Shaping and policy search in Reinforcement learning

by

Andrew Y. Ng

B.S. (Carnegie Mellon University) 1997
B.S. (Carnegie Mellon University) 1997
M.S. (Massachusetts Institute of Technology) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Michael I. Jordan, Chair
Professor Andy Packard
Professor Stuart Russell
Professor Shankar Sastry

Spring 2003

The dissertation of Andrew Y. Ng is approved:

Chair Date

Date

Date

Date

University of California, Berkeley

Spring 2003

Shaping and policy search in Reinforcement learning

Copyright Spring 2003

by

Andrew Y. Ng

Abstract

Shaping and policy search in Reinforcement learning

by

Andrew Y. Ng

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael I. Jordan, Chair

Reinforcement learning offers a powerful set of tools for sequential decision making under uncertainty. In this setting, an algorithm is required to learn to make sequences of decisions, and is evaluated by the long-term quality of its choices. As a concrete example, consider the problem of autonomous helicopter flight, in which an algorithm must repeatedly choose good control actions every fraction of a second.

Central to reinforcement learning is the idea of a *reward function*, which indicates to the learning algorithm what states of the world are preferred, and what states of the world should be avoided. The reward function specifies the learning task. To make reinforcement learning algorithms run in a reasonable amount of time, it is frequently necessary to use a well-chosen reward function that gives appropriate "hints" to the learning algorithm. But, the selection of these hints—called shaping rewards—often entails significant trial and error, and poorly chosen shaping rewards often change the problem in unanticipated ways that

cause poor solutions to be learned. In this dissertation, we give a theory of reward shaping that shows how these problems can be eliminated. This theory further gives guidelines for selecting good shaping rewards that in practice give significant speedups of the learning process. We also show that shaping can allow us to use "myopic" learning algorithms and still do well.

The "curse of dimensionality" refers to the observation that many simple reinforcement learning algorithms, ones based on discretization, scale exponentially with the size of the problem and are thus impractical for many applications. In this dissertation, we consider the policy search approach to reinforcement learning. Here, we wish to select a controller from among some restricted set of controllers for a task. We see that a key issue in policy search is obtaining uniformly good estimates of the quality of the controllers being considered. We show that simple Monte Carlo methods will not in general give uniformly good estimates. We then present the PEGASUS policy search method, which is derived using the surprising observation that all reinforcement learning problems can be transformed into ones in which all state transitions (given the current state and action) are deterministic. We show that PEGASUS has sample complexity that scales at most polynomially with the size of the problem, and give strong guarantees on the quality of the solutions it finds. In deriving these results, we also take the ideas of VC dimension and sample complexity that are familiar from supervised learning and apply them to the reinforcement learning setting, thus putting the two problems on a more equal footing.

Finally, we apply these ideas to designing a controller for an autonomous helicopter. Autonomous helicopter flight is widely viewed as a difficult control problem. Using

shaping and the PEGASUS policy search method, we are able to automatically design a stable hovering controller for a helicopter, as well as make it fly a number of challenging maneuvers taken from an RC helicopter competition.

Professor Michael I. Jordan
Dissertation Committee Chair

Contents

List of Figures **vi**

1 Introduction **1**
- 1.1 Introduction to Reinforcement learning 1
- 1.2 Comparison to supervised learning 5
- 1.3 Thesis outline and contributions 8

2 Reinforcement Learning and (PO)MDPs **10**
- 2.1 Markov decision processes 11
- 2.2 Some MDP properties and algorithms 15
- 2.3 MDP algorithms 18
- 2.4 Partially observable Markov decision processes 22

3 Shaping in Reinforcement Learning **25**
- 3.1 Changing the reward function 26
- 3.2 Shaping Rewards 29
- 3.3 Main shaping results 31
- 3.4 Experiments 37
- 3.5 Discussion 42

4 Pegasus: A policy search method for large MDPs and POMDPs **53**
- 4.1 Policy Search 54
- 4.2 Policy search framework 58
  - 4.2.1 Deterministic simulative models 59
  - 4.2.2 Policy search strategy 62
  - 4.2.3 VC dimension and complexity 67
- 4.3 The trajectory trees method 70
- 4.4 Policy search method 74
  - 4.4.1 Transformation of (PO)MDPs 75
  - 4.4.2 PEGASUS: A method for policy search 77
- 4.5 Main theoretical results 80
  - 4.5.1 The case of finite action spaces 80

4.5.2 The case of infinite action spaces: “Simple” Π is insufficient for uniform convergence 82
4.5.3 Uniform convergence in the case of infinite action spaces 84
4.6 Experiments 87
4.7 Discussion and related work 90

5 Autonomous helicopter flight via reinforcement learning 105
5.1 Introduction 106
5.2 Model identification 110
5.2.1 Locally weighted regression 111
5.2.2 Model selection and incorporating prior knowledge 114
5.3 Learning to Hover 121
5.4 Flying competition maneuvers 129

6 Conclusions 138

Bibliography 143

List of Figures

1.1 Berkeley autonomous helicopter. 2

2.1 5x5 grid-world in which the agent starts at S and must make its way to the goal G. 13

3.1 (a) 10x10 grid-world in which the agent starts at S and must make its way to the goal G. (b) 5x5 grid-world with 5 subgoals (including goal state), which must be visited in order $1, 2, 3, 4, G$. 38

3.2 (a) Experiment with 10x10 grid-world. Plot of steps taken to goal vs. trial number. Dotted line is with no shaping; dot-dash line is with $\Phi = 0.5\Phi_0$; solid line is with $\Phi = \Phi_0$. (b) Experiment with 50x50 grid-world. 39

3.3 (a) Results of experiment with 5x5 grid-world with subgoals. Plot of steps taken to goal vs. trial number. Dotted line is no shaping; dot-dash line is with $\Phi = \Phi_0$; solid line is with $\Phi = \Phi_1$. (b) Results of experiment with larger, 8x8 grid-world with more subgoals. 40

3.4 The unlabeled thick edges correspond to both actions. All edges have probability 1. The edge $(s_1, a, \hat{s}_0)$ carries a reward $\Delta/2$, and all other edges have zero reward. 47

4.1 (a) A simulator/generative model for an MDP, that takes as input any (s, a)-pair, and outputs $s' \sim P_{sa}(\cdot)$. (b) A typical computer implementation of a simulator, in which a random number generator is called to generate p, and the output s' is computed as a deterministic function $g(s, a, p)$ of p and of the inputs s, a. 60

4.2 Monte Carlo evaluation of a policy π, using a generative model/simulator. . 64

4.3 A trajectory tree. 71

4.4 PEGASUS evaluation of a policy π, using a generative model/simulator. . . . 75

4.5 (a) 5x5 gridworld, with the 8 observations. (b) PEGASUS results using the normal and complex deterministic simulative models. The topmost horizontal line shows the value of the best policy in Π; the solid curve is the mean policy value using the normal model; the lower curve is the mean policy value using the complex model. The (almost negligible) 1 s.e. bars are also plotted. 87

5.1 Berkeley autonomous helicopter. 107
5.2 Examples of plots comparing a (globally) linear model fit using the parameterization described in the text (solid lines) to some other models (dash-dot lines). Each point plotted shows the mean-squared error between the predicted value of a state variable—when a model (ignoring model noise) is used to the simulate the helicopter's dynamics for a certain duration indicated on the x-axis—and the true value of that state variable (as measured on test data) after the same duration. Top left: Comparison of $\dot{x}$-error to model not using extra a_{1s}, etc. variables. Top right: Comparison of $\dot{x}$-error to a model that omits the intercept (bias) term. Bottom: Comparison of $\dot{x}$ and $\dot{\theta}$ to linear deterministic model identified by [96]. 118
5.3 The solid line is the true helicopter $\dot{y}$ state on 10s of test data. The dash-dot line is the helicopter state predicted by our model, given the initial state at time 0 and all the intermediate control inputs. The dotted lines show 2 standard deviations in the estimated state. (Calculating these on a stochastic non-linear model is actually intractable, so these were estimated using an extended Kalman filter, and using a diagonal approximation to all the covariances as in [67, 21].) Every two seconds, the estimated state is "reset" to the true state, and the track starts again with zero error. Note that the estimated state is of the full, high-dimensional state of the helicopter, but only $\dot{y}$ is shown here. 119
5.4 Policy class Π used to learn a controller for hovering. The pictures inside the circles indicate whether each node computes and outputs the sum of its inputs, or the tanh of its input. Each edge with an arrow in the picture denotes a tunable parameter. 123
5.5 Helicopter hovering under control of learned policy. 126
5.6 Comparison of hovering performance of learned controller (blue solid line; colors where available) vs. trained human pilot (red dashed line). Shown here are the x^b, y^b and z^b (body coordinate) velocities. 127
5.7 Comparison of hovering performance of learned controller (blue solid line; colors where available) vs. trained human pilot (red dashed line). Shown here are the x, y and z (world coordinate) positions. 128
5.8 Diagrams of maneuvers from an RC helicopter competition organized by the Academy of Model Aeronautics. [Source: `www.modelaircraft.org`] 130
5.9 Policy class Π used to learn a controller for flying competition maneuvers. The dashed arrows show the newly-added edges. 132
5.10 Plots of example helicopter trajectories (as recorded by the onboard telemetry) flying the three competition maneuvers. 136

Acknowledgements

First, I must thank Professor Michael Jordan for endless advice, guidance and inspiration throughout my time as a graduate student. I am greatly indebted to Professor Jordan for all that I learned from him both in terms of technical knowledge and in terms of research style, and this work would have been impossible without his help and support.

Another person who had a large impact on my work is Professor Stuart Russell. Professor Russell's advice to me was invaluable on numerous occasions; I learned enormously from many enlightening discussions with him, and feel privileged to have had opportunities to work with him. I am also very grateful to Professor Shankar Sastry for his support and advice; working with members of his research group was also a hugely positive and enriching experience. I also thank Professor Andy Packard for helpful conversations about this dissertation.

I would also like to acknowledge a debt to Professor Michael Kearns. Working with him over the past 8 years has significantly influenced my thinking and research; indeed, many of the learning theory tools that I use regularly now were things that I'd originally learned from him when I was an undergraduate, and I count myself immeasurably lucky to have had him as a mentor and colleague over the years.

My experience as a graduate student would have been much poorer but for the post docs and fellow students at Berkeley. For their friendship and for many enlightening and edifying discussions and collaborations, I would like to thank particularly David Andre, Francis Bach, David Blei, Hoam Chung, Nando de Freitas, Daishi Harada, Hyounjin Kim, Gregory Lawrence, Jon McAuliffe, Kevin Murphy, Vassilis Papavassiliou, Mark Paskin,

Hanna Pasula, Doron Tal, Sekhar Tatikonda, Yair Weiss, Eric Xing, and Alice Zheng.

This work has also benefitted from conversations with Peter Bartlett, Yishay Mansour, Ronald Parr, Satinder Singh, and Ben Van Roy, and my thanks goes to them also.

Most of the work presented in Chapter 3 was done in collaboration with Daishi Harada and Stuart Russell. I also thank Jette Randløv and Preben Alstrøm for the use of the bicycle simulator, and Eric Xing for helpful conversations regarding the horizon time results.

The work presented in Chapter 5 was done in collaboration with Hyounjin Kim, and the helicopter on which the experiments were run belongs to Professor Shankar Sastry's research group. I am also indebted to Hyounjin for many enlightening conversations about control theory, which have significantly influenced the way I think about reinforcement learning. I am also grateful to Hoam Chung for the very significant amount of help that he gave in running helicopter experiments; to Drew Bagnell and to David Shim for many helpful conversations and suggestions about helicopters; to helicopter safety pilot Cedric Ma for his help in running the experiments; and to Peter Ray for his help organizing things for experiments. Conversations with Drew Bagnell have also significantly influenced my thinking about robust control.

The author was supported by a Berkeley fellowship and a Microsoft Research fellowship. This work was also supported by ARO MURI grant DAAH04-96-1-0341, ONR grant N00014-97-1-0941, and National Science Foundation grants ECS-9873474 and IIS-9988642.

Chapter 1

Introduction

In this chapter, we give an informal, non-mathematical overview of the reinforcement learning framework that we will consider in this thesis. We also describe some of the issues in reinforcement learning that we will try to address, and give an outline for the rest this dissertation.

1.1 Introduction to Reinforcement learning

Given a helicopter such as shown in Figure 1.1, how can we learn, or automatically design, a controller to make it fly?

One of the fundamental problems in Artificial Intelligence and control is that of sequential decision making in stochastic systems. A helicopter in flight is an example of a stochastic system, in that it exhibits random, unpredictable behavior, and wind gusts and other disturbances may cause it to move in unexpected ways. Helicopter control also represents a sequential decision-making problem, in that flying a helicopter entails contin-

Figure 1.1: Berkeley autonomous helicopter.

ually deciding in which direction to push the control stick. This makes it a harder problem than ones where we are required only to make a single good decision at a single instance in time, as it exhibits the property of *delayed consequences*: The quality of an autopilot is determined by its long-term performance, and if it makes a bad misstep now, the helicopter might still not crash for many seconds. Another aspect of helicopter control that makes it challenging is its partial observability. In particular, we usually cannot observe the helicopter position/state accurately; but despite our uncertainty about the state of the system, we are still required to compute good control commands every fraction of a second to keep it in the air.

We defer formalizing the Markov decision process (MDP) framework to Chapter 2. Briefly, it models a system (such as the helicopter) that we are interested in controlling as being in some "state" at each step in time. For instance, the state of a helicopter might be represented by its position and orientation. As a result of the actions we select, our system then moves through some sequence of states. Our task is to select actions so that the system tends to stay in "good" states, such as ones corresponding to hovering stably, and avoid "bad" states, such as ones corresponding to crashing. A large and diverse set of problems can be modeled using the MDP formalism. Some examples include planning and robot navigation [99], inventory management [85], machine maintenance [64], network routing [20], elevator control [26], and building recommender systems [47].

Reinforcement learning [99] gives a set of tools for solving problems posed in the MDP formalism. Despite its numerous successes, current algorithms still have difficulty with many problems, and a number of issues and challenges remain. We briefly describe

some of the issues that we think make certain reinforcement learning problems particularly challenging:

- First, there is the issue of high dimension problems. Specifically, simple reinforcement learning algorithms, ones based on discretization, often scale exponentially with the number of state variables. This problem, which we describe in more detail in Chapter 2, is known as the "curse of dimensionality" [14]. Can we design practical algorithms that provably work and that scale better to high dimensional problems?

- Also, there is the problem of how the "reward function" is chosen. In reinforcement learning, a designer has to specify a function that tells us (say) when the helicopter is flying well, and when it is flying badly. We have significant freedom in choosing this function, but as we will see in Chapter 3, certain choices can accelerate learning by orders of magnitude, while other, seemingly-benign choices can lead to very poor controllers being learned. Can we select reward functions that do not suffer from these problems, and that permit reinforcement learning algorithms to learn easily or quickly?

- Partial observability refers to the setting in which the state of the system being controlled cannot be observed exactly, such as when the sensors on a helicopter can measure its position only approximately. Partial observability makes the problem more difficult, and many standard reinforcement learning algorithms are inapplicable or become very difficult to apply to this setting. How can we still choose good controls for a system if we can only see approximately what it is doing?

In this thesis, we develop methods that attempt to address the first two of these

issues. Our resulting algorithms will also work well in the difficult, partially observable setting, and we apply our methods to controlling the helicopter shown in Figure 1.1. In doing so, we also visit topics such as system identification and verification that are familiar from classical control theory.

1.2 Comparison to supervised learning

Supervised learning is another standard problem in Artificial Intelligence. It can be thought of as a form of reinforcement learning in which we need to control a system only for a single time step, so that we need to make only a one-shot decision, rather than sequential decisions. While this difference may not seem significant, this turns out to make supervised learning a significantly easier problem.

For a concrete example, consider applying a supervised learning algorithm to the problem of predicting whether a patient has heart disease, given various measurements or "features" of the patient (such as heart rate, temperature, and results of various medical tests). Here, we imagine that we are given a training set consisting of some example patients' features, and information indicating whether each of these patients had heart disease. We might then use a supervised learning algorithm to fit some function—say a linear map [30] or a small neural network [69]—to this data. When a new patient arrives, we can then use our fitted function to try to predict, as a function of the new patient's features, whether she has heart disease; and as a result of this one-shot prediction, our patient then goes to meet her fate. If we make a mistake on a prediction (such as if we decide to rush a patient into surgery, but the subsequent operation reveals that there was nothing wrong after all

and the surgery was unnecessary), then we can also observe its consequences right away, and recognize and learn from the mistake.

In contrast, in reinforcement learning the consequences of our actions are often delayed, so that it becomes much harder to recognize and learn from the long-term effects of our actions. For instance, if we were to lose (or win) a game of chess on move 63, it may be non-trivial to recognize that the outcome had been predetermined by a blunder (or brilliant move) that we had made back in move 17. This "credit assignment" problem makes it much harder to learn to avoid past failures or repeat past successes.

Further, the sequential nature of reinforcement learning problems also makes it generally difficult to *reuse* data. In the supervised learning setting, if we had previously collected and stored away some set of examples of patients, and if we wanted to test a new neural network for predicting heart disease, then we can easily test how well the new neural network's predictions on our set of examples matches the actual outcomes, and declare the new model to be good or bad accordingly. In the reinforcement learning setting however, suppose we had previously extensively tested a controller that flies the helicopter upside down (or, for a less contrived example, one that flies the helicopter tilted slightly to the right; this is actually done for good reasons—see Chapter 5). The data gathered during these tests might then give us a very good idea of how the helicopter flies upside down, but it is unclear how this data might be reused to evaluate, say, a new controller that flies the helicopter straight-and-level. Thus, it seems that we might need to gather new data to test each new controller that might fly the helicopter in a slightly different way than previous ones. This property makes reinforcement learning typically require significantly

more "data" than supervised learning. One of the goals of this work will be to explore when we can efficiently reuse data in the setting of reinforcement learning, and see when these ideas can be exploited to give practical learning algorithms.

Lastly, one common theme in supervised learning is that of "agnostic learning." [52, 102]. (In AI, there is also the closely-related notion of bounded optimality [89].) This refers to the idea that it is often preferable to restrict the set of possible classifiers considered. For instance, in the heart disease example, rather than considering all possible functions mapping from patients' features to {`disease`, `no_disease`}—which is a huge space of functions—we might instead restrict our attention to some small set of functions, say all thresholded linear functions, or all functions representable by a medium-sized neural network. This significantly reduces the space of classifiers that we must consider. If it so happens that the "true" decision boundary separating patients with and patients without heart disease is so extremely complicated that no neural network can make accurate predictions for his problem, then since we have restricted our attention to functions represented by neural networks, we will not find a good classifier. But if the true decision boundary turns not to be that complicated, then our having restricted the set of classifiers considered allows one to show that only a "small" amount of training data is needed to fit our neural network well. Specifically, it turns out that the amount of data needed depends only on the number of "free parameters" in the neural network, but not on the underlying "complexity" of the input (distribution of) patients and heart disease occurrences [102, 52]. These results are proved in the supervised learning setting using the fact that data in supervised learning can be reused. In Chapter 4, we will generalize some of these notions to the setting of reinforce-

ment learning, and see how, by restricting our attention to some small set of controllers for a reinforcement learning problem, we may also derive related guarantees such as bounds on the "sample size" needed to learn well.

1.3 Thesis outline and contributions

The remainder of this dissertation is structured as follows. In Chapter 2, we begin by formalizing the Markov decision process (MDP) and partially observable Markov decision process (POMDP) frameworks. We also review some standard algorithms for solving MDPs, point out when they may or may not work well, and discuss some of the difficulties with scaling them to large problems or to POMDPs.

In Chapter 3, we describe *reward shaping*, which refers to the practice of choosing or modifying a reward function to help algorithms learn. We describe how seemingly natural attempts at shaping can lead to very poor solutions being learned, and give a theory of shaping that shows how these problems can be eliminated. We also give guidelines for designing good shaping functions that in practice result in significant speedups of the learning process.

Chapter 4 begins by outlining the policy search framework, in which we restrict our attention to a small set of possible controllers for an MDP, and we present a method for "reusing" data for evaluating and finding good controllers. Our methods also applies well to POMDPs, and scales well to large problems: We give bounds on the amount of data needed that, depending on the exact assumptions, either has no dependence or has at most polynomial dependence in the dimension of the problem.

Finally, in Chapter 5, we apply these ideas to designing a controller for flying the helicopter shown in Figure 1.1. We begin with a description of the system identification process, in which we learn a nonlinear stochastic model for the helicopter dynamics. We then apply our learning algorithms, first to make the helicopter hover, and then to make it fly challenging maneuvers taken from an RC helicopter competition.

Chapter 2

Reinforcement Learning and (PO)MDPs

In this chapter, we describe the reinforcement learning problem and introduce the Markov decision process (MDP) and partially observable Markov decision process (POMDP) formalisms. In doing so, we also introduce the notation that will be used in the remainder of this dissertation. We also describe some of the key ideas and algorithms in reinforcement learning, and show how many important problems in decision making and control can be posed in this framework.

For a more detailed introduction to reinforcement learning and MDPs than this, readers may also refer to standard texts such as [99, 17, 85], or the survey by Kaelbling et al. [48].

2.1 Markov decision processes

Markov decision processes provide a formalism for reasoning about planning and acting in the face of uncertainty. There are many possible ways of defining MDPs, and many of these definitions are equivalent up to small transformations of the problem. One definition is that an MDP M is a tuple $(S, D, A, \{P_{sa}(\cdot)\}, \gamma, R)$ consisting of:

- S: A set of possible **states** of the world.
- D: An **initial state distribution** (a probability distribution over S).
- A: A set of possible **actions** from which we may choose on each time step. ($|A| \geq 2$.)
- $P_{sa}(\cdot)$: The **state transition distributions**. For each state $s \in S$ and action $a \in A$, this gives the distribution over to which state we will randomly transition if we take action a in state s.
- γ: A number in $[0, 1]$ called the **discount factor**.
- $R : S \mapsto \mathbb{R}$: The reward function, bounded by $R_{\max}$. ($|R(s)| \leq R_{\max}$ for all s.)

Events in an MDP proceed as follows. We begin in an initial state s_0 drawn from the distribution D. At each time step t, we then have to pick an action a_t, as a result of which our state transitions to some s_{t+1} drawn from the probability transition distribution $P_{s_t a_t}(\cdot)$. By repeatedly picking actions, we traverse some sequence of states $s_0, s_1, \ldots$. Our total payoff is then the sum of discounted rewards along this sequence of states

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots . \tag{2.1}$$

Here, the discount factor γ, which is typically strictly less than one, causes rewards obtained far down the sequence to be given a smaller weight. In economic applications, γ has a natural interpretation via the risk-free interest rate, so that money (rewards) obtained immediately are more valuable than those obtained in the future. If we let γ be close to 1, we can also approximate an undiscounted problem.

Rewards can also be stochastic rather than deterministic functions of the state, and are also frequently written in slightly different forms, such as $R(s_t, a_t)$ (depending on the current state and action) or $R(s_t, a_t, s_{t+1})$ (depending on the current state, action, and the successor state); we will also occasionally use these forms of reward functions. In settings in which we need not be concerned with the initial state distribution, we also write an MDP as $(S, A, \{P_{sa}(\cdot)\}, \gamma, R)$.

In reinforcement learning, our goal is to find a way of choosing actions $a_0, a_1, \ldots$ over time, so as to maximize the expected value of the rewards given in Equation (2.1).

Figure 2.1 shows a standard, simple example problem that may be modeled as an MDP. Imagine a robot that lives in the 5x5 grid. The 25 cells thus comprise its state space S. It has a set A of four actions that try to move it in each of the four compass directions. But perhaps because its wheels slip, it doesn't always manage to go where it intends to, and hence with some small probability, its state transitions to a randomly chosen neighboring state rather than the one in the intended direction of movement. This stochastic behavior would be encoded in the state transition probabilities. (For instance, $P_{(3,3),\text{north}}((3,4)) = 0.85$, $P_{(3,3),\text{north}}((3,2)) = P_{(3,3),\text{north}}((2,3)) = P_{(3,3),\text{north}}((4,3)) = 0.05$.) The robot starts in the bottom-left state, so D assigns the $(1,1)$ state probability 1, and when it reaches the goal,

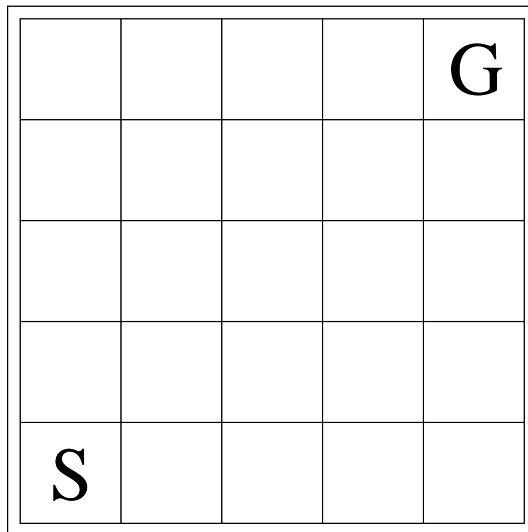


Figure 2.1: 5x5 grid-world in which the agent starts at S and must make its way to the goal G.

it receives a reward of plus one (thus, $R((5,5)) = 1$). If we are using some discount $\gamma < 1$, we would thus prefer to reach the goal, and obtain the reward, quickly rather than slowly.

In a Markov decision process, our agent observes at each step the current state s_t, and is thus allowed to choose its next action a_t as a function of the previous and current states $s_0, \ldots, s_t$. But because of the Markov property of MDPs (informally, that the future is conditionally independent of the past given the current state), to attain the optimal expected sum of rewards, it suffices to choose actions only a function of the current state s_t [19, 99]. Thus, the reinforcement learning can be posed as that of finding a good **policy** $\pi : S \mapsto A$ so that if in each state s we take action $\pi(s)$, we will obtain a large expected sum of rewards:

$$\mathbf{E}_\pi[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots]. \tag{2.2}$$

(Here, with some abuse of notation, we use "$\mathbf{E}_\pi$" to denote an expectation taken with respect to actions chosen according to π).

The reward function R is thus the "task description," and specifies the objective that we seek to optimize. Sometimes, we have significant freedom in choosing R. For instance in our previous gridworld example, rather than giving the agent the carrot of a +1 reward when the agent reaches the goal, we might instead give the stick of a -1 reward on every step until it reaches the goal. Alternatively, we might also try giving it a further +0.1 reward whenever it makes progress towards the goal. Certain choices of rewards may allow an agent to learn orders of magnitude faster; other choices may cause the agent to learn highly suboptimal solutions. The choice of reward can have a significant impact on the performance of our algorithms, and in Chapter 3, we will explore the freedom we have in choosing the reward function, and seek to understand how we might choose R to accelerate learning, but without compromising the quality of the solutions found.

Sometimes, the undiscounted setting of $\gamma = 1$ is also be of interest. In this case, we need to ensure that the expectation of the rewards in Equation (2.1) is still well-defined. When S is finite, one assumption that guarantees this is to assume that there is a distinguished "absorbing state" s_{end} so that the MDP "stops" if we ever enter this state, and moreover assume that the transition probabilities are so that no matter how an agent chooses the actions, the probability of eventually entering s_{end} is one. (Sometimes, the notation "s_0" is also used to denote an absorbing state rather than the initial state.) If these assumptions holds, we say that the transition probabilities $P_{sa}(\cdot)$ are **proper**.

Lastly, measurability is not a significant issue nor cause for concern in any of the applications we consider, and we will assume that anything that needs to be measurable is measurable, and otherwise ignore the issue of measurability in the presentation of our ideas. Readers interested in a detailed treatment of measurability in Markov chains and in Markov decision processes may refer to, e.g., [31, 45].

2.2 Some MDP properties and algorithms

We now review some standard definitions and results for MDPs, some of which will be useful in the subsequent chapters. Everything presented in this section holds straightforwardly for discounted MDPs with finite state and action spaces; with only small modifications, they also hold for undiscounted, proper, MDPs or for infinite MDPs. Some standard sources for proofs of the results we state here include, e.g., [85, 19, 98, 15, 16, 14].

In this section, we will use the "$R(s,a)$" form of rewards. Given a policy π, define its **value function** $V^\pi : S \mapsto \mathbb{R}$ to be the expected returns (sum of discounted rewards) for taking actions according to π and starting from a state s:

$$V^\pi(s) = \mathbf{E}_\pi[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \cdots | s_0 = s]. \tag{2.3}$$

Closely related to the value function is also the **Q-function**. For a given policy π, the Q-function for it, $Q^\pi : S \times A \mapsto \mathbb{R}$ gives the expected returns for starting in a given state, first taking a specified action, and then following policy π afterwards:

$$Q^\pi(s,a) = \mathbf{E}[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \cdots | s_0 = s, a_0 = a, \forall\, t > 0 \;\; a_t = \pi(s_t)]. \tag{2.4}$$

We also define the **optimal value function** $V^* : S \mapsto \mathbb{R}$ to be the optimal expected sum of rewards for starting from a state s:[1]

$$V^*(s) = \max_\pi V^\pi(s). \tag{2.5}$$

Similarly, the **optimal Q-function** can also be defined as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{2.6}$$

Sometimes, when we are working with multiple MDPs, we will also use a subscript M to indicate that a particular quantity is for a certain MDP M (as in $V^*_M(s), \pi_M$, etc).

The quantities V^* and V^π also satisfy following equations:[2]

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s')V^*(s') \tag{2.7}$$

$$V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s')V^\pi(s') \tag{2.8}$$

These **Bellman equations** give a recursive definition of V^* and V^π. For instance, the term in the max in Equation (2.7) is sum of the expected returns if we take action a (obtaining reward $R(s,a)$ immediately), and then behave optimally from then on (obtaining expected $\gamma V^*(s')$ returns from our successor state s'). Equation (2.7) thus states that $V^*(s)$ is given by picking the best action a, and then behaving optimally afterwards. Equation (2.8) has a similar interpretation: The expected returns under π is the immediate reward plus the future expected returns if we continue to behave according to π.

[1]Here and in the rest of this chapter, for the sake of simplicity we will generally forego distinguishing between max's and sup's, and indicating conditions for various max's and arg max's exist. In all the cases in which a max fails to exist, our arguments may be straightforwardly modified to instead consider an infinite sequence converging to the sup. Of course, in the case of finite MDPs, the max's in Equations (2.5-2.6) will exist and this is not a reason for concern.

[2]In the finite state case, our notation assumes $P_{sa}(\cdot)$ is a probability mass function over S, so that $p(s_{t+1} = s' | s_t = s, a_t = a) = P_{sa}(s')$. If S were continuous and $P_{sa}(\cdot)$ is a density, the sums in the Equations (2.7-2.8) would be replaced by integrals over S. In the fully general case, the summation terms should be replaced by Lebesgue integrals with respect to the measure $P_{sa}(\cdot)$ over the states.

It is also a fact that V^* and V^π (for a given π) not only satisfy Equations (2.7–2.8), but are also the *unique* solutions to these two equations.[3]

Many quantities can be written in terms of either the Q- or the value-functions, and we may usually pick whichever is more convenient to work with. Indeed, they are related via the identities

$$V^*(s) = \max_{a \in A} Q^*(s,a) \tag{2.9}$$

$$V^\pi(s) = Q^\pi(s, \pi(s)), \tag{2.10}$$

and there is an analogous form of Equations (2.7–2.8) for Q- instead of value-functions:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') \max_{a' \in A} Q^*(s', a') \tag{2.11}$$

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') Q^\pi(s', \pi(s')) \tag{2.12}$$

Depending on the initial-state distribution D and what state we start from, one may wonder if certain policies are better for starting from certain states, and other policies better from other states. It is a remarkable fact of MDPs that there exists an optimal policy $\pi^* : S \mapsto A$, so that starting from any state, π^* attains the optimal expected returns:

$$V^{\pi^*}(s) = V^*(s) \text{ for all } s. \tag{2.13}$$

(We will later see that this result unfortunately does not generalize to the POMDP setting, in which the agent does not always know the current state s_t.) Moreover, π^* is given by

[3]In the case of infinite MDPs, they are the unique *bounded* solutions to these equations. There may be other, unbounded solutions that are neither V^* not V^π

either of the following (equivalent) definitions:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a) \tag{2.14}$$

$$\pi^*(s) = \arg\max_{a \in A} R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s')V^*(s') \tag{2.15}$$

Note that while an agent knowing Q^* can thus easily compute the optimal policy, finding the optimal policy from V^* is slightly more complicated, and also requires knowledge of $P_{sa}(\cdot)$.

2.3 MDP algorithms

We now briefly review some standard algorithms for finding policies for MDPs. As before, readers interested in a more detailed treatment may refer to the previously referenced texts.

Since a value function (or a Q-function) defines an optimal policy, many algorithms attempt to find either Q^* or V^*. For instance, if the transition probabilities $P_{sa}(\cdot)$ are known, then Equation (2.7) defines a system of equations, the solution of which yields V^*. These equations may either be solved directly via solving a related linear program (e.g., [27, 41, 65]), or by iteratively performing the update

$$V(s) := \max_a R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s')V(s') \tag{2.16}$$

until it converges. The latter of these, a dynamic programming-based algorithm, is called **value iteration**.

Another standard algorithm, **policy iteration**, involves keeping in memory a policy π and an estimated value function V, and iteratively updating π according to Equa-

tion (2.15):

$$\pi(s) := \arg\max_{a \in A} R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s')V(s'), \tag{2.17}$$

and updating V to be the value function V^{π} for π by solving the system of linear equations given by Equation (2.8).

The methods described above assumed that we had knowledge of the state transition probabilities $P_{sa}(\cdot)$. If we do not know the transition probabilities, we may first try to learn them and then run our algorithms using the estimated probabilities. This approach is known as "model-based reinforcement learning." Alternatively, we may also try to learn either the Q- or value function directly, without learning a model (that is, the state transition probabilities) as an intermediate step. This latter approach is called "model-free reinforcement learning." For example, the Q-learning algorithm performs the following update upon seeing a transition from state s to s' when taking action a [106]:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha \left(R(s,a) + \gamma \max_{a' \in A} Q(s',a') \right). \tag{2.18}$$

(Compare this to Equation 2.11.) Here, α is a small number called the learning rate parameter.

In certain applications in which we need to learn online, one might also ask an agent to appropriately tradeoff between trying new actions to learn about them, vs. taking actions already known to be good to accumulate large rewards. This is known as the "exploration vs. exploitation" problem. (E.g., see [53].)

Most of the methods we have described work well for small MDPs, where the state space S is sufficiently small that $V : S \mapsto \mathbb{R}$ can be stored explicitly as a table, with one entry for each state (and likewise for π). For larger MDPs, these methods can be intractable

to apply exactly. Specifically, in many problems, the number of states grows exponentially in the number of state variables. For example, if the state space is $S = \{0,1\}^n$ consisting of n binary state variables, then the number of states is 2^n, and the cost of representing V or π explicitly would be exponentially large in n. Similarly, if we were to apply grid-based discretization to an n-dimensional continuous state space to reduce the problem to one with a finite number of states, we again end up with a number of discretized states that is exponential in n. Bellman called this problem the "curse of dimensionality" [13], and it makes the straightforward application of the simple reinforcement learning algorithms impractical even for many moderate-dimensional problems.

Thus, various approaches have also been proposed for finding approximations to the value function. A few examples of recent work proposing various approaches for doing so in different settings include [58, 42, 27, 41, 28], and this topic remains an area of active research.

The methods described so far all use (approximations to) V^* or Q^* to implicitly define, via Equations (2.14-2.15), a policy π. One alternative is to instead work directly in the space of policies. In Chapter 4, we will explore policy search in significantly greater detail, and also discuss some of its advantages and disadvantages compared to value-function based methods. One well-known example of a policy search algorithm is William's Reinforce algorithm [109]. (Some other, related, algorithms include [57, 9, 12].) Consider a simple 2-state MDP in which we start from s_0, pick an action a_i, receive reward $R(s_0, a_i)$, and transition to a zero-reward absorbing state s_{end}. Suppose further that we have some distribution over actions $p(a_i; \theta)$ smoothly parameterized by θ, and that this is used to define our policy

π. Specifically, $p(a_i;\theta)$ is the probability of our taking action a_i in state s_0. For example, this might be defined using the **softmax** parameterization, $p(a_i;\theta) = \exp(\theta_i)/\sum_j \exp(\theta_j)$. Now, suppose we currently have some choice for θ. Then to increase our expected payoff, we would like to take a hillclimbing step $\theta_i := \theta_i + \alpha \frac{\partial}{\partial\theta_i} V^\pi(s_0)$, where α is a learning rate parameter. We have:

$$\begin{aligned}
\frac{\partial}{\partial\theta_i} V^\pi(s_0) &= \frac{\partial}{\partial\theta_i} \sum_i R(s_0, a_i) p(a_i;\theta) \\
&= \sum_i R(s_0, a_i) \frac{\partial}{\partial\theta_i} p(a_i;\theta) \\
&= \sum_i R(s_0, a_i) p(a_i;\theta) \frac{\partial}{\partial\theta_i} \ln p(a_i;\theta) \\
&= \mathbf{E}_{a_i \sim p(\cdot;\theta)} \left[R(s_0, a_i) \frac{\partial}{\partial\theta_i} \ln p(a_i;\theta) \right].
\end{aligned}$$

Thus, if we act according to our current policy $p(a_i;\theta)$, and for the random a_i we picked, take a small step in the $R(s_0, a_i)\frac{d}{d\theta} \ln p(a_i;\theta)$ direction, we will be taking a **stochastic gradient ascent** step, that on expectation is a step uphill on the objective that we would like to maximize.

While our example used a trivial, single-step MDP, Reinforce (essentially) generalizes the above derivation to work for general problems. While Reinforce can be proved under various conditions to converge to either a local optima or a plateau, empirically, it can be very slow to do so. Indeed, in order to make progress, Reinforce may require up to an amount of sampling/number of steps that is *exponential* in the number of states or in the horizon time (an $\Omega(1/(1-\gamma))$-quantity that is more formally defined in Chapter 4) [49]. More seriously, Reinforce is also limited to *stochastic* policies. Apart from being a nontrivial restriction on the class of policies we are therefore allowed to consider, in safety-critical

applications (such as the helicopter we consider in Chapter 5), it seems very undesirable to add extra randomness to an already-stochastic problem by forcing our policies to randomly pick its actions.

In Chapter 4, we will explore policy search in significantly more detail, and also describe a policy search method that does not suffer from these problems. One of the key ideas is that, unlike Reinforce, which uses data sampled from the MDP once to take a small uphill step and then throws away the data, we will be *reusing* the data we obtain from the MDP. This will allow us to derive more efficient algorithms, for which we can prove nontrivial performance guarantees.

2.4 Partially observable Markov decision processes

So far, our discussion has centered on fully observable Markov decision processes, in which our agent can at each time step t observe its current state s_t. In the more general setting of **Partially observable Markov decision processes** (POMDPs), the agent has some set of possible observations O, and on each step it only sees an observation $o_t = o(s_t)$, where $o : S \mapsto O$ is a deterministic observation function. (The case of stochastic observations is straightforward generalization.) For instance, s may be a vector of state variables, and we may have sensors that measure only a subset of these variables; we will see an example of this in Chapter 5. Alternatively, certain states may be indistinguishable from each other from a sensor reading, in which case the observation $o(s_t)$ may be a list containing the equivalence-class of states that, based on our sensor reading, we might be in.

In the POMDP setting, it becomes significantly harder to find an optimal policy.

Indeed, even finding a near-optimal policy is, depending on the exact assumptions, typically at least NP-hard. (See [63] and references therein for an overview of many of these results.) Unlike the MDP setting, in a POMDP, an agent that knows Q^* will in general still be unable to behave optimally, because it does not always know what the current state s is, and thus cannot consistently pick $\arg\max_a Q^*(s, a)$.

One way to act optimally in a POMDP involves **belief state tracking**. In this approach, we will work with distributions $p(s)$ over states that represent our belief of what state we are in, given the observations we have seen so far and the actions we have taken. Specifically, we can define and compute the optimal Q- or value-functions over these **belief states** (which are distributions over S), and use that to act optimally. This problem essentially reduces the POMDP problem to an MDP one, since the belief states are fully observable. Note however that, assuming S was finite, this method requires computing value functions over probability distributions over S, which are continuous and live in the $(|S| - 1)$-dimensional simplex. Thus, this method has blown up a finite MDP problem to a high-dimensional, continuous/infinite-state one. Methods that attempt to use these ideas to find either optimal or near-optimal policies include [111, 60, 23, 107, 62, 61, 24].

While these belief-state based methods can work well on some problems, they seem difficult to generalize to large POMDPs, such as ones with high dimensional continuous/infinite state spaces. Indeed, if S were continuous, then it will typically be challenging to work with and learn value functions over belief states $p(s)$, which can in general be fairly complicated distributions over the state space. (These belief states can also be approximated using various methods. For instance, McAllester and Singh [68] show how, by

combing the approximate tracking algorithm of [21] with the algorithm of Kearns, Mansour and Ng [51], it is possible to obtain a policy that is computationally expensive to compute, but for which one can obtain non-trivial performance bounds.)

Various heuristics for designing controllers for POMDPs have also been proposed. For instance, we may find the state we are most likely to be in $\hat{s} = \arg\max_s p(s)$, and pick as our actions according to $\arg\max_a Q^*(\hat{s}, a)$. Whether these heuristics work well is highly problem-dependent, and it seems difficult to give general guarantees . (See, e.g., [22], for this and other examples of heuristics.)

Another, promising approach to finding good solutions to POMDPs involves using policy-search based methods. We defer a discussion of these ideas to Chapter 4, in which we will also see why, in many settings, policy-search methods generalize more easily than dynamic programming or value-function-based methods to the setting of POMDPs.

Chapter 3

Shaping in Reinforcement Learning[1]

In sequential decision problems, such as are studied in the dynamic programming and reinforcement learning literatures, the "task" is specified by the *reward function*. Given the reward function and a model of the domain, the optimal policy is determined. An elementary theoretical question that arises is this: What freedom do we have in specifying the reward function, such that the optimal policy remains unchanged?

Reward shaping refers to the practice of modifying the reward function to provide guidance or give "hints" to a learning agent to help it to learn faster. Yet, sometimes seemingly natural choices of shaping rewards can counter-intuitively result in the learning agent giving very poor solutions. In this chapter, we will give examples of how these problems can arise, and present a theory of reward shaping that addresses these problems

[1]Most of the work presented in this chapter first appeared in Ng, Harada and Russell (1999), "Policy invariance under reward transformations: Theory and application to reward shaping," Proceedings of the Sixteenth International Conference on Machine Learning, pp. 278–287.

and gives us sound, systematic ways of modifying reward functions to accelerate learning.

3.1 Changing the reward function

Reward shaping has the potential to be a very powerful technique for scaling up reinforcement learning methods to handle complex problems [29, 66, 86]. (Similar ideas have arisen in the animal training literature; see [91] for a discussion.) Indeed, a very simple pattern of extra rewards often suffices to render straightforward an otherwise intractable problem. However, a difficulty with reward shaping is that by modifying the reward function, it is changing the original problem M to some new problem M', and asking our algorithms to solve M' in the hope that solutions can be more easily or more quickly found there than in the original problem. But, it is not always clear that solutions/policies found for the modified problem M' will also be good for the original problem M. Ideally, we would like the set of optimal policies to be *invariant* to the changes we make to the reward function, so that good policies learned using shaping rewards will still be good for the original problem.

To see why policy invariance is important, consider the following examples of bugs that can arise:

- Randløv and P. Alstrøm describe a system that learns to ride a simulated bicycle to a particular location [86]. To speed up learning, they provided positive rewards whenever the agent made progress towards the goal. The agent learned to ride in tiny circles near the start state because no penalty was incurred for riding away from the goal.

- A similar problem occurred with a soccer-playing robot being trained by David Andre and Astro Teller (personal communication). Because possession of the ball is important in soccer, they provided a reward for touching the ball. The agent learned a policy whereby it remained next to the ball and "vibrated," touching the ball as frequently as possible.

These policies are clearly not optimal for the original MDPs. We would like to provide systematic ways of modifying a reward function while guaranteeing that the resulting learned policies will still be good. How is this possible?

The examples above suggest that the shaping rewards must obey certain conditions if they are not to mislead the agent into learning suboptimal policies. If one had *perfect* advance knowledge of the MDP, then it is possible to exactly characterize the set of all reward transformations that leave the optimal policy (or policies) invariant. Specifically, perfect knowledge of the MDP means that the optimal policy is exactly determined, and there is then a set of linear constraints that exactly characterize all the reward functions that do not change the optimal policy. This observation was the key idea used to derive the "inverse reinforcement learning" algorithms presented in Ng and Russell [78], which seek to recover a reward function given an optimal policy. But given that those methods start off by assuming knowledge of the optimal policy (and significant knowledge about the MDP), they do not seem to be immediately relevant to the more typical reinforcement learning setting in which we are trying to *find* the optimal policy (often starting with little or no information about the reinforcement learning problem).

More generally, we would like, even lacking intimate knowledge of the MDP, to be

able to give a non-trivial set possible changes to the reward function that preserve the optimality of the resulting learned policies. A degenerate special case of this problem is studied in utility theory, which is primarily concerned with single-step decisions. Here, corresponding guarantees for the utility function can be obtained very simply. For single-step decisions without uncertainty, any monotonic transformation on utilities leaves the optimal decision unchanged; with uncertainty, only positive linear transformations are allowed. [104] These results have important implications for designing evaluation functions in games, eliciting utility functions from humans, and many other areas.

To our knowledge, the question of policy invariance under reward function transformations has not been fully explored for sequential decision problems.[2] Policy-preserving transformations are also relevant to the task of of *structural estimation* of MDPs [90], which involves recovering the model and reward function from observed optimal behavior. Policy-preserving transformations determine the extent to which a reward function can be recovered.

Our two earlier examples of the bicycle and the soccer-playing robot had in common the property that the shaping reward caused the agent to choose a cyclical behavior, in which by repeatedly visiting a sequence of states in a cycle, the agent continually accumulates (shaping) reward. The difficulty with these positive-reward cycles leads one to consider rewards derived from a conservative potential—that is, the reward for executing a transition between two states is (essentially) the difference in the value of a potential function applied to each state. It turns out that not only is this a sufficient condition for

[2]Some results are known for *approximate* invariance: If rewards are perturbed by at most ε, the new policy's value is within $2\varepsilon/(1-\gamma)$ of the original optimal policy. [97, 110]

guaranteeing policy invariance under reward transformations, but that, assuming no prior knowledge of the MDP, this is also a *necessary* condition for being able to make such a guarantee.

The rest of this Chapter is structured as follows. In Section 3.2, we introduce our basic shaping framework, which is then used in Section 3.3 to give and prove our main shaping results. Section 3.4 gives some examples of how these results may be used to construct shaping potentials of various kinds and demonstrates their efficacy in speeding up learning on some simple domains. Finally, Section 3.5 shows that shaping allows us to learn using overly "myopic" algorithms (informally, ones that try to optimize only the short-term, rather than long-term, rewards,) and still do well, and closes with a discussion.

3.2 Shaping Rewards

We are trying to learn a policy for some MDP $M = (S, A, \{P_{sa}\}, \gamma, R)$, and wish to help our learning algorithm by giving it additional "shaping" rewards which will hopefully guide it towards learning a good (or optimal) policy faster. To formalize this, we assume that, rather than running our reinforcement learning algorithm on $M = (S, A, \{P_{sa}\}, \gamma, R)$, we will run it on some *transformed* MDP $M' = (S, A, \{P_{sa}\}, \gamma, R')$, where $R' = R + F$ is the reward function in the transformed MDP, and $F : S \times A \times S \mapsto \mathbb{R}$ is a bounded real-valued function called the **shaping reward function**. (Similar to R, the domain of F for the undiscounted case should strictly be $S - \{s_0\} \times A \times S$ where s_0 is a zero-reward absorbing state, but we will not be overly pedantic about this point for now.) So, if in the original MDP M we would have received reward $R(s, a, s')$ for transitioning from s to s' on action

a, then in the new MDP M' we would receive reward $R(s,a,s') + F(s,a,s')$ on the same event.

For any fixed MDP and assuming additive, memoryless shaping reward functions, this $R' = R + F$ is the most general possible form of shaping rewards.[3] Moreover, they cover a fairly large range of the possible shaping rewards one might come up with. For example, to encourage moving towards a goal, a shaping-reward function that one might choose is $F(s,a,s') = r$ whenever s' is closer (in whatever appropriate sense) to the goal than s, and $F(s,a,s') = 0$ otherwise, where r is some positive reward. Or, to encourage taking action a_1 in some set of states S_0, one might set $F(s,a,s') = r$ whenever $a = a_1, s \in S_0$, and $F(s,a,s') = 0$ otherwise.

One important property of this form of reward transformation is that it can generally be *implemented*: In many reinforcement learning applications, we are not explicitly given M as a tuple (S, A, T, γ, R), but are allowed to learn about M only through taking actions in the MDP and by observing the resulting state transitions and rewards. Given such access to M, we can simulate having the same type of access to M' simply by taking actions in M, and then "pretending" we observed reward $R(s,a,s') + F(s,a,s')$ whenever we actually observed reward $R(s,a,s')$ in M. Naturally, the reason that this works is that M and M' use the same actions, states and transition probabilities. Thus, online/offline model-based/model-free algorithms that may be applied to M may in general be readily

[3]It is possible to show a similar result to that given shortly for an even more general, not necessarily additive, form of shaped rewards: $R'(s,a,s') = F(r,s,a,s')$. Here, F is an arbitrary function, and $r = R(s,a,s')$ is the reward we would have received in the original MDP M. It turns out that if we are to give optimality guarantees similar to those we will give here, then under appropriate conditions, the only additional freedom that this gives us in choosing shaping rewards is that it allows us to rescale rewards by any fixed positive factor. Since this does not add any interesting richness to the possible choices for F, we will forgo using this more general formulation.

applied to M' in the same way.

In the sequel, we will use subscript M or M' to distinguish between various quantities computed in the original and in the transformed MDPs. Thus, π^*_M is an optimal policy in M, and $\pi^*_{M'}$ an optimal policy in M', and V^π_M and Q^π_M are the value and Q-functions in M, and so on. Since the cases of discounted ($\gamma < 1$) infinite horizon MDPs and undiscounted ($\gamma = 1$) MDPs with proper transition probabilities require slightly different treatments, we will deal with both of them explicitly in this chapter. For the remainder of this chapter, for simplicity it will be convenient to explicitly consider only MDPs with finite state spaces S, though the generalization offers no difficulties, and we will also give the key (technical) condition needed to generalize these results to MDPs with infinite state spaces.

We are learning a policy for M' in the hope of using it in M. The question at hand is thus the following: For what forms of shaping-reward functions F can we guarantee that $\pi^*_{M'}$, the optimal policy in M', will also be optimal in M? The next section will answer this to a fair degree of generality.

3.3 Main shaping results

In practical applications, we often do not exactly know the state transition probabilities $P_{sa}(\cdot)$ a priori (and may or may not know $R(s, a, s')$). Our goal is therefore, given only S and A (and possibly R), to come up with a shaping-reward function $F : S \times A \times S \mapsto \mathbb{R}$ that is "good" and so that $\pi^*_{M'}$ will be optimal in M. In this section, we will give a form for F under which we can guarantee $\pi^*_{M'}$ will be optimal in M. We also provide a weak converse showing that, without further knowledge of $P_{sa}(\cdot)$ and R, this is the only type of

shaping function that can always give this guarantee.

First ignoring discounting (i.e., letting $\gamma = 1$), let us try to gain some intuition about what F might give rise to the shaping "bug" pointed out earlier. On Randløv and Alstrøm's bicycle task, when the agent was rewarded for riding towards the goal but not punished for riding away from it, it learned to ride in a tiny circle and thereby obtain positive reward whenever it happened to be moving towards the goal. More generally, if there is some sequence of states $s_1, s_2, \ldots, s_n$ such that the agent can travel through them in a cycle ($s_1 \to s_2 \to \cdots \to s_n \to s_1 \to \cdots$), and gain net positive shaping-reward by doing so ($F(s_1, a_1, s_2) + \cdots + F(s_{n-1}, a_{n-1}, s_n) + F(s_n, a_n, s_1) > 0$), then it seems that the agent may be "distracted" from whatever it really should be trying to do (such as ride towards the goal,) and instead try to repeatedly go round this cycle.

To address this difficulty with cycles, a form for F that immediately comes to mind is to let F be a *difference of potentials*: $F(s, a, s') = \Phi(s') - \Phi(s)$, where Φ is some function over states. This way, $F(s_1, a_1, s_2) + \cdots + F(s_{n-1}, a_{n-1}, s_n) + F(s_n, a_n, s_1) = 0$, and we have eliminated the problem of cycles that "distract" the agent. Are there other ways to choose F? And aside from cycles, are there any other problems with shaping that we need to address? It turns out that, without more prior knowledge about P_{sa} and R, such potential-based shaping functions F are the only ones that will guarantee consistency with the optimal policy in M. Moreover, this turns out to be essentially all we need in order to make this guarantee. This is made formal in the following theorem:

Theorem 1 *Let any S, A, γ, and any shaping reward function $F : S \times A \times S \mapsto \mathbb{R}$ be given. We say F is a* **potential-based** *shaping function if there exists a real-valued*

function $\Phi : S \mapsto \mathbb{R}$ *such that for all* $s \in S - \{s_0\}, a \in A, s' \in S$,

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s), \tag{3.1}$$

(where $S - \{s_0\} = S$ *if* $\gamma < 1$*). Then, that* F *is a potential-based shaping function is a necessary and sufficient condition for it to guarantee consistency with the optimal policy (when learning from* $M' = (S, A, \{P_{sa}\}, \gamma, R+F)$ *rather than from* $M = (S, A, \{P_{sa}\}, \gamma, R)$*), in the following sense:*

- *(Sufficiency) If* F *is a potential-based shaping function, then every optimal policy in* M' *will also be an optimal policy in* M *(and vice versa).*
- *(Necessity) If* F *is not a potential-based shaping function (e.g., no such* Φ *exists satisfying Equation (3.1)), then there exist (proper) transition probabilities* P_{sa} *and a reward function* $R : S \times A \mapsto \mathbb{R}$*, such that no optimal policy in* M' *is optimal in* M*.*

The statements of the necessity and sufficiency conditions above might seem a little more complicated than expected, and this is because there can be multiple optimal policies in M or in M'. Nevertheless, it should be clear that the quantifications used make this the strongest possible theorem of this form. The sufficiency condition says that so long as we use a potential-based F, then we are guaranteed any $\pi^*_{M'}$ we might be trying to learn will also be optimal in M. The necessity condition says that if we have no knowledge of P_{sa} and R, then we must choose a potential-based F for learning in M', if we want to guarantee consistency with learning the optimal policy in M. (If we do have intimate knowledge of P_{sa}, R, then the necessity condition does not say much, and it is possible that we might be able to use other shaping functions.)

The proof of necessity is given in Appendix A. Here, we only prove that Equation (3.1) is a sufficient condition: that if F is indeed of the form in (3.1), then we may guarantee that every optimal policy in M' will also be optimal in M.

Proof (of sufficiency): Let F be of the form given in (3.1). If $\gamma = 1$, then since replacing $\Phi(s)$ with $\Phi'(s) = \Phi(s) - k$ for any constant k would not change the shaping rewards F (which is a difference of these potentials), we may, by replacing $\Phi(s)$ with $\Phi(s) - \Phi(s_0)$ if necessary, assume without loss of generality that the Φ used to express F via (3.1) satisfies $\Phi(s_0) = 0$. (Recall that s_0 here is the zero-reward absorbing state.)

For the original MDP M, we know that its optimal Q-function Q^*_M satisfies the Bellman Equations (see, e.g., [99])

$$Q^*_M(s,a) = \mathbf{E}_{s' \sim P_{sa}(\cdot)} \left[R(s,a,s') + \gamma \max_{a' \in A} Q^*_M(s',a') \right] \tag{3.2}$$

Some simple algebraic manipulation then gives us

$$Q^*_M(s,a) - \Phi(s) = \mathbf{E}_{s'} \left[R(s,a,s') + \gamma\Phi(s') - \Phi(s) + \gamma \max_{a' \in A} \left(Q^*_M(s',a') - \Phi(s') \right) \right] \tag{3.3}$$

If we now define $\hat{Q}_{M'}(s,a) \stackrel{\triangle}{=} Q^*_M(s,a) - \Phi(s)$ and substitute that and $F(s,a,s') = \gamma\Phi(s') - \Phi(s)$ back into the previous equation, we get

$$\begin{aligned} \hat{Q}_{M'}(s,a) &= \mathbf{E}_{s'} \left[R(s,a,s') + F(s,a,s') + \gamma \max_{a' \in A} \hat{Q}_{M'}(s',a') \right] \qquad (3.4) \\ &= \mathbf{E}_{s'} \left[R'(s,a,s') + \gamma \max_{a' \in A} \hat{Q}_{M'}(s',a') \right] \qquad (3.5) \end{aligned}$$

But this is exactly the Bellman equation for M'. For the undiscounted case, we moreover have $\hat{Q}_{M'}(s_0,a) = Q^*_M(s_0,a) - \Phi(s_0) = 0 - 0 = 0$. So, $\hat{Q}_{M'}(s,a)$ satisfies the Bellman equations for M', and must in fact be the unique optimal Q-function. Thus, $Q^*_{M'}(s,a) =$

$\hat{Q}_{M'}(s,a) = Q^*_M(s,a) - \Phi(s)$, and the optimal policy for M' therefore satisfies

$$\begin{aligned} \pi^*_{M'}(s) &\in \arg\max_{a \in A} Q^*_{M'}(s,a) && (3.6) \\ &= \arg\max_{a \in A} Q^*_M(s,a) - \Phi(s) && (3.7) \\ &= \arg\max_{a \in A} Q^*_M(s,a) && (3.8) \end{aligned}$$

and is therefore also optimal in M. To show every optimal policy in M is also optimal in M', simply apply the same proof with the roles of M and M' interchanged (and using the shaping function $-F$). This completes the proof. □

We also have the following closely-related result:

Lemma 2 *Under the conditions of Theorem 1, suppose that F does indeed take the form $F(s,a,s') = \gamma\Phi(s') - \Phi(s)$. Suppose further that $\Phi(s_0) = 0$ if $\gamma = 1$. Then for all $s \in S$, $a \in A$,*

$$\begin{aligned} Q^*_{M'}(s,a) &= Q^*_M(s,a) - \Phi(s), && (3.9) \\ V^*_{M'}(s) &= V^*_M(s) - \Phi(s). && (3.10) \end{aligned}$$

Proof: (3.9) was proved in the sufficiency proof above; (3.10) follows immediately from this using the identity $V^*(s) = \max_{a \in A} Q^*(s,a)$. □

Remark 1 (Robustness and learning). Although we have not proved it here, it is straightforward to generalize Lemma 2 to hold for arbitrary policies π, not just the optimal policy: $V^\pi_{M'}(s) = V^\pi_M(s) - \Phi(s)$ (and similarly for Q-functions). A consequence of this is that potential-based shaping is *robust* in the sense that near-optimal policies are also preserved; that is, if we learn a near-optimal policy π in M' (say, $|V^\pi_{M'}(s) - V^*_{M'}(s)| < \varepsilon$)

using potential-based shaping, then π will also be near-optimal in M ($|V_M^\pi(s) - V_M^*(s)| < \varepsilon$). (To see this, apply the identity we just pointed out to policies π and to $\pi_M^* = \pi_{M'}^*$, and subtract.)

Remark 2 (All policies optimal under Φ). To better understand why potential-based F preserve optimal policies, it is worth noting if we have an MDP M that has a potential-based *reward function* $R(s, a, s') = \gamma\Phi(s') - \Phi(s)$, then *any* policy is optimal in M. Thus, potential-based shaping functions are indifferent to policies, in the sense that they give us no reason to prefer any policy over any other; at an intuitive level, this accounts for why they do not give us any reason to prefer any policy other than π_M^* when we switch from M to M'.

We note that in the infinite-state discounted case, in order to guarantee policy invariance, we need an additional (benign) condition that Φ be *bounded*.[4] This is sufficient for ensuring that the rewards $R' = R + F$ in the modified MDP also be bounded, and more importantly is needed for certain steps of the proof of the theorem.[5] For the finite-state case, demanding that Φ be bounded is a vacuous condition since Φ, having a range of finite cardinality, is automatically bounded by the maximum element in its range.

From the theorem, we also recover the familiar result that, with $\gamma < 1$, constant offsets of the reward (i.e., letting $R' = R + c$) leaves the optimal policy (policies) invariant.

[4]Recall that $\Phi : S \mapsto \mathbb{R}$ is bounded iff there exists some constant $B < \infty$ such that $|\Phi(s)| < B$ for all $s \in S$.

[5]Specifically, under standard regularity conditions (e.g., [45, 98]), there is a unique *bounded* solution to the Bellman equations, but there may be many unbounded solutions. That Φ is bounded thus enables us to argue that $V - \Phi$ (or $Q - \Phi$) is also bounded, and is hence the unique bounded solution to the Bellman equations, which was used in the proof to conclude that $Q_{M'}^*(s, a) = \hat{Q}_{M'}(s, a)$. We note that in the context of reinforcement learning, the existence of unbounded solutions to the Bellman equations is also related to the "spurious" solutions discussed in [43].

Specifically, $F = c$ is simply achieved by letting $\Phi(s) = -c/(1-\gamma)$ for all s.

Theorem 1 suggests that we choose shaping rewards of the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$. In applications, Φ should of course be chosen using expert knowledge about the domain. As to how one may do this, Lemma 2 suggests a particularly nice form for Φ, if we know enough about the domain to try choosing it as such. We see that if $\Phi(s) = V_M^*(s)$, (with $\Phi(s_0) = 0$ in the undiscounted case), then Equation (3.10) tells us that the value function in M' is $V_{M'}^*(s) \equiv 0$. This is a particularly easy value function to learn; even lacking a model of the world, all that would remain to be done would be to learn the non-zero Q-values. Though to avoid misconception, we also stress this is not the only way of choosing useful Φ, and that such shaping rewards can help significantly *even if* Φ *is far from* V_M^* (say in the sup-norm), such as by guiding exploration, etc., and we will see examples of this in the next section. But in any case, so long as we choose potential-based F, we have the guarantee that any (near-)optimal policy we learn in M' will also be (near-)optimal in M. Let us now turn our attention to some small experiments that demonstrate how potential-based shaping might be applied in practice.

3.4 Experiments

Much empirical work before us has convincingly justified the use of shaping [66, 86], and we will not bother to try to further justify its use. Here, our goal instead is to show how potential-based shaping functions fit into the picture, and to demonstrate how such shaping functions might be derived in practice.

Towards these goals, we chose for simplicity and clarity to use very simple grid-

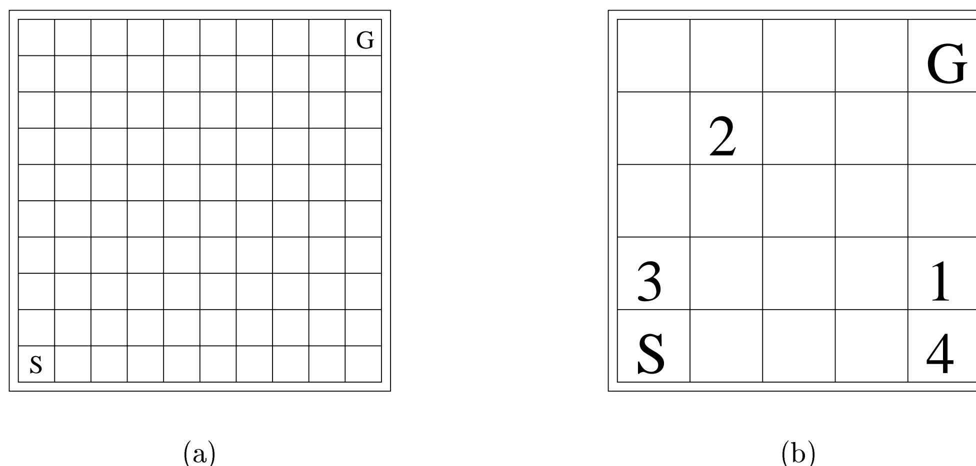
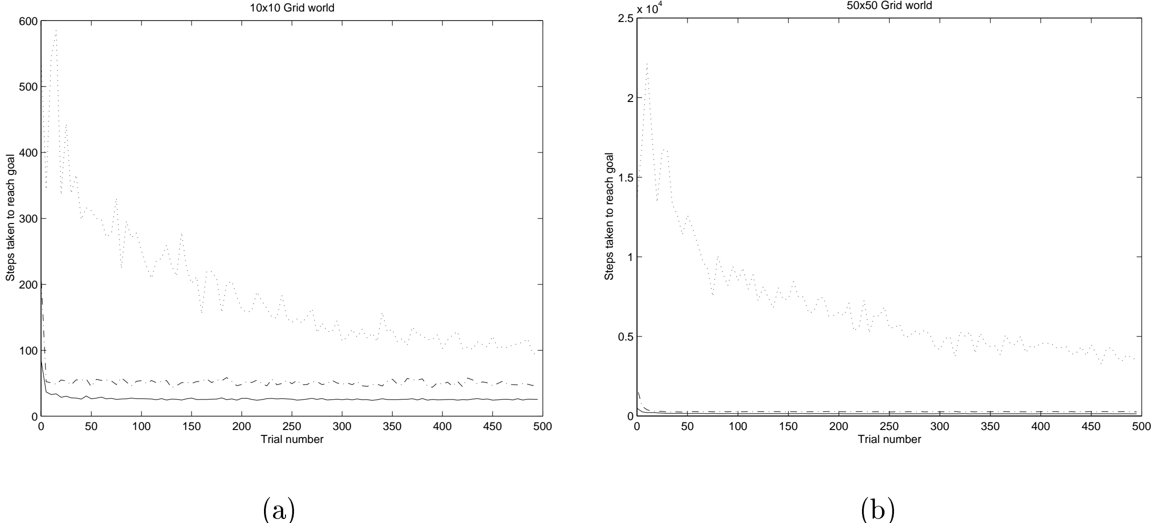


Figure 3.1: (a) 10x10 grid-world in which the agent starts at S and must make its way to the goal G. (b) 5x5 grid-world with 5 subgoals (including goal state), which must be visited in order $1, 2, 3, 4, G$.

world domains to showcase the interesting aspects of potential-based shaping. The first domain was a shortest-path-to-goal 10x10 grid-world (Figure 3.1a), with start and goal states in opposite corners, no discounting, and a -1 per-step reinforcement. Actions are the 4 compass directions, and move 1 step in the intended direction 80% of the time and a random direction 20% of the time, and the agent stays in the same place if it tries to walk off the grid. What might be a good shaping potential $\Phi(s)$? We had pointed out earlier that Equation (3.10) suggests $\Phi(s) = V_M^*(s)$ might be a good shaping potential. So let us now go through the type of reasoning that might suggest a crude estimate of V_M^*; by doing so, we hope to demonstrate how, with a little expert knowledge about distances and the location of the goal, similar reasoning may perhaps be used to similarly derive Φ for other minimum-cost-to-goal problems.

Upon trying to take a step towards the goal, we have an 80% chance of taking the desired step towards the goal, and a 20% chance of a random action. If we take a random

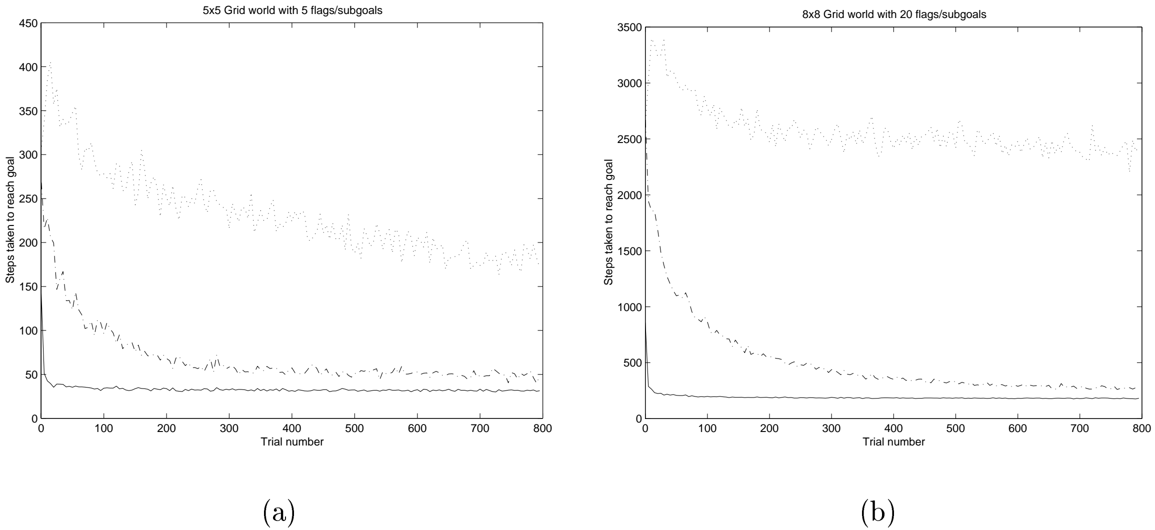


(a) (b)

Figure 3.2: (a) Experiment with 10x10 grid-world. Plot of steps taken to goal vs. trial number. Dotted line is with no shaping; dot-dash line is with $\Phi = 0.5\Phi_0$; solid line is with $\Phi = \Phi_0$. (b) Experiment with 50x50 grid-world.

action, then unless we are at the border of the gridworld, we are as likely to move towards as away from the goal. Hence, from most states, we would expect the optimal policy to make about 0.8 steps of (Manhattan distance) progress towards the goal per timestep, and a crude estimate of the expected number of steps needed to get to the goal from s would be MANHATTAN(s,GOAL)/0.8. Thus, we set our estimate of the value function and therefore $\Phi(s)$ to be $\Phi_0(s) = \hat{V}_M(s) = -\text{MANHATTAN}(s,\text{GOAL})/0.8$. This is what we used as our guess of a "good" shaping function. Also, as a shaping-reward that would be quite far (in the sup-norm) from $V^*_{M'}(s)$, we also tried using $\Phi(s) = 0.5\Phi_0(s)$. The results of this first experiment[6] are shown in Figure 3.2a. (All experiments reported in this section are averages over 40 independent runs.) As can be readily seen, using either of these shaping functions significantly helped speed up learning. Moreover, it is worth re-stressing that even though

[6]The learning algorithm used was Sarsa [99], with 0.10-greedy exploration, and learning rate 0.02. Experiments with Sarsa(λ) also gave analogous results showing shaping significantly speeding up learning.



(a) (b)

Figure 3.3: (a) Results of experiment with 5x5 grid-world with subgoals. Plot of steps taken to goal vs. trial number. Dotted line is no shaping; dot-dash line is with $\Phi = \Phi_0$; solid line is with $\Phi = \Phi_1$. (b) Results of experiment with larger, 8x8 grid-world with more subgoals.

$0.5\Phi_0$ is quite far from V_M^*, it still significantly helped the initial stages of learning. For a larger 50x50 grid-world, the results become even more dramatic: Figure 3.2b shows the result of the same experiment repeated on the larger grid. The plots for Φ_0 and $0.5\Phi_0$ are so low in the graph that they can barely be seen; learning without shaping is clearly losing badly to the potential-based shaping algorithm.

Reiterating, the goal of these experiments was not to try to justify shaping—that has been done far more convincingly by others. Instead, what we have demonstrated here is a style of some very simple reasoning that, by putting together a distance-to-goal heuristic, has enabled us to pick a sensible Φ that dramatically sped up learning.

Next, another class of problems for which a similar style of reasoning might work is domains where we can assign subgoals. Consider the grid-world in Figure 3.1b, where we start in the lower-left hand corner, and must pick up a set of "flags" in sequence before going to the final goal state. Actions and rewards are the same as in the previous grid-world, and

the state-space is expanded to keep track of the collected flags. Since each flag is a subgoal, it is tempting to choose F so that we are rewarded for visiting the subgoals. Let us now see how a potential-function style of reasoning can indeed lead us to choose such an F, and how Equation (3.10) further suggests magnitudes for the subgoal rewards.

With knowledge of the subgoal locations and using reasoning analogous to that suggested earlier (0.8 steps of progress per timestep, etc.), we may estimate the expected number of timesteps, say t, needed to reach the goal. If we imagine that each subgoal is about equally hard to reach from the previous one, then having reached the n-th subgoal, we would still have about $((5-n)/5)t$ steps to go. A slightly more refined argument changes this to $((5-n-0.5)/5)t$ steps (where 0.5 comes from the "typical case" where we are halfway between the n-th and $n+1$-st subgoals), and so our first choice of $\Phi(s)$ is $\Phi_0(s) = -((5-n_s-0.5)/5)t$, where n_s denotes the number of subgoals we have achieved when we are at s. Using this form of shaping-reward function, we see that $\Phi(s) = \Phi_0(s)$ jumps by $t/5$ whenever we reach any subgoal (other than the final goal state), and so the shaping reward function $F(s,a,s') = \Phi(s') - \Phi(s)$ is giving $t/5$ reward for reaching each of these subgoals. This is exactly what our intuition had suggested might be a good shaping reward. For comparison, we also carried out this experiment using a more fine-tuned shaping reward that, similar to the previous grid-world experiments, explicitly estimated the remaining time-to-goal for each state by counting the number of steps it is away from the goal and dividing by 0.8, and using that to construct the corresponding $\Phi_1(s) = \hat{V}_M(s)$ potential function. The result of these experiments are shown in Figure 3.3a, and we see that using our first crude shaping function Φ_0 has allowed us to significantly speed up learning

over not using shaping (and the fine-tuned Φ_1 unsurprisingly gave even better performance). When repeating this experiment on a larger 8x8 domain, the differences become even more dramatic (Figure 3.3b).

3.5 Discussion

We have given several intuitive examples of how shaping can reward "good" behavior and hence accelerate learning. We now show that a well-chosen shaping function can formally make a reinforcement learning "easier" for certain (myopic) learning algorithms, in sense that if a good shaping function is chosen, then we may learn using a reduced discount factor/lookahead horizon time, and still attain near-optimal behavior. What we mean by this is the following: One of the things that makes reinforcement learning interesting and difficult is that one's actions must be chosen to maximize the long-term, rather than only the immediate, rewards. The rate at which future rewards are discounted is controlled by γ, which thus also controls the "lookahead horizon time"—informally, the number of time steps the algorithms must plan ahead in order to do well. (Horizon times are made more formal in Chapter 4.) Intuitively, it is harder to learn in MDPs where γ close to 1, since the algorithm must then lookahead further; indeed, we will later see algorithms whose running time has an explicit (either exponential or polynomial) dependence on $1/(1-\gamma)$, so that a smaller γ means more efficient algorithms. We can show that if a good shaping function is chosen, then it is possible to learn using some $\gamma' < \gamma$ (i.e., use a smaller lookahead horizon), and still attain near-optimal performance.

Recall our previous discussion that a good shaping potential might be one close

to V_M^*. We can show the following result:

Theorem 3 *Let $M = (S, A, \{P_{sa}\}, \gamma, R)$ be an MDP ($\gamma < 1$), and suppose Φ satisfies $|\Phi(s) - V_M^*(s)| \leq \varepsilon$ for all s. Let F be a potential-based shaping reward defined using Φ, and consider learning using some discount $\gamma' < \gamma$ and the shaping rewards F. Specifically, let $\hat{\pi}$ be an optimal policy for the MDP $(S, A, \{P_{sa}\}, \gamma', R + F)$. Then*

$$V_M^{\hat{\pi}}(s) \geq V_M^*(s) - O_\gamma\left((\gamma - \gamma')\varepsilon\right). \tag{3.11}$$

(where the subscript in the big-O notation indicates that it is hiding constants that may depend on γ).

In other words, if we are using shaped rewards, then we may run our learning algorithm using some smaller discount factor $\gamma' < \gamma$ than the "original" γ, and still have the guarantee that, so long as our shaping function was a good one (i.e., ε is not too large), we will obtain a near-optimal policy. Given that many reinforcement learning algorithms will learn or converge faster if run with smaller discount factors, this shows another way that shaping can be used to accelerate learning, without sacrificing much performance. The proof of this Theorem is given in Appendix B.

Previously, we showed necessary and sufficient conditions for a shaping function F to leave optimal policies invariant. Here also are two easy generalizations worth mentioning: Aside from guaranteeing consistency while trying to learn the optimal policy, it is easy to show (by an argument similar to Remark 1 in Section 3.3) that potential-based F also work when trying to learn a good policy from within a *restricted* class of policies, such as in the policy-search framework that we study in the next chapter. Also, for semi-Markov decision processes (SMDPs) where actions take varying amounts of time to complete, Equation (3.1)

unsurprisingly generalizes to $F(s,a,s',\tau) = e^{-\beta\tau}\Phi(s') - \Phi(s)$, where τ is the time the action took to complete, and β is the discount rate.

Finally, the "$\gamma\Phi(s') - \Phi(s)$" form also seems on the surface reminiscent of terms in some of the equations used in Advantage learning [10] and λ-policy iteration [17]. At a very crude level, it turns out that each of them may be thought of as trying to modify Φ so as to gain some computational or representational advantage. If we consider the problem of modifying Φ, then trying to learn a rough shaping function seems to lead quite naturally to an algorithm for multi-scale value-function approximation; specifically, we may use a "course" approximation to Φ, and a "fine" approximation to learn a policy given the shaped rewards. Although it may initially seem unusual to try to *learn* a shaping function, it is the multiscale "rough vs. fine" approximation aspect that this leads to which makes it possibly powerful;[7] this may be an interesting subject for future work.

But more interestingly, the result of Theorem 3 also suggests that it might be fruitful to use a hybrid algorithm in which a value function approximation algorithm is used to approximate V^*_M (and the approximation is then used as Φ); and a policy-search algorithm is run using the shaped rewards and a reduced discount factor, for efficiency. (This also turns out to be closely related to actor-critic methods [99], but here we would run the "actor"/policy search algorithm with a reduced horizon time.)

In this chapter, we have shown that potential-based shaping rewards $\gamma\Phi(s') - \Phi(s)$ leave (near-)optimal policies unchanged. Moreover, potential-based shaping rewards were proved to be the only type of shaping that can guarantee such invariance unless we make

[7] This also relates to the observation that something like a learned shaping reward seems to be operating psychologically—e.g., the capture of a piece in chess operates as a reward even though the underlying MDP has rewards only for checkmate.

further assumptions about the MDP. But just as some practitioners use discounting even on undiscounted problems (perhaps to improve convergence of algorithms), we believe it will also be occasionally reasonable to try shaping rewards that are inspired by potentials, but which are perhaps not strictly of the form we have given. For example, by analogy to using discounting even on undiscounted problems, it is conceivable that for certain problems, it may be easier for an expert to propose a potential Φ for an "undiscounted" shaping function $\Phi(s') - \Phi(s)$, even when $\gamma \neq 1$. Even though our results may no longer guarantee optimality in this case, such a shaping function may, purely from an engineering point of view, still be worth trying, albeit judiciously and with care. In the same spirit, whereas our regularity conditions had demanded using bounded Φ, it is also plausible that some practitioners might want to try certain unbounded Φ. Naturally, if expert knowledge about the domain is available, then non-potential shaping functions might also be fully appropriate.

As guidelines for choosing shaping functions, we have suggested a distance-based heuristic and a subgoal-based heuristic for choosing potentials; because shaping is often crucial to making learning tractable, we believe the task of finding good shaping functions will be a problem of increasing importance.

Appendix 3.A: Proof of necessity

In this Appendix, we sketch the proof of the necessity part of Theorem 1. For brevity, we give the proof only for the case of $|A| = 2$; the generalization is obvious but more tedious. We begin with the following Lemma.

Lemma 4 *If there exist $s \in S - \{s_0\}$, $s' \in S$ and $a, a' \in A$ such that $F(s,a,s') \neq F(s,a',s')$, then there exist transition probabilities P_{sa} and a reward function R such that no optimal policy in M' is optimal in M.*

Proof (Sketch). Assume without loss of generality that $F(s,a,s') > F(s,a',s')$, and let $\Delta = F(s,a,s') - F(s,a',s') > 0$. In the undiscounted case, also assume for simplicity that $s \neq s'$. (When $\gamma = 1$, the proof for $s = s'$ is nearly the same, but having to ensure properness just makes it much more tedious.) We then construct M as follows: Let $P_{sa}(s') = P_{sa'}(s') = 1.0$, and let $R(s,a,s') = 0$ and $R(s,a',s') = \Delta/2$. Clearly $\pi^*_M(s) = a'$. On the other hand, since $R' = R + F$, we have $R'(s,a,s') = F(s,a,s')$ and $R'(s,a',s') = \Delta/2 + F(s,a',s') = F(s,a,s') - \Delta/2 < R'(s,a,s')$, and hence $\pi^*_{M'}(s) = a$. □

We are now ready to show the main necessity result.

Proof (of necessity). Assume F is not potential-based. We need to show we can construct P_{sa}, R such that no optimal policy $\pi^*_{M'}$ in M' is also optimal in M. By Lemma 4, if $F(s,a,s')$ depends on a, we are done; hence we need only consider shaping functions of the form $F(s,a,s') = F(s,s')$ (which do not depend on a).

If $\gamma = 1$, let $\hat{s}_0 = s_0$ be the distinguished absorbing state; otherwise let $\hat{s}_0$ be some fixed state. Noting that constant offsets of the reward do not affect the optimal policy when $\gamma < 1$, we may, by replacing all $F(s,s')$ with $F(s,s') - F(\hat{s}_0,\hat{s}_0)$ if necessary, assume without loss of generality that $F(\hat{s}_0,\hat{s}_0) = 0$. Now define $\Phi(s) = -F(s,\hat{s}_0)$ for all s. By assumption of F not being potential-based, there exist s_1, s_2 such that $\gamma\Phi(s_2) - \Phi(s_1) \neq F(s_1,s_2)$ (let us assume $s_1, s_2, \hat{s}_0$ are distinct; the other cases are either impossible or handled similarly). We then construct M in the following way (still assuming $|A| = 2$). From state s_1, let

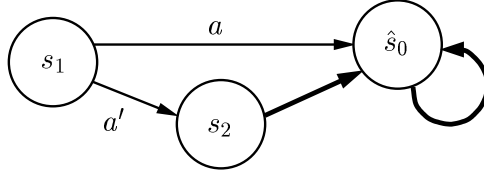


Figure 3.4: The unlabeled thick edges correspond to both actions. All edges have probability 1. The edge $(s_1, a, \hat{s}_0)$ carries a reward $\Delta/2$, and all other edges have zero reward.

$P_{s_1 a}(\hat{s}_0) = P_{s_1 a'}(s_2) = 1.0$, and from states s_2 and $\hat{s}_0$ let both actions a and a' lead to $\hat{s}_0$ with probability 1. Also define $\Delta = F(s_1, s_2) + \gamma F(s_2, \hat{s}_0) - F(s_1, \hat{s}_0)$ and let $R(s_1, a, \hat{s}_0) = \Delta/2$, $R(\cdot, \cdot, \cdot) = 0$ elsewhere. This model is illustrated in Figure 3.4. Then we have

$$Q^*_M(s_1, a) = \frac{\Delta}{2} \tag{3.12}$$

$$Q^*_M(s_1, a') = 0 \tag{3.13}$$

$$Q^*_{M'}(s_1, a) = \frac{\Delta}{2} + F(s_1, \hat{s}_0) \tag{3.14}$$

$$= F(s_1, s_2) + \gamma F(s_2, \hat{s}_0) - \frac{\Delta}{2} \tag{3.15}$$

$$Q^*_{M'}(s_1, a') = F(s_1, s_2) + \gamma F(s_2, \hat{s}_0), \tag{3.16}$$

where we have relied on the fact that $V^*_M(\hat{s}_0) = V^*_{M'}(\hat{s}_0) = 0$ by construction. Hence

$$\pi^*_M(s_1) = \begin{cases} a & \text{if } \Delta > 0, \\ a' & \text{otherwise} \end{cases} \tag{3.17}$$

$$\pi^*_{M'}(s_1) = \begin{cases} a' & \text{if } \Delta > 0, \\ a & \text{otherwise} \end{cases} \tag{3.18}$$

□

Appendix 3.B: Learning with a smaller horizon

In this Appendix, we give a fuller form of, and prove, Theorem 3, which shows that if a "good" shaping function is used, then we may learn using a smaller discount factor/smaller horizon time, and still attain near-optimal behavior.

Let $F(s,a,s') = \gamma\Phi(s') - \Phi(s)$ be a potential-based shaping function, and $R'(s,a,s') = R(s,a,s') + F(s,a,s')$. We previously showed that an MDP using reward R' and discount γ will have the same optimal policy (or policies) as one with rewards R and discount γ. We are now interested in the question of what happens if we perform shaping—thus learning with rewards R'—but use some new discount $\gamma' \leq \gamma$. Under γ', future rewards are discounted more heavily. Informally, this corresponds to letting our learning algorithm be more "myopic," so that it looks ahead fewer steps.

Whereas we had previously considered transforming the reward function of an MDP from R to R', we are now also interested in transforming the discount factor from γ to γ'. Modifying our previous notation, we will use "R, γ," "R, γ'," etc. subscripts to denote quantities from each of the four MDPs arising from combinations of the two rewards and two discount factors. For instance, $V^*_{R',\gamma}$ is the optimal value function in an MDP using reward function R' and discount γ (and the same state space, actions, and state transition probabilities as the original MDP). Similarly, $\pi^*_{R',\gamma}$ is an optimal policy in the MDP that uses R' and γ; and $V^{\pi^*_{R',\gamma}}_{R',\gamma'}$ is the value function for the policy $\pi^*_{R',\gamma}$ in the MDP using R', γ'. Note also that, by Theorem 1, we have that

$$\pi^*_{R',\gamma} = \pi^*_{R,\gamma} \,. \tag{3.19}$$

This says that optimal policies in the (R', γ)- and the (R, γ)-MDPs are the same. Restating

part of Remark 2 of Section 3.3, we also have that for any π,

$$V^{\pi}_{R',\gamma} = V^{\pi}_{R,\gamma} - \Phi. \tag{3.20}$$

The following result (essentially a restatement of Theorem 3) states that if we chose a "good" shaping function (if $\Phi(s)$ is close to $V^*_{R,\gamma}(s)$), then we may learn using some smaller horizon time (i.e., use some $\gamma' < \gamma$) and still attain near-optimal behavior. Note that we are always interested in evaluating policies with respect to the *original*, (R, γ)-MDP. And, the theorem guarantees that value of the policy $\pi^*_{R',\gamma'}$ (the result of learning an optimal policy using the modified rewards and discount factor), as evaluated in the original MDP, will be near-optimal.

Theorem 5 *Suppose* $|\Phi(s) - V^*_{R,\gamma}(s)| \leq \varepsilon$ *for all* $s \in S$, *and let* $\gamma < 1$. *Let* $R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)$, *and* $\gamma' < \gamma$. *Then*

$$V^{\pi^*_{R',\gamma'}}_{R,\gamma}(s) \geq V^*_{R,\gamma}(s) - \frac{2(\gamma - \gamma')\varepsilon}{(1-\gamma')(1-\gamma)} \tag{3.21}$$

Before proving the theorem, we first state, without proof, the following simple fact.

Proposition 6 *Let* $f, g : A \mapsto \mathbb{R}$ *be two real-valued functions with domain* A, *and suppose that* $|f(a) - g(a)| \leq \eta$ *for all* $a \in A$. *Then* $|\max_a f(a) - \max_a g(a)| \leq \eta$.

Proof (of Theorem 5). From Lemma 2, we have that $V^*_{R',\gamma}(s) = V^*_{R,\gamma}(s) - \Phi(s)$ for all s. Since moreover $|V^*_{R,\gamma}(s) - \Phi(s)| \leq \varepsilon$ for all s (by assumption), this shows that for all $s \in S$,

$$|V^*_{R',\gamma}(s)| \leq \varepsilon. \tag{3.22}$$

Let

$$\tau = \max_s |V^*_{R',\gamma}(s) - V^*_{R',\gamma'}(s)| \tag{3.23}$$

$$= \max_s |V^{\pi^*_{R',\gamma}}_{R',\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R',\gamma'}(s)|. \tag{3.24}$$

We will now show a bound on τ. For every s, a, we have that

$$\left|\gamma \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma'}(s')]\right| \tag{3.25}$$

$$= \left|\gamma \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] + \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma'}(s')]\right| \tag{3.26}$$

$$\leq \left|\gamma \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')]\right| + \left|\gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma'}(s')]\right| \tag{3.27}$$

$$= (\gamma - \gamma') \left|\mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')]\right| + \gamma' \left|\mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] - \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma'}(s')]\right| \tag{3.28}$$

$$\leq (\gamma - \gamma')\varepsilon + \gamma'\tau \tag{3.29}$$

This implies that for every s,

$$\left|V^*_{R',\gamma}(s) - V^*_{R',\gamma'}(s)\right| = \left|\left(\max_a \mathbf{E}_{s'\sim P_{sa}(\cdot)}[R'(s,a,s') + \gamma V^*_{R',\gamma}(s')]\right) - \left(\max_a \mathbf{E}_{s'\sim P_{sa}(\cdot)}[R'(s,a,s') + \gamma' V^*_{R',\gamma'}(s')]\right)\right| \tag{3.30}$$

$$\leq (\gamma - \gamma')\varepsilon + \gamma'\tau. \tag{3.31}$$

The second step above used Proposition 6, with

$$f(a) = \mathbf{E}_{s'\sim P_{sa}(\cdot)}[R'(s,a,s')] + \gamma \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma}(s')] \tag{3.32}$$

$$g(a) = \mathbf{E}_{s'\sim P_{sa}(\cdot)}[R'(s,a,s')] + \gamma' \mathbf{E}_{s'\sim P_{sa}(\cdot)}[V^*_{R',\gamma'}(s')], \tag{3.33}$$

and the fact that $|f(a) - g(a)| \leq (\gamma - \gamma')\varepsilon + \gamma'\tau$ for all a (as shown in Equation 3.29).

But since (3.31) holds for *any* state s, it must also be a bound on τ. (See Equation 3.23). Hence,

$$\tau \leq (\gamma - \gamma')\varepsilon + \gamma'\tau. \tag{3.34}$$

Some simple manipulation then allows us to show

$$\tau \leq \frac{(\gamma - \gamma')}{1 - \gamma'}\varepsilon, \tag{3.35}$$

and hence that, for all s,

$$|V^*_{R',\gamma}(s) - V^*_{R',\gamma'}(s)| \leq \frac{(\gamma - \gamma')}{1 - \gamma'}\varepsilon. \tag{3.36}$$

Recall that we are interested in evaluating $\pi^*_{R',\gamma'}$ with respect to the (R, γ)-MDP. Now, $\pi^*_{R',\gamma'}$ is the policy that acts greedily with respect to the value function $V^*_{R',\gamma'}(s)$; and, we have just shown that this value function is close (within $(\gamma - \gamma')\varepsilon/(1 - \gamma')$) to the optimal value function $V^*_{R',\gamma}(s)$ for the (R', γ)-MDP. A standard result (see, e.g., [97]) shows that the value of $\pi^*_{R',\gamma'}$ in the (R', γ)-MDP cannot be far from optimal. Specifically, we have that for all s,

$$V^*_{R',\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R',\gamma}(s) \leq 2\gamma\frac{(\gamma - \gamma')}{(1 - \gamma')(1 - \gamma)}\varepsilon \tag{3.37}$$

But using Equations (3.19-3.20), we also know that

$$\begin{aligned}
V^*_{R',\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R',\gamma}(s) &= V^{\pi^*_{R',\gamma}}_{R',\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R',\gamma}(s) && (3.38)\\
&= V^{\pi^*_{R,\gamma}}_{R',\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R',\gamma}(s) && (3.39)\\
&= \left(V^{\pi^*_{R,\gamma}}_{R,\gamma}(s) - \Phi(s)\right) - \left(V^{\pi^*_{R',\gamma'}}_{R,\gamma}(s) - \Phi(s)\right) && (3.40)\\
&= V^{\pi^*_{R,\gamma}}_{R,\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R,\gamma}(s) && (3.41)\\
&= V^*_{R,\gamma}(s) - V^{\pi^*_{R',\gamma'}}_{R,\gamma}(s). && (3.42)
\end{aligned}$$

Putting this together with (3.37) proves the theorem. $\square$

Lastly, to simplify this result to give Theorem 3, note that since $\gamma' \leq \gamma$, we have that $1/(1-\gamma') \leq 1/(1-\gamma) = O_\gamma(1)$.

Chapter 4

Pegasus: A policy search method for large MDPs and POMDPs[1]

In this chapter, we propose a method for searching within a space of policies for a Markov decision process (MDP) or a partially observable Markov decision process (POMDP). We will focus on the "planning" setting in which we are given a model of the MDP's state transition dynamics. We begin by describing the policy search framework and why naive algorithms do not work well. We then review the trajectory trees method of Kearns, Mansour and Ng [50], which solve some problems faced by the naive methods, but is still intractable for most reasonable problems. We then present the PEGASUS approach to policy search. PEGASUS is based on the following observation: Any (PO)MDP can be transformed into an "equivalent" POMDP in which all state transitions (given the current state and action) are deterministic. Thus, this reduces the general problem of policy search

[1] The PEGASUS algorithm presented this chapter first appeared in Ng and Jordan (2000), "PEGASUS: A policy search method for large MDPs and POMDPs," Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference, pp. 406–415.

to one in which we need only consider POMDPs with deterministic transitions. We give a natural way of estimating the quality of policies in these transformed POMDPs, leading to algorithms for efficiently evaluating and searching in the space of controllers. We also prove guarantees on the performance of our methods.

4.1 Policy Search

As discussed in Chapter 2, a recurring challenge in learning in large MDPs or POMDPs is the "curse of dimensionality." [13] For instance, consider learning in an MDP that has an n-dimensional state space $S = \mathbb{R}^n$ or $S = [0,1]^n$, where n is large. To obtain a controller for such an MDP, one must have a way of representing policies $\pi : S \mapsto A$. How can we represent a function with domain S? One simple method is to discretize the state space by "chopping" it into little grid cells of some edge-width ϵ, and then picking a separate action for each grid cell. Unfortunately, this method has a representational cost that is exponential in the dimension n, making it infeasible even for moderate-sized problems. Note that this problem arises even if S is not continuous. For instance, with a discrete n-dimensional state space $S = \{0,1\}^n$, the cost of representing $\pi : S \mapsto A$ is still exponential n if we use a naive lookup-table representation for π (storing a separate action for each state).

For a more practical way of representing policies, suppose we instead start by picking some restricted, tractably representable, set of functions Π that map S to A. Then, rather than trying to find the "universally optimal" policy among all possible policies mapping from from S to A, suppose we instead restrict our attention to Π, and seek only to find

a good policy within the set Π. Recent years have seen growing interest in such algorithms for *policy search* in large MDPs and POMDPs (e.g., [109, 57, 108, 77, 9, 50]).

As a concrete example, if $S = \mathbb{R}^n$ and $A = \mathbb{R}$, then one possible choice for Π might be the set of all linear functions mapping from the state variables to the real numbers. If there is indeed a linear controller that performs well on the MDP under consideration, then we hope that our learning algorithms will find it or something close to it. As another example, Π may also be the set of all possible functions representable by a medium-sized neural network [69] or any other smoothly parameterized function approximator. The goal of policy search will then be to find a setting of the neural network weights so that when the current state is set as the input into the neural network, it will compute a good choice of action to take at that state.

In contrast to dynamic programming-based algorithms that try to find an approximate value or Q-functions, policy search methods tend to apply straightforwardly to the setting of POMDPs, in which an agent cannot observe the state exactly at each step, and must act using only partial information. Specifically, the policy search framework we have discussed also encompasses cases where our family Π consists of policies that depend only on certain aspects of the state. In particular, to learn in a POMDP, we might use a class Π that contains only policies that depend only on the observables. (E.g., Π may be the set of all linear functions mapping from the observations to the actions.) This results in a class of stochastic memoryless (reactive) policies[2] that can be applied to the POMDP. By introducing artificial "memory variables" into the process state, we can also define limited-

[2] Although we have not explicitly addressed stochastic policies so far, they are a straightforward generalization (e.g. using the transformation to deterministic policies given in [50]).

memory policies [72] (which permits some *belief state* tracking, in which the agent uses past and present observations to try to estimate the true state).

Policy search methods may also enjoy other advantages over dynamic-programming and value-function based solutions. Specifically, policy search methods represent and work with policies directly, without the intermediate step of representing a value function. They can thus more readily exploit the fact that, for many MDPs, the value and Q-functions can be complicated and difficult to approximate, even though there may be simple, compactly representable policies that perform very well. Indeed, the existence of a good, compact representation of a Q-function (via a small neural network, say) implies the existence of a good, compact representation of a policy, because a Q-function defines a policy. In contrast, there is no guarantee that the existence of a good, compact representation of a policy implies a good, compact representation of a value or a Q-function.

Of course, while policy search provides a powerful tool for solving many problems in reinforcement and control, there are also settings in which other methods may be preferred. For instance, policy search typically requires explicitly searching in Π for a good policy. This may be computationally expensive and more prone to local optima than certain dynamic programming-based methods. So if these are issues, and particularly if there is reason to believe that the value function can be easily approximated, then value-function-based approaches would be competitive.

Policy search requires choosing Π. This choice is typically up to the user designing the system. When there is prior knowledge about a likely form of a good controller (such as a belief that there exists a good linear controller), then that can be used to choose Π.

When such prior knowledge is not available, then one may instead use a general function approximator (such as a neural network [69]) for Π.

A useful way of thinking about Π is to consider writing a computer program for controlling the MDP or POMDP, but one that has many parameters left unset. It will be then the role of our learning algorithm to automatically set those parameters. For example, if we were designing a controller to fly a helicopter through a sequence of maneuvers, we might write a program that explicitly loops through the different segments of the trajectory, taking the helicopter through each segment in turn. (We will, in fact, see this done in Chapter 5.) If, however, there are certain parameters that we do not know how to set (say, exactly how hard to pull on the control stick in order to make the helicopter accelerate forward smoothly), then we may leave these parameters undefined and leave the learning algorithm to set them to appropriate values.

This "philosophy" of designing Π via writing "partial programs" has also been espoused in a certain line of research that has attempted to construct specialized programming languages for representing prior knowledge in learning problems, and that then designs specialized algorithms that can take advantage of these representations. For example, Parr's HAMs [81] allow one to specify finite-automata controllers for (fully observed) MDPs; Andre and Russell's PHAMs encompass HAMs and generalizes it to permit more operators [4]; and Thrun's CES [100] is another programming language for machine learning. In contrast to being forced to use these *specialized* languages with all their limitations (e.g., HAMS and PHAMS do not support recursion/for-loops), in our view it is often preferable to allow users to use their favorite, fully expressive, programming language (such as C, C++, Java,

etc.) to specify Π. Despite not placing such restrictions on Π, we will see that, under certain assumptions about the programs (such as that it is "efficient" and terminates using only a small number of operations), we are able to show non-trivial guarantees about the performance of our algorithms.

In the next section, we set up the formalism giving our policy search framework, and describe our policy search strategy. Our eventual goal is an efficient algorithm whose "sample complexity" is polynomial in all quantities of interest. As an intermediate step, we review the "trajectory trees" method of Kearns, Mansour and Ng [50] in Section 4.3. The trajectory tree method requires an exponentially large number of samples, and does not apply to problems with an infinite-dimensional action space. In Section 4.4, we therefore describe the PEGASUS policy search method, which resolves these difficulties. In Section 4.5, we give guarantees on our method's performance; Section 4.6 presents the results of some experiments using these methods; and Section 4.7 closes with a discussion.

4.2 Policy search framework

For the sake of concreteness, we will assume, unless otherwise stated, that $S = [0,1]^{d_S}$ is a d_S-dimensional hypercube. For simplicity, we also assume rewards are deterministic, and written $R(s)$ rather than $R(s,a)$. We will also consider only the case of discounted MDPs ($\gamma < 1$), the extensions being straightforward.

Recall that the **value function** of a policy π is a map $V^\pi : S \mapsto \mathbb{R}$, so that $V^\pi(s)$ gives the expected discounted sum of rewards for executing π starting from state s. To simplify our subsequent notation, define the utility of a policy π, with respect to the

initial-state distribution D, to be

$$U(\pi) = \mathbf{E}_{s_0 \sim D}[V^\pi(s_0)] \tag{4.1}$$

(where the subscript $s_0 \sim D$ indicates that the expectation is with respect to s_0 drawn according to D). Thus, $U(\pi)$ is simply the expected discounted sum of rewards for executing π starting from s_0 drawn according to D. As before, when we are considering multiple MDPs and wish to make explicit that a value function or other quantity is for a particular MDP M, we will use a subscript M, as in $V^\pi_M(s)$, etc.

In the policy search setting, we have some fixed class Π of policies, and desire to find a good policy $\pi \in \Pi$. More precisely, for a given MDP M and policy class Π, define the best possible utility attainable by Π to be[3]

$$opt(M, \Pi) = \max_{\pi \in \Pi} U_M(\pi). \tag{4.2}$$

Our goal is to find a policy $\hat{\pi} \in \Pi$ that gives high utility; i.e., one where $U(\hat{\pi})$ is close to $opt(M, \Pi)$.

4.2.1 Deterministic simulative models

We are interested in the "planning" problem, and will assume that we are given a model of the (PO)MDP.

In order for an algorithm to find a good policy for an MDP (or POMDP), it must have access to, or some information about, the MDP. One common assumption is that a

[3]Here and in the rest of this chapter, we will be not be overly pedantic about the difference between max's and sup's, and whether various max's and arg max's exist. In all the cases in which the max fails to exist, our arguments may be straightforwardly modified to instead consider an infinite sequence converging to the sup.

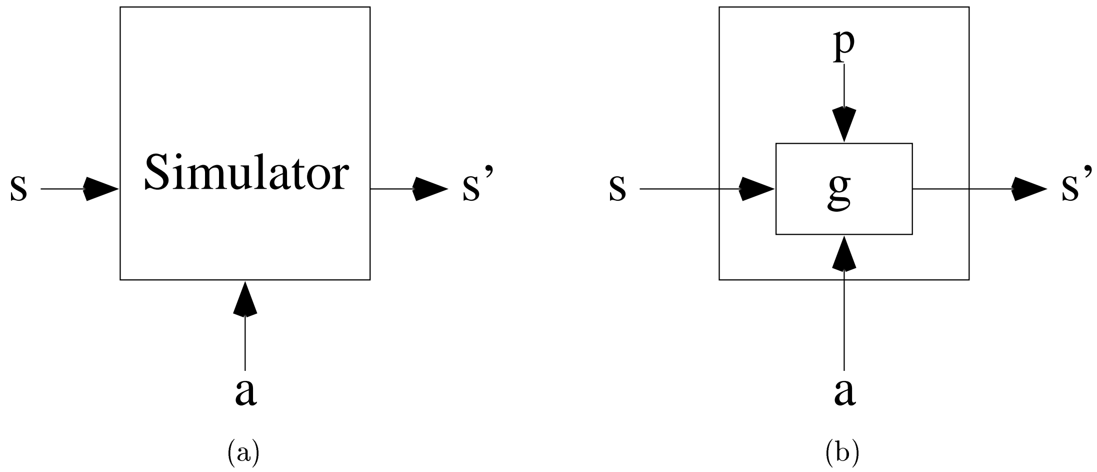


Figure 4.1: (a) A simulator/generative model for an MDP, that takes as input any (s, a)-pair, and outputs $s' \sim P_{sa}(\cdot)$. (b) A typical computer implementation of a simulator, in which a random number generator is called to generate p, and the output s' is computed as a deterministic function $g(s, a, p)$ of p and of the inputs s, a.

learning algorithm has access to the MDP via a generative model or a simulator [50, 98], which is a stochastic function that takes as input any (s, a) state-action pair, and outputs a successor state s' drawn according to $P_{sa}(\cdot)$. (In the POMDP setting, the simulator also returns an observation for the state.) This enables the learner to try actions from arbitrary states. Such a generative model/simulator is most commonly implemented via a computer program that takes as input any state $s \in S$, and action $a \in A$, and randomly samples $s' \sim P_{sa}(\cdot)$. This is depicted in Figure 4.1a.

In this work, we will assume access to such a generative model, but will take the assumption one step further. The only way (essentially) to implement a computer simulator of the form in Figure 4.1a is to write a computer program that takes as input s and a, then makes a call to a random number generator to obtain one (or several, say, d_P), random numbers p, and then computes some function g of the input s, a, and of the random p. Indeed, this is how (almost) all programs are written that need to sample a random variable

from some distribution $P_{sa}(\cdot)$—they compute some deterministic function of their inputs and the random number generator's output. We will assume that our learning algorithm has access not only to the generative model/simulator, but also access to the function g. I.e., we assume that our computer simulator *exposes its interface to the random number generator.* While this initially seems a fairly minor assumption, we will later see how demanding that the simulator expose its interface to the random generator allows us to derive significantly more powerful algorithms.

Since $g : S \times A \times [0,1]^{d_P} \mapsto S$ is a deterministic function, we will refer to it as a **deterministic simulative model**. To draw a sample from $P_{sa}(\cdot)$ for some fixed s and a, we need only draw $\vec{p}$ uniformly in $[0,1]^{d_P}$, and then calculate $g(s, a, \vec{p})$. A deterministic simulative model therefore allows us to simulate a generative model.

Let us examine some simple examples of deterministic simulative models. Suppose that for a state-action pair (s_1, a_1) and some states s' and s'', $P_{s_1 a_1}(s') = 1/3$, $P_{s_1 a_1}(s'') = 2/3$. Then we may choose $d_P = 1$ so that $\vec{p} = p$ is just a real number, and let $g(s_1, a_1, p) = s'$ if $p \leq 1/3$, and $g(s_1, a_1, p) = s''$ otherwise. As another example, suppose $S = \mathbb{R}$, and $P_{sa}(\cdot)$ is a normal distribution with a cumulative distribution function $F_{sa}(\cdot)$. Again letting $d_P = 1$, we may choose g to be $g(s, a, p) = F_{sa}^{-1}(p)$, so that $g(s, a, p)$ is distributed as $P_{sa}(\cdot)$.

It is a fact of probability theory that, given any transition distribution $P_{sa}(\cdot)$, such a deterministic simulative model g can always be constructed for it. (See, e.g. [31].) Indeed, some texts (e.g. [15]) routinely define POMDPs using essentially deterministic simulative models. However, there will often be many different choices of g for representing a (PO)MDP, and it will be up to the simulator's programmer to decide which one is most

"natural" to implement. As we will see later, the particular choice of g that is used can indeed impact the performance of our algorithm, In subsequent sections, one of our goals will be to bound the number of samples required to obtain good performance, and we will see that "simpler" (in a sense to be formalized) implementations of g are generally to be preferred.

4.2.2 Policy search strategy

Our goal is to find or approximate $\arg\max_{\pi\in\Pi} U(\pi)$, the best policy in Π. If we could easily compute $U(\pi)$, we could simply apply a search algorithm to search in Π for the utility-maximizing policy. But

$$U(\pi) = \mathbf{E}_{\pi}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots] \tag{4.3}$$

is the expected sum of discounted rewards for picking actions according to π, and calculating $U(\pi)$ is, in general, intractable.

Our strategy will therefore be to instead find a tractable estimate $\hat{U}$ of U, and then instead optimize $\hat{U}$, as a "proxy" for optimizing U. When will optimizing $\hat{U}$ be "nearly as good" as optimizing U? It is a straightforward to show that if $\hat{U}$ is a *uniformly good* estimate of U—that is, if

$$|\hat{U}(\pi) - U(\pi)| \leq \epsilon \text{ for all } \pi \in \Pi, \tag{4.4}$$

then optimizing $\hat{U}$ will result in a policy whose utility is within 2ϵ of the best possible utility. This result was also used in [50]. Readers may also be familiar with a form of this result commonly seen in the standard supervised learning setting.[4] But for the sake of

[4]E.g., in [54], it is shown that in the supervised learning setting, if we optimize the estimated generalization error of a classifier, and if the estimates are uniformly ϵ-close to the true generalization errors, then we will obtain a classifier within 2ϵ of the best possible error.

completeness, we state and formally prove this result now.

Letting $\hat{\pi} = \arg\max_{\pi\in\Pi} \hat{U}(\pi)$ be the policy we get from optimizing the estimated utilities, we have the following:

Proposition 7 *Let an MDP M be given, and suppose $|\hat{U}(\pi) - U(\pi)| \leq \epsilon$ for all $\pi \in \Pi$. Let $\hat{\pi} = \arg\max_{\pi\in\Pi} \hat{U}(\pi)$. Then*

$$U(\hat{\pi}) \geq opt(M, \Pi) - 2\epsilon. \tag{4.5}$$

Proof. Let $\pi^* = \arg\max_{\pi\in\Pi} U(\pi)$ be the best policy in Π. (So, $U(\pi^*) = opt(M, \Pi)$.) We then have:

$$\begin{aligned} U(\hat{\pi}) &\geq \hat{U}(\hat{\pi}) - \epsilon & (4.6)\\ &\geq \hat{U}(\pi^*) - \epsilon & (4.7)\\ &\geq U(\pi^*) - 2\epsilon & (4.8)\\ &= opt(M, \Pi) - 2\epsilon. & (4.9) \end{aligned}$$

Above, to obtain the first and third inequalities, we used the assumption that $\hat{U}$ is uniformly ϵ-close to U; for the second inequality, we used that fact that $\hat{\pi}$ maximizes $\hat{U}$ so that $\hat{U}(\hat{\pi}) \geq \hat{U}(\pi^*)$. □

Thus, if we can obtain $\hat{U}$ that is a uniformly good estimate of U, then optimizing $\hat{U}$ will give a policy that achieves nearly the best possible utility.

How might one try to obtain good estimates of U? Since we have a simulator for the (PO)MDP, one very natural method is to use Monte Carlo samples. (E.g., [98].) Specifically, to estimate the utility of a policy π, we can use our generative model to simulate controlling the MDP using π. I.e., we first draw $s_0 \sim D$ from the initial-state distribution,

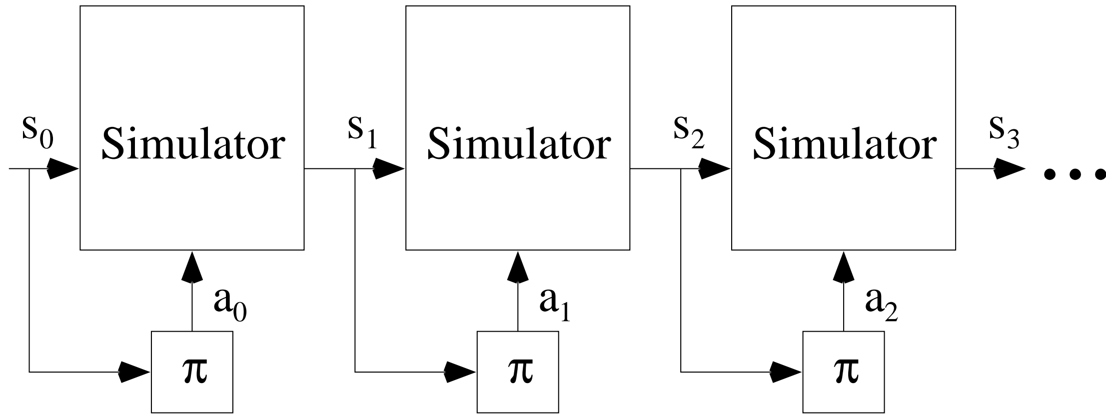


Figure 4.2: Monte Carlo evaluation of a policy π, using a generative model/simulator.

and then we repeatedly find the action $a_t = \pi(s_t)$ chosen by our policy at state s_t, and simulate taking action a_t by sampling a state transition $s_{t+1} \sim P_{s_t,a_t}(\cdot)$. By repeatedly doing this, we obtain a sequence of states $s_0, s_1, s_2, \ldots$. This process is shown in Figure 4.2.

By summing up the discounted rewards along the observed state sequence, we obtain $R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$, which is an unbiased estimate of $U(\pi)$. More generally, we may repeat this entire sampling process m times, and average over the m samples to obtain a better estimate of $U(\pi)$. Thus, letting $\{s_t^{(i)}\}$ denote the i-th sampled state sequence, we define our Monte Carlo estimate of $U(\pi)$ to be

$$\hat{U}(\pi) = \frac{1}{m}\sum_{i=1}^{m} R(s_0^{(i)}) + \gamma R(s_1^{(i)}) + \gamma^2 R(s_2^{(i)}) + \cdots . \tag{4.10}$$

To actually implement this, one last algorithmic detail is that, rather than working with infinite state sequences, we actually truncate the sequence after $H_\epsilon = \lceil \log_\gamma(\epsilon(1-\gamma)/2R_{\max}) \rceil$ steps. Here, H_ϵ is called the ϵ-horizon time, and it is easily verified that (because of discounting, so that rewards in the distant future are given a small weight,) the

truncation introduces at most $\epsilon/2$ error into the approximation.

Monte Carlo gives a simple way of obtaining estimates of policies' utilities. Unfortunately, if $\hat{U} : \Pi \mapsto \mathbb{R}$ is defined this way, then $\hat{U}$ is a stochastic function: Evaluating $\hat{U}$ involves a random sampling procedure, and thus each time we evaluate it, we may obtain a slightly different value. Thus, optimizing $\hat{U}$ to find

$$\hat{\pi} = \text{“}\arg\max_{\pi \in \Pi} \hat{U}(\pi)\text{”} \tag{4.11}$$

represents a non-trivial stochastic optimization problem, which presents a significant algorithmic challenge.

Even more seriously, Monte Carlo evaluation will in general fail to give the uniform convergence guarantee (Equation 4.4) when $|\Pi|$ is infinite.[5] This is true even if we were to average over arbitrarily large numbers m of Monte Carlo samples. We can formalize this in the following proposition:

Proposition 8 *Let $\pi_1, \pi_2, \ldots \subseteq \Pi$ be an infinite sequence of policies. Let there be an MDP that is "sufficiently random" that a single Monte Carlo sample cannot be guaranteed to give a good estimate for $U(\pi_i)$, in the sense that there exist some $B > 0, \delta > 0$ so that for every π_i, it holds true with probability at least δ that*

$$\left|(R(s_0) + \gamma R(s_1) + \ldots + \gamma^H R(s_H)) - U(\pi_i)\right| > B, \tag{4.12}$$

where the probability is over the random state sequence $s_0, \ldots, s_H$ generated by taking actions according to π_i. For each π_i, let $\hat{U}(\pi_i)$ be a Monte Carlo estimate of $U(\pi_i)$, determined by averaging over m sampled state sequences of H steps each. Here, an independent set of

[5] When $|\Pi|$ is finite, then with $m = O(\log|\Pi|)$ samples, the uniform convergence guarantee can be shown to hold with high probability. [50]

samples is used to determine each $\hat{U}(\pi_i)$, so that the $\hat{U}(\pi_i)$'s are mutually independent. Then with probability one,

$$\max_i |\hat{U}(\pi_i) - U(\pi_i)| > B$$

Proof. Consider any fixed π_i. For Equation (4.12) to hold, it must necessarily be the case that either

$$(R(s_0) + \gamma R(s_1) + \ldots \gamma^H R(s_H)) - U(\pi_i) > B \tag{4.13}$$

holds with probability at least $\delta/2$, or

$$(R(s_0) + \gamma R(s_1) + \ldots \gamma^H R(s_H)) - U(\pi_i) < -B \tag{4.14}$$

holds with probability at least $\delta/2$. We assume without loss of generality that the latter case holds, so that the probability of a single Monte Carlo sample being more that B below the true utility is at least $\delta/2$. Now, the probability of m independent samples all giving an estimate that is more than B below $U(\pi_i)$ is therefore at least $(\delta/2)^m$. Hence, with probability at least $(\delta/2)^m$, we have that

$$|\hat{U}(\pi_i) - U(\pi_i)| > B.$$

Let $A_i = 1$ if $|\hat{U}(\pi_i) - U(\pi_i)| > B$, and $A_i = 0$ otherwise. Then we have that $P(A_i = 1) \geq (\delta/2)^m$. Thus, $\sum_{i=1}^{\infty} P(A_i = 1) = \infty$. Moreover, the A_i's are mutually independent random variables. The Borel-Cantelli Lemma (e.g., [31]) states that if there is a sequence of independent events so that the probability of the i-th event is p_i, and if $\sum_{i=1}^{\infty} p_i = \infty$, then with probability one at least one of these events will occur. Hence, we conclude that with probability one, there will be some value of i so that $A_i = 1$, and hence $|\hat{U}(\pi_i) - U(\pi)| > B$ for that value of i. □

Less formally, consider sequentially evaluating a list of policies $\pi_1, \pi_2, \pi_3, \ldots$ via Monte Carlo, where the majority of the policies (say, $\pi_2, \pi_3, \ldots$) are very poorly-performing, risky policies, so that $U(\pi_i)$ for them is small. Because we are evaluating so many policies, with probability one we will eventually obtain a highly unrepresentative sample for some π_i, so that $\hat{U}(\pi_i)$ is very large. Thus, uniform convergence will, with probability one, fail to hold. To see why this is a problem, consider a setting in which π_1 is a good policy ($U(\pi_1)$ is large), but $\pi_2, \pi_3, \ldots$ are poor policies as discussed. Our argument indicates shows that, even if we average over an arbitrarily large m number of samples, we will, with probability 1, pick a policy "$\arg\max_{\pi \in \Pi} \hat{U}(\pi)$" that is significantly worse than π_1. Indeed, in our experiments on an autonomous helicopter (Chapter 5), all of our attempts to use this style of Monte Carlo evaluations resulted in very poor policies being found.

4.2.3 VC dimension and complexity

In the subsequent sections, one of our goals will be to derive algorithms for which we can bound the number of samples m we need to average over in order to derive uniformly good estimates of the utilities. This will allow us to give guarantees of the form in Proposition 7. We now introduce a few definitions and concepts that will later be useful to showing these results.

In supervised learning—say fitting a binary (0/1) classifier to training data—it is well known that the number of training examples needed to fit the parameters "well" is intimately related to the number of free parameters in the model we're fitting. Here, the number of "free parameters" must be defined appropriately. Indeed, since any d real

numbers can be encoded into a single real number,[6] we can reparameterize any d-parameter family of functions into a 1-parameter family of functions. Thus, it is clear that a useful notion of the "number of free parameters" parameterizing a set of functions cannot simply be the obvious one.[7]

The most common way of characterizing the number of free parameters in a set of functions Π is the Vapnik-Chervonenkis (VC) dimension. More precisely, consider a 2-action MDP ($|A| = 2$), and let Π be some set of policies. Π is thus some set of binary functions mapping from S to A. We say that Π **shatters** a set $\{s_1, \ldots, s_m\} \subseteq S$ if for every length-m string of actions $\vec{a} \in A^m$, there is a policy $\pi \in \Pi$ such that $\pi(s_i) = \vec{a}_i$ for every $i = 1, \ldots, m$. We define VC(Π) to be the size of the largest set of states that can be shattered by Π. [102] If Π can shatter arbitrarily large sets, then its VC dimension is infinite. So, if $d = \text{VC}(\Pi)$, there is a set of states $\{s_1, \ldots, s_d\}$ so that by picking an appropriate policy $\pi \in \Pi$, we can realize any combination of actions over these states. Intuitively, a set of functions that can shatter large sets—and hence has high VC dimension—is thus a rich or a "complex" set, whereas one that can shatter only small sets—and hence has small VC dimension—is thus "simple."

As a concrete example, let $A = \{a_1, a_2\}$, and suppose Π consists of the set of linear threshold functions over states $s \in \mathbb{R}^n$, so that

$$\pi(s) = \begin{cases} a_1 & \text{if } \theta^T s + \beta \geq 0, \\ a_2 & \text{otherwise} \end{cases}. \tag{4.15}$$

[6]For example, consider two real numbers $x, y \in [0, 1)$. To encode them into a single real number z, we can take the decimal representation of x and y, and let $z \in [0, 1)$'s digits be alternately taken from x and y. Thus, $x = 0.1234\ldots$ and $y = 0.5678\ldots$ can be encoded into $z = 0.15263748\ldots$, from which we can recover x and y exactly.

[7]It is possible to construct other examples of "complicated" functions that are parameterized by a single real number, that are less contrived than the example given in the previous footnote. (See [102].)

Here, $\theta \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ are the parameters controlling Π. Then, the VC dimension of Π equals $n + 1$. [102] In this case, the VC dimension coincides exactly with the number of parameters of the model. More generally, for many models, the VC dimension turns out to be roughly linear or at most some low-order polynomial in the number of parameters (see, e.g., [5, 40]).

In the setting of supervised learning, if one is trying to fit a function approximator to labeled training data, then it is known that sample complexity—that is, the number of training examples needed to fit our function approximator "well"—is linear in the VC dimension. [102] One of the results of this work will be to generalize these ideas to the setting of reinforcement learning, thus putting reinforcement learning on a more equal footing with simple supervised learning.

To close this section, we introduce one more concept that will be useful later. This concept captures the idea of the family of dynamics that a (PO)MDP and policy class can exhibit. Assume a deterministic simulative model g, and consider some fixed policy $\pi \in \Pi$. If we are executing π from some state s, the successor-state is determined by $f_\pi(s, \vec{p}) = g(s, \pi(s), \vec{p})$, which is a function of s and $\vec{p}$. Varying π over Π, we obtain a whole family of functions $\mathcal{F} = \{f_\pi | f_\pi(s, \vec{p}) = g(s, \pi(s), \vec{p}), \pi \in \Pi\}$ mapping from $S \times [0,1]^{d_P}$ into successor states S. This set of functions $\mathcal{F}$ should be thought of as the family of dynamics realizable by the POMDP and Π, though since its definition does depend on the particular deterministic simulative model g that we have chosen, this is "as expressed with respect to g." Another way of thinking about $\mathcal{F}$ is that its elements are simply the compositions of the transition dynamics (as represented by g) with policies. For each f, also let f_i be the

i-th coordinate function (so that $f_i(s, \vec{p})$ is the i-th coordinate of $f(s, \vec{p})$) and let $\mathcal{F}_i$ be the corresponding families of coordinate functions mapping from $S \times [0,1]^{d_P}$ into $[0,1]$. Thus, $\mathcal{F}_i$ captures all the ways that coordinate i of the state can evolve.

4.3 The trajectory trees method

We saw in Section 4.2.2 how naive Monte Carlo estimation failed to give (uniformly) good estimates of policies' utilities. In this section, we review the trajectory trees method of Kearns, Mansour and Ng [50], which gives a solution that does not suffer from this problem.

Consider a (PO)MDP with two actions, $A = \{a_1, a_2\}$. A trajectory tree is a data structure that, similar to the Monte Carlo evaluation discussed earlier, allows us to give an estimate of $U(\pi)$. However, the key difference between trajectory trees and Monte Carlo evaluation is that the evaluations will not be independent. Specifically, we will "reuse" the *same* samples to evaluate *different* policies.

A trajectory tree is a binary tree whose nodes are labeled with states. (See Figure 4.3.) A trajectory tree is built as follows. We begin by sampling $s_0 \sim D$, which we use to label the root of a binary tree. We then recursively "simulate" taking actions a_1 and a_2 from each state/node in the tree, and create children nodes labeled with the resulting states. Thus, we label the the a_1/left-child of the root with $s_1 \sim P_{s_0,a_1}(\cdot)$, and the a_2/right-child of the root with $s_1' \sim P_{s_0,a_2}(\cdot)$. The children of the s_1 node are in turn labeled with states drawn from $P_{s_1,a_1}(\cdot)$ and $P_{s_1,a_2}(\cdot)$, and so on. This process is repeated until we have a full binary tree of depth H_ϵ. Note that building a trajectory tree requires only a

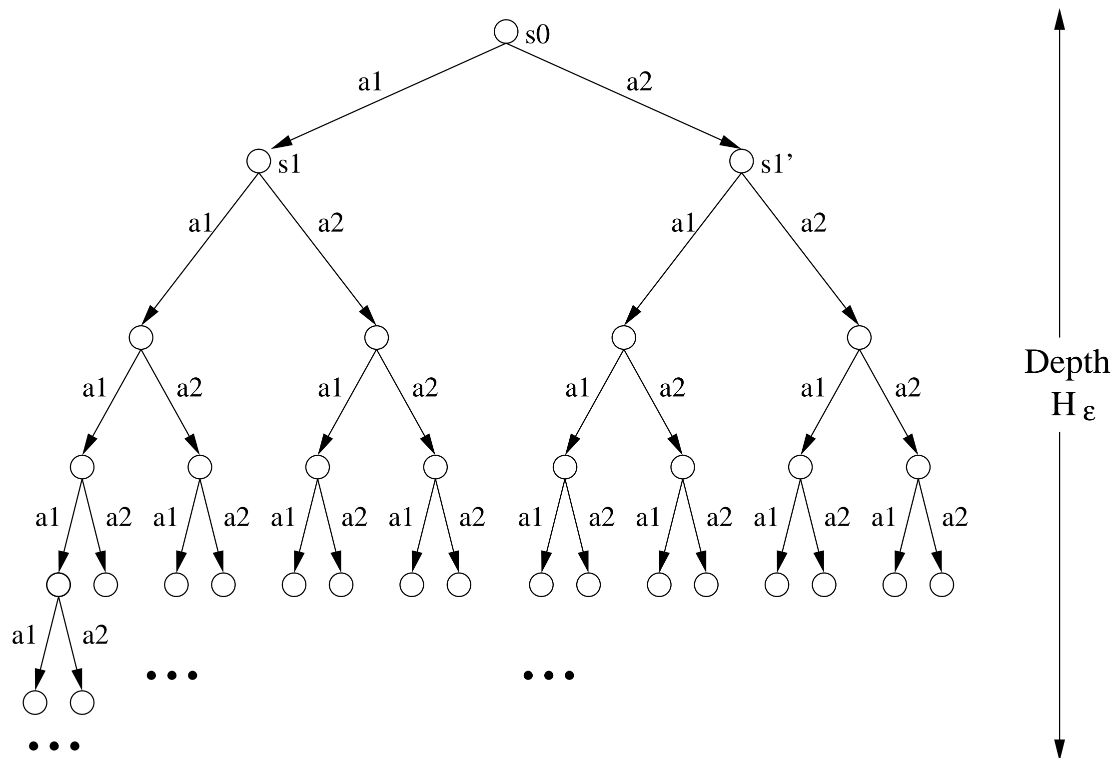


Figure 4.3: A trajectory tree.

simulator/generative model, but not a deterministic simulative model g.

To evaluate a policy π using a trajectory tree, we start at the root, and depending on whether $\pi(s_0)$ equals a_1 or a_2, we follow the path to either the a_1/left or the a_2/right child. From each node, we then repeatedly apply π to the state labeling that node, and follow the path corresponding to the action dictated by π. By continuing this process until it terminates at a leaf of the tree, we traverse a sequence of states $s_0, s_1, \ldots, s_{H_\epsilon}$. The reader can easily verify that for any fixed policy π, this sequence of states has exactly the same distribution as that obtained under the Monte Carlo evaluation method described in

Section 4.2.2. Thus, as before, the sequence $R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$ is an unbiased estimate of $U(\pi)$.[8]

If we construct m such trajectory trees independently, each policy then defines a path $(s_0^{(i)}, s_1^{(i)}, \dots s_{H_\epsilon}^{(i)})$ through each of them ($i = 1, \dots, m$), and we may average over the m resulting samples to obtain an estimate of $U(\pi)$:

$$\hat{U}(\pi) = \frac{1}{m} \sum_{i=1}^{m} R(s_0) + \gamma R(s_1) + \cdots + \gamma^{H_\epsilon} R(s_{H_\epsilon}). \tag{4.16}$$

Building m trees requires drawing only a finite number, about $m2^{H_\epsilon}$, of samples from the generative model/simulator, but the resulting set of trees then defines $\hat{U}(\pi)$ for *all* policies $\pi \in \Pi$, even if Π is infinite. Moreover, it is shown in [50] that the following uniform convergence guarantee holds:

Theorem 9 (Kearns, Mansour and Ng, 1999) *Let a (PO)MDP with actions $A = \{a_1, a_2\}$ be given, and let Π be a class of policies for this POMDP, with Vapnik-Chervonenkis dimension $d = \mathrm{VC}(\Pi)$. Also let any $\epsilon, \delta > 0$ be fixed, and let $\hat{U}$ be the utility estimates determined by using m trajectory trees and a horizon time of H_ϵ. If*

$$m = O\left(\mathrm{poly}\left(d, \frac{R_{\max}}{\epsilon}, \log\frac{1}{\delta}, \frac{1}{1-\gamma}\right)\right), \tag{4.17}$$

then with probability at least $1 - \delta$, $\hat{U}$ will be uniformly close to U:

$$|\hat{U}(\pi) - U(\pi)| \leq \epsilon \textit{ for all } \pi \in \Pi, \tag{4.18}$$

For the sake of completeness, we also include the proof of this theorem in Appendix A. Thus, by Proposition 7, we obtain as a corollary that, with high probability, the policy $\hat{\pi}$

[8]Up to truncation at the horizon time, H_ϵ steps.

obtained by optimizing $\hat{U}$ will have utility nearly as high as that of the best possible policy in Π:

$$U(\hat{\pi}) \geq opt(M, \Pi) - 2\epsilon. \tag{4.19}$$

It is surprising that, with only a *finite* set of samples drawn from the simulator, we obtain enough information to evaluate even an infinite set of policies well. Indeed, the bound in Theorem 9 on the number of trees required has no dependence on how complicated the (PO)MDP's state transitions $P_{sa}(\cdot)$ are, on how complicated the reward function is, or on the size of the state space S.

Informally, by sharing samples to evaluate different policies, we are causing the evaluations of different policies to become *correlated.* The problem with Monte Carlo evaluations was that, by using independent samples, we would almost certainly see at least one highly unrepresentative sample, which causes the uniform convergence condition to be violated (Proposition 8). Here, we draw only a finite set of trees, and so long as this finite set is a representative sample—which Theorem 9 guarantees to be true with high probability—then we are guaranteed uniformly good estimates of policies' utilities. Readers familiar with paired t-tests in statistics, and how they are often more powerful than (non-paired) t-tests, will also recognize this as another example of when having correlated estimates is desirable.

Of course, the number of samples that the trajectory tree method requires is still exponential in the horizon time H_ϵ, because the size of each tree is exponential in H_ϵ. This makes it impractical for most problems. Trajectory trees are also straightforward to generalize to k-action MDPs (by letting each node have a fan-out of k, and modifying Equation (4.17) to have an appropriate dependence on k). But, they do not apply to

problems with infinite action spaces, since then the fan-out of each node would have to be infinite.

In the next section, we describe a different way of estimating policies' utilities that resolves these problems. The method described in the next section is based on certain transformations of (PO)MDPs that, unlike trajectory trees, will require access to a deterministic simulative model g, and not only to a generative model/simulator. However, we will show how, by allowing access to g, we can reduce the exponential dependence on the horizon time to only a polynomial dependence. This makes the algorithm significantly more practical for many applications.

4.4 Policy search method

We show how, given a deterministic simulative model, we can reduce the problem of policy search in an arbitrary POMDP to one in which all the transitions are deterministic—that is, a POMDP in which taking an action a in a state s will always deterministically result in transitioning to some fixed state s'. This reduction is achieved by transforming the original POMDP into an "equivalent" one that has only deterministic transitions.

We may then perform policy search on these "simplified" transformed POMDPs. Specifically, in these simplified POMDPs, it is straightforward to define simple, natural, estimates $\hat{U}(\pi)$ of policies' utilities $U(\pi)$. We call our method PEGASUS (for Policy Evaluation-of-Goodness And Search Using Scenarios, for reasons that will become clear). Our algorithm also bears some similarity to one used in Van Roy [88] for value determination in the setting of fully observable MDPs.

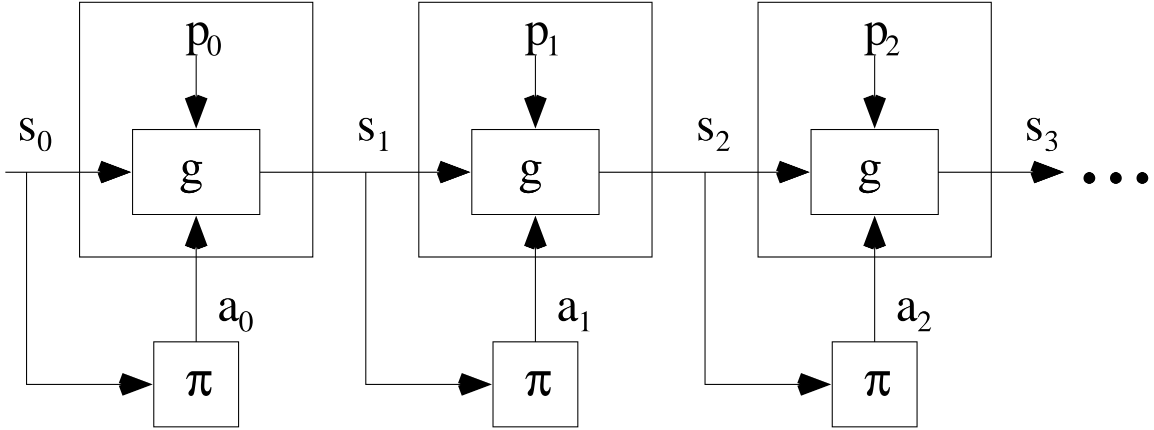


Figure 4.4: Pegasus evaluation of a policy π, using a generative model/simulator.

In Section 4.4.1, we give the transformation of POMDPs to deterministic ones, and in Section 4.4.2, we show how this transformation leads to our policy search algorithm. Theoretical results giving guarantees on the performance of our method are given in Section 4.5.

4.4.1 Transformation of (PO)MDPs

We begin by describing how, given a (PO)MDP $M = (S, D, A, \{P_{sa}(\cdot)\}, \gamma, R)$, a policy class Π and a deterministic simulative model g, we construct our transformed POMDP $M' = (S', D', A, \{P'_{sa}(\cdot)\}, \gamma, R')$ and corresponding class of policies Π'. Our transformed M' will have only deterministic transitions (though its initial state may still be random). To simplify the exposition, we assume $d_P = 1$, so that the terms $\vec{p}$ are just real numbers.

M' is constructed is as follows: The action space and discount factor for M' are the same as in M, so $A' = A$, $\gamma' = \gamma$. The state space for M' is $S \times [0, 1]^\infty$. In other words,

a typical state in M' can be written as a vector $(s, p_1, p_2, \ldots)$ — this consists of a state s from the original state space S, followed by an infinite sequence of real numbers in $[0, 1]$.

The rest of the transformation is straightforward. Upon taking action a in state $(s, p_1, p_2, \ldots)$ in M', we *deterministically* transition to the state $(s', p_2, p_3, \ldots)$, where $s' = g(s, a, p_1)$. In other words, the s portion of the state (which should be thought of as the "actual" state) changes to s', and one number in the infinite sequence $(p_1, p_2, \ldots)$ is used up to generate s' from the correct distribution. By the definition of the deterministic simulative model g, we see that so long as $p_1 \sim$ Uniform$[0, 1]$, then the "next-state" distribution of s' is the same as if we had taken action a in state s (randomization over p_1).

Finally, we choose D', the initial-state distribution over $S' = S \times [0, 1]^\infty$, so that $(s, p_1, p_2, \ldots)$ drawn according to D' has a distribution where $s \sim D$, and the p_i's are distributed independently according to a Uniform$[0, 1]$ distribution. For each policy $\pi \in \Pi$, also let there be a corresponding policy $\pi' \in \Pi'$, given by $\pi'(s, p_1, p_2, \ldots) = \pi(s)$, and let the reward be given by $R'(s, p_1, p_2, \ldots) = R(s)$.

If one observes only the "s"-portion (but not the p_i's) of a sequence of states generated in the transformed POMDP M', starting from the initial-state distribution D' and with actions chosen according to some $\pi' \in \Pi'$, then one obtains a sequence that is drawn from the same distribution as would have been generated from the original (PO)MDP M under the corresponding policy $\pi \in \Pi$. It follows that, for corresponding policies $\pi \in \Pi$ and $\pi' \in \Pi'$, we have that $U_M(\pi) = U_{M'}(\pi')$. This also implies that the best possible expected returns in both (PO)MDPs are the same: $opt(M, \Pi) = opt(M', \Pi')$. It also holds that for any state s, $V_M^\pi(s) = \mathbf{E}_{\{p_i\} \sim \mathrm{Unif}[0,1]}[V_{M'}^{\pi'}(s, p_1, p_2, \ldots)]$

To summarize, we have shown how, using a deterministic simulative model, we can transform any POMDP M and policy class Π into an "equivalent" POMDP M' and policy class Π', so that the transitions in M' are deterministic; i.e., given a state $s \in S'$ and an action $a \in A$, the next-state in M' is exactly determined. Since policies in Π and Π' have the same values, if we can find a policy $\pi' \in \Pi'$ that does well in M' starting from D', then the corresponding policy $\pi \in \Pi$ will also do well for the original POMDP M starting from D. Hence, the problem of policy search in general POMDPs is reduced to the problem of policy search in POMDPs with *deterministic* transition dynamics. In the next section, we show how we can exploit this fact to derive a simple and natural policy search method.

4.4.2 Pegasus: A method for policy search

As discussed, it suffices for policy search to find a good policy $\pi' \in \Pi'$ for the transformed POMDP, since the corresponding policy $\pi \in \Pi$ will be just as good. To do this, we first construct an approximation $\hat{U}_{M'}(\cdot)$ to $U_{M'}(\cdot)$, and then search over policies $\pi' \in \Pi'$ to optimize $\hat{U}_{M'}(\pi')$ (as a proxy for optimizing the hard-to-compute $U_{M'}(\pi)$), and thus find a (hopefully) good policy.

Recall that $U_{M'}$ is given by

$$U_{M'}(\pi) = \mathbf{E}_{s_0 \sim D'}\left[V^{\pi}_{M'}(s_0)\right], \tag{4.20}$$

where the expectation is over the initial state $s_0 \in S'$ drawn according to D'. The first step in our approximation is to replace the expectation over the distribution with a finite sample of states. More precisely, we first draw a sample $\{s_0^{(1)}, s_0^{(2)}, \ldots, s_0^{(m)}\}$ of m initial states according to D'. These states, also called "scenarios" (a term from the stochastic

optimization literature; see, e.g. [18]), define an approximation to $U_{M'}(\pi)$:

$$U_{M'}(\pi) \approx \frac{1}{m}\sum_{i=1}^{m} V^{\pi}_{M'}(s_0^{(i)}). \tag{4.21}$$

Since the transitions in M' are deterministic, for a given state $s \in S'$ and a policy $\pi \in \Pi'$, the sequence of states that will be visited upon executing π from s is exactly determined; hence the sum of discounted rewards for executing π from s is also exactly determined. Thus, to calculate one of the terms $V^{\pi}_{M'}(s_0^{(i)})$ in the summation in Equation (4.21) corresponding to scenario $s_0^{(i)}$, we need only use our deterministic simulative model to find the sequence of states visited by executing π from $s_0^{(i)}$, and sum up the resulting discounted rewards. Naturally, this would be an infinite sum, so as before as truncate the sum after H_ϵ steps.

To summarize, given m scenarios $s_0^{(1)}, \ldots, s_0^{(m)}$, our approximation to $U_{M'}$ is the deterministic function $\hat{U}_{M'} : \Pi' \mapsto \mathbb{R}$

$$\hat{U}_{M'}(\pi) = \frac{1}{m}\sum_{i=1}^{m} R'(s_0^{(i)}) + \gamma R'(s_1^{(i)}) + \cdots + \gamma^{H_\epsilon} R'(s_{H_\epsilon}^{(i)}) \tag{4.22}$$

where $(s_0^{(i)}, s_1^{(i)}, \ldots, s_{H_\epsilon}^{(i)})$ is the sequence of states deterministically visited by π starting from $s_0^{(i)}$. Given m scenarios, this defines an approximation to $U_{M'}(\pi)$ for *all* policies $\pi \in \Pi'$.

The final implementational detail is that, since the states $s_0^{(i)} \in S \times [0,1]^\infty$ are infinite-dimensional vectors, we have no way of representing them (and their successor states) explicitly. But because we will be simulating only H_ϵ steps, we need only represent $p_1^{(i)}, p_2^{(i)}, \ldots, p_{H_\epsilon}^{(i)}$, of the state $s_0^{(i)} = (s^{(i)}, p_1^{(i)}, p_2^{(i)}, \ldots)$, and so we will do just that.

Viewed in the space of the original, untransformed POMDP, evaluating a policy this way also has a simple interpretation. It is similar to the original Monte Carlo method

we had described earlier, with the key difference that the randomization is "fixed" in advance and "reused" for evaluating different π. Specifically, while carrying out Monte Carlo evaluations for the policies, we will always use the *same* sequences of random numbers to evaluate *all* the policies. As in the trajectory tree method, we thus draw one finite set of samples, and use them to evaluate all the policies in our class.

Having used m scenarios to define $\hat{U}_{M'}(\pi)$ for all π, we may now search over policies to optimize $\hat{U}_{M'}(\pi)$. We call this policy search method PEGASUS: Policy Evaluation-of-Goodness And Search Using Scenarios. Since $\hat{U}_{M'} : \Pi' \mapsto \mathbb{R}$ is an ordinary deterministic function, the search procedure only needs to optimize a deterministic function, and any number of standard optimization methods may be used. This should again be contrasted with the simple Monte Carlo setting, which resulted in a stochastic optimization problem (Equation 4.11) that we did not really know how to solve.

In the case that the action space is continuous and $\Pi = \{\pi_\theta | \theta \in \mathbb{R}^\ell\}$ is a smoothly parameterized family of policies (so $\pi_\theta(s)$ is differentiable in θ for all s) then if all the relevant quantities are differentiable, it is also possible to find the derivatives of $\hat{U}_{M'}(\pi_\theta)$ with respect to the parameters θ, and use algorithms such as gradient ascent or conjugate gradient, or higher order methods such as Newton's method, to optimize it. One common barrier to applying these ideas is that R is often discontinuous, being (say) 1 within a goal region and 0 elsewhere. One approach to dealing with this problem is to use a smoothed version of R, possibly in combination with "continuation" methods that gradually unsmooth it again. An alternative approach that may be useful in the setting of continuous dynamical systems is to alter the reward function to use a continuous-time model of discounting. Assuming that

the time at which the agent enters the goal region is differentiable, then $\hat{U}_{M'}(\pi_\theta)$ is again differentiable.[9] Of course, there are still many problems where $\hat{U}_{M'}$ may be discontinuous or non-differentiable. In these cases, other search algorithms or modified versions of gradient-based algorithms can still be applied. (For example, see the discussion in Chapter 5 on the helicopter problem.)

Unlike the trajectory tree method, PEGASUS no longer requires building exponentially large trees. Moreover, it applies readily to infinite-action MDPs, which will be necessary for the helicopter application we consider later. In the next section, we study theoretically how well this method works.

4.5 Main theoretical results

PEGASUS samples a number of scenarios from D', and uses them to form an approximation $\hat{U}(\pi)$ to $U(\pi)$. As before, we are interested in whether, and if so when, $\hat{U}$ will be a uniformly good approximation to U, so that Proposition 7 applies and we are guaranteed that optimizing $\hat{U}$ will result in a policy with utility close to $opt(M, \Pi)$. This section establishes conditions under which this occurs.

4.5.1 The case of finite action spaces

We begin by considering the case of two actions, $A = \{a_1, a_2\}$. For this case, we have the following theorem:

[9] More precisely, if the agent enters the goal region on some time step, then rather than giving it a reward of 1, we figure out what fraction $\tau \in [0, 1]$ of that time step (measured in continuous time) the agent had taken to enter the goal region, and then give it reward γ^τ instead. Assuming τ is differentiable in the system's dynamics, then γ^τ and hence $\hat{U}_{M'}(\pi_\theta)$ are now also differentiable (other than on a usually-measure 0 set, for example from truncation at H_ϵ steps).

Theorem 10 *Let a POMDP with actions* $A = \{a_1, a_2\}$ *be given, and let* Π *be a class of policies for this POMDP, with Vapnik-Chervonenkis dimension* $d = \mathrm{VC}(\Pi)$*. Also let any* $\epsilon, \delta > 0$ *be fixed, and let* $\hat{U}$ *be the utility estimates determined by* PEGASUS *using* m *scenarios and a horizon time of* H_ϵ*. If*

$$m = O\left(\mathrm{poly}\left(d, \frac{R_{\max}}{\epsilon}, \log\frac{1}{\delta}, \frac{1}{1-\gamma}\right)\right), \tag{4.23}$$

then with probability at least $1 - \delta$*,* $\hat{U}$ *will be uniformly close to* U*:*

$$\left|\hat{U}(\pi) - U(\pi)\right| \leq \epsilon \quad \text{for all } \pi \in \Pi \tag{4.24}$$

Proof (sketch). Observe that each scenario in PEGASUS can be viewed as a compact representation of a "trajectory tree," in that it gives a H_ϵ-step Monte Carlo evaluation of every policy. Specifically, given a scenario, we can construct a corresponding trajectory tree (as a deterministic function of that scenario), so that the a_i-child of a node $(s, p_1, p_2, \ldots)$ is $(g(s, a_i, p_1), p_2, \ldots)$, so that the tree gives the same utility estimate on any policy as the scenario. Actually, there is a technical difference between trajectory trees defined this way and trajectory trees as defined previously, in that here different subtrees are not constructed independently (since each level i of the tree was generated from the ancestor nodes using the same random number p_i in $g(s_{\text{parent}}, a, p_i)$); but the proof of Kearns et al. [50] applies without modification to give Theorem 10. □

Using the transformation given in Kearns et al. [50] , the case of a finite action space with $|A| > 2$ also gives rise to essentially the same uniform convergence result, so long as Π has low "complexity."

In Section 4.1, we also discussed designing a policy class Π via writing a partial program. When does a uniform convergence result such as the one given in Theorem 10

apply to policy classes defined this way? It turns out that so long as the "program" implementing Π calculates its output using only a small number of the usual arithmetic operations on the real numbers (that is, $+, -, \times, /$, and jumps based on inequality $=, <, \leq$ tests on pairs of real numbers), and so long as the number of real-valued parameters controlling Π is small, then $VC(\Pi)$ will also be small. Specifically, [40] shows that, if a set of functions Π is parameterized by k real numbers, and if there is a program that evaluates these functions (taking as input the k parameters and s) using at most T of the operations mentioned above, then $VC(\Pi)$ will be at most $O(Tk)$. Thus, Theorem 10 implies that, so long as the program we wrote is "efficient" (terminating in a small number of operations) and does not have too many parameters, then with a small number of samples m, our estimates of the utilities of the policies in Π will be uniformly good.

Again, we also note that the bound given in the theorem has *no dependence* on the size of the state space or on the "complexity" of the POMDP's transitions and rewards. Thus, so long as Π has low VC-dimension, uniform convergence will occur, independently of how complicated the POMDP is. As in Kearns et al., this theorem therefore recovers the best analogous results in supervised learning, in which uniform convergence occurs so long as the hypothesis class has low VC-dimension, regardless of the size or "complexity" of the underlying space and target function.

4.5.2 The case of infinite action spaces: "Simple" Π is insufficient for uniform convergence

We now consider the case of infinite action spaces. Whereas in the 2-action case, Π being "simple" was sufficient to ensure uniform convergence, this is not the case in POMDPs

with infinite action spaces.

Suppose A is a (countably or uncountably) infinite set of actions. A "simple" class of policies would be $\Pi = \{\pi_a | \pi_a(s) \equiv a, a \in A\}$ — the set of all policies that always choose the same action, regardless of the state. Intuitively, this seems to be the simplest policy that actually uses an infinite action space; also, any reasonable notion of complexity of policy classes should assign Π a low "dimension." If it were true that simple policy classes imply uniform convergence, then it is certainly true that this Π should always enjoy uniform convergence. Unfortunately, this is not the case, as we now show.

Theorem 11 *Let A be an infinite set of actions, and let $\Pi = \{\pi_a | \pi_a(s) \equiv a, a \in A\}$ be the corresponding set of all "constant valued" policies. Then there exists a finite-state MDP with action space A, and a deterministic simulative model for it, so that* PEGASUS*' estimates using the deterministic simulative model do not uniformly converge to their means. I.e., there is an $\epsilon > 0$, so that for estimates $\hat{U}$ derived using any finite number m of scenarios and any finite horizon time, there is with probability 1 a policy $\pi \in \Pi$ such that*

$$|\hat{U}(\pi) - U(\pi)| > \epsilon. \tag{4.25}$$

The proof of this Theorem, which is not difficult, is in Appendix B. This result shows that simplicity of Π is not sufficient for uniform convergence in the case of infinite action spaces. However, the counterexample used in the proof of Theorem 11 has a very complex g despite the MDP being quite simple. Indeed, we could have constructed a different, "simpler," choice for g for which uniform convergence would occur.[10] Thus, we might hypothesize that assumptions on the "complexity" of g are also needed to ensure

[10]For example, $g(s_0, a, p) = s_{-1}$ if $p \leq 0.5$, s_1 otherwise; see Appendix B.

uniform convergence in the general case. As we will shortly see, this intuition is roughly correct. Since actions affect transitions only through g, the crucial quantity is actually the composition of policies and the deterministic simulative model — in other words, the class $\mathcal{F}$ of the dynamics realizable in the POMDP and policy class, using a particular deterministic simulative model. In the next section, we show how assumptions on the complexity of $\mathcal{F}$ leads to uniform convergence bounds of the type we desire.

4.5.3 Uniform convergence in the case of infinite action spaces

For the remainder of this section, assume $S = [0,1]^{d_S}$. Then $\mathcal{F}$ is a class of functions mapping from $[0,1]^{d_S} \times [0,1]^{d_P}$ into $[0,1]^{d_S}$, and so a simple way to capture its "complexity" is to capture the complexity of its families of coordinate functions, $\mathcal{F}_i$, $i = 1, \ldots, d_S$. Each $\mathcal{F}_i$ is a family of functions mapping from $[0,1]^{d_S} \times [0,1]^{d_P}$ into $[0,1]$, the i-th coordinate of the state vector. Thus, $\mathcal{F}_i$ is just a family of real-valued functions — the family of i-th coordinate dynamics that Π can realize, with respect to g.

The complexity of a class of boolean functions is measured by its VC dimension, defined to be the size of the largest set shattered by the class. To capture the "complexity" of real-valued families of functions such as $\mathcal{F}_i$, we need a generalization of the VC dimension. The pseudo-dimension, due to Pollard [82], is defined as follows:

Definition (Pollard, 1990). Let $\mathcal{H}$ be a family of functions mapping from a space X into $\mathbb{R}$. Let a sequence of d points $x_1, \ldots, x_d \in X$ be given. We say $\mathcal{H}$ *shatters* $x_1, \ldots, x_d$ if there exists a sequence of real numbers $t_1, \ldots, t_d$ such that the subset of $\mathbb{R}^d$ given by $\{(h(x_1) - t_1, \ldots, h(x_d) - t_d) | h \in \mathcal{H}\}$ intersects all 2^d orthants of $\mathbb{R}^d$ (equivalently, if for any sequence of d bits $b_1, \ldots, b_d \in \{0,1\}$, there is a function $h \in \mathcal{H}$ such that $h(x_i) \geq t_i \Leftrightarrow b_i = 1$,

for all $i = 1, \ldots, d$). The **pseudo-dimension** of $\mathcal{H}$, denoted $\dim_P(\mathcal{H})$, is the size of the largest set that $\mathcal{H}$ shatters, or infinite if $\mathcal{H}$ can shatter arbitrarily large sets.

The pseudo-dimension generalizes the VC dimension, and coincides with it in the case that $\mathcal{H}$ maps into $\{0, 1\}$. We will use it to capture the "complexity" of the classes of the POMDP's realizable dynamics $\mathcal{F}_i$. We also remind readers of the definition of Lipschitz continuity.

Definition. A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is **Lipschitz continuous** (with respect to the Euclidean norm on its range and domain) if there exists a constant B such that for all $x, y \in \text{dom}(f)$, $||f(x) - f(y)||_2 \leq B||x - y||_2$. Here, B is called a **Lipschitz bound**. A family of functions $\mathcal{H}$ mapping from $\mathbb{R}^n$ into $\mathbb{R}$ is **uniformly Lipschitz continuous** with Lipschitz bound B if every function $h \in \mathcal{H}$ is Lipschitz continuous with Lipschitz bound B.

We now state our main theorem, with a corollary regarding when optimizing $\hat{U}$ will result in a provably good policy.

Theorem 12 *Let a POMDP with state space $S = [0, 1]^{d_S}$, and a possibly infinite action space be given. Also let a policy class Π, and a deterministic simulative model $g : S \times A \times [0, 1]^{d_P} \mapsto S$ for the POMDP be given. Let $\mathcal{F}$ be the corresponding family of realizable dynamics in the POMDP, and $\mathcal{F}_i$ the resulting families of coordinate functions. Suppose that $\dim_P(\mathcal{F}_i) \leq d$ for each $i = 1, \ldots, d_S$, and that each family $\mathcal{F}_i$ is uniformly Lipschitz continuous with Lipschitz bound at most B, and that the reward function $R : S \mapsto [-R_{\max}, R_{\max}]$ is also Lipschitz continuous with Lipschitz bound at most B_R. Finally, let $\epsilon, \delta > 0$ be given, and let $\hat{U}$ be the utility estimates determined by* PEGASUS *using m scenarios and a horizon*

time of H_ϵ. If

$$m = O\left(\text{poly}\left(d, \frac{R_{\max}}{\epsilon}, \log\frac{1}{\delta}, \frac{1}{1-\gamma}, \log B, \log\frac{B_R}{R_{\max}}, d_S, d_P\right)\right) \tag{4.26}$$

then with probability at least $1-\delta$, $\hat{U}$ will be uniformly close to U:

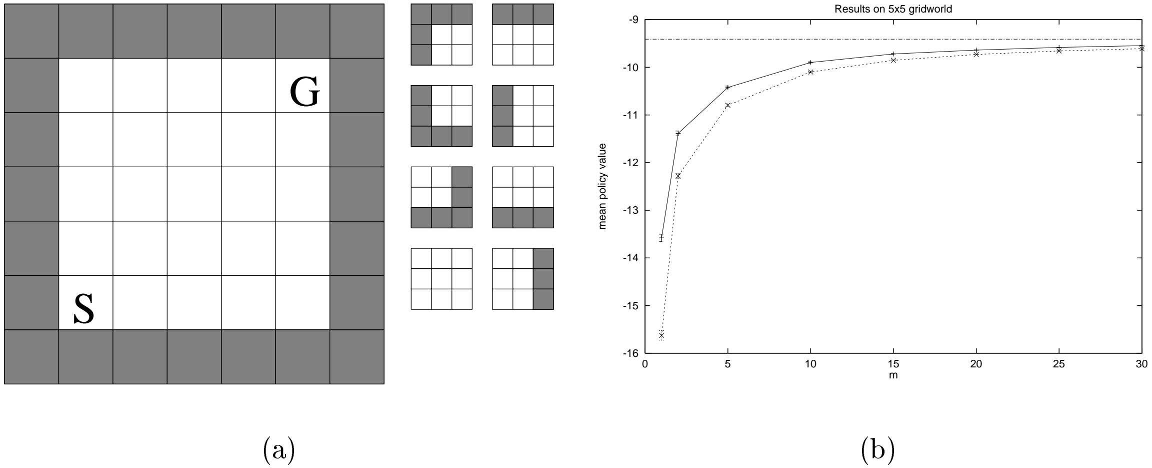
$$\left|\hat{U}(\pi) - U(\pi)\right| \leq \epsilon \quad \text{for all } \pi \in \Pi \tag{4.27}$$

Corollary 13 *Under the conditions of Theorem 10 or 12, let m be chosen as in the Theorem. Then with probability at least $1-\delta$, the policy $\hat{\pi}$ chosen by optimizing the value estimates, given by $\hat{\pi} = \arg\max_{\pi\in\Pi} \hat{U}(\pi)$, will be near-optimal in Π:*

$$U(\hat{\pi}) \geq opt(M, \Pi) - 2\epsilon \tag{4.28}$$

Remark. The (Lipschitz) continuity assumptions give a sufficient, but by no means necessary, set of conditions for the theorem, and other sets of sufficient conditions can be envisaged. For example, if we assume that the distribution on states induced by any policy at each time step has a bounded density, then we can show uniform convergence for a large class of ("reasonable") discontinuous reward functions such as $R(s) = 1$ if $s_1 > 0.5$, $R(s) = 0$ otherwise.[11] Using tools from [40], we can also show similar uniform convergence results without Lipschitz continuity assumptions, by assuming that the family π is parameterized by a small number of real numbers, and that π (for all $\pi \in \Pi$), g, and R are each implemented by a function that calculates their results using only a bounded number of operations (similar to the discussion in the previous section).

[11] Briefly, this is done by constructing two Lipschitz continuous reward functions R_U and R_L that are "close to" and which upper- and lower-bound R (and which hence give value estimates that also upper- and lower-bound our value estimates under R); using the assumption of bounded densities to show our values under R_U and R_L are ϵ-close to that of R; applying Theorem 12 to show uniform convergence occurs with R_U and R_L; and lastly deducing from this that uniform convergence occurs with R as well.



(a) (b)

Figure 4.5: (a) 5x5 gridworld, with the 8 observations. (b) PEGASUS results using the normal and complex deterministic simulative models. The topmost horizontal line shows the value of the best policy in Π; the solid curve is the mean policy value using the normal model; the lower curve is the mean policy value using the complex model. The (almost negligible) 1 s.e. bars are also plotted.

The proof of Theorem 12, which uses techniques first introduced by Haussler [44] and Pollard [82], is quite lengthy, and is deferred to Appendix C.

4.6 Experiments

In this section, we report the results from two experiments. The first, run to examine the behavior of PEGASUS parametrically, involves a simple gridworld POMDP. The second studies a complex continuous state/continuous action problem involving riding a bicycle.

Figure 4.5a shows the finite state and action POMDP used in our first experiment. In this problem, the agent starts in the lower-left corner, and receives a -1 reinforcement per step until it reaches the absorbing state in the upper-right corner. The eight possible observations, also shown in the figure, indicate whether each of the eight squares adjoining

the current position contains a wall. The policy class is small, consisting of all $4^8 = 65536$ functions mapping from the eight possible observations to the four actions corresponding to trying to move in each of the compass directions. Actions are noisy, and result in moving in a random direction 20% of the time. Since the policy class is small enough to exhaustively enumerate, our optimization algorithm for searching over policies was simply exhaustive search, trying all 4^8 policies on the m scenarios, and picking the best one. Our experiments were done with $\gamma = 0.99$ and a horizon time of $H = 100$, and all results reported on this problem are averages over 10000 trials. The deterministic simulative model was

$$g(s,a,p) = \begin{cases} \delta(s,\mathrm{up}) & \text{if } p \le 0.05 \\ \delta(s,\mathrm{left}) & \text{if } 0.05 < p \le 0.10 \\ \delta(s,\mathrm{down}) & \text{if } 0.10 < p \le 0.15 \\ \delta(s,\mathrm{right}) & \text{if } 0.15 < p \le 0.20 \\ \delta(s,a) & \text{otherwise} \end{cases}$$

where $\delta(s,a)$ denotes the result of moving one step from s in the direction indicated by a, and is s if this move would result in running into a wall.

Figure 4.5b shows the result of running this experiment, for different numbers of scenarios. The value of the best policy within Π is indicated by the topmost horizontal line, and the solid curve below that is the mean policy value when using our algorithm. As we see, even using surprisingly small numbers of scenarios, the algorithm manages to find good policies, and as m becomes large, the value also approaches the optimal value.

We have previously shown that "complicated" deterministic simulative model g can lead to poor results. For each (s,a)-pair, let $h_{s,a} : [0,1] \mapsto [0,1]$ be a hash function

that maps any Uniform$[0,1]$ random variable into another Uniform$[0,1]$ random variable.[12] Then if g is a deterministic simulative model, $g'(s,a,p) = g(s,a,h_{s,a}(p))$ is another one that, because of the presence of the hash function, is a much more "complex" model than g. (Here, we appeal to the reader's intuition about complex functions, rather than formal measures of complexity.) We would therefore predict that using PEGASUS with g' would give worse results than g, and indeed this prediction is borne out by the results as shown in Figure 4.5b (dashed curve). The difference between the curves is not large, and this is also not unexpected given the small size of the problem.[13]

Our second experiment uses Randløv and Alstrøm's [86] bicycle simulator, where the objective is to ride to a goal one kilometer away. The actions are the torque τ applied to the handlebars and the displacement ν of the rider's center-of-gravity from the center. The six-dimensional state used in [86] includes variables for the bicycle's tilt angle and orientation, and the handlebar's angle. If the bicycle tilt exceeds $\pi/15$, it falls over and enters an absorbing state, receiving a large negative reward. The randomness in the simulator arises from a uniformly distributed term added to the intended displacement of the center-of-gravity. Rescaled appropriately, this became the p term of our deterministic simulative model.

We performed policy search over the following space: We selected a vector $\vec{x}$ of fifteen (simple, manually-chosen but not fine tuned) features of each state; actions were then chosen with sigmoids: $\tau = \sigma(w_1 \cdot \vec{x})(\tau_{\max} - \tau_{\min}) + \tau_{\min}$, $\nu = \sigma(w_2 \cdot \vec{x})(\nu_{\max} - \nu_{\min}) + \nu_{\min}$,

[12] In our experiments, this was implemented by choosing, for each (s,a) pair, a random integer $k(s,a)$ from $\{1,\ldots,1000\}$, and then letting $h_{s,a}(p) = \mathrm{fract}(k(s,a)\cdot p)$, where $\mathrm{fract}(x)$ denotes the fractional part of x.

[13] Theory predicts that the difference between g and g''s performance should be at most $O(\sqrt{\log|\Pi|/m})$; see [50].

where $\sigma(z) = 1/(1+e^{-z})$, and w_1 and w_2 are the parameters. Note that since our approach can handle continuous actions directly, we did not, unlike [86], have to discretize the actions. The initial-state distribution was manually chosen to be representative of a "typical" state distribution when riding a bicycle, and was also not fine-tuned. We used only a small number $m = 30$ of scenarios, $\gamma = 0.998$, $H = 500$, with the continuous-time model of discounting discussed earlier, and a a slightly modified version of gradient ascent to optimize over the weights.[14] Shaping rewards, to reward progress towards the goal, were also used.[15]

We ran 10 trials using our policy search algorithm, testing each of the resulting solutions on 50 rides. Doing so, the median riding distances to the goal of the 10 different policies ranged from about 0.995km[16] to 1.07km. In all 500 evaluation runs for the 10 policies, the worst distance we observed was also about 1.07km. These results are *significantly* better than those of [86], which reported riding distances of about 7km (since their policies often took very "non-linear" paths to the goal), and a single "best-ever" trial of about 1.7km.

4.7 Discussion and related work

In this chapter, we showed how POMDPs may be transformed to deterministic ones, and used the transformation to define a policy search algorithm. Using the ideas of VC dimension and sample complexity familiar from supervised learning, we also proved

[14]Running experiments without the continuous-time model of discounting, we also obtained, using a non-gradient based hillclimbing algorithm, equally good results as those reported here. Our implementation of gradient ascent, using numerically evaluated derivatives, was run with a bound on the length of a step taken on any iteration; this helps the algorithm to avoid problems near $\hat{U}(\pi_\theta)$'s discontinuities.

[15]Other experimental details: The shaping reward was proportional to and signed the same as the amount of progress towards the goal. As in [86], we did not include the distance-from-goal as one of the state variables during training; training therefore proceeding "infinitely distant" from the goal.

[16]Distances under 1km are possible since, as in [86], the goal has a 10m radius.

guarantees on the quality of the solutions found by our method.

The goals of this work are also related in spirit to that of the large literature on variance reduction. (E.g., see [32].) With any Monte Carlo algorithm, variance is typically a source of error that we would like to diminish. Proposition 8 in essence showed that simple Monte Carlo has too large a variance to work well for this problem, and in particular leads to an overly optimistic, biased, estimate for $\max_\pi U(\pi)$. There are a number of standard variance reduction methods, many of which may also be applied in combination with PEGASUS. We briefly review a few examples here; our presentation is based in part on [32], which readers may refer to for more detailed descriptions.

Consider a variant on the bicycle problem in which with 50% chance we are asked to ride to some goal A, and with 50% chance we are asked to ride to some goal B.[17] One may then apply Monte Carlo in which each of the m samples has 50% chance each of selecting each of the two goals. Alternatively, one may set the goal to be goal A in exactly $m/2$ samples, and goal B in the remaining $m/2$ samples. This idea, an instance of stratified sampling, can often be shown to reduce the variance of Monte Carlo estimates. More broadly, there are also variants of stratified sampling where the number of samples allocated to each goal is not necessarily proportional to the probability of that goal, and where the samples are then reweighted accordingly.

Dimension reduction (also called Rao-Blackwellization or conditional Monte Carlo) refers to settings where we sample over only a subset of the variables, and integrate over the remainder. For instance, if we are using Monte Carlo to estimate $E[f(p_1, p_2)]$, but can

[17]More formally, imagine that the state is modified to have an extra bit specifying the goal, where this bit is determined randomly by the initial state distribution D.

calculate $E[f(p_1, p_2)|p_2]$ exactly, then we might sample only p_2 and average together the resulting $E[f(p_1, p_2)|p_2]$'s, rather than sampling both p_1 and p_2. In reinforcement learning problems where this idea applies, dimension reduction may be used (in combination with PEGASUS or not) to reduce variance. Some of these ideas are also explored in [56].

Systematic sampling refers to a setting in which, whenever some random numbers $p_1, \ldots, p_H$ are generated as a sample for Monte Carlo, we also generate and use some other sequence as a deterministic function of $p_1, \ldots, p_H$. For instance, having sampled $p_1, \ldots, p_H$, we may choose always to also include $1-p_1, \ldots, 1-p_H$ as an additional sample. If the Monte Carlo estimates using these two difference sequences have a strong negative correlation, then this method can significantly reduce variance. In the special case of $H = 1$, the particular method that we have described of using p and $1 - p$ is also called antithetic sampling.

Some other standard examples of variance reduction methods include the use of control variates, and randomized quadrature methods. Since they seem (to us) to be more difficult to apply to reinforcement learning problems, we leave the interested reader to refer to [32] for their descriptions.

At the end of the day, our goal in this chapter has been to give accurate estimates of the expectation $U(\pi) = E[R(s_0) + \gamma R(s_1) + \cdots |\pi]$. In addition to random Monte Carlo algorithms for estimating expectations, there are also many deterministic methods, such as numerical integration and quadrature algorithms. (E.g., [38, 32]). Many of these deterministic methods are known not to scale well with the dimension of the problem, but one that we think might be particularly promising for reinforcement learning is the method of quasi-Monte Carlo. (E.g., [83].) Here, rather than using random numbers $p_1, p_2, \ldots$, with

which to estimate the expectation, we use quasi-random numbers. Informally, quasi-random numbers are sequences chosen to be "spread out," so that they hopefully form a more "representative" sample than a random one. While the scalability of quasi-Monte Carlo to high dimensional problems is still a matter of some debate, Traub and Werschulz [101] show encouraging results demonstrating quasi-Monte Carlo to work on some large problems.

Finally, while our approach has focused on defining a good estimate of the U and treating the resulting $\hat{U}$, one may also apply similar ideas to estimate only the gradients (or finite differences) of U with respect to a policy's parameters, and use that in the inner-loop of stochastic gradient ascent to try to maximize U. Some work on infinitesimal perturbation analysis and gradient estimation attempts to characterize conditions under which $\partial\hat{U}(\pi_\theta)/\partial\theta$ is an unbiased estimate for $\partial U/\partial\theta$. If this is holds true, then $\partial\hat{U}(\pi_\theta)/\partial\theta$ may be used in stochastic gradient ascent to try to maximize $U(\pi_\theta)$. [87, 55, 46] (Here, a fresh set of Monte Carlo samples is typically drawn after each gradient ascent step.) Unfortunately, the conditions needed for this to be true typically fail to hold for the reinforcement learning problems that we are interested in, and may be difficult to apply outside a number of specialized applications (such as certain queuing problems) that have strictly continuous dynamics and rewards. For instance, the helicopter described in the next chapter has discontinuous dynamics/rewards, and straightforward implementations of these approaches do very poorly on it. Similarly, our implementation of policy search on the bicycle had to use a modified version of gradient ascent to do well. Nonetheless, we view this work as related in spirit to ours.

To summarize this chapter, we began by describing the policy search problem, and

saw how obtaining uniformly good estimates of policies' utilities plays a key role in finding good policies. We also saw how naive Monte Carlo methods did not work well, and then reviewed the trajectory trees method, which made certain theoretical results possible, but was still computationally too expensive to be practical. We then showed that any POMDP can be transformed into an "equivalent" one in which all transitions are deterministic. By approximating the transformed POMDP's initial state distribution with a sample of scenarios, we defined an estimate for the value of every policy, and finally performed policy search by optimizing these estimates. We also saw how this method was akin to the simple Monte Carlo method, with the key difference that random numbers are shared to evaluate different policies. Conditions were established under which this method gives uniformly good estimates, and experimental results showed our method working well. It is also straightforward to extend these methods and results to the cases of finite-horizon undiscounted reward; undiscounted rewards with proper state transitions and uniformly bounded expected-time-to-absorbing-state from all states; and infinite-horizon average reward with ϵ-mixing time H_ϵ.

Appendix 4.A: Proof of Theorem 9

In this appendix, we give the proof (by Kearns, Mansour and Ng, 1999) of Theorem 9.

The definition of the VC dimension of classes of binary functions was given in Section 4.2.3. There are several ways to generalize that definition to sets of real-valued functions. We now introduce one given in [103]. If $\mathcal{H} = \{h : X \mapsto [-B, B]\}$ is a set of

real-valued functions bounded by B, define $\mathrm{VC}_r(\mathcal{H})$ to be the (conventional) VC dimension of the set of binary functions $\{I(h,r,\cdot) : h \in \mathcal{H}, r \in (-B,B)\}$, where $I(h,r,x) = 1$ if $h(x) \geq r$, $I(h,r,x) = 0$ otherwise. That is, we take $\mathcal{H}$, introduce all possible thresholds to get indicator functions, and finally take the conventional VC dimension of the resulting set of indicators.

Let $V_{\max} = R_{\max}/(1-\gamma)$. For a fixed policy π, define a map $R(\pi,\cdot)$ from trajectory trees to real numbers $[-V_{\max}, V_{\max}]$, given by evaluating that policy π on the trajectory tree, Thus, Π may also be viewed as defining a set of real-valued functions whose domain is the space of trajectory trees, and whose range is $[-V_{\max}, V_{\max}]$. Thus, it makes sense to ask what $\mathrm{VC}_r(\Pi)$ is. (With some abuse of notation, here we are using Π to also denote this set of functions with domain being trees; this is *not* to be confused with the alternative/original view of Π as a set of functions mapping from states to actions.) We have the following lemma.

Lemma 14 *Let Π be a set of policies for a two-action POMDP, with VC dimension $\mathrm{VC}(\Pi)$ when viewed as a set of maps from states to actions. Then when viewed as a set of maps from the space of all depth-H trajectory trees to $[-V_{\max}, V_{\max}]$, the set Π has dimension bounded by*

$$\mathrm{VC}_r(\Pi) = O\left(H\mathrm{VC}(\Pi)\right) \tag{4.29}$$

Proof. Let $d = \mathrm{VC}(\Pi)$. From Sauer's Lemma (see [103]), Π can realize at most $(ek/d)^d$ different action labelings on any set of k states. Now on m trajectory trees, there are at most $k = m2^{(H+1)}$ different states (one for each node). So if we view each π as selecting a path through each tree, then Π can realize at most $(ek/d)^d = (em2^{(H+1)}/d)^d$ different

selections (where each "selection" is a set of m paths taken by a policy π, one per tree). Moreover, this set of trees has a total of $m2^H$ paths from roots to leaves, and so $R(\pi, T)$ can take on at most $m2^H$ values for $\pi \in \Pi$ and T in our set of m trees. Thus, we need consider only $m2^H$ settings of the threshold parameter r.

Multiplying the quantities together, we see therefore that the set of indicator functions used to define $\mathrm{VC}_r(\Pi)$ (where Π is now viewed as a map from trees to $[-V_{\max}, V_{\max}]$) can, on m trees, realize at most $m2^H(em2^{(H+1)}/d)^d$ different labelings. Now in order for Π (viewed a a set of real functions) to shatter m trees, it must be able to realize at least 2^m different labelings, so that

$$m2^H(em2^{(H+1)}/d)^d \geq 2^m \tag{4.30}$$

must hold. A little algebra shows this implies $m = O\left(H\mathrm{VC}(\Pi)\right)$, proving the lemma. $\square$

We now state one more result due to Vapnik [103], after which we will be ready to prove our theorem. Let $\mathcal{H}$ be a set of bounded real-valued functions $h : X \mapsto [-B, B]$ bounded by B, and let $d = \mathrm{VC}_r(\mathcal{H})$. Let D be some distribution over X, and let $x_1, \ldots, x_m$ be m iid samples drawn according to D. Then with probability $1 - \delta$,

$$\sup_{h \in \mathcal{H}} \left| E_D[h(x)] - \frac{1}{m}\sum_{i=1}^{m} h(x_i) \right| \leq O\left(B\sqrt{\frac{d\log\frac{m}{d} + \log\frac{1}{\delta}}{m}} \right) \tag{4.31}$$

holds (where the randomization is over the draw of the x_i's).

We are now ready to prove the theorem.

Proof (of Theorem 9). There are two sources in the error in our estimate of $V^\pi(s_0)$: Error from truncating at depth H, and error from the randomness in the sampling of trajectory trees. Let $V^\pi_{H_\epsilon}(s_0)$ be the expected H_ϵ-step sum of discounted reinforcements for π starting at s_0. Clearly, $\mathbf{E}_T[R(\pi, T)] = V^\pi_{H_\epsilon}(s_0)$ holds for all π. From our choice of H_ϵ, it also holds

by construction that $|V^\pi_{H_\epsilon}(s_0) - V^\pi(s_0)| \leq \epsilon/2$ for all π. Finally, we apply Equation (4.31) with $x = T$ being the trajectory trees, $\mathcal{H} = \Pi$ (viewed as a set of real-valued functions), $B = V_{\max}$, $h = \pi$, $h(x) = R(\pi, T)$, $d = \mathrm{VC}_r(\Pi)$ and $\mathbf{E}_D[h(x)] = \mathbf{E}_T[R(\pi, T)] = V^\pi_{H_\epsilon}(s_0)$, and find that with probability $1 - \delta$,

$$\left| V^\pi_{H_\epsilon}(s_0) - \frac{1}{m}\sum_{i=1}^{m} R(\pi, T_i) \right| \leq O\left(V_{\max}\sqrt{\frac{d \log \frac{m}{d} + \log \frac{1}{\delta}}{m}} \right) \tag{4.32}$$

holds simultaneously for all $\pi \in \Pi$. Substituting $d = O(H_\epsilon \mathrm{VC}(\Pi))$ from Lemma 14, we therefore see that with a choice of

$$m = O\left((V_{\max}/\epsilon)^2 (H_\epsilon \mathrm{VC}(\Pi) \log(V_{\max}/\epsilon) + \log(1/\delta)) \right) \tag{4.33}$$

we have that with probability $1 - \delta$, it holds simultaneously for all $\pi \in \Pi$ that $|V^\pi_{H_\epsilon}(s_0) - (1/m)\sum_{i=1}^m R(\pi, T_i)| \leq \epsilon/2$. When this is true, the triangle inequality gives $|V^\pi(s_0) - (1/m)\sum_{i=1}^m R(\pi, T_i)| \leq |V^\pi(s_0) - V^\pi_{H_\epsilon}(s_0)| + |V^\pi_{H_\epsilon}(s_0) - (1/m)\sum_{i=1}^m R(\pi, T_i)| \leq \epsilon/2 + \epsilon/2 = \epsilon$ simultaneously for all π, which proves the theorem. □

Appendix 4.B: Proof of Theorem 11

Proof (of Theorem 11). We construct an MDP with states s_{-1}, s_0, and s_1 plus an absorbing state. The reward function is $R(s_i) = i$ for $i = -1, 0, 1$. Discounting is ignored in this construction. Both s_{-1} and s_1 transition with probability 1 to the absorbing state regardless of the action taken. The initial-state s_0 has a .5 chance of transitioning to each of s_{-1} and s_1.

We now construct g, which will depend in a complicated way on the $\vec{p}$ term. Let $T = \{\cup_{i=1}^N [a_i, b_i] \,|\, a_i, b_i \in [0, 1] \cap \mathbb{Q}, a_i < b_i, 1 \leq N < \infty\}$ be the countable set of all

finite unions of intervals with rational endpoints in [0,1]. Let T' be the countable subset of T that contains all elements of T that have total length (Lebesgue measure) exactly 0.5. For example, $[1/3, 5/6]$ and $[0.0, 0.25] \cup [0.5, 0.75]$ are both in T'. Let $T_1, T_2, \ldots$ be an enumeration of the elements of T'. Also let $\{a_1, a_2, \ldots\}$ be an enumeration of (some countably infinite subset of) A. The deterministic simulative model on these actions is given by:

$$g(s_0, a_i, p) = \begin{cases} s_{-1} & \text{if } p \in T_i \\ s_1 & \text{otherwise} \end{cases}$$

So, $P_{s_0 a_i}(s_1) = P_{s_0 a_i}(s_{-1}) = 0.5$ for all a_i, and this is a correct model for the MDP. Note also that $U(\pi) = 0$ for all $\pi \in \Pi$.

For any finite sample of m scenarios $(s_0, \vec{p}^{(1)}), (s_0, \vec{p}^{(2)}), \ldots, (s_0, \vec{p}^{(m)})$, there exists some T_i such that $p_1^{(j)} \notin T_i$ for all $j = 1, \ldots, m$. Thus, evaluating $\pi_i \equiv a_i$ using this set of scenarios, all m simulated trajectories will transition from s_0 from s_1, so the value estimate (assuming $H_\epsilon \geq 1$) for π_i is $\hat{U}(\pi_i) = 1$. Since this argument holds for any finite number m of scenarios, we have shown that $\hat{U}$ does not uniformly converge to $U(\pi) = 0$ (over $\pi \in \Pi$). □

Appendix 4.C: Proof of Theorem 12

The proof techniques we use here are due to Haussler [44] and Pollard [82]. Haussler [44], to which we will be repeatedly referring, provides a readable introduction to most of the methods used here.

We begin with some standard definitions from [44]. For a subset T of a space X endowed with (pseudo-)metric ρ, we say $T_0 \subset X$ is an **ϵ-cover** for T if, for every $t \in T$,

there is some $t' \in T_0$ such that $\rho(t,t') \leq \epsilon$. For each $\epsilon > 0$, let $\mathcal{N}(\epsilon, T, \rho)$ denote the size of the smallest ϵ-cover for T.

Let $\mathcal{H}$ be a family of functions mapping from a set X into a bounded pseudo metric space (A, ρ), and let P be a probability measure on X. Define a pseudo metric on $\mathcal{H}$ by $d_{L^1(P,\rho)}(f,g) = \mathbf{E}_{x\sim P}[\rho(f(x), g(x))]$. Define the **capacity** of $\mathcal{H}$ to be $\mathcal{C}(\epsilon, \mathcal{H}, \rho) = \sup \mathcal{N}(\epsilon, \mathcal{H}, d_{L^1(P,\rho)})$, where the sup is over all probability measures P on X. The quantity $\mathcal{C}(\epsilon, \mathcal{H}, \rho)$ thus measures the "richness" of the class $\mathcal{H}$. Note that $\mathcal{C}$ and $\mathcal{N}$ are both decreasing functions of ϵ, and that $\mathcal{C}(\epsilon, \mathcal{H}, \rho) = \mathcal{C}(k\epsilon, \mathcal{H}, k\rho)$ for any $k > 0$.

The main results obtained with pseudo-dimension are uniform convergence of the empirical means of classes of random variables to their true means. Let $\mathcal{H}$ be a family of functions mapping from X into $[0, M]$, and let $\vec{x}$ (the "training set") be m i.i.d. draws from some probability measure P over X. Then for each $h \in \mathcal{H}$, let $\hat{r}_h(\vec{x}) = (1/m)\sum_{i=1}^{m} h(x_i)$ be the empirical mean of $h(x)$. Also let $r_h(P) = \mathbf{E}_{x\sim P}[h(x)]$ be the true mean.

We now state a few results from [44]. In [44], these are Theorem 6 combined with Theorem 12; Lemma 7; Lemma 8; and Theorem 9 (with Y being a singleton set, $\ell(y,a) = a$, $\alpha = \epsilon/4M$, and $\nu = 2M$). Below, ℓ_1 and ℓ_2 respectively denote the Manhattan and Euclidean metrics on $\mathbb{R}^n$. e.g. $\ell_1(\vec{x}, \vec{y}) = \sum_{i=1}^{n} |x_i - y_i|$.[18]

Lemma 15 *Let $\mathcal{H}$ be a family of functions mapping from X into $[0, M]$, and $d = \dim_P(\mathcal{H})$. Then for any probability measure P on X and any $0 < \epsilon \leq M$, we have that $\mathcal{N}(\epsilon, \mathcal{H}, d_{L^1(P,\ell_2)}) \leq 2((2eM/\epsilon)\ln(2eM/\epsilon))^d$.*

Lemma 16 *Let $\mathcal{H}_1, \ldots, \mathcal{H}_k$ each be a family of functions mapping from X into $[0, 1]$. The*

[18] This is inconsistent with the definition used in [44], which has an additional $(1/n)$ factor.

free product *of the* $\mathcal{H}_i$*'s is the class of functions* $\mathcal{H} = \{(f_1, \ldots, f_k) : f_j \in \mathcal{H}_j\}$ *mapping from* X *into* $[0,1]^k$ *(where* $(f_1, \ldots, f_k)(x) = (f_1(x), \ldots, f_k(x))$*). Then for any probability measure* P *on* X *and* $\epsilon > 0$,

$$\mathcal{N}(\epsilon, \mathcal{H}, d_{L^1(P,\ell_1)}) \leq \prod_{j=1}^{k} \mathcal{N}(\epsilon/k, \mathcal{H}_j, d_{L^1(P,\ell_2)}) \tag{4.34}$$

Lemma 17 *Let* $(X_1, \rho_1), \ldots, (X_{k+1}, \rho_{k+1})$ *be bounded metric spaces, and for each* $j = 1, \ldots, k$*, let* $\mathcal{H}_j$ *be a class of functions mapping from* X_j *into* X_{j+1}*. Suppose that each* $\mathcal{H}_j$ *is uniformly Lipschitz continuous (with respect to the metric* ρ_j *on its domain, and* ρ_{j+1} *on its range), with some Lipschitz bound* $b_j \geq 1$*. Let* $\mathcal{H} = \{f_k \circ \cdots \circ f_1 : f_j \in \mathcal{H}_j, 1 \leq j \leq k\}$ *be the class of functions mapping from* X_1 *into* X_{k+1} *given by composition of the functions in the* $\mathcal{H}_j$*'s. Let* $\epsilon_0 > 0$ *be given, and let* $\epsilon = k(\prod_{j=1}^{k} b_j)\epsilon_0$*. Then*

$$\mathcal{C}(\epsilon, \mathcal{H}, \rho_{k+1}) \leq \prod_{j=1}^{k} \mathcal{C}(\epsilon_0, \mathcal{H}_j, \rho_{j+1}) \tag{4.35}$$

Lemma 18 *Let* $\mathcal{H}$ *be a family of functions mapping from* X *into* $[0, M]$*, and let* P *be a probability measure on* X*. Let* $\vec{x}$ *be generated by* m *independent draws from* X*, and assume* $\epsilon > 0$*. Then*

$$\mathbf{Pr}[\exists h \in \mathcal{H} : |\hat{r}_h(\vec{x}) - r_h(P)| > \epsilon] \leq 4\mathcal{C}(\epsilon/16, \mathcal{H}, \ell_2)e^{-\epsilon^2 m/64M^2} \tag{4.36}$$

We are now ready to prove Theorem 12. No serious attempt has been made to tighten polynomial factors in the bound.

Proof (of Theorem 12). Our proof is in three parts. First, $\hat{U}$ gives an estimate of the discounted rewards summed over $(H_\epsilon + 1)$-steps; we reduce the problem of showing uniform convergence of $\hat{U}$ to one of proving that our estimates of the expected rewards on the H-th

step, $H = 0, \ldots, H_\epsilon$, all converge uniformly. Second, we carefully define the mapping from the scenarios $s^{(i)}$ to the H-th step rewards, and use Lemmas 15, 16 and 17 to bound its capacity. Lastly, applying Lemma 18 gives our result. To simplify the notation in this proof, assume $R_{\max} = 1$, and $B, B_R \geq 1$.

Part I: Reduction to uniform convergence of H-th step rewards. $\hat{U}$ was defined by

$$\hat{U}(\pi) = \frac{1}{m}\sum_{i=1}^{m} R(s_0^{(i)}) + \gamma R(s_1^{(i)}) + \cdots + \gamma^{H_\epsilon} R(s_{H_\epsilon}^{(i)}).$$

For each H, let $\hat{U}_H(\pi) = \frac{1}{m}\sum_{i=1}^{m} R(s_H^{(i)})$ be the empirical mean of the reward on the H-th step, and let $U_H(\pi) = \mathbf{E}_{s_H}[R(s_H)]$ be the true expected reward on the H-th step (starting from $s_0 \sim D$ and executing π). Thus, $U(\pi) = \sum_{H=0}^{\infty} \gamma^H U_H(\pi)$.

Suppose we can show, for each $H = 0, \ldots, H_\epsilon$, that with probability $1 - \delta/(H_\epsilon + 1)$,

$$|\hat{U}_H(\pi) - U_H(\pi)| \leq \epsilon/2(H_\epsilon + 1) \;\; \forall \pi \in \Pi \tag{4.37}$$

Then by the union bound, we know that with probability $1-\delta$, $|\hat{U}_H(\pi) - U_H(\pi)| \leq \epsilon/2(H_\epsilon + 1)$ holds simultaneously for all $H = 0, \ldots, H_\epsilon$ and for all $\pi \in \Pi$. This implies that, for all $\pi \in \Pi$,

$$\begin{aligned} |\hat{U}(\pi) - U(\pi)| &\leq |\hat{U}(\pi) - \sum_{H=0}^{H_\epsilon} \gamma^H U_H(\pi)| + |\sum_{H=0}^{H_\epsilon} \gamma^H U_H(\pi) - U(\pi)| && (4.38) \\ &\leq \sum_{H=0}^{H_\epsilon} |\hat{U}_H(\pi) - U_H(\pi)| + \epsilon/2 && (4.39) \\ &\leq \epsilon. && (4.40) \end{aligned}$$

where we used the fact that $|\sum_{H=0}^{H_\epsilon} \gamma^H U_H(\pi) - U(\pi)| \leq \epsilon/2$, by construction of the ϵ-horizon time. But this is exactly the desired result. Thus, we need only prove that Equation (4.37) holds with high probability for each $H = 0, \ldots, H_\epsilon$.

Part II: Bounding the capacity. Let $H \leq H_\epsilon$ be fixed. We now write out the mapping from a scenario $s^{(i)} \in S \times ([0,1]^{d_P})^\infty$ to the H-th step reward. Since this mapping depends only on the first $d_P \cdot H$ elements of the "p"s portion of the scenario, we will, with some abuse of notation, write the scenario as $s^{(i)} \in S \times [0,1]^{d_P H}$, and ignore its other coordinates. Thus, a scenario $s^{(i)}$ may now be written as $(s, p_1, p_2, \ldots, p_{d_P H})$.

Given a family of functions (such as $\mathcal{F}_i$) mapping from $S \times [0,1]^{d_P}$ into $[0,1]$, we extend its domain to $S \times [0,1]^{d_P+n}$ for any finite $n \geq 0$ simply by having it ignore the extra coordinates. Note this extension of the domain does not change the pseudo-dimension of a family of functions. Also, for each $n = 1, \ldots, n$, define a mapping I_n from $S \times [0,1]^n \mapsto [0,1]$ according to $I_n(s, p_1, p_2, \ldots, p_n) = p_n$. For each n, let $\mathcal{I}_n = \{I_n\}$ be singleton sets. Where necessary, I_n's domain is also extended as we have just described.

For each $i = 1, \ldots, H+1$, define $X_i = S \times ([0,1]^{d_P})^{H+1-i}$. For example, X_1 is just the space of scenarios (with only the first $d_P H$ elements of the p's kept), and $X_{H+1} = S$. For each $i = 1, \ldots, H$, define a family of maps from X_i into X_{i+1} according to $\mathcal{H}_i = \mathcal{F}_1 \times \mathcal{F}_2 \times \cdots \times \mathcal{F}_{d_S} \times \mathcal{I}_{d_P+1} \times \mathcal{I}_{d_P+2} \times \cdots \times \mathcal{I}_{(H-i+1)d_P}$ (where the definition of the free product of sets of functions is as given in Lemma 16); note such an $\mathcal{H}_i$ has Lipschitz bound at most $B_0 = (d_S + H d_P)B$. Also let $\mathcal{H}_{H+1} = \{R\}$ be a singleton set containing the reward function, and $X_{H+2} = [-R_{\max}, R_{\max}]$. Finally, let $\mathcal{H} = \mathcal{H}_{H+1} \circ \mathcal{H}_H \circ \cdots \circ \mathcal{H}_1$ be the family of maps from $S \times ([0,1]^{d_P})^H$ into $[-R_{\max}, R_{\max}]$.

Now, let $\hat{U}^\pi_{M',H} : S' \mapsto [-R_{\max}, R_{\max}]$ be the reward received on the H-th step when executing π from a scenario $s \in S'$. As we let π vary over Π, this defines a family of maps from scenarios into $[-R_{\max}, R_{\max}]$. Clearly, this family of maps is a subset of $\mathcal{H}$.

Thus, if we can bound the capacity of $\mathcal{H}$ (and hence prove uniform converge over $\mathcal{H}$), we have also proved uniform convergence for $\hat{U}^{\pi}_{M',H}$ (over all $\pi \in \Pi$).

For each $i = 1, \ldots, d_S$, since $\dim_P(F_i) \leq d$, Lemma 15 implies that $\mathcal{N}(\epsilon, \mathcal{F}_i, d_{L^1(P,\ell_2)}) \leq 2((2e/\epsilon)\ln(2e/\epsilon))^d$. Moreover, clearly $\mathcal{N}(\epsilon, \mathcal{I}_i, d_{L^1(P,\ell_2)}) = 1$ since each $\mathcal{I}_i$ is a singleton set. Combined with Lemma 16, this implies that, for each $i = 1, \ldots, H$ and $\epsilon \leq 1$,

$$\begin{aligned}
\mathcal{N}(\epsilon, \mathcal{H}_i, d_{L^1(P,\ell_1)}) &\leq \prod_{j=1}^{d_S} \mathcal{N}(\epsilon/(d_S + (H-i)d_P), \mathcal{F}_j, d_{L^1(P,\ell_2)}) && (4.41)\\
&\leq \prod_{j=1}^{d_S} \mathcal{N}(\epsilon/(d_S + H_\epsilon d_P), \mathcal{F}_j, d_{L^1(P,\ell_2)}) && (4.42)\\
&\leq 2^{d_S}\left(\frac{2e(d_S + H_\epsilon d_P)}{\epsilon}\ln\frac{2e(d_S + H_\epsilon d_P)}{\epsilon}\right)^{dd_S} && (4.43)\\
&\leq 2^{d_S}\left(\frac{2e(d_S + H_\epsilon d_P)}{\epsilon}\right)^{2dd_S} && (4.44)
\end{aligned}$$

where we have used the fact that $\mathcal{N}$ is decreasing in its ϵ parameter. By taking a sup over probability measures P, this is also a bound on $\mathcal{C}(\epsilon, \mathcal{H}_i, \ell_1)$. Now, as metrics over $\mathbb{R}^{(d_S+(H-i)d_P)}$, $\ell_2 \leq \ell_1$. Thus, this also gives

$$\mathcal{C}(\epsilon, \mathcal{H}_i, \ell_2) \leq 2^{d_S}\left(\frac{2e(d_S + H_\epsilon d_P)}{\epsilon}\right)^{2dd_S} \quad (4.45)$$

Finally, applying Lemma 17 with each of the ρ_k's being the ℓ_2 norm on the appropriate space, $k = H+1$, and $\epsilon = (H+1)B_0^H B_R \epsilon_0$, we find

$$\begin{aligned}
\mathcal{C}(\epsilon, \mathcal{H}, \ell_2) &\leq \prod_{j=1}^{H+1} \mathcal{C}(\epsilon/((H+1)B_0^H B_R), \mathcal{H}_j, \ell_2) && (4.46)\\
&\leq \prod_{j=1}^{H} 2^{d_S}\left(\frac{2e(d_S + H_\epsilon d_P)(H+1)B_0^H B_R}{\epsilon}\right)^{2dd_S} && (4.47)\\
&\leq 2^{d_S H_\epsilon}\left(\frac{2e(d_S + H_\epsilon d_P)(H_\epsilon+1)B_0^{H_\epsilon} B_R}{\epsilon}\right)^{2dd_S H_\epsilon} && (4.48)
\end{aligned}$$

Part III: Proving uniform convergence. Applying Lemma 18 with the above bound on $\mathcal{C}(\epsilon, \mathcal{H}, \ell_2)$, we find that for there to be a $1-\delta$ probability of our estimate of the expected H-th step reward to be ϵ-close to the mean, it suffices that

$$m = \frac{256}{\epsilon^2}\left(\log\frac{1}{\delta} + \log\left(4\mathcal{C}(\epsilon/16, \mathcal{H}, \ell_2)\right)\right) \tag{4.49}$$

$$= \mathrm{O}\left(\mathrm{poly}\left(d, \frac{1}{\epsilon}, \log\frac{1}{\delta}, \frac{1}{1-\gamma}, \log B, \log B_R, d_S, d_P\right)\right). \tag{4.50}$$

This completes the proof of the Theorem. □

Chapter 5

Autonomous helicopter flight via reinforcement learning[1]

Helicopters represent a challenging control problem with complex, highly coupled, asymmetric, noisy, non-linear, high-dimensional, MIMO, non-minimum phase dynamics. They are widely regarded to be significantly more difficult to control than fixed-wing aircraft. [59] Consider, for instance, the problem of designing a helicopter that hovers in place. We begin with a single, horizontally-oriented main rotor attached to the helicopter via the rotor shaft. Suppose our helicopter's main rotor rotates clockwise (viewed from above),[2] blowing air downwards and hence generating upward thrust. This keeps the helicopter in the air against the force of gravity, but by applying clockwise torque to the main rotor to make it rotate, our helicopter experiences an anti-torque that tends to cause the main chassis to spin anti-clockwise. Thus, in the invention of the helicopter, it was necessary to

[1]The work presented in this chapter was jointly done by the author and Hyounjin Kim.

[2]Throughout this chapter, we will adopt the convention of clockwise-rotating rotors.

add a tail rotor, which blows air sideways/rightwards to generate an appropriate moment to counteract the spin. But this creates another problem: This sideways force now causes the helicopter to tend to drift leftwards. So, for a helicopter to hover in place, it must actually be tilted slightly to the right, so that the main rotor's thrust is directed downwards and slightly to the left, to counteract this tendency to drift sideways.

The history of the development of the helicopter is rife with such tales of complex, nonintuitive dynamics and of ingenious solutions to problems caused by solutions to other problems. When the helicopter is in forward flight, the main rotor blades, because they are rotating clockwise, move through the air faster on the left than on the right, generating asymmetric lift. While ascending straight up is a fairly straightforward maneuver, moderate-speed descent involves the helicopter entering into noisy, turbulent air; surprisingly, these effects again disappear for a fast descent. Because of its asymmetric design, turning left and turning right are also very different maneuvers.

In this chapter, we describe the successful application of reinforcement learning to designing a controller for autonomous helicopter flight.

5.1 Introduction

In this section, we will begin by describing the helicopter hardware platform for which we were interested in designing a controller. We also describe the helicopter's instrumentation, its state space S, and its controls A. In Section 5.2, we then present how we identified (fit) a model of the helicopter's dynamics. Section 5.3 gives the policy class Π we chose for a controller to make the helicopter hover in place, and describes our application

Figure 5.1: Berkeley autonomous helicopter.

of reinforcement learning to this task. Section 5.4 then extends this work to making the helicopter fly challenging maneuvers taken from an RC helicopter competition.

The helicopter used in all of our experiments was the Berkeley autonomous helicopter (Figure 5.1) [96]. This is a Yamaha R-50 helicopter, approximately 3.6m long (including rotors), and carries a payload of up to about 20kg. The helicopter is instrumented with sensors and with a flight computer that is used to perform onboard control and navigation.

If we imagine our helicopter to be a rigid object in 3-space, then its state is characterized by 12 numbers: Its position (x, y, z), orientation (row ϕ, pitch θ, yaw ω), velocity $(\dot{x}, \dot{y}, \dot{z})$ and angular velocity $(\dot{\phi}, \dot{\theta}, \dot{\omega})$. Thus, we may make take an initial definition of the state space of the helicopter to be $S = \mathbb{R}^{12}$.[3] (We will add more state variables later.)

A detailed description of the design and instrumentation of this helicopter is given in Shim [96]. Briefly, it carries an Inertial Navigation System (INS) consisting of 3 accelerometers and 3 rate gyroscopes installed in exactly orthogonal x, y, and z directions. This gives estimates of linear accelerations and of turning rates. The helicopter is also instrumented with differential GPS. This consists (roughly) of a GPS receiver operating on the helicopter, and one operating on the ground at a precisely known location. By examining the (known) error of the GPS position estimate of the ground receiver and using that to correct the estimate of the GPS unit on the helicopter, this gives position estimates with a resolution of 2cm. A Kalman filter integrates the sensor information from the GPS, INS, and a digital compass, and reports (at 50Hz) 12 numbers corresponding to the estimates of

[3] Actually, the position and orientation of the helicopter would be more accurately modeled as lying in $\mathbb{R}^3 \times SO(3)$ since angles lie in a circle, etc. (E.g., see [74].) But for our purposes it suffices and is simpler to model the state as described above.

each of the 12 state variables described above.

Most helicopters built in this single main-rotor, single tail-rotor configuration are controlled via a 4-dimensional action space:

- a_1, a_2: The longitudinal (front-back) and latitudinal (left-right) cyclic pitch controls. The *rotor plane* is the plane through which the helicopters' rotors rotate. These controls tilt this plane either forwards/backwards or sideways, and by doing so cause the helicopter to accelerate forward/backwards or sideways.

- a_3: The (main rotor) collective pitch control. As the helicopter main-rotor's blades sweep through the air, they generate an amount of upward thrust that (generally) increases with the angle at which the rotor blades are tilted. E.g., a "flat"/horizontally oriented rotor blade slicing through the air generates no lift, whereas one tilted upwards would push air downwards and hence generate upward thrust. By varying the tilt angle of the rotor blades, the cyclic pitch control affects the main rotor's thrust.[4]

- a_4: The tail rotor collective pitch control. Using a mechanism similar to the main rotor collective pitch control, this controls the tail rotor's thrust.

The range of the actions are bounded, and we may thus take our space of actions to be $A = [-1, 1]^4$.

Readers interested in learning more about helicopters and helicopter dynamics may refer to such texts as Seddon [93], Leishman [59], and Wagtendonk [105]. The first two of these served as the author's primary references, whereas the third, written mainly for pilots,

[4]The helicopter throttle, a 5th control channel, is also commanded as a pre-set, deterministic, monotonic function of the cyclic pitch; we need not be concerned with it in the remainder of our discussion.

is less mathematical and more easily accessible. (For a general primer on aerodynamics, refer to, e.g., [2, 3].) Readers interested in reading more about the development of autonomous aerial vehicles may also see [96].

The hovering problem is as follows: Every 50th of a second, we obtain a new estimate of the helicopter state. Our task is to return some 4-dimensional vector $a \in A$, so that by executing our controls, the helicopter stays stably in the air.

5.2 Model identification

To run our reinforcement learning algorithms, we required an accurate model of the helicopter's state transition dynamics $P_{sa}(\cdot)$. To fit such a model, we chose a data-driven approach. We began by asking a human pilot to fly the helicopter for several minutes, and recorded the 12-dimensional helicopter state and 4-dimensional helicopter control inputs as it was flown. The pilot was asked to systematically perform frequency sweeps (this consists of oscillating a control at slowly increasing frequencies) [70] in each of the four control channels, while using the other three channels to stabilize the helicopter. In what follows, 339 seconds of flight data was used for fitting our model, with another 140 seconds of data used for hold-out testing.

For the remainder of this discussion, let us augment the helicopter state vector s with a constant 1 for notational convenience, so that $s = [x, y, z, \phi, \theta, \omega, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\omega}, 1]$. We modeled the continuous dynamics of the helicopter as a discrete time system. Following discussions with colleagues knowledgeable about helicopters, we decided to downsample our 50Hz data to 10Hz, and identify our model at 10Hz, so that the elapsed time between

successive states (s_t and s_{t+1}) was 1/10-th of a second.

5.2.1 Locally weighted regression

Since helicopters exhibit highly non-linear dynamics, we felt it was important to use a model that can capture these nonlinearities. We chose locally weighted linear regression (e.g., [6, 25]) as our main tool for fitting our model.

Locally weighted regression works as follows. Given a dataset $\{(x_i, y_i)\}_{i=1}^m$ where the x_i's are the vector-valued inputs and the y_i's are the real-valued outputs we are trying to predict, let X denote the design matrix whose i-th row is x_i, and let y be the vector of y_i's. In response to a query at a new x_{m+1}, standard (global) regression would perform a least-squares fit to the data:

$$\beta = (X^T X)^{-1} X^T y, \tag{5.1}$$

and evaluate the resulting linear function ($y = \beta^T x$) at $x = x_{m+1}$, yielding its prediction $\beta^T x_{m+1}$ for the value of y_{m+1}. Locally weighted regression does something slightly different. Specifically, it fits a linear function *locally*, using primarily the data close to the query x_{m+1}. Specifically, define $w_i = \exp(-\frac{1}{2}(x_{m+1} - x_i)^T \Sigma^{-1} (x_{m+1} - x_i))$ to be the weight of x_i ($i = 1, \ldots, m$). Thus, points close to the query x_{m+1} are given a large weight, and points far away are given a small weight. Here, Σ^{-1}, a positive semi-definite matrix, controls how fast the weights fall off with distance as we move away from x_{m+1}. Locally weighted regression then weights the data while performing its regression, and calculates:

$$\beta = (X^T W X)^{-1} X^T W y \tag{5.2}$$

Above, W is a diagonal matrix containing the w_i's. Finally, its prediction for y_{m+1} is given

by $\beta^T x_{m+1}$.

If we wish to use locally weighted regression to make a prediction at a new input different from x_{m+1}, then we would first again reweight all the data, fit a linear function to the data using those weights, and compute the prediction by evaluating the linear function at the location of the new query. Note that, since this procedure chooses a different linear function at every point, if we plot its predictions y as a function of the input query x, we obtain a function that is *not* locally linear or locally affine, nor piecewise linear nor piecewise affine. Instead, locally weighted regression is able to identify fully *nonlinear* models.

In our description above, Σ^{-1}, which determines the scaling of the input space controlling the weighting, still has to be specified. In our experiments, this was picked via leave-one-out cross validation.[5]

By performing regression with the current state and control s_t and a_t playing the role of the input x and, in turn, each variable of the next state s_{t+1} (or, more precisely, the change in the state $s_{t+1} - s_t$) playing the role of the output y, we obtain a nonlinear model of our helicopter dynamics. Specifically, given s_t, a_t, we now have a model that predicts what s_{t+1} will be. However, to build a realistic model of the helicopter, we also need to know the uncertainty in this estimate. In particular, we use the model $y = \beta^T x + \epsilon$, where ϵ is distributed as a Gaussian with zero mean and variance σ^2. We used the estimator for the noise variance σ^2 given in [25].[6] This ϵ term thus captures disturbances to the helicopter

[5]Actually, since we were fitting a model to a time-series, samples tend to be correlated in time, and the presence of temporally close-by samples—which will be spatially close-by as well—may make data seem more abundant than in reality (leading to bigger Σ^{-1} than might be optimal for test data). Thus, when leaving out a sample in cross validation, we actually left out a large window (16 seconds) of data around that sample, to diminish this bias.

[6]This is another estimator with a leave-one-out cross-validation flavor, and the windowing trick described in the previous footnote was also applied here.

state due to external, random perturbations (such as wind, etc).

Lastly, we also modified our model to capture uncertainty in the model itself. Specifically, if in certain areas of the state space we have very little data, then our local fit for β will be based on very little data, and we should not be confident of our estimate of the local β there. One standard way of capturing this uncertainty involves interpreting the locally weighted regression that we have described as a Bayesian procedure, so that via Bayes rule we can obtain a posterior distribution on β as $p(\beta|\text{data}) \propto p(\text{data}|\beta)p(\beta)$, where we choose $p(\beta)$ is a noninformative (flat) prior. (See, e.g., Gelman et al. [37].) Briefly, this method "pretends" that the outputs y_i for datapoints far away from the query x_{m+1} were observed with very high variance, so that the optimal, Bayesian regression on this data will automatically give them small weight. Armed with these posteriors, we thus have

$$\begin{aligned} \text{Var}(y_{m+1}|x_{m+1}) &= \text{Var}(\beta^T x_{m+1} + \epsilon) && (5.3) \\ &= x_{m+1}^T \Sigma_\beta x_{m+1} + \sigma^2, && (5.4) \end{aligned}$$

where Σ_β is the posterior covariance of β^T.[7]

Thus, to build our generative model or simulator for the helicopter—that is, a function that, given s and a, samples s' from $P_{sa}(\cdot)$—we write a function that, given s, a, performs locally weighted regression, calculates $\text{Var}(y_{m+1}|x_{m+1})$ using the formula above (where again y plays the role of, in turn, each coordinate of $s' - s$ and x_{m+1} plays the role of s, a), and finally returns $y_{m+1} = \beta^T x + \epsilon$, where ϵ is sampled from a zero mean gaussian with

[7] The Bayesian setup given in [37] also gives a way of estimating the variance σ^2 as well, if that is unknown. We note that it is important *not* to use that procedure if σ^2 is to be interpreted as the variance of ϵ as in our above description, since that estimator is hugely biased and will tend to significantly underestimate the variance. Given our application, underestimating the noise of the process may mean policies that fly well according to our model, but crash in real life, and is thus dangerous and unacceptable. Thus, we actually used the estimator for σ^2 described earlier.

the variance computed by Equation (5.4). To build a deterministic simulative model, we can use $g(s,a,p) = [g_1(s,a,p_1), \ldots, g_{d_S}(s,a,p_{d_S})]^T$, where each of the coordinate functions g_i is given by

$$g_i(s,a,p) = s_i + (s'_i - s_i) \tag{5.5}$$

$$= s_i + \beta^T x + F^{-1}_{s,a}(p). \tag{5.6}$$

Here, $F^{-1}_{s,a}$ is the inverse cdf of a gaussian with zero mean and variance as given by Equation (5.4).[8]

This completes our description of our main tool of locally weighted regression used to build our generative model. While locally weighted regression could, in principle, be applied straightforwardly to our raw training data to obtain a model of the helicopter dynamics, in the next section we will also see how this can significantly be improved on.

Readers interested in learning more about locally weighted regression may also see, e.g., the survey by Atkeson et al. [6] and the references therein.

5.2.2 Model selection and incorporating prior knowledge

In this section, we describe some of the choices we made in our attempts to design a high-fidelity model for the helicopter, and some of the rationales for them. In incorporating certain forms of prior knowledge into the model, our work also builds on that of Shim [96] (who, in turn, incorporated a number of ideas from Mettler et al. [71]), who built a deterministic, globally linear model for a helicopter.

[8] Actually, it would be easier to change the definitions of the p's from that given in Chapter 4 to be $p \sim \text{Normal}(0,1)$, and to instead use $g_i(s,a,p) = \beta^T x + \tau p$, where τ^2 is the variance computed using Equation (5.4).

There are many natural symmetries in helicopter flight. For instance, suppose we command a helicopter at position (10,10,50) facing east to move forward. This is related only by a translation and rotation to one at position (20,20,30) facing north that is issued the same command. Thus, it seems inefficient and unnecessary to try to learn different models for these two parts of the state space, which is what locally weighted regression would naively do if we were to give it the raw training data we had collected. Specifically, it seems more reasonable encode this type of symmetry into the model, rather than force our algorithms to learn them from scratch. In this example, this means we would like to learn what a helicopter will do if it is commanded to go forward, and then "apply" this single model to both the helicopter at (10,10,50) and the one at (20,20,30).

Thus, in system identification, it is standard to fit a model not in spatial (world) coordinates, but in the helicopter body coordinates. In this coordinate frame, the x, y, and z axes are forwards, sideways, and down relative to the current position of the helicopter. Where there is risk of confusion, we will use superscript s and b subscripts to distinguish between spatial and body coordinates; thus, $\dot{x}^b$ is forward velocity, regardless of the orientation of the helicopter. Our model is identified in the body coordinates $s^b = [\phi, \theta, \dot{x}^b, \dot{y}^b, \dot{z}^b, \dot{\phi}, \dot{\theta}, \dot{\omega}]$ which has four fewer variables than s^s. The equations transforming between spatial and body coordinates are easily derived (e.g., see [96]). As before, we can thus apply locally weighted regression to predict the one-step differences $s^b_{t+1} - s^b_t$ to build our model. It should also be clear that, given a model or simulator expressed in terms of body coordinates, we can easily obtain a simulator expressed in the original, spatial coordinates via an appropriate transformation of the variables (and it is the latter model that will be used by

our learning algorithms).

Apart from switching to body coordinates, we also considered several other ways of incorporating prior knowledge into the model.

First, we previously said that the cyclic pitch controls tilt the rotor plane and causes the helicopter to accelerate in a certain direction. But because of a mechanical damper ("a Bell-Hiller stabilizer"), the effects of this are not instantaneous, and there is significant latency between when the cyclic pitch controls are changed and when the rotor plane is tilted. Following [96], we also added two state variables a_{1s}, b_{1s} capturing orientation of the rotor-plane. Since rotor-plane tilt is not directly observed, however, we could not use locally weighted regression to directly fit a model to it, and thus instead used the model for a_{1s} and b_{1s} identified by [96]. Similarly, another state variable $\dot{\omega}_{fb}$ was also added to model the latency in $\dot{\omega}$ caused by a yaw-rate gyro feedback mechanism. On our held-out test data, these changes resulted in small but statistically significant increases in prediction accuracy, and were thus included in the final model.

In addition, similar to how using body coordinates exploits certain symmetries, there are other symmetries that can be incorporated into the model. For instance, since both ϕ_t and $\dot{\phi}_t$ are state variables, and we know that (at 10Hz) $\phi_{t+1} \approx \phi_t + \dot{\phi}_t/10$, there is no need to carry out a regression for ϕ. Similarly, we know that the row angle ϕ of the helicopter should have no direct effect on forward velocity $\dot{x}$. So, when performing regression to estimate $\dot{x}$, the coefficient in β corresponding to ϕ can be set to 0. This thus allows us to reduce the number of parameters that have to be fit by our regression. There are many other examples of these effects, and briefly, we ended up using the "template" given in Equation

(2.66-2.67) of [96] (plus adding a constant/bias term so that our regression doesn't have to intersect $y = 0$ at $x = 0$). Thus, coefficients in β were set to the constants to 0, ± 1, or $\pm g$ (gravity) if dictated so by the template, and left as a variable to be fit by the regression otherwise.

Briefly, some of the (other) choices that we considered in selecting a model include whether to use the a_{1s}, b_{1s} and/or $\dot{\omega}_{fb}$ terms; whether to include intercept terms in the regression; whether to identify the model at 5, 10, 25 or 50Hz; whether to set/hardwire certain coefficients as described earlier; and whether to use a globally linear regression or locally weighted regression.

Our main tool for choosing among the possible models was plots such as shown in Figure 5.2a, which we plotted for various state variables. (See figure caption.) We were particularly interested in checking how accurate a model is not just for predicting s_{t+1} from s_t, a_t, but how accurate the model is at longer time scales. (In the following paragraph and in footnote 9, we describe some of our motivations for examining different time scales.) Each of the panels in Figure 5.2a shows, for a model, the mean-squared error (as measured on test data) between the helicopter's true position and the estimated position at a certain time in the future (indicated on the x-axis), assuming the state estimate at time 0 is perfect.

Note also that, even though we previously discussed how certain coefficients of β can be set to zero so that certain coordinates of s_{t+1} depend only on a sub-vector of s_t, at larger time scales many more of the state variables become coupled. (E.g., for a single-step transition in the MDP, only $\dot{\phi}$ affects ϕ; but at longer time scales, other variables now have time to affect $\dot{\phi}$ and hence ϕ.) Thus, each of these plots to some degree tests the models

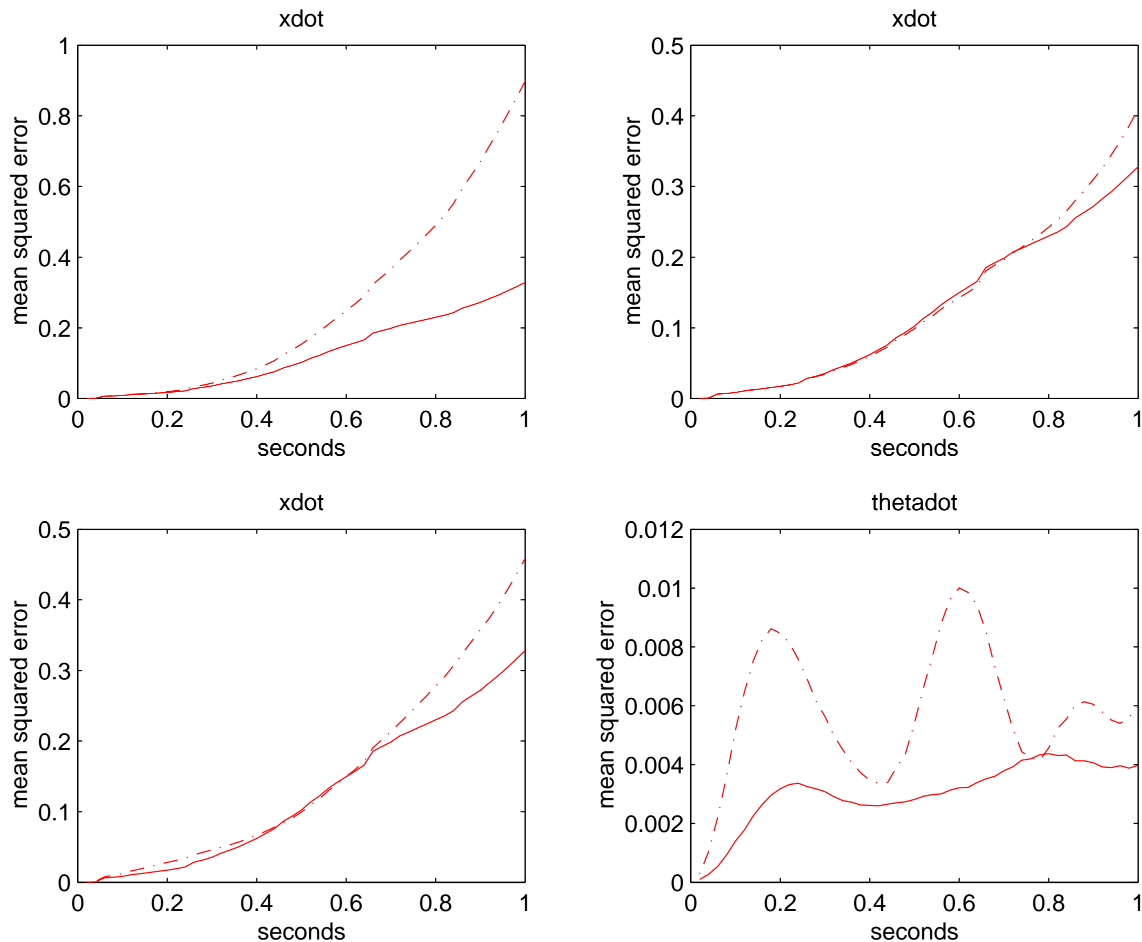


Figure 5.2: Examples of plots comparing a (globally) linear model fit using the parameterization described in the text (solid lines) to some other models (dash-dot lines). Each point plotted shows the mean-squared error between the predicted value of a state variable—when a model (ignoring model noise) is used to the simulate the helicopter's dynamics for a certain duration indicated on the x-axis—and the true value of that state variable (as measured on test data) after the same duration. Top left: Comparison of $\dot{x}$-error to model not using extra a_{1s}, etc. variables. Top right: Comparison of $\dot{x}$-error to a model that omits the intercept (bias) term. Bottom: Comparison of $\dot{x}$ and $\dot{\theta}$ to linear deterministic model identified by [96].

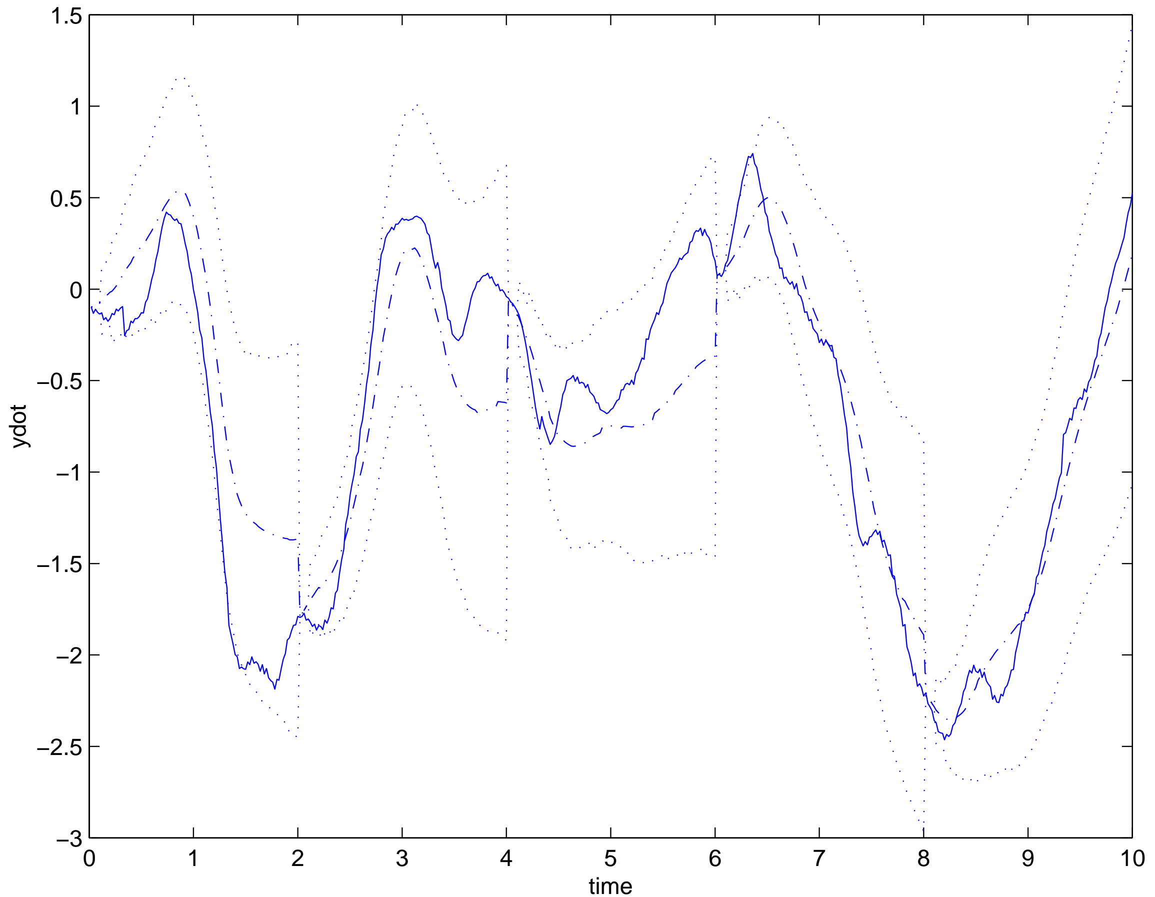


Figure 5.3: The solid line is the true helicopter $\dot{y}$ state on 10s of test data. The dash-dot line is the helicopter state predicted by our model, given the initial state at time 0 and all the intermediate control inputs. The dotted lines show 2 standard deviations in the estimated state. (Calculating these on a stochastic non-linear model is actually intractable, so these were estimated using an extended Kalman filter, and using a diagonal approximation to all the covariances as in [67, 21].) Every two seconds, the estimated state is "reset" to the true state, and the track starts again with zero error. Note that the estimated state is of the full, high-dimensional state of the helicopter, but only $\dot{y}$ is shown here.

for *all* the state variables, not just that of the variable whose error is being plotted. By generating these plots for different state variables, we learn how good the various models are at predicting the helicopter's dynamics at different time scales.

The blade-tip of the helicopter moves at over 1/3 of the speed of sound. Safety is a critical issue, and an accident could easily result in death or dismemberment for a person. Thus, having constructed a model, we felt it was important to make significant effort to verify that our stochastic model captures the dynamics well, so that we might be reasonably confident that a policy tested successfully in simulation will also be safe in real life. One of our main concerns was the possibility of unmodeled correlations in the ϵ noise terms (either across time or across the different state variables). Specifically, unmodeled correlations could mean that actual noise variance of the dynamics is much larger than that predicted by the model,[9] which would make it dangerous to fly a control policy that has been tested successfully only in simulation. To check against this, we examined many plots such as the one in Figure 5.3, to verify that the helicopter state "rarely" goes outside the error bars predicted by our model *at various time scales* (see caption).

Finally, we also used a simple trick to speed up our simulation. Recall that here we are using a discrete-time model to approximate a continuous time process, and the time-scale of the model was a parameter chosen by us. Specifically, if we have a model $s_{t+1} = s_t + f(s_t, a_t) + \epsilon$ identified at 10Hz, we may obtain a 50Hz model via the approximation

[9]Consider: Under the process $x_{t+1} = x_t + \epsilon_t$, if the ϵ_t's are strongly correlated in time, then in T time steps, x_t can easily drift $O(T)$ distance; but if the ϵ_t's are modeled as uncorrelated, the model would predict it drifting only $O(\sqrt{T})$ distance (the standard deviation of the sum of T independent random variables). Similarly, biases or errors in the model may also result in unmodeled, correlated errors: If in reality $x_{t+1} = 0.99x_t + \epsilon$, but our model is $x_{t+1} = 0.9x_t + \epsilon$ (perhaps because of oversmoothing), then successive errors ($\epsilon = x_{t+1} - 0.9x_t$) in time will either all tend to be all larger or all smaller than zero (depending on the sign of x_t).

$s_{t+1} = s_t + f(s_t, a_t)/5 + \epsilon'$, where the variance of ϵ' is 1/5 that of ϵ. Assuming that most of the dynamics of the helicopter are slower than 10Hz, both of these models would give about the same answers. Because the dynamics of some of the hidden state variables (a_{1s}, b_{1s}) were quite fast, and because we anticipated controlling the actual helicopter at 50Hz, we actually chose to use a 50Hz simulation. However, this also means needing 50 instead of 10 steps in the MDP to simulate one second of actual flight time, and hence a simulator that is five times slower.[10] However, the bottleneck turns out to be the fitting of the locally linear regressions to obtain β. As a compromise, we therefore decided to run the simulation at 50Hz, but to "refit" the linearity at 10Hz. Thus, every 5 steps in the simulation, we would examine the helicopter state and use its current location to refit a locally weighted regression to obtain β; then, for the next 5 steps, we will let the helicopter's dynamics evolve according to that same β. This approximation introduced negligible additional error into our model, and we felt represented a good compromise between having a fast simulator and being true to the 50Hz controller that we will be flying. (For another way of speeding up locally-weighted models, also see Moore et al. [73]. While an excellent choice for many applications, we chose not to use the ideas there because they can introduce unwanted discontinuities into the model of the dynamics.)

5.3 Learning to Hover

Armed with our simulator for the helicopter, we proceeded to use reinforcement learning to learn a controller to make the helicopter hover stably.

[10]By downsampling our training data from 50Hz to 10Hz, we also obtained almost a 5-fold speedup, since our training set became 1/5 as large and hence the locally weighted regression only needed to examine 1/5 as much data.

As discussed earlier, helicopter control is a challenging task. The only other attempt to use an automatic learning algorithm on the Berkeley autonomous helicopter was via μ-synthesis [11] (which, very informally, can be thought of as a "robust" version of H_∞-control). This succeeded in flying the helicopter in simulation, but not on the actual helicopter. [95] Similarly, preliminary experiments using H_2 and H_∞ controllers to fly a similar helicopter were also unsuccessful. [7] Of course, these comments should not be taken as criticism of any of these methods, all of which have succeeded on many difficult problems; rather, we take them to be indicative of the difficulty and subtlety involved in learning a helicopter controller.

We began by first learning a policy to keep the helicopter hovering in place. We proceeded by first specifying a policy class Π for hovering. In a way reminiscent of the discussion in Section 5.2.2 on model identification, there are many symmetries of which we can take advantage. For instance, if the helicopter is one meter to the left of the position (x^*, y^*, z^*) where we want it to hover, then the action we should take to move it a meter rightwards probably does not depend on whether the desired (x^*, y^*, z^*) is (10,10,50) or (20,20,30).

For our policy class, we chose the simple neural network depicted in Figure 5.4. We want a controller that, given the current helicopter state and a desired hovering position and orientation $(x^*, y^*, z^*, \omega^*)$, computes controls $a \in [-1, 1]^4$ to make it hover stably there. Each of the edges in the figure represents a weight, and the connections were chosen via simple reasoning about which control channel should be used to control which state variables. For instance, consider the control a_1 for the longitudinal (forward/backward)

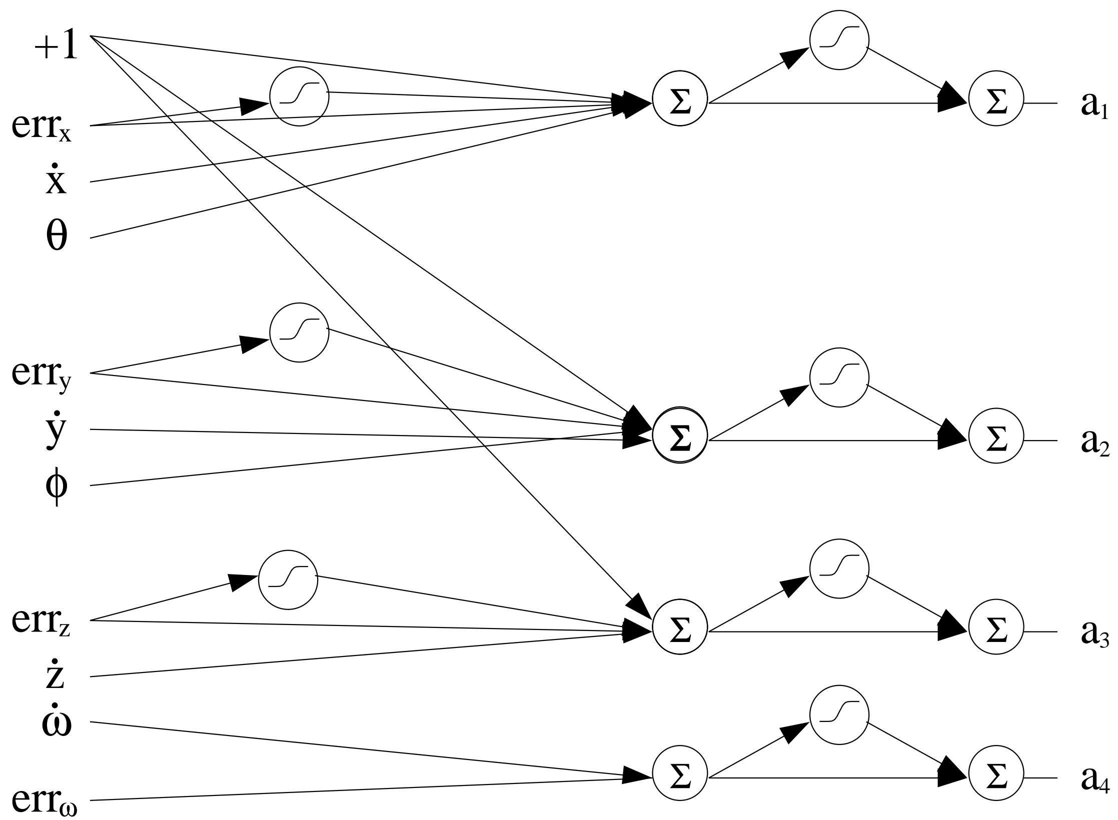


Figure 5.4: Policy class Π used to learn a controller for hovering. The pictures inside the circles indicate whether each node computes and outputs the sum of its inputs, or the tanh of its input. Each edge with an arrow in the picture denotes a tunable parameter.

cyclic pitch control. This causes the rotor plane to tilt forward/backward, thus causing the helicopter to pitch (and/or accelerate) forward or backward. From Figure 5.4, we can read off the cyclic pitch control to be given by

$$t_1 = w_1 + w_2 \,\mathrm{err}_x + w_3 \tanh(w_4 \mathrm{err}_x) + w_5 \dot{x}^b + w_6 \theta, \tag{5.7}$$

$$a_1 = w_7 \tanh(w_8 t_1) + w_9 t_1. \tag{5.8}$$

Here, the w_i's are the tunable parameters (weights) of the network, and $\mathrm{err}_x = x^b - x^b_{\mathrm{desired}}$ is defined to be the error in the x^b-position (forward direction, in body coordinates) between where the helicopter currently is and where we wish it to hover. Thus, whether the helicopter moves forwards or backwards is affected by the current error in the forward/backward direction, by the forward velocity, and by the forward pitch angle.[11]

For the reward function, we chose a (LQR/LQG [1] style) quadratic cost function on the (spatial representation of the) state, where[12]

$$\begin{aligned} R(s) = \quad & -(\alpha_x (x - x^*)^2 + \alpha_y (y - y^*)^2 + \alpha_z (y - y^*)^2 \\ & + \alpha_{\dot{x}} \dot{x}^2 + \alpha_{\dot{y}} \dot{y}^2 + \alpha_{\dot{z}} \dot{z}^2 + \alpha_\omega (\omega - \omega^*)^2). \end{aligned} \tag{5.9}$$

This encourages the helicopter to hover near $(x^*, y^*, z^*, \omega^*)$, while also keeping the velocity small and not making abrupt movements. The weights α_x, α_y, etc. (distinct from, and not to be confused with, the weights w_i parameterizing our policy class) were chosen using crude guesses on the "typical" magnitudes of each term, so as to scale them to be roughly the same order of magnitude. (This is a standard heuristic for choosing these weights. [1])

[11] We also considered letting the pitch-rate $\dot{\theta}$ be an input to a_1 (and the roll-rate $\dot{\phi}$ be an input to a_2), but decided against it for safety reasons because we were not confident that the Kalman filter could return consistently accurate estimates of these quantities.

[12] The $\omega - \omega^*$ error term is computed with appropriate wrapping about 2π rad, so that if we want to hover facing 0.01 rad, and the helicopter is currently facing $2\pi - 0.01$ rad, the error is 0.02, not $2\pi - 0.02$ rad.

To encourage small actions and smooth control of the helicopter, we also used a quadratic penalty for actions:

$$R(a) = -(\alpha_{a_1} a_1^2 + \alpha_{a_2} a_2^2 + \alpha_{a_3} a_3^2 + \alpha_{a_4} a_4^2), \tag{5.10}$$

and the overall reward was $R(s,a) = R(s) + R(a)$.

We now have a well-defined reinforcement learning problem. Specifically, the model we identified in Section 5.2 gives us the deterministic simulative model g, which with the reward function can then be used by the PEGASUS method to define approximations $\hat{U}(\pi)$ to the utilities of policies. Since policies are smoothly parameterized in the weights, and the dynamics are themselves continuous in the actions, the estimates of utilities are also continuous in the weights.[13] We may thus apply standard hillclimbing algorithms to search for a good setting of the weights. We tried both a gradient ascent algorithm, in which we numerically evaluate the derivative of $\hat{U}(\pi)$ with respect to the weights and then take a step in the indicated direction, and a random-walk algorithm in which we propose a random perturbation to the weights, and move there if it increases $\hat{U}(\pi)$. Both of these algorithms worked well, though with the gradient ascent method, it was important to scale the derivatives appropriately, since the estimates of the derivatives were sometimes numerically unstable.[14] It was also important to apply some standard heuristics to prevent its solutions from diverging (such as verifying after each step that we did indeed take a step

[13] Actually, this is not quite true. One last component of the reward that we did not mention earlier was that, if in performing the locally weighted regression, the matrix X^TWX is singular to numerical precision, then we declare the helicopter to have "crashed," terminate the simulation, and give it a huge negative (-50000) reward. The rationale is that, if X^TWX is singular, then Σ_β should be infinite in some direction, and hence the posterior variance of the state should also be (in general) infinite in some direction, thus incurring an "infinite" penalty via Equation (5.9). But because the test that checks if X^TWX is singular to numerical precision must necessarily return either 1 or 0, this means $\hat{U}(\pi)$ has a discontinuity between "crash" and "not-crash."

[14] A problem which seemed to be exacerbated significantly by the discontinuities described in the preceding footnote.

Figure 5.5: Helicopter hovering under control of learned policy.

uphill on the objective $\hat{U}$, and undoing/redoing the step using a smaller stepsize if this was not the case).

Lastly, we also found that, while performing policy search, the key computational step was the repeated Monte Carlo evaluation to obtain $\hat{U}(\pi)$. While we could have run our algorithms on a single computer, such operations are easily parallelized. Specifically, we can run Monte Carlo evaluations using different scenarios on different computers, and then aggregate the results centrally to obtain $\hat{U}(\pi)$. Using the Millennium cluster at Berkeley [84], we were thus able to use a parallel implementation to speed up policy search.

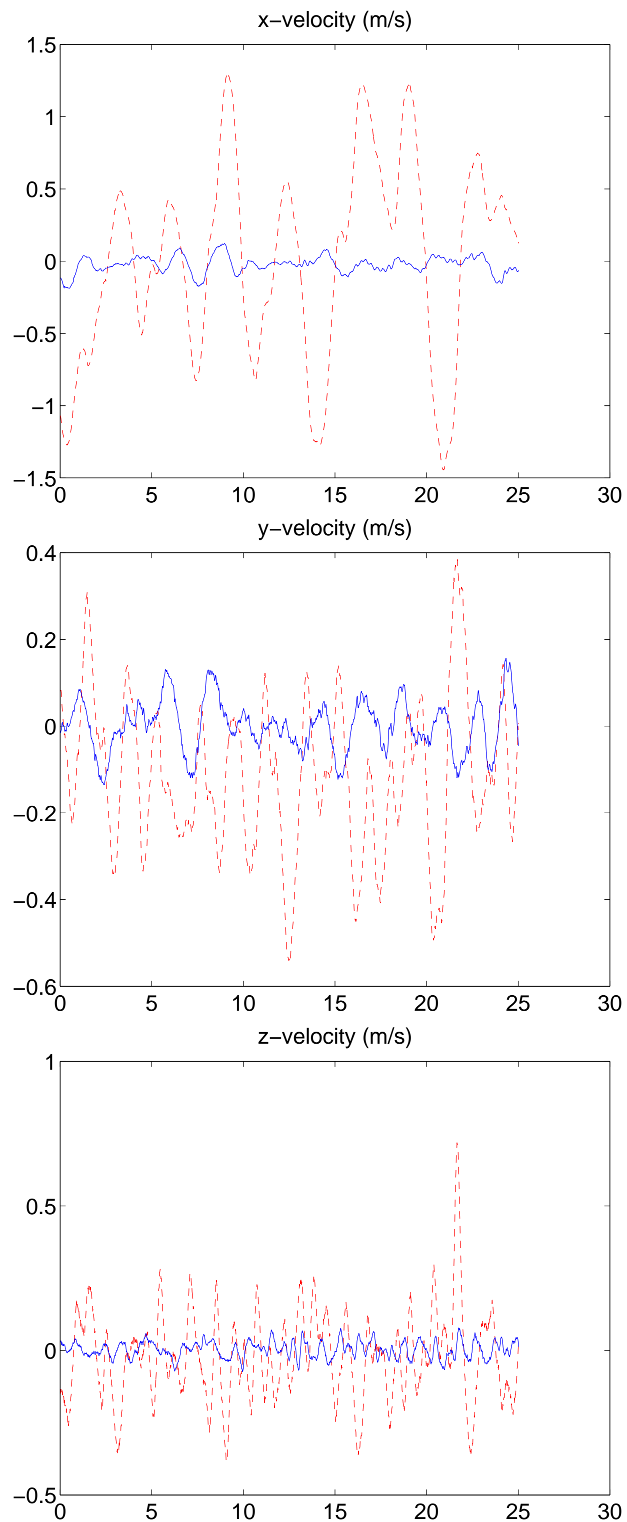


Figure 5.6: Comparison of hovering performance of learned controller (blue solid line; colors where available) vs. trained human pilot (red dashed line). Shown here are the x^b, y^b and z^b (body coordinate) velocities.

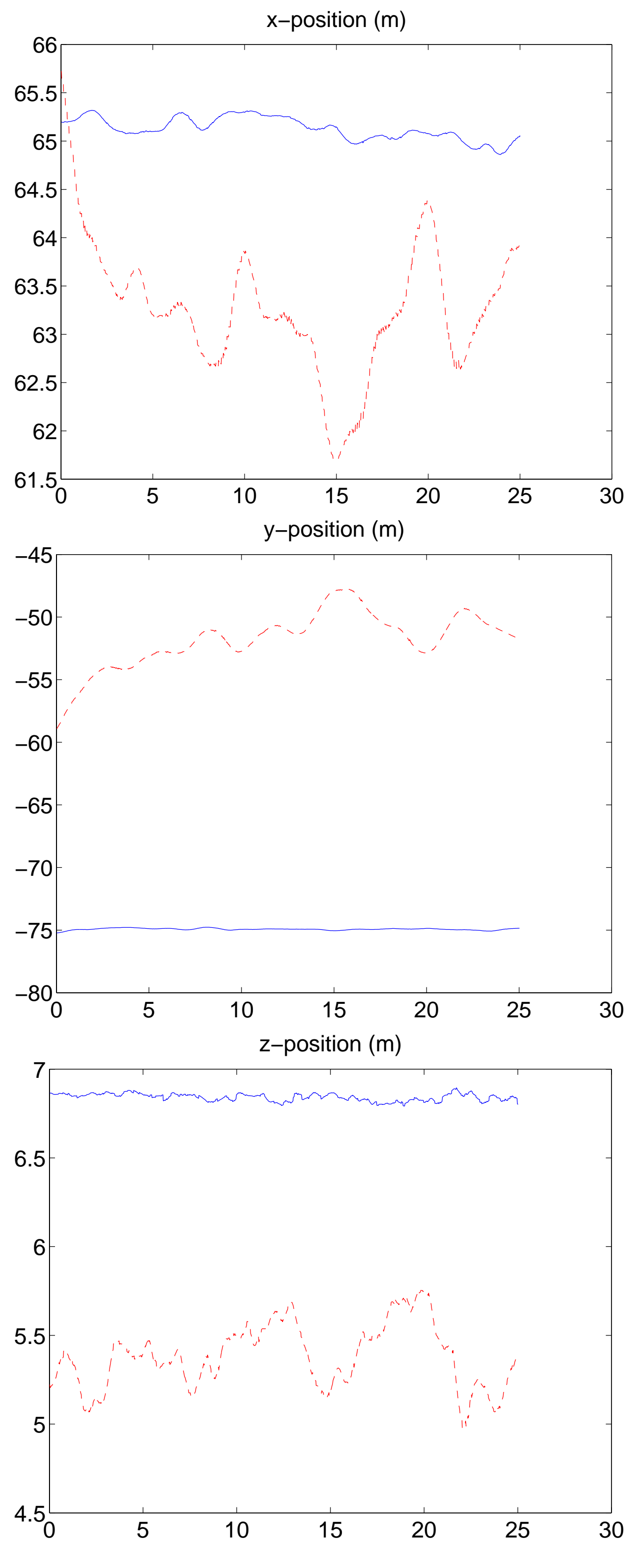


Figure 5.7: Comparison of hovering performance of learned controller (blue solid line; colors where available) vs. trained human pilot (red dashed line). Shown here are the x, y and z (world coordinate) positions.

We ran Pegasus using 30 scenarios of 35 seconds of flying time each, and a discount of $\gamma = 0.9995$. Figure 5.5 shows the result of implementing and running the resulting policy on the helicopter. On its maiden flight, our learned policy was successful in keeping the helicopter stabilized in the air. We note that [8] was also successful at modeling and using our Pegasus algorithm to control a subset (the cyclic pitch controls) of a helicopter's dynamics.

We also compare against the performance of our learned policy against that of our human pilot trained and licensed by Yamaha to fly the R-50 helicopter. Figures 5.6 and 5.7 shows the velocities and positions of the helicopter under our learned policy and under the human pilot's control. As we see, our controller was able to keep the helicopter flying more stably than was a human pilot.

5.4 Flying competition maneuvers

Having succeeded at learning to hover, we were next interested in making the helicopter automatically learn to fly several challenging maneuvers.

The Academy of Model Aeronautics (AMA) [79] organizes an annual RC helicopter competition. (The AMA boasts of a membership of over 170,000, and is to our knowledge by far the largest organization of this sort; it is also self-described as "The world's largest sport aviation organization.") In this competition, the helicopter has to be accurately flown through a number of maneuvers. The competition is organized into Class I (for beginners, with the easiest maneuvers) through Class III (with the most difficult maneuvers, for the most advanced pilots). We took the first three maneuvers from the most challenging, Class

III, segment of their competition.

Figure 5.8 shows maneuver diagrams from the web site of the AMA [79]. In the first of these maneuvers (III.1), the helicopter starts from the middle of the base of a triangle, flies backwards to the lower-right corner, performs a 180° pirouette (turning in place), flies backwards up an edge of the triangle, backwards down the other edge, performs another 180° pirouette, and lastly flies backwards to return to where it started. Flying a helicopter backwards is a significantly less stable maneuver than flying it forwards, which makes this maneuver interesting and challenging. In the second of these maneuvers (III.2), the helicopter has to perform a nose-in turn, in which it flies backwards out to the edge of a circle, pauses, and then accelerates into a trajectory that forms a full circle, but always keeping the nose of the helicopter pointed at center of rotation. This continues until the helicopter returns to where it had started circling, and it lastly flies forward back to the center of the circle. Many human pilots seem to find this second maneuver particularly challenging to fly. Lastly, maneuver III.3 involves flying the helicopter in a vertical rectangle, with two 360° pirouettes in opposite directions halfway along the vertical segments of the rectangle.

How does one design a controller for making a helicopter fly trajectories? Given a controller for keeping a system's state at a point $(x^*, y^*, z^*, \omega^*)$, one standard way to make the system move through a particular trajectory is to slowly vary $(x^*, y^*, z^*, \omega^*)$ along a sequence of set points on that trajectory. (E.g., see [33].) For instance, if we ask our helicopter to hover at $(0, 0, 0, 0)$, then a fraction of a second later ask it to hover at $(0.01, 0, 0, 0)$, then at $(0.02, 0, 0, 0)$ and so on, our helicopter will slowly fly in the x^s-

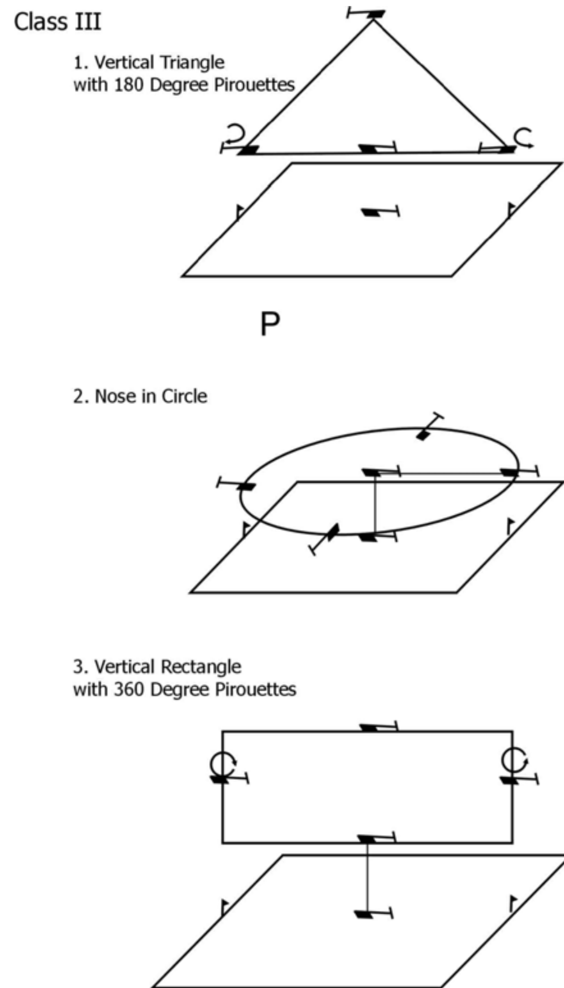


Figure 5.8: Diagrams of maneuvers from the Class III segment of an RC helicopter competition organized by the Academy of Model Aeronautics. [Source: `www.modelaircraft.org`]

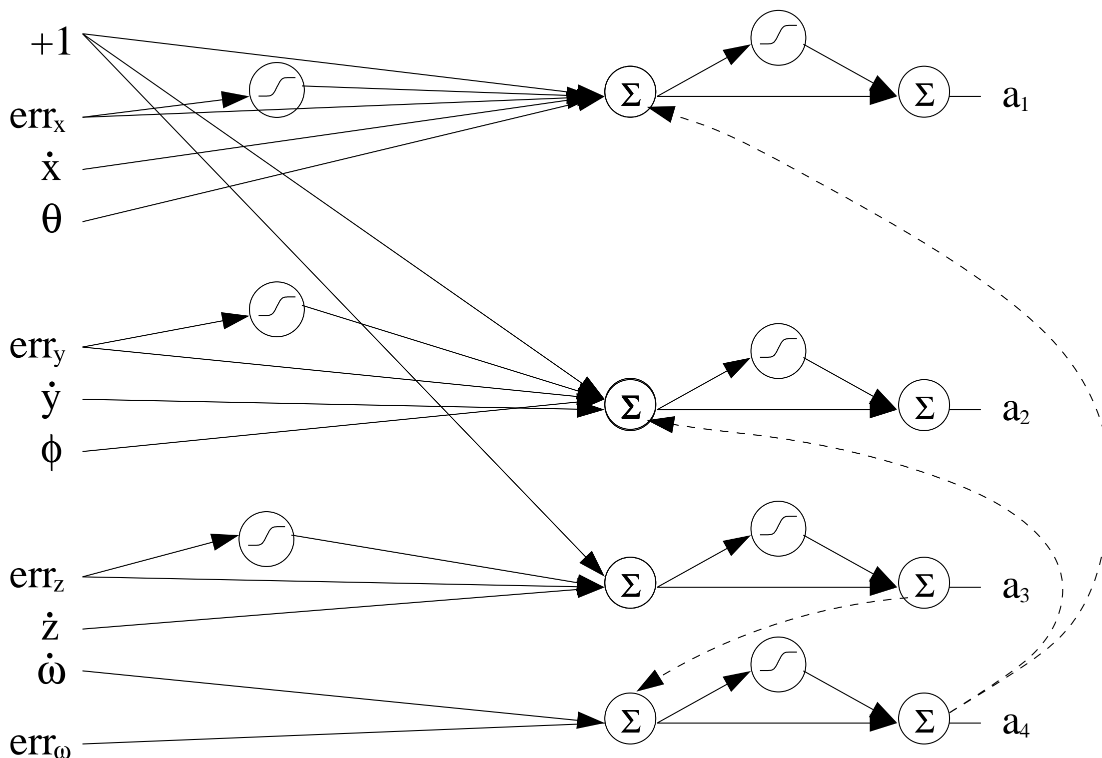


Figure 5.9: Policy class Π used to learn a controller for flying competition maneuvers. The dashed arrows show the newly-added edges.

direction. As in [96], to make the helicopter fly a particular trajectory, we thus made $(x^*, y^*, z^*, \omega^*)$ accelerate smoothly along the desired trajectory up to the speed limit, and decelerate smoothly at the end of the trajectory. By taking this procedure and "wrapping" it around our old policy class from Figure 5.4, we thus obtain a computer program—that is, a new policy class—not just for hovering, but also for flying arbitrary trajectories. I.e., we now have a family of policies that take as input a trajectory, and that attempt to make the helicopter fly that trajectory. Moreover, we can now also *retrain the policy's parameters for accurate trajectory following*, not just hovering.

Because we are now flying trajectories and not only hovering, we also augmented

the policy class to take into account more of the coupling between the helicopter's different subdynamics. For instance, the "straightforward" way to turn is to change the tail rotor collective pitch/thrust, so that it yaws either left or right. For small corrections to direction, this works well; but for larger turns, the thrust from the tail rotor also tends to cause the helicopter to drift sideways. Turning left this way causes the helicopter to drift rightwards, and turning right causes it to drift leftwards. To correct for it, we enriched the policy class to allow it to correct for this drift by applying the appropriate collective pitch controls (which, we recall, can be used to make the helicopter accelerate sideways/in the direction opposite to the drift). Also, having a helicopter climb or descend changes the amount of work done by the main rotor, and hence the amount of torque/anti-torque generated, which can cause the helicopter to enter an unwanted turn. To allow the policy class to correct for this, we also added a link between the collective pitch control a_3 and the tail rotor collective pitch control a_4. The result of these modifications to the policy are shown in Figure 5.9.

To apply our reinforcement learning algorithms, we also needed a reward function. One simple choice for R would have been to use Equation (5.9) with the newly-defined (time-varying) $(x^*, y^*, z^*, \omega^*)$. But we did not consider this to be a good choice. Specifically, consider making the helicopter fly in the increasing x-direction, so that $(x^*, y^*, z^*, \omega^*)$ starts off as $(0, 0, 0, 0)$ (say), and has its first coordinate x^* slowly increased over time. Then, while the actual helicopter position x^s will indeed increase, it will also almost certainly lag consistently behind x^*. This is because the hovering controller is always trying to "catch up" to the moving $(x^*, y^*, z^*, \omega^*)$. Thus, $x - x^*$ may remain large, and the helicopter will continuously incur a $x - x^*$ cost, even if it is in fact flying a very straight and accurate

trajectory in the increasing x-direction exactly as desired. It would be undesirable to have the helicopter risk trying to fly more aggressively to reduce this fake "error," particularly if it is at the cost of increased error in the other coordinates. So, we changed the reward function to penalize deviation not from $(x^*, y^*, z^*, \omega^*)$, but instead deviation from $(x_p, y_p, z_p, \omega_p)$, where $(x_p, y_p, z_p, \omega_p)$ is the "projection" of the helicopter's actual position onto the path of the idealized, desired trajectory. (In our example of flying in a straight line, for a helicopter at (x, y, z, ω), we easily see $(x_p, y_p, z_p, \omega_p) = (x, 0, 0, 0)$.) Thus, we imagine an "external observer" that looks at the actual helicopter state and estimates which part of the idealized trajectory the helicopter is trying to fly through (taking care not to be confused if a trajectory loops back on itself, such as in maneuver III.2), and the learning algorithm pays a penalty that is quadratic between the actual position and the "tracked" position on the idealized trajectory.

We also needed to make sure the helicopter is rewarded for actually making progress along the trajectory (as opposed to, say, staying at its initial position and not moving anywhere). To accomplish this, we used the shaping rewards of Chapter 3. Specifically, since we were already tracking where along the desired trajectory the helicopter was, we could then easily choose a potential function Φ that increases gradually along the desired trajectory. Thus, whenever the helicopter's "tracked" position $(x_p, y_p, z_p, \omega_p)$ makes forward progress along this trajectory, it would be climbing up the potential function, and hence receive positive $\Phi(s') - \Phi(s)$ reward (ignoring γ in this discussion).

Finally, our modifications have decoupled our definition of the reward function from $(x^*, y^*, z^*, \omega^*)$ and the evolution of $(x^*, y^*, z^*, \omega^*)$ in time. So, we are now also free

to consider allowing $(x^*, y^*, z^*, \omega^*)$ to evolve in a way that is *different* from the path of the desired trajectory, but nonetheless in way that hopefully allows the helicopter to follow the actual, desired trajectory more accurately. (In control theory, there is also a related practice of using the inverse dynamics to obtain better tracking behavior.) We considered a few alternatives, such as allowing $(x^*, y^*, z^*, \omega^*)$ to follow a larger or smaller circle than the desired one in maneuver III.2, but the main one that we used ended up being a modification to how we fly trajectories that have both a vertical and a horizontal component (such as along the two upper edges of the triangle in III.1). Specifically, it turns out that the z (vertical)-response of the helicopter is very fast: For example, to climb, we need only increase the cyclic pitch control, which almost immediately affects thrust, so that the helicopter starts accelerating upwards very quickly. In contrast, the x and y responses seem much slower. Thus, if $(x^*, y^*, z^*, \omega^*)$ moves at 45° upwards (say) as in maneuver III.1, the helicopter will tend to track the z-component of the trajectory much more quickly, so that, rather than flying in a straight line, it will tend to initially accelerate into a climb steeper than 45°, resulting in a "bowed-out" trajectory. Similarly concerns also apply for an angled descent, resulting in a "bowed-in" trajectory. To correct for this, we therefore artificially slowed down the z-response. In particular, when $(x^*, y^*, z^*, \omega^*)$ is moving into an angled climb or descent, the (x^*, y^*, ω^*) portion will evolve normally with time, but the changes to z^* will be delayed by t seconds, where t here is another parameter in our policy class, to be automatically learned by our algorithm.

Using this setup and retraining our policy class' parameters for accurate trajectory following, we were able to learn a policy that flies all three of the competition maneuvers

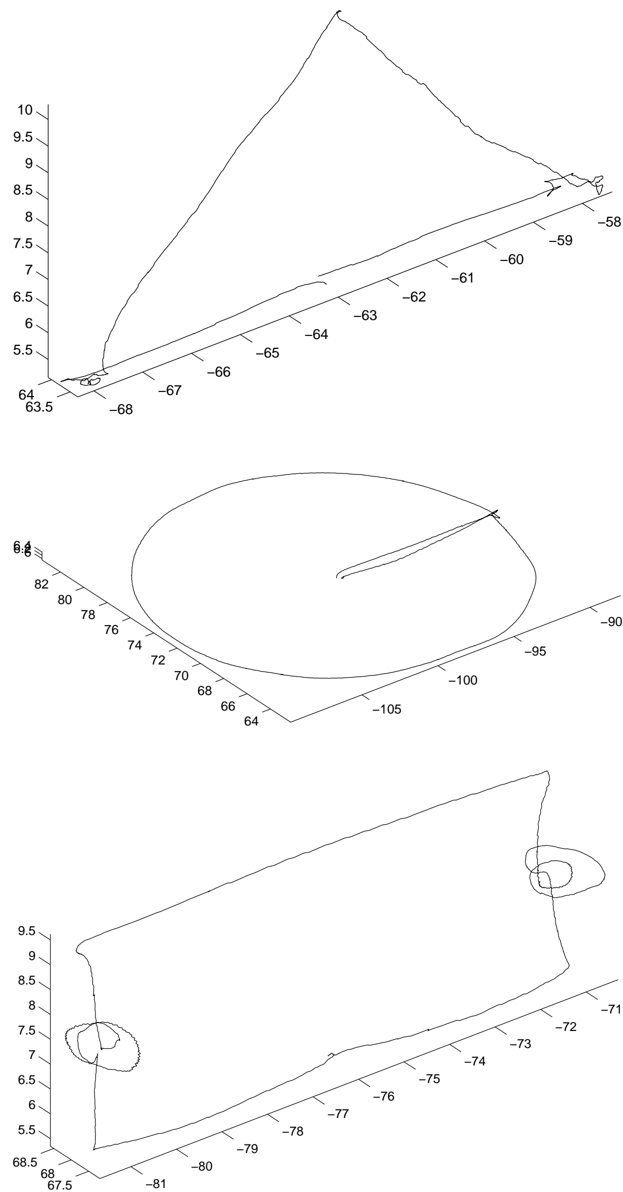


Figure 5.10: Plots of example helicopter trajectories (as recorded by the onboard telemetry) flying the three competition maneuvers.

fairly accurately. Figure 5.10 shows actual trajectories taken by the helicopter while flying these maneuvers.

Chapter 6

Conclusions

Recent years have seen numerous successes of reinforcement learning approaches to control and decision making under uncertainty. [99, 85, 64, 20, 26, 47] Yet, many issues pertaining to the practical application of reinforcement learning algorithms remain. In this dissertation, we presented some methods for reinforcement learning that sought to address some of these issues.

One of these issues was that of specifying the task description, or reward function. Specifically, shaping rewards are often used to provide necessary hints to a learning algorithm to enable it to learn in a reasonable amount of time. But, poorly chosen shaping rewards can result in poor policies being learned, in which case some amount of human trial and error is typically needed to design better shaping rewards. In Chapter 3, we characterized necessary and sufficient conditions under which shaping rewards may be proved to guarantee optimal policies being learned. Our analysis also gave guidelines for choosing shaping rewards. We then showed that shaping can permit learning algorithms using a

reduced horizon-time to learn well, and thus in some sense formally reduces the "difficulty" of a reinforcement learning problem for these (myopic) algorithms. The form of the shaping rewards proposed were also demonstrated to work well on some problems.

In Chapter 4, we considered the policy search problem, and saw that a key issue in policy search was obtaining uniformly good estimates of policies' utilities. We saw that simple Monte Carlo methods cannot accomplish this, and also discussed the trajectory tree method, which obtains uniformly good estimates but at a prohibitively (exponentially) large computational cost. We then showed that all reinforcement learning problems can be reduced to ones in which all the state transitions are deterministic. This was used to derive the PEGASUS algorithm, which guaranteed uniformly good estimates of policies' utilities, and had at most a polynomial sample complexity. In presenting these results, we also used a generalization of the familiar ideas of VC dimension and sample complexity from the setting of supervised learning to that of reinforcement learning, thus putting the two problems on a more equal footing.

In Chapter 5, these ideas were combined to design a controller for an autonomous helicopter. Autonomous control of helicopters is widely considered a challenging problem, but using these algorithms, we were able to automatically design a very stable hovering controller, as well as fly a number of maneuvers taken from an RC helicopter competition.

This dissertation also provides the ground work, and suggests some directions, for future work in reinforcement learning and adaptive control. Many pressing problems still remain; some of them are:

- **Specification of reward function.** While we have given a method for specifying

shaping rewards, we view the problem of specifying rewards in general as still a very difficult one. Specifically, consider the task of learning to drive on a highway. Here, we might want to trade off many factors such as maintaining a safe following distance, keeping a safe distance from the curb, maintaining a reasonable speed, avoiding pedestrians, avoiding jerky starts and stops, preferring driving in the middle lane, and so on. However, it is very difficult to sit down and write out a reward function capturing exactly how much weight to give to each of these things and exactly how these things should be traded off against each other. Is there a better way to specify reward functions? We think inverse reinforcement learning algorithms [78] might provide one alternative if a teacher is available to demonstrate driving, but are the other ways?

- **Safety and robustness.** While our results guarantee that we can trust the estimated utilities of policies to be close to the true utilities, traditional control theory also defines notions such as the stability and robustness of a policy. For instance, if our model for the MDP were slightly incorrect, can we still guarantee good performance? In safety-critical applications including helicopter flight, it would be also desirable to give guarantees in terms of the traditional notions of stability and robustness. Some of the ideas in this dissertation can be applied to these problems, and we think safety guarantees is an important issue for broad acceptance of reinforcement learning in safety-critical applications. More broadly, fairly different problems and problem formulations have traditionally been considered in the reinforcement learning and in the classical control literatures, and we believe it will be a fruitful endeavor to explore connections that may bring the two fields closer together.

- **Multi-agent systems.** Throughout this dissertation, we have discussed reinforcement learning for a single agent. But a number of applications require multiple, perhaps distributed, agents, where sometimes the agents have limited communication bandwidth. One example is that of distributed nodes controlling a power-grid [92]. Another example is multiple economic agents acting in a market [36]. If the agents do not have necessarily identical goals, then the characterization of the agents' behaviors fall into the purview of game theory [80, 35]. How can reinforcement learning algorithms be adapted to these settings?

- **Unsupervised learning.** In Chapter 1, we gave a comparison between supervised learning and reinforcement learning, in which we said that some properties of the latter that distinguish it from supervised learning and that make it difficult are the sequential nature of the decision making problem, and the issue of delayed rewards and consequences. In this dissertation, we applied some ideas familiar from supervised learning, such as VC dimension and sample complexity, to the reinforcement learning setting. In Artificial Intelligence, there is also the unsupervised learning problem, in which our algorithms are given data and asked to find interesting "structure" in the data, rather asked to learn to perform a specific (supervised) classification or regression task. By analogy, we believe it would also be interesting to explored "unsupervised" sequential decision making problems, in which rather than learning to maximize a particular reward function, our algorithms are asked only to discover interesting structure in a problem.

This dissertation has presented methods and theory for reinforcement learning

that we believe will be useful for many problems. And, despite the many successes that reinforcement learning has seen, many interesting and important problems in learning and adaptive control remain. We believe that further efforts to study these problems will pay off richly.

Bibliography

[1] B. D. O. Anderson and J. B. Moore. *Optimal Control: Linear Quadratic Methods.* Prentice-Hall, 1989.

[2] John D. Anderson. *Fundamentals of Aerodynamics, 3rd ed.* McGraw-Hill, 2001.

[3] John D. Anderson. *Introduction to Flight, 4th ed.* McGraw-Hill, 2001.

[4] David Andre and Stuart Russell. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, pages 1019–1025, 2001.

[5] M. Anthony and P. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 1999.

[6] C. Atkeson, S. Schaal, and A. Moore. Locally weighted learning. *AI Review*, 11, 1997.

[7] J. Bagnell, 2001. Pers. Comm.

[8] J. Bagnell and J. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the International Conference on Robotics and Automation.* IEEE, 2001.

[9] L. Baird and A.W. Moore. Gradient descent for general Reinforcement Learning. In *NIPS 11*, 1999.

[10] Leemon C. Baird. Reinforcement Learning in continuous time: Advantage updating. In *Proceedings of the International Conference on Neural Networks*, 1994.

[11] G. Balas, J. Doyle, K. Glover, A. Packard, and R. Smith. μ-analysis and synthesis toolbox user's guide, 1995.

[12] J. Baxter and P. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[13] R. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, 1961.

[14] Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[15] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Volume I.* Athena Scientific, 1995.

[16] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Volume II.* Athena Scientific, 1995.

[17] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic Programming.* Athena Scientific, 1996.

[18] John R. Birge and Francois Louveaux. *Introduction to Stochastic Programming.* Springer, 1997.

[19] D. Blackwell. Discrete dynamic programming. *Annals of Mathematical Statistics*, 33:719–726, 1962.

[20] Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems 6*, pages 671–678, 1993.

[21] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Proc. UAI*, pages 33–42, 1998.

[22] A. Cassandra. *Exact and approximate algorithms for partially observable Markov decision processes.* PhD thesis, Brown University, 1998.

[23] A. Cassandra, L. Kaelbling, and M. Littman. Efficient dynamic-programming updates in partially observable Markov decision processes. Technical Report CS-95-19, Brown University, 1995.

[24] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[25] W. Cleveland. Robust locally weighted regression and smoothing scatterplots. *J. Amer. Stat. Assoc*, 74, 1979.

[26] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1017–1023, 1996.

[27] Daniela Pucci de Farias. *The linear programming approach to approximate dynamic*

programming: Theory and application. PhD thesis, Department of Management Science and Engineering, Stanford University, 2002.

[28] T. G. Dietterich and X. Wang. Batch value function approximation via support vectors. In *Advances in Neural Information Processing Systems 14*, 2002.

[29] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[30] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley and Sons, 1973.

[31] Richard Durrett. *Probability : Theory and Examples, 2nd edition.* Duxbury, 1996.

[32] Michael Evans and Tim Swartz. *Approximating integrals via Monte Carlo and deterministic methods.* Number 20 in Oxford Statistical Science Series. Oxford University Press, 2000.

[33] Gene F. Franklin, J. David Powell, and Abbas Emani-Naeini. *Feedback Control of Dynamic Systems.* Addison-Wesley, 1995.

[34] M. Fu and J. Hu. *Conditional Monte Carlo.* Kluwer Academic Publishers, 1997.

[35] D. Fudenberg and J. Tirole. *Game Theory.* MIT Press, 1991.

[36] Drew Fudenberg and David K. Levine. *The Theory of Learning in Games.* MIT Press, 1998.

[37] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis.* Chapman & Hall/CRC, 2000. Chapter 8.

[38] Curtis F. Geralk and Patrick O. Wheatley. *Applied Numerical Analysis (4th Ed.)*. Addison-Wesley, 1989.

[39] P. Glasserman. *Gradient estimation via pertubation analysis.* Kluwer Academic Publishers, 1991.

[40] P. W. Goldberg and M. R. Jerrum. Bounding the Vapnik-Chervonenkis dimension of concept classes parameterized by real numbers. *Machine Learning*, 18:131–148, 1995.

[41] Geoffrey Gordon. *Approximate Solutions to Markov Decision Processes.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

[42] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projections for factored mdps. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[43] M. E. Harmon and L. C. Baird. Spurious solutions to the bellman equation. Technical Report WL-TR-96-'To Be Assigned', Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1996.

[44] D. Haussler. Decision-theoretic generalizations of the PAC model for neural networks and other applications. *Information and Computation*, 100:78–150, 1992.

[45] O. Hernández-Lerma. *Adaptive Markov Control Processes.* Number 79 in Applied Mathematical Sciences. Springer-Verlag, 1989.

[46] Y. Ho and X. Cao. *Pertubation analysis of discrete event dynamic systems.* Kluwer Academic Publishers, 1991.

[47] T. Joachims, D. Freitag, and T. Mitchell. Webwatcher: A tour guide for the world wide web. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[48] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[49] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 2002.

[50] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. Approximate planning in large POMDPs via reusable trajectories. *(extended version of paper in NIPS 12)*, 1999.

[51] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[52] Michael Kearns, Robert Schapire, and Linda Sellie. Towards efficient agnostic learning. *Machine Learning*, 17:115–141, 1994.

[53] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

[54] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, Massachusetts, 1994.

[55] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[56] Hyounjin Kim and Andrew Y. Ng. Policy search, robust control, and linear quadratic regulators. (In preparation), 2002.

[57] H. Kimura, M. Yamamura, and S. Kobayashi. Reinforcement learning by stochastic hill climbing on discounted reward. In *Proceedings of the Twelfth International Conference on Machine Learning*, 1995.

[58] Daphne Koller and Ronald Parr. Policy iteration for factored mdps. In *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference*, 2000.

[59] J. Gordan Leishman. *Principles of Helicopter Aerodynamics.* Cambridge Aerospace Series. Cambridge University Press, 2000.

[60] M. Littman, T. Dean, and L. Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

[61] W. Lovejob. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47–65, 1991.

[62] W. Lovejoy. Computationally feasible bounds for partially observed Markov decision process. *Operations Research*, 39, 1991.

[63] Christopher Lusena, Martin Mundhenk, and Judy Goldsmith. Nonapproximability

results for partially observable markov decision processes. *Journal of AI Research*, 14:83–103, 2001.

[64] S. Mahadevan, N. Marchalleck, T. Das, and G. Abhihit. Self improving factory simulation using continuous-time reinforcement learning. In *Proceedings of the 14th International Conference on Machine Learning*, pages 202–210, 1997.

[65] A. Manne. Linear programming and sequential decisions. *Management Science*, 6:259–267, 1960.

[66] Maja J Mataric. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[67] P. S. Maybeck. *Stochastic models, estimation and control.* Academic Press, 1982.

[68] D. McAllester and S. Singh. Approximate planning for factored POMDPs using simplified belief states. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, 1999.

[69] J. L. McClelland and D. E. Rumelhart. *Parallel Distributed Processing.* MIT Press, 1986.

[70] B. Mettler, M. Tischler, and T. Kanade. System identification of small-size unmanned helicopter dynamics. In *American Helicopter Society, 55th Forum*, 1999.

[71] Bernard Mettler, Mark Tischler, and Takeo Kanade. System identification of a model-scale helicopter. Technical Report CMU-RI-TR-00-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[72] N. Meuleau, L. Peshkin, K-E. Kim, and L.P. Kaelbling. Learning finite-state controllers for partially observable environments. In *Uncertainty in Artificial Intelligence, Proceedings of the Fifteenth conference*, 1999.

[73] A. W. Moore, J. Schneider, and K. Deng. Efficient locally weighted polynomial regression predictions. In *Proceedings of the 1997 International Machine Learning Conference*. Morgan Kaufmann Publishers, 1997.

[74] Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[75] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287, Bled, Slovenia, July 1999. Morgan Kaufmann.

[76] Andrew Y. Ng and Michael I. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference*, pages 406–415, 2000.

[77] Andrew Y. Ng, Ronald Parr, and Daphne Koller. Policy search via density estimation. Advances in Neural Information Processing Systems 12, 1999.

[78] Andrew Y. Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[79] Academy of Model Aeronautics. `http://www.modelaircraft.org`.

[80] G. Owen. *Game Theory.* Academic Press, 1995.

[81] Ronald Parr. *Hierarchical Control and Learning in Markov Decision Processes.* PhD thesis, Computer Science Division, University of California, Berkeley, 1998.

[82] D. Pollard. *Empirical Processes: Theory and Applications.* NSF-CBMS Regional Conference Series in Probability and Statistics, Vol. 2. Inst. of Mathematical Statistics and American Statistical Assoc., 1990.

[83] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[84] UC Berkeley Millennium Project. `http://www.millennium.berkeley.edu`.

[85] M. Puterman. *Markov Decision Processes.* Wiley, 1994.

[86] J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

[87] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:40–407, 1951.

[88] Benjamin Van Roy. *Learning and Value Function Approximation in Complex Decision Processes.* PhD thesis, Massachusetts Institute of Technology, 1998.

[89] Stuart Russell and Eric Wefald. *Do the right thing : studies in limited rationality.* Artificial intelligence. MIT Press, Cambridge, Mass, 1991.

[90] John Rust. Do people behave according to Bellman's principal of optimality? Submitted to Journal of Economic Perspectives, 1994.

[91] L.M. Saksida, S.M. Raymond, and D.S. Touretzky. Shaping robot behavior using principles from instrumental conditioning. *Robotics and Autonomous Systems*, 22(3–4):231–249, 1997.

[92] Jeff Schneider, Weng-Keen Wong, Andrew Moore, and Martin Riedmiller. Distributed value functions. In *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999.

[93] J. Seddon. *Basic Helicopter Aerodynamics*. AIAA Education Series. America Institute of Aeronautics and Astronautics, 1990.

[94] Christian R. Shelton. Policy improvement for pomdps using normalized importance sampling. In *Proceedings of the Seventeenth International Conference on Uncertainty in Artificial Intelligence*, pages 496–503, 2001.

[95] D. Shim, 2001. Pers. Comm.

[96] Hyunchul Shim. *Hierarchical Flight Contorl System Synthesis for Rotorcraft-based Unmanned Aerial Vehicles*. PhD thesis, Mechanical Engineering, University of California, Berkeley, 2000.

[97] Satinder Singh and Richard Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.

[98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[99] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[100] Sebastian Thrun. A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, School of Computer Science, Carnegie Mellon University, 1998.

[101] Joseph F. Traub and Arthur G. Werschulz. *Complexity and information.* Lezioni Lincee, Accademia Nazionale dei Lincei. Cambridge University Press, 1999.

[102] V. N. Vapnik. *Statistical Learning Theory.* John Wiley & Sons, 1998.

[103] V.N. Vapnik. *Estimation of Dependences Based on Empirical Data.* Springer-Verlag, 1982.

[104] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, Princeton, New Jersey, first edition, 1944.

[105] W. J. Wagtendonk. *Principles of Helicopter Flight.* Aviation Supplies and Academics, 1996.

[106] C. J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, Cambridge University, 1989.

[107] C. White. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 32, 1991.

[108] J.K. Williams and S. Singh. Experiments with an algorithm which learns stochastic memoryless policies for POMDPs. In *NIPS 11*, 1999.

[109] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[110] Ronald J. Williams and Leemon C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. In *Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems*, 1994.

[111] N. Zhang, S. Lee, and W. Zhang. A method for speeding up value iteration in partially observable Markov decision processes. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, 1999.