EFFICIENT MODEL-BASED EXPLORATION IN CONTINUOUS STATE-SPACE ENVIRONMENTS

BY ALI NOURI

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Michael L. Littman

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2011

ABSTRACT OF THE DISSERTATION

Efficient Model-based Exploration in Continuous State-space Environments

by Ali Nouri

Dissertation Director: Michael L. Littman

The impetus for exploration in reinforcement learning (RL) is decreasing uncertainty about the environment for the purpose of better decision making. As such, exploration plays a crucial role in the efficiency of RL algorithms. In this dissertation, I consider continuous state control problems and introduce a new methodology for representing uncertainty that engenders more efficient algorithms. I argue that the new notion of uncertainty allows for more efficient use of function approximation, which is essential for learning in continuous spaces. In particular, I focus on a class of algorithms referred to as model-based methods and develop several such algorithms that are much more efficient than the current state-of-the-art methods. These algorithms attack the long-standing "curse of dimensionality"— learning complexity often scales exponentially with problem dimensionality. I introduce algorithms that can exploit the dependency structure between state variables to exponentially decrease the sample complexity of learning, both in cases where the dependency structure is provided by the user *a priori* and cases where the algorithm has to find it on its own. I also use the new uncertainty notion to derive a multi-resolution exploration scheme, and demonstrate how this new technique achieves *anytime* behavior, which is very important in real-life applications. Finally, using a set of rich experiments, I show how the new exploration mechanisms

affect the efficiency of learning, especially in real-life domains where acquiring samples is expensive.

Acknowledgements

It would have been a near impossible to accomplish this dissertation without the help and support of many people. I am greatly humbled for all their assistance and encouragements.

I would like to express my deepest gratitude to my advisor, Michael L. Littman, who has been an extraordinary mentor for me throughout my whole graduate study. He patiently spent enormous amount of his time teaching me how to think about problems and how to approach them in a systematic way. He showed me how to organize my solutions and how to write and present them in a professional manner. I greatly admire his wisdom, friendship , and last but not least, his sense of humor.

I would also like to thank my committee members, Michael L. Littman, Vladimir Pavlovic, Casimir Kulikowski, and Doina Precup, for their support and invaluable comments.

A huge portion of the ideas in this document stem from collaborations with my colleagues in RL^3 lab. I cannot imagine a more vibrant and exciting environment to work in. I learned a great deal from the members of the lab, both scientifically and culturally, and I believe they played a very important role in my getting mature over the past several years. A special thanks goes to Tom Walsh and Lihong Li for their significant contributions to my research path.

I am very lucky to be surrounded by so many brilliant and talented friends, whose support, ideas and academic suggestions have inspired me in my research career. I would like to thank my amazing wife Zahra Mohajerani and dear friend Mohammad Sedighi for the numerous times they helped me with linear algebra and mathematics. And member of the Arian RoboCup team, Mazda Ahmadi, Mayssam Sayyadian and Mayssam Mohammadi for the incredible research journey we had together.

Last, but certainly not least, I would like to thank my lovely family and amazing wife, Zahra, for their unconditional love and support and encouragement. Maman, I cannot begin to imagine how I could've emotionally pulled off my journey to USA if it weren't for your blessing. I know I wasn't there for you in the hardest of times, but you always were there for me. Shiva, Shadi, and Afshan, thank you for always believing in me and encouraging me to pursue my research interests. I would've definitely not be here if I didn't have you. Zahra, thank you so much for always being there for me, during the happy times with laughter and love, during the deadlines with patience and support, and during the sad times with encouragement and hope. Thank you for keeping me sane!

Dedication

This dissertation is dedicated to my father who once told me his biggest wish is for me to get my Ph.D. and make him proud. His place is empty among us, but his memories give me encouragement and hope!

A word from the author

According to ..., as of December 2010:

1. Paper consumption has increased six-fold over the past 50 years.
2. The average US office worker uses over 10,000 sheets of printing and copying paper per year and generates 2 *lbs.* of paper waste per day.
3. Paper and packaging waste make up over 40% of North American solid waste landfill.

According to Rutgers annual report, over 15 million pages of paper have been printed at school alone in the year 2010.

Please save a tree by not printing this document!

Table of Contents

Abstract ii

Acknowledgements iv

Dedication vi

List of Tables xii

List of Figures xiii

List of algorithms 1

1. Introduction 2

1.1. Contributions 4

1.2. Overview 6

2. Reinforcement-Learning Background 8

2.1. Learning Through Interaction 8

2.2. Markov Decision Processes (MDPs) 10

2.3. Planning vs. Learning in MDPs 15

2.4. Challenges 16

2.5. Efficiency of RL Algorithms 18

2.6. Reinforcement-Learning Algorithms 20

2.6.1. Policy-Search Methods 20

2.6.2. Value-based Methods 21

2.6.3. Model-based Methods 22

3. Self-aware Exploration 24

3.1. Known vs. Unknown 24
3.1.1. KWIK and Rmax 27
3.1.2. More Examples 29
3.1.3. Disadvantages of the KWIK framework 35
3.2. Continuous Knownness 36
3.2.1. CKWIK-Rmax 38

4. Model-based Learning with CKWIK Framework 41
4.1. CF-Rmax 41
4.1.1. CKWIK Kernel Regression 42
Sample-complexity Analysis 46
4.1.2. The Proposed Algorithm 51
4.1.3. Empirical Results 52
Experiment Setups 53
4.2. Factored Learner for Continuous Spaces 55
4.2.1. Factored-state MDPs 58
4.2.2. FF-Rmax 60
4.2.3. Discussion 62
4.2.4. Analysis 65
4.2.5. Experimental Results 66

5. Automatic Discovery of Relevant Features 74
5.1. Background 75
5.2. Dimension Reduction in Regression 76
5.2.1. Kernel Regression and Metric Learning 76
5.2.2. Factorization of MLKR 84
5.3. The Proposed Algorithm 86
5.4. Discussion 87
5.5. Related Work and Future Extensions 89

5.5.1. Kernel Regression with Unsupervised Dimension Reduction . . . 90
5.5.2. Partial Least Squares . . . 92
5.5.3. Gaussian Processes . . . 94
5.6. Experimental Results . . . 97

6. Extension: Multi-resolution Exploration . . . 105
6.1. Multi-resolution Discretization . . . 107
6.1.1. Uncertainty Trees . . . 109
6.2. Proposed Algorithm . . . 112
6.3. Discussion . . . 114
6.3.1. Application to Value-based RL . . . 114
6.4. Experimental Results . . . 115
6.5. Conclusion . . . 119

7. Concluding Remarks . . . 121

Appendix A. Proofs . . . 123
A.1. Proof of Theorem 14 . . . 123
A.2. Proof of Lemma 17 . . . 127

Appendix B. Planning in Markov Decision Processes . . . 131
B.1. Value Iteration . . . 131
B.2. Approximate Planning in Continuous Spaces . . . 134
B.2.1. Discretization . . . 134
B.2.2. Fitted Value Iteration (FVI) . . . 139
B.3. Forward Planning . . . 140
B.3.1. UCT . . . 142

Appendix C. Technical Details of Environments . . . 145
C.1. Mountaincar . . . 145
C.1.1. n-Mountaincar . . . 146

C.2. Puddleworld 146
C.2.1. n-Puddleworld 148
C.3. Warpedworld 148
C.4. Bumbleball 149

Appendix D. Mathematical Facts 153
D.1. Learnability of Continuous MDPs 153
D.2. Kernel Functions 154

References 155
Vita 168

List of Tables

4.1. Performance of three algorithms in the Bumbleball domain. 71

5.1. Performance of DRE in the Bumbleball domain. 104

List of Figures

2.1. A diagram of reinforcement learning 9

2.2. A simple finite-state MDP example 11

2.3. Problem configurations stretching between planning and learning. 16

2.4. Structure of a typical model-based algorithm 23

3.1. Example of a DBN describing a dependency structure 30

3.2. Illustration of ball-heuristic 35

4.1. Effect of the number of neighbors used in kernel regression 45

4.2. CF-Rmax and fitted-Rmax learning in MOUNTAINCAR 53

4.3. CF-Rmax and fitted-Rmax in PUDDLEWORLD 54

4.4. Comparison of knownness functions in fitted-Rmax and CF-Rmax 55

4.5. The effect of kernel width on CF-Rmax and fitted-Rmax 56

4.6. A simple example of dependency graphs for a 3-dimensional MDP. . . . 60

4.7. Performance of FF-Rmax and CF-Rmax in 4-PUDDLEWORLD. 68

4.8. Performance of FF-Rmax and CF-Rmax in n-PUDDLEWORLD 68

4.9. Comparison of knownness in CF-Rmax and FF-Rmax 69

4.10. The true dependency graph of n-MOUNTAINCAR 71

4.11. Two alternative dependency graphs for 3-MOUNTAINCAR 72

4.12. CF-Rmax and variations of FF-Rmax in n-MOUNTAINCAR 73

5.1. Illustrations of different distance metrics 78

5.2. Illustration of the kernel width σ 82

5.3. Comparison of MLKR and FMLKR on a simple regression problem. . . 86

5.4. Examples of three hyper-parameter classes for GP 96

5.5. Offline evaluation of three algorithms in n-MOUNTAINCAR 98

5.6. Comparison of three algorithms in n-PUDDLEWORLD 99
5.7. Three knownness functions: fitted-Rmax, CF-Rmax, and DRE 100
5.8. Comparison of fitted-Rmax and DRE on (1,3)-MOUNTAINCARS 101
5.9. Parameter selection of Q-learning in various environments 102
5.10. DRE, and two versions of FF-Rmax in 3-WARPEDWORLD 104
6.1. An example of the discretization used in MRE 109
6.2. Illustration of an uncertainty tree 110
6.3. Comparison of MRE with two fixed discretizations in MOUNTAINCAR . . 116
6.4. Stage-wise Comparison of three algorithms in MOUNTAINCAR 117
6.5. Value function of MRE and fixed discretization in MOUNTAINCAR 117
6.6. Fitted Q-iteration with three different explorations in MOUNTAINCAR . . . 119
6.7. Knownness of $\mathrm{UT}_{\mathrm{boolean}}$ and $\mathrm{UT}_{\mathrm{continuous}}$ in MOUNTAINCAR 120
B.1. Discretizing a 2D state space using the cell-state method 136
B.2. Illustration of Kuhn-triangulation in 2D and 3D state spaces 137
B.3. Illustration of sparse sampling algorithm 141
C.1. MOUNTAINCAR domain. 145
C.2. PUDDLEWORLD domain. 147
C.3. A top view of the BUMBLEBALL field. 149
C.4. Pictures of three models of Aibo 150
C.5. Picture of a bumble ball. 150
C.6. Distributed architecture of the BUMBLEBALL environment. 151
D.1. Several examples of kernel functions 155
D.2. Different values of σ in the Gaussian kernel. 156

List of Algorithms

1. Rmax 26
2. KWIK-Rmax 27
3. KWIK-LR 32
4. CKWIK-Rmax 39
5. CKWIK-KR 46
6. CF-Rmax 51
7. CKWIK-FKR 62
8. FF-Rmax 62
9. Multivariate MLKR 83
10. Multivariate MLKR with explicit dimension reduction. 84
11. DRE 88
12. DRE with unsupervised dimension reduction. 91
13. SIMPLS 93
14. MRE 113
15. Value Iteration 132
16. Sample-based Value Iteration 133
17. Generating samples from $\tilde{T}(\xi_i, a)$ 139
18. UCT 143

Chapter 1

Introduction

Many real-life learning applications, such as robotics and process control, can be formulated as sequential decision-making problems. The essence of learning in these systems is their temporal aspect, which requires the agents to chain actions together to form behaviors. In these problems, the utility associated with a behavior is usually revealed to the learner piece by piece through interaction with the environment.

For example, consider an automated pilot that is learning to fly an airplane. At any time during flying, the pilot observes the current readings of all the sensors on board and has to decide what controls to apply. To achieve optimality (flying smoothly and avoid crashing), the learner needs to apply many different controls one after another to keep the plane stable during the flight. The autopilot is never told directly what set of controls to apply at each situation to achieve optimality. The only feedback it receives is a penalty whenever the plane crashes. It is up to the learner to infer what set of controls in the long history of actions it took are responsible for the crash.

Techniques from the field of reinforcement learning are an excellent choice for attacking these tasks, since they address the problem of learning through interaction. Unlike supervised learning algorithms, these algorithms carry out their learning without requiring access to explicit examples of correct or incorrect behaviors. Instead, knowledge is acquired via actively experimenting with different control strategies. Many of the aforementioned learning problems, which includes robotics, computer games, adaptive optimal control, sensor networks and many others, have the flavor of learning through interaction and that is why reinforcement learning fits nicely with this rich body of learning problems.

One of the central properties of real-life learning problems is that they all have very

large state spaces. For example, a typical computer game can have more than 10^{10} distinct states, and some others have effectively an infinite number of states, as they can take on a continuous range of states. For example, a very simple-minded representation of state in our auto-pilot example has to at least include position, velocity and acceleration in the $(x, y, z, yaw, pitch)$ coordinates, which is a 15-dimensional vector.

Although all machine-learning techniques are susceptible to having trouble with large-scale problems, algorithms for sequential decision making are even more sensitive to the size of the state space because of the need for active exploration to acquire useful data, the temporal aspect of learning, and the nature of tabula-rasa search.

RL algorithms can affect the samples they collect during the learning. But, they can only do so indirectly via the actions they choose. Therefore, they are faced with a double optimization task: First, they need to take actions that maximize the performance given the current knowledge about the environment (exploitation of knowledge), and second they need to take actions that result in collecting potentially useful samples, which in part can be used to achieve better performance (exploration of the environment). These two optimization tasks may be conflicting because actions that gather information about the environment are not necessarily the optimal ones. As a result, an effective strategy for exploring the environment and a way to balance between exploration and exploitation is a vital component of any RL algorithm.

Continuous spaces broaden the applicability of RL algorithms to a whole new level. But, unfortunately, most of the theoretically sound and practical RL algorithms are designed for finite spaces. This thesis takes a step toward constructing efficient algorithms for continuous spaces. In particular, we focus on the problem of efficiently balancing exploration and exploitation in RL algorithms in continuous state-space problems.

The challenging fact about continuous state spaces is that since they have an infinite number of states, lookup-table approaches, which are widely used in RL algorithms for finite spaces, cannot be applied directly. Hence, some sort of function approximator that can generalize knowledge from one state to another must be used. The bulk of this thesis focuses on investigating efficient exploration techniques when a function

approximator is used within an RL algorithm. In particular, it introduces a new family of model-based algorithms that use an uncertainty measure to actively drive exploration toward less-known areas of the state space. It both presents theoretical properties of this group of algorithms and empirically evaluates several instances of such algorithms in multiple environments including simulations and a real robotics domain.

1.1 Contributions

This thesis introduces a family of new techniques for efficiently balancing exploration and exploitation in model-based algorithms in continuous state spaces. At the heart of these algorithms is an uncertainty-driven exploration mechanism that enables these algorithms to systematically drive exploration toward less-known areas of the state space, resulting in a more accurate model from which higher rewards can be obtained. This strategy is achieved by the introduction of a new concept called "knownness".

The knownness is a continuous variable in the range from 0 to 1 that measures the certainty level of an agent's function approximator for any given input. The algorithms proposed in this dissertation directly incorporate this uncertainty level into the value of states in the form of a bonus value, thus forming an implicit exploration strategy.

A similar but more restrictive concept was already presented in the literature— widely known as "Rmax exploration" (Brafman and Tennenholtz, 2002). This approach divides states into two groups— known and unknown states—and constructs an exploration strategy that focuses on reaching unknown states by assigning them maximum possible values. The knownness concept generalizes this approach to something that better suits function approximation and is far more data efficient. To demonstrate the versatility of this approach, we introduce several algorithms using this new concept, all of which follow the same skeleton structure.

The first algorithm in this family is an extension of the existing algorithm fitted-Rmax, which uses Rmax exploration. This algorithm is designed to attack a broad class of continuous state-space MDPs. The only assumption the algorithm makes about the environment is that its dynamics are smooth and that the transition function of

each state-action pair can be modeled using the mean of the probability distribution it generates over the states. A formal analysis of the sample complexity of this algorithm is provided, making it the first algorithm in this class of environments with guarantees in the *probably approximately correct MDP (PAC-MDP)* framework. Experimental data shows that the new algorithm is empirically much more data efficient than its predecessors.

The next method we examine is an algorithm that can take in prior knowledge from the user to exponentially increase the learning speed in factored continuous spaces. The user provides information about what state variables are irrelevant to the prediction of the next state using a dependency structure similar to the ones used in graphical models such as dynamic Bayesian networks. The algorithm uses this information in its function approximator and knownness function. In a sense, the information is used to perform *dimension reduction* while approximating the transition function. By reducing the space in which the algorithm makes predictions and computes knownness, the algorithm is able to dramatically reduce its sample complexity. This algorithm is reminiscent of the factored-Rmax algorithm for finite spaces, but extends it to continuous state spaces and the cases where function approximation is used.

The next algorithm introduced in this dissertation is a technique that employs dimension reduction in exploration without any prior knowledge. This algorithm automatically discovers relevant dimensions of the environment and maintains a low-dimensional representation of the transition function using techniques from dimension reduction in regression. Using the same "self-aware" exploration scheme as before, in combination with the compact representation of the system dynamics, this algorithm is able to achieve significant speedup in learning—just like the previous algorithm—but without any prior knowledge.

Finally, an algorithm is developed that uses the same knownness concept to derive a hierarchical exploration strategy. The performance metric that Rmax exploration achieves is one that demands near-optimal behavior in all but polynomial number of timesteps with high probability, but unlike regret minimization techniques (Auer et al.,

2010), it does not insist on improving performance after near-optimality is achieved, nor does it have any guarantee before that. We show that a hierarchical exploration strategy can be used to achieve this *anytime* behavior.

This algorithm achieves anytime behavior by forming a multi-resolution discretization of the state space and variable levels of generalization. These features, which are made possible by using the continuous knownness metric, allow the agent to make very accurate predictions in parts of the state space with a lot of samples (that is, using a fine discretization), while still managing to use data efficiently in other parts of the state space with sparser data (using coarse discretization).

1.2 Overview

The remainder of this dissertation is organized as follows: Chapter 2 is an overview of the reinforcement-learning problem. It provides some background information about reinforcement learning, interesting problems in the field, and also nomenclature necessary for the development of the rest of the dissertation.

Chapter 3 focuses on model-based algorithms and a mechanism for performing exploration called *self-aware exploration*. It first surveys existing work in this class of algorithms, and then introduces the concept of *knownness*. It formalizes this concept using a learning framework called "Continuous Knows What It Knows" or CKWIK, develops an abstract RL algorithm called CKWIK-Rmax that uses this newly developed framework, and analyzes its sample complexity in the PAC-MDP framework.

Chapter 4 introduces two instantiations of the CKWIK-Rmax algorithm for the cases where no information about the dynamics of the environment is available (other than general Lipschitz assumption), and the cases where the user can provide dependency structures between the state variables to the algorithm. A sample complexity analysis of these two algorithms is also provided. This analysis shows that the second algorithm can use prior knowledge to learn significantly faster.

Chapter 5 extends the algorithms in the previous chapter to improve learning in cases where no prior knowledge is received from the user. The algorithm developed in

this chapter uses dimension-reduction techniques from supervised learning to automatically discover relevant features in the state space, and dramatically improve the sample complexity without any prior knowledge about the environment.

Chapter 6 introduces an algorithm that uses the knownness concept to derive a hierarchical exploration mechanism. This algorithm, called MRE, expands the effectiveness of model-based techniques beyond the PAC-MDP framework, by creating behaviors that are more *anytime*.

I conclude the dissertation in Chapter 7. Also, two appendices are provided. The first one provides technical details of the environments used to evaluate the algorithms in the dissertation, and the second one gives some useful mathematical facts that were used in this document.

Chapter 2

Reinforcement-Learning Background

The purpose of this chapter is to familiarize the reader with the concept of *reinforcement learning*, both as a learning paradigm and a subfield of *machine learning*. We will discuss some of the basic results and challenges and introduce the necessary notation that will be used in the rest of this dissertation. However, the content of this chapter is not intended to be a complete RL overview; only materials necessary for the development of the rest of the dissertation are included. Readers who need more thorough information can refer to more general surveys about RL, such as Sutton and Barto (1998), (Littman, 1996) and (Szepesvári, 2010).

2.1 Learning Through Interaction

Reinforcement learning (RL) is the study of learning through interaction between an intelligent agent and an environment. The ultimate goal of the learning process is for the agent to behave optimally in the environment. The interaction between these two entities refers to the agent's ability to both perceive information about the state of the environment, and send back commands to potentially influence the environment state. The optimality of agent's behavior is measured using a reinforcement signal for the task that signifies how good each state is. Figure 2.1 is a pictorial illustration of this interaction between the agent and the environment.

According to this definition, reinforcement learning is a very broad concept. In fact, one can argue that the whole universe and the life we are living is an instance of a reinforcement-learning problem. In this instance, an individual is born into the world, having very little knowledge about its surrounding environment; through various sets of sensory information, it then collects information about the world. Furthermore, the

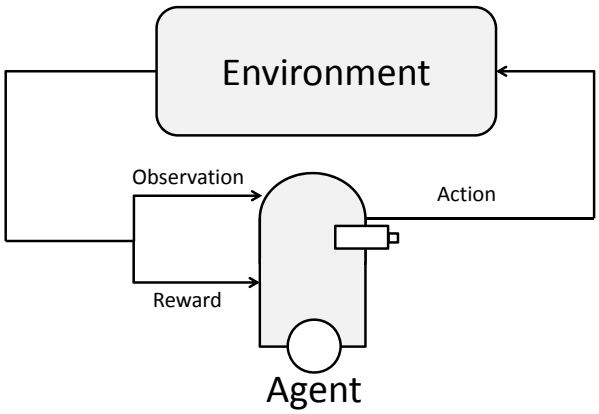


Figure 2.1: Interaction between the agent and the environment in a reinforcement-learning problem.

universe constantly changes due to the collective actions of all the beings in it. Each individual is motivated by various types of intrinsic and/or extrinsic concerns to behave optimally in the world and has to achieve this goal by learning through interaction.

Of course, as scientists we are faced with the challenge of constructing a framework that can capture as broad of a learning paradigm as possible, while keeping it mathematically justifiable and viable. Numerous mathematical formulations of RL have been used in the literature depending on many different factors, such as how accurately the agent perceives the world, what the nature of the states and actions is, how complicated the dynamics of the world are and how the interaction occurs between the two entities. Each model makes some compromises and assumptions so as to make the development of algorithms feasible. Below, we discuss assumptions that have been made in a rich body of the RL literature and are also adopted in this dissertation:

Single agent. Unlike in the general RL setting, where multiple agents can coexist in an environment and interact with each other, the single-agent setting considers the case where only one agent interacts with the environment. The notion of *system control* has also been used to emphasize that the agent tries to control a *system* (environment) in an optimal fashion (Szepesvári, 2010; Ernst et al., 2009).

Synchronous discrete timesteps. In this setting, learning happens during a series of discrete time intervals—we call them *timesteps*. In each timestep t, the agent

first makes an observation about the environment's state, denoted by o_t, along with a scalar reward signal r_t. It then executes an action a_t. At the end, the environment transitions to another state based on the action it receives from the agent and advances the timestep. Some contrary configurations include having a continuous notion of time (Doya, 2000), and the case where the agent's observation of the current state is delayed due to some sort of latency (Walsh et al., 2009).

Full Observability. In this setting, the agent is capable of observing the full state of the environment. When this assumption holds true, we use state (denoted s) and observation (denoted o) interchangeably. Other frameworks include having partial observability of the states, or no observability altogether (Littman, 1996; Kaelbling et al., 1998; Jaakkola et al., 1995).

Markovian Assumption. The behavior of a dynamical system is a function that maps the entire history $h_t = \{s_1, r_1, a_1, ..., s_t, r_t\}$ and a_t to a probability distribution over the next state. The Markovian assumption dictates that s_t is a sufficient statistic of this function, so $\Pr(s|h_t, a_t) = \Pr(s|s_t, a_t)$.

When these assumptions hold true, a mathematical framework called the *Markov Decision Process*—or MDP for short—can be used to model the environment (Puterman, 1994). The following section provides a summary of this framework.

2.2 Markov Decision Processes (MDPs)

A *Markov decision process* is a mathematical model of a dynamical system with the property that the current state is a sufficient statistic of the dynamics. This assumption provides a huge advantage in terms of the learnability of the system because important quantities can be estimated directly, and therefore a much more compact representation of the behavior can be established. An MDP is formally defined using a 5-tuple: $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$. The two variables $\mathcal{S}$ and $\mathcal{A}$ are the sets of possible states of the system and available actions to the agent, respectively. Function T is the transition kernel of the system that defines how the system responds to agents' actions. More formally, it maps a state-action pair to a probability distribution over the next states $T : \mathcal{S} \times \mathcal{A} \to \mathcal{P}_{\mathcal{S}}$.

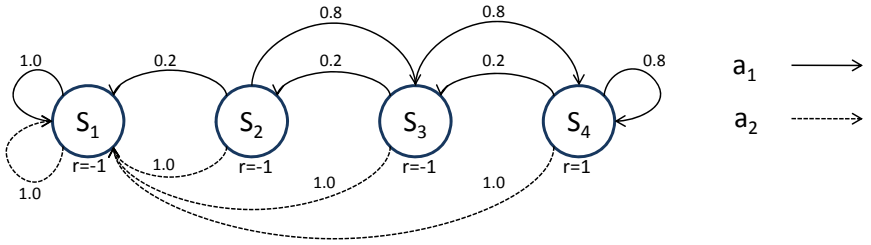


Figure 2.2: Combination lock, a simple stochastic finite MDP. The probability of each transition is specified on its corresponding edge in the graph.

In the degenerate case, when the model forces all the probability distributions to concentrate on only one state, the MDP is termed *deterministic*. Function $R : \mathcal{S} \to \mathbb{R}$ is the reward function that specifies the reinforcement signal in each state, and $0 \leq \gamma < 1$ is a discount factor that modifies how much future rewards are worth to the agent.

Depending on the types of the variables, we can construct several MDP definitions with different complexity levels. In all the models considered here, $\mathcal{A}$ is a discrete nonempty set of actions $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$. In this work, two types of MDPs are mainly addressed, which are formally defined next.

Discrete stochastic MDP. In this model, also called the *finite stochastic MDP* model, the state space is a countable nonempty set of states $\mathcal{S} = \{s_1, \ldots, s_{|\mathcal{S}|}\}$ and we allow for arbitrary stochastic transition functions.

Figure 2.2 shows an example of such an MDP. In this example, there are four states marked by s_1 to s_4 and two actions marked by a_1 (the solid lines) and a_2 (the dashed lines). The reward function is also specified under each state. The transition function of this MDP is graphically displayed using edges of the graph. The number on each edge specifies the probability of that transition. For example, if we perform a_1 in s_3, the system moves to s_4 with 0.8 probability and s_2 with probability 0.2. In more complex systems, however, the transition function is often described intentionally as opposed to being enumerated extensionally for all the state-action pairs.

Continuous MDP. In this model, the state space is a bounded, closed subset of the Euclidean space $\mathcal{S} \subset \mathbb{R}^n$. Although similar to the finite case—the transition function is a mapping from state-action pairs to a probability distribution over the next states—we

need to put some constraints on the possible transition functions to assure learnability. In particular, there are two factors we need to pay attention to: (1) The shape of the probability distributions over the next states, or the *stochasticity assumptions*, and (2) The relationship between the transition functions of two nearby states s_1 and s_2, or the *smoothness assumptions*. These factors are important because they allow for generalization in learning—something essential for learning in continuous spaces. For example, one can show that learning in continuous spaces becomes impractical if no assumption is made about the smoothness and the stochasticity of the environment (refer to Appendix D.1).

Below are some of the commonly-made assumptions about the stochasticity of an environment.

Assumption 1 (Deterministic)**.** *In this class of MDPs, the probability distribution $T(s, a)$ is a Dirac function. In other words, we have $T(s'|s, a) = 1.0$ for some state s' and 0 everywhere else. In this setting, the transition function can also be regarded as a mapping from state-action pairs to next states. Therefore, the transition function can be written as $s_{t+1} = f(s_t, a)$.*

Assumption 2 (Parametric distribution)**.** *In this class of MDPs, the distribution over the next state is a known parametric one. The set of parameters θ that define the probability distribution can vary from one model to another, but usually the mean and the covariance of the distribution is considered. An example of this class is when the transition function is a multivariate normal distribution. It is also customary to assume that the agent learns only the mean of the distribution and the other parameters are known beforehand. Therefore, the transition function is fully described by its mean function $\mu_{s,a}$:*

$$s_{t+1} \sim \mathcal{P}(\mu_{s_t,a_t}, \theta_{s_t,a_t}),$$

where θ_{s_t,a_t} is the set of other parameters for the transition function.

Assumption 3 (White noise)**.** *This class is a particular instance of the previous class*

that allows for limited stochasticity in terms of white noise at the destination state. Instead of defining the transition function in terms of a probability distribution, we usually write the transition function as: $s_{t+1} = f(s_t, a) + \omega_t$. *The second term is a white noise vector with each of its components selected i.i.d. from a distribution with 0 mean.*

Most of the results in this dissertation assume that the environment conforms to Assumption 2. These environments are usually referred to as *general continuous MDPs* in this document, which distinguishes them from other classes that enforce more constrained assumptions.

While stochasticity measures how complex the transition out of one state is, smoothness measures how much the transition is allowed to vary from one starting state to another. We quantify the smoothness of a domain using the *Lipschitz continuity measure*. Depending on the stochasticity assumption, the definition of Lipschitz continuity measure varies. For example, if the environment is deterministic, the smoothness can be specified as:

$$\| T(s_1, a) - T(s_2, a) \|_2^2 \leq C_T \| s_1, s_2 \|_2^2, \quad \forall s_1, s_2 \in \mathcal{S}, a \in \mathcal{A}, \tag{2.1}$$

where $C_T \in \mathbb{R}^+$ is called the Lipschitz constant. We can also define smoothness in stochastic domains. For example, suppose our transition function is in the form of multivariate normal distribution with constant variance: $T(s, a) \sim \mathcal{N}(\mu_{s,a}, \Sigma)$. Then we can write the smoothness as:

$$\| \mu_{s_1,a} - \mu_{s_2,a} \|_2^2 \leq C_T \| s_1, s_2 \|_2^2, \quad \forall s_1, s_2 \in \mathcal{S}, a \in \mathcal{A}, \tag{2.2}$$

which relates the distance between the means of the two probability distributions over the next states to the distance of the two starting states. As a general rule of thumb, we can define smoothness based on how much the parameters of distributions change from one state to another.

Given an MDP, a *policy*, denoted by π, is defined as a function that specifies what

action to take from each state. If the policy is deterministic, it simply outputs an action for each state: $\pi \in \Pi_{det} : \mathcal{S} \rightarrow \mathcal{A}$. On the other hand, stochastic policies produce probability distributions over actions at each state: $\pi \in \Pi_{stoch} : \mathcal{S} \rightarrow \mathcal{P}_{\mathcal{A}}$. We concentrate on deterministic policies in most of this document, and therefore use π to mean a deterministic policy unless specified otherwise. This constraint is not very limiting because although deterministic policies are less complex and easier to work with, existence of an optimal deterministic policy in any MDP is always guaranteed (Puterman, 1994). Following policy π means that for any timestep t, the agent executes $\pi(s_t)$, where s_t is the state observed at time t.

Given a policy π, the expected total discounted return that the agent collects over an infinite horizon starting from an initial state s and following π is referred to as the *value* of that state under π,

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t^{\pi} | s^0 = s\right], \tag{2.3}$$

where R_t^{π} is a random variable that indicates the reward collected at timestep t when following policy π. The goal of learning is to find a policy that achieves the maximum value for all starting states. Such a policy is referred to as the *optimal policy* and is denoted by π^*. Other quantities related to the optimal policy are also indicated by the asterisk. For example, optimal value function $V^* = V^{\pi^*}$ is the value of an optimal policy and optimal action a^* is the action that an optimal policy takes. We also use a slight variation of the value function, denoted by $Q^{\pi}(s, a)$, to indicate the expected total discounted reward when the agent starts at state s, executes action a, and then follows policy π.

A recursive formulation of the value function can also be constructed to relate the value of different states to each other. This recursive equation is known as the *Bellman equation*:

$$\begin{aligned} V^{\pi}(s) &= R(s) + \gamma \textstyle\sum_{s' \in \mathcal{S}} T(s'|s, \pi(s)) V^{\pi}(s') \quad \text{Discrete MDP} \\ V^{\pi}(s) &= R(s) + \gamma \textstyle\int_{\mathcal{S}} T(s'|s, \pi(s)) V^{\pi}(s') ds' \quad \text{Continuous MDP.} \end{aligned} \tag{2.4}$$

It can be shown that the optimal value function V^*, which we sometimes simply call the value function V, can be written as:

$$\begin{aligned} V(s) &= R(s) + \max_a \left(\gamma \textstyle\sum_{s' \in \mathcal{S}} T(s'|s,a) V(s')\right) \quad \text{Discrete MDP} \\ V(s) &= R(s) + \max_a \left(\gamma \textstyle\int_{\mathcal{S}} T(s'|s,a) V(s') ds'\right) \quad \text{Continuous MDP.} \end{aligned} \tag{2.5}$$

Similarly, we can derive the equations for the Q-function (Sutton and Barto, 1998):

$$\begin{aligned} Q(s,a) &= R(s) + \gamma \textstyle\sum_{s' \in \mathcal{S}} T(s'|s,a) V(s') \quad \text{Discrete MDP} \\ Q(s,a) &= R(s) + \gamma \textstyle\int_{\mathcal{S}} T(s'|s,a) V(s') ds' \quad \text{Continuous MDP.} \end{aligned} \tag{2.6}$$

2.3 Planning vs. Learning in MDPs

As mentioned earlier, the goal of solving an MDP is to find an optimal policy. When all the parameters of the MDP are given, the task of finding the optimal policy is called *planning* (Puterman, 1994). But, when only $\mathcal{S}$, $\mathcal{A}$, and the discount factor are specified, and the agent needs to find the optimal policy by interacting with the MDP, the task is called *learning*.

These definitions are of course not unique, and we can find several variations of them in the literature.

In particular, there is not a fine line between planning and learning when dealing with settings that provide more information to the agent than our definition of learning, but less than what we defined for planning. Figure 2.3 shows some of these settings in a spectrum between learning and planning. The most complete information is available when the solver has access to the full MDP tuple (Puterman, 1994). A *generative model* provides access to the transition function only through sampling, so instead of providing the full description of the probability distribution, it allows the solver to query the transition function for any state-action pair (Kearns et al., 1999; Kocsis and Szepesvári, 2006). *Trajectories with reset* is a setting where the notion of current state and trajectory come into play, just like the learning problem; the difference between the two settings is that the solver in the former can push a *reset button* at anytime during

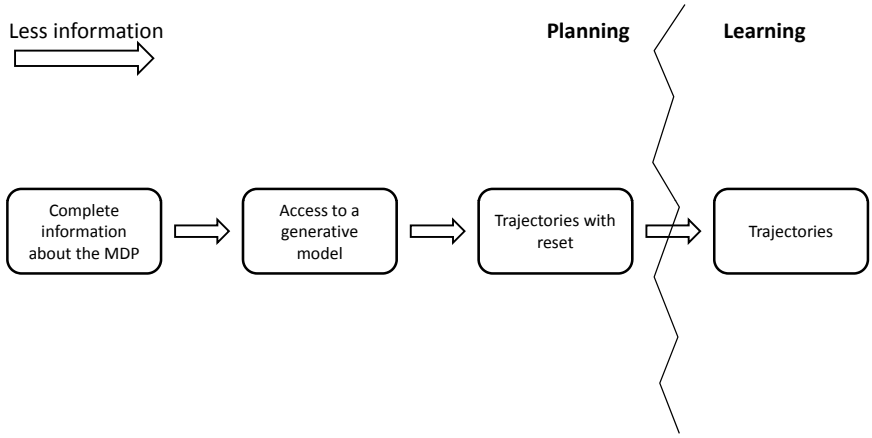


Figure 2.3: Problem configurations stretching between planning and learning.

a trajectory to go back to the first state (Fiechter, 1997; Kakade and Langford, 2002).

We focus most of our attention on the learning problem in this dissertation. However, Appendix B provides a quick overview of some of the existing algorithms for planning in MDPs.

2.4 Challenges

All machine-learning techniques are susceptible to having trouble in large-scale domains. This problem is mostly due to the fact that they fail to use information efficiently. Humans, as an example, are very good at learning complicated tasks with little information, because they are able to generalize experience from one situation to others very efficiently. Machine-learning methods on the other hand, have not been as successful in making good use of available knowledge. They either do not exploit available data to reason about unvisited situations, or they have so much data to process that learning becomes intractable in large environments.

RL algorithms are even more sensitive to the size of the problem because they contend with unique challenges that algorithms in other paradigms, such as supervised learning, do not need to face. Below, some of these challenges are summarized.

Delayed partial feedback. We can think of the Q-function as the utility function of actions because it reveals what the best action is in each state. Unfortunately, RL

agents do not immediately receive the utility associated with their actions from the environment. That is, the reinforcement signal of a particular state does not reveal the correct action to be taken from that state. In fact, the agent never receives the correct actions from a supervisor, and the only information it receives from the environment is some partial data that can be used to construct the utility of the action it chooses. Therefore, the algorithm has to infer the optimal behavior by putting together the partial information collected over time. This indirect information is completely different from supervised learning, where correct labels are provided for some training data.

Complex data. In many machine-learning paradigms, including those used in the majority of the supervised-learning literature, the learner does not have any control over either the training, or the testing data. The data is usually selected from some fixed distribution and the learner has to focus all its attention on how to predict labels for the testing data. Learning becomes more complex when the learner has the ability to choose the input data for training. The reason for this complication is two-fold: First, apart from having to predict labels, the learner has to figure out a way to choose good training data. Second, since the learner affects the distribution of the training data, the hypothesis it learns could be biased toward that particular choice, which might lead to sub-optimality in the testing data.

RL agents directly affect the data they observe by choosing different actions. Moreover, since there is no notion of training and testing phases in RL, the agent faces a unique dual control problem that we study next. In a sense, reinforcement learning has a flavor of both *active learning* (Freund et al., 1995; Settles, 2010), in which the agent selects the training data itself, and *online learning* (Littlestone, 1988; Vovk et al., 2005), in which learning examples are presented in a sequence, one data at a time.

Exploration vs. exploitation. As mentioned before, RL agents control their input data in a unique way by deciding what action to take at each timestep. The goal of the agent is to take actions that accrue maximum total reward. A second goal of the agent, which is internal to its structure, is to select actions that result in a good distribution of data for learning. In other words, at each timestep the agent has

to decide whether to select an action that results in exploring of the environment to gain better information, or to select an action that seems to collect maximum utility according to the current knowledge. This dual control problem is referred to as the exploration vs. exploitation dilemma in reinforcement learning, and is an important component of any successful RL algorithm. An algorithm that always seeks to explore the environment suffers because it is totally blind to the utility of its actions. On the other hand, an algorithm that always tries to maximize its performance might never learn the optimal behavior because it does not have thorough knowledge about the environment.

Although neither of these challenges is unique to reinforcement learning, their combination creates a unique framework that is both very general and also complex. Other challenges that arise in machine learning, such as generalization and overfitting, also apply in reinforcement learning.

2.5 Efficiency of RL Algorithms

Performance of RL algorithms is usually measured using two completely different metrics: *Sample complexity* measures how much interaction with the environment is needed for the agent to learn, while *computational complexity* measures the amount of computation the agent needs per timestep. Both of these metrics are important for the success of any algorithm because a computationally fast algorithm that learns very slowly wastes a lot of valuable experience, and a fast learning algorithm that needs a lot of computation might not be practical to implement or may not react fast enough in real-time. Nevertheless, most of the attention of this dissertation is focused on the study of sample complexity of algorithms.

During the history of the RL field, different criteria have been used for evaluating the sample complexity of algorithms. Here, we summarize some of these criteria.

Asymptotic convergence. This type of criteria states that the algorithm always finds and converges to the optimal policy if it is given enough time. More formally, let V_t be the set of value functions at timestep t. The asymptotic convergence states that

$V_t \to V^*$ as $t \to \infty$. This metric provides some basic information about the stability of the algorithm and was the method of choice for analyzing RL algorithms in the 1990s (Sutton and Barto, 1998). While it is a useful theoretical analysis tool, it barely provides any practical information about the behavior of an algorithm, as we cannot execute an algorithm for an infinite number of steps. Also, the metric does not provide any information about the actual followed policy by the agent. The next two metrics provide information about the goodness of learning when the algorithm has experienced a finite number of samples.

Regret minimization. This metric is borrowed from the field of optimization in k-armed bandits (Auer et al., 2002). It compares the difference between the reward obtained by the algorithm to that of the optimal one over time. Arguably, this metric is ideal because it exactly measures the quantity that the learner tries to minimize. More technically, the *regret* of an algorithm at any time t is measured by:

$$Reg(t) = \mathbb{E}\left[\sum_{i=1}^{\mathrm{T}}(r(i) - r^*(i))\right], \tag{2.7}$$

where $r(i)$ and $r^*(i)$ are two random variables denoting the reward the agent and the optimal policy receive at time i, respectively. Unfortunately, while the formulation of this metric is very appealing, it is very hard to analyze if no further assumption is made. Therefor, algorithms often make very strict assumptions about the dynamics of the MDP to be able to derive their bounds (Auer and Ortner, 2007; Auer et al., 2010). At the time of this dissertation, no regret-minimization analysis has been done for RL in continuous spaces to the best of my knowledge.

PAC-MDP learning. Probably approximately correct (PAC) learning has been used for analysis of distribution-free supervised-learning model for a long time (Valiant, 1984). However, the PAC-MDP metric was first introduced by Fiechter (1994) for reinforcement learning, and then refined by Kearns and Singh (1998), Kakade (2003) and Strehl (2007). This metric bounds, with high probability, the total number of timesteps that the algorithm does not follow a near-optimal policy. The reason we try to settle on near-optimality with high probability is that we cannot expect to fully learn

a stochastic system using a finite number of samples with full confidence. Therefore, the algorithm takes in a near-optimality parameter $\epsilon > 0$ and a failure probability $\delta > 0$ beforehand and achieves ϵ-optimal behavior with a probability more than $(1 - \delta)$. There are subtle differences between the two available definitions for PAC-MDP. We shall mostly follow the second definition used by Strehl (2007) because it is a simpler definition that avoids mixing times in MDPs. The following definition formally defines the sample complexity of an algorithm according to the following:

Definition 4. *Let $h_t = (s_1, r_1, a_1, ..., s_t, r_t)$ be a path generated by an RL algorithm up to time t. Denote the algorithm's policy at time t by χ_t. For any fixed $\epsilon > 0$, the sample complexity of the algorithm is the total number of timesteps in which the algorithm's policy at that time is not ϵ-optimal: $V^{\chi_t}(s_t) \leq V^*(s_t) - \epsilon$.*

According to this definition, the sample complexity of algorithms is measured by the number of times they make mistakes (that is, does not follow a near-optimal policy). In PAC-MDP analysis, we bound the number of times the algorithm makes such mistakes. Note that we do not state when those mistakes occur in the execution path and only put a cap on their total number.

2.6 Reinforcement-Learning Algorithms

The ultimate goal of an RL algorithm is to behave (near) optimally. To achieve this goal, several approaches have been widely used in the literature. These algorithms are mainly distinguished from each other by the types of quantities they maintain in their internal structures and how they update them to generate the final policy. Here, we briefly mention some of these techniques.

2.6.1 Policy-Search Methods

These algorithms are considered to be the most direct methods in reinforcement learning because they explicitly search the space of policies (Baxter and Bartlett, 2001; Sutton et al., 2000). Many policy-search algorithms have been studied in the literature, most of which rely on parametrization of the policy that allows for gradient-ascent searches to be

applied. The general architecture of such algorithms consists of maintaining a specific policy as the current policy, developing a mechanism to evaluate the performance of the current policy, and finally a mechanism to derive another policy from the current one that has a better performance.

2.6.2 Value-based Methods

Value-based methods take a more indirect approach to solving the reinforcement-learning problem by maintaining information about the value of states, either in the form of Equation 2.5 or 2.6. Since the Bellman equation defines the value of a state recursively based on the value of other states, to which the agent does not have access, it has to use some sort of *bootstrapping* in learning. A family of algorithms called "temporal difference learning methods" do exactly that by using their own predictions as targets during the course of learning (Sutton, 1988, 1996). Below, we briefly describe one of the algorithms in this family called *Q-learning* (Watkins, 1989).

Q-learning maintains an estimate of $Q^*(s, a)$ and updates its estimates based on transition tuples $(s_t, a_t, r_{t+1}, s_{t+1})$. The algorithm performs a stochastic approximation of the Bellman equation by moving its estimate $Q_t(s_t, a_t)$ toward the value of s_{t+1}:

$$\begin{aligned} \delta_{t+1} &= r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \\ Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \alpha_t \delta_{t+1}. \end{aligned} \tag{2.8}$$

The first equation defines δ_{t+1}, which is the difference between the estimated values of two consecutive states s_t and s_{t+1}—hence the name of *temporal difference*. The algorithm requires a sequence of small nonnegative numbers $[\alpha^t]_{t \in \mathbb{N}}$, called *learning rate*, to operate. Implementation of this algorithm is very easy when dealing with finite state spaces because a lookup table can be used to store the Q-function. However, it becomes more complicated when using some sort of generalization (Boyan and Moore, 1995).

2.6.3 Model-based Methods

Model-based algorithms take an even more indirect approach to solving the RL problem. These methods learn the value of states by first directly estimating the parameters of the unknown MDP (the transition function, the reward function or both), and then using a conventional planning algorithm to solve the estimated MDP (Kumar and Varaiya, 1986; Sutton, 1990). One of the philosophies behind these algorithms is that learning the unknown parameters of the MDP, which is very close to regular supervised learning, is much easier than learning the indirect and complex value function. Once the parameters are estimated, the planning step to derive the value function is again an easier task than learning the value function in an online fashion using data. Therefore, these algorithms break the complicated task of online learning the value function into two simpler subtasks.

There are two important caveats in this approach that need to be carefully addressed. First, learning the parameters of the MDP is not exact, therefore the constructed internal MDP will always be different from the true one. For the model-based methods to be successful, an analysis needs to be done on the relationship between the model estimation error and the sub-optimality of the final policy. Second, it is not obvious how the decision making should be done before the parameters are fully learned.

Several people have shown the relationship between the similarity of two MDPs and their corresponding policies both for finite and continuous MDPs, which takes care of the first caveat. The general result is that if the dynamics of two MDPs are close to each other, so are their optimal policies (Kearns and Singh, 1998; Kakade et al., 2003).

The second problem is dealt with using the exploration-exploitation strategy of the algorithm. A standard technique for doing so is to continuously use the partial information to build new models and replan as new information comes in. Meanwhile, an exploration scheme has to be devised to make sure that the internal model gets closer to the true MDP as more data comes in. A schematic illustration of such an algorithm is depicted in Figure 2.4.

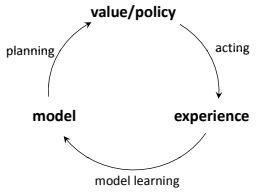


Figure 2.4: Illustration of the structure of a typical model-based algorithm (Sutton and Barto, 1998).

The next chapter provides detailed information about a class of model-based algorithms that use *self-aware* exploration to solve the exploration-exploitation dilemma.

Chapter 3

Self-aware Exploration

This chapter studies the concept of self-aware exploration in model-based algorithms and how it helps with the efficient discovery of information in unknown environments. These algorithms are called "self-aware" because at any time during learning, in addition to being able to estimate the parameters of the environment, they are able to identify whether their estimates are accurate or not. So, in a sense, they are aware of how much they know about each part of the state space. This extra information allows them to focus their attention on those parts where their estimates are not as good.

The main idea behind this type of exploration is the so-called "optimism in the face of uncertainty" maxim (Thrun, 1992). According to this concept, learning agents need to be optimistic in the quantities they are estimating. In reinforcement learning, this behavior can be realized by acting according to an overly optimistic value function with respect to the observations so far. Model-based algorithms are studied here because they have been particularly successful in incorporating this idea.

In the rest of this chapter, the implementation of optimism in the face of uncertainty is explored in model-based methods in two sections: First, an overview of the existing literature is presented in Section 3.1. Then, in Section 3.2, a new concept called "knownness" is developed as an extension of the commonly-used *known*/*unknown* concepts.

3.1 Known vs. Unknown

To study the concept of *known states*, we consider the Rmax algorithm (Brafman and Tennenholtz, 2002), which was one of the first techniques that exploited the idea of

self-aware exploration in finite MDPs. Although the focus of this document is on continuous spaces, Rmax is selected as an introductory example because it is simple to study, yet provides insight into how self-aware exploration can be used in model-based RL algorithms in general. In what follows, Rmax is explained in the setting where only the transition function is unknown. Learning algorithms for other scenarios are constructed in a similar fashion and have been studied elsewhere, (Brafman and Tennenholtz, 2002; Strehl, 2007).

The Rmax agent (described in Algorithm 1) maintains an internal model of the environment that—given the data—has the most likelihood. Since the transition function of each state-action pair forms a multinomial distribution over the next states, the maximum-likelihood (ML) estimate can be constructed using a table of counts: $c(s';s,a)$ keeps the number of times we ended up in s' after taking a from s, and $c(s,a)$ keeps track of how many times action a was executed in s. The maximum-likelihood estimate is computed using the following equation:

$$\hat{T}(s'|s,a) = \frac{c(s';s,a)}{c(s,a)}. \tag{3.1}$$

The agent incorporates exploration into its internal model by using only those estimates with guaranteed high accuracy, and replacing the rest with optimistic initialization. More technically, the algorithm distinguishes between two types of state-action pairs when estimating the model: *knowns* and *unknowns* [1]. The agent starts out by considering all the state-action pairs unknown. As it collects data during learning, it marks a pair as known whenever the total variation between the estimated transition function for that pair and the true one becomes less than ϵ_T with probability at least $(1-\delta_T)$. It can be shown that for the values of $c(s,a) > C$, where $C = \frac{2(\ln(2^{|S|}-2)-\ln(\delta_T))}{\epsilon_T^2}$, the ML estimate satisfies this condition (Strehl et al., 2009). The value of the two constants ϵ_T and δ_T are computed based on user inputs before the learning starts.

During the construction of the internal model, the agent replaces its estimates for

[1] Sometimes, these notions are used with *states*. There, a known state s means that (s,a) is known for all a's.

unknown pairs by a bogus transition function that causes their Q-functions to become $V_{\max}$—the maximum achievable value. With this technique, the agent is attracted to unknown regions because of their high values.

Algorithm 1 Rmax, a model-based algorithm for solving finite MDPs.

Input: Threshold constant C.
Initialize counter $c(s,a)$ to 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.
Initialize internal MDP $\hat{M} = \langle \mathcal{S}, \mathcal{A}, \hat{T}, \hat{R}, \gamma \rangle$:

$$\hat{T}(s'|s,a) = \mathbb{I}(s' = s), \quad \hat{R}(s) = R_{\max}.$$

Compute $\pi^*_{\hat{M}}$ as the optimal policy of $\hat{M}$.
for all timesteps $t = 1, 2, \cdots$ **do**
  Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
  Increment counter: $c(s_t, a_t) \leftarrow c(s_t, a_t) + 1$.
  if $c(s_t, a_t) = C$ **then**
    Update $\hat{T}(s_t, a_t)$ and $\hat{R}(s_t)$ using ML estimate of the data (eqn. 3.1).
    Update $\hat{M}$ and subsequently $\pi^*_{\hat{M}}$.
  end if
end for

There are two key factors that guarantee the success of Rmax theoretically. First, it can be shown that if the parameters of two MDPs (that is, their transition and reward functions) are close to each other, their value functions are also close. Moreover, executing the optimal policy of one of the MDPs in the other one will not incur a huge loss. This result can be used to show that as long as the agent remains inside the known parts of the state space, its performance will be close to optimal with high probability. Second, based on the pigeon-hole principle, the number of times the agent can be in an unknown state-action pair is bounded. Therefore, we can bound the number of times the algorithm makes mistakes by not following a near-optimal policy.

While the table-based estimation of the model in Rmax limits its applicability to finite MDPs with no function approximator to carry information from one state to another, it turns out that the principle of known and unknown states can be used in a variety of different settings. The KWIK framework, which will be briefly studied next, captures this insight.

3.1.1 KWIK and Rmax

Several independent analyses have shown that Rmax is PAC-MDP (Strehl et al., 2009; Brafman and Tennenholtz, 2002). But, perhaps the most comprehensive analysis is due to Li (2009), which provided a relationship between the efficiency of learning the transition and reward functions in a learning framework called "knows what it knows" (Li et al., 2008), or KWIK for short, to that of a general model-based algorithm called KWIK-Rmax. This algorithm, which is described in Algorithm 2, maintains one KWIK learner for each unknown parameter of the MDP. At each timestep, it rebuilds its internal representation of the world using its KWIK learners, forming known and unknown state-action pairs depending on whether the KWIK learners return an estimate or say "I don't know" respectively. Similarly to Rmax, it then computes the optimal value function of its internal model and follows the greedy policy. The details of the KWIK framework will be discussed shortly.

Algorithm 2 KWIK-Rmax, a general model-based algorithm.

Input: KWIK learner $\mathcal{F}_T$, accuracy parameter ϵ_T, and confidence parameter δ_T.
Initialize transition function approximator $\mathcal{F}_T$ with ϵ_T and δ_T.
Initialize the internal MDP $\hat{M}$ and its optimal policy $\pi^*_{\hat{M}}$.
for all timesteps $t = 1, 2, \cdots$ **do**
  Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
  Update $\mathcal{F}_T$ using (s_t, a_t, s_{t+1}).
  Update internal MDP $\hat{M} = \langle \mathcal{S}, \mathcal{A}, \hat{T}, R', \gamma \rangle$, where $\hat{T}$ and R' are defined as:
  if $\mathcal{F}_T(s, a)$ returns $\sqcap$ **then**
    $\hat{T}(s'|s, a) = \mathbb{I}(s' = s)$ and $R'(s) = R_{\max}$.
  else
    $\hat{T}(s'|s, a)$ is the output of $\mathcal{F}_T$ and $R'(s) = R(s)$.
  end if
end for

In this dissertation, the definition of KWIK online learning is slightly abused to be specific to learning the transition function. Otherwise, KWIK can be used as a general online-learning framework. With this in mind, let us first formalize the notion of closeness for transition functions:

Let the *total variation* between two continuous probability distributions $\mathcal{P}_1$ and $\mathcal{P}_2$

be:

$$\mathrm{d_{var}}(\mathcal{P}_1, \mathcal{P}_2) \stackrel{\text{def}}{=} \int_x |\mathrm{Pr}_1(x) - \mathrm{Pr}_2(x)| dx. \tag{3.2}$$

The closeness of two transition functions is defined as follows:

Definition 5. *Let T_1 and T_2 be two transition functions defined over the same state/action spaces. We say $T_1(s,a)$ is ϵ-close to $T_2(s,a)$ if the total variation between $T_1(s,a)$ and $T_2(s,a)$ is less than ϵ.*

Definition 6. *Let $\hat{T}$ be an estimate of a transition function T. We say $\hat{T}(s,a)$ is (ϵ,δ)-close to $T(s,a)$ if $\hat{T}(s,a)$ is ϵ-close to $T(s,a)$ with probability at least $(1-\delta)$. Similarly, $\hat{T}(s,.)$ is (ϵ,δ)-close to $T(s,.)$ when $\hat{T}(s,a)$ is $(\epsilon,\delta/|\mathcal{A}|)$-close for all $a \in \mathcal{A}$.*

Definition 7 (KWIK online learning)**.** *An algorithm online-learns a transition function in the KWIK framework using the following protocol: The learner is first given an accuracy parameter $\epsilon > 0$ and an allowed probability of failure $0 < \delta < 1$ (we are mostly interested in the cases where ϵ and δ are close to 0). At each timestep $t = 1, 2, \ldots$, the learner is presented with (s_t, a_t) and is asked for an estimate of $T(s_t, a_t)$. The learner can either present its estimate $\hat{T}(s_t, a_t)$, in which case it has to be (ϵ,δ)-close, or it can say its estimate is not (ϵ,δ)-close (that is, it says I don't know or the special symbol $\sqcap$). In the latter case, the algorithm is given an i.i.d. sample from $T(s,a)$ as a training example. An algorithm has a KWIK bound of $B(\epsilon,\delta,\mathcal{S},\mathcal{A})$ if the number of times it outputs $\sqcap$ is bounded by the function B for any adversarially selected sequence.*

Observation 8. *The ML estimator for the transition function in* Rmax *has a KWIK bound of $C|\mathcal{S}||\mathcal{A}|$. The reason is that for any (s_t, a_t), the algorithm returns $\sqcap$ if $c(s_t, a_t) < C$. Otherwise, it returns the ML estimate, which is in fact (ϵ,δ)-close to the true function.*

In fact, Rmax forms known and unknown state-action pairs based on whether its transition function estimator returns a value or $\sqcap$. Theorem 21 of Li (2009), which we summarize here in Theorem 9, generalizes this connection and provides a relationship between the PAC-MDP learnability of KWIK-Rmax and the KWIK bound of its transition (and reward) function learner.

Theorem 9. *If the transition function of a class of MDPs can be KWIK-learned with a bound B that is polynomial w.r.t. its arguments $(\epsilon, \delta, \mathcal{S}, \mathcal{A})$, then* KWIK-Rmax *is PAC-MDP. In particular, if the following parameter values are used:*

$$\epsilon_T = \Theta\left(\epsilon(1-\gamma)^2\right), \qquad \delta_T = \Theta(\delta),$$

then the sample complexity of KWIK-Rmax *is*[2]*:*

$$\mathcal{O}\left(\frac{V_{\max}}{\epsilon(1-\gamma)}\left(B(\epsilon(1-\gamma)/V_{\max}, \delta, \mathcal{S}, \mathcal{A}) + \ln\frac{1}{\delta}\right)\ln\frac{1}{\epsilon(1-\gamma)}\right).$$

Proof. Readers are referred to (Li, 2009; Theorem 21) for the proof of this theorem. □

According to this theorem, the sample complexity of KWIK-Rmax is dependent on $\mathcal{S}$ and $\mathcal{A}$ only through a linear dependence on B—the KWIK sample complexity of $\mathcal{F}_T$. Therefore, it is highly desirable to study different classes of MDPs that allow for faster KWIK learners. Below, some of these classes are briefly described.

3.1.2 More Examples

The relationship between KWIK-learnability of MDPs and the sample complexity of model-based learning has been investigated for several classes of MDPs in the past few years. Some of these classes are described below.

Factored-Rmax (Guestrin et al., 2002). This algorithm is a variant of Rmax for environments in which the state space is finite but is represented in a factored form (Kearns and Koller, 1999): $s = \{s(1), \ldots, s(k)\}$. Furthermore, the value of each component of the next state is dependent only on a subset of the current state's components. That is, we have:

$$T(s'|s,a) = \prod_{i=1}^{k} T_i(s'(i)|s(\mathrm{dep}_a(i)), a), \tag{3.3}$$

where $\mathrm{dep}_a(i)$ is the list of components that are sufficient to estimate $s'(i)$ when action a is applied, and $s(\mathrm{dep}_a(i))$ is a vector constructed from s by selecting those components

[2]The function Θ is the standard growth rate function.

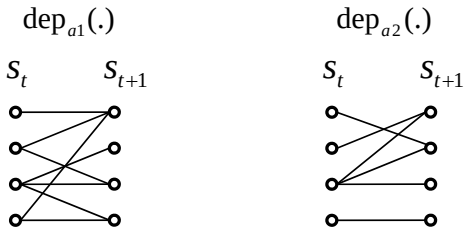


Figure 3.1: The DBN structure of an MDP whose transition function is described in Eqn. 3.4

that are also in $\text{dep}_a(i)$. If the same ML estimator as Rmax is used, a sample complexity that scales exponentially in the factor size k is obtained because k can be much smaller than $|\mathcal{S}|$ or even $\log(\mathcal{S})$. Factored-Rmax uses the dependency structure of the individual components to (exponentially) reduce the sample complexity, while remaining in the KWIK framework.

Factored-Rmax uses a set of ML estimators $\mathcal{F}_a^i(.)$, each estimating the value of the i-th component of the next state when action a is applied. The algorithm takes as input the dependency sets $\text{dep}_a(i)$ in the form of *dynamic Bayesian networks (DBNs)*. The input to $\mathcal{F}_a^i$ is $s(\text{dep}_a(i))$. Each time a transition (s, a, s') is observed, the algorithm updates $\mathcal{F}_a^i$ for all the i's. When a query for $T(s, a)$ is received, the algorithm queries $\mathcal{F}_a^i$ for all i's, and concatenates all the results. If any individual $\mathcal{F}_a^i$ returns $\sqcap$, the algorithm returns $\sqcap$. Figure 3.1 shows an example of such dependency structures for an MDP with two actions $\{a_1, a_2\}$ and the following transition function structure:

$$T(., a_1) : \begin{cases} s'(1) = f_1(s(1), s(2), s(4), a_1) \\ s'(2) = f_2(s(3), a_1) \\ s'(3) = f_3(s(2), s(3), a_1) \\ s'(4) = f_4(s(3), s(4), a_1) \end{cases} \qquad T(., a_2) : \begin{cases} s'(1) = f_5(s(2), s(3), a_2) \\ s'(2) = f_6(s(1), s(3), a_2) \\ s'(3) = f_7(s(3), a_2) \\ s'(4) = f_8(s(4), a_2). \end{cases} \tag{3.4}$$

It can be shown that if $\hat{T}_i$ is $(\epsilon/k, \delta/k)$-close to T_i for all i's, then $\hat{T}$ is (ϵ, δ)-close to the true transition function (Strehl, 2007). A maximum-likelihood estimator is able to

learn a $(\epsilon/k, \delta/k)$-close estimation of T_i with a sample complexity that is exponential in $|\text{dep}_a(i)|$. Assuming that the size of $\text{dep}_a(i)$ is bounded by k, the KWIK sample complexity of this learning scheme can be exponentially smaller than the one we studied in Rmax provided that k is sufficiently small. Consequently, the exploration sample complexity of factored-Rmax, which according to Theorem 9 is linearly dependent on the KWIK bound, is exponentially smaller than the sample complexity of Rmax.

The relationship between KWIK-learnability of the transition function and the sample complexity of model-based learning carries over to the continuous spaces. For example, as we shall see later, the PAC-MDP sample complexity of learning in Lipschitz-continuous MDPs has a lower bound that scales exponentially in the dimensionality of the state space. However, by considering a subclass of continuous MDPs that can be KWIK learned faster, we can construct RL algorithms that scale polynomially as the dimensionality increases. One such subclass is described below.

Linear Transition Function. Let us consider continuous MDPs whose transition functions can be captured by a noisy linear function. In other words, the dynamics of the world is in the form of:

$$s_{t+1} = \mathbf{A}\Phi_{a_t}(s_t) + \omega_t, \tag{3.5}$$

where $\mathbf{A}$ is a $(|S| \times n)$ matrix and $\Phi_a(.) : \mathbb{R}^{|S|} \to \mathbb{R}^n$ is a (basis or kernel) function associated with action a and satisfying $\| \Phi_a(.) \| \leq 1$. Finally, ω_t is a white noise with each of its components selected *independent and identically distributed* (i.i.d.) with mean 0 and a known variance of σ^2.

Strehl and Littman (2008a) introduced an algorithm that is able to learn a noisy linear function $f : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}$ in the KWIK framework (Algorithm 3). They showed that the number of times the algorithm outputs $\sqcap$ is bounded by $\tilde{\mathcal{O}}(|\mathcal{S}|^3/\epsilon^4)$.

An Rmax-type algorithm can be constructed based on KWIK-LR in the following way: The algorithm needs to maintain $|\mathcal{A}| \times |\mathcal{S}|$ instances of the regressor —denoted by $\mathcal{F}_a^i$. Upon observing a transition (s, a, s'), the algorithm updates $\mathcal{F}_a^i$ for $i = 1 \ldots |\mathcal{S}|$ with $(s, s'(i))$ as the training example. To answer a query (s, a), the algorithm queries $\mathcal{F}_a^i$ for all $i = 1 \ldots |\mathcal{S}|$ and outputs an answer by concatenating all of them into a vector. If

Algorithm 3 KWIK-LR, a linear regression algorithm in the KWIK framework.

Inputs: Accuracy parameter ϵ, confidence parameter δ.
Initialize $\mathbf{X} = [\,]$ and $y = [\,]$.
for $t = 1, 2, 3, \ldots$ **do**
  Let x_t denote the input at time t.
  Compute $\bar{q}$ and $\bar{v}$ as follows:

$$\begin{aligned} \mathbf{X}^{\mathrm{T}} X &= U \Lambda U^{\mathrm{T}} \\ \bar{q} &\stackrel{\text{def}}{=} \mathbf{X}\bar{\mathbf{U}}\bar{\mathbf{\Lambda}}^{-\mathbf{1}}\bar{\mathbf{U}}^{\mathrm{T}} x_t \\ \bar{v} &\stackrel{\text{def}}{=} [0, \ldots, 0, v_{k+1}^{\mathrm{T}} x_t, \ldots, v_n^{\mathrm{T}} x_t]^{\mathrm{T}} \end{aligned}$$

  if $\| \bar{q} \| \leq \epsilon$ and $\| \bar{v} \| \leq \delta$ **then**
    Choose $\hat{\theta}$ that minimizes $\Sigma_i [y(i) - \bar{\theta}^{\mathrm{T}} \mathbf{X}(i)]^2$ subject to $\| \bar{\theta} \| \leq 1$, where $\mathbf{X}(i)$ is the transpose of the i-th row of $\mathbf{X}$ and $y(i)$ is the i-th component of y.
    Output valid prediction $x^{\mathrm{T}} \hat{\theta}$.
  else
    Output $\sqcap$.
    Receive output y_t.
    Append x_t^{T} as a new row to the matrix $\mathbf{X}$.
    Append y_t as a new element to the vector y.
  end if
end for

any of the regressors return $\sqcap$, the algorithm marks that state-action pair as unknown.

Again, we can use the result of Theorem 9 that relates the sample complexity of exploration to the KWIK sample complexity of learning the transition function. In this case, the sample complexity of exploration depends polynomially on k (it is $\tilde{\mathcal{O}}(k^3)$ to be exact)[3] instead of exponentially (Strehl and Littman, 2008a).

Constructing model-based algorithms using the concept of known states has also been studied in the scenarios where theoretically sound KWIK-learners have not yet been found for the underlying transition function or a particular suitable function approximator in mind is not compatible with the KWIK framework. In these situations, the known states are computed based on some sort of *heuristic function* in the learner. For example, the next algorithm learns the general continuous-space MDPs with smooth stochastic transition functions using *kernel regression*.

Fitted-Rmax (Jong and Stone, 2007). This algorithm is a counterpart of Rmax

[3]The notation $\tilde{\mathcal{O}}$ is the same as $\mathcal{O}$ but ignoring the logarithmic factors.

for continuous spaces and has the same structure as KWIK-Rmax. It uses *instance-based function approximation* (Russell and Norvig, 1994) for estimating the transition function.

To better study this algorithm, let us first take a closer look at kernel regression. A kernel regressor $\mathcal{F}$ is a local function approximator that receives training data in the form of $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, $x_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}^k$, and outputs $\hat{y}$ as:

$$\hat{y} \stackrel{\text{def}}{=} \mathcal{F}(x; D) = \frac{\sum_{x_i \in D} k(x, x_i) y_i}{\sum_{x_i \in D} k(x, x_i)}, \tag{3.6}$$

where $k(.,.)$ is called the "kernel function" and is a metric that measures the similarity between the two arguments. Equation 3.6 uses an average of all the training instances, weighted by their similarities to the query point according to the kernel metric. A detailed overview of kernel functions can be found in Appendix D.2, but one of the functions called *Gaussian kernel* is reiterated here:

$$k_g(x_1, x_2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\|x_1 - x_2\|_2^2}{\sigma^2}}. \tag{3.7}$$

This function has the shape of a bell and is peaked when $x_1 = x_2$ and diminishes to 0 as the distance between the two arguments goes to infinity. The parameter σ controls how fast the function degrades to 0 (refer to Appendix D.2 for more information).

Fitted-Rmax maintains one instance of kernel regression for each action, $\mathcal{F}_a(.)$, to estimate the transition function. Updating $\mathcal{F}_a$ is done naturally using D_a, which are the samples in D with their actions equal to a. Unlike Rmax, this algorithm cannot use counts for each state to distinguish between knowns and unknowns because there is an infinite number of states in a continuous space and we may never visit any state twice. Therefore, the algorithm needs to make some kind of generalization to use information from nearby states to compute the known regions. Jong and Stone (2007) suggested that a heuristic based on the sum of the kernel values of the training data and the query point be used to decide whether a query point is known or not. In particular, they used

the following formula:

$$\hat{T}(s,a): \begin{cases} \text{known} & , \text{ if } \tilde{n}(s,a) \stackrel{\text{def}}{=} \sum_{s_i \in D_a} \exp\left(-\frac{\|s-s_i\|_2^2}{\sigma^2}\right) > C \\ \text{unknown} & , \text{ otherwise,} \end{cases} \tag{3.8}$$

where C is a threshold parameter. Similar to the counter in Rmax, $\tilde{n}(s,a)$ counts the samples that exist in the training set for (s,a). But, it also partially counts the training samples that are close to the query points, weighted by their un-normalized kernel values.

The generalization used in the computation of $\tilde{n}$ can be taken even further if the user has prior knowledge about the relationship between different actions. That is, if $T(s,a_1)$ is related to $T(s,a_2)$, we can compute $\tilde{n}$ as follow:

$$\tilde{n}(s,a) \stackrel{\text{def}}{=} \sum_{s_i \in D} \delta_{aa_i} \exp\left(-\frac{\| s - s_i \|_2^2}{\sigma^2}\right), \tag{3.9}$$

where $0 \leq \delta_{a_1 a_2} \leq 1$ is a function that provides the similarity of actions a_1 and a_2. This algorithm does not have the PAC-MDP guarantee of the previous methods we discussed, but can tackle a broader class of environments and has been shown to be very successful in practice (Jong and Stone, 2007).

It is easy to change the heuristic function for computing the known states. For example, the *ball-heuristic* is another heuristic we can use in fitted-Rmax. This method partitions the state space into knowns and unknowns based on whether enough training points exist in an ϵ-ball around the query point—hence the name. More precisely, let $\mathcal{N}_\epsilon(s,a)$ be the set of points in D_a that are ϵ-close to s:

$$\mathcal{N}_\epsilon(s,a) = \{s' \in D_a | \quad \| s' - s \|_2^2 \leq \epsilon\}. \tag{3.10}$$

The ball-heuristic identifies a state-action pair as known iff $|\mathcal{N}_\epsilon(s,a)| > C$ for predefined values of ϵ and C. Figure 3.2 depicts the ball-heuristic schematically in 2D when the value of C is 5.

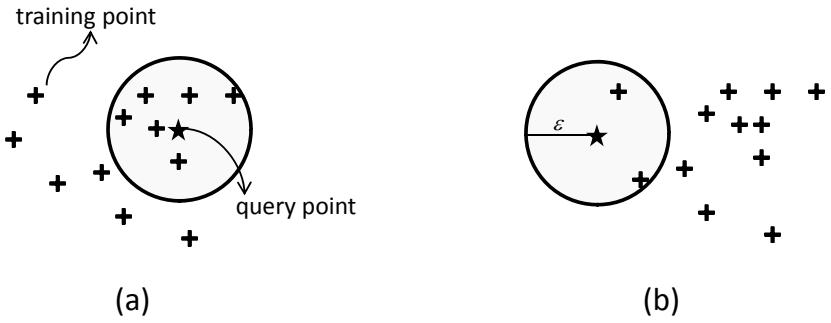


Figure 3.2: A schematic illustration of the ball-heuristic. The query point in part (a) is known because there are more than C training points in its ϵ-vicinity. The point in (b) is unknown because only two points exist in the ϵ-ball around it.

Although these two heuristics do not provide a formal guarantee about our estimation at the query point, they are related to the smoothness assumptions of the transition function. For example, the ball-heuristic uses an ϵ-ball because, according to Lipschitz-continuity, the transition function of any point s' in the ϵ-ball around s can only be different from the transition function of s by a maximum amount of ϵC_T. For a transition function that has the form of a normal distribution (Assumption 2), it means we have: $\parallel \mu(s,a) - \mu(s',a) \parallel_2^2 \leq \epsilon C_T$.

3.1.3 Disadvantages of the KWIK framework

Self-aware exploration still suffers from a kind of data inefficiency due to the way the knownness concept is designed and also the nature of the KWIK framework. The algorithms developed with a discrete separation between known and unknown states have to sacrifice data for estimation accuracy because they are only allowed to make predictions if they are ϵ-accurate. For example, Rmax does not make any estimation of $T(s,a)$ unless it has seen C samples from (s,a), effectively ignoring all the training samples in the first $C-1$ steps. In real-life applications, where collecting data is typically very expensive, ignoring precious data points is often unacceptable [4].

[4] An exception to this rule is the *Model-based Interval Estimation* algorithm (Strehl and Littman, 2005) that uses all the available information instead of forming known and unknown states. Unfortunately, no version of this algorithm has been published detailing its use in continuous spaces.

In most implementations of these algorithms, the user has no choice but to replace the conservative large values of C—dictated by the bounds[5]—with smaller hand-selected values to overcome the inefficient use of data. This approximation often has its own set of problems. For example, apart from the difficulty of choosing an appropriate value of C for a domain, this crude approximation sometimes undermines the effectiveness of self-aware exploration. In this framework, once a state-action becomes known, the algorithm stops systematically searching for samples from that pair. If the value of C is set too small (to increase data efficiency of the algorithm), there is a high chance of underestimating the value of good states and failing to converge to the optimal policy altogether.

An exception algorithm that avoids these problems, yet achieves a PAC-MDP guarantee is *model-based interval estimation* (MBIE) (Strehl and Littman, 2005) for finite state space MDPs. This algorithm does not form known/unknown state-action pairs and uses confidence intervals along with its parameter estimations to maintain an optimistic value function. Because of this approach, this algorithm makes a better use of data by using all of the available experience to build its internal model (as opposed to throwing out the experience collected from unknown state-action pairs). Another variation of MBIE was later developed by Strehl and Littman (2008b), which converted confidence intervals into bonus values—a concept closer to known/unknown partitioning of state-action pairs. Unfortunately, the ideas behind MBIE has not been successfully applied to continuous space MDPs.

In the next section, a novel extension of the knownness concept is developed that allows for more efficient use of samples during learning for continuous MDPs.

3.2 Continuous Knownness

Continuous knownness (or simply knownness) is a generalization of the known function with binary output $\{0, 1\}$. It is a function, denoted by ψ, that given a set of transition data, maps state-action pairs to the interval $[0, 1]$ and identifies how *much* the agent

[5]For example, the threshold for Rmax is $C = \mathcal{O}\left(\frac{\mathcal{S}+\ln(\mathcal{SA}/\delta)}{\epsilon^2(1-\gamma)^4}\right)$.

knows about the transition function for that pair. This section provides details about how this function can be constructed and how it can help with the data efficiency of model-based algorithms. A new framework called *CKWIK* is also developed to formalize the knownness concept, similar to the way KWIK formalized the use of Boolean knownness function.

Definition 10 (CKWIK online learning)**.** *An algorithm online-learns the transition function of an MDP in the CKWIK framework using the following protocol: Unlike the KWIK protocol, the learner does not receive any accuracy parameters upfront. At each timestep $t = 1, 2, \ldots$, the learner is presented with (s_t, a_t) and δ_t. It is then asked for $T(s_t, a_t)$. The learner outputs $\hat{T}(s_t, a_t)$ and $\epsilon(s_t, a_t)$ and guarantees that $\hat{T}(s_t, a_t)$ is $(\epsilon(s_t, a_t), \delta_t)$-close to $T(s_t, a_t)$ (According to Def. 6). Context permitting, we sometimes use ϵ_t to mean $\epsilon(s_t, a_t)$. The learner is then given access to s', which is an i.i.d. sample from $T(s_t, a_t)$. The agent has a CKWIK bound of $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$ if the following holds for any input sequence and any ϵ and δ:*

$$B(\epsilon, \delta, \mathcal{S}, \mathcal{A}) \geq \sum_{t=1}^{\infty} \mathbb{I}(\epsilon_t \geq \epsilon | \delta_t \geq \delta). \tag{3.11}$$

The difference between the new framework and KWIK is that the learner always outputs an estimate of the transition function and never returns $\sqcap$. It also outputs the accuracy of the estimate, according to the given failure probability. The next proposition shows that CKWIK is a special case of KWIK.

Proposition 11. *If Algorithm $\mathcal{A}$ solves a problem in the CKWIK framework with the bound $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$, the same problem can be learned in the KWIK framework with the same bound $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$.*

Proof. We can prove this statement by constructing Algorithm $\mathcal{B}$ from $\mathcal{A}$ as follows:

- **Initialize:** save ϵ and δ internally, and initialize $\mathcal{A}$.
- **Query** (s_t, a_t)**:** Query $\mathcal{A}$ with (s_t, a_t, δ), receive output $\hat{T}(s_t, a_t)$ and ϵ_t. Output $\sqcap$ if $\epsilon_t > \epsilon$, otherwise return $\hat{T}(s_t, a_t)$. After receiving s', pass it to $\mathcal{A}$.

Each time $\mathcal{B}$ returns an estimate, it is indeed (ϵ, δ)-close according to the protocol of CKWIK. The number of times it returns $\sqcap$ is bounded by the number of times $\mathcal{A}$ returns an accuracy $\epsilon_t > \epsilon$ (note that we use $\delta_t = \delta$ for all timesteps). This number is bounded by $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$ according to our assumptions. □

A direct result of this proposition, which extends Theorem 9, is presented in the following corollary.

Corollary 12. *If the transition function of a class of MDPs $\mathcal{M}$ is CKWIK-learnable with a bound B that is polynomial w.r.t. its arguments, we can construct an algorithm for the online RL problem that is PAC-MDP in $\mathcal{M}$.*

Although the new framework is stronger than KWIK, it turns out that most of the known techniques in KWIK can be easily converted to work in CKWIK. In fact, one of the intuitions behind this new setup is that most of the available learners in the KWIK framework know the accuracy of their estimates for each input, but use a threshold to convert that information to a binary output, just to be compatible with the Boolean knownness concept in Rmax-type exploration. By changing this concept to have a continuous range of values, we effectively allow the agent to use all of the available information at anytime during learning (see next section).

Given the CKWIK framework, we are now ready to establish the relationship between CKWIK learning and model-based PAC-MDP learning.

3.2.1 CKWIK-Rmax

Li (2009) introduced the abstract algorithm KWIK-Rmax that formally relates KWIK learning to the sample complexity of model-based RL algorithm (Li et al., 2008). Here, a similar relationship is developed between CKWIK learning and the sample complexity of exploration by constructing a new algorithm called CKWIK-Rmax.

This algorithm extends Boolean knownness to a continuous function based on the prediction accuracy of its CKWIK learner. Continuous knownness can be formally defined as follows:

Definition 13. *For a given desired accuracy ϵ_T, and allowed failure probability δ_T, and a CKWIK learner $\mathcal{F}_T$, the (continuous) knownness of a state-action pair is denoted by $\psi(s,a)$ and is defined as:*

$$\psi(s,a) = \min\left(\frac{1-\epsilon(s,a)}{1-\epsilon_T}, 1.0\right),$$

where $\epsilon(s,a)$ is the output of $\mathcal{F}_T(s,a,\delta_T)$.

Given the above definition for knownness, CKWIK-Rmax constructs an augmented transition function $\hat{T}'$ based on its estimation of the transition function $\hat{T}$ and the knownness function ψ as follows:

$$\hat{T}'(s'|s,a) = \psi(s,a)\hat{T}(s,a). \tag{3.12}$$

To make sure $\hat{T}'$ is a well-defined probability distribution and an optimistic value function, the algorithm assigns the remaining probability, which is $1-\psi(s,a)$, to a fictitious state s^f that has the highest possible value $V_{\max}$. Algorithm 4 shows the details of CKWIK-Rmax. The two inputs (ϵ_T, δ_T) are constructed from user inputs (ϵ, δ) as will be described later.

Algorithm 4 CKWIK-Rmax: A general model-based algorithm that works with any CKWIK learner.

1: **Input:** CKWIK learner $\mathcal{F}_T$, accuracy parameter ϵ_T, and confidence δ_T.
2: Initialize the internal MDP $\hat{M}$ and its optimal policy $\pi^*_{\hat{M}}$.
3: **for** all timesteps $t = 1, 2, \cdots$ **do**
4: Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
5: Update $\mathcal{F}_T$ using (s_t, a, s_{t+1}).
6: Update internal MDP $\hat{M} = \langle \mathcal{S}, \mathcal{A}, \hat{T}', R', \gamma \rangle$, where $\hat{T}'$ is computed from $\hat{T}$ using Eqn. 3.12, and $\hat{T}$ is computed by using $\mathcal{F}_T$ with $\delta_t = \delta_T$.
7: **end for**

Sample-Complexity Analysis. The sample complexity of CKWIK-Rmax can be analyzed similarly to the analysis of KWIK-Rmax. Assume that $\mathcal{F}_T$ has a CKWIK bound of $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$. The following theorem provides a PAC-MDP bound for the sample complexity of CKWIK-Rmax.

Theorem 14. *Let $\mathcal{M}$ be a class of MDPs and assume that $\mathcal{F}_T$ is a CKWIK-learner*

of the transition function of $\mathcal{M}$ with a bound of $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$. Then, CKWIK-Rmax *is PAC-MDP with a sample complexity of:*

$$\mathcal{O}\left(\frac{V_{\max}}{\epsilon(1-\gamma)}\left(B(\epsilon(1-\gamma)/V_{\max}, \delta, \mathcal{S}, \mathcal{A}) + \ln\frac{1}{\delta}\right)\ln\frac{1}{\epsilon(1-\gamma)}\right),$$

provided that the accuracy parameters for $\mathcal{F}_T$ are set as:

$$\epsilon_T = \Theta(\epsilon(1-\delta)^2), \quad \delta_T = \Theta(\delta).$$

Proof (sketch). The proof of this theorem follows the same steps as in the proof of (Li, 2009; Thm. 21). We just need to define the set K_t from (Strehl, 2007; Thm. 1) as:

$$K_t \stackrel{\text{def}}{=} \{(s,a) \in \mathcal{S} \times \mathcal{A} | \epsilon(s,a) \leq \epsilon_T\},$$

Due to the length of the proof and the similarity, readers are referred to (Li, 2009) for details. □

This sample-complexity bound is very loose for CKWIK-Rmax. In fact, although this algorithm is more data efficient than KWIK-Rmax in general, the two bounds are the same. The reason for having the same bound is that the loss caused by the agent when navigating in state-action pairs outside the set K_t is upper-bounded by $V_{\max}$ in the proof. While this upper bound is logical in KWIK-Rmax because no intelligent estimation is made for these pairs, it is very loose for CKWIK-Rmax because state-action pairs outside K_t can still have knownness values more than 0. The purpose of the theorem is to show that this algorithm, with clear practical advantages, does not pay a theoretical cost in the worst case. It is still an open question to find a tighter bound for CKWIK-Rmax.

In the next chapters, we introduce some algorithms instantiated from CKWIK-Rmax and study their properties.

Chapter 4

Model-based Learning with CKWIK Framework

This chapter introduces two novel algorithms derived from CKWIK-Rmax in two different classes of continuous MDPs. The first algorithm, which is an extension of fitted-Rmax called Cfitted-Rmax (or CF-Rmax), learns in the general continuous MDP setting that follows Assumption 2. While this algorithm makes a few assumptions about the smoothness of the environment, it does not receive any other information about the world. This algorithm is the first practical method to have a PAC-MDP guarantee in general Lipschitz-smooth continuous domains.

The next algorithm, which is called factored fitted-Rmax or FF-Rmax, builds on the idea of CF-Rmax for scenarios in which the user can provide extra information about the dependency between state variables. It is shown that this algorithm can use the extra information to gain an exponential speedup in learning.

4.1 CF-Rmax

Fitted-Rmax, as discussed in the previous chapter, is a model-based algorithm designed for continuous spaces. It uses instance-based function approximation (kernel regression to be exact) to estimate the transition function and a heuristic to compute the known state-action pairs. Although the specific heuristic that is used in the algorithm breaks the PAC-MDP guarantee of the algorithm, it is still widely used in practice because it makes fewer limiting assumptions about the environment than the ones with the PAC-MDP guarantee.

This section introduces CF-Rmax, which is a similar algorithm to fitted-Rmax in that they both use kernel regression for estimating the transition function. However, the new

algorithm is designed to work in the newly developed CKWIK framework, and in that regard, is an instantiation of the CKWIK-Rmax introduced in the previous chapter.

Unlike its counterpart in the KWIK framework, CKWIK-Rmax has a provably efficient PAC-MDP sample complexity in continuous domains as well. This achievement derives from the development of a kernel regression algorithm with an efficient CKWIK sample complexity. CKWIK-Rmax marks the first practical algorithm with PAC-MDP guarantee in general continuous MDPs.

4.1.1 CKWIK Kernel Regression

The CKWIK framework is an online learning setting. On the other hand, kernel regression (KR) takes place in a batch setting; similar to the original fitted-Rmax, we need to repeatedly use KR at each timestep.

Suppose that an MDP M satisfies the parametric form of Assumption 2 and we wish to CKWIK-learn its transition function using kernel regression. Also, suppose that the only unknown parameter of the transition function for each state-action pair is its mean $\mu(s, a)$. The rest of this section assumes that the transition function has the shape of a multivariate normal distribution for each state-action pair with a fixed covariance matrix Σ. However, the results carry over to more general known distributions. Finally, let the smoothness assumption be defined according to Equation 2.2—the means of the two distributions are close to each other if the two starting states are close.

At each timestep $t = 1, 2, \ldots$, the algorithm is given a state-action pair (s_t, a_t) along with a failure probability δ_t and is asked for an estimate for $T(s_t, a_t)$ (or equivalently $\mu(s_t, a_t)$). The regressor needs to return $\hat{\mu}(s_t, a_t)$ along with $\epsilon(s_t, a_t)$ and guarantee that $\hat{T}(s_t, a_t)$ is $(\epsilon(s_t, a_t), \delta_t)$-close to $T(s_t, a_t)$. A kernel regressor $\mathcal{F}$, when given a set of training data points $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, and a kernel function k, estimates the function at the query point x by:

$$\hat{y} \stackrel{\text{def}}{=} \mathcal{F}(x; D) = \frac{\sum_{x_i \in D} k(x, x_i) y_i}{\sum_{x_i \in D} k(x, x_i)}. \tag{4.1}$$

Since the smoothness assumption is defined on individual actions, no generalization

is allowed between different actions. Therefore, one data set for each action a should be maintained in the kernel regression. Let D_a be the training points for action a and $\mathcal{F}_a$ be the corresponding regressor. Each data set is constructed as follows:

$$D_a = \{(s_i, s'_i) |\ (s_i, a_i, s'_i) \in D, a_i = a\}. \tag{4.2}$$

Note that the prediction at time t is performed before s'_t is provided, so the training data contains points up to timestep $t-1$. The kernel regressor's output at each timestep is then $\mathcal{F}_{a_t}(s_t, D_a)$. The regressor also outputs $\epsilon(s_t, a_t)$, whose value will be determined shortly.

Several variations of kernel regression exist. We build our algorithm based on one of these variations that approximates Equation 4.1 using a smaller number of points. The basic idea behind this approximation is improving computational complexity. The running time of kernel regression is linearly dependent on the number of samples in D_a, which makes it impractical in many applications. Approximate kernel regression is a technique that combines kernel regression with another nonparametric method called "k-nearest neighbors" (Russell and Norvig, 1994). In this method, the sum $\sum_{x_i \in D}$ is replaced with another sum $\sum_{x_i \in \mathcal{N}_c(x)}$, where $\mathcal{N}_c(x)$ is the set of c-closest points to x. Hence, the approximate kernel regression becomes:

$$\hat{y} \stackrel{\text{def}}{=} \mathcal{F}(x; D) = \frac{\sum_{x_i \in \mathcal{N}_c(x)} k(x, x_i) y_i}{\sum_{x_i \in \mathcal{N}_c(x)} k(x, x_i)}. \tag{4.3}$$

The running time of this new method is much faster because it only depends linearly on c plus the time needed to compute $\mathcal{N}_c(x)$ from D_a. There is a huge literature on performing k-nearest neighbors in sub-linear time, (Arya et al., 1994; Vijayakumar and Schaal, 2006).

Equation 4.3 is usually a good approximation of the original kernel regression. The reason is that kernel functions typically decay to 0 very fast as the distance between the points increases. Therefore, most of the contributions to the output of the kernel regression come from the nearby points anyways, so eliminating far points from

computation of the regression does not incur much loss.

The CKWIK-KR algorithm is an extension of approximate KR that computes a different neighbor count c for each individual query point. We show that if c's are selected carefully, CKWIK-KR will have a CKWIK bound. The variable c is computed based on the relative position of the query point w.r.t. other points in D_a, as well as the allowed failure probability δ_t and the Lipschitz smoothness constant C_T. The algorithm computes the optimal c using a loss function $\mathcal{L}$. We will see later on that the particular choice of $\mathcal{L}$ that is defined next helps the algorithm achieve a CKWIK bound:

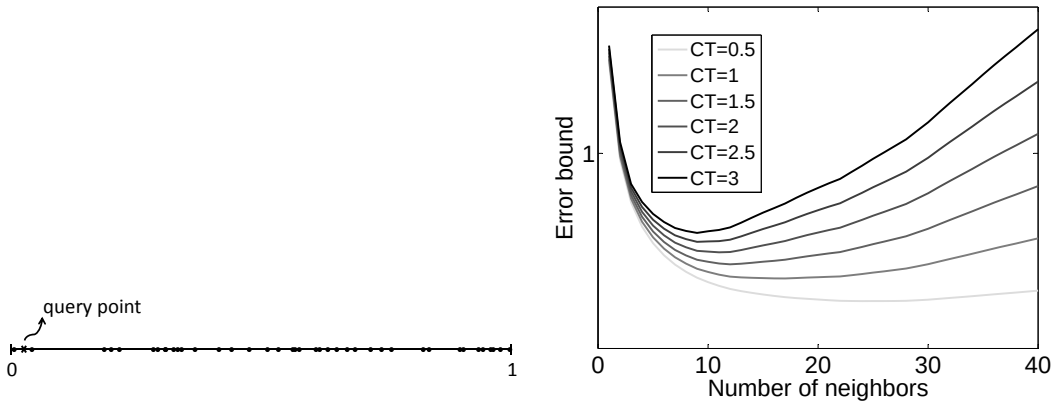
$$\mathcal{L}(c,x) \stackrel{\text{def}}{=} \sqrt{\frac{\ln(2/\delta_t)\sum_{s_j\in\mathcal{N}_c(x)} k(s_j,s^*)^2}{2[\sum_{s_j\in\mathcal{N}_c(x)} k(s_j,s^*)]^2}} + \frac{C_T}{c}\sum_{i=1}^{c} \parallel s_i - s^* \parallel_2^2 . \tag{4.4}$$

CKWIK-KR computes c_s for each query point as:

$$c_s = \underset{c}{\operatorname{argmin}}\, \mathcal{L}(c,x). \tag{4.5}$$

If optimizing $\mathcal{L}$ over all possible values of c is computationally hard, a greedy search approximation can be applied by sorting all the points according to their distance to the query point and introducing them one at a time to Equation 4.4. It is straightforward to update this equation in constant time with each new point. The algorithm then stops whenever $\mathcal{L}(c+1) > \mathcal{L}(c)$ (described in Algorithm 5). Figure 4.1 shows the behavior of $\mathcal{L}$ as a function of c for a simple example. The picture on the left is a set of 40 training points scattered in $[0,1]$ along with a query point. The graph on the right shows the value of $\mathcal{L}$ when c is varied. Different lines denote different values for C_T. As you can see, the loss function first goes down as more points are included in the set. But, at some point it goes back up as including more training points means using farther and farther points that might not be related to the query point. The higher values of C_T have a more steep incline because they are less smooth (their transition function is allowed to change more).

This algorithm dynamically adjusts how many points from the data set are used to estimate the value at any query point. The number of nearest points used for each



(a) (b)

Figure 4.1: A simple example showing the effect of number of neighbors used in kernel regression on the error bound function $\mathcal{L}(x, c)$. (a) shows a training set along with a query point, (b) shows the error bound computed for different values of c.

state is selected such that the loss function is minimized. In general, the more samples we use to estimate a transition function, the closer the estimation becomes to the true function. On the other hand, the farther the points are from the query points (which is a direct result of including more points), the less reliable they become. The loss function $\mathcal{L}$ tries to capture this tradeoff. The first term tends to go down as c grows larger, while the second term goes up. That is why we need to find the optimal c that minimizes the loss function. As we will see later on, function $\mathcal{L}$ bounds the difference between the means of the estimated and the true transition functions. The total variation between $\hat{T}(s, a)$ and $T(s, a))$ is related to the difference between their means:

$$\mathrm{d}_{\mathrm{var}}(\hat{T}(s,a), T(s,a)) \leq \sqrt{\frac{\| \hat{\mu}_{s,a} - \mu_{s,a} \|_2^2}{\lambda_n}}, \tag{4.6}$$

where λ_n is the smallest eigenvalue of the covariance matrix Σ (Li, 2009; Lemma 19). Therefore, the algorithm returns $\mathcal{L}_{\mathrm{var}}$ as the accuracy of the estimate:

$$\mathcal{L}_{\mathrm{var}}(s) = \lambda_n \mathcal{L}(s, c_s). \tag{4.7}$$

Next, the theoretical properties of this algorithm is studied.

Algorithm 5 CKWIK-KR, a learner of transition functions in the CKWIK framework.

Inputs: the kernel function k, Lipschitz constant C_T.
Initialize dataset D_a for all a's.
for all timesteps $t = 1, 2, \cdots$ **do**
  Observe (s_t, a_t).
  Initialize counter i to 0.
  repeat
    $i \leftarrow i + 1$.
    Compute $\mathcal{L}(i, s_t)$ using Eqn. 4.4 and D_{a_t}.
    Compute $\hat{y}_i$ using Eqn. 4.3 and $\mathcal{N}_i(s_t)$.
  until $\mathcal{L}(i, s_t) > \mathcal{L}(i-1, s_t)$ AND $i > 1$
  output $(\hat{y}_{i-1}, \mathcal{L}_{\text{var}}(i-1, s_t))$.
  Receive s_{t+1} as an i.i.d. sample from $T(s_t, a_t)$.
  Add (s_t, s_{t+1}) to D_{a_t}.
end for

Sample-complexity Analysis

Let us start by investigating an individual $\mathcal{F}_a$. The set D_a contains $\{s_1, s_2, \ldots, s_n\}$ along with samples from their transition functions $\{s'_1, s'_2, \ldots, s'_n\}$. Let $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ be the multivariate normal distributions representing the transition functions for each s_i. In other words, we have $s'_i \sim \mathcal{P}_i = \mathcal{N}(\mu_i, \Sigma)$. The following lemma relates the distribution mean of a query point to that of nearby points, when kernel regression is applied.

Lemma 15. *Let s^* be a query point and μ^* be mean of its transition function. If we have $\| \mu_i - \mu^* \|_2^2 \leq \epsilon_i$ for all the states $s_i \in \mathcal{N}_c(s^*)$, we then have:*

$$\| \mathbb{E}\left[\mathcal{F}_a(s^*; \mathcal{N}_c(s^*)\right] - \mu^* \|_2^2 \leq \frac{1}{c} \sum_{i=1}^{c} \epsilon_i. \tag{4.8}$$

Proof. Without loss of generality, assume that $d(s^*, s_1) \leq d(s^*, s_2) \leq \cdots \leq d(s^*, s_c)$. Define random variable Z to be $\mathcal{F}_a(s^*; \mathcal{N}_c(s^*))$, X_i to be the random variable drawn

from $\mathcal{P}_i$, and $k_i = \frac{k(s_i,s^*)}{\sum_j k(s_j,s^*)}$. We have:

$$\| \mathbb{E}[Z] - \mu^* \|_2^2 = \| \sum_{i=1}^{c} k_i \mathbb{E}[X_i] - \mu^* \|_2^2 \tag{4.9}$$

$$= \| k_1\mu_1 + \cdots + k_c\mu_c - \mu^* \|_2^2 \tag{4.10}$$

$$= \| k_1\mu_1 + \cdots + k_c\mu_c - k_1\mu^* - \cdots - k_c\mu^* \|_2^2$$

$$\leq k_1 \| \mu_1 - \mu^* \|_2^2 + \cdots + k_n \| \mu_c - \mu^* \|_2^2 \tag{4.11}$$

$$\leq (1/c) \| \mu_1 - \mu^* \|_2^2 + \cdots + (1/c) \| \mu_c - \mu^* \|_2^2$$

$$= \frac{1}{c} \sum_{i=1}^{c} \epsilon_i. \tag{4.12}$$

In this derivation, (4.10) is because of the linearity of expectation operator, (4.11) is because $\sum k_i = 1$ and finally, (4.12) is because we assumed the points are sorted according to their distance to the query point and k_i's sum up to 1. □

The following lemma analyzes the accuracy of kernel regression based on the set of points in $\mathcal{N}_c(s^*)$.

Lemma 16. *Given $0 < \delta < 1$, a set of training data D_a, and positive integer $c \geq 1$, let $\mathcal{L}_a(s^*, D_a, c)$ be the loss occurred by using c neighbors of s^* in the kernel regression:*

$$\mathcal{L}_a(s^*, D_a, c) = \| \mathcal{F}_a(s^*; \mathcal{N}_c(s^*)) - \mu^* \|_2^2 .$$

The following accuracy holds with probability at least $(1-\delta)$:

$$\mathcal{L}_a(s^*, D_a, c) \leq \sqrt{\frac{\ln(2/\delta) \sum_{s_j \in \mathcal{N}_c(s^*)} k(s_j, s^*)^2}{2[\sum_{s_j \in \mathcal{N}_c(s^*)} k(s_j, s^*)]^2}} + \frac{C_T}{c} \sum_{i=1}^{c} \| s_i - s^* \|_2^2 . \tag{4.13}$$

Proof. Let us first bound the difference $\| \mathcal{F}_a(s^*; \mathcal{N}_c(s^*)) - \mathbb{E}[\mathcal{F}_a(s^*; \mathcal{N}_c(s^*))] \|_2^2$ or equivalently $\| Z - \mathbb{E}[Z] \|_2^2$. We can rewrite kernel regression as:

$$\mathcal{F}_a(s^*; \mathcal{N}_c(s^*)) = \sum_{i=1}^{c} k_i X_i. \tag{4.14}$$

Consider random variables $k_i X_i$. The boundary of these variables is $0 \leq k_i X_i \leq \frac{k(x_i,x^*)}{\sum k(s_j,s^*)}$. Now, apply the Hoeffding inequality (Hoeffding, 1963) to the sum of these random variables:

$$\Pr(\parallel Z - E[Z] \parallel_2^2 \geq t) \leq 2\exp\left(\frac{-2t^2[\sum k(s_j,s^*)]^2}{\sum k(s_j,s^*)^2}\right). \tag{4.15}$$

Given a failure probability δ, we can compute t based on the number of samples c as follows:

$$2\exp(\frac{-2t^2[\sum k(s_j,s^*)]^2}{\sum k(s_j,s^*)^2}) \leq \delta \tag{4.16}$$

$$t \geq \sqrt{\frac{\ln(2/\delta)\sum k(s_j,s^*)^2}{2[\sum k(s_j,s^*)]^2}}. \tag{4.17}$$

So, for any t that satisfies Equation 4.17, it is guaranteed that $\Pr(\parallel Z - E[Z] \parallel_2^2 \geq t) \leq \delta$. Using Lemma 15 and the result above, we have:

$$\Pr\left(\parallel \mathcal{F}_a(s^*;\mathcal{N}_c(s^*)) - \mu^* \parallel_2^2 \geq (\frac{1}{c}\sum_{i=1}^{c}\epsilon_i + t)\right) \leq \Pr(\parallel Z - \mathbb{E}[Z] \parallel_2^2 \geq t). \tag{4.18}$$

Therefore, for c samples, the following inequality holds with probability at least $(1-\delta)$:

$$\parallel \mathcal{F}_a(s^*;\mathcal{N}_c(s^*)) - \mu^* \parallel_2^2 \leq \sqrt{\frac{\ln(2/\delta)\sum k(s_j,s^*)^2}{2[\sum k(s_j,s^*)]^2}} + \frac{C_T}{c}\sum_{i=1}^{c} \parallel s_i - s^* \parallel_2^2,$$

where the second term is due to the expansion of the definition of ϵ_i. □

This lemma guarantees that the output of CKWIK-KR is indeed ϵ_t-accurate as computed in the algorithm. This lemma also gives detailed information about the behavior of kernel regression and establishes the relationship between the accuracy of the estimator and the points in the training set. In particular, the estimation error in Equation 4.13 is decomposed into two different errors: The first term measures how accurate

we estimate a stochastic quantity—the mean of a probability distribution—using a finite set of samples; therefore, the error goes down as the sample size goes up. The second term measures the error caused by not using samples from the same distribution, but rather from similar distributions (gathered from nearby points); therefore, as we include more points that are farther away from the query point in our estimation, the accuracy goes down. Computation of the optimal value of c has to be done online and is dependent on the relative position of the training data to each query point.

The following lemma bounds the error of kernel regression based on the distance of the points in $\mathcal{N}_c(s^*)$ for any data set D_a.

Lemma 17. *Given a training data D_a, a query point s^*, and an $\frac{\epsilon_1}{2C_T}$-ball around s^*, if c_1 points from D_a are in the $\frac{\epsilon_1}{2C_T}$-ball, where c_1 is:*

$$c_1 > \frac{18}{\sqrt{\frac{18\epsilon_1^2}{\ln(2/\delta)}+1}-1}+1, \tag{4.19}$$

the accuracy of CKWIK-KR is bounded by:

$$\| \mathcal{F}_a(s^*; D_a) - \mu^* \|_2^2 \leq \epsilon_1, \tag{4.20}$$

with probability at least $(1-\delta)$ for any input D_a, provided that the kernel width σ is larger than $\sqrt{\frac{\epsilon_1}{2C_T}}$.

The proof of this lemma is provided in Appendix A.2 due to its length. The next step completes our analysis of the sample complexity of CKWIK-KR in the CKWIK framework.

Theorem 18. *Let $\mathcal{M}$ be a class of MDPs as defined in Assumption 2, with $\| \mathcal{S} \|_\infty \leq 1$. The CKWIK-KR algorithm CKWIK-learns the transition function of any MDP in $\mathcal{M}$ with the following bound:*

$$B(\epsilon, \delta, \mathcal{S}, \mathcal{A}) = \mathcal{O}\left(\sqrt{\ln(1/\delta)}|\mathcal{A}|\left(\frac{|\mathcal{S}|}{\epsilon}\right)^{|\mathcal{S}|/2}\right). \tag{4.21}$$

Proof. The theorem is proved by bounding the number of times the algorithm outputs an accuracy $\epsilon_t > \epsilon$ when $\delta_t = \delta$. It is trivial to show that having $\delta_t > \delta$ will not increase this bound. Consider one of the $\mathcal{F}_a$'s. Denote $|\mathcal{S}|$ by m and let ξ be a uniform discretization over the state space with the following resolution along each axis:

$$h = \sqrt{\frac{2mC_T}{\lambda_n \epsilon^2}}. \tag{4.22}$$

This discretization populates $|\xi| = h^m$ cells. The distance between any two points in each cell is at most:

$$\max_{x_i, x_j \in \xi_k} \| x_i - x_j \|_2^2 \leq \frac{\lambda_n \epsilon^2}{2C_T}.$$

We apply Lemma 17 to each cell ξ setting $\epsilon_1 = \lambda_n \epsilon^2$. For any query point in a cell, if there are c_1 training points inside $\xi(s^*)$ (as defined in Eqn. A.6), the output of the regressor is at least ϵ_1-close to the true mean w.p. at least $(1 - \delta)$. According to (Li, 2009; Lemma 19), when the mean of two normal distributions are ϵ-close, their total variations are $\left(\sqrt{\frac{\epsilon}{\lambda_n}}\right)$-close. Applying this lemma to ϵ_1, we get that the total variation between the transition function generated by the kernel regression and the true one is ϵ.

The only step left to do is to count the number of times a query point with less than c_1 training points in its cell can be encountered—denoted by event $\mathcal{G}$. But, that quantity is bounded by:

$$\begin{aligned}
|\mathcal{G}| &\leq c_1 |\xi| \\
&= c_1 h^m \\
&= \left(\frac{18}{\sqrt{\frac{18\epsilon^2}{\ln(2/\delta)} + 1} - 1} + 1 \right) \left(\frac{2mC_T}{\lambda_n \epsilon^2} \right)^{m/2} \\
&= \mathcal{O}\left((\ln(1/\delta))^{1/2} \left(\frac{m}{\epsilon^2} \right)^{m/2} \right).
\end{aligned} \tag{4.23}$$

The proof is completed by extending this result to consider all the actions. □

4.1.2 The Proposed Algorithm

Algorithm 6 shows the pseudo-code of CF-Rmax, which is an implementation of CKWIK-Rmax. This algorithm resembles fitted-Rmax in that they both use KR for estimating the transition function, and they both turn uncertainty in model estimation into optimistic values. However, the new algorithm takes into account the partially-known estimates of the transition function in a principled way when it solves its internal model. Therefore, it creates optimistic values that are tighter to the true value function.

Algorithm 6 CF-Rmax, a model-based algorithm for continuous space MDPs.

1: **Inputs:** CKWIK-KR regressor $\mathcal{F}$, accuracy ϵ_T and confidence δ_T.
2: Initialize the internal MDP $\hat{M}$ and its optimal policy $\pi^*_{\hat{M}}$.
3: **for** all timesteps $t = 1, 2, \cdots$ **do**
4: Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
5: Update $\mathcal{F}$ using (s_t, a_t, s_{t+1}).
6: Update internal MDP $\hat{M} = \langle \mathcal{S} + s^f, \mathcal{A}, \hat{T}', R', \gamma \rangle$,
where $\hat{T}(., a) = \mathcal{F}_a(., \delta_T)$ and $\hat{T}'$ is defined as:

$$\hat{T}'(s'|s,a) = \begin{cases} (1 - \psi(s,a)), & \text{if } s' = s^f \\ \psi(s,a)\hat{T}(s'|s,a), & \text{otherwise.} \end{cases} \tag{4.24}$$

7: **end for**

Sample-complexity Analysis. The sample complexity of CF-Rmax can be directly derived using the result of Theorem 14 using the CKWIK bounds of CKWIK-KR. In particular, since the sample complexity of CKWIK-Rmax is:

$$\mathcal{O}\left(\frac{V_{\max}}{\epsilon(1-\gamma)}\left(B(\epsilon(1-\gamma)/V_{\max}, \delta, \mathcal{S}, \mathcal{A}) + \ln\frac{1}{\delta}\right)\ln\frac{1}{\epsilon(1-\gamma)}\right),$$

and the CKWIK sample complexity of $\mathcal{A}_T$ in CF-Rmax is:

$$B(\epsilon, \delta, \mathcal{S}, \mathcal{A}) = \mathcal{O}\left(\sqrt{\ln(1/\delta)}|\mathcal{A}|\left(\frac{|\mathcal{S}|}{\epsilon}\right)^{|\mathcal{S}|/2}\right)$$

The PAC-MDP sample complexity of CF-Rmax becomes:

$$\mathcal{O}\left(\sqrt{\ln(1/\delta)}|\mathcal{A}|\left(\frac{V_{\max}|\mathcal{S}|}{\epsilon(1-\gamma)}\right)^{|\mathcal{S}|/2} + \ln\frac{1}{\delta}\right). \tag{4.25}$$

Again, it is important to note that the lower bound for *planning* an ϵ-optimal policy, which is a much easier problem, is $\Omega\left(\frac{1}{\epsilon}\right)^{|\mathcal{S}|}$.

4.1.3 Empirical Results

This section provides some experimental results, demonstrating the performance of CF-Rmax in several test domains. Two widely-used domains in the reinforcement-learning literature were used for comparison, as described next.

Mountaincar (Sutton and Barto, 1998). In this domain, an underpowered car tries to climb up to the right side of a valley, but has to gain its energy through several back and forth travels to the left of the valley. The state space is 2-dimensional and consists of the horizontal position of the car x, and its velocity v. The action set is *forward*, *backward*, and *neutral*, which correspond to accelerating in the intended direction. Agent receives -1 penalty at each timestep except for when it escapes the valley to receive a reward of 0 and end the episode. Technical details of this domain can be found in Appendix C.1.

Puddleworld (Sutton, 1996). In this domain, the agent is placed inside a bounded region ([0,1],[0,1]) and its goal is to move to a small goal region using 4 available actions: {north, east, south, west}. Along the way, the agent has to avoid moving over some parts of the state space marked as puddles. The state space of this domain is the position of the agent in the world (X, Y). Each of the 4 available actions move the agent in the intended direction by some fixed step size to which a small amount of Gaussian noise is added. Each timestep in PUDDLEWORLD accrues -1 penalty if the agent is outside the puddle regions. The agent receives more penalty when it enters the puddle. The amount of the penalty depends on how close to the center of the puddle the agent is. The episode ends when the agent gets to the goal region or a cap of 300 steps is reached. The details of this domain can be found in Appendix C.2.

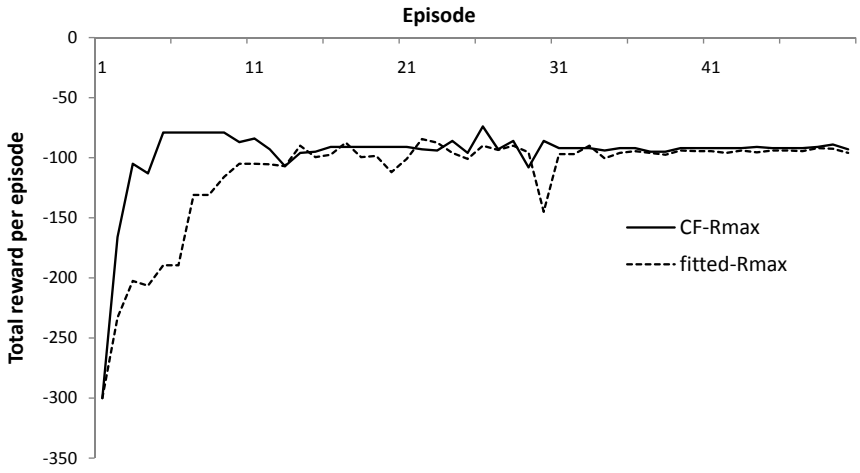


Figure 4.2: Performance of CF-Rmax and fitted-Rmax algorithms in MOUNTAINCAR.

Experiment Setups

To compare the performance of different algorithms, we evaluated them in an online reinforcement-learning setting. The parameters of the domains were their default values as explained in the Appendix C unless specified otherwise. In each experiment, the algorithm was executed for some fixed E episodes and its sum of collected rewards for each episode was saved. The whole experiment was repeated 20 times for the sake of statistical significance.

We first compared CF-Rmax with its sister algorithm fitted-Rmax in MOUNTAINCAR. Both algorithms used $\sigma = 0.3$, fitted value iteration (FVI) (§ B.2.2) as the planner and resolved their models every 50 timesteps. The parameters for FVI for both algorithms were as follows: a discretization with $h = 30$ was used to generate the backup points and 5 samples were used for each Bellman backup. Figure 4.2 shows the result of the two algorithms in MOUNTAINCAR when 50 episodes are used for learning.

Although fitted-Rmax is a very powerful algorithm, CF-Rmax managed to converge faster to the optimal policy. In particular, fitted-Rmax spent a couple of episodes just collecting data without using them because most of the states rendered unknown due to the nature of the Boolean function used to compute known states. CF-Rmax started using the data right from the beginning, using the continuous knownness function.

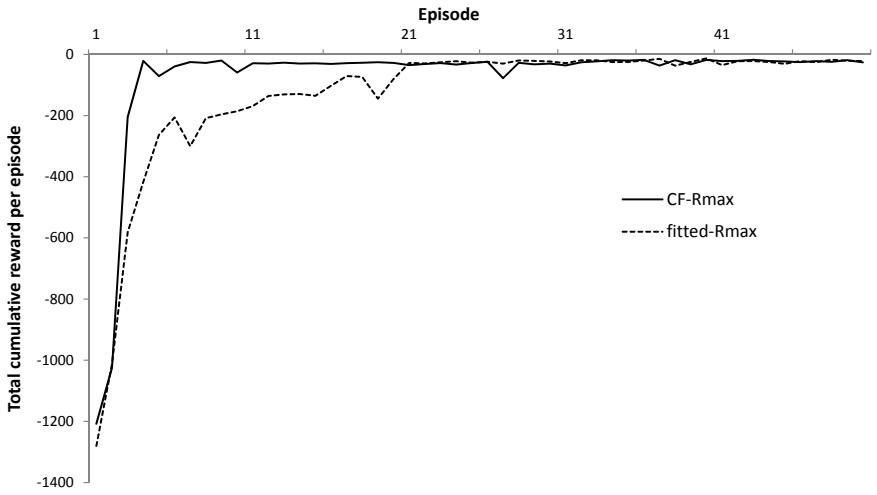


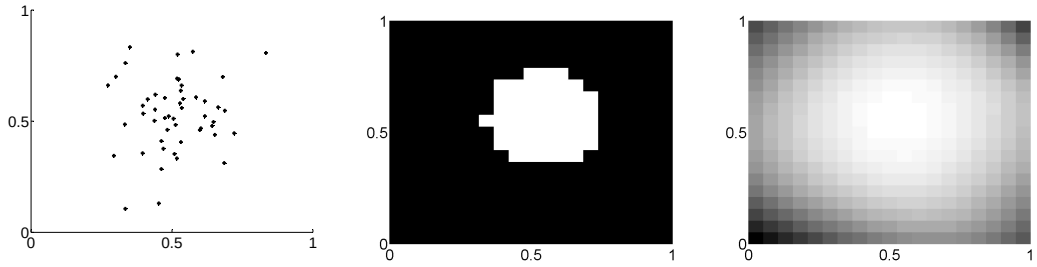
Figure 4.3: Performance of CF-Rmax and fitted-Rmax algorithms in PUDDLEWORLD.

Therefore, it did a better job in the early episodes.

The same experiment was performed in PUDDLEWORLD. As Figure 4.3 shows, similar results were obtained. Both algorithms were able to converge to the optimal policy, but CF-Rmax did better in the early stages of learning.

To get more insight into why CF-Rmax was superior to fitted-Rmax, we investigated the knownness function in a simple experiment. We selected a part of MOUNTAINCAR's transition function: $f(x, y) = v_t - 0.0025\cos(3x_t)$. We then compared the knownness functions in both algorithms using a training set of 50 points. Figure 4.4(a) shows the training points in the normalized space $[0, 1]^2$, along with the knownness values across the entire space for both algorithms in part (b) and (c). The darker parts of the image indicate smaller knownness values (0 is black and 1 is white). Since fitted-Rmax used a Boolean function for the knownness, it only generated black and white regions. Parameters for computation of knownness in both algorithms were selected in such a way that the algorithms created similar regions with knownness equal to 1. As you can see in part (b), fitted-Rmax created hard boundaries between known and unknown regions. This algorithm was not data efficient because it refused to make any predictions for the black region. The fact that the algorithm completely trusted its estimates in the close-to-boundary points in the white region and did not make any

predictions for close-to-boundary points in the black region explains why the algorithm is data inefficient. On the other hand, CF-Rmax produced a smooth knownness function that decayed to 0 as it got farther from the training points.



(a) Training points (b) fitted-Rmax (c) CF-Rmax

Figure 4.4: Comparison of the knownness functions in fitted-Rmax and CF-Rmax using data from MOUNTAINCAR. Graph (a) shows the training data, (b) and (c) show the knownness computed in fitted-Rmax and CF-Rmax respectively.

We then tested the effect of varying the kernel width on the performance of both algorithms in MOUNTAINCAR with the same setup. Both algorithms were sensitive to the value of the kernel width in a classical tradeoff between bias and variance. Small kernel widths created regressors with very low bias, but spiky, with high variance outcomes. The effect of this configuration on the performance of CF-Rmax was detrimental. On the other hand, very high values of kernel width created regressors with low variance but high bias, which were not able to capture the shape of the transition function. Again, the effect of this configuration on the RL algorithm was detrimental. The best configurations happened to be values that created a balance between the bias and variance of the regressor.

4.2 Factored Learner for Continuous Spaces

In the previous section, we saw how continuous knownness helped balancing exploration-exploitation in a more data-efficient way in continuous spaces. We also developed the first algorithm using this concept in the CKWIK framework, and showed that it is PAC-MDP.

This algorithm is appealing because it marries two lines of works for solving general

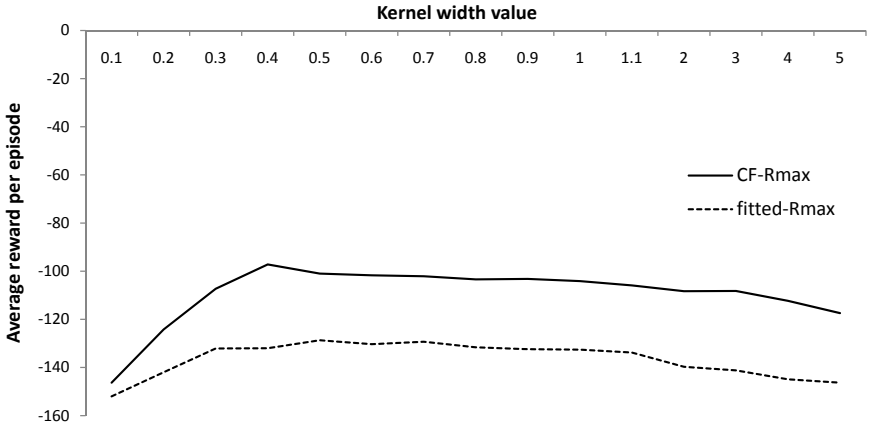


Figure 4.5: The effect of varying the kernel width on the performance of CF-Rmax and fitted-Rmax in MOUNTAINCAR.

continuous MDPs. Kakade et al. (2003) introduced an abstract algorithm called Metric E^3 that learns efficiently in the PAC-MDP framework, but left important implementation details—such as a concrete way to learn the model and how to glue it to the planner—to the user. On the other hand, some algorithms (for example, fitted-Rmax) provided a concrete way of learning the model, without providing any convergence guarantees. The algorithm we proposed in the previous chapter followed the practicality of fitted-Rmax by providing a concrete algorithm designed for general continuous MDPs, yet inherited the theoretical properties of Metric E^3 by providing a PAC-MDP bound. This bound relates the number of mistakes the algorithm makes to some relevant quantities, including $\frac{1}{\epsilon}$, $\frac{1}{1-\delta}$ and the complexity of the environment with high probability.

Unfortunately, one of the common misconceptions about the efficiency of learning in continuous spaces is that the complexity of a system is measured by the dimensionality of its state space, which is not the case for general continuous MDPs. In fact, one of the difficulties of learning in general continuous MDPs is that their complexity scales exponentially with respect to the dimensionality of their state spaces.

The reason for this scaling mismatch is due to the way the smoothness conditions are defined in these MDPs. According to Lipschitz-continuity assumption, the difference between the transition functions of two points is bounded by a linear function of their

Euclidian distance. With this condition, any information about the transition function at one point can be used as an ϵ-close approximation for the transition function of another point only if the second point lies inside a ball around the first point with a radius of $\frac{\epsilon}{C_T}$, where C_T is the Lipschitz constant.

It can be shown that given a filling of the space with $\frac{\epsilon}{C_T}$-balls, the value of the transition function needs to be specified in at least one point in each ball, or otherwise an ϵ-approximation cannot be guaranteed.

It is well-known that the volume of a mathematical space grows exponentially as the dimensionality increases. For example, if we use 100 points evenly distributed in a unit interval, we have effectively covered the space with 0.01-balls. Achieving the same covering in a 10-dimensional unit hypercube will require 10^{20} points, which is in fact 10^{18} times larger than the former[1]. Therefore, the complexity of a continuous MDP scales exponentially in the dimensionality of its state space under the Lipshitz-continuous assumption (2.2).

As a consequence of this phenomenon, our reinforcement-learning algorithm is also bound to the exponential blowup. In fact, one can show that even computing a near-optimal policy of a completely known continuous MDP (the planning problem) has a lower bound that scales exponentially w.r.t the dimensionality of the state space in general. In particular, Chow and Tsitsiklis (1989) showed that any algorithm computing an ϵ-optimal policy in a d-dimensional continuous MDP needs at least $\Omega(\epsilon^{-d})$ samples from T and R. If the planning problem, which is a much easier problem than learning, has an exponential lower bound, we certainly cannot hope to construct an RL algorithm that scales polynomially in the dimensionality of the state space. To summarize, an algorithm that is allowed to make an exponential number of mistakes w.r.t. the state-space dimensionality is the best one can achieve in general continuous MDPs in the PAC-MDP framework.

The same problem, which is also referred to as "curse of dimensionality" or the "Hughes effect" (Bellman, 1961), has been studied in many disciplines in machine

[1]Example borrowed from Bellman (1957).

learning and optimization (Pearson, 1901; Guyon and Elisseeff, 2003; Kriegel et al., 2009). An intuitive way to attack this problem is to investigate algorithms that avoid the exponential blowup either by assuming a more structured model class (Draper and Smith, 1998) or by trying to find patterns and structures directly from the data (Liu and Motoda, 1998; Guyon and Elisseeff, 2003).

Some of these techniques have been followed up in the reinforcement-learning literature as well. For example, Strehl and Littman (2008a) studied an algorithm that provably scaled polynomially with the number of state variables, provided that the transition function had the form of a linear function. While this restriction may look very limiting, it provided an insight into how RL methods can be related to methods from control theory. Also, Brunskill et al. (2009) studied environments where the transition function can be described in terms of a series of constant functions for a non-overlapping partitioning of the state space. They introduced an algorithm that would scale polynomially with the number of partitions of the state space. Assuming that the environment has relatively few partitions, this algorithm can exploit this information to provide exponential speedup.

In this section, we consider factorized environments and construct an algorithm based on CF-Rmax that can exploit prior knowledge about the dependency structure in the environment (if any) to gain (possibly) exponential speedup in learning.

4.2.1 Factored-state MDPs

In continuous spaces, the state space of an MDP is naturally defined in a factored form. The state space $\mathcal{S}$ is a closed subset of $\mathbb{R}^n$, which means each state $s \in \mathcal{S}$ can be represented by a vector of size n. We call each element of this vector a *state variable* or a *factor* of the state. The transition function in general MDPs on the other hand, is not necessarily representable in factored form. This function maps state-action pairs to a probability distribution over the next states in their entirety, and therefore the function $T(s'|s,a)$ might not be decomposable in general. We say a transition function is decomposable if the state variables of s' are conditionally independent of each other.

In other words, we have:

$$T(s'|s,a) = \prod_{i=1}^{|S|} T(s'(i)|s,a). \tag{4.26}$$

The above equation states that each component of the state at time $t+1$ is only dependent on the components of the state and action at time t, and not on any information from time $t+1$. This setup is reminiscent of the way DBNs are constructed, where a graph is formed by using variables at time t and $t+1$ and the dependency set of all the variables at time $t+1$ contains only variables from time t.

This class of environments is actually very rich and covers a lot of real-life domains. In fact, it is not very easy to construct a physical control task outside the scope of this class. Even more so, it is typical for an environment to exhibit some weak form of decoupling between state variables even across timesteps. In other words, we do not usually need all the state variables at time t to predict each individual component of the next state. For example, consider several robots that are placed in an environment. Each robot has its own set of state variables to identify its current state, and the state of the environment is the concatenation of all state variables of all the robots. Predicting the value of each robots' variables in the next timestep typically requires the current state of that particular robot, but not the status of the others. We are interested in providing learning algorithms for environments that fall into this category.

Factorization of a transition function can be formally represented by the following structure: For each action a, a bipartite graph $G_a(X, Y, E)$ with $|X| = |S|$ and $|Y| = |S|$ is used to represent the interdependency of factors when action a is performed. Each node in the X and Y parts of the graph represents one state variable at time t and $t+1$ respectively. The edge between $x(i)$ and $y(j)$ is missing from E if and only if we have:

$$T(s'(j)|s,a) = T(s'(j)|s(1),\ldots,s(j-1),s(j+1),\ldots,s(|S|),a), \tag{4.27}$$

that is, there is conditional independence between the two variables. For example, Figure 4.6 shows the dependency graphs of a 3-dimensional MDP with 2 actions. The graph on the left indicates that the value of the first state variable, $s(1)$, at time $t+1$

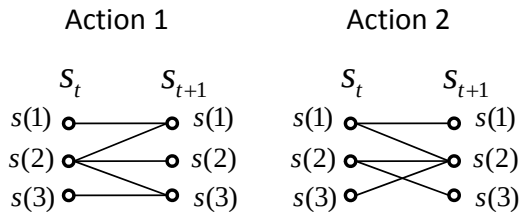


Figure 4.6: A simple example of dependency graphs for a 3-dimensional MDP.

is only dependent on the values of $s(1)$ and $s(2)$ at time t when action 1 is executed, while the value of $s(2)$ at time $t+1$ is dependent only on the value of $s(2)$ at time t. The rest of the graph is interpreted the same way.

The goal of this section is to construct an algorithm that can use these dependency structures to improve the sample complexity of learning.

4.2.2 FF-Rmax

Factored fitted-Rmax—or FF-Rmax for short—behaves very similarly to CF-Rmax in that they both use CKWIK-KR to maintain an internal model of the world. They also augment their internal models by bonus values based on the knownness of state-action pairs. And finally, they both act greedily with respect to their internal model while updating their models on a regular basis. The key difference between the two algorithms is in the way they train their function approximators and how they compute knownness. This difference is best understood by a simple example of an online regression problem in the CKWIK framework (Nouri and Littman, 2008).

Suppose we want to online-learn the function $z \stackrel{\text{def}}{=} f(x,y) = ax+b$, where a and b are constants. This function maps $\mathbb{R}^2$ to $\mathbb{R}$, but its output is independent of the variable y, in other words $\Pr(z|x,y) = \Pr(z|x)$. CKWIK-KR can solve this regression with a sample complexity that is exponential in the dimensionality (Thm. 18). However, if the dependency structure in f is known (that is, the independence of z on y), the algorithm can transform this regression problem into another one that maps $\mathbb{R}$ to $\mathbb{R}$. Consequently, any training data point $((x,y),z)$ can be converted to (x,z) and fed into the new regressor. It is obvious that this space reduction does not incur any

information loss as y does not have any effect on the value of z. After this reduction, the same regressor can be used with a new sample complexity that is dependent on the new dimensionality. In general, depending on how many variables are eliminated from the regression, the sample complexity can be improved exponentially. The process of mapping the regression problem into another one with a smaller input space is also referred to as *dimension reduction* in regression or *feature selection* in supervised-learning literature (Geladi, 1986; Fukumizu et al., 2009; Cook, 2007).

The main idea of FF-Rmax is to use the same technique described above to learn the transition function faster when the dependency structures are known *a priori*. Since the output components are uncorrelated, $T(s'|s,a)$ can be factored into $T(s'(i)|s,a)$ for all i's. Each of these individual univariate regressions in turn can be transformed into a simpler regression based on the dependency structure of $s'(i)$ as we saw earlier.

To make things more concrete, let us define *factored kernel regression* as follows: Suppose the dependency graph $G_a = (X, Y, E)$ is available for each action a. Let $\text{dep}_{a,j}$ be the set of nodes in X that are adjacent to $y(j)$ in G_a:

$$\text{dep}_{a,j} = \{x(i)|\ x(i) \in X,\ (x(i), y(j)) \in E,\ y(j) \in Y,\ G_a = (X, Y, E)\}. \tag{4.28}$$

Also, let $(s \perp \text{dep}_{a,j})$ be the projection of vector s into the space spanned by the components of $\text{dep}_{a,j}$. Assuming that the output components are uncorrelated, a factored kernel regressor (FKR) breaks up the estimation of $T(.|.,a)$ into $|\mathcal{S}|$ univariate kernel regressors $\mathcal{F}_a^j$, each estimating the value of one of the components of the next state. With any training data (s_t, a_t, s_{t+1}), each $\mathcal{F}_{a_t}^j$ is updated by $(s_t \perp \text{dep}_{a_t,j}, s_{t+1}(j))$. The response for a query (s_t, a_t, δ_t) is constructed using the response of $\mathcal{F}_{a_t}^j$'s for the query point $(s_t \perp \text{dep}_{a_t,j}, \frac{\delta_t}{|\mathcal{S}|})$. Let $(\hat{y}_j, \epsilon_t^j)$ be the output of $\mathcal{F}_a^j$. The final output $\hat{y}$ is constructed by concatenating all the $\hat{y}_j$'s. The accuracy of the estimate is computed using:

$$\epsilon_t = \sum_{i=1}^{|\mathcal{S}|} \epsilon_t^i. \tag{4.29}$$

When each $\mathcal{F}^j$ is a CKWIK-KR, we call the factored state-space learner CKWIK-FKR.

The pseudo-code is available in Algorithm 7.

Algorithm 7 CKWIK-FKR, a factored learner of transition functions in the CKWIK framework.

Inputs: Kernel function k, Lipschitz constant C_T, and G_a's .
Initialize dataset D_a for all a's.
Initialize all $\mathcal{F}^j$'s with (k, C_T).
for all timesteps $t = 1, 2, \cdots$ **do**
  Observe (s_t, a_t).
  Query $\mathcal{F}^j$ with $(s_t \perp \text{dep}_{a_t,j}, a_t)$ using the data $(D_{a_t} \perp \text{dep}_{a_t,j})$ for all j's.
  Construct $\hat{y}$ by concatenating all $\hat{y}_j$'s.
  Compute $\epsilon(s_t, a_t)$ according to Eqn. 4.29.
  Observe s_{t+1} as an i.i.d. sample of $T(s_t, a_t)$.
  Add (s_t, s_{t+1}) to D_{a_t}.
end for

FF-Rmax uses CKWIK-FKR as the regressor for learning an internal model of the world. Apart from this regressor, this algorithm is the same as CF-Rmax developed earlier. Algorithm 8 contains the pseudo-code of FF-Rmax.

Algorithm 8 FF-Rmax, a factored model-based learner for continuous MDPs.

Inputs: Accuracy ϵ_T, confidence δ_T, and dependency graphs G_a for all a's.
Initialize CKWIK-FKR regressor $\mathcal{F}$ using G_a's.
for all timesteps $t = 1, 2, \cdots$ **do**
  Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
  Update $\mathcal{F}$ using (s_t, a_t, s_{t+1}).
  Update internal MDP $\hat{M} = \langle \mathcal{S} + s^f, \mathcal{A}, \hat{T}', R', \gamma \rangle$,
  where $\hat{T}(., s) = \mathcal{F}_a(., \delta_T)$ and $\hat{T}'$ is defined as:

$$\hat{T}'(s'|s,a) = \begin{cases} (1 - \psi(s,a)) & \text{if } s' = s^f \\ \psi(s,a)\hat{T}(s'|s,a) & \text{otherwise.} \end{cases} \tag{4.30}$$

end for

4.2.3 Discussion

This algorithm is closely related to factored E^3 (Kearns and Koller, 1999) and factored-Rmax (Guestrin et al., 2002; Strehl, 2007) for finite MDPs. The lower bound for planning in finite MDPs has a polynomial dependency on the number of available states. However, the state space of many complex environments is presented to the learner in terms of a combination of factors (or state variables), each taking up value from a finite domain. This set of environments, which are modeled by the so-called *DBN-MDPs*,

has been widely used in many real-life applications (Boutilier et al., 1996). But, it also brings challenges to the learner as the total number of states in such MDPs grows exponentially in the number of factors. The two aforementioned algorithms are variations of E^3 and Rmax algorithms. They take in the graphical structure of the dependency between the factors as input and use it to learn the transition function faster. The sample complexity of both algorithms are proved to have an exponential dependency only on the size of the maximum dependency set of any factor instead of the total number of factors. FF-Rmax implements the same machinery used in these algorithms to achieve dramatically better sample complexity in continuous spaces.

The factorization of the state space is even more natural in continuous spaces. Finite MDPs are defined using a nonempty set of states, that is the states are usually indexed by natural numbers. Unless an explicit factorization is believed to exist for the problem at hand, it is not possible to provide a graphical dependency structure for it. Therefore, factored state-space learners in finite spaces become relevant only if the underlying domain admits to the factorization.

The state space of continuous MDPs, on the other hand, are defined as a closed subset of $\mathbb{R}^n$, which is by definition in a factored form. Therefore, any continuous MDP has a corresponding dependency graph in principle. Of course, this fact does not mean FF-Rmax always perform exponentially better than CF-Rmax. But, we argued that in the cases where the dependency structure is a bipartite graph and the maximum dependency set of nodes is smaller than the number of state dimensions, the algorithm can use this prior information to dramatically decrease its sample complexity.

The two above conditions that allow FF-Rmax to gain performance improvement over CF-Rmax are not very limiting. The conditional independence of state variables at each timestep, which results in a bipartite graph, naturally occurs in many real-life systems, especially in those based on physical domains. For example, it is easy to show that all the deterministic domains belong to this class, as well as all the stochastic domains with a noise term selected i.i.d. from each dimension.

The second assumption is perhaps more restricting because it requires that only a

subset of state variables at time t be necessary for estimating the value of each factor at time $t+1$. But, we can get insight into the generality of this constraint by looking more closely at the process of constructing the state space of a complex domain. From the perspective of an application designer, the factors that constitute a state usually come from different sources, like different sensory information on a robot or different properties of an object in an environment. It is natural in many complex domains that some of the factors be independent of each other. For example, a controller for a car might need information about the position of the car as well as the amount of gas in the tank, but predicting the amount of gas is independent of the position of the car. Another set of environments that makes a good example of factored-state MDP are the ones that track information about multiple objects. For example, if we maintain information about multiple cars in an environment, chances are that predicting the properties of one car is independent of predicting those of others.

Perhaps the most limiting part of the algorithm that may affect its usability in practice is that the user needs to know the dependency structure of the state variables beforehand. The algorithm can still use partial information if a complete dependency relationship cannot be established. In this case, the more thorough the user's information is about the independence structure, the faster the learning becomes. To formalize this idea, let $G1 \geq G2$ on graphs denote that edge set $E1$ is a superset of $E2$. The algorithm always converges to a near-optimal policy if we have $G_a \geq G_a^*$ for all a's, where G_a^* is the true dependency graph of action a and G_a is the graph that was given to the learner. In other words, as long as all the true dependencies are present in the input graph, the algorithm will indeed converge to a near-optimal solution. Of course, the number of mistakes will increase exponentially as the size of the maximum dependency set increases, and in the degenerate case, if all G_a's are complete bipartite graphs, the sample complexity of the algorithm becomes the same as CF-Rmax. The algorithm might never find a near-optimal solution if any dependency edge is omitted from the input graphs. Hence, a good practice in the design stage is to be cautious about the dependencies and if one is not sure whether two variables are dependent on each other or not, it is important to assume they are.

4.2.4 Analysis

According to Theorem 14, the sample complexity of FF-Rmax is directly dependent on the CKWIK sample complexity of the factored transition function learner. So, we start by analyzing the sample complexity of Algorithm 7 in the CKWIK framework. The next theorem states that the sample complexity of CKWIK-FKR is exponentially dependent on k instead of $|\mathcal{S}|$, where k is the maximum size of the dependency sets.

Proposition 19. *Let M be a continuous state-space MDP according to Assumption 2. Assume that a set of dependency graphs G_a is provided. Let k be the maximum size of the dependency sets. The CKWIK-FKR algorithm CKWIK-learns the transition function of M with the following bound:*

$$B(\epsilon, \delta, \mathcal{S}, \mathcal{A}) = \mathcal{O}\left(\sqrt{\frac{\ln(1/\delta)}{|\mathcal{S}|}}|\mathcal{A}|^2\left(\frac{|\mathcal{S}|^2}{\epsilon}\right)^{k/2}\right).$$

Proof. The proof is done in two steps: First, we show that the output of the algorithm is indeed ϵ_t-close to the true function with probability at least $(1-\delta_t)$, and second, we bound the number of times ϵ_t is bigger than ϵ.

To show that the first condition holds, consider two points x_1 and x_2 in $\mathbb{R}^n$. If we have $(x_1(i) - x_2(i))^2 \leq \epsilon_i$ with probability at least $(1-\delta/n)$ for all i's, we also have:

$$\| x_1 - x_2 \|_2^2 \leq \sum_{i=1}^{n} \epsilon_i, \qquad \text{w.p. } (1-\delta). \tag{4.31}$$

To see why, let A_i be the event that $(x_1(i) - x_2(i))^2 > \epsilon_i$, which according to the assumption has probability at most $\Pr(A_i) = \delta/n$. Apply the Union bound on the set of events A_i. The probability that none of these events happens is bounded by:

$$\Pr\left(\bigcap_{i=1}^{n} \overline{A_i}\right) \geq 1 - \sum_{i=1}^{n} \frac{\delta}{n} = 1 - \delta.$$

This quantity is the probability that $(x_1(i) - x_2(i))^2 \leq \epsilon_i$ holds for all i's. In this

case, the Euclidian distance between the two points is computed as:

$$\begin{aligned} \| x_1 - x_2 \|_2^2 &= \sum_{i=1}^{n} (x_1(i) - x_2(i))^2 \\ &\leq \sum_{i=1}^{n} \epsilon_i, \end{aligned}$$

which is exactly the accuracy parameter computed by CKWIK-FKR in Equation 4.29.

The second part is proved by observing that $\epsilon_t > \epsilon$ iff $\epsilon_t^i > \epsilon/n$ for at least one ϵ_t^i. But, the number of times each ϵ_t^i is larger than ϵ/n when δ/n is given is bounded by $B_{\text{CKWIK-KR}}(\epsilon/n, \delta/n, k_i, |\mathcal{A}|)$, where $k_i = \max_a \text{dep}_{a,i}$. Putting it all together, we get:

$$B(\epsilon, \delta, \mathcal{S}, \mathcal{A}) = \mathcal{O}\left(\sqrt{\frac{\ln(1/\delta)}{|\mathcal{S}|}} |\mathcal{A}|^2 \left(\frac{|\mathcal{S}|^2}{\epsilon}\right)^{k/2}\right).$$

□

The sample complexity of FF-Rmax can be computed based on the result of Theorem 14 and the CKWIK sample complexity we just derived. Putting these results together, we get:

$$\mathcal{O}\left(\sqrt{\frac{\ln(1/\delta)}{|\mathcal{S}|}} |\mathcal{A}|^2 \left(\frac{V_{\max}|\mathcal{S}|^2}{\epsilon(1-\gamma)}\right)^{k/2} + \ln\frac{1}{\delta}\right),$$

which is exponentially dependent on the maximum size of the dependency set. This result contrasts the exponential dependence on the dimensionality of the state space in the CF-Rmax algorithm.

4.2.5 Experimental Results

This section provides empirical results for FF-Rmax. Several experiments were performed to illustrate the benefits of factored learning.

The first environment in which we tested FF-Rmax was an extension of Puddleworld called n-Puddleworld. In this domain, the 2-dimensional box of the original domain was replaced by an n-dimensional unit hypercube. The set of actions was also

extended to include two actions per dimension for moving the agent along that dimension, resulting in a total of $2n$ actions. The puddles and the goal region were projected into the extra dimensions (that is, only the first two state variables were used to decide whether the agent was in a puddle or the goal region). With this setup, it is straightforward to show that the value functions of all the n-PUDDLEWORLDs are the same. The technical details of this domain are presented in Appendix C.2.1.

The n-PUDDLEWORLD domain provides a natural way to factor the transition function. Independent of the state space dimensionality, the dependency sets are always $\text{dep}_{a,j} = \{x_j\}$ because the value of a state variable is independent of other variables (due to the grid-based navigation).

The first experiment compared the performance of CF-Rmax with FF-Rmax in 4-PUDDLEWORLD. The experiment was performed in 50 episodes, each having a cap of 300 steps. FF-Rmax was given the dependency graphs of the environment. Other parameters were selected as in Section 4.1.3. Figure 4.7 shows the result of this experiment. As was expected, CF-Rmax was not able to learn the optimal policy in the short amount of time given to it because it had to explore a 4-dimensional space. It was only successful in learning to avoid the puddles. However, FF-Rmax used the independence sets to eliminate 3 out of 4 input dimensions for estimating each output component. Therefore, its performance hardly changed from 2-PUDDLEWORLD.

To put this result into perspective, we tried the same experiment in n-PUDDLEWORLD for $n = 2, 3, 4, 5$. For each domain, the average collected reward per episode was reported. The x-axis in Figure 4.8 shows the dimensionality of the domain, while the y-axis shows the average collected reward per episode.

Although CF-Rmax is a very powerful algorithm, its performance dropped quickly because the number of collected samples in 50 episodes was just not enough to cover higher than a 3-dimensional space. On the other hand, the performance of FF-Rmax was not directly dependent on the dimensionality of the space. Instead, it was dependent on the maximum size of the dependency set, which was always 1 independent of n. For that reason, the algorithm performance did not suffer much with the introduction of

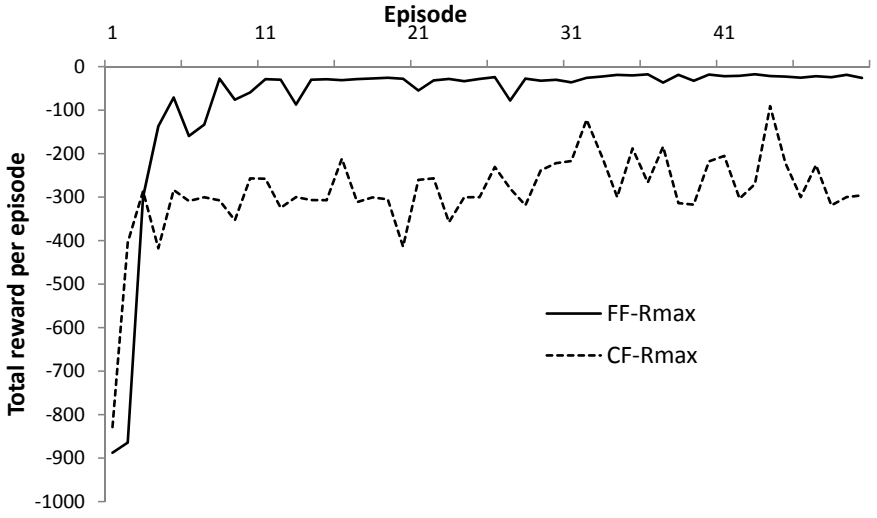


Figure 4.7: Performance of FF-Rmax and CF-Rmax in 4-PUDDLEWORLD.

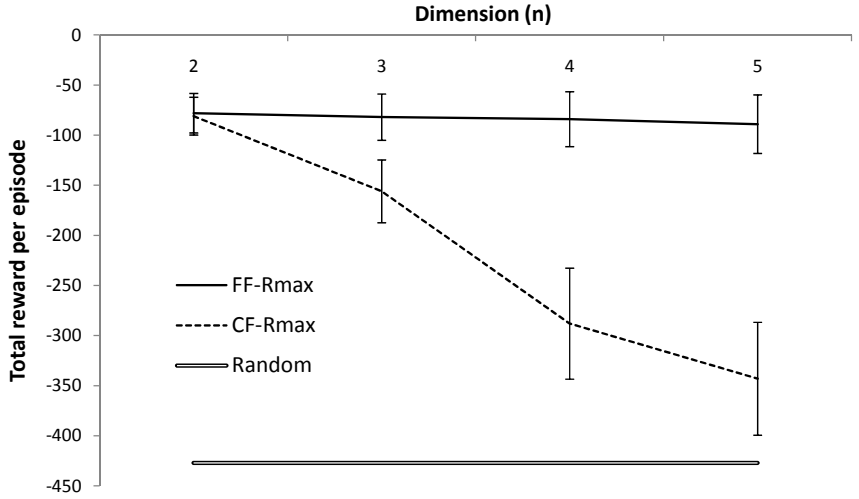
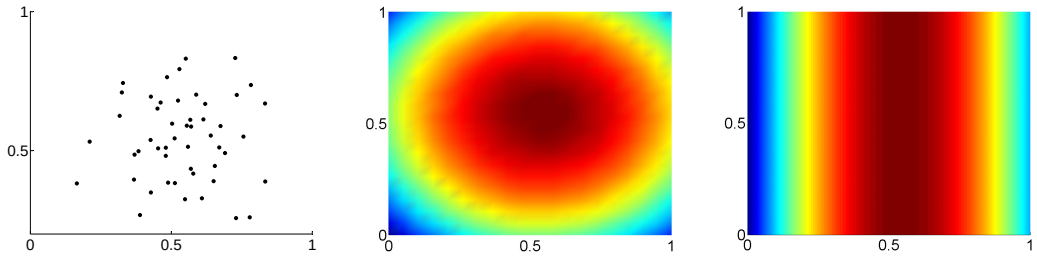


Figure 4.8: Performance of FF-Rmax and CF-Rmax in n-PUDDLEWORLD for different values of n.

more dimensions in the environment.

The gap between the two lines signifies the effectiveness of factored learning. In particular, the difference between the performance of the two algorithms in 4-PUDDLEWORLD is close to the difference between the optimal and random policies.

Similar to the experiment for CF-Rmax, we examined the knownness function to get more insight into why the performance of FF-Rmax was much better. Fifty points were generated from the state space—shown in Figure 4.9(a)—with the target function $f(x, y) = x + 0.05 + \mathcal{N}(0, 0.01)$, which is part of the transition function of PUDDLEWORLD. The FF-Rmax algorithm was told that the output is not dependent on y. The heat-map graph in Figure 4.9(b) shows the knownness computed in the entire state space using CF-Rmax and Figure 4.9(c) shows the knownness of FF-Rmax. Dark red signifies a completely known state and dark blue means a completely unknown state[2]. While the number of training points in the top center part of the state space was not very high, FF-Rmax produced a very high knownness value because it knew the y dimension is irrelevant to the regression problem at hand. So, in a sense, it allowed for a much broader generalization along the y-axis.



(a) Training points (b) CF-Rmax (c) FF-Rmax

Figure 4.9: Comparison of the knownness function in CF-Rmax and FF-Rmax using data from PUDDLEWORLD. The dark red in the heat-map graphs signifies a completely known state and dark blue is completely unknown.

The next experiment the algorithms were tested on was a real robotic task. This environment was inspired by PUDDLEWORLD, but implemented a navigation task using

[2]For a print in black/white, the dark red area will be dark gray (the area in the middle), and the areas with lower knownness values are represented with lighter grays. The completely unknown areas are black around the edges of the pictures, which correspond to dark blue.

an Aibo Robot—which is a four-legged dog robot by SONY. In this task, an Aibo was placed inside a closed field and had to navigate to a tiny goal region. To create a more interesting version of the puddles, a moving object called *Bumbleball* was introduced into the environment, and the robot had to avoid it. Bumbleball is a motorized toy ball that randomly moves around. The state of the system consisted of five variables: position and orientation of the robot and position of the ball. The robot was equipped with 6 actions for moving forward and backward, turning right and left, and strafing right and left. This set of actions was more restrictive than the one in PUDDLEWORLD. Since the robot was not holonomic, it had to mix turns and walks to move in an intended direction. Technical details of this environment can be found in Appendix C.4.

This experiment was selected because it provided an example of a situation where factorization could be performed easily in a real-life domain. Given the two objects in the world (*i.e.*, the robot and the ball), it was easy to observe that the movement of the robot and the ball were almost independent of each other. Therefore, a set of dependency graphs that reflected this independence was easily created without having an in-depth knowledge of how the robot works.

The goal of this experiment was to learn as much as possible with only 3000 steps. Two algorithms were tested in this domain: CF-Rmax and FF-Rmax. Both algorithms used the same set of parameter values. The kernel width was hand-tuned on a simulator to 0.8 and planning was redone every 20 steps. Finally, the whole experiment was repeated 3 times.

Table 4.1 summarizes the result of this experiment. To get a sense of what the numbers mean in the table, we also included the results of a randomly moving agent. As you can see, CF-Rmax was not able to perform much better than the random agent. Again, FF-Rmax learned much faster in this domain because it had prior access to the dependency graphs of the state variables. In particular, it quickly learned to avoid the ball even in the early stages of learning. This experiment showed how factorization can be used in applications with real data to significantly improve upon state-of-the-art algorithms.

Algorithms:	Random	CF-Rmax	FF-Rmax
Total cumulative reward:	-24269	-21461	-3917.0
Number of collisions:	533	463.7	38
Percent finished episodes:	8.6%	13%	81.3%

Table 4.1: Performance of three algorithms in the Bumbleball domain.

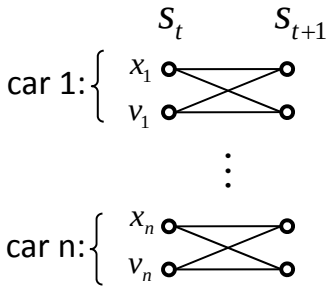


Figure 4.10: The true dependency graph of n-MOUNTAINCAR. This structure was the same for all actions.

The last experiment we considered for FF-Rmax investigated the effect of having imperfect dependency graphs. For this experiment, we selected another domain called n-MOUNTAINCAR as it had a more complicated dependency structure than n-PUDDLEWORLD. This new environment extended MOUNTAINCAR in the same way n-PUDDLEWORLD extended PUDDLEWORLD. In an instance of n-MOUNTAINCAR, n cars were placed in n parallel worlds with the same three available actions. At the beginning of each experiment, the effect of different actions on each car was randomly permuted. For example, action right might have moved the first car to the right while moving the second car to the left. The goal of the experiment was to drive the first car to the top of the hill. Hence, similar to n-PUDDLEWORLD, the value function of n-MOUNTAINCAR was the same for all values of n.

The dependency graphs of this environment (which are the same for all the actions) are presented in Figure 4.10. The next state of each car is dependent only on the previous state of that particular car. We called this set of graphs G_{true}.

In addition to the true dependency graph, two other graphs were also given to FF-Rmax to illustrate what happens when the graphs are not accurate. The first set of

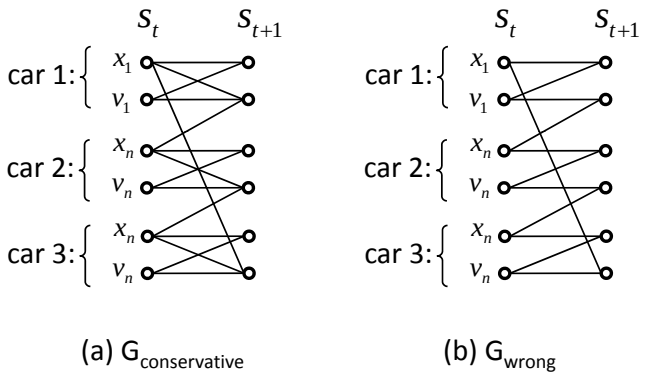


Figure 4.11: Two alternative dependency graphs for 3-MOUNTAINCAR. The one in (a) is a conservative one because its edge set is a superset of the true edge set. The one in (b) is completely wrong because it misses some important dependencies.

graphs were generated from G_{true} by making the velocity of each car dependent on the position of half of the cars. We called these graphs $G_{\text{conservative}}$ as they were a superset of the true graphs. The second set was constructed from G_{true} by connecting the velocity of each car to the position of the next car instead of its own. We called these G_{wrong} as they described a wrong set of dependencies. Figure 4.11 shows these two dependency sets for 3-MOUNTAINCAR.

Figure 4.12 shows the results of executing CF-Rmax as well as FF-Rmax on n-MOUNTAINCAR for different values of n when the three aforementioned graphs were provided. FF-Rmax$_{\text{true}}$ did not suffer much from adding more dimensions to the environment because it was only dependent on the maximum size of the dependency set, which was 2 in all the domains. FF-Rmax$_{\text{conservative}}$ did not scale well as its maximum dependency set was dependent on n. Therefore, as the dimensionality grew larger, it needed exponentially more samples to learn. But, it still performed better than CF-Rmax, which treats all the variables as dependent on each other. FF-Rmax$_{\text{wrong}}$ was not able to solve any of the environments as it did not include the variables that were necessary to estimate the transition function. This experiment demonstrated the fact that it is important to make sure none of the true dependencies between the variables are missing from G_a's. Although being conservative when the true dependencies are not known might hurt the sample complexity of the algorithm, removing an important

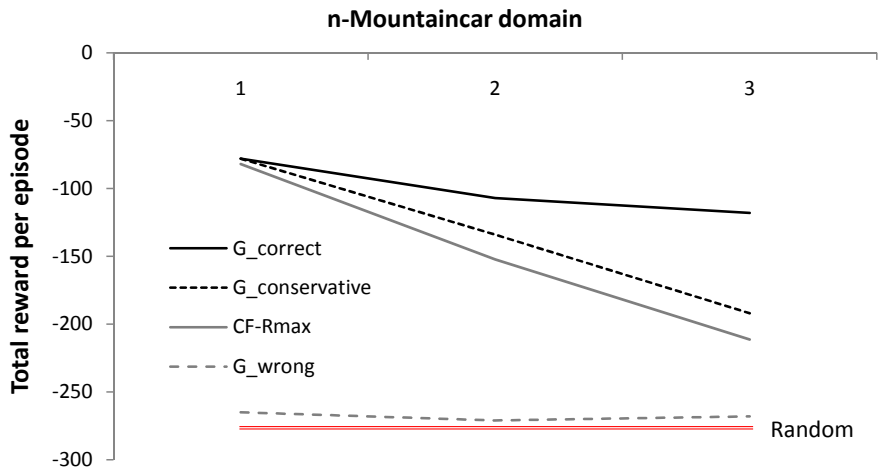


Figure 4.12: The result of CF-Rmax and three versions of FF-Rmax on n-MOUNTAINCAR.

edge by mistake can completely ruin the convergence behavior of the algorithm.

Chapter 5

Automatic Discovery of Relevant Features

Chapter 3 introduced a novel PAC-MDP algorithm for behavior learning in general continuous MDPs. While this method was the first practical algorithm with theoretical guarantees in general continuous MDPs, Section 4.2 showed that the problem of the "curse of dimensionality" still looms large in complex domains, and unless we find a way to take out the exponential dependency on the dimensionality of the state space, our algorithms cannot hope to scale well to real-life complex domains.

The last chapter also introduced one way to get rid of the exponential dependency. It showed how prior knowledge about the dependency structure between the state variables can be used to exponentially decrease the sample complexity of CF-Rmax in continuous spaces. However, a big challenge still remained for using FF-Rmax in practice, and that was obtaining the dependency structures beforehand.

From a system designer's perspective, unless extensive knowledge of the system can be obtained from an expert, only the most obvious independencies can be established. So, in the more general cases where the environment is not fully known to the designer in advance, FF-Rmax does not quite achieve the goal of dealing with the curse of dimensionality.

This chapter investigates some other machine-learning techniques for dealing with high-dimensional data and the curse of dimensionality, and introduces a way to incorporate them into a reinforcement-learning algorithm. The goal of the proposed algorithm is to: (1) automatically discover the relevant dimensions of the data without the need for the user to provide them *a priori*, and (2) incorporate that knowledge into exploration to reduce the sample complexity of learning.

5.1 Background

A classic approach in machine learning for dealing with high-dimensional spaces is to explicitly use a simpler representation of data by projecting it to lower-dimensional spaces—known as *dimension reduction*. In fact, the history of using dimension reduction in machine learning goes back several decades, with a large number of success stories (Jolliffe, 1986). Methods such as principal component analysis (PCA) have long been used in various scientific disciplines as a preprocessing step for handling high-dimensional data, and are now considered standard for dealing with complex data. More recently however, the applicability of these methods has been extended a great deal, thanks to advances in the field of statistical learning theory. Robust dimension reduction in regression using nonlinear kernel transformation functions is an example of such an advance (Weinberger and Tesauro, 2007; Fukumizu et al., 2009).

The idea of dimension reduction has also been studied in the reinforcement-learning community. For example, Kolter and Ng (2009) and Parr et al. (2008) learned the relevant basis functions (for example, from a large pool), when approximating the value function in the context of least-squares temporal difference learning (LSTD). Discarding irrelevant basis functions reduces the number of free parameters and provides a more overfitting-resistant estimation. Some research makes an even tighter connection to the dimension-reduction literature by directly using some of the existing techniques and tailoring them to the RL framework. For example, Smart (2004) used manifold learning for low-dimensional representation of the value function, and Mahadevan (2009) proposed a framework using Laplacian operators for representing and controlling MDPs.

The main contribution of this dissertation has been providing data-efficient exploration techniques. In particular, this chapter provides a method for using dimension reduction to attack the exploration problem in continuous state-space problems, and because of the focus on exploration, this research is orthogonal to existing dimension-reduction work in RL, which has emphasized on either statistical efficiency of learning or exploitation.

In what follows, dimension reduction is first studied in a broader context of regression. Then, an RL algorithm is introduced that uses dimension reduction for faster learning.

5.2 Dimension Reduction in Regression

The task of dimension reduction in regression (DRR) is to find a low-dimensional representation of the input space such that the transformed data can predict the output independent of the original covariates. To be more precise, let us define the data as a set of observations of the form (x, y), where $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^l$, and the regression as the problem of estimating the conditional probability density of y given x.

The task of DRR is to find a *transformation function* $\Phi : \mathbb{R}^m \to \mathbb{R}^r$, with $r < m$, such that x and y are conditionally independent given the transformation $\Phi(x)$ (Fukumizu et al., 2009). For convenience, we use matrix notation $\mathbf{X}$ to denote the row-wise concatenation of x_i^{T} for $i = 1 \ldots n$. We call the regression *univariate* whenever $l = 1$, and *multivariate* if l is greater than 1.

Here, DRR is investigated through *kernel regression*, which is a nonparametric and nonlinear technique (as discussed in Chapter 4). We show how computing a linear transformation function can be translated into learning a customized metric for kernel regression, and provide an efficient way of doing so. We also briefly overview some other techniques for performing DRR from the literature that might be useful for RL algorithms.

5.2.1 Kernel Regression and Metric Learning

Chapter 4 introduced CKWIK-KR, which performed kernel regression in the CKWIK framework. Here, a special type of kernel regression is developed to perform dimension reduction in regression. This work is built on the work of Weinberger and Tesauro (2007) on univariate kernel-metric learning.

To reiterate, kernel regression computes an estimate using the following equation:

$$\widehat{y^*} \stackrel{\text{def}}{=} \hat{f}(x^*) = \frac{\sum_{i=1}^{n} k(x^*, x_i) y_i}{\sum_{i=1}^{n} k(x^*, x_i)}, \tag{5.1}$$

where $k(.,.) \geq 0$ is the *kernel* function. Here, we focus on the *Gaussian kernel*:

$$k(x_i, x_j) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{d^2(x_i, x_j)}{\sigma^2}}, \tag{5.2}$$

where d is the distance metric and σ is a constant that determines how fast the kernel decays with respect to d.

Metric learning refers to the tuning of the distance function in the kernel so as to minimize the regression error. For example, if one of the dimensions of the input space is irrelevant to the true function f, a distance metric that is oblivious to that dimension is expected to achieve better results.

The example in Figure 5.1 demonstrates this relationship. Suppose we need to estimate the value of the function $f(x, y) = ax + b$ at the query point q_1 using the value of the function at the training points $t_1, \ldots, t_{12}$. The ellipsoids in both images are equidistant contours of two different metrics: The one on the left is the Euclidian distance and the one on the right is a metric that is stretched along the y dimension. It is easy to see that the kernel regressor that uses the metric on the right side will have a better estimation at the query point because it puts more weights on the values of $t_1, \ldots, t_6$ than the points $t_7, \ldots, t_{12}$. Since f is only dependent on x and not y, all the points $t_1, \ldots, t_6$ and q_1 have the same output value. Therefore, putting more weight on $t_1, \ldots, t_6$ results in a better prediction of function at a_1 by making better use of the data.

To tune the metric, we must first select a differentiable distance function with respect to some parameter θ. This setup allows us to perform gradient descent to find the optimal value. More precisely, let $\mathcal{L}$ be a loss function defined by the cumulative leave-one-out error of the training set: $\mathcal{L} = \sum_i \parallel y_i - \hat{y}_i \parallel_2^2$. The metric-learning algorithm updates θ iteratively using the gradient descent rule: $\Delta\theta = -\alpha \frac{\partial \mathcal{L}}{\partial \theta}$, where

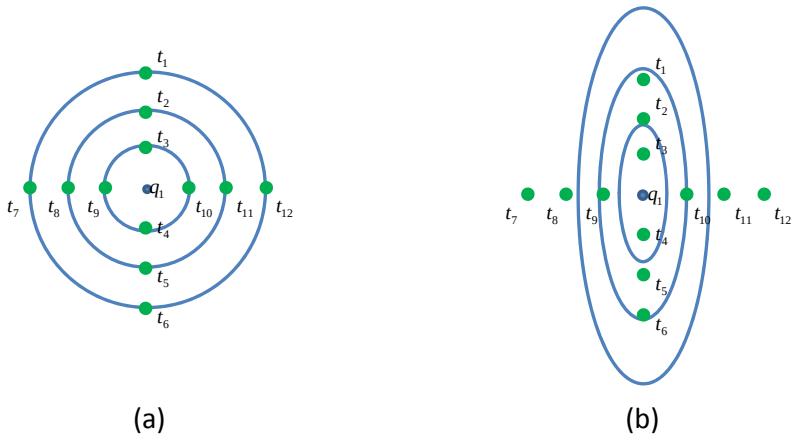


Figure 5.1: A simple example of a set of training points and a single query point in 2D along with two different metrics: (a) Euclidian, (b) a customized metric based on Euclidian that is stretched along the Y axis.

α is a learning rate (Nouri and Littman, 2010). Any differentiable distance function works in this procedure. Here, we use the *Mahalanobis* metric, which can be written as:

$$d_m^2(x_i, x_j) = (x_i - x_j)^{\mathrm{T}} \mathbf{M} (x_i - x_j), \tag{5.3}$$

where $\mathbf{M}$ can be any symmetric positive semi-definite matrix. Basically, the matrix $\mathbf{M}$ provides a way to customize the Euclidian metric by skewing and rotating the equidistant contours in any direction; Figure 5.1 shows one such transformation. In the degenerate case, setting $\mathbf{M} = \mathbf{I}$ allows Mahalanobis metric to capture the Euclidian metric. Unfortunately, it is not easy to learn the matrix $\mathbf{M}$ directly. Since the matrix has to be positive semi-definite, the learning becomes a nonlinear optimization, which is computationally hard. But, we can use the decomposition $\mathbf{M} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$ to learn $\mathbf{A}$ instead of $\mathbf{M}$. The benefit of this decomposition is twofold: First, the matrix $\mathbf{A}$ is totally unconstrained (Strang, 1980) and optimization can be done via regular gradient descent methods; second, matrix $\mathbf{A}$ provides a mapping between dimension reduction and metric learning as we will see shortly. Given the decomposition, we can write the Mahalanobis metric as:

$$d_m^2(x_i, x_j) = \| \mathbf{A}(x_i - x_j) \|_2^2 \, . \tag{5.4}$$

We use subscripts on k to indicate what metric is being used inside the kernel. For example, k_u denotes the kernel function with the Euclidian metric and k_m denotes Mahalanobis. The following result shows how to compute the gradient of the loss function when the Mahalanobis metric with a Gaussian kernel is used.

Proposition 20. *The gradient of $\mathcal{L}$ when the Mahalanobis metric is used in the kernel function is:*

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = 4 \sum_i (\hat{y}_i - y_i) \frac{\sum_j (\hat{y}_i - y_j) \mathbf{A} k_m(x_i, x_j; \mathbf{A}) x_{ij} x_{ij}^{\mathrm{T}}}{\sum_{j \neq i} k_m(x_i, x_j; \mathbf{A})}. \tag{5.5}$$

Proof. Let x_{ij} be $(x_i - x_j)$ and k_{ij} be $k_m(x_{ij})$. The gradient can be expanded as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial k} \cdot \frac{\partial k}{\partial d^2} \cdot \frac{\partial d^2}{\partial \mathbf{A}}.$$

To compute $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$, we can start backward and compute $\frac{\partial d^2}{\partial \mathbf{A}}$, $\frac{\partial k}{\partial \mathbf{A}}$, and $\frac{\partial \hat{y}}{\partial \mathbf{A}}$ in turn.

$$\frac{\partial d^2}{\partial \mathbf{A}} = 2 \mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}$$

Therefore, the gradient of the kernel function is:

$$\begin{aligned} \frac{\partial k_m}{\partial \mathbf{A}} &= \frac{\partial}{\partial \mathbf{A}} \exp(-d_m^2(x_{ij})) \\ &= -2(\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij}. \end{aligned}$$

The gradient of $\hat{y}_i$ is:

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial \mathbf{A}} &= \frac{\partial}{\partial \mathbf{A}} \frac{\sum_j k_{ij} y_j}{\sum_j k_{ij}} \\
&= \frac{-2\left[\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij} y_j\right] \cdot \sum_j k_{ij}}{\left(\sum_j k_{ij}\right)^2} \\
&\qquad - \frac{-2\left[\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij}\right] \cdot \sum_j k_{ij} y_j}{\left(\sum_j k_{ij}\right)^2} \\
&= \frac{-2\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij} y_j}{\sum_j k_{ij}} - \frac{-2\left[\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij}\right] \cdot \hat{y}_i}{\sum_j k_{ij}} \\
&= \frac{-2\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij} (y_j - \hat{y}_i)}{\sum_j k_{ij}}.
\end{aligned}$$

Given these results, the gradient of $\mathcal{L}$ is:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{A}} &= \frac{\partial}{\partial \mathbf{A}} \sum_i (y_i - \hat{y}_i)^{\mathrm{T}} (y_i - \hat{y}_i) \\
&= 4 \sum_i (\hat{y}_i - y_i) \frac{\sum_j (\mathbf{A} x_{ij} x_{ij}^{\mathrm{T}}) k_{ij} (\hat{y}_i - y_j)}{\sum_j k_{ij}}. \qquad (5.6)
\end{aligned}$$

□

As with all other local search methods, this process of learning $\mathbf{A}$ is dependent on its initial value and might get stuck in local minima. Multiple starting points for $\mathbf{A}$ can be used to resolve this issue, although we did not find it necessary to do so in our implementations. Initial values of $\mathbf{A}$ could be $\mathbf{I}$ or a totally random matrix. When used in the RL setting, where the dimension reduction is applied repetitively, previous values of $\mathbf{A}$ are also good choices of starting points.

A challenging problem in kernel regression in general is to make sure the different dimensions of the input space have the same meaning, otherwise using the Euclidian metric does not make sense. For example, if one of the input dimensions is the weight of an object measured in kg, and another dimension is its velocity in m/s, these two are not directly comparable with each other unless they are somehow intelligently normalized.

This normalization is not always easy. The user has to first decide the right unit for each dimension to make the Euclidian distance makes sense. For example, given that the unit used in the velocity is fixed, representing weight in terms of *grams, pounds,* or *kilograms* creates very different distances. Furthermore, the user has to make a justifiable numerical connection between possibly two totally unrelated numbers, which is not always easy.

Since the optimized $\mathbf{A}$ is a completely data-driven distance metric, we do not have to normalize the data points anymore. The self-adjusting $\mathbf{A}$ automatically finds a normalization that best fits the data. In fact, the parameter σ and the leading coefficient in the kernel function can also be omitted, as they are also captured inside $\mathbf{A}$. Therefore, we can simply replace the Gaussian kernel in Equation 5.2 with the following one:

$$k(x_i, x_j) = e^{-d^2(x_i, x_j)}. \tag{5.7}$$

Tuning the constants in Equation 5.2, which are eliminated in the above equation, is one of the crucial steps in regular kernel regression. For example, the parameter σ, which is known as *kernel width* or *kernel radius*, acts as a smoothing parameter. High values of σ create kernel functions that are more widely-spread, and smaller values create more spiky functions. Fine-tuning the value of this parameter involves having a good understanding of the problem at hand, and its value greatly affects the performance of the regression. As an example, Figure 5.2 compares two values of σ for a 1D regression problem using a set of training points. Part (a) shows the shape of the kernel function when $\sigma = 0.1$ and (c) shows the same kernel with $\sigma = 1.0$. As you can see, the larger values of σ produce flatter functions. The slim dotted lines in parts (b) and (d) are both the same function $y = \cos(x)$. The regressors have access to training data from this function, which are shown in both images by the (+) signs. The solid line in part (b) is the outcome of kernel regression when the kernel function in (a) is used, and part (d) shows the same kernel regression when the kernel in (c) is used. The output in (d) is much smoother than (b) because it allows for more generalization to happen across the space. Because of the smoothing, (b) provides a regressor with less variance, but

higher bias. Unfortunately, deciding on the right value of σ that best balances variance and bias is a challenging task and often requires an optimization of its own (Sheather and Jones, 1991). The approach in this chapter for self-adjusting $\mathbf{A}$ automatically eliminates the need to pick the right bandwidth.

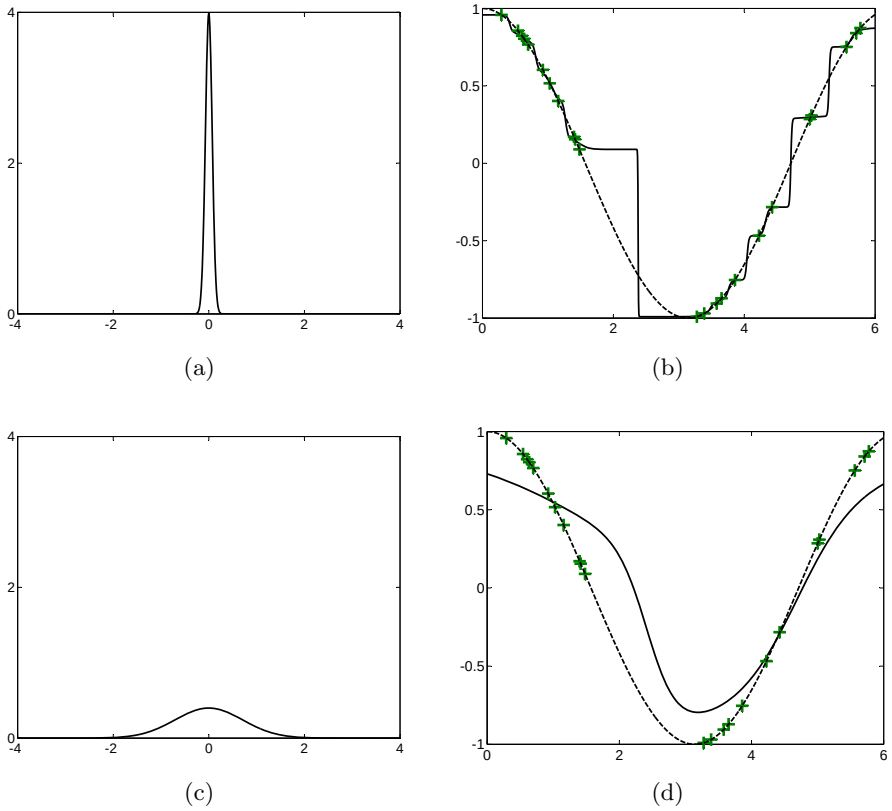


Figure 5.2: Two kernel functions with σ equal to 0.1 and 1.0 are displayed in (a) and (c) respectively. Kernel regression with kernels from (a) and (c) are used to estimate a function in (b) and (d).

Equation 5.4 reveals that kernel regression with the Mahalanobis metric is equivalent to regular kernel regression after the transformation $\mathbf{X} \leftarrow \mathbf{X}\mathbf{A}^{\mathrm{T}}$. So, $\mathbf{A}$ plays the role of the transformation function in DRR. It is important to note that this transformation does not explicitly reduce the dimensionality of the data as in a typical DRR application. But, kernel regression does not care about the dimensionality of the points, as long as the correct distance metric is used. Therefore, explicit dimension reduction is not necessary in this case.

Nevertheless, we can still incorporate explicit dimension reduction into the metric-learning process if our algorithm design requires it. For example, we can construct more complex regressors by transforming the data into a lower-dimensional subspace as a preprocessing step, and then using a more appropriate regressor afterward. This step is reminiscent of the commonly-used ideas of unsupervised data preprocessing to simplify data representation before performing the actual learning (Jolliffe, 1986; Fukumizu et al., 2009).

If the desired target dimensionality is known beforehand, forcing $\mathbf{A}$ to be an $(r \times m)$ matrix ensures that a transformation that maps the data into the r-dimensional space is directly learned. If the target dimensionality is not known ahead of time, we can use an unsupervised dimension-reduction method after the $\mathbf{X}\mathbf{A}^{\mathrm{T}}_{m\times m}$ transformation. In particular, it can be shown that directly learning $\mathbf{A}_{r\times m}$ is similar to learning $\mathbf{A}_{m\times m}$ and mapping $(\mathbf{X}\mathbf{A}^{\mathrm{T}}_{m\times m})$ into an r-dimensional space using PCA. Refer to Weinberger and Tesauro (2007) for more details.

Algorithms 9 and 10 highlight the details of the whole process depending whether we wish to perform the explicit dimension reduction using PCA or not. The $\mathbf{W}$ in line 7 of Algorithm 10 is the transformation matrix of PCA.

Algorithm 9 Multivariate MLKR, a metric learning algorithm for kernel regression.

1: **function** train($\mathbf{X}, \mathbf{Y}$)
2: Initialize $\mathbf{A}$.
3: **repeat**
4: $\quad \Delta\mathbf{A} \leftarrow -\alpha \frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ {using Eqn. 5.6}
5: **until** $\| \Delta\mathbf{A} \|_\infty <$ threshold
6: Return $\mathbf{A}$.
7: **end function**

8: **function** test($x, \mathbf{A}$)
9: return $\hat{f}(x; \mathbf{A})$ using Eqn. 5.1;
10: **end function**

We are now ready to introduce a variant of MLKR that is suitable for model-based RL algorithms.

Algorithm 10 Multivariate MLKR with explicit dimension reduction.

function train($\mathbf{X}, \mathbf{Y}$)
Initialize $\mathbf{A}$.
repeat
  $\Delta\mathbf{A} \leftarrow -\alpha \frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ {using Eqn. 5.6}
until $\| \Delta\mathbf{A} \|_\infty <$ threshold
$\tilde{\mathbf{X}} \leftarrow \mathbf{X}\mathbf{A}^{\mathrm{T}}$;
{ explicit dimension reduction step:}
$\mathbf{W} = PCA(\tilde{\mathbf{X}})$;
$\tilde{\mathbf{X}} \leftarrow \tilde{\mathbf{X}}\mathbf{W}^{\mathrm{T}}$;
Return $\mathbf{W}$.
end function

function test($x, \mathbf{W}$)
$\tilde{x} \leftarrow \mathbf{W}x$;
return $\hat{f}(\tilde{x})$ using Eqn. 5.1 with k_u kernel, and trained on $(\tilde{\mathbf{X}}, \mathbf{Y})$;
end function

5.2.2 Factorization of MLKR

For a given regression, the minimal of all the input subspaces that maintains the conditional independence of y and x is called the *central subspace* (Fukumizu et al., 2009). This concept provides important insight into the statistical efficiency of dimension reduction as it signifies what portion of the input data is redundant or irrelevant.

Estimating the transition function of a continuous state-space MDP involves solving a regression problem from $\mathbb{R}^{|\mathcal{S}|}$ to itself, with the target covariates being the next-state components. A large class of real-life environments including most physical control problems have a factorized transition function in which the individual components of the next state are independent of each other (*i.e.*, $\Pr(y(i)|x, y(j)) = \Pr(y(i)|x)$ when $i \neq j$). In fact, coming up with a control problem that is not in this class is not an easy task. For this type of environments, we introduce a factorized variation of MLKR, or FMLKR, that achieves better statistical efficiency. This improvement is achieved by breaking up the original regression into several easier ones with smaller central subspaces.

This extension is similar to the factorization used in FF-Rmax and is pretty straightforward to construct: the $\mathbb{R}^m \rightarrow \mathbb{R}^l$ regression is broken up into l univariate MLKR problems, one for each output component. Upon receiving the training data set, the

algorithm feeds $\{(x_1, y_1(j)), ..., (x_n, y_n(j))\}$ to the j-th MLKR learner, where $y_i(j)$ is the j-th component of y_i. To estimate the value of a query point x^*, it queries all MLKR learners and constructs $\widehat{y^*}$ from their outputs.

It can be shown that the central subspace of each univariate regression is smaller than or equal to the central subspace of the original multivariate formulation. In fact, since each univariate regression is dealing with only one component of the output, less information from the input space is typically needed, yielding smaller subspaces. We demonstrate this claim by a simple example. Consider learning the function $f(x) = I(x)+rd(x)$, where I is the identity function and the rd operator shifts the components of x downward. For this specific function, the dependency set of all the output components is all of the input variables, and therefore the whole input space is required to describe f. However, each output component depends on only two dimensions of the input.

The factored MLKR turns its regression into l sub-regressions and one might be concerned about the accumulation of errors caused by this process. But, fortunately, since the output variables are independent of each other, the sum of the errors of the univariate regressors is not more than the error of the multivariate regressor. On the other hand, smaller central subspaces of the factored regressors create exponentially better estimations due to the properties of the curse of dimensionality.

As an example, we estimated the above function in $\mathbb{R}^{10}$ using the two methods, FMLKR and MLKR. Figure 5.3 shows what happens if we force different target dimensionalities on the regressors. To produce this graph, we generated 100 points uniformly distributed in the unit square and used the above function plus a small amount of Gaussian noise to construct the training set. The x-axis shows the internal dimensionality we forced on the regressors. The y-axis shows the mean-squared-error measured on another set of 100 randomly selected points.

Since only one dimension is statistically sufficient to produce each component of the output (using the linear combination of the two dependent components), FMLKR quickly achieves good performance, even when the algorithm has to map the input data

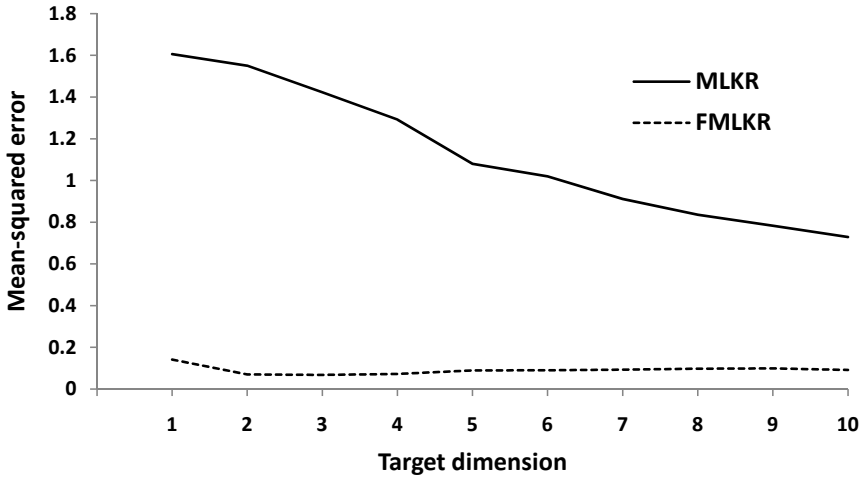


Figure 5.3: Comparison of MLKR and FMLKR on a simple regression problem.

into scalars. On the other hand, MLKR requires all the input dimensions in order to maintain the link between the input and output. That is why the result for MLKR improves as more dimensions are allowed in the transformation. MLKR cannot solve the regression problem as well as FMLKR even when it uses the whole input space, because a training set of 100 points is simply not enough to cover a 10-dimensional space.

5.3 The Proposed Algorithm

This section introduces a novel model-based algorithm called *Dimension Reduction in Exploration* (DRE). This algorithm is derived in the spirit of the algorithms in Chapter 3 and 4, and also several other published papers in model-based reinforcement learning that use model uncertainty to drive the exploration toward parts of the state space in which the algorithm is uncertain about its predictions (Brafman and Tennenholtz, 2002; Kakade, 2003). In particular, the skeleton of DRE is very similar to CF-Rmax and differs only by the way the transition function is trained and the knownness is computed in the kernel regression. While CF-Rmax used CKWIK-KR to model the transition function, DRE uses FMLKR (or MLKR).

DRE uses $|\mathcal{A}|$ multivariate FMLKR regressors to estimate the transition function,

each responsible for estimating the next state for one action—denoted by $\mathcal{F}_a$. Each of these regressors in turn consists of $|\mathcal{S}|$ MLKR regressors inside, each responsible for estimating one of the components of the next state. Let $\mathcal{F}_a^i$ be the univariate MLKR regressor responsible for estimating the i-th component of the next state when action a is used. Upon receiving a query point (s_t, a_t), the estimated transition function $\hat{T}$ is constructed by concatenating the output of $\mathcal{F}_{a_t}^i$ for all i's.

The accuracy of component i at time t—denoted ϵ_t^i—is computed by the kernel regressor $\mathcal{F}_{a_t}^i$ using the same procedure as in CKWIK-FKR with the accuracy of the transition function as $\epsilon_t = \sum_i \epsilon_t^i$.

The rest of the algorithm works exactly the same as CF-Rmax. In particular, let s_f be a new special state with self-loop transition on all actions and a reward of $R_{\max}$. DRE constructs its internal model as $\hat{M} = \langle \mathcal{S} + s_f, \mathcal{A}, \hat{T}', R, \gamma \rangle$, where the augmented transition function $\hat{T}'$ is computed as follows:

$$\hat{T}'(s'|s,a) = \begin{cases} 1 - \psi(s,a), & \text{if } s' = s^f \\ \psi(s,a)\hat{T}(s'|s,a), & \text{otherwise.} \end{cases} \tag{5.8}$$

The knownness function $\psi(s,a)$ is computed from ϵ_t using Definition 13. After the construction of $\hat{M}$, the algorithm uses an approximate planner to find a near-optimal policy for it. It then takes the greedy action according to this policy.

The pseudo-code of DRE, which is provided in Algorithm 11, is exactly the same as the one we developed for FF-Rmax except that it uses FMLKR instead of CKWIK-FKR.

The pseudo-code of DRE is provided in Algorithm 11. Although this algorithm has a similar structure to CF-Rmax, it performs a much better job in practice because of the way it handles the estimation of the transition function and computes the knownness function in the reduced-dimension space.

5.4 Discussion

There are two important characteristics that are vital to the success of DRE. First, the algorithm uses metric learning for estimating the transition function, and second, it

Algorithm 11 DRE, a model-based algorithm for continuous state space MDPs.

1: **Inputs:** Accuracy parameter ϵ_T , confidence parameter δ_T.
2: Initialize FMLKR regressor $\mathcal{F}$.
3: **for** all timesteps $t = 1, 2, \ldots$ **do**
4: Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
5: Update $\mathcal{F}$ using (s_t, a_t, s_{t+1}).
6: Update internal MDP $\hat{M}$ using the augmented transition function in Eqn. 5.8.
7: **end for**

computes the knownness function in a subspace of the original state space.

One of the most important properties of dimension-reduction techniques in regression is that they provide stable approximation when the sample size is small. In fact, many practical applications of these methods are for when the number of samples is on the order of the number of variables, in which case the classic approaches typically fail (Geladi, 1986; Fukumizu et al., 2009). DRE is able to build very realistic models of the world in the early stages of learning due to the efficiency of dimension reduction in regression.

The space in which the knownness is computed directly affects the sample complexity of the algorithm. For a query point to have a high knownness value, several points need to exist in its vicinity. Therefore, covering a space with known points requires a training set that is exponential in size with respect to the dimensionality of that space. By reducing the dimensionality of the space in which the knownness is computed, far fewer samples are needed to get high knownness values for the entire space.

The computational complexity of relearning the metric every $planFreq$ steps seems burdensome because $|\mathcal{A}| \times |\mathcal{S}|$ gradient descent instances need to be solved. However, our experiments indicate that the most time-consuming component of the algorithm is still the planning step. Part of this phenomenon stems from the way gradient descent searches the solution space. If we use the current $\mathbf{A}$ as the starting point of the gradient descent (Line 2 of Algorithm 9), after performing dimension-reduction once or twice, the starting point is usually very close to the optimal solution. As a result, gradient descent returns very quickly. If local optima are a concern and we can afford more computation, we can start the search using several initial matrices; we did not see

much improvement using this technique in practice.

DRE is closely related to FF-Rmax too because in FF-Rmax, the dependency graphs given to the algorithm effectively reduce the size of the model class of possible transition functions by projecting the data into the space spanned by only those variables inside the dependency set of the output component. Essentially, each G_a provides the transformation function Φ used in dimension reduction.

The transformation used in DRE has two main advantages to the DBNs used in FF-Rmax: First, the DBNs had to be provided by the user beforehand whereas DRE automatically discovers the transformations during the learning; and second, the DBN supported only those transformations that could be made by eliminating some of the input variables. Thus, it was always aligned with the original axis and it also had all the problems a kernel regressor is faced with in the design stage, such as picking the right value for σ and normalizing the variables. The dimension reduction process used in FMLKR, on the other hand, can handle any linear transformation of the original space, which effectively eliminates the problem of working with normalization and kernel width.

The next section provides some avenues of further research for dimension reduction in model-based RL.

5.5 Related Work and Future Extensions

MLKR was developed as a seamless integration of kernel regression with dimension reduction. We saw earlier that incorporating this technique into DRE decreases the sample complexity of exploration. However, it is important to note that DRE is not limited to kernel regression as the only choice for the transition-function estimator, nor is it restricted to the use of the particular dimension-reduction technique in MLKR. As mentioned earlier, the determinant factor in the success of DRE is the ability to discover the underlying relevant subspace of the state space w.r.t. the transition function, which allows the algorithm to compute the knownness values in that reduced-dimensional space.

This section investigates how some other dimension-reduction techniques can be incorporated into DRE. In particular, three case studies are presented in the rest of this section: 1) an unsupervised explicit dimension reduction used in combination with kernel regression, 2) a linear dimension reduction in combination with a multivariate linear regressor, and finally 3) dimension reduction in feature space using Gaussian processes.

5.5.1 Kernel Regression with Unsupervised Dimension Reduction

MLKR used the relationship between the input and output variables to discover the relevant subspace when estimating the transition function. In some situations, the distribution of the input data itself comes from within a subspace of the original space independent of the regression. In fact, examples of such scenarios are so abundant in practice that the majority of the dimension-reduction literature focuses on unsupervised methods. These algorithms typically compute a simpler representation of data while trying to minimize the reconstruction error.

Some of the applications of unsupervised dimension reduction occur in scenarios where the states are represented using many correlated variables, or cases where the input data can exist naturally in a smaller subspace of the original state space. For example, the state of a humanoid robot might be represented by its joint positions as well as the current torques at its motors; but, not all combinations of joint positions or motor torques are physically possible for the robot.

Many machine-learning techniques have been presented to deal with high-dimensional data that may have come from some underlying low-dimensional space. For example, as we saw earlier, PCA is an algorithm that linearly projects data into a subspace that captures as much variance of the data as possible. Some other techniques such as Isomap (Tenenbaum et al., 2000), local linear embedding (Roweis and Saul, 2000), Laplacian eigenmaps (Belkin and Niyogi, 2001), local multidimensional scaling (Venna and Kaski, 2006), and many of their variations discover manifolds with smaller dimensionality using a nonlinear transformation of the data.

Although these nonlinear embedding methods have been successfully applied to many industrial applications, no robust algorithm to the date of this writing is available to address the problem of online nonlinear dimension reduction with non-i.i.d. data. Nevertheless, we show here how to conceptually add unsupervised dimension reduction to DRE in case more suitable algorithms for performing nonlinear dimension reduction are developed in the near future. It is straightforward to modify DRE to work with any unsupervised dimension-reduction technique that explicitly represents data using a new coordinate system with a fewer number of dimensions.

Let ϕ_a be the transformation function that is learned using D_a as the training data. To incorporate ϕ_a into DRE, one needs to train and query the kernel regressors using the data points after the ϕ transformation. In this way, the regression (and subsequently, the computation of knownness function ψ) takes place in the reduced-dimensional space, and therefore, we get the benefits of unsupervised dimension reduction in exploration similar to the way the original DRE did. Algorithm 12 summarizes these changes.

Algorithm 12 DRE with unsupervised dimension reduction.

1: **Inputs:** Kernel regressor $\mathcal{F}_a$, dimension reduction transformer function ϕ_a.
2: Initialize history data-sets D_a.
3: **for** all timesteps $t = 1, 2, \ldots$ **do**
4: Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
5: Add (s_t, s_{t+1}) to the set D_{a_t}.
6: Update the transformation function ϕ_{a_t} based on the new D_{a_t}.
7: Retrain F_{a_t} using the training points $\phi_{a_t}(D_{a_t})$.
8: Estimate the transition function using:

$$\hat{T}(s, a) = \mathcal{F}_a(\phi_a(s)).$$

9: Update internal MDP $\hat{M}$ using the augmented transition function in Eqn. 5.8.
10: **end for**

As mentioned earlier, this algorithm treats the transformation function and how it is updated as a black box, and any linear or nonlinear dimension-reduction technique can be used. It is important to note however, that most of the practical nonlinear dimension-reduction algorithms at the time of this writing are not quite suitable for use in this procedure as they might produce unstable results when points are not sampled

i.i.d. from the state space. In particular, applying the current nonlinear dimension reduction techniques to an online RL setting is problematic. These algorithms look at the distances between nearby points to discover manifolds that the data could have come from. In most RL settings, the set of states in the history form a smooth trajectory inside the state space. This trajectory will look like a one-dimensional manifold, no matter how complicated the space is. Perhaps linear methods will suffer less from this phenomenon as they produce simpler transformations.

5.5.2 Partial Least Squares

Partial least squares (PLS) is an old but effective way to perform dimension reduction with linear regression. It is one of the covariance-based statistical methods also known as "structural equation modeling", and is particularly powerful in dealing with high-dimensional data with few training points (Geladi, 1986).

Given a training set $(\mathbf{X}_{n\times m}, \mathbf{Y}_{n\times l})$, PLS identifies a set of factors constructed by a linear combination of the input variables that best describe the output data when least-squares linear regression is used. The idea behind PLS is very simple. A linear regressor constructs its model by forming $\mathbf{Y} = \mathbf{XA}$ and estimating its parameter matrix $\mathbf{A}$ using the data. It is well-known that when $\mathbf{X}$ is ill-conditioned, computation of an optimal $\mathbf{A}$ is not easy. Several factors contribute to having an ill-conditioned input matrix. For example, when data points are very high dimensional and the training points are relatively few, matrix $\mathbf{X}$ will not be full rank. Another frequent situation that results in having an ill-conditioned matrix is when individual factors of the data are highly correlated with each other, for example when several variables that are related to each other by a fixed equation are used to describe the data. Ordinary least-squares methods cannot easily deal with these kinds of situations.

PLS extends the linear regression by computing a factor score matrix $\mathbf{T} = \mathbf{X}_{n\times m}\mathbf{W}_{m\times r}$ for an appropriate weight matrix $\mathbf{W}$. This matrix transforms data from m-dimensional space into a smaller r-dimensional one. Once this transformation is performed, we can solve the equation $\mathbf{Y} = \mathbf{TQ}$ for the parameter matrix $\mathbf{Q}$. Once $\mathbf{Q}$ is computed, we can

substitute it in the original linear formulation: $\mathbf{Y} = \mathbf{X}(\mathbf{WQ})$.

Computing $\mathbf{Q}$ and $\mathbf{W}$ simultaneously, such that the final regression has minimum error in the training set, is the goal of PLS algorithms. Several techniques exist for performing PLS, some of which work only for univariate regression and some others for multivariate regression. Two widely-used methods for performing PLS are NIPALS (Geladi, 1986) and SIMPLS (de Jong, 1993). A detailed overview of these algorithms is outside the scope of this document. However, pseudo-code for SIMPLS is provided in Algorithm 13 for reference.

Algorithm 13 SIMPLS (de Jong, 1993), an algorithm for performing partial least squares regression.

Inputs: Dimensionality of the latent matrix c.
Set $\mathbf{B}_0 = \mathbf{X}^{\mathrm{T}}\mathbf{Y}$.
Set $\mathbf{M}_0 = \mathbf{X}^{\mathrm{T}}\mathbf{X}$.
Set $\mathbf{C}_0 = \mathbf{I}$.
for each $t = 1, \ldots, c$ **do**
  Compute q_t, the dominant eigenvector of $\mathbf{B}_t^{\mathrm{T}}\mathbf{B}_t$.
  $w_t = \mathbf{B}_t q_t, c_t = w_t^{\mathrm{T}}\mathbf{M}_t w_t, w_t = \frac{w_t}{\sqrt{c_t}}$.
  Store w_t into $\mathbf{W}$ as a column.
  $p_t = \mathbf{M}_t w_t$, and store p_t into $\mathbf{P}$ as a column.
  $q_t = \mathbf{B}_t^{\mathrm{T}} w_t$, and store q_t into $\mathbf{Q}$ as a column.
  $v_t = \mathbf{C}_t p_t$, and $v_t = \frac{v_t}{\|v_t\|}$.
  $\mathbf{C}_{t+1} = \mathbf{C}_t = v_t v_t^{\mathrm{T}}$ and $\mathbf{M}_{t+1} = \mathbf{M}_t - p_t p_t^{\mathrm{T}}$.
  $\mathbf{B}_{t+1} = \mathbf{C}_t \mathbf{B}_t$.
end for
Return transformation matrix $\mathbf{W}$ and regression coefficient matrix $\mathbf{A} = \mathbf{WQ}^{\mathrm{T}}$.

PLS methods are not designed for CKWIK (or KWIK) frameworks because they do not provide a way to assess the uncertainty of their predictions. To use PLS in DRE, we need to find a mechanism to create the knownness values.

One such method that was studied in this dissertation was the ball-heuristic. This function uses ϵ-balls to generate knownness values. To cover any space using ϵ-balls, a number that is exponential in the dimensionality is required. However, if the balls are used in the subspace generated by the $\mathbf{W}$ transformation instead of the original space, exponentially fewer samples will be required to cover the space.

Another way we can apply PLS to a model-based algorithm is to use the setup of KWIK linear regression in Algorithm 3, if we decide to use a Boolean knownness

instead. Again, we use that algorithm on the input data after the transformation into the smaller subspace.

5.5.3 Gaussian Processes

Recently, another technique was developed by Jung and Stone (2010) that is worth mentioning here; this technique uses Gaussian processes (GPs) in a model-based algorithm similar to DRE. A Gaussian process is a nonparametric regression technique that assumes the entire function is selected from a Gaussian distribution over functions (Rasmussen and Williams, 2006). GP provides a way to compute the posterior over this distribution given a prior and some data from the actual function. GP has been used in many different disciplines as a robust and powerful regression technique, and since it is a nonparametric method, it provides great modeling flexibility.

There are three more properties of GP that make it very suitable for use in a model-based RL algorithm. (1) GP easily integrates with projection of the input data into a feature space using basis functions. Once the original data is transformed into the high-dimensional space of features, GP becomes much more expressive and powerful. (2) GP provides a way to learn the hyper-parameters of the basis functions, which can be used to perform dimension reduction in the feature space. (3) GP provides a natural way to compute the uncertainty of its estimations because it expresses its predictions using the posterior in the form of a Gaussian distribution. The mean of the distribution can be used as the output of the regressor and the variance can be used as an uncertainty measure. The uncertainty measure can be translated into a knownness value using the techniques from Chapters 3 and 4. These three properties make GP a good fit for model-based reinforcement learning.

Let us see how to incorporate GP into CKWIK-Rmax. The $\mathcal{F}_T$ regressor is a mixture model that combines $|S|$ univariate GP regressors. The factorization is performed the same way as in FMLKR. Let ϕ be the feature transformation function and $\mathbf{\Phi}$ be the input matrix after the data has been transformed from m-dimensional space into the feature space with dimension p using ϕ (for now we assume the basis functions are

given). It can be shown that the output associated with a query point x has the following posterior distribution:

$$f(x; \mathbf{X}, y) \sim \mathcal{N}\left(\frac{1}{\sigma_n^2}\phi(x)^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{\Phi}y, \phi(x)^{\mathrm{T}}\mathbf{B}^{-1}\phi(x)\right), \tag{5.9}$$

where $\mathbf{B} = \sigma_n^{-2}\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}} + \Sigma_p^{-1}$. Readers are referred to Rasmussen and Williams (2006) for more details about the procedure. GP outputs a Gaussian distribution for each query point, which can be used both as a prediction and an uncertainty measure. At each timestep, calling GP with $(\mathbf{X}, y) = D_{a_t}$ and $x = s_t$ will provide an estimate of the transition function as well as a variance that can be used directly as ϵ_t^j after normalization. Knownness can be computed using the same definition used in CKWIK-Rmax.

GPs can be described using an alternative way of interpreting the same results directly in the function space. In this view, any GP is fully described by its mean function $m(x)$ and a covariance function $k(x, x')$. The covariance function can be thought of as a function that measures the relationship between the two points x and x'. From this perspective, we can treat them as the kernel function in MLKR. In fact, a popular covariance function that is widely used in GP is the squared exponential function that is very similar to the Gaussian kernel in Equation 5.2:

$$k(x_1, x_2) = \exp(-\frac{1}{2} \parallel x_1 - x_2 \parallel_2^2). \tag{5.10}$$

We can customize this covariance function in the same way we customized the Gaussian kernel:

$$k(x_1, x_2; v_0, b, \mathbf{M}) = v_0 \exp(-\frac{1}{2}(x_1 - x_2)^{\mathrm{T}}\mathbf{M}(x_1 - x_2)) + b, \tag{5.11}$$

where $\theta = \{v_0, b, \mathbf{M}\}$ is the set of hyper-parameters of the covariance function that need

to be learned. Some of the widely-used structures of $\mathbf{M}$ are:

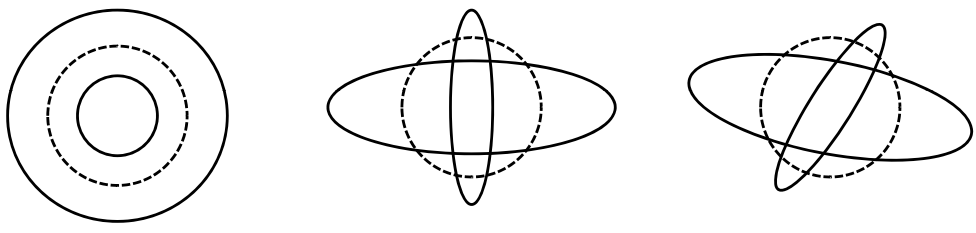
$$\mathbf{M_1} = l\mathbf{I}, \qquad \text{uniform distance} \tag{5.12}$$

$$\mathbf{M_2} = \text{diag}(l_1, \ldots, l_m), \qquad \text{variable selection} \tag{5.13}$$

$$\mathbf{M_3} = \mathbf{B}\mathbf{B}^{\mathrm{T}} + (l_1, \ldots, l_m), \qquad \text{linear transformation,} \tag{5.14}$$

where the parameter matrix $\mathbf{M}$ is reduced to scalar l in $\mathbf{M_1}$, vector $(l_1, \ldots, l_m)$ in $\mathbf{M_2}$, and $\{\mathbf{M}, (l_1, \ldots, l_m)\}$ in $\mathbf{M_3}$. The hyper-parameters in $\mathbf{M_2}$ play the role of *characteristic length-scales* for each dimension, which basically means how far in each dimension we need to go before the points become uncorrelated with each other. This covariance function is a way to detect the relevant features because it measures the inter-dependency of points along each dimension differently—something also known as "automatic relevance determination" (Rasmussen and Williams, 2006). The last function resembles the decomposition we used in the Mahalanobis metric of the Gaussian kernel.

Figure 5.4 illustrates three classes. The dotted line in each example is an equidistant contour if $\mathbf{M} = \mathbf{I}$ was used. The solid lines are some examples when the parameters of the matrix are changed. As we can see, $\mathbf{M_1}$ can only control how wide the contour is without changing its shape (similar to the width parameter of the Gaussian kernel). $\mathbf{M_2}$ can vary the width of the contour along each dimension individually (producing vertical and horizontal ellipsoids), and finally $\mathbf{M_3}$ allows for rotation and scaling along any direction.



(a) (b) (c)

Figure 5.4: Examples of different parameter values for $\mathbf{M_1}$, $\mathbf{M_2}$ and $\mathbf{M_3}$ in (a), (b) and (c) respectively. In each figure, the dotted line represents an equidistance contour when $\mathbf{I}$ is used. The solid lines show how different parameter values change that contour.

The parameter set θ can be learned using likelihood maximization. It is common to have a mixture of different classes of functions and have the GP figure out the right model. This process creates a hierarchical specification of models. At the lower level, we have the parameters of a specific model, for example $(l_1, \ldots, l_m)$ in $\mathbf{M}_2$. At the top level, we have a set of possible model classes $\mathcal{H}_i$, for example a probability distribution over $\mathbf{M}_1, \mathbf{M}_2$ and $\mathbf{M}_3$.

Inference on these parameters can be done one level at a time from bottom to top. At the lower level, the posterior is computed as:

$$\Pr(\theta|y, \mathbf{X}, \mathcal{H}_i) = \frac{\Pr(y|\mathbf{X}, \theta, \mathcal{H}_i)\Pr(\theta|\mathcal{H}_i)}{\Pr(y|\mathbf{X}, \mathcal{H}_i)}, \tag{5.15}$$

where $\Pr(\theta|\mathcal{H}_i)$ is the prior for the hyper-parameters. The normalizing constant in the denominator is computed as:

$$\Pr(y|\mathbf{X}, \mathcal{H}_i) = \int \Pr(y|\mathbf{X}, \theta, \mathcal{H}_i)\Pr(\theta|\mathcal{H}_i)d\theta. \tag{5.16}$$

At the top level, the posterior for the model is computed as:

$$\Pr(\mathcal{H}_i|y, \mathbf{X}) = \frac{\Pr(y|\mathbf{X}, \mathcal{H}_i)\Pr(\mathcal{H}_i)}{\Pr(y|\mathbf{X})}, \tag{5.17}$$

where $\Pr(y|\mathbf{X}) = \sum_i \Pr(y|\mathbf{X}, \mathcal{H}_i)\Pr(\mathcal{H}_i)$ assuming the model classes are finite. Of course, computation of some of these equations, particularly the integral in Equation 5.16, requires numerical approximations. Efficient implementation details of GP can be found in the numerous available handbooks (Rasmussen and Williams, 2006).

5.6 Experimental Results

We used several experiments to evaluate DRE and investigate its properties. The first experiment was designed to demonstrate the statistical efficiency of learning the transition function using dimension reduction. For this purpose, we ignored the exploration problem and evaluated the algorithms on offline data. We used n-MOUNTAINCAR with

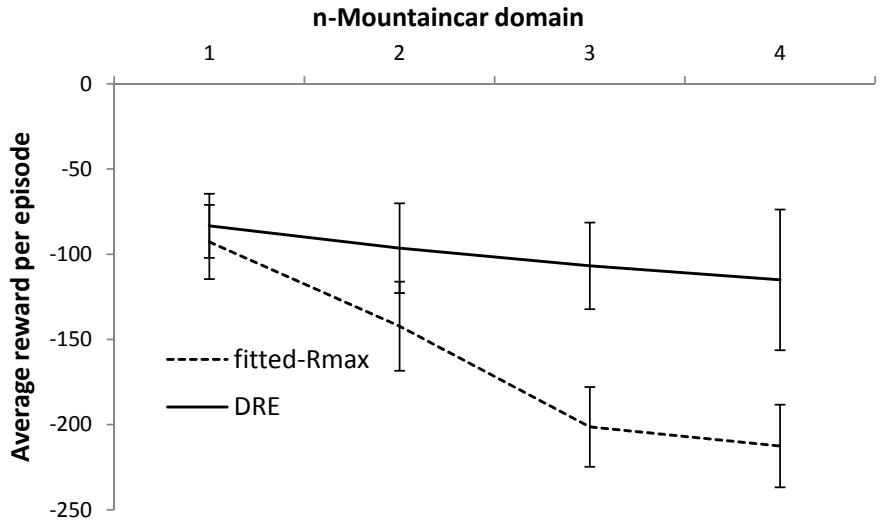


Figure 5.5: Batch reinforcement learning results in n-MOUNTAINCAR using two algorithms.

different values of n as the environment for this experiment. For each individual domain, we collected 500 transitions using a policy that selected actions randomly from 100 random start states and ran for 5 timesteps. Each algorithm learned a model using the data. We then chose another 100 random start states as the test set, and evaluated each algorithm by starting from these states and running its learned policy. For the sake of statistical significance, the experiment was repeated 20 times.

Fitted-Rmax and DRE were tested in this experiment (the behavior of CF-Rmax would have been similar to fitted-Rmax because they both use a regular kernel regression to estimate the model). The y-axis of Figure 5.5 shows the average reward-per-episode when the agent started the episode from the test points. The x-axis shows the number of cars in the world (state-space dimensionality is $2x$). Error bars represent standard deviation.

All the algorithms used the same planner (FVI with a resolution of 30 per dimension and $C = 5$). Fitted-Rmax used $\sigma = 0.3$ as the kernel width. As mentioned earlier, the optimal value function was the same in all n-MOUNTAINCARS, and any performance degradation was only due to the model-approximation error.

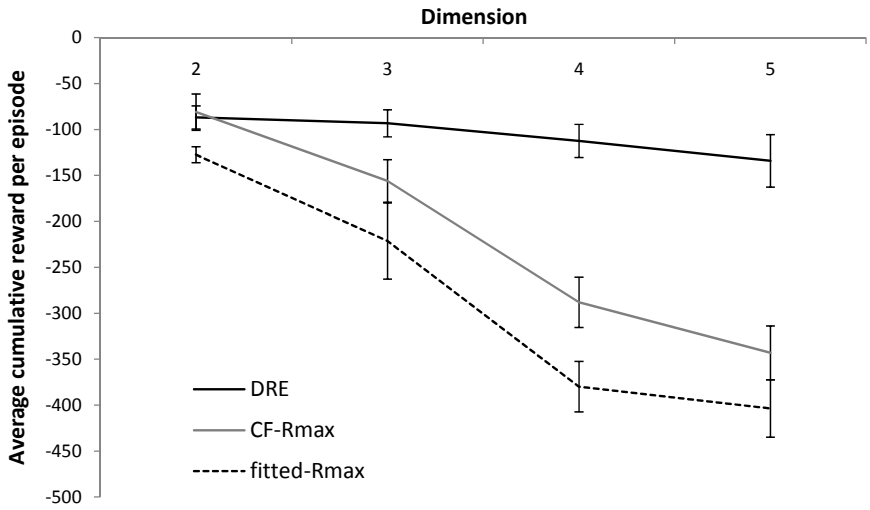


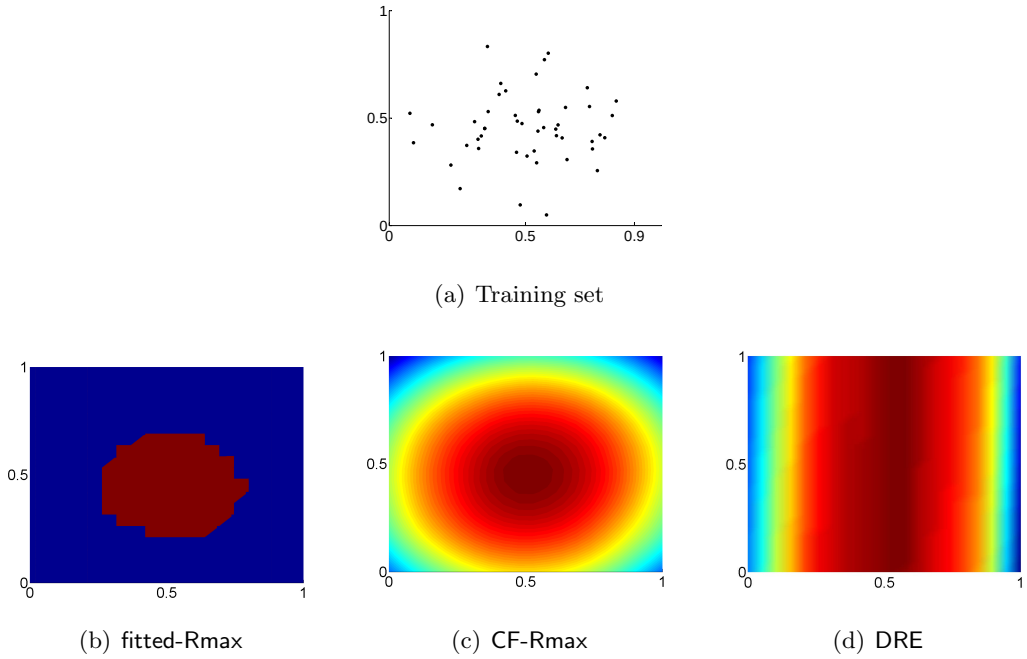
Figure 5.6: Online evaluation of three algorithms in n-Puddleworld for different dimensions.

As expected, the performance of fitted-Rmax degraded as the dimensionality increased because 500 samples were not enough to learn a kernel regression in higher than four-dimension spaces. Therefore, at 3-Mountaincar, the fitted-Rmax algorithm behaved very close to the random policy. DRE, on the other hand, managed to keep the dimensionality low even in 4-Mountaincar because of the fact that only a subset of dimensions were necessary to predict each component of the output. The internal dimension of the univariate MLKRs never went above 2 in this experiment.

The second experiment tested a set of algorithms in an online setting. For that reason, we included CF-Rmax in the experiment, along with DRE and fitted-Rmax. We used n-Puddleworld as the first environment. Each algorithm ran 50 episodes in each n-dimensional Puddleworld—with a cap of 300 steps—and the results were averaged over 20 runs. Figure 5.6 shows the average performance-per-episode (cumulative reward divided by the number of episodes) as dimensionality increased. Similarly to the previous result, DRE retained a low-level representation in all the domains (it never used more than one dimension in each MLKR). Therefore, it created high knownness values very quickly, as they were computed in low-dimensional spaces. CF-Rmax and fitted-Rmax did not perform well in high-dimensional Puddleworlds.

We then compared the knownness functions of these three algorithms using a simple

experiment. We selected part of PUDDLEWORLD's transition function: $f(x) = x(2) + 0.05 + \mathcal{N}(0, 0.01)$, and compared the knownness functions of the algorithms after training on 50 samples. The heat-map graphs in Figure 5.7 show the knownness values across the entire space. Dark red indicates a completely known state and dark blue means an unknown state. The training points are shown in part (a). DRE discovered the fact that the vertical dimension was not relevant to the function, so it accumulated knownness values along that axis. Since CF-Rmax had to work in 2D, which resulted in weaker generalization, it produced much smaller knownness values. Fitted-Rmax did a worse job of generalizing because it had to convert the knownnesses into a Boolean output.



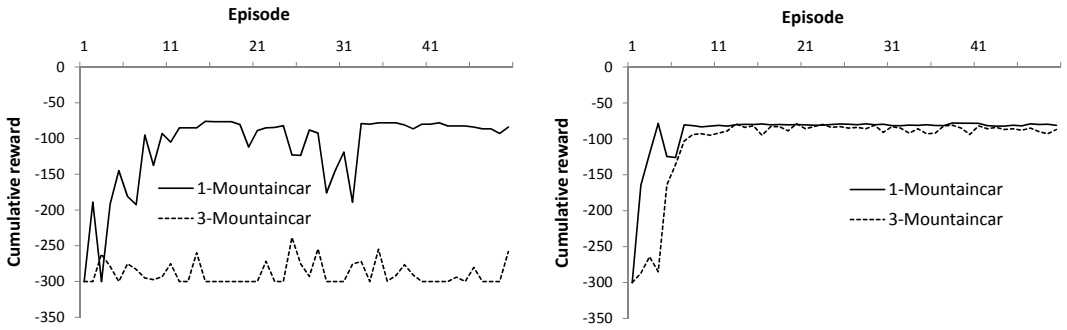
(a) Training set

(b) fitted-Rmax (c) CF-Rmax (d) DRE

Figure 5.7: Comparison of the knownness function computed by three algorithms using the same set of training points. Sub-figures (b),(c), and (d) show the knownness function computed by fitted-Rmax, CF-Rmax, and DRE, respectively.

Next, a similar online experiment was performed in n-MOUNTAINCAR for two values $n = 1$ and $n = 3$. Figure 5.8 provides a more detailed comparison of DRE and fitted-Rmax in 1-MOUNTAINCAR (the solid lines) and 3-MOUNTAINCAR(the dotted lines). This set of graphs shows the learning curves for the two algorithms (the x-axis is the episode number and the y-axis shows the collected rewards per episode). Results are averaged over 20 runs and smoothed over a window of size 5.

Both algorithms performed highly comparably in 1-MOUNTAINCAR; but, fitted-Rmax completely failed to learn in 3-MOUNTAINCAR using 50 episodes, whereas DRE did not suffer much.



(a) fitted-Rmax (b) DRE

Figure 5.8: The learning curves of (a) fitted-Rmax and (b) DRE in two environments: 1-MOUNTAINCAR and 3-MOUNTAINCAR.

We tried DRE in several other benchmarks and have found it very robust and stable. One of the contributing factors to this behavior is that it has very few parameters to tune in, and that the algorithm performance is not very sensitive to their values. In fact, as mentioned above, we used only one set of parameter values to run the algorithm across all the environments.

To showcase this advantage using empirical data, we tested Q-learning with a widely-used function approximator, CMAC (Sutton and Barto, 1998), on several different RL benchmarks and measured the sensitivity of the algorithm performance on CMAC parameters. In other words, we were interested in knowing what the best parameter values were for each test benchmark, and how much they varied from one domain to another. A set of seven domains were used for this experiment: (1,2,3)-MOUNTAINCAR, (2,3,4)-PUDDLEWORLD, and another continuous control problem called ACROBOT (Sutton and Barto, 1998). Four different tiling resolutions were used as the parameter values of CMAC: 1,2,3, and 5 tiles per dimension.

Figure 5.9 shows the effect of these parameters on the performance of Q-learning. The X-axis contains the environments and the Y-axis is the average cumulative reward per episode. Each line in the graph corresponds to a fixed tile resolution for CMAC. For

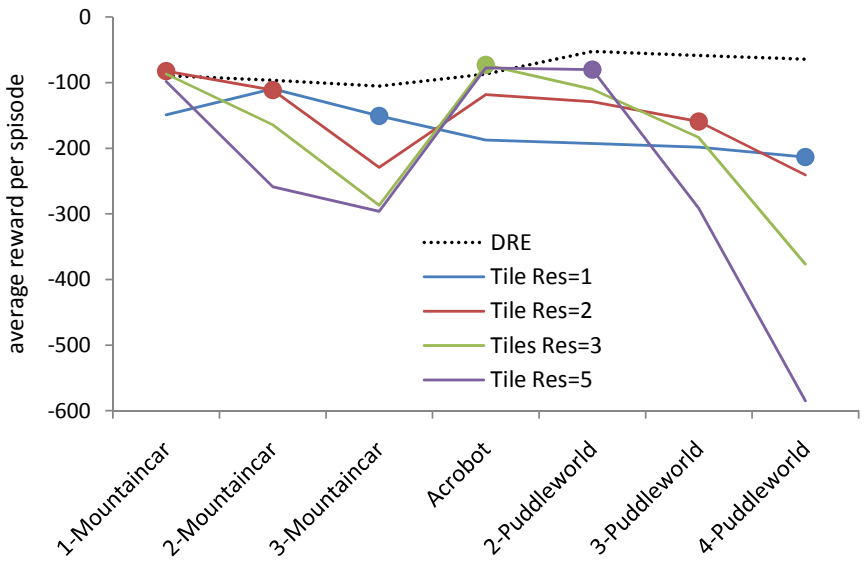


Figure 5.9: Evaluating Q-learning in seven environment when different parameter values are used. One instance of DRE outperforms the best instance of Q-learning in nearly every domains.

each environment, the parameter that achieved the best result is identified by a circle. Comparison of the different lines in different domains revealed interesting observations. For example, while a resolution of size 3 achieved the best performance in Acrobot, it had almost the worst performance in 3-Mountaincar. DRE with one fixed set of parameter values performed comparatively to the best instance of Q-learning in all domains. In fact, it outperformed Q-learning in most of the domains.

The next experiment investigated the relationship between FF-Rmax and DRE. We mentioned earlier that an independence graph provides a special way of performing dimension reduction because the agent can eliminate some of the input variables during the model estimation process. However, this particular type of graph is not able to model transformation functions other than variable elimination. In particular, DRE computes a linear transformation of the input space, which is a much broader transformation than variable elimination. The next experiment demonstrated this difference.

Warpedworld is a generalization of the Puddleworld domain. These environments are exactly the same except for their transition functions. While executing each action in Puddleworld results in a movement in the intended direction by a constant value,

actions in WARPEDWORLD move the agent in the intended direction by a length that is computed using a linear combination of the state variables. For example, the action *right* would change $s(1)$ to $s(1)+0.05$ in PUDDLEWORLD, but to $as(1)+bs(2)$ in WARPEDWORLD (a and b are two constants). Similarly, one can generate n-WARPEDWORLD from n-PUDDLEWORLD. Please refer to Appendix C.3 for more details.

We compared three algorithms in 3-WARPEDWORLD: DRE and FF-Rmax with two different inputs. FF-Rmax$_{\text{correct}}$ used the correct DBN of WARPEDWORLD and FF-Rmax$_{\text{wrong}}$ used the graph for 3-PUDDLEWORLD, which was not a good representation of the dependencies in 3-WARPEDWORLD. It is easy to see that each component of the next state variable in WARPEDWORLD is dependent on all the input variables because it is computed using a linear combination of all the variables. So, the true maximum dependency size was 3 for all the graphs. FF-Rmax$_{\text{wrong}}$ used the DBN of PUDDLEWORLD. Since the transition function of PUDDLEWORLD was much simpler than WARPEDWORLD, these DBNs missed several important edges.

Figure 5.10 presents the learning results of the above algorithms. As was expected, FF-Rmax$_{\text{wrong}}$ was not able to learn anything as it could not model the dynamics of the world. But, an interesting outcome of this experiment was that FF-Rmax$_{\text{correct}}$ was not able to learn a good behavior either, although the true dependency graphs were given to it. Given the limited amount of experience it had, it was not able to fully learn the 3-dimensional regression. However, DRE achieved a much better result because it used a linear transformation that mapped the input space into a 1-dimensional one without losing information.

The final experiment we considered was testing DRE in BUMBLEBALL. Attempting to make the Table 4.1 from the previous chapter more complete, we tested DRE using the same settings as before. Table 5.1 provides a complete summary of all four algorithms tested in BUMBLEBALL. Obviously, FF-Rmax performed better than DRE because the structure was given to it. But, DRE was also able to learn this 5-dimensional task using only 3000 samples, without any prior information from the user.

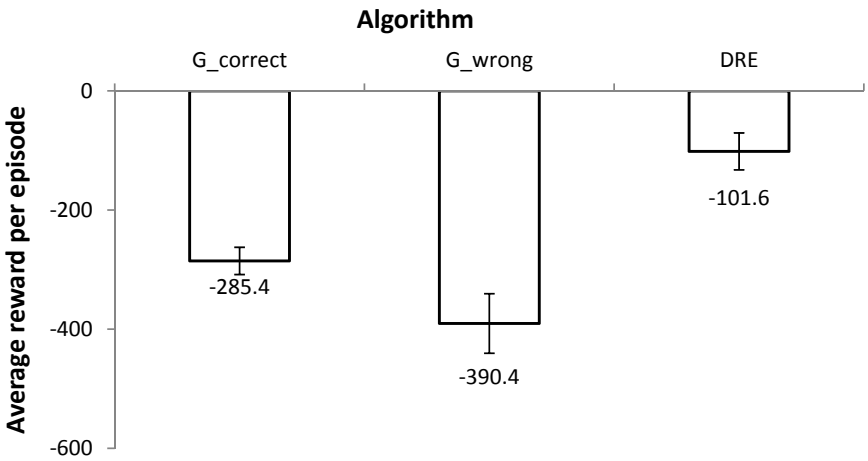


Figure 5.10: Evaluating three algorithms in 3-Warpedworld: DRE and two versions of FF-Rmax that used correct and incorrect dependency graphs.

Algorithms:	Random	CF-Rmax	FF-Rmax	DRE
Total cumulative reward:	−24269	−21461	−3917	−4077
Number of collisions:	533.0	463.7	38.0	53.3
Percent finished episodes:	8.6%	13.0%	81.3%	77.0%

Table 5.1: Performance of DRE in the Bumbleball domain.

Chapter 6

Extension: Multi-resolution Exploration

This chapter extends the utility of the knownness concept, which was introduced earlier for data-efficient model-based learning, to a novel hierarchical exploration scheme. This multi-resolution exploration expands the effectiveness of the model-based approach beyond what has been achieved in the PAC-MDP framework. The algorithm developed in this chapter successively refines how it interprets the uncertainty level, which allows it to systematically look for better and better predictions about the world as the learning progresses.

The performance metric used in all the algorithms we have studied so far demands (1) near-optimal behavior in all but polynomial number of timesteps with high probability, but does not insist on (2) performance improvements after convergence, nor does it (3) provide performance guarantee at any particular timestep. Such "anytime" behavior is encouraged by algorithms with regret bounds (Auer and Ortner, 2007), but unfortunately no such algorithm has yet been developed for regret minimization in continuous state spaces. The goal of this chapter is to devise an algorithm that has these properties. That is, we are interested in an algorithm that never stops systematically searching for better behaviors and can provide a performance guarantee anytime the learning is stopped.

As a motivating example for the work presented in this chapter, consider how a discrete state-space algorithm might be adapted to work for a continuous state-space problem. The practitioner must decide how to discretize the state space. While finer discretizations allow the learning algorithm to learn more accurate policies, they require much more experience to learn well. The dilemma of picking fine or coarse resolution has to be resolved in advance. Some of the factors that are involved in this decision making

are based on the dynamics and reward structure of the environment, the desired level of optimality and also on the quantity of resources (in terms of samples) are available. For example, if you are working with a robot, you need to know beforehand how many samples you can afford to collect with the robot. The more samples you can collect, the finer the discretizations you can afford with the algorithm. Meanwhile, the performance of the algorithm depends critically on these *a priori* choices, because the algorithm does not respond dynamically based on the available resources.

This chapter develops an algorithm called *multi-resolution exploration* or MRE to address this situation. This algorithm has a similar skeleton to CKWIK-Rmax, except that it does not receive the PAC-MDP input parameters (ϵ, δ) from the user. To satisfy conditions (1) and (2) mentioned earlier, the algorithm changes the way $\epsilon(s, a)$ is computed in $\mathcal{F}_T$ and how it is interpreted. In fact, there is a fundamental difference between the way uncertainty is measured in MRE and the algorithms developed in the previous chapters. The uncertainty parameter $\epsilon(s, a)$ used to measure how close the estimation was to the actual transition function. This interpretation of uncertainty is independent of time and depends only on the distance between the estimation and the true value. The knownness value was then computed based on the relative values of $\epsilon(s, a)$ and the target accuracy ϵ_T. To make sure the algorithm always searches for better behaviors, MRE successively refines the interpretation of ϵ_T during learning. This process, which is borrowed from the literature on regret minimization, changes ϵ_T such that a fixed estimation looks less accurate as time goes on. MRE also maintains a hierarchical mapping structure over the state space to make sure it can provide an estimation of its performance at any time the learning stops.

In the rest of this chapter, the concept of hierarchical mapping and multi-resolution exploration is first developed, and then it is integrated into a model-based algorithm. Following the same pattern used in previous chapters, a discussion section investigates the properties of the algorithm including, in this case, how this exploration can be extended to value-based RL algorithms.

6.1 Multi-resolution Discretization

Let us go back to the idea of the ball-heuristic from Chapter 3 and investigate its properties a little further. To reiterate, ball-heuristic is used in regression to measure uncertainty. It does so by forming an ϵ-ball around any query point and counting the number of training points inside that ball. We can extend this heuristic to something computationally faster by discretizing the space using an ϵ-granularity. Such a discretization creates a structure ζ such that each cell $\varsigma \in \zeta$ can be contained inside an ϵ-ball. Given this discretization, the uncertainty of a query point is computed based on the number of training points inside the cell that contains the query point. Since any point in that cell is also inside the ϵ-ball of the query point, this method can be thought of as a more constrained version of the ball-heuristic.

The intuition behind this heuristic comes directly from the smoothness assumptions in the regression space. For example, consider learning a transition function that has the form of a parametric distribution $\mathcal{P}(\mu, \Sigma)$ with a Lipschitz continuity constant C_T. Let $\varsigma(s^*)$ be the cell containing s^* and $\mathcal{N}_\varsigma(s^*; D_a)$ be the set of points in D_a that lie inside $\varsigma(s^*)$. Also, suppose we use the following simple averager as the function approximator:

$$\mathcal{F}_a(s^*; D_a) = \frac{1}{k} \sum_{s_i \in \mathcal{N}_\varsigma(s^*; D_a)} s'_i, \tag{6.1}$$

where s'_i is a sample from the transition function of s_i or $\mathcal{P}_i$ and $c = |\mathcal{N}_\varsigma(s^*; D_a)|$. Similar to Lemma 15, it can be established that:

$$\| \mathbb{E}\left[\mathcal{F}_a(s^*)\right] - \mu^* \|_2^2 \leq \epsilon C_T, \tag{6.2}$$

where C_T is the Lipschitz constant. Again, we can use the same technique used in Lemma 16 to show that the following equation holds with probability at least $1 - \delta$.

$$\| \mathcal{F}_a(s^*; D_a) - \mu^* \|_2^2 \leq \sqrt{\frac{\ln(2/\delta)}{2c^2}} + \epsilon C_T, \tag{6.3}$$

where the first term is because we used a finite number of samples to estimate μ^*

and the second term is because the samples were taken not from $\mathcal{P}^*$ but from some other distributions $\mathcal{P}_i$'s. Therefore, Equation 6.1 provides a principled way to perform exploration that can be used in CKWIK-Rmax because in addition to making predictions, it also provides the accuracy of its predictions (according to Equation 6.3).

In this setup, the discretization resolution has a crucial role in the outcome of the function approximator because it controls the degree of generalization in Equation 6.1. The larger the value of ϵ is, the further the training points are allowed to be in order to have an impact on the outcome of the averager at a certain query point. In a sense, this parameter plays the role of the characteristic length in GP (for example, in Equation 5.12). In Chapter 5, we saw how hyper-parameters like this one can be optimized using data in an RL algorithm.

Instead of optimizing for a single value of ϵ, MRE automatically adjusts the degree of generalization for different parts of the state space depending on the distribution of the input data. In other words, MRE constructs a variable degree of generalization across the state space.

Figure 6.1 compares the discretizations created by the ball-heuristic and MRE. Figure 6.1(a) shows a fine resolution discretization of the state space. This partitioning produces accurate estimations, but with the cost of creating narrow generalization in the entire space—high sample complexity. Figure 6.1(b) shows a discretization that creates wide generalizations due to its big cell size, but might result in poor estimations. Figure 6.1(c) shows a variable discretization created by MRE. Cells with small sizes correspond to parts of the state space with a lot of sample points. The algorithm makes narrow, but accurate, generalizations in areas with a high-density of data, and wide generalizations in low-density ones.

These two properties that allow variable generalization across space and time provide the necessary tools for MRE to develop the two behavioral conditions we were looking for at the beginning of the chapter. To implement these ideas, MRE maintains a data structure called an "uncertainty tree" that is explained next.

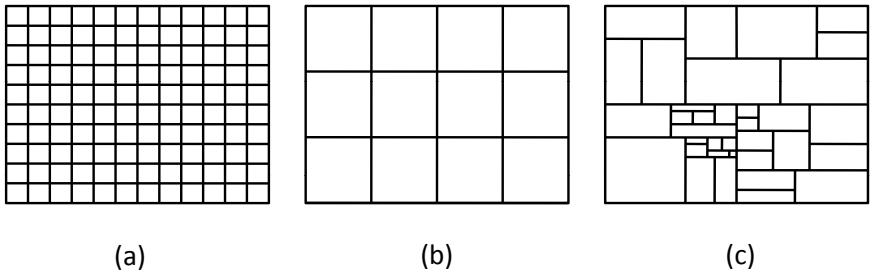


Figure 6.1: (a) a fine resolution discretization of the space that allows for accurate estimates, but requires a lot of samples, (b) a discretization that allows for wide generalization, but with less accuracy, (c) variable discretization in MRE that combines the good properties of (a) and (b).

6.1.1 Uncertainty Trees

Regression trees are a well-known class of local function approximators that partition the input space into non-overlapping regions and use the training samples of each region for predicting the value of query points inside that region. Their ability to maintain a non-uniform discretization of high-dimensional spaces with relatively fast query time has proven to be very useful in various algorithms (Ernst et al., 2005; Munos and Moore, 2002).

For the purpose of our RL algorithm, we develop a new regression tree algorithm called "uncertainty tree" that conforms to the protocol of the CKWIK framework.

An uncertainty tree ζ is an online multivariate local regressor. Since this algorithm works in the CKWIK framework, it can be used to construct an RL algorithm using the structure of CKWIK-Rmax. In particular, we can use $|\mathcal{A}|$ uncertainty trees, each estimating the transition function when one of the actions is performed. The following explains how $\hat{T}(s_t, a_t)$ and ϵ_t are computed by uncertainty trees.

In an uncertainty tree ζ, each node ς covers a bounded region in the state space and keeps track of the points inside that region, with the root covering the whole space. Let R_ς be the region of ς and denote its boundaries by two k-dimensional vectors $r_<$ and $r_>$, representing the minimum and maximum values in each dimension, respectively.

Each internal node splits its region into two half-regions along one of the dimensions to create two children. Parameter ν determines the maximum number of points allowed

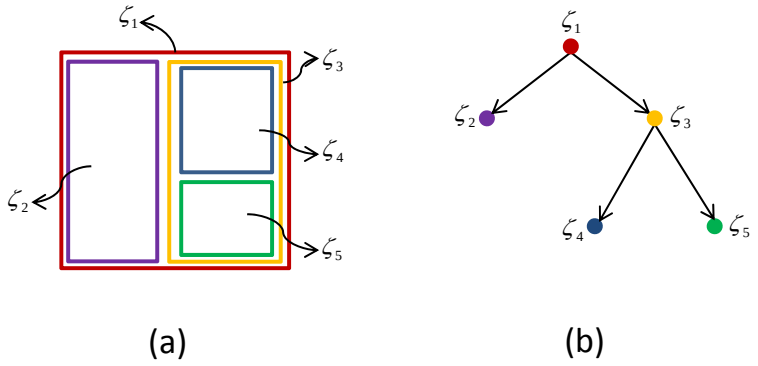


Figure 6.2: Illustration of an uncertainty tree: (a) shows the structure of a tree and (b) shows how the tree partitions a 2D space.

in each leaf. For a given node ς, let ϵ_ς be the smallest ϵ-ball that contains ς and $D(\varsigma)$ be the set of data pairs in the training set D that fall inside R_ς. Also, the normalizing size of the tree, denoted by τ, is defined as the region size of a hypothetical uniform discretization of the space that would have (on average) $\frac{\nu}{|\mathcal{S}|}$ data points inside each cell, provided that the points were uniformly distributed in the space:

$$\tau(D) = \frac{1}{\lfloor \sqrt[|\mathcal{S}|]{|D| \times |\mathcal{S}|/\nu} \rfloor}. \tag{6.4}$$

Figure 6.2 illustrates the structure of an uncertainty tree. The tree structure on the left induces a partitioning of the state space, which is depicted on the right.

The algorithm maintains the structure in an online fashion. At each timestep $t = 1, 2, \ldots$, a query point x_t is given, for which the algorithm outputs $\hat{y}_t$ along with $\epsilon(x_t)$. It then receives a sample from the true function $x'_t \sim f(x_t)$.

To insert (x_t, y_t), the algorithm starts at the root and travels down the tree, depending on which child contains x_t. Once inside a leaf ς, the algorithm adds (x_t, y_t) to ς's data set. if $|D(\varsigma)|$ is more than ν, the node splits and creates two new half-regions. Splitting is performed by selecting a dimension $j \in [1..|\mathcal{S}|]$, and splitting along the j-th

dimension according to a split value sv.

Similar to conventional regression trees, computing j and sv can be done according to a number of different splitting heuristics. For example, a standard way to select j is to use a round-robin strategy. If the parent of a node has split in the direction j, the child splits in $(j+1) \mod |\mathcal{S}|$. The value of sv can be selected to be the middle of the region (that is, $(r_<(j)+r_>(j))/2$) or the value of $s_i(j)$, where $s_i \in \varsigma$ is the median point of the cell's data set according to dimension j. More detailed discussions on various splitting criteria can be found in (Preparata and Shamos, 1985; de Berg et al., 2008).

Given the query point (x_t), the tree first locates the leaf containing x_t by traveling down the tree like before, denoted by $\varsigma(x_t)$. Then, it uses the local function approximator used in $\varsigma(x_t)$ to compute the estimate $\hat{y}_t$. For the sake of concreteness, consider a simple averager as the function approximator inside each leaf:

$$\mathcal{F}_\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{|D(\varsigma(x))|} \sum_{(x_i,y_i)\in D(\varsigma(x))} y_i.$$

This regressor creates piece-wise constant estimation of the transition function for each cell. A more sophisticated regressor would be to use a least squares linear regression trained on $D(\varsigma(s))$.

As mentioned before, the accuracy output $\epsilon(x_t)$ does not have the same meaning as before. Intuitively, this output provides an incentive for the RL agent to explore different regions of the state space. Here, we consider two heuristic functions for computing $\epsilon(x_t)$. These heuristics will not create algorithms with PAC-MDP guarantee because they do not satisfy the closeness assumption of the CKWIK framework. Instead, they are constructed from heuristic functions that have been used in both k-armed bandit (Auer et al., 2002) and reinforcement-learning (Auer and Ortner, 2007) literatures.

The first heuristic function uses a mixture of the relative size of the cell w.r.t. the normalizing size of the tree and a function based on ball-heuristic to compute $\epsilon(x_t)$:

$$\epsilon(x_t) = \left(1 - \frac{|D(\varsigma(s))|}{\nu}\right)\left(\frac{\| \varsigma(s) \|_2^2}{\tau(D)}\right). \tag{6.5}$$

The first term is based on the idea of ball-heuristic. It measures the accuracy based on how many points exist in the cell (after normalizing it with the maximum number of points allowed in each cell). The second term is based on the size of the cell; smaller cells have better accuracies (smaller ϵ's). The normalizing size of the tree is used in the denominator as a mean to provide dependency on the sample size. The more time is passed (bigger sample size), the smaller the normalizing size becomes. Therefore, given a fixed cell size, the value of $\epsilon(x_t)$ will get bigger as $|D|$ increases.

The second heuristic is based directly on the bonus values used in regret-minimization algorithms for k-armed bandit problems. These algorithms add a bonus value of the form $\sqrt{\frac{\ln(t)}{n_a}}$ to their estimates of each arm's payoff function, where t is the current time and n_a is the number of times arm a is pulled (Auer et al., 2002; Bubeck et al., 2008; Kocsis and Szepesvári, 2006). It has been shown that this type of bonus function results in an exploration strategy that pulls each arm infinitely many times, but pulls the optimal arm exponentially more often than the others.

The second uncertainty tree heuristic uses the same function to add uncertainty to its estimates. Suppose ζ_a is used for estimating $T(.,a)$ using D_a. The accuracy is computed as:

$$\epsilon(x_t,a) = \left(1 - \frac{|D(\varsigma_a(s))|}{\nu}\right) + C_T \parallel \varsigma_a(s) \parallel_2^2 + \sqrt{\frac{\ln(|D(\varsigma(x_t))|)}{|D_a(\varsigma_a(x_t))|}}, \tag{6.6}$$

where $|D(\varsigma_a(x_t))|$ is the number of training points in $\varsigma_a(x_t)$ and $|D(\varsigma(x_t)|$ is the sum of $|D(\varsigma_i(x_t))|$ for all i's. In other words, the bonus in the third term is computed based on the total number of samples the algorithm has seen so far from the neighborhood of x_t and the number of times it has selected to execute a. The first two term of this function resemble the accuracy estimate of ball-heuristic.

6.2 Proposed Algorithm

This section constructs an RL algorithm that uses uncertainty trees to maintain an internal model of the world. This algorithm, which is called *Multi-resolution Exploration*

(MRE), is very similar to CKWIK-Rmax because they both use the knownness function to provide exploration bonuses to those state-action pairs with uncertain transition-function estimates. The only difference between the two algorithms is that MRE does not receive ϵ and δ from the user, and therefore does not work in the PAC-MDP framework. As a result of this change, the transition-function estimator does not receive the previously-used δ_t as an allowed probability of failure. Also, the knownness function is computed differently because no ϵ_T is available. The new knownness function is computed as:

$$\psi(s,a) \stackrel{\text{def}}{=} \max\left(1 - \epsilon(s,a), 0\right), \tag{6.7}$$

where $\epsilon(s,a)$ is provided by the transition function estimator. MRE uses $|\mathcal{A}|$ uncertainty trees for estimating the transition function—denoted by ζ_a. Once the knownness is computed, the algorithm uses the same technique from CKWIK-Rmax to build its internal model. In particular, it constructs its internal model of the world as $\hat{M} = \langle \mathcal{S} + s_f, \mathcal{A}, \hat{T}', R, \gamma \rangle$, where ζ_a is used to create $\hat{T}$ and the augmented transition function $\hat{T}'$ is computed from $\hat{T}$ as:

$$\hat{T}'(s'|s,a) = \begin{cases} 1 - \psi(s,a), & \text{if } s' = s^f \\ \psi(s,a)\hat{T}(s'|s,a), & \text{otherwise.} \end{cases} \tag{6.8}$$

A sketch of MRE is provided in Algorithm 14.

Algorithm 14 MRE, a model-based algorithm for continuous state space MDPs.

1: Initialize uncertainty trees ζ_a for all a's.
2: **for** all timesteps $t = 1, 2, \ldots$ **do**
3: Observe s_t and r_t, execute action $a_t = \pi^*_{\hat{M}}(s_t)$, transition to s_{t+1}.
4: Update ζ_a using (s_t, s_{t+1}).
5: Update internal MDP $\hat{M}$ using the augmented transition function in Eqn. 6.8.
6: **end for**

6.3 Discussion

The two main properties of MRE are its multi-resolution discretization that allows it to apply variable-resolution generalization across the state space and its dynamic accuracy measure that allows it to keep exploring the environment as more experience is obtained. This method shares many positive properties with existing algorithms. For example, if the dynamic accuracy measure, which is dependent on time, is replaced with something similar to the ball-heuristic, the algorithm will be similar to CF-Rmax, because it works in the CKWIK framework (maintains a continuous knownness) and applies a variable degree of generalization across the state space. Furthermore, if the variable discretization in MRE is also replaced with a uniform discretization, the algorithm becomes similar to MBIE (Strehl and Littman, 2005) for finite spaces. This similarity is a result of MBIE being the only previously developed algorithm in the KWIK framework that used all the available data instead of forming a binary distinction between known/unknown states. Finally, if the knownness in MRE is forced into a binary output, the algorithm will become similar to metric E^3 (Kakade et al., 2003) and fitted-Rmax.

6.3.1 Application to Value-based RL

The idea of multi-resolution exploration can also be used in other approaches like value-based methods. This section explains how to incorporate uncertainty trees into a value-based algorithm called *fitted Q-iteration*. This algorithm is an offline batch learner, but it has been integrated with other exploration mechanisms like ϵ-greedy or Rmax-type exploration to work as an online algorithm (Li et al., 2009). The same procedure can be used to apply this exploration strategy to other value-based methods, such as LSPI (Lagoudakis and Parr, 2003).

The fitted Q-iteration algorithm accepts a set of four-tuple samples:

$$S = \{(s^l, a^l, r^l, s'^l), l = 1..n\}$$

and uses regression trees to iteratively compute increasingly accurate $\hat{Q}$-functions. In particular, let $\hat{Q}_i^j$ be the regression tree used to approximate $Q(\cdot, j)$ in the i-th iteration. Let $S^j \subset S$ be the set of samples with action equal to j. The training samples for $\hat{Q}_0^j$ are $S_0^j = \{(s^l, r^l) | (s^l, a^l, r^l, s'^l) \in S^j\}$. $\hat{Q}_{i+1}^j$ is constructed based on $\hat{Q}_i$ in the following way:

$$\begin{aligned} x^l &= \{s^l | (s^l, a^l, r^l, s'^l) \in S^j\} \\ y^l &= \{r^l + \gamma \max_{a \in A} \hat{Q}_i^a(s'^l) | (s^l, a^l, r^l, s'^l) \in S^j\} \\ S_{i+1}^j &= \{(x^l, y^l)\}. \end{aligned} \tag{6.9}$$

Random sampling is usually used to collect S for fitted Q-iteration when used as an offline algorithm. In online settings, ϵ-greedy can be used as the exploration scheme to collect samples. The batch portion of the algorithm is applied periodically to incorporate the new collected samples.

Combining uncertainty trees with fitted Q-iteration is very simple. Let ζ_j correspond to $\hat{Q}_i^j$ for all i's, and be trained on the same samples. The only change in the algorithm is the computation of Equation 6.9. To use optimistic values, we elevate $\hat{Q}$-functions according to their knownness:

$$y^l = \psi(s^l, j)\Big(r^l + \gamma \max_{a \in A} Q_i^a(s'^l)\Big) + \Big(1 - \psi(s^l, j)\Big) V_{\max}.$$

6.4 Experimental Results

The first experiment was performed on MOUNTAINCAR and compared MRE against two other algorithms that used fixed-length discretizations instead of uncertainty trees. One of these algorithms had a fine discretization (normalized length of 0.05), whereas the other used a coarse one (normalized length of 0.3). MRE applied the first heuristic defined earlier and set the maximum number of points in each cell to 10. FVI was used as the planner and was called every 50 steps. The learning was conducted in 200

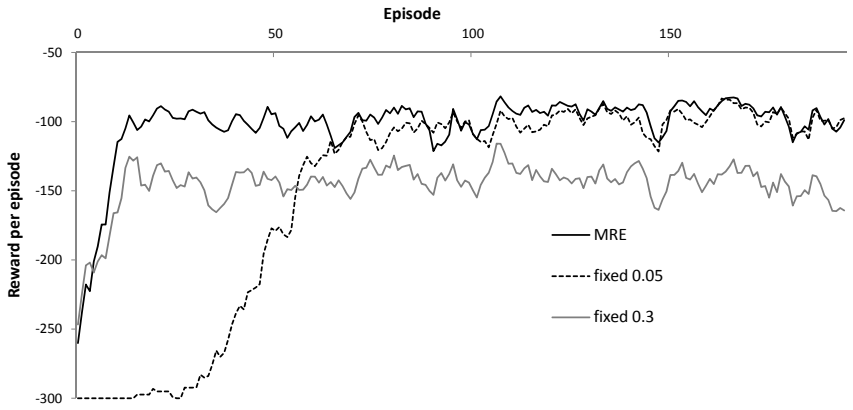


Figure 6.3: Performance of three algorithms in MOUNTAINCAR: MRE and two algorithms based on fine and coarse discretizations.

episodes, where each episode had a cap of 300 steps. The results were averaged over 20 runs and smoothed over a window of size 5 to avoid a cluttered graph.

The learning curves of these algorithms are shown in Figure 6.3. The algorithm with finer fixed discretization converged to a very good policy, but took a long time to do so because it trusted only very accurate estimations throughout the learning. The one with coarse discretization, on the other hand, converged very fast, but not to a very good policy; it constructed rough estimations early on and did not compensate as more samples were collected. Since MRE refined the notion of knownness over time, it managed to make rough estimations at the beginning and accurate ones later on. Therefore, it quickly converged to a good policy.

A more detailed comparison of this result is available in Figure 6.4, where the average performance of each algorithm is provided for three different periods: at the early stages of learning (episode $1 - 100$), in the middle of learning (episode $100 - 200$), and near the end (episode $200 - 300$). Standard deviation is used as the error bar. Performance of MRE was very close to the one with fine resolution in the last stage because they both had learned the optimal policy; however, it performed much better in the first period. Also, it learned as fast as the one with coarse discretization in the beginning, but managed to converge to the optimal policy as more samples were gathered.

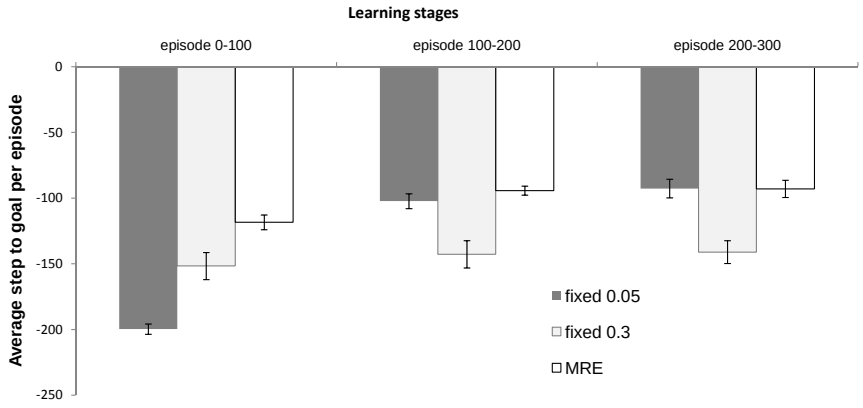


Figure 6.4: Performance of three algorithms in MOUNTAINCAR broken down into three stages of learning. A fine resolution discretization learns the optimal policy, but takes a long time. A coarse discretization learns fast, but cannot make good estimates. MRE learns fast and converges to good estimates.

To take a closer look at why MRE performed better than the one with fine resolution during the early stages of learning (note that both of them achieved the same performance level at the end), the two algorithms' value functions were examined at $t = 1500$ (Figure 6.5).

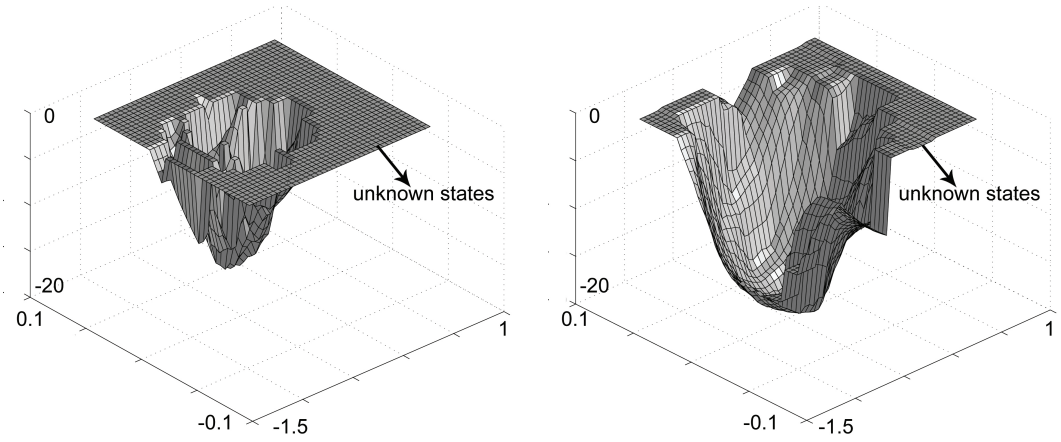


Figure 6.5: Snapshot of the value function at timestep 1500 in MRE and another algorithm that used a fixed discretization of size 0.05 in MOUNTAINCAR.

In the algorithm with fixed discretization, a large portion of the state space was unknown. These were the states with 0 values (the flat surface on the top of the value function). The reason for having a lot of states with 0 knownness was the narrow

generalization of the algorithm. MRE, on the other hand, achieved a much more realistic and smooth value function by allowing coarser generalizations in parts of the state space with fewer samples.

A similar experiment was also tried with fitted Q-iteration and compared different exploration strategies and their effects on the algorithm performance. Three exploration techniques were tested: ϵ-greedy and two versions of uncertainty trees. The first one, denoted by $\text{UT}_{\text{continuous}}$, computed the knownness based on the second heuristic we discussed for uncertainty trees. The second one, denoted by $\text{UT}_{\text{boolean}}$, used a threshold C to convert the knownness value to a Boolean function similar to the algorithms in the KWIK framework.

For ϵ-greedy, the value of ϵ was set to 0.3 at the beginning and decayed linearly to 0.03 at $t = 10000$, and kept constant afterward. The constant C was set to 0.8 in $\text{UT}_{\text{boolean}}$. These parameter settings were hand-tuned by a rough optimization, through a few trial and error runs.

The result of this experiment is shown in Figure 6.6. As expected, ϵ-greedy performed poorly, as it could not collect good samples to feed the batch learner. Both versions of uncertainty tree converged to the same policy at the end, although the one that used continuous knownness did it faster.

For a better understanding of why the continuous knownness helped fitted Q-iteration at the early stages of learning, snapshots of the knownness functions from the two versions were compared to each other at timestep 1500.

Figure 6.7 shows the knownness functions of action *right* computed in each algorithm, along with the set of visited states. Black indicates a completely unknown region, while white means completely known; gray is used for intermediate values.

Although $\text{UT}_{\text{boolean}}$ used a variable discretization, it had to output a Boolean function, so it ran into the same data-inefficiency issues that fitted-Rmax did (*c.f.* (§ 3.1.2)). In particular, the set of visited states in $\text{UT}_{\text{continuous}}$ had a better covering of the whole space, including several trajectories to the goal region. We believe this covering of the state space helped $\text{UT}_{\text{continuous}}$ to perform better than the other version.

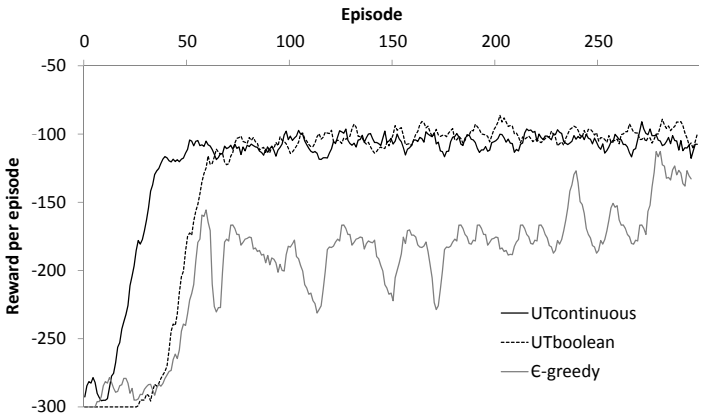
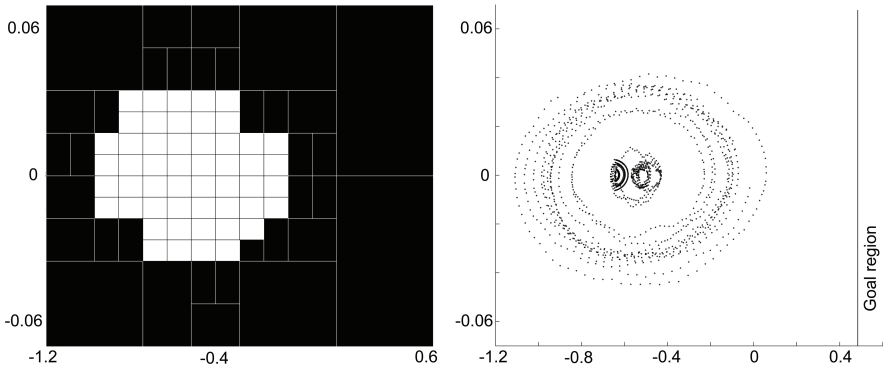


Figure 6.6: Comparison of three different exploration strategies for fitted Q-iteration tested in MOUNTAINCAR.

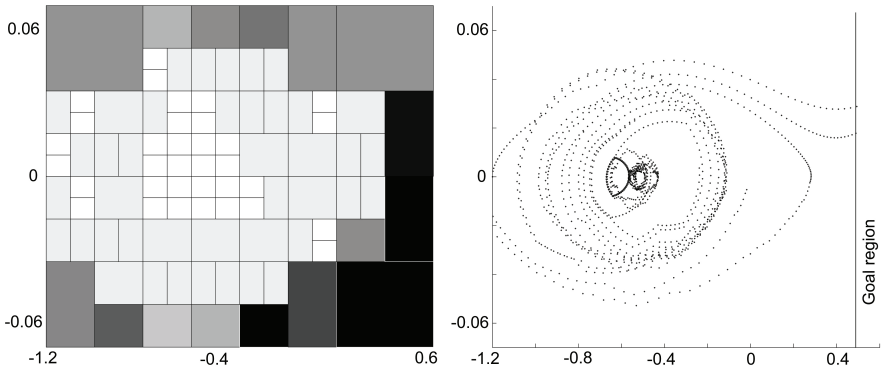
6.5 Conclusion

This chapter showed that the same knownness concept can be used to derive a hierarchical exploration strategy, which encourages *anytime* behavior. This development of showed that the knownness idea can be used to expand the effectiveness of model-based exploration to beyond what has been achieved by the PAC-MDP framework.

This algorithm achieved anytime behavior by forming a multi-resolution discretization of the state space and variable levels of generalization. These features, which were made possible by using the continuous knownness metric, allowed the agent to make very accurate predictions in parts of the state space with a lot of samples (that is, using a fine discretization), while still managing to use data efficiently in other parts of the state space with sparser data (using coarse discretization). This chapter also showed how to use the same knownness idea in a value-based algorithm. Although this technique could not be used to derive a PAC-MDP bound in the case of a value-based algorithm, empirical evaluations showed that it can be used to boost up the performance of many existing value-based algorithms that are often used with primitive exploration schemes.



(a) $\mathrm{UT}_{\mathrm{boolean}}$



(b) $\mathrm{UT}_{\mathrm{continuous}}$

Figure 6.7: Knownness function computed in two versions of uncertainty trees with fitted Q-iteration: one that outputs Boolean values, and one that works with continuous ones. Black indicates completely unknown and white means completely known. Collected samples are also shown for the same two versions at timestep 1500.

Chapter 7

Concluding Remarks

This dissertation considered continuous-state control problems and introduced a family of new techniques for efficiently balancing exploration and exploitation in model-based algorithms. At the heart of these methods was a new learning framework called CKWIK. This learning paradigm introduced a new methodology for representing uncertainty in the model estimation part of these algorithms, which could be used to achieve a more data-efficient exploration of the environment. These algorithms represented uncertainty using a concept called *knownness*, which was a continuous variable in the range of 0 to 1 indicating how much the algorithm would know about the transition function of each state-action pair.

Several algorithms were introduced that used this uncertainty measure to accomplish more data-efficient learning. The first algorithm to use the CKWIK framework was called CF-Rmax and was a continuation of an existing algorithm from the Rmax-exploration family called fitted-Rmax. This algorithm targeted a broad class of continuous state-space MDPs and was shown to have better performance than its sister algorithm in practice. A formal analysis of the sample complexity of this algorithm was also provided. Two more methods were introduced based on CF-Rmax, which targeted a smaller class of environments. These algorithms, called FF-Rmax and DRE, were designed to achieve even better performance in the case where the environment could be represented in a more compact way. FF-Rmax achieved this speedup by using the domain knowledge provided by the user *a priori*. DRE accomplished the same goal by automatically discovering these compact representations without the help of the user. Finally, an algorithm was introduced that used the same knownness concept to derive a hierarchical exploration strategy. It was shown that this exploration scheme insists

on continuously exploring the environment while making efficient use of available data. These behaviors are especially beneficial in real-life situations where the user does not know *a priori* the quantity of resources he/she has during the learning.

Appendix A

Proofs

This appendix contains the proofs of the theorems that either were lengthy or tangent to the flow of the dissertation, and thus were removed from the main content.

A.1 Proof of Theorem 14

We first state the theorem one more time:

Theorem 21. *Let $\mathcal{M}$ be a class of MDPs and assume that $\mathcal{F}_T$ is a CKWIK-learner of the transition function of $\mathcal{M}$ with a bound of $B(\epsilon, \delta, \mathcal{S}, \mathcal{A})$. Then, CKWIK-Rmax is PAC-MDP with a sample complexity of:*

$$\mathcal{O}\left(\frac{V_{\max}}{\epsilon(1-\gamma)}\left(B(\epsilon(1-\gamma)/V_{\max}, \delta, \mathcal{S}, \mathcal{A}) + \ln\frac{1}{\delta}\right)\ln\frac{1}{\epsilon(1-\gamma)}\right),$$

provided that the accuracy parameters for $\mathcal{F}_T$ are set as:

$$\epsilon_T = \Theta(\epsilon(1-\delta)^2), \quad \delta_T = \Theta(\delta).$$

The proof relies on a general PAC-MDP result from Strehl et al. (2006), which is provided here as a reference:

Theorem 22. *Let $\mathcal{A}(\epsilon, \delta)$ be an algorithm that takes ϵ and δ as inputs (in addition to other algorithm-specific inputs), acts greedily according to its estimated state-action value function, denoted Q_t at timestep t. Define $V_t(s) \stackrel{def}{=} \text{argmax}_{a \in A} Q_t(s, a)$. Suppose that on every timestep t, there exists a set K_t of state-acion pairs that depends only on the agent's history up to timestep t. We assume that $K_t = K_{t+1}$ unless, during timestep t, an update to some state-action value occurs or the escape event A_K happens. Let*

M_{K_t} be an MDP that has the same transition and reward function as M on set K_t, and let π_t be the greedy policy with respect to Q_t. Suppose that for any inputs ϵ and δ, with probability at least $1 - delta/2$, the following conditions hold for all timesteps t:

1. *(Optimism) $V_t(s_t) \geq V^*(s_t) - \epsilon/4$,*
2. *(Accuracy) $V_t(s_t) - V^{\pi_t}_{M_{K_t}}(s_t) \leq \epsilon/4$, and*
3. *(Bounded surprises) The total number of updates of action-value estimates plus the number of times the escape event from K_t, A_K, can occur is bounded by a function $\zeta(\epsilon, \delta)$. The function ζ may depend on $|M|$.*

Then, with probability at least $1 - \delta$, the sample complexity of exploration of $\mathcal{A}(\epsilon, \delta)$ is

$$\mathcal{O}\left(\frac{V_{\max}}{\epsilon(1-\delta)}\left(\zeta(\epsilon,\delta) + \ln\frac{1}{\delta}\right)\ln\frac{1}{\epsilon(1-\delta)}\right).$$

We use the above theorem by defining the set K_t as follows:

Definition 23. *Let $M = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ be an MDP. At timestep t of the execution of CKWIK-Rmax, define the set K_t of state-action pairs as follows:*

$$K_t \stackrel{def}{=} \{(s,a) \in \mathcal{S} \times \mathcal{A} | \epsilon(s,a) \leq \epsilon_T\}.$$

We first provide a version of the simulation lemma (Li, 2009; Lemma 33), and then verify the three conditions in Theorem 22 in Lemmas **??**, **??**, and **??** to show that CKWIK-Rmax is PAC-MDP. We choose the following parameter values:

$$\epsilon_T = \frac{\epsilon(1-\gamma)}{\gamma V_{\max}}, \qquad \delta_T = \delta/4 \tag{A.1}$$

Lemma 24. *(Simulation Lemma) Let $M_1 = \langle \mathcal{S}, \mathcal{A}, T_1, R, \gamma \rangle$ and $M_2 = \langle \mathcal{S}, \mathcal{A}, T_2, R, \gamma \rangle$ be two MDPs with the same state/action spaces and discount factor. Let Q_1^* and Q_2^* (V_1^* and V_2^*) be their optimal state-action value (state-value) functions, respectively.*

Assume the two transition functions are close in the following sense: there exists a constant ϵ_T, such that for every (s, a), we have:

$$|T_1(\cdot|s,a) - T_2(\cdot|s,a)| \leq \epsilon_T.$$

Then, for any $s \in \mathcal{S}$ and $a \in \mathcal{A}$, we have:

$$|Q_1^*(s,a) - Q_2^*(s,a)| \leq \frac{\gamma V_{\max} \epsilon_T}{1-\gamma}$$
$$|V_1^*(s) - V_2^*(s)| \leq \frac{\gamma V_{\max} \epsilon_T}{1-\gamma}.$$

Proof. Define the *Bellman operators*, $\mathcal{B}_1$ and $\mathcal{B}_2$, for M_1 and M_2, respectively: for $i = 1, 2$ and any state-action value function $Q \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$,

$$\mathcal{B}_i Q(s,a) \stackrel{\text{def}}{=} R(s,a) + \gamma \int_{s' \in \mathcal{S}} T_i(s'|s,a) \sup_{a' \in \mathcal{A}} Q(s',a') ds'.$$

It is known that Q_i^* is the *fixed point* of $\mathcal{B}_i$: $\mathcal{B}_i Q_i^* = Q_i^*$. Define two errors: the ℓ_∞ approximation error $e = \| Q_1^* - Q_2^* \|_\infty$ and the ℓ_∞ Bellman backup error $b = \| \mathcal{B}_1 Q_2^* - \mathcal{B}_2 Q_2^* \|_\infty$. Then,

$$\begin{aligned} e =& \| \mathcal{B}_1 Q_1^* - \mathcal{B}_2 Q_2^* \|_\infty \\ \leq& \| \mathcal{B}_1 Q_1^* - \mathcal{B}_1 Q_2^* \|_\infty + \| \mathcal{B}_1 Q_2^* - \mathcal{B}_2 Q_2^* \|_\infty \\ \leq& \gamma \| Q_1^* - Q_2^* \|_\infty + \| \mathcal{B}_1 Q_2^* - \mathcal{B}_2 Q_2^* \|_\infty \\ =& \gamma e + b, \end{aligned} \tag{A.2}$$

where the first step is due to the fixed-point property of $\mathcal{B}_i$, the second due to the triangle inequality, the third due to the contraction property of $\mathcal{B}_i$, and the last due to the definitions of e and b. It follows immediately that $(1-\gamma)e \leq b$, and so

$$e \leq \frac{b}{1-\gamma}. \tag{A.3}$$

We now give an upper bound for b:

$$\begin{aligned}
b &= \sup_{s,a} |\mathcal{B}_1 Q_2^*(s,a) - \mathcal{B}_2 Q_2^*(s,a)| \\
&= \sup_{s,a} |\gamma \int_{\mathcal{S}} (T_1(s'|s,a) - T_2(s'|s,a)) \sup_{a'} Q_2^*(s',a') ds'| \\
&\leq \gamma \sup_{s,a} \int_{\mathcal{S}} |T_1(s'|s,a) - T_2(s'|s,a))| \sup_{a'} |Q_2^*(s',a')| ds' \\
&\leq \gamma V_{\max} \sup_{s,a} \int_{\mathcal{S}} |T_1(s'|s,a) - T_2(s'|s,a)| ds' \\
&\leq \gamma V_{\max} \epsilon_T, .
\end{aligned}$$

Combining this result with Equation A.3, we have for all (s,a) that

$$|Q_1^*(s,a) - Q_2^*(s,a)| \leq e \leq \frac{b}{1-\gamma} \leq \frac{\gamma V_{\max} \epsilon_T}{1-\gamma}.$$

The second part of the lemma follows immediate based on the relationship between the value function and state-action value functions. □

Lemma 25. *With probability at least* $1 - \delta/2$, $Q_t(s,a) \geq Q^*(s,a) - \epsilon/4$ *for all* t *and* (s,a).

Proof. Let $\hat{M}_t$ be the internal MDP of the algorithm at time t. Also, let M_{K_t} be the known state-action MDP, where K_t is given in Definition 23. The transition function in M_{K_t} agrees with $\hat{M}_t$ for state-action pairs outside K_t and with M for the pairs inside K_t. Since the transition function of $\hat{M}_t$ for state-action pairs in K_t are accurate with probability at least $\delta_T = \delta/4$, using Lemma 24 we get:

$$|Q_{\hat{M}_t}(s,a) - Q^*_{M_{K_t}}(s,a)| \leq \frac{\gamma V_{\max} \epsilon_T}{1-\gamma}. \tag{A.4}$$

On the other hand, according to the construction of M_{K_t} and since it is identical to M for all the state-action pairs inside K_t and the assumption about the $\mathcal{F}_T$, we have that the optimal state-action value function of M_{K_t} is optimistic with probability at least

$1-\delta/4$. Combining the inequalities, we have:

$$\begin{aligned} Q_t(s,a) - Q^*(s,a) &\geq Q^*_{\hat{M}_t}(s,a) - Q^*(s,a) \\ &\geq Q^*_{M_{K_t}}(s,a) - \frac{\gamma V_{\max}\epsilon_T}{1-\gamma} - Q^*(s,a) \\ &\geq -\frac{\gamma V_{\max}\epsilon_T}{1-\gamma}. \end{aligned} \tag{A.5}$$

with probability at least $1-\delta/2$. The parameter values chosen in Equation A.1 satisfies the requirement of the lemma. □

Lemma 26. *The total number of timesteps in which Q_t changes or a state outside K_t is visited, denoted by $\zeta(\epsilon,\delta)$, is at most $B_T(\epsilon_T,\delta_T)$.*

Proof. Since Q_t is only changed when K_t changes (the value of $\epsilon(s,a)$ for at least one state-action pairs becomes less than ϵ_T), we may only bound the number of timesteps in which the outcome of $\mathcal{F}_T$ for a query point has $\epsilon(s,a)$ greater than ϵ_T when δ_T is provided, which is indeed $B_T(\epsilon_T,\delta_T)$. □

The proof of Theorem 14 is completed by using the previous lemmas and the result of Theorem 22.

A.2 Proof of Lemma 17

This section provides the proof of lemma 17 in Chapter **??**, which was skipped for making the flow of the chapter more coherent. The lemma is restated here:

Lemma 27. *Given a training data D_a, a query point s^*, and an $\frac{\epsilon_1}{2C_T}$-ball around s^*, if c_1 points from D_a are in the $\frac{\epsilon_1}{2C_T}$-ball, where c_1 is:*

$$c_1 > \frac{18}{\sqrt{\frac{18\epsilon_1^2}{\ln(2/\delta)}+1}-1} + 1, \tag{A.6}$$

the accuracy of CKWIK-KR is bounded by:

$$\| \mathcal{F}_a(s^*; D_a) - \mu^* \|_2^2 \leq \epsilon_1, \tag{A.7}$$

with probability at least $(1-\delta)$ *for any input* D_a*, provided that the kernel width* σ *is larger than* $\sqrt{\frac{\epsilon_1}{2C_T}}$.

Proof (of Lemma 17). For a query point x^*, fix the training data D_a. We can write the accuracy of kernel regression as:

$$\| \mathcal{F}_a(x^*; D_a) \|_2^2 = \min_c \mathcal{L}(s^*, D_a, c).$$

In particular, this equation holds for $c = c_1$:

$$\begin{aligned} \| \mathcal{F}_a(x^*; D_a) \|_2^2 &\leq \mathcal{L}(s^*, D_a, c_1) \\ &\leq \sqrt{\frac{\ln(2/\delta) \sum_{s_j \in \mathcal{N}_{c_1}(s^*)} k(s_j, s^*)^2}{2[\sum_{s_j \in \mathcal{N}_{c_1}(s^*)} k(s_j, s^*)]^2}} + \frac{C_T}{c_1} \sum_{i=1}^{c_1} \| s_i - s^* \|_2^2 \qquad \text{(A.8)} \\ &\leq \mathcal{L}_1 + \mathcal{L}_2, \qquad \text{(A.9)} \end{aligned}$$

where $\mathcal{L}_1$ and $\mathcal{L}_2$ denote the loss due to the corresponding terms of the previous sum. According to the assumption, at least c_1 points from D_a lie in the $\frac{\epsilon_1}{2C_T}$-ball around s^*. We need to bound the loss function for any such D_a. To achieve this objective, we bound each term in the above inequality separately. For any D_a that has c_1 points in the $\frac{\epsilon_1}{2C_T}$-ball around s^*, $\mathcal{L}_2$ is bounded by:

$$\begin{aligned} \mathcal{L}_2 &= \frac{C_T}{c_1} \sum_{i=1}^{c_1} \| s_i - s^* \|_2^2 \\ &\leq \frac{C_T}{c_1} \sum_{i=1}^{c_1} \frac{\epsilon_1}{2C_T} \\ &\leq \frac{\epsilon_1}{2}. \end{aligned} \tag{A.10}$$

We now bound the first term. Let k_j be defined as:

$$k_i = \frac{k(s_i, s^*)}{\sum_{s_j \in \mathcal{N}_{c_1}(s^*)} k(s_j, s^*)}.$$

The maximum loss of the first term is:

$$\max_{D_a} \sqrt{\frac{\ln(2/\delta)}{2} \cdot \sum_{i=1}^{c_1} (k_i)^2}, \quad \text{//subject to having } c_1 \text{ points in the ball.} \tag{A.11}$$

It is straightforward to show that this term is maximized when one of the points equals s^* and the rest are located at the border of the ball. Without loss of generality, suppose $s_1 = s^*$ and let $s_{i>1}$ be the points located at the boundary of the ball. We have:

$$k_{i>1} = \frac{\frac{1}{\sigma\sqrt{2\pi}} \cdot \exp(-\frac{\epsilon_1}{2C_T\sigma^2})}{\sum k(s_j, s^*)} \tag{A.12}$$

$$= \frac{\frac{1}{\sigma\sqrt{2\pi}} \cdot \exp(-\frac{\epsilon_1}{2C_T\sigma^2})}{(c_1 - 1)\left(\frac{1}{\sigma\sqrt{2\pi}} \cdot \exp(-\frac{\epsilon_1}{2C_T\sigma^2})\right) + \frac{1}{\sigma\sqrt{2\pi}}} \tag{A.13}$$

$$= \frac{\exp(-\frac{\epsilon_1}{2C_T\sigma^2})}{(c_1 - 1)\exp(-\frac{\epsilon_1}{2C_T\sigma^2}) + 1} \tag{A.14}$$

$$= \frac{1}{(c-1) + \exp(\frac{\epsilon_1}{2C_T\sigma^2})}. \tag{A.15}$$

Perform a variable exchange of f for $\exp(\frac{\epsilon_1}{2C_T\sigma^2})$. We get:

$$k_{i>1} = \frac{1}{(c_1 - 1) + f}. \tag{A.16}$$

Furthermore, if we denote $k_{i>1}$ by w, we can write k_1 using w:

$$k_1 = (1 - (c_1 - 1)w).$$

Now, let us rewrite $\mathcal{L}_1$ using these recent developments:

$$(\mathcal{L}_1)^2 = \max_{D_a} \frac{\ln(2/\delta)}{2} \cdot \sum_{i=1}^{c_1} (k_i)^2 \tag{A.17}$$

$$= \frac{\ln(2/\delta)}{2} \cdot \left((c_1 - 1)w^2 + (1 - (c_1 - 1)w)^2\right) \tag{A.18}$$

$$= \frac{\ln(2/\delta)}{2} \cdot \left[\frac{(c_1 - 1)}{[(c_1 - 1) + f]^2} + \left(\frac{f}{(c_1 - 1) + f}\right)^2\right] \tag{A.19}$$

$$= \frac{\ln(2/\delta)}{2} \cdot \left[\frac{x}{[x + f]^2} + \left(\frac{f}{x + f}\right)^2\right] \tag{A.20}$$

$$\leq \frac{\ln(2/\delta)}{2} \cdot \left[\frac{1}{x} + \left(\frac{3}{x}\right)^2\right], \tag{A.21}$$

where the last step follows because of our assumption that $f \leq 3$. The last function is strictly decreasing w.r.t. x, so its maximum occurs for the minimum value of x. But, the minimum value of x (or $c - 1$) is bounded according to our assumptions. Plugging in the value of c_1 in the equation, we get:

$$(\mathcal{L}_1)^2 \leq \frac{\ln(2/\delta)}{2} \cdot \left[\frac{\sqrt{\frac{18\epsilon_1^2}{\ln(2/\delta)} + 1} - 1}{18} + \left(\frac{\sqrt{\frac{18\epsilon_1^2}{\ln(2/\delta)} + 1} - 1}{6}\right)^2\right] \tag{A.22}$$

$$\leq \left(\frac{\epsilon_1}{2}\right)^2 \qquad \text{//simplifying the equation.} \tag{A.23}$$

Therefore, we have $\mathcal{L}_1 \leq \frac{\epsilon_1}{2}$.

Putting it all back together, the following holds with probability at least $(1 - \delta)$:

$$\| \mathcal{F}_a(x^*; D_a) \|_2^2 \leq \mathcal{L}(s^*, D_a, c_1) \tag{A.24}$$

$$\leq \mathcal{L}_1 + \mathcal{L}_2 \tag{A.25}$$

$$\leq \frac{\epsilon_1}{2} + \frac{\epsilon_1}{2} = \epsilon_1. \tag{A.26}$$

□

Appendix B

Planning in Markov Decision Processes

Planning in an MDP refers to the act of computing the optimal policy when the description of the MDP is given. Planning is closely related to reinforcement learning because they both have the same goal of behaving optimally in an environment (which is modeled by an MDP). Furthermore, planning is an important sub-step in all model-based algorithms as explained in Figure 2.4.

In all the algorithms that were developed in this dissertation, the reader was left to decide how to perform the planning step using the internal model. In practice, planning is typically computationally challenging and an approximation to the optimal policy needs to be made, especially in continuous spaces. Sine the performance of any model-based algorithm is highly dependent on the accuracy of its planner (Li, 2009), this chapter surveys some useful techniques for planning in continuous state-space MDPs.

B.1 Value Iteration

Value iteration is one of the oldest and simplest algorithms for solving an MDP (Puterman, 1994). Although it is designed to solve finite MDPs, it is interesting to study here because it is the core of most of the sophisticated techniques for solving both finite and continuous MDPs. Value iteration maintains estimates of value function (or action-value function) and is therefore a value-based method.

Starting with an estimate of the value function V_0, it repetitively uses the Bellman operator to improve the estimate. Alternatively, the algorithm can maintain the Q-function during the whole process. Algorithm 15 shows the pseudo-code for Value iteration.

Algorithm 15 Value Iteration: A planning algorithm for finite MDPs.

1: **Inputs:** $M = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$.
2: Initialize $V(s) = V_{\max}$.
3: **for** t=1, 2, ... **do**
4: **for** all $s \in \mathcal{S}$ **do**
5: Update $V(s)$ according to Bellman equation (2.5).
6: **end for**
7: **end for**

There are two important properties of value iteration that are worth mentioning. First, it can be shown that as t goes to infinity, V_t converges to the optimal value function. This convergence is independent of the value used in the initialization step at line 2—we could have used any value instead of $V_{\max}$. Second, the sequence of value function estimates V_t approaches the optimal value function at a geometric rate γ:

$$\| V_{t+1} - V^* \|_\infty \leq \gamma \| V_t - V^* \|_\infty . \tag{B.1}$$

This result is due to an important property of the Bellman operator called *contraction*. This property relates the difference of two functions before and after an operator is performed on them. In particular, Bellman operator $\mathcal{B}$ is a γ-contraction because:

$$\| \mathcal{B}V_1 - \mathcal{B}V_2 \|_\infty \leq \gamma \| V_1 - V_2 \|_\infty . \tag{B.2}$$

The geometric convergence is obtained simply by observing that V^* is the fixed point of the Bellman operator: $\mathcal{B}V^* = V^*$. Given the geometric convergence, it is important to have a starting estimate V_0 that is as close to V^* as possible to ensure the algorithm converges to the optimal value function faster. Although the convergence to V^* occurs only in the limit, one does not need to compute the exact value function to find an optimal policy. In fact, it can be shown that value iteration converges to the optimal policy in finite time (Puterman, 1994). One way to detect convergence is to stop the algorithm when $\| V_{t+1} - V_t \|_\infty$ drops below a certain threshold (Williams and Baird, 1993).

There are many variations of the value iteration algorithm. A large number of

algorithms known as *asynchronous value iteration* (Bertsekas, 1982) replace the inner loop of Algorithm 15 by something that updates the value of only one state. In other words, these algorithms select a state according to some strategy at each iteration and update its value using the Bellman equation. Different algorithms adopt different strategies for selecting states in each iteration. For example, prioritized sweeping (Moore and Atkeson, 1993; Peng and Williams, 1993) updates states based on a priority queue maintained over the states. States have more priority if their Bellman error (*aka* $|V_{t+1} - V_t|$) is higher. Another example is real-time dynamic programming (RTDP) (Barto et al., 1995), which updates the value of states that are actually visited by an agent (forming an online planning problem).

Value iteration can be extended to the situations where the complete description of the transition function is not available. If access to the transition function is only provided through samples from the transition function, the model is called a *generative* one because we can generate samples from the transition function, but do not have access to the description of the entire probability distribution. The Bellman backup in value iteration cannot be applied in this case because we do not have access to $T(s'|s,a)$. Sample-based value iteration, described in Algorithm 16, uses C samples from the transition function of each state-action pair to estimate the Bellman backup operator.

Algorithm 16 Sample-based Value Iteration, a planning algorithm using generative MDP models.

Inputs: $M = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$, C.
Initialize $V(s) = V_{\max}$.
for t=1, 2, ... **do**
  for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ **do**
    Generate C samples from the transition function of (s,a): $s'_i \sim T(s,a), i = 1, \ldots, C$.
    Update the Q-function of (s,a) as follows:

$$Q(s,a) = R(s) + \frac{\gamma}{C} \sum_{i=1}^{C} V(s'_i). \tag{B.3}$$

  end for
end for

B.2 Approximate Planning in Continuous Spaces

The value iteration algorithm, as well as other exact planning algorithms, can only be used in finite MDPs with relatively few number of states because they store state values in lookup tables. Most of the finite MDPs that are of practical interest have too many states to be presentable this way. A classical example of this is the popular game of GO, which is estimated to have 2.08×10^{170} different legal board configurations. Continuous MDPs have infinite number of states by nature, which effectively eliminates the possibility of using lookup table to store the value function. In these domains, except for few special cases, neither the exact computation of the the Bellman operator, nor the exact representation of the resulting functions is possible. Therefore, approximate planning techniques are required for computing the optimal policy in these MDPs. Some of these techniques are presented next.

B.2.1 Discretization

Abstraction is a widely-used technique in artificial intelligence to make compact representations of large problems (Giunchiglia and Walsh, 1992). Abstraction has also been extensively studied in the reinforcement-learning community. In an MDP framework, abstraction is any mechanism that provides compact representation of a quantity related to the MDP. In particular, *state abstraction* aggregates the states into a (smaller) set with the hope of constructing a more compact representation of functions defined over the state space (such as value function or policy). A comprehensive list of different state abstractions and their properties can be found in (Li et al., 2006). Some planning algorithms (Tsitsiklis and Van Roy, 1996; Van Roy, 2006; Taylor et al., 2009) have used abstraction for faster computation of the optimal policy.

Discretization is a particular type of state abstraction in continuous spaces, and in mathematics, it is generally referred to as the process of transferring a continuous model into a discrete one. Discretization of the state space is typically carried out by partitioning the state space into a finite number of non-overlapping regions (Dougherty et al., 1995). For example, suppose $\mathcal{S}$ is a unit hyper-cube of dimension n. A uniform

discretization with resolution h creates a grid that evenly splits each dimension into h intervals, each having a length of $\frac{1}{h}$. Therefore, a total of (h^n) cells will be created—denoted by $\xi_1 \dots \xi_{(h^n)}$.

A finite MDP $\tilde{M} = \langle \tilde{\mathcal{S}}, \mathcal{A}, \tilde{T}, \tilde{R}, \gamma \rangle$ can be constructed from the original MDP using discretization. Once this model is built, it can be solved using one of the existing methods for finite spaces. Below, two ways for constructing $\tilde{M}$ are explored.

Cell state. In this method, each cell in the discretization becomes a state in the new MDP. More precisely, $\tilde{s}_i \in \tilde{\mathcal{S}}$ is represented by ξ_i. The transition and the reward function of $\tilde{s}_i$ are computed using samples from T and R, similar to sample-based value iteration algorithm. First, C samples from ξ_i are generated according to some fixed distribution (for example, uniform distribution)—denote them by $(s_1, \dots, s_C)$. The mean of the reward function of these points is used as the reward function of ξ_i, that is $R(\tilde{s}_i) = \frac{1}{C} \sum_i R(s_i)$. For computing the transition function of $(\tilde{s}_i, a)$, one sample is generated from the transition function of each s_i, that is $s'_i \sim T(s_i, a)$. Then the set $\xi(s'_1), \dots, \xi(s'_C)$ is constructed, where $\xi(s'_i)$ is the cell containing s'_i. Finally, the maximum-likelihood estimate is used to compute $\tilde{T}(\tilde{s'}|\tilde{s}, a)$ based on this set. Likewise, if we only need a generative model of the MDP, each time we are asked for a sample from $\tilde{T}(\tilde{s}, a)$, we uniformly sample one point s from the corresponding cell, generate a sample from its transition function $s' \sim T(s, a)$, and return $\xi(s')$ as the answer. Figur B.1 shows the details of this process. The picture on the left shows a 2-dimensional state space along with a discretization with resolution 3. The picture in the middle is a zoom-in on the cell at the bottom-right corner. It shows 5 randomly selected states from this cell along with their next states when action a_1 is performed. The picture on the right shows the discretized MDP along with the transition function $T(\xi_9, a_1)$.

Corner state. In this method, the corners of the grid are used as the states in $\tilde{M}$. For constructing the transition function, we need to first study *Kuhn-triangulation* (Moore, 1992) and an interpolation mechanism called "barycentric interpolators" (Munos and Moore, 1998).

Kuhn-triangulation is a technique for partitioning a k-dimensional hyper-cube into

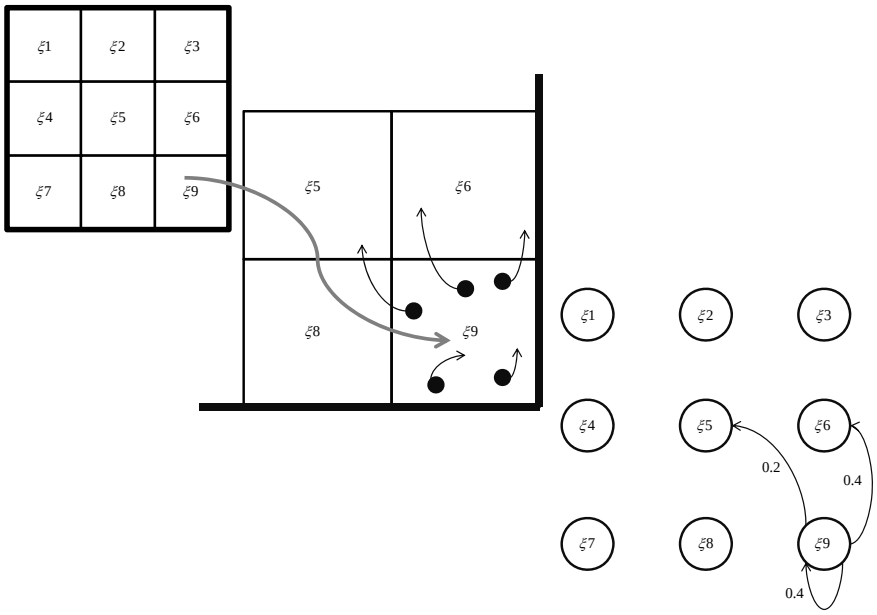


Figure B.1: The process of discretizing a 2D state space using the cell-state method.

non-overlapping k-dimensional simplexes. Given this partitioning, we can associate each point in a cell to one of the simplexes. This process is explained below.

For ease of exposition, let us consider an ordering for the corners of the i-th cell. The corners $(\xi_{i,j}|j = 1 \ldots 2^k - 1)$ can be ordered using the binary encoding of j. This process is best explained using a picture. Figure B.2 shows the ordering for 2D and 3D cells. As you can see, the first two points $\xi_{i,0}, \xi_{i,1}$ always cover the corners of the line spanning the first axis, the four points $\xi_{i,0}, \ldots, \xi_{i,3}$ cover the corners of the square spanning the first two axes, and so on.

Let $j_0, j_1, \ldots, j_{d-1}$ be a permutation of components of s such that we have $s(j_0) \geq s(j_1) \geq \ldots \geq s(j_{d-1})$, that is a permutation achieved by sorting the components of s.

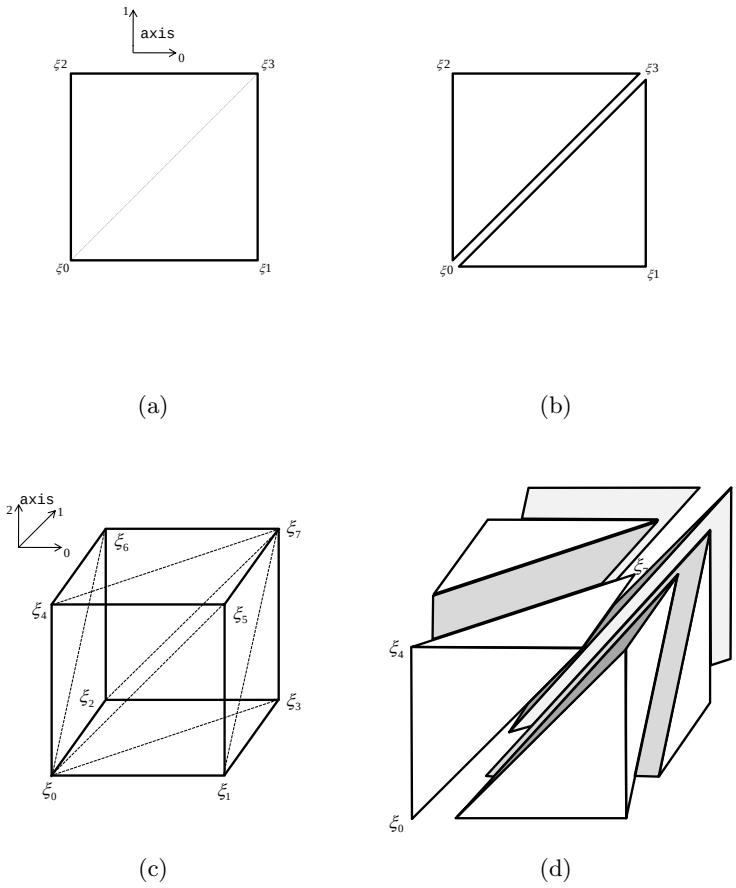


Figure B.2: The ordering of the cell corners are depicted in 2D and 3D worlds in (a) and (c) respectively. (b) and (d) show the simplexes formed by the corner-state method in the corresponding worlds.

The $k+1$ indices of the corresponding simplex are then:

$$
\begin{aligned}
i_0 &= 0 \\
i_1 &= i_0 + 2^{j_0} \\
&\vdots \\
i_l &= i_{l-1} + 2^{j_{l-1}} \\
&\vdots \\
i_k &= i_{k-1} + 2^{j_{k-1}} = 2^k - 1.
\end{aligned}
$$

Figure B.2 (b) and (d) show how the simplexes are formed in 2D and 3D hypercubes.

Now, let us look at how the barycentric interpolation works given a Kuhn-triangulation. For any point $s \in \mathcal{S}$, let $\xi_{i_0}, \ldots, \xi_{i_k}$ be the set of the corners of the simplex that s belongs to. The barycentric coordinates $\lambda_0, \ldots, \lambda_k$ of s are coefficients of a linear combination of $\xi_{i_0}, \ldots, \xi_{i_k}$ used to represent s. In other words, the barycentric coordinates for a state s satisfy the following two conditions:

1. Their sum equals to 1: $\sum_{j=0}^{k} \lambda_j = 1$
2. They are the coefficients of the linear function of corners that produce that state: $s = \sum_{j=0}^{k} \lambda_j \xi_{i_j}$

It is straightforward to show that given the Kuhn-triangulation, barycentric coordinates are (uniquely) defined as:

$$\begin{aligned}
\lambda_0 &= 1 - s_{j_0} \\
\lambda_1 &= s_{j_0} - s_{j_1} \\
&\vdots \\
\lambda_l &= s_{j_{l-1}} - s_{j_l} \\
&\vdots \\
\lambda_k &= sj_{k-1} - 0.
\end{aligned}$$

To recap, these two steps allow us to represent any state using a linear combination of the $k+1$ corners of a simplex containing that point. Given this setup, it is easy to construct $\tilde{T}$: If a generative model is intended, a sample from the original transition function is generated (that is, $s' \sim T(\xi_i, a)$). Then, the corners of the simplex containing s' and their corresponding barycentric coefficients are computed. And finally, one of the corners is selected randomly according to the distribution defined by the barycentric coefficients. Algorithm 17 shows the detail of this process. The construction of the full transition function follows the same procedure.

Algorithm 17 Generating samples from $\tilde{T}(\xi_i, a)$ in the corner-state model.

1: Generate a sample from the underlying continuous MDP: $s' \sim T(\xi_i, a)$.
2: Compute the indices of the corners of the simplex containing s': $(\xi_{i_0}, \ldots, \xi_{i_k})$.
3: Compute the corresponding barycentric coefficients: $(\lambda_0, \ldots, \lambda_k)$.
4: Randomly return ξ_{i_j} according to the probability λ_j.

B.2.2 Fitted Value Iteration (FVI)

Fitted value iteration (Gordon, 1995) is a generalization of the corner-state method we studied earlier. This algorithm generates a sequence of value functions $V_1, V_2, \ldots$, each V_t computed from the previous one V_{t-1} by applying an approximate Bellman operator. Assuming that V_t is presented using the function approximator $\mathcal{F}_t$, the algorithm goes through two procedures to obtain $\mathcal{F}_{t+1}$. First, an approximate Bellman operator is applied to a set of backup points s_i by using a Monte-Carlo estimate.

$$\hat{V}_{t+1}(s_i) = R(s_i) + \max_a \frac{\gamma}{C} \sum_{j=1}^{C} \mathcal{F}_t(y_j^{(s_i,a)}), \quad i = 1, \ldots, N, \tag{B.4}$$

where each $y_j^{(s_i,a)}$ is a sample from $T(s_i, a)$. This estimate is reminiscent of the technique used in sample-based value iteration. The second step in each iteration is to learn $\mathcal{F}_{t+1}$ using the training set $\{(s_i, \hat{V}_{t+1}(s_i)) | i = 1..N\}$.

There are a couple of important parameters in FVI that need to be examined. The Monte-Carlo parameter C specifies how accurately the Bellman backup for a particular state needs to be approximated. In the degenerate case of deterministic transition function, setting $C = 1$ is all we need. As a general rule of thumb, the more stochastic the environment is, the higher the value of C should be. The number of samples N and the way the samples are selected from the state space has a direct impact on the expressiveness of the approximation. Intuitively, N should be small enough so that we can perform the backup operator on all of the data points. But, at the same time, it should be large enough so that the entire value function can be reasonably represented by only N points. The choice of the distribution used to select these samples is more complicated (Munos and Szepesvári, 2008). Again, the distribution should be such that the value function of the entire MDP can be reasonably expressed by only the values

at the points from it. An empirical evaluation of different heuristics will be provided later on.

Perhaps the most important part of the algorithm is the choice of $\mathcal{F}$. We saw earlier that the Bellman operator is a contraction operator and that is the reason why estimations of value iteration approaches the optimal value function at a geometric rate. Unfortunately, the combination of the approximate Bellman operator and the function approximation might not always be a contraction operator, and therefore, the whole process might never converge. In fact, many people have showed that this situation arises in the context of a lot of popular choices of function approximators in many MDPs (Boyan and Moore, 1995; Tsitsiklis and Van Roy, 1996). But, the situation is not that hopeless either. Gordon (1995) showed that at least for one class of function approximators, called "averagers", the combination with approximate Bellman operator is indeed a contraction, and therefore, convergence to the optimal value function is guaranteed.

A function approximator is an averager if $\mathcal{F}$ is a weighted average of zero or more of y_i's (a constant term is also allowed), where (x_i, y_i) is in the training data. The weights of this average can be dependent on x_i's but not on any of the y_i's.

More than a decade later, Munos and Szepesvári (2008) provided a detailed analysis on the finite-time convergence of FVI depending on the choice of its parameters.

B.3 Forward Planning

All the algorithms presented in the previous sections compute the value function for the entire MDP. That is, after the planning is done, the value function $V(s)$ is readily available for all states. In this section, another set of algorithms are presented that compute the value function of only one state (that is, the current state). The intuition behind this approach is that sometimes a huge portion of the state space is not relevant to the planning problem at hand. For example, consider a player in the game of GO. We saw that the state space of this game is bigger than the total number of atoms in the whole universe. However, at any given time, the player does not need to know the

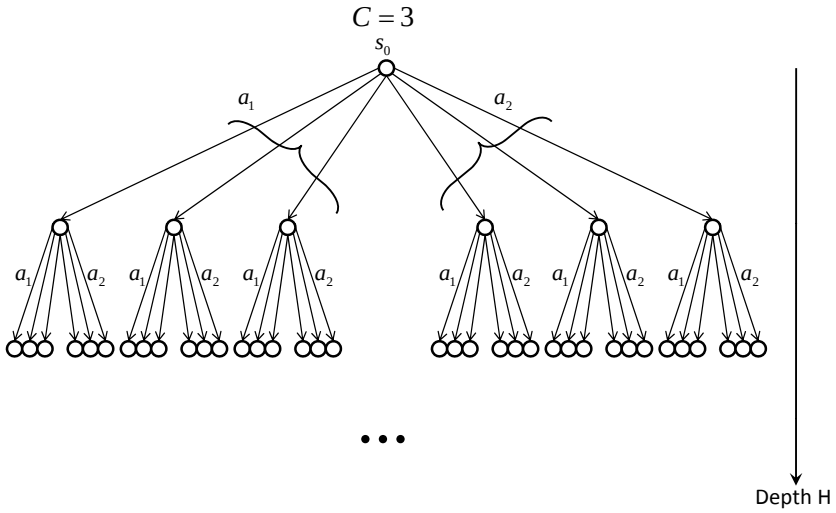


Figure B.3: A schematic view of a sparse-sampling tree. Each node has $C|\mathcal{A}|$ children.

value function of all the states to be able to play the game. All it needs to know is the Q-function of the current board configuration, from which it can compute a policy to play the game. Assuming that the planning for one state is faster than solving the entire MDP, the player can repetitively replan for the current state as the game goes on. These planning algorithms usually create a forward search tree to compute the value of the current state—hence the name.

Indeed, it has been shown that forward planning can be done (arbitrarily) faster than full planning. A forward-planning algorithm called "sparse sampling" was introduced by Kearns et al. (1999), whose running time was independent of the size of the state space. This algorithm creates a new MDP M' whose state space is the set of reachable states from the current state and an ϵ-horizon time.

This MDP is generated as a tree: The root of the tree is the current state. Each internal node of this tree is expanded by trying each action C times, and the expansion stops at the depth of:

$$H = \log_\gamma \left(\frac{\epsilon(1-\gamma)}{R_{\max}} \right). \tag{B.5}$$

Figure B.3 shows a schematic view of this tree[1]. It can be shown that the estimate

[1]This figure is taken from (Kearns et al., 1999).

of the value function at the root is ϵ-close to the true value function, provided that the value of each node at depth h is computed as:

$$\hat{V}_h(s) = R(s) + \max_a \frac{\gamma}{C} \sum_{s' \sim T(s,a)} \hat{V}_{h+1}(s'), \tag{B.6}$$

and $V_H(s) = 0$ by default. So, the value at the root is computed recursively using the value of all its descendent nodes. Sparse sampling generates a full tree of depth H with a fanning parameter $C|\mathcal{A}|$. Therefore, while its running time is independent of the size of the state space, it is exponential in $\frac{1}{\epsilon}$ and $\frac{1}{1-\gamma}$, which makes it computationally not feasible in practice as they both are usually very large numbers. This type of tree construction is sometimes called "stage-wise construction" because the values are computed recursively in stages from bottom to top. Therefore, the value of an internal node can be computed only when the entire sub-tree of that node is in place.

B.3.1 UCT

Forward planning can be effective because instead of wasting a lot of resources for computing the value at all the states, it only computes the value at the current state. However, the full tree that sparse sampling generates makes the algorithm very inefficient. The upper confidence bounds applied to trees (UCT) was introduced by Kocsis and Szepesvári (2006) as a Monte-Carlo search planner to address the inefficiency of sparse sampling.

The intuition behind UCT is to focus the computation resources on parts of the search-tree that are more promising (that is, states that are more probable to be part of the optimal policy). To achieve this goal, UCT uses another approach to building the tree. UCT constructs the tree using *rollouts* from the current state, instead of using a stage-wise approach. In Monte-Carlo rollout construction, trajectories of length H are generated from the current state one at a time. UCT uses the available information at each node (that is, the current estimated value function) to help with the process of generating the next trajectories.

In particular, UCT uses a technique from k-armed bandit called UCB (Auer et al.,

2002) to select an action at each node when generating a trajectory. This algorithm selects arms based on their estimated payoff and a specially constructed bonus function. UCT applies the same algorithm at each node of the tree using the estimated Q-function of that node. Algorithm 18 shows the detail of this process.

Algorithm 18 UCT: A Monte-Carlo tree-search planner.

Function UCT(s, C_p, H, g)**:**
while we have time **do**
  $V = \text{search}(s, 0, C_p, H, g)$;
end while
return V;
End Function

Function search(s, d, C_p, H, h)**:**
if $d = H$ **then**
  $Q_d(s, a) = g(s)$ for all $a \in \mathcal{A}$.
  Return 0.
else
  $a^* = \text{argmax}_a \left(Q_d(s, a) + C_p \sqrt{\frac{\ln(n_{sd})}{n_{a^* sd}}} \right)$.
  $s' \sim T(s, a^*)$.
  $V = R(s) + \gamma \text{UCT}(s', d + 1)$. //recursively generate the trajectory.
  $\text{Inc}(n_{sd})$.
  $\text{Inc}(n_{a^* sd})$.
  $Q_d(s, a^*) = \frac{V + (n_{a^* sd} - 1) Q_d(s, a^*)}{n_{a^* sd}}$. //maintain an average for Q_d.
  Return V.
end if
End Function

The heuristic function g that estimates the value of states at the leaves of the tree is usually a vanilla Monte-Carlo method that runs random trajectories from that leaf (perhaps, running until the end of the episode if possible). Please refer to the work of Gelly and Silver (2007) for more details on the heuristic function. The constants H and C_p indicate the depth of the search tree and the exploration-exploitation tradeoff constant, respectively.

UCT has been extremely successful in practice. It has been used for solving some computer games with huge number of states, such as Go (Gelly and Wang, 2006) and poker (Van den Broeck et al., 2009). One of the factors for the success of UCT is that it is possible to run deep trajectories in practice. In contrary, sparse sampling can only be

run with a very short values of H due to the exponential blowup. Due to the nature of sequential decision making, many times the agent has to chain a long series of actions to each other to form the optimal policy. In these cases, short sequences of actions will not be sufficient to distinguish the optimal policy from the others.

While UCT is designed for finite MDPs, we can use the same discretization techniques used in SectionB.2.1 to run UCT in continuous spaces.

Appendix C

Technical Details of Environments

This appendix explains the technical details of the environments used for evaluating different algorithms.

C.1 Mountaincar

This domain is probably the most widely-used environment in the field of reinforcement learning. It describes an underpowered car trying to move up a hill (Sutton and Barto, 1998). But, since it does not have enough power to do so, it needs to travel back and forth the hill several times to build up the necessary speed. Figure C.1 shows an illustration of this domain.

The state space of this domain consists of two variables: the horizontal position of the car that has a range in $[-1.2, 0.6]$ and its velocity in the range of $[-0.07, 0.07]$. The car is equipped with three action: *left*, *neutral*, and *right*. These actions accelerate the car in the intended direction. The reward function of this environment is computed as:

$$R(x, v) = \begin{cases} -1 & \text{if } x < 0.5 \\ 0 & \text{if } x \geq 0.5 \end{cases}$$

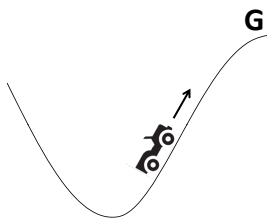


Figure C.1: MOUNTAINCAR domain.

The next state is computed using the following equations:

$$v_{t+1} = 0.0001 f(a_t) - 0.0025 \cos(3x_t)$$
$$s_{t+1} = s_t + v_{t+1},$$

where $f(a_t)$ is defined as (-1) for left, (0) for neutral, and (1) for right. The state variables were brought back within the boundaries of the state space if they went outside.

Each time the car reached the goal region, the episode ended. Also, the episode ended if the car did not manage to get to the top after 300 steps. The starting state was selected to be hard. The car was always started at the bottom of the hill with a tiny random velocity selected from $\mathcal{N}(0, 0.005)$.

C.1.1 n-Mountaincar

This environment is an extension of MOUNTAINCAR that contains multiple cars (Nouri and Littman, 2010). In an instance of n-MOUNTAINCAR, n cars are placed in n parallel worlds—cars do not clash with each other—and are controlled using the same set of three actions. The effect of each action is different on each car. For example, while action *right* accelerates the first car to the right, it might accelerate the second car to the left. The effect of each action on each car is picked at random at the beginning of each experiment. The goal of this domain is to drive the first car to the top of the hill. Therefore, the value functions of all n-MOUNTAINCAR domains are the same, independent of n.

C.2 Puddleworld

This environment is a simulated navigation problem. An agent is placed inside a bounded region $[0, 1]^2$. Its task is to navigate to a goal region while avoiding two existing puddles in the box (Sutton, 1996). Figure C.2 explains how the environment works. The state space of this environment is the position of the agent:

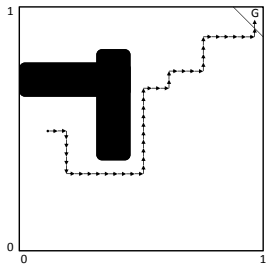


Figure C.2: PUDDLEWORLD domain.

$$\mathcal{S} = \{(x, y) | x \in [0, 1], y \in [0, 1]\}.$$

The agent is equipped with four actions: *up*, *down*, *left*, *right*. Intuitively, these actions are designed to move the agent in the intended direction by a fixed amount plus some noise. More precisely, the transition function is:

$$s_{t+1}|s_t, \textit{up} = s_t + (0, 0.05) + \omega_t \tag{C.1}$$

$$s_{t+1}|s_t, \textit{down} = s_t + (0, -0.05) + \omega_t \tag{C.2}$$

$$s_{t+1}|s_t, \textit{right} = s_t + (0.05, 0) + \omega_t \tag{C.3}$$

$$s_{t+1}|s_t, \textit{left} = s_t + (-0.05, 0) + \omega_t, \tag{C.4}$$

$$\tag{C.5}$$

where ω_t is selected from a normal distribution as follows: $\omega_t \sim \mathcal{N}(0, 0.01)$. The goal region is defined as the set of points that are 0.1-close to $(1, 1)$ in 1-norm:

$$G(s) \stackrel{\text{def}}{=} \| x - (1, 1) \|_1 \leq 0.1.$$

Two puddles are present in the environment in the form of two ellipsoids, whose centers are located at:

$$\text{pud}_1 = \langle (0.1, 0.75), (0.45, 0.75) \rangle$$

$$\text{pud}_2 = \langle (0.45, 0.40), (0.45, 0.80) \rangle.$$

The reward function is computed based on the position of the agent. If it is in the goal region, a reward of 0 is received and the episode is terminated. If its distance to the center of any of the puddles is less than 0.1, the reward is computed as $R(s) = -400(0.1 - d)$, where d is the distance to the center of the ellipsoid. If the agent is outside the puddles, it receives a -1 penalty. Finally, the episodes are started by selecting a random state according to a uniform distribution outside of the goal region and the puddles.

C.2.1 n-Puddleworld

This environment is an extension of PUDDLEWORLD to n-dimensional space (Nouri and Littman, 2010). The agent is placed in the n-dimensional unit hyper-cube $[0, 1]^n$, and its goal—similarly to PUDDLEWORLD—is to reach the goal region while avoiding the puddles. The goal and the puddles are projected into the dimensions higher than 2, such that the value function of all n-PUDDLEWORLDS are the same. That is, the goal region is defined as:

$$G(s) = \| (s(1), s(2)) - (1, 1) \|_1 \leq 0.1,$$

and the distance to the puddles is computed the same way. Actions are also manipulated to match the new space; there are $2n$ actions in n-PUDDLEWORLD. Actions a_{2i} and a_{2i+1} move the agent along the i-th dimension by the same constant amount as in PUDDLEWORLD.

C.3 Warpedworld

This environment is basically a more complicated version of PUDDLEWORLD. Everything in these two environments is exactly the same, except for the transition function. While in PUDDLEWORLD, the agent moves by a constant amount after executing each action, the effect of actions in WARPEDWORLD is based on a linear combination of the state variables:

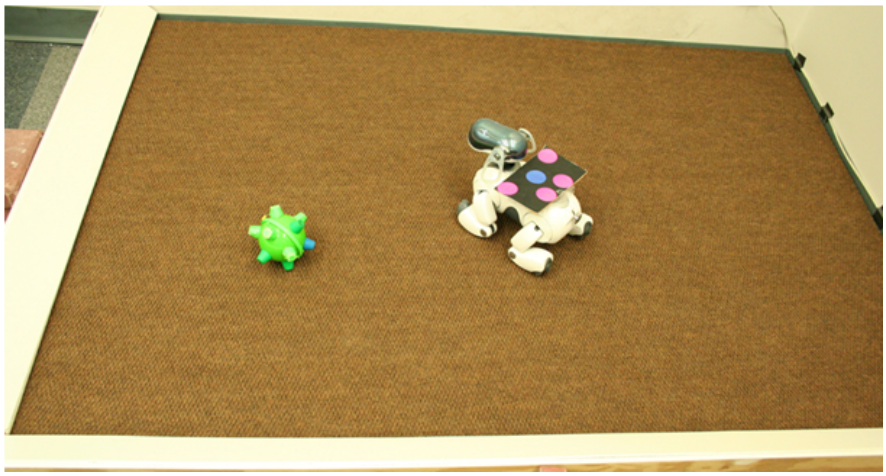


Figure C.3: A top view of the BUMBLEBALL field.

$$s_{t+1}|s_t, a = \sum_{i=1}^{n} C_a^i s_t(i) + \omega_t,$$

where ω_t is the same noise function used in PUDDLEWORLD and C_a is a vector of coefficients defined individually for each action. In our experiments, these constants were selected to be the same for all actions and were set to $(0.01, 0.02)$ and $(0.01, 0.02, 0.03)$ for 2-WARPEDWORLD and 3-WARPEDWORLD, respectively. Note that because of these new dynamics, the value functions of n-WARPEDWORLD would not be the same for different values of n.

C.4 Bumbleball

This section explains the technical details of how BUMBLEBALL was implemented.

The BUMBLEBALL domain was inspired by a widely-used simulation environment in the RL community called PUDDLEWORLD (Sutton, 1996). In this robotic navigation task, an Aibo robot was placed inside a box with a size of approximately 4×6 feet, and had to navigate to a goal region, while avoiding contact with a randomly moving ball (Figure C.3).

Aibo is a programmable four-legged dog robot that was first introduced in 1999 by Sony as an intelligent pet toy. However, it received a lot of attention from robotics

Figure C.4: Pictures of three generations of Aibo: ERS-111 was the first model, followed by ERS-210 and finally ERS-7.

Figure C.5: Picture of a bumble ball.

researchers, especially after it was introduced as a league in international RoboCup competition. The production of Aibo was discontinued in 2006 after three generations of different models. Figure C.4 shows pictures of different Aibo generations.

The last generation of the Aibo is equipped with a lot of sensors, including pressure sensors, infra-red sensors, acceleration and vibration sensors, and microphone. It is equipped with 64MB RAM, 64-bit RISC processor, and wireless network card. The operating system that runs on Aibo is called MIPS, but a cross-compiler for Linux is available. Furthermore, an impressive library called Tekkotsu is available for developing applications for Aibo in C++.

The moving ball used in the experiment was called "bumble ball". It was a motorized toy ball from Ertl toys company. A picture of it can be found in Figure C.5.

The Bumbleball environment was implemented using a distributed system architecture. A camera mounted on the top of the field captured the current state of the environment. This information was processed in a PC computer. This machine also ran

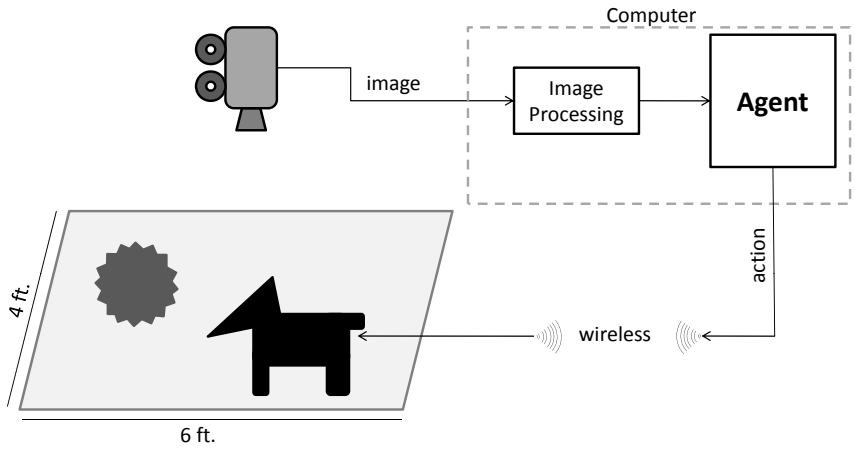


Figure C.6: Distributed architecture of the BUMBLEBALL environment.

the code for the RL experiment including the agent. The actions executed by the agent was transferred over the wireless network to another program running on the Aibo. This program executed the encoded action and the loop repeated. This architecture is shown in Figure C.6.

The state space of the environment consisted of 5 variables: Two for the position of the robot (its center of mass), two for the position of the ball, and one for the orientation of the robot. The current state was captured by placing a special marker on the back of the robot and using a video-processing program called "dogtracker" developed by Chris Mansley, who is a current graduate student in RL^3 lab at Rutgers.

The Aibo had 6 available macro actions that allowed the robot to move around. These actions were *MoveForward*, *MoveBackward*, *TurnRight*, *TurnLeft*, *StrafeLeft*, and *StrafeRight*. These low-level implementation of these actions were hard-coded into the robot using gait patterns in the Tekkotsu library. Specifically, applying each of these actions resulted in the robot executing the corresponding gait pattern for a fixed amount of time. This execution time was based on whether the current action executed was the same as the previous one or not. If it was the same, the gait pattern was executed for 1.0 second, and if it was not, the gait pattern was executed for 1.3 second. The reason for this change was to make sure the effect of actions were as close to Markovian as possible (going forward when the robot is already moving forward is

easier than moving forward when the robot is already moving backward).

The Bumbleball domain was implemented as an *Environment-module* in the RL-GLUE framework and will be available for download from the RLGLUE website.

Appendix D

Mathematical Facts

D.1 Learnability of Continuous MDPs

We discussed several sample-complexity learning analyses. The following theorem states that learning—using any of these analyses—becomes impossible in continuous state-space MDPs if no assumption is made about the smoothness of the transition function.

Theorem 28. *Given a continuous state-space MDP $M = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$, if no smoothness assumption is made on T, that is, if $T(s_1, a)$ and $T(s_2, a)$ can be arbitrarily different for any pairs of states $s_1 \neq s_2$, no algorithm is able to guarantee learning of M according to any of the following MDP learning frameworks: asymptotic convergence, regret minimization or PAC-MDP analysis.*

Proof (sketch). This problem is unlearnable in all the frameworks because of the same reason: there are an infinite number of states in a continuous state space and no matter how much data the agent collects, there are always some unvisited states. Since no smoothness assumption is available, the agent has no information about those states, so it can behave arbitrarily badly in those states. Here is an example of such domain.

Consider an environment with a one-dimensional state space in the interval $[0, 1]$ and two actions. Imagine that the agent starts at $s = 1$. Both actions result in a transition to $s/2$. However, the payoff of one action is 0 and the other action is 1, as a function of the state s. This mapping can be arbitrary, so, on every step, the agent has a 50-50 chance of getting high reward and a 50-50 chance of getting low reward. No learning can take place because no state is ever visited twice and no generalization is possible. □

D.2 Kernel Functions

Kernel functions are widely used in statistical machine learning as a particular kind of similarity function. For example, they are used in *kernel density estimation* (Hastie et al., 2003), *time series* (Warren Liao, 2005), *kernel regression*, and many other research disciplines.

A kernel function is a non-negative integrable scalar function defined on u, satisfying the following two conditions:

- $\int_{-\infty}^{+\infty} k(u)du = 1$.
- $k(-u) = k(u)$.

In the cases where kernel function is used to measure the similarity between two points x_1 and x_2 in some space $\mathbb{R}^n$, u becomes the distance between the two points. Let $d(x_1, x_2)$ be a distance metric in $\mathbb{R}^n$. Then, the kernel function becomes $k(x_1, x_2) = k(d(x_1, x_2))$. Many forms of kernel functions have been used in various literatures. Here, we summarize some of the more popular ones:

- **Uniform:** $k(u) = \frac{1}{2\sigma}\mathbb{I}(|u| \leq \sigma)$.
- **Triangle:** $k(u) = (\sigma - |u|)\mathbb{I}(|u| \leq \sigma)$.
- **Cosine:** $k(u) = \frac{\pi}{4\sigma}\cos\left(\frac{\pi}{2\sigma}u\right)\mathbb{I}(|u| \leq \sigma)$.
- **Gaussian:** $k(u) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{u^2}{\sigma^2}\right)$.

All these kernel functions are peaked at $u = 0$ and decay to 0 when $|u| \to \infty$. The difference between them is how do so, while maintaining the two above assumptions. The parameter σ is usually referred to as "kernel width" or "smoothing parameter". It controls the rate at which the kernel function decays to 0. Typically, larger values of σ means that the kernel function is wider. That is, the rate at which it goes to 0 is smaller. The shape of these kernel functions are depicted in Figure D.1 for a fixed value of $\sigma = 1$.

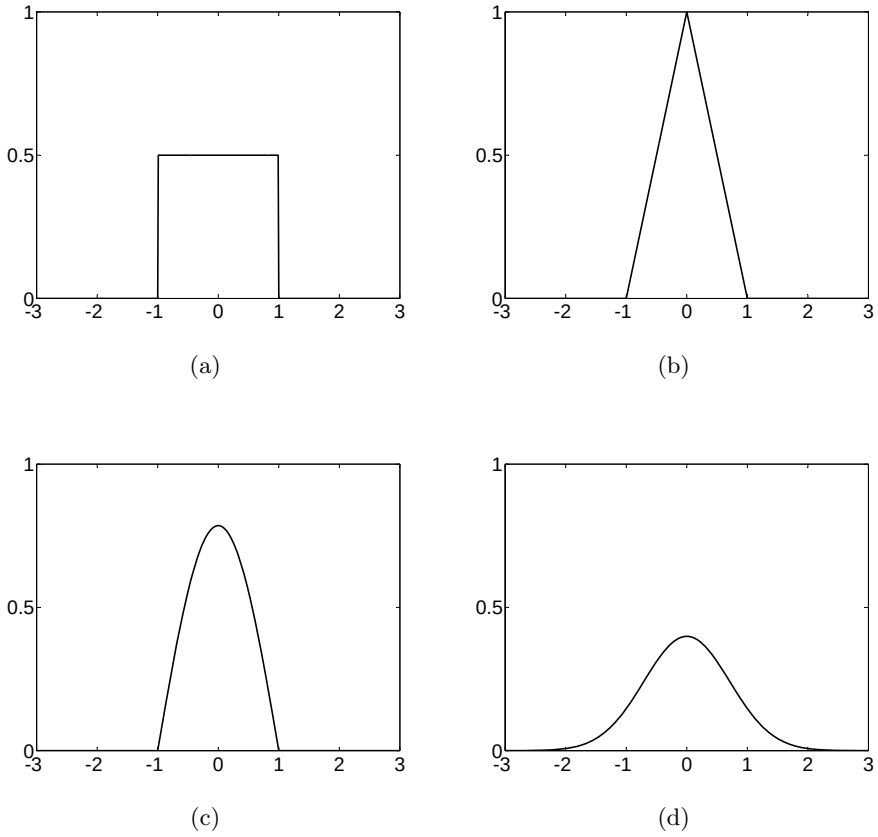


Figure D.1: Several examples of kernel functions. The value of σ was set to 1 for all the functions.

Figure D.2 illustrates the effect of changing the kernel width in Gaussian kernel. Four values are shown in the figure. Smaller values of σ create functions that are closer to a dirac function, whereas larger values make the function more spread-out. Usually, depending on the type of application, one needs to search for the right value of σ that optimizes performance (Sheather and Jones, 1991).

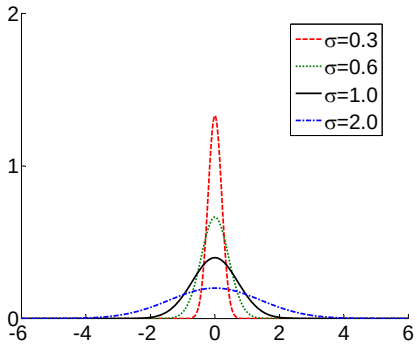


Figure D.2: Different values of σ in the Gaussian kernel.

Bibliography

Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

Peter Auer and Ronald Ortner. Logarithmic online regret bounds for undiscounted reinforcement learning. In *Advances in Neural Information Processing Systems 19*, pages 49–56, 2007.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning Journal*, 47(2-3):235–256, 2002. ISSN 0885-6125.

Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. *Machine Learning Research*, 11:1563–1600, April 2010.

Andrew G. Barto, S. J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

Jonathan Baxter and Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15(1):319–350, 2001. ISSN 1076-9757.

Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in Neural Information Processing Systems 14*, pages 585–591. MIT Press, 2001.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

Richard Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, NJ, 1961.

Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 23(3):610–616, 1982.

Craig Boutilier, Thomas Dean, and Steve Hanks. Planning under uncertainty: structural assumptions and computational leverage. *New directions in AI planning*, pages 157–171, 1996.

Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995.

Ronen I. Brafman and Moshe Tennenholtz. R-MAX—A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002.

Emma Brunskill, Bethany R. Leffler, Lihong Li, Michael L. Littman, and Nicholas Roy. Provably efficient learning with typed parametric models. *Journal of Machine Learning Research*, 10:1955–1988, December 2009.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization in χ-armed bandits. In *In Advances in Neural Information Processing Systems 22*, 2008.

Chee-Seng Chow and J. N. Tsitsiklis. The complexity of dynamic programming. *Journal of Complexity*, 5(4):466–488, 1989.

Dennis Cook. Fisher lecture: Dimension reduction in regression. *Statistical Science*, 22 (1):1–26, 2007.

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, April 2008.

S. de Jong. SIMPLS: An alternative approach to partial least squares regression. *Chemometrics and Intelligent Laboratory Systems*, 18(3):251–263, March 1993.

J. Dougherty, R. Kohavi, and M Sahami. Supervised and unsupervised discretization of continuous features. In *12th International Conference on Machine Learning*, pages 194–202, 1995.

Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12:219–245, 2000.

Norman R. Draper and Harry Smith. *Applied Regression Analysis (Wiley Series in Probability and Statistics)*. Wiley-Interscience, April 1998.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Maching Learning Research*, 6:503–556, 2005.

Damien Ernst, Mevludin Glavic, Florin Capitanescu, and Louis Wehenkel. Reinforcement learning versus model predictive control: a comparison on a power system problem. *IEEE Transactions on Systems, Man and Cybernetics Part B*, 39(2):517–529, 2009.

Claude-Nicolas Fiechter. Efficient reinforcement learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, pages 88–97. Association of Computing Machinery, 1994.

Claude-Nicolas Fiechter. Expected mistake bound model for on-line reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 116–124, 1997.

Yoav Freund, H. Sebastian Seung, Eli Shamir, and Naftali Tishby. Selective sampling using the query by committee algorithm. In *Machine Learning*, pages 133–168, 1995.

Kenji Fukumizu, Francis R. Bach, and Michael I. Jordan. Kernel dimension reduction in regression. *Annals of Statistics*, 37:1871, 2009.

P. Geladi. Partial least-squares regression: a tutorial. *Analytica Chimica Acta*, 185(1): 1–17, 1986. ISSN 00032670.

Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the International Conference of Machine Learning*, pages 273–280, 2007.

Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *Neural Information Processing Systems Conference, On-line trading of Exploration and Exploitation Workshop*, December 2006.

Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57 (2-3):323–389, 1992. ISSN 0004-3702.

Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, 1995.

Carlos Guestrin, Relu Patrascu, and Dale Schuurmans. Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the International Conference on Machine Learning*, pages 235–242, 2002.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, corrected edition, July 2003.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

Tommi Jaakkola, Satinder P. Singh, and Michael I. Jordan. Reinforcement learning algorithm for partially observable markov decision problems. In *Advances in Neural Information Processing Systems 7*, pages 345–352. MIT Press, 1995.

I.T. Jolliffe. *Principal Component Analysis*. Springer, New York, NY, 1986.

Nicholas K. Jong and Peter Stone. Model-based exploration in continuous state spaces. In *the 7th Symposium on Abstraction, Reformulation, and Approximation*, July 2007.

Tobias Jung and Peter Stone. Gaussian processes for sample efficient reinforcement learning with RMAX-like exploration. In *Proceedings of the European Conference on Machine Learning*, September 2010.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2): 99–134, 1998.

Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning*, pages 267–274, 2002.

Sham Kakade, Michael Kearns, and John Langford. Exploration in metric state spaces. In *Proceedings of the 20th International Conference on Machine Learning*, pages 306–312, 2003.

Sham M. Kakade. *On the Sample Complexity of Reinforcement Learning.* PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the 15th International Conference on Machine Learning*, pages 260–268, 1998.

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 (IJCAI-99)*, pages 1324–1331, 1999.

Michael J. Kearns and Daphne Koller. Efficient reinforcement learning in factored MDPs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 740–747, 1999.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

J. Zico Kolter and Andrew Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *The 26th International Conference on Machine Learning*, pages 521–528, 2009.

Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transanction on Knowledge Discovery from Data*, 3(1):1–58, 2009. ISSN 1556-4681.

P. R. Kumar and P. P. Varaiya. *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice Hall, Englewood Cliffs, NJ, 1986.

Michail Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research (JMLR)*, 4:1107–1149, 2003.

Lihong Li. *A Unifying Framework for Computational Reinforcement Learning Theory*. PhD thesis, Rutgers, the State University of New Jersey, 2009.

Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for MDPs. In *Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006.

Lihong Li, Michael L. Littman, and Thomas J. Walsh. Knows what it knows: A framework for self-aware learning. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML-08)*, pages 568–575, 2008.

Lihong Li, Michael L. Littman, and Christopher R. Mansley. Online exploration in least-squares policy iteration. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 733–739, 2009.

N. Littlestone. Learning when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning*, 2:285–318, 1988.

Michael L. Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, February 1996.

Huan Liu and Hiroshi Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

Sridhar Mahadevan. Learning representation and control in markov decision processes: New frontiers. 1(4):403–565, 2009.

Andrew W. Moore and Christopher G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 263–270, San Mateo, CA, 1993. Morgan Kaufmann.

Douglas William Moore. *Simplicial mesh generation with applications*. PhD thesis, Department of Computer Science, Cornell University, 1992.

Rémi Munos and Andrew Moore. Barycentric interpolators for continuous space & time reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1024–1030. MIT Press, 1998.

Rémi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49:291–323, 2002.

Rémi Munos and Csaba Szepesvári. Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, 9:815–857, 2008.

Ali Nouri and Michael L. Littman. Multi-resolution exploration in continuous spaces. In *Proceedings of the 22nd Neural Information Processing Systems*, pages 1209–1216, 2008.

Ali Nouri and Michael L. Littman. Dimension reduction and its application to model-based exploration in continuous spaces. *Machine Learning*, 81(1):85–98, 2010.

Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, 2008.

K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

Jing Peng and Ronald J. Williams. Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction*. Springer, 1985. ISBN 3-540-96131-3.

Martin L. Puterman. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

Carl E. Rasmussen and Christopher Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

Sam Roweis and Lawrence Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1994. ISBN 0-13-103805-2.

Burr Settles. Active learning literature survey. Technical report, Carnegie Mellon University, 2010.

S. J. Sheather and M. C. Jones. A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, 53(3):683–690, 1991.

William D. Smart. Explicit manifold representations for value-function approximation in reinforcement learning. In *Prceedings of the 8th International Symposium on Artificial Intelligence and Mathematics*, 2004.

Gilbert Strang. *Linear Algebra and its Applications*. Academic Press, Orlando, FL, 2nd edition, 1980.

Alexander L. Strehl. *Probably Approximately Correct (PAC) Exploration in Reinforcement Learning*. PhD thesis, Department of Computer Science, Rutgers, the State University of New Jersey, 2007.

Alexander L. Strehl and Michael L. Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the Twenty-second International Conference on Machine Learning (ICML-05)*, pages 857–864, 2005.

Alexander L. Strehl and Michael L. Littman. Online linear regression and its application to model-based reinforcement learning. In *Advances in Neural Information Processing Systems 20*, pages 737–744, 2008a.

Alexander L. Strehl and Michael L. Littman. An analysis of model-based interval estimation for Markov decision processes. *Journal of Computer and System Sciences*, 74:1309–1331, 2008b. special issue on Learning Theory.

Alexander L. Strehl, Lihong Li, and Michael L. Littman. Incremental model-based learners with formal learning-time guarantees. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006)*, 2006. URL `http://www.cs.rutgers.edu/~strehl/papers/UAI06IncrementalModelBasedRL.pdf`.

Alexander L. Strehl, Lihong Li, and Michael L. Littman. Reinforcement learning in finite MDPs: PAC Analysis. *Journal of Machine Learning Research*, 10:2413–2444, 2009.

Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

Richard S. Sutton. First results with Dyna, an integrated architecture for learning, planning and reacting. In *Neural networks for control*, pages 179–189. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-13261-3.

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044, Cambridge, MA, 1996.

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

Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063, 2000.

Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.

Jonathan Taylor, Prakash Panangaden, and Doina Precup. Bounding performance loss in approximate MDP homomorphisms. In *Advances in Neural Information Processing Systems*, pages 1649–1656, 2009.

Joshua B. Tenenbaum, Vin Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.

Sebastian B. Thrun. The role of exploration in learning control. In *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 527–559. 1992.

John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1/2/3):59–94, 1996.

L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *Advances in Machine Learning*, volume 5828 of *Lecture Notes in Computer Science*, pages 367–381. 2009.

Benjamin Van Roy. Performance loss bounds for approximate value iteration with state aggregation. *Mathematics of Operations Research*, 31(2):234–244, 2006.

Jarkko Venna and Samuel Kaski. Local multidimensional scaling. *Neural Networks*, 19 (6):889–899, 2006.

Sethu Vijayakumar and Stefan Schaal. Approximate nearest neighbor regression in very high dimensions. In *Nearest Neighbor Methods in Learning and Vision*. MIT press, 2006.

Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 0387001522.

Thomas J. Walsh, Ali Nouri, Lihong Li, and Michael L. Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18(1):83–105, 2009.

T. Warren Liao. Clustering of time series data–a survey. *Pattern Recognition*, 38(11): 1857–1874, 2005.

Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

Kilian Q. Weinberger and Gerry Tesauro. Metric learning for kernel regression. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, 2007.

Ronald J. Williams and Leemon C. Baird, III. Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-14, Northeastern University, College of Computer Science, Boston, MA, November 1993.

Vita

Ali Nouri

EDUCATION

November 2010 Ph.D. in Computer Science, Rutgers University, New Brunswick, NJ, USA

October 2003 B.Sc. in Computer Science, Sharif University of Technology, Tehran, Iran

EXPERIENCE

09/2005—06/2010	Graduate Assistant, Department of Computer Science, Rutgers University, New Brunswick, NJ, USA
02/2006—06/2010	Part-time Research Intern, Center for Advanced Infrastructure and Transportation, NJ, USA
09/2004—06/2005	Teaching/Research Assistant, Department of Computing Science, University of Alberta, Edmonton, AB, Canada

PUBLICATION

- A. Nouri, M. L. Littman: Dimension Reduction and Its Application to Model-based Exploration in Continuous Spaces, *Machine Learning Journal, Pages 85–98, 2010.*
- T. J. Walsh, A. Nouri, L. Li, M. L. Littman: Planning and Learning in Environments with Delayed Feedback , *Journal of Autonomous Agents and Multi-Agent Systems , Pages 83–105, 2009.*
- J. Asmuth, L. Li, M. L. Littman, A. Nouri, D. Wingate: A Bayesian Sampling Approach to Exploration in Reinforcement Learning, *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI), 2009.*
- A. Nouri, M. L. Littman: Multi-resolution Exploration in Continuous Spaces, *Proceedings of the 21st Conference on Advances in Neural Information Processing Systems (NIPS), 2009.*
- A. Nouri, M. L. Littman, L. Li, R. Parr, C. Painter-Wakefield, G. Taylor: A Novel Benchmark Methodology and Data Repository for Real-life Reinforcement Learning, *Multidisciplinary Symposium on Reinforcement Learning*, Montreal, Canada, 2009.
- A. Nouri, M. L. Littman, L. Li: PAC-MDP Reinforcement Learning with Bayesian Priors in Deterministic MDPs, *NIPS 08 workshop on Model Uncertainty and Risk in Reinforcement Learning, Whistler, Canada 2008.*
- T. J. Walsh, A. Nouri, L. Li, M. L. Littman: Planning and Learning in Environments with Delayed Feedback , *Proceedings of the 18th European Conference on Machine Learning (ECML) , 2007.*

- A. Nouri, R. Zamani-Nasab, J. Habibi, A. Geramifard: Task Allocation in Complex Multi-agent Systems with Parallel Scheduling, *Proceedings of the 2nd Workshop on Information Technology & Its Disciplines (WITID), Kish Island, Iran, 2004.*
- A. Nouri, J. Habibi, Using Layered Genetic Algorithm in Multi-Agent Learning, *CSI Journal on Computer Science & Engineering, Vol. 1, No2, October 2003.*
- J. Habibi, M. Ahmadi, A. Nouri, M. Sayadian, Implementing Heterogeneous Agents in Dynamic Environment, A Case Study in RoboCup Rescue, *The German Conference on Multiagent System Technologies (MATES), 2003.*
- J. Habibi, A. Nouri, M. Ahmadi: Utilizing Different Multiagent Methods in RoboCupRescue Simulation, *Proceedings of the RoboCup-2002 Symposium, Fukuoka, Japan, 2002.*