HOME

ALGORITHMS FOR DECISION MAKING

MYKEL J. KOCHENDERFER, TIM A. WHEELER,
AND KYLE H. WRAY

MIT PRESS, 2022

INTRO

DOWNLOAD

BUY

OUTLINE

ANCILLARIES

ERRATA

Algorithms for Decision Making

Algorithms for Decision Making

Mykel J. Kochenderfer
Tim A. Wheeler
Kyle H. Wray

The MIT Press
Cambridge, Massachusetts
London, England

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in TEX Gyre Pagella by the authors in LATEX.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Kochenderfer, Mykel J., 1980– author. | Wheeler, Tim A. (Tim Allan), author. | Wray, Kyle H., author.
Title: Algorithms for decision making / Mykel J. Kochenderfer, Tim A. Wheeler, Kyle H. Wray.
Description: Cambridge : Massachusetts Institute of Technology, [2022] |
Includes bibliographical references and index.
Identifiers: LCCN 2021038701 | ISBN 9780262047012 (hardcover)
Subjects: LCSH: Decision support systems–Mathematics. | Algorithms.
Classification: LCC T58.62 .K666 2022 | DDC 658.4/03—dc23
LC record available at https://lccn.loc.gov/2021038701

10 9 8 7 6 5 4 3 2 1

To our families

Contents

PART II SEQUENTIAL PROBLEMS

part v multiagent systems

Preface

This book provides a broad introduction to algorithms for decision making under uncertainty. We cover a wide variety of topics related to decision making, introducing the underlying mathematical problem formulations and the algorithms for solving them. Figures, examples, and exercises are provided to convey the intuition behind the various approaches.

This book is intended for advanced undergraduates and graduate students, as well as professionals. It requires some mathematical maturity and assumes prior exposure to multivariable calculus, linear algebra, and probability concepts. Some review material is provided in the appendices. Disciplines where the book would be especially useful include mathematics, statistics, computer science, aerospace, electrical engineering, and operations research.

Fundamental to this textbook are the algorithms, which are all implemented in the Julia programming language. We have found this language to be ideal for specifying algorithms in human-readable form. The priority in the design of the algorithmic implementations was interpretability rather than efficiency. Industrial applications, for example, may benefit from alternative implementations. Permission is granted, free of charge, to use the code snippets associated with this book, subject to the condition that the source of the code is acknowledged.

Mykel J. Kochenderfer
Tim A. Wheeler
Kyle H. Wray
Stanford, California
February 28, 2022

Acknowledgments

This textbook has grown from a course on decision making under uncertainty taught at Stanford. We are grateful to the students and teaching assistants who have helped shape the course over the past six years.

The authors wish to thank the many individuals who have provided valuable feedback on early drafts of our manuscript, including Dylan Asmar, Drew Bagnell, Safa Bakhshi, Edward Balaban, Jean Betterton, Raunak Bhattacharyya, Kelsey Bing, Maxime Bouton, Hugo Buurmeijer, Austin Chan, Simon Chauvin, Shushman Choudhury, Jon Cox, Matthew Daly, Victoria Dax, Harrison Delecki, Richard Dewey, Paul Diederichs, Apoorva Dixit, Dea Dressel, Ben Duprey, Torstein Eliassen, Johannes Fischer, Rushil Goradia, Jayesh Gupta, Griffin Holt, Arec Jamgochian, Rohan Kapre, Mark Koren, Liam Kruse, Jonathan Larkin, Tor Lattimore, Bernard Lange, Ritchie Lee, Sheng Li, Michael Littman, Robert Moss, Joshua Ott, Emma Passmore, Oriana Peltzer, Francesco Piccoli, Nikhil Raghuraman, Jeffrey Sarnoff, Marc Schlichting, Ransalu Senanayake, Chelsea Sidrane, Michael Sheehan, Chris Strong, Zach Sunberg, Abiy Teshome, Alexandros Tzikas, Kemal Ure, Ziyu Wang, Josh Wolff, Zhen Wu, Anıl Yıldız, Shengtong Zhang, and Zongzhang Zhang. We also would like to thank Sydney Katz, Kunal Menda, and Ayan Mukhopadhyay for their contributions to the discussion in chapter 1. Ross Alexander produced many of the exercises throughout the book. It has been a pleasure working with Elizabeth Swayze from the MIT Press in preparing this manuscript for publication.

The style of this book was inspired by Edward Tufte. Among other stylistic elements, we adopted his wide margins and use of small multiples. The typesetting of this book is based on the Tufte-LaTeX package by Kevin Godby, Bil Kleb, and Bill Wood. The book's color scheme was adapted from the Monokai theme by Jon Skinner of Sublime Text (sublimetext.com) and a palette that better

accommodates individuals with color blindness.[1] For plots, we use the viridis color map defined by Stéfan van der Walt and Nathaniel Smith.

We have also benefited from the various open-source packages on which this textbook depends (see appendix G). The typesetting of the code was done with the help of pythontex, which is maintained by Geoffrey Poore. The typeface used for the algorithms is JuliaMono (github.com/cormullion/juliamono). The plotting was handled by pgfplots, which is maintained by Christian Feuersänger.

[1] B. Wong, "Points of View: Color Blindness," *Nature Methods*, vol. 8, no. 6, pp. 441–442, 2011.

1 Introduction

Many important problems involve decision making under uncertainty, including aircraft collision avoidance, wildfire management, and disaster response. When designing automated decision-making systems or decision-support systems, it is important to account for various sources of uncertainty while carefully balancing multiple objectives. We will discuss these challenges from a computational perspective, aiming to provide the theory behind decision-making models and computational approaches. This chapter introduces the problem of decision making under uncertainty, provides some examples of applications, and outlines the space of computational approaches. It then summarizes how various disciplines have contributed to our understanding of intelligent decision making and highlights areas of potential societal impact. We conclude with an outline of the remainder of the book.

1.1 Decision Making

An *agent* is an entity that acts based on observations of its environment. Agents may be physical entities, like humans or robots, or they may be nonphysical entities, such as decision support systems that are implemented entirely in software. As shown in figure 1.1, the interaction between the agent and the environment follows an *observe-act cycle* or *loop*.

The agent at time t receives an *observation* of the environment, denoted as o_t. Observations may be made, for example, through a biological sensory process, as in humans, or by a sensor system, like radar in an air traffic control system. Observations are often incomplete or noisy; humans may not see an approaching aircraft or a radar system might miss a detection due to electromagnetic interference. The agent then chooses an action a_t through some decision-making process.

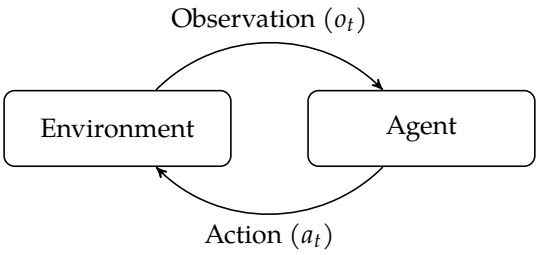


Figure 1.1. Interaction between an agent and its environment.

This action, such as sounding an alert, may have a nondeterministic effect on the environment.

Our focus is on agents that interact intelligently to achieve their objectives over time. Given the past sequence of observations, $o_1, \ldots, o_t$, and knowledge of the environment, the agent must choose an action a_t that best achieves its objectives in the presence of various sources of uncertainty,[1] including the following:

- *outcome uncertainty*, where the effects of our actions are uncertain,
- *model uncertainty*, where our model of the problem is uncertain,
- *state uncertainty*, where the true state of the environment is uncertain, and
- *interaction uncertainty*, where the behavior of the other agents interacting in the environment is uncertain.

[1] We focus here on discrete time problems. Continuous time problems are studied in the field of *control theory*. See D. E. Kirk, *Optimal Control Theory: An Introduction*. Prentice-Hall, 1970.

This book is organized around these four sources of uncertainty. Making decisions in the presence of uncertainty is central to the field of *artificial intelligence*,[2] as well as many other fields, as outlined in section 1.4. We will discuss a variety of algorithms, or descriptions of computational processes, for making decisions that are robust to uncertainty.

[2] A comprehensive introduction to artificial intelligence is provided by S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

1.2 Applications

The decision-making framework presented in the previous section can be applied to a wide variety of domains. This section discusses a few conceptual examples with real-world applications. Appendix F outlines additional notional problems that are used throughout this text to demonstrate the algorithms we discuss.

1.2.1 *Aircraft Collision Avoidance*

To help prevent midair collisions between aircraft, we want to design a system that can alert pilots to potential threats and direct them how to maneuver to avoid them.[3] The system communicates with the transponders of other aircraft to identify their positions with some degree of accuracy. Deciding what guidance to provide to the pilots is challenging. There is uncertainty in how quickly the pilots will respond and how aggressively they will comply with the guidance. In addition, there is uncertainty in the behavior of other aircraft. We want our system to alert sufficiently early to provide enough time for pilots to maneuver their aircraft to avoid collisions, but we do not want our system to issue alerts too early, which would result in many unnecessary maneuvers. Since this system is to be used continuously worldwide, we need the system to provide an exceptional level of safety.

[3] This application is discussed in a chapter titled ''Collision Avoidance'' by M. J. Kochenderfer, *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

1.2.2 *Automated Driving*

We want to build an autonomous vehicle that can safely drive in urban environments.[4] The vehicle must rely on a suite of sensors to perceive its environment in order to make safe decisions. One type of sensor is lidar, which involves measuring laser reflections off the environment to determine distances to obstacles. Another type of sensor is a camera, which, through computer vision algorithms, can detect pedestrians and other vehicles. Both of these types of sensors are imperfect and susceptible to noise and occlusions. For example, a parked truck may occlude a pedestrian who may be trying to cross at a crosswalk. Our system must predict the intentions and future paths of other vehicles, pedestrians, and other road users from their observable behaviors in order to navigate safely to our destination.

[4] A similar application was explored by M. Bouton, A. Nakhaei, K. Fujimura, and M. J. Kochenderfer, "Safe Reinforcement Learning with Scene Decomposition for Navigating Complex Urban Environments," in *IEEE Intelligent Vehicles Symposium (IV)*, 2019.

1.2.3 *Breast Cancer Screening*

Worldwide, breast cancer is the most common cancer in women. Detecting breast cancer early can help save lives, with mammography being the most effective screening tool available. However, mammography carries with it potential risks, including false positives, which can result in unnecessary and invasive diagnostic follow-up. Research over the years has resulted in various population-based screening schedules based on age to balance the benefits and risks of testing. Developing a system that can make recommendations based on personal risk

characteristics and screening history has the potential to result in better health outcomes.[5] The success of such a system can be compared to populationwide screening schedules in terms of total expected quality-adjusted life years, the number of mammograms, the prevalence of false positives, and the risk of undetected, invasive cancer.

[5] Such a concept is proposed by T. Ayer, O. Alagoz, and N. K. Stout, "A POMDP Approach to Personalize Mammography Screening Decisions," *Operations Research*, vol. 60, no. 5, pp. 1019–1034, 2012.

1.2.4 *Financial Consumption and Portfolio Allocation*

Suppose that we want to build a system that recommends how much of an individual's wealth should be consumed that year and how much should be invested.[6] The investment portfolio may include stocks and bonds with different levels of risk and expected return. The evolution of wealth is stochastic due to uncertainty in both earned and investment income, often increasing until the investor is near retirement, and then steadily decreasing. The enjoyment that comes from the consumption of a unit of wealth in a year typically diminishes with the amount consumed, resulting in a desire to smooth consumption over the lifespan of the individual.

[6] A related problem was studied by R. C. Merton, "Optimum Consumption and Portfolio Rules in a Continuous-Time Model," *Journal of Economic Theory*, vol. 3, no. 4, pp. 373–413, 1971.

1.2.5 *Distributed Wildfire Surveillance*

Situational awareness is a major challenge when fighting wildfires. The state of a fire evolves over time, influenced by factors such as wind and the distribution of fuel in the environment. Many wildfires span large geographic regions. One concept for monitoring a wildfire is to use a team of drones equipped with sensors to fly above it.[7] The sensing range of individual drones is limited, but the information from the team can be fused to provide a unified snapshot of the situation to drive resource allocation decisions. We would like the team members to autonomously determine how to collaborate with each other to provide the best coverage of the fire. Effective monitoring requires deciding how to maneuver to cover areas where new sensor information is likely to be useful; spending time in areas where we are certain of whether the fire is burning or not would be wasteful. Identifying important areas to explore requires reasoning about the stochastic evolution of the fire, given only imperfect knowledge of its current state.

[7] This application was explored by K. D. Julian and M. J. Kochenderfer, "Distributed Wildfire Surveillance with Autonomous Aircraft Using Deep Reinforcement Learning," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 42, no. 8, pp. 1768–1778, 2019.

1.2.6 Mars Science Exploration

Rovers have made important discoveries on and increased our understanding of Mars. However, a major bottleneck in scientific exploration has been the communication link between the rover and the operations team on Earth. It can take as much as half an hour for sensor information to be sent from Mars to Earth and for commands to be sent from Earth to Mars. In addition, guidance to rovers needs to be planned in advance because there are limited upload and download windows with Mars due to the positions of orbiters serving as information relays between the planets. Recent research has suggested that the efficiency of science exploration missions can be improved by a factor of five through the introduction of greater levels of autonomy.[8] Human operators would still provide high-level guidance on mission objectives, but the rover would have the flexibility to select its own science targets using the most up-to-date information. In addition, it would be desirable for rovers to respond appropriately to various hazards and system failures without human intervention.

[8] This concept is presented and evaluated by D. Gaines, G. Doran, M. Paton, B. Rothrock, J. Russino, R. Mackey, R. Anderson, R. Francis, C. Joswig, H. Justice, K. Kolcio, G. Rabideau, S. Schaffer, J. Sawoniewicz, A. Vasavada, V. Wong, K. Yu, and A.-a. Agha-mohammadi, "Self-Reliant Rovers for Increased Mission Productivity," *Journal of Field Robotics*, vol. 37, no. 7, pp. 1171–1196, 2020.

1.3 Methods

There are many methods for designing decision-making agents. Depending on the application, some may be more appropriate than others. They differ in the responsibilities of the designer and the tasks left to automation. This section briefly overviews a collection of these methods. The book will focus primarily on planning and reinforcement learning, but some of the techniques will involve elements of supervised learning and optimization.

1.3.1 Explicit Programming

The most direct method for designing a decision-making agent is to anticipate all the scenarios that the agent might find itself in and explicitly program what the agent should do in response to each one. The explicit programming approach may work well for simple problems, but it places a large burden on the designer to provide a complete strategy. Various agent programming languages and frameworks have been proposed to make programming agents easier.

1.3.2 *Supervised Learning*

With some problems, it may be easier to show an agent what to do rather than to write a program for the agent to follow. The designer provides a set of training examples, and an automated learning algorithm must generalize from these examples. This approach is known as *supervised learning* and has been widely applied to classification problems. This technique is sometimes called *behavioral cloning* when applied to learning mappings from observations to actions. Behavioral cloning works well when an expert designer actually knows the best course of action for a representative collection of situations. Although a wide variety of different learning algorithms exist, they generally cannot perform better than human designers in new situations.

1.3.3 *Optimization*

Another approach is for the designer to specify the space of possible decision strategies and a performance measure to be maximized. Evaluating the performance of a decision strategy generally involves running a batch of simulations. The optimization algorithm then performs a search in this space for the optimal strategy. If the space is relatively small and the performance measure does not have many local optima, then various local or global search methods may be appropriate. Although knowledge of a dynamic model is generally assumed to run the simulations, it is not otherwise used to guide the search, which can be important for complex problems.

1.3.4 *Planning*

Planning is a form of optimization that uses a model of the problem dynamics to help guide the search. A broad base of literature explores various planning problems, much of it focused on deterministic problems. For some problems, it may be acceptable to approximate the dynamics with a deterministic model. Assuming a deterministic model allows us to use methods that can more easily scale to high-dimensional problems. For other problems, accounting for future uncertainty is critical. This book focuses entirely on problems in which accounting for uncertainty is important.

1.3.5 Reinforcement Learning

Reinforcement learning relaxes the assumption in planning that a model is known ahead of time. Instead, the decision-making strategy is learned while the agent interacts with the environment. The designer only has to provide a performance measure; it is up to the learning algorithm to optimize the behavior of the agent. One of the interesting complexities that arises in reinforcement learning is that the choice of action affects not only the immediate success of the agent in achieving its objectives, but also the agent's ability to learn about the environment and identify the characteristics of the problem that it can exploit.

1.4 History

The theory of automating the process of decision making has its roots in the dreams of early philosophers, scientists, mathematicians, and writers. The ancient Greeks began incorporating automation into myths and stories as early as 800 BC. The word *automaton* was first used in Homer's *Iliad*, which contains references to the notion of automatic machines, including mechanical tripods used to serve dinner guests.[9] In the seventeenth century, philosophers proposed the use of logic rules to automatically settle disagreements. Their ideas created the foundation for mechanized reasoning.

Beginning in the late eighteenth century, inventors began creating automatic machines to perform labor. In particular, a series of innovations in the textile industry led to the development of the automatic loom, which in turn laid the foundation for the first factory robots.[10] In the early nineteenth century, the use of intelligent machines to automate labor began to make its way into science fiction novels. The word *robot* originated in the Czech writer Karel Čapek's play titled *R.U.R.*, short for *Rossum's Universal Robots*, about machines that could perform work that humans would prefer not to do. The play inspired other science fiction writers to incorporate robots into their writing. In the mid-twentieth century, the notable writer and professor Isaac Asimov laid out his vision for robotics in his famous *Robot* series.

A major challenge in practical implementations of automated decision making is accounting for uncertainty. Even at the end of the twentieth century, George Dantzig, most famous for developing the simplex algorithm, stated in 1991:

[9] S. Vasileiadou, D. Kalligeropoulos, and N. Karcanias, "Systems, Modelling and Control in Ancient Greece: Part 1: Mythical Automata," *Measurement and Control*, vol. 36, no. 3, pp. 76–80, 2003.

[10] N. J. Nilsson, *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.

> In retrospect it is interesting to note that the original problem that started my research is still outstanding—namely the problem of planning or scheduling dynamically over time, particularly planning dynamically under uncertainty. If such a problem could be successfully solved it could (eventually through better planning) contribute to the well-being and stability of the world.[11]

[11] G. B. Dantzig, "Linear Programming," *Operations Research*, vol. 50, no. 1, pp. 42–47, 2002.

While decision making under uncertainty still remains an active area of research, over the past few centuries, researchers and engineers have come closer to making the concepts posed by these early dreamers possible. Current state-of-the-art decision-making algorithms rely on a convergence of concepts developed in multiple disciplines, including economics, psychology, neuroscience, computer science, engineering, mathematics, and operations research. This section highlights some major contributions from these disciplines. The cross-pollination between disciplines has led to many recent advances and will likely continue to support growth in the future.

1.4.1 Economics

Economics requires models of human decision making. One approach to building such models involves utility theory, which was first introduced in the late eighteenth century.[12] Utility theory provides a means to model and compare the desirability of various outcomes. For example, utility can be used to compare the desirability of monetary quantities. In the *Theory of Legislation*, Jeremy Bentham summarized the nonlinearity in the utility of money:

[12] G. J. Stigler, "The Development of Utility Theory. I," *Journal of Political Economy*, vol. 58, no. 4, pp. 307–327, 1950.

> *1st.* Each portion of wealth has a corresponding portion of happiness.
> *2nd.* Of two individuals with unequal fortunes, he who has the most wealth has the most happiness.
> *3rd.* The excess in happiness of the richer will not be so great as the excess of his wealth.[13]

[13] J. Bentham, *Theory of Legislation*. Trübner & Company, 1887.

By combining the concept of utility with the notion of rational decision making, economists in the mid-twentieth century established a basis for the maximum expected utility principle. This principle is a key concept behind the creation of autonomous decision-making agents. Utility theory also gave rise to the development of game theory, which attempts to understand the behavior of multiple agents acting in the presence of one another to maximize their interests.[14]

[14] O. Morgenstern and J. von Neumann, *Theory of Games and Economic Behavior*. Princeton University Press, 1953.

1.4.2 *Psychology*

Psychologists also study human decision making, typically from the perspective of human behavior. By studying the reactions of animals to stimuli, psychologists have been developing theories of trial-and-error learning since the nineteenth century. Researchers noticed that animals tend to make decisions based on the satisfaction or discomfort they experienced in previous similar situations. Russian psychologist Ivan Pavlov combined this idea with the concept of reinforcement after observing the salivation patterns of dogs when fed. Psychologists found that a pattern of behavior could be strengthened or weakened using continuous reinforcement of a particular stimulus. In the mid-twentieth century, the mathematician and computer scientist Alan Turing expressed the possibility of allowing machines to learn in the same manner:

> The organization of a machine into a universal machine would be most impressive if the arrangements of interference involve very few inputs. The training of a human child depends largely on a system of rewards and punishments, and this suggests that it ought to be possible to carry through the organising with only two interfering inputs, one for ''pleasure'' or ''reward'' (R) and the other for ''pain'' or ''punishment'' (P).[15]

[15] A. M. Turing, "Intelligent Machinery," National Physical Laboratory, Report, 1948.

The work of psychologists laid the foundation for the field of reinforcement learning, a critical technique used to teach agents to make decisions in uncertain environments.[16]

[16] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

1.4.3 *Neuroscience*

While psychologists study human behavior as it happens, neuroscientists focus on the biological processes used to create the behavior. At the end of the nineteenth century, scientists found that the brain is composed of an interconnected network of neurons, which is responsible for its ability to perceive and reason about the world. Artificial intelligence pioneer Nils Nilsson describes the application of these findings to decision making as follows:

> Because it is the *brain* of an animal that is responsible for converting sensory information into action, it is to be expected that several good ideas can be found in the work of neurophysiologists and neuroanatomists who study brains and their fundamental components, neurons.[17]

[17] N. J. Nilsson, *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.

In the 1940s, researchers first proposed that neurons could be considered as individual ''logic units'' capable of performing computational operations when pieced together into a network. This work served as a basis for neural networks, which are used in the field of artificial intelligence to perform a variety of complex tasks.

1.4.4 Computer Science

In the mid-twentieth century, computer scientists began formulating the problem of intelligent decision making as a problem of symbolic manipulation through formal logic. The computer program Logic Theorist, written in the mid-twentieth century to perform automated reasoning, used this way of thinking to prove mathematical theorems. Herbert Simon, one of its inventors, addressed the symbolic nature of the program by relating it to the human mind:

> We invented a computer program capable of thinking nonnumerically, and thereby solved the venerable mind/body problem, explaining how a system composed of matter can have the properties of mind.[18]

[18] Quoted by J. Agar, *Science in the 20th Century and Beyond*. Polity, 2012.

These symbolic systems relied heavily on human expertise. An alternative approach to intelligence, called *connectionism*, was inspired in part by developments in neuroscience and focuses on the use of artificial neural networks as a substrate for intelligence. With the knowledge that neural networks could be trained for pattern recognition, connectionists attempt to learn intelligent behavior from data or experience rather than the hard-coded knowledge of experts. The connectionist paradigm underpinned the success of AlphaGo, the autonomous program that beat a human professional at the game of Go, as well as much of the development of autonomous vehicles. Algorithms that combine both symbolic and connectionist paradigms remain an active area of research today.

1.4.5 Engineering

The field of engineering has focused on allowing physical systems, such as robots, to make intelligent decisions. World-renowned roboticist Sebastian Thrun describes the components of these systems as follows:

> Robotics systems have in common that they are situated in the physical world, perceive their environments through sensors, and manipulate their environment through things that move.[19]

[19] S. Thrun, "Probabilistic Robotics," *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

To design these systems, engineers must address perception, planning, and acting. Physical systems perceive the world by using their sensors to create a representation of the salient features of their environment. The field of state estimation has focused on using sensor measurements to construct a belief about the state of the world. Planning requires reasoning about the ways to execute the tasks they are designed to perform. The planning process has been enabled by advances in the semiconductor industry spanning many decades.[20] Once a plan has been devised, an autonomous agent must act on it in the real world. This task requires both hardware (in the form of actuators) and algorithms to control the actuators and reject disturbances. The field of control theory has focused on the stabilization of mechanical systems through feedback control.[21] Automatic control systems are widely used in industry, from the regulation of temperature in an oven to the navigation of aerospace systems.

[20] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[21] D. A. Mindell, *Between Human and Machine: Feedback, Control, and Computing Before Cybernetics*. JHU Press, 2002.

1.4.6 Mathematics

An agent must be able to quantify its uncertainty to make informed decisions in uncertain environments. The field of decision making relies heavily on probability theory for this task. In particular, Bayesian statistics plays an important role in this text. In 1763, a paper of Thomas Bayes was published posthumously, containing what would later be known as Bayes' rule. His approach to probabilistic inference fell in and out of favor until the mid-twentieth century, when researchers began to find Bayesian methods useful in a number of settings.[22] Mathematician Bernard Koopman found practical use for the theory during World War II:

> Every operation involved in search is beset with uncertainties; it can be understood quantitatively only in terms of [...] probability. This may now be regarded as a truism, but it seems to have taken the developments in operational research of the Second World War to drive home its practical implications.[23]

Sampling-based methods (sometimes referred to as *Monte Carlo methods*) developed in the early twentieth century for large-scale calculations as part of the Manhattan Project, made some inference techniques possible that would previously have been intractable. These foundations serve as a basis for Bayesian networks, which increased in popularity later in the twentieth century in the field of artificial intelligence.

[22] W. M. Bolstad and J. M. Curran, *Introduction to Bayesian Statistics*. Wiley, 2016.

[23] B. O. Koopman, *Search and Screening: General Principles with Historical Applications*. Pergamon Press, 1980.

1.4.7 Operations Research

Operations research is concerned with finding optimal solutions to decision-making problems such as resource allocation, asset investment, and maintenance scheduling. In the late nineteenth century, researchers began to explore the application of mathematical and scientific analysis to the production of goods and services. The field was accelerated during the Industrial Revolution when companies began to subdivide their management into departments responsible for distinct aspects of overall decisions. During World War II, the optimization of decisions was applied to allocating resources to an army. Once the war came to an end, businesses began to notice that the same operations research concepts previously used to make military decisions could help them optimize business decisions. This realization led to the development of management science, as described by the organizational theorist Harold Koontz:

> The abiding belief of this group is that, if management, or organization, or planning, or decision making is a logical process, it can be expressed in terms of mathematical symbols and relationships. The central approach of this school is the model, for it is through these devices that the problem is expressed in its basic relationships and in terms of selected goals or objectives.[24]

[24] H. Koontz, "The Management Theory Jungle," *Academy of Management Journal*, vol. 4, no. 3, pp. 174–188, 1961.

This desire to be able to better model and understand business decisions sparked the development of a number of concepts used today, such as linear programming, dynamic programming, and queuing theory.[25]

[25] F. S. Hillier, *Introduction to Operations Research*. McGraw-Hill, 2012.

1.5 Societal Impact

Algorithmic approaches to decision making have transformed society and will likely continue to in the future. This section briefly highlights a few ways that decision-making algorithms can contribute to society and introduces challenges that remain when attempting to ensure a broad benefit.[26]

[26] A much more thorough discussion is provided by Z. R. Shi, C. Wang, and F. Fang, "Artificial Intelligence for Social Good: A Survey," 2020. arXiv: 2001.01818v1.

Algorithmic approaches have contributed to environmental sustainability. In the context of energy management, for example, Bayesian optimization has been applied to automated home energy management systems. Algorithms from the field of multiagent systems are used to predict the operation of smart grids, design markets for trading energy, and predict rooftop solar-power adoption. Algorithms have also been developed to protect biodiversity. For example, neural networks are used to automate wildlife censuses, game-theoretic approaches are used to

combat poaching in forests, and optimization techniques are employed to allocate resources for habitat management.

Decision-making algorithms have found success in the field of medicine for decades. Such algorithms have been used for matching residents to hospitals and organ donors to patients in need. An early application of Bayesian networks, which we will cover in the first part of this book, was disease diagnosis. Since then, Bayesian networks have been widely used in medicine for the diagnosis and prognosis of diseases. The field of medical image processing has been transformed by deep learning, and algorithmic ideas have recently played an important role in understanding the spread of disease.

Algorithms have enabled us to understand the growth of urban areas and facilitate their design. Data-driven algorithms have been widely used to improve public infrastructure. For example, stochastic processes have been used to predict failures in water pipelines, deep learning has improved traffic management, and Markov decision processes and Monte Carlo methods have been employed to improve emergency response. Ideas from decentralized multiagent systems have optimized travel routes, and path-planning techniques have been used to optimize the delivery of goods. Decision-making algorithms have been used for autonomous cars and improving aircraft safety.

Algorithms for optimizing decisions can amplify the impact of its users, regardless of their intention. If the objective of the user of these algorithms, for example, is to spread misinformation during a political election, then optimization processes can help facilitate this. However, similar algorithms can be used to monitor and counteract the spread of false information. Sometimes the implementation of these decision-making algorithms can lead to downstream consequences that their users did not intend.[27]

Although algorithms have the potential to bring significant benefits, there are also challenges associated with their implementation in society. Data-driven algorithms often suffer from inherent biases and blind spots due to the way that data is collected. As algorithms become part of our lives, it is important to understand how the risk of bias can be reduced and how the benefits of algorithmic progress can be distributed in a manner that is equitable and fair. Algorithms can also be vulnerable to adversarial manipulation, and it is critical that we design algorithms that are robust to such attacks. It is also important to extend moral and legal frameworks for preventing unintended consequences and assigning responsibility.

[27] For a general discussion, see B. Christian, *The Alignment Problem*. Norton & Company, 2020. See also the discussion by D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete Problems in AI Safety," 2016. arXiv: 1606.06565v2.

1.6 Overview

This book is divided into five parts. The first part addresses the problem of reasoning about uncertainty and objectives in simple decisions at a single point in time. The second extends decision making to sequential problems, where we must make a sequence of decisions in response to information about the outcomes of our actions as we proceed. The third addresses model uncertainty, where we do not start with a known model and must learn how to act through interaction with the environment. The fourth addresses state uncertainty, where imperfect perceptual information prevents us from knowing the full environmental state. The final part discusses decision contexts involving multiple agents.

1.6.1 Probabilistic Reasoning

Rational decision making requires reasoning about our uncertainty and objectives. This part of the book begins by discussing how to represent uncertainty as a probability distribution. Real-world problems require reasoning about distributions over many variables. We will discuss how to construct these models, how to use them to make inferences, and how to learn their parameters and structure from data. We then introduce the foundations of *utility theory* and show how it forms the basis for rational decision making under uncertainty through the maximum expected utility principle. We then discuss how notions of utility theory can be incorporated into the probabilistic graphical models introduced earlier in this chapter to form what are called *decision networks*.

1.6.2 Sequential Problems

Many important problems require that we make a series of decisions. The same principle of maximum expected utility still applies, but optimal decision making in a sequential context requires reasoning about future sequences of actions and observations. This part of the book will discuss sequential decision problems in stochastic environments, where the outcomes of our actions are uncertain. We will focus on a general formulation of sequential decision problems under the assumption that the model is known and that the environment is fully observable. We will relax both of these assumptions later in the book. Our discussion will begin with the introduction of the *Markov decision process* (*MDP*), the standard

mathematical model for sequential decision problems. We will discuss several approaches for finding exact solutions to these types of problems. Because large problems sometimes do not permit exact solutions to be found efficiently, we will discuss a collection of both offline and online approximate solution methods, along with a type of method that involves directly searching the space of parameterized decision policies. Finally, we will discuss approaches for validating that our decision strategies will perform as expected when deployed in the real world.

1.6.3 Model Uncertainty

In our discussion of sequential decision problems up to this point, we have assumed that the transition and reward models are known. In many problems, however, the dynamics and rewards are not known exactly, and the agent must learn to act through experience. By observing the outcomes of its actions in the form of state transitions and rewards, the agent is to choose actions that maximize its long-term accumulation of rewards. Solving such problems in which there is model uncertainty is the subject of the field of *reinforcement learning* and the focus of this part of the book. We will discuss several challenges in addressing model uncertainty. First, the agent must carefully balance the exploration of the environment with the exploitation of knowledge gained through experience. Second, rewards may be received long after the important decisions have been made, so credit for later rewards must be assigned to earlier decisions. Third, the agent must generalize from limited experience. We will review the theory and some of the key algorithms for addressing these challenges.

1.6.4 State Uncertainty

In this part, we extend uncertainty to include the state. Instead of observing the state exactly, we receive observations that have only a probabilistic relationship with the state. Such problems can be modeled as a *partially observable Markov decision process* (*POMDP*). A common approach to solving POMDPs involves inferring a belief distribution over the underlying state at the current time step and then applying a policy that maps beliefs to actions. This part begins by discussing how to update our belief distribution, given a past sequence of observations and actions. It then discusses a variety of exact and approximate methods for solving POMDPs.

1.6.5 *Multiagent Systems*

Up to this point, there has only been one agent making decisions within the environment. This part expands the previous four parts to multiple agents, discussing the challenges that arise from interaction uncertainty. We begin by discussing simple games, where a group of agents simultaneously each select an action. The result is an individual reward for each agent based on the combined joint action. The *Markov game* (*MG*) represents a generalization of both simple games to multiple states and the MDP to multiple agents. Consequently, the agents select actions that can stochastically change the state of a shared environment. Algorithms for MGs rely on reinforcement learning due to uncertainty about the policies of the other agents. A *partially observable Markov game* (*POMG*) introduces state uncertainty, further generalizing MGs and POMDPs, as agents now receive only noisy local observations. The *decentralized partially observable Markov decision process* (*Dec-POMDP*) focuses the POMG on a collaborative, multiagent team where there is a shared reward among the agents. This part of the book presents these four categories of problems and discusses exact and approximate algorithms that solve them.

PART I

PROBABILISTIC REASONING

Rational decision making requires reasoning about our uncertainty and objectives. Uncertainty arises from practical and theoretical limitations on our ability to predict future events. For example, predicting exactly how a human operator will respond to advice from a decision support system would require, among other things, a detailed model of how the human brain works. Even the paths of satellites can be difficult to predict. Although Newtonian physics permit highly precise predictions of satellite trajectories, spontaneous failures in attitude thrusters can result in large deviations from the nominal path, and even small imprecisions can compound over time. To achieve its objectives, a robust decision-making system must account for various sources of uncertainty in the current state of the world and future events. This part of the book begins by discussing how to represent uncertainty using probability distributions. Real-world problems require reasoning about distributions over many variables. We will discuss how to construct these models, use them to make inferences, and learn their parameters and structure from data. We then introduce the foundations of utility theory and show how it forms the basis for rational decision making under uncertainty. Utility theory can be incorporated into the probabilistic graphical models introduced earlier to form what are called decision networks. We focus on single-step decisions, reserving discussion of sequential decision problems for the next part of the book.

2 Representation

Computationally accounting for uncertainty requires a formal representation. This chapter discusses how to represent uncertainty.[1] We begin by introducing the notion of degree of belief and show how a set of axioms results in our ability to use probability distributions to quantify our uncertainty.[2] We discuss several useful forms of distributions over both discrete and continuous variables. Because many important problems involve probability distributions over a large number of variables, we discuss a way to represent joint distributions efficiently that takes advantage of conditional independence between variables.

[1] A detailed discussion of a variety of approaches to representing uncertainty is provided by F. Cuzzolin, *The Geometry of Uncertainty*. Springer, 2021.

[2] For a more comprehensive elaboration, see E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

2.1 Degrees of Belief and Probability

In problems involving uncertainty, it is essential to be able to compare the plausibility of different statements. We would like to be able to represent, for example, that proposition A is more plausible than proposition B. If A represents ''my actuator failed,'' and B represents ''my sensor failed,'' then we would write $A \succ B$. Using this basic relation $\succ$, we can define several other relations:

$$A \prec B \text{ if and only if } B \succ A \tag{2.1}$$

$$A \sim B \text{ if and only if neither } A \succ B \text{ nor } B \succ A \tag{2.2}$$

$$A \succeq B \text{ if and only if } A \succ B \text{ or } A \sim B \tag{2.3}$$

$$A \preceq B \text{ if and only if } B \succ A \text{ or } A \sim B \tag{2.4}$$

We want to make certain assumptions about the relationships induced by the operators $\succ$, $\sim$, and $\prec$. The assumption of *universal comparability* requires exactly one of the following to hold: $A \succ B$, $A \sim B$, or $A \prec B$. The assumption of *transitivity* requires that if $A \succeq B$ and $B \succeq C$, then $A \succeq C$. Universal comparability

and transitivity assumptions lead to an ability to represent plausibility by a real-valued function P that has the following two properties:[3]

$$P(A) > P(B) \text{ if and only if } A \succ B \tag{2.5}$$

$$P(A) = P(B) \text{ if and only if } A \sim B \tag{2.6}$$

If we make a set of additional assumptions[4] about the form of P, then we can show that P must satisfy the basic *axioms of probability* (see appendix A.2). If we are certain of A, then $P(A) = 1$. If we believe that A is impossible, then $P(A) = 0$. Uncertainty in the truth of A is represented by values between the two extrema. Hence, probability masses must lie between 0 and 1, with $0 \leq P(A) \leq 1$.

[3] See discussion in E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[4] The axiomatization of subjective probability is given by P. C. Fishburn, "The Axioms of Subjective Probability," *Statistical Science*, vol. 1, no. 3, pp. 335–345, 1986. A more recent axiomatization is contained in M. J. Dupré and F. J. Tipler, "New Axioms for Rigorous Bayesian Probability," *Bayesian Analysis*, vol. 4, no. 3, pp. 599–606, 2009.

2.2 Probability Distributions

A *probability distribution* assigns probabilities to different outcomes.[5] There are different ways to represent probability distributions depending on whether they involve discrete or continuous outcomes.

[5] For an introduction to probability theory, see D. P. Bertsekas and J. N. Tsitsiklis, *Introduction to Probability*. Athena Scientific, 2002.

2.2.1 Discrete Probability Distributions

A *discrete probability distribution* is a distribution over a discrete set of values. We can represent such a distribution as a *probability mass function*, which assigns a probability to every possible assignment of its input variable to a value. For example, suppose that we have a variable X that can take on one of n values: $1, \ldots, n$, or, using *colon notation*, $1{:}n$.[6] A distribution associated with X specifies the n probabilities of the various assignments of values to that variable, in particular $P(X = 1), \ldots, P(X = n)$. Figure 2.1 shows an example of a discrete distribution.

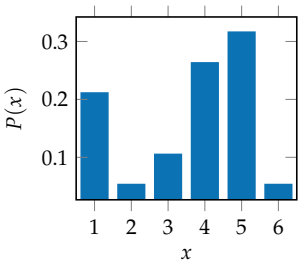


Figure 2.1. A probability mass function for a distribution over $1{:}6$.

There are constraints on the probability masses associated with discrete distributions. The masses must sum to 1:

$$\sum_{i=1}^{n} P(X = i) = 1 \tag{2.7}$$

and $0 \leq P(X = i) \leq 1$ for all i.

For notational convenience, we will use lowercase letters and superscripts as shorthand when discussing the assignment of values to variables. For example, $P(x^3)$ is shorthand for $P(X = 3)$. If X is a *binary variable*, it can take on the value of true or false.[7] We will use 0 to represent false and 1 to represent true. For example, we use $P(x^0)$ to represent the probability that X is false.

[6] We will often use this colon notation for compactness. Other texts sometimes use the notation $[1 \,..\, n]$ for integer intervals from 1 to n. We will also use this colon notation to index into vectors and matrices. For example $x_{1:n}$ represents $x_1, \ldots, x_n$. The colon notation is sometimes used in programming languages, such as Julia and MATLAB.

[7] Julia, like many other programming languages, similarly treats Boolean values as 0 and 1 in numerical operations.

The *parameters* of a distribution govern the probabilities associated with different assignments. For example, if we use X to represent the outcome of a roll of a six-sided die, then we would have $P(x^1) = \theta_1, \ldots, P(x^6) = \theta_6$, with $\theta_{1:6}$ being the six parameters of the distribution. However, we need only five *independent parameters* to uniquely specify the distribution over the outcomes of the roll because we know that the distribution must sum to 1.

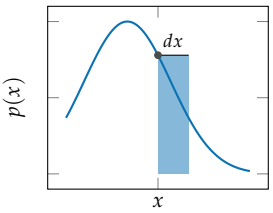


Figure 2.2. Probability density functions are used to represent continuous probability distributions. If $p(x)$ is a probability density, then $p(x)dx$ indicated by the area of the blue rectangle is the probability that a sample from the random variable falls within the interval $(x, x + dx)$ as $dx \to 0$.

2.2.2 *Continuous Probability Distributions*

A *continuous probability distribution* is a distribution over a continuous set of values. Representing a distribution over a continuous variable is a little less straightforward than for a discrete variable. For instance, in many continuous distributions, the probability that a variable takes on a particular value is infinitesimally small. One way to represent a continuous probability distribution is to use a *probability density function* (see figure 2.2), represented with lowercase letters. If $p(x)$ is a probability density function over X, then $p(x)dx$ is the probability that X falls within the interval $(x, x + dx)$ as $dx \to 0$. Similar to how the probability masses associated with a discrete distribution must sum to 1, a probability density function $p(x)$ must integrate to 1:

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1 \tag{2.8}$$

Another way to represent a continuous distribution is with a *cumulative distribution function* (see figure 2.3), which specifies the probability mass associated with values below some threshold. If we have a cumulative distribution function P associated with variable X, then $P(x)$ represents the probability mass associated with X taking on a value less than or equal to x. A cumulative distribution function can be defined in terms of a probability density function p as follows:

$$\mathrm{cdf}_X(x) = P(X \leq x) = \int_{-\infty}^{x} p(x')\,\mathrm{d}x' \tag{2.9}$$

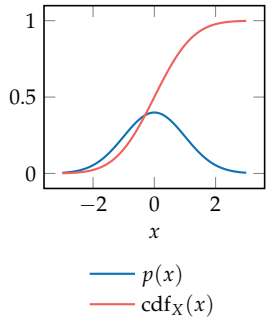


Figure 2.3. The probability density function and cumulative distribution function for a standard Gaussian distribution.

Related to the cumulative distribution function is the *quantile function*, also called the *inverse cumulative distribution function* (see figure 2.4). The value of $\mathrm{quantile}_X(\alpha)$ is the value x such that $P(X \leq x) = \alpha$. In other words, the quantile function returns the minimum value of x whose cumulative distribution value is greater than or equal to α. Of course, we have $0 \leq \alpha \leq 1$.

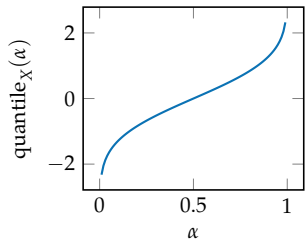


Figure 2.4. The quantile function for a standard Gaussian distribution.

There are many different parameterized families of distributions. We outline several in appendix B. A simple distribution family is the *uniform distribution*

$\mathcal{U}(a, b)$, which assigns probability density uniformly between a and b, and zero elsewhere. Hence, the probability density function is $p(x) = 1/(b - a)$ for x in the interval $[a, b]$. We can use $\mathcal{U}(x \mid a, b)$ to represent the density at x.[8] The *support* of a distribution is the set of values that are assigned nonzero density. In the case of $\mathcal{U}(a, b)$, the support is the interval $[a, b]$. See example 2.1.

[8] Some texts use a semicolon to separate the parameters of the distribution. For example, one can also write $\mathcal{U}(x; a, b)$.

Example 2.1. An example of a uniform distribution with a lower bound of 0 and an upper bound of 10.

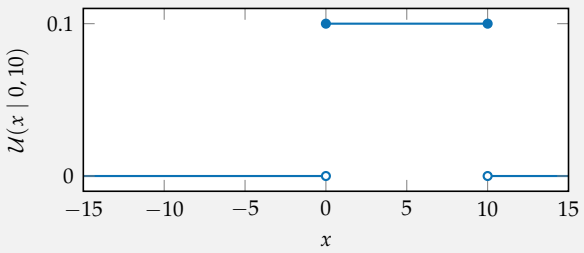
The uniform distribution $\mathcal{U}(0, 10)$ assigns equal probability to all values in the range $[0, 10]$ with a probability density function:

$$\mathcal{U}(x \mid 0, 10) = \begin{cases} 1/10 & \text{if } 0 \leq x \leq 10 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

The probability that a random sample from this distribution is equal to the constant π is essentially zero. However, we can define nonzero probabilities for samples being within some interval, such as $[3, 5]$. For example, the probability that a sample lies between 3 and 5 given the distribution plotted here is:

$$\int_3^5 \mathcal{U}(x \mid 0, 10)\, \mathrm{d}x = \frac{5 - 3}{10} = \frac{1}{5} \tag{2.11}$$

The support of this distribution is the interval $[0, 10]$.



Another common distribution for continuous variables is the *Gaussian distribution* (also called the *normal distribution*). The Gaussian distribution is parameterized by a mean μ and variance σ^2:

$$p(x) = \mathcal{N}(x \mid \mu, \sigma^2) \tag{2.12}$$

Here, σ is the *standard deviation*, which is the square root of the variance. The variance is also commonly denoted by ν. We use $\mathcal{N}(\mu, \sigma^2)$ to represent a Gaus-

sian distribution with parameters μ and σ^2 and $\mathcal{N}(x \mid \mu, \sigma^2)$ to represent the probability density at x, as given by

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right) \tag{2.13}$$

where ϕ is the *standard normal density function*:

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tag{2.14}$$

Appendix B shows plots of Gaussian density functions with different parameters.

Although a Gaussian distribution is often convenient because it is defined by only two parameters and makes computation and derivation easy, it has some limitations. It assigns nonzero probability to large positive and negative values, which may not be appropriate for the quantity we are trying to model. For example, we might not want to assign nonzero probabilities for aircraft flying below the ground or at infeasible altitudes. We can use a *truncated Gaussian distribution* (see figure 2.5) to bound the *support* of possible values; that is, the range of values assigned nonzero probabilities. The density function is given by

$$\mathcal{N}(x \mid \mu, \sigma^2, a, b) = \frac{\frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)}{\Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)} \tag{2.15}$$

when x is within the interval (a, b).

The function Φ is the *standard normal cumulative distribution function*, as given by

$$\Phi(x) = \int_{-\infty}^{x} \phi(x')\,\mathrm{d}x' \tag{2.16}$$

The Gaussian distribution is *unimodal*, meaning that there is a point in the distribution at which the density increases on one side and decreases on the other side. There are different ways to represent continuous distributions that are *multimodal*. One way is to use a *mixture model*, which is a mixture of multiple distributions. We mix together a collection of unimodal distributions to obtain a multimodal distribution. A *Gaussian mixture model* is a mixture model that is simply a weighted average of various Gaussian distributions. The parameters of a Gaussian mixture model include the parameters of the Gaussian distribution components $\mu_{1:n}, \sigma^2_{1:n}$, as well as their weights $\rho_{1:n}$. The density is given by

$$p(x \mid \mu_{1:n}, \sigma^2_{1:n}, \rho_{1:n}) = \sum_{i=1}^{n} \rho_i \mathcal{N}(x \mid \mu_i, \sigma_i^2) \tag{2.17}$$

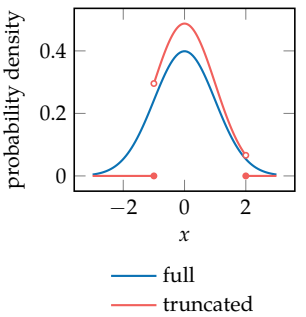
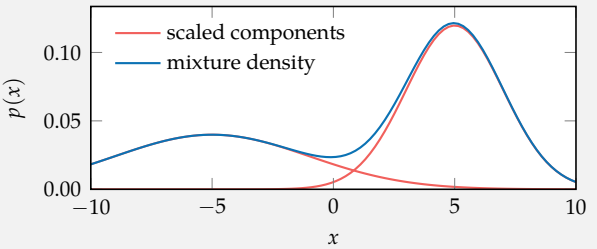


Figure 2.5. The probability density functions for a unit Gaussian distribution and the same distribution truncated between −1 and 2.

where the weights must sum to 1. Example 2.2 shows a Gaussian mixture model with two components.

Example 2.2. An example of a Gaussian mixture model.

We can create a Gaussian mixture model with components $\mu_1 = 5$, $\sigma_1 = 2$ and $\mu_2 = -5$, $\sigma_2 = 4$, weighted according to $\rho_1 = 0.6$ and $\rho_2 = 0.4$. Here we plot the density of two components scaled by their weights:



Another approach to representing multimodal continuous distributions is through discretization. For example, we can represent a distribution over a continuous variable as a *piecewise-uniform density*. The density is specified by the bin edges, and a probability mass is associated with each bin. Such a piecewise-uniform distribution is a type of mixture model where the components are uniform distributions.

2.3 Joint Distributions

A *joint distribution* is a probability distribution over multiple variables. A distribution over a single variable is called a *univariate distribution*, and a distribution over multiple variables is called a *multivariate distribution*. If we have a joint distribution over two discrete variables X and Y, then $P(x, y)$ denotes the probability that both $X = x$ and $Y = y$.

From a joint distribution, we can compute a *marginal* distribution of a variable or a set of variables by summing out all other variables using what is known as the *law of total probability*:[9]

$$P(x) = \sum_y P(x, y) \tag{2.18}$$

This property is used throughout this book.

[9] If our distribution is continuous, then we integrate out the other variables when marginalizing. For example:

$$p(x) = \int p(x, y)\, \mathrm{d}y$$

Real-world decision making often requires reasoning about joint distributions involving many variables. Sometimes there are complex relationships between the variables that are important to represent. We may use different strategies to represent joint distributions depending on whether the variables involve discrete or continuous values.

2.3.1 Discrete Joint Distributions

If the variables are discrete, the joint distribution can be represented by a table like the one shown in table 2.1. That table lists all the possible assignments of values to three binary variables. Each variable can only be 0 or 1, resulting in $2^3 = 8$ possible assignments. As with other discrete distributions, the probabilities in the table must sum to 1. It follows that although there are eight entries in the table, only seven of them are *independent*. If θ_i represents the probability in the ith row in the table, then we only need the parameters $\theta_1, \ldots, \theta_7$ to represent the distribution because we know that $\theta_8 = 1 - (\theta_1 + \ldots + \theta_7)$.

Table 2.1. Example of a joint distribution involving binary variables X, Y, and Z.

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

If we have n binary variables, then we need as many as $2^n - 1$ independent parameters to specify the joint distribution. This exponential growth in the number of parameters makes storing the distribution in memory difficult. In some cases, we can assume that our variables are *independent*, which means that the realization of one does not affect the probability distribution of the other. If X and Y are independent, which is sometimes written as $X \perp Y$, then we know that $P(x, y) = P(x)P(y)$ for all x and y. Suppose we have binary variables $X_1, \ldots, X_n$ that are all independent of each other, resulting in $P(x_{1:n}) = \prod_i P(x_i)$. This factorization allows us to represent this joint distribution with only n independent parameters instead of the $2^n - 1$ required when we cannot assume independence (see table 2.2). Independence can result in an enormous savings in terms of representational complexity, but it is often a poor assumption.

Table 2.2. If we know the variables in table 2.1 are independent, we can represent $P(x, y, z)$ using the product $P(x)P(y)P(z)$. This representation requires only one parameter for each of the three univariate distributions.

X	$P(X)$
0	0.85
1	0.15

Y	$P(Y)$
0	0.45
1	0.55

Z	$P(Z)$
0	0.20
1	0.80

We can represent joint distributions in terms of factors. A *factor* ϕ over a set of variables is a function from assignments of those variables to the real numbers. In order to represent a probability distribution, the real numbers in the factor must be nonnegative. A factor with nonnegative values can be normalized such that it represents a probability distribution. Algorithm 2.1 provides an implementation for discrete factors, and example 2.3 demonstrates how they work.

Another approach to reduce the storage required to represent joint distributions with repeated values is to use a *decision tree*. A decision tree involving three discrete

```
struct Variable
    name::Symbol
    r::Int # number of possible values
end

const Assignment = Dict{Symbol,Int}
const FactorTable = Dict{Assignment,Float64}

struct Factor
    vars::Vector{Variable}
    table::FactorTable
end

variablenames(ϕ::Factor) = [var.name for var in ϕ.vars]

select(a::Assignment, varnames::Vector{Symbol}) =
    Assignment(n⇒a[n] for n in varnames)

function assignments(vars::AbstractVector{Variable})
    names = [var.name for var in vars]
    return vec([Assignment(n⇒v for (n,v) in zip(names, values))
                for values in product((1:v.r for v in vars)...)])
end

function normalize!(ϕ::Factor)
    z = sum(p for (a,p) in ϕ.table)
    for (a,p) in ϕ.table
        ϕ.table[a] = p/z
    end
    return ϕ
end
```

Algorithm 2.1. Types and functions relevant to working with factors over a set of discrete variables. A variable is given a name (represented as a symbol) and may take on an integer from 1 to m. An assignment is a mapping from variable names to values represented as integers. A factor is defined by a factor table, which assigns values to different assignments involving a set of variables and is a mapping from assignments to real values. This mapping is represented by a dictionary. Any assignments not contained in the dictionary are set to 0. Also included in this algorithm block are some utility functions for returning the variable names associated with a factor, selecting a subset of an assignment, enumerating possible assignments, and normalizing factors. As discussed in appendix G.3.3, `product` produces the Cartesian product of a set of collections. It is imported from `Base.Iterators`.

We can instantiate the table from table 2.1 using the `Factor` type using the following code:

```
# requires convenience functions from appendix G.5
X = Variable(:x, 2)
Y = Variable(:y, 2)
Z = Variable(:z, 2)
ϕ = Factor([X, Y, Z], FactorTable(
    (x=1, y=1, z=1) ⇒ 0.08, (x=1, y=1, z=2) ⇒ 0.31,
    (x=1, y=2, z=1) ⇒ 0.09, (x=1, y=2, z=2) ⇒ 0.37,
    (x=2, y=1, z=1) ⇒ 0.01, (x=2, y=1, z=2) ⇒ 0.05,
    (x=2, y=2, z=1) ⇒ 0.02, (x=2, y=2, z=2) ⇒ 0.07,
))
```

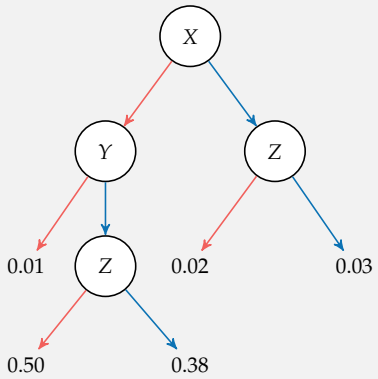
Example 2.3. Constructing a discrete factor. *The construction of the factor table using named tuples takes advantage of the utility functions defined in appendix G.5.*

variables is shown in example 2.4. Although the savings in this example in terms of the number of parameters may not be significant, it can become quite substantial when there are many variables and many repeated values.

Example 2.4. A decision tree can be a more efficient representation of a joint distribution than a table.

Suppose we have the following table representing a joint probability distribution. We can use the decision tree to the right of it to more compactly represent the values in the table. Red arrows are followed when a variable is 0, and blue arrows are followed when a variable is 1. Instead of storing eight probabilities, we store only five, along with a representation of the tree.

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.01
0	0	1	0.01
0	1	0	0.50
0	1	1	0.38
1	0	0	0.02
1	0	1	0.03
1	1	0	0.02
1	1	1	0.03



2.3.2 *Continuous Joint Distributions*

We can also define joint distributions over continuous variables. A rather simple distribution is the *multivariate uniform distribution*, which assigns a constant probability density everywhere there is support. We can use $\mathcal{U}(\mathbf{a}, \mathbf{b})$ to represent a uniform distribution over a *box*, which is a Cartesian product of intervals, with the ith interval being $[a_i, b_i]$. This family of uniform distributions is a special type of *multivariate product distribution*, which is a distribution defined in terms of the product of univariate distributions. In this case,

$$\mathcal{U}(\mathbf{x} \mid \mathbf{a}, \mathbf{b}) = \prod_i \mathcal{U}(x_i \mid a_i, b_i) \tag{2.19}$$

We can create a mixture model from a weighted collection of multivariate uniform distributions, just as we can with univariate distributions. If we have a joint distribution over n variables and k mixture components, we need to define $k(2n+1)-1$ independent parameters. For each of the k components, we need to define the upper and lower bounds for each of the variables as well as their weights. We can subtract 1 because the weights must sum to 1. Figure 2.6 shows an example that can be represented by five components.

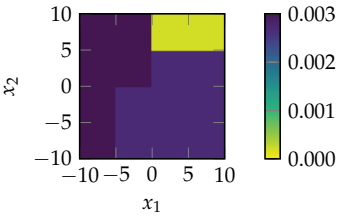


Figure 2.6. A density function for a mixture of multivariate uniform distributions.

It is also common to represent piecewise constant density functions by discretizing each of the variables independently. The discretization is represented by a set of bin edges for each variable. These bin edges define a grid over the variables. We then associate a constant probability density with each grid cell. The bin edges do not have to be uniformly separated. In some cases, it may be desirable to have increased resolution around certain values. Different variables might have different bin edges associated with them. If there are n variables and m bins for each variable, then we need $m^n - 1$ independent parameters to define the distribution—in addition to the values that define the bin edges.

In some cases, it may be more memory efficient to represent a continuous joint distribution as a decision tree in a manner similar to what we discussed for discrete joint distributions. The internal nodes compare variables against thresholds and the leaf nodes are density values. Figure 2.7 shows a decision tree that represents the density function in figure 2.6.

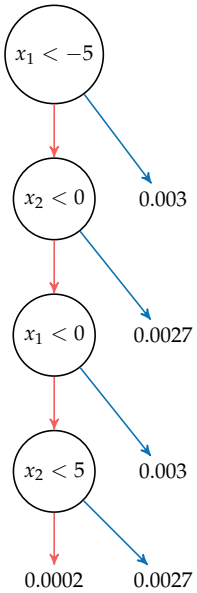


Figure 2.7. An example of a decision tree that represents a piecewise constant joint probability density defined over x_1 and x_2 over the interval $[-10, 10]^2$.

Another useful distribution is the *multivariate Gaussian distribution* with the density function

$$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{2.20}$$

where $\mathbf{x}$ is in $\mathbb{R}^n$, $\boldsymbol{\mu}$ is the *mean vector*, and $\boldsymbol{\Sigma}$ is the *covariance matrix*. The density function given here requires that $\boldsymbol{\Sigma}$ be *positive definite*.[10] The number of independent parameters is equal to $n + (n+1)n/2$, the number of components in $\boldsymbol{\mu}$ added to the number of components in the upper triangle of matrix $\boldsymbol{\Sigma}$.[11] Appendix B shows plots of different multivariate Gaussian density functions. We can also define *multivariate Gaussian mixture models*. Figure 2.8 shows an example of one with three components.

If we have a multivariate Gaussian with all the variables independent, then the covariance matrix $\boldsymbol{\Sigma}$ is diagonal with only n independent parameters. In fact,

[10] This definition is reviewed in appendix A.5.

[11] If we know the parameters in the upper triangle of $\boldsymbol{\Sigma}$, we know the parameters in the lower triangle as well, because $\boldsymbol{\Sigma}$ is symmetric.

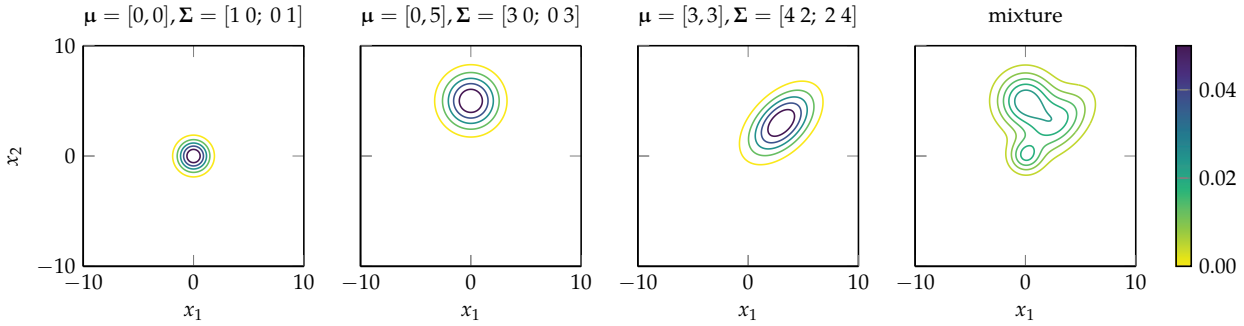


Figure 2.8. Multivariate Gaussian mixture model with three components. The components are mixed together with the weights 0.1, 0.5, and 0.4, respectively.

we can write the density function as a product of univariate Gaussian densities:

$$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_i \mathcal{N}(x_i \mid \mu_i, \Sigma_{ii}) \tag{2.21}$$

2.4 Conditional Distributions

The previous section introduced the idea of independence, which can help reduce the number of parameters used to define a joint distribution. However, as was mentioned, independence can be too strong of an assumption. This section will introduce the idea of conditional independence, which can help reduce the number of independent parameters without making assumptions that are as strong. Before discussing conditional independence, we will first introduce the notion of a *conditional distribution*, which is a distribution over a variable given the value of one or more other ones.

The definition of *conditional probability* states that

$$P(x \mid y) = \frac{P(x, y)}{P(y)} \tag{2.22}$$

where $P(x \mid y)$ is read as "probability of x given y." In some contexts, it is common to refer to y as *evidence*.

Since a conditional probability distribution is a probability distribution over one or more variables given some evidence, we know that

$$\sum_x P(x \mid y) = 1 \tag{2.23}$$

for a discrete X. If X is continuous, it integrates to 1.

We can incorporate the definition of conditional probability into equation (2.18) to obtain a slightly different form of the law of total probability:

$$P(x) = \sum_y P(x \mid y)P(y) \tag{2.24}$$

for a discrete distribution.

Another useful relationship that follows from the definition of conditional probability is *Bayes' rule*:[12]

$$P(x \mid y) = \frac{P(y \mid x)P(x)}{P(y)} \tag{2.25}$$

If we have a representation of a conditional distribution $P(y \mid x)$, we can apply Bayes' rule to swap y and x to obtain the conditional distribution $P(x \mid y)$.

We will now discuss a variety of ways to represent conditional probability distributions over discrete and continuous variables.

[12] Named for the English statistician and Presbyterian minister Thomas Bayes (c. 1701–1761) who provided a formulation of this theorem. A history is provided by S. B. McGrayne, *The Theory That Would Not Die*. Yale University Press, 2011.

2.4.1 Discrete Conditional Models

A conditional probability distribution over discrete variables can be represented using a table. In fact, we can use the same discrete factor representation that we used in section 2.3.1 for joint distributions. Table 2.3 shows an example of a table representing $P(X \mid Y, Z)$ with all binary variables. In contrast with a joint table (e.g., table 2.1), the column containing the probabilities need not sum to 1. However, if we sum over the probabilities that are consistent with what we are conditioning on, we must get 1. For example, conditioning on y^0 and z^0 (the evidence), we have

$$P(x^0 \mid y^0, z^0) + P(x^1 \mid y^0, z^0) = 0.08 + 0.92 = 1 \tag{2.26}$$

Conditional probability tables can become quite large. If we were to create a table like table 2.3, in which all variables can take on m values and we are conditioning on n variables, there would be m^{n+1} rows. However, since the m values of the variable we are not conditioning on must sum to 1, there are only $(m-1)m^n$ independent parameters. There is still an exponential growth in the number of variables on which we condition. When there are many repeated values in the conditional probability table, a decision tree (introduced in section 2.3.1) may be a more efficient representation.

Table 2.3. An example of a conditional distribution involving the binary variables X, Y, and Z.

X	Y	Z	$P(X \mid Y, Z)$
0	0	0	0.08
0	0	1	0.15
0	1	0	0.05
0	1	1	0.10
1	0	0	0.92
1	0	1	0.85
1	1	0	0.95
1	1	1	0.90

2.4.2 Conditional Gaussian Models

A *conditional Gaussian* model can be used to represent a distribution over a continuous variable given one or more discrete variables. For example, if we have a continuous variable X and a discrete variable Y with values $1:n$, we can define a conditional Gaussian model as follows:[13]

$$p(x \mid y) = \begin{cases} \mathcal{N}(x \mid \mu_1, \sigma_1^2) & \text{if } y^1 \\ \vdots & \\ \mathcal{N}(x \mid \mu_n, \sigma_n^2) & \text{if } y^n \end{cases} \tag{2.27}$$

with parameter vector $\theta = [\mu_{1:n}, \sigma_{1:n}]$. All $2n$ of those parameters can be varied independently. If we want to condition on multiple discrete variables, we just need to add more cases and associated parameters.

[13] This definition is for a mixture of univariate Gaussians, but the concept can be easily generalized to a mixture of multidimensional Gaussians.

2.4.3 Linear Gaussian Models

The *linear Gaussian* model of $P(X \mid Y)$ represents the distribution over a continuous variable X as a Gaussian distribution with the mean being a linear function of the value of the continuous variable Y. The conditional density function is

$$p(x \mid y) = \mathcal{N}(x \mid my + b, \sigma^2) \tag{2.28}$$

with parameters $\theta = [m, b, \sigma]$. The mean is a linear function of y defined by parameters m and b. The variance is constant. Figure 2.9 shows an example.

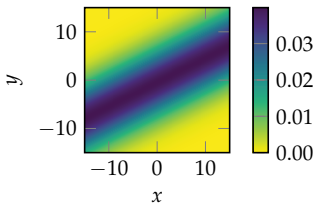


Figure 2.9. A linear Gaussian model with $p(x \mid y) = \mathcal{N}(x \mid 2y + 1, 10^2)$

2.4.4 Conditional Linear Gaussian Models

The *conditional linear Gaussian* model combines the ideas of conditional Gaussian and linear Gaussian models to be able to condition a continuous variable on both discrete and continuous variables. Suppose that we want to represent $p(X \mid Y, Z)$, where X and Y are continuous and Z is discrete with values $1:n$. The conditional density function is then

$$p(x \mid y, z) = \begin{cases} \mathcal{N}(x \mid m_1 y + b_1, \sigma_1^2) & \text{if } z^1 \\ \vdots & \\ \mathcal{N}(x \mid m_n y + b_n, \sigma_n^2) & \text{if } z^n \end{cases} \tag{2.29}$$

Here, the parameter vector $\theta = [m_{1:n}, b_{1:n}, \sigma_{1:n}]$ has $3n$ components.

2.4.5 Sigmoid Models

We can use a *sigmoid*[14] model to represent a distribution over a binary variable conditioned on a continuous variable. For example, we may want to represent $P(x^1 \mid y)$, where x is binary and y is continuous. Of course, we could just set a threshold θ and say that $P(x^1 \mid y) = 0$ if $y < \theta$, and $P(x^1 \mid y) = 1$ otherwise. However, in many applications, we may not want to have such a hard threshold that results in assigning zero probability to x^1 for certain values of y.

Instead of a hard threshold, we could use a *soft threshold*, which assigns low probabilities when below a threshold and high probabilities when above a threshold. One way to represent a soft threshold is to use a *logit model*, which produces a sigmoid curve:

$$P(x^1 \mid y) = \frac{1}{1 + \exp\left(-2\frac{y-\theta_1}{\theta_2}\right)} \tag{2.30}$$

The parameter θ_1 governs the location of the threshold, and θ_2 controls the ''softness'' or spread of the probabilities. Figure 2.10 shows a plot of $P(x^1 \mid y)$ with a logit model.

[14] A sigmoid is an S-shaped curve. There are different ways to define such a curve mathematically, but we will focus on the logit model.

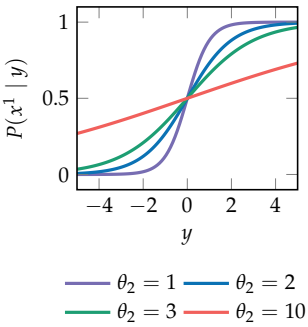


Figure 2.10. The logit model with $\theta_1 = 0$ and different values for θ_2.

2.4.6 Deterministic Variables

Some problems may involve a *deterministic variable*, whose value is fixed given evidence. In other words, we assign probability 1 to a value that is a deterministic function of its evidence. Using a conditional probability table to represent a discrete deterministic variable is possible, but it is wasteful. A single variable instantiation will have probability 1 for each parental instantiation, and the remaining entries will be 0. Our implementation can take advantage of this sparsity for a more compact representation. Algorithms in this text using discrete factors treat any assignments missing from the factor table as having value 0, making it so that we have to store only the assignments that have nonzero probability.

2.5 Bayesian Networks

A *Bayesian network* can be used to represent a joint probability distribution.[15] The structure of a Bayesian network is defined by a *directed acyclic graph* consisting of nodes and directed edges.[16] Each node corresponds to a variable. Directed edges connect pairs of nodes, with cycles in the graph being prohibited. The directed

[15] For an in-depth treatment of Bayesian networks and other forms of probabilistic graphical models, see D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[16] Appendix A.16 reviews common graph terminology.

edges indicate direct probabilistic relationships.[17] Associated with each node X_i is a conditional distribution $P(X_i \mid \text{Pa}(X_i))$, where $\text{Pa}(X_i)$ represents the parents of X_i in the graph. Algorithm 2.2 provides an implementation of a Bayesian network data structure. Example 2.5 illustrates the application of Bayesian networks to a satellite-monitoring problem.

[17] In *causal networks*, the direction of the edges indicate causal relationships between variables. However, causality is not required in general Bayesian networks. J. Pearl, *Causality: Models, Reasoning, and Inference*, 2nd ed. Cambridge University Press, 2009.

```
struct BayesianNetwork
    vars::Vector{Variable}
    factors::Vector{Factor}
    graph::SimpleDiGraph{Int64}
end
```

Algorithm 2.2. A discrete Bayesian network representation in terms of a set of variables, factors, and a graph. The graph data structure is provided by `Graphs.jl`.

The *chain rule* for Bayesian networks specifies how to construct a joint distribution from the local conditional probability distributions. Suppose that we have the variables $X_{1:n}$ and want to compute the probability of a particular assignment of all these variables to values $P(x_{1:n})$. The chain rule says

$$P(x_{1:n}) = \prod_{i=1}^{n} P(x_i \mid \text{pa}(x_i)) \tag{2.31}$$

where $\text{pa}(x_i)$ is the particular assignment of the parents of X_i to their values. Algorithm 2.3 provides an implementation for Bayesian networks with conditional probability distributions represented as discrete factors.

```
function probability(bn::BayesianNetwork, assignment)
    subassignment(ϕ) = select(assignment, variablenames(ϕ))
    probability(ϕ) = get(ϕ.table, subassignment(ϕ), 0.0)
    return prod(probability(ϕ) for ϕ in bn.factors)
end
```

Algorithm 2.3. A function for evaluating the probability of an assignment given a Bayesian network `bn`. For example, if `bn` is as defined in example 2.5, then `a = (b=1,s=1,e=1,d=2,c=1)` `probability(bn, Assignment(a))` returns `0.034228655999999996`.

In the satellite example, suppose we want to compute the probability that nothing is wrong; that is, $P(b^0, s^0, e^0, d^0, c^0)$. From the chain rule,

$$P(b^0, s^0, e^0, d^0, c^0) = P(b^0)P(s^0)P(e^0 \mid b^0, s^0)P(d^0 \mid e^0)P(c^0 \mid e^0) \tag{2.32}$$

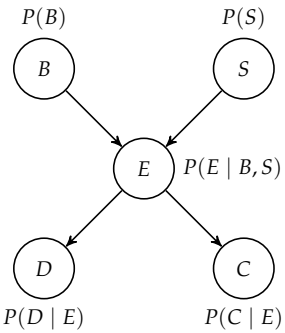
If we had fully specified a joint distribution over the five variables B, S, E, D, and C, then we would have needed $2^5 - 1 = 31$ independent parameters. The structure assumed in our Bayesian network allows us to specify the joint distribution using only $1 + 1 + 4 + 2 + 2 = 10$ independent parameters. The difference between

In the margin is a Bayesian network for a satellite-monitoring problem involving five binary variables. Fortunately, battery failure and solar panel failures are both rare, although solar panel failures are somewhat more likely than battery failures. Failures in either can lead to electrical system failure. There may be causes of electrical system failure other than battery or solar panel failure, such as a problem with the power management unit. An electrical system failure can result in trajectory deviation, which can be observed from the Earth by telescope, as well as a communication loss that interrupts the transmission of telemetry and mission data down to various ground stations. Other anomalies not involving the electrical system can result in trajectory deviation and communication loss.

Associated with each of the five variables are five conditional probability distributions. Because B and S have no parents, we only need to specify $P(B)$ and $P(S)$. The code here creates a Bayesian network structure with example values for the elements of the associated factor tables. The tuples in the factor tables index into the domains of the variables, which is $\{0, 1\}$ for all the variables. For example, `(e=2,b=1,s=1)` corresponds to (e^1, b^0, s^0).

```
# requires convenience functions from appendix G.5
B = Variable(:b, 2); S = Variable(:s, 2)
E = Variable(:e, 2)
D = Variable(:d, 2); C = Variable(:c, 2)
vars = [B, S, E, D, C]
factors = [
    Factor([B], FactorTable((b=1,) ⇒ 0.99, (b=2,) ⇒ 0.01)),
    Factor([S], FactorTable((s=1,) ⇒ 0.98, (s=2,) ⇒ 0.02)),
    Factor([E,B,S], FactorTable(
        (e=1,b=1,s=1) ⇒ 0.90, (e=1,b=1,s=2) ⇒ 0.04,
        (e=1,b=2,s=1) ⇒ 0.05, (e=1,b=2,s=2) ⇒ 0.01,
        (e=2,b=1,s=1) ⇒ 0.10, (e=2,b=1,s=2) ⇒ 0.96,
        (e=2,b=2,s=1) ⇒ 0.95, (e=2,b=2,s=2) ⇒ 0.99)),
    Factor([D, E], FactorTable(
        (d=1,e=1) ⇒ 0.96, (d=1,e=2) ⇒ 0.03,
        (d=2,e=1) ⇒ 0.04, (d=2,e=2) ⇒ 0.97)),
    Factor([C, E], FactorTable(
        (c=1,e=1) ⇒ 0.98, (c=1,e=2) ⇒ 0.01,
        (c=2,e=1) ⇒ 0.02, (c=2,e=2) ⇒ 0.99))
]
graph = SimpleDiGraph(5)
add_edge!(graph, 1, 3); add_edge!(graph, 2, 3)
add_edge!(graph, 3, 4); add_edge!(graph, 3, 5)
bn = BayesianNetwork(vars, factors, graph)
```

Example 2.5. A Bayesian network representing a satellite-monitoring problem. Here is the structure of the network represented as a directed acyclic graph. Associated with each node is a conditional probability distribution.



B battery failure
S solar panel failure
E electrical system failure
D trajectory deviation
C communication loss

10 and 31 does not represent an especially significant savings in the number of parameters, but the savings can become enormous in larger Bayesian networks. The power of Bayesian networks comes from their ability to reduce the number of parameters required to specify a joint probability distribution.

2.6 Conditional Independence

The reason that a Bayesian network can represent a joint distribution with fewer independent parameters than would normally be required is the *conditional independence* assumptions encoded in its graphical structure.[18] Conditional independence is a generalization of the notion of independence introduced in section 2.3.1. Variables X and Y are conditionally independent given Z if and only if $P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. The assertion that X and Y are conditionally independent given Z is written as $(X \perp Y \mid Z)$. It is possible to show from this definition that $(X \perp Y \mid Z)$ if and only if $P(X \mid Z) = P(X \mid Y, Z)$. Given Z, information about Y provides no additional information about X, and vice versa. Example 2.6 shows an instance of this.

[18] If the conditional independence assumptions made by the Bayesian network are invalid, then we run the risk of not properly modeling the joint distribution, as will be discussed in chapter 5.

Example 2.6. Conditional independence in the satellite-tracking problem.

Suppose the presence of satellite trajectory deviation (D) is conditionally independent of whether we have a communication loss (C) given knowledge of whether we have an electrical system failure (E). We would write this $(D \perp C \mid E)$. If we know that we have an electrical system failure, then the fact that we observe a loss of communication has no impact on our belief that there is a trajectory deviation. We may have an elevated expectation that there is a trajectory deviation, but that is only because we know that an electrical system failure has occurred.

We can use a set of rules to determine whether the structure of a Bayesian network implies that two variables must be conditionally independent given a set of other evidence variables.[19] Suppose that we want to check whether $(A \perp B \mid \mathcal{C})$ is implied by the network structure, where $\mathcal{C}$ is a set of evidence variables. We have to check all possible undirected paths from A to B for what is called *d-separation*. A path between A and B is d-separated by $\mathcal{C}$ if any of the following is true:

[19] Even if the structure of a network does not imply conditional independence, there may still be conditional independence due to the choice of conditional probability distributions. See exercise 2.10.

1. The path contains a *chain* of nodes, $X \rightarrow Y \rightarrow Z$, such that Y is in $\mathcal{C}$.

2. The path contains a *fork*, $X \leftarrow Y \rightarrow Z$, such that Y is in $\mathcal{C}$.

3. The path contains an *inverted fork* (also called a *v-structure*), $X \rightarrow Y \leftarrow Z$, such that Y is *not* in $\mathcal{C}$ and no descendant of Y is in $\mathcal{C}$. Example 2.7 provides some intuition for this rule.

We say that A and B are d-separated by $\mathcal{C}$ if all the paths between A and B are d-separated by $\mathcal{C}$. This d-separation implies that $(A \perp B \mid \mathcal{C})$.[20] Example 2.8 demonstrates this process for checking whether a graph implies a particular conditional independence assumption.

[20] An algorithm for efficiently determining d-separation is a bit complicated. See algorithm 3.1 in D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

Example 2.7. Intuition behind conditional independence assumptions implied (and not implied) in chains, forks, and inverted forks.

If we have $X \rightarrow Y \rightarrow Z$ (chain) or $X \leftarrow Y \rightarrow Z$ (fork) with evidence at Y, then X and Z are conditionally independent, meaning that $P(X \mid Y, Z) = P(X \mid Y)$. Interestingly, if the directions of the arrows were slightly different, with $X \rightarrow Y \leftarrow Z$ (inverted fork), then X and Z may no longer be conditionally independent given Y. In other words, it may be the case that $P(B \mid E) \neq P(B \mid S, E)$. To provide some intuition, consider the inverted fork path from battery failure B to solar panel failure S via electrical system failure E. Suppose we know that we have an electrical failure. If we know that we do not have a battery failure, then we are more inclined to believe that we have a solar panel failure because it is an alternative cause of the electrical failure. Conversely, if we found out that we do have a battery failure, then our belief that we have a solar panel failure decreases. This effect is referred to as *explaining away*. Observing a solar panel failure explains away the cause of the electrical system failure.

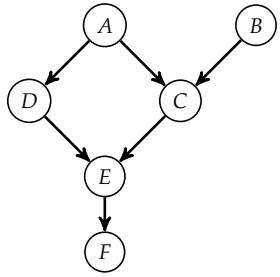
Sometimes the term *Markov blanket*[21] of node X is used to refer to the minimal set of nodes that, if their values were known, make X conditionally independent of all other nodes. A Markov blanket of a particular node turns out to consist of its parents, its children, and the other parents of its children.

[21] Named after the Russian mathematician Andrey Andreyevich Markov (1856–1922). J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

2.7 Summary

- Representing uncertainty as a probability distribution is motivated by a set of axioms related to the comparison of the plausibility of different statements.

Example 2.8. Conditional independence assumptions implied by the graphical structure below.



Suppose that we want to determine whether the network shown in the margin implies that $(D \perp B \mid F)$. There are two undirected paths from D to B. We need to check both paths for d-separation.

The path $D \leftarrow A \rightarrow C \leftarrow B$ involves the fork $D \leftarrow A \rightarrow C$, followed by an inverted fork, $A \rightarrow C \leftarrow B$. There is no evidence at A, so there is no d-separation from the fork. Since F is a descendant of C, there is no d-separation along the inverted fork. Hence, there is no d-separation along this path.

The second path, $D \rightarrow E \leftarrow C \leftarrow B$, involves the inverted fork $D \rightarrow E \leftarrow C$ and a chain, $E \leftarrow C \leftarrow B$. Since F is a descendant of E, there is no d-separation along the inverted fork. Because there is no d-separation along the chain part of this path either, there is no d-separation along this path from D to B.

For D and B to be conditionally independent given F, there must be d-separation along all undirected paths from D to B. In this case, neither of the two paths has d-separation. Hence, conditional independence is not implied by the network structure.

- There are many families of both discrete and continuous probability distributions.
- Continuous probability distributions can be represented by density functions.
- Probability distribution families can be combined in mixtures to create more flexible distributions.
- Joint distributions are distributions over multiple variables.
- Conditional distributions are distributions over one or more variables given the values of evidence variables.
- A Bayesian network is defined by a graphical structure and a set of conditional distributions.
- Depending on the structure of the Bayesian network, we can represent joint distributions with fewer parameters due to conditional independence assumptions.

2.8 Exercises

Exercise 2.1. Consider a continuous random variable X that follows the *exponential distribution* parameterized by λ with density $p(x \mid \lambda) = \lambda \exp(-\lambda x)$ with nonnegative support. Compute the cumulative distribution function of X.

Solution: We start with the definition of the cumulative distribution function. Since the support of the distribution is lower-bounded by $x = 0$, there is no probability mass in the interval $(-\infty, 0)$, allowing us to adjust the lower bound of the integral to 0. After computing the integral, we obtain $\text{cdf}_X(x)$:

$$\begin{aligned}
\text{cdf}_X(x) &= \int_{-\infty}^{x} p(x')\,\mathrm{d}x' \\
\text{cdf}_X(x) &= \int_{0}^{x} \lambda e^{-\lambda x'}\,\mathrm{d}x' \\
\text{cdf}_X(x) &= -e^{-\lambda x'}\Big|_0^x \\
\text{cdf}_X(x) &= 1 - e^{-\lambda x}
\end{aligned}$$

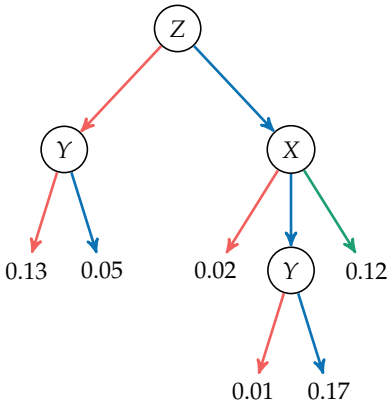
Exercise 2.2. For the density function in figure 2.6, what are the five components of the mixture? (There are multiple valid solutions.)

Solution: One solution is $\mathcal{U}([-10,-10],[-5,10])$, $\mathcal{U}([-5,0],[0,10])$, $\mathcal{U}([-5,-10],[0,0])$, $\mathcal{U}([0,-10],[10,5])$, and $\mathcal{U}([0,5],[10,10])$.

Exercise 2.3. Given the following table representation of $P(X, Y, Z)$, generate an equivalent compact decision tree representation:

X	Y	Z	$P(X,Y,Z)$
0	0	0	0.13
0	0	1	0.02
0	1	0	0.05
0	1	1	0.02
1	0	0	0.13
1	0	1	0.01
1	1	0	0.05
1	1	1	0.17
2	0	0	0.13
2	0	1	0.12
2	1	0	0.05
2	1	1	0.12

Solution: We start with the most common probabilities: 0.13, which occurs when $Z = 0$ and $Y = 0$, and 0.05, which occurs when $Z = 0$ and $Y = 1$. We choose to make Z the root of our decision tree, and when $Z = 0$, we continue to a Y node. Based on the value of Y, we branch to either 0.13 or 0.05. Next, we continue with cases when $Z = 1$. The most common probabilities are 0.02, which occurs when $Z = 1$ and $X = 0$, and 0.12, which occurs when $Z = 1$ and $X = 2$. So, when $Z = 1$, we choose to continue to an X node. Based on the whether X is 0, 1, or 2, we continue to 0.02, a Y node, or 0.12, respectively. Finally, based on the value of Y, we branch to either 0.01 or 0.17.



Exercise 2.4. Suppose that we want to specify a multivariate Gaussian mixture model with three components defined over four variables. We require that two of the three Gaussian distributions assume independence between the four variables, while the other Gaussian distribution is defined without any independence assumptions. How many independent parameters are required to specify this mixture model?

Solution: For a Gaussian distribution over four variables ($n = 4$) with independence assumptions, we need to specify $n + n = 2n = 8$ independent parameters; there are four parameters for the mean vector and four parameters for the covariance matrix (which is equivalent to the mean and variance parameters of four independent univariate Gaussian distributions). For a Gaussian distribution over four variables without independence assumptions, we need to specify $n + n(n+1)/2 = 14$ independent parameters; there are 4 parameters for the mean vector and 10 parameters for the covariance matrix. In addition, for our three mixture components ($k = 3$), we need to specify $k - 1 = 2$ independent parameters for the weights. Thus, we need $2(8) + 1(14) + 2 = 32$ independent parameters to specify this mixture distribution.

Exercise 2.5. We have three independent variables $X_{1:3}$ defined by piecewise-constant densities with 4, 7, and 3 bin edges, respectively. How many independent parameters are required to specify their joint distribution?

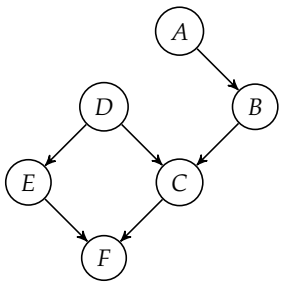
Solution: If we have a piecewise-constant density with m bin edges, then there are $m - 1$ bins and $m - 2$ independent parameters. For this problem, there will be $(4 - 2) + (7 - 2) + (3 - 2) = 8$ independent parameters.

Exercise 2.6. Suppose that we have four continuous random variables, X_1, X_2, Y_1, and Y_2, and we want to construct a linear Gaussian model of $X = X_{1:2}$ given $Y = Y_{1:2}$; that is, $p(X \mid Y)$. How many independent parameters are required for this model?

Solution: In this case, our mean vector for the Gaussian distribution is two-dimensional and requires four independent parameters for the transformation matrix $\mathbf{M}$ and two independent parameters for the bias vector $\mathbf{b}$. We also require three independent parameters for the covariance matrix $\mathbf{\Sigma}$. In total, we need $4 + 2 + 3 = 9$ independent parameters to specify this model:

$$p(\mathbf{x} \mid \mathbf{y}) = \mathcal{N}(\mathbf{x} \mid \mathbf{M}\mathbf{y} + \mathbf{b}, \mathbf{\Sigma})$$

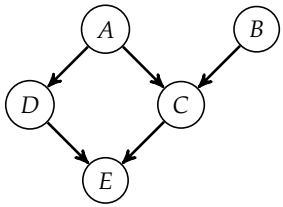
Exercise 2.7. Given the following Bayesian network, in which each node can take on one of four values, how many independent parameters are there? What is the percent reduction in the number of independent parameters required when using the following Bayesian network compared to using a full joint probability table?



Solution: The number of independent parameters for each node is equal to $(k - 1)k^m$, where k is the number of values that the node can take on and m is the number of parents that the node has. Variable A has 3, B has 12, C has 48, D has 3, E has 12, and F has 48 independent parameters. There are 126 total independent parameters for this Bayesian network.

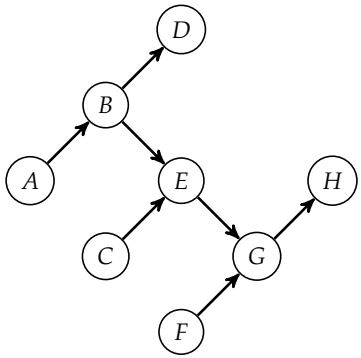
The number of independent parameters required to specify a joint probability table over n variables that can take on k values is equal to $k^n - 1$. Therefore, specifying a joint probability table would require $4^6 - 1 = 4096 - 1 = 4095$ independent parameters. The percent reduction in the number of independent parameters required is $(4095 - 126)/4095 \approx 96.9\,\%$.

Exercise 2.8. Given the following Bayesian network, is A d-separated from E, given C?



Solution: There are two paths from A to E: $A \to D \to E$ and $A \to C \to E$. There is d-separation along the second path, but not the first. Hence, A is not d-separated from E given C.

Exercise 2.9. Given the following Bayesian network, determine the Markov blanket of B:



Solution: Paths from B to A can only be d-separated given A. Paths from B to D can only be d-separated given D. Paths from B to E, and simultaneously F, G, and H, can be efficiently d-separated given E. Paths from B to C are naturally d-separated due to a v-structure; however, since E must be contained in our Markov blanket, paths from B to C given E can only be d-separated given C. So, the Markov blanket of B is $\{A, C, D, E\}$.

Exercise 2.10. In a Bayesian network with structure $A \to B$, is it possible for A to be independent of B?

Solution: There is a direct arrow from A to B, which indicates that independence is not implied. However, this does not mean that they are not independent. Whether A and B are independent depends on the choice of conditional probability tables. We can choose the tables so that there is independence. For example, suppose that both variables are binary and $P(a) = 0.5$ is uniform and $P(b \mid a) = 0.5$. Clearly, $P(A)P(B \mid A) = P(A)P(B)$, which means they are independent.

3 Inference

The previous chapter explained how to represent probability distributions. This chapter will show how to use these probabilistic representations for *inference*, which involves determining the distribution over one or more unobserved variables given the values associated with a set of observed variables. It begins by introducing exact inference methods. Because exact inference can be computationally intractable depending on the structure of the network, we will also discuss several algorithms for approximate inference.

3.1 Inference in Bayesian Networks

In inference problems, we want to infer a distribution over *query variables* given some observed *evidence variables*. The other nodes are referred to as *hidden variables*. We often refer to the distribution over the query variables, given the evidence, as a *posterior distribution*.

To illustrate the computations involved in inference, recall the Bayesian network from example 2.5, the structure of which is reproduced in figure 3.1. Suppose we have B as a query variable and evidence $D = 1$ and $C = 1$. The inference task is to compute $P(b^1 \mid d^1, c^1)$, which corresponds to computing the probability that we have a battery failure given an observed trajectory deviation and communication loss.

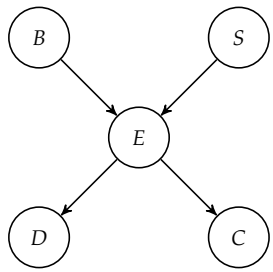


Figure 3.1. Bayesian network structure from example 2.5.

From the definition of conditional probability introduced in equation (2.22), we know that

$$P(b^1 \mid d^1, c^1) = \frac{P(b^1, d^1, c^1)}{P(d^1, c^1)} \tag{3.1}$$

To compute the numerator, we must use a process known as *marginalization*, where we sum out variables that are not involved (in this case S and E):

$$P(b^1, d^1, c^1) = \sum_s \sum_e P(b^1, s, e, d^1, c^1) \tag{3.2}$$

We know from the chain rule for Bayesian networks introduced in equation (2.31) that

$$P(b^1, s, e, d^1, c^1) = P(b^1)P(s)P(e \mid b^1, s)P(d^1 \mid e)P(c^1 \mid e) \tag{3.3}$$

All the components on the right side are specified in the conditional probability distributions associated with the nodes in the Bayesian network. We can compute the denominator in equation (3.1) using the same approach, but with an additional summation over the values for B.

This process of using the definition of conditional probability, marginalization, and applying the chain rule can be used to perform exact inference in any Bayesian network. We can implement exact inference using factors. Recall that factors represent discrete multivariate distributions. We use the following three operations on factors to achieve this:

- We use the *factor product* (algorithm 3.1) to combine two factors to produce a larger factor whose scope is the combined scope of the input factors. If we have $\phi(X, Y)$ and $\psi(Y, Z)$, then $\phi \cdot \psi$ will be over X, Y, and Z with $(\phi \cdot \psi)(x, y, z) = \phi(x, y)\psi(y, z)$. The factor product is demonstrated in example 3.1.
- We use *factor marginalization* (algorithm 3.2) to sum out a particular variable from the entire factor table, removing it from the resulting scope. Example 3.2 illustrates this process.
- We use *factor conditioning* (algorithm 3.3) with respect to some evidence to remove any rows in the table inconsistent with that evidence. Example 3.3 demonstrates factor conditioning.

These three factor operations are used together in algorithm 3.4 to perform exact inference. It starts by computing the product of all the factors, conditioning on the evidence, marginalizing out the hidden variables, and normalizing. One potential issue with this approach is the size of the product of all the factors. The size of the factor product is equal to the product of the number of values each variable can assume. For the satellite example problem, there are only $2^5 = 32$ possible assignments, but many interesting problems would have a factor product that is too large to enumerate practically.

```
function Base.:*(ϕ::Factor, ψ::Factor)
    ϕnames = variablenames(ϕ)
    ψnames = variablenames(ψ)
    ψonly = setdiff(ψ.vars, ϕ.vars)
    table = FactorTable()
    for (ϕa,ϕp) in ϕ.table
        for a in assignments(ψonly)
            a = merge(ϕa, a)
            ψa = select(a, ψnames)
            table[a] = ϕp * get(ψ.table, ψa, 0.0)
        end
    end
    vars = vcat(ϕ.vars, ψonly)
    return Factor(vars, table)
end
```

Algorithm 3.1. An implementation of the factor product, which constructs the factor representing the joint distribution of two smaller factors `ϕ` and `ψ`. If we want to compute the factor product of `ϕ` and `ψ`, we simply write `ϕ*ψ`.

The factor product of two factors produces a new factor over the union of their variables. Here, we produce a new factor from two factors that share a variable:

X	Y	$\phi_1(X,Y)$
0	0	0.3
0	1	0.4
1	0	0.2
1	1	0.1

Y	Z	$\phi_2(Y,Z)$
0	0	0.2
0	1	0.0
1	0	0.3
1	1	0.5

X	Y	Z	$\phi_3(X,Y,Z)$
0	0	0	0.06
0	0	1	0.00
0	1	0	0.12
0	1	1	0.20
1	0	0	0.04
1	0	1	0.00
1	1	0	0.03
1	1	1	0.05

Example 3.1. An illustration of a factor product combining two factors representing $\phi_1(X,Y)$ and $\phi_2(Y,Z)$ to produce a factor representing $\phi_3(X,Y,Z)$.

```
function marginalize(ϕ::Factor, name)
    table = FactorTable()
    for (a, p) in ϕ.table
        a′ = delete!(copy(a), name)
        table[a′] = get(table, a′, 0.0) + p
    end
    vars = filter(v -> v.name != name, ϕ.vars)
    return Factor(vars, table)
end
```

Algorithm 3.2. A method for marginalizing a variable named `name` from a factor `ϕ`.

Recall the joint probability distribution $P(X, Y, Z)$ from table 2.1. We can marginalize out Y by summing the probabilities in each row that have matching assignments for X and Z:

X	Y	Z	$\phi(X,Y,Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

X	Z	$\phi(X,Z)$
0	0	0.17
0	1	0.68
1	0	0.03
1	1	0.12

Example 3.2. A demonstration of factor marginalization.

```
in_scope(name, ϕ) = any(name == v.name for v in ϕ.vars)

function condition(ϕ::Factor, name, value)
    if !in_scope(name, ϕ)
        return ϕ
    end
    table = FactorTable()
    for (a, p) in ϕ.table
        if a[name] == value
            table[delete!(copy(a), name)] = p
        end
    end
    vars = filter(v -> v.name != name, ϕ.vars)
    return Factor(vars, table)
end

function condition(ϕ::Factor, evidence)
    for (name, value) in pairs(evidence)
        ϕ = condition(ϕ, name, value)
    end
    return ϕ
end
```

Algorithm 3.3. Two methods for factor conditioning given some evidence. The first takes a factor `ϕ` and returns a new factor whose table entries are consistent with the variable named `name` having the value `value`. The second takes a factor `ϕ` and applies evidence in the form of a named tuple. The `in_scope` method returns true if the variable named `name` is within the scope of the factor `ϕ`.

Factor conditioning involves dropping any rows inconsistent with the evidence. Here is the factor from table 2.1, and we condition on $Y = 1$. All rows for which $Y \neq 1$ are removed:

X	Y	Z	$\phi(X,Y,Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

$Y = 1$

X	Z	$\phi(X,Z)$
0	0	0.09
0	1	0.37
1	0	0.02
1	1	0.07

Example 3.3. An illustration of setting evidence, in this case for Y, in a factor. The resulting values must be renormalized.

```
struct ExactInference end

function infer(M::ExactInference, bn, query, evidence)
    ϕ = prod(bn.factors)
    ϕ = condition(ϕ, evidence)
    for name in setdiff(variablenames(ϕ), query)
        ϕ = marginalize(ϕ, name)
    end
    return normalize!(ϕ)
end
```

Algorithm 3.4. A naive exact inference algorithm for a discrete Bayesian network `bn`, which takes as input a set of query variable names `query` and `evidence` associating values with observed variables. The algorithm computes a joint distribution over the query variables in the form of a factor. We introduce the `ExactInference` type to allow `infer` to be called with different inference methods, as shall be seen in the rest of this chapter.

3.2 *Inference in Naive Bayes Models*

The previous section presented a general method for performing exact inference in any Bayesian network. This section discusses how this same method can be used to solve *classification* problems for a special kind of Bayesian network structure known as a *naive Bayes* model. This structure is given in figure 3.2. An equivalent but more compact representation is shown in figure 3.3 using a *plate*, shown here as a rounded box. The $i = 1:n$ in the bottom of the box specifies that the i in the subscript of the variable name is repeated from 1 to n.

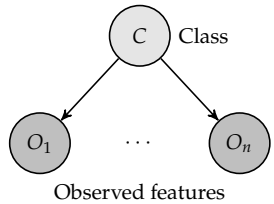


Figure 3.2. A naive Bayes model.

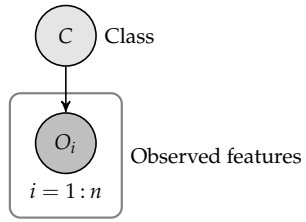


Figure 3.3. Plate representation of a naive Bayes model.

In the naive Bayes model, class C is the query variable, and the observed features $O_{1:n}$ are the evidence variables. The naive Bayes model is called naive because it assumes conditional independence between the evidence variables given the class. Using the notation introduced in section 2.6, we can say $(O_i \perp O_j \mid C)$ for all $i \neq j$. Of course, if these conditional independence assumptions do not hold, then we can add the necessary directed edges between the observed features.

We have to specify the *prior* $P(C)$ and the *class-conditional distributions* $P(O_i \mid C)$. As done in the previous section, we can apply the chain rule to compute the joint distribution:

$$P(c, o_{1:n}) = P(c) \prod_{i=1}^{n} P(o_i \mid c) \tag{3.4}$$

Our classification task involves computing the conditional probability $P(c \mid o_{1:n})$. From the definition of conditional probability, we have

$$P(c \mid o_{1:n}) = \frac{P(c, o_{1:n})}{P(o_{1:n})} \tag{3.5}$$

We can compute the denominator by marginalizing the joint distribution:

$$P(o_{1:n}) = \sum_c P(c, o_{1:n}) \tag{3.6}$$

The denominator in equation (3.5) is not a function of C and can therefore be treated as a constant. Hence, we can write

$$P(c \mid o_{1:n}) = \kappa P(c, o_{1:n}) \tag{3.7}$$

where κ is a *normalization constant* such that $\sum_c P(c \mid o_{1:n}) = 1$. We often drop κ and write

$$P(c \mid o_{1:n}) \propto P(c, o_{1:n}) \tag{3.8}$$

where the *proportional to* symbol $\propto$ is used to represent that the left side is proportional to the right side. Example 3.4 illustrates how inference can be applied to classifying radar tracks.

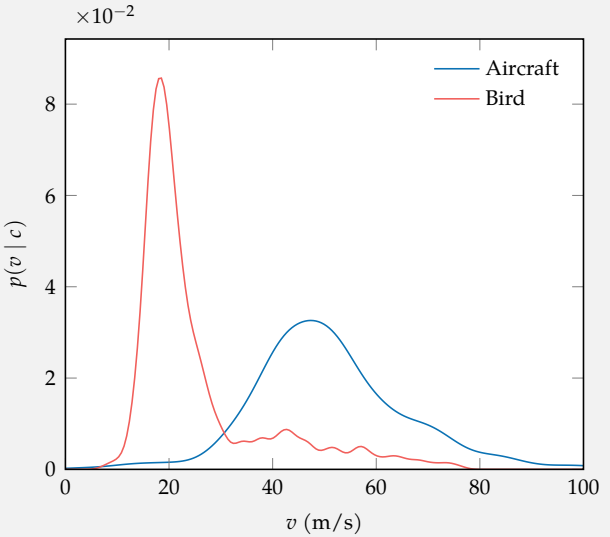
We can use this method to infer a distribution over classes, but for many applications, we have to commit to a particular class. It is common to classify according to the class with the highest posterior probability, $\arg\max_c P(c \mid o_{1:n})$. However, choosing a class is really a decision problem that often should take into account the consequences of misclassification. For example, if we are interested in using our classifier to filter out targets that are not aircraft for the purpose of air traffic control, then we can afford to occasionally let a few birds and other clutter tracks through our filter. However, we would want to avoid filtering out any real aircraft because that could lead to a collision. In this case, we would probably want to classify a track as a bird only if the posterior probability were close to 1. Decision problems will be discussed in chapter 6.

3.3 Sum-Product Variable Elimination

A variety of methods can be used to perform efficient inference in more complicated Bayesian networks. One method is known as *sum-product variable elimination*, which interleaves eliminating hidden variables (summations) with applications of the chain rule (products). It is more efficient to marginalize variables out as early as possible to avoid generating large factors.

Example 3.4. Radar target classification in which we want to determine whether a radar track corresponds to a bird or an aircraft.

Suppose that we have a radar track and we want to determine whether it was generated by a bird or an aircraft. We base our inference on airspeed and the amount of heading fluctuation. The first represents our belief about whether a target is a bird or an aircraft in the absence of any information about the track. Here are example class-conditional distributions for airspeed v as estimated from radar data:



Suppose from the chain rule, we determine:

$$P(\text{bird}, \text{slow}, \text{little heading fluctuation}) = 0.03$$
$$P(\text{aircraft}, \text{slow}, \text{little heading fluctuation}) = 0.01$$

Of course, these probabilities do not sum to 1. If we want to determine the probability that a target is a bird given the evidence, then we would make the following calculation:

$$P(\text{bird} \mid \text{slow}, \text{little heading fluctuation}) = \frac{0.03}{0.03 + 0.01} = 0.75$$

We will illustrate the variable elimination algorithm by computing the distribution $P(B \mid d^1, c^1)$ for the Bayesian network in figure 3.1. The conditional probability distributions associated with the nodes in the network can be represented by the following factors:

$$\phi_1(B), \phi_2(S), \phi_3(E, B, S), \phi_4(D, E), \phi_5(C, E) \tag{3.9}$$

Because D and C are observed variables, the last two factors can be replaced with $\phi_6(E)$ and $\phi_7(E)$ by setting the evidence $D = 1$ and $C = 1$.

We then proceed by eliminating the hidden variables in sequence. Different strategies can be used for choosing an ordering, but for this example, we arbitrarily choose the ordering E and then S. To eliminate E, we take the product of all the factors involving E and then marginalize out E to get a new factor:

$$\phi_8(B, S) = \sum_e \phi_3(e, B, S)\phi_6(e)\phi_7(e) \tag{3.10}$$

We can now discard ϕ_3, ϕ_6, and ϕ_7 because all the information we need from them is contained in ϕ_8.

Next, we eliminate S. Again, we gather all remaining factors that involve S and marginalize out S from the product of these factors:

$$\phi_9(B) = \sum_s \phi_2(s)\phi_8(B, s) \tag{3.11}$$

We discard ϕ_2 and ϕ_8 and are left with $\phi_1(B)$ and $\phi_9(B)$. Finally, we take the product of these two factors and normalize the result to obtain a factor representing $P(B \mid d^1, c^1)$.

This procedure is equivalent to computing the following:

$$P(B \mid d^1, c^1) \propto \phi_1(B) \sum_s \left(\phi_2(s) \sum_e \left(\phi_3(e \mid B, s)\phi_4(d^1 \mid e)\phi_5(c^1 \mid e) \right) \right) \tag{3.12}$$

This produces the same result as, but is more efficient than, the naive procedure of taking the product of all the factors and then marginalizing:

$$P(B \mid d^1, c^1) \propto \sum_s \sum_e \phi_1(B)\phi_2(s)\phi_3(e \mid B, s)\phi_4(d^1 \mid e)\phi_5(c^1 \mid e) \tag{3.13}$$

The sum-product variable elimination algorithm is implemented in algorithm 3.5. It takes as input a Bayesian network, a set of query variables, a list of observed values, and an ordering of the variables. We first set all observed values. Then, for each variable, we multiply all factors containing it and then marginalize that variable out. This new factor replaces the consumed factors, and we repeat the process for the next variable.

For many networks, variable elimination allows inference to be done in an amount of time that scales linearly with the size of the network, but it has exponential time complexity in the worst case. What influences the amount of computation is the variable elimination order. Choosing the optimal elimination order is *NP-hard*,[1] meaning that it cannot be done in polynomial time in the worst case (section 3.5). Even if we found the optimal elimination order, variable elimination can still require an exponential number of computations. Variable elimination heuristics generally try to minimize the number of variables involved in the intermediate factors generated by the algorithm.

[1] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of Finding Embeddings in a k-Tree," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 2, pp. 277–284, 1987.

```
struct VariableElimination
    ordering # array of variable indices
end

function infer(M::VariableElimination, bn, query, evidence)
    Φ = [condition(ϕ, evidence) for ϕ in bn.factors]
    for i in M.ordering
        name = bn.vars[i].name
        if name ∉ query
            inds = findall(ϕ->in_scope(name, ϕ), Φ)
            if !isempty(inds)
                ϕ = prod(Φ[inds])
                deleteat!(Φ, inds)
                ϕ = marginalize(ϕ, name)
                push!(Φ, ϕ)
            end
        end
    end
    return normalize!(prod(Φ))
end
```

Algorithm 3.5. An implementation of the sum-product variable elimination algorithm, which takes in a Bayesian network `bn`, a list of query variables `query`, and evidence `evidence`. The variables are processed in the order given by `ordering`.

3.4 Belief Propagation

An approach to inference known as *belief propagation* works by propagating ''messages'' through the network using the *sum-product algorithm* in order to compute the marginal distributions of the query variables.[2] Belief propagation requires linear time but provides an exact answer only if the network does not have undirected cycles. If the network has undirected cycles, then it can be converted to a tree by combining multiple variables into single nodes by using what is known as the *junction tree algorithm*. If the number of variables that have to be combined into any one node in the resulting network is small, then inference can be done efficiently. A variation of belief propagation known as *loopy belief propagation* can provide approximate solutions in networks with undirected cycles. Although this approach does not provide any guarantees and may not converge, it can work well in practice.[3]

[2] A tutorial on the sum-product algorithm with a discussion of its connections to many other algorithms developed in separate communities is provided by F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor Graphs and the Sum-Product Algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[3] Belief propagation and related algorithms are covered in detail by D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

3.5 Computational Complexity

We can show that inference in Bayesian networks is NP-hard by using an NP-complete problem called 3SAT.[4] It is easy to construct a Bayesian network from an arbitrary 3SAT problem. For example, consider the following 3SAT formula:[5]

$$F(x_1, x_2, x_3, x_4) = \begin{array}{lllllll} (& x_1 & \vee & x_2 & \vee & x_3 &) \\ (& \neg x_1 & \vee & \neg x_2 & \vee & x_3 &) \\ (& x_2 & \vee & \neg x_3 & \vee & x_4 &) \end{array} \begin{array}{l} \wedge \\ \wedge \\ \end{array} \tag{3.14}$$

where $\neg$ represents *logical negation* (''not''), $\wedge$ represents *logical conjunction* (''and''), and $\vee$ represents *logical disjunction* (''or''). The formula consists of a conjunction of *clauses*, which are disjunctions of what are called *literals*. A literal is simply a variable or its negation.

Figure 3.4 shows the corresponding Bayesian network representation. The variables are represented by $X_{1:4}$, and the clauses are represented by $C_{1:3}$. The distributions over the variables are uniform. The nodes representing clauses have as parents the participating variables. Because this is a 3SAT problem, each clause node has exactly three parents. Each clause node assigns probability 0 to assignments that do not satisfy the clause and probability 1 to all satisfying assignments. The remaining nodes assign probability 1 to true if all their parents

[4] G. F. Cooper, "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks," *Artificial Intelligence*, vol. 42, no. 2–3, pp. 393–405, 1990. The Bayesian network construction in this section follows that text. See appendix C for a brief review of complexity classes.

[5] This formula also appears in example C.3 in appendix C.

are true. The original problem is satisfiable if and only if $P(y^1) > 0$. Hence, inference in Bayesian networks is at least as hard as 3SAT.

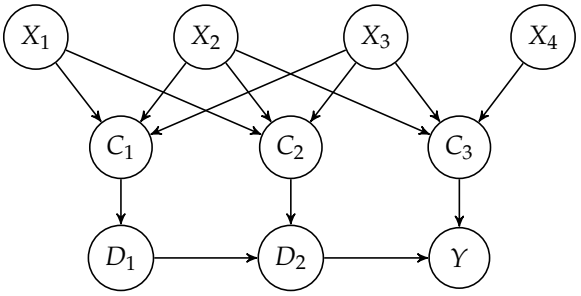


Figure 3.4. Bayesian network representing a 3SAT problem.

The reason we go to the effort of showing that inference in Bayesian networks is NP-hard is so that we know to avoid wasting time looking for an efficient, exact inference algorithm that works on all Bayesian networks. Therefore, research over the past couple of decades has focused on approximate inference methods, which are discussed next.

3.6 Direct Sampling

Motivated by the fact that exact inference is computationally intractable, many approximation methods have been developed. One of the simplest methods for inference is based on *direct sampling*, where random samples from the joint distribution are used to arrive at a probability estimate.[6] To illustrate this point, suppose that we want to infer $P(b^1 \mid d^1, c^1)$ from a set of n samples from the joint distribution $P(b, s, e, d, c)$. We use parenthetical superscripts to indicate the index of a sample, where we write $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$ for the ith sample. The direct sample estimate is

$$P(b^1 \mid d^1, c^1) \approx \frac{\sum_i (b^{(i)} = 1 \wedge d^{(i)} = 1 \wedge c^{(i)} = 1)}{\sum_i (d^{(i)} = 1 \wedge c^{(i)} = 1)} \tag{3.15}$$

We use the convention where a logical statement in parentheses is treated numerically as 1 when true and 0 when false. The numerator is the number of samples consistent with b, d, and c all set to 1, and the denominator is the number of samples consistent with d and c all set to 1.

[6] Sometimes approaches involving random sampling are referred to as *Monte Carlo methods*. The name comes from the Monte Carlo Casino in Monaco. An introduction to randomized algorithms and their application to a variety of problem domains is provided by R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

Sampling from the joint distribution represented by a Bayesian network is straightforward. The first step involves finding a *topological sort* of the nodes in the Bayesian network. A topological sort of the nodes in a directed acyclic graph is an ordered list such that if there is an edge $A \to B$, then A comes before B in the list.[7] For example, a topological sort for the network in figure 3.1 is B, S, E, D, C. A topological sort always exists, but it may not be unique. Another topological sort for the network is S, B, E, C, D.

[7] A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962. An implementation of topological sorting is provided by the `Graphs.jl` package.

Once we have a topological sort, we can begin sampling from the conditional probability distributions. Algorithm 3.6 shows how to sample from a Bayesian network given an ordering $X_{1:n}$ that represents a topological sort. We draw a sample from the conditional distribution associated with X_i given the values of the parents that have already been assigned. Because $X_{1:n}$ is a topological sort, we know that all the parents of X_i have already been instantiated, allowing this sampling to be done. Direct sampling is implemented in algorithm 3.7 and is demonstrated in example 3.5.

```
function Base.rand(ϕ::Factor)
    tot, p, w = 0.0, rand(), sum(values(ϕ.table))
    for (a,v) in ϕ.table
        tot += v/w
        if tot >= p
            return a
        end
    end
    return Assignment()
end

function Base.rand(bn::BayesianNetwork)
    a = Assignment()
    for i in topological_sort(bn.graph)
        name, ϕ = bn.vars[i].name, bn.factors[i]
        a[name] = rand(condition(ϕ, a))[name]
    end
    return a
end
```

Algorithm 3.6. A method for sampling an assignment from a Bayesian network `bn`. We also provide a method for sampling an assignment from a factor `ϕ`.

Suppose we draw 10 random samples from the network in figure 3.1. We are interested in inferring $P(b^1 \mid d^1, c^1)$. Only 2 of the 10 samples (pointed to in the table) are consistent with observations d^1 and c^1. One sample has $b = 1$, and the other sample has $b = 0$. From these samples, we infer that $P(b^1 \mid d^1, c^1) = 0.5$. Of course, we would want to use more than just 2 samples to accurately estimate $P(b^1 \mid d^1, c^1)$.

B	S	E	D	C	
0	0	1	1	0	
0	0	0	0	0	
1	0	1	0	0	
1	0	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	
0	0	0	0	1	
0	1	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	

Example 3.5. An example of how direct samples from a Bayesian network can be used for inference.

```
struct DirectSampling
    m # number of samples
end

function infer(M::DirectSampling, bn, query, evidence)
    table = FactorTable()
    for i in 1:(M.m)
        a = rand(bn)
        if all(a[k] == v for (k,v) in pairs(evidence))
            b = select(a, query)
            table[b] = get(table, b, 0) + 1
        end
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.7. The direct sampling inference method, which takes a Bayesian network `bn`, a list of query variables `query`, and evidence `evidence`. The method draws `m` samples from the Bayesian network and retains those samples that are consistent with the evidence. A factor over the query variables is returned. This method can fail if no samples that satisfy the evidence are found.

3.7 Likelihood Weighted Sampling

The problem with direct sampling is that we may waste time generating samples that are inconsistent with the observations, especially if the observations are unlikely. An alternative approach is called *likelihood weighted sampling*, which involves generating weighted samples that are consistent with the observations.

To illustrate, we will again attempt to infer $P(b^1 \mid d^1, c^1)$. We have a set of n samples, where the ith sample is again denoted $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$. The weight of the ith sample is w_i. The weighted estimate is

$$P(b^1 \mid d^1, c^1) \approx \frac{\sum_i w_i(b^{(i)} = 1 \land d^{(i)} = 1 \land c^{(i)} = 1)}{\sum_i w_i(d^{(i)} = 1 \land c^{(i)} = 1)} \tag{3.16}$$

$$= \frac{\sum_i w_i(b^{(i)} = 1)}{\sum_i w_i} \tag{3.17}$$

To generate these weighted samples, we begin with a topological sort and sample from the conditional distributions in sequence. The only difference in likelihood weighting is how we handle observed variables. Instead of sampling their values from a conditional distribution, we assign variables to their observed values and adjust the weight of the sample appropriately. The weight of a sample is simply the product of the conditional probabilities at the observed nodes. Likelihood weighted sampling is implemented in algorithm 3.8. Example 3.6 demonstrates inference with likelihood weighted sampling.

```
struct LikelihoodWeightedSampling
    m # number of samples
end

function infer(M::LikelihoodWeightedSampling, bn, query, evidence)
    table = FactorTable()
    ordering = topological_sort(bn.graph)
    for i in 1:(M.m)
        a, w = Assignment(), 1.0
        for j in ordering
            name, ϕ = bn.vars[j].name, bn.factors[j]
            if haskey(evidence, name)
                a[name] = evidence[name]
                w *= ϕ.table[select(a, variablenames(ϕ))]
            else
                a[name] = rand(condition(ϕ, a))[name]
            end
        end
        b = select(a, query)
        table[b] = get(table, b, 0) + w
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.8. The likelihood weighted sampling inference method, which takes a Bayesian network `bn`, a list of query variables `query`, and evidence `evidence`. The method draws `m` samples from the Bayesian network but sets values from evidence when possible, keeping track of the conditional probability when doing so. These probabilities are used to weight the samples such that the final inference estimate is accurate. A factor over the query variables is returned.

Example 3.6. Likelihood weighted samples from a Bayesian network.

The table here shows five likelihood weighted samples from the network in figure 3.1. We sample from $P(B)$, $P(S)$, and $P(E \mid B, S)$, as we would with direct sampling. When we come to D and C, we assign $D = 1$ and $C = 1$. If the sample has $E = 1$, then the weight is $P(d^1 \mid e^1)P(c^1 \mid e^1)$; otherwise, the weight is $P(d^1 \mid e^0)P(c^1 \mid e^0)$. If we assume

$$P(d^1 \mid e^1)P(c^1 \mid e^1) = 0.95$$
$$P(d^1 \mid e^0)P(c^1 \mid e^0) = 0.01$$

then we may approximate from the samples in the table:

$$P(b^1 \mid d^1, c^1) = \frac{0.95}{0.95 + 0.95 + 0.01 + 0.01 + 0.95} \approx 0.331$$

B	S	E	D	C	Weight
1	0	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$
0	1	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$
0	0	0	1	1	$P(d^1 \mid e^0)P(c^1 \mid e^0)$
0	0	0	1	1	$P(d^1 \mid e^0)P(c^1 \mid e^0)$
0	0	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$

Although likelihood weighting makes it so that all samples are consistent with the observations, it can still be wasteful. Consider the simple chemical detection Bayesian network shown in figure 3.5, and assume that we detected a chemical of interest. We want to infer $P(c^1 \mid d^1)$. Because this network is small, we can easily compute this probability exactly by using Bayes' rule:

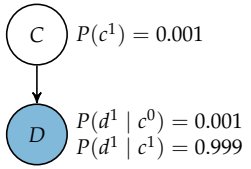


Figure 3.5. Chemical detection Bayesian network, with C indicating whether the chemical is present and D indicating whether the chemical is detected.

$$P(c^1 \mid d^1) = \frac{P(d^1 \mid c^1)P(c^1)}{P(d^1 \mid c^1)P(c^1) + P(d^1 \mid c^0)P(c^0)} \tag{3.18}$$

$$= \frac{0.999 \times 0.001}{0.999 \times 0.001 + 0.001 \times 0.999} \tag{3.19}$$

$$= 0.5 \tag{3.20}$$

If we use likelihood weighting, then 99.9 % of the samples will have $C = 0$, with a weight of 0.001. Until we get a sample of $C = 1$, which has an associated weight of 0.999, our estimate of $P(c^1 \mid d^1)$ will be 0.

3.8 Gibbs Sampling

An alternative approach to inference is to use *Gibbs sampling*,[8] which is a kind of *Markov chain Monte Carlo* technique. Gibbs sampling involves drawing samples consistent with the evidence in a way that does not involve weighting. From these samples, we can infer the distribution over the query variables.

[8] Named for the American scientist Josiah Willard Gibbs (1839–1903), who, with James Clerk Maxwell and Ludwig Boltzman, created the field of statistical mechanics.

Gibbs sampling involves generating a sequence of samples, starting with an initial sample, $x_{1:n}^{(1)}$, generated randomly with the evidence variables set to their observed values. The kth sample $x_{1:n}^{(k)}$ depends probabilistically on the previous sample, $x_{1:n}^{(k-1)}$. We modify $x_{1:n}^{(k-1)}$ in place to obtain $x_{1:n}^{(k)}$ as follows. Using any ordering of the unobserved variables, which need not be a topological sort, $x_i^{(k)}$ is sampled from the distribution represented by $P(X_i \mid x_{-i}^{(k)})$. Here, $x_{-i}^{(k)}$ represents the values of all other variables except X_i in sample k. Sampling from $P(X_i \mid x_{-i}^{(k)})$ can be done efficiently because we only need to consider the Markov blanket of variable X_i (see section 2.6).

Unlike the other sampling methods discussed so far, the samples produced by this method are not independent. However, it can be proven that, in the limit, samples are drawn exactly from the joint distribution over the unobserved

variables given the observations. Algorithm 3.9 shows how to compute a factor for $P(X_i \mid x_{-i})$. Gibbs sampling is implemented in algorithm 3.10.

```
function blanket(bn, a, i)
    name = bn.vars[i].name
    val = a[name]
    a = delete!(copy(a), name)
    Φ = filter(ϕ -> in_scope(name, ϕ), bn.factors)
    ϕ = prod(condition(ϕ, a) for ϕ in Φ)
    return normalize!(ϕ)
end
```

Algorithm 3.9. A method for obtaining $P(X_i \mid x_{-i})$ for a Bayesian network `bn` given a current assignment `a`.

Gibbs sampling can be applied to our running example. We can use our m samples to estimate

$$P(b^1 \mid d^1, c^1) \approx \frac{1}{m} \sum_i (b^{(i)} = 1) \tag{3.21}$$

Figure 3.6 compares the convergence of the estimate of $P(c^1 \mid d^1)$ in the chemical detection network using direct, likelihood weighted, and Gibbs sampling. Direct sampling takes the longest to converge. The direct sampling curve has long periods during which the estimate does not change because samples are inconsistent with the observations. Likelihood weighted sampling converges faster in this example. Spikes occur when a sample is generated with $C = 1$, and then gradually decrease. Gibbs sampling, in this example, quickly converges to the true value of 0.5.

As mentioned earlier, Gibbs sampling, like other Markov chain Monte Carlo methods, produces samples from the desired distribution in the limit. In practice, we have to run Gibbs for some amount of time, called the *burn-in period*, before converging to a steady-state distribution. The samples produced during burn-in are normally discarded. If many samples are to be used from a single Gibbs sampling series, it is common to *thin* the samples by keeping only every hth sample because of potential correlation between samples.

```
function update_gibbs_sample!(a, bn, evidence, ordering)
    for i in ordering
        name = bn.vars[i].name
        if !haskey(evidence, name)
            b = blanket(bn, a, i)
            a[name] = rand(b)[name]
        end
    end
end

function gibbs_sample!(a, bn, evidence, ordering, m)
    for j in 1:m
        update_gibbs_sample!(a, bn, evidence, ordering)
    end
end

struct GibbsSampling
    m_samples # number of samples to use
    m_burnin  # number of samples to discard during burn-in
    m_skip    # number of samples to skip for thinning
    ordering  # array of variable indices
end

function infer(M::GibbsSampling, bn, query, evidence)
    table = FactorTable()
    a = merge(rand(bn), evidence)
    gibbs_sample!(a, bn, evidence, M.ordering, M.m_burnin)
    for i in 1:(M.m_samples)
        gibbs_sample!(a, bn, evidence, M.ordering, M.m_skip)
        b = select(a, query)
        table[b] = get(table, b, 0) + 1
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.10. Gibbs sampling implemented for a Bayesian network `bn` with evidence `evidence` and an ordering `ordering`. The method iteratively updates the assignment `a` for `m` iterations.

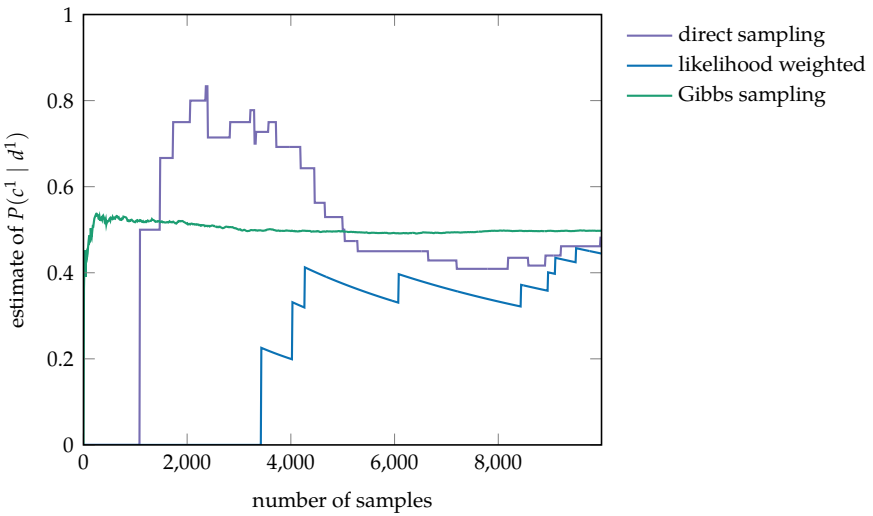


Figure 3.6. A comparison of sampling-based inference methods on the chemical detection network. Both likelihood weighted and direct sampling have poor convergence due to the rarity of events, whereas Gibbs sampling is able to converge to the true value efficiently, even with no burn-in period or thinning.

3.9 *Inference in Gaussian Models*

If the joint distribution is Gaussian, we can perform exact inference analytically. Two jointly Gaussian random variables $\mathbf{a}$ and $\mathbf{b}$ can be written

$$\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\boldsymbol{\mu}_\mathbf{a}\\ \boldsymbol{\mu}_\mathbf{b}\end{bmatrix}, \begin{bmatrix}\mathbf{A} & \mathbf{C}\\ \mathbf{C}^\top & \mathbf{B}\end{bmatrix}\right) \tag{3.22}$$

The marginal distribution of a multivariate Gaussian is also Gaussian:

$$\mathbf{a} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{a}, \mathbf{A}) \qquad \mathbf{b} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{b}, \mathbf{B}) \tag{3.23}$$

The conditional distribution of a multivariate Gaussian is also Gaussian, with a convenient closed-form solution:

$$p(\mathbf{a} \mid \mathbf{b}) = \mathcal{N}\left(\mathbf{a} \mid \boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}}, \boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}}\right) \tag{3.24}$$

$$\boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}} = \boldsymbol{\mu}_\mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_\mathbf{b}) \tag{3.25}$$

$$\boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top \tag{3.26}$$

Algorithm 3.11 shows how to use these equations to infer a distribution over a set of query variables given evidence. Example 3.7 illustrates how to extract the marginal and conditional distributions from a multivariate Gaussian.

```
function infer(D::MvNormal, query, evidencevars, evidence)
    μ, Σ = D.μ, D.Σ.mat
    b, μa, μb = evidence, μ[query], μ[evidencevars]
    A = Σ[query,query]
    B = Σ[evidencevars,evidencevars]
    C = Σ[query,evidencevars]
    μ = μa + C * (B\(b - μb))
    Σ = A - C * (B \ C')
    return MvNormal(μ, Σ)
end
```

Algorithm 3.11. Inference in a multivariate Gaussian distribution `D`. A vector of integers specifies the query variables in the `query` argument, and a vector of integers specifies the evidence variables in the `evidencevars` argument. The values of the evidence variables are contained in the vector `evidence`. The `Distributions.jl` package defines the `MvNormal` distribution.

Consider

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \right)$$

The marginal distribution for x_1 is $\mathcal{N}(0, 3)$, and the marginal distribution for x_2 is $\mathcal{N}(1, 2)$.

The conditional distribution for x_1 given $x_2 = 2$ is

$$\begin{aligned} \boldsymbol{\mu}_{x_1 \mid x_2 = 2} &= 0 + 1 \cdot 2^{-1} \cdot (2 - 1) = 0.5 \\ \boldsymbol{\Sigma}_{x_1 \mid x_2 = 2} &= 3 - 1 \cdot 2^{-1} \cdot 1 = 2.5 \\ x_1 \mid (x_2 = 2) &\sim \mathcal{N}(0.5, 2.5) \end{aligned}$$

We can perform this inference calculation using algorithm 3.11 by constructing the joint distribution

```
D = MvNormal([0.0,1.0],[3.0 1.0; 1.0 2.0])
```

and then calling `infer(D, [1], [2], [2.0])`.

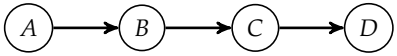
Example 3.7. Marginal and conditional distributions for a multivariate Gaussian.

3.10 Summary

- Inference involves determining the probability of query variables given some evidence.
- Exact inference can be done by computing the joint distribution over the variables, setting evidence, and marginalizing out any hidden variables.
- Inference can be done efficiently in naive Bayes models, in which a single parent variable affects many conditionally independent children.
- The variable elimination algorithm can make exact inference more efficient by marginalizing variables in sequence.
- Belief propagation represents another method for inference, in which information is iteratively passed between factors to arrive at a result.
- Inference in a Bayesian network can be shown to be NP-hard through a reduction to the 3SAT problem, motivating the development of approximate inference methods.
- Approximate inference can be done by directly sampling from the joint distribution represented by a Bayesian network, but it may involve discarding many samples that are inconsistent with the evidence.
- Likelihood weighted sampling can reduce computation required for approximate inference by only generating samples that are consistent with the evidence and weighting each sample accordingly.
- Gibbs sampling generates a series of unweighted samples that are consistent with the evidence and can greatly speed approximate inference.
- Exact inference can be done efficiently through matrix operations when the joint distribution is Gaussian.

3.11 Exercises

Exercise 3.1. Given the following Bayesian network and its associated conditional probability distributions, write the equation required to perform exact inference for the query $P(a^1 \mid d^1)$.



Solution: We first expand the inference expression using the definition of conditional probability.

$$P(a^1 \mid d^1) = \frac{P(a^1, d^1)}{P(d^1)}$$

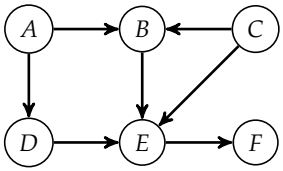
We can rewrite the numerator as a marginalization over the hidden variables and we can rewrite the denominator as a marginalization over both the hidden and query variables:

$$P(a^1 \mid d^1) = \frac{\sum_b \sum_c P(a^1, b, c, d^1)}{\sum_a \sum_b \sum_c P(a, b, c, d^1)}$$

The definition of the joint probability in both the numerator and the denominator can be rewritten using the chain rule for Bayesian networks and the resulting equation can be simplified by removing constants from inside the summations:

$$\begin{aligned} P(a^1 \mid d^1) &= \frac{\sum_b \sum_c P(a^1)P(b \mid a^1)P(c \mid b)P(d^1 \mid c)}{\sum_a \sum_b \sum_c P(a)P(b \mid a)P(c \mid b)P(d^1 \mid c)} \\ &= \frac{P(a^1)\sum_b \sum_c P(b \mid a^1)P(c \mid b)P(d^1 \mid c)}{\sum_a \sum_b \sum_c P(a)P(b \mid a)P(c \mid b)P(d^1 \mid c)} \\ &= \frac{P(a^1)\sum_b P(b \mid a^1)\sum_c P(c \mid b)P(d^1 \mid c)}{\sum_a P(a)\sum_b P(b \mid a)\sum_c P(c \mid b)P(d^1 \mid c)} \end{aligned}$$

Exercise 3.2. Given the following Bayesian network and its associated conditional probability distributions, write the equation required to perform an exact inference for the query $P(c^1, d^1 \mid a^0, f^1)$.



Solution: We first expand the inference expression using the definition of conditional probability:

$$P(c^1, d^1 \mid a^0, f^1) = \frac{P(a^0, c^1, d^1, f^1)}{P(a^0, f^1)}$$

We can rewrite the numerator as a marginalization over the hidden variables, and we can rewrite the denominator as a marginalization over both the hidden and query variables:

$$P(c^1, d^1 \mid a^0, f^1) = \frac{\sum_b \sum_e P(a^0, b, c^1, d^1, e, f^1)}{\sum_b \sum_c \sum_d \sum_e P(a^0, b, c, d, e, f^1)}$$

The definition of the joint probability in both the numerator and the denominator can be rewritten using the chain rule for Bayesian networks, and the resulting equation can be simplified by removing constants from inside the summations. Note that there are multiple possible orderings of the summations in the final equation:

$$\begin{aligned} P(c^1, d^1 \mid a^0, f^1) &= \frac{\sum_b \sum_e P(a^0) P(b \mid a^0, c^1) P(c^1) P(d^1 \mid a^0) P(e \mid b, c^1, d^1) P(f^1 \mid e)}{\sum_b \sum_c \sum_d \sum_e P(a^0) P(b \mid a^0, c) P(c) P(d \mid a^0) P(e \mid b, c, d) P(f^1 \mid e)} \\ &= \frac{P(a^0) P(c^1) P(d^1 \mid a^0) \sum_b \sum_e P(b \mid a^0, c^1) P(e \mid b, c^1, d^1) P(f^1 \mid e)}{P(a^0) \sum_b \sum_c \sum_d \sum_e P(b \mid a^0, c) P(c) P(d \mid a^0) P(e \mid b, c, d) P(f^1 \mid e)} \\ &= \frac{P(c^1) P(d^1 \mid a^0) \sum_b P(b \mid a^0, c^1) \sum_e P(e \mid b, c^1, d^1) P(f^1 \mid e)}{\sum_c P(c) \sum_b P(b \mid a^0, c) \sum_d P(d \mid a^0) \sum_e P(e \mid b, c, d) P(f^1 \mid e)} \end{aligned}$$

Exercise 3.3. Suppose that we are developing an object detection system for an autonomous vehicle driving in a city. Our vehicle's perception system reports an object's size S (either small, medium, or large) and speed V (either slow, moderate, or fast). We want to design a model that will determine the class C of an object—either a vehicle, pedestrian, or a ball—given observations of the object's size and speed. Assuming a naive Bayes model with the following class prior and class-conditional distributions, what is the detected class given observations S = medium and V = slow?

C	$P(C)$
vehicle	0.80
pedestrian	0.19
ball	0.01

C	S	$P(S \mid C)$
vehicle	small	0.001
vehicle	medium	0.009
vehicle	large	0.990
pedestrian	small	0.200
pedestrian	medium	0.750
pedestrian	large	0.050
ball	small	0.800
ball	medium	0.199
ball	large	0.001

C	V	$P(V \mid C)$
vehicle	slow	0.2
vehicle	moderate	0.2
vehicle	fast	0.6
pedestrian	slow	0.5
pedestrian	moderate	0.4
pedestrian	fast	0.1
ball	slow	0.4
ball	moderate	0.4
ball	fast	0.2

Solution: To compute the posterior distribution $P(c \mid o_{1:n})$, we use the definition of the joint distribution for a naive Bayes model in equation (3.4):

$$P(c \mid o_{1:n}) \propto P(c) \prod_{i=1}^{n} P(o_i \mid c)$$

$$P(\text{vehicle} \mid \text{medium}, \text{slow}) \propto P(\text{vehicle}) P(S = \text{medium} \mid \text{vehicle}) P(V = \text{slow} \mid \text{vehicle})$$

$$P(\text{vehicle} \mid \text{medium}, \text{slow}) \propto (0.80)(0.009)(0.2) = 0.00144$$

$$P(\text{pedestrian} \mid \text{medium}, \text{slow}) \propto P(\text{pedestrian}) P(S = \text{medium} \mid \text{pedestrian}) P(V = \text{slow} \mid \text{pedestrian})$$

$$P(\text{pedestrian} \mid \text{medium}, \text{slow}) \propto (0.19)(0.75)(0.5) = 0.07125$$

$$P(\text{ball} \mid \text{medium}, \text{slow}) \propto P(\text{ball}) P(S = \text{medium} \mid \text{ball}) P(V = \text{slow} \mid \text{ball})$$

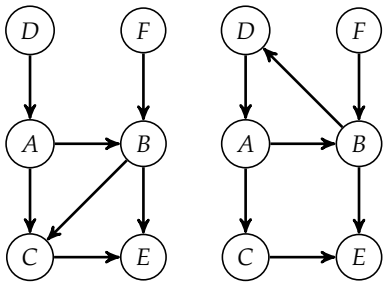
$$P(\text{ball} \mid \text{medium}, \text{slow}) \propto (0.01)(0.199)(0.4) = 0.000796$$

Since $P(\text{pedestrian} \mid \text{medium}, \text{slow})$ has the largest probability, the object is classified as a pedestrian.

Exercise 3.4. Given the 3SAT formula in equation (3.14) and the Bayesian network structure in figure 3.4, what are the values of $P(c_3^1 \mid x_2^1, x_3^0, x_4^1)$ and $P(y^1 \mid d_2^1, c_3^0)$?

Solution: We have $P(c_3^1 \mid x_2^1, x_3^0, x_4^1) = 1$ because x_2^1, x_3^0, x_4^1 makes the third clause true, and $P(y^1 \mid d_2^1, c_3^0) = 0$, because $Y = 1$ requires that both D_2 and C_3 be true.

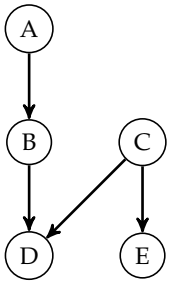
Exercise 3.5. Give a topological sort for each of the following directed graphs:



(1) (2)

Solution: There are three valid topological sorts for the first directed graph (Bayesian network): (F, D, A, B, C, E), (D, A, F, B, C, E), and (D, F, A, B, C, E). There are no valid topological sorts for the second directed graph since it is cyclic.

Exercise 3.6. Suppose that we have the following Bayesian network and we are interested in generating an approximation of the inference query $P(e^1 \mid b^0, d^1)$ using likelihood weighted sampling. Given the following samples, write the expressions for each of the sample weights. In addition, write the equation for estimating $P(e^1 \mid b^0, d^1)$ in terms of the sample weights w_i.



A	B	C	D	E
0	0	0	1	0
1	0	0	1	0
0	0	0	1	1
1	0	1	1	1
0	0	1	1	0
1	0	1	1	1

Solution: For likelihood weighted sampling, the sample weights are the product of the distributions over evidence variables conditioned on the values of their parents. Thus, the general form for our weights is $P(b^0 \mid a)P(d^1 \mid b^0, c)$. We then match each of the values for each sample from the joint distribution:

A	B	C	D	E	Weight
0	0	0	1	0	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^0)$
1	0	0	1	0	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^0)$
0	0	0	1	1	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^0)$
1	0	1	1	1	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^1)$
0	0	1	1	0	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^1)$
1	0	1	1	1	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^1)$

To estimate $P(e^1 \mid b^0, d^1)$, we simply need to sum the weights of samples consistent with the query variable and divide this by the sum of all the weights:

$$P(e^1 \mid b^0, d^1) = \frac{\sum_i w_i(e^{(i)} = 1)}{\sum_i w_i} = \frac{w_3 + w_4 + w_6}{w_1 + w_2 + w_3 + w_4 + w_5 + w_6}$$

Exercise 3.7. Each year, we receive student scores on standardized mathematics M, reading R, and writing W exams. Using data from prior years, we create the following distribution:

$$\begin{bmatrix} M \\ R \\ W \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 81 \\ 82 \\ 80 \end{bmatrix}, \begin{bmatrix} 25 & -9 & -16 \\ -9 & 36 & 16 \\ -16 & 16 & 36 \end{bmatrix} \right)$$

Compute the parameters of the conditional distribution over a student's math and reading test scores, given a writing score of 90.

Solution: If we let **a** represent the vector of math and reading scores and **b** represent the writing score, the joint and conditional distributions are as follows:

$$\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\boldsymbol{\mu}_a\\ \boldsymbol{\mu}_b\end{bmatrix}, \begin{bmatrix}\mathbf{A} & \mathbf{C}\\ \mathbf{C}^\top & \mathbf{B}\end{bmatrix}\right)$$

$$p(\mathbf{a} \mid \mathbf{b}) = \mathcal{N}\left(\mathbf{a} \mid \boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b}\right)$$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_b)$$

$$\boldsymbol{\Sigma}_{a|b} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top$$

In the example, we have the following definitions:

$$\boldsymbol{\mu}_a = \begin{bmatrix}81\\ 82\end{bmatrix} \qquad \boldsymbol{\mu}_b = \begin{bmatrix}80\end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix}25 & -9\\ -9 & 36\end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix}36\end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix}-16\\ 16\end{bmatrix}$$

Thus, the parameters of our conditional distribution given $\mathbf{b} = W = 90$ are

$$\boldsymbol{\mu}_{M,R|W=90} = \begin{bmatrix}81\\ 82\end{bmatrix} + \begin{bmatrix}-16\\ 16\end{bmatrix}\frac{1}{36}(90 - 80) \approx \begin{bmatrix}76.5\\ 86.4\end{bmatrix}$$

$$\boldsymbol{\Sigma}_{M,R|W=90} = \begin{bmatrix}25 & -9\\ -9 & 36\end{bmatrix} - \begin{bmatrix}-16\\ 16\end{bmatrix}\frac{1}{36}\begin{bmatrix}-16 & 16\end{bmatrix} \approx \begin{bmatrix}25 & -9\\ -9 & 36\end{bmatrix} - \begin{bmatrix}7.1 & -7.1\\ -7.1 & 7.1\end{bmatrix} = \begin{bmatrix}17.9 & -1.9\\ -1.9 & 28.9\end{bmatrix}$$

Given that the student scores a 90 on the writing test, based on our conditional distribution, we expect the student to earn a 76.5 on the math test, with a standard deviation of $\sqrt{17.9}$, and an 86.4 on the reading test, with a standard deviation of $\sqrt{28.9}$.

4 Parameter Learning

We have assumed so far that the parameters and structure of our probabilistic models were known. This chapter addresses the problem of *learning* or *fitting* model parameters from data.[1] We begin by introducing an approach where we identify the parameters of a model that maximize the likelihood of observing the data. After discussing the limitations of such an approach, we introduce an alternative Bayesian approach, in which we start with a probability distribution over the unknown parameters and then update that distribution based on the observed data using the laws of probability. We then discuss probabilistic models that avoid committing to a fixed number of parameters.

[1] This chapter focuses on learning model parameters from data, which is an important component of the field of *machine learning*. A broad introduction to the field is provided by K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

4.1 Maximum Likelihood Parameter Learning

In *maximum likelihood parameter learning*, we attempt to find the parameters of a distribution that maximize the likelihood of observing the data. If θ represents the parameters of a distribution, then the *maximum likelihood estimate* is

$$\hat{\theta} = \arg\max_{\theta} P(D \mid \theta) \tag{4.1}$$

where $P(D \mid \theta)$ is the likelihood that our probability model assigns to the data D occurring when the model parameters are set to θ.[2] We often use the ''hat'' accent (''$\hat{\cdot}$'') to indicate an estimate of a parameter.

[2] Here, we write $P(D \mid \theta)$ as if it is a probability mass associated with a discrete distribution. However, our probability model may be continuous, in which case we are working with densities.

There are two challenges associated with maximum likelihood parameter learning. One is to choose an appropriate probability model by which we define $P(D \mid \theta)$. We often assume that the samples in our data D are *independently and identically distributed*, which means that our samples $D = o_{1:m}$ are drawn from a

distribution $o_i \sim P(\cdot \mid \theta)$ with

$$P(D \mid \theta) = \prod_i P(o_i \mid \theta) \tag{4.2}$$

Probability models could include, for example, the categorical distributions or Gaussian distributions mentioned in earlier chapters.

The other challenge is performing the maximization in equation (4.1). For many common probability models, we can perform this optimization analytically. Others may be difficult. A common approach is to maximize the *log-likelihood*, often denoted as $\ell(\theta)$. Since the log-transformation is monotonically increasing, maximizing the log-likelihood is equivalent to maximizing the likelihood:[3]

$$\hat{\theta} = \arg\max_\theta \sum_i \log P(o_i \mid \theta) \tag{4.3}$$

[3] Although it does not matter whether we maximize the natural logarithm (base e) or the common logarithm (base 10) in this equation, throughout this book we will use $\log(x)$ to mean the logarithm of x with base e.

Computing the sum of log-likelihoods is typically much more numerically stable compared to computing the product of many small probability masses or densities. The remainder of this section will demonstrate how to optimize equation (4.1) for different types of distributions.

4.1.1 Maximum Likelihood Estimates for Categorical Distributions

Suppose that the random variable C represents whether a flight will result in a midair collision, and we are interested in estimating the distribution $P(C)$. Because C is either 0 or 1, it is sufficient to estimate the parameter $\theta = P(c^1)$. What we want to do is infer θ from data D. We have a historical database spanning a decade consisting of m flights with n midair collisions. Our intuition, of course, tells us that a good estimate for θ, given the data D, is n/m. Under the assumption of independence of outcomes between flights, the probability of a sequence of m outcomes in D with n midair collisions is:

$$P(D \mid \theta) = \theta^n (1-\theta)^{m-n} \tag{4.4}$$

The maximum likelihood estimate $\hat{\theta}$ is the value for θ that maximizes equation (4.4), which is equivalent to maximizing the logarithm of the likelihood:

$$\ell(\theta) = \log\left(\theta^n (1-\theta)^{m-n}\right) \tag{4.5}$$

$$= n \log \theta + (m-n) \log(1-\theta) \tag{4.6}$$

We can use the standard technique for finding the maximum of a function by setting the first derivative of ℓ to 0 and then solving for θ. The derivative is given by

$$\frac{\partial}{\partial\theta}\ell(\theta) = \frac{n}{\theta} - \frac{m-n}{1-\theta} \tag{4.7}$$

We can solve for $\hat{\theta}$ by setting the derivative to 0:

$$\frac{n}{\hat{\theta}} - \frac{m-n}{1-\hat{\theta}} = 0 \tag{4.8}$$

After a few algebraic steps, we see that, indeed, $\hat{\theta} = n/m$.

Computing the maximum likelihood estimate for a variable X that can assume k values is also straightforward. If $n_{1:k}$ are the observed counts for the k different values, then the maximum likelihood estimate for $P(x^i \mid n_{1:k})$ is given by

$$\hat{\theta}_i = \frac{n_i}{\sum_{j=1}^{k} n_j} \tag{4.9}$$

4.1.2 *Maximum Likelihood Estimates for Gaussian Distributions*

In a Gaussian distribution, the log-likelihood of the mean μ and variance σ^2 with m samples is given by

$$\ell(\mu, \sigma^2) \propto -m \log \sigma - \frac{\sum_i (o_i - \mu)^2}{2\sigma^2} \tag{4.10}$$

Again, we can set the derivative to 0 with respect to the parameters and solve for the maximum likelihood estimate:

$$\tfrac{\partial}{\partial\mu}\ell(\mu, \sigma^2) = \frac{\sum_i (o_i - \hat{\mu})}{\hat{\sigma}^2} = 0 \tag{4.11}$$

$$\tfrac{\partial}{\partial\sigma}\ell(\mu, \sigma^2) = -\frac{m}{\hat{\sigma}} + \frac{\sum_i (o_i - \hat{\mu})^2}{\hat{\sigma}^3} = 0 \tag{4.12}$$

After some algebraic manipulation, we get

$$\hat{\mu} = \frac{\sum_i o_i}{m} \qquad \hat{\sigma}^2 = \frac{\sum_i (o_i - \hat{\mu})^2}{m} \tag{4.13}$$

Figure 4.1 provides an example of fitting a Gaussian to data.

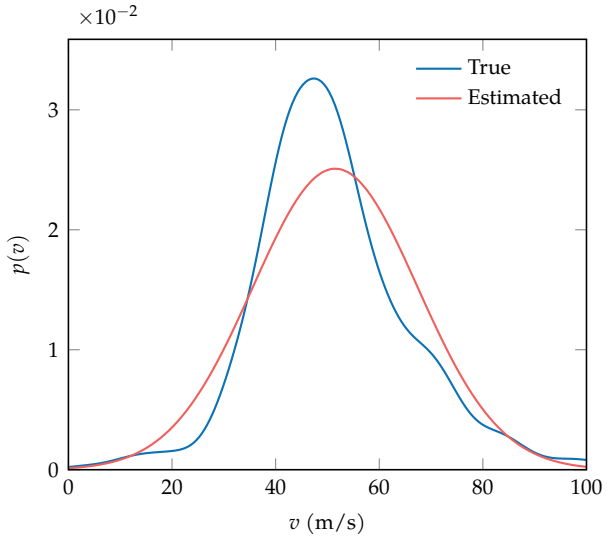


Figure 4.1. Suppose that we have airspeed measurements $o_{1:m}$ from m aircraft tracks, and we want to fit a Gaussian model. This figure shows a Gaussian with the maximum likelihood estimates $\hat{\mu} = 51.5\,\text{m/s}$ and $\hat{\sigma} = 15.9\,\text{m/s}$. The "true" distribution is shown for comparison. In this case, the Gaussian is a fairly reasonable approximation of the true distribution.

4.1.3 *Maximum Likelihood Estimates for Bayesian Networks*

We can apply maximum likelihood parameter learning to Bayesian networks. Here, we will assume that our network is composed of a set of n discrete variables that we denote as $X_{1:n}$. Our data $D = \{\mathbf{o}_1, \ldots, \mathbf{o}_m\}$ consists of observed samples from those variables. In our network with structure G, r_i is the number of instantiations of X_i, and q_i is the number of instantiations of the parents of X_i. If X_i has no parents, then $q_i = 1$. The jth instantiation of the parents of X_i is denoted as $\boldsymbol{\pi}_{ij}$.

The factor table for X_i thus has $r_i q_i$ entries, resulting in a total of $\sum_{i=1}^{n} r_i q_i$ parameters in our Bayesian network. Each parameter is written as θ_{ijk} and determines

$$P(X_i = k \mid \boldsymbol{\pi}_{ij}) = \theta_{ijk} \tag{4.14}$$

Although there are $\sum_{i=1}^{n} r_i q_i$ parameters, only $\sum_{i=1}^{n}(r_i - 1)q_i$ are independent. We use $\boldsymbol{\theta}$ to represent the set of all parameters.

We use m_{ijk} to represent the number of times $X_i = k$ given parental instantiation j in the data set. Algorithm 4.1 provides an implementation of a function for extracting these counts or statistics from a data set. The likelihood is given in

terms of m_{ijk}:

$$P(D \mid \boldsymbol{\theta}, G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \prod_{k=1}^{r_i} \theta_{ijk}^{m_{ijk}} \tag{4.15}$$

Similar to the maximum likelihood estimate for the univariate distribution in equation (4.9), the maximum likelihood estimate in our discrete Bayesian network model is

$$\hat{\theta}_{ijk} = \frac{m_{ijk}}{\sum_{k'} m_{ijk'}} \tag{4.16}$$

Example 4.1 illustrates this process.

```
function sub2ind(siz, x)
    k = vcat(1, cumprod(siz[1:end-1]))
    return dot(k, x .- 1) + 1
end

function statistics(vars, G, D::Matrix{Int})
    n = size(D, 1)
    r = [vars[i].r for i in 1:n]
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n]
    M = [zeros(q[i], r[i]) for i in 1:n]
    for o in eachcol(D)
        for i in 1:n
            k = o[i]
            parents = inneighbors(G,i)
            j = 1
            if !isempty(parents)
                j = sub2ind(r[parents], o[parents])
            end
            M[i][j,k] += 1.0
        end
    end
    return M
end
```

Algorithm 4.1. A function for extracting the statistics, or counts, from a discrete data set `D`, assuming a Bayesian network with variables `vars` and structure `G`. The data set is an $n \times m$ matrix, where n is the number of variables and m is the number of data points. This function returns an array `M` of length n. The ith component consists of a $q_i \times r_i$ matrix of counts. The `sub2ind(siz, x)` function returns a linear index into an array with dimensions specified by `siz` given coordinates `x`. It is used to identify which parental instantiation is relevant to a particular data point and variable.

4.2 Bayesian Parameter Learning

Bayesian parameter learning addresses some of the drawbacks of maximum likelihood estimation, especially when the amount of data is limited. For example, suppose that our aviation safety database was limited to the events of the past week, and we found no recorded midair collisions. If θ is the probability that a flight results in a midair collision, then the maximum likelihood estimate would

Example 4.1. Using the `statistics` function for extracting the statistics from a data set. Bayesian parameter learning can be used to avoid NAN values, but we must specify a prior.

Suppose that we have a small network, $A \rightarrow B \leftarrow C$, and we want to extract the statistics from data matrix `D`. We can use the following code:

```
G = SimpleDiGraph(3)
add_edge!(G, 1, 2)
add_edge!(G, 3, 2)
vars = [Variable(:A,2), Variable(:B,2), Variable(:C,2)]
D = [1 2 2 1; 1 2 2 1; 2 2 2 2]
M = statistics(vars, G, D)
```

The output is an array `M` consisting of these three count matrices, each of size $q_i \times r_i$:

$$\begin{bmatrix} 2 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 4 \end{bmatrix}$$

We can compute the maximum likelihood estimate by normalizing the rows in the matrices in `M`:

```
θ = [mapslices(x->normalize(x,1), Mi, dims=2) for Mi in M]
```

which produces

$$\begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \quad \begin{bmatrix} \text{NAN} & \text{NAN} \\ \text{NAN} & \text{NAN} \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \end{bmatrix}$$

As we can see, the first and second parental instantiations of the second variable B leads to NAN ("not a number") estimates. Because there are no observations of those two parental instantiations in the data, the denominator of equation (4.16) equals zero, making the parameter estimate undefined. Most of the other parameters are not NAN. For example, the parameter $\theta_{112} = 0.5$ means that the maximum likelihood estimate of $P(a^2)$ is 0.5.

be $\hat{\theta} = 0$. Believing that there is zero chance of a midair collision is not a reasonable conclusion unless our prior hypothesis was, for example, that all flights were perfectly safe.

The Bayesian approach to parameter learning involves estimating $p(\theta \mid D)$, the posterior distribution over θ given our data D. Instead of obtaining a point estimate $\hat{\theta}$ as in maximum likelihood estimation, we obtain a distribution. This distribution can help us quantify our uncertainty about the true value of θ. We can convert this distribution into a point estimate by computing the expectation:

$$\hat{\theta} = \mathbb{E}_{\theta \sim p(\cdot \mid D)}[\theta] = \int \theta p(\theta \mid D) \, \mathrm{d}\theta \tag{4.17}$$

In some cases, however, the expectation may not be an acceptable estimate, as illustrated in figure 4.2. An alternative is to use the *maximum a posteriori* estimate:

$$\hat{\theta} = \arg\max_{\theta} p(\theta \mid D) \tag{4.18}$$

This estimate corresponds to a value of θ that is assigned the greatest density. This is often referred to as the *mode* of the distribution. As shown in figure 4.2, the mode may not be unique.

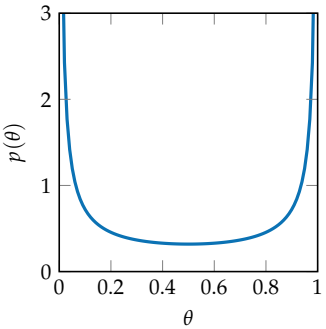


Figure 4.2. An example of a distribution where the expected value of θ is not a good estimate. The expected value of 0.5 has a lower density than occurs at the extreme values of 0 or 1. This distribution happens to be a beta distribution, a type of distribution we will discuss shortly, with parameters $(0.2, 0.2)$.

Bayesian parameter learning can be viewed as inference in a Bayesian network with the structure in figure 4.3, which makes the assumption that the observed variables are conditionally independent of each other. As with any Bayesian network, we must specify $p(\theta)$ and $P(O_i \mid \theta)$. We often use a uniform prior $p(\theta)$. The remainder of this section discusses how to apply Bayesian parameter learning to different models of $P(O_i \mid \theta)$.

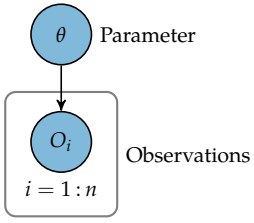


Figure 4.3. Bayesian network representing parameter learning.

4.2.1 *Bayesian Learning for Binary Distributions*

Suppose we want to learn the parameters of a binary distribution. Here, we will use $P(o^1 \mid \theta) = \theta$. To infer the distribution over θ in the Bayesian network in figure 4.3, we can proceed with the standard method for performing inference

discussed in chapter 3. Here, we will assume a uniform prior:

$$p(\theta \mid o_{1:m}) \propto p(\theta, o_{1:m}) \tag{4.19}$$

$$= p(\theta) \prod_{i=1}^{m} P(o_i \mid \theta) \tag{4.20}$$

$$= \prod_{i=1}^{m} P(o_i \mid \theta) \tag{4.21}$$

$$= \prod_{i=1}^{m} \theta^{o_i} (1-\theta)^{1-o_i} \tag{4.22}$$

$$= \theta^n (1-\theta)^{m-n} \tag{4.23}$$

The posterior is proportional to $\theta^n(1-\theta)^{m-n}$, where n is the number of times $O_i = 1$. To find the normalization constant, we integrate

$$\int_0^1 \theta^n (1-\theta)^{m-n} \, \mathrm{d}\theta = \frac{\Gamma(n+1)\Gamma(m-n+1)}{\Gamma(m+2)} \tag{4.24}$$

where Γ is the *gamma function*. The gamma function is a real-valued generalization of the factorial. If m is an integer, then $\Gamma(m) = (m-1)!$. Taking normalization into account, we have

$$p(\theta \mid o_{1:m}) = \frac{\Gamma(m+2)}{\Gamma(n+1)\Gamma(m-n+1)} \theta^n (1-\theta)^{m-n} \tag{4.25}$$

$$= \text{Beta}(\theta \mid n+1, m-n+1) \tag{4.26}$$

The *beta distribution* $\text{Beta}(\alpha, \beta)$ is defined by parameters α and β, and curves for this distribution are shown in figure 4.4. The distribution $\text{Beta}(1, 1)$ corresponds to the uniform distribution spanning 0 to 1.

The distribution $\text{Beta}(\alpha, \beta)$ has mean

$$\frac{\alpha}{\alpha + \beta} \tag{4.27}$$

When α and β are both greater than 1, the mode is

$$\frac{\alpha - 1}{\alpha + \beta - 2} \tag{4.28}$$

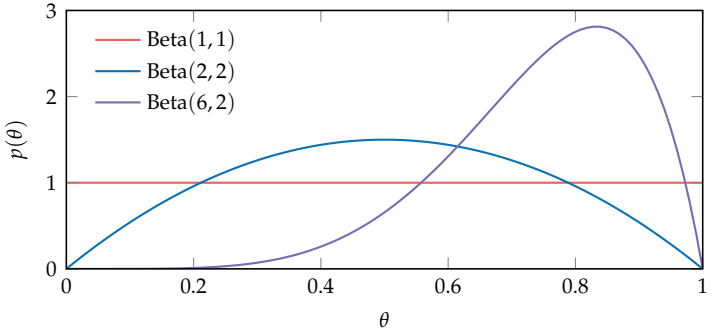


Figure 4.4. An overlay of several beta probability densities.

Conveniently, if a beta distribution is used as a prior over a parameter of a binary distribution, then the posterior is also a beta distribution. In particular, if the prior is given by $\text{Beta}(\alpha, \beta)$ and we make an observation o_i, then we get a posterior of $\text{Beta}(\alpha + 1, \beta)$ if $o_i = 1$ and $\text{Beta}(\alpha, \beta + 1)$ if $o_i = 0$. Hence, if we started with a prior given by $\text{Beta}(\alpha, \beta)$ and our data showed that there were n collisions out of m flights, then the posterior would be given by $\text{Beta}(\alpha + n, \beta + m - n)$. The α and β parameters in the prior are sometimes called *pseudocounts* because they are treated similarly to the observed counts of the two outcome classes in the posterior, although the pseudocounts need not be integers.

Choosing the prior, in principle, should be done without knowledge of the data used to compute the posterior. Uniform priors often work well in practice, although if expert knowledge is available, then it can be encoded into the prior. For example, suppose that we had a slightly bent coin and we wanted to estimate θ, the probability that the coin would land heads. Before we collected any data by flipping the coin, we would start with a belief θ that is likely to be around 0.5. Instead of starting with a uniform prior $\text{Beta}(1, 1)$, we might use $\text{Beta}(2, 2)$ (shown in figure 4.4), which gives more weight to values near 0.5. If we were more confident in an estimate near 0.5, then we could reduce the variance of the prior by increasing the pseudocounts. The prior $\text{Beta}(10, 10)$ is much more peaked than $\text{Beta}(2, 2)$. In general, however, the importance of the prior diminishes with the amount of data used to compute the posterior. If we observe m flips and n were heads, then the difference between $\text{Beta}(1 + n, 1 + m - n)$ and $\text{Beta}(10 + n, 10 + m - n)$ is negligible if we observe thousands of coin flips.

4.2.2 Bayesian Learning for Categorical Distributions

The *Dirichlet distribution*[4] is a generalization of the beta distribution and can be used to estimate the parameters of categorical distributions. Suppose that X is a discrete random variable that takes integer values from 1 to n. We define the parameters of the distribution to be $\theta_{1:n}$, where $P(x^i) = \theta_i$. Of course, the parameters must sum to 1, and so only the first $n - 1$ parameters are independent. The Dirichlet distribution can be used to represent both the prior and the posterior distribution and is parameterized by $\alpha_{1:n}$. The density is given by

$$\text{Dir}(\theta_{1:n} \mid \alpha_{1:n}) = \frac{\Gamma(\alpha_0)}{\prod_{i=1}^{n} \Gamma(\alpha_i)} \prod_{i=1}^{n} \theta_i^{\alpha_i - 1} \tag{4.29}$$

where α_0 is used to denote the summation of the parameters $\alpha_{1:n}$.[5] If $n = 2$, then it is easy to see that equation (4.29) is equivalent to the beta distribution.

It is common to use a uniform prior where all the Dirichlet parameters $\alpha_{1:n}$ are set to 1. As with the beta distribution, the parameters in the Dirichlet are often referred to as *pseudocounts*. If the prior over $\theta_{1:n}$ is given by $\text{Dir}(\alpha_{1:n})$ and there are m_i observations of $X = i$, then the posterior is given by

$$p(\theta_{1:n} \mid \alpha_{1:n}, m_{1:n}) = \text{Dir}(\theta_{1:n} \mid \alpha_1 + m_1, \ldots, \alpha_n + m_n) \tag{4.30}$$

The distribution $\text{Dir}(\alpha_{1:n})$ has a mean vector whose ith component is

$$\frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j} \tag{4.31}$$

When $\alpha_i > 1$, the ith component of the mode is

$$\frac{\alpha_i - 1}{\sum_{j=1}^{n} \alpha_j - n} \tag{4.32}$$

As we have seen, Bayesian parameter estimation is straightforward for binary and discrete random variables because it involves simply counting the various outcomes in the data. Bayes' rule can be used to infer the distribution over the parameters for other parametric distributions. Depending on the choice of prior and the form of the parametric distribution, calculating the posterior over the space of parameters also might be done analytically.

[4] This distribution is named after the German mathematician Johann Peter Gustav Lejeune Dirichlet (1805–1859).

[5] See appendix B for plots of Dirichlet distribution densities for different parameters.

4.2.3 *Bayesian Learning for Bayesian Networks*

We can apply Bayesian parameter learning to discrete Bayesian networks. The prior over the Bayesian network parameters $\boldsymbol{\theta}$ can be factorized as follows:

$$p(\theta \mid G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} p(\theta_{ij}) \tag{4.33}$$

where $\theta_{ij} = (\theta_{ij1}, \ldots, \theta_{ijr_i})$. The prior $p(\theta_{ij})$, under some weak assumptions, can be shown to follow a Dirichlet distribution $\text{Dir}(\alpha_{ij1}, \ldots, \alpha_{ijr_i})$. Algorithm 4.2 provides an implementation for creating a data structure holding α_{ijk}, where all entries are 1, corresponding to a uniform prior.

After observing data in the form of m_{ijk} counts (as introduced in section 4.1.3), the posterior is then

$$p(\theta_{ij} \mid \alpha_{ij}, m_{ij}) = \text{Dir}(\theta_{ij} \mid \alpha_{ij1} + m_{ij1}, \ldots, \alpha_{ijr_i} + m_{ijr_i}) \tag{4.34}$$

similar to equation (4.30). Example 4.2 demonstrates this process.

```
function prior(vars, G)
    n = length(vars)
    r = [vars[i].r for i in 1:n]
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n]
    return [ones(q[i], r[i]) for i in 1:n]
end
```

Algorithm 4.2. A function for generating a prior α_{ijk} where all entries are 1. The array of matrices that this function returns takes the same form as the statistics generated by algorithm 4.1. To determine the appropriate dimensions, the function takes as input the list of variables `vars` and structure `G`.

Example 4.2. Computing the posterior parameters in a Bayesian network. Note that unlike example 4.1, here we do not have NAN values.

We can compute the parameters of the posterior associated with a Bayesian network through simple addition of the prior parameters and counts (equation (4.34)). If we use the matrix of counts `M` obtained in example 4.1, we can add it to the matrices of prior parameters `α = prior(vars, G)` to obtain the set of posterior parameters `M + α`:

$$\begin{bmatrix} 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 3 & 1 \\ 1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 5 \end{bmatrix}$$

4.3 Nonparametric Learning

The previous two sections assumed that the probabilistic model was of a fixed form and that a fixed set of parameters were to be learned from the data. An alternative approach is based on *nonparametric* methods in which the number of parameters scales with the amount of data. A common nonparametric method is *kernel density estimation* (algorithm 4.3). Given observations $o_{1:m}$, kernel density estimation represents the density as follows:

$$p(x) = \frac{1}{m}\sum_{i=1}^{m} \phi(x - o_i) \tag{4.35}$$

where ϕ is a *kernel function*, which integrates to 1. The kernel function is used to assign greater density to values near the observed data points. A kernel function is generally symmetric, meaning that $\phi(x) = \phi(-x)$. A common kernel is the zero-mean Gaussian distribution. When such a kernel is used, the standard deviation is often referred to as the *bandwidth*, which can be tuned to control the smoothness of the density function. Larger bandwidths generally lead to smoother densities. Bayesian methods can be applied to the selection of the appropriate bandwidth based on the data. The effect of the bandwidth choice is shown in figure 4.5.

```
gaussian_kernel(b) = x->pdf(Normal(0,b), x)

function kernel_density_estimate(ϕ, O)
    return x -> sum([ϕ(x - o) for o in O])/length(O)
end
```

Algorithm 4.3. The method `gaussian_kernel` returns a zero-mean Gaussian kernel $\phi(x)$ with bandwidth `b`. Kernel density estimation is also implemented for a kernel `ϕ` and list of observations `O`.

4.4 Learning with Missing Data

When learning the parameters of our probabilistic model we may have *missing* entries in our data.[6] For example, if we are conducting a survey, some respondents may decide to skip a question. Table 4.1 shows an example of a data set with missing entries involving three binary variables: A, B, and C. One approach to handling missing data is to discard all the instances that are *incomplete*, where there are one or more missing entries. Depending on how much of the data is missing, we might have to discard much of it. In table 4.1, we would have to discard all but one of the rows, which can be wasteful.

[6] Learning with missing data is the subject of a large body of literature. A comprehensive introduction and review is provided by G. Molenberghs, G. Fitzmaurice, M. G. Kenward, A. Tsiatis, and G. Verbeke, eds., *Handbook of Missing Data Methodology*. CRC Press, 2014.

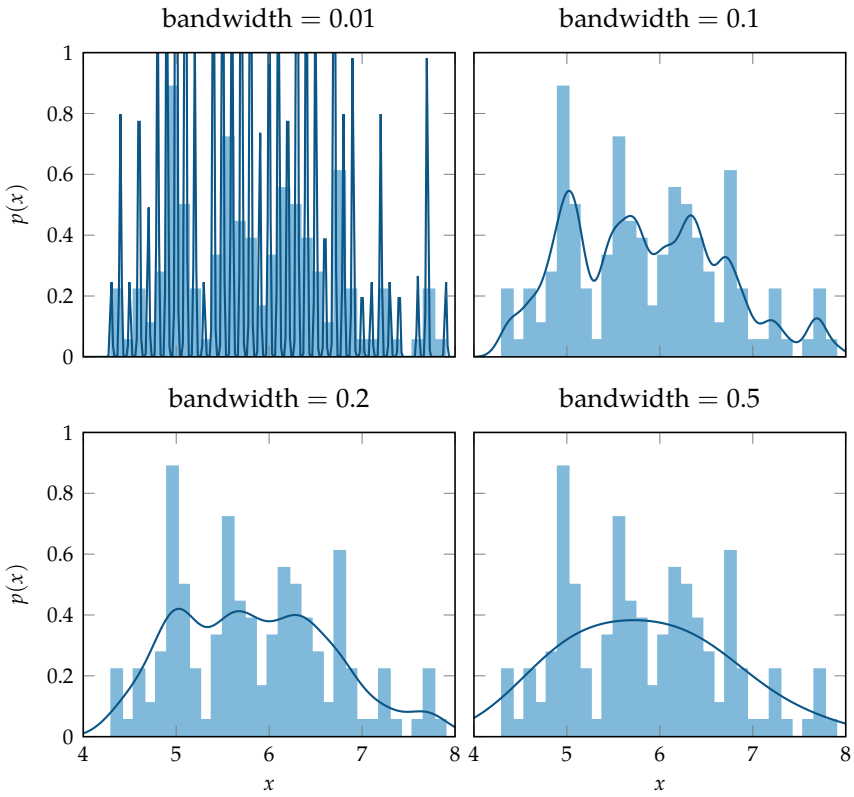


Figure 4.5. Kernel density estimation applied to the same data set using zero-mean Gaussian kernels with different bandwidths. The histogram in blue shows the underlying data set frequencies, and the black lines indicate the probability density from kernel density estimation. Larger bandwidths smooth out the estimate, whereas smaller bandwidths can overfit to specific samples.

We can learn model parameters from missing data using either a maximum likelihood or a Bayesian approach. If taking a Bayesian maximum a posteriori approach, we want to find the estimate

$$\hat{\theta} = \arg\max_{\theta} p(\theta \mid D_{\text{obs}}) \tag{4.36}$$

$$= \arg\max_{\theta} \sum_{D_{\text{mis}}} p(\theta \mid D_{\text{obs}}, D_{\text{mis}}) P(D_{\text{mis}} \mid D_{\text{obs}}) \tag{4.37}$$

where D_{obs} and D_{mis} consist of all the observed and missing data, respectively. If the data is continuous, then the sum would be replaced by an integral. The marginalization over the missing data can be computationally expensive. The same marginalization also affects the computational tractability of a Bayesian approach.

This section discusses two general approaches for learning with missing data without having to enumerate over all the possible combinations of missing values. The first involves learning the distribution parameters using predicted values of the missing entries. The second involves an iterative approach for improving our parameter estimates.

We will focus on the context where data is *missing at random*, meaning that the probability that an entry is missing is conditionally independent of its value, given the values of the observed variables. An example of a situation that does not adhere to this assumption might include radar data containing measurements of the distance to a target, but the measurement may be missing either due to noise or because the target is beyond the sensing range. The fact that an entry is missing is an indication that the value is more likely to be high. Accounting for this form of missingness requires different models and algorithms from what we discuss here.[7]

A	B	C
1	1	0
?	1	1
1	?	?
?	?	?

Table 4.1. Example of data consisting of four instances with six missing entries.

[7] Different *missingness mechanisms* and associated inference techniques are reviewed by R. J. A. Little and D. B. Rubin, *Statistical Analysis with Missing Data*, 3rd ed. Wiley, 2020.

4.4.1 Data Imputation

An alternative to discarding incomplete instances is to impute the values of missing entries. *Data imputation* is the process of inferring values for missing entries. One way to view imputation is as an approximation of equation (4.37), where we find

$$\hat{D}_{\text{mis}} = \arg\max_{D_{\text{mis}}} p(D_{\text{mis}} \mid D_{\text{obs}}) \tag{4.38}$$

Once we have the imputed missing values, we can use that data to produce a maximum posteriori estimate:

$$\hat{\theta} = \arg\max_{\theta} p(\theta \mid D_{\text{obs}}) \approx \arg\max_{\theta} p(\theta \mid D_{\text{obs}}, \hat{D}_{\text{mis}}) \tag{4.39}$$

or, alternatively, we can take a maximum likelihood approach.

Solving equation (4.38) may still be computationally challenging. One simple approach for discrete data sets is to replace missing entries with the most commonly observed value, called the *marginal mode*. For example, in table 4.1, we might replace all the missing values for A with its marginal mode of 1.

A	B	C
−6.5	0.9	4.2
?	4.4	9.2
7.8	?	?
?	?	?

Table 4.2. Example of data with continuous values.

Continuous data often lacks duplicates. However, we can fit a distribution to continuous values and then use the mode of the resulting distribution. For example, we might fit a Gaussian distribution to the data in table 4.2, and then fill in the missing entries with the mean of the observed values associated with each variable. The top-left plot in figure 4.6 illustrates the effect of this approach on two-dimensional data. The red lines show how values with missing first or second components are paired with their imputed counterparts. We can then use the observed and imputed data to arrive at a maximum likelihood estimate of the parameters of a joint Gaussian distribution. As we can see, this method of imputation does not always produce sensible predictions and the learned model is quite poor.

We can often do better if we account for the probabilistic relationships between the observed and unobserved variables. In figure 4.6, there is clearly correlation between the two variables; hence, knowing the value of one variable can help predict the value of the other variable. A common approach to imputation, called *nearest-neighbor imputation*, is to use the values associated with the instance that is nearest with respect to a distance measure defined on the observed variables. The top-right plot in figure 4.6 uses the Euclidean distance for imputation. This approach tends to lead to better imputations and learned distributions.

An alternative approach is to fit a distribution to the fully observed data and then use that distribution to infer the missing values. We can use the inference algorithms from the previous chapter to perform this inference. For example, if our data is discrete and we can assume a Bayesian network structure, we can use variable elimination or Gibbs sampling to produce a distribution over the missing variables for an instance from the observed variables. From this distribution, we might use the mean or mode to impute the missing values. Alternatively, we can

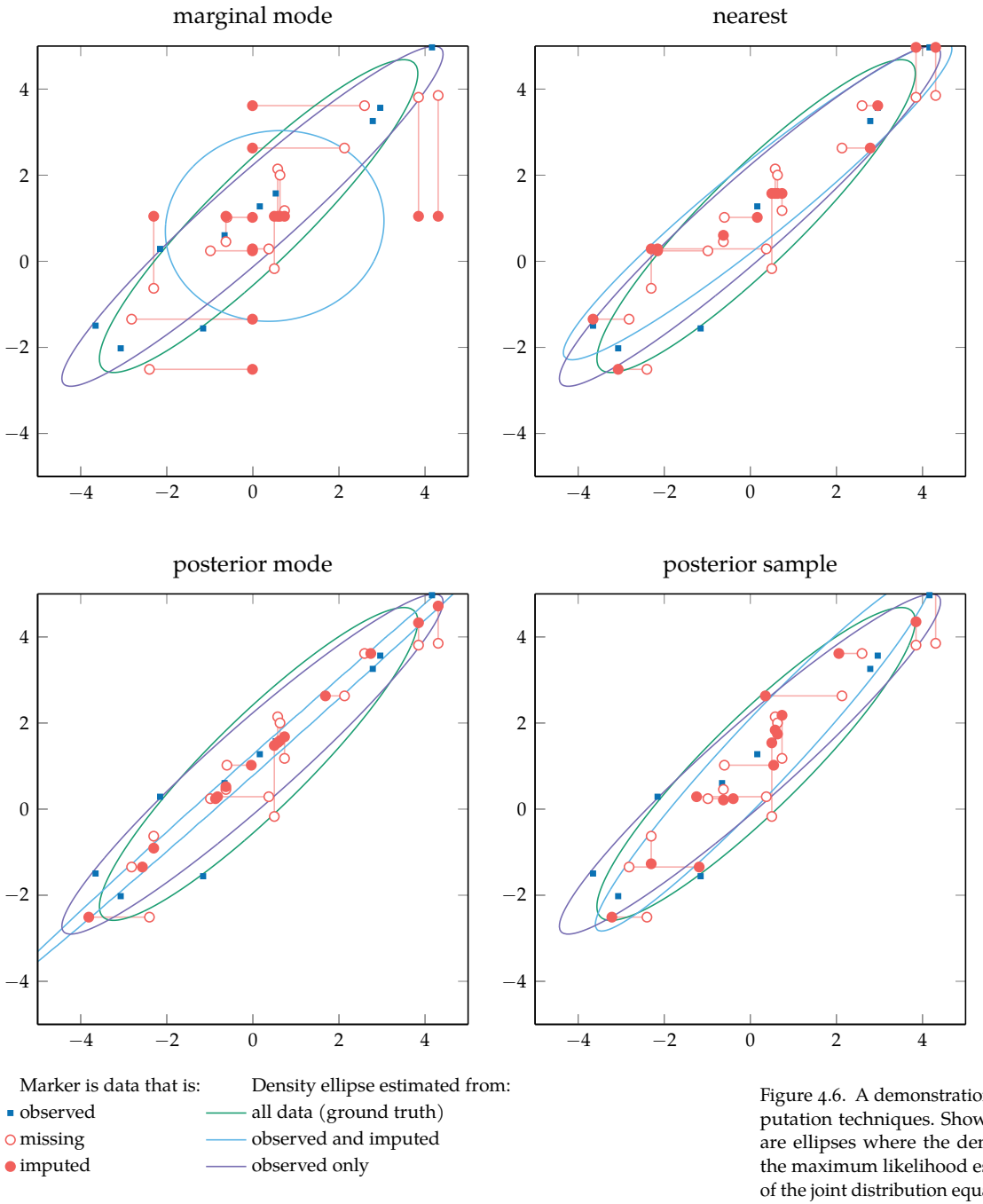


Figure 4.6. A demonstration of imputation techniques. Shown here are ellipses where the density of the maximum likelihood estimate of the joint distribution equals 0.02.

pull a sample from this distribution. If our data is continuous and we can assume that the data is jointly Gaussian, we can use algorithm 3.11 to infer the posterior distribution. The bottom plots in figure 4.6 demonstrate imputation using these posterior mode and posterior sampling approaches.

4.4.2 Expectation-Maximization

The *expectation-maximization* (*EM*) category of approaches involves iterative improvement of the distribution parameter estimate $\hat{\theta}$.[8] We begin with an initial $\hat{\theta}$, which may be a guess, randomly sampled from a prior distribution over distribution parameters, or estimated using one of the methods discussed in section 4.4.1. At each iteration, we perform a two-step process to update $\hat{\theta}$.

[8] Expectation-maximization was introduced by A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society, Series B* (*Methodological*), vol. 39, no. 1, pp. 1–38, 1977.

The first step is called the *expectation step* (*E-step*), where we use the current estimate of θ to infer completions of the data. For example, if we are modeling our data using a discrete Bayesian network, we can use one of our inference algorithms to infer a distribution over the missing entries for each instance. When extracting the counts, we apply a weighting proportional to the likelihood of the completions as shown in example 4.3. In cases where there are many missing variables, there may be too many possible completions to practically enumerate, making a sampling-based approach attractive. We may also want to use sampling as an approximation method when our variables are continuous.

The second step is called the *maximization step* (*M-step*), where we attempt to find a new $\hat{\theta}$ that maximizes the likelihood of the completed data. If we have a discrete Bayesian network with the weighted counts in the form shown in example 4.3, then we can perform the same maximum likelihood estimate as discussed earlier in this chapter. Alternatively, we can use a maximum a posteriori estimate if we want to incorporate a prior.

This approach is not guaranteed to converge to model parameters that maximize the likelihood of the observed data, but it can work well in practice. To reduce the risk of the algorithm converging to only a local optimum, we can run the algorithm to convergence from many different initial points in the parameter space. We simply choose the resulting parameter estimate in the end that maximizes likelihood.

Expectation-maximization can even be used to impute values for variables that are not observed at all in the data. Such variables are called *latent variables*. To illustrate, suppose we have a Bayesian network $Z \rightarrow X$, where X is continuous

Example 4.3. Expanding an incomplete data set using assumed model parameters.

Suppose that we have a binary Bayesian network with $A \to B$. We start by assuming that $\hat{\theta}$ implies

$$P(a^1) = 0.5 \qquad P(b^1 \mid a^0) = 0.2 \qquad P(b^1 \mid a^1) = 0.6$$

Using these parameters, we can expand the data set with missing values (left) to a weighted data set with all possible individual completions (right):

A	B
1	1
0	1
0	?
?	0

A	B	weight
1	1	1
0	1	1
0	0	$1 - P(b^1 \mid a^0) = 0.8$
0	1	$P(b^1 \mid a^0) = 0.2$
0	0	$\alpha P(a^0)P(b^0 \mid a^0) = \alpha 0.4 = 2/3$
1	0	$\alpha P(a^1)P(b^0 \mid a^1) = \alpha 0.2 = 1/3$

The α in the calculation here is a normalization constant, which enforces that each instance is expanded to instances whose weights sum to 1. The count matrices are then

$$\begin{bmatrix} (2 + 2/3) & (1 + 1/3) \end{bmatrix} \qquad \begin{bmatrix} (0.8 + 2/3) & 1.2 \\ 1/3 & 1 \end{bmatrix}$$

and Z is discrete and can take on one of three values. Our model assumes $p(x \mid z)$ is conditional Gaussian. Our data set contains only values for X, but none for Z. We start with an initial $\hat{\theta}$ and use it to infer a probability distribution over the values of Z, given the value of X for each instance. The distribution over entry completions are then used to update our estimate of the parameters of $P(Z)$ and $P(X \mid Z)$ as illustrated in example 4.4. We iterate to convergence, which often occurs very quickly. The parameters that we obtain in this example define a Gaussian mixture model, which was introduced in section 2.2.2.

4.5 Summary

- Parameter learning involves inferring the parameters of a probabilistic model from data.
- A maximum likelihood approach to parameter learning involves maximizing a likelihood function, which can be done analytically for some models.
- A Bayesian approach to parameter learning involves inferring a probability distribution over the underlying parameter using Bayes' rule.
- The beta and Dirichlet distributions are examples of Bayesian priors that are easily updated with evidence.
- In contrast with parametric learning, which assumes a fixed parameterization of a probability model, nonparametric learning uses representations that grow with the amount of data.
- We can approach the problem of learning parameters from missing data using methods such as data imputation or expectation-maximization, where we make inferences based on observed values.

4.6 Exercises

Exercise 4.1. Suppose that Anna is shooting basketball free throws. Before we see her play, we start with an independent uniform prior over the probability that she successfully makes a basket per shot. We observe her take three shots, with two of them resulting in successful baskets. What is the probability that we assign to her making the next basket?

Example 4.4. Expectation maximization applied to learning the parameters of a Gaussian mixture model.

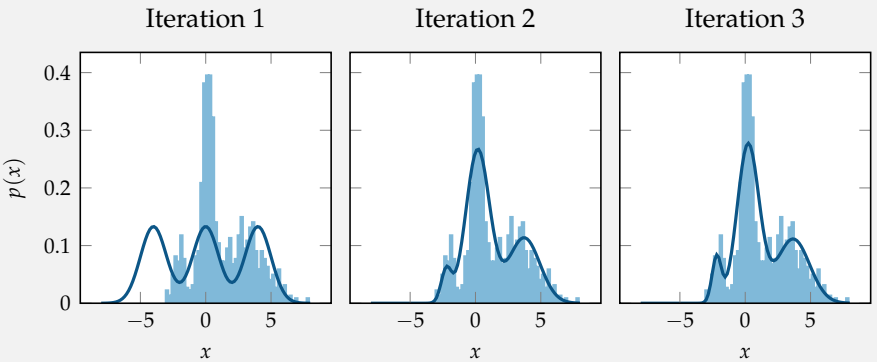
We have a Bayesian network $Z \rightarrow X$, where Z is a discrete latent variable with three values and X is continuous with $p(x \mid z)$ modeled as a conditional Gaussian. Hence, we have parameters defining $P(z^1)$, $P(z^2)$, and $P(z^3)$, as well as μ_i and σ_i for each of the three Gaussian distributions associated with different values of Z. In this example, we use an initial parameter vector $\hat{\theta}$ that specifies $P(z^i) = 1/3$ and $\sigma_i = 1$ for all i. We spread out the means with $\mu_1 = -4$, $\mu_2 = 0$, and $\mu_3 = 4$.

Suppose our first instance in our data has $X = 4.2$. We want to infer the distribution over Z for that instance:

$$P(z^i \mid X = 4.2) = \frac{P(z^i)\,\mathcal{N}(4.2 \mid \mu_i, \sigma_i^2)}{\sum_j P(z^j)\,\mathcal{N}(4.2 \mid \mu_j, \sigma_j^2)}$$

We compute this distribution for all the instances in our data set. For the weighted completions, we can obtain a new estimate for $\hat{\theta}$. We estimate $P(z^i)$ by taking the mean across the instances in our data set. To estimate μ_i and σ_i, we use the mean and standard deviation of the values for X over the instances in our data set, weighted by the probability of z^i associated with the various instances.

We repeat the process until convergence occurs. The plot here shows three iterations. The histogram was generated from the values of X. The dark blue function indicates the inferred density. By the third iteration, our parameters of the Gaussian mixture model closely represent the data distribution.



Solution: We denote the probability of making a basket as θ. Since we start with a uniform prior $\text{Beta}(1,1)$ and observe two baskets and one miss, our posterior is then $\text{Beta}(1+2,1+1) = \text{Beta}(3,2)$. We want to compute the probability of a basket as follows:

$$P(\text{basket}) = \int P(\text{basket} \mid \theta)\, \text{Beta}(\theta \mid 3,2)\, \mathrm{d}\theta = \int \theta\, \text{Beta}(\theta \mid 3,2)\, \mathrm{d}\theta$$

This expression is just the expectation (or mean) of a beta distribution, which gives us $P(\text{basket}) = 3/5$.

Exercise 4.2. Consider a continuous random variable X that follows the *Laplace distribution* parameterized by μ and b, with density

$$p(x \mid \mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right)$$

Compute the maximum likelihood estimates of the parameters of a Laplace distribution given a data set D of m independent observations $x_{1:m}$. Note that $\partial |u| / \partial x = \text{sign}(u) \partial u / \partial x$, where the sign function returns the sign of its argument.

Solution: Since the observations are independent, we can write the log-likelihood function as the summation:

$$\begin{aligned}
\ell(\mu, b) &= \sum_{i=1}^{m} \log\left[\frac{1}{2b} \exp\left(-\frac{|x_i - \mu|}{b}\right)\right] \\
&= -\sum_{i=1}^{m} \log 2b - \sum_{i=1}^{m} \frac{|x_i - \mu|}{b} \\
&= -m \log 2b - \frac{1}{b} \sum_{i=1}^{m} |x_i - \mu|
\end{aligned}$$

To obtain the maximum likelihood estimates of the true parameters μ and b, we take the partial derivatives of the log-likelihood with respect to each of the parameters, set them to zero, and solve for each parameter. First, we solve for $\hat{\mu}$:

$$\begin{aligned}
\tfrac{\partial}{\partial \mu} \ell(\mu, b) &= \frac{1}{\hat{b}} \sum_{i=1}^{m} \text{sign}(x_i - \mu) \\
0 &= \frac{1}{\hat{b}} \sum_{i=1}^{m} \text{sign}(x_i - \hat{\mu}) \\
0 &= \sum_{i=1}^{m} \text{sign}(x_i - \hat{\mu}) \\
\hat{\mu} &= \text{median}(x_{1:m})
\end{aligned}$$

Now, solving for $\hat{b}$:

$$\begin{aligned}
\tfrac{\partial}{\partial b}\ell(\mu, b) &= -\frac{m}{b} + \frac{1}{b^2}\sum_{i=1}^{m}|x_i - \hat{\mu}| \\
0 &= -\frac{m}{\hat{b}} + \frac{1}{\hat{b}^2}\sum_{i=1}^{m}|x_i - \hat{\mu}| \\
\frac{m}{\hat{b}} &= \frac{1}{\hat{b}^2}\sum_{i=1}^{m}|x_i - \hat{\mu}| \\
\hat{b} &= \frac{1}{m}\sum_{i=1}^{m}|x_i - \hat{\mu}|
\end{aligned}$$

Thus, the maximum likelihood estimates for the parameters of a Laplace distribution are $\hat{\mu}$, the median of the observations, and $\hat{b}$, the mean of absolute deviations from the median.

Exercise 4.3. This question explores the application of maximum likelihood estimation to *censored data*, where some measurements are only partially known. Suppose that we are building electric motors for a quadcopter drone, and we want to produce a model of how long they last until failure. Although there may be more suitable distributions for modeling the reliability of components,[9] we will use an *exponential distribution* parameterized by λ with probability density function $\lambda\exp(-\lambda x)$ and cumulative distribution function $1 - \exp(-\lambda x)$. We fly five drones. Three have motor failures after 132 hours, 42 hours, and 89 hours. We stopped testing the other two after 200 hours without failure; we do not know their failure times; we just know that they are greater than 200 hours. What is the maximum likelihood estimate for λ given this data?

[9] K. S. Trivedi and A. Bobbio, *Reliability and Availability Engineering*. Cambridge University Press, 2017.

Solution: This problem has $n = 3$ fully observed measurements and $m = 2$ censored measurements. We use t_i to represent the ith fully observed measurement and $\underline{t}_j$ to represent the jth censored measurement. The likelihood of a single measurement above $\underline{t}_j$ is the complement of the cumulative distribution function, which is simply $\exp(-\lambda\underline{t}_j)$. Hence, the likelihood of the data is

$$\left(\prod_{i=1}^{n}\lambda e^{-\lambda t_i}\right)\left(\prod_{j=1}^{m}e^{-\lambda\underline{t}_j}\right)$$

We use our standard approach of maximizing the log-likelihood, which is given by

$$\ell(\lambda) = \sum_{i=1}^{n}(\log\lambda - \lambda t_i) + \sum_{j=1}^{m}-\lambda\underline{t}_j$$

The derivative with respect to λ is

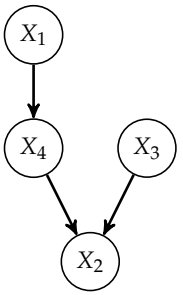
$$\frac{\partial\ell}{\partial\lambda} = \frac{n}{\lambda} - \sum_{i=1}^{n}t_i - \sum_{j=1}^{m}\underline{t}_j$$

Setting this derivative to 0, we can solve for λ to obtain the maximum likelihood estimate:

$$\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} t_i + \sum_{j=1}^{m} \underline{t}_j} = \frac{3}{132 + 42 + 89 + 200 + 200} \approx 0.00452$$

The mean of the exponential distribution is $1/\lambda$, making the mean in our problem 221 hours.

Exercise 4.4. We have a Bayesian network where the variables $X_{1:3}$ can take on values in $\{1, 2\}$ and X_4 can take on values in $\{1, 2, 3\}$. Given the data set D of observations $\mathbf{o}_{1:m}$, as illustrated here, generate the maximum likelihood estimates of the associated conditional distribution parameters $\boldsymbol{\theta}$.



$$\mathbf{D} = \begin{bmatrix} 1 & 2 & 1 & 1 & 1 & 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 & 1 & 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 2 & 1 \\ 3 & 2 & 1 & 1 & 1 & 3 & 3 & 1 & 1 & 1 \end{bmatrix}$$

Solution: We can generate count matrices $\mathbf{M}_i$ of size $q_i \times r_i$ for each node by iterating through the data set and storing the counts. We then normalize each row in the count matrices to yield the matrices containing the maximum likelihood estimates of the parameters:

$$\mathbf{M}_1 = \begin{bmatrix} 7 & 3 \end{bmatrix} \qquad \mathbf{M}_2 = \begin{bmatrix} 3 & 1 \\ 0 & 0 \\ 2 & 0 \\ 0 & 2 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{M}_3 = \begin{bmatrix} 6 & 4 \end{bmatrix} \qquad \mathbf{M}_4 = \begin{bmatrix} 5 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\hat{\boldsymbol{\theta}}_1 = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \quad \hat{\boldsymbol{\theta}}_2 = \begin{bmatrix} 0.75 & 0.25 \\ \text{NAN} & \text{NAN} \\ 1.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 1.0 \\ 0.0 & 1.0 \end{bmatrix} \quad \hat{\boldsymbol{\theta}}_3 = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix} \quad \hat{\boldsymbol{\theta}}_4 \approx \begin{bmatrix} 0.71 & 0.0 & 0.29 \\ 0.33 & 0.33 & 0.34 \end{bmatrix}$$

Exercise 4.5. We have a biased coin, and we want to estimate the Bernoulli parameter ϕ that specifies the probability the coin lands on heads. If the first toss lands on heads ($o_1 = 1$), answer the following questions:

- What is the maximum likelihood estimate of ϕ?

- Using a uniform prior, what is the maximum a posteriori estimate of ϕ?
- Using a uniform prior, what is the expectation of our posterior distribution over ϕ?

Solution: Since our first toss lands on heads, we have $n = 1$ successes and $m = 1$ trials.

- The maximum likelihood estimate of ϕ is $n/m = 1$.
- Using a uniform $\text{Beta}(1, 1)$ prior, the posterior distribution is $\text{Beta}(1 + n, 1 + m - n) = \text{Beta}(2, 1)$. The maximum a posteriori estimate of ϕ or mode of the posterior distribution is
$$\frac{\alpha - 1}{\alpha + \beta - 2} = \frac{2 - 1}{2 + 1 - 2} = 1$$
- The mean of the posterior distribution is
$$\frac{\alpha}{\alpha + \beta} = \frac{2}{2 + 1} = \frac{2}{3}$$

Exercise 4.6. Suppose we are given the following data set, with one missing value. What is the value that will be imputed using marginal mode imputation, assuming that the marginal distribution is a Gaussian? What is the value that will be imputed using nearest-neighbor imputation?

X_1	X_2
0.5	1.0
?	0.3
−0.6	−0.3
0.1	0.2

Solution: Assuming that the marginal distribution over X_1 is a Gaussian, we can compute the marginal mode, which is the mean parameter of the Gaussian distribution:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x_i = \frac{0.5 - 0.6 + 0.1}{3} = 0$$

Thus, for marginal mode imputation, the missing value will be set to 0. For nearest-neighbor imputation, the nearest sample to $X_2 = 0.3$ is the fourth sample, so the missing value will be set to 0.1.

Exercise 4.7. Suppose we are given a data set over two variables $X_{1:2}$, with several missing values. We assume that $X_{1:2}$ are jointly Gaussian and use the fully-observed samples to fit the following distribution:

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 5 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 & 1 \\ 1 & 2 \end{bmatrix} \right)$$

What is the value that will be imputed for X_1 for the sample $X_2 = 1.5$ using posterior mode imputation? What distribution do we need to sample from for posterior sample imputation?

Solution: Since we assumed that $X_{1:2}$ are jointly Gaussian, the posterior distribution over X_1 given X_2 is also Gaussian, and its mode is the mean parameter of the posterior distribution. We can compute the mean of the posterior distribution as follows:

$$p(x_1 \mid x_2) = \mathcal{N}\left(x_1 \mid \mu_{x_1|x_2}, \sigma^2_{x_1|x_2}\right)$$
$$\mu_{x_1|x_2=1.5} = 5 + (1)(2)^{-1}(1.5 - 2) = 4.75$$

Thus, for posterior mode imputation, the missing value will be set to 4.75. For posterior sample imputation, we will sample a value $X_1 \sim \mathcal{N}(4.75, 3.5)$.

5 Structure Learning

The previous chapters of this book assumed that the structures of our probabilistic models were known. This chapter discusses methods for learning the structure of models from data.[1] We begin by explaining how to compute the probability of a graphical structure, given the data. Generally, we want to maximize this probability. Because the space of possible graphical structures is usually too large to enumerate, we also discuss ways to search this space efficiently.

[1] Overviews of Bayesian network structure learning can be found in the following textbooks: D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009. R. E. Neapolitan, *Learning Bayesian Networks*. Prentice Hall, 2003.

5.1 Bayesian Network Scoring

We want to be able to score a network structure G based on how well it models the data. A maximum a posteriori approach to structure learning involves finding a G that maximizes $P(G \mid D)$. We first explain how to compute a Bayesian score based on $P(G \mid D)$ to measure how well G models the data. We then explain how to go about searching the space of networks for the highest-scoring network. Like inference in Bayesian networks, it can be shown that for general graphs and input data, learning the structure of a Bayesian network is NP-hard.[2]

[2] See D. M. Chickering, "Learning Bayesian Networks is NP-Complete," in *Learning from Data: Artificial Intelligence and Statistics V*, D. Fisher and H.-J. Lenz, eds., Springer, 1996, pp. 121–130. D. M. Chickering, D. Heckerman, and C. Meek, "Large-Sample Learning of Bayesian Networks is NP-Hard," *Journal of Machine Learning Research*, vol. 5, pp. 1287–1330, 2004.

We compute $P(G \mid D)$ using Bayes' rule and the law of total probability:

$$P(G \mid D) \propto P(G)P(D \mid G) \tag{5.1}$$

$$= P(G) \int P(D \mid \theta, G) p(\theta \mid G) \, \mathrm{d}\theta \tag{5.2}$$

where θ contains the network parameters as introduced in the previous chapter. Integrating with respect to θ results in[3]

$$P(G \mid D) = P(G) \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0} + m_{ij0})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + m_{ijk})}{\Gamma(\alpha_{ijk})} \tag{5.3}$$

[3] For the derivation, see the appendix of G. F. Cooper and E. Herskovits, "A Bayesian Method for the Induction of Probabilistic Networks from Data," *Machine Learning*, vol. 4, no. 9, pp. 309–347, 1992.

where the values for α_{ijk} are the pseudocounts and m_{ijk} are the counts, as introduced in the previous chapter. We also define

$$\alpha_{ij0} = \sum_{k=1}^{r_i} \alpha_{ijk} \qquad m_{ij0} = \sum_{k=1}^{r_i} m_{ijk} \tag{5.4}$$

Finding the G that maximizes equation (5.2) is the same as finding the G that maximizes what is called the *Bayesian score*:

$$\log P(G \mid D) = \log P(G) + \sum_{i=1}^{n} \sum_{j=1}^{q_i} \left(\log \left(\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0} + m_{ij0})} \right) + \sum_{k=1}^{r_i} \log \left(\frac{\Gamma(\alpha_{ijk} + m_{ijk})}{\Gamma(\alpha_{ijk})} \right) \right) \tag{5.5}$$

The Bayesian score is more convenient to compute numerically because it is easier to add the logarithm of small numbers together than to multiply small numbers together. Many software libraries can compute the logarithm of the gamma function directly.

A variety of graph priors have been explored in the literature, although a uniform prior is often used in practice, in which case $\log P(G)$ can be dropped from the computation of the Bayesian score in equation (5.5). Algorithm 5.1 provides an implementation.

```
function bayesian_score_component(M, α)
    p =  sum(loggamma.(α + M))
    p -= sum(loggamma.(α))
    p += sum(loggamma.(sum(α,dims=2)))
    p -= sum(loggamma.(sum(α,dims=2) + sum(M,dims=2)))
    return p
end

function bayesian_score(vars, G, D)
    n = length(vars)
    M = statistics(vars, G, D)
    α = prior(vars, G)
    return sum(bayesian_score_component(M[i], α[i]) for i in 1:n)
end
```

Algorithm 5.1. An algorithm for computing the Bayesian score for a list of variables `vars` and a graph `G` given data `D`. This method uses a uniform prior $\alpha_{ijk} = 1$ for all i, j, and k as generated by algorithm 4.2. The `loggamma` function is provided by `SpecialFunctions.jl`. Chapter 4 introduced the `statistics` and `prior` functions. Note that $\log(\Gamma(\alpha)/\Gamma(\alpha + m)) = \log\Gamma(\alpha) - \log\Gamma(\alpha + m)$, and that $\log\Gamma(1) = 0$.

A by-product of optimizing the structure with respect to the Bayesian score is that we are able to find the right balance in the model complexity, given the available data. We do not want a model that misses out on capturing important relationships between variables, but we also do not want a model that has too many parameters to be adequately learned from limited data.

To illustrate how the Bayesian score helps us balance model complexity, consider the network in figure 5.1. The value of A weakly influences the value of B, and C is independent of the other variables. We sample from this ''true'' model to generate data D, and then try to learn the model structure. There are 25 possible network structures involving three variables, but we will focus on the scores for the models in figure 5.2.

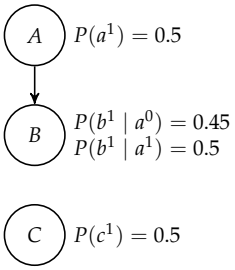


Figure 5.1. A simple Bayesian network to illustrate how the Bayesian score helps us balance model complexity.

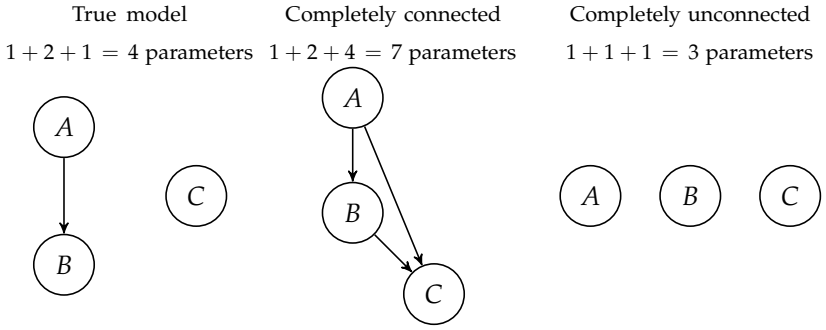


Figure 5.2. Three Bayesian network structures with varying levels of complexity.

Figure 5.3 shows how the Bayesian scores of the completely connected and unconnected models compare to the true model as the amount of data increases. In the plot, we subtract the score of the true model, so values above 0 indicate that the model provides a better representation than the true model, given the available data. The plot shows that the unconnected model does better than the true model when there are fewer than 5×10^3 samples. The completely connected model never does better than the true model, but it starts to do better than the unconnected model at about 10^4 samples because there are sufficient data to adequately estimate its seven independent parameters.

5.2 Directed Graph Search

In a *directed graph search*, we search the space of directed acyclic graphs for one that maximizes the Bayesian score. The space of possible Bayesian network structures grows superexponentially.[4] With 10 nodes, there are 4.2×10^{18} possible directed acyclic graphs. With 20 nodes, there are 2.4×10^{72}. Except for Bayesian networks with few nodes, we cannot enumerate the space of possible structures to find the highest-scoring network. Therefore, we have to rely on a search strategy.

[4] R. W. Robinson, "Counting Labeled Acyclic Digraphs," in *Ann Arbor Conference on Graph Theory*, 1973.

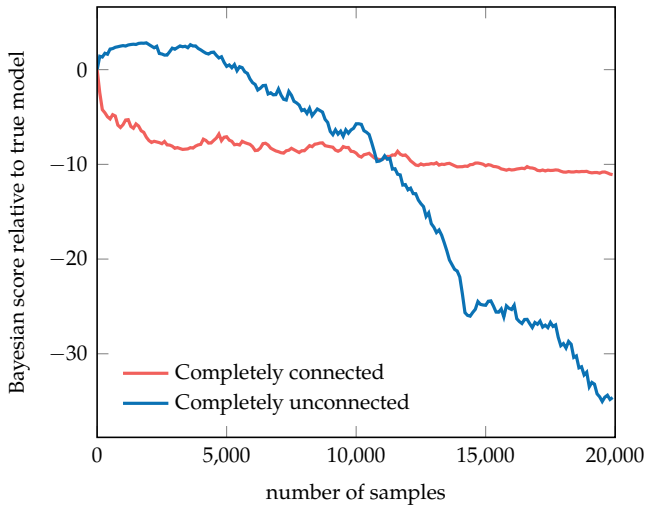


Figure 5.3. Bayesian network structure learning balances model complexity with the available data. The completely connected model never outperforms the true model, whereas the completely unconnected model eventually underperforms when more than about 5×10^3 samples have been drawn. This result indicates that simpler models can outperform complicated models when data is scarce—even when a more complicated model generated the samples.

Fortunately, search is a general problem, and a wide variety of generic search algorithms have been studied over the years.

One of the most common search strategies is called *K2*.[5] The search (algorithm 5.2) runs in polynomial time but does not guarantee finding a globally optimal network structure. It can use any scoring function, but it is often used with the Bayesian score because of its ability to balance the complexity of the model with the amount of data available. K2 begins with a graph with no directed edges and then iterates over the variables according to a provided ordering, greedily adding parents to the nodes in a way that maximally increases the score. It is common for K2 to impose an upper bound on the number of parents for any one node to reduce the required computation. The original K2 algorithm assumed a unit uniform Dirichlet prior with $\alpha_{ijk} = 1$ for all i, j, and k, but any prior can be used in principle.

A general search strategy is *local search*, which is sometimes called *hill climbing*. Algorithm 5.3 provides an implementation of this concept. We start with an initial graph and then move to the highest-scoring neighbor. The neighborhood of a graph consists of the graphs that are only one basic graph operation away, where the basic graph operations include introducing an edge, removing an edge, and reversing an edge. Of course, not all operations are possible from a particular

[5] The name comes from the fact that it is an evolution of a system called Kutató. The algorithm was introduced by G. F. Cooper and E. Herskovits, "A Bayesian Method for the Induction of Probabilistic Networks from Data," *Machine Learning*, vol. 4, no. 9, pp. 309–347, 1992.

```
struct K2Search
    ordering::Vector{Int} # variable ordering
end

function fit(method::K2Search, vars, D)
    G = SimpleDiGraph(length(vars))
    for (k,i) in enumerate(method.ordering[2:end])
        y = bayesian_score(vars, G, D)
        while true
            y_best, j_best = -Inf, 0
            for j in method.ordering[1:k]
                if !has_edge(G, j, i)
                    add_edge!(G, j, i)
                    y′ = bayesian_score(vars, G, D)
                    if y′ > y_best
                        y_best, j_best = y′, j
                    end
                    rem_edge!(G, j, i)
                end
            end
            if y_best > y
                y = y_best
                add_edge!(G, j_best, i)
            else
                break
            end
        end
    end
    return G
end
```

Algorithm 5.2. K2 search of the space of directed acyclic graphs using a specified variable ordering. This variable ordering imposes a topological ordering in the resulting graph. The `fit` function takes an ordered list variables `vars` and a data set `D`. The method starts with an empty graph and iteratively adds the next parent that maximally improves the Bayesian score.

graph, and operations that introduce cycles into the graph are invalid. The search continues until the current graph scores no lower than any of its neighbors.

An *opportunistic* version of local search is implemented in algorithm 5.3. Rather than generating all graph neighbors at every iteration, this method generates a single random neighbor and accepts it if its Bayesian score is greater than that of the current graph.

```
struct LocalDirectedGraphSearch
    G     # initial graph
    k_max # number of iterations
end

function rand_graph_neighbor(G)
    n = nv(G)
    i = rand(1:n)
    j = mod1(i + rand(2:n)-1, n)
    G′ = copy(G)
    has_edge(G, i, j) ? rem_edge!(G′, i, j) : add_edge!(G′, i, j)
    return G′
end

function fit(method::LocalDirectedGraphSearch, vars, D)
    G = method.G
    y = bayesian_score(vars, G, D)
    for k in 1:method.k_max
        G′ = rand_graph_neighbor(G)
        y′ = is_cyclic(G′) ? -Inf : bayesian_score(vars, G′, D)
        if y′ > y
            y, G = y′, G′
        end
    end
    return G
end
```

Algorithm 5.3. Local directed graph search, which starts with an initial directed graph `G` and opportunistically moves to a random graph neighbor whenever its Bayesian score is greater. It repeats this process for `k_max` iterations. Random graph neighbors are generated by either adding or removing a single edge. This algorithm can be extended to include reversing the direction of an edge. Edge addition can result in a graph with cycles, in which case we assign a score of $-\infty$.

Local search can get stuck in *local optima*, preventing it from finding the globally optimal network structure. Various strategies have been proposed for addressing local optima, including the following:[6]

- *Randomized restart*. Once a local optima has been found, simply restart the search at a random point in the search space.
- *Simulated annealing*. Instead of always moving to the neighbor with greatest fitness, the search can visit neighbors with lower fitness according to some randomized exploration strategy. As the search progresses, the randomness in

[6] The field of optimization is quite vast, and many methods have been developed for addressing local optima. This textbook provides an overview: M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

the exploration decreases according to a particular schedule. This approach is called simulated annealing because of its inspiration from annealing in metallurgy.

- *Genetic algorithms*. The procedure begins with an initial random population of points in the search space represented as binary strings. Each bit in a string indicates the presence or absence of an arrow between two nodes. String manipulation thus allows for searching the space of directed graphs. The individuals in the population reproduce at a rate proportional to their score. Individuals selected for reproduction have their strings recombined randomly through genetic crossover, which involves selecting a crossover point on two randomly selected individuals and then swapping the strings after that point. Mutations are also introduced randomly into the population by randomly flipping bits in the strings. The process of evolution continues until a satisfactory point in the search space is found.
- *Memetic algorithms*. This approach, sometimes called *genetic local search*, is simply a combination of genetic algorithms and local search. After genetic recombination, local search is applied to the individuals.
- *Tabu search*. Previous methods can be augmented to maintain a *tabu list* containing recently visited points in the search space. The search algorithm avoids neighbors in the tabu list.

Some search strategies may work better than others on certain data sets, but in general, finding the global optima remains NP-hard. Many applications, however, do not require the globally optimal network structure. A locally optimal structure is often acceptable.

5.3 Markov Equivalence Classes

As discussed earlier, the structure of a Bayesian network encodes a set of conditional independence assumptions. An important observation to make when trying to learn the structure of a Bayesian network is that two different graphs can encode the same independence assumptions. As a simple example, the two-variable network $A \rightarrow B$ has the same independence assumptions as $A \leftarrow B$. Solely on the basis of the data, we cannot justify the direction of the edge between A and B.

If two networks encode the same conditional independence assumptions, we say that they are *Markov equivalent*. It can be proven that two graphs are Markov equivalent if and only if they have (1) the same edges, without regard to direction; and (2) the same set of immoral v-structures. An *immoral v-structure* is a v-structure $X \rightarrow Y \leftarrow Z$, with X and Z not directly connected, as shown in figure 5.4. A *Markov equivalence class* is a set containing all the directed acyclic graphs that are Markov equivalent to each other. A method for checking Markov equivalence is given in algorithm 5.4.

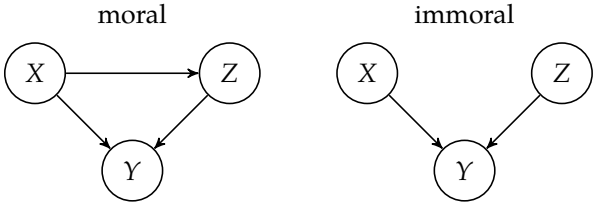


Figure 5.4. Moral and immoral v-structures.

In general, two structures belonging to the same Markov equivalence class may be given different scores. However, if the Bayesian score is used with Dirichlet priors such that $\kappa = \sum_j \sum_k \alpha_{ijk}$ is constant for all i, then two Markov equivalent structures are assigned the same score.[7] Such priors are called *BDe*, and a special case is the *BDeu* prior,[8] which assigns $\alpha_{ijk} = \kappa/(q_i r_i)$. Although the commonly used uniform prior $\alpha_{ijk} = 1$ does not always result in identical scores being assigned to structures in the same equivalence class, they are often fairly close. A scoring function that assigns the same score to all structures in the same class is called *score equivalent*.

[7] This was shown by D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian Networks: The Combination of Knowledge and Statistical Data," *Machine Learning*, vol. 20, no. 3, pp. 197–243, 1995.

[8] W. L. Buntine, "Theory Refinement on Bayesian Networks," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1991.

5.4 Partially Directed Graph Search

A Markov equivalence class can be represented as a *partially directed graph*, sometimes called an *essential graph* or a *directed acyclic graph pattern*. A partially directed graph can contain both directed edges and undirected edges. An example of a partially directed graph that encodes a Markov equivalence class is shown in figure 5.5. A directed acyclic graph G is a member of the Markov equivalence class encoded by a partially directed graph G' if and only if G has the same edges as G' without regard to direction and has the same immoral v-structures as G'.

```
function are_markov_equivalent(G, H)
    if nv(G) != nv(H) || ne(G) != ne(H) ||
        !all(has_edge(H, e) || has_edge(H, reverse(e))
                                        for e in edges(G))
        return false
    end
    for (I, J) in [(G,H), (H,G)]
        for c in 1:nv(I)
            parents = inneighbors(I, c)
             for (a, b) in subsets(parents, 2)
                 if !has_edge(I, a, b) && !has_edge(I, b, a) &&
                    !(has_edge(J, a, c) && has_edge(J, b, c))
                     return false
                 end
             end
        end
    end

    return true
end
```

Algorithm 5.4. A method for determining whether the directed acyclic graphs `G` and `H` are Markov equivalent. The `subsets` function from `IterTools.jl` returns all subsets of a given set and a specified size.

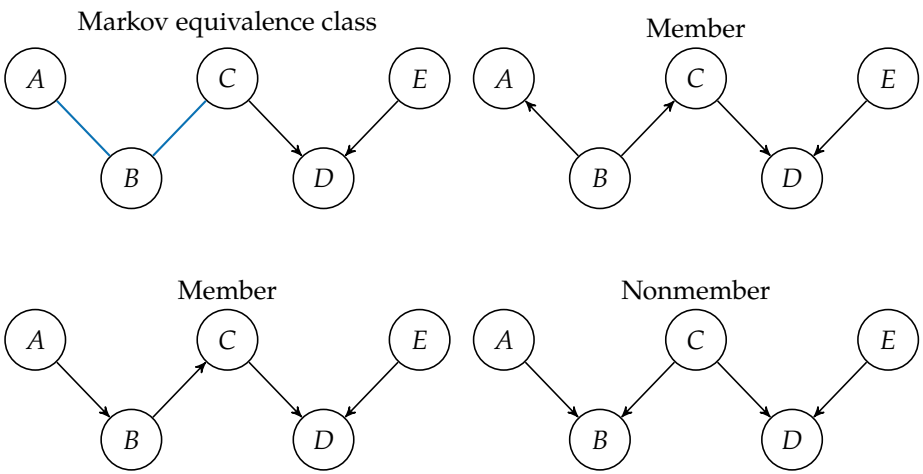


Figure 5.5. A Markov equivalence class and examples of members and a nonmember. The nonmember does not belong to the Markov equivalence class because it introduces an immoral v-structure, $A \rightarrow B \leftarrow C$, which is not indicated in the partially directed graph.

Instead of searching the space of directed acyclic graphs, we can search the space of Markov equivalence classes represented by partially directed graphs.[9] Although the space of Markov equivalence classes is, of course, smaller than the space of directed acyclic graphs, it is not significantly smaller; the ratio of directed acyclic graphs to equivalence classes asymptotes to around 3.7 fairly quickly.[10] A problem with hill climbing in the space of directed acyclic graphs is that the neighborhood may consist of other graphs that are in the same equivalence class with the same score, which can lead to the search becoming stuck in a local optimum. Searching the space of equivalence classes allows us to jump to different directed acyclic graphs outside the current equivalence class.

[9] Details of how to search this space are provided by D. M. Chickering, "Learning Equivalence Classes of Bayesian-Network Structures," *Journal of Machine Learning Research*, vol. 2, pp. 445–498, 2002.

[10] S. B. Gillispie and M. D. Perlman, "The Size Distribution for Markov Equivalence Classes of Acyclic Digraph Models," *Artificial Intelligence*, vol. 141, no. 1–2, pp. 137–155, 2002.

Any of the general search strategies presented in section 5.2 can be used. If a form of local search is used, then we need to define the local graph operations that define the neighborhood of the graph. Examples of local graph operations include:

- If an edge between X and Y does not exist, add either $X - Y$ or $X \rightarrow Y$.
- If $X - Y$ or $X \rightarrow Y$, then remove the edge between X and Y.
- If $X \rightarrow Y$, then reverse the direction of the edge to get $X \leftarrow Y$.
- If $X - Y - Z$, then add $X \rightarrow Y \leftarrow Z$.

To score a partially directed graph, we generate a member of its Markov equivalence class and compute its score.

5.5 Summary

- Fitting a Bayesian network to data requires selecting the Bayesian network structure that dictates the conditional dependencies between variables.
- Bayesian approaches to structure learning maximize the Bayesian score, which is related to the probability of the graph structure given a data set.
- The Bayesian score promotes simpler structures for smaller data sets and supports more complicated structures for larger data sets.
- The number of possible structures is superexponential in the number of variables, and finding a structure that maximizes the Bayesian score is NP-hard.

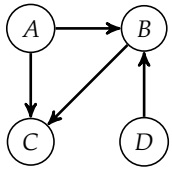
- Directed graph search algorithms like K2 and local search can be efficient but do not guarantee optimality.
- Methods like partially directed graph search traverse the space of Markov equivalence classes, which may be more efficient than searching the larger space of directed acyclic graphs.

5.6 Exercises

Exercise 5.1. How many neighbors does an edgeless directed acyclic graph with m nodes have?

Solution: Of the three basic graph operations, we can only add edges. We can add any edge to an edgeless directed acyclic graph and it will remain acyclic. There are $m(m-1) = m^2 - m$ node pairs, and therefore that many neighbors.

Exercise 5.2. How many networks are in the neighborhood of the following Bayesian network?

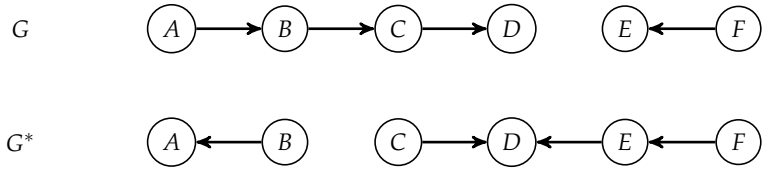


Solution: We can perform the following graph operations:

- Add $A \to D$, $D \to A$, $D \to C$
- Remove $A \to B$, $A \to C$, $B \to C$, $D \to B$
- Flip $A \to B$, $B \to C$, $D \to B$

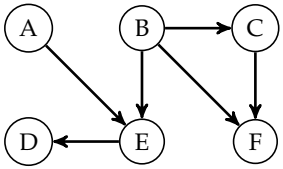
Thus, there are 10 Bayesian networks in the neighborhood.

Exercise 5.3. Suppose we start local search with a Bayesian network G. What is the fewest number of iterations of local search that could be performed to converge to the optimal Bayesian network G^*?

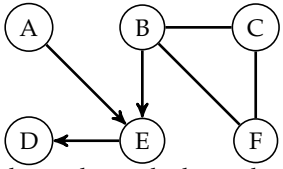


Solution: At each iteration, local search can move from the original network to a network in its neighborhood, which is at most one edge operation from the original network. Since there are three differences between the edges of G and G^*, performing local search from G would require a minimum of three iterations to arrive at G^*. One potential minimal sequence of local search iterations could be flipping $A \to B$, removing $B \to C$, and adding $E \to D$. We assume that the graphs formed with these edge operations yielded the highest Bayesian scores of all graphs in the considered neighborhood.

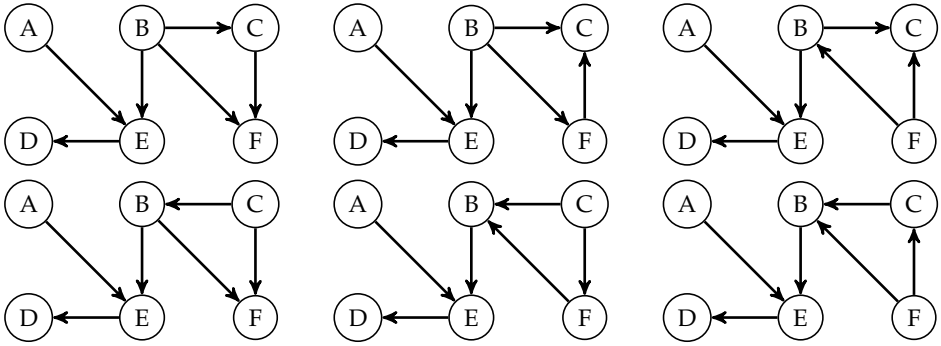
Exercise 5.4. Draw the partially directed acyclic graph representing the Markov equivalence class of the following Bayesian network. How many graphs are in this Markov equivalence class?



Solution: The Markov equivalence class can be represented by the following partially directed acyclic graph:

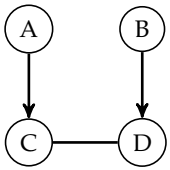


There are six networks in this Markov equivalence class, which are shown here:



Exercise 5.5. Give an example of a partially directed acyclic graph with four nodes that does not define a nonempty Markov equivalence class.

Solution: Consider the following partially directed acyclic graph:



We cannot replace the undirected edge with a directed edge because doing so would introduce a new v-structure.

6 Simple Decisions

This chapter introduces the notion of *simple decisions*, where we make a single decision under uncertainty.[1] We will study the problem of decision making from the perspective of *utility theory*, which involves modeling the preferences of an agent as a real-valued function over uncertain outcomes.[2] This chapter begins by discussing how a small set of constraints on rational preferences can lead to the existence of a utility function. This utility function can be inferred from a sequence of preference queries. We then introduce the maximum expected utility principle as a definition of rationality, a central concept in *decision theory* that will be used as a driving principle for decision making in this book.[3] We show how decision problems can be represented as decision networks and show an algorithm for solving for an optimal decision. The concept of value of information is introduced, which measures the utility gained through observing additional variables. The chapter concludes with a brief discussion of how human decision making is not always consistent with the maximum expected utility principle.

[1] Simple decisions are simple compared to sequential problems, which are the focus of the rest of the book. Simple decisions are not necessarily simple to solve, though.

[2] Schoemaker provides an overview of the development of utility theory. See P. J. H. Schoemaker, "The Expected Utility Model: Its Variants, Purposes, Evidence and Limitations," *Journal of Economic Literature*, vol. 20, no. 2, pp. 529–563, 1982. Fishburn surveys the field. See P. C. Fishburn, "Utility Theory," *Management Science*, vol. 14, no. 5, pp. 335–378, 1968.

[3] A survey of the field of decision theory is provided by M. Peterson, *An Introduction to Decision Theory*. Cambridge University Press, 2009.

6.1 Constraints on Rational Preferences

We began our discussion on uncertainty in chapter 2 by identifying the need to compare our degree of belief in different statements. This chapter requires the ability to compare the degree of desirability of two different outcomes. We state our preferences using the following operators:

- $A \succ B$ if we prefer A over B.
- $A \sim B$ if we are indifferent between A and B.
- $A \succeq B$ if we prefer A over B or are indifferent.

Just as beliefs can be subjective, so can preferences.

In addition to comparing events, our preference operators can be used to compare preferences over uncertain outcomes. A *lottery* is a set of probabilities associated with a set of outcomes. For example, if $S_{1:n}$ is a set of outcomes and $p_{1:n}$ are their associated probabilities, then the lottery involving these outcomes and probabilities is written as

$$[S_1 : p_1;\ \ \ldots;\ \ S_n : p_n] \tag{6.1}$$

The existence of a real-valued measure of utility emerges from a set of assumptions about preferences.[4] From this utility function, it is possible to define what it means to make rational decisions under uncertainty. Just as we imposed a set of constraints on beliefs, we will impose some constraints on preferences:[5]

- *Completeness.* Exactly one of the following holds: $A \succ B$, $B \succ A$, or $A \sim B$.
- *Transitivity.* If $A \succeq B$ and $B \succeq C$, then $A \succeq C$.
- *Continuity.* If $A \succeq C \succeq B$, then there exists a probability p such that $[A : p; B : 1 - p] \sim C$.
- *Independence.* If $A \succ B$, then for any C and probability p, $[A : p; C : 1 - p] \succeq [B : p; C : 1 - p]$.

These are constraints on *rational preferences*. They say nothing about the preferences of actual human beings; in fact, there is strong evidence that humans are not always rational (a point discussed further in section 6.7). Our objective in this book is to understand rational decision making from a computational perspective so that we can build useful systems. The possible extension of this theory to understanding human decision making is only of secondary interest.

[4] The theory of expected utility was introduced by the Swiss mathematician and physicist Daniel Bernoulli (1700–1782) in 1738. See D. Bernoulli, "Exposition of a New Theory on the Measurement of Risk," *Econometrica*, vol. 22, no. 1, pp. 23–36, 1954.

[5] These constraints are sometimes called the *von Neumann–Morgenstern axioms*, named after the Hungarian-American mathematician and physicist John von Neumann (1903–1957) and the Austrian-American economist Oskar Morgenstern (1902–1977). They formulated a variation of these axioms. See J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton University Press, 1944. Critiques of these axioms are discussed by P. Anand, "Are the Preference Axioms Really Rational?" *Theory and Decision*, vol. 23, no. 2, pp. 189–214, 1987.

6.2 Utility Functions

Just as constraints on the comparison of the plausibility of different statements lead to the existence of a real-valued probability measure, constraints on rational preferences lead to the existence of a real-valued *utility* measure. It follows from our constraints on rational preferences that there exists a real-valued utility function U such that

- $U(A) > U(B)$ if and only if $A \succ B$, and
- $U(A) = U(B)$ if and only if $A \sim B$.

The utility function is unique up to a *positive affine transformation*. In other words, for any constants $m > 0$ and b, $U'(S) = mU(S) + b$ if and only if the preferences induced by U' are the same as U. Utilities are like temperatures: you can compare temperatures using Kelvin, Celsius, or Fahrenheit, all of which are affine transformations of each other.

It follows from the constraints on rational preferences that the utility of a lottery is given by

$$U([S_1 : p_1;\ \ldots;\ S_n : p_n]) = \sum_{i=1}^{n} p_i U(S_i) \tag{6.2}$$

Example 6.1 applies this equation to compute the utility of outcomes involving a collision avoidance system.

Example 6.1. A lottery involving the outcomes of a collision avoidance system.

Suppose that we are building a collision avoidance system. The outcome of an encounter of an aircraft is defined by whether the system alerts (A) and whether a collision occurs (C). Because A and C are binary, there are four possible outcomes. So long as our preferences are rational, we can write our utility function over the space of possible lotteries in terms of four parameters: $U(a^0, c^0)$, $U(a^1, c^0)$, $U(a^0, c^1)$, and $U(a^1, c^1)$. For example,

$$U([a^0, c^0 : 0.5; \quad a^1, c^0 : 0.3; \quad a^0, c^1 : 0.1; \quad a^1, c^1 : 0.1])$$

is equal to

$$0.5U(a^0, c^0) + 0.3U(a^1, c^0) + 0.1U(a^0, c^1) + 0.1U(a^1, c^1)$$

If the utility function is bounded, then we can define a *normalized utility function*, where the best possible outcome is assigned utility 1 and the worst possible outcome is assigned utility 0. The utility of each of the other outcomes is scaled and translated as necessary.

6.3 Utility Elicitation

In building a decision-making or decision support system, it is often helpful to infer the utility function from a human or a group of humans. This approach is called *utility elicitation* or *preference elicitation*.[6] One way to go about doing this is to fix the utility of the worst outcome $\underline{S}$ to 0 and the best outcome $\overline{S}$ to 1. So long as the utilities of the outcomes are bounded, we can translate and scale the utilities without altering our preferences. If we want to determine the utility of outcome S, then we determine probability p such that $S \sim [\overline{S} : p; \underline{S} : 1 - p]$. It then follows that $U(S) = p$. Example 6.2 applies this process to determine the utility function associated with a collision avoidance problem.

[6] A variety of methods for utility elicitation are surveyed by P. H. Farquhar, "Utility Assessment Methods," *Management Science*, vol. 30, no. 11, pp. 1283–1300, 1984.

Example 6.2. Utility elicitation applied to collision avoidance.

In our collision avoidance example, the best possible event is to not alert and not have a collision, and so we set $U(a^0, c^0) = 1$. The worst possible event is to alert and have a collision, and so we set $U(a^1, c^1) = 0$. We define the lottery $L(p)$ to be $[a^0, c^0 : p;\ \ a^1, c^1 : 1 - p]$. To determine $U(a^1, c^0)$, we must find p such that $(a^1, c^0) \sim L(p)$. Similarly, to determine $U(a^0, c^1)$, we find p such that $(a^0, c^1) \sim L(p)$.

It may be tempting to use monetary values to infer utility functions. For example, if we are building a decision support system for managing wildfires, it may be tempting to define a utility function in terms of the monetary cost incurred by property damage and the monetary cost for deploying fire suppression resources. However, it is well known in economics that the utility of wealth, in general, is not linear.[7] If there were a linear relationship between utility and wealth, then decisions should be made in terms of maximizing expected monetary value. Someone who tries to maximize expected monetary value would have no use for insurance because the expected monetary values of insurance policies are generally negative.

[7] H. Markowitz, "The Utility of Wealth," *Journal of Political Economy*, vol. 60, no. 2, pp. 151–158, 1952.

Instead of trying to maximize expected wealth, we generally want to maximize the expected utility of wealth. Of course, different people have different utility functions. Figure 6.1 shows an example of a utility function. For small amounts of wealth, the curve is roughly linear, where $100 is about twice as good at $50. For larger amounts of wealth, however, the curve tends to flatten out; after all,

$1000 is worth less to a billionaire than it is to the average person. The flattening of the curve is sometimes referred to as *diminishing marginal utility*.

When discussing monetary utility functions, the three terms listed here are often used. To illustrate this, assume that A represents being given $50 and B represents a 50 % chance of winning $100.

- *Risk neutral.* The utility function is linear. There is no preference between $50 and the 50 % chance of winning $100 ($A \sim B$).
- *Risk seeking.* The utility function is convex. There is a preference for the 50 % chance of winning $100 ($A \prec B$).
- *Risk averse.* The utility function is concave. There is a preference for the $50 ($A \succ B$).

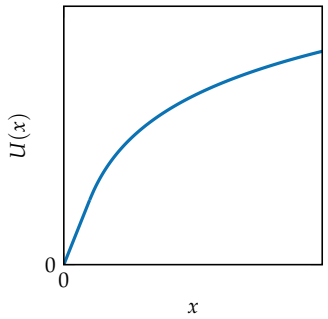


Figure 6.1. The utility of wealth x is often modeled as linear for small values and then concave for larger values, exhibiting risk aversion.

There are several common functional forms for modeling risk aversion of scalar quantities,[8] such as wealth or the availability of hospital beds. One is *quadratic utility*:

$$U(x) = \lambda x - x^2 \tag{6.3}$$

where the parameter $\lambda > 0$ controls the risk aversion. Since we generally want this utility function to be monotonically increasing when modeling the utility of quantities like wealth, we would cap this function at $x = \lambda/2$. After that point, the utility starts decreasing. Another simple form is *exponential utility*:

$$U(x) = 1 - e^{-\lambda x} \tag{6.4}$$

with $\lambda > 0$. Although it has a convenient mathematical form, it is generally not viewed as a plausible model of the utility of wealth. An alternative is the *power utility*:

$$U(x) = \frac{x^{1-\lambda} - 1}{1 - \lambda} \tag{6.5}$$

with $\lambda \geq 0$ and $\lambda \neq 1$. The *logarithmic utility*

$$U(x) = \log x \tag{6.6}$$

with $x > 0$ can be viewed as a special case of the power utility where $\lambda \to 1$. Figure 6.2 shows a plot of the power utility function with the logarithmic utility as a special case.

[8] These functional forms have been well studied within economics and finance. J. E. Ingersoll, *Theory of Financial Decision Making*. Rowman and Littlefield Publishers, 1987.

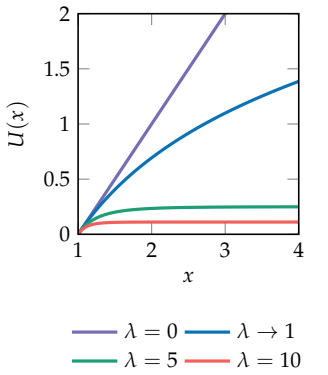


Figure 6.2. Power utility functions.

6.4 Maximum Expected Utility Principle

We are interested in the problem of making rational decisions with imperfect knowledge of the state of the world. Suppose that we have a probabilistic model $P(s' \mid o, a)$, which represents the probability that the state of the world becomes s', given that we observe o and take action a. We have a utility function $U(s')$ that encodes our preferences over the space of outcomes. Our *expected utility* of taking action a, given observation o, is given by

$$EU(a \mid o) = \sum_{s'} P(s' \mid a, o) U(s') \tag{6.7}$$

The *principle of maximum expected utility* says that a rational agent should choose the action that maximizes expected utility:

$$a^* = \arg\max_a EU(a \mid o) \tag{6.8}$$

Because we are interested in building rational agents, equation (6.8) plays a central role in this book.[9] Example 6.3 applies this principle to a simple decision problem.

[9] The importance of the maximum expected utility principle to the field of artificial intelligence is discussed by S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

6.5 Decision Networks

A *decision network*, sometimes called an *influence diagram*, is a generalization of a Bayesian network to include action and utility nodes so that we may compactly represent the probability and utility models defining a decision problem.[10] The state, action, and observation spaces in the previous section may be factored, and the structure of a decision network captures the relationships between the various components.

[10] An extensive discussion of decision networks can be found in F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. Springer, 2007.

Decision networks are composed of three types of nodes:

- A *chance node* corresponds to a random variable (indicated by a circle).
- An *action node* corresponds to a decision variable (indicated by a square).
- A *utility node* corresponds to a utility variable (indicated by a diamond) and cannot have children.

Example 6.3. Applying the principle of maximum expected utility to the simple decision of whether to bring an umbrella.

Suppose that we are trying to decide whether to bring an umbrella on our vacation given the weather forecast for our destination. We observe the forecast o, which may be either rain or sun. Our action a is either to bring our umbrella or leave our umbrella. The resulting state s' is a combination of whether we brought our umbrella and whether there is sun or rain at our destination. Our probabilistic model is as follows:

o	a	s'	$P(s' \mid a, o)$
forecast rain	bring umbrella	rain with umbrella	0.9
forecast rain	leave umbrella	rain without umbrella	0.9
forecast rain	bring umbrella	sun with umbrella	0.1
forecast rain	leave umbrella	sun without umbrella	0.1
forecast sun	bring umbrella	rain with umbrella	0.2
forecast sun	leave umbrella	rain without umbrella	0.2
forecast sun	bring umbrella	sun with umbrella	0.8
forecast sun	leave umbrella	sun without umbrella	0.8

As shown in the table, we assume that our forecast is imperfect; rain forecasts are right 90 % of the time and sun forecasts are right 80 % of the time. In addition, we assume that bringing an umbrella does not affect the weather, though some may question this assumption. The utility function is as follows:

s'	$U(s')$
rain with umbrella	-0.1
rain without umbrella	-1
sun with umbrella	0.9
sun without umbrella	1

We can compute the expected utility of bringing our umbrella if we forecast rain using equation (6.7):

$$EU(\text{bring umbrella} \mid \text{forecast rain}) = 0.9 \times -0.1 + 0.1 \times 0.9 = 0$$

Likewise, we can compute the expected utility of leaving our umbrella if we forecast rain using equation (6.7):

$$EU(\text{leave umbrella} \mid \text{forecast rain}) = 0.9 \times -1 + 0.1 \times 1 = -0.8$$

Hence, we will want to bring our umbrella.

There are three kinds of directed edges:

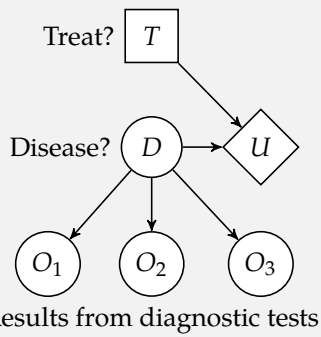
- A *conditional edge* ends in a chance node and indicates that the uncertainty in that chance node is conditioned on the values of all its parents.
- An *informational edge* ends in an action node and indicates that the decision associated with that node is made with knowledge of the values of its parents. (These edges are often drawn with dashed lines and are sometimes omitted from diagrams for simplicity.)
- A *functional edge* ends in a utility node and indicates that the utility node is determined by the outcomes of its parents.

Like Bayesian networks, decision networks cannot have cycles. The utility associated with an action is equal to the sum of the values at all the utility nodes. Example 6.4 illustrates how a decision network can model the problem of whether to treat a disease, given the results of diagnostic tests.

Example 6.4. An example of a decision network used to model whether to treat a disease, given information from diagnostic tests.

We have a set of results from diagnostic tests that may indicate the presence of a particular disease. Given what is known about the tests, we need to decide whether to apply a treatment. The utility is a function of whether a treatment is applied and whether the disease is actually present. Conditional edges connect D to O_1, O_2, and O_3. Informational edges are not explicitly shown in the illustration, but they would connect the observations to T. Functional edges connect T and D to U.

T	D	$U(T, D)$
0	0	0
0	1	−10
1	0	−1
1	1	−1



Solving a simple problem (algorithm 6.1) requires iterating over all possible decision instantiations to find a decision that maximizes expected utility. For each

instantiation, we evaluate the associated expected utility. We begin by instantiating the action nodes and observed chance nodes. We can then apply any inference algorithm to compute the posterior over the inputs to the utility function. The expected utility is the sum of the values at the utility nodes. Example 6.5 shows how this process can be applied to our running example.

```
struct SimpleProblem
    bn::BayesianNetwork
    chance_vars::Vector{Variable}
    decision_vars::Vector{Variable}
    utility_vars::Vector{Variable}
    utilities::Dict{Symbol, Vector{Float64}}
end

function solve(𝒫::SimpleProblem, evidence, M)
    query = [var.name for var in 𝒫.utility_vars]
    U(a) = sum(𝒫.utilities[uname][a[uname]] for uname in query)
    best = (a=nothing, u=-Inf)
    for assignment in assignments(𝒫.decision_vars)
        evidence = merge(evidence, assignment)
        ϕ = infer(M, 𝒫.bn, query, evidence)
        u = sum(p*U(a) for (a, p) in ϕ.table)
        if u > best.u
            best = (a=assignment, u=u)
        end
    end
    return best
end
```

Algorithm 6.1. A simple problem as a decision network. A decision network is a Bayesian network with chance, decision, and utility variables. Utility variables are treated as deterministic. Because variables in our Bayesian network take values from $1:r_i$, the utility variables are mapped to real values by the `utilities` field. For example, if we have a utility variable `:u1`, the `i`th utility associated with that variable is `utilities[:u1][i]`. The `solve` function takes as input the problem, evidence, and an inference method. It returns the best assignment to the decision variables and its associated expected utility.

A variety of methods have been developed to make evaluating decision networks more efficient.[11] One method involves removing action and chance nodes from decision networks if they have no children, as defined by conditional, informational, or functional edges. In example 6.5, we can remove O_2 and O_3 because they have no children. We cannot remove O_1 because we treated it as observed, indicating that there is an informational edge from O_1 to T (although it is not drawn explicitly).

6.6 Value of Information

We make decisions based on what we observe. In many applications, it is natural to want to quantify the *value of information*, which is how much observing additional variables is expected to increase our utility.[12] For example, in the disease treatment

[11] R. D. Shachter, "Evaluating Influence Diagrams," *Operations Research*, vol. 34, no. 6, pp. 871–882, 1986. R. D. Shachter, "Probabilistic Inference and Influence Diagrams," *Operations Research*, vol. 36, no. 4, pp. 589–604, 1988.

[12] R. A. Howard, "Information Value Theory," *IEEE Transactions on Systems Science and Cybernetics*, vol. 2, no. 1, pp. 22–26, 1966. Applications to decision networks can be found in: S. L. Dittmer and F. V. Jensen, "Myopic Value of Information in Influence Diagrams," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1997. R. D. Shachter, "Efficient Value of Information Computation," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.

Example 6.5. Decision network evaluation of the diagnostic test problem.

We can use equation (6.7) to compute the expected utility of treating a disease for the decision network in example 6.4. For now, we will assume that we have the result from only the first diagnostic test and it came back positive. If we wanted to make the knowledge of the first diagnostic test explicit in the diagram, then we would draw an informational edge from O_1 to T, and we would have

$$EU(t^1 \mid o_1^1) = \sum_{o_3} \sum_{o_2} \sum_{d} P(d, o_2, o_3 \mid t^1, o_1^1) U(t^1, d, o_1^1, o_2, o_3)$$

We can use the chain rule for Bayesian networks and the definition of conditional probability to compute $P(d, o_2, o_3 \mid t^1, o_1^1)$. Because the utility node depends only on whether the disease is present and whether we treat it, we can simplify $U(t^1, d, o_1^1, o_2, o_3)$ to $U(t^1, d)$. Hence,

$$EU(t^1 \mid o_1^1) = \sum_{d} P(d \mid t^1, o_1^1) U(t^1, d)$$

Any of the exact or approximate inference methods introduced in the previous chapter can be used to evaluate $P(d \mid t^1, o_1^1)$. To decide whether to apply a treatment, we compute $EU(t^1 \mid o_1^1)$ and $EU(t^0 \mid o_1^1)$ and make the decision that provides the highest expected utility.

application in example 6.5, we assumed that we have only observed o_1^1. Given the positive result from that one diagnostic test alone, we may decide against treatment. However, it may be beneficial to administer additional diagnostic tests to reduce the risk of not treating a disease that is really present.

In computing the value of information, we will use $EU^*(o)$ to denote the expected utility of an optimal action, given observation o. The value of information about variable O', given o, is

$$VOI(O' \mid o) = \left(\sum_{o'} P(o' \mid o) EU^*(o, o')\right) - EU^*(o) \tag{6.9}$$

In other words, the value of information about a variable is the increase in expected utility if that variable is observed. Algorithm 6.2 provides an implementation of this.

```
function value_of_information(𝒫, query, evidence, M)
    ϕ = infer(M, 𝒫.bn, query, evidence)
    voi = -solve(𝒫, evidence, M).u
    query_vars = filter(v->v.name ∈ query, 𝒫.chance_vars)
    for o′ in assignments(query_vars)
        oo′ = merge(evidence, o′)
        p = ϕ.table[o′]
        voi += p*solve(𝒫, oo′, M).u
    end
    return voi
end
```

Algorithm 6.2. A method for computing the value of information of a query `query` given observed chance variables and their values `evidence`. The method additionally takes a simple problem $\mathcal{P}$ and an inference strategy `M`.

The value of information is never negative. The expected utility can increase only if additional observations can lead to different optimal decisions. If observing a new variable O' makes no difference in the choice of action, then $EU^*(o, o') = EU^*(o)$ for all o', in which case equation (6.9) evaluates to 0. For example, if the optimal decision is to treat the disease regardless of the outcome of the *diagnostic test*, then the value of observing the outcome of the test is 0.

The value of information only captures the increase in expected utility from making an observation. A cost may be associated with making a particular observation. Some diagnostic tests may be inexpensive, such as a temperature reading; other diagnostic tests are more costly and invasive, such as a lumbar puncture. The value of information obtained by a lumbar puncture may be much greater than that of a temperature reading, but the costs of the tests should be taken into consideration.

Value of information is an important and often-used metric for choosing what to observe. Sometimes the value of information metric is used to determine an appropriate sequence of observations. After each observation, the value of information is determined for the remaining unobserved variables. The unobserved variable with the greatest value of information is then selected for observation. If there are costs associated with making different observations, then these costs are subtracted from the value of information when determining which variable to observe. The process continues until it is no longer beneficial to observe any more variables. The optimal action is then chosen. This greedy selection of observations is only a heuristic; it may not represent the truly optimal sequence of observations. The optimal selection of observations can be determined by using the techniques for sequential decision making introduced in later chapters.

6.7 Irrationality

Decision theory is a *normative theory*, which is prescriptive, not a *descriptive theory*, which is predictive of human behavior. Human judgment and preference often do not follow the rules of rationality outlined in section 6.1.[13] Even human experts may have an inconsistent set of preferences, which can be problematic when designing a decision support system that attempts to maximize expected utility.

Example 6.6 shows that certainty often exaggerates losses that are certain compared to losses that are merely probable. This *certainty effect* works with gains as well. A smaller gain that is certain is often preferred over a much greater gain that is only probable, in a way that the axioms of rationality are necessarily violated.

Example 6.7 demonstrates the *framing effect*, where people decide on options based on whether they are presented as a loss or as a gain. Many other cognitive biases can lead to deviations from what is prescribed by utility theory.[14] Special care must be given when trying to elicit utility functions from human experts to build decision support systems. Although the recommendations of the decision support system may be rational, they may not exactly reflect human preferences in certain situations.

[13] Kahneman and Tversky provide a critique of expected utility theory and introduce an alternative model called *prospect theory*, which appears to be more consistent with human behavior. D. Kahneman and A. Tversky, "Prospect Theory: An Analysis of Decision Under Risk," *Econometrica*, vol. 47, no. 2, pp. 263–292, 1979.

[14] Several recent books discuss apparent human irrationality. D. Ariely, *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2008. J. Lehrer, *How We Decide*. Houghton Mifflin, 2009.

Example 6.6. An experiment demonstrating that certainty often exaggerates losses that are certain relative to losses that are merely probable. A. Tversky and D. Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, vol. 211, no. 4481, pp. 453–458, 1981.

Tversky and Kahneman studied the preferences of university students who answered questionnaires in a classroom setting. They presented students with questions dealing with the response to an epidemic. The students were to reveal their preference between the following two outcomes:

- A: 100 % chance of losing 75 lives
- B: 80 % chance of losing 100 lives

Most preferred B over A. From equation (6.2), we know

$$U(\text{lose } 75) < 0.8U(\text{lose } 100) \tag{6.10}$$

They were then asked to choose between the following two outcomes:

- C: 10 % chance of losing 75 lives
- D: 8 % chance of losing 100 lives

Most preferred C over D. Hence, $0.1U(\text{lose } 75) > 0.08U(\text{lose } 100)$. We multiply both sides by 10 and get

$$U(\text{lose } 75) > 0.8U(\text{lose } 100) \tag{6.11}$$

Of course, equations (6.10) and (6.11) result in a contradiction. We have made no assumption about the actual value of $U(\text{lose } 75)$ and $U(\text{lose } 100)$—we did not even assume that losing 100 lives was worse than losing 75 lives. Because equation (6.2) follows directly from the von Neumann–Morgenstern axioms given in section 6.1, there must be a violation of at least one of the axioms, even though many people who select B and C seem to find the axioms agreeable.

Tversky and Kahneman demonstrated the *framing effect* using a hypothetical scenario in which an epidemic is expected to kill 600 people. They presented students with the following two outcomes:

- E: 200 people will be saved.
- F: 1/3 chance that 600 people will be saved and 2/3 chance that no people will be saved.

The majority of students chose E over F. They then asked them to choose between the following:

- G: 400 people will die.
- H: 1/3 chance that nobody will die and 2/3 chance that 600 people will die.

The majority of students chose H over G, even though E is equivalent to G and F is equivalent to H. This inconsistency is due to how the question is framed.

Example 6.7. An experiment demonstrating the framing effect. A. Tversky and D. Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, vol. 211, no. 4481, pp. 453–458, 1981.

6.8 Summary

- Rational decision making combines probability and utility theory.
- The existence of a utility function follows from constraints on rational preferences.
- A rational decision is one that maximizes expected utility.
- Decision problems can be modeled using decision networks, which are extensions of Bayesian networks that include actions and utilities.
- Solving a simple decision involves inference in Bayesian networks and is thus NP-hard.
- The value of information measures the gain in expected utility should a new variable be observed.
- Humans are not always rational.

6.9 Exercises

Exercise 6.1. Suppose that we have a utility function $U(s)$ with a finite maximum value $\overline{U}$ and a finite minimum value $\underline{U}$. What is the corresponding normalized utility function $\hat{U}(s)$ that preserves the same preferences?

Solution: A normalized utility function has a maximum value of 1 and a minimum value of 0. Preferences are preserved under affine transforms, so we determine the affine transform of $U(s)$ that matches the unit bounds. This transform is

$$\hat{U}(s) = \frac{U(s) - \underline{U}}{\overline{U} - \underline{U}} = \frac{1}{\overline{U} - \underline{U}} U(s) - \frac{\underline{U}}{\overline{U} - \underline{U}}$$

Exercise 6.2. If $A \succeq C \succeq B$ and the utilities of each outcome are $U(A) = 450$, $U(B) = -150$, and $U(C) = 60$, what is the lottery over A and B that will make us indifferent between the lottery and C?

Solution: A lottery over A and B is defined as $[A : p; B : 1 - p]$. To satisfy indifference between the lottery and C ($[A : p; B : 1 - p] \sim C$), we must have $U([A : p; B : 1 - p]) = U(C)$. Thus, we must compute p that satisfies the equality

$$\begin{aligned} U([A : p; B : 1 - p]) &= U(C) \\ pU(A) + (1 - p)U(B) &= U(C) \\ p &= \frac{U(C) - U(B)}{U(A) - U(B)} \\ p &= \frac{60 - (-150)}{450 - (-150)} = 0.35 \end{aligned}$$

This implies that the lottery $[A : 0.35; B : 0.65]$ is equally as desired as C.

Exercise 6.3. Suppose that for a utility function U over three outcomes A, B, and C, that $U(A) = 5$, $U(B) = 20$, and $U(C) = 0$. We are given a choice between a lottery that gives us a 50 % probability of B and a 50 % probability of C and a lottery that guarantees A. Compute the preferred lottery and show that, under the positive affine transformation with $m = 2$ and $b = 30$, that we maintain a preference for the same lottery.

Solution: The first lottery is given by $[A : 0.0; B : 0.5; C : 0.5]$, and the second lottery is given by $[A : 1.0; B : 0.0; C : 0.0]$. The original utilities for each lottery are given by

$$\begin{aligned} U([A : 0.0; B : 0.5; C : 0.5]) &= 0.0U(A) + 0.5U(B) + 0.5U(C) = 10 \\ U([A : 1.0; B : 0.0; C : 0.0]) &= 1.0U(A) + 0.0U(B) + 0.0U(C) = 5 \end{aligned}$$

Thus, since $U([A : 0.0; B : 0.5; C : 0.5]) > U([A : 1.0; B : 0.0; C : 0.0])$, we prefer the first lottery. Under the positive affine transformation $m = 2$ and $b = 30$, our new utilities can be computed as $U' = 2U + 30$. The new utilities are then $U'(A) = 40$, $U'(B) = 70$, and $U'(C) = 30$. The new utilities for each lottery are

$$\begin{aligned} U'([A : 0.0; B : 0.5; C : 0.5]) &= 0.0U'(A) + 0.5U'(B) + 0.5U'(C) = 50 \\ U'([A : 1.0; B : 0.0; C : 0.0]) &= 1.0U'(A) + 0.0U'(B) + 0.0U'(C) = 40 \end{aligned}$$

Since $U'([A : 0.0; B : 0.5; C : 0.5]) > U'([A : 1.0; B : 0.0; C : 0.0])$, we maintain a preference for the first lottery.

Exercise 6.4. Prove that the power utility function in equation (6.5) is risk averse for all $x > 0$ and $\lambda > 0$ with $\lambda \neq 1$.

Solution: Risk aversion implies that the utility function is concave, which requires that the second derivative of the utility function is negative. The utility function and its derivatives are computed as follows:

$$U(x) = \frac{x^{1-\lambda} - 1}{1 - \lambda}$$
$$\frac{dU}{dx} = \frac{1}{x^{\lambda}}$$
$$\frac{d^2U}{dx^2} = \frac{-\lambda}{x^{\lambda+1}}$$

For $x > 0$ and $\lambda > 0$, $\lambda \neq 1$, $x^{\lambda+1}$ is a positive number raised to a positive exponent, which is guaranteed to be positive. Multiplying this by $-\lambda$ guarantees that the second derivative is negative. Thus, for all $x > 0$ and $\lambda > 0$, $\lambda \neq 1$, the power utility function is risk averse.

Exercise 6.5. Using the parameters given in example 6.3, compute the expected utility of bringing our umbrella if we forecast sun and the expected utility of leaving our umbrella behind if we forecast sun. What is the action that maximizes our expected utility, given that we forecast sun?

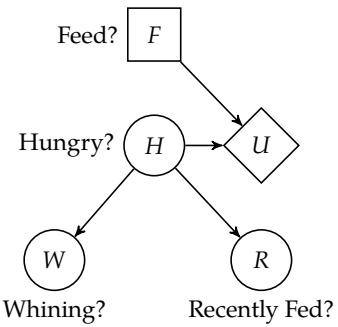
Solution:

$$EU(\text{bring umbrella} \mid \text{forecast sun}) = 0.2 \times -0.1 + 0.8 \times 0.9 = 0.7$$
$$EU(\text{leave umbrella} \mid \text{forecast sun}) = 0.2 \times -1.0 + 0.8 \times 1.0 = 0.6$$

The action that maximizes our expected utility if we forecast sun is to bring our umbrella!

Exercise 6.6. Suppose that we are trying to optimally decide whether or not to feed (F) our new puppy based on the likelihood that the puppy is hungry (H). We can observe whether the puppy is whining (W) and whether someone else has recently fed the puppy (R). The utilities of each combination of feeding and hunger and the decision network representation are provided here:

F	H	$U(F, H)$
0	0	0.0
0	1	−1.0
1	0	−0.5
1	1	−0.1



Given that $P(h^1 \mid w^1) = 0.78$, if we observe the puppy whining (w^1), what are the expected utilities of not feeding the puppy (f^0) and feeding the puppy (f^1)? What is the optimal action?

Solution: We start with the definition of expected utility and recognize that the utility depends only on H and F:

$$EU(f^0 \mid w^1) = \sum_h P(h \mid w^1) U(f^0, h)$$

Now, we can compute the expected utility of feeding the puppy given that it is whining and, in a similar fashion as before, the expected utility of not feeding the puppy given that it is whining:

$$EU(f^0 \mid w^1) = 0.22 \times 0.0 + 0.78 \times -1.0 = -0.78$$
$$EU(f^1 \mid w^1) = 0.22 \times -0.5 + 0.78 \times -0.1 = -0.188$$

Thus, the optimal action is to feed the puppy (f^1) since this maximizes our expected utility $EU^*(w^1) = -0.188$.

Exercise 6.7. Using the results from exercise 6.6, if $P(r^1 \mid w^1) = 0.2$, $P(h^1 \mid w^1, r^0) = 0.9$, and $P(h^1 \mid w^1, r^1) = 0.3$, what is the value of information of asking someone else if the puppy has recently been fed, given that we observe the puppy to be whining (w^1)?

Solution: We are interested in computing

$$\text{VOI}(R \mid w^1) = \left(\sum_r P(r \mid w^1) EU^*(w^1, r) \right) - EU^*(w^1)$$

We start by computing $EU(f \mid w^1, r)$ for all f and r. Following a similar derivation as in exercise 6.6, we have

$$EU(f^0 \mid w^1, r^0) = \sum_h P(h \mid w^1, r^0) U(f^0, h)$$

So, for each combination of F and R, we have the following expected utilities:

$$EU(f^0 \mid w^1, r^0) = \sum_h P(h \mid w^1, r^0) U(f^0, h) = 0.1 \times 0.0 + 0.9 \times -1.0 = -0.9$$
$$EU(f^1 \mid w^1, r^0) = \sum_h P(h \mid w^1, r^0) U(f^1, h) = 0.1 \times -0.5 + 0.9 \times -0.1 = -0.14$$
$$EU(f^0 \mid w^1, r^1) = \sum_h P(h \mid w^1, r^1) U(f^0, h) = 0.7 \times 0.0 + 0.3 \times -1.0 = -0.3$$
$$EU(f^1 \mid w^1, r^1) = \sum_h P(h \mid w^1, r^1) U(f^1, h) = 0.7 \times -0.5 + 0.3 \times -0.1 = -0.38$$

The optimal expected utilities are

$$EU^*(w^1, r^0) = -0.14$$
$$EU^*(w^1, r^1) = -0.3$$

Now, we can compute the value of information:

$$\text{VOI}(R \mid w^1) = 0.8(-0.14) + 0.2(-0.3) - (-0.188) = 0.016$$

PART II

SEQUENTIAL PROBLEMS

Up to this point, we have assumed that we make a single decision at one point in time, but many important problems require that we make a series of decisions. The same principle of maximum expected utility still applies, but optimal decision making in a sequential context requires reasoning about future sequences of actions and observations. This part of the book will discuss sequential decision problems in stochastic environments. We will focus on a general formulation of sequential decision problems under the assumption that the model is known and that the environment is fully observable. We will relax both of these assumptions later. Our discussion will begin with the introduction of the *Markov decision process* (*MDP*), the standard mathematical model for sequential decision problems. We will discuss several approaches for finding exact solutions. Because large problems sometimes do not permit exact solutions to be efficiently found, we will discuss a collection of both offline and online approximate solution methods, along with a type of method that involves directly searching the space of parameterized decision policies. Finally, we will discuss approaches for validating that our decision strategies will perform as expected when deployed in the real world.

7 Exact Solution Methods

This chapter introduces a model known as a *Markov decision process* (*MDP*) to represent sequential decision problems where the effects of our actions are uncertain.[1] We begin with a description of the model, which specifies both the stochastic dynamics of the system as well as the utility associated with its evolution. Different algorithms can be used to compute the utility associated with a decision strategy and to search for an optimal strategy. Under certain assumptions, we can find exact solutions to MDPs. Later chapters will discuss approximation methods that tend to scale better to larger problems.

[1] Such models were originally studied in the 1950s. R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957. A modern treatment can be found in M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

7.1 Markov Decision Processes

In an MDP (algorithm 7.1), we choose action a_t at time t based on observing state s_t. We then receive a reward r_t. The *action space* $\mathcal{A}$ is the set of possible actions, and the *state space* $\mathcal{S}$ is the set of possible states. Some of the algorithms assume that these sets are finite, but this is not required in general. The state evolves probabilistically based on the current state and action we take. The assumption that the next state depends only on the current state and action and not on any prior state or action is known as the *Markov assumption*.

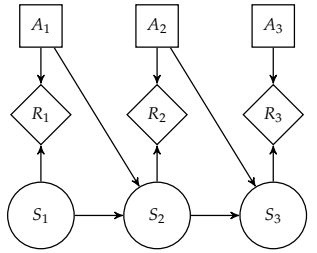


Figure 7.1. MDP decision network diagram.

An MDP can be represented using a decision network as shown in figure 7.1. There are informational edges (not shown here) from $A_{1:t-1}$ and $S_{1:t}$ to A_t. The utility function is decomposed into rewards $R_{1:t}$. We focus on *stationary* MDPs in which $P(S_{t+1} \mid S_t, A_t)$ and $P(R_t \mid S_t, A_t)$ do not vary with time. Stationary MDPs can be compactly represented by a dynamic decision diagram as shown in figure 7.2. The *state transition model* $T(s' \mid s, a)$ represents the probability of transitioning from state s to s' after executing action a. The *reward function* $R(s, a)$ represents the expected reward received when executing action a from state s.

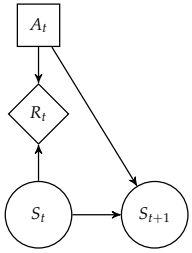


Figure 7.2. Stationary MDP decision network diagram. All MDPs have this general structure.

The reward function is a deterministic function of s and a because it represents an expectation, but rewards may be generated stochastically in the environment or even depend on the resulting next state.[2] Example 7.1 shows how to frame a collision avoidance problem as an MDP.

[2] For example, if the reward depends on the next state as given by $R(s, a, s')$, then the expected reward function would be

$$R(s, a) = \sum_{s'} T(s' \mid s, a) R(s, a, s')$$

The problem of aircraft collision avoidance can be formulated as an MDP. The states represent the positions and velocities of our aircraft and the intruder aircraft, and the actions represent whether we climb, descend, or stay level. We receive a large negative reward for colliding with the other aircraft and a small negative reward for climbing or descending.

Given knowledge of the current state, we must decide whether an avoidance maneuver is required. The problem is challenging because the positions of the aircraft evolve probabilistically, and we want to make sure that we start our maneuver early enough to avoid collision, but late enough so that we avoid unnecessary maneuvering.

Example 7.1. Aircraft collision avoidance framed as an MDP. Many other real-world applications are discussed in D. J. White, "A Survey of Applications of Markov Decision Processes," *Journal of the Operational Research Society*, vol. 44, no. 11, pp. 1073–1096, 1993.

```
struct MDP
    γ  # discount factor
    S  # state space
    A  # action space
    T  # transition function
    R  # reward function
    TR # sample transition and reward
end
```

Algorithm 7.1. Data structure for an MDP. We will use the `TR` field later to sample the next state and reward given the current state and action: `s', r = TR(s, a)`. In mathematical writing, MDPs are sometimes defined in terms of a tuple consisting of the various components of the MDP, written $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$.

The rewards in an MDP are treated as components in an additively decomposed utility function. In a *finite horizon* problem with n decisions, the utility associated with a sequence of rewards $r_{1:n}$ is simply

$$\sum_{t=1}^{n} r_t \tag{7.1}$$

The sum of rewards is sometimes called the *return*.

In an *infinite horizon* problem in which the number of decisions is unbounded, the sum of rewards can become infinite.[3] There are several ways to define utility in terms of individual rewards in infinite horizon problems. One way is to impose

[3] Suppose that strategy A results in a reward of 1 per time step and strategy B results in a reward of 100 per time step. Intuitively, a rational agent should prefer strategy B over strategy A, but both provide the same infinite expected utility.

a *discount factor* γ between 0 and 1. The utility is then given by

$$\sum_{t=1}^{\infty} \gamma^{t-1} r_t \tag{7.2}$$

This value is sometimes called the *discounted return*. So long as $0 \leq \gamma < 1$ and the rewards are finite, the utility will be finite. The discount factor makes it so that rewards in the present are worth more than rewards in the future, a concept that also appears in economics.

Another way to define utility in infinite horizon problems is to use the *average reward*, also called the *average return*, given by

$$\lim_{n \to \infty} \frac{1}{n} \sum_{t=1}^{n} r_t \tag{7.3}$$

This formulation can be attractive because we do not have to choose a discount factor, but there is often no practical difference between this formulation and a discounted return with a discount factor close to 1. Because the discounted return is often computationally simpler to work with, we will focus on the discounted formulation.

A *policy* tells us what action to select given the past history of states and actions. The action to select at time t, given the *history* $h_t = (s_{1:t}, a_{1:t-1})$, is written $\pi_t(h_t)$. Because the future states and rewards depend only on the current state and action (as made apparent in the conditional independence assumptions in figure 7.1), we can restrict our attention to policies that depend only on the current state. In addition, we will primarily focus on *deterministic policies* because there is guaranteed to exist in MDPs an optimal policy that is deterministic. Later chapters discuss *stochastic policies*, where $\pi_t(a_t \mid s_t)$ denotes the probability that the policy assigns to taking action a_t in state s_t at time t.

In infinite horizon problems with stationary transitions and rewards, we can further restrict our attention to *stationary policies*, which do not depend on time. We will write the action associated with stationary policy π in state s as $\pi(s)$, without the temporal subscript. In finite horizon problems, however, it may be beneficial to select a different action depending on how many time steps are remaining. For example, when playing basketball, it is generally not a good strategy to attempt a half-court shot unless there are only a couple of seconds remaining in the game. We can make stationary policies account for time by incorporating time as a state variable.

The expected utility of executing π from state s is denoted as $U^\pi(s)$. In the context of MDPs, U^π is often referred to as the *value function*. An *optimal policy* π^* is a policy that maximizes expected utility:[4]

$$\pi^*(s) = \arg\max_\pi U^\pi(s) \tag{7.4}$$

[4] Doing so is consistent with the maximum expected utility principle introduced in section 6.4.

for all states s. Depending on the model, there may be multiple policies that are optimal. The value function associated with an optimal policy π^* is called the *optimal value function* and is denoted as U^*.

An optimal policy can be found by using a computational technique called *dynamic programming*,[5] which involves simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner. Although we will focus on dynamic programming algorithms for MDPs, dynamic programming is a general technique that can be applied to a wide variety of other problems. For example, dynamic programming can be used in computing a Fibonacci sequence and finding the longest common subsequence between two strings.[6] In general, algorithms that use dynamic programming for solving MDPs are much more efficient than brute force methods.

[5] The term ''dynamic programming'' was coined by the American mathematician Richard Ernest Bellman (1920–1984). Dynamic refers to the fact that the problem is time-varying and programming refers to a methodology to find an optimal program or decision strategy. R. Bellman, *Eye of the Hurricane: An Autobiography*. World Scientific, 1984.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

7.2 Policy Evaluation

Before we discuss how to go about computing an optimal policy, we will discuss *policy evaluation*, where we compute the value function U^π. Policy evaluation can be done iteratively. If the policy is executed for a single step, the utility is $U_1^\pi(s) = R(s, \pi(s))$. Further steps can be obtained from the *lookahead* equation:

$$U_{k+1}^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_k^\pi(s') \tag{7.5}$$

This equation is implemented in algorithm 7.2. Iterative policy evaluation is implemented in algorithm 7.3. Several iterations are shown in figure 7.3.

The value function U^π can be computed to an arbitrary precision given sufficient iterations of the lookahead equation. Convergence is guaranteed because the update in equation (7.5) is a *contraction mapping* (reviewed in appendix A.15).[7] At convergence, the following equality holds:

[7] See exercise 7.12.

$$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s') \tag{7.6}$$

```
function lookahead(𝒫::MDP, U, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U(s′) for s′ in 𝒮)
end
function lookahead(𝒫::MDP, U::Vector, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U[i] for (i,s′) in enumerate(𝒮))
end
```

Algorithm 7.2. Functions for computing the lookahead state-action value from a state `s` given an action `a` using an estimate of the value function `U` for the MDP $\mathcal{P}$. The second version handles the case when `U` is a vector.

```
function iterative_policy_evaluation(𝒫::MDP, π, k_max)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    U = [0.0 for s in 𝒮]
    for k in 1:k_max
        U = [lookahead(𝒫, U, s, π(s)) for s in 𝒮]
    end
    return U
end
```

Algorithm 7.3. Iterative policy evaluation, which iteratively computes the value function for a policy `π` for MDP $\mathcal{P}$ with discrete state and action spaces using `k_max` iterations.

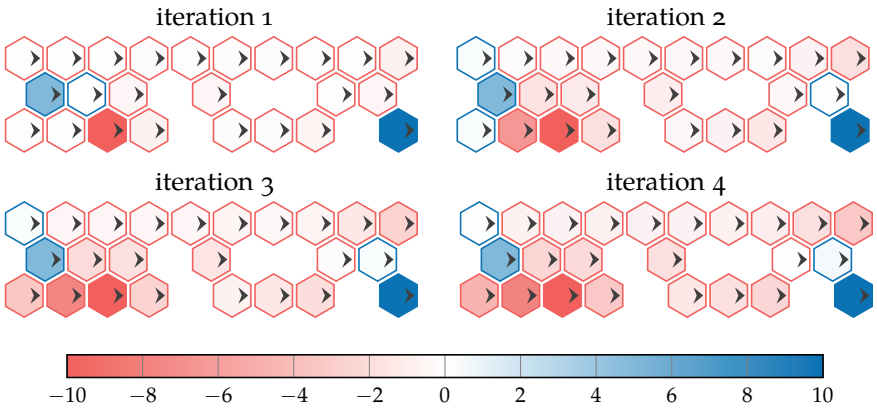


Figure 7.3. Iterative policy evaluation used to evaluate an east-moving policy on the hex world problem (see appendix F.1). The arrows indicate the direction recommended by the policy (i.e., always move east), and the colors indicate the values associated with the states. The values change with each iteration.

This equality is called the *Bellman expectation equation*.[8]

Policy evaluation can be done without iteration by solving the system of equations in the Bellman expectation equation directly. Equation (7.6) defines a set of $|S|$ linear equations with $|S|$ unknowns corresponding to the values at each state. One way to solve this system of equations is to first convert it into matrix form:

$$\mathbf{U}^\pi = \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}^\pi \tag{7.7}$$

where $\mathbf{U}^\pi$ and $\mathbf{R}^\pi$ are the utility and reward functions represented in vector form with $|\mathcal{S}|$ components. The $|\mathcal{S}| \times |\mathcal{S}|$ matrix $\mathbf{T}^\pi$ contains state transition probabilities where T_{ij}^π is the probability of transitioning from the ith state to the jth state.

The value function is obtained as follows:

$$\mathbf{U}^\pi - \gamma \mathbf{T}^\pi \mathbf{U}^\pi = \mathbf{R}^\pi \tag{7.8}$$

$$(\mathbf{I} - \gamma \mathbf{T}^\pi)\mathbf{U}^\pi = \mathbf{R}^\pi \tag{7.9}$$

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1}\mathbf{R}^\pi \tag{7.10}$$

This method is implemented in algorithm 7.4. Solving for $\mathbf{U}^\pi$ in this way requires $O(|\mathcal{S}|^3)$ time. The method is used to evaluate a policy in figure 7.4.

[8] This equation is named for Richard E. Bellman, one of the pioneers of dynamic programming. R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

```
function policy_evaluation(𝒫::MDP, π)
    𝒮, R, T, γ = 𝒫.𝒮, 𝒫.R, 𝒫.T, 𝒫.γ
    R′ = [R(s, π(s)) for s in 𝒮]
    T′ = [T(s, π(s), s′) for s in 𝒮, s′ in 𝒮]
    return (I - γ*T′)\R′
end
```

Algorithm 7.4. Exact policy evaluation, which computes the value function for a policy `π` for an MDP `𝒫` with discrete state and action spaces.

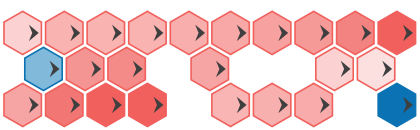


Figure 7.4. Exact policy evaluation used to evaluate an east-moving policy for the hex world problem. The exact solution contains lower values than what was contained in the first few steps of iterative policy evaluation in figure 7.3. If we ran iterative policy evaluation for more iterations, it would converge to the same value function.

7.3 Value Function Policies

The previous section showed how to compute a value function associated with a policy. This section shows how to extract a policy from a value function, which we later use when generating optimal policies. Given a value function U, which may or may not correspond to the optimal value function, we can construct a policy π that maximizes the lookahead equation introduced in equation (7.5):

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \tag{7.11}$$

We refer to this policy as a *greedy policy* with respect to U. If U is the optimal value function, then the extracted policy is optimal. Algorithm 7.5 implements this idea.

An alternative way to represent a policy is to use the *action value function*, sometimes called the *Q-function*. The action value function represents the expected return when starting in state s, taking action a, and then continuing with the greedy policy with respect to Q:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \tag{7.12}$$

From this action value function, we can obtain the value function,

$$U(s) = \max_a Q(s,a) \tag{7.13}$$

as well as the policy,

$$\pi(s) = \arg\max_a Q(s,a) \tag{7.14}$$

Storing Q explicitly for discrete problems requires $O(|\mathcal{S}| \times |\mathcal{A}|)$ storage instead of $O(|\mathcal{S}|)$ storage for U, but we do not have to use R and T to extract the policy.

Policies can also be represented using the *advantage function*, which quantifies the advantage of taking an action in comparison to the greedy action. It is defined in terms of the difference between Q and U:

$$A(s,a) = Q(s,a) - U(s) \tag{7.15}$$

Greedy actions have zero advantage, and nongreedy actions have negative advantage. Some algorithms that we will discuss later in the book use U representations, but others will use Q or A.

```
struct ValueFunctionPolicy
    𝒫 # problem
    U # utility function
end

function greedy(𝒫::MDP, U, s)
    u, a = findmax(a→lookahead(𝒫, U, s, a), 𝒫.𝒜)
    return (a=a, u=u)
end

(π::ValueFunctionPolicy)(s) = greedy(π.𝒫, π.U, s).a
```

Algorithm 7.5. A value function policy extracted from a value function `U` for an MDP `𝒫`. The `greedy` function will be used in other algorithms.

7.4 Policy Iteration

Policy iteration (algorithm 7.6) is one way to compute an optimal policy. It involves iterating between policy evaluation (section 7.2) and policy improvement through a greedy policy (algorithm 7.5). Policy iteration is guaranteed to converge given any initial policy. It converges in a finite number of iterations because there are finitely many policies and every iteration improves the policy if it can be improved. Although the number of possible policies is exponential in the number of states, policy iteration often converges quickly. Figure 7.5 demonstrates policy iteration on the hex world problem.

```
struct PolicyIteration
    π # initial policy
    k_max # maximum number of iterations
end

function solve(M::PolicyIteration, 𝒫::MDP)
    π, 𝒮 = M.π, 𝒫.𝒮
    for k = 1:M.k_max
        U = policy_evaluation(𝒫, π)
        π′ = ValueFunctionPolicy(𝒫, U)
        if all(π(s) == π′(s) for s in 𝒮)
            break
        end
        π = π′
    end
    return π
end
```

Algorithm 7.6. Policy iteration, which iteratively improves an initial policy `π` to obtain an optimal policy for an MDP `𝒫` with discrete state and action spaces.

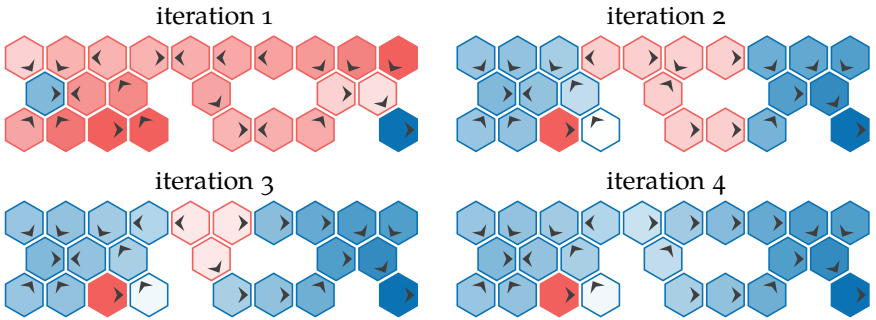


Figure 7.5. Policy iteration used to iteratively improve an initially east-moving policy in the hex world problem to obtain an optimal policy. In the first iteration, we see the value function associated with the east-moving policy and arrows indicating the policy that is greedy with respect to that value function. Policy iteration converges in four iterations; if we ran for a fifth or more iterations, we would get the same policy.

Policy iteration tends to be expensive because we must evaluate the policy in each iteration. A variation of policy iteration called *modified policy iteration*[9] approximates the value function using iterative policy evaluation instead of exact policy evaluation. We can choose the number of policy evaluation iterations between steps of policy improvement. If we use only one iteration between steps, then this approach is identical to value iteration.

[9] M. L. Puterman and M. C. Shin, "Modified Policy Iteration Algorithms for Discounted Markov Decision Problems," *Management Science*, vol. 24, no. 11, pp. 1127–1137, 1978.

7.5 Value Iteration

Value iteration is an alternative to policy iteration that is often used because of its simplicity. Unlike policy improvement, value iteration updates the value function directly. It begins with any bounded value function U, meaning that $|U(s)| < \infty$ for all s. One common initialization is $U(s) = 0$ for all s.

The value function can be improved by applying the *Bellman backup*, also called the *Bellman update*:[10]

$$U_{k+1}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \tag{7.16}$$

[10] It is referred to as a backup operation because it transfers information back to a state from its future states.

This backup procedure is implemented in algorithm 7.7.

```
function backup(𝒫::MDP, U, s)
    return maximum(lookahead(𝒫, U, s, a) for a in 𝒫.𝒜)
end
```

Algorithm 7.7. The backup procedure applied to an MDP `𝒫`, which improves a value function `U` at state `s`.

Repeated application of this update is guaranteed to converge to the optimal value function. Like iterative policy evaluation, we can use the fact that the update

is a contraction mapping to prove convergence.[11] This optimal policy is guaranteed to satisfy the *Bellman optimality equation*:

[11] See exercise 7.13.

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right) \tag{7.17}$$

Further applications of the Bellman backup once this equality holds do not change the value function. An optimal policy can be extracted from U^* using equation (7.11). Value iteration is implemented in algorithm 7.8 and is applied to the hex world problem in figure 7.6.

The implementation in algorithm 7.8 stops after a fixed number of iterations, but it is also common to terminate the iterations early based on the maximum change in value $\|U_{k+1} - U_k\|_\infty$, called the *Bellman residual*. If the Bellman residual drops below a threshold δ, then the iterations terminate. A Bellman residual of δ guarantees that the optimal value function obtained by value iteration is within $\epsilon = \delta\gamma/(1-\gamma)$ of U^*.[12] Discount factors closer to 1 significantly inflate this error, leading to slower convergence. If we heavily discount future reward (γ closer to 0), then we do not need to iterate as much into the future. This effect is demonstrated in example 7.2.

[12] See exercise 7.8.

Knowing the maximum deviation of the estimated value function from the optimal value function, $\|U_k - U^*\|_\infty < \epsilon$, allows us to bound the maximum deviation of reward obtained under the extracted policy π from an optimal policy π^*. This *policy loss* $\|U^\pi - U^*\|_\infty$ is bounded by $2\epsilon\gamma/(1-\gamma)$.[13]

[13] S. P. Singh and R. C. Yee, "An Upper Bound on the Loss from Approximate Optimal-Value Functions," *Machine Learning*, vol. 16, no. 3, pp. 227–233, 1994.

```
struct ValueIteration
    k_max # maximum number of iterations
end

function solve(M::ValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        U = [backup(𝒫, U, s) for s in 𝒫.𝒮]
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

Algorithm 7.8. Value iteration, which iteratively improves a value function `U` to obtain an optimal policy for an MDP `𝒫` with discrete state and action spaces. The method terminates after `k_max` iterations.

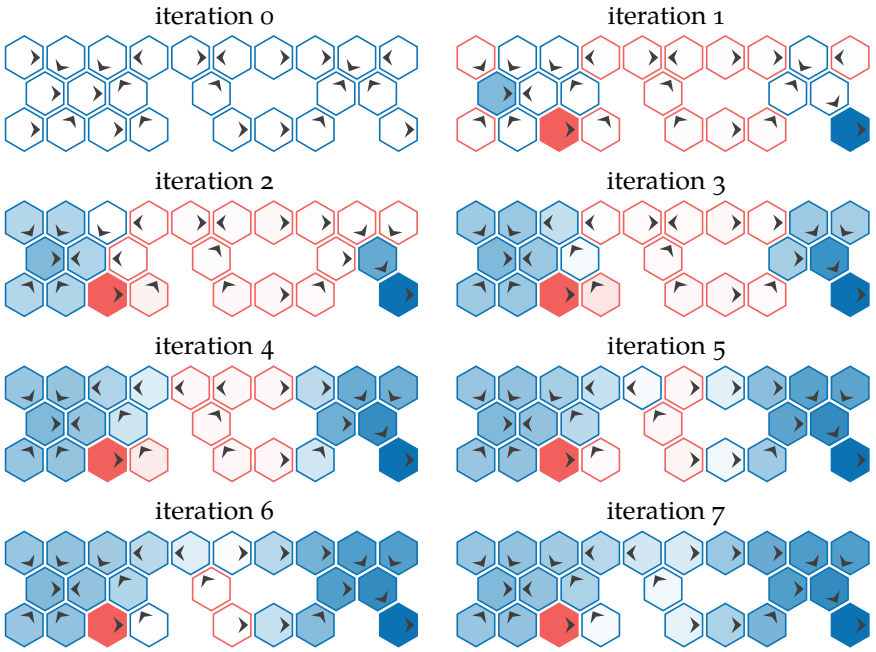
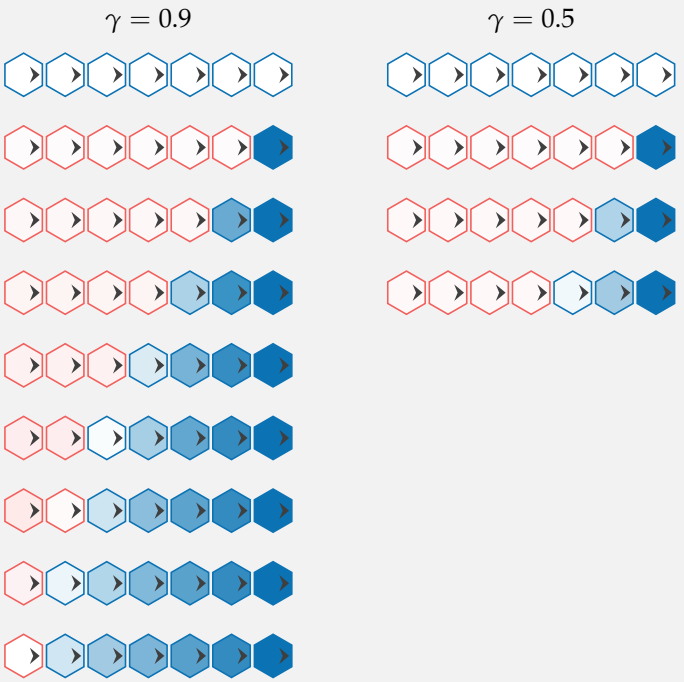


Figure 7.6. Value iteration in the hex world problem to obtain an optimal policy. Each hex is colored according to the value function, and arrows indicate the policy that is greedy with respect to that value function.

Example 7.2. The effect of the discount factor on convergence of value iteration. In each case, value iteration was run until the Bellman residual was less than 1.

Consider a simple variation of the hex world problem, consisting of a straight line of tiles with a single consuming tile at the end producing a reward of 10. The discount factor directly affects the rate at which reward from the consuming tile propagates down the line to the other tiles, and thus how quickly value iteration converges.



7.6 Asynchronous Value Iteration

Value iteration tends to be computationally intensive, as every entry in the value function U_k is updated in each iteration to obtain U_{k+1}. In *asynchronous value iteration*, only a subset of the states are updated with each iteration. Asynchronous value iteration is still guaranteed to converge on the optimal value function, provided that each state is updated an infinite number of times.

One common asynchronous value iteration method, *Gauss-Seidel value iteration* (algorithm 7.9), sweeps through an ordering of the states and applies the Bellman update in place:

$$U(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \tag{7.18}$$

The computational savings lies in not having to construct a second value function in memory with each iteration. Gauss-Seidel value iteration can converge more quickly than standard value iteration, depending on the ordering chosen.[14] In some problems, the state contains a time index that increments deterministically forward in time. If we apply Gauss-Seidel value iteration starting at the last time index and work our way backward, this process is sometimes called *backward induction value iteration*. An example of the impact of the state ordering is given in example 7.3.

[14] A poor ordering in Gauss-Seidel value iteration cannot cause the algorithm to be slower than standard value iteration.

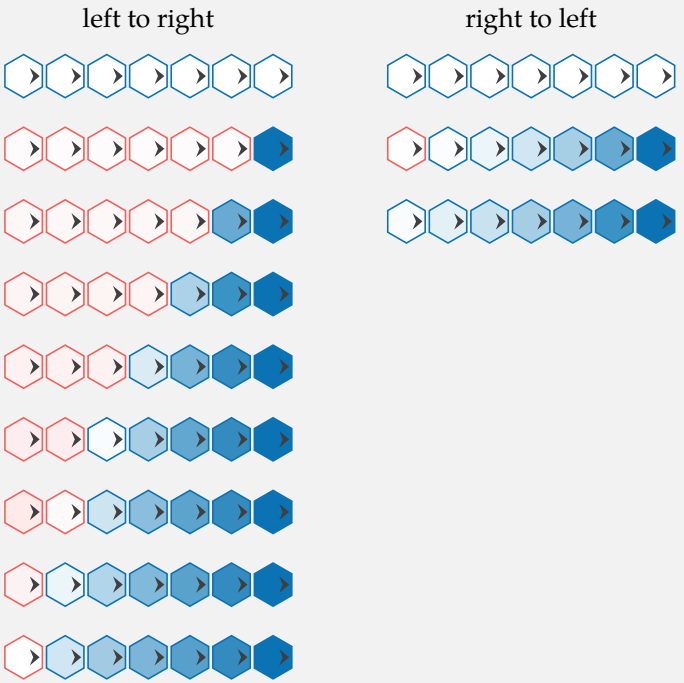
```
struct GaussSeidelValueIteration
    k_max # maximum number of iterations
end

function solve(M::GaussSeidelValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        for (i, s) in enumerate(𝒫.𝒮)
            U[i] = backup(𝒫, U, s)
        end
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

Algorithm 7.9. Asynchronous value iteration, which updates states in a different manner than value iteration, often saving computation time. The method terminates after `k_max` iterations.

Example 7.3. The effect of the state ordering on convergence of asynchronous value iteration. In this case, evaluating right to left allows convergence to occur in far fewer iterations.

Consider the linear variation of the hex world problem from example 7.2. We can solve the same problem using asynchronous value iteration. The ordering of the states directly affects the rate at which reward from the consuming tile propagates down the line to the other tiles, and thus how quickly the method converges.



7.7 Linear Program Formulation

The problem of finding an optimal policy can be formulated as a *linear program*, which is an optimization problem with a linear objective function and a set of linear equality or inequality constraints. Once a problem is represented as a linear program, we can use one of many linear programming solvers.[15]

To show how we can convert the Bellman optimality equation into a linear program, we begin by replacing the equality in the Bellman optimality equation with a set of inequality constraints while minimizing $U(s)$ at each state s:[16]

$$\begin{aligned} &\text{minimize} \sum_s U(s) \\ &\text{subject to } U(s) \geq \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \text{ for all } s \end{aligned} \tag{7.19}$$

The variables in the optimization are the utilities at each state. Once we know those utilities, we can extract an optimal policy using equation (7.11).

The maximization in the inequality constraints can be replaced by a set of linear constraints, making it a linear program:

$$\begin{aligned} &\text{minimize} \sum_s U(s) \\ &\text{subject to } U(s) \geq R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \text{ for all } s \text{ and } a \end{aligned} \tag{7.20}$$

In the linear program shown in equation (7.20), the number of variables is equal to the number of states and the number of constraints is equal to the number of states times the number of actions. Because linear programs can be solved in polynomial time,[17] MDPs can be solved in polynomial time as well. Although a linear programming approach provides this asymptotic complexity guarantee, it is often more efficient in practice to simply use value iteration. Algorithm 7.10 provides an implementation of this.

7.8 Linear Systems with Quadratic Reward

So far, we have assumed discrete state and action spaces. This section relaxes this assumption, allowing for continuous, vector-valued states and actions. The Bellman optimality equation for discrete problems can be modified as follows:[18]

[15] For an overview of linear programming, see R. Vanderbei, *Linear Programming, Foundations and Extensions*, 4th ed. Springer, 2014.

[16] Intuitively, we want to push the value $U(s)$ at all states s down in order to convert the inequality constraints into equality constraints. Hence, we minimize the sum of all utilities.

[17] This was proved by L. G. Khachiyan, "Polynomial Algorithms in Linear Programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53–72, 1980. Modern algorithms tend to be more efficient in practice.

[18] This section assumes that the problem is undiscounted and finite horizon, but these equations can be easily generalized.

```
struct LinearProgramFormulation end

function tensorform(𝒫::MDP)
    𝒮, 𝒜, R, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T
    𝒮′ = eachindex(𝒮)
    𝒜′ = eachindex(𝒜)
    R′ = [R(s,a) for s in 𝒮, a in 𝒜]
    T′ = [T(s,a,s′) for s in 𝒮, a in 𝒜, s′ in 𝒮]
    return 𝒮′, 𝒜′, R′, T′
end

solve(𝒫::MDP) = solve(LinearProgramFormulation(), 𝒫)

function solve(M::LinearProgramFormulation, 𝒫::MDP)
    𝒮, 𝒜, R, T = tensorform(𝒫)
    model = Model(GLPK.Optimizer)
    @variable(model, U[𝒮])
    @objective(model, Min, sum(U))
    @constraint(model, [s=𝒮,a=𝒜], U[s] ≥ R[s,a] + 𝒫.γ*T[s,a,:]⋅U)
    optimize!(model)
    return ValueFunctionPolicy(𝒫, value.(U))
end
```

Algorithm 7.10. A method for solving a discrete MDP using a linear program formulation. For convenience in specifying the linear program, we define a function for converting an MDP into its tensor form, where the states and actions consist of integer indices, the reward function is a matrix, and the transition function is a three-dimensional tensor. It uses the `JuMP.jl` package for mathematical programming. The optimizer is set to use `GLPK.jl`, but others can be used instead. We also define the default solve behavior for MDPs to use this formulation.

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}} \left(R(\mathbf{s}, \mathbf{a}) + \int T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) U_h(\mathbf{s}') \, \mathrm{d}\mathbf{s}' \right) \tag{7.21}$$

where s and a in equation (7.16) are replaced with their vector equivalents, the summation is replaced with an integral, and T provides a probability density rather than a probability mass. Computing equation (7.21) is not straightforward for an arbitrary continuous transition distribution and reward function.

In some cases, exact solution methods do exist for MDPs with continuous state and action spaces.[19] In particular, if a problem has *linear dynamics* and has *quadratic reward*, then the optimal policy can be efficiently found in closed form. Such a system is known in control theory as a *linear quadratic regulator* (*LQR*) and has been well studied.[20]

A problem has linear dynamics if the next state $\mathbf{s}'$ after taking action $\mathbf{a}$ from state $\mathbf{s}$ is determined by an equation of the form:

$$\mathbf{s}' = \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w} \tag{7.22}$$

where $\mathbf{T}_s$ and $\mathbf{T}_a$ are matrices and $\mathbf{w}$ is a random disturbance drawn from a zero mean, finite variance distribution that does not depend on $\mathbf{s}$ and $\mathbf{a}$. One common choice is the multivariate Gaussian.

[19] For a detailed overview, see chapter 4 of volume I of D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 2007.

[20] For a compact summary of LQR and other related control problems, see A. Shaiju and I. R. Petersen, "Formulas for Discrete Time LQR, LQG, LEQG and Minimax LQG Optimal Control Problems," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 8773–8778, 2008.

A reward function is quadratic if it can be written in the form:[21]

$$R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} \tag{7.23}$$

[21] A third term, $2\mathbf{s}^\top \mathbf{R}_{sa}\mathbf{a}$, can also be included. For an example, see Shaiju and Petersen (2008).

where $\mathbf{R}_s$ and $\mathbf{R}_a$ are matrices that determine how state and action component combinations contribute reward. We additionally require that $\mathbf{R}_s$ be negative semidefinite and $\mathbf{R}_a$ be negative definite. Such a reward function penalizes states and actions that deviate from $\mathbf{0}$.

Problems with linear dynamics and quadratic reward are common in control theory where one often seeks to regulate a process such that it does not deviate far from a desired value. The quadratic cost assigns a much higher cost to states far from the origin than to those near it. The optimal policy for a problem with linear dynamics and quadratic reward has an analytic, closed-form solution. Many MDPs can be approximated with linear quadratic MDPs and solved, often yielding reasonable policies for the original problem.

Substituting the transition and reward functions into equation (7.21) produces

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int p(\mathbf{w}) U_h(\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w}) \, \mathrm{d}\mathbf{w} \right) \tag{7.24}$$

where $p(\mathbf{w})$ is the probability density of the random, zero-mean disturbance $\mathbf{w}$.

The optimal one-step value function is

$$U_1(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} \right) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} \tag{7.25}$$

for which the optimal action is $\mathbf{a} = \mathbf{0}$.

We will show through induction that $U_h(\mathbf{s})$ has a quadratic form, $\mathbf{s}^\top \mathbf{V}_h \mathbf{s} + q_h$, with symmetric matrices $\mathbf{V}_h$. For the one-step value function, $\mathbf{V}_1 = \mathbf{R}_s$ and $q_1 = 0$. Substituting this quadratic form into equation (7.24) yields

$$U_{h+1}(\mathbf{s}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int p(\mathbf{w}) \left((\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w})^\top \mathbf{V}_h (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w}) + q_h \right) \mathrm{d}\mathbf{w} \right) \tag{7.26}$$

This can be simplified by expanding and using the fact that $\int p(\mathbf{w}) \, \mathrm{d}\mathbf{w} = 1$ and $\int \mathbf{w} p(\mathbf{w}) \, \mathrm{d}\mathbf{w} = 0$:

$$\begin{aligned} U_{h+1}(\mathbf{s}) = {} & \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} \\ & + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + 2\mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} + \mathbf{a}^\top \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} \right) \\ & + \int p(\mathbf{w}) \left(\mathbf{w}^\top \mathbf{V}_h \mathbf{w} \right) \mathrm{d}\mathbf{w} + q_h \end{aligned} \tag{7.27}$$

We can obtain the optimal action by differentiating with respect to $\mathbf{a}$ and setting it to $\mathbf{0}$:[22]

$$\begin{aligned}\mathbf{0} &= \left(\mathbf{R}_a + \mathbf{R}_a^\top\right)\mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} + \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a + \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^\top\right)\mathbf{a} \\ &= 2\mathbf{R}_a\mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a}\end{aligned} \tag{7.28}$$

[22] Recall that

$$\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$$
$$\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$$

Solving for the optimal action yields[23]

$$\mathbf{a} = -\left(\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^{-1} \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} \tag{7.29}$$

[23] The matrix $\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a$ is negative definite, and thus invertible.

Substituting the optimal action into $U_{h+1}(\mathbf{s})$ yields the quadratic form that we were seeking, $U_{h+1}(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}_{h+1} \mathbf{s} + q_{h+1}$, with[24]

$$\mathbf{V}_{h+1} = \mathbf{R}_s + \mathbf{T}_s^\top \mathbf{V}_h^\top \mathbf{T}_s - \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s\right)^\top \left(\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^{-1} \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s\right) \tag{7.30}$$

and

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] \tag{7.31}$$

If $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, then

$$q_{h+1} = \sum_{i=1}^{h} \operatorname{Tr}(\boldsymbol{\Sigma} \mathbf{V}_i) \tag{7.32}$$

We can compute $\mathbf{V}_h$ and q_h up to any horizon h starting from $\mathbf{V}_1 = \mathbf{R}_s$ and $q_1 = 0$ and iterating using equations (7.30) and (7.31). The optimal action for an h-step policy comes directly from equation (7.29):

$$\pi_h(\mathbf{s}) = -\left(\mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_a + \mathbf{R}_a\right)^{-1} \mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_s \mathbf{s} \tag{7.33}$$

[24] This equation is sometimes referred to as the *discrete-time Riccati equation*, named after the Venetian mathematician Jacopo Riccati (1676–1754).

Note that the optimal action is independent of the zero-mean disturbance distribution.[25] The variance of the disturbance, however, does affect the expected utility. Algorithm 7.11 provides an implementation. Example 7.4 demonstrates this process on a simple problem with linear Gaussian dynamics.

[25] In this case, we can replace the random disturbances with its expected value without changing the optimal policy. This property is known as *certainty equivalence*.

7.9 Summary

- Discrete MDPs with bounded rewards can be solved exactly through dynamic programming.

```
struct LinearQuadraticProblem
    Ts # transition matrix with respect to state
    Ta # transition matrix with respect to action
    Rs # reward matrix with respect to state (negative semidefinite)
    Ra # reward matrix with respect to action (negative definite)
    h_max # horizon
end

function solve(𝒫::LinearQuadraticProblem)
    Ts, Ta, Rs, Ra, h_max = 𝒫.Ts, 𝒫.Ta, 𝒫.Rs, 𝒫.Ra, 𝒫.h_max
    V = zeros(size(Rs))
    πs = Any[s -> zeros(size(Ta, 2))]
    for h in 2:h_max
        V = Ts'*(V - V*Ta*((Ta'*V*Ta + Ra) \ Ta'*V))*Ts + Rs
        L = -(Ta'*V*Ta + Ra) \ Ta' * V * Ts
        push!(πs, s -> L*s)
    end
    return πs
end
```

Algorithm 7.11. A method that computes an optimal policy for an `h_max`-step horizon MDP with stochastic linear dynamics parameterized by matrices `Ts` and `Ta` and quadratic reward parameterized by matrices `Rs` and `Ra`. The method returns a vector of policies where entry h produces the optimal first action in an h-step policy.

- Policy evaluation for such problems can be done exactly through matrix inversion or can be approximated by an iterative algorithm.
- Policy iteration can be used to solve for optimal policies by iterating between policy evaluation and policy improvement.
- Value iteration and asynchronous value iteration save computation by directly iterating the value function.
- The problem of finding an optimal policy can be framed as a linear program and solved in polynomial time.
- Continuous problems with linear transition functions and quadratic rewards can be solved exactly.

7.10 Exercises

Exercise 7.1. Show that for an infinite sequence of constant rewards ($r_t = r$ for all t), the infinite horizon discounted return converges to $r/(1-\gamma)$.

Example 7.4. Solving a finite horizon MDP with a linear transition function and quadratic reward. The illustration shows the progression of the system from $[-10, 0]$. The blue contour lines show the Gaussian distributions over the state at each iteration. The initial belief is circular, but it gets distorted to a noncircular shape as we propagate the belief forward using the Kalman filter.

Consider a continuous MDP where the state is composed of a scalar position and velocity $s = [x, v]$. Actions are scalar accelerations a that are each executed over a time step $\Delta t = 1$. Find an optimal five-step policy from $s_0 = [-10, 0]$, given a quadratic reward:

$$R(\mathbf{s}, a) = -x^2 - v^2 - 0.5a^2$$

such that the system tends toward rest at $s = \mathbf{0}$.

The transition dynamics are

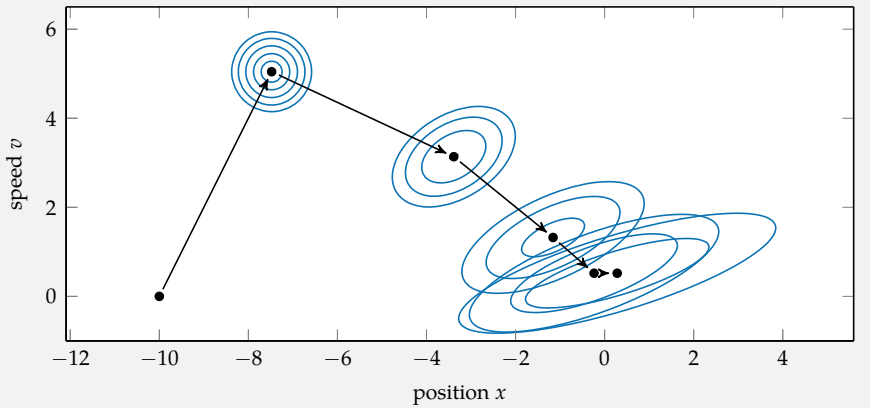
$$\begin{bmatrix} x' \\ v' \end{bmatrix} = \begin{bmatrix} x + v\Delta t + \frac{1}{2}a\Delta t^2 + w_1 \\ v + a\Delta t + w_2 \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0.5\Delta t^2 \\ \Delta t \end{bmatrix} [a] + \mathbf{w}$$

where $\mathbf{w}$ is drawn from a zero-mean multivariate Gaussian distribution with covariance $0.1\mathbf{I}$.

The reward matrices are $\mathbf{R}_s = -\mathbf{I}$ and $\mathbf{R}_a = -[0.5]$.

The resulting optimal policies are:

$$\begin{aligned}
\pi_1(\mathbf{s}) &= \begin{bmatrix} 0 & 0 \end{bmatrix} \mathbf{s} \\
\pi_2(\mathbf{s}) &= \begin{bmatrix} -0.286 & -0.857 \end{bmatrix} \mathbf{s} \\
\pi_3(\mathbf{s}) &= \begin{bmatrix} -0.462 & -1.077 \end{bmatrix} \mathbf{s} \\
\pi_4(\mathbf{s}) &= \begin{bmatrix} -0.499 & -1.118 \end{bmatrix} \mathbf{s} \\
\pi_5(\mathbf{s}) &= \begin{bmatrix} -0.504 & -1.124 \end{bmatrix} \mathbf{s}
\end{aligned}$$



Solution: We can prove that the infinite sequence of discounted constant rewards converges to $r/(1-\gamma)$ in the following steps:

$$\begin{aligned}\sum_{t=1}^{\infty} \gamma^{t-1} r_t &= r + \gamma^1 r + \gamma^2 r + \cdots \\ &= r + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t\end{aligned}$$

We can move the summation to the left side and factor out $(1-\gamma)$:

$$\begin{aligned}(1-\gamma) \sum_{t=1}^{\infty} \gamma^{t-1} r &= r \\ \sum_{t=1}^{\infty} \gamma^{t-1} r &= \frac{r}{1-\gamma}\end{aligned}$$

Exercise 7.2. Suppose we have an MDP consisting of five states, $s_{1:5}$, and two actions, to stay (a_S) and continue (a_C). We have the following:

$$\begin{aligned}T(s_i \mid s_i, a_S) &= 1 \text{ for } i \in \{1,2,3,4\} \\ T(s_{i+1} \mid s_i, a_C) &= 1 \text{ for } i \in \{1,2,3,4\} \\ T(s_5 \mid s_5, a) &= 1 \text{ for all actions } a \\ R(s_i, a) &= 0 \text{ for } i \in \{1,2,3,5\} \text{ and for all actions } a \\ R(s_4, a_S) &= 0 \\ R(s_4, a_C) &= 10\end{aligned}$$

What is the discount factor γ if the optimal value $U^*(s_1) = 1$?

Solution: The optimal value of $U^*(s_1)$ is associated with following the optimal policy π^* starting from s_1. Given the transition model, the optimal policy from s_1 is to continue until reaching s_5, which is a terminal state where we can no longer transition to another state or accumulate additional reward. Thus, the optimal value of s_1 can be computed as

$$\begin{aligned}U^*(s_1) &= \sum_{t=1}^{\infty} \gamma^{t-1} r_t \\ U^*(s_1) &= R(s_1, a_C) + \gamma^1 R(s_2, a_C) + \gamma^2 R(s_3, a_C) + \gamma^3 R(s_4, a_C) + \gamma^4 R(s_5, a_C) + \cdots \\ U^*(s_1) &= 0 + \gamma^1 \times 0 + \gamma^2 \times 0 + \gamma^3 \times 10 + \gamma^4 \times 0 + 0 \\ 1 &= 10\gamma^3\end{aligned}$$

Thus, the discount factor is $\gamma = 0.1^{1/3} \approx 0.464$.

Exercise 7.3. What is the time complexity of performing k steps of iterative policy evaluation?

Solution: Iterative policy evaluation requires computing the lookahead equation:

$$U^{\pi}_{k+1}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^{\pi}_{k}(s')$$

Updating the value at a single state requires summing over all $|\mathcal{S}|$ states. For a single iteration over all states, we must do this operation $|\mathcal{S}|$ times. Thus, the time complexity of k steps of iterative policy evaluation is $O(k|\mathcal{S}|^2)$.

Exercise 7.4. Suppose that we have an MDP with six states, $s_{1:6}$, and four actions, $a_{1:4}$. Using the following tabular form of the action value function $Q(s, a)$, compute $U(s)$, $\pi(s)$, and $A(s, a)$.

$Q(s,a)$	a_1	a_2	a_3	a_4
s_1	0.41	0.46	0.37	0.37
s_2	0.50	0.55	0.46	0.37
s_3	0.60	0.50	0.38	0.44
s_4	0.41	0.50	0.33	0.41
s_5	0.50	0.60	0.41	0.39
s_6	0.71	0.70	0.61	0.59

Solution: We can compute $U(s)$, $\pi(s)$, and $A(s, a)$ using the following equations:

$$U(s) = \max_a Q(s, a) \qquad \pi(s) = \arg\max_a Q(s, a) \qquad A(s, a) = Q(s, a) - U(s)$$

s	$U(s)$	$\pi(s)$	$A(s,a_1)$	$A(s,a_2)$	$A(s,a_3)$	$A(s,a_4)$
s_1	0.46	a_2	−0.05	0.00	−0.09	−0.09
s_2	0.55	a_2	−0.05	0.00	−0.09	−0.18
s_3	0.60	a_1	0.00	−0.10	−0.22	−0.16
s_4	0.50	a_2	−0.09	0.00	−0.17	−0.09
s_5	0.60	a_2	−0.10	0.00	−0.19	−0.21
s_6	0.71	a_1	0.00	−0.01	−0.10	−0.12

Exercise 7.5. Suppose that we have a three-tile, straight-line hex world (appendix F.1) where the rightmost tile is an absorbing state. When we take any action in the rightmost state, we get a reward of 10 and we are transported to a fourth terminal state where we no longer receive any reward. Use a discount factor of $\gamma = 0.9$, and perform a single step of policy iteration where the initial policy π has us move east in the first tile, northeast in the second tile, and southwest in the third tile. For the policy evaluation step, write out the transition matrix $\mathbf{T}^{\pi}$ and the reward vector $\mathbf{R}^{\pi}$, and then solve the infinite horizon value function $\mathbf{U}^{\pi}$ directly using matrix inversion. For the policy improvement step, compute the updated policy π' by maximizing the lookahead equation.

Solution: For the policy evaluation step, we use equation (7.10), repeated here:

$$\mathbf{U}^{\pi} = (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R}^{\pi}$$

Forming the transition matrix $\mathbf{T}^\pi$ and reward vector $\mathbf{R}^\pi$ with an additional state for the terminal state, we can solve for the infinite horizon value function $\mathbf{U}^\pi$:[26]

$$\mathbf{U}^\pi = \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - (0.9) \begin{bmatrix} 0.3 & 0.7 & 0 & 0 \\ 0 & 0.85 & 0.15 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -0.3 \\ -0.85 \\ 10 \\ 0 \end{bmatrix} \approx \begin{bmatrix} 1.425 \\ 2.128 \\ 10 \\ 0 \end{bmatrix}$$

For the policy improvement step, we apply equation (7.11) using the updated value function. The actions in the arg max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}:

$$\pi(s_1) = \arg\max(1.425, 0.527, 0.283, 0.283, 0.283, 0.527) = a_E$$
$$\pi(s_2) = \arg\max(6.575, 2.128, 0.970, 1.172, 0.970, 2.128) = a_E$$
$$\pi(s_3) = \arg\max(10, 10, 10, 10, 10, 10) \text{ (all actions are equally desirable)}$$

Exercise 7.6. Perform two steps of value iteration to the problem in exercise 7.5, starting with an initial value function $U_0(s) = 0$ for all s.

Solution: We need to use the Bellman backup (equation (7.16)) to iteratively update the value function. The actions in the max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}. For our first iteration, the value function is zero for all states, so we only need to consider the reward component:

$$U_1(s_1) = \max(-0.3, -0.85, -1, -1, -1, -0.85) = -0.3$$
$$U_1(s_2) = \max(-0.3, -0.85, -0.85, -0.3, -0.85, -0.85) = -0.3$$
$$U_1(s_3) = \max(10, 10, 10, 10, 10, 10) = 10$$

For the second iteration,

$$U_2(s_1) = \max(-0.57, -1.12, -1.27, -1.27, -1.27, -1.12) = -0.57$$
$$U_2(s_2) = \max(5.919, 0.271, -1.12, -0.57, -1.12, 0.271) = 5.919$$
$$U_2(s_3) = \max(10, 10, 10, 10, 10, 10) = 10$$

Exercise 7.7. Apply one sweep of asynchronous value iteration to the problem in exercise 7.5, starting with an initial value function $U_0(s) = 0$ for all s. Update the states from right to left.

[26] The hex world problem defines $R(s, a, s')$, so in order to produce entries for $\mathbf{R}^\pi$, we must compute

$$R(s, a) = \sum_{s'} T(s' \mid s, a) R(s, a, s')$$

For example, -0.3 comes from the 30 % chance that moving east causes a collision with the border, with cost -1.

Solution: We use the Bellman backup (equation (7.16)) to iteratively update the value function over each state following our ordering. The actions in the max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}:

$$\begin{aligned}
U(s_3) &= \max(10, 10, 10, 10, 10, 10) = 10 \\
U(s_2) &= \max(6, 0.5, -0.85, -0.3, -0.85, 0.5) = 6 \\
U(s_1) &= \max(3.48, -0.04, -1, -1, -1, -0.04) = 3.48
\end{aligned}$$

Exercise 7.8. Prove that a Bellman residual of δ guarantees that the value function obtained by value iteration is within $\delta\gamma/(1-\gamma)$ of $U^*(s)$ at every state s.

Solution: For a given U_k, suppose we know that $\|U_k - U_{k-1}\|_\infty < \delta$. Then we bound the improvement in the next iteration:

$$\begin{aligned}
U_{k+1}(s) - U_k(s) &= \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\quad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_{k-1}(s') \right) \\
&< \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\quad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \left(U_k(s') - \delta \right) \right) \\
&= \delta\gamma
\end{aligned}$$

Similarly,

$$\begin{aligned}
U_{k+1}(s) - U_k(s) &> \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\quad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \left(U_k(s') + \delta \right) \right) \\
&= -\delta\gamma
\end{aligned}$$

The accumulated improvement after infinite iterations is thus bounded by

$$\|U^*(s) - U_k(s)\|_\infty < \sum_{i=1}^{\infty} \delta\gamma^i = \frac{\delta\gamma}{1-\gamma}$$

A Bellman residual of δ thus guarantees that the optimal value function obtained by value iteration is within $\delta\gamma/(1-\gamma)$ of U^*.

Exercise 7.9. Suppose that we run policy evaluation on an expert policy to obtain a value function. If acting greedily with respect to that value function is equivalent to the expert policy, what can we deduce about the expert policy?

Solution: We know from the Bellman optimality equation that greedy lookahead on an optimal value function is stationary. If the greedy policy matches the expert policy, then both policies are optimal.

Exercise 7.10. Show how an LQR problem with a quadratic reward function $R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a}$ can be reformulated so that the reward function includes linear terms in $\mathbf{s}$ and $\mathbf{a}$.

Solution: We can introduce an additional state dimension that is always equal to 1, yielding a new system with linear dynamics:

$$\begin{bmatrix} \mathbf{s}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{T}_s & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} + \mathbf{T}_a \mathbf{a}$$

The reward function of the augmented system can now have linear state reward terms:

$$\begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix}^\top \mathbf{R}_{\text{augmented}} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + 2\mathbf{r}_{s,\text{linear}}^\top \mathbf{s} + r_{s,\text{scalar}}$$

Similarly, we can include an additional action dimension that is always 1 in order to obtain linear action reward terms.

Exercise 7.11. Why does the optimal policy obtained in example 7.4 produce actions with greater magnitude when the horizon is greater?

Solution: The problem in example 7.4 has quadratic reward that penalizes deviations from the origin. The longer the horizon, the greater the negative reward that can be accumulated, making it more worthwhile to reach the origin sooner.

Exercise 7.12. Prove that iterative policy evaluation converges to the solution of equation (7.6).

Solution: Consider iterative policy evaluation applied to a policy π as given in equation (7.5):

$$U_{k+1}^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_k^\pi(s')$$

Let us define an operator B_π and rewrite this as $U_{k+1}^\pi = B_\pi U_k^\pi$. We can show that B_π is a contraction mapping:

$$\begin{aligned}
B_\pi U^\pi(s) &= R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s') \\
&= R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) \left(U^\pi(s') - \hat{U}^\pi(s') + \hat{U}^\pi(s')\right) \\
&= B_\pi \hat{U}^\pi(s) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) \left(U^\pi(s') - \hat{U}^\pi(s')\right) \\
&\leq B_\pi \hat{U}^\pi(s) + \gamma \|U^\pi - \hat{U}^\pi\|_\infty
\end{aligned}$$

Hence, $\|B_\pi U^\pi - B_\pi \hat{U}^\pi\|_\infty \leq \alpha \|U^\pi - \hat{U}^\pi\|_\infty$ for $\alpha = \gamma$, implying that B_π is a contraction mapping. As discussed in appendix A.15, $\lim_{t \to \infty} B_\pi^t U_1^\pi$ converges to a unique fixed point U^π, for which $U^\pi = B_\pi U^\pi$.

Exercise 7.13. Prove that value iteration converges to a unique solution.

Solution: The value iteration update (equation (7.16)) is

$$U^{k+1}(s) = \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) U_k(s') \right)$$

We will denote the Bellman operator as B and rewrite an application of the Bellman backup as $U_{k+1} = BU_k$. As with the previous problem, if B is a contraction mapping, then repeated application of B to U will converge to a unique fixed point.

We can show that B is a contraction mapping:

$$\begin{aligned}
BU(s) &= \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) U(s') \right) \\
&= \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) \left(U(s') - \hat{U}(s') + \hat{U}(s')\right) \right) \\
&\leq B\hat{U}(s) + \gamma \max_a \sum_{s'} T(s' \mid s, a) \left(U(s') - \hat{U}(s')\right) \\
&\leq B\hat{U}(s) + \alpha \|U - \hat{U}\|_\infty
\end{aligned}$$

for $\alpha = \gamma \max_s \max_a \sum_{s'} T(s' \mid s, a)$, with $0 \leq \alpha < 1$. Hence, $\|BU - B\hat{U}\|_\infty \leq \alpha \|U - \hat{U}\|_\infty$, which implies that B is a contraction mapping.

Exercise 7.14. Show that the point to which value iteration converges corresponds to the optimal value function.

Solution: Let U be the value function produced by value iteration. We want to show that $U = U^*$. At convergence, we have $BU = U$. Let U_0 be a value function that maps all states to 0. For any policy π, it follows from the definition of B_π that $B_\pi U_0 \leq BU_0$. Similarly, $B_\pi^t U_0 \leq B^t U_0$. Because $B_{\pi^*}^t U_0 \to U^*$ and $B^t U_0 \to U$ as $t \to \infty$, it follows that $U^* \leq U$, which can be the case only if $U = U^*$.

Exercise 7.15. Suppose that we have a linear Gaussian problem with disturbance $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ and quadratic reward. Show that the scalar term in the utility function has the form:

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] = \sum_{i=1}^{h} \text{Tr}(\mathbf{\Sigma}\mathbf{V}_i)$$

You may want to use the *trace trick*:

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} = \text{Tr}\left(\mathbf{x}^\top \mathbf{A}\mathbf{x}\right) = \text{Tr}\left(\mathbf{A}\mathbf{x}\mathbf{x}^\top\right)$$

Solution: This equation is true if $\mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] = \text{Tr}(\mathbf{\Sigma}\mathbf{V}_i)$. Our derivation is

$$\begin{aligned}
\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] &= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\text{Tr}\left(\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right)\right] \\
&= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\text{Tr}\left(\mathbf{V}_i \mathbf{w}\mathbf{w}^\top\right)\right] \\
&= \text{Tr}\left(\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{V}_i \mathbf{w}\mathbf{w}^\top\right]\right) \\
&= \text{Tr}\left(\mathbf{V}_i \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{w}\mathbf{w}^\top\right]\right) \\
&= \text{Tr}(\mathbf{V}_i\mathbf{\Sigma}) \\
&= \text{Tr}(\mathbf{\Sigma}\mathbf{V}_i)
\end{aligned}$$

Exercise 7.16. What is the role of the scalar term q in the LQR optimal value function, as given in equation (7.31)?

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right]$$

Solution: A matrix $\mathbf{M}$ is positive definite if, for all nonzero $\mathbf{x}$, $\mathbf{x}^\top \mathbf{M}\mathbf{x} > 0$. In equation (7.31), every $\mathbf{V}_i$ is negative semidefinite, so $\mathbf{w}^\top \mathbf{V}\mathbf{w} \leq 0$ for all $\mathbf{w}$. Thus, these q terms are guaranteed to be nonpositive. This should be expected, as it is impossible to obtain positive reward in LQR problems, and we seek instead to minimize cost.

The q scalars are offsets in the quadratic optimal value function:

$$U(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}\mathbf{s} + q$$

Each q represents the baseline reward around which the $\mathbf{s}^\top \mathbf{V}\mathbf{s}$ term fluctuates. We know that $\mathbf{V}$ is negative definite, so $\mathbf{s}^\top \mathbf{V}\mathbf{s} \leq 0$, and q thus represents the expected reward that one could obtain if one were at the origin, $\mathbf{s} = \mathbf{0}$.

8 Approximate Value Functions

Up to this point, we have assumed that the value function can be represented as a table. Tables are useful representations only for small, discrete problems. Problems with larger state spaces may require an infeasible amount of memory, and the exact methods discussed in the previous chapter may require an infeasible amount of computation. For such problems, we often have to resort to *approximate dynamic programming*, where the solution may not be exact.[1] One way to approximate solutions is to use *value function approximation*, which is the subject of this chapter. We will discuss different approaches to approximating the value function and how to incorporate dynamic programming to derive approximately optimal policies.

[1] A deeper treatment of this topic is provided by W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. Wiley, 2011. Relevant insights can be drawn from a variety of fields as discussed by W. B. Powell, *Reinforcement Learning and Stochastic Optimization*. Wiley, 2022.

8.1 Parametric Representations

We will use $U_\theta(s)$ to denote our *parametric representation* of the value function, where θ is the vector of *parameters*. There are many ways to represent $U_\theta(s)$, several of which will be mentioned later in this chapter. Assuming that we have such an approximation, we can extract an action according to

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_\theta(s') \right) \tag{8.1}$$

Value function approximations are often used in problems with continuous state spaces, in which case the summation above may be replaced with an integral. The integral can be approximated using transition model samples.

An alternative to the computation in equation (8.1) is to approximate the action value function $Q(s,a)$. If we use $Q_\theta(s,a)$ to represent our parametric

approximation, we can obtain an action according to

$$\pi(s) = \arg\max_a Q_\theta(s, a) \tag{8.2}$$

This chapter discusses how we can apply dynamic programming at a finite set of states $S = s_{1:m}$ to arrive at a parametric approximation of the value function over the full state space. Different schemes can be used to generate this set. If the state space is relatively low-dimensional, we can define a grid. Another approach is to use random sampling from the state space. However, some states are more likely to be encountered than others and are therefore more important in constructing the value function. We can bias the sampling toward more important states by running simulations with some policy (perhaps initially random), from a plausible set of initial states.

An iterative approach can be used to enhance our approximation of the value function at the states in S. We alternate between improving our value estimates at S through dynamic programming and refitting our approximation at those states. Algorithm 8.1 provides an implementation where the dynamic programming step consists of Bellman backups as done in value iteration (see section 7.5). A similar algorithm can be created for action value approximations Q_θ.[2]

[2] Several other categories of approaches for optimizing value function approximations are surveyed by A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, and J. P. How, "A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning," *Foundations and Trends in Machine Learning*, vol. 6, no. 4, pp. 375–451, 2013.

```
struct ApproximateValueIteration
    U0    # initial parameterized value function that supports fit!
    S     # set of discrete states for performing backups
    k_max # maximum number of iterations
end

function solve(M::ApproximateValueIteration, 𝒫::MDP)
    U0, S, k_max = M.U0, M.S, M.k_max
    for k in 1:k_max
        U = [backup(𝒫, U0, s) for s in S]
        fit!(U0, S, U)
    end
    return ValueFunctionPolicy(𝒫, U0)
end
```

Algorithm 8.1. Approximate value iteration for an MDP with the parameterized value function approximation `U0`. We perform backups (defined in algorithm 7.7) at the states in `S` to obtain a vector of utilities `U`. We then call `fit!(U0, S, U)`, which modifies the parametric representation `U0` to better match the value of the states in `S` to the utilities in `U`. Different parametric approximations have different implementations for `fit!`.

All of the parametric representations discussed in this chapter can be used with algorithm 8.1. To be used with that algorithm, a representation needs to support the evaluation of U_θ and the fitting of U_θ to estimates of the utilities at the points in S.

We can group the parametric representations into two categories. The first category includes *local approximation* methods, where θ corresponds to the values at the states in S. To evaluate $U_\theta(s)$ at an arbitrary state s, we take a weighted sum of the values stored in S. The second category includes *global approximation* methods, where θ is not directly related to the values at the states in S. In fact, θ may have far fewer or even far more components than there are states in S.

Both local approximation and many global approximations can be viewed as a *linear function approximation* $U_\theta(s) = \theta^\top \beta(s)$, where methods differ in how they define the vector function β. In local approximation methods, $\beta(s)$ determines how to weight the utilities of the states in S to approximate the utility at state s. The weights are generally nonnegative and sum to 1. In many global approximation methods, $\beta(s)$ is viewed as a set of basis functions that are combined in a linear fashion to obtain an approximation for an arbitrary s.

We can also approximate the action value function using a linear function, $Q_\theta(s, a) = \theta^\top \beta(s, a)$. In the context of local approximations, we can provide approximations over continuous action spaces by choosing a finite set of actions $A \subset \mathcal{A}$. Our parameter vector θ would then consist of $|S| \times |A|$ components, each corresponding to a state-action value. Our function $\beta(s, a)$ would return a vector with the same number of components that specifies how to weight together our finite set of state-action values to obtain an estimate of the utility associated with state s and action a.

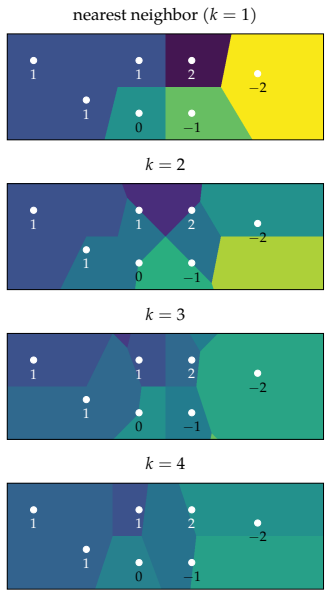


Figure 8.1. Approximating the values of states in a two-dimensional, continuous state space using the mean of the utility values of their k-nearest neighbors according to Euclidean distance. The resulting value function is piecewise constant.

8.2 Nearest Neighbor

A simple approach to local approximation is to use the value of the state in S that is the *nearest neighbor* of s. In order to use this approach, we need a *distance metric* (see appendix A.3). We use $d(s, s')$ to denote the distance between two states s and s'. The approximate value function is then $U_\theta(s) = \theta_i$, where $i = \arg\min_{j \in 1:m} d(s_j, s)$. Figure 8.1 shows an example of a value function represented using the nearest neighbor scheme.

We can generalize this approach to average together the values of the *k-nearest neighbors*. This approach still results in piecewise constant value functions, but different values for k can result in better approximations. Figure 8.1 shows examples of value functions approximated with different values for k. Algorithm 8.2 provides an implementation of this.

```
mutable struct NearestNeighborValueFunction
    k # number of neighbors
    d # distance function d(s, s')
    S # set of discrete states
    θ # vector of values at states in S
end

function (U0::NearestNeighborValueFunction)(s)
    dists = [U0.d(s,s') for s' in U0.S]
    ind = sortperm(dists)[1:U0.k]
    return mean(U0.θ[i] for i in ind)
end

function fit!(U0::NearestNeighborValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.2. The k-nearest neighbors method, which approximates the value of a state `s` based on the `k` closest states in `S`, as determined by a distance function `d`. The vector `θ` contains the values of the states in `S`. Greater efficiency can be achieved by using specialized data structures, such as kd-trees, implemented in `NearestNeighbors.jl`.

8.3 Kernel Smoothing

Another local approximation method is *kernel smoothing*, where the utilities of the states in S are smoothed over the entire state space. This method requires defining a *kernel function* $k(s, s')$ that relates pairs of states s and s'. We generally want $k(s, s')$ to be higher for states that are closer together because those values tell us how to weight together the utilities associated with the states in S. This method results in the following linear approximation:

$$U_\theta(s) = \sum_{i=1}^{m} \theta_i \beta_i(s) = \theta^\top \beta(s) \tag{8.3}$$

where

$$\beta_i(s) = \frac{k(s, s_i)}{\sum_{j=1}^{m} k(s, s_j)} \tag{8.4}$$

Algorithm 8.3 provides an implementation of this.

There are many ways that we can define a kernel function. We can define our kernel to simply be the inverse of the distance between states:

$$k(s, s') = \max(d(s, s'), \epsilon)^{-1} \tag{8.5}$$

where ϵ is a small positive constant in order to avoid dividing by zero when $s = s'$. Figure 8.2 shows value approximations using several distance functions. As we can see, kernel smoothing can result in smooth value function approximations, in contrast with k-nearest neighbors. Figure 8.3 applies this kernel to a discrete hex world problem and shows the outcome of a few iterations of approximate value iteration (algorithm 8.1). Figure 8.4 shows a value function and policy learned for the mountain car problem (appendix F.4) with a continuous state space.

Another common kernel is the *Gaussian kernel*:

$$k(s, s') = \exp\left(-\frac{d(s, s')^2}{2\sigma^2}\right) \tag{8.6}$$

where σ controls the degree of smoothing.

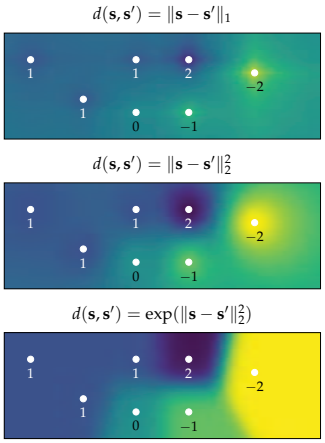


Figure 8.2. Approximating the values of states in a two-dimensional continuous state space by assigning values based on proximity to several states with known values. Approximations are constructed using several distance functions.

```
mutable struct LocallyWeightedValueFunction
    k # kernel function k(s, s′)
    S # set of discrete states
    θ # vector of values at states in S
end

function (U0::LocallyWeightedValueFunction)(s)
    w = normalize([U0.k(s,s′) for s′ in U0.S], 1)
    return U0.θ ⋅ w
end

function fit!(U0::LocallyWeightedValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.3. Locally weighted value function approximation defined by a kernel function `k` and a vector of utilities `θ` at states in `S`.

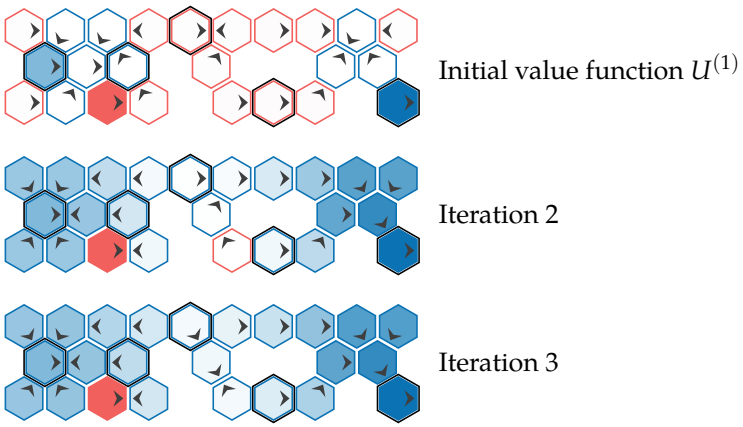


Figure 8.3. Local approximation value iteration used to iteratively improve an approximate value function on the hex world problem. The five outlined states are used to approximate the value function. The value of the remaining states are approximated using the distance function $\|\mathbf{s} - \mathbf{s}'\|_2^2$. The resulting policy is reasonable but nevertheless suboptimal. Positive reward is shown in blue, and negative reward is shown in red.

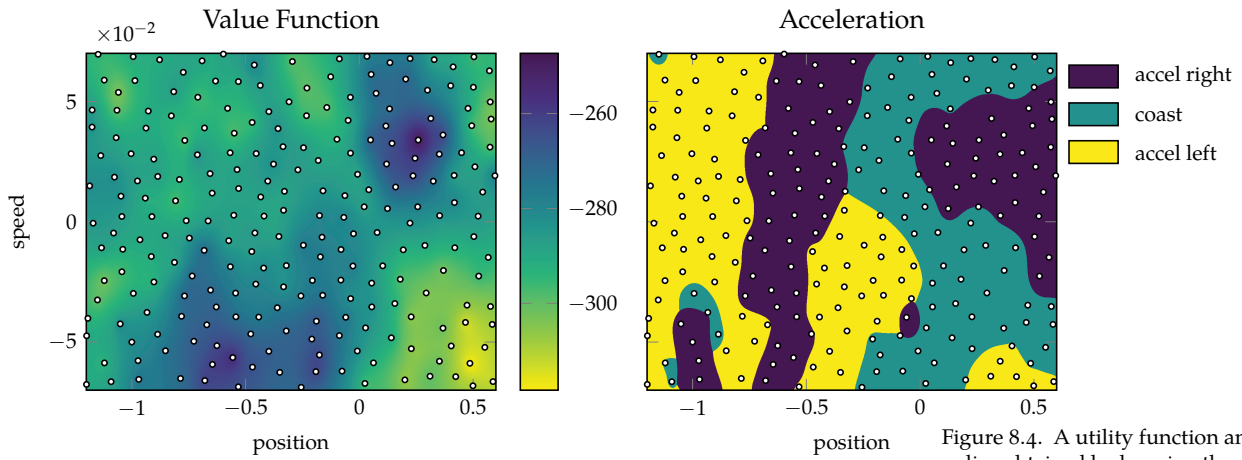


Figure 8.4. A utility function and policy obtained by learning the action values for a finite set of states (white) in the mountain car problem using the distance function $\|\mathbf{s} - \mathbf{s}'\|_2 + 0.1$.

8.4 Linear Interpolation

Linear interpolation is another common approach to local approximation. The one-dimensional case is straightforward, in which the approximated value for a state s between two states s_1 and s_2 is

$$U_\theta(s) = \alpha\theta_1 + (1-\alpha)\theta_2 \tag{8.7}$$

with $\alpha = (s_2 - s)/(s_2 - s_1)$. This case is shown in figures 8.5 and 8.6.

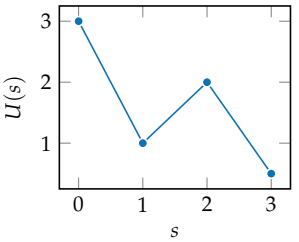


Figure 8.5. One-dimensional linear interpolation produces interpolated values along the line segment connecting two points.

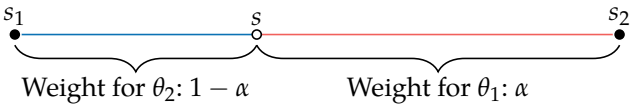
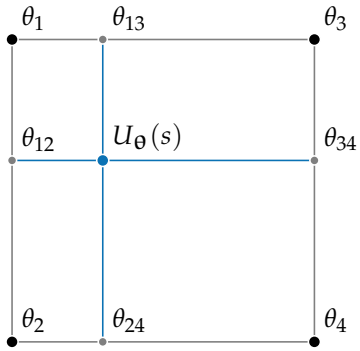


Figure 8.6. The weight assigned to each point in one dimension is proportional to the length of the segment on the opposite side of the interpolation state.

Linear interpolation can be extended to a multidimensional grid. In the two-dimensional case, called *bilinear interpolation*, we interpolate among four vertices. Bilinear interpolation is done through single-dimensional linear interpolation, once in each axis, requiring the utility of four states at the grid vertices. This interpolation is shown in figure 8.7.



θ_{12} = 1D interpolation between θ_1 and θ_2 along the vertical axis
θ_{24} = 1D interpolation between θ_2 and θ_4 along the horizontal axis
θ_{13} = 1D interpolation between θ_1 and θ_3 along the horizontal axis
θ_{34} = 1D interpolation between θ_3 and θ_4 along the vertical axis

$$U_\theta(s) = \begin{cases} \text{1D interpolation between } \theta_{12} \text{ and } \theta_{34} \text{ along the horizontal axis} \\ \text{or} \\ \text{1D interpolation between } \theta_{13} \text{ and } \theta_{24} \text{ along the vertical axis} \end{cases}$$

Figure 8.7. Linear interpolation on a two-dimensional grid is achieved through linear interpolation on each axis in turn, in either order.

Given four vertices with coordinates $s_1 = [x_1, y_1]$, $s_2 = [x_1, y_2]$, $s_3 = [x_2, y_1]$, and $s_4 = [x_2, y_2]$, and a sample state $s = [x, y]$, the interpolated value is

$$
\begin{aligned}
U_\theta(s) &= \alpha\theta_{12} + (1-\alpha)\theta_{34} && (8.8)\\
&= \frac{x_2 - x}{x_2 - x_1}\theta_{12} + \frac{x - x_1}{x_2 - x_1}\theta_{34} && (8.9)\\
&= \frac{x_2 - x}{x_2 - x_1}(\alpha\theta_1 + (1-\alpha)\theta_2) + \frac{x - x_1}{x_2 - x_1}(\alpha\theta_3 + (1-\alpha)\theta_4) && (8.10)\\
&= \frac{x_2 - x}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_1 + \frac{y - y_1}{y_2 - y_1}\theta_2\right) + \frac{x - x_1}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_3 + \frac{y - y_1}{y_2 - y_1}\theta_4\right) && (8.11)\\
&= \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4 && (8.12)
\end{aligned}
$$

The resulting interpolation weighs each vertex according to the area of its opposing quadrant, as shown in figure 8.8.

Multilinear interpolation in d dimensions is similarly achieved by linearly interpolating along each axis, requiring 2^d vertices. Here too, the utility of each vertex is weighted according to the volume of the opposing hyperrectangle. Multilinear interpolation is implemented in algorithm 8.4. Figure 8.9 demonstrates this approach on a two-dimensional state space.

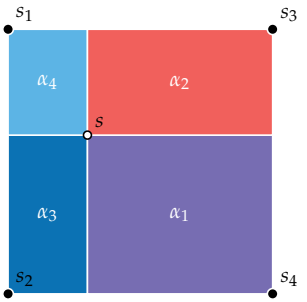


Figure 8.8. Linear interpolation on a two-dimensional grid results in a contribution of each vertex equal to the relative area of its opposing quadrant: $U_\theta(s) = \alpha_1\theta_1 + \alpha_2\theta_2 + \alpha_3\theta_3 + \alpha_4\theta_4$.

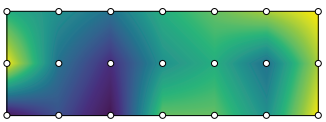


Figure 8.9. Two-dimensional linear interpolation over a 3×7 grid.

8.5 Simplex Interpolation

Multilinear interpolation can be inefficient in high dimensions. Rather than weighting the contributions of 2^d points, *simplex interpolation* considers only $d + 1$ points in the neighborhood of a given state to produce a continuous surface that matches the known sample points.

We start with a multidimensional grid and divide each cell into $d!$ *simplexes*, which are multidimensional generalizations of triangles defined by the *convex hull* of $d + 1$ vertices. This process is known as *Coxeter-Freudenthal-Kuhn triangulation*,[3] and it ensures that any two simplexes that share a face will produce equivalent values across the face, thus producing continuity when interpolating, as shown in figure 8.10.

[3] A. W. Moore, "Simplicial Mesh Generation with Applications," Ph.D. dissertation, Cornell University, 1992.

```
mutable struct MultilinearValueFunction
    o # position of lower-left corner
    δ # vector of widths
    θ # vector of values at states in S
end

function (U0::MultilinearValueFunction)(s)
    o, δ, θ = U0.o, U0.δ, U0.θ
    Δ = (s - o)./δ
    # Multidimensional index of lower-left cell
    i = min.(floor.(Int, Δ) .+ 1, size(θ) .- 1)
    vertex_index = similar(i)
    d = length(s)
    u = 0.0
    for vertex in 0:2^d-1
        weight = 1.0
        for j in 1:d
            # Check whether jth bit is set
            if vertex & (1 << (j-1)) > 0
                vertex_index[j] = i[j] + 1
                weight *= Δ[j] - i[j] + 1
            else
                vertex_index[j] = i[j]
                weight *= i[j] - Δ[j]
            end
        end
        u += θ[vertex_index...]*weight
    end
    return u
end

function fit!(U0::MultilinearValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.4. A method for conducting multilinear interpolation to estimate the value of state vector `s` for known state values `θ` over a grid defined by a lower-left vertex `o` and vector of widths `δ`. Vertices of the grid can all be written `o + δ.*i` for some nonnegative integral vector `i`. The package `Interpolations.jl` also provides multilinear and other interpolation methods.

To illustrate, suppose that we have translated and scaled the cell containing a state such that the lowest vertex is $\mathbf{0}$ and the diagonally opposite vertex is $\mathbf{1}$. There is a simplex for each permutation of $1:d$. The simplex given by permutation $\mathbf{p}$ is the set of points $\mathbf{x}$ satisfying

$$0 \leq x_{p_1} \leq x_{p_2} \leq \cdots \leq x_{p_d} \leq 1 \tag{8.13}$$

Figure 8.11 shows the simplexes obtained for the unit cube.

Simplex interpolation first translates and scales a state vector $\mathbf{s}$ to the unit hypercube of its corresponding cell to obtain $\mathbf{s}'$. It then sorts the entries in $\mathbf{s}'$ to determine which simplex contains $\mathbf{s}'$. The utility at $\mathbf{s}'$ can then be expressed by a unique linear combination of the vertices of that simplex.

Example 8.1 provides an example of simplex interpolation. The process is implemented in algorithm 8.5.

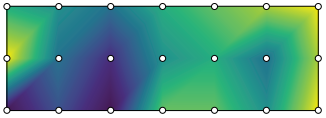


Figure 8.10. Two-dimensional simplex interpolation over a 3×7 grid.

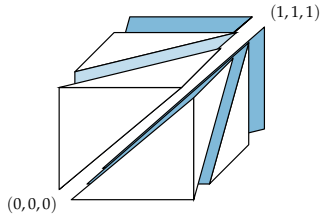


Figure 8.11. A triangulation of a unit cube. Based on figure 2.1 of A. W. Moore, "Simplicial Mesh Generation with Applications," Ph.D. dissertation, Cornell University, 1992.

Example 8.1. Simplex interpolation in three dimensions.

Consider a three-dimensional simplex given by the permutation $\mathbf{p} = [3, 1, 2]$ such that points within the simplex satisfy $0 \leq x_3 \leq x_1 \leq x_2 \leq 1$. This simplex has vertices $(0, 0, 0)$, $(0, 1, 0)$, $(1, 1, 0)$, and $(1, 1, 1)$.

Any point $\mathbf{s}$ belonging to the simplex can thus be expressed by a weighting of the vertices:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

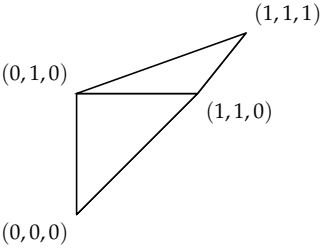
We can determine the values of the last three weights in succession:

$$w_4 = s_3 \qquad w_3 = s_1 - w_4 \qquad w_2 = s_2 - w_3 - w_4$$

We obtain w_1 by enforcing that the weights sum to 1.

If $\mathbf{s} = [0.3, 0.7, 0.2]$, then the weights are

$$w_4 = 0.2 \qquad w_3 = 0.1 \qquad w_2 = 0.4 \qquad w_1 = 0.3$$



```
mutable struct SimplexValueFunction
    o # position of lower-left corner
    δ # vector of widths
    θ # vector of values at states in S
end

function (U0::SimplexValueFunction)(s)
    Δ = (s - U0.o)./U0.δ
    # Multidimensional index of upper-right cell
    i = min.(floor.(Int, Δ) .+ 1, size(U0.θ) .- 1) .+ 1
    u = 0.0
    s′ = (s - (U0.o + U0.δ.*(i.-2))) ./ U0.δ
    p = sortperm(s′) # increasing order
    w_tot = 0.0
    for j in p
        w = s′[j] - w_tot
        u += w*U0.θ[i...]
        i[j] -= 1
        w_tot += w
    end
    u += (1 - w_tot)*U0.θ[i...]
    return u
end

function fit!(U0::SimplexValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.5. A method for conducting simplex interpolation to estimate the value of state vector `s` for known state values `θ` over a grid defined by a lower-left vertex `o` and a vector of widths `δ`. Vertices of the grid can all be written `o + δ.*i` for some nonnegative integral vector `i`. Simplex interpolation is also implemented in the general `GridInterpolations.jl` package.

8.6 Linear Regression

A simple global approximation approach is *linear regression*, where $U_\theta(s)$ is a linear combination of *basis functions*, also commonly referred to as *features*. These basis functions are generally a nonlinear function of the state s and are combined into a vector function $\beta(s)$ or $\beta(s, a)$, resulting in the approximations

$$U_\theta(s) = \theta^\top \beta(s) \qquad Q_\theta(s, a) = \theta^\top \beta(s, a) \tag{8.14}$$

Although our approximation is linear with respect to the basis functions, the resulting approximation may be nonlinear with respect to the underlying state variables. Figure 8.12 illustrates this concept. Example 8.2 provides an example of global linear value approximation using polynomial basis functions for the continuous mountain car problem, resulting in a nonlinear value function approximation with respect to the state variables.

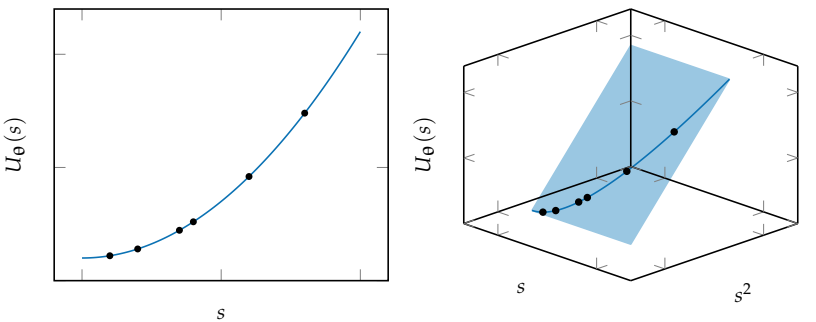


Figure 8.12. Linear regression with nonlinear basis functions is linear in higher dimensions. Here, polynomial regression can be seen as linear in a three-dimensional space. The function exists in the plane formed from its bases, but it does not occupy the entire plane because the terms are not independent.

Adding more basis functions generally improves the ability to match the target utilities at the states in S, but too many basis functions can lead to poor approximations at other states. Principled methods exist for choosing an appropriate set of basis functions for our regression model.[4]

Fitting linear models involves determining the vector θ that minimizes the squared error of the predictions at the states in $S = s_{1:m}$. If the utilities associated with those states are denoted as $u_{1:m}$, then we want to find the θ that minimizes

$$\sum_{i=1}^{m} (\hat{U}_\theta(s_i) - u_i)^2 = \sum_{i=1}^{m} (\theta^\top \beta(s_i) - u_i)^2 \tag{8.15}$$

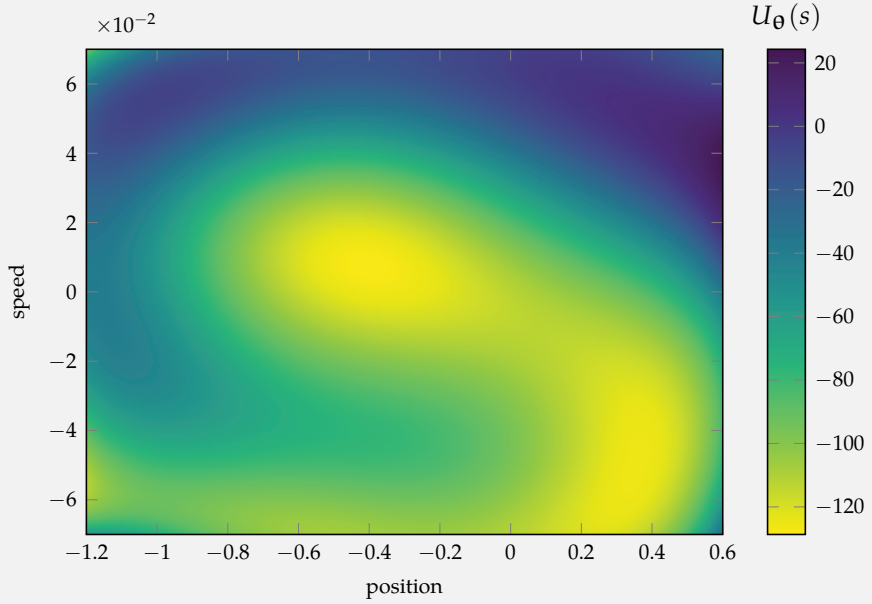
[4] See chapter 14 of M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019. or chapter 7 of T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

Example 8.2. Using a linear approximation to the mountain car value function. The choice of basis functions makes a big difference. The optimal value function for the mountain car is nonlinear, with a spiral shape and discontinuities. Even sixth-degree polynomials do not produce a perfect fit.

We can approximate the value function for the mountain car problem using a linear approximation. The problem has a continuous state space with two dimensions consisting of position x and speed v. Here are the basis functions up to degree six:

$$\boldsymbol{\beta}(s) = \begin{bmatrix} 1, & & & & & & \\ x, & v, & & & & & \\ x^2, & xv, & v^2, & & & & \\ x^3, & x^2v, & xv^2, & v^3, & & & \\ x^4, & x^3v, & x^2v^2, & xv^3, & v^4, & & \\ x^5, & x^4v, & x^3v^2, & x^2v^3, & xv^4, & v^5, & \\ x^6, & x^5v, & x^4v^2, & x^3v^3, & x^2v^4, & xv^5, & v^6 \end{bmatrix}$$

Here is a plot of an approximate value function fit to state-value pairs from an expert policy:



The optimal θ can be computed through some simple matrix operations. We first construct a matrix $\mathbf{X}$ where each of the m rows $\mathbf{X}_{i,:}$ contains $\beta(s_i)^\top$.[5] It can be shown that the value of θ that minimizes the squared error is

$$\theta = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top u_{1:m} = \mathbf{X}^+ u_{1:m} \tag{8.16}$$

where $\mathbf{X}^+$ is the *Moore-Penrose pseudoinverse* of matrix $\mathbf{X}$. The pseudoinverse is often implemented by first computing the *singular value decomposition*, $\mathbf{X} = \mathbf{U}\Sigma\mathbf{U}^*$. We then have

$$\mathbf{X}^+ = \mathbf{U}\Sigma^+\mathbf{U}^* \tag{8.17}$$

The pseudoinverse of the diagonal matrix Σ is obtained by taking the reciprocal of each nonzero element of the diagonal and then transposing the result.

Figure 8.13 shows how the utilities of states in S are fit with several basis function families. Different choices of basis functions result in different errors.

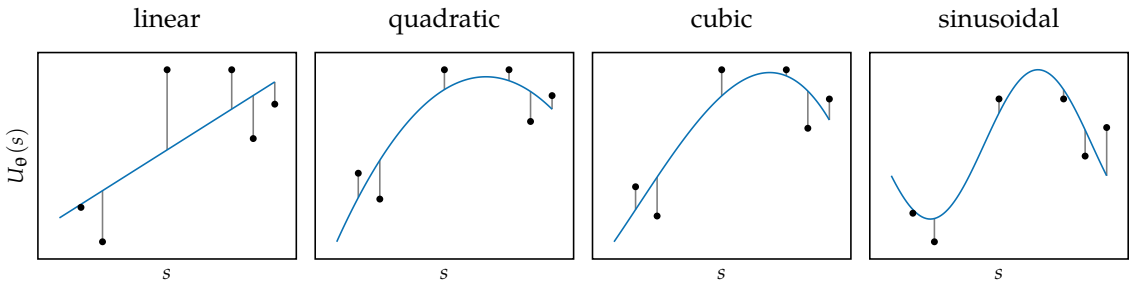


Figure 8.13. Linear regression with different basis function families.

Algorithm 8.6 provides an implementation for evaluating and fitting linear regression models of the value function. Example 8.3 demonstrates this approach with the mountain car problem.

8.7 Neural Network Regression

Neural network regression relieves us of having to construct an appropriate set of basis functions as required in linear regression. Instead, a *neural network* is used to represent our value function. For a review of neural networks, see appendix D. The input to the neural network would be the state variables, and the output would be the utility estimate. The parameters θ would correspond to the *weights* in the neural network.

[5] For an overview of the mathematics involved in linear regression as well as more advanced techniques, see T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

```
mutable struct LinearRegressionValueFunction
    β # basis vector function
    θ # vector of parameters
end

function (U0::LinearRegressionValueFunction)(s)
    return U0.β(s) ⋅ U0.θ
end

function fit!(U0::LinearRegressionValueFunction, S, U)
    X = hcat([U0.β(s) for s in S]...)'
    U0.θ = pinv(X)*U
    return U0
end
```

Algorithm 8.6. Linear regression value function approximation, defined by a basis vector function `β` and a vector of parameters `θ`. The function `pinv` implements the psuedoinverse. Julia and other languages support the *backslash* operator, which allows us to write `X \ U` in place of `pinv(X)*U` in the `fit!` function.

As discussed in appendix D, we can optimize the network weights to achieve a particular objective. In the context of approximate dynamic programming, we would want to minimize the error of our predictions, just as we did in the previous section. However, minimizing the squared error cannot be done through simple matrix operations. Instead, we generally have to rely on optimization techniques such as gradient descent. Fortunately, computing the gradient of neural networks can be done exactly through straightforward application of the derivative chain rule.

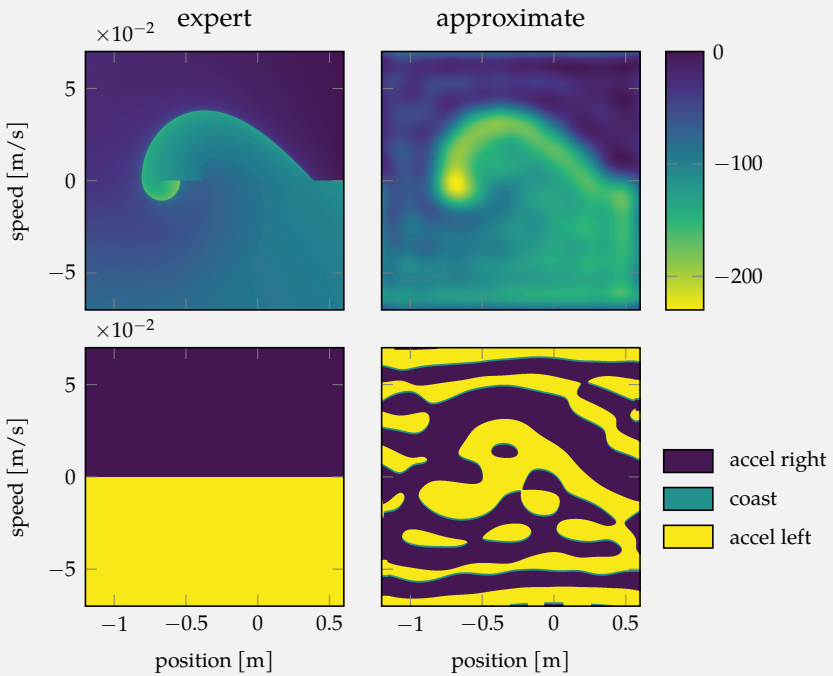
8.8 Summary

- For large or continuous problems, we can attempt to find approximate policies represented by parameterized models of the value function.
- The approaches taken in this chapter involve iteratively applying steps of dynamic programming at a finite set of states and refining our parametric approximation.
- Local approximation techniques approximate the value function based on the values of nearby states with known values.
- A variety of local approximation techniques include nearest neighbor, kernel smoothing, linear interpolation, and simplex interpolation.
- Global approximation techniques include linear regression and neural network regression.

We can apply linear regression to learn a value function for the mountain car problem. The optimal value function has the form of a spiral, which can be difficult to approximate with polynomial basis functions (see example 8.2). We use Fourier basis functions whose components take the following form:

$$
\begin{aligned}
b_0(x) &= 1/2 \\
b_{s,i}(x) &= \sin(2\pi i x/T) \text{ for } i = 1, 2, \ldots \\
b_{c,i}(x) &= \cos(2\pi i x/T) \text{ for } i = 1, 2, \ldots
\end{aligned}
$$

where T is the width of the component's domain. The multidimensional Fourier basis functions are all combinations of the one-dimensional components across the state-space axes. Here we use an eighth-order approximation, so i ranges up to 8. The expert policy is to accelerate in the direction of motion.



Example 8.3. Linear regression using Fourier bases used to approximate the value function for the mountain car problem (appendix F.4). Value functions (top row) and resulting policies (bottom row) are shown. The globally approximated value function is a poor fit despite using eighth-order Fourier basis functions. The resulting approximate policy is not a close approximation to the expert policy. The small time step in the mountain car problem causes even small changes in the value function landscape to affect the policy. Optimal utility functions often have complex geometries that can be difficult to capture with global basis functions.

- Nonlinear utility functions can be obtained when using linear regression when combined with an appropriate selection of nonlinear basis functions.
- Neural network regression relieves us of having to specify basis functions, but fitting them is more complex and generally requires us to use gradient descent to tune our parametric approximation of the value function.

8.9 Exercises

Exercise 8.1. The value function approximation methods presented in this chapter have mostly assumed continuous state spaces. The hex world problem, appendix F.1, is discrete, but most of its states can be mapped to two-dimensional locations. It does, however, have an additional terminal state that produces zero reward, which does not have a two-dimensional location. How can one modify the continuous value function approximation methods in this chapter to handle such a state?

Solution: The hex world problem has the agent navigate through a two-dimensional hexagonal grid. However, the agent can enter a single terminal state from one of several grid hexes. This single terminal state presents a challenge for value function approximation methods, which often rely on proximity to infer a state's value.

While the terminal state could be projected to the same state space as the other states, perhaps far away, this hack would nevertheless force a form of proximity into the terminal state's value calculation. Selecting a single position for a state that should be equidistant to multiple predecessor states introduces bias.

One alternative is to treat the terminal state as a special case. The kernel function could be modified to produce infinite distance between the terminal state and any other states.

Another option is to adjust the problem to have a terminal state for every hex that produces a terminal reward. Each terminal state can be coincident with its predecessor state, but offset in an additional dimension. This transformation maintains proximity at the expense of additional states.

Exercise 8.2. A tabular representation is a special case of linear approximate value functions. Show how, for any discrete problem, a tabular representation can be framed as a linear approximate value function.

Solution: Consider a discrete MDP with m states $s_{1:m}$ and n actions $a_{1:n}$. A tabular representation associates a value with each state or state-action pair. We can recover the same behavior using a linear approximate value function. We associate an indicator function with each state or state-action pair, whose value is 1 when the input is the given state or state-action pair and 0 otherwise:

$$\beta_i(s) = (s = s_i) = \begin{cases} 1 & \text{if } s = s_i \\ 0 & \text{otherwise} \end{cases}$$

or

$$\beta_{ij}(s,a) = ((s,a) = (s_i,a_j)) = \begin{cases} 1 & \text{if } (s,a) = (s_i,a_j) \\ 0 & \text{otherwise} \end{cases}$$

Exercise 8.3. Suppose that we have a problem with continuous state and action spaces and we would like to construct both a local approximation and a global approximation of the action value function $Q(s,a) = \theta^\top \beta(s,a)$. For global approximation, we choose the basis functions

$$\beta(s,a) = \left[1, s, a, s^2, sa, a^2\right]$$

Given a set of 100 states $S = s_{1:100}$ and a set of five actions $A = a_{1:5}$, how many parameters are in θ for a local approximation method? How many parameters are in θ for the specified global approximation method?

Solution: In local approximation methods, the state-action values are the parameters. We will have $|S| \times |A| = 100 \times 5 = 500$ parameters in θ. In global approximation methods, the coefficients of the basis functions are the parameters. Since there are six components in $\beta(s,a)$, we will have six parameters in θ.

Exercise 8.4. We are given the states $s_1 = (4,5)$, $s_2 = (2,6)$, and $s_3 = (-1,-1)$, and their corresponding values, $U(s_1) = 2$, $U(s_2) = 10$, and $U(s_3) = 30$. Compute the value at state $s = (1,2)$ using 2-nearest neighbor local approximation with an L_1 distance metric, with an L_2 distance metric, and with an L_∞ distance metric.

Solution: We tabulate the distances from s to the points $s' \in S$ as given here:

$s' \in S$	L_1	L_2	L_∞
$s_1 = (4,5)$	6	$\sqrt{18}$	3
$s_2 = (2,6)$	5	$\sqrt{17}$	4
$s_3 = (-1,-1)$	5	$\sqrt{13}$	3

Using the L_1 norm, we estimate $U(s) = (10+30)/2 = 20$. Using the L_2 norm, we estimate $U(s) = (10+30)/2 = 20$. Using the L_∞ norm, we estimate $U(s) = (2+30)/2 = 16$.

Exercise 8.5. We would like to estimate the value at a state s given the values at a set of two states $S = \{s_1, s_2\}$. If we want to use local approximation value iteration, which of the following weighting functions are valid? If they are invalid, how could the weighting functions be modified to make them valid?

- $\beta(s) = [1,1]$
- $\beta(s) = [1-\lambda, \lambda]$ where $\lambda \in [0,1]$
- $\beta(s) = \left[e^{(s-s_1)^2}, e^{(s-s_2)^2}\right]$

Solution: The first set of weighting functions is not valid, as it violates the constraint $\sum_i \beta_i(s) = 1$. We can modify the weighting functions by normalizing them by their sum:

$$\boldsymbol{\beta}(s) = \left[\frac{1}{1+1}, \frac{1}{1+1}\right] = \left[\frac{1}{2}, \frac{1}{2}\right]$$

The second set of weighting functions is valid. The third set of weighting functions is not valid, as it violates the constraint $\sum_i \beta_i(s) = 1$. We can modify the weighting functions by normalizing them by their sum:

$$\boldsymbol{\beta}(s) = \left[\frac{e^{(s-s_1)^2}}{e^{(s-s_1)^2}+e^{(s-s_2)^2}}, \frac{e^{(s-s_2)^2}}{e^{(s-s_1)^2}+e^{(s-s_2)^2}}\right]$$

Exercise 8.6. Prove that bilinear interpolation is invariant under (nonzero) linear grid scaling.

Solution: It is straightforward to show that the interpolated value is invariant to a linear scaling on one or both axes, such as, $\tilde{U}_\theta(\tilde{s}) = U_\theta(s)$. We show this by substituting all x- and y-values by their scaled versions $\tilde{x} = \beta x$ and $\tilde{y} = \gamma y$, and showing that the grid scalings cancel out:

$$\begin{aligned}
\tilde{U}_\theta(\tilde{s}) &= \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_1 + \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_2 + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_3 + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_4 \\
\tilde{U}_\theta(\tilde{s}) &= \frac{\beta(x_2 - x)\gamma(y_2 - y)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_1 + \frac{\beta(x_2 - x)\gamma(y - y_1)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_2 + \frac{\beta(x - x_1)\gamma(y_2 - y)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_3 + \frac{\beta(x - x_1)\gamma(y - y_1)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_4 \\
\tilde{U}_\theta(\tilde{s}) &= \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4 \\
\tilde{U}_\theta(\tilde{s}) &= U_\theta(s)
\end{aligned}$$

Exercise 8.7. Given the four states $s_1 = [0, 5]$, $s_2 = [0, 25]$, $s_3 = [1, 5]$, and $s_4 = [1, 25]$, and a sample state $s = [0.7, 10]$, generate the interpolant equation $U_\theta(s)$ for arbitrary θ.

Solution: The general form for bilinear interpolation is given in equation (8.12) and reproduced here. To generate the interpolant, we substitute our values into the equation and simplify:

$$\begin{aligned}
U_\theta(s) &= \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4 \\
U_\theta(s) &= \frac{(1 - 0.7)(25 - 10)}{(1 - 0)(25 - 5)}\theta_1 + \frac{(1 - 0.7)(10 - 5)}{(1 - 0)(25 - 5)}\theta_2 + \frac{(0.7 - 0)(25 - 10)}{(1 - 0)(25 - 5)}\theta_3 + \frac{(0.7 - 0)(10 - 5)}{(1 - 0)(25 - 5)}\theta_4 \\
U_\theta(s) &= \frac{9}{40}\theta_1 + \frac{3}{40}\theta_2 + \frac{21}{40}\theta_3 + \frac{7}{40}\theta_4
\end{aligned}$$

Exercise 8.8. Following example 8.1, what are the simplex interpolant weights for a state $\mathbf{s} = [0.4, 0.95, 0.6]$?

Solution: For the given state $\mathbf{s}$, we have $0 \leq x_1 \leq x_3 \leq x_2 \leq 1$, and so our permutation vector is $\mathbf{p} = [1, 3, 2]$. The vertices of our simplex can be generated by starting from $(0, 0, 0)$ and changing each 0 to a 1 in reverse order of the permutation vector. Thus, the vertices of the simplex are $(0, 0, 0)$, $(0, 1, 0)$, $(0, 1, 1)$, and $(1, 1, 1)$.

Any point $\mathbf{s}$ belonging to the simplex can thus be expressed by a weighting of the vertices:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

We can determine the values of the weights in reverse order, finally solving for w_1 by applying the constraint that the weights must sum to 1. We can then compute the weights for $\mathbf{s} = [0.4, 0.95, 0.6]$:

$$\begin{array}{llll} w_4 = s_1 & w_3 = s_3 - w_4 & w_2 = s_2 - w_3 - w_4 & w_1 = 1 - w_2 - w_3 - w_4 \\ w_4 = 0.4 & w_3 = 0.2 & w_2 = 0.35 & w_1 = 0.05 \end{array}$$

9 Online Planning

The solution methods we have discussed so far compute policies offline, before any actions are executed in the real problem. Even offline approximation methods can be intractable in many high-dimensional problems. This chapter discusses *online planning* methods that find actions based on reasoning about states that are reachable from the current state. The *reachable state space* is often orders of magnitude smaller than the full state space, which can significantly reduce storage and computational requirements compared to offline methods. We will discuss a variety of algorithms that aim to make online planning efficient, including pruning the state space, sampling, and planning more deeply along trajectories that appear more promising.

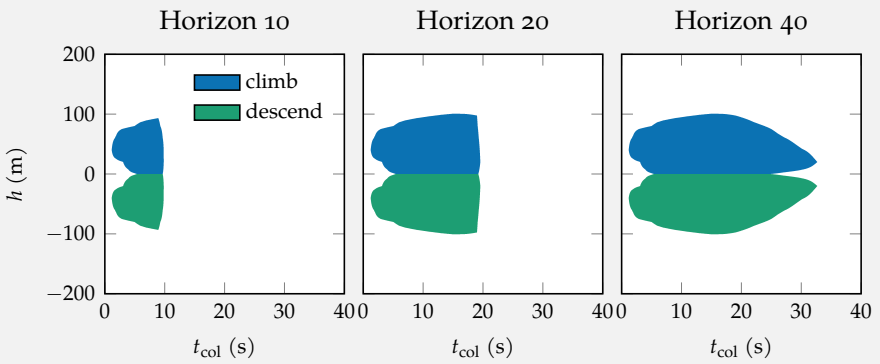
9.1 Receding Horizon Planning

In *receding horizon planning*, we plan from the current state to a maximum fixed horizon or depth d. We then execute the action from our current state, transition to the next state, and replan. The online planning methods discussed in this chapter follow this receding horizon planning scheme. They differ in how they explore different courses of action.

A challenge in applying receding horizon planning is determining the appropriate depth. Deeper planning generally requires more computation. For some problems, a shallow depth can be quite effective; the fact that we replan at each step can compensate for our lack of longer-term modeling. In other problems, greater planning depths may be necessary so that our planner can be driven toward goals or away from unsafe states, as illustrated in example 9.1.

Example 9.1. Receding horizon planning for collision avoidance to different planning depths. In this problem, there are four state variables. These plots show slices of the state space under the assumption that the aircraft is currently level and there has not yet been an advisory. The horizontal axis is the time to collision t_{col}, and the vertical axis is our altitude h relative to the intruder. Appendix F.6 provides additional details about this problem.

Suppose we want to apply receding horizon planning to aircraft collision avoidance. The objective is to provide descend or climb advisories when necessary to avoid collision. A collision occurs when our altitude relative to the intruder h is within ± 50 m and the time to potential collision t_{col} is zero. We want to plan deeply enough so that we can provide an advisory sufficiently early to avoid collision with a high degree of confidence. The plots here show the actions that would be taken by a receding horizon planner with different depths.



If the depth is $d = 10$, we provide advisories only within 10 s of collision. Due to the limitations of the vehicle dynamics and the uncertainty of the behavior of the other aircraft, providing advisories this late compromises safety. With $d = 20$, we can do better, but there are cases where we would want to alert a little earlier to further reduce collision risk. There is no motivation to plan deeper than $d = 40$ because we do not need to advise any maneuvers that far ahead of potential collision.

9.2 Lookahead with Rollouts

Chapter 8 involved extracting policies that are greedy with respect to an approximate value function U through the use of one-step lookahead.[1] A simple online strategy involves acting greedily with respect to values estimated through simulation to depth d. To run a simulation, we need a policy to simulate. Of course, we do not know the optimal policy, but we can use what is called a *rollout policy* instead. Rollout policies are typically stochastic, with actions drawn from a distribution $a \sim \pi(s)$. To produce these rollout simulations, we use a *generative model* $s' \sim T(s, a)$ to generate successor states s' from the distribution $T(s' \mid s, a)$. This generative model can be implemented through draws from a random number generator, which can be easier to implement in practice compared to explicitly representing the distribution $T(s' \mid s, a)$.

[1] The lookahead strategy was originally introduced in algorithm 7.2 as part of our discussion of exact solution methods.

Algorithm 9.1 combines one-step lookahead with values estimated through rollout. This approach often results in better behavior than that of the original rollout policy, but optimality is not guaranteed. It can be viewed as an approximate form of policy improvement used in the policy iteration algorithm (section 7.4). A simple variation of this algorithm is to use multiple rollouts to arrive at a better estimate of the expected discounted return. If we run m simulations for each action and resulting state, the time complexity is $O(m \times |\mathcal{A}| \times |\mathcal{S}| \times d)$.

9.3 Forward Search

Forward search determines the best action to take from an initial state s by expanding all possible transitions up to depth d. These expansions form a *search tree*.[2] Such search trees have a worst-case branching factor of $|\mathcal{S}| \times |\mathcal{A}|$, yielding a computational complexity of $O((|\mathcal{S}| \times |\mathcal{A}|)^d)$. Figure 9.1 shows a search tree applied to a problem with three states and two actions. Figure 9.2 visualizes the states visited during forward search on the hex world problem.

[2] The exploration of the tree occurs as a *depth-first search*. Appendix E reviews depth-first search and other standard search algorithms in the deterministic context.

Algorithm 9.2 calls itself recursively to the specified depth. Once reaching the specified depth, it uses an estimate of the utility provided by the function U. If we simply want to plan to the specified horizon, we set $U(s) = 0$. If our problem requires planning beyond the depth that we can afford to compute online, we can use an estimate of the value function obtained offline using, for example, one of the value function approximations described in the previous chapter. Combining

```
struct RolloutLookahead
    𝒫 # problem
    π # rollout policy
    d # depth
end

randstep(𝒫::MDP, s, a) = 𝒫.TR(s, a)

function rollout(𝒫, s, π, d)
    ret = 0.0
    for t in 1:d
        a = π(s)
        s, r = randstep(𝒫, s, a)
        ret += 𝒫.γ^(t-1) * r
    end
    return ret
end

function (π::RolloutLookahead)(s)
    U(s) = rollout(π.𝒫, s, π.π, π.d)
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.1. A function that runs a rollout of policy `π` in problem `𝒫` from state `s` to depth `d`. It returns the total discounted reward. This function can be used with the `greedy` function (introduced in algorithm 7.5) to generate an action that is likely to be an improvement over the original rollout policy. We will use this algorithm later for problems other than MDPs, requiring us to only have to modify `randstep` appropriately.

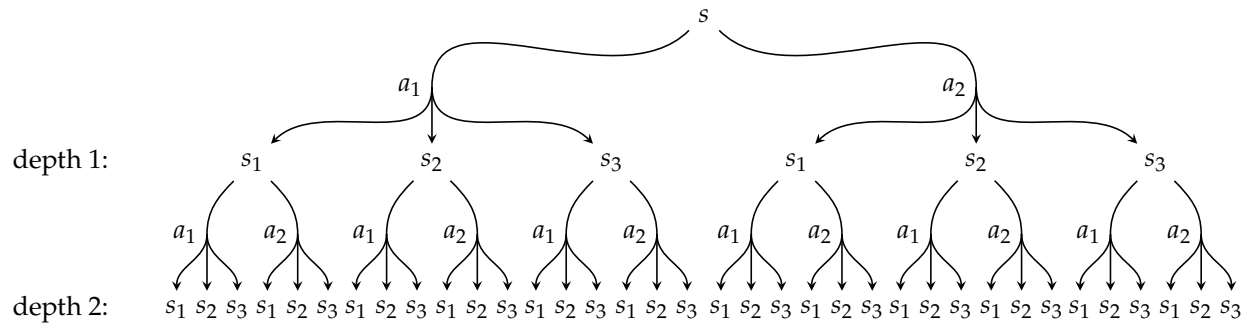


Figure 9.1. A forward search tree for a problem with three states and two actions.

online and offline approaches in this way is sometimes referred to as *hybrid planning*.

```
struct ForwardSearch
    𝒫 # problem
    d # depth
    U # value function at depth d
end

function forward_search(𝒫, s, d, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    U′(s) = forward_search(𝒫, s, d-1, U).u
    for a in 𝒫.𝒜
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::ForwardSearch)(s) = forward_search(π.𝒫, s, π.d, π.U).a
```

Algorithm 9.2. The forward search algorithm for finding an approximately optimal action online for a problem `𝒫` from a current state `s`. The search is performed to depth `d`, at which point the terminal value is estimated with an approximate value function `U`. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`. The problem type is not constrained to be an MDP; section 22.2 uses this same algorithm in the context of partially observable problems with a different implementation for `lookahead`.

9.4 Branch and Bound

Branch and bound (algorithm 9.3) attempts to avoid the exponential computational complexity of forward search. It prunes branches by reasoning about bounds on the value function. The algorithm requires knowing a lower bound on the value function $\underline{U}(s)$ and an upper bound on the action value function $\overline{Q}(s,a)$. The lower bound is used to evaluate the states at the maximum depth. This lower bound is propagated upward through the tree through Bellman updates. If we find that the upper bound of an action at a state is lower than the lower bound of a previously explored action from that state, then we need not explore that action, allowing us to *prune* the associated subtree from consideration.

Branch and bound will give the same result as forward search, but it can be more efficient depending on how many branches are pruned. The worst-case complexity of branch and bound is still the same as forward search. To facilitate

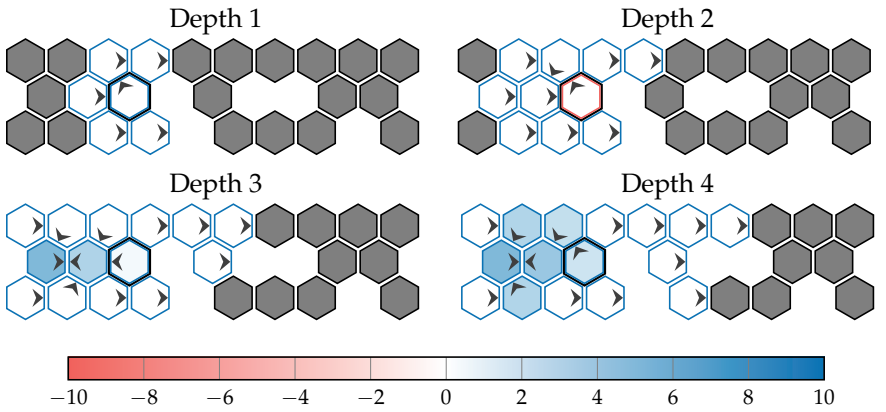


Figure 9.2. Forward search applied to the hex world problem with four maximum depths. The search can visit a node multiple times. The actions and colors for visited states were chosen according to the shallowest, highest-value node in the search tree for that state. The initial state has an additional black border.

```
struct BranchAndBound
    𝒫   # problem
    d   # depth
    Ulo # lower bound on value function at depth d
    Qhi # upper bound on action value function
end

function branch_and_bound(𝒫, s, d, Ulo, Qhi)
    if d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    U′(s) = branch_and_bound(𝒫, s, d-1, Ulo, Qhi).u
    best = (a=nothing, u=-Inf)
    for a in sort(𝒫.𝒜, by=a->Qhi(s,a), rev=true)
        if Qhi(s, a) < best.u
            return best # safe to prune
        end
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::BranchAndBound)(s) = branch_and_bound(π.𝒫, s, π.d, π.Ulo, π.Qhi).a
```

Algorithm 9.3. The branch and bound algorithm for finding an approximately optimal action online for a discrete MDP $\mathcal{P}$ from a current state `s`. The search is performed to depth `d` with value function lower bound `Ulo` and action value function upper bound `Qhi`. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`. This algorithm is also used for POMDPs.

pruning, actions are traversed in descending order by upper bound. Tighter bounds will generally result in more pruning, as shown in example 9.2.

Consider applying branch and bound to the mountain car problem. We can use the value function of a heuristic policy for the lower bound $\underline{U}(s)$, such as a heuristic policy that always accelerates in the direction of motion. For our upper bound $\overline{Q}([x, v], a)$, we can use the return expected when accelerating toward the goal with no hill. Branch and bound visits about a third as many states as forward search.

Example 9.2. Branch and bound applied to the mountain car problem (appendix F.4). Branch and bound can achieve a significant speedup over forward search.

9.5 Sparse Sampling

A method known as *sparse sampling*[3] (algorithm 9.4) attempts to reduce the branching factor of forward search and branch and bound. Instead of branching on all possible next states, we consider only a limited number of samples of the next state. Although the sampling of the next state results in an approximation, this method can work well in practice and can significantly reduce computation. If we draw m samples of the next state for each action node in the search tree, the computational complexity is $O\left((m \times |\mathcal{A}|)^d\right)$, which is still exponential in the depth but no longer depends on the size of the state space. Figure 9.3 shows an example.

[3] M. J. Kearns, Y. Mansour, and A. Y. Ng, "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes," *Machine Learning*, vol. 49, no. 2–3, pp. 193–208, 2002.

9.6 Monte Carlo Tree Search

Monte Carlo tree search (algorithm 9.5) avoids the exponential complexity in the horizon by running m simulations from the current state.[4] During these simulations, the algorithm updates estimates of the action value function $Q(s, a)$ and a record of the number of times a particular state-action pair has been selected, $N(s, a)$. After running these m simulations from our current state s, we simply choose the action that maximizes our estimate of $Q(s, a)$.

A simulation (algorithm 9.6) begins by traversing the explored state space, consisting of the states for which we have estimates of Q and N. We follow an exploration strategy to choose actions from the various states. A common approach is to select the action that maximizes the *UCB1 exploration heuristic*:[5]

[4] For a survey, see C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] UCB stands for upper confidence bound. This is one of many strategies discussed by P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-Time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2–3, pp. 235–256, 2002. The equation is derived from the Chernoff-Hoeffding bound.

```
struct SparseSampling
    𝒫 # problem
    d # depth
    m # number of samples
    U # value function at depth d
end

function sparse_sampling(𝒫, s, d, m, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒫.𝒜
        u = 0.0
        for i in 1:m
            s′, r = randstep(𝒫, s, a)
            a′, u′ = sparse_sampling(𝒫, s′, d-1, m, U)
            u += (r + 𝒫.γ*u′) / m
        end
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::SparseSampling)(s) = sparse_sampling(π.𝒫, s, π.d, π.m, π.U).a
```

Algorithm 9.4. The sparse sampling algorithm for finding an approximately optimal action online for a discrete problem `𝒫` from a current state `s` to depth `d` with `m` samples per action. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`.

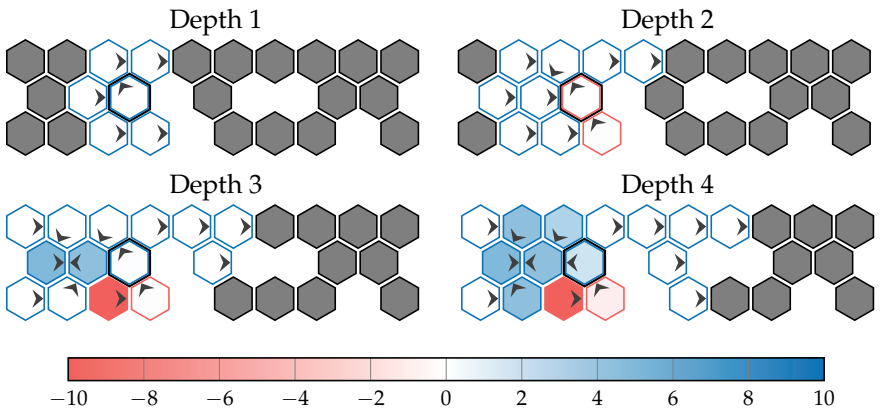


Figure 9.3. Sparse sampling with $m = 10$ applied to the hex world problem. Visited tiles are colored according to their estimated value. The bordered tile is the initial state. Compare to forward search in figure 9.2.

```
struct MonteCarloTreeSearch
    𝒫 # problem
    N # visit counts
    Q # action value estimates
    d # depth
    m # number of simulations
    c # exploration constant
    U # value function estimate
end

function (π::MonteCarloTreeSearch)(s)
    for k in 1:π.m
        simulate!(π, s)
    end
    return argmax(a→π.Q[(s,a)], π.𝒫.𝒜)
end
```

Algorithm 9.5. The Monte Carlo tree search policy for finding an approximately optimal action from a current state `s`.

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{9.1}$$

where $N(s) = \sum_a N(s,a)$ is the total visit count to s and c is an exploration parameter that scales the value of unexplored actions. The second term corresponds to an *exploration bonus*. If $N(s,a) = 0$, the bonus is defined to be infinity. With $N(s,a)$ in the denominator, the exploration bonus is higher for actions that have not been tried as frequently. Algorithm 9.7 implements this exploration strategy. We will discuss many other exploration strategies later in chapter 15.

As we take actions specified by algorithm 9.7, we step into new states sampled from the generative model $T(s,a)$, similar to the sparse sampling method. We increment the visit count $N(s,a)$ and update $Q(s,a)$ to maintain the mean value.

At some point, we will either reach the maximum depth or a state that we have not yet explored. If we reach an unexplored state s, we initialize $N(s,a)$ and $Q(s,a)$ to zero for each action a. We may modify algorithm 9.6 to initialize these counts and value estimates to some other values based on prior expert knowledge of the problem. After initializing N and Q, we then return a value estimate at the state s. It is common to estimate this value through a rollout of some policy using the process outlined in section 9.2.

Examples 9.3 to 9.7 work through an illustration of Monte Carlo tree search applied to the 2048 problem. Figure 9.4 shows a search tree generated by running Monte Carlo tree search on 2048. Example 9.8 discusses the impact of using different strategies for estimating values.

```
function simulate!(π::MonteCarloTreeSearch, s, d=π.d)
    if d ≤ 0
        return π.U(s)
    end
    𝒫, N, Q, c = π.𝒫, π.N, π.Q, π.c
    𝒜, TR, γ = 𝒫.𝒜, 𝒫.TR, 𝒫.γ
    if !haskey(N, (s, first(𝒜)))
        for a in 𝒜
            N[(s,a)] = 0
            Q[(s,a)] = 0.0
        end
        return π.U(s)
    end
    a = explore(π, s)
    s′, r = TR(s,a)
    q = r + γ*simulate!(π, s′, d-1)
    N[(s,a)] += 1
    Q[(s,a)] += (q-Q[(s,a)])/N[(s,a)]
    return q
end
```

Algorithm 9.6. A method for running a Monte Carlo tree search simulation starting from state `s` to depth `d`.

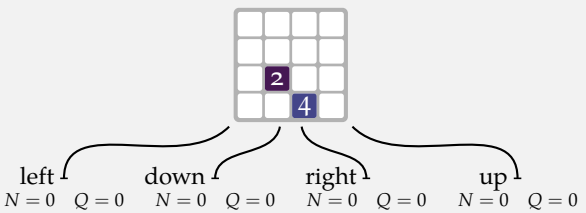
```
bonus(Nsa, Ns) = Nsa == 0 ? Inf : sqrt(log(Ns)/Nsa)

function explore(π::MonteCarloTreeSearch, s)
    𝒜, N, Q, c = π.𝒫.𝒜, π.N, π.Q, π.c
    Ns = sum(N[(s,a)] for a in 𝒜)
    return argmax(a->Q[(s,a)] + c*bonus(N[(s,a)], Ns), 𝒜)
end
```

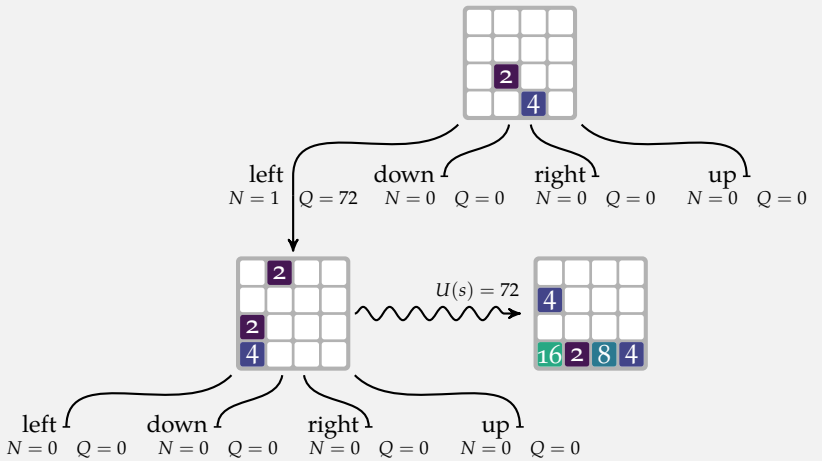
Algorithm 9.7. An exploration policy used in Monte Carlo tree search when determining which nodes to traverse through the search tree. The policy is determined by a dictionary of state-action visitation counts `N` and values `Q`, as well as an exploration parameter `c`. When `N[(s,a)] = 0`, the policy returns infinity.

Example 9.3. An example of solving 2048 with Monte Carlo tree search.

Consider using Monte Carlo tree search to play 2048 (appendix F.2) with a maximum depth $d = 10$, an exploration parameter $c = 100$, and a 10-step random rollout to estimate $U(s)$. Our first simulation expands the starting state. The count and value are initialized for each action from the initial state:



The second simulation begins by selecting the best action from the initial state according to our exploration strategy in equation (9.1). Because all states have the same value, we arbitrarily choose the first action, `left`. We then sample a new successor state and expand it, initializing the associated counts and value estimates. A rollout is run from the successor state and its value is used to update the value of `left`:



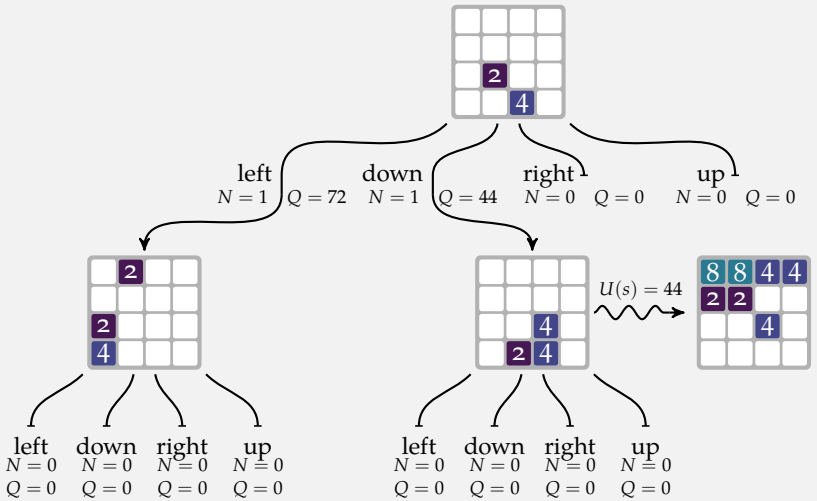
Example 9.4. A (continued) example of solving 2048 with Monte Carlo tree search.

Example 9.5. A (continued) example of solving 2048 with Monte Carlo tree search.

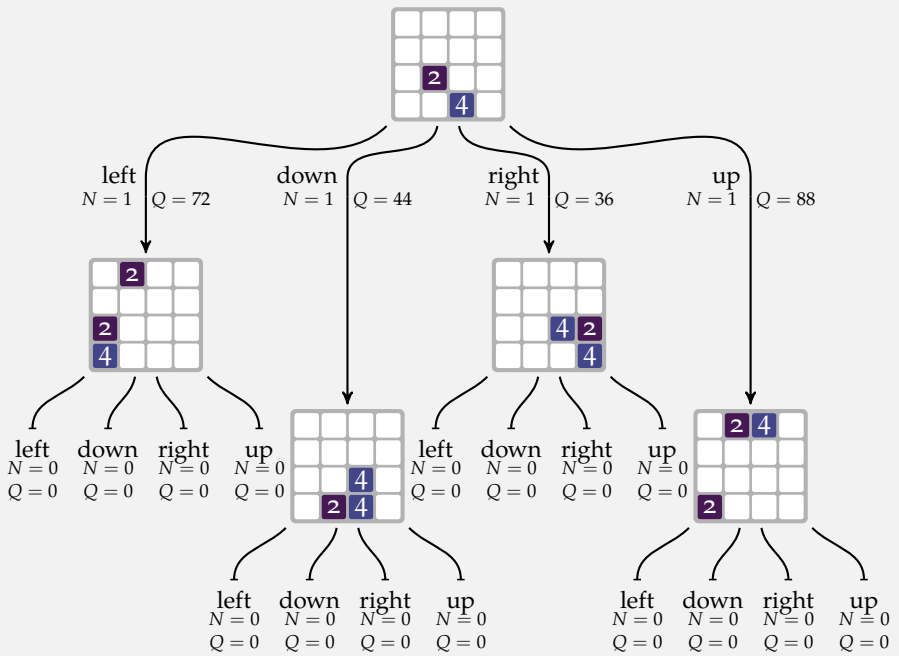
The third simulation begins by selecting the second action, `down`, because it has infinite value due to the exploration bonus given for unexplored actions. The first action has finite value:

$$Q(s_0, \texttt{left}) + c\sqrt{\frac{\log N(s_0)}{N(s_0, \texttt{left})}} = 72 + 100\sqrt{\frac{\log 1}{1}} = 72$$

We take the `down` action and sample a new successor state, which is expanded. A rollout is run from the successor state and its value is used to update the value of `down`:

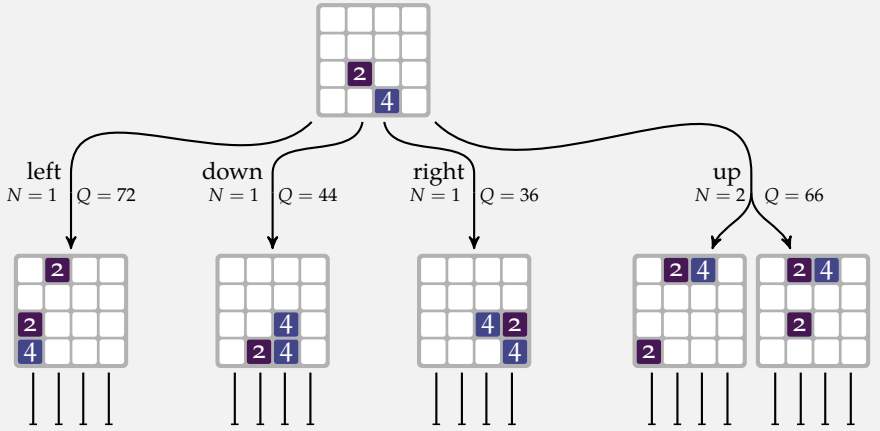


The next two simulations select `right` and `up`, respectively. This results in the following:



Example 9.6. A (continued) example of solving 2048 with Monte Carlo tree search.

In the fifth simulation, up has the highest value. The successor state after taking up in the source state will not necessarily be the same as the first time up was selected. We evaluate $U(s) = 44$ and update our visitation count to 2 and our estimated value to $Q \leftarrow 88 + (44 - 88)/2 = 66$. A new successor node is created:



Example 9.7. A (continued) example of solving 2048 with Monte Carlo tree search.

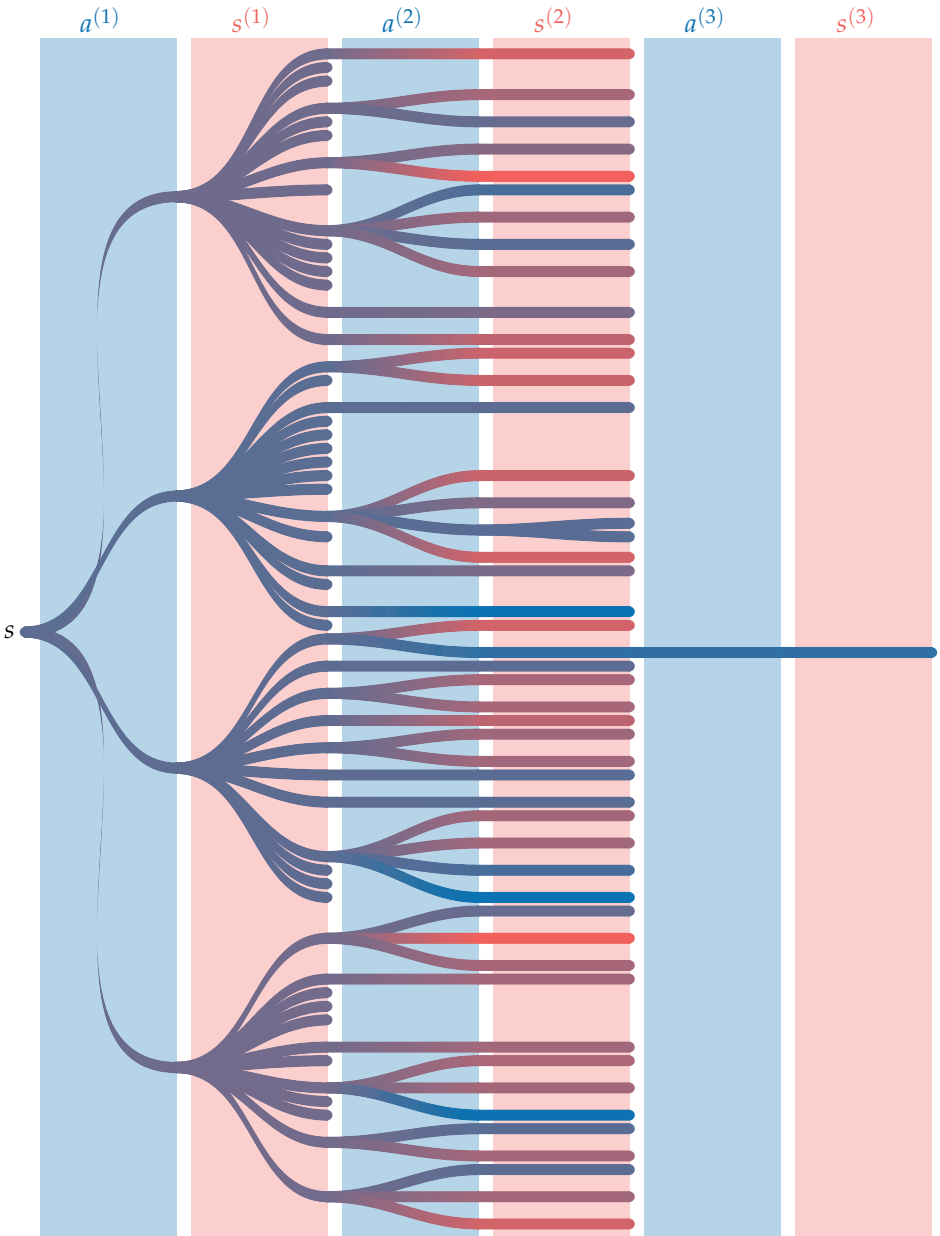


Figure 9.4. A Monte Carlo tree search tree on 2048 after 100 simulations. In general, Monte Carlo tree search for MDPs produces a search graph because there can be multiple ways to reach the same state. The colors in the tree indicate the estimated values at the nodes, with high values in blue and low values in red. The tree is shallow, with a fairly high branching factor, because 2048 has many reachable states for each action.

Example 9.8. The performance of Monte Carlo tree search varies with the number of simulations and as the board evaluation method is changed. Heuristic board evaluations tend to be efficient and can more effectively guide the search when run counts are low. Lookahead rollout evaluations take about 18 times longer than heuristic evaluations.

Rollouts are not the only means by which we can estimate utilities in Monte Carlo tree search. Custom evaluation functions can often be crafted for specific problems to help guide the algorithm. For example, we can encourage Monte Carlo tree search to order its tiles in 2048 using evaluation functions that return the weighted sum across tile values:

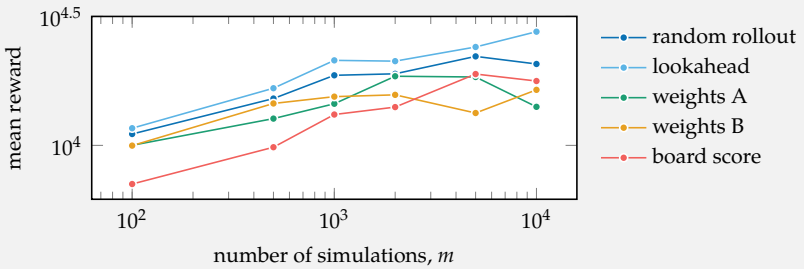
1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

heuristic A weights

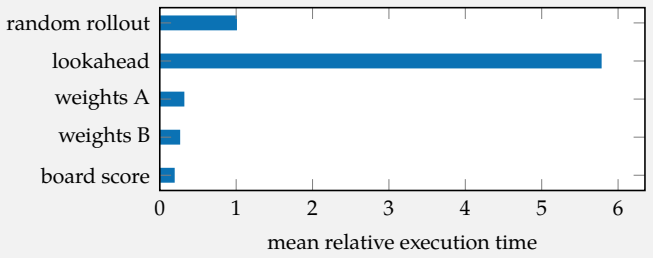
0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

heuristic B weights

The plot here compares Monte Carlo tree search on 2048 using rollouts with a uniform random policy, rollouts with a one-step lookahead policy, the two evaluation functions, and using the current board score:



Rollouts perform well but require more execution time. Here we plot the average execution time relative to random rollouts for $m = 100$ from a starting state:



There are variations of this basic Monte Carlo tree search algorithm that can better handle large action and state spaces. Instead of expanding all the actions, we can use *progressive widening*. The number of actions considered from state s is limited to $\theta_1 N(s)^{\theta_2}$, where θ_1 and θ_2 are hyperparameters. Similarly, we can limit the number of states that result from taking action a from state s in the same way, using what is called *double progressive widening*. If the number of states that have been simulated from state s after action a is below $\theta_3 N(s,a)^{\theta_4}$, then we sample a new state; otherwise, we sample one of the previously sampled states with probability proportional to the number of times it has been visited. This strategy can be used to handle large as well as continuous action and state spaces.[6]

[6] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Learning and Intelligent Optimization (LION)*, 2011.

9.7 Heuristic Search

Heuristic search (algorithm 9.8) uses m simulations of a greedy policy with respect to a value function U from the current state s.[7] The value function U is initialized to an upper bound of the value function $\overline{U}$, which is referred to as a *heuristic*. As we run these simulations, we update our estimate of U through lookahead. After running these simulations, we simply select the greedy action from s with respect to U. Figure 9.5 shows how U and the greedy policy changes with the number of simulations.

[7] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, vol. 72, no. 1–2, pp. 81–138, 1995. Other forms of heuristic search are discussed by Mausam and A. Kolobov, *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool, 2012.

Heuristic search is guaranteed to converge to the optimal utility function so long as the heuristic $\overline{U}$ is indeed an upper bound on the value function.[8] The efficiency of the search depends on the tightness of the upper bound. Unfortunately, tight bounds can be difficult to obtain in practice. While a heuristic that is not a true upper bound may not converge to the optimal policy, it may still converge to a policy that performs well. The time complexity is $O(m \times d \times |S| \times |A|)$.

[8] Such a heuristic is referred to as an *admissible heuristic*.

9.8 Labeled Heuristic Search

Labeled heuristic search (algorithm 9.9) is a variation of heuristic search that runs simulations with value updates while labeling states based on whether their value is solved.[9] We say that a state s is solved if its utility residual falls below a threshold $\delta > 0$:

$$|U_{k+1}(s) - U_k(s)| < \delta \tag{9.2}$$

[9] B. Bonet and H. Geffner, "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

```
struct HeuristicSearch
    𝒫   # problem
    Uhi # upper bound on value function
    d   # depth
    m   # number of simulations
end

function simulate!(π::HeuristicSearch, U, s)
    𝒫 = π.𝒫
    for d in 1:π.d
        a, u = greedy(𝒫, U, s)
        U[s] = u
        s = rand(𝒫.T(s, a))
    end
end

function (π::HeuristicSearch)(s)
    U = [π.Uhi(s) for s in π.𝒫.𝒮]
    for i in 1:π.m
        simulate!(π, U, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.8. Heuristic search runs `m` simulations starting from an initial state `s` to a depth `d`. The search is guided by a heuristic initial value function `Uhi`, which leads to optimality in the limit of simulations if it is an upper bound on the optimal value function.

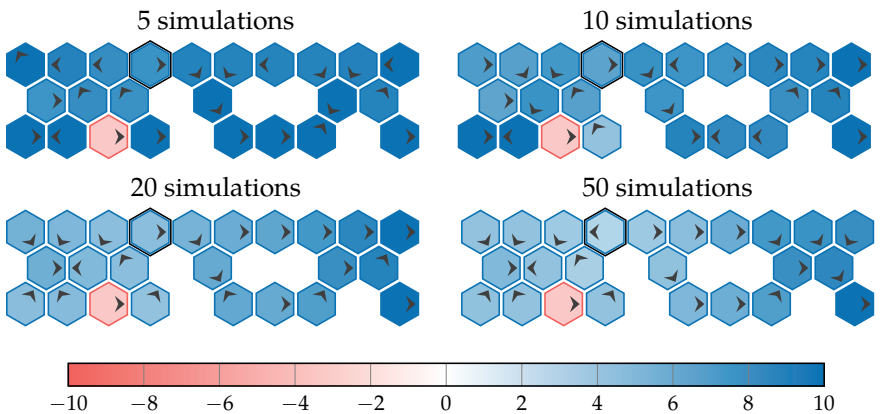


Figure 9.5. Heuristic search runs simulations with Bellman updates to improve a value function on the hex world problem to obtain a policy from an initial state, shown here with an additional black border. These simulations are run to depth 8 with heuristic $\overline{U}(s) = 10$. Each hex is colored according to the utility function value in that iteration. We see that the algorithm eventually finds an optimal policy.

We run simulations with value updates until the current state is solved. In contrast with the heuristic search in the previous section, which runs a fixed number of iterations, this labeling process focuses computational effort on the most important areas of the state space.

```
struct LabeledHeuristicSearch
    𝒫     # problem
    Uhi   # upper bound on value function
    d     # depth
    δ     # gap threshold
end

function (π::LabeledHeuristicSearch)(s)
    U, solved = [π.Uhi(s) for s in 𝒫.𝒮], Set()
    while s ∉ solved
        simulate!(π, U, solved, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.9. Labeled heuristic search, which runs simulations starting from the current state to depth `d` until the current state is solved. The search is guided by a heuristic upper bound on the value function `Uhi` and maintains a growing set of solved states. States are considered solved when their utility residuals fall below `δ`. A value function policy is returned.

Simulations in labeled heuristic search (algorithm 9.10) begin by running to a maximum depth of d by following a policy that is greedy with respect to our estimated value function U, similar to the heuristic search in the previous section. We may stop a simulation before a depth of d if we reach a state that has been labeled as solved in a prior simulation.

```
function simulate!(π::LabeledHeuristicSearch, U, solved, s)
    visited = []
    for d in 1:π.d
        if s ∈ solved
            break
        end
        push!(visited, s)
        a, u = greedy(π.𝒫, U, s)
        U[s] = u
        s = rand(π.𝒫.T(s, a))
    end
    while !isempty(visited)
        if label!(π, U, solved, pop!(visited))
            break
        end
    end
end
```

Algorithm 9.10. Simulations are run from the current state to a maximum depth `d`. We stop a simulation at depth `d` or if we encounter a state that is in the set `solved`. After a simulation, we call `label!` on the states we visited in reverse order.

After each simulation, we iterate over the states we visited during that simulation in reverse order, performing a labeling routine on each state and stopping if a state is found that is not solved. The labeling routine (algorithm 9.11) searches the states in the *greedy envelope* of s, which is defined to be the states reachable from s under a greedy policy with respect to U. The state s is considered not solved if there is a state in the greedy envelope of s whose utility residual is greater than threshold δ. If no such state is found, then s is marked as solved—as well as all states in the greedy envelope of s because they must have converged as well. If a state with a sufficiently large utility residual is found, then the utilities of all states traversed during the search of the greedy enveloped are updated.

Figure 9.6 shows several different greedy envelopes. Figure 9.7 shows the states traversed in a single iteration of labeled heuristic search. Figure 9.8 shows the progression of heuristic search on the hex world problem.

9.9 Open-Loop Planning

The online methods discussed in this chapter, as well as the offline methods discussed in the previous chapters, are examples of *closed-loop planning*, which involves accounting for future state information in the planning process.[10] Often, *open-loop planning* can provide a satisfactory approximation of an optimal closed-loop plan while greatly enhancing computational efficiency by avoiding having to reason about the acquisition of future information. Sometimes this open-loop planning approach is referred to as *model predictive control*.[11] As with receding horizon control, model predictive control solves the open-loop problem, executes the action from our current state, transitions to the next state, and then replans.

[10] The loop in this context refers to the observe-act loop introduced in section 1.1.

[11] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2019.

Open-loop plans can be represented as a sequence of actions up to a depth d. The planning process reduces to an optimization problem:

$$\underset{a_{1:d}}{\text{maximize}} \quad U(a_{1:d}) \tag{9.3}$$

where $U(a_{1:d})$ is the expected return when executing the sequence of actions $a_{1:d}$. Depending on the application, this optimization problem may be convex or lend itself to a convex approximation, meaning that it can be solved quickly using a variety of algorithms.[12] Later in this section, we will discuss a few different formulations that can be used to transform equation (9.3) into a convex problem.

[12] Appendix A.6 reviews convexity. An introduction to convex optimization is provided by S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

```
function expand(π::LabeledHeuristicSearch, U, solved, s)
    𝒫, δ = π.𝒫, π.δ
    𝒮, 𝒜, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T
    found, toexpand, envelope = false, Set(s), []
    while !isempty(toexpand)
        s = pop!(toexpand)
        push!(envelope, s)
        a, u = greedy(𝒫, U, s)
        if abs(U[s] - u) > δ
            found = true
        else
            for s′ in 𝒮
                if T(s,a,s′) > 0 && s′ ∉ (solved ∪ envelope)
                    push!(toexpand, s′)
                end
            end
        end
    end
    return (found, envelope)
end

function label!(π::LabeledHeuristicSearch, U, solved, s)
    if s ∈ solved
        return false
    end
    found, envelope = expand(π, U, solved, s)
    if found
        for s ∈ reverse(envelope)
            U[s] = greedy(π.𝒫, U, s).u
        end
    else
        union!(solved, envelope)
    end
    return found
end
```

Algorithm 9.11. The `label!` function will attempt to find a state in the greedy envelope of `s` whose utility residual exceeds a threshold `δ`. The function `expand` computes the greedy envelope of `s` and determines whether any of those states have utility residuals above the threshold. If a state has a residual that exceeds the threshold, then we update the utilities of the states in the envelope. Otherwise, we add that envelope to the set of solved states.

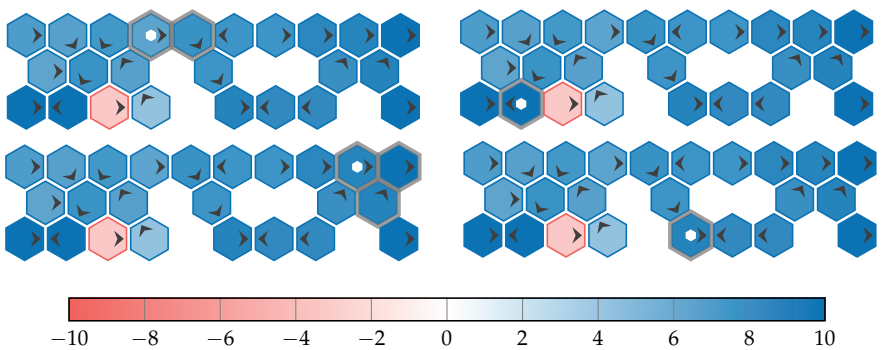


Figure 9.6. The greedy envelope for $\delta = 1$ for several states visualized for a value function on the hex world problem. The value function was obtained by running basic heuristic search for 10 iterations from an initial state, shown with a white hex center, to a maximum depth of 8. We find that the size of the greedy envelope, outlined in gray, can vary widely depending on the state.

Figure 9.7. A single iteration of labeled heuristic search conducts an exploratory run (arrows), followed by labeling (hexagonal border). Only two states are labeled in this iteration: the hidden terminal state and the state with a hexagonal border. Both the exploratory run and the labeling step update the value function.

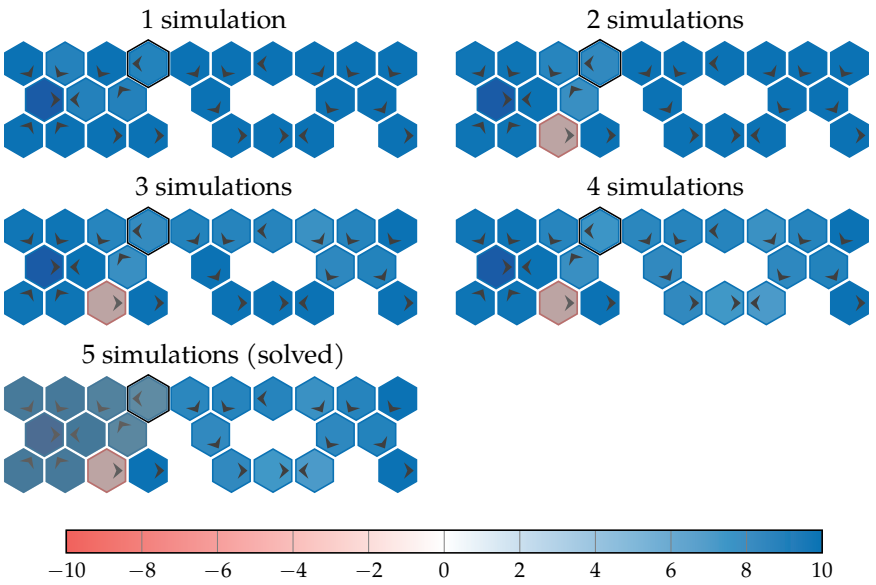
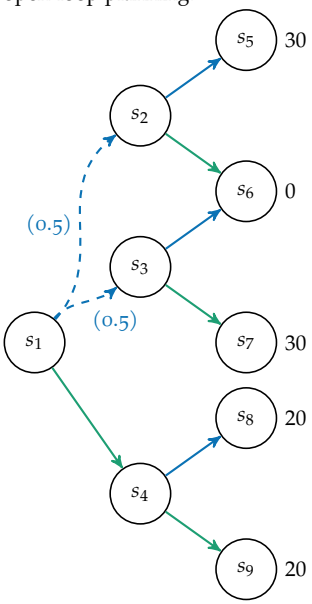


Figure 9.8. A progression of heuristic search on the hex world problem using $\delta = 1$ and a heuristic $\overline{U}(s) = 10$. The solved states in each iteration are covered in a gray wash. The set of solved states grows from the terminal reward state back toward the initial state with the dark border.

Open-loop planning can often allow us to devise effective decision strategies in high-dimensional spaces where closed-loop planning is computationally infeasible. This type of planning gains this efficiency by not accounting for future information. Example 9.9 provides a simple instance of where open-loop planning can result in poor decisions, even when we account for stochasticity.

Example 9.9. Suboptimality of open-loop planning.



Consider a problem with nine states, as shown in the margin, with two decision steps starting from the initial state s_1. In our decisions, we must decide between going up (blue arrows) and going down (green arrows). The effects of these actions are deterministic, except that if we go up from s_1, then we end up in state s_2 half the time and in state s_3 half the time. We receive a reward of 30 in states s_5 and s_7 and a reward of 20 in states s_8 and s_9, as indicated in the illustration.

There are exactly four open-loop plans: (up, up), (up, down), (down, up), and (down, down). In this simple example, it is easy to compute their expected utilities:

- $U(\text{up}, \text{up}) = 0.5 \times 30 + 0.5 \times 0 = 15$
- $U(\text{up}, \text{down}) = 0.5 \times 0 + 0.5 \times 30 = 15$
- $U(\text{down}, \text{up}) = 20$
- $U(\text{down}, \text{down}) = 20$

According to the set of open-loop plans, it is best to choose down from s_1 because our expected reward is 20 instead of 15.

Closed-loop planning, in contrast, takes into account the fact that we can base our next decision on the observed outcome of our first action. If we choose to go up from s_1, then we can choose to go down or up depending on whether we end up in s_2 or s_3, thereby guaranteeing a reward of 30.

9.9.1 Deterministic Model Predictive Control

A common approximation to make $U(a_{1:d})$ amenable to optimization is to assume deterministic dynamics:

$$\begin{aligned} \underset{a_{1:d}, s_{2:d}}{\text{maximize}} \quad & \sum_{t=1}^{d} \gamma^t R(s_t, a_t) \\ \text{subject to} \quad & s_{t+1} = T(s_t, a_t), \quad t \in 1:d-1 \end{aligned} \tag{9.4}$$

where s_1 is the current state and $T(s, a)$ is a deterministic transition function that returns the state that results from taking action a from state s. A common strategy for producing a suitable deterministic transition function from a stochastic transition function is to use the most likely transition. If the dynamics in equation (9.4) are linear and the reward function is convex, then the problem is convex.

Example 9.10 provides an instance involving navigating to a goal state while avoiding an obstacle and minimizing acceleration effort. Both the state space and action space are continuous, and we can find a solution in well under a second. Replanning after every step can help compensate for stochasticity or unexpected events. For example, if the obstacle moves, we can readjust our plan, as illustrated in figure 9.9.

9.9.2 Robust Model Predictive Control

We can change the problem formulation to provide robustness to outcome uncertainty. There are many *robust model predictive control* formulations,[13] but one involves choosing the best open-loop plan given the worst-case state transitions. This formulation defines $T(s, a)$ to be an *uncertainty set* consisting of all possible states that can result from taking action a in state s. In other words, the uncertainty set is the support of the distribution $T(\cdot \mid s, a)$. Optimizing with respect to worst-case state transitions requires transforming the optimization problem in equation (9.4) into a *minimax* problem:

$$\begin{aligned} \underset{a_{1:d}}{\text{maximize}} \quad & \underset{s_{2:d}}{\text{minimize}} \sum_{t=1}^{d} \gamma^t R(s_t, a_t) \\ \text{subject to} \quad & s_{t+1} \in T(s_t, a_t), \quad t \in 1:d-1 \end{aligned} \tag{9.5}$$

[13] A. Bemporad and M. Morari, "Robust Model Predictive Control: A Survey," in *Robustness in Identification and Control*, A. Garulli, A. Tesi, and A. Vicino, eds., Springer, 1999, pp. 207–226.

Example 9.10. Open-loop planning in a deterministic environment. We attempt to find a path around a circular obstacle. This implementation uses the `JuMP.jl` interface to the Ipopt solver. A. Wächter and L. T. Biegler, "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2005.

In this problem, our state $\mathbf{s}$ represents our agent's two-dimensional position concatenated with its two-dimensional velocity vector, with $\mathbf{s}$ initially set to $[0, 0, 0, 0]$. Our action $\mathbf{a}$ is an acceleration vector, where each component must be between ± 1. At each step, we use our action to update our velocity, and we use our velocity to update our position. Our objective is to reach a goal state of $\mathbf{s}_{\text{goal}} = [10, 10, 0, 0]$. We plan up to $d = 10$ steps with no discounting. With each step, we accumulate a cost of $\|\mathbf{a}_t\|_2^2$ to minimize acceleration effort. At the last step, we want to be as close to the goal state as possible, with a penalty of $100\|\mathbf{s}_d - \mathbf{s}_{\text{goal}}\|_2^2$. We also have to ensure that we avoid a circular obstacle with radius 2 centered at $[3, 4]$. We can formulate this problem as follows and extract the first action from the plan:

```
model = Model(Ipopt.Optimizer)
d = 10
current_state = zeros(4)
goal = [10,10,0,0]
obstacle = [3,4]
@variables model begin
    s[1:4, 1:d]
    -1 ≤ a[1:2,1:d] ≤ 1
end
# velocity update
@constraint(model, [i=2:d,j=1:2], s[2+j,i] == s[2+j,i-1] + a[j,i-1])
# position update
@constraint(model, [i=2:d,j=1:2], s[j,i] == s[j,i-1] + s[2+j,i-1])
# initial condition
@constraint(model, s[:,1] .== current_state)
# obstacle
@constraint(model, [i=1:d], sum((s[1:2,i] - obstacle).^2) ≥ 4)
@objective(model, Min, 100*sum((s[:,d] - goal).^2) + sum(a.^2))
optimize!(model)
action = value.(a[:,1])
```

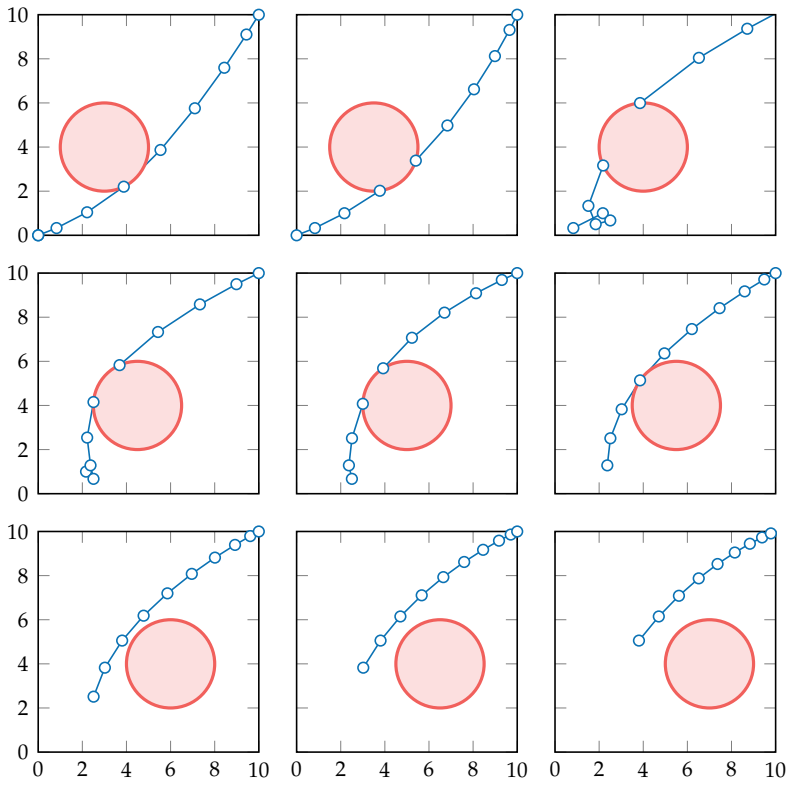


Figure 9.9. Model predictive control applied to the problem in example 9.10, with the addition of a moving obstacle. The sequence progresses left-to-right, and top-to-bottom. Initially, we have a plan that passes to the right of the obstacle, but in the third cell, we see that we must change our mind and pass to the left. We have to maneuver around a little to adjust our velocity vector appropriately with minimal effort. Of course, we could have created a better path (in terms of our utility function) if our planning process had known that the obstacle was moving in a particular direction.

Unfortunately, this formulation can result in extremely conservative behavior. If we adapt example 9.10 to model the uncertainty in the motion of the obstacle, the accumulation of uncertainty may become quite large, even when planning with a relatively short horizon. One way to reduce the accumulation of uncertainty is to restrict the uncertainty set output by $T(s, a)$ to contain only, say, 95 % of the probability mass. Another issue with this approach is that the minimax optimization problem is often not convex and difficult to solve.

9.9.3 Multiforecast Model Predictive Control

One way to address the computational challenge within the minimax problem in equation (9.5) is to use m forecast scenarios, each of which follows its own deterministic transition function.[14] There are various formulations of this kind of *multiforecast model predictive control*, which is a type of *hindsight optimization*.[15] One common approach is to have the deterministic transition functions depend on the step k, $T_i(s, a, k)$, which is the same as augmenting the state space to include depth. Example 9.11 demonstrates how this might be done for a linear Gaussian model.

[14] S. Garatti and M. C. Campi, "Modulating Robustness in Control Design: Principles and Algorithms," *IEEE Control Systems Magazine*, vol. 33, no. 2, pp. 36–51, 2013.

[15] It is called hindsight optimization because it represents a solution that is optimizing using knowledge about action outcomes that can only be known in hindsight. E. K. P. Chong, R. L. Givan, and H. S. Chang, "A Framework for Simulation-Based Network Control via Hindsight Optimization," in *IEEE Conference on Decision and Control (CDC)*, 2000.

Example 9.11. Modeling linear Gaussian transition dynamics in multiforecast model predictive control.

Suppose we have a problem with linear Gaussian dynamics:

$$T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a}, \mathbf{\Sigma})$$

The problem in figure 9.9 is linear, with no uncertainty, but if we allow the obstacle to move according to a Gaussian distribution at each step, then the dynamics become linear Gaussian. We can approximate the dynamics using a set of m forecast scenarios, each consisting of d steps. We can pull $m \times d$ samples $\epsilon_{ik} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ and define the deterministic transition functions:

$$T_i(\mathbf{s}, \mathbf{a}, k) = \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \epsilon_{ik}$$

We try to find the best sequence of actions for the worst sampled scenario:

$$\begin{aligned} &\underset{a_{1:d}}{\text{maximize}} \quad \underset{i,\, s_{2:d}}{\text{minimize}} \sum_{k=1}^{d} \gamma^k R(s_k, a_k) \\ &\text{subject to} \quad s_{k+1} = T_i(s_k, a_k, k), \quad k \in 1 : d-1 \end{aligned} \tag{9.6}$$

This problem can be much easier to solve than the original robust problem.

We can also use a multiforecast approach to optimize the average case.[16] The formulation is similar to equation (9.6), except that we replace the minimization with an expectation and allow different action sequences to be taken for different scenarios, with the constraint that the first action must agree:

$$\begin{aligned} \underset{a_{1:d}^{(1:m)}, s_{2:d}^{(i)}}{\text{maximize}} \quad & \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{d} \gamma^k R(s_k^{(i)}, a_k^{(i)}) \\ \text{subject to} \quad & s_{k+1}^{(i)} = T_i(s_k^{(i)}, a_k^{(i)}, k), \quad k \in 1:d-1, i \in 1:m \\ & a_1^{(i)} = a_1^{(j)}, \qquad\qquad\qquad i \in 1:m, j \in 1:m \end{aligned} \tag{9.7}$$

This formulation can result in robust behavior without being overly conservative, while still maintaining computational tractability. Both formulations in equations (9.6) and (9.7) can be made more robust by increasing the number of forecast scenarios m at the expense of additional computation.

[16] This approach was applied to optimizing power flow policies by N. Moehle, E. Busseti, S. Boyd, and M. Wytock, "Dynamic Energy Management," in *Large Scale Optimization in Supply Chains and Smart Manufacturing*, Springer, 2019, pp. 69–126.

9.10 Summary

- Online methods plan from the current state, focusing computation on states that are reachable.
- Receding horizon planning involves planning to a certain horizon and then replanning with each step.
- Lookahead with rollouts involves acting greedily with respect to values estimated using simulations of a rollout policy; it is computationally efficient compared to other algorithms, but there are no guarantees on performance.
- Forward search considers all state-action transitions up to a certain depth, resulting in computational complexity that grows exponentially in both the number of states and the number of actions.
- Branch and bound uses upper and lower bound functions to prune portions of the search tree that will not lead to a better outcome in expectation.
- Sparse sampling avoids the exponential complexity in the number of states by limiting the number of sampled transitions from every search node.

- Monte Carlo tree search guides search to promising areas of the search space by taking actions that balance exploration with exploitation.
- Heuristic search runs simulations of a policy that is greedy with respect to a value function that is updated along the way using lookahead.
- Labeled heuristic search reduces computation by not reevaluating states whose values have converged.
- Open-loop planning aims to find the best possible sequence of actions and can be computationally efficient if the optimization problem is convex.

9.11 Exercises

Exercise 9.1. Why does branch and bound have the same worst-case computational complexity as forward search?

Solution: In the worst case, branch and bound will never prune, resulting in a traversal of the same search tree as forward search with the same complexity.

Exercise 9.2. Given two admissible heuristics h_1 and h_2, how can we use both of them in heuristic search?

Solution: Create a new heuristic $h(s) = \min\{h_1(s), h_2(s)\}$ and use it instead. This new heuristic is guaranteed to be admissible and cannot be a worse bound than either h_1 or h_2. Both $h_1(s) \geq U^*(s)$ and $h_2(s) \geq U^*(s)$ imply that $h(s) \geq U^*(s)$.

Exercise 9.3. Given two inadmissible heuristics h_1 and h_2, describe a way we can use both of them in heuristic search.

Solution: We could define a new heuristic $h_3(s) = \max(h_1(s), h_2(s))$ to get a potentially admissible, or ''less-inadmissible,'' heuristic. It may be slower to converge, but it may be more likely to not miss out on a better solution.

Exercise 9.4. Suppose we have a discrete MDP with state space $\mathcal{S}$ and action space $\mathcal{A}$ and we want to perform forward search to depth d. Due to computational constraints and the requirement that we must simulate to depth d, we decide to generate new, smaller state and action spaces by re-discretizing the original state and action spaces on a coarser scale with $|\mathcal{S}'| < |\mathcal{S}|$ and $|\mathcal{A}'| < |\mathcal{A}|$. In terms of the original state and action spaces, what would the size of the new state and action spaces need to be in order to make the computational complexity of forward search approximately depth-invariant with respect to the size of our original state and action spaces, that is, $O(|\mathcal{S}||\mathcal{A}|)$?

Solution: We need

$$|\mathcal{S}'| = |\mathcal{S}|^{\frac{1}{d}} \qquad \text{and} \qquad |\mathcal{A}'| = |\mathcal{A}|^{\frac{1}{d}}$$

This results in the following complexity:

$$O\left(|\mathcal{S}'|^d |\mathcal{A}'|^d\right) = O\left(\left(|\mathcal{S}|^{\frac{1}{d}}\right)^d \left(|\mathcal{A}|^{\frac{1}{d}}\right)^d\right) = O(|\mathcal{S}||\mathcal{A}|)$$

Exercise 9.5. Building on the previous exercise, suppose now that we want to keep all the original actions in our action space and only re-discretize the state space. What would the size of the new state space need to be to make the computational complexity of forward search approximately depth-invariant with respect to the size of our original state and action spaces?

Solution: The computational complexity of forward search is given by $O\left((|\mathcal{S}||\mathcal{A}|)^d\right)$, which can also be written as $O\left(|\mathcal{S}|^d |\mathcal{A}|^d\right)$. Thus, in order for our coarser state space to lead to forward search that is approximately depth-invariant with respect to the size of our original state and action spaces, we need

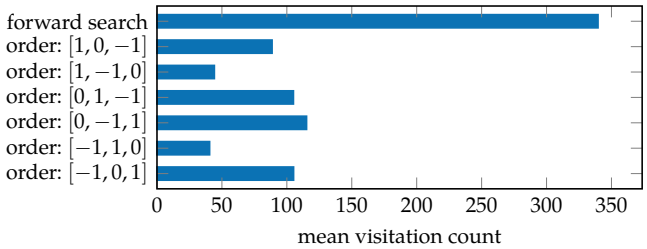
$$|\mathcal{S}'| = \left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}}\right)^{\frac{1}{d}}$$

This gives us:

$$O\left(|\mathcal{S}'|^d |\mathcal{A}'|^d\right) = O\left(\left[\left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}}\right)^{\frac{1}{d}}\right]^d |\mathcal{A}|^d\right) = O\left(|\mathcal{S}| \frac{|\mathcal{A}|^d}{|\mathcal{A}|^{d-1}}\right) = O(|\mathcal{S}||\mathcal{A}|)$$

Exercise 9.6. Will changing the ordering of the action space cause forward search to take different actions? Will changing the ordering of the action space cause branch and bound to take different actions? Can the ordering of the action space affect how many states are visited by branch and bound?

Solution: Forward search enumerates over all possible future actions. It may return different actions if there are ties in their expected utilities. Branch and bound maintains the same optimality guarantee over the horizon as forward search by sorting by upper bound. The ordering of the action space can affect branch and bound's visitation rate when the upper bound produces the same expected value for two or more actions. Below we show this effect on the modified mountain car problem from example 9.2. The plot compares the number of states visited in forward search to that of branch and bound for different action orderings to depth 6. Branch and bound consistently visits far fewer states than forward search, but action ordering can still affect state visitation.



Exercise 9.7. Is sparse sampling with $m = |\mathcal{S}|$ equivalent to forward search?

Solution: No. While the computational complexities are identical at $O\left(|\mathcal{S}|^d |\mathcal{A}|^d\right)$, forward search will branch on all states in the state space, while sparse sampling will branch on $|\mathcal{S}|$ randomly sampled states.

Exercise 9.8. Given an MDP with $|\mathcal{S}| = 10$, $|\mathcal{A}| = 3$, and a uniform transition distribution $T(s' \mid s, a) = 1/|\mathcal{S}|$ for all s and a, what is the probability that sparse sampling with $m = |\mathcal{S}|$ samples and depth $d = 1$ yields the exact same search tree produced by forward search with depth $d = 1$?

Solution: For both forward search and sparse sampling, we branch on all actions from the current state node. For forward search, at each of these action nodes, we branch on all states, while for sparse sampling, we will branch on $m = |\mathcal{S}|$ sampled states. If these sampled states are exactly equal to the state space, that action branch is equivalent to the branch produced in forward search. Thus, for a single action branch we have:

$$\begin{aligned}
&\text{the probability the first state is unique} && \frac{10}{10} \\
&\text{the probability the second state is unique (not equal to the first state)} && \frac{9}{10} \\
&\text{the probability the third state is unique (not equal to the first or second state)} && \frac{8}{10} \\
&\vdots && \vdots
\end{aligned}$$

Since each of these sampled states is independent, this leads to the probability of all unique states in the state space being selected with probability

$$\frac{10 \times 9 \times 8 \times \cdots}{10 \times 10 \times 10 \times \cdots} = \frac{10!}{10^{10}} \approx 0.000363$$

Since each of the sampled states across different action branches is independent, the probability that all three action branches sample the unique states in the state space is

$$\left(\frac{10!}{10^{10}}\right)^3 \approx (0.000363)^3 \approx 4.78 \times 10^{-11}$$

Exercise 9.9. Given the following tables of $Q(s,a)$ and $N(s,a)$, use the upper confidence bound in equation (9.1) to compute the MCTS traversal action for each state with an exploration parameter of $c_1 = 10$ and again for $c_2 = 20$.

	$Q(s,a_1)$	$Q(s,a_2)$
s_1	10	-5
s_2	12	10

	$N(s,a_1)$	$N(s,a_2)$
s_1	27	4
s_2	32	18

Solution: For the first exploration parameter $c_1 = 10$, we tabulate the upper confidence bound of each state-action pair and select the action maximizing the bound for each state:

	$UCB(s,a_1)$	$UCB(s,a_2)$	$\arg\max_a UCB(s,a)$
s_1	$10 + 10\sqrt{\frac{\log 31}{27}} \approx 13.566$	$-5 + 10\sqrt{\frac{\log 31}{4}} \approx 4.266$	a_1
s_2	$12 + 10\sqrt{\frac{\log 50}{32}} \approx 15.496$	$10 + 10\sqrt{\frac{\log 50}{18}} \approx 14.662$	a_1

And for $c_2 = 20$, we have:

	$UCB(s,a_1)$	$UCB(s,a_2)$	$\arg\max_a UCB(s,a)$
s_1	$10 + 20\sqrt{\frac{\log 31}{27}} \approx 17.133$	$-5 + 20\sqrt{\frac{\log 31}{4}} \approx 13.531$	a_1
s_2	$12 + 20\sqrt{\frac{\log 50}{32}} \approx 18.993$	$10 + 20\sqrt{\frac{\log 50}{18}} \approx 19.324$	a_2

10 Policy Search

Policy search involves searching the space of policies without directly computing a value function. The policy space is often lower-dimensional than the state space and can often be searched more efficiently. Policy optimization optimizes the parameters in a *parameterized policy* in order to maximize utility. This parameterized policy can take many forms, such as neural networks, decision trees, and computer programs. This chapter begins by discussing a way to estimate the value of a policy given an initial state distribution. We will then discuss search methods that do not use estimates of the gradient of the policy, saving gradient methods for the next chapter. Although local search can be quite effective in practice, we will also discuss a few alternative optimization approaches that can avoid local optima.[1]

[1] There are many other optimization approaches, as discussed by M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

10.1 Approximate Policy Evaluation

As introduced in section 7.2, we can compute the expected discounted return when following a policy π from a state s. This expected discounted return $U^\pi(s)$ can be computed iteratively (algorithm 7.3) or through matrix operations (algorithm 7.4) when the state space is discrete and relatively small. We can use these results to compute the expected discounted return of π:

$$U(\pi) = \sum_s b(s) U^\pi(s) \tag{10.1}$$

assuming an *initial state distribution* $b(s)$.

We will use this definition of $U(\pi)$ throughout this chapter. However, we often cannot compute $U(\pi)$ exactly when the state space is large or continuous. Instead, we can approximate $U(\pi)$ by sampling *trajectories*, consisting of states, actions,

and rewards when following π. The definition of $U(\pi)$ can be rewritten as

$$U(\pi) = \mathbb{E}_\tau[R(\tau)] = \int p_\pi(\tau) R(\tau) \, \mathrm{d}\tau \tag{10.2}$$

where $p_\pi(\tau)$ is the probability density associated with trajectory τ when following policy π, starting from initial state distribution b. The *trajectory reward* $R(\tau)$ is the discounted return associated with τ. Figure 10.1 illustrates the computation of $U(\pi)$ in terms of trajectories sampled from an initial state distribution.

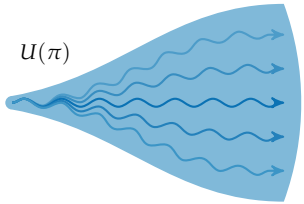


Figure 10.1. The utility associated with a policy from an initial state distribution is computed from the return associated with all possible trajectories under the given policy, weighted according to their likelihood.

Monte Carlo policy evaluation (algorithm 10.1) involves approximating equation (10.2) with m trajectory rollouts of π:

$$U(\pi) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \tag{10.3}$$

where $\tau^{(i)}$ is the ith trajectory sample.

```
struct MonteCarloPolicyEvaluation
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
end

function (U::MonteCarloPolicyEvaluation)(π)
    R(π) = rollout(U.𝒫, rand(U.b), π, U.d)
    return mean(R(π) for i = 1:U.m)
end

(U::MonteCarloPolicyEvaluation)(π, θ) = U(s->π(θ, s))
```

Algorithm 10.1. Monte Carlo policy evaluation of a policy `π`. The method runs `m` rollouts to depth `d` according to the dynamics specified by the problem `𝒫`. Each rollout is run from an initial state sampled from state distribution `b`. The final line in this algorithm block evaluates a policy `π` parameterized by `θ`, which will be useful in the algorithms in this chapter that attempt to find a value of `θ` that maximizes `U`.

Monte Carlo policy evaluation is stochastic. Multiple evaluations of equation (10.1) with the same policy can give different estimates. Increasing the number of rollouts decreases the variance of the evaluation, as demonstrated in figure 10.2.

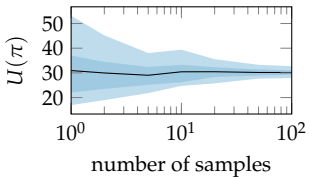


Figure 10.2. The effect of the depth and sample count for Monte Carlo policy evaluation of a uniform random policy on the cart-pole problem (appendix F.3). The variance decreases as the number of samples increases. The blue regions indicate the 5 % to 95 % and 25 % to 75 % empirical quantiles of $U(\pi)$.

We will use π_θ to denote a policy parameterized by θ. For convenience, we will use $U(\theta)$ as shorthand for $U(\pi_\theta)$ in cases where it is not ambiguous. The parameter θ may be a vector or some other more complex representation. For example, we may want to represent our policy using a neural network with a particular structure. We would use θ to represent the weights in the network. Many optimization algorithms assume that θ is a vector with a fixed number of

components. Other optimization algorithms allow more flexible representations, including representations like decision trees or computational expressions.[2]

[2] We will not be discussing those representations here, but some are implemented in `ExprOptimization.jl`.

10.2 Local Search

A common approach to optimization is *local search*, where we begin with an initial parameterization and incrementally move from neighbor to neighbor in the search space until convergence occurs. We discussed this type of approach in chapter 5, in the context of optimizing Bayesian network structures with respect to the Bayesian score. Here, we are optimizing policies parameterized by θ. We are trying to find a value of θ that maximizes $U(\theta)$.

There are many local search algorithms, but this section will focus on the *Hooke-Jeeves method* (algorithm 10.2).[3] This algorithm assumes that our policy is parameterized by an n-dimensional vector θ. The algorithm takes a step of size $\pm\alpha$ in each of the coordinate directions from the current θ. These $2n$ points correspond to the neighborhood of θ. If no improvements to the policy are found, then the step size α is decreased by some factor. If an improvement is found, it moves to the best point. The process continues until α drops below some threshold $\epsilon > 0$. An example involving policy optimization is provided in example 10.1, and figure 10.3 illustrates this process.

[3] R. Hooke and T. A. Jeeves, "Direct Search Solution of Numerical and Statistical Problems," *Journal of the ACM (JACM)*, vol. 8, no. 2, pp. 212–229, 1961.

10.3 Genetic Algorithms

A potential issue with local search algorithms like the Hooke-Jeeves method is that the optimization can get stuck in a local optimum. There are a wide variety of approaches that involve maintaining a *population* consisting of samples of points in the parameter space, evaluating them in parallel with respect to our objective, and then recombining them in some way to drive the population toward a global optimum. A *genetic algorithm*[4] is one such approach, which derives inspiration from biological evolution. It is a general optimization method, but it has been successful in the context of optimizing policies. For example, this approach has been used to optimize policies for Atari video games, where the policy parameters correspond to weights in a neural network.[5]

A simple version of this approach (algorithm 10.3) begins with a population of m random parameterizations, $\theta^{(1)}, \ldots, \theta^{(m)}$. We compute $U(\theta^{(i)})$ for each sample

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[5] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," 2017. arXiv: 1712.06567v3. The implementation in this section follows their relatively simple formulation. Their formulation does not include crossover, which is typically used to mix parameterizations across a population.

```
struct HookeJeevesPolicySearch
    θ # initial parameterization
    α # step size
    c # step size reduction factor
    ϵ # termination step size
end

function optimize(M::HookeJeevesPolicySearch, π, U)
    θ, θ′, α, c, ϵ = copy(M.θ), similar(M.θ), M.α, M.c, M.ϵ
    u, n = U(π, θ), length(θ)
    while α > ϵ
        copyto!(θ′, θ)
        best = (i=0, sgn=0, u=u)
        for i in 1:n
            for sgn in (-1,1)
                θ′[i] = θ[i] + sgn*α
                u′ = U(π, θ′)
                if u′ > best.u
                    best = (i=i, sgn=sgn, u=u′)
                end
            end
            θ′[i] = θ[i]
        end
        if best.i != 0
            θ[best.i] += best.sgn*α
            u = best.u
        else
            α *= c
        end
    end
    return θ
end
```

Algorithm 10.2. Policy search using the Hooke-Jeeves method, which returns a `θ` that has been optimized with respect to `U`. The policy `π` takes as input a parameter `θ` and state `s`. This implementation starts with an initial value of `θ`. The step size `α` is reduced by a factor of `c` if no neighbor improves the objective. Iterations are run until the step size is less than `ϵ`.

Example 10.1. Using a policy optimization algorithm to optimize the parameters of a stochastic policy.

Suppose we want to optimize a policy for the simple regulator problem described in appendix F.5. We define a stochastic policy π parameterized by θ such that the action is generated according to

$$a \sim \mathcal{N}(\theta_1 s, (|\theta_2| + 10^{-5})^2) \tag{10.4}$$

The following code defines the parameterized stochastic policy `π`, evaluation function `U`, and method `M`. It then calls `optimize(M, π, U)`, which returns an optimized value for `θ`. In this case, we use the Hooke-Jeeves method, but the other methods discussed in this chapter can be passed in as `M` instead:

```
function π(θ, s)
    return rand(Normal(θ[1]*s, abs(θ[2]) + 0.00001))
end
b, d, n_rollouts = Normal(0.3,0.1), 10, 3
U = MonteCarloPolicyEvaluation(𝒫, b, d, n_rollouts)
θ, α, c, ϵ = [0.0,1.0], 0.75, 0.75, 0.01
M = HookeJeevesPolicySearch(θ, α, c, ϵ)
θ = optimize(M, π, U)
```

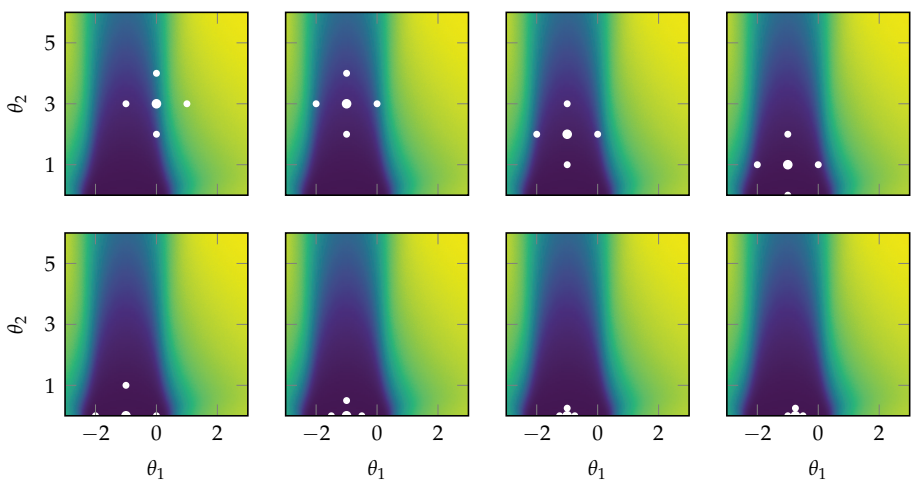


Figure 10.3. The Hooke-Jeeves method applied to optimizing a policy in the simple regulator problem discussed in example 10.1. The evaluations at each iteration are shown as white points. Iterations proceed left to right and top to bottom, and the background is colored according to the expected utility, with yellow indicating lower utility and dark blue indicating higher utility.

i in the population. Since these evaluations potentially involve many rollout simulations and are therefore computationally expensive, they are often run in parallel. These evaluations help us identify the *elite samples*, which are the top m_{elite} samples according to U.

The population at the next iteration is generated by producing $m - 1$ new parameterizations by repeatedly selecting a random elite sample $\boldsymbol{\theta}$ and perturbing it with isotropic Gaussian noise, $\boldsymbol{\theta} + \sigma\boldsymbol{\epsilon}$, where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The best parameterization, unperturbed, is included as the mth sample. Because the evaluations involve stochastic rollouts, a variation of this algorithm could involve running additional rollouts to help identify which of the elite samples is truly the best. Figure 10.4 shows several iterations, or *generations*, of this approach in a sample problem.

```
struct GeneticPolicySearch
    θs      # initial population
    σ       # initial standard deviation
    m_elite # number of elite samples
    k_max   # number of iterations
end

function optimize(M::GeneticPolicySearch, π, U)
    θs, σ = M.θs, M.σ
    n, m = length(first(θs)), length(θs)
    for k in 1:M.k_max
        us = [U(π, θ) for θ in θs]
        sp = sortperm(us, rev=true)
        θ_best = θs[sp[1]]
        rand_elite() = θs[sp[rand(1:M.m_elite)]]
        θs = [rand_elite() + σ.*randn(n) for i in 1:(m-1)]
        push!(θs, θ_best)
    end
    return last(θs)
end
```

Algorithm 10.3. A genetic policy search method for iteratively updating a population of policy parameterizations `θs`, which takes a policy evaluation function `U`, a policy `π(θ, s)`, a perturbation standard deviation `σ`, an elite sample count `m_elite`, and an iteration count `k_max`. The best `m_elite` samples from each iteration are used to generate the samples for the subsequent iteration.

10.4 Cross Entropy Method

The *cross entropy method* (algorithm 10.4) involves updating a *search distribution* over the parameterized space of policies at each iteration.[6] We parameterize this search distribution $p(\theta \mid \psi)$ with ψ.[7] This distribution can belong to any family, but a Gaussian distribution is a common choice, where ψ represents the mean and

[6] S. Mannor, R. Y. Rubinstein, and Y. Gat, "The Cross Entropy Method for Fast Policy Search," in *International Conference on Machine Learning (ICML)*, 2003.

[7] Often, θ and ψ are vectors, but because this assumption is not required for this method, we will not bold them in this section.

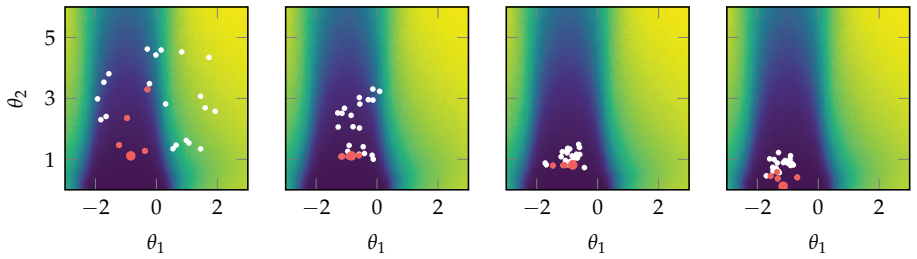


Figure 10.4. Genetic policy search with $\sigma = 0.25$ applied to the simple regulator problem using 25 samples per iteration. The five elite samples in each generation are shown in red, with the best sample indicated by a larger dot.

covariance of the distribution. The objective is to find a value of ψ^* that maximizes the expectation of $U(\theta)$ when θ is drawn from the search distribution:

$$\psi^* = \arg\max_{\psi} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \psi)} [U(\theta)] = \arg\max_{\psi} \int U(\theta) p(\theta \mid \psi) \, \mathrm{d}\theta \tag{10.5}$$

Directly maximizing equation (10.5) is typically computationally infeasible. The approach taken in the cross entropy method is to start with an initial value of ψ, typically chosen so that the distribution is spread over the relevant parameter space. At each iteration, we draw m samples from the associated distribution and then update ψ to fit the elite samples. For the fit, we typically use the maximum likelihood estimate (section 4.1).[8] We stop after a fixed number of iterations, or until the search distribution becomes highly focused on an optimum. Figure 10.5 demonstrates the algorithm on a simple problem.

[8] The maximum likelihood estimate corresponds to the choice of ψ that minimizes the *cross entropy* (see appendix A.9) between the search distribution and the elite samples.

10.5 Evolution Strategies

Evolution strategies[9] update a search distribution parameterized by a vector $\boldsymbol{\psi}$ at each iteration. However, instead of fitting the distribution to a set of elite samples, they update the distribution by taking a step in the direction of the gradient.[10] The gradient of the objective in equation (10.5) can be computed as follows:[11]

[9] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, vol. 15, pp. 949–980, 2014.

[10] We are effectively doing gradient ascent, which is reviewed in appendix A.11.

[11] The policy parameter θ is not bolded here because it is not required to be a vector. However, $\boldsymbol{\psi}$ is in bold because we require it to be a vector when we work with the gradient of the objective.

```
struct CrossEntropyPolicySearch
    p       # initial distribution
    m       # number of samples
    m_elite # number of elite samples
    k_max   # number of iterations
end

function optimize_dist(M::CrossEntropyPolicySearch, π, U)
    p, m, m_elite, k_max = M.p, M.m, M.m_elite, M.k_max
    for k in 1:k_max
        θs = rand(p, m)
        us = [U(π, θs[:,i]) for i in 1:m]
        θ_elite = θs[:,sortperm(us)[(m-m_elite+1):m]]
        p = Distributions.fit(typeof(p), θ_elite)
    end
    return p
end

function optimize(M, π, U)
    return Distributions.mode(optimize_dist(M, π, U))
end
```

Algorithm 10.4. Cross entropy policy search, which iteratively improves a search distribution initially set to `p`. This algorithm takes as input a parameterized policy `π(θ, s)` and a policy evaluation function `U`. In each iteration, `m` samples are drawn and the top `m_elite` are used to refit the distribution. The algorithm terminates after `k_max` iterations. The distribution `p` can be defined using the `Distributions.jl` package. For example, we might define

```
μ = [0.0,1.0]
Σ = [1.0 0.0; 0.0 1.0]
p = MvNormal(μ,Σ)
```

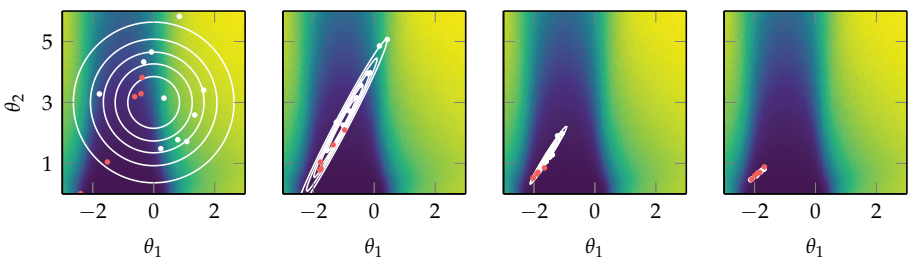


Figure 10.5. The cross entropy method applied to the simple regulator problem using a multivariate Gaussian search distribution. The five elite samples in each iteration are shown in red. The initial distribution is set to $\mathcal{N}([0, 3], 2\mathbf{I})$.

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] = \nabla_{\boldsymbol{\psi}} \int U(\theta) p(\theta \mid \boldsymbol{\psi}) \, \mathrm{d}\theta \tag{10.6}$$

$$= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) \, \mathrm{d}\theta \tag{10.7}$$

$$= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) \frac{p(\theta \mid \boldsymbol{\psi})}{p(\theta \mid \boldsymbol{\psi})} \, \mathrm{d}\theta \tag{10.8}$$

$$= \int \left(U(\theta) \nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})\right) p(\theta \mid \boldsymbol{\psi}) \, \mathrm{d}\theta \tag{10.9}$$

$$= \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}\left[U(\theta) \nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})\right] \tag{10.10}$$

The introduction of the logarithm above comes from what is called the *log derivative trick*, which observes that $\nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi}) = \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) / p(\theta \mid \boldsymbol{\psi})$. This computation requires knowing $\nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})$, but we can often compute this analytically, as discussed in example 10.2.

The search gradient can be estimated from m samples: $\theta^{(1)}, \ldots, \theta^{(m)} \sim p(\cdot \mid \boldsymbol{\psi})$:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] \approx \frac{1}{m} \sum_{i=1}^{m} U(\theta^{(i)}) \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)} \mid \boldsymbol{\psi}) \tag{10.11}$$

This estimate depends on the evaluated expected utility, which itself can vary widely. We can make our gradient estimate more resilient with *rank shaping*, which replaces the utility values with weights based on the relative performance of each sample to the other samples in its iteration. The m samples are sorted in descending order of expected utility. Weight $w^{(i)}$ is assigned to sample i according to a weighting scheme with $w^{(1)} \geq \cdots \geq w^{(m)}$. The search gradient becomes

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] \approx \sum_{i=1}^{m} w^{(i)} \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)} \mid \boldsymbol{\psi}) \tag{10.12}$$

A common weighting scheme is[12]

$$w^{(i)} = \frac{\max\left(0, \log\left(\frac{m}{2} + 1\right) - \log(i)\right)}{\sum_{j=1}^{m} \max\left(0, \log\left(\frac{m}{2} + 1\right) - \log(j)\right)} - \frac{1}{m} \tag{10.13}$$

These weights, shown in figure 10.6, favor better samples and give most samples a small negative weight. Rank-shaping reduces the influence of outliers.

Algorithm 10.5 provides an implementation of the evolution strategies method. Figure 10.7 shows an example of a search progression.

[12] N. Hansen and A. Ostermeier, "Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation," in *IEEE International Conference on Evolutionary Computation*, 1996.

The multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, is a common distribution family. The likelihood in d dimensions takes the form

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{d}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right)$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. The log likelihood is

$$\log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{d}{2} \log(2\pi) - \frac{1}{2} \log |\boldsymbol{\Sigma}| - \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$$

The parameters can be updated using their log likelihood gradients:

$$\nabla_{\boldsymbol{\mu}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$$

$$\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \frac{1}{2} \boldsymbol{\Sigma}^{-1}$$

The term $\nabla_{\boldsymbol{\Sigma}}$ contains the partial derivative of each entry of $\boldsymbol{\Sigma}$ with respect to the log likelihood.

Directly updating $\boldsymbol{\Sigma}$ may not result in a positive definite matrix, as is required for covariance matrices. One solution is to represent $\boldsymbol{\Sigma}$ as a product $\mathbf{A}^\top \mathbf{A}$, which guarantees that $\boldsymbol{\Sigma}$ remains positive semidefinite, and then to update $\mathbf{A}$ instead. Replacing $\boldsymbol{\Sigma}$ by $\mathbf{A}^\top \mathbf{A}$ and taking the gradient with respect to $\mathbf{A}$ yields

$$\nabla_{(\mathbf{A})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{A}) = \mathbf{A} \left[\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})^\top \right]$$

Example 10.2. A derivation of the log likelihood gradient equations for the multivariate Gaussian distribution. For the original derivation and several more sophisticated solutions for handling the positive definite covariance matrix, see D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, vol. 15, pp. 949–980, 2014.

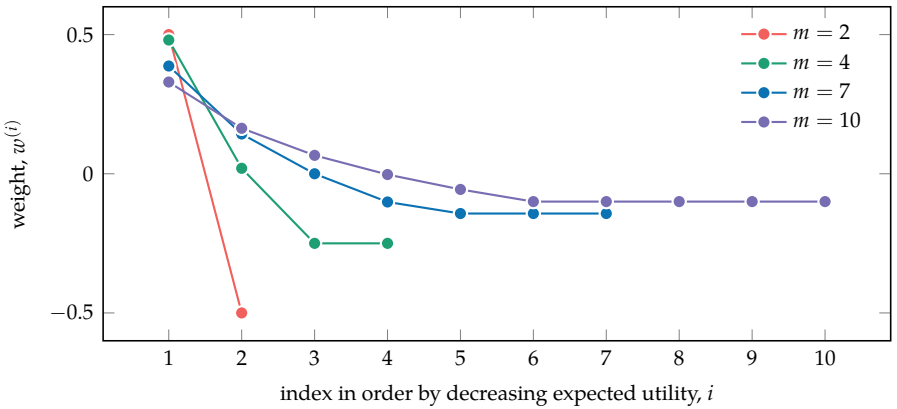


Figure 10.6. Several weightings constructed using equation (10.13).

```
struct EvolutionStrategies
    D       # distribution constructor
    ψ       # initial distribution parameterization
    ∇logp   # log search likelihood gradient
    m       # number of samples
    α       # step factor
    k_max   # number of iterations
end

function evolution_strategy_weights(m)
    ws = [max(0, log(m/2+1) - log(i)) for i in 1:m]
    ws ./= sum(ws)
    ws .-= 1/m
    return ws
end

function optimize_dist(M::EvolutionStrategies, π, U)
    D, ψ, m, ∇logp, α = M.D, M.ψ, M.m, M.∇logp, M.α
    ws = evolution_strategy_weights(m)
    for k in 1:M.k_max
        θs = rand(D(ψ), m)
        us = [U(π, θs[:,i]) for i in 1:m]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*∇logp(ψ, θs[:,i]) for (w,i) in zip(ws,sp))
        ψ += α.*∇
    end
    return D(ψ)
end
```

Algorithm 10.5. An evolution strategies method for updating a search distribution `D(ψ)` over policy parameterizations for policy `π(θ, s)`. This implementation also takes an initial search distribution parameterization `ψ`, the log search likelihood gradient `∇logp(ψ, θ)`, a policy evaluation function `U`, and an iteration count `k_max`. In each iteration, `m` parameterization samples are drawn and are used to estimate the search gradient. This gradient is then applied with a step factor `α`. We can use `Distributions.jl` to define `D(ψ)`. For example, if we want to define `D` to construct a Gaussian with a given mean `ψ` and fixed covariance `Σ`, we can use `D(ψ) = MvNormal(ψ, Σ)`.

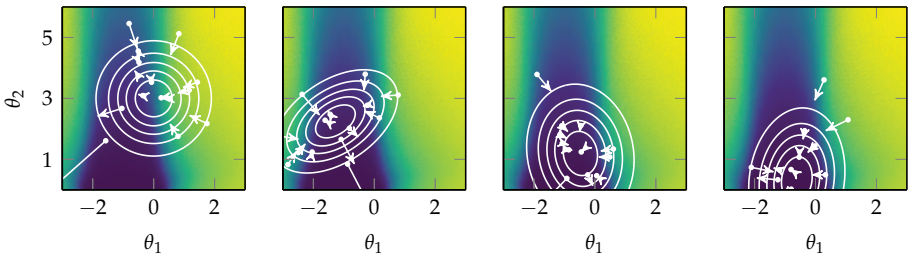


Figure 10.7. Evolution strategies (algorithm 10.5) applied to the simple regulator problem using a multivariate Gaussian search distribution. Samples are shown in white, along with their search gradient contributions, $w\nabla \log p$.

10.6 Isotropic Evolutionary Strategies

The previous section introduced evolutionary strategies that can work with general search distributions. This section will make the assumption that the search distribution is a *spherical* or *isotropic* Gaussian, where the covariance matrix takes the form $\sigma^2\mathbf{I}$.[13] Under this assumption, the expected utility of the distribution introduced in equation (10.5) simplifies to[14]

$$\mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})] = \mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})] \tag{10.14}$$

The search gradient reduces to

$$\begin{aligned}
\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})] &= \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}\left[U(\boldsymbol{\theta})\nabla_{\boldsymbol{\psi}} \log p(\boldsymbol{\theta} \mid \boldsymbol{\psi}, \sigma^2\mathbf{I})\right] && (10.15)\\
&= \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}\left[U(\boldsymbol{\theta})\frac{1}{\sigma^2}(\boldsymbol{\theta}-\boldsymbol{\psi})\right] && (10.16)\\
&= \mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})\frac{1}{\sigma^2}(\sigma\boldsymbol{\epsilon})\right] && (10.17)\\
&= \frac{1}{\sigma}\mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})\boldsymbol{\epsilon}] && (10.18)
\end{aligned}$$

Algorithm 10.6 provides an implementation of this strategy. This implementation incorporates *mirrored sampling*.[15] We sample $m/2$ values from the search distribution and then generate the other $m/2$ samples by mirroring them about the mean. Mirrored samples reduce the variance of the gradient estimate.[16] The benefit of using this technique is shown in figure 10.8.

[13] An example of this approach applied to policy search is explored by T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864v2.

[14] In general, if $\mathbf{A}^\top\mathbf{A} = \boldsymbol{\Sigma}$, then $\boldsymbol{\theta} = \boldsymbol{\mu} + \mathbf{A}^\top\boldsymbol{\epsilon}$ transforms $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ into a sample $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

[15] D. Brockhoff, A. Auger, N. Hansen, D. Arnold, and T. Hohm, "Mirrored Sampling and Sequential Selection for Evolution Strategies," in *International Conference on Parallel Problem Solving from Nature*, 2010.

[16] This technique was implemented by T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864v2. They included other techniques as well, including weight decay.

```
struct IsotropicEvolutionStrategies
    ψ       # initial mean
    σ       # initial standard deviation
    m       # number of samples
    α       # step factor
    k_max   # number of iterations
end

function optimize_dist(M::IsotropicEvolutionStrategies, π, U)
    ψ, σ, m, α, k_max = M.ψ, M.σ, M.m, M.α, M.k_max
    n = length(ψ)
    ws = evolution_strategy_weights(2*div(m,2))
    for k in 1:k_max
        ϵs = [randn(n) for i in 1:div(m,2)]
        append!(ϵs, -ϵs) # weight mirroring
        us = [U(π, ψ + σ.*ϵ) for ϵ in ϵs]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*ϵs[i] for (w,i) in zip(ws,sp)) / σ
        ψ += α.*∇
    end
    return MvNormal(ψ, σ)
end
```

Algorithm 10.6. An evolution strategies method for updating an isotropic multivariate Gaussian search distribution with mean `ψ` and covariance $\sigma^2\mathbf{I}$ over policy parameterizations for a policy `π(θ, s)`. This implementation also takes a policy evaluation function `U`, a step factor `α`, and an iteration count `k_max`. In each iteration, `m/2` parameterization samples are drawn and mirrored and are then used to estimate the search gradient.

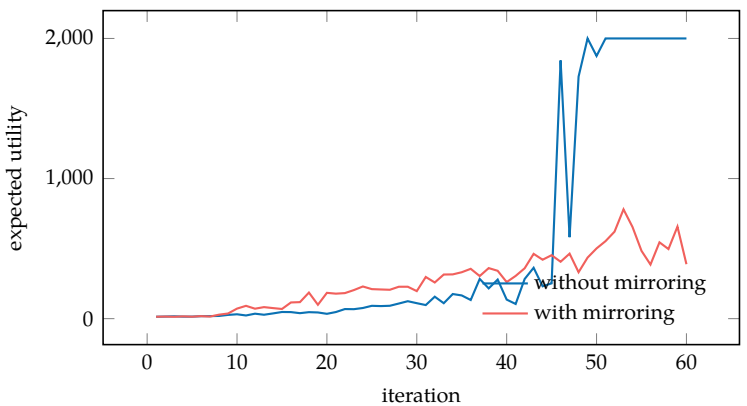


Figure 10.8. A demonstration of the effect that mirrored sampling has on isotropic evolution strategies. Two-layer neural network policies were trained on the cart-pole problem (appendix F.3) using `m = 10`, and `σ = 0.25`, with six rollouts per evaluation. Mirrored sampling significantly speeds and stabilizes learning.

10.7 Summary

- Monte Carlo policy evaluation involves computing the expected utility associated with a policy using a large number of rollouts from states sampled from an initial state distribution.
- Local search methods, such as the Hooke-Jeeves method, improve a policy based on small, local changes.
- Genetic algorithms maintain a population of points in the parameter space, recombining them in different ways in attempt to drive the population toward a global optimum.
- The cross entropy method iteratively improves a search distribution over policy parameters by refitting the distribution to elite samples at each iteration.
- Evolutionary strategies attempt to improve the search distribution using gradient information from samples from that distribution.
- Isotropic evolutionary strategies make the assumption that the search distribution is an isotropic Gaussian.

10.8 Exercises

Exercise 10.1. In Monte Carlo policy evaluation, how is the variance of the utility estimate affected by the number of samples?

Solution: The variance of Monte Carlo policy evaluation is the variance of the mean of m samples. These samples are assumed to be independent, and so the variance of the mean is the variance of a single rollout evaluation divided by the sample size:

$$\text{Var}[\hat{U}(\pi)] = \text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} R(\tau^{(i)})\right] = \frac{1}{m^2}\text{Var}\left[\sum_{i=1}^{m} R(\tau^{(i)})\right] = \frac{1}{m^2}\left(\sum_{i=1}^{m}\text{Var}\left[R(\tau^{(i)})\right]\right) = \frac{1}{m}\text{Var}_\tau[R(\tau)]$$

where $\hat{U}(\pi)$ is the utility from Monte Carlo policy evaluation and $R(\tau)$ is the trajectory reward for a sampled trajectory τ. The sample variance, therefore, decreases with $1/m$.

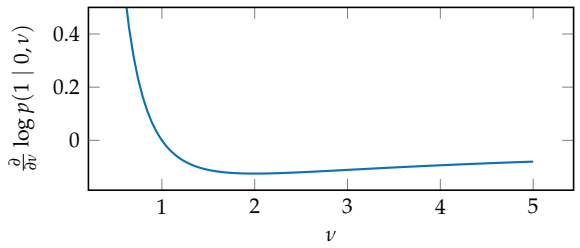
Exercise 10.2. What effect does varying the number of samples m and the number of elite samples m_{elite} have on cross entropy policy search?

Solution: The computational cost per iteration scales linearly with the number of samples. More samples will better cover the search space, resulting in a better chance of identifying better elite samples to improve the policy. The number of elite samples also has an effect. Making all samples elite provides no feedback to the improvement process. Having too few elite samples can lead to premature convergence to a suboptimal solution.

Exercise 10.3. Consider using evolution strategies with a univariate Gaussian distribution, $\theta \sim \mathcal{N}(\mu, \nu)$. What is the search gradient with respect to the variance ν? What issue arises as the variance becomes small?

Solution: The search gradient is the gradient of the log-likelihood:

$$\begin{aligned}\frac{\partial}{\partial \nu} \log p(x \mid \mu, \nu) &= \frac{\partial}{\partial \nu} \log \frac{1}{\sqrt{2\pi\nu}} \exp\left(-\frac{(x-\mu)^2}{2\nu}\right) \\ &= \frac{\partial}{\partial \nu}\left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\nu) - \frac{(x-\mu)^2}{2\nu}\right) \\ &= -\frac{1}{2\nu} + \frac{(x-\mu)^2}{2\nu^2}\end{aligned}$$



We find that the gradient goes to infinity as the variance approaches zero. This is a problem because the variance should be small when the search distribution converges. Very large gradients can cause simple ascent methods to overshoot optima.

Exercise 10.4. Equation (10.14) defines the objective in terms of a search distribution $\theta \sim \mathcal{N}(\psi, \boldsymbol{\Sigma})$. What advantage does this objective have over directly optimizing θ using the expected utility objective in equation (10.1)?

Solution: The added Gaussian noise around the policy parameters can smooth discontinuities in the original objective, which can make optimization more reliable.

Exercise 10.5. Which of the methods in this chapter are best suited to the fact that multiple types of policies could perform well in a given problem?

Solution: The Hooke-Jeeves method improves a single policy parameterization, so it cannot retain multiple policies. Both the cross entropy method and evolution strategies use search distributions. In order to successfully represent multiple types of policies, a multimodal distribution would have to be used. One common multimodal distribution is a mixture of Gaussians. A mixture of Gaussians cannot be fit analytically, but they can be reliably fit using expectation maximization (EM), as demonstrated in example 4.4. Genetic algorithms can retain multiple policies if the population size is sufficiently large.

Exercise 10.6. Suppose we have a parameterized policy π_θ that we would like to optimize using the Hooke-Jeeves method. If we initialize our parameter $\theta = 0$ and the utility function is $U(\theta) = -3\theta^2 + 4\theta + 1$, what is the largest step size α that would still guarantee policy improvement in the first iteration of the Hooke-Jeeves method?

Solution: The Hooke-Jeeves method evaluates the objective function at the center point $\pm\alpha$ along each coordinate direction. In order to guarantee improvement in the first iteration of Hooke-Jeeves search, at least one of the objective function values at the new points must improve the objective function value. For our policy optimization problem, this means that we are searching for the largest step size α such that either $U(\theta + \alpha)$ or $U(\theta - \alpha)$ is greater than $U(\theta)$.

Since the underlying utility function is parabolic and concave, the largest step size that would still lead to improvement is slightly less than the width of the parabola at the current point. Thus, we compute the point on the parabola opposite the current point, θ' at which $U(\theta') = U(\theta)$:

$$\begin{aligned}
U(\theta) &= -3\theta^2 + 4\theta + 1 = -3(0)^2 + 4(0) + 1 = 1 \\
U(\theta) &= U(\theta') \\
1 &= -3\theta'^2 + 4\theta' + 1 \\
0 &= -3\theta'^2 + 4\theta' + 0 \\
\theta' &= \frac{-4 \pm \sqrt{4^2 - 4(-3)(0)}}{2(-3)} = \frac{-4 \pm 4}{-6} = \frac{2 \pm 2}{3} = \left\{0, \tfrac{4}{3}\right\}
\end{aligned}$$

The point on the parabola opposite the current point is thus $\theta' = \frac{4}{3}$. The distance between θ and θ' is $\frac{4}{3} - 0 = \frac{4}{3}$. Thus, the maximal step size we can take and still guarantee improvement in the first iteration is just under $\frac{4}{3}$.

Exercise 10.7. Suppose we have a policy parameterized by a single parameter θ. We take an evolution strategies approach with a search distribution that follows a Bernoulli distribution $p(\theta \mid \psi) = \psi^\theta(1 - \psi)^{1-\theta}$. Compute the log-likelihood gradient $\nabla_\psi \log p(\theta \mid \psi)$.

Solution: The log-likelihood gradient can be computed as follows:

$$\begin{aligned}
p(\theta \mid \psi) &= \psi^\theta (1-\psi)^{1-\theta} \\
\log p(\theta \mid \psi) &= \log\left(\psi^\theta (1-\psi)^{1-\theta}\right) \\
\log p(\theta \mid \psi) &= \theta \log \psi + (1-\theta)\log(1-\psi) \\
\nabla_\psi \log p(\theta \mid \psi) &= \frac{d}{d\psi}\left[\theta \log \psi + (1-\theta)\log(1-\psi)\right] \\
\nabla_\psi \log p(\theta \mid \psi) &= \frac{\theta}{\psi} - \frac{1-\theta}{1-\psi}
\end{aligned}$$

Exercise 10.8. Compute the sample weights for search gradient estimation with rank shaping given $m = 3$ samples.

Solution: We first compute the numerator of the first term from equation (10.13), for all i:

$$\begin{aligned}
i = 1 \qquad & \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 1\right) = \log \tfrac{5}{2} \\
i = 2 \qquad & \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 2\right) = \log \tfrac{5}{4} \\
i = 3 \qquad & \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 3\right) = 0
\end{aligned}$$

Now, we compute the weights:

$$\begin{aligned}
w^{(1)} &= \frac{\log \frac{5}{2}}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = 0.47 \\
w^{(2)} &= \frac{\log \frac{5}{4}}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = -0.14 \\
w^{(3)} &= \frac{0}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = -0.33
\end{aligned}$$

11 Policy Gradient Estimation

The previous chapter discussed several ways to go about directly optimizing the parameters of a policy to maximize expected utility. In many applications, it is often useful to use the gradient of the utility with respect to the policy parameters to guide the optimization process. This chapter discusses several approaches to estimating this gradient from trajectory rollouts.[1] A major challenge with this approach is the variance of the estimate due to the stochastic nature of the trajectories arising from both the environment and our exploration of it. The next chapter will discuss how to use these algorithms to estimate gradients for the purpose of policy optimization.

[1] An additional resource on this topic is M. C. Fu, "Gradient Estimation," in *Simulation*, S. G. Henderson and B. L. Nelson, eds., Elsevier, 2006, pp. 575–616.

11.1 Finite Difference

Finite difference methods estimate the gradient of a function from small changes in its evaluation. Recall that the derivative of a univariate function f is

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{11.1}$$

The derivative at x can be approximated by a sufficiently small step $\delta > 0$:

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) \approx \frac{f(x+\delta) - f(x)}{\delta} \tag{11.2}$$

This approximation is illustrated in figure 11.1.

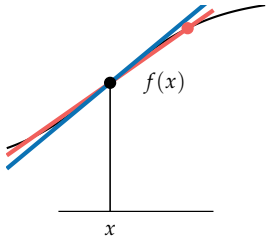


Figure 11.1. The finite difference method approximates the derivative of $f(x)$ using an evaluation of a point near x. The finite-difference approximation, in red, is not a perfect match for the true derivative, in blue.

The gradient of a multivariate function f with an input of length n is

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x})\right] \tag{11.3}$$

Finite differences can be applied to each dimension to estimate the gradient.

In the context of policy optimization, we want to estimate the gradient of the utility expected from following a policy parameterized by θ:

$$\nabla U(\theta) = \left[\frac{\partial U}{\partial \theta_1}(\theta), \ldots, \frac{\partial U}{\partial \theta_n}(\theta)\right] \tag{11.4}$$

$$\approx \left[\frac{U(\theta + \delta \mathbf{e}^{(1)}) - U(\theta)}{\delta}, \ldots, \frac{U(\theta + \delta \mathbf{e}^{(n)}) - U(\theta)}{\delta}\right] \tag{11.5}$$

where $\mathbf{e}^{(i)}$ is the ith *standard basis* vector, consisting of zeros except for the ith component, which is set to 1.

As discussed in section 10.1, we need to simulate policy rollouts to estimate $U(\theta)$. We can use algorithm 11.1 to generate trajectories. From these trajectories, we can compute their return and estimate the utility associated with the policy. Algorithm 11.2 implements the gradient estimate in equation (11.5) by simulating m rollouts for each component and averaging the returns.

```
function simulate(𝒫::MDP, s, π, d)
    τ = []
    for i = 1:d
        a = π(s)
        s′, r = 𝒫.TR(s,a)
        push!(τ, (s,a,r))
        s = s′
    end
    return τ
end
```

Algorithm 11.1. A method for generating a trajectory associated with problem $\mathcal{P}$ starting in state `s` and executing policy π to depth `d`. It creates a vector τ containing state-action-reward tuples.

A major challenge in arriving at accurate estimates of the policy gradient is the fact that the variance of the trajectory rewards can be quite high. One approach to reduce the resulting variance in the gradient estimate is to have each rollout share the same random generator seeds.[2] This approach can be helpful, for example, in cases where one rollout happens to hit a low-probability transition early on. Other rollouts will have the same tendency due to the shared random generator, and their rewards will tend to be biased in the same way.

Policy representations have a significant effect on the policy gradient. Example 11.1 demonstrates the sensitivity of the policy gradient to the policy parameterization. Finite differences for policy optimization can perform poorly when the parameters differ in scale.

[2] This random seed sharing is used in the PEGASUS algorithm. A. Y. Ng and M. Jordan, "A Policy Search Method for Large MDPs and POMDPs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2000.

```
struct FiniteDifferenceGradient
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
    δ # step size
end

function gradient(M::FiniteDifferenceGradient, π, θ)
    𝒫, b, d, m, δ, γ, n = M.𝒫, M.b, M.d, M.m, M.δ, M.𝒫.γ, length(θ)
    Δθ(i) = [i == k ? δ : 0.0 for k in 1:n]
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    U(θ) = mean(R(simulate(𝒫, rand(b), s->π(θ, s), d)) for i in 1:m)
    ΔU = [U(θ + Δθ(i)) - U(θ) for i in 1:n]
    return ΔU ./ δ
end
```

Algorithm 11.2. A method for estimating a policy gradient using finite differences for a problem $\mathcal{P}$, a parameterized policy `π(θ, s)`, and a policy parameterization vector `θ`. Utility estimates are made from `m` rollouts to depth `d`. The step size is given by `δ`.

Consider a single-state, single-step MDP with a one-dimensional continuous action space and a reward function $R(s, a) = a$. In this case, larger actions produce higher rewards.

Suppose we have a stochastic policy π_θ that samples its action according to a uniform distribution between θ_1 and θ_2 for $\theta_2 > \theta_1$. The expected value is

$$U(\theta) = \mathbb{E}[a] = \int_{\theta_1}^{\theta_2} a \frac{1}{\theta_2 - \theta_1} \, \mathrm{d}a = \frac{\theta_1 + \theta_2}{2}$$

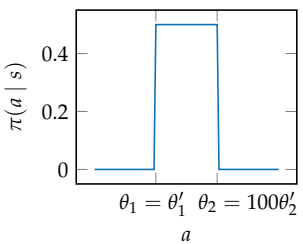
The policy gradient is

$$\nabla U(\theta) = [1/2, 1/2]$$

The policy could be reparameterized to draw actions from a uniform distribution between θ_1' and $100\theta_2'$, for $100\theta_2' > \theta_1'$. Now the expected reward is $(\theta_1' + 100\theta_2')/2$ and the policy gradient is $[1/2, 50]$.

The two parameterizations can represent the same policies, but they have very different gradients. Finding a suitable perturbation scalar for the second policy is much more difficult because the parameters vary widely in scale.

Example 11.1. An example of how policy parameterization has a significant impact on the policy gradient.



11.2 Regression Gradient

Instead of estimating the gradient at θ by taking a fixed step along each coordinate axis, as done in the previous section, we can use *linear regression*[3] to estimate the gradient from the results of random perturbations from θ. These perturbations are stored in a matrix as follows:[4]

$$\Delta\boldsymbol{\Theta} = \begin{bmatrix} (\Delta\theta^{(1)})^\top \\ \vdots \\ (\Delta\theta^{(m)})^\top \end{bmatrix} \tag{11.6}$$

More policy parameter perturbations will tend to produce better gradient estimates.[5]

For each of these perturbations, we perform a rollout and estimate the change in utility:[6]

$$\Delta\mathbf{U} = \left[U(\theta + \Delta\theta^{(1)}) - U(\theta), \ldots, U(\theta + \Delta\theta^{(m)}) - U(\theta)\right] \tag{11.7}$$

The policy gradient estimate using linear regression is then[7]

$$\nabla U(\theta) \approx \Delta\boldsymbol{\Theta}^{+}\Delta\mathbf{U} \tag{11.8}$$

Algorithm 11.3 provides an implementation of this approach in which the perturbations are drawn uniformly from a hypersphere with radius δ. Example 11.2 demonstrates this approach with a simple function.

[3] Linear regression is covered in section 8.6.

[4] This general approach is sometimes referred to as *simultaneous perturbation stochastic approximation* by J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley, 2003. The general connection to linear regression is provided by J. Peters and S. Schaal, "Reinforcement Learning of Motor Skills with Policy Gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

[5] A recommended rule of thumb is to use about twice as many perturbations as the number of parameters.

[6] This equation shows the *forward difference*. Other finite-difference formulations, such as the central difference, can also be used.

[7] As discussed in section 8.6, $\mathbf{X}^{+}$ denotes the pseudoinverse of $\mathbf{X}$.

11.3 Likelihood Ratio

The *likelihood ratio* approach[8] to gradient estimation uses an analytical form of $\nabla\pi_\theta$ to improve our estimate of $\nabla U(\theta)$. Recall from equation (10.2) that

$$U(\theta) = \int p_\theta(\tau) R(\tau)\, \mathrm{d}\tau \tag{11.9}$$

[8] P. W. Glynn, "Likelihood Ratio Gradient Estimation for Stochastic Systems," *Communications of the ACM*, vol. 33, no. 10, pp. 75–84, 1990.

```
struct RegressionGradient
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
    δ # step size
end

function gradient(M::RegressionGradient, π, θ)
    𝒫, b, d, m, δ, γ = M.𝒫, M.b, M.d, M.m, M.δ, M.𝒫.γ
    Δθ = [δ.*normalize(randn(length(θ)), 2) for i = 1:m]
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    U(θ) = R(simulate(𝒫, rand(b), s->π(θ,s), d))
    ΔU = [U(θ + Δθ) - U(θ) for Δθ in Δθ]
    return pinv(reduce(hcat, Δθ)') * ΔU
end
```

Algorithm 11.3. A method for estimating a policy gradient using finite differences for an MDP `𝒫`, a stochastic parameterized policy `π(θ, s)`, and a policy parameterization vector `θ`. Policy variation vectors are generated by normalizing normally distributed samples and scaling by a perturbation scalar `δ`. A total of `m` parameter perturbations are generated, and each is evaluated in a rollout from an initial state drawn from `b` to depth `d` and compared to the original policy parameterization.

Hence,

$$\nabla U(\theta) = \nabla_\theta \int p_\theta(\tau) R(\tau) \, \mathrm{d}\tau \tag{11.10}$$

$$= \int \nabla_\theta p_\theta(\tau) R(\tau) \, \mathrm{d}\tau \tag{11.11}$$

$$= \int p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} R(\tau) \, \mathrm{d}\tau \tag{11.12}$$

$$= \mathbb{E}_\tau \left[\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} R(\tau) \right] \tag{11.13}$$

The name for this method comes from this trajectory likelihood ratio. This likelihood ratio can be seen as a weight in likelihood weighted sampling (section 3.7) over trajectory rewards.

Applying the log derivative trick,[9] we have

$$\nabla U(\theta) = \mathbb{E}_\tau[\nabla_\theta \log p_\theta(\tau) R(\tau)] \tag{11.14}$$

We can estimate this expectation using trajectory rollouts. For each trajectory τ, we need to compute the product $\nabla_\theta \log p_\theta(\tau) R(\tau)$. Recall that $R(\tau)$ is the return associated with trajectory τ. If we have a stochastic policy,[10] the gradient $\nabla_\theta \log p_\theta(\tau)$ is

$$\nabla_\theta \log p_\theta(\tau) = \sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \tag{11.15}$$

[9] The log derivative trick was introduced in section 10.5. It uses the following equality:

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta p_\theta(\tau) / p_\theta(\tau)$$

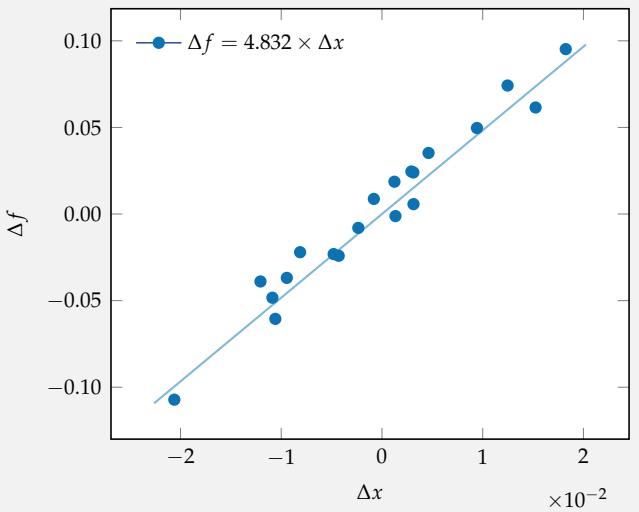
[10] We use $\pi_\theta(a \mid s)$ to represent the probability (either density or mass) the policy π_θ assigns to taking action a from state s.

Example 11.2. Using the regression gradient method on a one-dimensional function.

We would like to apply the regression gradient to estimate the gradient of a simple, one-dimensional function $f(x) = x^2$, evaluated at $x_0 = 2$ from $m = 20$ samples. To imitate the stochasticity inherent in policy evaluation, we add noise to the function evaluations. We generate a set of disturbances ΔX, sampled from $\mathcal{N}(0, \delta^2)$, and evaluate $f(x_0 + \Delta x) - f(x_0)$ for each disturbance Δx in ΔX. We can then estimate the one-dimensional gradient (or derivative) $\Delta X^+ \Delta F$ with this code:

```
f(x) = x^2 + 1e-2*randn()
m = 20
δ = 1e-2
ΔX = [δ.*randn() for i = 1:m]
x0 = 2.0
ΔF = [f(x0 + Δx) - f(x0) for Δx in ΔX]
pinv(ΔX) * ΔF
```

The samples and linear regression are shown here. The slope of the regression line is close to the exact solution of 4:



because $p_\theta(\tau)$ takes the form

$$p_\theta(\tau) = p(s^{(1)}) \prod_{k=1}^{d} T(s^{(k+1)} \mid s^{(k)}, a^{(k)}) \pi_\theta(a^{(k)} \mid s^{(k)}) \tag{11.16}$$

where $s^{(k)}$ and $a^{(k)}$ are the kth state and action, respectively, in trajectory τ. Algorithm 11.4 provides an implementation in which m trajectories are sampled to arrive at a gradient estimate. Example 11.3 illustrates the process.

If we have a deterministic policy, the gradient requires computing:[11]

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \log \left[p(s^{(1)}) \prod_{k=1}^{d} T(s^{(k+1)} \mid s^{(k)}, \pi_\theta(s^{(k)})) \right] \tag{11.17}$$

$$= \sum_{k=1}^{d} \nabla_\theta \pi_\theta(s^{(k)}) \frac{\partial}{\partial a^{(k)}} \log T(s^{(k+1)} \mid s^{(k)}, a^{(k)}) \tag{11.18}$$

Equations (11.17) and (11.18) require knowing the transition likelihood, which is in contrast with equation (11.15) for stochastic policies.

[11] Many problems have vector-valued actions $\mathbf{a} \in \mathbb{R}^n$. In this case, $\nabla_\theta \pi_\theta(s^{(k)})$ is replaced with a Jacobian matrix whose jth column is the gradient with respect to the jth action component, and the $\frac{\partial}{\partial a^{(k)}} \log T(s^{(k+1)} \mid s^{(k)}, a^{(k)})$ is replaced with an action gradient.

```
struct LikelihoodRatioGradient
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
    ∇logπ # gradient of log likelihood
end

function gradient(M::LikelihoodRatioGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇U(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)*R(τ)
    return mean(∇U(simulate(𝒫, rand(b), πθ, d)) for i in 1:m)
end
```

Algorithm 11.4. A method for estimating a policy gradient of a policy `π(s)` for an MDP `𝒫` with initial state distribution `b` using the likelihood ratio trick. The gradient with respect to the parameterization vector `θ` is estimated from `m` rollouts to depth `d` using the log policy gradients `∇logπ`.

11.4 Reward-to-Go

The likelihood ratio policy gradient method is unbiased but has high variance. Example 11.4 reviews bias and variance. The variance generally increases significantly with rollout depth due to the correlation between actions, states, and

Example 11.3. Applying the likelihood ratio trick to estimate a policy gradient in a simple problem.

Consider the single-step, single-state problem from example 11.1. Suppose we have a stochastic policy π_θ that samples its action according to a Gaussian distribution $\mathcal{N}(\theta_1, \theta_2^2)$, where θ_2^2 is the variance.

$$\begin{aligned}\log \pi_\theta(a \mid s) &= \log\left(\frac{1}{\sqrt{2\pi\theta_2^2}} \exp\left(-\frac{(a-\theta_1)^2}{2\theta_2^2}\right)\right) \\ &= -\frac{(a-\theta_1)^2}{2\theta_2^2} - \frac{1}{2}\log\left(2\pi\theta_2^2\right)\end{aligned}$$

The gradient of the log policy likelihood is

$$\begin{aligned}\frac{\partial}{\partial\theta_1}\log \pi_\theta(a \mid s) &= \frac{a-\theta_1}{\theta_2^2} \\ \frac{\partial}{\partial\theta_2}\log \pi_\theta(a \mid s) &= \frac{(a-\theta_1)^2 - \theta_2^2}{\theta_2^3}\end{aligned}$$

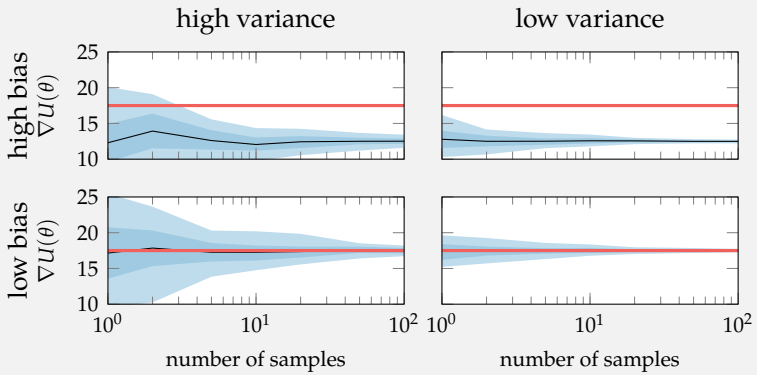
Suppose we run three rollouts with $\theta = [0, 1]$, taking actions $\{0.5, -1, 0.7\}$ and receiving the same rewards ($R(s, a) = a$). The estimated policy gradient is

$$\begin{aligned}\nabla U(\theta) &\approx \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log p_\theta(\tau^{(i)}) R(\tau^{(i)}) \\ &= \frac{1}{3}\left(\begin{bmatrix}0.5\\-0.75\end{bmatrix} 0.5 + \begin{bmatrix}-1.0\\0.0\end{bmatrix}(-1) + \begin{bmatrix}0.7\\-0.51\end{bmatrix} 0.7\right) \\ &= [0.58, -0.244]\end{aligned}$$

Example 11.4. An empirical demonstration of bias and variance when estimating $\nabla U(\theta)$.

When estimating a quantity of interest from a collection of simulations, we generally want to use a scheme that has both low *bias* and low *variance*. In this chapter, we want to estimate $\nabla U(\theta)$. Generally, with more simulation samples, we can arrive at a better estimate. Some methods can lead to bias, where—even with infinitely many samples—it does not lead to an accurate estimate. Sometimes methods with nonzero bias may still be attractive if they also have low variance, meaning that they require fewer samples to converge.

Here are plots of the estimates from four notional methods for estimating $\nabla U(\theta)$. The true value is 17.5, as indicated by the red lines. We ran 100 simulations 100 times for each method. The variance decreases as the number of samples increases. The blue regions indicate the 5 % to 95 % and 25 % to 75 % empirical quantiles of the estimates.



rewards across time steps. The *reward-to-go* approach attempts to reduce the variance in the estimate.

To derive this approach, we begin by expanding equation (11.14):

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\left(\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\right)\left(\sum_{k=1}^{d} r^{(k)}\gamma^{k-1}\right)\right] \tag{11.19}$$

Let $f^{(k)}$ replace $\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})$ for convenience. We then expand as follows:

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\left(\sum_{k=1}^{d} f^{(k)}\right)\left(\sum_{k=1}^{d} r^{(k)}\gamma^{k-1}\right)\right] \tag{11.20}$$

$$= \mathbb{E}_\tau\left[\left(f^{(1)} + f^{(2)} + f^{(3)} + \cdots + f^{(d)}\right)\left(r^{(1)} + r^{(2)}\gamma + r^{(3)}\gamma^2 + \cdots + r^{(d)}\gamma^{d-1}\right)\right] \tag{11.21}$$

$$= \mathbb{E}_\tau\begin{bmatrix} f^{(1)}r^{(1)} + f^{(1)}r^{(2)}\gamma + f^{(1)}r^{(3)}\gamma^2 + \cdots + f^{(1)}r^{(d)}\gamma^{d-1} \\ +f^{(2)}r^{(1)} + f^{(2)}r^{(2)}\gamma + f^{(2)}r^{(3)}\gamma^2 + \cdots + f^{(2)}r^{(d)}\gamma^{d-1} \\ +f^{(3)}r^{(1)} + f^{(3)}r^{(2)}\gamma + f^{(3)}r^{(3)}\gamma^2 + \cdots + f^{(3)}r^{(d)}\gamma^{d-1} \\ \vdots \\ +f^{(d)}r^{(1)} + f^{(d)}r^{(2)}\gamma + f^{(d)}r^{(3)}\gamma^2 + \cdots + f^{(d)}r^{(d)}\gamma^{d-1} \end{bmatrix} \tag{11.22}$$

The first reward, $r^{(1)}$, is affected only by the first action. Thus, its contribution to the policy gradient should not depend on subsequent time steps. We can remove other such causality-violating terms as follows:[12]

[12] The term $\sum_{\ell=k}^{d} r^{(\ell)}\gamma^{\ell-k}$ is often called the *reward-to-go* from step k.

$$\nabla U(\theta) = \mathbb{E}_\tau\begin{bmatrix} f^{(1)}r^{(1)} + f^{(1)}r^{(2)}\gamma + f^{(1)}r^{(3)}\gamma^2 + \cdots + f^{(1)}r^{(d)}\gamma^{d-1} \\ + f^{(2)}r^{(2)}\gamma + f^{(2)}r^{(3)}\gamma^2 + \cdots + f^{(2)}r^{(d)}\gamma^{d-1} \\ + f^{(3)}r^{(3)}\gamma^2 + \cdots + f^{(3)}r^{(d)}\gamma^{d-1} \\ \vdots \\ + f^{(d)}r^{(d)}\gamma^{d-1} \end{bmatrix} \tag{11.23}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\left(\sum_{\ell=k}^{d} r^{(\ell)}\gamma^{\ell-1}\right)\right] \tag{11.24}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\left(\gamma^{k-1}\sum_{\ell=k}^{d} r^{(\ell)}\gamma^{\ell-k}\right)\right] \tag{11.25}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\gamma^{k-1} r^{(k)}_{\text{to-go}}\right] \tag{11.26}$$

Algorithm 11.5 provides an implementation of this.

Notice that the reward-to-go for a state-action pair (s, a) under a policy parameterized by θ is really an approximation of the state-action value from that state, $Q_\theta(s, a)$. The action value function, if known, can be used to obtain the policy gradient:

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} Q_\theta\left(s^{(k)}, a^{(k)}\right)\right] \tag{11.27}$$

```
struct RewardToGoGradient
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
    ∇logπ # gradient of log likelihood
end

function gradient(M::RewardToGoGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    ∇U(τ) = sum(∇logπ(θ, a, s)*R(τ,j) for (j, (s,a,r)) in enumerate(τ))
    return mean(∇U(simulate(𝒫, rand(b), πθ, d)) for i in 1:m)
end
```

Algorithm 11.5. A method that uses reward-to-go for estimating a policy gradient of a policy `π(s)` for an MDP `𝒫` with initial state distribution `b`. The gradient with respect to the parameterization vector `θ` is estimated from `m` rollouts to depth `d` using the log policy gradient `∇logπ`.

11.5 Baseline Subtraction

We can further build on the approach presented in the previous section by subtracting a *baseline* value from the reward-to-go[13] to reduce the variance of the gradient estimate. This subtraction does not bias the gradient.

[13] We could also subtract a baseline from a state-action value.

We now subtract a baseline $r_{\text{base}}(s^{(k)})$:

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1}\left(r_{\text{to-go}}^{(k)} - r_{\text{base}}(s^{(k)})\right)\right] \tag{11.28}$$

To show that baseline subtraction does not bias the gradient, we first expand:

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} r_{\text{to-go}}^{(k)} - \sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} r_{\text{base}}(s^{(k)})\right] \tag{11.29}$$

The *linearity of expectation* states that $\mathbb{E}[a + b] = \mathbb{E}[a] + \mathbb{E}[b]$, so it is sufficient to prove that equation (11.29) is equivalent to equation (11.26), if for each step k, the expected associated baseline term is $\mathbf{0}$:

$$\mathbb{E}_\tau\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right] = \mathbf{0} \tag{11.30}$$

We begin by converting the expectation into nested expectations, as illustrated in figure 11.2:

$$\mathbb{E}_\tau\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right] = \mathbb{E}_{\tau_{1:k}}\left[\mathbb{E}_{\tau_{k+1:d}}\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right]\right] \tag{11.31}$$

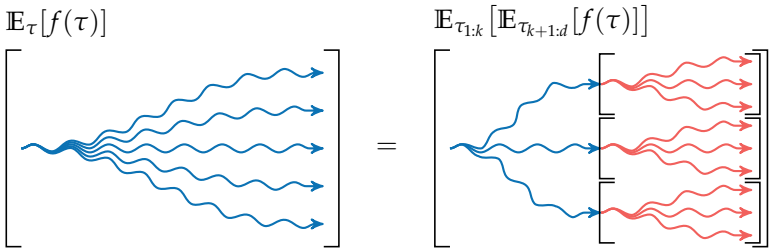


Figure 11.2. The expectation of a function of trajectories sampled from a policy can be viewed as an expectation over a nested expectation of subtrajectories. For a mathematical derivation, see exercise 11.4.

We continue with our derivation, using the same log derivative trick from section 11.3:

$$\begin{aligned}
&\mathbb{E}_{\tau_{1:k}}\left[\mathbb{E}_{\tau_{k+1:d}}\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right]\right] \\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)})\ \mathbb{E}_{\tau_{k+1:d}}\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\right]\right] && (11.32)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)})\ \mathbb{E}_{a^{(k)}}\left[\nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)})\right]\right] && (11.33)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)}) \int \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \pi_\theta(a^{(k)} \mid s^{(k)})\, \mathrm{d}a^{(k)}\right] && (11.34)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)}) \int \frac{\nabla_\theta \pi_\theta(a^{(k)} \mid s^{(k)})}{\pi_\theta(a^{(k)} \mid s^{(k)})} \pi_\theta(a^{(k)} \mid s^{(k)})\, \mathrm{d}a^{(k)}\right] && (11.35)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)})\, \nabla_\theta \int \pi_\theta(a^{(k)} \mid s^{(k)})\, \mathrm{d}a^{(k)}\right] && (11.36)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)})\, \nabla_\theta 1\right] && (11.37)\\
&\quad = \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1} r_{\text{base}}(s^{(k)})\, \mathbf{0}\right] && (11.38)
\end{aligned}$$

Therefore, subtracting a term $r_{\text{base}}(s^{(k)})$ does not bias the estimate. This derivation assumed continuous state and action spaces. The same result applies to discrete spaces.

We can choose a different $r_{\text{base}}(s)$ for every component of the gradient, and we will select them to minimize the variance. For simplicity, we will drop the dependence on s and treat each baseline component as constant.[14] For compactness in writing the equations in our derivation, we define

$$\ell_i(a, s, k) = \gamma^{k-1} \frac{\partial}{\partial \theta_i} \log \pi_\theta(a \mid s) \tag{11.39}$$

[14] Some methods approximate a state-dependent baseline using $r_{\text{base}}(s^{(k)}) = \boldsymbol{\phi}(s^{(k)})^\top \mathbf{w}$. Selecting appropriate baseline functions tends to be difficult. J. Peters and S. Schaal, "Reinforcement Learning of Motor Skills with Policy Gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

The variance of the ith component of our gradient estimate in equation (11.28) is

$$\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\left(\ell_i(a, s, k)\left(r_{\text{to-go}} - r_{\text{base},i}\right)\right)^2\right] - \mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a, s, k)\left(r_{\text{to-go}} - r_{\text{base},i}\right)\right]^2 \tag{11.40}$$

where the expectation is over the $(a, s, r_{\text{to-go}})$ tuples in our trajectory samples, and k is each tuple's depth.

We have just shown that the second term is zero. Hence, we can focus on choosing $r_{\text{base},i}$ to minimize the first term by taking the derivative with respect to the baseline and setting it to zero:

$$\begin{aligned}
&\frac{\partial}{\partial r_{\text{base},i}} \mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\left(\ell_i(a, s, k)\left(r_{\text{to-go}} - r_{\text{base},i}\right)\right)^2\right] \\
&\quad = \frac{\partial}{\partial r_{\text{base},i}}\left(\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a, s, k)^2 r_{\text{to-go}}^2\right] - 2 \mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a, s, k)^2 r_{\text{to-go}} r_{\text{base},i}\right] + r_{\text{base},i}^2 \mathop{\mathbb{E}}_{a,s,k}\left[\ell_i(a, s, k)^2\right]\right) && (11.41) \\
&\quad = -2 \mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a, s, k)^2 r_{\text{to-go}}\right] + 2 r_{\text{base},i} \mathop{\mathbb{E}}_{a,s,k}\left[\ell_i(a, s, k)^2\right] = 0 && (11.42)
\end{aligned}$$

Solving for $r_{\text{base},i}$ yields the baseline component that minimizes the variance:

$$r_{\text{base},i} = \frac{\mathbb{E}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a, s, k)^2 r_{\text{to-go}}\right]}{\mathbb{E}_{a,s,k}\left[\ell_i(a, s, k)^2\right]} \tag{11.43}$$

```
struct BaselineSubtractionGradient
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
    ∇logπ # gradient of log likelihood
end

function gradient(M::BaselineSubtractionGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    ℓ(a, s, k) = ∇logπ(θ, a, s)*γ^(k-1)
    R(τ, k) = sum(r*γ^(j-1) for (j,(s,a,r)) in enumerate(τ[k:end]))
    numer(τ) = sum(ℓ(a,s,k).^2*R(τ,k) for (k,(s,a,r)) in enumerate(τ))
    denom(τ) = sum(ℓ(a,s,k).^2 for (k,(s,a)) in enumerate(τ))
    base(τ) = numer(τ) ./ denom(τ)
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    rbase = mean(base(τ) for τ in trajs)
    ∇U(τ) = sum(ℓ(a,s,k).*(R(τ,k).-rbase) for (k,(s,a,r)) in enumerate(τ))
    return mean(∇U(τ) for τ in trajs)
end
```

Algorithm 11.6. Likelihood ratio gradient estimation with reward-to-go and baseline subtraction for an MDP $\mathcal{P}$, policy π, and initial state distribution b. The gradient with respect to the parameterization vector θ is estimated from m rollouts to depth d using the log policy gradients $\nabla\log\pi$.

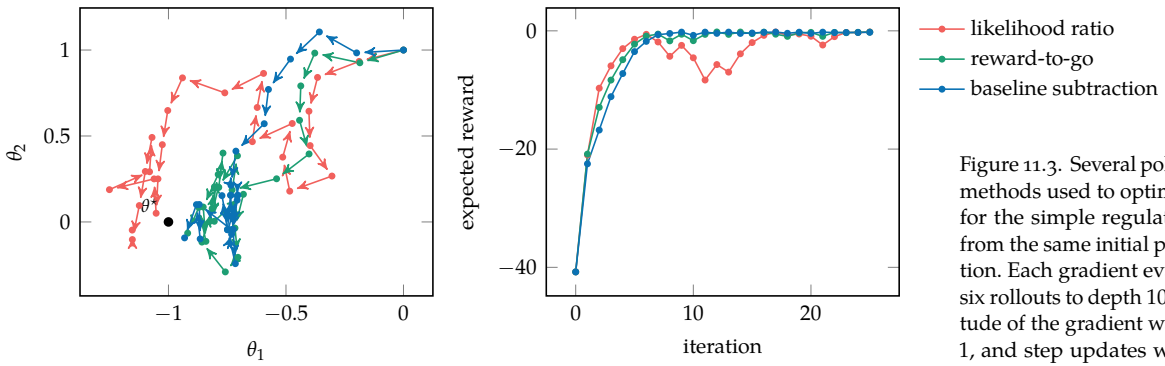


Figure 11.3. Several policy gradient methods used to optimize policies for the simple regulator problem from the same initial parameterization. Each gradient evaluation ran six rollouts to depth 10. The magnitude of the gradient was limited to 1, and step updates were applied with step size 0.2. The optimal policy parameterization is shown in black.

It is common to use likelihood ratio policy gradient estimation with this baseline subtraction (algorithm 11.6).[15]. Figure 11.3 compares the methods discussed here.

[15] This combination is used in the class of algorithms called *REINFORCE* as introduced by R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

Qualitatively, when considering the gradient contribution of state-action pairs, what we really care about is the relative value of one action over another. If all actions in a particular state produce the same high value, there is no real signal in the gradient, and baseline subtraction can zero that out. We want to identify the actions that produce a higher value than others, regardless of the mean value across actions.

An alternative to the action value is the *advantage*, $A(s, a) = Q(s, a) - U(s)$. Using the state value function in baseline subtraction produces the advantage. The policy gradient using the advantage is unbiased and typically has much lower variance. The gradient computation takes the following form:

$$\nabla U(\theta) = \mathbb{E}_\tau \left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} A_\theta\left(s^{(k)}, a^{(k)}\right) \right] \tag{11.44}$$

As with the state and action value functions, the advantage function is typically unknown. Other methods, covered in chapter 13, are needed to approximate it.

11.6 Summary

- A gradient can be estimated using finite differences.
- Linear regression can also be used to provide more robust estimates of the policy gradient.
- The likelihood ratio can be used to derive a form of the policy gradient that does not depend on the transition model for stochastic policies.
- The variance of the policy gradient can be significantly reduced using the reward-to-go and baseline subtraction.

11.7 Exercises

Exercise 11.1. If we estimate the expected discounted return of a given parameterized policy π_θ defined by an n-dimensional vector of parameters θ using m rollouts, how many total rollouts do we need to perform to compute the policy gradient using a finite difference approach?

Solution: In order to estimate the policy gradient using a finite difference approach, we need to estimate the utility of the policy given the current parameter vector $U(\theta)$, as well as all n variations of the current parameter vector $U(\theta + \delta \mathbf{e}^{(i)})$ for $i = 1:n$. Since we estimate each of these using m rollouts, we need to perform a total of $m(n+1)$ rollouts.

Exercise 11.2. Suppose we have a robotic arm with which we are able to run experiments manipulating a wide variety of objects. We would like to use the likelihood ratio policy gradient or one of its extensions to train a policy that is efficient at picking up and moving these objects. Would it be more straightforward to use a deterministic or a stochastic policy, and why?

Solution: The likelihood ratio policy gradient requires an explicit representation of the transition likelihood when used with deterministic policies. Specifying an accurate explicit transition model for a real-world robotic arm manipulation task would be challenging. Computing the policy gradient for a stochastic policy does not require having an explicit representation of the transition likelihood, making the use of a stochastic policy more straightforward.

Exercise 11.3. Consider policy gradients of the form

$$\nabla_\theta U(\theta) = \mathbb{E}_\tau \left[\sum_{k=1}^{d} \gamma^{k-1} y \, \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \right]$$

Which of the following values of y result in a valid policy gradient? Explain why.

(a) $\gamma^{1-k} \sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}$

(b) $\sum_{\ell=k}^{\infty} r^{(\ell)} \gamma^{\ell-k}$

(c) $\left(\sum_{\ell=k}^{\infty} r^{(\ell)} \gamma^{\ell-k} \right) - r_{\text{base}}(s^{(k)})$

(d) $U(s^{(k)})$

(e) $Q(s^{(k)}, a^{(k)})$

(f) $A(s^{(k)}, a^{(k)})$

(g) $r^{(k)} + \gamma U(s^{(k+1)}) - U(s^{(k)})$

Solution:

(a) $\sum_{\ell=1}^{\infty} r^{(\ell)}$ results in the total discounted reward, as

$$\gamma^{k-1}\gamma^{1-k}\sum_{\ell=1}^{\infty} r^{(\ell)}\gamma^{\ell-1} = \sum_{\ell=1}^{\infty} r^{(\ell)}\gamma^{\ell-1}$$

and produces a valid policy gradient, as given in equation (11.19).

(b) $\sum_{\ell=k}^{\infty} r^{(\ell)}\gamma^{\ell-k}$ is the reward-to-go and produces a valid policy gradient, as given in equation (11.26).

(c) $\left(\sum_{\ell=k}^{\infty} r^{(\ell)}\right) - r_{\text{base}}(s^{(k)})$ is the baseline subtracted reward-to-go and produces a valid policy gradient, as given in equation (11.28).

(d) $U(s^{(k)})$ is the state value function and does not produce a valid policy gradient.

(e) $Q(s^{(k)}, a^{(k)})$ is the state-action value function and produces a valid policy gradient, as given in equation (11.27).

(f) $A(s^{(k)}, a^{(k)})$ is the advantage function and produces a valid policy gradient, as given in equation (11.44).

(g) $r^{(k)} + \gamma U(s^{(k+1)}) - U(s^{(k)})$ is the temporal difference residual (to be discussed further in chapter 13) and produces a valid policy gradient because it is an unbiased approximation of the advantage function.

Exercise 11.4. Show that $\mathbb{E}_{\tau\sim\pi}[f(\tau)] = \mathbb{E}_{\tau_{1:k}\sim\pi}[\mathbb{E}_{\tau_{k:d}\sim\pi}[f(\tau)]]$ for step k.

Solution: The nested expectations can be proven by writing the expectation in integral form and then converting back:

$$\begin{aligned}
&\mathbb{E}_{\tau\sim\pi}[f(\tau)] = \\
&= \int p(\tau) f(\tau)\,\mathrm{d}\tau \\
&= \int \left(p(s^{(1)})\prod_{k=1}^{d} p(s^{(k+1)} \mid s^{(k)}, a^{(k)})\pi(a^{(k)} \mid s^{(k)})\right) f(\tau)\,\mathrm{d}\tau \\
&= \int\int\int\int\cdots\int \left(p(s^{(1)})\prod_{k=1}^{d} p(s^{(k+1)} \mid s^{(k)}, a^{(k)})\pi(a^{(k)} \mid s^{(k)})\right) f(\tau)\,\mathrm{d}s^{(d)}\cdots\mathrm{d}a^{(2)}\,\mathrm{d}s^{(2)}\,\mathrm{d}a^{(1)}\,\mathrm{d}s^{(1)} \\
&= \mathop{\mathbb{E}}_{\tau_{1:k}\sim\pi}\left[\int\int\int\int\cdots\int \left(\prod_{q=k}^{d} p(s^{(q+1)} \mid s^{(q)}, a^{(q)})\pi(a^{(q)} \mid s^{(q)})\right) f(\tau)\,\mathrm{d}s^{(d)}\cdots\mathrm{d}a^{(k+1)}\,\mathrm{d}s^{(k+1)}\,\mathrm{d}a^{(k)}\,\mathrm{d}s^{(k)}\right] \\
&= \mathbb{E}_{\tau_{1:k}\sim\pi}[\mathbb{E}_{\tau_{k:d}\sim\pi}[f(\tau)]]
\end{aligned}$$

Exercise 11.5. Our implementation of the regression gradient (algorithm 11.3) fits a linear mapping from perturbations to the difference in returns, $U(\theta + \Delta\theta^{(i)}) - U(\theta)$. We evaluate $U(\theta + \Delta\theta^{(i)})$ and $U(\theta)$ for each of the m perturbations, thus reevaluating $U(\theta)$ a total of m times. How might we reallocate the samples in a more effective manner?

Solution: One approach is to evaluate $U(\theta)$ once and use the same value for each perturbation, thereby conducting only $m + 1$ evaluations. Having an accurate estimate of $U(\theta)$ is particularly important for an accurate regression gradient estimate. An alternative is to still compute $U(\theta)$ once, but use m rollouts, thus preserving the total number of rollouts per iteration. This approach uses the same amount of computation as algorithm 11.3, but it can produce a more reliable gradient estimate.

12 Policy Gradient Optimization

We can use estimates of the policy gradient to drive the search of the parameter space toward an optimal policy. The previous chapter outlined methods for estimating this gradient. This chapter explains how to use these estimates to guide the optimization. We begin with gradient ascent, which simply takes steps in the direction of the gradient at each iteration. Determining the step size is a major challenge. Large steps can lead to faster progress to the optimum, but they can overshoot. The natural policy gradient modifies the direction of the gradient to better handle variable levels of sensitivity across parameter components. We conclude with the trust region method, which starts in exactly the same way as the natural gradient method to obtain a candidate policy. It then searches along the line segment in policy space connecting the original policy to this candidate to find a better policy.

12.1 Gradient Ascent Update

We can use *gradient ascent* (reviewed in appendix A.11) to find a policy parameterized by θ that maximizes the expected utility $U(\theta)$. Gradient ascent is a type of *iterated ascent* method, which involves taking steps in the parameter space at each iteration in an attempt to improve the quality of the associated policy. All the methods discussed in this chapter are iterated ascent methods, but they differ in how they take steps. The gradient ascent method discussed in this section takes steps in the direction of $\nabla U(\theta)$, which may be estimated using one of the methods discussed in the previous chapter. The update of θ is

$$\theta \leftarrow \theta + \alpha \nabla U(\theta) \tag{12.1}$$

where the step length is equal to a step factor $\alpha > 0$ times the magnitude of the gradient.

Algorithm 12.1 implements a method that takes such a step. This method can be called for either a fixed number of iterations or until θ or $U(\theta)$ converges. Gradient ascent, as well as the other algorithms discussed in this chapter, is not guaranteed to converge to the optimal policy. However, there are techniques to encourage convergence to a *locally optimal* policy, in which taking an infinitesimally small step in parameter space cannot result in a better policy. One approach is to decay the step factor with each step.[1]

[1] This approach, as well as many others, are covered in detail by M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

```
struct PolicyGradientUpdate
    ∇U # policy gradient estimate
    α  # step factor
end

function update(M::PolicyGradientUpdate, θ)
    return θ + M.α * M.∇U(θ)
end
```

Algorithm 12.1. The gradient ascent method for policy optimization. It takes a step from a point `θ` in the direction of the gradient `∇U` with step factor `α`. We can use one of the methods in the previous chapter to compute `∇U`.

Very large gradients tend to overshoot the optimum and may occur due to a variety of reasons. Rewards for some problems, such as for the 2048 problem (appendix F.2), can vary by orders of magnitude. One approach for keeping the gradients manageable is to use *gradient scaling*, which limits the magnitude of a gradient estimate before using it to update the policy parameterization. Gradients are commonly limited to having an L_2-norm of 1. Another approach is *gradient clipping*, which conducts elementwise clamping of the gradient before using it to update the policy. Clipping commonly limits the entries to lie between ± 1. Both techniques are implemented in algorithm 12.2.

```
scale_gradient(∇, L2_max) = min(L2_max/norm(∇), 1)*∇
clip_gradient(∇, a, b) = clamp.(∇, a, b)
```

Algorithm 12.2. Methods for gradient scaling and clipping. Gradient scaling limits the magnitude of the provided gradient vector `∇` to `L2_max`. Gradient clipping provides elementwise clamping of the provided gradient vector `∇` to between `a` and `b`.

Scaling and clipping differ in how they affect the final gradient direction, as demonstrated in figure 12.1. Scaling will leave the direction unaffected, whereas clipping affects each component individually. Whether this difference is advantageous depends on the problem. For example, if a single component dominates the gradient vector, scaling will zero out the other components.

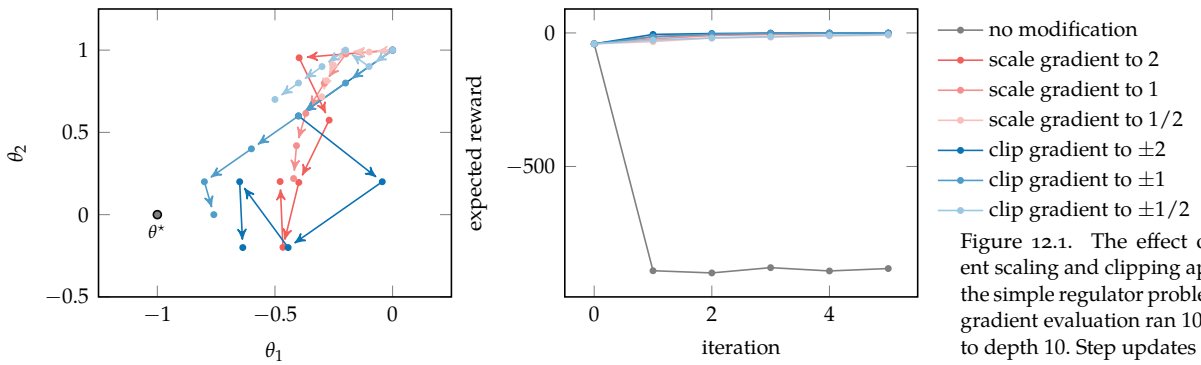


Figure 12.1. The effect of gradient scaling and clipping applied to the simple regulator problem. Each gradient evaluation ran 10 rollouts to depth 10. Step updates were applied with a step size of 0.2. The optimal policy parameterization is shown in black.

12.2 *Restricted Gradient Update*

The remaining algorithms in this chapter attempt to optimize an approximation of the objective function $U(\theta)$, subject to a constraint that the policy parameters at the next step θ' are not too far from θ at the current step. The constraint takes the form $g(\theta, \theta') \leq \epsilon$, where $\epsilon > 0$ is a free parameter in the algorithm. The methods differ in their approximation of $U(\theta)$ and the form of g. This section describes a simple *restricted step* method.

We use the first-order Taylor approximation (appendix A.12) obtained from our gradient estimate at θ to approximate U:

$$U(\theta') \approx U(\theta) + \nabla U(\theta)^\top (\theta' - \theta) \tag{12.2}$$

For the constraint, we use

$$g(\theta, \theta') = \frac{1}{2}(\theta' - \theta)^\top \mathbf{I}(\theta' - \theta) = \frac{1}{2}\|\theta' - \theta\|_2^2 \tag{12.3}$$

We can view this constraint as limiting the step length to no more than $\sqrt{2\epsilon}$. In other words, the feasible region in our optimization is a ball of radius $\sqrt{2\epsilon}$ centered at θ.

The optimization problem is, then,

$$\begin{aligned} \underset{\theta'}{\text{maximize}} \quad & U(\theta) + \nabla U(\theta)^\top (\theta' - \theta) \\ \text{subject to} \quad & \frac{1}{2}(\theta' - \theta)^\top \mathbf{I}(\theta' - \theta) \leq \epsilon \end{aligned} \tag{12.4}$$

We can drop $U(\theta)$ from the objective since it does not depend on θ'. In addition, we can change the inequality to an equality in the constraint because the linear objective forces the optimal solution to be on the boundary of the feasible region. These changes result in an equivalent optimization problem:

$$\begin{aligned} \underset{\theta'}{\text{maximize}} \quad & \nabla U(\theta)^\top (\theta' - \theta) \\ \text{subject to} \quad & \frac{1}{2}(\theta' - \theta)^\top \mathbf{I}(\theta' - \theta) = \epsilon \end{aligned} \tag{12.5}$$

This optimization problem can be solved analytically:

$$\theta' = \theta + \mathbf{u}\sqrt{\frac{2\epsilon}{\mathbf{u}^\top \mathbf{u}}} = \theta + \sqrt{2\epsilon}\frac{\mathbf{u}}{\|\mathbf{u}\|} \tag{12.6}$$

where the unnormalized search direction $\mathbf{u}$ is simply $\nabla U(\theta)$. Of course, we do not know $\nabla U(\theta)$ exactly, but we can use any of the methods described in the previous chapter to estimate it. Algorithm 12.3 provides an implementation.

```
struct RestrictedPolicyUpdate
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    ∇logπ # gradient of log likelihood
    π     # policy
    ϵ     # divergence bound
end

function update(M::RestrictedPolicyUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    u = mean(∇U(τ) for τ in τs)
    return θ + u*sqrt(2*M.ϵ/dot(u,u))
end
```

Algorithm 12.3. The update function for the restricted policy gradient method at `θ` for a problem `𝒫` with initial state distribution `b`. The gradient is estimated from an initial state distribution `b` to depth `d` with `m` simulations of parameterized policy `π(θ, s)` with log policy gradient `∇logπ`.

12.3 Natural Gradient Update

The *natural gradient* method[2] is a variation of the restricted step method discussed in the previous section to better handle situations when some components of the parameter space are more sensitive than others. *Sensitivity* in this context refers to how much the utility of a policy varies with respect to small changes in one of the parameters. The sensitivity in gradient methods is largely determined by the choice of scaling of the policy parameters. The natural policy gradient method makes the search direction $\mathbf{u}$ invariant to parameter scaling. Figure 12.2 illustrates the differences between the true gradient and the natural gradient.

[2] S. Amari, "Natural Gradient Works Efficiently in Learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998.

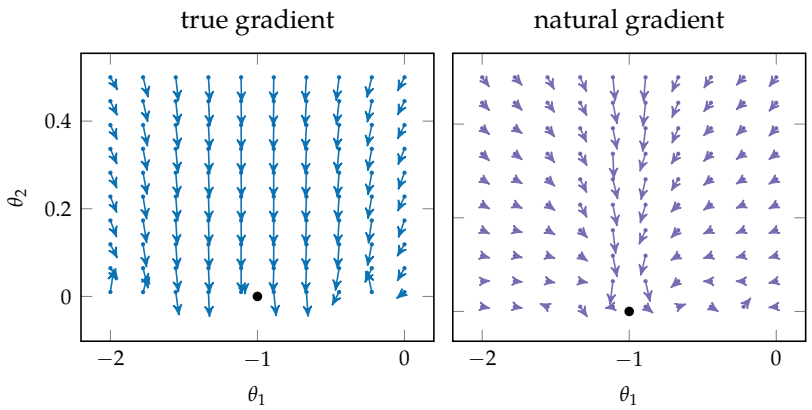


Figure 12.2. A comparison of the true gradient and the natural gradient on the simple regulator problem (see appendix F.5). The true gradient generally points strongly in the negative θ_2 direction, whereas the natural gradient generally points toward the optimum (black dot) at $[-1, 0]$. A similar figure is presented in J. Peters and S. Schaal, "Reinforcement Learning of Motor Skills with Policy Gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

The natural policy gradient method uses the same first-order approximation of the objective as in the previous section. The constraint, however, is different. The intuition is that we want to restrict changes in θ that result in large changes in the distribution over trajectories. A way to measure how much a distribution changes is to use the *Kullback-Leibler divergence*, or KL divergence (appendix A.10). We could impose the constraint

$$g(\theta, \theta') = D_{\text{KL}}\big(p(\cdot \mid \theta) \,||\, p(\cdot \mid \theta')\big) \leq \epsilon \tag{12.7}$$

but instead we will use a second-order Taylor approximation:

$$g(\theta, \theta') = \frac{1}{2}(\theta' - \theta)^\top \mathbf{F}_\theta (\theta' - \theta) \leq \epsilon \tag{12.8}$$

where the *Fisher information matrix* has the following form:

$$\mathbf{F}_\theta = \int p(\tau \mid \theta) \nabla \log p(\tau \mid \theta) \nabla \log p(\tau \mid \theta)^\top \, \mathrm{d}\tau \tag{12.9}$$

$$= \mathbb{E}_\tau \left[\nabla \log p(\tau \mid \theta) \nabla \log p(\tau \mid \theta)^\top \right] \tag{12.10}$$

The resulting optimization problem is

$$\begin{aligned} \underset{\theta'}{\text{maximize}} \quad & \nabla U(\theta)^\top (\theta' - \theta) \\ \text{subject to} \quad & \frac{1}{2} (\theta' - \theta)^\top \mathbf{F}_\theta (\theta' - \theta) = \epsilon \end{aligned} \tag{12.11}$$

which looks identical to equation (12.5) except that instead of the identity matrix $\mathbf{I}$, we have the Fisher matrix $\mathbf{F}_\theta$. This difference results in an ellipsoid feasible set. Figure 12.3 shows an example in two dimensions.

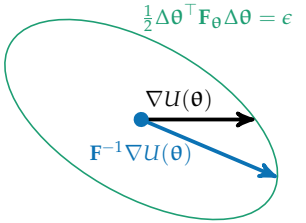


Figure 12.3. The natural policy gradient places a constraint on the approximated Kullback-Leibler divergence. This constraint takes the form of an ellipse. The ellipse may be elongated in certain directions, allowing larger steps if the gradient is rotated.

This optimization problem can be solved analytically and has the same form as the update in the previous section:

$$\theta' = \theta + \mathbf{u} \sqrt{\frac{2\epsilon}{\nabla U(\theta)^\top \mathbf{u}}} \tag{12.12}$$

except that we now have[3]

$$\mathbf{u} = \mathbf{F}_\theta^{-1} \nabla U(\theta) \tag{12.13}$$

We can use sampled trajectories to estimate $\mathbf{F}_\theta$ and $\nabla U(\theta)$. Algorithm 12.4 provides an implementation.

[3] This computation can be done using conjugate gradient descent, which reduces computation when the dimension of θ is large. S. M. Kakade, "A Natural Policy Gradient," in *Advances in Neural Information Processing Systems* (*NIPS*), 2001.

12.4 Trust Region Update

This section discusses a method for searching within the *trust region*, defined by the elliptical feasible region from the previous section. This category of approach is referred to as *trust region policy optimization* (*TRPO*).[4] It works by computing the next evaluation point θ' that would be taken by the natural policy gradient and then conducting a *line search* along the line segment connecting θ to θ'. A key property of this line search phase is that evaluations of the approximate objective and constraint do not require any additional rollout simulations.

[4] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust Region Policy Optimization," in *International Conference on Machine Learning* (*ICML*), 2015.

```
struct NaturalPolicyUpdate
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    ∇logπ # gradient of log likelihood
    π     # policy
    ϵ     # divergence bound
end

function natural_update(θ, ∇f, F, ϵ, τs)
    ∇fθ = mean(∇f(τ) for τ in τs)
    u = mean(F(τ) for τ in τs) \ ∇fθ
    return θ + u*sqrt(2ϵ/dot(∇fθ,u))
end

function update(M::NaturalPolicyUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    F(τ) = ∇log(τ)*∇log(τ)'
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return natural_update(θ, ∇U, F, M.ϵ, τs)
end
```

Algorithm 12.4. The update function for the natural policy gradient, given policy `π(θ, s)`, for an MDP `𝒫` with initial state distribution `b`. The natural gradient with respect to the parameter vector `θ` is estimated from `m` rollouts to depth `d` using the log policy gradients `∇logπ`. The `natural_update` helper method conducts an update according to equation (12.12), given an objective gradient `∇f(τ)` and a Fisher matrix `F(τ)` for a list of trajectories.

During the line search phase, we no longer use a first-order approximation. Instead, we use an approximation derived from an equality involving the advantage function[5]

$$U(\theta') = U(\theta) + \mathop{\mathbb{E}}_{\tau \sim \pi_{\theta'}}\left[\sum_{k=1}^{d} A_\theta(s^{(k)}, a^{(k)})\right] \tag{12.14}$$

[5] A variation of this equality is proven in lemma 6.1 of S. M. Kakade and J. Langford, "Approximately Optimal Approximate Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2002.

Another way to write this is to use $b_{\gamma,\theta}$, which is the *discounted visitation distribution* of state s under policy π_θ, where

$$b_{\gamma,\theta}(s) \propto P(s^{(1)} = s) + \gamma P(s^{(2)} = s) + \gamma^2 P(s^{(3)} = s) + \cdots \tag{12.15}$$

Using the discounted visitation distribution, the objective becomes

$$U(\theta') = U(\theta) + \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta'}}\left[\mathop{\mathbb{E}}_{a \sim \pi_{\theta'}(\cdot|s)}[A_\theta(s,a)]\right] \tag{12.16}$$

We would like to pull our samples from our policy parameterized by θ instead of θ' so that we do not have to run more simulations during the line search. The samples associated with the inner expectation can be replaced with samples from our original policy so long as we appropriately weight the advantage:[6]

$$U(\theta') = U(\theta) + \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta'}}\left[\mathop{\mathbb{E}}_{a \sim \pi_\theta(\cdot|s)}\left[\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} A_\theta(s,a)\right]\right] \tag{12.17}$$

[6] This weighting comes from *importance sampling*, which is reviewed in appendix A.14.

The next step involves replacing the state distribution with $b_{\gamma,\theta}$. The quality of the approximation degrades as θ' gets further from θ, but it is hypothesized that it is acceptable within the trust region. Since $U(\theta)$ does not depend on θ', we can drop it from the objective. We can also drop the state value function from the advantage function, leaving us with the action value function. What remains is referred to as the *surrogate objective*:

$$f(\theta, \theta') = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a \sim \pi_\theta(\cdot|s)}\left[\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} Q_\theta(s,a)\right]\right] \tag{12.18}$$

This equation can be estimated from the same set of trajectories that was used to estimate the natural gradient update. We can estimate $Q_\theta(s,a)$ using the reward-to-go in the sampled trajectories.[7]

[7] Algorithm 12.5 instead uses $\sum_{\ell=k} r^{(\ell)} \gamma^{\ell-1}$, which effectively discounts the reward-to-go by γ^{k-1}. This discount is needed to weight each sample's contribution to match the discounted visitation distribution. The surrogate constraint is similarly discounted.

The *surrogate constraint* in the line search is given by

$$g(\theta, \theta') = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta}}[D_{\mathrm{KL}}(\pi_\theta(\cdot \mid s) \,||\, \pi_{\theta'}(\cdot \mid s))] \leq \epsilon \tag{12.19}$$

Line search involves iteratively evaluating our surrogate objective f and surrogate constraint g for different points in the policy space. We begin with the θ' obtained from the same process as the natural gradient update. We then iteratively apply

$$\theta' \leftarrow \theta + \alpha(\theta' - \theta) \tag{12.20}$$

until we have an improvement in our objective with $f(\theta, \theta') > f(\theta, \theta)$ and our constraint is met with $g(\theta, \theta') \leq \epsilon$. The step factor $0 < \alpha < 1$ shrinks the distance between θ and θ' at each iteration, with α typically set to 0.5.

Algorithm 12.5 provides an implementation of this approach. Figure 12.4 illustrates the relationship between the feasible regions associated with the natural gradient and the line search. Figure 12.5 demonstrates the approach on a regulator problem, and example 12.1 shows an update for a simple problem.

12.5 Clamped Surrogate Objective

We can avoid detrimental policy updates from overly optimistic estimates of the trust region surrogate objective by *clamping*.[8] The surrogate objective from equation (12.18), after exchanging the action value advantage, is

$$\mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} A_\theta(s,a)\right]\right] \tag{12.21}$$

[8] Clamping is a key idea in what is known as *proximal policy optimization* (*PPO*) as discussed by J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017. arXiv: 1707.06347v2.

The probability ratio $\pi_{\theta'}(a \mid s)/\pi_\theta(a \mid s)$ can be overly optimistic. A pessimistic lower bound on the objective can significantly improve performance:

$$\mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\min\left(\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} A_\theta(s,a), \text{clamp}\left(\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)}, 1-\epsilon, 1+\epsilon\right) A_\theta(s,a)\right)\right]\right] \tag{12.22}$$

where ϵ is a small positive value[9] and $\text{clamp}(x, a, b)$ forces x to be between a and b. By definition, $\text{clamp}(x, a, b) = \min\{\max\{x, a\}, b\}$.

[9] While this ϵ does not directly act as a threshold on divergence, as it did in previous algorithms, its role is similar. A typical value is 0.2.

Clamping the probability ratio alone does not produce a lower bound; we must also take the minimum of the clamped and original objectives. The lower bound is shown in figure 12.6, together with the original and clamped objectives. The end result of the lower bound is that the change in probability ratio is ignored when it would cause the objective to improve significantly. Using the lower bound thus prevents large, often detrimental, updates in these situations and removes the need for the trust region surrogate constraint equation (12.19). Without the

```
struct TrustRegionUpdate
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    π     # policy π(s)
    p     # policy likelihood p(θ, a, s)
    ∇logπ # log likelihood gradient
    KL    # KL divergence KL(θ, θ′, s)
    ϵ     # divergence bound
    α     # line search reduction factor (e.g., 0.5)
end

function surrogate_objective(M::TrustRegionUpdate, θ, θ′, τs)
    d, p, γ = M.d, M.p, M.𝒫.γ
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    w(a,s) = p(θ′,a,s) / p(θ,a,s)
    f(τ) = mean(w(a,s)*R(τ,k) for (k,(s,a,r)) in enumerate(τ))
    return mean(f(τ) for τ in τs)
end

function surrogate_constraint(M::TrustRegionUpdate, θ, θ′, τs)
    γ = M.𝒫.γ
    KL(τ) = mean(M.KL(θ, θ′, s)*γ^(k-1) for (k,(s,a,r)) in enumerate(τ))
    return mean(KL(τ) for τ in τs)
end

function linesearch(M::TrustRegionUpdate, f, g, θ, θ′)
    fθ = f(θ)
    while g(θ′) > M.ϵ || f(θ′) ≤ fθ
        θ′ = θ + M.α*(θ′ - θ)
    end
    return θ′
end

function update(M::TrustRegionUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    F(τ) = ∇log(τ)*∇log(τ)'
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    θ′ = natural_update(θ, ∇U, F, M.ϵ, τs)
    f(θ′) = surrogate_objective(M, θ, θ′, τs)
    g(θ′) = surrogate_constraint(M, θ, θ′, τs)
    return linesearch(M, f, g, θ, θ′)
end
```

Algorithm 12.5. The update procedure for trust region policy optimization, which augments the natural gradient with a line search. It generates `m` trajectories using policy `π` in problem `𝒫` with initial state distribution `b` and depth `d`. To obtain the starting point of the line search, we need the gradient of the log-probability of the policy generating a particular action from the current state, which we denote as `∇logπ`. For the surrogate objective, we need the probability function `p`, which gives the probability that our policy generates a particular action from the current state. For the surrogate constraint, we need the divergence between the action distributions generated by π_θ and $\pi_{\theta'}$. At each step of the line search, we shrink the distance between the considered point `θ′` and `θ` while maintaining the search direction.

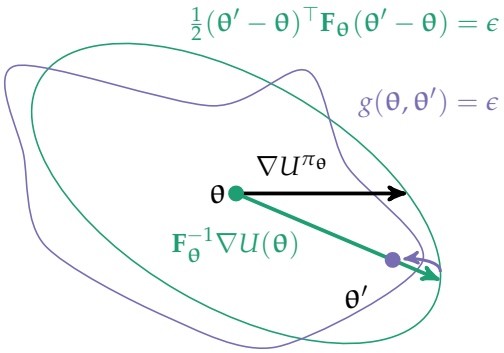


Figure 12.4. Trust region policy optimization searches within the elliptical constraint generated by a second-order approximation of the Kullback-Leibler divergence. After computing the natural policy gradient ascent direction, a line search is conducted to ensure that the updated policy improves the policy reward and adheres to the divergence constraint. The line search starts from the estimated maximum step size and reduces the step size along the ascent direction until a satisfactory point is found.

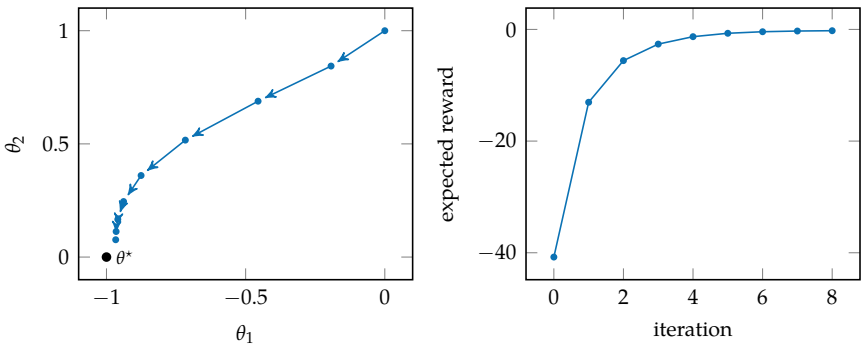


Figure 12.5. Trust region policy optimization applied to the simple regulator problem with rollouts to depth 10 with $\epsilon = 1$ and $c = 2$. The optimal policy parameterization is shown in black.

Example 12.1. An example of one iteration of trust region policy optimization.

Consider applying TRPO to the Gaussian policy $\mathcal{N}(\theta_1, \theta_2^2)$ from example 11.3 to the single-state MDP from example 11.1 with $\gamma = 1$. Recall that the gradient of the log policy likelihood is

$$\frac{\partial}{\partial \theta_1} \log \pi_{\boldsymbol{\theta}}(a \mid s) = \frac{a - \theta_1}{\theta_2^2}$$
$$\frac{\partial}{\partial \theta_2} \log \pi_{\boldsymbol{\theta}}(a \mid s) = \frac{(a - \theta_1)^2 - \theta_2^2}{\theta_2^3}$$

Suppose that we run two rollouts with $\boldsymbol{\theta} = [0, 1]$ (this problem only has one state):

$$\tau_1 = \{(a = r = -0.532), (a = r = \quad 0.597), (a = r = 1.947)\}$$
$$\tau_2 = \{(a = r = -0.263), (a = r = -2.212), (a = r = 2.364)\}$$

The estimated Fisher information matrix is

$$\begin{aligned}\mathbf{F}_{\boldsymbol{\theta}} &= \frac{1}{2}\left(\nabla \log p(\tau^{(1)}) \nabla \log p(\tau^{(1)})^\top + \nabla \log p(\tau^{(2)}) \nabla \log p(\tau^{(2)})^\top\right) \\ &= \frac{1}{2}\left(\begin{bmatrix} 4.048 & 2.878 \\ 2.878 & 2.046 \end{bmatrix} + \begin{bmatrix} 0.012 & -0.838 \\ -0.838 & 57.012 \end{bmatrix}\right) = \begin{bmatrix} 2.030 & 1.020 \\ 1.019 & 29.529 \end{bmatrix}\end{aligned}$$

The objective function gradient is $[2.030, 1.020]$. The resulting descent direction $\mathbf{u}$ is $[1, 0]$. Setting $\epsilon = 0.1$, we compute our updated parameterization vector and obtain $\boldsymbol{\theta}' = [0.314, 1]$.

The surrogate objective function value at $\boldsymbol{\theta}$ is 1.485. Line search begins at $\boldsymbol{\theta}'$, where the surrogate objective function value is 2.110 and the constraint yields 0.049. This satisfies our constraint (as $0.049 < \epsilon$), so we return the new parameterization.

constraint, we can also eliminate line search and use standard gradient ascent methods.

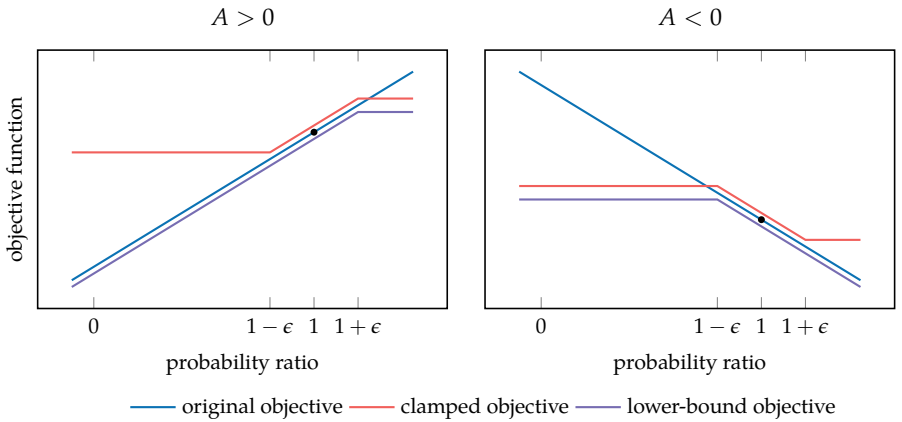


Figure 12.6. A visualization of the lower-bound objective for positive and negative advantages compared to the original objective and the clamped objective. The black point shows the baseline around which the optimization is performed, $\pi_{\theta'}(a \mid s)/\pi_\theta(a \mid s) = 1$. The three line plots in each axis are vertically separated for clarity.

The gradient of the unclamped objective equation (12.21) with action values is

$$\nabla_{\theta'} f(\theta, \theta') = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta}} \left[\mathop{\mathbb{E}}_{a \sim \pi_\theta(\cdot|s)} \left[\frac{\nabla_{\theta'} \pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} Q_\theta(s, a) \right] \right] \tag{12.23}$$

where $Q_\theta(s, a)$ can be estimated from reward-to-go. The gradient of the lower-bound objective equation (12.22) (with clamping), is the same, except there is no contribution from experience tuples for which the objective is actively clamped. That is, if either the reward-to-go is positive and the probability ratio is greater than $1 + \epsilon$, or if the reward-to-go is negative and the probability ratio is less than $1 - \epsilon$, the gradient contribution is zero.

Like TRPO, the gradient can be computed for a parameterization θ' from experience generated from θ. Hence, several gradient updates can be run in a row using the same set of sampled trajectories. Algorithm 12.6 provides an implementation of this.

The clamped surrogate objective is compared to several other surrogate objectives in figure 12.7, which includes a line plot for the effective objective for TRPO:

$$\mathop{\mathbb{E}}_{\substack{s \sim b_{\gamma,\theta} \\ a \sim \pi_\theta(\cdot|s)}} \left[\frac{\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} A_\theta(s, a) - \beta D_{\text{KL}}(\pi_\theta(\cdot \mid s) \mid\mid \pi_{\theta'}(\cdot \mid s)) \right] \tag{12.24}$$

```
struct ClampedSurrogateUpdate
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of trajectories
    π     # policy
    p     # policy likelihood
    ∇π    # policy likelihood gradient
    ϵ     # divergence bound
    α     # step size
    k_max # number of iterations per update
end

function clamped_gradient(M::ClampedSurrogateUpdate, θ, θ′, τs)
    d, p, ∇π, ϵ, γ = M.d, M.p, M.∇π, M.ϵ, M.𝒫.γ
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    ∇f(a,s,r_togo) = begin
        P = p(θ, a,s)
        w = p(θ′,a,s) / P
        if (r_togo > 0 && w > 1+ϵ) || (r_togo < 0 && w < 1-ϵ)
            return zeros(length(θ))
        end
        return ∇π(θ′, a, s) * r_togo / P
    end
    ∇f(τ) = mean(∇f(a,s,R(τ,k)) for (k,(s,a,r)) in enumerate(τ))
    return mean(∇f(τ) for τ in τs)
end

function update(M::ClampedSurrogateUpdate, θ)
    𝒫, b, d, m, π, α, k_max= M.𝒫, M.b, M.d, M.m, M.π, M.α, M.k_max
    πθ(s) = π(θ, s)
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    θ′ = copy(θ)
    for k in 1:k_max
        θ′ += α*clamped_gradient(M, θ, θ′, τs)
    end
    return θ′
end
```

Algorithm 12.6. An implementation of clamped surrogate policy optimization, which returns a new policy parameterization for policy `π(s)` of an MDP `𝒫` with initial state distribution `b`. This implementation samples `m` trajectories to depth `d`, and then uses them to estimate the policy gradient in `k_max` subsequent updates. The policy gradient using the clamped objective is constructed using the policy gradients `∇p` with clamping parameter `ϵ`.

which is the trust region policy objective where the constraint is implemented as a penalty for some coefficient β. TRPO typically uses a hard constraint rather than a penalty because it is difficult to choose a value of β that performs well within a single problem, let alone across multiple problems.

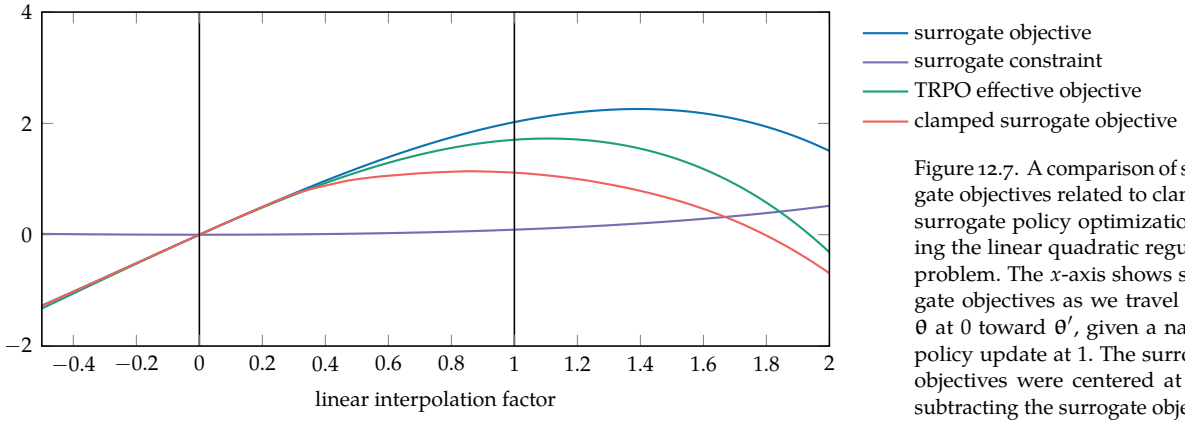


Figure 12.7. A comparison of surrogate objectives related to clamped surrogate policy optimization using the linear quadratic regulator problem. The x-axis shows surrogate objectives as we travel from θ at 0 toward θ', given a natural policy update at 1. The surrogate objectives were centered at 0 by subtracting the surrogate objective function value for θ. We see that the clamped surrogate objective behaves very similarly to the effective TRPO objective without needing a constraint. Note that ϵ and β can be adjusted for both algorithms, which would affect where the maximum is in each case.

12.6 Summary

- The gradient ascent algorithm can use the gradient estimates obtained from the methods discussed in the previous chapter to iteratively improve our policy.
- Gradient ascent can be made more robust by scaling, clipping, or forcing the size of the improvement steps to be uniform.
- The natural gradient approach uses a first-order approximation of the objective function with a constraint on the divergence between the trajectory distribution at each step, approximated using an estimate of the Fisher information matrix.
- Trust region policy optimization involves augmenting the natural gradient method with a line search to further improve the policy without additional trajectory simulations.
- We can use a pessimistic lower bound of the TRPO objective to obtain a clamped surrogate objective that performs similarly without the need for line search.

12.7 Exercises

Exercise 12.1. TRPO starts its line search from a new parameterization given by a natural policy gradient update. However, TRPO conducts the line search using a different objective than the natural policy gradient. Show that the gradient of the surrogate objective equation (12.18) used in TRPO is actually the same as the reward-to-go policy gradient equation (11.26).

Solution: The gradient of TRPO's surrogate objective is

$$\nabla_{\theta'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(a \mid s)}{\pi_\theta(a \mid s)} Q_\theta(s,a)\right]\right]$$

When conducting the initial natural policy gradient update, the search direction is evaluated at $\theta' = \theta$. Furthermore, the action value is approximated with the reward-to-go:

$$\nabla_{\theta'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} r_{\text{to-go}}\right]\right]$$

Recall that the derivative of $\log f(x)$ is $f'(x)/f(x)$. It thus follows that

$$\nabla_{\theta'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\nabla_\theta \log \pi_\theta(a \mid s) r_{\text{to-go}}\right]\right]$$

which takes the same form as the reward-to-go policy gradient equation (11.26).

Exercise 12.2. Perform the calculations of example 12.1. First, compute the inverse of the Fisher information matrix $\mathbf{F}_\theta^{-1}$, compute $\mathbf{u}$, and compute the updated parameters θ'.

Solution: We start by computing the inverse of the Fisher information matrix:

$$\mathbf{F}_\theta^{-1} \approx \frac{1}{0.341(29.529) - 0.332(0.332)}\begin{bmatrix} 29.529 & -0.332 \\ -0.332 & 0.341 \end{bmatrix} \approx \begin{bmatrix} 0.501 & -0.017 \\ -0.017 & 0.034 \end{bmatrix}$$

Now, we update $\mathbf{u}$ as follows:

$$\mathbf{u} = \mathbf{F}_\theta^{-1}\nabla U(\theta) \approx \begin{bmatrix} 0.501 & -0.017 \\ -0.017 & 0.034 \end{bmatrix}\begin{bmatrix} 2.030 \\ 1.020 \end{bmatrix} \approx \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Finally, we estimate the updated parameters θ:

$$\begin{aligned}
\theta' &= \theta + \mathbf{u}\sqrt{\frac{2\epsilon}{\nabla U(\theta)^\top \mathbf{u}}} \\
&\approx \begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\sqrt{\frac{2(0.1)}{\begin{bmatrix}2.030 & 1.020\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}}} \\
&\approx \begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\sqrt{\frac{0.2}{2.030}} \\
&\approx \begin{bmatrix}0.314\\1\end{bmatrix}
\end{aligned}$$

Exercise 12.3. Suppose we have the parameterized policies π_θ and $\pi_{\theta'}$ given in the following table:

	a_1	a_2	a_3	a_4
$\pi_\theta(a \mid s_1)$	0.1	0.2	0.3	0.4
$\pi_{\theta'}(a \mid s_1)$	0.4	0.3	0.2	0.1
$\pi_\theta(a \mid s_2)$	0.1	0.1	0.6	0.2
$\pi_{\theta'}(a \mid s_2)$	0.1	0.1	0.5	0.3

Given that we sample the following five states, s_1, s_2, s_1, s_1, s_2, approximate $\mathbb{E}_s\left[D_{\text{KL}}(\pi_\theta(\cdot \mid s) \mid\mid \pi_{\theta'}(\cdot \mid s))\right]$ using the definition

$$D_{\text{KL}}(P \mid\mid Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

Solution: First, we compute the KL divergence for a state sample s_1:

$$D_{\text{KL}}(\pi_\theta(\cdot \mid s_1) \mid\mid \pi_{\theta'}(\cdot \mid s_1)) = 0.1 \log\left(\tfrac{0.1}{0.4}\right) + 0.2 \log\left(\tfrac{0.2}{0.3}\right) + 0.3 \log\left(\tfrac{0.3}{0.3}\right) + 0.4 \log\left(\tfrac{0.4}{0.1}\right) \approx 0.456$$

Now, we compute the KL divergence for a state sample s_2:

$$D_{\text{KL}}(\pi_\theta(\cdot \mid s_2) \mid\mid \pi_{\theta'}(\cdot \mid s_2)) = 0.1 \log\left(\tfrac{0.1}{0.1}\right) + 0.1 \log\left(\tfrac{0.1}{0.1}\right) + 0.6 \log\left(\tfrac{0.6}{0.5}\right) + 0.2 \log\left(\tfrac{0.2}{0.3}\right) \approx 0.0283$$

Finally, we compute the approximation of the expectation, which is the average KL divergence of the parameterized policies over the n state samples:

$$\begin{aligned}
\mathbb{E}_s[D_{\text{KL}}(\pi_\theta(\cdot \mid s) \mid\mid \pi_{\theta'}(\cdot \mid s))] &\approx \frac{1}{n}\sum_{i=1}^{n} D_{\text{KL}}\left(\pi_\theta(\cdot \mid s^{(i)}) \,\middle\|\, \pi_{\theta'}(\cdot \mid s^{(i)})\right) \\
&\approx \frac{1}{5}\left(0.456 + 0.0283 + 0.456 + 0.456 + 0.0283\right) \\
&\approx 0.285
\end{aligned}$$

13 Actor-Critic Methods

The previous chapter discussed ways to improve a parameterized policy through gradient information estimated from rollouts. This chapter introduces *actor-critic methods*, which use an estimate of a value function to help direct the optimization. The actor, in this context, is the policy, and the critic is the value function. Both are trained in parallel. We will discuss several methods that differ in whether they approximate the value function, advantage function, or action value function. Most focus on stochastic policies, but we will also discuss one method that supports deterministic policies that output continuous actions. Finally, we will discuss a way to incorporate an online method for generating more informative trajectories for training the actor and critic.

13.1 Actor-Critic

In actor-critic methods, we have an actor represented by a policy π_θ, parameterized by θ with the help of a critic that provides an estimate of the value function $U_\phi(s)$, $Q_\phi(s,a)$, or $A_\phi(s,a)$ parameterized by ϕ. We will start this chapter with a simple actor-critic approach in which the optimization of π_θ is done through gradient ascent, with the gradient of our objective being the same as in equation (11.44):

$$\nabla U(\theta) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} A_\theta\left(s^{(k)}, a^{(k)}\right)\right] \tag{13.1}$$

The advantage when following a policy parameterized by θ can be estimated using a set of observed transitions from s to s' with reward r:

$$A_\theta(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma U^{\pi_\theta}(s') - U^{\pi_\theta}(s)\right] \tag{13.2}$$

The $r + \gamma U^{\pi_\theta}(s') - U^{\pi_\theta}(s)$ inside the expectation is referred to as the *temporal difference residual*.

The critic allows us to estimate the true value function U^{π_θ} when following π_θ, resulting in the following gradient for the actor:

$$\nabla U(\theta) \approx \mathbb{E}_\tau \left[\sum_{k=1}^{d} \nabla_\theta \log \pi_\theta(a^{(k)} \mid s^{(k)}) \gamma^{k-1} \left(r^{(k)} + \gamma U_\phi(s^{(k+1)}) - U_\phi(s^{(k)}) \right) \right] \tag{13.3}$$

This expectation can be estimated through rollout trajectories, as done in chapter 11.

The critic is also updated through gradient optimization. We want to find a ϕ that minimizes our loss function:

$$\ell(\phi) = \frac{1}{2} \mathbb{E}_s \left[\left(U_\phi(s) - U^{\pi_\theta}(s) \right)^2 \right] \tag{13.4}$$

To minimize this objective, we can take steps in the opposite direction of the gradient:

$$\nabla \ell(\phi) = \mathbb{E}_s \left[\left(U_\phi(s) - U^{\pi_\theta}(s) \right) \nabla_\phi U_\phi(s) \right] \tag{13.5}$$

Of course, we do not know U^{π_θ} exactly, but it can be estimated using the reward-to-go along rollout trajectories, resulting in

$$\nabla \ell(\phi) = \mathbb{E}_\tau \left[\sum_{k=1}^{d} \left(U_\phi(s^{(k)}) - r_{\text{to-go}}^{(k)} \right) \nabla_\phi U_\phi(s^{(k)}) \right] \tag{13.6}$$

where $r_{\text{to-go}}^{(k)}$ is the reward-to-go at step k in a particular trajectory τ.

Algorithm 13.1 shows how to estimate $\nabla U(\theta)$ and $\nabla \ell(\phi)$ from rollouts. With each iteration, we step θ in the direction of $\nabla U(\theta)$ to maximize utility, and we step ϕ in the opposite direction of $\nabla \ell(\phi)$ to minimize our loss. This approach can become unstable due to the dependency between the estimation of θ and ϕ, but this approach has worked well for a variety of problems. It is a common practice to update the policy more frequently than the value function to improve stability. The implementations in this chapter can easily be adapted to update the value function only for a subset of the iterations that the policy is updated.

```
struct ActorCritic
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    ∇logπ # gradient of log likelihood ∇logπ(θ,a,s)
    U     # parameterized value function U(ϕ, s)
    ∇U    # gradient of value function ∇U(ϕ,s)
end

function gradient(M::ActorCritic, π, θ, ϕ)
    𝒫, b, d, m, ∇logπ = M.𝒫, M.b, M.d, M.m, M.∇logπ
    U, ∇U, γ = M.U, M.∇U, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ,j) = sum(r*γ^(k-1) for (k,(s,a,r)) in enumerate(τ[j:end]))
    A(τ,j) = τ[j][3] + γ*U(ϕ,τ[j+1][1]) - U(ϕ,τ[j][1])
    ∇Uθ(τ) = sum(∇logπ(θ,a,s)*A(τ,j)*γ^(j-1) for (j, (s,a,r))
                 in enumerate(τ[1:end-1]))
    ∇ℓϕ(τ) = sum((U(ϕ,s) - R(τ,j))*∇U(ϕ,s) for (j, (s,a,r))
                 in enumerate(τ))
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓϕ(τ) for τ in trajs)
end
```

Algorithm 13.1. A basic actor-critic method for computing both a policy gradient and a value function gradient for an MDP `𝒫` with initial state distribution `b`. The policy `π` is parameterized by `θ` and has a log-gradient `∇logπ`. The value function `U` is parameterized by `ϕ` and the gradient of its objective function is `∇U`. This method runs `m` rollouts to depth `d`. The results are used to update `θ` and `ϕ`. The policy parameterization is updated in the direction of `∇θ` to maximize the expected value, whereas the value function parameterization is updated in the negative direction of `∇ϕ` to minimize the value loss.

13.2 *Generalized Advantage Estimation*

Generalized advantage estimation (algorithm 13.2) is an actor-critic method that uses a more general version of the advantage estimate shown in equation (13.2) that allows us to balance between bias and variance.[1] Approximation with the temporal difference residual has low variance, but it introduces bias due to a potentially inaccurate U_ϕ used to approximate U^{π_θ}. An alternative is to replace $r + \gamma U^{\pi_\theta}(s')$ with the sequence of rollout rewards $r_1, \ldots, r_d$:

$$A_\theta(s,a) = \mathbb{E}_{r_1,\ldots,r_d}\left[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{d-1} r_d - U^{\pi_\theta}(s)\right] \tag{13.7}$$

$$= \mathbb{E}_{r_1,\ldots,r_d}\left[-U^{\pi_\theta}(s) + \sum_{\ell=1}^{d} \gamma^{\ell-1} r_\ell\right] \tag{13.8}$$

We can obtain an unbiased estimate of this expectation through rollout trajectories, as done in the policy gradient estimation methods (chapter 11). However, the estimate is high variance, meaning that we need many samples to arrive at an accurate estimate.

[1] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," in *International Conference on Learning Representations (ICLR)*, 2016. arXiv: 1506.02438v6.

The approach taken by generalized advantage estimation is to balance between these two extremes of using temporal difference residuals and full rollouts. We define $\hat{A}^{(k)}$ to be the advantage estimate obtained from k steps of a rollout and the utility associated with the resulting state s':

$$\hat{A}^{(k)}(s,a) = \mathbb{E}_{r_1,\ldots,r_k,s'}\left[r_1 + \gamma r_2 + \cdots + \gamma^{k-1} r_k + \gamma^k U^{\pi_\theta}(s') - U^{\pi_\theta}(s)\right] \tag{13.9}$$

$$= \mathbb{E}_{r_1,\ldots,r_k,s'}\left[-U^{\pi_\theta}(s) + \gamma^k U^{\pi_\theta}(s') + \sum_{\ell=1}^{k} \gamma^{\ell-1} r_\ell\right] \tag{13.10}$$

An alternative way to write $\hat{A}^{(k)}$ is in terms of an expectation over temporal difference residuals. We can define

$$\delta_t = r_t + \gamma U(s_{t+1}) - U(s_t) \tag{13.11}$$

where s_t, r_t, and s_{t+1} are the state, reward, and subsequent state along a sampled trajectory and U is our value function estimate. Then,

$$\hat{A}^{(k)}(s,a) = \mathbb{E}\left[\sum_{\ell=1}^{k} \gamma^{\ell-1} \delta_\ell\right] \tag{13.12}$$

Instead of committing to a particular value for k, generalized advantage estimation introduces a parameter $\lambda \in [0,1]$ that provides an *exponentially weighted average* of $\hat{A}^{(k)}$ for k ranging from 1 to d:[2]

[2] The exponentially weighted average of a series $x_1, x_2, \ldots$ is $(1-\lambda)(x_1 + \lambda x_2 + \lambda^2 x_3 + \cdots)$.

$$\hat{A}^{\text{GAE}}(s,a) \mid_{d=1} = \hat{A}^{(1)} \tag{13.13}$$

$$\hat{A}^{\text{GAE}}(s,a) \mid_{d=2} = (1-\lambda)\hat{A}^{(1)} + \lambda \hat{A}^{(2)} \tag{13.14}$$

$$\hat{A}^{\text{GAE}}(s,a) \mid_{d=3} = (1-\lambda)\hat{A}^{(1)} + \lambda\left((1-\lambda)\hat{A}^{(2)} + \lambda \hat{A}^{(3)}\right) \tag{13.15}$$

$$= (1-\lambda)\hat{A}^{(1)} + \lambda(1-\lambda)\hat{A}^{(2)} + \lambda^2 \hat{A}^{(3)} \tag{13.16}$$

$$\vdots$$

$$\hat{A}^{\text{GAE}}(s,a) = (1-\lambda)\left(\hat{A}^{(1)} + \lambda \hat{A}^{(2)} + \lambda^2 \hat{A}^{(3)} + \cdots + \lambda^{d-2} \hat{A}^{(d-1)}\right) + \lambda^{d-1} \hat{A}^{(d)} \tag{13.17}$$

For an infinite horizon, the generalized advantage estimate simplifies to

$$\hat{A}^{\text{GAE}}(s,a) = (1-\lambda)\left(\hat{A}^{(1)} + \lambda\hat{A}^{(2)} + \lambda^2\hat{A}^{(3)} + \cdots\right) \tag{13.18}$$

$$= (1-\lambda)\left(\delta_1\left(1+\lambda+\lambda^2+\cdots\right) + \gamma\delta_2\left(\lambda+\lambda^2+\cdots\right) + \gamma^2\delta_3\left(\lambda^2+\cdots\right) + \cdots\right) \tag{13.19}$$

$$= (1-\lambda)\left(\delta_1\frac{1}{1-\lambda} + \gamma\delta_2\frac{\lambda}{1-\lambda} + \gamma^2\delta_3\frac{\lambda^2}{1-\lambda} + \cdots\right) \tag{13.20}$$

$$= \mathbb{E}\left[\sum_{k=1}^{\infty}(\gamma\lambda)^{k-1}\delta_k\right] \tag{13.21}$$

We can tune parameter λ to balance between bias and variance. If $\lambda = 0$, then we have the high-bias, low-variance estimate for the temporal difference residual from the previous section. If $\lambda = 1$, we have the unbiased full rollout with increased variance. Figure 13.1 demonstrates the algorithm with different values for λ.

```
struct GeneralizedAdvantageEstimation
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    ∇logπ # gradient of log likelihood ∇logπ(θ,a,s)
    U     # parameterized value function U(ϕ, s)
    ∇U    # gradient of value function ∇U(ϕ,s)
    λ     # weight ∈ [0,1]
end

function gradient(M::GeneralizedAdvantageEstimation, π, θ, ϕ)
    𝒫, b, d, m, ∇logπ = M.𝒫, M.b, M.d, M.m, M.∇logπ
    U, ∇U, γ, λ = M.U, M.∇U, M.𝒫.γ, M.λ
    πθ(s) = π(θ, s)
    R(τ,j) = sum(r*γ^(k-1) for (k,(s,a,r)) in enumerate(τ[j:end]))
    δ(τ,j) = τ[j][3] + γ*U(ϕ,τ[j+1][1]) - U(ϕ,τ[j][1])
    A(τ,j) = sum((γ*λ)^(ℓ-1)*δ(τ, j+ℓ-1) for ℓ in 1:d-j)
    ∇Uθ(τ) = sum(∇logπ(θ,a,s)*A(τ,j)*γ^(j-1)
                 for (j, (s,a,r)) in enumerate(τ[1:end-1]))
    ∇ℓϕ(τ) = sum((U(ϕ,s) - R(τ,j))*∇U(ϕ,s)
                 for (j, (s,a,r)) in enumerate(τ))
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓϕ(τ) for τ in trajs)
end
```

Algorithm 13.2. Generalized advantage estimation for computing both a policy gradient and a value function gradient for an MDP `𝒫` with initial state distribution `b`. The policy is parameterized by `θ` and has a log-gradient `∇logπ`. The value function `U` is parameterized by `ϕ` and has gradient `∇U`. This method runs `m` rollouts to depth `d`. The generalized advantage is computed with exponential weighting `λ` using equation (13.21) with a finite horizon. The implementation here is a simplified version of what was presented in the original paper, which included aspects of trust regions when taking steps.

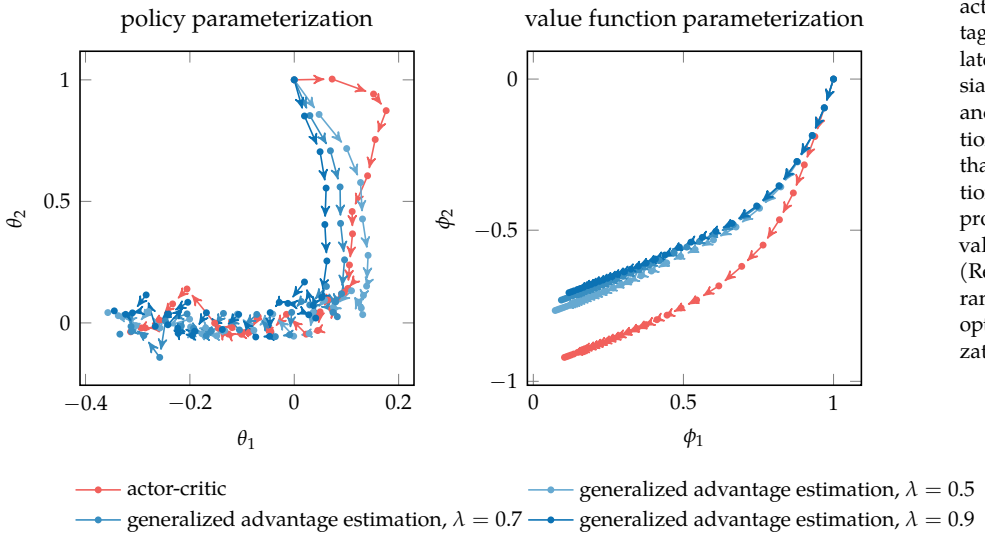


Figure 13.1. A comparison of basic actor-critic to generalized advantage estimation on the simple regulator problem with $\gamma = 0.9$, a Gaussian policy $\pi_\theta(s) = \mathcal{N}(\theta_1 s, \theta_2^2)$, and an approximate value function $U_\phi(s) = \phi_1 s + \phi_2 s^2$. We find that generalized advantage estimation is more efficiently able to approach well-performing policy and value function parameterizations. (Recall that the optimal policy parameterization is $[-1, 0]$ and the optimal value function parameterization is near $[0, -0.7]$.)

13.3 Deterministic Policy Gradient

The *deterministic policy gradient* approach[3] involves optimizing a deterministic policy $\pi_\theta(s)$ that produces continuous actions with the help of a critic in the form of a parameterized action value function $Q_\phi(s, a)$. As with the actor-critic methods discussed so far, we define a loss function with respect to the parameterization ϕ:

$$\ell(\phi) = \frac{1}{2} \mathop{\mathbb{E}}_{s,a,r,s'} \left[\left(r + \gamma Q_\phi(s', \pi_\theta(s')) - Q_\phi(s, a) \right)^2 \right] \tag{13.22}$$

where the expectation is over the experience tuples generated by rollouts of π_θ. This loss function attempts to minimize the residual of Q_ϕ, similar to how the actor-critic method in the first section tried to minimize the residual of U_ϕ.

Similar to the other methods, we update ϕ by taking a step in the opposite direction of the gradient:

$$\nabla \ell(\phi) = \mathop{\mathbb{E}}_{s,a,r,s'} \left[\left(r + \gamma Q_\phi(s', \pi_\theta(s')) - Q_\phi(s, a) \right) \left(\gamma \nabla_\phi Q_\phi(s', \pi_\theta(s')) - \nabla_\phi Q_\phi(s, a) \right) \right] \tag{13.23}$$

We thus need a differentiable parameterized action value function from which we can compute $\nabla_\phi Q_\phi(s, a)$, such as a neural network.

[3] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *International Conference on Machine Learning (ICML)*, 2014.

For the actor, we want to find a value of θ that maximizes

$$U(\theta) = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\theta}} \left[Q_\phi(s, \pi_\theta(s)) \right] \tag{13.24}$$

where the expectation is over the states from the discounted visitation frequency when following π_θ. Again, we can use gradient ascent to optimize θ with the gradient given by

$$\nabla U(\theta) = \mathbb{E}_s \left[\nabla_\theta Q_\phi(s, \pi_\theta(s)) \right] \tag{13.25}$$

$$= \mathbb{E}_s \left[\nabla_\theta \pi_\theta(s) \nabla_a Q_\phi(s, a)|_{a=\pi_\theta(s)} \right] \tag{13.26}$$

Here, $\nabla_\theta \pi_\theta(s)$ is a Jacobian matrix whose ith column is the gradient with respect to the ith action dimension of the policy under parameterization θ. An example for this term is given in example 13.1. The gradient $\nabla_a Q_\phi(s, a)|_{a=\pi_\theta(s)}$ is a vector that indicates how much our estimated action value changes as we perturb the action given by our policy at state s. In addition to the Jacobian, we need to supply this gradient to use this method.

Example 13.1. An example of the Jacobian in the deterministic policy gradient.

Consider the following deterministic policy for a two-dimensional action space and a one-dimensional state space:

$$\pi_\theta(s) = \begin{bmatrix} \theta_1 + \theta_2 s + \theta_3 s^2 \\ \theta_1 + \sin(\theta_4 s) + \cos(\theta_5 s) \end{bmatrix}$$

The matrix $\nabla_\theta \pi_\theta(s)$ then takes the following form:

$$\nabla_\theta \pi_\theta(s) = \begin{bmatrix} \nabla_\theta \pi_\theta(s) \,|_{a_1} & \nabla_\theta \pi_\theta(s) \,|_{a_2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ s & 0 \\ s^2 & 0 \\ 0 & \cos(\theta_4 s) s \\ 0 & -\sin(\theta_5 s) s \end{bmatrix}$$

As with the other actor-critic methods, we perform gradient descent on $\ell(\phi)$ and gradient ascent on $U(\theta)$. For this approach to work in practice, a few additional techniques are needed. One is to generate experiences from a stochastic policy to allow better exploration. It is often adequate to simply add zero-mean

Gaussian noise to actions generated by our deterministic policy π_θ, as done in algorithm 13.3. To encourage stability when learning θ and ϕ, we can use experience replay.[4]

An example of this method and the effect of σ on performance is given in example 13.2.

```
struct DeterministicPolicyGradient
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    m     # number of samples
    ∇π    # gradient of deterministic policy π(θ, s)
    Q     # parameterized value function Q(ϕ,s,a)
    ∇Qϕ   # gradient of value function with respect to ϕ
    ∇Qa   # gradient of value function with respect to a
    σ     # policy noise
end

function gradient(M::DeterministicPolicyGradient, π, θ, ϕ)
    𝒫, b, d, m, ∇π = M.𝒫, M.b, M.d, M.m, M.∇π
    Q, ∇Qϕ, ∇Qa, σ, γ = M.Q, M.∇Qϕ, M.∇Qa, M.σ, M.𝒫.γ
    π_rand(s) = π(θ, s) + σ*randn()*I
    ∇Uθ(τ) = sum(∇π(θ,s)*∇Qa(ϕ,s,π(θ,s))*γ^(j-1) for (j,(s,a,r))
                 in enumerate(τ))
    ∇ℓϕ(τ,j) = begin
        s, a, r = τ[j]
        s′ = τ[j+1][1]
        a′ = π(θ,s′)
        δ = r + γ*Q(ϕ,s′,a′) - Q(ϕ,s,a)
        return δ*(γ*∇Qϕ(ϕ,s′,a′) - ∇Qϕ(ϕ,s,a))
    end
    ∇ℓϕ(τ) = sum(∇ℓϕ(τ,j) for j in 1:length(τ)-1)
    trajs = [simulate(𝒫, rand(b), π_rand, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓϕ(τ) for τ in trajs)
end
```

Algorithm 13.3. The deterministic policy gradient method for computing a policy gradient `∇θ` for a deterministic policy `π` and a value function gradient `∇ϕ` for a continuous action MDP `𝒫` with initial state distribution `b`. The policy is parameterized by `θ` and has a gradient `∇π` that produces a matrix where each column is the gradient with respect to that continuous action component. The value function `Q` is parameterized by `ϕ` and has a gradient `∇Qϕ` with respect to the parameterization and gradient `∇Qa` with respect to the action. This method runs `m` rollouts to depth `d`, and performs exploration using 0-mean Gaussian noise with standard deviation `σ`.

[4] We will discuss experience replay in section 17.7 in the context of reinforcement learning. Other techniques for stabilizing learning include using *target parameterizations*, described in the context of neural representations by T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous Control with Deep Reinforcement Learning," in *International Conference on Learning Representations (ICLR)*, 2016. arXiv: 1509.02971v6.

13.4 Actor-Critic with Monte Carlo Tree Search

We can extend concepts from online planning (chapter 9) to the actor-critic setting in which we improve a parameterized policy $\pi_\theta(a \mid s)$ and a parameterized value function $U_\phi(s)$.[5] This section discusses the application of Monte Carlo tree search (section 9.6) to learning a stochastic policy with a discrete action space. We use our parameterized policy and value function to guide Monte Carlo tree search,

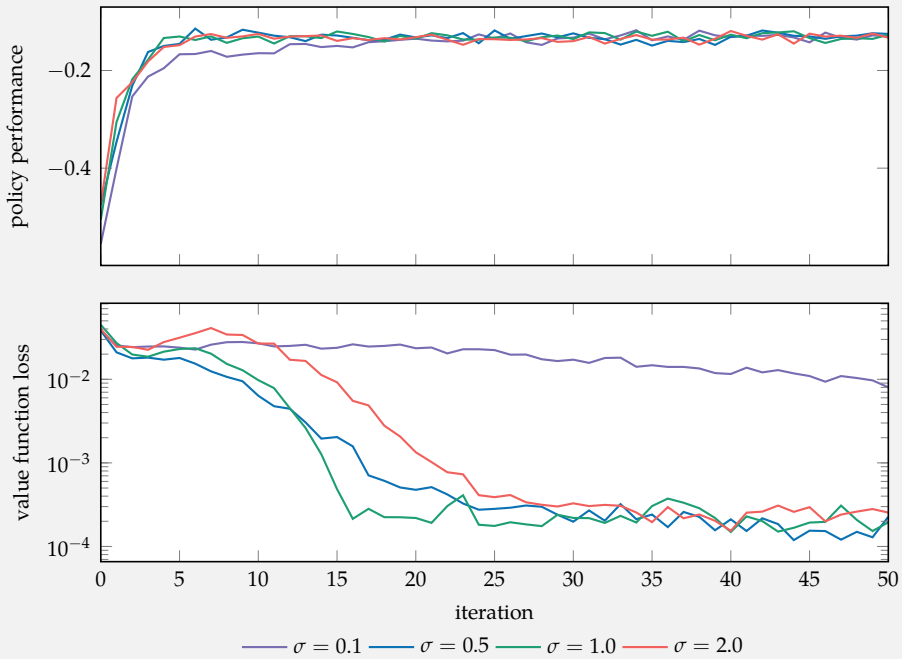
[5] Deterministic policy gradient used Q_ϕ, but this approach uses U_ϕ like the other actor-critic methods discussed in this chapter.

Example 13.2. An application of the deterministic policy gradient method to the simple regulator problem and an exploration of the impact of the policy stochasticity parameter σ.

Consider applying the deterministic policy gradient algorithm to the simple regulator problem. Suppose we use a simple parameterized deterministic policy $\pi_{\boldsymbol{\theta}}(s) = \theta_1$ and the parameterized state-action value function:

$$Q_{\boldsymbol{\phi}}(s, a) = \phi_1 + \phi_2 s + \phi_3 s^2 + \phi_4 (s + a)^2$$

Here, we plot a progression of the deterministic policy gradient algorithm starting with $\boldsymbol{\theta} = [0]$ and $\boldsymbol{\phi} = [0, 1, 0, -1]$ for different values of σ. Each iteration was run with five rollouts to depth 10 with $\gamma = 0.9$.



For this simple problem, the policy quickly converges to optimality almost regardless of σ. However, if σ is either too small or too large, the value function takes longer to improve. In the case of very small values of σ, our policy conducts insufficient exploration from which to effectively learn the value function. For larger values of σ, we explore more, but we also tend to make poor move choices more frequently.

and we use the results from Monte Carlo tree search to refine our parameterized policy and value function. As with the other actor critic methods, we apply gradient-based optimization of θ and ϕ.[6]

As we perform Monte Carlo tree search, we want to direct our exploration to some extent by our parameterized policy $\pi_\theta(a \mid s)$. One approach is to use an action that maximizes the *probabilistic upper confidence bound*:

$$a = \arg\max_a Q(s,a) + c\pi_\theta(a \mid s)\frac{\sqrt{N(s)}}{1+N(s,a)} \tag{13.27}$$

where $Q(s,a)$ is the action value estimated through the tree search, $N(s,a)$ is the visit count as discussed in section 9.6, and $N(s) = \sum_a N(s,a)$.[7]

After running tree search, we can use the statistics that we collect to obtain $\pi_{\text{MCTS}}(a \mid s)$. One way to define this is in terms of the counts:[8]

$$\pi_{\text{MCTS}}(a \mid s) \propto N(s,a)^\eta \tag{13.28}$$

where $\eta \geq 0$ is a hyperparameter that controls the greediness of the policy. If $\eta = 0$, then π_{MCTS} will generate actions at random. As $\eta \to \infty$, it will select the action that was selected the most from that state.

In our optimization of θ, we want our model π_θ to match what we obtain through Monte Carlo tree search. One loss function that we can define is the expected cross entropy of $\pi_\theta(\cdot \mid s)$ relative to $\pi_{\text{MCTS}}(\cdot \mid s)$:

$$\ell(\theta) = -\mathbb{E}_s\left[\sum_a \pi_{\text{MCTS}}(a \mid s)\log \pi_\theta(a \mid s)\right] \tag{13.29}$$

where the expectation is over states experienced during the tree exploration. The gradient is

$$\nabla\ell(\theta) = -\mathbb{E}_s\left[\sum_a \frac{\pi_{\text{MCTS}}(a \mid s)}{\pi_\theta(a \mid s)}\nabla_\theta \pi_\theta(a \mid s)\right] \tag{13.30}$$

To learn ϕ, we define a loss function in terms of a value function generated during the tree search:

$$U_{\text{MCTS}}(s) = \max_a Q(s,a) \tag{13.31}$$

which is defined at least at the states that we explore during tree search. The loss function aims to make U_ϕ agree with the estimates from the tree search:

$$\ell(\phi) = \frac{1}{2}\mathbb{E}_s\left[\left(U_\phi(s) - U_{\text{MCTS}}(s)\right)^2\right] \tag{13.32}$$

[6] This general approach was introduced by D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., "Mastering the Game of Go Without Human Knowledge," *Nature*, vol. 550, pp. 354–359, 2017. The discussion here loosely follows their *AlphaGo Zero* algorithm, but instead of trying to solve the game of Go, we are trying to solve a general MDP. Both the fact that Alpha Zero plays as both Go players and that games tend to have a winner and a loser allow the original method to reinforce the winning behavior and punish the losing behavior. The generalized MDP formulation will tend to suffer from sparse rewards when applied to similar problems.

[7] There are some notable differences from the upper confidence bound presented in equation (9.1); for example, there is no logarithm in equation (13.27) and we add 1 to the denominator to follow the form used by AlphaGo Zero.

[8] In algorithm 9.5, we select the greedy action with respect to Q. Other strategies are surveyed by C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. The approach suggested here follows AlphaGo Zero.

The gradient is

$$\nabla \ell(\boldsymbol{\phi}) = \mathbb{E}_s\left[\left(U_{\boldsymbol{\phi}}(s) - U_{\text{MCTS}}(s)\right)\nabla_{\boldsymbol{\phi}} U_{\boldsymbol{\phi}}(s)\right] \tag{13.33}$$

Like the actor-critic method in the first section, we need to be able to compute the gradient of our parameterized value function.

After performing some number of Monte Carlo tree search simulations, we update $\boldsymbol{\theta}$ by stepping in the direction opposite to $\nabla \ell(\boldsymbol{\theta})$ and $\boldsymbol{\phi}$ by stepping in the direction opposite to $\nabla \ell(\boldsymbol{\phi})$.[9]

[9] The AlphaGo Zero implementation uses a single neural network to represent both the value function and the policy instead of independent parameterizations as discussed in this section. The gradient used to update the network parameters is a mixture of equations (13.30) and (13.33). This enhancement significantly reduces evaluation time and feature learning time.

13.5 Summary

- In actor-critic methods, an actor attempts to optimize a parameterized policy with the help of a critic that provides a parameterized estimate of the value function.
- Generally, actor-critic methods use gradient-based optimization to learn the parameters of both the policy and value function approximation.
- The basic actor-critic method uses a policy gradient for the actor and minimizes the squared temporal difference residual for the critic.
- The generalized advantage estimate attempts to reduce the variance of its policy gradient at the expense of some bias by accumulating temporal difference residuals across multiple time steps.
- The deterministic policy gradient can be applied to problems with continuous action spaces and uses a deterministic policy actor and an action value critic.
- Online methods, such as Monte Carlo tree search, can be used to direct the optimization of the policy and value function estimate.

13.6 Exercises

Exercise 13.1. Would the actor-critic method with Monte Carlo tree search, as presented in section 13.4, be a good method for solving the cart-pole problem (appendix F.3)?

Solution: The Monte Carlo tree search expands a tree based on visited states. The cart-pole problem has a continuous state space, leading to a search tree with an infinite branching factor. Use of this algorithm would require adjusting the problem, such as discretizing the state space.