NOTE TO USERS

ON CHARACTERISTICS OF MARKOV DECISION PROCESSES AND REINFORCEMENT LEARNING IN LARGE DOMAINS

Bohdana Ratitch

School of Computer Science
McGill University, Montréal

February 2005

A thesis submitted to McGill University
in partial fulfilment of the requirements of the degree of
Doctor of Philosophy

Your file Votre référence
ISBN: 0-494-12934-4
Our file Notre référence
ISBN: 0-494-12934-4

Acknowledgments

My deepest gratitude goes to my thesis advisors, Doina Precup and Denis Thérien.

Denis Thérien was my advisor from the time I started my graduate studies at McGill as a Master student. He helped me to learn good research practices and techniques. In addition to his own guidance, he provided me with valuable opportunities to learn from and work with his colleagues on academic and industrial research projects, which strengthened my interests in machine learning and inspired me to venture into the doctoral program. His constant support and belief in me were important factors in my career choices as well as in my progress during the doctoral studies.

Doina Precup was a great source of inspiration throughout my work on this thesis. She constantly provided me with support and rich feedback, which allowed me to become more competent and mature as a researcher and a writer. Her open, encouraging attitude allowed me to shape and develop my own interests while greatly benefiting from her valuable expertise in the field. I am deeply grateful for all the time she devoted to our discussions, to writing of joint papers and to editing of this thesis. She also supported me in numerous opportunities to obtain exposure to our research community, to teach and to work on interesting projects outside the scope of this thesis.

I would also like to thank Prakash Panangaden, who was a member of my Ph.D. progress committee, for the feedback he provided and interest he showed in my work.

I am grateful to Charles Anderson, the external examiner of this thesis, as well as to all the members of the thesis defense committee, Greg Dudek, Ricard Gavaldà, Shie Mannor, Prakash Panangaden, Joelle Pineau, Doina Precup and Denis Thérien,

for their valuable comments and suggestions that helped me to improve the final version of the thesis.

Thanks to Michael Kearns for having introduced me to the field of reinforcement learning and for having inspired me to work in this area.

The last part of this thesis was inspired and benefited from interesting discussions with Chris Hobbs from Nortel Networks.

Thanks to Ricard Gavaldà for his collaboration on the work pertaining to my Master's thesis, which was an important stepping stone in my graduate program.

The School of Computer Science at McGill University and the Learning and Reasoning Lab, in particular, provided a stimulating environment for research. I benefited from many interesting courses offered by the department and an infrastructure conducive to successful graduate studies. I am thankful to many current and former graduate students and lab members with whom I had many interesting discussions. Most of all, I would like to thank Danielle Azar, Masoumeh T. Izadi, Swaminathan Mahadevan and Martin Stole. Danielle Azar has also been my dearest friend; her presence, help and support were extremely precious to me.

I am grateful for the financial support that I received over the years of my graduate studies. Most of this research was supported by NSERC: through research grants of my thesis advisors, Denis Thérien and Doina Precup, as well as through an NSERC doctoral scholarship that I received during 1999-2000. I also benefited from a CCIFQ scholarship that enabled my internship at the University of Paris 6. I am thankful to Vincent Corruble from the University of Paris 6 for providing me with an opportunity to visit LIP6 and to collaborate in his research projects.

Finally, I want to extend my gratitude to all of my family. To my parents, Olesia and Irynej, and to my sister, Zoriana, for encouraging and supporting my interests in science and in research. Without their help and sacrifices, I would not have been able to reach this point. To my husband, Andrij, and to our son, Borys, for all the patience and understanding that they had for me during the challenging and

demanding times of the graduate program, and most of all, for their love that makes my accomplishments so much more enjoyable.

Abstract

Reinforcement learning is a general computational framework for learning sequential decision strategies from the interaction of an agent with a dynamic environment. In this thesis, we focus on value-based learning methods, which rely on computing utility values for different behavior strategies. Value-based reinforcement learning methods have a solid theoretical foundation and a growing history of successful applications to real-world problems. However, most existing theoretically-sound algorithms work for small problems only. For complex real-world decision tasks, approximate methods have to be used; in this case there is a significant gap between the existing theoretical results and the methodologies applied in practice. This thesis is devoted to the analysis of various factors that contribute to the difficulty of learning with popular reinforcement learning algorithms, as well as to developing new methods that facilitate the practical application of reinforcement learning techniques.

In the first part of this thesis, we investigate properties of reinforcement learning tasks that influence the performance of value-based algorithms. We present five domain-independent quantitative attributes that can be used to measure various task characteristics. We study the effect of these characteristics on learning and how they can be used for improving the efficiency of existing algorithms. In particular, we develop one application that uses measurements of the proposed attributes for improving exploration (the process by which the agent gathers experience for learning good behavior strategies).

In large realistic domains, function approximation methods have to be incorporated into reinforcement learning algorithms. The second part of this thesis is devoted to the use of a function approximation model based on Sparse Distributed Memories

(SDMs) in approximate value-based methods. Like for all other function approximators, the success of using SDMs in reinforcement learning depends, to a large extent, on a good choice of the structure of the approximator. We propose a new technique for automatically selecting certain structural parameters of the SDM model on-line based on training data. Our algorithm takes into account the interaction of function approximation with reinforcement learning algorithms and avoids some of the difficulties faced by other methods from the existing literature. In our experiments, this method provides very good performance and is computationally efficient.

Résumé

L'apprentissage par renforcement est une approche computationnelle générale pour l'apprentissage de stratégies de décision séquentielles à partir de l'interaction de l'agent avec un environnement dynamique. Dans cette thèse, on se concentre sur les méthodes d'apprentissage basées sur les valeurs qui dépendent du calcul des valeurs d'utilité pour différentes stratégies de comportement. Les méthodes d'apprentissage par renforcement basées sur les valeurs ont une fondation théorique solide et un historique croissant d'applications à des problèmes réels. Cependant, la plupart des resultats théoriques sont basés sur des suppositions restrictives qui ne sont satisfaites que par des problèmes simples. Pour des problèmes de décision réels plus complexes, des méthodes d'approximation doivent être utilisées. Dans ce cas-là, il existe une grande différence entre les résultats théoriques existants et les méthodologies appliquées en pratique. Cette thèse est consacrée à l'analyse des divers facteurs contribuant à la difficulté de l'apprentissage par des algorithmes bien connus d'apprentissage par renforcement ainsi qu'au développement de nouvelles méthodes facilitant l'application pratique de ces techniques.

Dans la première partie de cette thèse, on recherche les propriétés des problèmes d'apprentissage par renforcement qui influencent la performance des algorithmes basés sur les valeurs. On présente cinq attributs quantitatifs indépendants du domaine qui peuvent être utilisés dans le but de mesurer plusieurs caractéristiques du problème. On étudie l'effet de ces caractéristiques sur l'apprentissage et la manière dont ils peuvent être utilisés pour améliorer l'efficacité de ce dernier. En particulier, on développe

une application qui utilise des mesures des attributs proposés pour améliorer l'exploration (la procédure par laquelle l'agent accumule de l'expérience pour l'apprentissage de bonnes stratégies de comportement).

Dans des domaines réalistes de grande taille, les méthodes d'approximation de fonctions doivent être incoporées dans les algorithmes d'apprentissage par renforcement. La seconde partie de cette thèse est consacrée à l'utilisation d'un modèle d'approximation de fonctions basé sur "Sparse Distributed Memories" (SDM) dans les méthodes approximatives basées sur les valeurs. Comme c'est le cas pour tous les approximateurs de fonctions, le succès de l'usage des SDM en apprentissage par renforcement dépend, pour une grande partie, du bon choix de la structure de l'approximateur. On propose une nouvelle technique pour le choix automatique de certains parmètres structuraux du modéle SDM en-ligne basé sur un ensemble d'apprentissage. Notre algorithme prend en considération l'interaction particulière de l'approximation de fonction avec les algorithmes d'apprentissage par renforcement et évite certaines des difficultés envisagées par d'autres méthodes semblables qui existent dans la litérature. On démontre empiriquement que notre méthode fournit une très bonne performance et qu'elle est efficace du point de vue calcul.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Reinforcement learning [Sutton and Barto, 1998] is a computational approach that allows a software *agent* to learn how to behave by interacting with its environment. For example, imagine an office-helper robot that needs to learn how to optimally schedule and manage tasks such as delivering mail and coffee to the employees as well as tidying up their offices. The learning agent is situated in a *dynamic environment*, which is characterized by a particular *state* at any given time. In the previous example, the state description could include the current location of the robot in the office space, its current engagement in the execution of any task and outstanding requests from the employees. The agent interacts with the environment at discrete time steps by observing the corresponding states of the environment and performing certain *actions*. For instance, the helper robot's actions could include picking up mail at the mail boxes, delivering mail or coffee to a particular employee, tidying up an office. The dynamics of the environment is influenced by the agent's actions as well as factors external to the agent (out of its control). For example, when the helper robot completes an outstanding task, this task will not be part of the state description on the next time step (effect of the agent's action) but, at the same time, employees can make new requests (these are external factors, not under the agent's control).

One of the distinguishing features of reinforcement learning is that neither the designer of the learning system nor the environment provide examples of "correct" actions. The only feedback that the agent receives is in the form of numerical *immediate rewards*. For example, the helper robot can receive negative rewards (penalties) proportional to the number and duration of the outstanding requests from the employees. In general, both the rewards and the environment's dynamics (state transitions) can be stochastic.

The goal of the agent is to learn a way of choosing actions, called a *policy* which optimizes some *long-term performance criterion*. Typically, this criterion is related to the total reward accumulated by the agent over time. The agent's policy must take into account the uncertainty (stochasticity) pertaining to the environment's state dynamics and rewards.

In the absence of a "teacher", the agent needs to actively *explore* its environment by trying out different actions in different states and observing the corresponding outcomes. Thus, the agent can learn without any prior knowledge about its environment and without any supervision. The ability of the agent to learn in this manner is very valuable in domains, in which it is easier to design a reasonable immediate reward function than to find a good (optimal) long-term decision strategy analytically.

Reinforcement learning has been applied successfully in various domains. Prominent examples include Samuel's checkers player [Samuel, 1959; 1967], robot navigation control [Lin, 1992], Tesauro's backgammon player, TD-Gammon, [Tesauro, 1994], packet routing in dynamically changing networks [Boyan and Littman, 1994], job-shop scheduling strategies (in the context of NASA space shuttle missions) [Zhang and Dietterich, 1995], elevator dispatch control [Crites and Barto, 1996], maintenance and repair strategies [Bertsekas and Tsitsiklis, 1996], dynamic channel allocation in cellular telephone networks [Singh and Bertsekas, 1997], call admission control and routing in communication networks [Marbach *et al.*, 1998; Brown *et al.*, 1998], switch packet arbitration [Brown, 2001], computing investment strategies for multi-market

commodity trading [Hauskrecht *et al.*, 2001], helicopter control [Bagnell and Schneider, 2001; Ng *et al.*, 2004], irrigation network regulation [Guestrin *et al.*, 2004] and robotic soccer (see e.g., [Fidelman and Stone, 2004; Riedmiller and Merke, 2001; Riedmiller *et al.*, 2000; Stone and Sutton, 2001; Wiering *et al.*, 1999]).

This list illustrates the generality of the reinforcement learning framework and the fact that it can be applied to diverse real-world problems. However, current state-of-the-art reinforcement learning techniques for large-scale real-world problems can be hard to tune and their theoretical properties are not well-understood. Practical applications are often built by employing techniques that are well-studied and known to work well in other contexts. Often, common-sense heuristics and background knowledge of the domain are used to tailor such techniques for a particular task. Unfortunately, the design and implementation process is typically very time-consuming and involves a lot of trial-and-error steps. Despite the generality of the reinforcement learning approach, the process of selection and fine-tuning of the individual components of the learning system often seems to be very problem-specific. One of the goals of this thesis is to investigate how various domain characteristics affect the performance of popular reinforcement learning techniques and how the knowledge of such characteristics can be used to improve the efficiency of learning. The other goal of this thesis is to develop and test certain techniques that allow us to automate (at least partially) the design of reinforcement learning systems.

1.1. Value-Based Reinforcement Learning

In this thesis, we focus on reinforcement learning techniques that are based on computing *value functions*. Value functions map states of the environment or state-action pairs to their utilities. Utilities reflect the long-term desirability of states and actions and accommodate any stochasticity that may be present in the system. Typically, such utilities are defined as expected values of some additive function of the rewards received by the agent over time. Algorithms that compute a value function and then implicitly derive a policy from it are called as *value-based methods*.

Value-based reinforcement learning algorithms have been analyzed thoroughly for problems, in which the state space is finite and small. It is also usually assumed that the actions available to the agent come from a discrete set, which is small relative to the state space. In this case, value functions can be represented by look-up tables, with one value (utility) assigned to each state or state-action pair. Methods using such a representation are often called *tabular methods.* However, if the state space is very large or continuous, this approach is infeasible. In this case, *function approximation* techniques [Friedman, 1994] are used to represent the value functions approximately. We will refer to such techniques as *approximate value-based methods.*

Tabular value-based methods are well understood. In this case, most of the existing algorithms have provable convergence [Bertsekas and Tsitsiklis, 1996] in the limit (with an infinite amount of experience), under the assumption that the agent encounters all states and tries all actions infinitely often. Some results on computational complexity and convergence rates are also available, e.g., [Blondel and Tsitsiklis, 2000; Kakade, 2003; Littman *et al.*, 1995; Szepesvári, 1997] (see Chapter 2 for more details), but few of the algorithms that are most popular in practice are well studied in this respect.

When approximate representations of the value functions are used, even convergence results are much harder to establish. Although function approximation techniques can be incorporated into reinforcement learning methods quite naturally, the theoretical analysis of the combined systems becomes very difficult. This is due to the fact that many of the fundamental assumptions, on which the theoretical analysis is typically based when studying just one class of algorithms in isolation, are violated in the combined setting. At present, theoretical results are available only for some reinforcement learning algorithms, and mostly for a restricted class of value function representations (see Chapters 2 and 5 for more details). However, approximate methods are indispensable for the practical use of reinforcement learning. In fact, approximate algorithms were successfully used in most of the applications mentioned above, even though their theoretical properties are not fully understood. Thus, there

is still a big gap between the available theoretical results and the methodologies that are applied in practice.

1.2. Problem Statement and Objectives

In this thesis, we rely on a standard framework of *Markov Decision Processes* (MDPs) [Bellman, 1957], which is commonly used in reinforcement learning to define formally the interactions between the agent and its environment, as well as the agent's long-term performance goals. An MDP is defined by a tuple $\langle S, A, P, R \rangle$, where S is the environment's state space, A is the agent's action space, P is the probability distribution for state transitions (also called a transition function), and R is the function that represents expected immediate rewards (see Chapter 2 for more details). However, the availability of the transition and reward functions, P and R, in an explicit analytical form is not required as long as the agent can observe samples of state transitions and rewards produced according to these functions.

In this thesis, we focus primarily on value-based reinforcement learning algorithms that learn on-line, by obtaining samples of state transitions and rewards from the direct interaction of the agent with its environment and by processing these samples one at a time. This is in contrast with off-line *planning* algorithms, that rely on a complete and explicit knowledge of the transition and reward functions and use dynamic programming algorithms to compute optimal policies (see Chapter 2). In on-line learning, there is a close interplay between the agent's state of knowledge about its environment and the experience obtained from the environment (see Figure 1.1). The agent's knowledge is reflected in the values of states or state-action pairs, which, in turn, can determine the policy used by the agent to act in the environment while learning. This policy then affects what experience is obtained by the agent for further learning. Of course, the experience is also determined by the environment, as it responds to the agent's actions according to its dynamical model.

In reinforcement learning, most research has focused so far on the interactions between the agent's policy, the experience it generates and the values that are learned

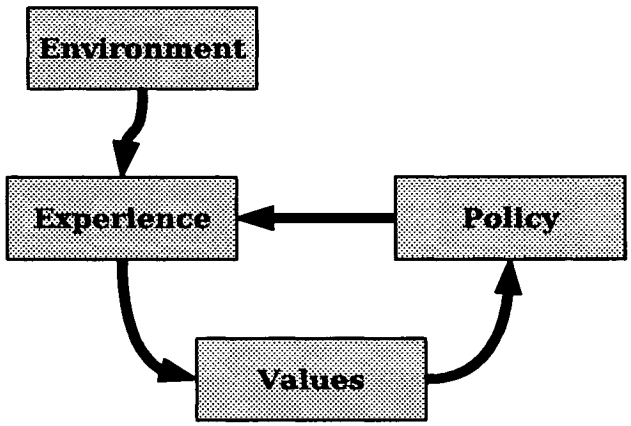


FIGURE 1.1. Main components of on-line value-based learning and direct relationships between them.

from it. The influence of the environment and its properties on the generated experience and subsequently on learning has been studied very little so far. One part of this thesis is devoted to investigating this issue. The other part is focused on the link between the experience and the representation of the value function. In particular, we investigate how to automatically build good approximate representations of values functions.

1.3. Contributions

1.3.1. MDP Characteristics and Their Effect on Learning

In the current reinforcement learning literature, theoretical studies considered mainly a very general definition of Markov Decision Processes. This is in contrast to other related disciplines, e.g., combinatorial optimization [Hoos and Stutzle, 2000; Lagoudakis and Littman, 2000] and supervised learning [Köpf *et al.*, 2000; Linder and Studer, 1999; Peng *et al.*, 2002], where several studies investigated how problem characteristics influence the complexity and behavior of various algorithms (see Chapter 3 for more details). A few existing results in reinforcement learning also provide an indication that not all reinforcement learning tasks are equally easy to solve [Blondel and Tsitsiklis, 2000; Dean *et al.*, 1995; Kakade, 2003; Kirman, 1995;

Littman *et al.*, 1995]. In particular, stochasticity in the environmental responses appears to be one of the key factors influencing the complexity of finding good policies.

One part of this thesis is devoted to defining and analyzing certain properties of Markov Decision Processes that influence the performance of incremental value-based reinforcement learning algorithms. We build on the prior work of Kirman (1995), who proposed several attributes for MDP characterization in order to construct numerical statistical models for predicting the performance of planning algorithms used off-line to solve MDPs with known transition and reward functions. We extend this work to on-line reinforcement learning, both for tabular and approximate methods. We refine the definitions of certain attributes introduced in [Kirman, 1995] and propose some new attributes. The attributes that we investigate are quantitative; most of them measure the amount of stochasticity that comes from different sources and the amount of control that the agent can exercise over its environment. We provide a detailed discussion of the effect that such MDP properties have on the learning process. We formulate specific hypotheses concerning the nature of these effects and experimentally validate some of these hypotheses. For instance, we show how the stochasticity of the state dynamics affects the quality of policies learned by on-line learning techniques in a limited amount of time, as well as how it relates to the agent's ability to explore its environment efficiently. Our analysis opens a new perspective on the difficulties encountered by incremental value-based algorithms and better explains variations in their performance in different domains. This work is presented in Chapter 3 of this thesis. A part of it has been previously published in [Ratitch and Precup, 2002].

1.3.2. Using Knowledge of MDP Properties for Improving Exploration

The insights obtained through the analysis of the MDP characteristics can be used to adjust various parameters of the learning algorithms to better suit the problem at hand. In this thesis, we develop one specific method that relies on two of the proposed MDP attributes to guide the process by which the agent explores its environment

during learning. Exploration is essential for reinforcement learning, as it is the only means by which the agent gathers the information that is necessary to compute good behavior strategies. Efficient exploration methods are of great practical importance and have been studied by many researchers (see Chapter 4 for a review of the research in this area). Our analysis of MDP characteristics led to the design of a new technique that can be used to improve the performance of existing exploration strategies. The agent uses measurements of the MDP attributes in order to focus its exploration effort where it is particularly beneficial. More specifically, the agent gathers more experience for the state-action pairs, whose consequences exhibit high variability; such state-action pairs need more training data in order to obtain good estimates of their values. Also, the agent gives priority to learning about those states, in which its choice of action is expected to have most impact on performance. Our experimental results demonstrate that this technique can improve the quality of the learned policies and the speed of learning. This material is presented in Chapter 4 of this thesis. Parts of this work have also been published in [Ratitch and Precup, 2003b; 2003a].

1.3.3. On-Line Structure Selection for Sparse Distributed Memories in Reinforcement Learning

The second part of this thesis is devoted to approximate value-based reinforcement learning methods. We investigate the use of a function approximation model based on Sparse Distributed Memories (SDMs) [Kanerva, 1993] for representing value functions. A Sparse Distributed Memory can be viewed as a general purpose memory, where states or state-action pairs serve as addresses of memory locations and their values (utilities) represent the contents of locations. When the state space is very large or infinite, memory locations are created only for a small subset of state-action pairs. In this case, like other function approximators, the SDM generalizes the values across "nearby" states. The *structure of the SDM* is defined by the subset of states, for which memory locations are created (*locations' addresses*), as well as by

the *radii of activation* of each location, which determine how values are generalized across different states (see Chapter 5 for more details). We present a new approach for configuring on-line the structure of SDMs by automatically choosing the addresses of the memory locations based on the training data obtained by the agent. Our method has emerged as a result of prior experimentation with other related methods from the existing literature. It avoids some of the difficulties faced by other methods when used in the context of on-line value-based reinforcement learning. Our method provides good performance empirically, is simple to implement and is very efficient in terms of computational time and space requirements. This work is presented in Chapter 5 of this thesis. A part of it has been previously published in [Ratitch and Precup, 2004; Ratitch *et al.*, 2004].

1.4. Statement of Originality

Portions of this thesis have been previously published in peer-reviewed conference proceedings [Ratitch and Precup, 2002; 2003b; 2004] and presented at workshops [Ratitch and Precup, 2003a; Ratitch *et al.*, 2004]. The material presented in this thesis contains more extensive discussions of the proposed techniques compared to the content of the published papers. The thesis also contains a significantly more extensive literature review and in-depth discussions of the relationships between the methods proposed and existing approaches. The experimental studies presented in chapters 3 and 4 contain additional results not included in [Ratitch and Precup, 2002; 2003b].

1.5. Thesis Outline

The rest of the thesis is structured as follows.

Chapter 2 contains background material on reinforcement learning and function approximation. First, we give an overview of tabular value-based methods. Then we briefly discuss the problem of function approximation from a supervised learning perspective and introduce the approximate value-based methods discussed in this

thesis. We then give an overview of the state of knowledge in reinforcement learning combined with function approximation. Finally, we briefly discuss existing results on the complexity of reinforcement learning.

In **Chapter 3**, we define five attributes that can be used to characterize Markov Decision Processes. We propose hypotheses concerning their effect on learning and provide suggestions for their potential use in practical applications. We discuss ways to compute or approximately estimate these attributes. We then present the results of an empirical study illustrating the presence and nature of the effect of two of the proposed MDP attributes on both tabular and approximate value-based methods.

Chapter 4 presents a practical application of the two MDP attributes studied in depth in Chapter 3. We introduce a new exploration strategy, which encourages the agent to gather experience in those regions of the state-action space, where the training samples are either expected to be very noisy, or where learning a good policy will have most impact on the agent's performance. Our empirical results demonstrate that this technique can improve the performance of existing exploration strategies at low additional computation cost.

Chapter 5 is devoted to the use of Sparse Distributed Memories for approximately representing value functions. In this chapter, we discuss how the SDM model relates to the existing theoretical results on approximate value-based methods and how it can be used with several existing approaches. Then, we present the main contribution of this chapter - a new method for configuring the structure of the memory semi-automatically, based on training data obtained on-line. We provide experimental results on two different domains and compare our technique to other existing approaches. The results show that our method performs well in practice and is computationally efficient.

Chapter 6 summarizes the contributions in this thesis and provides directions for future work.

Appendices provide details on some technical issues discussed in this thesis.

CHAPTER 2

Background

Chapter Outline

In this chapter, we provide background information on reinforcement learning. We present definitions of the key concepts needed to formulate and solve reinforcement learning problems and discuss several standard algorithms. Then we introduce the issue of using function approximation to solve large reinforcement learning tasks. We present various ways in which approximate reinforcement learning algorithms can be designed and give an overview of the related theoretical results from the current literature.

2.1. Reinforcement Learning Problem

Reinforcement learning [Sutton and Barto, 1998; Puterman, 1994; Bertsekas and Tsitsiklis, 1996] is a framework for computational learning agents that use experience from their interaction with an environment to improve performance over time. There is no explicit teacher to guide the agent. Instead the agent interacts directly with the dynamic environment in which it operates. The agent uses its sensors to perceive the current state of the environment and is able to perform actions that cause the environment to change its state. The agent also receives a numerical reward signal from the environment, which represents feedback on the agent's immediate performance. But maximizing the immediate reward is not the ultimate goal of the learning agent. Based on this signal, the agent forms a performance criterion, which reflects what is

good in the long term. This criterion incorporates any uncertainty due to the system dynamics and future course of events.

We will consider reinforcement learning problems where time is discrete. We will sometimes refer to the time steps as *stages*. At each stage t, the environment is in some *state* $s_t \in S$, where S is a set of all states. The state space S may be finite or infinite. The *action* a_t performed by the agent at the stage t is selected from a finite[1] set of actions $A(s_t)$ available from state s_t. In Section 2.2, we review reinforcement learning methods that assume discrete, finite state and action spaces. In Section 2.4, we discuss methods for continuous state spaces.

As a result of performing action a_t in state s_t, on the next time step, the agent receives a numerical *reward* signal r_{t+1} and the environment makes a transition to a new state s_{t+1}. The numerical reward signal that the environment provides is the primary means for the agent to evaluate its own performance. In general, this signal is stochastic. It is the means by which the designer of the reinforcement learning system can tell the agent *what* it is supposed to achieve, but not *how*. This constitutes a major difference from supervised learning systems where examples of the desired response are always provided by a teacher. The goal of the agent is to *maximize the cumulative reward* - the long-term *return* - which is an additive function of the reward sequence. Since the environment is stochastic, the agent is supposed to maximize the *expected return*, taking into account any uncertainty pertaining to the system. The agent strives to find a *policy*, a way of behaving, which maximizes this return. Mathematically, a policy is described by a probability distribution $\pi : S \times A \rightarrow [0, 1]$, where $\pi(s, a)$ denotes the probability of taking action a in state s.

One can distinguish two main types of reinforcement learning tasks: *episodic* and *continuing tasks*. In episodic (finite horizon) tasks, there exists at least one terminal state, in which an episode ends. The system then can be reinitialized to some starting

[1]In the current literature, there is some research devoted to MDPs with continuous action spaces, e.g., [Williams, 1992; Baird and Klopf, 1993; Ravindran, 1996; Strösslin and Gerstner, 2003]. We do not consider continuous actions in this thesis.

state and a new episode begins. Continuing tasks or infinite horizon problems, on the other hand, consist of just one infinite sequence of states, actions and rewards.

In general, a return represents a cumulative (additive) function of the reward sequence. For an episodic task, for example, the return is usually defined as the sum of all rewards received from the beginning of an episode until its end:

$$R(s_0) = \sum_{k=0}^{T-1} r_{k+1} \tag{2.1}$$

where T is the number of stages in the episode and s_0 is a starting state. In the case of continuing tasks, there are many problems where one values rewards obtained in the near future more than those received later. In this case, future rewards are discounted by a factor $\gamma \in (0, 1]$ and the return is defined as:

$$R_t(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \text{ , where } s_t = s \tag{2.2}$$

Discounting with $0 < \gamma < 1$ ensures that the returns from all states are finite. When $\gamma = 1$ (i.e., in the undiscounted case), in order to ensure that the returns are finite, some additional assumptions about the problem need to be satisfied. In particular, there has to exist a set of absorbing states (from which there is a zero probability of exiting), which are reached with probability 1 on any trajectory through the state space, and immediate rewards in these states have to be zero. Undiscounted problems can be considered episodic tasks, for which the number of stages in an episode is not known in advance and is stochastic. An episode ends when the system enters an absorbing state.

There are problems for which discounting future rewards is not appropriate, but there is no absorbing set of states either. In this case, a performance criterion often used is the *average cost per stage*:

$$R_t(s) = \lim_{T \to \infty} (\frac{1}{T} \sum_{k=0}^{T} r_{t+k+1}) \tag{2.3}$$

In order to optimize this criterion, the limit must exist and be finite. The algorithms for optimizing the average cost per stage criterion are, in general, different from methods designed for discounted and episodic tasks [Mahadevan, 1996; Bertsekas and Tsitsiklis, 1996; Kearns and Singh, 1998]. In this thesis, we focus on episodic and discounted problems.

Reinforcement learning relies on the assumption that the system dynamics has the *Markov property*:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} = Pr\{s_{t+1} = s', r_{t+1} = r | s_0, a_0, r_0, s_1, a_1, r_1, ..., s_t, a_t, r_t\}$$
$$\forall s', s_t \in S, a_t \in A(s_t) \tag{2.4}$$

The Markov property means that the next state and immediate reward depend only on the current state and action. Systems that have the Markov property are called *Markov Decision Processes (MDPs)*. In reality, most systems are not strictly Markov, but a lot of them are quite close to MDPs. Reinforcement learning algorithms relying on the Markov property are easier to analyze theoretically and are very useful in practice even when the Markov property does not strictly hold in all states.

An MDP can be characterized by the *transition probabilities*:

$$P^a_{s,s'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \; \forall s, s' \in S, a \in A(s) \tag{2.5}$$

and the *expected value of the immediate reward*

$$R^a_{s,s'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \; \forall s, s' \in S, a \in A(s) \tag{2.6}$$

Together these form the *model of the MDP*.

2.2. Reinforcement Learning Algorithms

The goal of a reinforcement learning algorithm is either to evaluate the performance of a given policy (*prediction problem*) or to find an optimal policy (*control problem*). Many reinforcement learning algorithms are based on estimating *value*

functions. Value functions are defined with respect to policies and reflect their goodness (evaluations). They represent the desirability of different states or state-action pairs with respect to the performance goals of the agent. The *state-value function for policy* π for the discounted infinite horizon problem is defined as the expected return:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\},\ \forall s \in S \tag{2.7}$$

Another value function that is often useful is the *action-value function*:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\}\ ,\forall s \in S, a \in A(s) \tag{2.8}$$

It represents the expected return starting from state s, taking action a in s and then following policy π thereafter.

The state-value function satisfies a recursive equation that, for the discounted case, has the following form:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = \sum_{a}\pi(s,a)\sum_{s'}P^{a}_{s,s'}[R^{a}_{s,s'} + \gamma V^{\pi}(s')]\ ,\forall s \in S \tag{2.9}$$

This is the *Bellman equation* for state values and represents the relationship between the value of a state and the values of its successors [Bellman, 1957]. This system of equations has a unique solution, which is the state-value function for policy π [Puterman, 1994]. A similar equation is satisfied by the action-value function:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} = \sum_{s'}P^{a}_{s,s'}[R^{a}_{s,s'} + \gamma V^{\pi}(s')]\ ,\forall s \in S \tag{2.10}$$

Value functions are useful because they define a partial ordering over policies. A policy π is considered to be better than another policy π' if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$, $\forall s \in S$. The *optimal policy* is a policy corresponding to the maximum state-value function V^*, which is called the *optimal state-value function*[2]. The optimal state-value

[2]The optimal policy can also be defined as the policy corresponding to the minimum state-value function, in which case the return can be viewed as a function of immediate costs or penalties.

function satisfies the *Bellman optimality equation*:

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} P^a_{s,s'} [R^a_{s,s'} + \gamma V^*(s')] \, , \forall s \in S \tag{2.11}$$

For any finite MDP, there exists a unique solution V^* to this system of equations, which is achievable by a deterministic optimal policy π^* (see e.g., [Puterman, 1994]). All algorithms discussed in this section assume that the value functions are represented by look up tables, in which there is an entry for the value of each state (or state-action pair).

The optimal policy can be obtained from the optimal state-value function by one-step look-ahead search. For each state, there will be at least one action at which the maximum is attained in the Bellman optimality equation. A policy that assigns non-zero probabilities to such actions and zero probability to all others will be an optimal policy. This policy is *greedy* with respect to V^* as well as optimal in the long run.

2.2.1. Dynamic Programming

Dynamic programming methods [Puterman, 1994] can compute optimal policies for MDPs, given a model of the environment (2.5)-(2.6), by using the value functions and Bellman equations to guide the search for optimal policies. Rather than solving the Bellman equations directly, dynamic programming methods treat them as recursive update rules. Dynamic programming algorithms are *bootstrapping* - they update the estimates of state values based on the estimates of the values for the successor states. We present a brief overview of the most popular dynamic programming algorithms.

The *policy evaluation* task is concerned with estimating the values of states when the agent acts according to some fixed policy π. We discuss the case of deterministic policies, but all results extend easily to the general case of stochastic policies. Assume that the agent has adopted a deterministic policy $\pi : S \rightarrow A$, where $\pi(s) = a \in A(s)$ and is interested in computing the state-value function V^π associated with this policy.

The agent starts with some arbitrary initial estimate of the state-value function, V_0^π, and uses the Bellman equation for the state-value function as a recursive update rule to improve the estimates:

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} P_{s,s'}^a [R_{s,s'}^a + \gamma V_k^\pi(s')], \ \forall s \in S, \ a = \pi(s), \ k = 0, 1, 2, ... \tag{2.12}$$

This is an iterative algorithm for policy evaluation, where one iteration consists of the updates being made to all states. Policy evaluation can be shown to converge in the limit with probability 1 to the correct state-value function V^π, which is the unique solution to the Bellman equation. This can be shown based on the fact that the operator $(T_\pi V_k^\pi)(s) = \sum_{s'} P_{s,s'}^a [R_{s,s'}^a + \gamma V_k^\pi(s')]$ is a *contraction mapping*[3] (see e.g., [Bertsekas and Tsitsiklis, 1996] for details). A similar algorithm can be used to estimate the action-value function Q^π.

Estimating value functions is particularly useful for finding better policies. The *policy improvement* algorithm uses the action-value function to improve the current policy. If $Q^\pi(s,a) \geq V^\pi(s)$ for some $a \neq \pi(s)$, then it is better to select action a in state s rather than to select $\pi(s)$. This follows from the policy improvement theorem [Bellman, 1957], which states that for any pair of deterministic policies π and π', such that $\forall s \in S, Q^\pi(s, \pi'(s)) \geq V^\pi(s)$, policy π' must be as good as or better than π. In this manner one can construct a new improved policy π', which is greedy with respect to V^π:

$$\pi'(s) = \arg\max_{a \in A(s)} Q^\pi(s,a) = \arg\max_{a \in A(s)} \sum_{s'} P_{s,s'}^a [R_{s,s'}^a + \gamma V^\pi(s')] \tag{2.13}$$

Policy evaluation and policy improvement can be interleaved to construct a sequence of successively improving policies. This algorithm, known as *Policy Iteration*, constructs a sequence: $\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow ... \rightarrow \pi^* \rightarrow V^{\pi^*}$. Policy Iteration

[3]We say that the operator T_π is a *contraction mapping* if there exists a vector $\xi = (\xi_1, ..., \xi_n)$, such that each $\xi_i > 0$ and there exists a scalar $\beta < 1$ such that $\|T_\pi V - T_\pi \bar{V}\|_\xi \leq \beta \|V - \bar{V}\|_\xi$, where $\|V\|_\xi = \max_{i=1,...,n} \frac{|V(i)|}{\xi_i}$. If the operator T_π is a contraction mapping than it is guaranteed to have a *fixed point* V_{fp}, that is a vector that satisfies $T_\pi V_{fp} = V_{fp}$ (see e.g., [Bertsekas and Tsitsiklis, 1989]).

converges to the optimal policy, because there is a finite number of policies in a finite MDP (i.e., an MDP with the finite state and action spaces) and each new policy improves on the previous one.

One difficulty with the Policy Iteration algorithm is the fact that the policy evaluation step converges only in the limit. However, policy evaluation can be stopped before convergence occurs. The *Value Iteration* algorithm performs just one policy evaluation iteration (one sweep over the state space) followed by a policy improvement step:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma V_k(s')] \ , \ \forall s \in S \tag{2.14}$$

Value Iteration estimates the value of the optimal policy directly and can be seen as turning the Bellman optimality equation into an update rule, similarly to policy evaluation. Value Iteration converges in the limit to the optimal value function V^* due to the contraction property of the operator (2.14) [Bertsekas and Tsitsiklis, 1996].

Policy Iteration relies on the full convergence of policy evaluation while Value Iteration performs just one policy evaluation step between successive policy improvement steps. An intermediate solution is to perform some fixed number ($k > 1$) of policy evaluation steps before the policy improvement step. This variation, called *Optimistic Policy Iteration*, also converges under certain conditions [Tsitsiklis, 2002], and it can be more efficient than Value Iteration because policy evaluation iterations are less expensive than Value Iteration ones when the number of actions is large.

One drawback of the above algorithms is that they require an update of the value function over the entire state space on each iteration. In domains with large state spaces, one spends a long time on one iteration before any improvements in performance are made. *Asynchronous dynamic programming* algorithms are iterative dynamic programming algorithms that allow updating states in an arbitrary order. In fact, some of the states may be updated several times before the others get their turn. To converge correctly, these algorithms require that all states continue to be updated infinitely often in the limit. Asynchronous dynamic programming does not guarantee that an optimal policy is reached with less computation, but it obtains

policy improvements earlier. This approach allows running dynamic programming algorithms on-line (although the MDP model is still required). In this case, the states actually observed by the agent during the interaction with the environment are selected for updates.

2.2.2. Monte Carlo Methods

Dynamic programming methods can only be used when a model of the system, i.e. transition probabilities and expected rewards as defined in Equations (2.5)-(2.6), is available. Of course, an agent can learn a model and then use it in dynamic programming methods. However, learning a value function directly from interaction with the environment is also possible. Monte Carlo methods estimate value functions directly based on the experience of the agent. By experience we mean sample sequences of states, actions and rewards obtained from interaction with the environment. The idea underlying Monte Carlo methods in general is to use samples of a random variable to estimate its expected value as the sample mean. Recall that value functions are actually expected values of long-term returns, which are random variables. Monte Carlo methods estimate state or state-action values based on *averaging sample returns* observed during the interaction of the agent with its environment. Monte Carlo methods are well-defined for episodic tasks, since samples of complete returns can be obtained for finite-horizon tasks. For each state (or state-action pair) a sample return is the sum of the discounted rewards received starting from the occurrence of the state (or state-action pair) until the end of an episode. As more samples are observed, their average converges to the true expected value of the return under a fixed policy used by the agent for generating the sample sequences.

One can design a Policy Iteration algorithm, in which the policy evaluation step estimates the value function using Monte Carlo methods. There is one complication, however, that does not arise in synchronous dynamic programming. If the agent adopts a deterministic policy π, then the experience generated by its interaction with the environment contains samples only for actions suggested by policy π. The values

for other actions will not be estimated and there will be no information on which to base the policy improvement step. Therefore, *maintaining sufficient exploration* is key for the success of Policy Iteration using Monte Carlo methods. One solution is for the agent to adopt a stochastic policy with non-zero probabilities of selecting all actions from all states: a *soft stochastic policy*, such that $\pi(s,a) > 0,\ \forall s \in S, \forall a \in A(s)$. There are different ways to implement this approach.

In the case of *on-policy methods*, the agent uses a soft stochastic policy when it interacts with the environment to generate experience, and evaluates its performance under this policy. In order to benefit from the currently available knowledge and to do sufficient exploration at the same time, the agent can bias its policy to take greedy actions more often. For instance, the agent can use an *ϵ-greedy policy*, which selects with probability $(1-\epsilon)$ the action that is greedy with respect to the current estimate of the action-value function and with probability ϵ any action uniformly randomly (where ϵ has a small positive value). Another popular choice of soft policy relies on the Boltzman distribution:

$$\pi(s,a) = e^{\frac{Q(s,a)}{\tau}} / \sum_{b \in A(s)} e^{\frac{Q(s,b)}{\tau}}$$

where τ is a positive temperature parameter. In this case, an action with a higher value will have a higher probability of being selected. However, the differences between the action selection probabilities are typically much smoother than in the case of the ϵ-greedy policy: two actions with similar values would have similar probabilities of being selected. Both the parameter ϵ in the ϵ-greedy strategy and the temperature τ in the Boltzman distribution can decrease to zero in the limit, in order to gradually approach a greedy policy.

Another approach is *off-policy* learning: the agent uses one policy to interact with the environment and generate experience (*behavior policy*), but estimates the value function for a different policy (*estimation policy*). This can be done by weighting the samples of returns by their relative probabilities under the estimation and behavior policies [Sutton and Barto, 1998]. Such weights can be computed based only on the

knowledge of the action selection probabilities of the estimation and behavior policies, without the need to know transition probabilities. An immediate advantage of this approach is that the estimation policy can be deterministic while the behavior policy is stochastic and fixed, ensuring sufficient exploration. In particular, an agent can try to learn the value of the optimal policy while following an arbitrary stochastic policy, as we will discuss later.

Policy Iteration with Monte Carlo based policy evaluation converges in the limit to the optimal policy (both for on-policy and off-policy learning) as long as every state-action pair is visited infinitely often [Bertsekas and Tsitsiklis, 1996]. But in practice, one encounters the same problem as for dynamic programming methods: one cannot wait forever until the policy evaluation step converges.

Variants of Optimistic Policy Iteration with Monte Carlo based policy evaluation were recently analyzed in [Tsitsiklis, 2002]. In this case a policy evaluation step consists in updating the values of all states (a synchronous algorithm) using an *incremental Monte Carlo update*:

$$V_{k+1}(s) \leftarrow (1 - \alpha_k)V_k(s) + \alpha_k \bar{V}^{\pi_k}(s), \forall s \in S \tag{2.15}$$

where $\bar{V}^{\pi_k}(s)$ is a sample return observed from state s while following a policy that is greedy with respect to the value function V_k. The step-size parameter α_k has to satisfy the following *stochastic approximation conditions*:

$$\begin{aligned} &\alpha_k \geq 0 \text{ for all } k \text{ and with probability 1:} \\ &\textstyle\sum_{k=0}^{\infty} \alpha_k = \infty \\ &\textstyle\sum_{k=0}^{\infty} \alpha_k^2 < \infty \end{aligned} \tag{2.16}$$

This synchronous Optimistic Policy Iteration was shown to converge to the optimal policy with probability 1. A similar algorithm for state-action values $Q_k(s, a)$ was also shown to converge.

Practical implementations of the methods based on Monte Carlo sampling are feasible mostly for episodic tasks, where it is possible to obtain sample returns on

complete trajectories. For continuing tasks, as discussed in [Tsitsiklis, 2002], one possibility is to generate each trajectory for some large number of steps K, such that γ^K is very close to zero and the contribution of the tail of the trajectory is negligible.

2.2.3. Temporal Difference Learning

A combination of the ideas from dynamic programming and Monte Carlo methods yields temporal difference (TD) learning [Sutton, 1988]. Similarly to the Monte Carlo method, this approach allows learning directly from experience without any prior knowledge of the system's model. The feature shared by TD learning and dynamic programming methods is that they both use bootstrapping for estimating value functions.

Policy Evaluation

Let us first discuss TD learning in the context of policy evaluation. TD algorithms make updates of the estimated values based on each observed state transition and on the immediate reward received from the environment on this transition: $(s_t, a_t) \xrightarrow{r_t} s_{t+1}$. One-step TD performs the following update on every time step:

$$V(s_t) \leftarrow V(s_t) + \alpha_t[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.17}$$

This method bootstraps, but it *uses sample updates* instead of full updates as in the case of dynamic programming. Only one successor state, observed during the interaction with the environment, is used to update V instead of using values of all the possible successors and weighting them according to their probabilities as in Equation (2.12).

The major difference between TD learning and Monte Carlo methods is that Monte Carlo methods perform updates based on the entire sequence of observed rewards until the end of an episode, while TD methods use the samples of immediate rewards and next states for the updates. An intermediate approach is to use the *n-step truncated return*, $R_t^{(n)}$, obtained from a sequence of $n > 1$ transitions and

rewards:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^n V_t(s_{t+n+1}) \tag{2.18}$$

To go one step further, one can compute the updates to the value function estimate based on several n-step returns. The family of methods TD(λ), with $0 \leq \lambda \leq 1$, combine n-step returns weighted proportionally to λ^{n-1}:

$$\begin{aligned} R_t^\lambda &= (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}, \quad 0 \leq \lambda < 1 \\ R_t^\lambda &= R_t^{(\infty)}, \qquad\qquad\qquad\quad \lambda = 1 \end{aligned} \tag{2.19}$$

Weights decay by λ with each additional time step. It is easy to see that we obtain Monte Carlo updates by setting $\lambda = 1$, whereas by setting $\lambda = 0$ we obtain the one-step TD method in Equation (2.17). The above algorithm is known as the *forward view of the TD(λ) algorithm*, where the updates to the value function estimates are calculated as:

$$V(s_t) \leftarrow V(s_t) + \alpha_t[R_t^\lambda - V(s_t)] \tag{2.20}$$

Obviously, to implement this algorithm directly, we still need to wait indefinitely in order to compute R_t^λ. This method is rarely used in practice, but it is useful for the theoretical analysis of TD methods.

The *backward view of the TD(λ) algorithm* is a way to implement incrementally the forward view algorithm described above. This variant introduces the use of *eligibility traces*, which establish the eligibility of a particular event for participating in updates of the value function. Eligibility traces are variables $e_t(s)$ associated with each state $s \in S$. These variables are initialized to 0 at the beginning of learning and are updated at each stage t as follows:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{2.21}$$

where λ is the *trace decay* parameter. The trace for a state is increased every time the state is visited and decreases exponentially otherwise. We denote by δ_t the *temporal*

difference or *TD error* at stage t:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.22}$$

The TD error is the difference between the value estimate that can be formed for state s_t based on the currently observed transition, $[r_{t+1} + \gamma V(s_{t+1})]$, and the current value estimate, $V(s_t)$. On every time step, all states are updated proportionally to their eligibility traces:

$$V(s) \leftarrow V(s) + \alpha \delta_t e_t(s), \forall s \in S \tag{2.23}$$

As a result, all states receive credit (or blame) for the current TD error, but states visited in the distant past are given less credit (or blame), since their eligibility traces have decreased.

There are two ways of performing updates. In on-line (intra-sequence) updating, changes to the value function are made as soon as the appropriate increment $\alpha \delta_t e_t(s)$ is computed. In off-line (after-sequence) updating, the updates are accumulated and the estimates are changed to new values only at the end of an episode. Like Monte Carlo methods, off-line updating is well-defined for finite-horizon tasks.

TD(1) with off-line updating is the Monte Carlo method. TD(1) with on-line updating yields an approximation of Monte Carlo estimation [Sutton, 1988]. This incremental implementation of Monte Carlo methods is much more general: it can be directly applied to discounted continuing tasks, not just to episodic ones.

The TD(λ) method for policy evaluation was introduced in the work of Sutton (1988). There, two convergence results were presented: (i) The tabular after-sequence TD(0) algorithm for evaluating a *proper* policy[4] converges in the limit in expected value to the true value function of the policy. (ii) The off-line TD(0) method converges in the limit to the *optimal predictions* when it is repeatedly presented with a finite training set. Optimal predictions in this case are expected values of the states

[4] As defined in [Bertsekas and Tsitsiklis, 1996], a stationary policy π is called *proper* if, when using this policy, there is a positive probability that the termination state will be reached after at most n stages, regardless of the initial state.

under the maximum-likelihood estimates of the true process parameters (transition probabilities and rewards) obtained from the data. Optimality is ensured for the states that appear in the training set only. Under the same conditions, TD(1) converges to the estimates that minimize the error on the training set (in general, these are different from the optimal predictions in the above sense).

In [Sutton, 1988], the equivalence of forward (intra-sequence) updating and backward (after-sequence) updating for general λ was shown in the context of undiscounted episodic tasks. In [Watkins, 1989], this result was extended to discounted continuing tasks based on the n-step and λ returns, as explained above. The terms "forward" and "backward" views were first introduced in [Sutton and Barto, 1998].

The proof of convergence in the mean of the tabular TD(λ) algorithm for general λ was presented in [Dayan, 1992] based on the ideas involving n-step returns and λ-returns introduced in [Watkins, 1989]. Dayan (1992) also showed convergence with probability 1 of the tabular TD(0) algorithm.

In [Jaakkola *et al.*, 1994], convergence with probability 1 was shown for TD(λ) for general λ both under off-line and on-line training. Their proof relies on techniques from stochastic approximation theory, which were extended to cover asynchronous processes having a contraction property with respect to some maximum norm. The step-size parameter α_t has to satisfy the stochastic approximation conditions, given in Equation (2.16).

Control Problem

The temporal difference approach can be used for the policy evaluation step of the (generalized) Policy Iteration algorithm. As with Monte Carlo methods, sufficient exploration must be ensured in order to find the optimal policy. Again, one can use either on-policy or off-policy approaches to ensure adequate exploration.

An example of TD-based on-policy learning is the *Sarsa* algorithm, first introduced in [Rummery and Niranjan, 1994], which performs the following update on

every time step:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.24}$$

The algorithm estimates the action value function for the current behavior policy, which is near-greedy (e.g., ϵ-greedy) with respect to the last estimate of the action-value function Q. This algorithm converges in the limit with probability 1 to an optimal near-greedy policy if all state-action pairs are visited infinitely often. Convergence to the optimal greedy policy occurs if the amount of exploration (as controlled, for example, by ϵ) diminishes appropriately over time [Singh *et al.*, 2000].

Eligibility traces can also be used with the Sarsa algorithm. The resulting method is known as Sarsa(λ), where the parameter λ plays the same role as in TD(λ). In this case, one trace $e_t(s, a)$ is associated with each state-action pair. On each time step t, the value update is then performed as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t e_t(s, a)[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)], \forall s \in S, \forall a \in A(s) \tag{2.25}$$

The traces $e(s, a)$ accumulate visitation counts for each state-action pair. On each time step, the value update can be propagated to the state-action pairs that were visited earlier on the current trajectory. This can speed up learning, as illustrated in Figure 2.1.

There are two ways in which the traces are usually updated: replacing traces and accumulating traces methods. For the *accumulating traces* method, the eligibilities are updated similarly to the TD(λ) algorithm:

$$e_t(s, a) := \lambda\gamma e_t(s, a) + \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{otherwise} \end{cases} \tag{2.26}$$

With this method, similar to the case of policy evaluation, traces fade away gradually (by the trace-decay factor λ) when the state-action pair is not visited and are incremented otherwise. In practice, in the control setting, it was noticed that there can potentially be a problem with the accumulating trace method. Consider a situation in

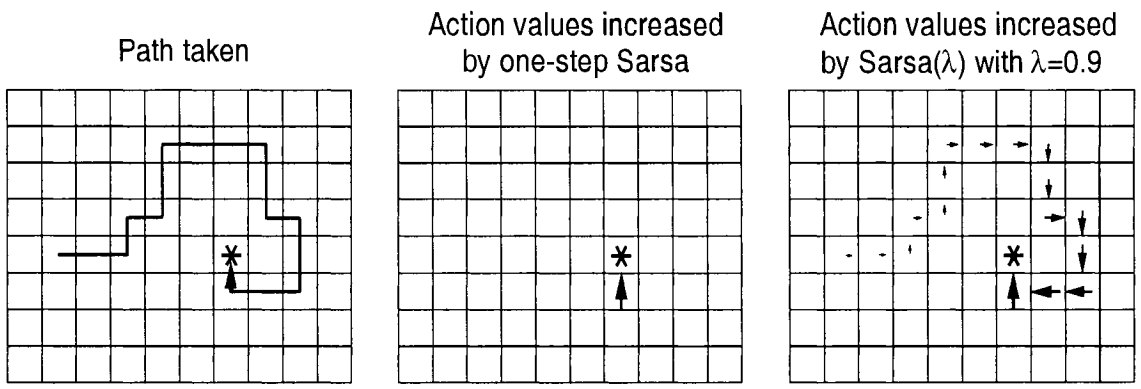


FIGURE 2.1. Effect of using the eligibility traces. One-step Sarsa corresponds to the Sarsa(0) algorithm. In this task, there is one goal state, marked by an asterisk. The rewards are zero everywhere except in the goal state. This picture is taken from [Sutton and Barto, 1998], Figure 7.11, page 181.

which, for a given state, a good action was taken more recently but a bad action was taken more frequently before. In this case, the trace for the bad action is larger than the trace for the good action. When a high reward is finally received, it is thanks to the good action. However, the value of the bad action will go up more than the value of the good action, due to the eligibility traces. Learning can thus be significantly slower. The *replace trace* method was introduced to remedy this problem. With this method, the traces are updated as follows:

$$e_t(s,a) := \begin{cases} 1 & \text{for } s = s_t, a = a_t \\ 0 & s = s_t \text{ and } \forall a \neq a_t \\ \lambda\gamma e_t(s,a) & s \in S,\ s \neq s_t \text{ and } \forall a \end{cases} \tag{2.27}$$

With this method, the trace of the action that was taken is reset to one, and the traces of the actions that were not taken in the visited state are cleared (replaced by zero).

The theoretical convergence properties of the tabular Sarsa(λ) algorithm for $\lambda > 1$ are still poorly understood (see [Gordon, 2000] for some related results). However, this method is widely used in practice and using values of $\lambda > 0$ is known to provide a significant speedup during learning.

A very popular representative of the off-policy approach is the *Q-learning algorithm*, introduced in [Watkins, 1989]. This algorithm performs the following updates to the action-value function on every time step:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.28}$$

Note that this algorithm can be viewed as a variant of the Value Iteration algorithm, presented in Equation (2.14), where updates to the value function are performed based on transition samples instead of the MDP model.

This Q-learning algorithm estimates the optimal action-value function Q^*, regardless of what behavior policy is followed, since the action value estimate in the update is selected according to the greedy policy from each successor state (maximization at s_{t+1}). If all state-action pairs continue to be updated infinitely often and the step-size parameter satisfies the stochastic approximation conditions (2.16), the algorithm converges in the limit to the optimal action-value function Q^* with probability 1. The first to prove this fact was Watkins (1989) based on n-step and λ returns for discounted continuing tasks using the contraction property argument. More convergence results for the Q-learning algorithm can be found in [Jaakkola *et al.*, 1994] and [Tsitsiklis, 1994].

Two variants of the Q-learning algorithm using eligibility traces were proposed in [Watkins, 1989] and [Peng and Williams, 1996]. Special care must be taken when introducing eligibility traces into Q-learning, since it is an off-policy method. In this case, when an off-policy action is taken, the current TD error should not be used to update the values of the on-policy actions taken on previous steps. The algorithm of Watkins (1989) clears all the traces when an off-policy action is taken. Peng's Q(λ) algorithm is actually a mixture of on-policy and off-policy updates (see [Peng and Williams, 1996] for details). The convergence of these algorithms has not yet been established.

2.2.4. Summary of Reinforcement Learning Algorithms

We will briefly summarize the main dimensions along which the methods were distinguished in this section. The first distinction is between model-based and direct methods. Model-based methods rely on the availability of an explicit system model (either given or estimated from experience) and are represented by the dynamic programming algorithms. Direct methods learn optimal behavior on-line, from the interaction of the agent with the environment without explicitly building a model of the system's dynamics. Monte Carlo and TD methods belong to this class. With these methods, it is extremely important to have good exploration strategies that provide a rich experience for the agent to learn from, but at the same time allow it to reap the benefits of the currently available knowledge (exploration-exploitation trade-off). The exploration issue is also important for asynchronous dynamic programming algorithms that determine which states (or state-action pairs) to update based on the agent's experience.

Another important distinction is between bootstrapping methods (e.g., dynamic programming and temporal difference) and non-bootstrapping (Monte Carlo). Monte Carlo methods can suffer from high variance in the return samples when the environment is stochastic and the episodes are long. TD methods reduce this variance by truncating the returns based on the bootstrapping mechanism. However, the latter can introduce bias into sampled value estimates. A formal theoretical analysis [Kearns and Singh, 2000] of the bias-variance trade-off of the TD(λ) method confirmed that in the tabular case, larger values of λ lead to faster convergence but to higher asymptotic error.

Finally, reinforcement learning includes on-policy and off-policy algorithms: on-policy methods require the behavior and estimation policies to be the same, while off-policy methods allow them to be different.

2.3. Function Approximation

The methods discussed so far were designed and analyzed for the case, in which value functions are represented as tables, with a separate entry holding the value of each state or state-action pair. Such a representation is impractical for large problems. First, the memory requirements cannot always be satisfied. In addition, it is practically impossible to explore exhaustively very large (continuous) state spaces. The application of reinforcement learning algorithms to large problems requires using generalization techniques to approximately represent either value functions or policies.

The idea of using function approximation in reinforcement learning can be traced back to the 1950s. Farley and Clark (1955) represented policies using linear threshold functions and adjusted their parameters by reinforcement learning. Samuel (1959, 1967) was the first to use function approximation methods to learn value functions for his famous reinforcement learning application to the game of checkers. At about the same time, Bellman and Dreyfus (1959) proposed using function approximation methods with dynamic programming.

Today, reinforcement learning with function approximation is an area of very active research. Later in this chapter, we will discuss some of the mainstream approaches in this domain. Chapter 5 of this thesis is devoted to an application of a particular function approximation model, the Sparse Distributed Memory, to on-line value-based reinforcement learning. In that chapter, we will provide more discussion of approximate reinforcement learning methods.

Here we will give a brief overview of the classical function approximation problem, as addressed in supervised learning. In the next section, we discuss the use of function approximation with reinforcement learning.

2.3.1. Function Approximation Problem

Function approximation has been addressed in many disciplines in the past. We will focus on methods for function approximation in the context of predictive learning

[Friedman, 1994]. In this context, knowledge about the system is given by some number of measurable quantities, called variables. The goal is to develop a computational relationship between the input and output variables based on a training set of known input-output samples. Once such a mapping is inferred from the training data, it can be used for predicting the output values of previously unseen data given only the values of the input variables. The inputs and outputs can be continuous and/or categorical variables. When the outputs are continuous variables, the problem is known as regression or function approximation; when the outputs are categorical, the problem is known as classification. Here we will consider only the regression problem with a single output, i.e., we seek to approximate a function f of n input variables, denoted by vector $\mathbf{x} = (x_1, ..., x_n)$, from a given set of L training samples $(\mathbf{x}^i, y_i), (i = 1, ..., L)$, where $y_i = f(\mathbf{x}^i)$. The distribution of the training data $\mathbf{x}^i$ can be arbitrary and is usually unknown.

The problem of function approximation is easy to formulate but difficult to solve. The main source of difficulty is the finite size of the training set in all practical applications. In the case of a training set of size L, the goal is to find an approximation that minimizes some error function on the training set, with the hope that it will be a good predictor for unseen examples too. One possibility is to find an approximation function $\tilde{f}$ that minimizes the Mean Squared Error (MSE):

$$MSE = \sum_{i=1}^{L} P(\mathbf{x}^i)[y_i - \tilde{f}(\mathbf{x}^i)]^2 \tag{2.29}$$

where $P(\mathbf{x}^i)$ is the probability of sample $\mathbf{x}^i$.

The solution to the minimization problem is not unique. There are, in fact, an infinite number of functions that can interpolate L data points yielding the minimum value for the MSE. Thus, one must restrict the search to some set of eligible solutions. This restriction is imposed by a designer, often based on considerations outside the training data.

The distribution of inputs P in (2.29) weighs the error for different training samples. This is very important, because usually it is not possible to reduce the

error to zero for all training samples due to the limited expressive power of the approximation architecture. A better approximation at some points can be obtained only at the expense of a worse approximation at other points. The distribution P specifies how this trade-off should be made. Usually, P is the distribution from which the training samples are drawn.

Function approximation has two degrees of freedom: the choice of an approximation architecture and the choice of a training (learning) method. These choices are often (but not always) coupled, since some learning methods are either specifically designed or are guaranteed to work properly only with certain architectures. We will now briefly review some of the popular architectures and learning methods. This overview is not meant to be exhaustive or to provide a lot of details; we will focus mainly on methods related to the research developed in this thesis.

2.3.2. Approximation Architectures

Many of the existing approximation approaches belong to the class of *dictionary methods* [Cherkassky and Mulier, 1996]. These methods restrict their solution space to a particular class of parameterized functions:

$$f_{\mathbf{w},\mathbf{q}}(\mathbf{x}) = \sum_{m=1}^{M} w_m \phi^m(\mathbf{x}, \mathbf{q}^m) \tag{2.30}$$

The functions ϕ^m are often referred to as basis functions. A particular set of basis functions constitutes a dictionary. These methods can be further subdivided into non-adaptive and adaptive methods. *Non-adaptive methods* use fixed basis functions ϕ^m (with the parameters $\mathbf{q}^m$ preset before any learning takes place) and estimate only the coefficients w_m of the linear combination of the basis functions. *Adaptive methods* also fit the parameters of the basis functions $\mathbf{q}^m$ to the training data. Non-adaptive and adaptive methods are also referred to as *linear* and *non-linear* architectures respectively, although in general, non-linear methods also include architectures that combine fixed basis functions in a non-linear manner.

Approximation methods can be further classified as global and local. *Global methods* use basis functions defined (or with nonzero values) on the entire domain of the input variables (or a large portion of it). *Local methods* use locally defined basis functions. Artificial neural networks [Haykin, 1994] are an example of global non-linear methods. Radial Basis Function Networks are an example of local methods. (See more details below). A mixture of local and global architectures is also possible [Bertsekas and Tsitsiklis, 1996], in which case local approximations can be used to enhance the quality of approximation in parts of the space where the target function has special characteristics.

We will now briefly describe several examples of function approximation architectures related to this thesis.

CMACs Approximators

The CMAC architecture [Albus, 1981] is a linear local function approximator, quite popular in the reinforcement learning community (see e.g., [Santamaria *et al.*, 1998; Sutton and Barto, 1998; Tham, 1995]). The architecture consists of a number of superimposed *tilings*, which are discretization grids (see Figure 2.2). Each tiling is displaced by a small (usually random) amount with respect to all others. Each tiling consists of *tiles*, which are the grid cells. Each input *activates* one tile in each tiling. Each tile is considered to be a binary feature of the input and corresponds to one basis function in the dictionary methods representation (2.30). Thus, when the input activates certain tiles, it is mapped to a set of corresponding active features. This model is sometimes called *tile coding* architecture in the literature for obvious reasons. Each tile has an associated parameter (weight). The approximate function represented by this architecture is computed as follows:

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{Tilings} \sum_{j=1}^{Tiles_i} w_{ij} t_{ij}(\mathbf{x}) \tag{2.31}$$

where w_{ij} are the parameters associated with each tile and $t_{ij}(\mathbf{x}) = 1$ if the input $\mathbf{x}$ activates tile j in tiling i and 0 otherwise. Since the tilings are shifted with respect to

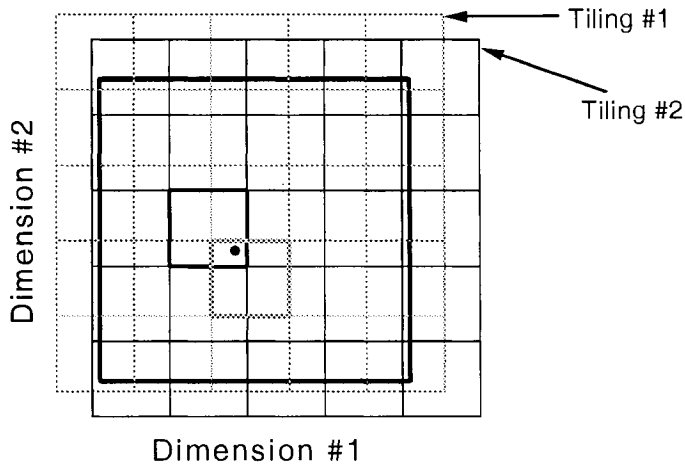


FIGURE 2.2. CMAC architecture for a two-dimensional space. This picture is taken from [Sutton and Barto, 1998], Figure 8.5, page 206.

each other, two inputs that activate the same tile in one tiling can activate different tiles in another tiling. Thus, the values of such inputs will be different. The number of tiles per tiling and the number of tilings determine the resolution of the architecture.

In general, the tiles do not have to be hyper-rectangular; they can be, for example, diagonal stripes, or even have irregular shapes. However, in such cases, an efficient mechanism for identifying active tiles has to be designed. In most practical applications, regular discretization grids are used to form tilings. In this case, the size of the CMAC architecture scales exponentially with the dimensionality of the input space. This factor can be limiting for the applicability of the model to highly dimensional problems. One implementation that is sometimes used to remedy this problem is based on *hashing* [Sutton and Barto, 1998], where multiple tiles are collapsed into one in a consistent pseudo-random manner. In this case, the model is no longer local, since tiles corresponding to distant regions of the input space correspond to one feature.

Radial Basis Function Networks

Radial Basis Function Networks (RBFNs) (see e.g., [Haykin, 1994; Sutton and Barto, 1998; Blanzieri, 2003]) can be seen as a generalization of tile coding to a model

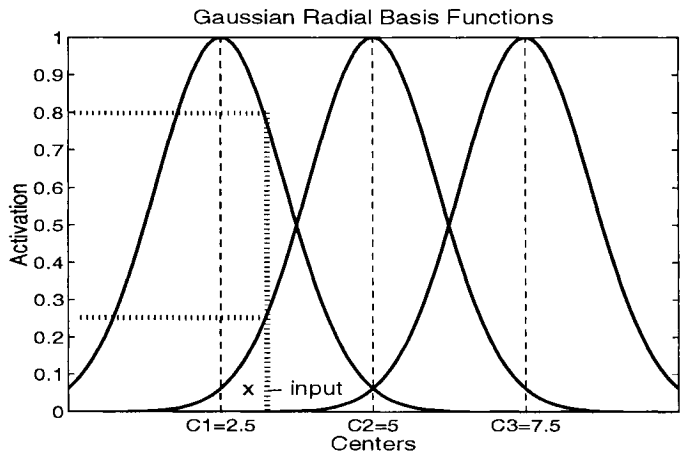


FIGURE 2.3. Radial Basis Functions in one-dimensional input space.

that consists of continuous features instead of binary ones[5]. A Radial Basis Function ϕ^m is a smooth differentiable function that is associated with some center $\mathbf{c}^m$ (a point in the input space) and an activation width σ_m. Probably the most popular type of RBFs is Gaussian (bell-shaped):

$$\phi^m(\mathbf{x}, \mathbf{c}^m, \sigma_m) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}^m\|^2}{\sigma_m^2}\right) \tag{2.32}$$

where $\|\cdot\|$ is some distance metric (most often, the (weighted) Euclidean distance). The shape of the function is such that it gradually diminishes to zero as the input gets farther away from its center, $\mathbf{c}^m$, depending on the value of the RBF's width, σ_m. The centers and widths of different RBFs in the network are chosen such that some of them overlap (see Figure 2.3). Thus, input $\mathbf{x}$ typically activates several basis functions, similar to the case of the CMAC model. However, the activation is not binary, but continuous in the interval $[0, 1]$.

Each RBF has a parameter w_m associated with it, which is sometimes referred to as its height. The approximate function is then represented as follows:

$$f_{\mathbf{w},\mathbf{c},\sigma} = \sum_{m=1}^{M} w_m \phi_m(\mathbf{x}, \mathbf{c}^m, \sigma_m) \tag{2.33}$$

[5]Some implementations of the CMAC architecture are available for non-binary tile activation functions. See, for example, http://www.ece.unh.edu/robots/cmac.htm).

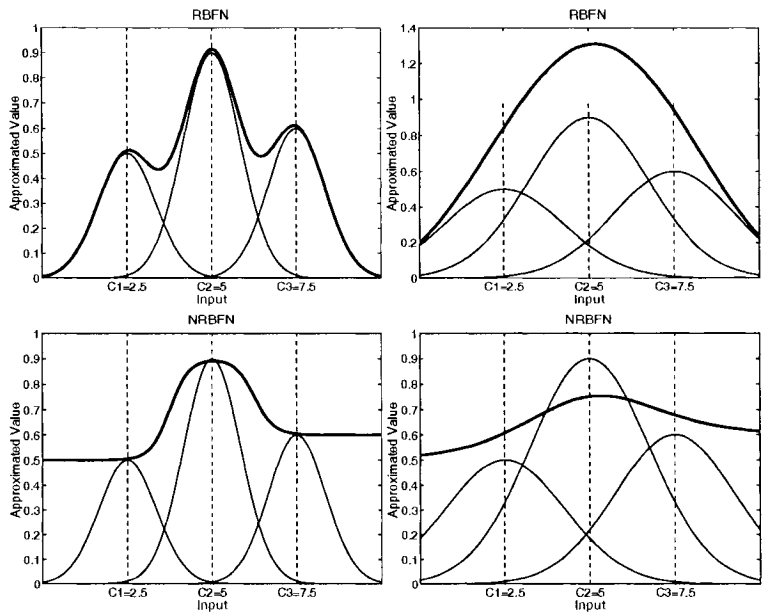


FIGURE 2.4. Radial Basis Functions Networks. Top row: standard RBFNs; bottom row: normalized RBFNs. Weight vectors $\mathbf{w}$ are the same in all four graphs.

Two examples of functions represented by a network of three RBFs are shown in the top row of Figure 2.4. One can see that the smoothness of the approximated function decreases as the overlap between the RBFs decreases.

The RBFN model is local due to the nature of its radial basis functions. When the centers and widths of the radial basis functions are tuned during learning to minimize the objective error function, the model is non-linear, otherwise, it is linear.

A variation on the above RBFN model is the Normalized RBFNs (NRBFNs), in which the approximate function is represented as follows:

$$f_{\mathbf{w},\mathbf{c},\sigma} = \sum_{m=1}^{M} w_m \frac{\phi^m(\mathbf{x}, \mathbf{c^m}, \sigma_m)}{\sum_{k=1}^{M} \phi^k(\mathbf{x}, \mathbf{c^k}, \sigma_k)} \tag{2.34}$$

Using normalization increases the smoothness in the regions of the input space where the basis functions overlap only a little (see Figure 2.4, bottom row). Note that it also affects the way in which the network extrapolates. In RBFNs, the activations of RBFs drop off rapidly towards zero for data points that are far from the centers of

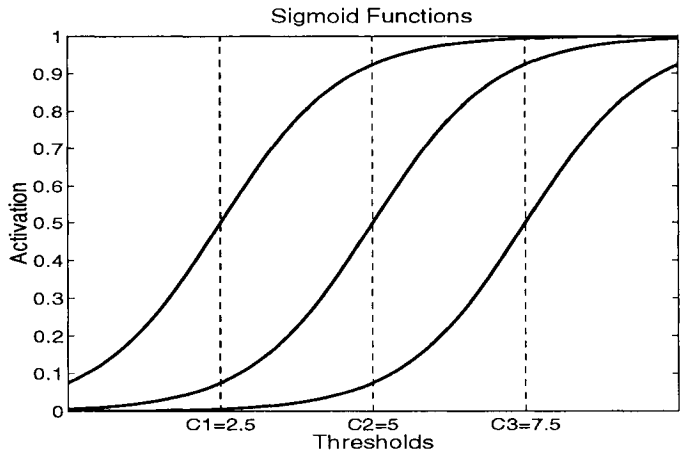


FIGURE 2.5. Sigmoid function in one-dimensional input space. For each sigmoid function, $q_1 = 1$.

the RBFs. Thus, RBFNs are not inclined to extrapolate. However, in an NRBFN, one RBF can dominate the output and hence, extrapolate more significantly.

Sigmoid Neural Networks

A one-layer feed-forward sigmoid neural network (see e.g., [Haykin, 1994]) has the same overall structure as the dictionary models in (2.30). Each basis (or activation) function ϕ^m has a set of associated parameters $q_1^m, ..., q_n^m$, where n is the dimensionality of the input space, and q_0^m, which is often referred to as a threshold. The *standard sigmoid* function is defined as follows:

$$\begin{aligned} &\phi^m(\mathbf{x}, \mathbf{q}^m) = \phi^m(y) = \tfrac{1}{1+\exp(-y)} \text{ , where} \\ &y = \textstyle\sum_{i=1}^{n} q_i^m x_i - q_0^m \end{aligned} \tag{2.35}$$

This function has the following properties (see Figure 2.5):

$$\begin{aligned} &\phi^m(y)_{y \to -\infty} 0 \\ &\phi^m(y)_{y \to \infty} 1 \\ &\phi^m(0) = \tfrac{1}{2} \end{aligned} \tag{2.36}$$

A sigmoid basis function is global, i.e., it has values significantly larger than zero on a potentially unbounded region of the input space. The parameters $\mathbf{q}^m$ of the sig-

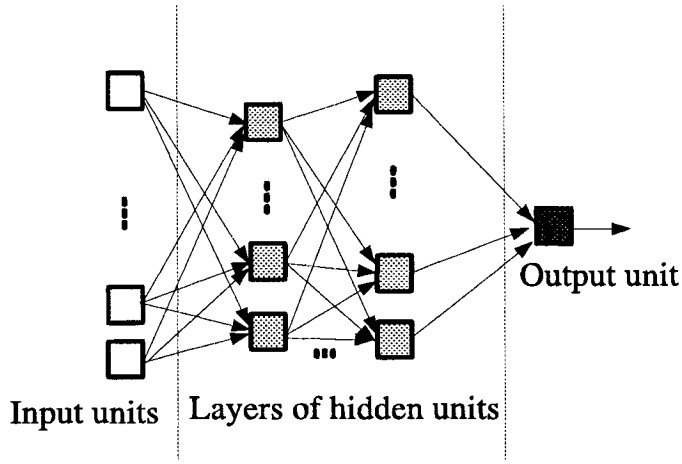


FIGURE 2.6. Feed-forward neural network architecture.

moid functions are typically tuned during learning by a gradient descent method (see Section 2.3.3). Sigmoid neural networks are non-linear, however, when the elements of $\mathbf{q}^m$ are close to zero, the network is close to linear.

There are architectures of sigmoid neural networks which consist of multiple layers of such sigmoid units (see Figure 2.6), which are often called hidden layers or layers of hidden units. In this case, the units are not combined with each other in a linear manner, as in dictionary methods. Every unit in a layer is connected with all the units in the previous layer. These connections are not all equal, each connection may have a different strength or weight. Data enters at the input layer and passes through the network, layer by layer, until it arrives at the output(s).

Classification and Regression Trees

Classification and regression tree (CART) [Breiman *et al.*, 1984] is a model that amounts to partitioning the input space into variable size rectangular regions. Classification trees are used for target functions with a finite discrete set of possible values (e.g., binary), while regression trees are used for continuous target functions. Each internal node of the tree is associated with a test that splits the range of some input variable in two. For example, a real valued variable x_1 can be compared to some real value w_1 (see Figure 2.7).

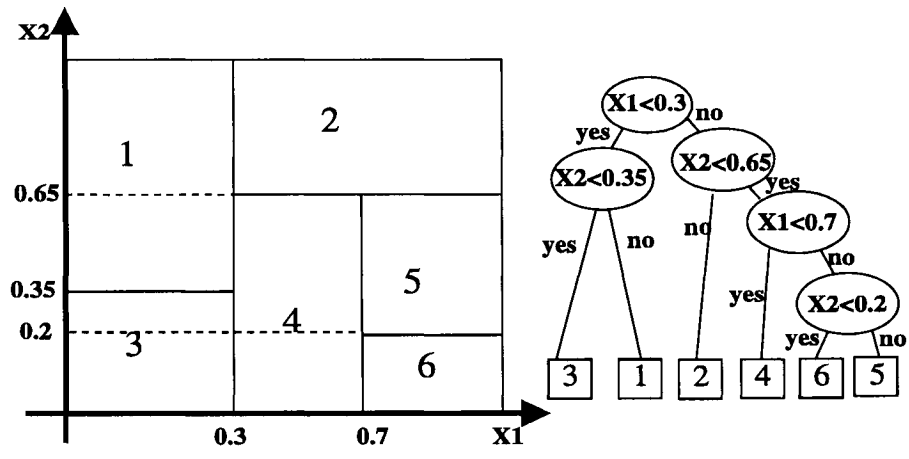


FIGURE 2.7. Classification tree in a two-dimensional input space.

In the case of classification trees, the leaves of the tree contain the classification labels; in the case of regression trees, the leaves contain real constant values (in the simplest case). In order to find a predicted value for some input $\mathbf{x} = (x_1, ..., x_n)$, the tree is traversed from the root down to the leaves. At each internal node, the associated test is performed for the corresponding input variable and depending on the outcome of the test, either the left or right edge is taken to a lower level node. When a leaf node is reached, the predicted value is obtained as a classification label or a constant value associated with this leaf. Such a model provides a local architecture.

For regression trees, the leaves may contain a linear function instead of a constant. In this case, the function at the leaf node reached for a specific input is evaluated for the input variables' values, in order to determine the predicted value. This allows to have multiple approximators, each associated with a different local region. The resulting approximate function is then either piecewise constant or piecewise linear.

Manifold Representations

Manifold representations [Boothby, 1986; Grimm and Hughes, 1995] are another example of local models. In mathematics, a manifold is a topological space that looks locally like the "ordinary" Euclidean space $\Re^n$ and is a Hausdorff space[6]. An example

[6] X is a Hausdorff space, or separated space, iff, given any distinct points $\mathbf{x}$ and $\mathbf{y}$, there exists a neighborhood $U_\mathbf{x}$ of $\mathbf{x}$ and a neighborhood $U_\mathbf{y}$ of $\mathbf{y}$ that are disjoint.

is the surface of a sphere, which globally is not a plane, however its small patches are topologically equivalent to patches of the Euclidean plane. To make precise the notion of "looks locally like", a manifold representation uses local coordinate systems or charts. An atlas of the manifold describes how a complicated space is patched together from simpler pieces (charts) that partially overlap at the edges. This is exactly analogous to the common meaning of atlas, where each individual map in an atlas of the world gives a neighborhood of each point on the globe. While each individual map does not exactly line up with other maps that it overlaps with (because of the Earth's curvature), the overlap of two maps can still be compared (by using latitude and longitude lines, for example).

It is possible to embed a function $F : M \rightarrow \Re$ on a manifold M. Assume that we have D charts $\psi^d : U_d \rightarrow \Re^n$ on a manifold M, where U_d is a subset of M. For each chart ψ^d, we define two functions: $f^d : M \rightarrow \Re$ and $b^d : M \rightarrow [0, 1]$. Every b^d is a blend function, which is smooth with derivatives equal 0 at the boundary of U_d. The functions f^d are some (parametric) approximation functions defined on U_d. Then the approximate function is defined as follows:

$$F(\mathbf{x}) = \frac{\sum_d b^d(\mathbf{x}) f^d(\mathbf{x}, \mathbf{q})}{\sum_d b^d(\mathbf{x})} \tag{2.37}$$

2.3.3. Training Methods for Function Approximation

Gradient Methods

Training methods for function approximation are the algorithms used to find an appropriate setting of the model parameters, w_m and possibly $\mathbf{q}^m$. Let us denote by a vector $\mathbf{w} = (w_1, ..., w_K)$ all the parameters that have to be tuned. One can design a training method by defining a path through the multi-dimensional parameter space of the approximation architecture, and picking the parameter setting along the path that minimizes some cost function $g : \Re^K \rightarrow \Re$, e.g., MSE as in Equation (2.29). Many algorithms specify the path using a descent direction: $\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha_t \mathbf{d}(t)$ where $t = 0, 1, ...$ are the iteration indexes and α_t is a positive learning rate (also

often called a step size). The vector $\mathbf{w}(0)$ corresponds to an initial guess and $\mathbf{d}(t)$ is a direction of the parameter update, such that $\nabla_{\mathbf{w}} g(\mathbf{w}(t))'\mathbf{d}(t) < 0$. I.e., $\mathbf{d}(t)$ is close to the direction of the negative gradient of the error function with respect to the model parameters. One of the most widely used algorithms is the *steepest descent* method (see e.g., [Bertsekas and Tsitsiklis, 1996]), which defines the descent direction as $\mathbf{d}(t) = -\nabla_{\mathbf{w}} g(\mathbf{w}(t))$. When the steepest descent is performed on the MSE function, the approach is known as Least Mean Squares training and is due to [Widrow and Hoff, 1960]. Other gradient methods use second order derivatives to define $\mathbf{d}(t)$, which significantly improves the convergence rate. Among such methods are Newton's method, Quasi-Newton method and Gauss-Newton method (see e.g.,[Haykin, 1994]). In general, these algorithms are more complex and have very time-consuming computations on each iteration. For that reason, they are rarely used in practice, despite the advantage of increased convergence speed (in terms of the required number of iterations).

The step size α_t can be either constant or diminishing to zero over time. A constant step size yields a simple algorithm, but it is not easy to select the right value for it. If the step size is too large, divergence can occur, and if the step size is too small, the rate of convergence may be very slow. For some versions of the gradient descent algorithm, in order to guarantee convergence, it is necessary to ensure that the stochastic approximation conditions (2.16) are satisfied. The diminishing step size rule has good theoretical convergence properties, but the associated convergence rates tend to be slow. It is used primarily in situations in which convergence with a constant step size cannot be achieved.

The steepest descent method, which minimizes the cost function

$$MSE_{\mathbf{w}} = \sum_{i=1}^{L} [y_i - f_{\mathbf{w}}(\mathbf{x}^i)]^2 \tag{2.38}$$

and processes all available data samples before updating the parameter vector, is called a *batch method.* Its update equation is the following:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha_t \sum_{i=1}^{L} [y_i - f_{\mathbf{w}}(\mathbf{x}^i)] \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^i) \tag{2.39}$$

Note that the cost function in Equation (2.38) does not contain explicit weighting with probabilities $P(\mathbf{x}^i)$ as in Equation (2.29), because they are typically unknown. However, training samples are assumed to be drawn independently from the distribution $P(\mathbf{x})$.

When the training set is large, each iteration takes a long time, and progress is delayed. In this case, one can use an *incremental gradient descent method* that cycles through the data and updates the estimates of the parameters $\mathbf{w}$ after each datum is processed (one data point per iteration):

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha_t [y(t) - f_{\mathbf{w}}(\mathbf{x}(t))] \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}(t)) \tag{2.40}$$

Here the error is determined by the training sample $\langle \mathbf{x}(t), y(t) \rangle$ processed at time t.

A cycle of the incremental gradient method through the entire data set differs from the pure steepest descent iteration only in that the gradient is evaluated at the corresponding current estimates of the parameters rather than at the estimates available at the start of the cycle. For very large data sets, the incremental method may converge faster than the batch version. The choice of the step-size α_t plays an important role in the performance of the incremental gradient method. In theory, a diminishing step-size that satisfies conditions (2.16) is essential for convergence to a stationary point of the cost function. The incremental gradient method is particularly suitable for reinforcement learning, because it can be applied on-line, processing training samples one at a time as they become available during the interaction of the agent with the environment.

Memory-Based Learning

Another approach to learning an unknown function is a class of learning methods known as *memory-based learning* or *instance-based learning* [Atkeson *et al.*, 1997a]. They are also often referred to as lazy learning. These methods store all training data samples without processing them in any way. Then, when a query is made, the estimated value is determined by combining the stored data points close to the query in some way. This is why the approach is called lazy learning: no learning takes place until a prediction has to be made. Lazy learning is often used in combination with local approximation architectures and uses locally weighted training to average, interpolate between or otherwise combine the training data. Weighting the data can be viewed as replicating relevant instances and discarding irrelevant instances. Relevance is measured by calculating the distance between the query point and each data point stored in memory. The structure of the function approximator is automatically determined by the data instances that are stored in the memory.

One of the most popular examples of instance-based methods is the *k-nearest neighbor*. With this method, when a prediction has to be made for some query input point $\mathbf{x}$, the distances from this point to all samples stored in the memory are calculated, $D_m = \|\mathbf{x} - \mathbf{x}^m\|, m = 1, ..., M$ (e.g., in terms of the Euclidean distance). Then k sample points from the memory are selected such that their distance to the query point is smallest[7]. The approximate value is then obtained as follows:

$$\tilde{f}(\mathbf{x}) = \sum_{i=1}^{k} \phi^i(D_i) y_i \tag{2.41}$$

where ϕ^i is a weighting kernel function (e.g., Gaussian) and y_i is a target value stored in the memory for the sample input $\mathbf{x}^i$.

This approach to function approximation is *non-parametric*, since the approximate function does not depend on any tunable parameters and is directly determined only by the previously seen training samples.

[7]More efficient implementations exist to quickly identify k nearest samples.

Locally weighted regression uses the same paradigm of lazy learning, but instead of computing the approximate value as in (2.41), it fits a local surface (e.g., quadratic) to the selected nearby points. These points are still weighted by some kernel function proportionally to their distance to the query point (see [Atkeson *et al.*, 1997a] for an excellent introduction to this subject).

2.4. Reinforcement Learning with Function Approximation

In most cases, the incorporation of generalization techniques into reinforcement learning is achieved by applying the algorithms developed for the tabular case to approximate representations of the value functions. If the approximate value function V' is a good approximation of the optimal value function V^* then a greedy policy based on V' is close to optimal [Singh and Yee, 1994; Bertsekas and Tsitsiklis, 1996]. An alternative approach that has become popular in recent research is to search directly for a policy, without relying on an approximation of the value function (see, e.g., [Baxter and Bartlett, 1999; Bartlett and Baxter, 2000; Sutton *et al.*, 2000]). In this case, a function approximator is needed to represent the policy. In this thesis, we do not consider this approach.

The function approximation task in the context of reinforcement learning is quite different from the classical, supervised learning setting. In supervised learning, the training data contains values of the unknown target function at some sample points. Many learning techniques assume a static training set over which multiple passes are performed to find an appropriate setting of the parameters of the function approximation. In value-based reinforcement learning, the objective is to approximate value functions, either optimal or for a particular policy, for which the desired target values are not known in advance. Training examples are generated by a reinforcement learning method, which changes the estimates of the value functions as the learning progresses. Because of this, the target function appears to be non-stationary from the function approximator's point of view. Also, the sampling strategy (behavior policy)

can change during learning if an on-policy near-greedy control algorithm is used. In this case, even the distribution of the training inputs in non-stationary.

While studying the implications of combining function approximation with reinforcement learning algorithms, one has to consider a particular type of architecture and an associated training algorithm. When approximation is introduced into value-based reinforcement learning methods, one should not expect convergence to the optimal solutions. It can simply be impossible to represent the corresponding value functions exactly with a given architecture. Theoretical questions can be asked mainly about convergence in general. Does the algorithm converge? If it does, what are the properties of the limit and how close is the resulting solution to the optimal one? If the algorithm does not converge, does it oscillate in some bounded region or does it diverge? We now give a brief overview of reinforcement learning methods combined with function approximation and of the related theoretical results.

2.4.1. Reinforcement Learning with Least Mean Squares Training

As previously mentioned, Least Mean Squares training aims at minimizing the Mean Squared Error criterion, as in (2.29). The minimization process is performed by the steepest descent algorithm, as described above. For example, in the case of a value-based reinforcement learning algorithm that estimates the action-value function, training samples have the form of $\langle (s_t, a_t), \bar{Q}(s_t, a_t) \rangle$. The inputs (state-action pairs) are sampled according to the agent's behavior policy. The target values $\bar{Q}(s_t, a_t)$ can be formed in various ways. For example, target values can be sample returns obtained by Monte Carlo methods. When such a method is used for the policy evaluation problem, it converges to a locally optimal set of parameters for the function approximation.

Temporal difference methods can also form targets by using their bootstrapping mechanism. For example, in the case of the Sarsa(0) control algorithm, after observing a transition $(s_t, a_t) \xrightarrow{r_t} (s'_t, a'_t)$, the agent can provide the following target for the

function approximation algorithm :

$$\bar{Q}(s_t, a_t) = r_t + \gamma Q_{\mathbf{w}}(s'_t, a'_t) \tag{2.42}$$

where $Q_{\mathbf{w}}$ is the current approximation of the action-value function, which depends on some parameter vector $\mathbf{w}$. In this case, the parameters of the function approximator will be updated in the following way:

$$\mathbf{w}(t+1) := \mathbf{w}(t) + \alpha_t \left[r_t + \gamma Q_{\mathbf{w}}(s'_t, a'_t) - Q_{\mathbf{w}}(s_t, a_t) \right] \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s_t, a_t) \tag{2.43}$$

LMS training is a standard technique in supervised learning. However, in the context of reinforcement learning, various difficulties can arise with the application of this approach to value-function approximation, as we explain next.

The MSE function (2.29) is defined based on the distribution of training inputs, $P(\mathbf{x})$, which serves as the means to balance approximation errors. Thus, inputs that are less frequent in the training data can have greater errors, since their contribution to the overall error function is small based on their occurrence probabilities. When the parameters of an approximator are updated using the incremental gradient descent method as in (2.40) the error at some input $\mathbf{x}$ is reduced. But it is possible that the parameter update, at the same time, increases the errors for some other inputs. In the case of supervised learning, when the distribution of the training samples is *fixed*, this fact does not hinder the convergence of gradient based algorithms. As learning progresses, the function approximator eventually reaches some local optimum at a parameter setting that balances the errors according to the underlying fixed sampling distribution $P(\mathbf{x})$.

With reinforcement learning, however, the fact that the error can increase for some states after a parameter update can be a problem due to the bootstrapping mechanism through which training target values are formed, e.g., as in Equation (2.42). The state-action pair (s'_t, a'_t) is sometimes referred to as a *bootstrapped* pair and its value $Q_{\mathbf{w}}(s'_t, a'_t)$ - as a bootstrapped value. If the error for the bootstrapped state-action pair (s'_t, a'_t) increased on previous steps, such an increased error can now

be incorporated back into the training data through the bootstrapped target. If the state-action pairs, which provide bootstrapped target value estimates, do not appear as training inputs themselves often enough throughout learning, the error is not sufficiently minimized for them, and the parameters of the function approximator may get tuned to minimize the error for unrelated state-action pairs instead. In this case, the errors will propagate through the function approximator, which can even result in divergence. If training data is drawn from an on-policy distribution (i.e., the distribution of the evaluated policy), such a phenomenon can be prevented. In this case, the state-action pairs that provide bootstrapped target values relevant to the evaluated policy are naturally updated with appropriate frequencies[8].

Despite the challenges involved in using LMS training with reinforcement learning, as discussed above, the current literature contains both theoretical analyses and empirical evidence of successful applications of the LMS training in certain contexts. We will give a brief overview of the related theoretical results next.

Theoretical Results on LMS Training with Reinforcement Learning

The TD(λ) method for policy evaluation, combined with the LMS rule and based on a linear function approximator, has been proven to converge with probability one provided that the updates are made according to the on-policy distribution [Tsitsiklis and Van Roy, 1997]. The error bound (in the weighted maximum norm) of the resulting approximation of the value function is derived in terms of the error of the best possible approximation given the chosen architecture and depends on λ. The most favorable bound is obtained for $\lambda = 1$ and the bound deteriorates when λ decreases. But empirically, for many problems, bootstrapping methods (i.e., $\lambda < 1$) converge faster and sometimes lead to better policies when used in conjunction with the policy iteration algorithm [Sutton and Barto, 1998]. A similar (but more general) result was presented in [Tadic, 2001].

[8]This is true mainly for the case of the policy evaluation problem. In the case of the control problem, in which the policy can change during learning, it is necessary that these changes are not too drastic and frequent, so that the approximator has sufficient time to learn with the current sample distribution.

For the control problem, Optimistic Policy Iteration can be used with function approximation. The motivation and the idea of Optimistic Policy Iteration in this case are the same as in the tabular representation: perform a policy update early, before approximate policy evaluation converges. This method is widely used in practice, even though its convergence behavior is not fully understood. Intuitively, Optimistic Policy Iteration can have an additional advantage over the classical Policy Iteration method in the case of function approximation. In approximate Policy Iteration, the policy may become much worse after the policy update step. Since policy evaluation is performed "fully", the new value vector will be very far away from the optimal one at the end of the policy evaluation step. With Optimistic Policy Iteration, once a bad policy is adopted, the new value vector also moves away from the optimal one, but only gradually. There is hope that while this happens, a new, better policy will be adopted. With Optimistic Policy Iteration, the selection of a bad policy may be corrected much faster and the size of the oscillation in the approximate value function will usually be smaller. However, it is unclear if this self-correcting mechanism always works.

As indicated in [Munos, 2003], in Policy Iteration, empirically, a rapid improvement is observed during the first few iterations, but then oscillations tend to occur. After obtaining some policy that is relatively close to the optimal one, the errors in the value function approximation prevent significant improvements and then oscillatory behavior begins.

In Optimistic Policy Iteration, the oscillations are more complex than in classical Policy Iteration. A phenomenon called *chattering* often occurs, in which oscillation in the policy space occurs simultaneously with convergence in the parameter space of the approximation architecture. The limit to which the parameters converge need not correspond to any policy for the problem. It may be a boundary point that separates policies in parameter space and cannot always be used to construct an approximation of the value function of any policy. Optimistic Policy Iteration may

have multiple stable and unstable equilibriums in the parameter space of the value function, depending on the initial setting of the parameter vector.

The bounds on the loss resulting from using the policy greedy with respect to the approximated value function are presented in terms of the maximum norm in [Bertsekas and Tsitsiklis, 1996]. The work in [Munos, 2003] provides similar bounds in terms of value approximation errors according to a weighted quadratic norm, which is usually used by function approximation algorithms. Experiments suggest that error bounds are often comparable for Optimistic Policy Iteration and classical Policy Iteration. However, convergence, as opposed to oscillation, happens rarely. Theoretical results are known only for the case of linear function approximation, as we outline below.

The result in [Gordon, 2000] proves the non-divergence of the Sarsa(λ) algorithm with linear function approximation and LMS training for the case of finite state spaces. Some of the key requirements in this result are the on-policy distribution of the training samples and the fact that the policy does not change during the trajectories (episodes) but only between them. That is, during the entire episode, the agent uses a semi-greedy policy based on the approximation of the value function available at the beginning of the episode. It is only at the end of the episode that updates to the parameters of the function approximator are committed and the new approximation is used to obtain a new semi-greedy policy for the next episode. In this case, the policy, and thus the distribution of states in the training data, does not change too frequently, which allows approximation errors to be better balanced across different states. In essence, on every trajectory, the method behaves as on-policy TD(λ) and the whole learning process converges to some region in the parameter space of the value function. The proof applies to finite MDPs only, as it relies on the fact that there are only a finite number of policies for a given MDP.

This result does not provide much reassurance for practical applications, since learning can either converge to a value function far from the optimal one or oscillate between different policies. Nevertheless, it is an important theoretical result, which

sheds light on the behavior of the Sarsa control algorithm and emphasizes important requirements for a guaranteed non-divergent behavior. In practical on-line applications, however, most often parameter updates are committed after each observed transition and thus the policy can change on every time step.

A related result was obtained in [de Farias and Van Roy, 2000], where it was shown that the update operator corresponding to the Sarsa(0) algorithm with linear function approximation has at least one fixed point when using on-policy exploration based on Boltzman distribution. However, the question of convergence to these fixed points still remains open.

In [Perkins and Precup, 2002], a model-free variant of approximate Policy Iteration, which uses Sarsa updates with linear action-value function approximation for policy evaluation steps, was shown to converge to a unique solution. This result pointed out that divergence in approximate reinforcement learning algorithms for control can be caused by the use of ϵ-greedy behavior policies in the policy improvement step. The ϵ-greedy strategy can make the agent's behavior discontinuous in the value estimates: small changes in the value estimates may result in large changes in the agent's behavior. This can dramatically change the distribution of state-action pairs experienced under subsequent policies and, as explained before, can cause value estimates to be learned under dramatically different weightings of errors on subsequent policy evaluation steps. This makes it difficult to ensure an actual improvement on policy improvement steps. A solution to this problem, suggested in [Perkins and Precup, 2002], was to make the agent's semi-greedy behavior policy be a continuous function of state-action values, e.g., based on the Boltzman distribution. This allows only small changes in the agent's behavior if there are only small changes in the value function. Then, if the behavior policy is Lipschitz continuous[9] in the action values with a constant that is not too large, then the sequence of policies is guaranteed to converge. A suitable magnitude of the Lipschitz constant depends on the environment and on properties of the linear function approximator. However, if the MDP model

[9]A function $f : \Re \rightarrow \Re$ is Lipschitz if there exists a constant $c \in \Re$ such that for all $x, y \in \Re$, $|f(x) - f(y)| \leq c|x - y|$.

is not known, then an appropriate choice of the policy representation is not obvious. If the choice is made on the safe side, by taking a very small constant, then the agent may have a limited ability to exploit its current knowledge, i.e., take greedy actions often enough. In anecdotal evidence, exploration approaches based on the Boltzman distribution often exhibit slower learning and converge to worse policies, compared to ϵ-greedy exploration, when the latter provides stable behavior.

For reinforcement learning methods with off-policy LMS training, examples of divergence exist in the current literature even for linear function approximators, for instance, the divergence of the off-policy TD(λ) [Tsitsiklis and Van Roy, 1996; 1997] and of the Q-learning algorithm [Baird, 1995; Thrun and Schwartz, 1993] with a fixed stochastic exploration policy. Importance sampling can be used to weight the parameter updates by the relative probabilities of the observed training samples under the behavior policy and the evaluated policy. A convergence result for policy evaluation with TD(λ) and importance sampling was presented in [Precup *et al.*, 2001] for episodic tasks and linear function approximators. Q-learning has never been reported to diverge in practice with linear architectures when a soft near-greedy behavior policy is used, but no theoretical analysis exists for this case.

Different attempts were made in the past to deal with the difficulties presented by standard LMS training in the reinforcement learning context. In the following section, we discuss some alternative training methods from the current literature, which have better theoretical guarantees for the control problem. Most of them require linear approximators. More discussion on this subject will also be provided in Chapter 5.

2.4.2. Alternatives to the Least Mean Squares in Reinforcement Learning

Residual Gradient Algorithms

The LMS method for value function approximation, as described before, treats each target value, e.g., $[r + \gamma Q_{\mathbf{w}}(s', a')]$, as a scalar and ignores the fact that, in reinforcement learning, this target depends on the current approximation of the value function. The LMS updates simply attempt to make the value of each state-action

pair (s, a) to match the values of its successors, without taking into account the effect that such updates can have on the corresponding matches for values of their successors and predecessors. Residual gradient algorithms in [Baird, 1995] address this issue directly. These methods explicitly optimize the Mean Squared Bellman Error (MSBE) that can be derived from the Bellman equations corresponding to either the policy evaluation, as in (2.9), or the control problem, as in (2.11). For example, in the case of the control problem, the MSBE measure is as follows:

$$MSBE_{\mathbf{w}} = \frac{1}{2} \sum_{\text{sample } (s,a)} \left[r + \gamma \max_b Q_{\mathbf{w}}(s', b) - Q_{\mathbf{w}}(s, a) \right]^2 \tag{2.44}$$

In this case, incremental updates of the parameters w_i of a parametric function approximator would be performed in the direction of the negative gradient of the MSBE function as follows:

$$\begin{aligned} \mathbf{w}(t+1) := \mathbf{w}(t) - \alpha_t \quad & [r + \gamma \max_b Q_{\mathbf{w}}(s'_1, b) - Q_{\mathbf{w}}(s_t, a_t)] \\ \cdot \quad & [\gamma \nabla_{\mathbf{w}} \max_b Q_{\mathbf{w}}(s'_2, b) - \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s_t, a_t)] \end{aligned} \tag{2.45}$$

Here, we take the gradient with respect to the parameters $\mathbf{w}$ of the target estimate $[r + \gamma \max_b Q_{\mathbf{w}}(s'_1, b)]$ as well as at the current prediction of the state-action value $Q_{\mathbf{w}}(s_t, a_t)$. Note that here, two independent samples of the next state, s'_1 and s'_2, have to be used to ensure an unbiased estimate of the product of two random factors. Generating two independent samples of the next state requires either a model of the MDP or storing a set of state-successor pairs (s, s'), observed during learning, and sampling from this set.

The algorithm described above, with the update rule (2.45), is the *residual gradient* variant of Q-learning. Residual gradient variants were also proposed in [Baird, 1995] for other methods, including TD(0) and Value Iteration. Each of these residual gradient algorithms is guaranteed to converge to a local minimum of the Mean Squared Bellman Error. However, terms like $\left[\gamma \nabla_{\mathbf{w}} \max_b Q_{\mathbf{w}}(s'_2, b) - \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s_t, a_t)\right]$ in Equation (2.45) can be very small, especially with γ values close to 1, in which case learning can be very slow. In [Baird, 1995], another approach was proposed

which combines a residual gradient update and the "traditional" LMS update, so that convergence properties are preserved and some speedup is achieved.

Grow Support and Related Methods

Another way to prevent propagating approximation errors through the bootstrapping LMS updates was proposed in [Boyan and Moore, 1995] and [Boyan and Moore, 1996]. The two algorithms, known as Grow Support and ROUT respectively, provide a safe (converging) approach to value-function approximation in reinforcement learning. However, they are intended for episodic tasks only and for off-line training scenarios, where a model of the MDP is available or when transitions from any state can be generated at will by a simulator. These algorithms alternate between the following two phases.

During the exploration phase, trajectories are generated in an attempt to find fragments of the optimal policy. The algorithm constructs a set of states, called a *Support Set*, $\bar{S}$, for which the approximate optimal values $V_{\mathbf{w}}(s)$ are known with high accuracy. This process starts from terminal states and then the Support Set is extended "backwards" to other states. In other words, a new state s can only be added to the Support Set $\bar{S}$ if its optimal value can be reliably estimated from an accurate approximation of the optimal values of its successor states.

During the function approximation phase, a function approximator is trained using the LMS rule in batch mode on the training samples represented by the Support Set, $\langle s, V_{\mathbf{w}}(s)\rangle$, $s \in \bar{S}$. Then the Support Set is augmented again in an iterative manner.

The BFBP algorithm in [Wang and Dietterich, 2001] is a modification to the GrowSupport and ROUT methods. It is a provably convergent algorithm, intended for deterministic MDPs only and is to be applied in an off-line manner. It modifies the Grow Support and ROUT algorithms by applying a different exploration process to search for new states to be included into the Support Set and by including additional error terms into the the overall objective function for value function approximation.

Another approach, which has the same flavor as the Grow Support algorithm, was proposed in [Atkeson and Morimoto, 2002]. It builds a non-parametric representation of the value function based on points in the state-action space that lie close to the optimal trajectories (which is conceptually similar to the Support Set). The Grow Support method is conservative, as it only allows points to be included in the training set when an accurate estimation of the optimal value function is available for them. The method in [Atkeson and Morimoto, 2002] starts with points lying on some initially chosen trajectories, then several techniques are applied to improve these trajectories (make them closer to optimal). When improved trajectories are found, old ones are discarded. In other words, the old non-parametric value function representation is abandoned, and a new improved one is adopted. Unlike the Grow Support and BFBP algorithms, this approach is applicable to continuing stochastic tasks. Like BFBP, it is appropriate mostly for off-line learning with access to either an MDP model or a generative simulator model.

TD and Q-learning with Soft State Aggregation

Convergence of TD(0) and Q-learning was proven for value-function approximation based on soft state aggregation [Singh *et al.*, 1995]. In this case, the state space is divided into a number of *soft clusters* so that each state s belongs to a cluster x with some probability $P(x|s)$. The value function is defined only at the level of clusters and the value of each cluster generalizes to all states in proportion to the probabilities $P(x|s)$. Convergence was shown for the case, in which TD(0) or Q-learning are used to update the values of the clusters. The error bound on the resulting value function depends on the particular clustering of the state space. Good clustering is thus essential for achieving good performance with this method. The fact that the clusters are soft allows one to define a parameterization of the cluster probabilities and potentially learn a good clustering using gradient descent. Despite its good theoretical properties, this algorithm is not widely used in practice.

LSTD-based methods

Approximate TD learning performs updates of the approximator's weights iteratively, proportionally to some learning step. An alternative approach, the Least Squares TD (LSTD) method, was introduced in [Bradtke and Barto, 1996] and extended for use with eligibility traces in [Boyan, 1999]. This method eliminates the use of a step size and instead accumulates the information contained in the training samples in a matrix and a vector, which can be viewed as an approximation model. Then an appropriate setting of the approximator's parameters is computed as a solution of a system of linear equations by matrix inversion. Thus, no divergence is possible in this case, no matter what the distribution of the training samples is[10]. The LSTD method has also been used in the policy evaluation step of Policy Iteration in [Lagoudakis and Parr, 2003b]. More discussion of this method will be provided in Chapter 5.

Approximate Value iteration

The Value Iteration method generally assumes the availability of a model of the MDP, as given by (2.5)-(2.6). If such a model is available or can be learned, the Value Iteration algorithm can be applied in combination with value-function approximation. In this case, a subset of prototype states $\bar{S} = \{s^1, ..., s^M\}$ is chosen, and their values are arbitrarily initialized to some values $V_0(s^m)$, $m = 1, ..., M$. On every iteration t, a training set, consisting of samples $\langle s^m, V_t(s^m) \rangle$, $m = 1, ..., M$, is presented to a function approximator, which learns by an appropriate training method to generalize the values of the states in $\bar{S}$ to the entire state space. Then, one iteration of the Value Iteration method is performed synchronously for the states in $\bar{S}$ only, and the next estimate of the value function, V_{t+1}, is obtained as follows:

$$V_{t+1}(s^m) := \max_a \sum_{s'} P^a_{s^m,s'} \left[R^a_{s^m,s'} + \gamma V_t(s') \right] \ , \ m = 1, ..., M \tag{2.46}$$

[10]Special care has to be taken to ensure that the formed matrix is not singular.

Note that the successor states s' may be outside of the set $\bar{S}$ of prototype states, in which case the current approximation is used to estimate their values $V_t(s')$. This method is practical only if the set of possible successors is reasonably small so that the summation in Equation (2.46) can be efficiently computed.

In the tabular case, the convergence of the Value Iteration algorithm can be established based on the contraction property of its update operator. However, not all function approximation architectures preserve this property when the method is used as described before. Convergence of the approximate Value Iteration and the corresponding error bounds in the weighted maximum norm were established for function approximators known as *averagers* in [Gordon, 1995] and for linear function approximators satisfying certain conditions in [Tsitsiklis and Van Roy, 1996]. We will discuss these methods in detail in Chapter 5. Recently, error bounds for the approximate Value Iteration algorithm, applied to a restricted class of MDPs, were analyzed in [Munos, 2004] in terms of weighted L_1 and L_2 norms[11], which are usually used by function approximation training methods.

Q-learning with Averager Updates

Q-learning can be combined with interpolative function approximators while using an update mechanism slightly different from LMS training [Szepesvàri, 2001; Reynolds, 2002; Szepesvári and Smart, 2004], in which case, a convergent algorithm can be obtained. We discuss this approach in detail in Chapter 5.

Direct Policy Learning

So far we have discussed methods that approximate the optimal value function and define the optimal policy as a greedy policy with respect to this approximation. An alternative is to learn an approximate parameterized representation of a stochastic policy by using the gradient of the expected return with respect to the policy parameters. This method is, in fact, much easier to analyze theoretically, since

[11]Let $\mathbf{V} : S \rightarrow \Re$, where S is a finite state space such that $|S| = n$. Let $\mu(s)$ be a distribution on S. Then, $\|V\|_{L_1} = \sum_{s \in S} \mu(s)|V(s)|$ and $\|V\|_{L_2} = \left(\sum_{s \in S} \mu(s)|V(s)|^2\right)^{\frac{1}{2}}$. The result in [Munos, 2004] extends to infinite state spaces as well.

small incremental changes of the policy architecture's parameters can cause only small changes in the policy itself and in the state visitation probabilities. This idea serves as a basis for two recently proposed algorithms, the Policy Gradient Method [Sutton *et al.*, 2000] and the Value and Policy Search algorithm [Baird and Moore, 1998], which are provably convergent. In practice, however, methods based on the policy gradient search tend to be much slower than pure value-based algorithms.

Current theoretical results for RL with FA

Prediction Problem

RL method	Linear Approximations	Non-linear Approximations
Monte Carlo Sampling	Converges to the minimum solution of the MSE cost function.	Converges to a local optimum.
TD(λ)	Converges to a near-minimum solution of the MSE cost function.	Possibility of divergence demonstrated for TD(0).
Bellman error methods	Converges to a local optimum.	

Control Problem

RL method	Various Approximations
Value Iteration	Divergence is possible. Converges with averagers and linear architectures under certain conditions.
Policy Iteration	Convergence with linear architectures if policy improvement step generates a policy that is smooth in state-action values.
Optimistic Policy Iteration	Oscillation: chattering phenomenon. Non-divergence of Sarsa(λ) proved with linear approximators.
Q-learning	Divergence is possible with LMS training. Converges with averagers and soft state aggregation. Converges with interpolative approximators and with averager update rule.
Bellman error methods	Converges to a local optimum of the approximator's parameters. Chattering may occur.
Grow-Support Method	Stable learning. Terminates with a solution that is optimal for the states in the support set and near-optimal for the original state space.
Policy Gradient Method	Converges to a locally optimal policy for average return per stage problems.
Value and Policy Search	Converges to a locally optimal policy; value approximation errors are weighted by the state visitation probabilities. Result applies to episodic tasks and discounted problems with some conditions.

2.5. Complexity Analysis in Reinforcement Learning

Most theoretical studies in the reinforcement learning literature focus on convergence issues, whereas fewer results exist on the complexity of reinforcement learning and convergence rates.

One of the results providing polynomial time exact solutions for MDPs relies on formulating an MDP as a linear program (see e.g., [Bertsekas and Tsitsiklis, 1996]) and solving it by linear programming methods [Chvatal, 1983] in time polynomial in the number of states n, the number of actions m, and the size of the representation of the state transition probability matrix and the reward function[12] [Puterman, 1994; Bertsekas, 1995]. However, in the past, this approach to solving MDPs has been considered to be inefficient in practice [Littman *et al.*, 1995]. Only recently, the results in [de Farias, 2002; Guestrin, 2003; Hauskrecht and Kveton, 2004; Guestrin *et al.*, 2004] provided evidence that linear programming approaches can be practical for solving large MDPs.

The sample and computational complexity of "traditional" MDP-specific algorithms has been studied to some extent [Papadimitriou and Tsitsiklis, 1987; Littman *et al.*, 1995; Fiechter, 1997; Szepesvári, 1997; Kearns and Singh, 1998; 1999; Papadimitriou and Tsitsiklis, 1999; Kearns and Singh, 2000; Blondel and Tsitsiklis, 2000; Brafman and Tennenhotlz, 2002; Kakade, 2003]. For example, the Value Iteration algorithm was shown to solve exactly continuing discounted finite state MDPs in time polynomial in n, m, the maximum number of bits necessary to represent any component of the state transition probability matrix and the reward function, and the factor $\frac{1}{1-\gamma}$ (where γ is the discount factor) [Littman *et al.*, 1995]. On the other hand, no bounds exist on the number of iterations required by the Policy Iteration algorithm on general MDPs. Most approaches analyze tabular methods and guarantee approximate solutions in polynomial time. However, they either assume a constrained sampling method or additional knowledge about the MDP (e.g., accurate bounds on

[12]Note, that the actual time needed to find an exact solution, in general, depends on the numerical values of the input data. No linear programming algorithm is known to run in time polynomial in n and m.

the oprimal values [Kearns and Singh, 1998]), which typically is not available in practice. There are still few results for non-restrictive on-line sampling scenarios [Kakade, 2003].

The convergence rate of Q-learning has been analyzed by Szepesvari (1997) and Even-Dar and Mansour (2003). In both cases, the bounds depend on the particular schedule by which the learning rate is decreased, as well as on certain characteristics of the underlying MDP. For instance, Even-Dar and Mansour (2003) introduce the notion of covering time, which is the number of time steps necessary to try out all state-action pairs, regardless of the start state. They show that if the learning rate is decreased according to a linear schedule, the convergence rate depends exponentially on $\frac{1}{1-\gamma}$. If, on the other hand, the schedule is polynomial, the dependence on $\frac{1}{1-\gamma}$ is also polynomial. Szepesvari establishes the convergence rate assuming that the actions are sampled according to a fixed policy (independent of the action value estimates). His bounds depend on the ratio between the minimum and maximum state-action occupation frequencies. If it is easy to visit all state-action pairs, then obviously fewer time steps would be necessary to achieve a prescribed degree of accuracy.

For the case of continuous state spaces, computational time complexity analysis was performed for MDPs that satisfy certain Lipschitz continuity assumptions (i.e., with sufficiently smooth transition probabilities and reward functions). For example, it was shown in [Chow and Tsitsiklis, 1989; 1991] that when an MDP is discretized with a multigrid algorithm, $O(\frac{1}{\eta^{2n+m}})$ arithmetic operations are necessary and sufficient in order to approximate the optimal state-value function uniformly with accuracy η on a state and action spaces contained in the hypercubes $[0,1]^n$ and $[0,1]^m$ respectively, where η is the step of the discretization grid. Rust (1997) extended this approach by using a randomized variant of the Value Iteration algorithm. This randomized algorithm has been shown to require only polynomial time, where the complexity estimate depends on the Lipschitz constants of the MDP transition and reward functions. Unfortunately, as indicated in [Blondel and Tsitsiklis, 2000], this

constant tends to increase exponentially for practical problems of increasing dimension. The approach of Rust (1997) was further analyzed in [Szepesvàri, 2001].

CHAPTER 3

Characteristics of Markov Decision Processes

Chapter Outline

In this chapter, we present five quantitative attributes, which can be used to measure certain properties of Markov Decision Processes that affect performance of on-line value-based reinforcement learning algorithms. The considered properties are related to the amount of stochasticity in the MDPs, the required amount and ease of exploration, the amount of control that the agent has over its environment as well as risk tolerance constraints. We formulate hypotheses regarding the effects that the MDP properties, captured by the proposed attributes, have on learning and present results of an empirical study aimed at verifying these hypotheses. We discuss how the MDP attributes can be used in practice and how they can be computed.

3.1. Motivation

Reinforcement learning has already proved to be quite successful in solving interesting control and sequential decision-making problems and in handling large, realistic domains e.g., [Crites and Barto, 1996; Tesauro, 1994; Samuel, 1967; Singh and Bertsekas, 1997; Ng *et al.*, 2004; Stone and Sutton, 2001] (see Chapter 1 for more examples). However, even though reinforcement learning algorithms have a solid theoretical foundation, as discussed in Chapter 2, available theoretical results do not

always provide an adequate guidance for practical applications of reinforcement learning methods. For instance, even for tabular algorithms, in most cases, convergence is guaranteed only *in the limit*. As we discussed in Chapter 2, there are few results in the existing literature providing polynomial-time solutions for MDPs (either exact or approximate), and most of them rely on assumptions that can be restrictive for practical applications. In general, it is difficult to predict the quality of the policies obtained by various popular reinforcement learning methods after a limited amount of *on-line* training.

Most efforts for analyzing reinforcement learning techniques assume that the problem to be solved is a general stochastic Markov Decision Process, while very little research has been devoted to defining or studying sub-classes of MDPs. This is in contrast with the prior research in other related disciplines. For example, in combinatorial optimization [Papadimitriou and Steiglitz, 1982; Hogg *et al.*, 1996], it has been shown that the performance of approximate optimization algorithms can be drastically affected by characteristics of the problem at hand. The performance of local search algorithms is affected by characteristics of the search space for a given problem instance, such as the number of local optima, the sizes of the regions of attraction, and the diameter of the search space[1]. Recent research (e.g., [Hoos and Stutzle, 2000; Lagoudakis and Littman, 2000]) has shown that such problem characteristics can be used to predict the behavior of local search algorithms, and improve algorithm selection.

In supervised learning (mainly in classification), there is also a growing research area, known as Meta-Learning [Köpf *et al.*, 2000; Linder and Studer, 1999; Peng *et al.*, 2002], devoted to identifying the characteristics of datasets that can be used in order to select the most appropriate learning algorithms for the data at hand. The considered attributes range from those characterizing the dataset size, the number of attributes, the number of missing values to statistical and information-theoretical measures, such as the entropy of classes, mutual information between data attributes

[1]The diameter of the search space is defined as the maximal distance between any two points in a given instance of the search space.

and classes, noise-to-signal ratio, etc. (see, e.g., [Henery, 1994; Peng *et al.*, 2002]). Based on the performance profiles of different classification algorithms, regression models are built which predict the performance of these algorithms on new data sets based on dataset characteristics[2].

In the MDP literature, prior theoretical and empirical results also suggest that certain MDP characteristics can influence the complexity of finding an optimal policy. For example, it has been shown [Papadimitriou and Tsitsiklis, 1987] that deterministic MDPs can be solved efficiently in parallel with a number of arithmetic operations that depends only on the number of states n and the number of actions m (i.e., the problem is in the complexity class NC)[3]. However, under the general conditions of stochastic state transitions, the problem is P-complete[4]. Also, the work in [Papadimitriou and Tsitsiklis, 1999] analyzed a problem of routing and scheduling in closed queuing networks. It was shown that, in a general setting, the problem is EXP-complete (has a provably exponential complexity). However deterministic instances of this problem are PSPACE-complete. This suggests that stochasticity is one of the major factors responsible for the complexity of solving MDPs. Previous empirical studies [Dean *et al.*, 1995; Kirman, 1995] uncovered the existence of various MDP properties affecting the performance of dynamic programming methods under time constraints. Stochasticity of state transitions was conjectured and empirically shown to be amongst such influential factors.

One of the goals of this thesis is to investigate a similar conjecture but in the context of on-line value-based reinforcement learning and in a more general context than previously addressed in the literature. Our work builds on the doctoral dissertation of Kirman (1995). He studied the performance of dynamic programming

[2]METAL: Data Mining Advisor, http://www.metal-kdd.org/

[3]In complexity theory, the class NC ("Nick's Class") is the set of problems decidable in polylogarithmic time on a *parallel* computer with a polynomial number of processors. In other words, a problem is in NC if there are constants c and k such that it can be solved in time $O((\log n)^c)$ using $O(n^k)$ parallel processors.

[4]The class P consists of all those problems that can be solved on a deterministic *sequential* machine in an amount of time that is polynomial in the size of the input. If an efficient (logarithmic time) *parallel* algorithm were available, then all problems in P would be solvable efficiently in parallel, which is considered unlikely by the researchers in complexity theory.

algorithms (planners) on three classes of discrete-state MDPs. Kirman proposed a number of domain-specific parameters as well as some domain-independent attributes to measure certain properties of the MDPs. His goal was to build statistical models, based on the proposed MDP characteristics, for direct *quantitative* prediction of the performance of specific dynamic programming algorithms on instances of the studied classes of MDPs. He found some of the proposed domain-independent attributes to be statistically significant predictors of performance. However, the obtained quantitative statistical models did not appear to directly transfer to other types of domains.

In our work, we do not build numerical predictive models of performance. We start with analyzing value-based on-line reinforcement learning algorithms, such as Sarsa and Q-learning [Sutton and Barto, 1998], both tabular and approximate. In the context of such algorithms, we investigate the effect of certain attributes proposed by Kirman (1995) as well as suggest new attributes. We formulate hypotheses about the effect of these attributes on learning which can be verified empirically. We validate experimentally our hypotheses concerning two of the proposed attributes. In Chapter 4, we present an approach to using these two attributes in a domain-independent context for improving the exploration efficiency of on-line reinforcement learning. We discuss other potential applications of the proposed attributes in this chapter.

The rest of this chapter is organized as follows. In Section 3.2, we discuss certain properties of MDPs related to the performance of on-line value-based algorithms and motivate the use of the proposed attributes. In Section 3.3, we present definitions of the MDP attributes, explain which MDP properties they capture and how they relate to the algorithms' performance. We also give our suggestions as to the potential practical usage of these attributes. In Section 3.4, we discuss how to compute the values of the proposed attributes. In Section 3.5, we present the results of an empirical study investigating the effects of two of the proposed attributes on tabular and approximate reinforcement learning methods. We end with a summary of this chapter in Section 3.6.

3.2. Some Challenges in On-Line Reinforcement Learning

3.2.1. Mean Estimation and Sample Variability

Recall from Chapter 2 that value-based reinforcement learning methods incrementally estimate *expected* values of the (discounted) returns, i.e., value functions (see e.g., the Value Iteration algorithm in Equation (2.14)). Model-free algorithms, such as Sarsa and Q-learning, estimate these expected values using samples as opposed to using MDP models (transition probabilities and expectations of immediate rewards). Hence, in certain aspects, such algorithms can be related to incremental estimation of the mean of a random variable.

Consider a general incremental mean estimator:

$$\bar{X}_k = (1 - \alpha_k)\bar{X}_{k-1} + \alpha_k x_k \text{ , k=1,2,3,...} \tag{3.1}$$

where $\bar{X}_k$ is the estimated sample mean of a random variable X after receiving a new sample x_k of this random variable. The parameters $\alpha_k \in [0, 1]$ are the learning steps that control the rate of convergence of the estimated mean to the true expectation of the random variable X:

$$\lim_{k\to\infty} \bar{X}_k = E\{X\} \tag{3.2}$$

The standard Robbins-Monro conditions [Robbins and Monro, 1951] on the choice of the learning steps α_k ensure that such convergence is guaranteed:

$$\begin{aligned} &(1) \quad \textstyle\sum_{k=1}^{\infty} \alpha_k = \infty \\ &(2) \quad \textstyle\sum_{k=1}^{\infty} \alpha_k^2 < \infty \end{aligned} \tag{3.3}$$

The first condition ensures that, if the current estimate is biased in any way, the bias will be eliminated in the limit, since the sum of the future learning steps is infinite (and thus the contribution of the future samples is always non-zero). The second condition is concerned with overcoming the variance of samples by reducing the learning steps eventually to a sufficiently small value. These conditions are required for convergence of many stochastic approximation algorithms (recall, for example,

Temporal Difference learning and the incremental gradient descent method discussed in Chapter 2).

In practice, with a limited number of samples available and a finite numerical precision, it is not always easy to find an optimal decreasing schedule for the learning steps α_k, satisfying conditions (3.3), so that the trade-off between lowering the bias and overcomming the variance is resolved. Intuitively, the higher the variance of the considered random variable, the more samples are necessary to increase the confidence in the accuracy of the sample mean estimate and the smaller the values of the learning steps should be. However, if the learning steps are reduced too quickly, the estimation process may converge to a biased value (different from the true expectation $E\{X\}$).

In practical applications, a *recency weighted average estimator* is often used, where the learning step α is constant for the entire estimation time:

$$\bar{X}_k = (1 - \alpha)\bar{X}_{k-1} + \alpha x_k \tag{3.4}$$

This approach is well suited for the case, where the distribution of the considered random variable is non-stationary. A constant setting of α allows to continue tracking changes in the sample distribution without premature convergence. Unfortunately, constant settings are particularly sensitive to the variance of samples. Ideally, we would like to use a decreasing schedule for α_k withing each of the stationary periods and then reset to a new schedule (starting with a larger setting of α_k) when the distribution of the random variable changes significantly. However, in practice, it is not always possible to determine when such changes occur.

Value Non-Stationarity

Recall from Chapter 2 that in the case of value-based reinforcement learning algorithms, we are, in fact, dealing with samples that come from non-stationary distributions. Consider learning optimal state-action values, $Q(s, a)$, which are the means of the random variables $R(s, a)$ (i.e., returns). In the case of on-line learning, these values are usually estimated from samples of one-step-truncated discounted

returns, i.e., bootstrapped estimates (see Equation (2.18)). Abusing the notation, we will call them *state-action value samples.* These samples are obtained by observing the state transitions and immediate rewards and by using bootstrapping to estimate the values of the successor state-action pairs. For example, recall that at each time step t, the Sarsa algorithm forms and learns a new sample of the value for the current state-action pair (s_t, a_t) based on the observed reward r_{t+1} and the next state-action pair (s_{t+1}, a_{t+1}) as follows:

$$\underbrace{Q(s_t, a_t)}_{\text{Estimated mean}} := (1-\alpha)Q(s_t, a_t) + \alpha\underbrace{[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})]}_{\text{Sample of } Q(s_t, a_t)} \tag{3.5}$$

where the action a_{t+1} is chosen from a probability distribution defined by the agent's current estimation policy. Consider time steps t_1 and t_2, where two subsequent samples for the state-action value $Q(s, a)$ are obtained, such that $(s_{t_1}, a_{t_1}) = (s_{t_2}, a_{t_2}) = (s, a)$. These samples are based on the bootstrapped estimates of the successor state-action values, $Q(s_{t_1+1}, a_{t_1+1})$ and $Q(s_{t_2+1}, a_{t_2+1})$. These estimates may, in general, be different even if the same successor state-action pair is considered at both times, i.e., $(s_{t_1+1}, a_{t_1+1}) = (s_{t_2+1}, a_{t_2+1}) = (s', a')$. This is because the action-value estimates are updated whenever the corresponding state-action pairs are experienced by the agent. Hence, the value of $Q(s', a')$ could have changed between the times t_1 and t_2. This kind of non-stationarity can produce sudden and significant changes in the observed samples. This is why small values of the learning step parameter are usually necessary to avoid instability in the estimation process.

Sample distribution non-stationarity is also caused by changes in the estimation policy. Consider, for example, an off-policy learning algorithm, such as Q-learning. A sample of the state-action value $Q(s_{t_1}, a_{t_1})$ at time t_1 is formed as $[r_{t_1+1} + \gamma \max_{a \in A} Q(s_{t_1+1}, a)]$. As the estimated state-action values are updated during learning, some time after t_1, the agent can learn that a different action is better in the state s_{t_1+1}. Thus, the action that maximizes the value in the successor state s_{t_1+1} can change before the time step t_2 when the value of the state-action pair

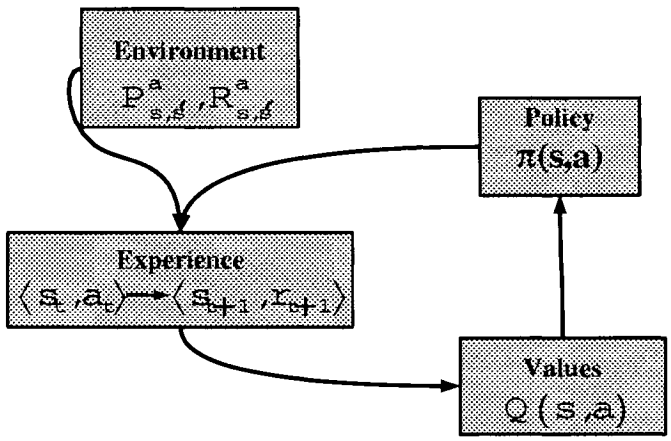


FIGURE 3.1. Value-based on-line learning.

$(s, a) = (s_{t_1}, a_{t_1}) = (s_{t_2}, a_{t_2})$ is updated again. Thus, the value of $\max_{a \in A} Q(s_{t_1+1}, a)$ can be different at time t_2 as compared to time t_1. The situation is the same with the on-policy Sarsa method, which uses policies that are nearly greedy, e.g., the ϵ-greedy policy.

Thus, an on-line tabular reinforcement learning algorithm can be viewed as a general estimator of the mean of a random variable that operates on top of the process that supplies value samples with non-stationary distribution. Of course, the overall process is much more sophisticated, since there is a circular dependency between the mean estimators and the sampling mechanism through bootstrapping (see Figure 3.1). As discussed in Chapter 2, convergence of tabular reinforcement learning algorithms can be analyzed using stochastic approximation theory based on the the contraction property of the iterative updates derived from the Bellman equations (see e.g., [Jaakkola *et al.*, 1994] for a proof of convergence of the Q-learning algorithm and [Singh *et al.*, 2000] for convergence of Sarsa(0)). However, looking at a (crude) interpretation of the value function estimation as the general mean estimation helps to isolate certain important issues that are not explicitly addressed in the existing theoretical analysis. In particular, we will now focus on the relationship between the environment, the value samples and the estimators (see Figure 3.1) and investigate how various MDP properties can affect the sampling and estimation process.

As already mentioned, convergence of a general mean estimator, e.g., as in Equation (3.1), is affected by the variance of obtained samples, even in the case of a stationary sample distribution. Let us examine samples obtained in the course of on-line value-based reinforcement learning and whether they are associated with any variance.

Stochasticity of the Environment

Consider the Sarsa update in Equation (3.5) and let us identify some factors that can create variability in the corresponding state-action value samples. First of all, the stochasticity of the environment can be responsible for the variance. Suppose that the same state-action pair (s, a) is encountered at time steps t_1 and t_2 during learning, and the corresponding value samples are obtained as follows:

$$\begin{aligned} &\text{Sample on time step } t_1: \quad [r_{t_1+1} + \gamma Q(s_{t_1+1}, a_{t_1+1})] \\ &\text{Sample on time step } t_2: \quad [r_{t_2+1} + \gamma Q(s_{t_2+1}, a_{t_2+1})] \end{aligned} \tag{3.6}$$

Table 3.1 summarizes three factors that can contribute to the variance of these samples. The stochasticity of the state transitions and rewards is responsible for the two components of the sample variance: the variance in immediate rewards and the variance in long-term rewards, estimated by the values of the successor state-action pairs. More specifically, if the environment provides stochastic rewards, the reward component of the state-action value sample will contribute to the sample variance (Factor 1, Table 3.1) . If state transitions are stochastic, both the rewards and the values of the successor states associated with transitions to different states can vary at times t_1 and t_2 (factors 2 and 3 in Table 3.1) and hence, introduce variance into the value samples. In Section 3.3, we will present two attributes, the State Transition Entropy and the Variance of Immediate Rewards, which quantify the amount of stochasticity in the environment that contributes to these factors. Note that the magnitude of the variance in long-term rewards depends in the end on the actual shape of the action-value function, that is on the differences in the state-action values of the successor states s_{t_1+1} and s_{t_2+1}.

TABLE 3.1. Factors contributing to the variance of state-action value samples due to the stochasticity of the environment.

Factor 1:	Stochastic immediate rewards $\Longrightarrow$	$r_{t_1+1} \neq r_{t_2+1}$
Factor 2:	Stochastic transitions: $s_{t_1+1} \neq s_{t_2+1} \Longrightarrow$	$r_{t_1+1} \neq r_{t_2+1}$
Factor 3:	Stochastic transitions: $s_{t_1+1} \neq s_{t_2+1} \Longrightarrow$	$Q(s_{t_1+1}, a_{t_1+1}) \neq Q(s_{t_2+1}, a_{t_2+1})$

Variance Due to Exploration

When an on-policy algorithm is used, such as Sarsa, samples are formed as in Equation (3.6), where the action a_{t_1+1}, selected in the successor state s_{t_1+1}, is chosen according to the current behavior policy. The behavior policy must ensure sufficient exploration and thus must be stochastic, such that it assigns a non-zero probability of selecting every action in every state. Because of this, the choice of a_{t_1+1} and a_{t_2+1} is randomized and can be different at times t_1 and t_2 even if the successor state $s_{t_2+1} = s_{t_1+1}$ and its state-action values have not changed between t_1 and t_2. Thus, the values of the successor state-action pairs can be different due to different actions chosen at times t_1 and t_2 for exploration purposes:

$$a_{t_1+1} \neq a_{t_2+1} \Longrightarrow Q(s_{t_1+1}, a_{t_1+1}) \neq Q(s_{t_2+1}, a_{t_2+1}) \text{ , even if } s_{t_1+1} = s_{t_2+1} \qquad (3.7)$$

Two of the attributes, the Controllability and the Reward Information Content, presented in Section 3.3, will allow us to assess whether significant differences in the values of different actions could be expected.

Function Approximation: Bias and Variance

In the case of using function approximation for representing value functions, the general mean estimators, discussed before, are replaced by a learning algorithm associated with the function approximator employed (see Chapter 2). Different function approximators have different tolerance to the variance (noise) in training samples, but, in general, learning and tuning the learning step parameters is more difficult in the presence of a significant variance.

On the other hand, the function approximator can itself affect the variance of value samples during on-line reinforcement learning. Let us illustrate this point in

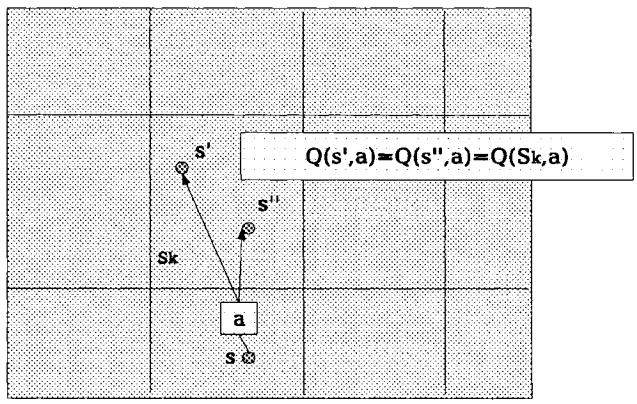


FIGURE 3.2. State aggregation: bias and variance

the case of a local function approximator, for instance, state aggregation. In this case, the state space is partitioned into a number of non-overlapping partitions (see Figure 3.2). States s' and s'' that belong to the same partition, say S_k, share common state-action values, $Q(s', a) = Q(s'', a) = Q(S_k, a)$. Partitioning or grouping states together in such a manner introduces a *bias* into the action-value function, so that its shape is in part restricted by the chosen grouping of states. The situation is similar with any kind of function approximation that generalizes the values across different states in some manner[5]. Bias introduced by the function approximator can influence the variance of value samples in several ways.

As before, consider two samples of $Q(s, a)$ at different times during learning, t_1 and t_2. Now assume that the successor state-action pairs are identical at both times: $(s_{t_1+1}, a_{t_1+1}) = (s_{t_2+1}, a_{t_2+1}) = (s', a')$. Let a state s'' belong to the same partition S_k as the state s'. If the state-action pair (s'', a') has been encountered between t_1 and t_2, an update of the aggregate value $Q(S_k, a')$ of the states s' and s'' has been performed. Hence, the value of $Q(s', a')$ might have significantly changed during the period of time between steps t_1 and t_2 because of the total accumulated updates

[5]This is sometimes called the *inductive* or *representational* bias [Mitchel, 1997]. A different type of bias that is often considered in supervised learning is the *statistical bias* (see e.g., [Geman *et al.*, 1992]), which is the difference between the expected value of an estimator (taken over all possible datasets) and the correct values.

caused by *other* states in the same aggregate. In this case, the resulting difference between the estimates $Q(s', a')$ at times t_1 and t_2 can be significant. Thus, the bias of the function approximator can increase the variance of value samples associated with the non-stationarity of the value estimates.

At the same time, the function approximation bias can decrease the variance associated with the stochasticity of state transitions discussed earlier. This will be the case if states s' and s'', which are both possible successors of the state-action pair (s, a), belong to the same aggregate and thus share the same values (see Figure 3.2). In this case, no variance of the samples for $Q(s, a)$ will be introduced due to Factor 3 in Table 3.1, that is, due to differences in the values of successor states s' and s''.

Finally, the bias introduced by the choice of the structure of the function approximator can increase the variance of value samples because of the following reason. If the states s' and s'' that belong to the same partition are very different in terms of their true values, such states can provide very different (high variance) samples for their common aggregate values $Q(S_k, \cdot)$. This makes the estimation processes more difficult and complicates convergence of the estimation process. Grouping together the states with different values can also prevent learning the action-value function with an accuracy sufficient for finding a good policy. Several studies can be found in the literature that attempt to assess the differences between the states with respect to their values in order to find an appropriate state aggregation, see e.g., [Givan *et al.*, 2003; Ferns *et al.*, 2004; Munos and Moore, 2001; Reynolds, 2002].

Discussion

As already mentioned, the magnitude of the variance of value samples depends on immediate rewards and the actual shape of the value function. It should be noted, however, that in stochastic environments or when using function approximation, it is unlikely *not* to encounter any significant variance throughout the training process because of the following fact. During on-line learning, the agent performs a form of Generalized Policy Iteration [Sutton and Barto, 1998], that is, it partially estimates

the values of many different intermediate policies before it reaches the optimal behavior. It is quite likely that at least under some of those intermediate policies, there will be a significant variance of value samples due to the factors discussed above. This variance can render learning more difficult, lead to a higher sample complexity and hence result in a longer convergence time. In general, it is hard to avoid or reduce this variance, but the ability to assess this variance can be useful for (dynamic) tuning of various parameters of learning algorithms (e.g., the learning step and the exploration rate) in order to mitigate the effect of the high variance and improve performance. Several attributes that we discuss later in this chapter help to make such an assessment.

In the current literature, there are some approaches that attempted to measure and use the information about the stochasticity of the environment or the variance of value samples. The work in [Kirman, 1995; Dean *et al.*, 1995], to the best of our knowledge, was the only one that used measurements of the environment's stochasticity based on the MDP model only and independently of the estimated value functions. As we already mentioned, these measurements were used for building numerical (regression) models of the performance of specific dynamic programming algorithms on several classes of MDPs. We will point out the relation between the attributes presented in this chapter to the work of Kirman in the next section where we will present the definitions of the MDP attributes.

The methods in [Kaelbling, 1993; Meuleau and Bourgine, 1999] used the estimates of the variance of state-action value samples for improving exploration efficiency (see Section 4.2 for more details). The variance estimates were computed in parallel with the value functions during learning. Since during on-line learning the state-action value estimates constantly change, a form of a forgetting mechanism has to be employed to track these changes and to get an appropriate (up-to-date) estimate of the variance. Such a forgetting mechanism was implemented by maintaining a sliding window of a batch of previously observed samples for each state-action pair and calculating the sample variance based on these batches. Thus, this approach requires

a considerable amount of additional memory. In addition, it is not obvious what the size of the sliding windows should be to capture the non-stationarity properly.

Another (cheaper) heuristic for assessing and accounting for the value sample variance was used in [Meuleau and Bourgine, 1999]. It relied on an assumption about the maximum bound on the variance of value samples. Suppose that the initial state-action values (at the onset of learning) satisfy the following condition:

$$\frac{R_{min}}{1-\gamma} \leq Q(s,a) \leq \frac{R_{max}}{1-\gamma}, \text{ for all } (s,a) \tag{3.8}$$

where R_{max} and R_{min} are the maximum and minimum rewards over all (s,a,s') triples. Then, the variance of state-action values satisfies the following bound:

$$Var[Q(s,a)] \leq \left(\frac{R_{max}-R_{min}}{2(1-\gamma)}\right)^2, \text{ for all } (s,a), \text{ at all times} \tag{3.9}$$

This estimate assumes a uniform variance across all the state-action pairs.

An approach for exact computation of the variance of returns associated with a fixed policy π was developed in [Munos and Moore, 2001]. It was used to assess the variance (error) introduced by approximating a deterministic continuous state and time MDP with another stochastic discrete MDP. The variance of the return, $\sigma^2(s) = E\{[R(s)-V(s)]^2\}$, where $R(s)$ is the discounted return and $V(s)$ is the value function (expected value of the return) for policy π, was proved to be a solution to the following Bellman equation:

$$\begin{aligned} \sigma^2(s) &= \gamma^2 \textstyle\sum_{s' \in S} P_{ss'}^{\pi(s)} \sigma^2(s') + e(s) \ , \forall s \in S \\ e(s) &= \textstyle\sum_{s' \in S} P_{ss'}^{\pi(s)} \left[\gamma V(s') - V(s) + r_{ss'}^{\pi(s)}\right]^2 \end{aligned} \tag{3.10}$$

The proof of this fact from [Munos and Moore, 2001] is provided in Appendix A.1. Based on this Bellman equation, it is possible to compute the variance of returns using a dynamic programming algorithm, since Equation (3.10) is a fixed-point equation of a contractant operator (in max norm, with a contraction factor γ^2). The variance $\sigma^2(s)$ takes into account the variation in the values of the subsequent states (s and its successors s') as well as the discounted expected variance $\sigma^2(s')$ of these successors.

Thus, this approach provides a way to compute the variance of the return for a fixed policy accurately, however, such computation is as costly as computing the value function. In fact, the value function has to be already computed in order to use Equation (3.10). Such computation can be justified in the contexts similar to the one in [Munos and Moore, 2001], where it was used in order to assess the goodness of a value function approximation and iteratively improve it. An approximate value function and the variance of the return were computed using dynamic programming assuming the availability of the MDP model.

We will now mention several other challenging issues in on-line reinforcement learning, which are related to MDP properties captured by our attributes.

3.2.2. Exploration

As discussed in Chapter 2, the reinforcement learning agent needs to learn by actively exploring its environment. The difficulty of exploration can be associated with several factors. For example, the ease of movement around the state space will affect the speed, with which the agent can discover new, previously unvisited parts of the state space or revisit states again, and hence, learn their values. Also, the agent often faces the problem of delayed rewards, that is long delays in the resolution of the action consequences (e.g., MDPs with zero rewards everywhere except at a goal state). If the immediate rewards provide little information about the (relative) desirability of different actions, the speed of learning and the duration of the initial exploration phase will be affected.

Once again, knowledge of such properties of the MDP at hand can be helpful for making an informed choice of the learning and exploration algorithms and for taking advantage of the underlying MDP characteristics in order to facilitate the exploration process. The question of the ease of movement through the state space can be related to the notion of conductance studied in the context of Markov chains [Jerrum and Sinclair, 1988], which measures the rate at which the process can flow around the state space. Such a measure would be most useful in the case when a

fixed behavior policy (that defines a specific Markov chain) is applied for a long time. Hence it would be relevant for off-policy algorithms mainly and not so much for the on-policy algorithms where the behavior policy can typically change quickly as the value estimates get updated. Computing conductance associated with a particular Markov chain would require applying complex off-line methods [Jerrum and Sinclair, 1988].

In the next section, we discuss how several MDP attributes considered in this thesis, namely, the State Transition Entropy, the Controllability and the Reward Information Content, can provide some indication about the MDP properties discussed above at a low computational cost. In Chapter 4, we show how to use two of these attributes for the design of an efficient exploration strategy.

3.2.3. Amount of Control

The success of the agent's learning effort depends on the way the control system is designed and the way the interaction between the agent and the environment occurs. In particular, the performance of the agent depends on the amount of control that it can exercise over the environment: if the environment simply ignores the agent's actions so that the state transitions and rewards are independent of what the agent does, then, of course, the performance of the learning agent will not be different from the performance of an agent acting randomly. In the next section we will present two MDP attributes, the Controllability and the Reward Information Content, that measure the amount of the agent's influence on the environment's responses. In general, such characteristics may be non-uniform across the MDP's state space. The ability to assess such properties can help to evaluate the agent's limitations as well as prioritize the agent's learning effort on the states and actions that can have the largest impact on its performance. The controllability also has other implications on the variance of value samples as well as on the exploration process, as we will explain in detail after we will have defined the corresponding attributes in the next section.

3.2.4. Risk Management

Another issue that the agent, learning on-line, may need to take into consideration is related to risk tolerance during learning. Recently, some work, mostly in the domain of decision theory, has been done for addressing the question of risk management in MDPs. Below, we will briefly highlight some existing approaches to risk management and in Section 3.3, we will discuss another MDP attribute, the Risk Factor, that allows to reflect the agent's risk tolerance with respect to the properties of the MDP's reward function.

In reinforcement learning, the focus is on computing and acting according to the expected values of returns. Decision theory offers alternative decision-making criteria [Taha, 1987] arguing that expected values are not always appropriate in practice because of the following considerations. Criteria based on expected values assume that the decision process is repeated a sufficiently large number of times from the same state. In practice, however, it might not always be the case and acting in such a manner may not be best when possible consequences or their probabilities have extreme values. The agent may also need to adhere to certain constraints, safety issues and performance expectations in addition to best-on-average criteria. Examples include tasks, where the agent is required to get to a goal within a certain time with probability one, or where the agent has to minimize the probability of entering fatal states.

For such tasks, some approaches aim at minimizing α-value criterion. The agent using this criterion secures that with probability $1-\alpha$, its return will not be less than some threshold m_α, where the optimal policy maximizes m_α [Heger, 1994a]. A special case of this approach is when $\alpha = 0$, where the agent is optimizing the *worst-case* total discounted reward (also known as mini-max criterion). A counterpart of the Q-learning algorithm for this criterion is presented in [Heger, 1994b]. This approach was further developed and generalized in [Coraluppi and Markus, 1999].

Optimization of the worst-case criterion, however, is considered to be too conservative and pessimistic for certain tasks. For example, in [Gaskell, 2003], the experiments were conducted on a goal-directed task where the agent had to walk in a state space containing a deadly cliff. On this task, the minimax algorithm learns to jump from the cliff from the start to avoid a possibility of occasionally getting a slightly lower return later (associated with per-step delay penalties). It was also pointed in [Neuneier and Mihatsch, 1999] that minimax learner for an investment management problem learns to never make investments. The approach in [Gaskell, 2003] proposes a β-pessimistic learning, which is a compromise between standard (optimistic) Q-learning and extreme pessimistic minimax methods. In this case, the resulting policy chooses an action with the highest return with probability $1-\beta$ and an action with the lowest return with probability β. A Q-learning-type algorithm was proposed in order to learn the corresponding value functions and was experimentally demonstrated to produce relatively safe solutions.

A different approach was proposed in [Geibel, 2001], where the optimization problem is formulated as a *constrained* MDP with two criteria: a standard discounted return and a non-discounted criterion that reflects a probability that a trajectory from some state s under a policy π eventually ends up in a fatal state. A heuristic algorithm is designed to find policies that balance the two criteria by means of a weight parameter. This weight is optimized by slowly shifting the focus from risk avoidance to value optimization until the constraints on risk tolerance are violated.

3.2.5. Summary

In this section, we discussed some of the challenges of value-based on-line reinforcement learning algorithms, which can be summarized as follows:

(1) Dealing with the non-stationarity of the sampling distribution. Theoretically, this is the question of convergence of on-line reinforcement learning algorithms for control, see e.g., [Watkins and Dayan, 1992; Jaakkola *et al.*, 1994; Singh *et al.*, 1995; Bertsekas and Tsitsiklis, 1996; Singh *et al.*, 2000]. In practice, this issue relates to the

selection of learning and exploration parameters as well as function approximation representations for value functions in a manner that allows successful tracking of the non-stationary value samples provided by the underlying reinforcement learning algorithm.

(2) Dealing with the variance of value samples that is caused by the stochasticity of the environment and the exploration process. Once again, this issue is in practice related to the selection of learning parameters and improving efficiency of the sampling process (see e.g., [Kaelbling, 1993; Kirman, 1995; Meuleau and Bourgine, 1999] for existing work).

(3) Providing good exploration for efficiently uncovering new information while satisfying risk constraints (see Chapter 4 for a literature review on exploration methods as well as e.g., [Heger, 1994b; Geibel, 2001; Hauskrecht *et al.*, 2001; Gaskell, 2003] for some studies on risk management).

In the following section, we present five quantitative attributes that capture certain properties of MDPs related to the challenges (2) and (3). Our goal is to identify relevant MDP characteristics that can be assessed without relying on the state-action values, but only based on the information captured in the standard MDP model (the state transition probabilities and the immediate rewards). Our attributes are intended mainly for measuring MDP properties prior to learning or during learning but in a manner that is independent of the value function estimates. As we mentioned before and as pointed out in [Wyatt, 2001], it is easier to evaluate such properties during on-line learning, because they are *stationary*, whereas the heuristics based on the state-action values need to track the dynamically changing value estimates.

3.3. Quantitative Attributes to Measure MDP Properties

In this section, we present five domain-independent attributes that can be used to quantitatively describe certain characteristics of an MDP relevant to the performance of on-line value-based reinforcement learning algorithms. All attributes are defined based on the information contained in the MDP model only. For simplicity, we

first assume discrete and finite state and action spaces and an availability of the MDP model (state transition probabilities and the expected values of immediate rewards). Nevertheless, the definitions naturally extend to the continuous case and all the attributes can be approximately estimated from samples of state transitions and rewards obtained from the interaction with the environment, as we discuss in Section 3.4.

3.3.1. State Transition Entropy

The *State Transition Entropy* measures the amount of stochasticity due to the environment's state dynamics. Let $O_{s,a} \in S$ denote a random variable representing the outcome (next state) of the transition from state s when the agent performs action a. We use the standard information-theoretic definition of entropy (see, e.g., [MacKay, 2003]) to measure the State Transition Entropy for a state-action pair (s, a) (note, that the same definition was used in [Kirman, 1995]):

$$STE(s,a) = H(O_{s,a}) = -\sum_{s' \in S} P^a_{s,s'} \log P^a_{s,s'} \tag{3.11}$$

A high value of $STE(s, a)$ means that there are many possible next states s' (with $P^a_{s,s'} \neq 0$). The entropy is maximized for the uniform distribution. The agent is more likely to encounter many different states by performing the same action a in some state s if $STE(s, a)$ is high. This can have several implications on learning. First, in this case, the variance in the samples of the state-action value $Q(s, a)$ may be high, as discussed in the previous section and summarized in Table 3.1 (see Factors 2 and 3), which makes learning more difficult. Of course, the actual variance depends on the immediate rewards and the values of different successor states.

Second, in environments that have many states with high State Transition Entropy, state space exploration happens naturally to some extent, and state-space coverage is facilitated. Since extensive and fast exploration is essential for reinforcement learning algorithms, the presence of state-action pairs with high State Transition Entropy can result in faster learning. In some cases, however, exploration is not

improved in states with high State Transition Entropy. For instance, a robot may bounce against a wall and non-deterministically end-up in the nearby states. If, some of the time it bounces back to the same state, then state-space exploration is actually somewhat impeded. If desired, this case can be identified by considering the value of $(1 - P^a_{ss})$, the probability of making a transition to a state, different from the current one[6]. Kirman (1995) also suggested using the M-Step State Transition Entropy (entropy of state transitions after M time steps) as a measure of ease of movement through the state space. In this case, the rate of increase of this entropy with M can be considered.

We hypothesize that MDPs with high average values of the State Transition Entropy will exhibit a trade-off between a positive effect on state-space exploration and a negative effect of the increased sample variance during learning.

We envisage the following practical uses of the State Transition Entropy attribute. First, it can be used to improve the efficiency of exploration by giving a higher priority to actions with high State Transition Entropy values. Such a strategy can facilitate state-space exploration and gather more samples for value estimates with a potentially high variance of value samples (see Chapter 4 for an implementation).

Second, the State Transition Entropy can be used for tuning the learning step parameter. As discussed in the previous section, the state-action pairs with high variances of value samples (which can be indicated by high $STE(s, a)$ values) should use lower settings of the learning step. If an MDP has high values of this attribute throughout the state-action space, the learning steps in general should be set to small values, especially when using high values of the eligibility trace parameter λ (see Chapter 2). This is because using high values of λ by itself increases the variance of value samples, since the variance accumulates as the samples are back-propagated along long trajectories (see Chapter 2 as well as [Kearns and Singh, 2000] for the analysis of the bias-variance trade-off of the TD(λ) method).

[6]In this thesis, we do not further investigate the use of such an indicator.

Finally, when using local function approximators to represent value functions, the State Transition Entropy can be useful for choosing a structure of the approximator. For example, as pointed out in [Menache *et al.*, 2004], when Radial Basis Function Networks are trained so that the centers and widths of the basis functions are adjusted with the gradient descent learning (see Chapter 2), it is possible that the agent will learn a policy that avoids states, which often provide a high instantaneous (per sample) error. If the error is associated with a high sample variance (noise), it cannot be removed by learning a better value function approximation so the agent can simply learn to not visit such states even if they have good values. The State Transition Entropy attribute can help to determine whether the error is due to the stochasticity of the environment, and hence to avoid such undesirable learning outcomes. On the other hand, there are a number of heuristic methods (see e.g., [Anderson, 1993; Fritzke, 1997; Wiering, 1999; Millán *et al.*, 2002] as well as Chapter 5 for more details) that allocate units (basis functions) for local function approximators so that the regions of high approximation errors get more units. Allocating more units in the areas of the state space, which show errors due to the stochasticity of the environment as opposed to poor approximation, can result in overfitting, i.e., the situation where the architecture that is too complex learns to fit the noise and not just the true underlying signal. Again, the State Transition Entropy attribute can help to identify and avoid such situations.

As pointed out before, the stochasticity-related variance of value samples comes from two components: the variance of immediate rewards and the variability in the values of successor state-action pairs. The two attributes presented next aim at providing additional information in this respect, which is not captured by the State Transition Entropy.

3.3.2. Variance of Immediate Rewards

The *Variance of Immediate Rewards*, measured for state-action pairs, accounts for the variability in the immediate rewards due to both the variability of the next

states and the environmental responses to fixed (s, a, s') triples. Let $r_s^a \in \Re$ denote a random variable that represents the rewards observed after taking an action a in a state s. The Variance of Immediate Rewards is defined as follows:

$$VIR(s,a) = E\{(r_s^a)^2\} - (E\{r_s^a\})^2 \tag{3.12}$$

where the expectation is over the distribution of the next states, $P_{ss'}^a$, and the distribution of rewards for each (s, a, s') triple, which we denote $P_{rew}^{\langle s,a,s'\rangle}$.

This attribute can be estimated from the experience (see Section 3.4), or computed analytically if the reward variances $\sigma_{ss'}^a$ of the (s, a, s') triples are known. In the latter case, it can be computed as follows. The first term in Equation (3.12) can be expressed as follows:

$$E\{(r_s^a)^2\} = \sum_{s' \in S} P_{ss'}^a E_{P_{rew}^{\langle s,a,s'\rangle}}\{(r_{ss'}^a)^2\} \tag{3.13}$$

where $r_{ss'}^a \in \Re$ denotes a random variable that represents the rewards observed after taking action a in state s and making a transition to state s'. The variance of $r_{ss'}^a$, that is $\sigma_{ss'}^a$, is defined as:

$$\sigma_{ss'}^a = E_{P_{rew}^{\langle s,a,s'\rangle}}\{(r_{ss'}^a)^2\} - (E_{P_{rew}^{\langle s,a,s'\rangle}}\{r_{ss'}^a\})^2 = E_{P_{rew}^{\langle s,a,s'\rangle}}\{(r_{ss'}^a)^2\} - (R_{ss'}^a)^2 \tag{3.14}$$

If $\sigma_{ss'}^a$ is known, we can find:

$$E_{P_{rew}^{\langle s,a,s'\rangle}}\{(r_{ss'}^a)^2\} = \sigma_{ss'}^a + (R_{ss'}^a)^2 \tag{3.15}$$

Thus, from Equations (3.13)-(3.15), we obtain:

$$VIR(s,a) = \sum_{s' \in S} P_{ss'}^a \left[\sigma_{ss'}^a + (R_{ss'}^a)^2\right] - \left(\sum_{s' \in S} P_{ss'}^a R_{ss'}^a\right)^2 \tag{3.16}$$

The state-action pairs with high values of the Variance of Immediate Rewards should be sampled more extensively in order to obtain accurate estimates of their values $Q(s, a)$.

This attribute can be used for the purposes similar to the State Transition Entropy discussed earlier. It can be used in combination with the State Transition Entropy, as it provides an additional information, which is not explicitly contained in the latter attribute.

3.3.3. Controllability

The *Controllability* of a state s is a normalized measure of the information gain when predicting the next state based on the knowledge of the action taken, as opposed to making the prediction before an action is identified. A similar, but not identical, attribute was used by Kirman (1995). Let $O_s \in S$ denote a random variable representing the outcome of a uniformly random action in state s. Let A_s denote a random variable representing the action taken in state s. We consider A_s to be chosen from a uniform distribution. Now, given the value of A_s, the corresponding information gain is the reduction in the entropy of O_s:

$$H(O_s) - H(O_s|A_s) \tag{3.17}$$

The first term, $H(O_s)$, is the entropy of state transitions assuming no knowledge of the performed action. It can be computed as follows:

$$\begin{aligned} H(O_s) &= -\sum_{s' \in S} P_{s,s'} \log P_{s,s'} \\ P_{s,s'} &= \sum_{a \in A} \pi(s,a) P^a_{s,s'} \\ &= \sum_{a \in A} \frac{1}{|A|} P^a_{s,s'} \end{aligned} \tag{3.18}$$

Thus, from the above,

$$H(O_s) = -\sum_{s' \in S} \left(\frac{\sum_{a \in A} P^a_{s,s'}}{|A|}\right) \log\left(\frac{\sum_{a \in A} P^a_{s,s'}}{|A|}\right) \tag{3.19}$$

The second term in Equation (3.17) is the conditional entropy of state transitions provided that it is known which action has been performed. To compute this term

we use the standard definition of the conditional entropy which, in our case, represents the entropies $H(O_{s,a})$ averaged over all actions $a \in A$ assuming uniform action selection probabilities:

$$H(O_s|A_s) = -\sum_{a \in A} \frac{1}{|A|} \sum_{s' \in S} P^a_{s,s'} \log(P^a_{s,s'}) \tag{3.20}$$

Finally, the Controllability in the state s is defined as:

$$C(s) = \frac{H(O_s) - H(O_s|A_s)}{H(O_s)} \text{ , when } H(O_s) \neq 0 \tag{3.21}$$

If all actions are deterministic, then $H(O_s|A_s) = 0$ and $C(s) = 1$. If $H(O_s) = 0$ (all actions deterministically lead to the same state), then $C(s)$ is defined to be 0.

The Controllability reflects the difference in the state transition distributions of different actions in the state s. High controllability (e.g., greater than 0.5) means that the outcomes of different actions are quite different in terms of successor states. Consider, for example, the cases in Figure 3.3. The Controllability of a state on the left is the lowest, as two actions make a transition to the same state with a relatively high probability. The transition probability distributions of the two actions are very similar as well. A state on the middle picture has a higher Controllability, because the transition to the common successor state s_2 is performed with different probabilities with actions a_1 and a_2. Finally, a state on the right has the highest Controllability, because the sets of possible successor states are disjoint for the two actions. The Controllability is still less than 1 in this case, since each action has a relatively high entropy by itself, and hence, it is not possible to eliminate entirely the uncertainty about state transitions by the knowledge of the performed action.

In a highly controllable environment, the agent can exercise a lot of control over which trajectories (sequences of states) it goes through, by choosing appropriate actions. This enables the agent to reap higher returns in environments where small differences in the trajectories lead to significant differences in the expected returns. In such cases, the agent operating in a highly controllable environment could be expected to significantly outperform an untrained agent with a random behavior. States

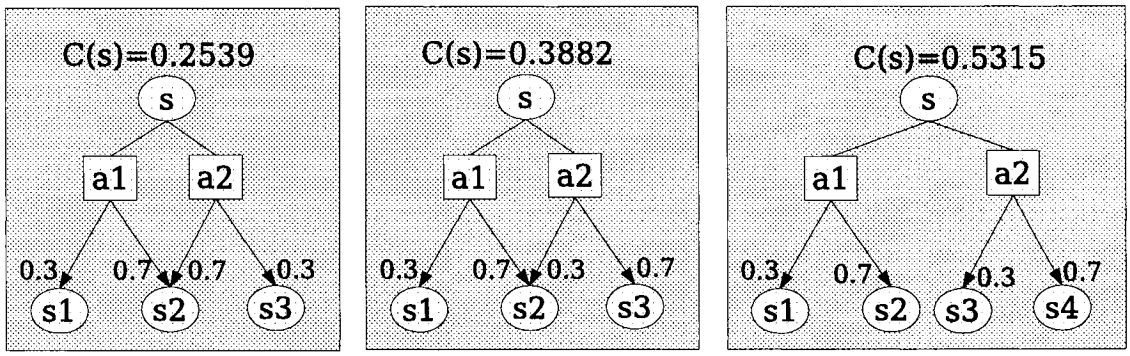


FIGURE 3.3. Examples of states with different Controllability values.

with high Controllability are very important, because in these states the agent can potentially impact most its performance.

Similar to the State Transition Entropy, the level of Controllability in an MDP also influences exploration of the state space. Because in a highly controllable state s the outcomes of different actions are quite different, the agent can choose what areas to explore. This can be advantageous for reinforcement learning, because it can facilitate directing exploration to desired areas through an appropriate choice of actions.

States with high Controllability also require more training. In such states, the agent has to learn about the distinct outcomes of different actions and thus it has to explore well distinct parts of the state space in order to properly estimate relative values of different actions. In other words, assume that a state s' on Figure 3.4 is highly controllable. This indicates that the sets of possible successor states, S_i, are fairly different for different actions a_i, $i = 1, ..., k$. The values of successor states in the set, say S_1, are used to estimate the value of the state-action pair (s', a_1). Unfortunately, most values from the set S_1 cannot be reused to estimate the value of the state-action pair (s', a_2), if the the sets S_1 and S_2 are quite different.

When the agent is learning with an on-policy method, such as Sarsa, a state s' with a high value $C(s')$ can increase the variance of value samples of its predecessor

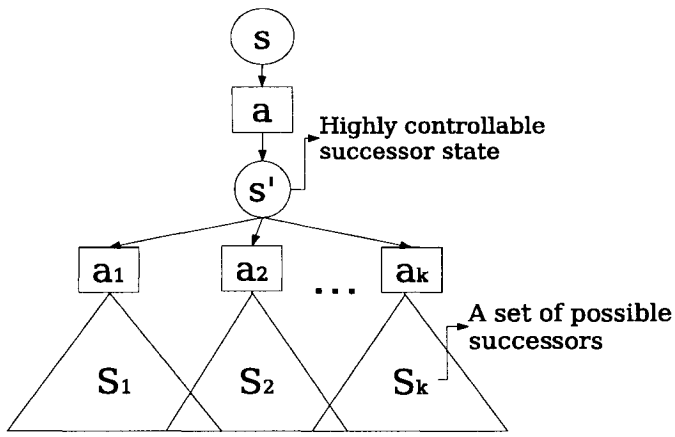


FIGURE 3.4. Effects of the Controllability.

state-action pairs (s, a) (see Figure 3.4 and Equation (3.7)). This happens if different actions a_i, $i = 1, ..., k$ are selected for exploration from the state s' and if the values $Q(s', a_i)$ of different actions a_i differ significantly. This is more likely if the state s' is highly controllable, because, as explained before, the actions' outcomes are significantly different.

In summary, knowing the Controllability can be helpful for identifying important states in the MDP. The ability to anticipate the Controllability of the successor states can be useful for assessing the potential variance introduced by these states into the value samples of their predecessors, as well as the exploration requirements for these states (see Chapter 4 for an application of these ideas). Thus, we also measure the *Forward Controllability* of a state-action pair (s, a), which is the expected Controllability of the successor state:

$$FC(s, a) = \sum_{s' \in S} P^a_{s,s'} C(s') \tag{3.22}$$

This attribute indicates whether the state-action pairs have, on average, highly controllable successors.

The values of the Controllability and the Forward Controllability attributes can be used in the following ways. First, these attributes can be used for improving efficiency of exploration (see Chapter 4). In this case, states with high Controllability

(or state-action pairs with high Forward Controllability) should be given a priority of visitation because they are very important and potentially require more training. The Controllability values can be also used to tune the learning rate parameter for on-policy reinforcement learning methods, so that lower settings are used for the state-action pairs with high Forward Controllability in order to mitigate the effect of a potentially high variance of value samples.

3.3.4. Reward Information Content

As explained above, the Controllability reflects the difference in the distribution of the next states for different actions, and thus indicates whether the outcomes of different actions are potentially different in terms of their long-term values. From a similar perspective, we can consider whether different actions in a particular state have different immediate rewards. This way, we would also measure how informative the immediate rewards are for differentiating between the values of different actions. For example, in goal-oriented tasks, it is typical to have zero-valued rewards everywhere except at the goal state. In this case, the immediate rewards contain little information for most of the state space. Learning and exploration in such MDPs is very difficult, since the agent usually has to revisit states many times until the reward information from the goal state propagates back across the entire state space. Such tasks are good candidates for using more sophisticated exploration techniques (see Section 4.2) as well as eligibility traces. However, these methods require extra computational overhead on every iteration. Thus, it is important to make an informed decision regarding the fact whether they can significantly affect performance.

The *Reward Information Content* estimates how different the mean rewards of different actions are in some state s. Let R_s^a be the average reward obtained after performing an action a in a state s:

$$R_s^a = \sum_{s' \in S} P_{ss'}^a R_{ss'}^a \tag{3.23}$$

Let R_s be the average reward obtained when performing a uniformly random action in the state s:

$$R_s = \sum_{a \in A} \frac{1}{|A|} \sum_{s' \in S} P^a_{ss'} R^a_{ss'} = \frac{1}{|A|} \sum_{a \in A} R^a_s \tag{3.24}$$

Then the Reward Information Content is defined as the variance of the average immediate rewards R^a_s across different actions:

$$RIC(s) = \frac{1}{|A|} \sum_{a \in A} (R_s - R^a_s)^2 \tag{3.25}$$

If $RIC(s) = 0$, then all actions have the same short-term value, and thus rewards by themselves do not help to determine relative advantages of different actions in the state s.

Note that we could estimate the Reward Information Content in several other ways. For example, we could use mutual information (see e.g., [MacKay, 2003]) between the actions and immediate rewards. This measure would indicate whether the actions have different short-term effects based on the difference of the *distributions* of their rewards, even if their means are similar. But since the state-action values are the estimates of the mean returns and action selection is based on these mean values, the definition that compares the means of rewards is more relevant. Also, we could consider only the maximum difference between values R^a_s of any pair of actions, namely, $\max_{a_1, a_2 \in A} |R^{a_1}_s - R^{a_2}_s|$. However, in this case, we would capture the distinction only between two actions that differ most. We would not be able to assess how the immediate rewards help to differentiate between the consequences of *all* actions in the considered state s.

The states that have a high value of the Reward Information Content attribute as well as a high value of the Controllability attribute have, in general, a high information content required to be learned[7]. In such states, the agent may need to learn a trade-off between short-term versus long-term benefits of actions, which potentially requires a lot of training experience. Thus, learning can benefit from prioritizing visitations of

[7]By information content we do not mean the technical Shannon's measure of the compressibility of the information description, but rather the associated complexity.

such states. On the other hand, states with very low values of the Reward Information Content and low Controllability values are of the lesser importance to the learning agent, since the state-action values of different policies will likely be similar.

Nevertheless, states with low Reward Information Content and low Controllability should not always be deemphasized, because accurate estimates of action-values in these states may be necessary to compute good value estimates in other states. This can be related to the notion of *influence* introduced in [Munos and Moore, 2001]. The influence of a state s_1 on another state s_2 under policy π is defined as a measure of the extent to which the state s_1 contributes to the value of the state s_2. It was shown in [Munos and Moore, 2001] that it can be computed exactly by an iterative procedure, which is as costly as computing the value function of the corresponding policy[8]. If such a measure of influence can be computed (either exactly or approximately), it would be useful to assess whether a state s with a low Reward Information Content and a low Controllability has a significant influence on other important states, in which case the value of s should be learned well.

Similar to the Controllability attribute, the Reward Information Content of a state s' can give an indication about the variance of value samples introduced for the predecessor state-action pairs if an on-policy algorithm, such as Sarsa, is used.

The Reward Information Content attribute can potentially be used in the following ways. Assessing the values of the Reward Information Content throughout the state space can help to improve exploration: MDPs with large areas of low Reward Information Content are good candidates for directed exploration methods as well as high values of the eligibility traces. This attribute can also be used for tuning the learning step parameter, similar to as it was already discussed in the case of the Controllability attribute. The Reward Information Content should be considered in

[8]In some cases, high computational cost of this method is justified, e.g., for improving the design of a function approximator. In [Munos and Moore, 2001], states with the high return variance (computed in the way mentioned in the previous section) were checked for whether they significantly influence other states considered important. If this was the case, an attempt was made to approximate more accurately the values of the high-variance states.

combination with other attributes discussed above in order to identify important and difficult states.

3.3.5. Risk Factor

As discussed in Section 3.2, the learning agent has to perform exploration during on-line learning while sometimes being constrained by a certain risk tolerance level. In Section 3.2, we mentioned several reinforcement learning approaches that treat risk considerations explicitly. Alternatively, risk tolerance can be taken into account during learning in the following way. We define the *Risk Factor* attribute, which measures the likelihood of getting a low reward after the agent performs a uniformly random action.

Let $r_s \in \Re$ denote a reward observed on a transition from a state s after performing a uniformly random action. The Risk Factor in the state s is defined as follows:

$$RF(s) = Pr[r_s < R_s - \epsilon(s)], \tag{3.26}$$

where R_s is the average reward observed after taking a uniformly randomly chosen action in state s (see Equation (3.24)), and $\epsilon(s)$ is a positive number, possibly dependent on the state, which quantifies the tolerance to lower-than-usual rewards.

This definition is similar in spirit to the α-value criterion discussed in the previous section. However, the Risk Factor provides only a myopic assessment of the risk, since low immediate rewards do not necessarily mean low long-term returns and vice versa. But this attribute is easy to estimate and it can be helpful for minimizing losses, especially during the early stages of learning. A similar (myopic) definition of risk was used in [Hauskrecht *et al.*, 2001] for the problem of computing risk-sensitive investment strategies for multi-market commodity trading. In [Hauskrecht *et al.*, 2001], this problem, formulated as an MDP, was solved by linear optimization techniques taking advantage of a particular problem structure. In future work, we plan to investigate whether a myopic definition of risk, such as the Risk Factor attribute, can

be successfully used with standard reinforcement learning algorithms for computing (long-term) optimal value functions.

The Risk Factor attribute can also be useful in environments, where the rewards are not strictly Markovian. For instance, a person interacting with the learning agent gets irritated by the agent's frequent non-sense (exploratory) actions and starts to give bigger penalties. If the Risk Factor is estimated high in some state s, the agent can lower the exploration rate in this state.

3.4. Computing the Attributes

The attributes discussed in the previous section, can be measured locally, for each state or state-action pair, or globally, as an average over all states or state-action pairs of the MDP. Local measures are most useful for fine-tuning parameters of reinforcement learning algorithms for individual states or state-action pairs. For example, in Chapter 4, we present an exploration strategy that uses local attribute values to decide on the action selection probabilities for exploration. On the other hand, in the empirical study presented in Section 3.5, we use global measures to study an impact of the overall MDP characteristics on the performance of learning algorithms. To obtain the global measures, we can use sample averages of the attribute values over multiple state-action pairs. One possibility is to assume that all states and actions are equally probable or to use a distribution corresponding to following a uniformly random policy. This choice of the sampling distribution allows to characterize the MDPs before any learning takes place without having to fix any particular (non-uniform) policy. However, under some circumstances, weighted averages might be of more interest, e.g., the attribute estimates computed based on a behavior generated by a restricted class of policies. If the sample averages are estimated on-line, during learning, they naturally reflect the state distribution that the agent is actually experiencing.

So far we presented the definitions of the attributes for discrete finite state spaces. All the definitions extend naturally to the continuous state spaces by integrating over the domains of the state variables instead of using summations. In practice, in the

case of a continuous state space, it is also possible to approximate the attributes by discretizing the state space and using a histogram approximation of the transition probability distribution and a piece-wise constant approximation of the immediate reward function. However, using a discretization may not always be possible, for instance, when the state space is high- dimensional. Also, it can fail to provide the most useful information in the case, where a different kind of a function approximator is used to represent the value function. If a local architecture is used to approximate the value function, the attributes can be associated with local units (e.g., basis functions) as opposed to states (similar to the way in which the eligibility traces are used in this case, see e.g., [Santamaria *et al.*, 1998]). The attributes will then reflect the MDP properties with respect to the chosen approximate representation over the state space and can indicate relative characteristics of different representations. We present more discussion about this option at the end of this section.

In the presentation of the attributes above, we also assumed that the transition probabilities and means of the immediate rewards were known. If this is not the case, the attributes can be estimated from observed samples of state transitions and rewards, both for discrete and continuous state spaces. In the following section, we describe these estimators. All the incremental estimators discussed below allow the computation of the attributes on-line, either prior to or in parallel with learning the value functions. Thus we can use the attributes to adjust certain learning parameters, as we do for the attribute-based exploration strategy presented in Chapter 4.

3.4.1. Transition Probabilities

Estimates of the transition probabilities can be used in order to compute some of the attributes, in particular the State Transition Entropy and the Controllability. These estimates can be updated incrementally as follows.

In the case of a discrete finite state space, for each state-action pair (s, a), we need to keep a counter N_s^a for the number of state transitions observed *from* the state-action pair (s, a). This counter is incremented every time the corresponding

state-action pair is visited and no termination condition is satisfied. For each state-action pair (s, a), we also need to keep a *set* of counters $N^a_{ss'}$, one for each possible next state s', of the number of observed transitions *to* the corresponding next state s'. Each counter is incremented by one every time a transition to the corresponding next state is observed from the state-action pair (s, a). The approximate transition probabilities are then calculated as follows:

$$P^a_{ss'} = \frac{N^a_{ss'}}{N^a_s} \ , \forall s, s' \in S \text{ and } a \in A \tag{3.27}$$

In the case of a continuous state space, the state space can be discretized into a set of disjoint bins (hypercubes), b_i, $i = 1, ..., B$. Then we maintain a counter N^a_i for each bin-action pair which corresponds to the number of state transition samples observed when performing the action a *from* any state that belongs to the bin b_i. Then, for each bin-action pair (i, a), we also maintain a *set* of counters N^a_{ij}, for each bin $j = 1, ..., B$ corresponding to the next state. Each counter is incremented by one every time a transition to any state in the bin b_j is observed after performing the action a from a state in the bin b_i. We can then calculate *bin-to-bin* transition probabilities as

$$P^a_{ij} = \frac{N^a_{ij}}{N^a_i} \ , \forall i, j = 1, ..., B \text{ and } a \in A \tag{3.28}$$

3.4.2. Mean of a Random Variable

We need to estimate the mean of a random variable for several attributes. First of all, we need the means of immediate rewards: $R^a_{ss'}$ for the (s, a, s') triples, if they are not known a priori; R^a_s for the state-action pairs (s, a), as defined in Equation (3.23); and finally R_s for the states s, as defined in Equation (3.24). We need these estimates in order to compute the Reward Information Content and the Risk Factor attributes. As discussed in Section 3.2, the mean of a random variable can be estimated incrementally as in Equation (3.1). Since the rewards have a stationary probability distribution, we can use the *running average* mean estimator. For example, assuming a discrete state space, in order to estimate $R^a_{ss'}$, we use the following

update after observing a reward $r^a_{ss'}$ on a transition from state s to state s' with action a:

$$R^a_{ss'} := \frac{N^a_{ss'} R^a_{ss'} + r^a_{ss'}}{N^a_{ss'} + 1} \tag{3.29}$$

where $N^a_{ss'}$ is the number of reward samples observed so far for the corresponding triple (s, a, s') not counting the last sample $r^a_{ss'}$.

We can estimate R^a_s in a similar manner by using all the reward samples observed after performing the action a in state s, regardless of the next state.

Finally, in order to estimate R_s, the mean rewards in the state s under the uniform action selection probability distribution, we can use Equation (3.24) and the estimates of the means R^a_s for all actions.

The estimation of the mean is also required for the computation of the Forward Controllability, as in Equation (3.22), which is the mean of the Controllability in the successor states with respect to the state transition probability distribution. To estimate this attribute for each state-action pair, $FC(s, a)$, we would first need to estimate the values $C(s')$ for every state s' and then to estimate the Forward Controllability itself for the state-action pairs (s, a). It is possible to update the estimates of $C(s)$ and $FC(s, a)$ in parallel (see below for a discussion of an incremental entropy estimator involved in computation of $C(s)$ values). In this case, the values $FC(s, a)$ can be computed using the mean estimator for a random variable with a non-stationary probability distribution in order to account for the changing estimates of $C(s')$ in the successor states. The incremental recency weighted average estimator, as in Equation (3.4) can be used for this purpose. After every observation of a state transition from state s with action a to state s', the estimate of the Forward Controllability is updated as follows:

$$FC(s, a) := (1 - \alpha) FC(s, a) + \alpha C(s') \tag{3.30}$$

We can choose either a constant step-size α or use a decreasing schedule, for instance,

$$\alpha_{N^a_s} = \frac{\beta - 1}{\beta^{N^a_s} - 1} \tag{3.31}$$

where $\beta \in (0,1)$ and N_s^a is the number of samples received so far for the estimation of $FC(s,a)$. This rule corresponds to starting with $\alpha = 1$ (the first sample is the estimate of the mean) and then decreasing the learning step α every time a new sample is received until eventually the asymptote value of $\alpha = (1-\beta)$ is reached, behaving as an estimator with a constant α thereafter. It is appropriate to start with high values, because the values of $C(s')$ are expected to change most during the initial period and this change should be tracked quickly.

In the case of a continuous state space, piece-wise constant sample mean estimates can be obtained by discretizing the state space as discussed above. Alternatively, as pointed out before, each sample mean estimator can be associated with a local feature of a local function approximator and updated every time the local unit is activated.

3.4.3. Sample Variance

We can also estimate the Variance of Immediate Rewards from samples as well. Since we are computing the sample variance of a stationary random variable, the estimation can be done incrementally as follows. Suppose we have a set of samples of immediate rewards, $r_s^a(i), i = 1, ..., N_s^a$, received after taking action a in state s. Then the Variance of Immediate Rewards can be computed as the sample variance:

$$VIR(s,a) = \frac{1}{N_s^a - 1} \sum_{i=1}^{N_s^a} \left[\bar{R}_s^a - r_s^a(i)\right]^2 \tag{3.32}$$

where $\bar{R}_s^a = \frac{1}{N_s^a} \sum_{i=1}^{N_s^a} r_s^a(i)$ is the sample mean, obtained after observing N_s^a samples.

We can compute the sample variance incrementally, that is by updating the estimate of $VIR(s,a)$ each time we receive a new sample $r_s^a(i)$ and then discarding this sample. This can be done using the following transformation of Equation (3.32):

$$\begin{aligned} VIR(s,a) &= \frac{1}{N_s^a - 1} \sum_{i=1}^{N_s^a} \left[\bar{R}_s^a - r_s^a(i)\right]^2 \\ &= \frac{1}{N_s^a - 1} \sum_{i=1}^{N_s^a} \left([\bar{R}_s^a]^2 + [r_s^a(i)]^2 - 2\bar{R}_s^a r_s^a(i)\right) \\ &= \frac{1}{N_s^a - 1} \left(N_s^a [\bar{R}_s^a]^2 + \sum_{i=1}^{N_s^a} [r_s^a(i)]^2 - 2N_s^a [\bar{R}_s^a]^2\right) \\ &= \frac{1}{N_s^a - 1} \left(\sum_{i=1}^{N_s^a} [r_s^a(i)]^2 - N_s^a [\bar{R}_s^a]^2\right) \end{aligned} \tag{3.33}$$

We can maintain the incremental estimates of the sum of squares, $\sum_{i=1}^{N_s^a}[r_s^a(i)]^2$, and the sample mean, $\bar{R}_s^a$, and use them to compute the sample variance on any step by employing the formula in the last row of Equation (3.33). The sample mean $\bar{R}_s^a$ can be computed by the running average estimator and is equivalent to the estimate of R_s^a discussed earlier.

Once again, in the case of a continuous state space, we can either use a discretization of the state space or associate the estimates with the units of a local function approximator to compute the attributes, as discussed above.

3.4.4. Entropy of a Random Variable

In order to estimate the State Transition Entropy and the Controllability attributes, it is necessary to estimate the entropies of random variables. One possibility would be to first estimate the corresponding transition probabilities, as discussed earlier, and then compute all the entropies. Alternatively, we can estimate the entropies directly in an incremental manner using the algorithm from [Vignat and Bercher, 1999]. For example, to estimate the value of $STE(s,a)$ in the case of a discrete state space, we can proceed as follows.

We keep the counters N_s^a and $N_{ss'}^a$ as described in Section 3.4.1. Every time we receive a new $(N_s^a+1)^{th}$ transition sample (s,a,s'), we update the entropy estimator using the following formula (see [Vignat and Bercher, 1999] for the derivation of this update rule:)

$$\begin{aligned} &STE(s,a) := \tfrac{N_s^a}{N_s^a+1} STE(s,a) + \tfrac{1}{N_s^a+1}\left[g(N_s^a) - g(N_{ss'}^a)\right] \text{ , for } N_s^a > 1 \\ &g(x) = (x+1)\log(x+1) - x\log x \end{aligned} \tag{3.34}$$

After that the counters N_s^a and $N_{ss'}^a$ are also updated. In general, entropies are initialized to 0 for $N_s^a = 1$.

It is also possible to estimate incrementally the entropy of a random variable with a non-stationary probability distribution. In this case, we only need to change the weights used to combine the previous estimate and the contribution of the new sample, just like in the case of the incremental mean estimator, discussed in Section

3.4.2. The update rule then is as follows:

$$STE(s,a) := \frac{\beta^{N_s^a+1}-\beta}{\beta^{N_s^a+1}-1} STE(s,a) + \frac{\beta-1}{\beta^{N_s^a+1}-1}\left[g(N_s^a) - g(N_{ss'}^a)\right] \text{ , for } N_s^a > 1 \quad (3.35)$$

where β is a parameter implementing the decreasing schedule. When N_s^a approaches infinity, the term $\beta^{N_s^a+1}$ approaches zero. In this case, current estimates $STE(s,a)$ will be weighted with a limiting factor β and new samples will be weighted with a factor of $1-\beta$. Usually β is selected to be greater than 0.5 in order to eventually make the contribution of new samples smaller than that of the current estimate. If the distribution is expected to become stationary with time, the value of β can be chosen very close to 1, e.g., $\beta = 0.997$. We use this variant of the entropy estimator when we want to initialize the estimator of $STE(s,a)$ to a value different from zero, for example an over-estimate of the entropy, which is desirable in some situations (see Chapter 4). Then gradually the estimate will approach the true value of $STE(s,a)$.

The entropy can be estimated in the same manner for continuous random variables, which we need to do for a continuous state space. Just like before, we can discretize the state space into a set of disjoint bins and keep the corresponding counters for the bins, N_i^a and N_{ij}^a, as explained before. Then the estimator of the State Transition Entropy is piece-wise constant over the state space, with constant values inside the discretization bins. The following update rule, equivalent to the rule in Equation (3.34), can be used upon observing a sample of a transition from the state $s \in b_i$ to the state $s' \in b_j$ after performing the action a:

$$STE(b_i,a) := \frac{N_i^a}{N_i^a+1} STE(s,a) + \frac{1}{N_i^a+1}\left[g(N_i^a) - g(U_{ij}^a) + \log h_j\right] \text{ , for } N_i^a > 1$$
where h_j is the volume of the bin b_j. (3.36)

An update rule, equivalent to Equation (3.35) can be obtained by substituting the same weighting factors as in (3.35).

In order to estimate the Controllability, we need to compute the entropies $H(O_s)$ and $H(O_s|A_s)$, as defined in Equation (3.21). The entropy $H(O_s|A_s)$ can be computed as the average of the State Transition Entropies $STE(s,a)$ over all actions in the state

s (see Equation (3.20)). The entropy $H(O_s)$ can be computed incrementally just like the State Transition Entropy in Equations (3.34)-(3.36). In this case we need to keep the following counters in each state s: N_s (instead of N_s^a) for the number of observed transitions from the state s and a set of counters $N_{ss'}$ (instead of $N_{ss'}^a$), one for each successor state s', for the number of observed transitions to the state s' from the state s. Recall, however, that $H(O_s)$ is the entropy of the random variable O_s that corresponds to the outcomes of the state transitions under the uniformly random action selection. If the agent's behavior policy is not uniformly random, we can proceed in one of two ways as follows. If the behavior policy is ϵ-greedy, then we would collect samples for this entropy estimate only on exploration steps when actions are chosen uniformly randomly. Alternatively, regardless of the behavior policy, we can have a "pigeon-hole" holder, where we would try to collect exactly one sample for every action in the state s, then process all these samples and discard them.

3.4.5. Estimating with Local Function Approximators

As already pointed out, in the case of a continuous state space, the attributes can be associated with units of a local function approximator (e.g., basis functions of a RBFN) instead of discretization bins. Below we provide a discussion about how this can be done in the case when all actions have the same structure of the function approximator. For example, if RBFNs are used to represent the action-value function of each action, the sets of centers and widths of the RBFs used for different actions should be the same.

We can use the incremental estimators discussed above, updating them on-line upon the activation of the local units. For many approximators, multiple (overlapping) units can be activated at the same time by each state (recall, e.g., tiles in the CMAC architecture discussed in Chapter 2). In this case, estimators associated with all activated units have to be updated upon observing a transition for a state-action pair (s, a).

Also, in this case, we will need to keep the following counters for the estimation of entropies as well as for the estimation of $R^a_{ss'}$ in Equation (3.29). Let ϕ_i, $i = 1, ..., N$ be a set of units of the function approximator, e.g., all the tiles in the CMAC architecture. For each pair of a unit ϕ_i and action a, we keep a counter N^a_i for the number of times this unit was activated and the action a was performed. Then, let Ψ be a set of all possible subsets of units, where the sizes of these subsets depend on the minimum and maximum number of units that can be active at the same time. For instance, in the CAMC architecture, exactly T tiles are always active, where T is the number of tilings. In this case, the size of each subset of units, $\psi_j \in \Psi$, would be T.

For each unit-action pair (ϕ_i, a) we then keep a set of counters N^a_{ij} for the number of times that a subset of units ψ_j was activated by a successor state s' of any state-action pair (s, a) such that ϕ_i was activated by the state s. Of course, the total number of all possible subsets ψ_j can be large, but it is only necessary to keep the record for those subsets that have non-zero counters. Typically, there are few of them in practice.

An approach that would require less counters to be maintained for each unit could be used with function approximators, in which activations are continuous as opposed to binary, e.g. RBFNs or Sparse Distributed Memories (see Chapter 5) as opposed to CMACs. In this case, for each unit ϕ_i, we can keep a set of counters N^a_{ij}, where $j = 1, ..., N$ (N is the number of units and not subsets of units as before). Each counter N^a_{ij} will be incremented on every transition $(s, a) \rightarrow s'$, such that the state s activates ϕ_i and ϕ_j has the *largest* activation value among all units activated by the state s'. In this case, we would account for units that contribute most to the values of the successor states. We could also use (continuous) amounts of activation instead of mere activation counts.

3.5. Effect of MDP Attributes on Learning: Empirical Study

In Section 3.3, we introduced a number of attributes to quantitatively measure certain properties of Markov Decision Processes. These attributes reflect mainly the amount and the nature of the environment's stochasticity, as well as the amount of control that the agent has over its environment. We hypothesized that MDP characteristics measured by the proposed attributes have an effect on the quality of policies obtained after some limited training time with on-line value-based reinforcement learning methods.

In this section, we focus on empirically verifying this hypothesis with respect to two attributes: the State Transition Entropy and the Controllability. In the work of Kirman (1995) these attributes appeared to influence the performance of off-line dynamic programming algorithms significantly more than other attributes considered in his work. This fact was one of the motivations for starting our empirical investigation with these two attributes. Also, these attributes give the most general characterization of the domain's stochasticity. Thus, we wanted to investigate whether it is possible to detect the effects of these two attributes on the performance of the learning agent without using more detailed measurements of the reward and value-function structure.

As previously indicated, it is not our objective to build accurate performance prediction models based on attribute values, as it was done in [Kirman, 1995]. The work of Kirman indicated that such models do not seem to be generally useful for domains not involved in the model construction. Here, the objective is to verify our hypothesis concerning the existence of a statistically significant relationship between the attribute values and the empirical performance of on-line value-based reinforcement learning algorithms, both tabular and approximate. Effects of the MDP properties measured by these attributes were not studied for such algorithms in the existing literature.

3.5.1. Experimental Methodology

This section presents a number of experiments aimed at evaluating performance of reinforcement learning algorithms on different MDPs. We will first describe the type of data and performance measures, which we use to analyze our experimental results.

Experimental Data and Performance Measures

Because of the stochasticity of MDPs and the randomized nature of reinforcement learning algorithms, multiple *learning runs* are performed for each task, algorithm and parameter setting. Each learning run consists either of an unbroken chain of state transitions or a number of *trials*. In the case of episodic MDPs, trials represent episodes, and in the case of continuing MDPs, they represent a limited, sufficiently large number of simulated steps, after which the system is reset to a new starting state.

One popular way of visualizing experimental results is by plotting average *learning curves*. Learning curves represent a performance measure of interest (for example, returns accumulated during learning) as a function of the number of learning trials performed on a given MDP. Each point on the learning curve usually represents an average of performance measurements over multiple learning runs.

In the case of on-line reinforcement learning methods, the following performance characteristics are usually of interest:

(i) Solution quality that can be measured by the returns obtained using the policy that is greedy with respect to the value function learned.

(ii) Speed and stability of learning, i.e., how quickly the solution improves and how much the performance deviates from its mean or maximum throughout learning.

(iii) Performance during learning, which is reflected by the rewards accumulated by the agent during the entire learning process.

In the empirical studies presented in this thesis, we focus mainly on the solution quality and the evolution of the solution during learning. Performance during learning

is usually of interest if the agent has to minimize its loss while learning in a real-world setting.

In order to obtain measurements of the solution quality, we empirically evaluate the policies that are greedy with respect to action-value functions obtained at different stages of learning. We select a set of test states, from which we start the evaluation trials. Such starting test states are usually sampled from the same distribution as the starting states for the learning trials. A set of test states is fixed beforehand and then used for all evaluations in order to minimize unexplained variance in the data.

For each start state in the test set, S_{test}, we estimate the average return of the greedy policy, $R_{t,k}(s), s \in S_{test}$, where t is the number of elapsed learning trials on the k^{th} learning run. Usually, evaluation of greedy policies is performed with a certain frequency, so that $t \in \{0, G, 2G, ..., lG\}$ where lG is the total number of learning trials on each run. In order to estimate the average return $R_{t,k}(s)$ for each test state, we simulate a certain number of trajectories from the corresponding state and average the returns observed. Then we take the average over all test states to obtain *greedy returns*:

$$R_{t,k} = \frac{1}{|S_{test}|} \sum_{s \in S_{test}} R_{t,k}(s) \ , \ t \in \{0, G, 2G, ..., lG\} \ , \ k \in \{1, ..., K\} \tag{3.37}$$

The estimated greedy returns $R_{t,k}$ are then averaged over all learning runs executed:

$$R_t = \frac{1}{K} \sum_{k=1}^{K} R_{t,k} \ , \ t \in \{0, G, 2G, ..., lG\} \tag{3.38}$$

These average measurements are used to plot learning curves (of R_t as a function of t).

For the experimental studies that involve many MDPs or algorithms, it can be difficult to visualize and compare many learning curves. In this case, other more succinct performance measures can be used.

As previously discussed, convergence cannot always be achieved in practice in reinforcement learning (e.g., due to the use of function approximation, limited training

time, sub-optimal settings of user-tunable parameters, etc.). Typically, the agent's behavior does not improve monotonically with every learning trial and thus certain variations in performance may occur throughout learning. Because of this, the agent would usually keep a record of the *best* greedy policy found in the course of a learning run and consider it to be a solution instead of the policy obtained at the end of a designated learning period. Hence, one possibility is to measure the solution quality using the average returns of the best greedy policies obtained in the course of each learning run. We call this measure the average *Best Greedy Return*:

$$BGR = \frac{1}{K}\sum_{k=1}^{K}\left(\max_{t\in\{0,G,2G,\ldots,lG\}} R_{t,k}\right) \tag{3.39}$$

This measure is used as an indicator of the overall solution quality achieved, on average, by the algorithm on a given task.

We are also interested in analyzing the evolution of the solution quality over the learning period. One possibility is to compute the sum of greedy returns over all trials on each learning run and average it over all runs. We call this measure the *Cumulative Greedy Return*:

$$CGR = \frac{1}{K}\sum_{k=1}^{K}\sum_{t\in\{0,G,2G,\ldots,lG\}} R_{t,k} \tag{3.40}$$

Although learning speed and stability are very important performance characteristics, to the best of our knowledge, there is no standard measure to evaluate them accurately. The Cumulative Greedy Return measure would fail to capture performance differences if, for example, one learning curve is steeper than another learning curve at the beginning but then exhibits asymptotic performance that is lower or has more variance. So we use a variant of the Cumulative Greedy Return measure that helps, to some extent, in highlighting differences in such situations. It is usually possible to estimate an upper bound, R^{max}, on the return of an optimal policy for a given task. For instance, we can obtain such an estimate by assuming that the maximum possible reward can be obtained on each step. We can assign penalties

for not reaching the R^{max} bound at different stages of learning and then accumulate such penalties on a given run. More specifically, we use the following performance measure, which we call the *Cumulative Weighted Penalty*:

$$CWP = \frac{1}{K}\sum_{k=1}^{K}\left[\sum_{t\in\{0,G,2G,\ldots,lG\}}\frac{t}{lG}(R^{max} - R_{t,k})\right] \tag{3.41}$$

Because of the weights $\frac{t}{lG}$, failure to come close to the upper bound R^{max} is penalized more as the number of learning trials increases, because we expect the performance to get (and stay) better with learning. Hence, this measure gives lower penalties to those learning methods that achieve better asymptotic performance and do not deviate much from their best solutions. Figure 3.5 shows an example of two curves. Neither the Best Greedy Return measure nor the Cumulative Greedy Return measure find much difference in the performance they represent, whereas the Cumulative Weighted Penalty measure indicates that the performance represented by Curve 1 is better (it has a lower penalty).

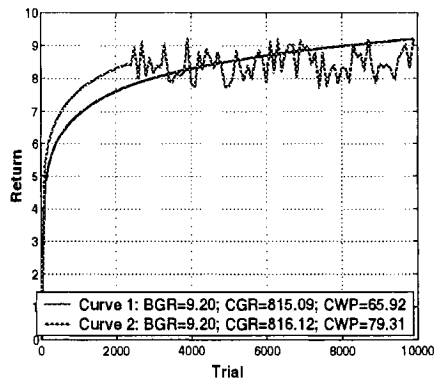


FIGURE 3.5. Example of different performance measures. BGR stands for the Best Greedy Return measure, CGR - for the Cumulative Greedy Return measure and CWP - for the Cumulative Weighted Penalty.

In this section, we want to study the performance of reinforcement learning algorithms on classes of MDPs, such that MDPs in a given class are characterized by particular values of the studied MDP attributes. To draw general conclusions about the performance of an algorithm on a class of MDPs, the experiments should be

performed on multiple MDPs belonging to that class. For different MDPs, optimal policies may have different returns, hence upper bounds on the performance measures are different. This is why it is not always meaningful to aggregate directly (e.g., average) performance measurements, such as discussed above, for different MDPs in the class. In this case, it is necessary to normalize performance measurements for each MDP in order to obtain a common baseline.

It would be best to normalize with respect to the expected returns of the optimal policy of the corresponding MDP, if it can be obtained by independent means, for example, by applying a dynamic programming algorithm. In this case, it is also desirable to account for a difference in returns attainable by the optimal policy and by the uniformly random policy (behavior of an untrained agent[9]). We use the following normalization for greedy returns:

$$\hat{R}_{t,k} = \frac{R_{t,k} - R^{rand}}{R^{op} - R^{rand}} \ , \ t \in \{0, G, 2G, ..., lG\} \ , \ k \in \{1, ..., K\} \tag{3.42}$$

where R^{op} represents the average return of the optimal policy on the starting test states and R^{rand} represents the average return of the uniformly random policy on the same set of start states. Greedy returns $R_{t,k}$ are computed as explained above. In Equation (3.42), the denominator represents the advantage of the optimal policy over the random one (i.e., the amount to be learned), and the numerator represents the advantage of the learned policy over the random one. Hence, the ratio represents how much of the required performance gain the agent achieved through learning.

Normalized greedy returns can be used to compute other performance measures, such as the Best Greedy Return measure and the Cumulative Weighted Penalty measure. Such measures can then be averaged over all tested MDPs that belong to a particular class.

Unfortunately, optimal policies cannot always be obtained by independent means. In some experiments, we use MDPs with continuous state spaces, for which only

[9]Note that the agent's behavior is usually initialized to the uniformly random policy at the beginning of learning.

simulation models are available, but not explicit MDP models. In this case, an obvious performance baseline is the performance of the uniformly random policy. As we will show later, it can also provide an acceptable normalization factor and yield results that are qualitatively similar to those obtained by using normalization with respect to the optimal performance.

3.5.2. Benchmark Domains

In this empirical study, our objective is to consider both tabular and approximate reinforcement learning algorithms. Thus, we conduct our experiments on domains with discrete (small) state spaces as well as on MDPs with continuous state spaces. In order to study empirically the effect of the State Transition Entropy and the Controllability attributes on learning, it is desirable to consider a range of values of these attributes and to vary them independently. Unfortunately, the main collection of experimental tasks available from the "Reinforcement Learning Repository" at the University of Massachusetts, Amherst[10], that are currently used in the reinforcement learning community, contains only a handful of domains, and the continuous tasks are mostly deterministic. So our experiments were performed on artificial, randomly generated MDPs (both discrete and continuous) as well as on randomized versions of a well-studied Mountain-Car task [Sutton and Barto, 1998]. We will now describe these domains.

Random MDPs

Random discrete MDPs have already been used in the literature for experimental studies with tabular reinforcement learning algorithms. We use as a starting point a design suggested by Sutton and Kautz for discrete, enumerated state spaces[11], but we extend it in order to allow feature-vector representations of the states, where the features can be either discrete or continuous. Figure 3.6 shows how transitions are performed in a random MDP. The left panel shows the case of a discrete, enumerated

[10] www-anw.cs.umass.edu/rlr
[11] www.cs.umass.edu/~rich/RandomMDPs.html

state space, as used by Sutton and Kautz. For each state-action pair (s, a) the next state s' is selected from a set of b possible next states, according to the probability distribution $P(s, a, s'_j), j = 1, ..., b$. The reward is then sampled from a normal distribution with mean $R(s, a, s')$ and variance $V(s, a, s')$. Such random MDPs are easy to generate automatically.

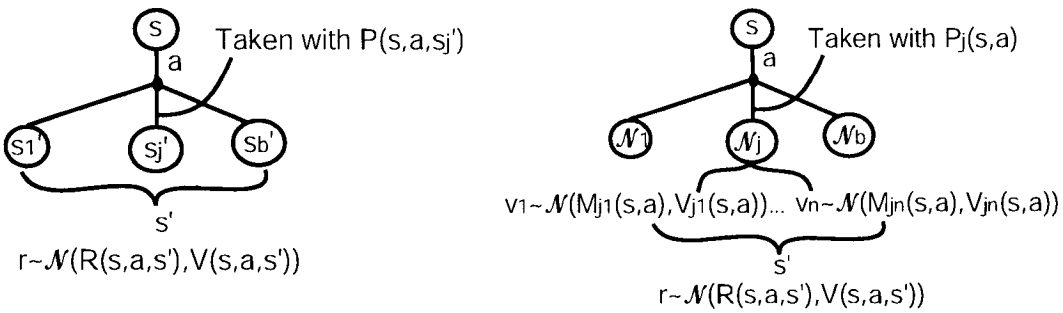


FIGURE 3.6. Random MDPs. Left: enumerated states; right: feature-vector states.

Our random MDPs are a straightforward extension of this design. A state is described by a feature vector: $s = \langle x_1, ..., x_n \rangle$, with $x_i \in [0, 1]$, $i \in \{1, ..., n\}$. State transitions are governed by a mixture of b multivariate normal distributions $\mathcal{N}(\mu_\mathbf{j}, \sigma_\mathbf{j})$, with means $\mu_\mathbf{j} = \langle \mu_j^1, \ldots \mu_j^n \rangle$ and variances $\sigma_\mathbf{j} = \langle \sigma_j^1, \ldots \sigma_j^n \rangle$, $j = 1, ..., b$ (Gaussians with diagonal covariance matrices). The parameter b is called the branching factor. The means $\mu_j^i = M_j^i(s, a)$ and variances $\sigma_j^i = V_j^i(s, a)$ are functions of the current state-action pair (s, a). Sampling from this mixture is performed hierarchically: first, one of the b Gaussian components is selected according to probabilities $P_j(s, a), j = 1, \ldots b$, then the next state s' is sampled from the selected component, $\mathcal{N}(\mu_j, \sigma_j)$. Sampling from such a component can be done independently for each state variable, since we use diagonal covariance matrices. Note, that even though this assumes independence of state variables on the *same* time step, each state variable depends on values of *all* state variables on the previous time step through functions $\mu_j^i = M_j^i(s, a)$, $\sigma_j^i = V_j^i(s, a)$ and $P_j(s, a)$. We can always recover a discrete model from this case by setting the variances σ_j^i to zero and using discrete feature values.

Mixtures of Gaussians are a natural and non-restrictive choice for modeling multivariate distributions. Of course, one can use other basis distributions as well. Once the next state s' is determined, the reward for the transition is sampled from a normal distribution with mean $R(s,a,s')$ and variance $V(s,a,s')$. Episodic tasks can be modeled by using a termination probability distribution $P(s')$ over states.

We designed a generator for random MDPs of this form, which uses as input a textual specification of various parameters, for instance, the number of state variables, actions, a branching factor (number of Gaussian components) for each action, lower and upper bounds for the variance of Gaussian components, etc. The branching factor can be either uniform across the entire state space or vary randomly across state-action pairs within a specified range.

In our experiments we used piecewise constant functions to represent $P_j(s,a)$, $M_j^i(s,a)$, $V_j^i(s,a)$, $R(s,a,s')$ and $V(s,a,s')$. For each such piecewise constant function, the origin of the discretization grid is displaced randomly in a small neighborhood of the zero feature vector[12]. In this way, the states that, for example, share the same state transition mean, do not necessarily have the same transition variances, transition probabilities and rewards. In order to obtain random MDPs, the parameters of these functions (constants for discrete partitions) are randomly generated.

Mountain-Car Domain

The Mountain-Car domain [Sutton and Barto, 1998] is a very well-studied minimum time-to-goal task. The agent has to drive a car up a steep hill by using three actions: full throttle forward, full throttle reverse, or no throttle. The engine is not sufficiently strong to drive up the hill directly, so the agent has to build up sufficient energy first, by accelerating away from the goal. The state is described by two continuous state variables: the current position and velocity of the car. We use the state dynamics exactly as described in [Sutton and Barto, 1998]. The rewards are -1 for every time step, until the goal is reached. If the goal has not been reached after 1000

[12]It is similar to the CMAC architecture (see Chapter 2), where tilings are displaced with respect to one another by a small random amount.

time steps, the episode is terminated. This task is usually modeled in a deterministic setting. For our experiments, we introduced some noise into the classical version. More specifically, we perturbed the car position on every time step (after applying the original state transition function) by a zero-mean Gaussian noise. The corresponding Gaussian variance was selected so that the resulting task does not become trivial (that is, the agent should not end up in the goal state by chance with a very high probability).

3.5.3. Experimental Results

This section presents results of our experiments with the on-line value-based reinforcement learning algorithm Sarsa (see Chapter 2) on several sets of MDPs, characterized by different values of the State Transition Entropy and Controllability attributes. We consider tabular Sarsa(0) as well as approximate Sarsa(λ) using CMACs to represent action-value functions.

3.5.3.1. Results for Discrete Random MDPs

Test MDPs

This experiment was performed on a set of 65 discrete random MDPs. All of them have two state variables and two actions. Each state variable can take 25 discrete values (625 states total).

For each test MDP, we computed the value of the State Transition Entropy (STE) for each state-action pair and averaged them over all pairs in order to obtain a global attribute value. Similarly, we averaged the values of the Controllability over all states to obtain the global attribute value for each task.

We formed 13 groups of discrete random MDPs with different global values of the State Transition Entropy and the Controllability attributes. Each group contains 5 MDPs. For the purpose of these experiments, we chose MDPs with the attribute values distributed in such a way that we could study the effect of one attribute while keeping the other attribute fixed. The global values of the attributes in the groups of random MDPs tested are shown in Figure 3.7. It is not possible to obtain a

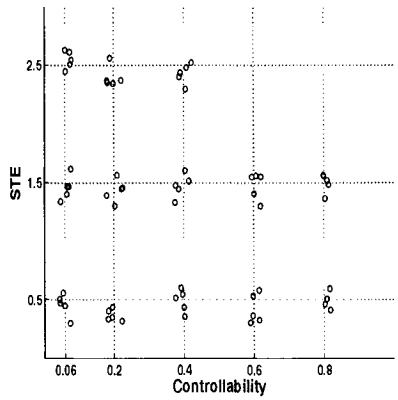


FIGURE 3.7. Global values of the State Transition Entropy and the Controllability attributes for discrete random MDPs in the test set. Each circle corresponds to a combination of attribute values for one MDP.

complete two-way factorial experimental design (where all fixed levels of one attribute are completely crossed with all fixed levels of the other attribute), because the upper bound on the Controllability values is dependent on the values of the State Transition Entropy. Note that each group of random MDPs contains tasks that have similar attribute values, but which were obtained with different parameter settings for the random MDP generator, for example, different bounds on the branching factor. Thus, MDPs within each group are in fact quite different in terms of their state transition structure and rewards. All MDPs are continuing tasks with a discount factor of 0.95.

Experimental Results

The experimental settings are summarized in Table 3.2. In order to evaluate the performance of the Sarsa(0) algorithm, we used two performance measures discussed earlier in this section: the Best Greedy Return measure (BGR, see Equation (3.39)) and the Cumulative Weighted Penalty measure (CWP, see Equation (3.41)). To compute these measures, we used normalized greedy returns, because we aggregated measurements for all MDPs in each group, as will be explained below. Figure 3.8 shows results for the case, in which greedy returns were normalized with respect to the returns of the optimal policy on the test states, as in Equation (3.42). Optimal policies were computed by the Value Iteration algorithm (see Chapter 2) using exact

RL algorithm	Sarsa(0)
Value-function representation	Table
Exploration strategy	ϵ-greedy, constant ϵ
Exploration parameter settings tested	$\epsilon \in \{0.01, 0.1, 0.3, 0.5\}$
Learning rate schedule	Decreasing: $\alpha_N = \frac{\beta - 1}{\beta^{N+1} - 1}$
Learning rate settings tested	$\beta \in \{0.75, 0.9, 0.95, 0.98\}$
Number of learning trials per run	10000
Number of runs	15
Frequency of greedy policy evaluation	Every 100 trials
Number of simulated trajectories for greedy policy evaluation	30
Start state distribution	Uniform over the whole state space
Number of test states	100

TABLE 3.2. Experimental settings for discrete random MDPs.

MDP models. In order to obtain the returns of the optimal policy on the test states, the optimal policy was simulated for 30 trials (1000 time steps each) from each test state and the returns observed were averaged over all trials and test states. Thus, the returns of the optimal policy were estimated in the same way as the returns of the greedy policies obtained during learning with Sarsa(0). The returns of the uniformly random policy, which are used in Equation (3.42), were estimated in the same way.

Experiments on each task were performed using a number of settings for the exploration and learning rate parameters[13], as specified in Table 3.2. Performance measures were first obtained for each task and each combination of the exploration parameter and the learning rate parameter setting tested. We then identified parameter settings that produced best results for each task[14] and used the corresponding performance measurements to produce the graphs in Figure 3.8. Measurements on the graphs represent averages over the 5 tasks in each MDP group and over 15 learning runs for each task.

[13]The ranges of the parameter values tested were selected based on prior experiments. We selected those values, with which the agent was making a reasonable progress.
[14]In these experiments, both the Best Greedy Return measure and the Cumulative Weighted Penalty measure attained best values for the same parameter settings.

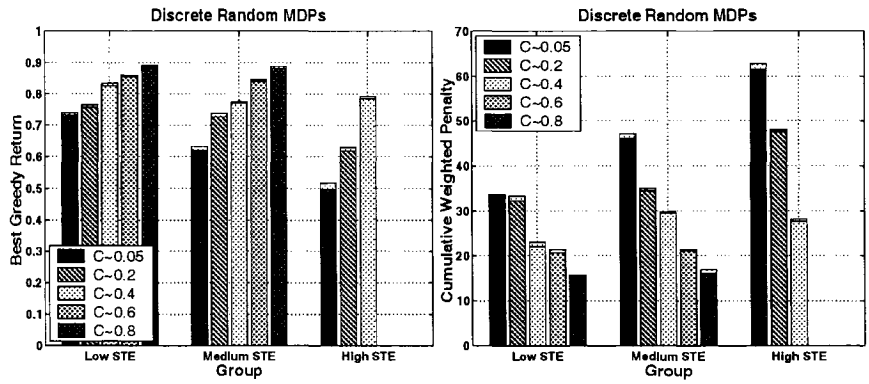


FIGURE 3.8. Discrete random MDPs. Performance measures are computed based on greedy returns normalized with respect to the returns of the optimal policy. Performance measurements represent averages over 5 MDPs in each group and over 15 learning runs for each MDP. Each bar is topped with the standard deviation. Low STE corresponds to values ~ 0.5, medium STE corresponds to values ~ 1.5 and high STE corresponds to values ~ 2.5.

From Figure 3.8, we can see that for each level of the State Transition Entropy tested, the Controllability has an effect on performance. As the Controllability increases, the performance improves, both in terms of the Best Greedy Return measure (see the left panel) and the Cumulative Weighted Penalty measure (see the right panel, where the lower the bars, the smaller the penalties and hence, the better the performance). As discussed in Section 3.3, in MDPs with high Controllability values, the agent has better control over the outcomes of its actions. For highly controllable MDPs, action outcomes differ significantly in terms of their state transitions and potentially in terms of the values of their successor states. Thus, it may be easier for the agent to differentiate between the values of different actions, which can facilitate learning. Also, as previously discussed, exploration can be more efficient in highly controllable MDPs, since taking different exploratory actions leads to different parts of the state space. This can also affect performance in a positive way.

We can also see from Figure 3.8 that an increase in the State Transition Entropy can have a negative effect on performance. As discussed in Section 3.3, high values of the State Transition Entropy can be associated with an increased variance of value samples and thus can make learning more difficult.

From Figure 3.8, we can also observe some interaction effect between the two attributes. In particular, the positive effect of high Controllability on performance appears to be more pronounced as the State Transition Entropy increases. From a different perspective, the negative effect of high State Transition Entropy diminishes as the Controllability increases. High State Transition Entropy makes it even harder for the agent to determine relative goodness of different actions when the Controllability is low. Hence, its negative effect is more pronounced across the low Controllability groups. On the other hand, with high Controllability, the distinctions between different actions are clearer, even if each action is highly stochastic. In this case, high State Transition Entropy can exhibit a positive effect through improved exploration, as we explained in Section 3.3. This, as discussed in Section 3.3, may be particularly beneficial for highly controllable MDPs, where many states should be explored as soon as possible to be able to tease apart differences in the values of different actions.

In Figure 3.9, we provide examples of selected learning curves, where error bars represent standard deviations computed over 15 learning runs. We selected one MDP from three groups of random MDPs with low State Transition Entropy values (see the left panel) and one MDP from the groups with high State Transition Entropy values (see the right panel). We show learning curves for one MDP only in each group (as opposed to averaging over all MDPs in each group) in order to clearly illustrate the learning progress and variance on any given MDP. The graphs are quite similar for different selections of MDPs. On these graphs, we can see the same trends as those reflected on the bar-graphs in Figure 3.8. Note that the variance in the performance decreases as the Controllability increases, whereas the variance increases as the State Transition Entropy increases, as would be expected.

We performed statistical tests to determine statistical significance of the empirical results obtained. In particular, we performed classical two-way analysis of variance (ANOVA) (see, e.g., [Cohen, 1995]). Two-way-ANOVA estimates the statistical significance of the effects of two independent factors on a dependent variable. In our

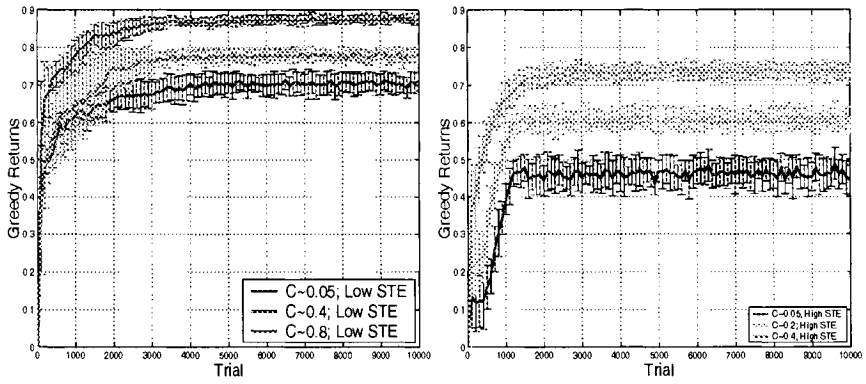


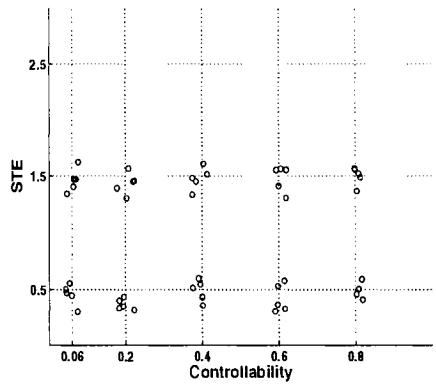
FIGURE 3.9. Selected learning curves, based on greedy returns normalized with respect to the returns of the optimal policy, for discrete random MDPs.

case, the two independent factors are the State Transition Entropy and the Controllability attributes, and the dependent variable is the algorithm's performance. Two-way-ANOVA evaluates the independent effects of the two factors as well as the interactions among the factors that help to explain the variance in the dependent variable.

Classical two-way-ANOVA relies on a two-way factorial design, where all tested levels of one factor are crossed with all tested levels of the other factor (i.e., all combinations are tested). However, recall that we could not create a complete two-way factorial experimental design, because it is impossible to generate MDPs with certain combinations of values of the State Transition Entropy and the Controllability (see Figure 3.7). So we performed two-way-ANOVA on two subsets of MDP groups separately, such that each subset represents a complete factorial experiment. One test was performed for the MDP groups with the attribute values $STE \in \{0.5, 1.5\} \times C \in \{0.05, 0.2, 0.4, 0.6, 0.8\}$. The second test was performed for the groups with the attribute values $STE \in \{0.5, 1.5, 2.5\} \times C \in \{0.05, 0.2, 0.4\}$ (see figures 3.10 and 3.11).

We analyzed the results with respect to the two performance measures considered. Data analyzed by the two-way-ANOVA test has a form of a two-dimensional table, where each cell corresponds to an experimental condition defined by a particular combination of factor values. Each cell contains a number of samples of the

dependent variable in the corresponding experimental condition. In our case, each sample represents an average performance measurement (the Best Greedy Return or the Cumulative Weighted Penalty) for Sarsa(0) on one MDP. Thus, the number of samples in each cell is equal to 5, i.e., the number of MDPs in each group. Our experiment does not contain repeated measures, as each MDP is tested only in one experimental condition. Hence, we perform two-way-ANOVA without repeated measures. The results are summarized in figures 3.10 and 3.11, for two factorial designs respectively.



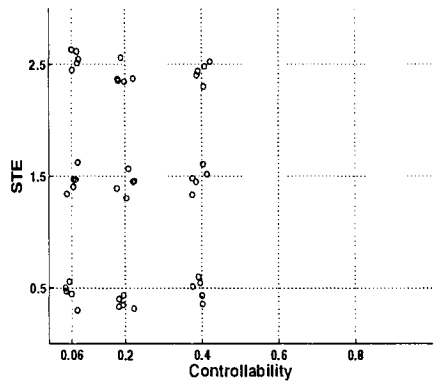
$STE \in \{0.5, 1.5\} \times C \in \{0.05, 0.2, 0.4, 0.6, 0.8\}$; ddf=40

	BGR	CWP
F_{STE} (ndf=1)	44.08	4.09
$1 - p_{STE}$	0.99	0.95
F_C (ndf=4)	15.05	2.54
$1 - p_C$	0.99	0.95
F_{int} (ndf=4)	2.04	2.19
$1 - p_{int}$	0.89	0.91

FIGURE 3.10. Two-way-ANOVA for subset 1 of discrete random MDPs. Performance measures are normalized w.r.t. the optimal policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.

We can see from Figure 3.10 that both the State Transition Entropy and the Controllability have independent effects, which are statistically significant at confidence levels of 0.95 or higher for both performance measures considered. This test does not reveal that the interaction effect between the two attributes is significant. The

second test, summarized in Figure 3.11, also shows significant independent effects of the two attributes (confidence levels are 0.98 and higher). In this test, the effect of the Controllability is even larger, as can be seen from higher values of the corresponding F statistic. We can also see from Figure 3.8 that the effect of the Controllability is the strongest for the highest level of the State Transition Entropy, which is included in this test. This test also shows a significant interaction between the two attributes. Indeed, we could perceive some interaction effect by visually inspecting Figure 3.8, as already discussed.



$STE \in \{0.5, 1.5, 2.5\} \times C \in \{0.05, 0.2, 0.4\}$; ddf=36

	BGR	CWP
F_{STE} (ndf=2)	38.76	5.03
$1 - p_{STE}$	0.99	0.98
F_C (ndf=2)	25.49	9.73
$1 - p_C$	0.99	0.99
F_{int} (ndf=4)	7.85	4.16
$1 - p_{int}$	0.99	0.99

FIGURE 3.11. Two-way-ANOVA for subset 2 of discrete random MDPs. Performance measures are normalized w.r.t. the optimal policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.

We also computed the Hays' statistic (see e.g., [Cohen, 1995]) for our experimental results, which measures the amount of predictive power of an independent factor under investigation. Although we are not trying to build predictive models of performance based on the attribute values, this statistic gives another assessment of the

	BGR	CWP
STE	10%	7%
C	13%	9%
Group	18%	20%

FIGURE 3.12. Predictive power of the attribute values (Hays statistic) for discrete random MDPs. Performance measures are normalized w.r.t. the optimal policy.

strength of the statistical relationship between the independent and dependent variables. As discussed in [Cohen, 1995], values of this statistic that are greater than 10 indicate practical usability of the considered independent factors for building predictive models. In our case, we measured the predictive power of the MDP group, which corresponds to a particular combination of the two attribute values. As in the case of two-way-ANOVA, each data sample represents an average performance measurement on one MDP in the considered group. We also measured the predictive power of the State Transition Entropy and the Controllability attributes separately. The results are summarized in Figure 3.12. We can see that the combination of the two attribute values (predictive power of the MDP group) indicates potential practical usefulness of the attribute values. The predictive power of each attribute in isolation appears to be lower, which is consistent with the fact that two-way-ANOVA detected a significant interaction effect between the two attributes.

We also performed a similar analysis for the performance measures obtained based on greedy returns normalized with respect to the performance of the uniformly random policy. The objective was to verify whether performance measures based on such a normalization would lead to similar conclusions. If the results are similar for the normalizations with respect to the random and optimal policies, then we can use random policy normalization in our further experiments with continuous state space MDPs, for which we cannot compute optimal policies by independent means, such as dynamic programming.

The returns of the uniformly random policy on the test start states were estimated in the same manner as described earlier. Each performance measurement (the Best

Greedy Return and the Cumulative Weighted Penalty) was then computed using ratios of greedy returns $R_{t,k}$ (see Equation (3.37)) to the estimated returns of the uniformly random policy on the corresponding task:

$$\tilde{R}_{t,k} = \frac{R_{t,k}}{R^{rand}} \ , \ k \in \{1, ..., K\}, \ t \in \{0, G, 2G, ..., lG\} \tag{3.43}$$

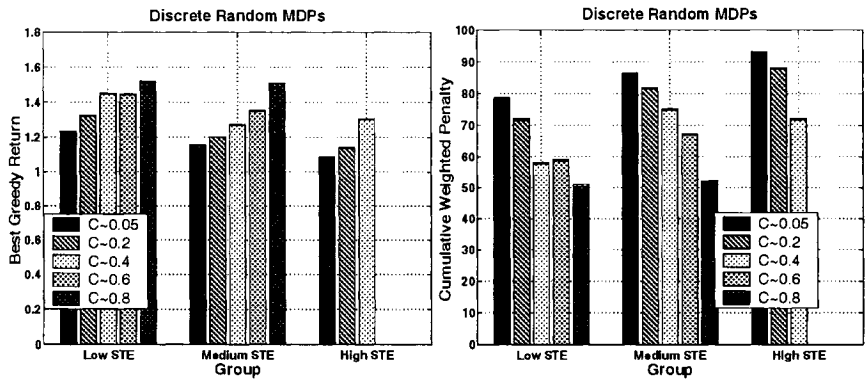
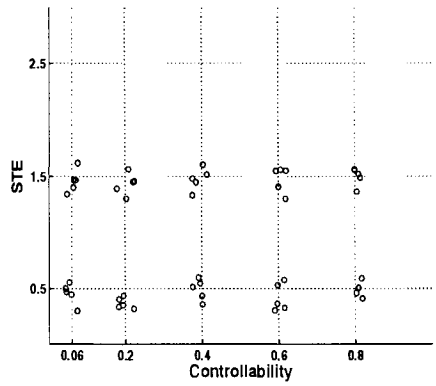


FIGURE 3.13. Discrete random MDPs. Performance measures are computed based on greedy returns normalized with respect to returns of the uniformly random policy. Performance measurements represent averages over 5 MDPs in each group and over 15 learning runs for each MDP. Each bar is topped with the standard deviation.

Figure 3.13 shows experimental results based on this normalization. The graphs were produced using performance measurements for the same optimal settings of the exploration and the learning rate parameters as before. We can see that the results are very similar qualitatively. Most trends that were previously observed are preserved in these results: performance improves as the Controllability increases and degrades as the State Transition Entropy increases. As before, a negative effect of the State Transition Entropy diminishes as the Controllability increases.

We performed statistical tests in the same manner as before. They are summarized in figures 3.14 and 3.15. The results show that the independent effects of the two attributes are statistically significant. The effect of the Controllability appears more significant with this analysis. This can be expected, because the Controllability measures differences in the state transitions of different actions compared to the uniform action selection (which is a baseline policy in this analysis). The interaction effect



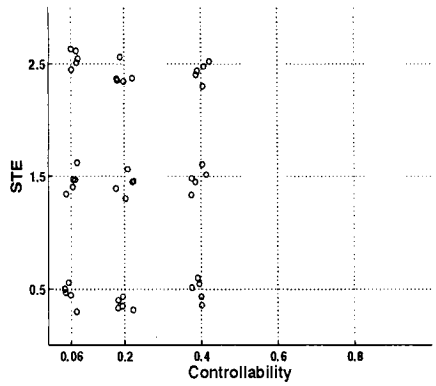
$STE \in \{0.5, 1.5\} \times C \in \{0.05, 0.2, 0.4, 0.6, 0.8\}$; ddf=40

	BGR	CWP
F_{STE} (ndf=1)	11.74	12.18
$1 - p_{STE}$	0.99	0.99
F_C (ndf=4)	15.78	18.62
$1 - p_C$	0.99	0.99
F_{int} (ndf=4)	0.96	1.02
$1 - p_{int}$	0.56	0.59

FIGURE 3.14. Two-way-ANOVA for subset 1 of discrete random MDPs. Performance measures are normalized w.r.t. the uniformly random policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.

between the two attributes is not as pronounced with these measurements compared to the case of normalization with respect to the performance of optimal policies.

We also estimated the predictive power of the attribute values in this case, similar as explained above. The results are summarized in Figure 3.16. We see similar trends here as well: the MDP group (combination of the two attribute values) has the most predictive power and each attribute has less predictive power in isolation.



$STE \in \{0.5, 1.5, 2.5\} \times C \in \{0.05, 0.2, 0.4\}$; ddf=36

	BGR	CWP
F_{STE} (ndf=2)	16.30	19.11
$1 - p_{STE}$	0.99	0.99
F_C (ndf=2)	20.31	25.78
$1 - p_C$	0.99	0.99
F_{int} (ndf=4)	0.77	0.98
$1 - p_{int}$	0.44	0.56

FIGURE 3.15. Two-way-ANOVA for subset 2 of discrete random MDPs. Performance measures are normalized w.r.t. the uniformly random policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.

	BGR	CWP
STE	6%	7%
C	10%	10%
Group	24%	25%

FIGURE 3.16. Predictive power of the attribute values (Hays statistic) for discrete random MDPs. Performance measures are normalized w.r.t. the uniformly random policy.

3.5.3.2. Results on Continuous Random MDPs

In this set of experiments, our objective was to investigate whether the effects of the State Transition Entropy and the Controllability attributes would be similar in the case of the approximate Sarsa algorithm compared to tabular Sarsa, which we studied in our experiments with discrete random MDPs. So in these experiments, we use MDPs with continuous state spaces and we use CMAC function approximators (see Chapter 2) to represent action-value functions.

Test MDPs

We tested 45 continuous random MDPs. All of them have two state variables and two actions. Each state variable takes values from the interval $[0, 1]$. The piecewise constant functions $P_j(s, a)$, $M_j^i(s, a)$, $V_j^i(s, a)$, $R(s, a, s')$ and $V(s, a, s')$, used to specify the dynamics of random MDPs as discussed above, were generated using discretizations of size 25×25, where the grid origin for each function was displaced by a random amount $\Delta < 0.03$ in each state dimension. For each state variable, we imposed an upper bound of 0.1 on the absolute difference between the value of the corresponding state variable in the current state and its value in the next state, in order to prevent big erratic jumps across the state space.

For each task, attribute values were estimated using a 10×10 discretization of the state space, as described in Section 3.4. We sampled uniformly randomly 100 states (independently of the discretization), and for each of the sampled states, we collected 500 samples of state transitions, which provided counts of transitions to each of the discrete bins. Values of the State Transition Entropy and the Controllability were estimated for each of the sample states (and each action, in the case of the State Transition Entropy) and then averaged to obtain global attribute values for each MDP[15].

[15]Attribute values were also estimated for a sample of states obtained using the state distribution generated by following a uniformly random policy instead of the uniform state sampling. For our random MDPs, global attribute values were very similar under the two distributions.

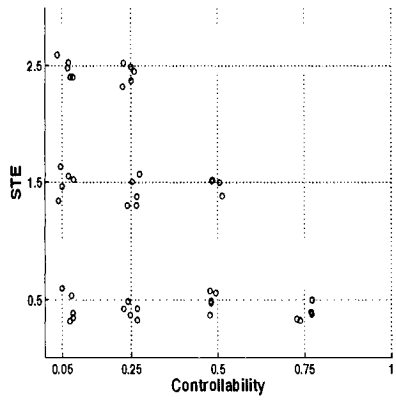


FIGURE 3.17. Global values of the attributes for continuous random MDPs in the test set. Each circle represents a combination of the attribute values of one MDP.

We studied 9 groups of continuous random MDPs, where each group contained 5 tasks. Global values of the attributes in the considered MDP groups are shown in Figure 3.17. As in the case of the experiments with discrete random MDPs, it is not possible to obtain a complete two-way factorial experimental design. Like in the case of discrete random MDPs, continuous random MDPs in each group have similar attribute values but are quite different in terms of their state transitions and rewards. This could be achieved by using different settings for the random MDP generator, such as different combinations of bounds on the state transition and reward variance and the branching factor. All MDPs are continuing tasks with a discount factor of 0.95.

Experimental Results

The experimental settings are summarized in Table 3.3. The performance measurements that are shown on the figures below were obtained based on greedy returns normalized by the returns of the uniformly random policy. Normalization was done in the same manner as in the case of discrete random MDPs. For the continuous random MDPs, we are not able to obtain independent estimates of the optimal policy, and hence we are not able to do normalization with respect to the returns of the optimal policy. As in the previous experiments, we tested a number of settings of

RL algorithm	Sarsa(0)
Value-function representation	CMACs of size 500
Exploration strategy	ϵ-greedy, constant ϵ
Exploration parameter settings tested	$\epsilon \in \{0.01, 0.1, 0.3, 0.5\}$
Learning rate schedule	Constant
Learning rate settings tested	$\beta \in \{0.025, 0.05, 0.1, 0.15\}$
Number of learning trials per run	10000
Number of runs	15
Frequency of greedy policy evaluations	Every 100 trials
Number of simulated trajectories for greedy policy evaluation	30
Start state distribution	Uniform over the whole state space
Number of test states	100

TABLE 3.3. Experimental settings for continuous random MDPs.

the exploration and learning rate parameters for each task. The figures below were produced using the best parameter settings for each task, as described earlier in the case of discrete random MDPs.

Figure 3.18 shows the results for the experiments with continuous random MDPs. The graphs show the same performance trends with respect to the attribute values as in the case of discrete random MDPs. Thus the results appear to be consistent for both tabular and approximate Sarsa using linear function approximation architecture CMAC. We experimented with different sizes of function approximators and obtained qualitatively similar results in all cases. The only differences observed were that the coarser the function approximator, the less of a positive effect is observed with an increase of the Controllability values, especially for small values of this attribute. This would be expected, because a coarse function approximator (that aggregates large areas of neighboring states) is not capable of discriminating small differences in state transitions.

For the results obtained in these experiments, we performed statistical tests in a similar manner as for the discrete random MDPs. As before, we performed two separate two-way-ANOVA tests for two subsets of MDP groups: one test for $STE \in$

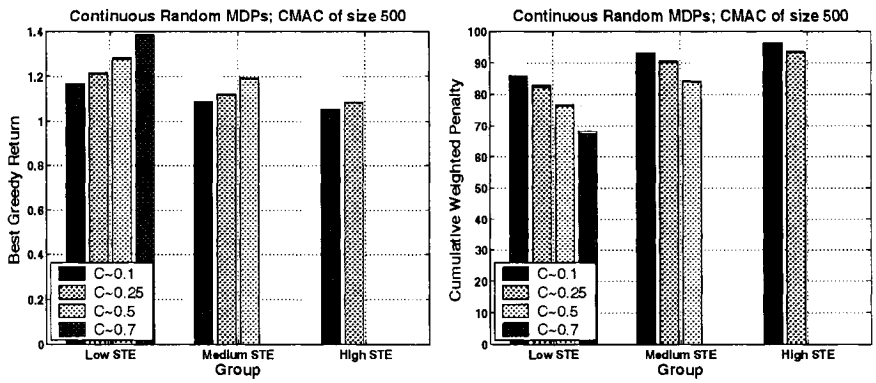
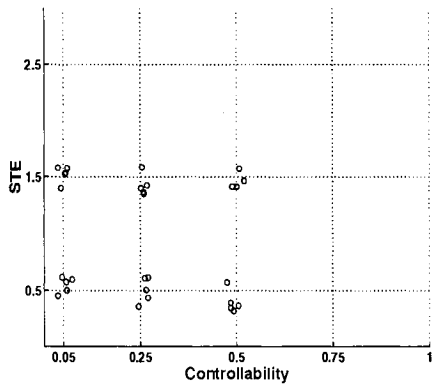


FIGURE 3.18. Performance of the Sarsa(0) algorithm for continuous random MDPs. The action-value functions are represented by CMACs of size 500. Performance measures are computed based on greedy returns normalized with respect to returns of the uniformly random policy. Performance measurements represent averages over 5 MDPs in each group and over 15 learning runs for each MDP. Each bar is topped with the standard deviation.

$\{0.5, 1.5\} \times C \in \{0.05, 0.25, 0.5\}$ and the other test for $STE \in \{0.5, 1.5, 2.5\} \times C \in \{0.05, 0.25\}$. The results are summarized in figures 3.19 and 3.20. The statistical results are similar to those for discrete random MDPs with performance measures normalized with respect to returns of the uniformly random policy. The tests show statistically significant independent effects of the two attributes but no interaction effect.

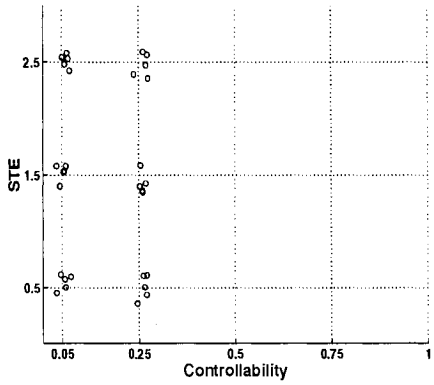
We also estimated the predictive power of the attribute values, as explained before. The results are shown in Figure 3.21, indicating a significant statistical relationship between the combination of the attribute values and the performance of approximate Sarsa.



$STE \in \{0.5, 1.5\} \times C \in \{0.05, 0.25, 0.5\}$; ddf=24

	BGR	CWP
F_{STE} (ndf=1)	41.01	40.65
$1 - p_{STE}$	0.99	0.99
F_C (ndf=4)	22.37	9.53
$1 - p_C$	0.99	0.99
F_{int} (ndf=4)	0.06	0.05
$1 - p_{int}$	0.05	0.04

FIGURE 3.19. Two-way-ANOVA for subset 1 of continuous random MDPs. Performance measures are normalized w.r.t. the uniformly random policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.



$STE \in \{0.5, 1.5, 2.5\} \times C \in \{0.05, 0.25\}$; ddf=24

	BGR	CWP
F_{STE} (ndf=2)	31.41	31.97
$1 - p_{STE}$	0.99	0.99
F_C (ndf=2)	7.79	4.91
$1 - p_C$	0.99	0.99
F_{int} (ndf=4)	0.08	0.07
$1 - p_{int}$	0.08	0.06

FIGURE 3.20. Two-way-ANOVA for subset 2 of continuous random MDPs. Performance measures are normalized w.r.t. the uniformly random policy. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts STE and C refer to the effects of the State Transition Entropy and the Controllability. Subscript int refers to the interaction effect between the two factors.

	BGR	CWP
STE	8%	7%
C	12%	10%
Group	20%	21%

FIGURE 3.21. Predictive power of the attribute values (Hays statistic) for continuous random MDPs.

3.5.3.3. Results on the Mountain Car Domain

The objective of this set of experiments was to verify the presence of the effects of the attribute values in a domain that is well understood and widely used in the reinforcement learning community. The Mountain-Car domain, as previously mentioned, is a classical reinforcement learning benchmark.

Test MDPs

As explained before, stochastic variants of the Mountain-Car task have some variance in the state transition function for the state variable corresponding to the car position. We considered four tasks characterized by the following values of the position variance: $\{0, 0.00009, 0.00038, 0.0008\}$ (the first variant is the classical deterministic task). Global values of the State Transition Entropy and Controllability for these tasks are shown in Figure 3.22. Note that for stochastic Mountain-Car tasks, values of the two attributes are anti-correlated. The attribute values were computed in a similar manner as in the case of continuous random MDPs, as described above. Note that the range of values of the State Transition Entropy is much smaller in this experiment, since adding more stochasticity to state transitions made the task virtually uncontrollable but trivial (the agent ended up in the goal state with a very high probability simply by chance).

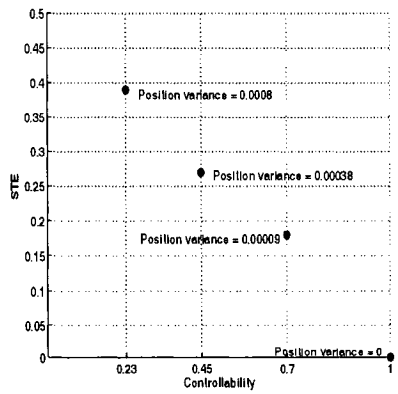


FIGURE 3.22. Global values of the attributes for stochastic Mountain-Car tasks. Each circle corresponds to the attribute values for one variant tested.

Experimental Results

The experimental settings are summarized in Table 3.4. Note that in this experiment, we also tested several settings of the eligibility trace parameter λ (see Chapter 2).

RL algorithm	Sarsa(λ)
Eligibility trace parameter settings	$\lambda \in \{0, 0.3, 0.5, 0.7, 0.9, 0.99\}$
Value-function representation	CMACs of size 500.
Exploration strategy	ϵ-greedy, constant ϵ
Exploration parameter settings tested	$\epsilon \in \{0.01, 0.1, 0.3\}$
Learning rate schedule	Constant
Learning rate settings tested	$\beta \in \{0.05, 0.1, 0.15, 0.2\}$
Number of learning trials per run	10000
Number of runs	20
Frequency of greedy policy evaluations	Every 100 trials
Number of simulated trajectories for greedy policy evaluation	30
Start state distribution	Uniform over the entire state space.
Number of test states	50

TABLE 3.4. Experimental settings for the Mountain-Car domain.

The experimental results for the Sarsa(0) algorithm are shown on Figure 3.23. Performance measurements were obtained by normalizing greedy returns with respect to the performance of the uniformly random policy on the corresponding variant and on the set of starting test states. Since all rewards in this domain are negative, normalization was performed in the following way:

$$\bar{R}_{t,k} = \frac{R_{rand}}{R_{t,k}},\ k \in \{1, ...20\},\ t \in \{0, G, 2G, ..., lG\} \tag{3.44}$$

As before, graphs were obtained with the performance measurements corresponding to the best settings of the exploration parameter and the learning rate parameter.

We can see from Figure 3.23 that performance deteriorates as the stochasticity of the task increases. The performance change is monotonic with respect to the Best Greedy Return measure. For the Cumulative Weighted Penalty measure, a

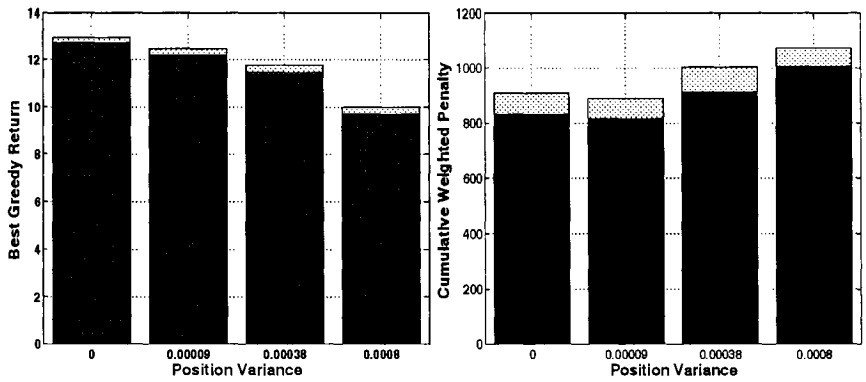


FIGURE 3.23. Performance of the Sarsa(0) algorithm on deterministic and stochastic variants of the Mountain-Car domain. Larger values of the position variance correspond to lower Controllability and higher State Transition Entropy.

slight increase in the task stochasticity does not lead to an immediate increase of the penalties, possibly because of an improved learning speed, as suggested by the learning curves in Figure 3.24. However, as the stochasticity increases further, the performance deteriorates with respect to both measures. Thus, the experiments on the Mountain-Car domain also provide evidence for the existence of an effect of domain stochasticity on the performance of the Sarsa algorithm.

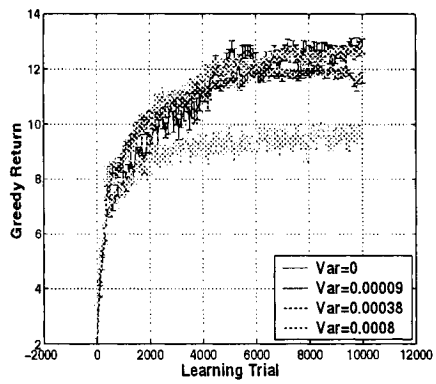


FIGURE 3.24. Learning curves of the Sarsa(0) algorithm on deterministic and stochastic variants of the Mountain-Car domain.

We performed classical one-way-ANOVA (see e.g., [Cohen, 1995]) for the results of these experiments. This test estimates statistical significance of the effect of one independent factor on a dependent variable. In our case, the independent factor

	BGR	CWP
F (ndf=3, ddf=76)	5.7313	3.1898
1-p	0.99	0.97
Pred. power	12%	9.5%

FIGURE 3.25. One-way ANOVA and predictive power of the stochasticity level for the Mountain-Car domain.

is the amount of stochasticity of the task, for which four levels were tested. Each cell in a one-dimensional data table analyzed by this test contains samples of the performance measurements for the Sarsa(0) algorithm. Each sample corresponds to a measurement for one run of the algorithm on the corresponding variant of the task. Thus, there were 20 samples for each level of the independent factor. The results of this test are presented in Figure 3.25. We can see that the observed effects are statistically significant at the confidence levels of 0.99 and 0.97 for the Best Greedy Return and the Cumulative Weighted Penalty measures respectively. The predictive power of the stochasticity level of the task also indicates a statistical relationship between the stochasticity level and the algorithm's performance.

Experimental results for the performance of the Sarsa(λ) algorithm (using eligibility traces) are shown on Figure 3.26. With respect to the Best Greedy Return measure, higher values of λ lead to better solutions for all stochastic variants. However, the Cumulative Weighted Penalty measure increases with $\lambda = 0.99$ for two variants with higher stochasticity, which indicates a decrease in performance. As suggested by the learning curves in Figure 3.27, for variants with high stochasticity, the variance in the performance measure increases when λ increases (see the right panel). On the other hand, we do not observe such a phenomenon for the deterministic version of the domain, as can be seen from the left panel of Figure 3.27. This trend is consistent with the theoretical findings [Kearns and Singh, 2000], which demonstrated that learning with higher values of λ is associated with a lower bias but a higher variance[16]. In the case of highly stochastic domains, the variance can be even more significant, as we see from these experiments.

[16]This results was obtained for TD(λ).

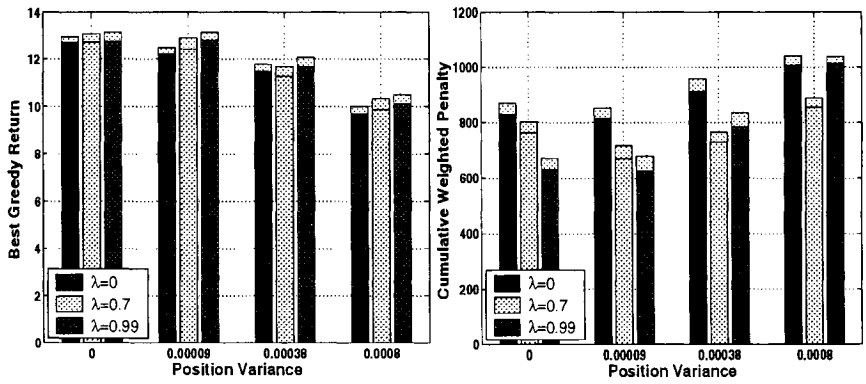


FIGURE 3.26. Performance of the Sarsa(λ) algorithm on deterministic and stochastic variants of the Mountain-Car domain.

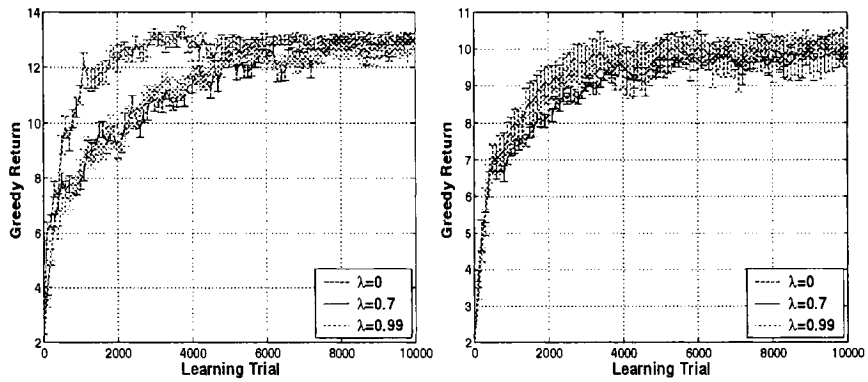


FIGURE 3.27. Selected learning curves for the Sarsa(λ) algorithm on the Mountain-Car domain. The left panel shows the performance on the deterministic variant of the domain and the right panel shows the performance on the stochastic variant with the position variance equal to 0.0008.

We also performed similar experiments with the approximate Q-learning algorithm (see Chapter 2) on the stochastic Mountain Car domain. The results obtained were very similar to those of the Sarsa algorithm.

3.6. Summary

In this chapter, we discussed the potential usefulness of studying certain properties of Markov Decision Processes. We presented five quantitative MDP attributes: the State Transition Entropy, Variance of Immediate Rewards, Controllability, Reward Information Content and Risk Factor. These attributes measure MDP properties related mostly to the amount of stochasticity in the environment and the amount of control that the agent has over the environment. We discussed how they can affect the difficulty of learning with on-line value-based algorithms, mainly with respect to the sample variance and exploration. We also discussed the implications of such MDP characteristics as the amount of control and the risk tolerance. We suggested various ways, in which knowledge of the MDP attributes can be used for improving learning efficiency. The attributes complement each other and thus could be most helpful when used in combinations.

Our goal was to identify attributes that rely on information pertaining only to the model of the underlying MDP and not to the value estimates. This way, the estimation of attribute values is easier, since we need to estimate stationary properties only. They can be computed either prior to learning or during learning, but without being affected by dynamic changes in the value estimates. Such attributes can provide useful information, which is independent of the value estimation process. We discussed how the attributes can be computed, both off-line and on-line.

We presented results of an experimental study that provided empirical evidence for our hypotheses regarding the effects of the State Transition Entropy and Controllability attributes on the performance of on-line value-based reinforcement learning. The results obtained showed a consistent pattern across tabular and approximate reinforcement learning algorithms. These experimental results warranted our further research aimed at developing a practical application of these two attributes for improving exploration, which will be presented in the next chapter. A further study and practical application of the other attributes will be done in future work.

CHAPTER 4

Using MDP Attributes for Exploration in Reinforcement Learning

Chapter Outline

In this chapter, we present a new approach for efficient exploration in reinforcement learning. Our technique relies on information about certain properties of MDPs as computed by two attributes studied in the previous chapters, namely the State Transition Entropy and the Forward Controllability. Our approach can be used in combination with other existing exploration techniques, and we experimentally demonstrate that it can improve their performance. In contrast to most other existing methods, the exploration-relevant information used by our approach can be precomputed beforehand and then used during learning without additional computation cost as well as transferred between similar tasks.

4.1. Introduction

As previously discussed, one of the key features of reinforcement learning is that a learning agent is not instructed what actions it should perform. This creates a need for the agent to actively explore its environment in order to discover good behavior strategies. Ensuring an efficient exploration process and balancing the risk of taking exploratory actions with the benefit of information gathering are of great practical importance for reinforcement learning agents, and have been the topic of

much recent research, e.g., [Thrun, 1992; Kaelbling, 1993; Meuleau and Bourgine, 1999; Dearden *et al.*, 1999; Kearns and Singh, 1998; Sutton, 1990; Wiering, 1999; Wyatt, 2001].

In general, exploration research addresses two main issues: (1) balancing the exploration-exploitation trade-off, i.e., maximizing the agent's performance during learning; (2) exploring the environment in a way that allows finding a good policy given a limited amount of training time. This chapter is devoted to the second issue.

Existing exploration strategies can be divided into two broad classes: undirected and directed methods. *Undirected methods* are concerned only with ensuring *sufficient* exploration, by selecting all actions infinitely often. The ϵ-greedy and Boltzman exploration strategies are notable examples of such methods (see Chapter 2). Undirected strategies are concerned only with action selection frequencies and do not address the issue of efficient state space exploration. These techniques are attractive because of their simplicity and because they have no additional requirements of storage or computation. In fact, they are the most commonly used in practice. However, the potential inefficiency of the undirected approaches is intuitive and can be proved for certain types of tasks. For example, in [Whitehead, 1991; Thrun, 1992], it is shown that in a class of deterministic goal-directed tasks with a positive reward received only upon entering a goal state, undirected learning is exponential in the number of steps that the agent with an optimal policy requires in order to reach a goal state. On the other hand, with a simple form of directed exploration (using some information about the course of learning), the same deterministic tasks can be learned in time polynomial in the number of states and the maximum number of actions available at each state. The impact is believed to be very important for stochastic tasks as well [Thrun, 1992; Kearns and Singh, 1998; Fiechter, 1997].

Directed exploration strategies, on the other hand, attempt not only to ensure a sufficient amount of exploration, but also to make exploration *efficient*. These techniques often aim to achieve a more uniform exploration of the state space, or to

balance the relative profit of discovering new information versus exploiting current knowledge. Typically, directed approaches keep track of certain information about the course of the learning process. Often, they also rely on the use of an MDP model, which can be estimated during learning if not known a priori. This typically requires extra computation and storage in addition to the resources needed by general reinforcement learning algorithms. However, these additional requirements can be well justified by the benefits of faster learning, better solutions found and a better balancing of the exploration-exploitation trade-off.

The main contribution of this chapter is a new directed exploration approach, which takes into account certain properties of the Markov Decision Process being solved. In Chapter 3, we introduced several attributes that can be used to provide a quantitative characterization of MDPs. Our approach to exploration is based on the use of two attributes: the State Transition Entropy and Forward Controllability. Recall from the previous chapters that the State Transition Entropy measures the amount of stochasticity in the environment while the Forward Controllability reflects how much the agent's actions impact the trajectories that the agent follows. Experimental results presented in the previous chapter showed that these attributes can significantly affect the quality of learning with on-line reinforcement learning algorithms. In this chapter, we show how to use these MDP attributes to guide exploration in order to improve the chances of finding a good policy as quickly as possible. Here, we focus on a design of the attribute-based exploration strategies for finite MDPs and tabular representations of the action-value functions. Possibilities for extending our approach to approximate representations of the action-value functions are discussed in the future work section. Our approach can be combined with existing exploration methods, both undirected and directed, and we empirically demonstrate that it can boost their performance.

Using MDP attributes can improve the exploration process in three ways. First, it encourages a more homogeneous visitation of the state space, similar to other existing directed methods. Second, it encourages more frequent sampling for actions with

potentially high variance in their action-value estimates. Finally, it encourages the learning agent to focus more on the states in which its actions have more impact on the course of events. One important difference between our exploration strategy and other directed techniques is that the exploration-related information we use reflects only properties of the task at hand, and does not depend on the history of learning. Hence, this information does not carry the bias of previous, possibly unfortunate exploration decisions. Additionally, in some cases, the MDP attributes can be pre-computed beforehand and then used during learning without any additional computational cost. The attributes' values can also be transferred between tasks if the agent is faced with solving multiple related tasks in an environment in which the dynamics does not change much. As discussed in Chapter 3, the attributes can also be estimated during learning, which requires only a small constant amount of additional resources. This is in contrast to most other directed methods, which typically tend to involve a significant amount of extra computation and memory.

The rest of the chapter is organized as follows. In Section 4.2, we provide an overview of the existing exploration approaches. The details of the proposed attribute-based exploration method are presented in Section 4.3. Empirical results are discussed in Section 4.4. Conclusions and directions for future work are presented in Section 4.5.

4.2. Overview of Exploration Approaches

As previously mentioned, the ultimate goal of an exploration policy is to allow the reinforcement learning agent to gather experience with the environment in such a way as to find an optimal policy as quickly as possible, while also gathering as much reward as possible during learning. This goal can be itself cast as a learning problem, often called *optimal learning* [Kumar, 1985]. Solving this problem would require the agent to have a probabilistic model of the uncertainty about its own knowledge of the environment, and to update this model as learning progresses. Solving the optimal learning problem then becomes equivalent to solving a partially observable MDP

(POMDP) defined by this model of uncertainty, which is generally intractable [Sondik, 1971; Littman, 1996]. However, various heuristics can be used to obtain a good exploration policy based only on estimates of certain aspects of the uncertainty about the agent's knowledge of the environment. As explained above, existing exploration strategies can be divided into two broad classes: *undirected* and *directed*. We will discuss them next.

4.2.1. Undirected Methods

In undirected methods, exploration is driven only by randomness, which is incorporated into the agent's behavior policy to ensure that each action will be selected with non-zero probability in each visited state. An example of undirected techniques is the ϵ-greedy exploration strategy. As discussed in Chapter 2, in every state, the agent selects a greedy action with probability $(1-\epsilon)$ (performs exploitation) and uniformly randomly selects any action with probability ϵ (performs exploration). Another representative of undirected exploration methods relies on the Boltzman distribution and is more value-driven. With this strategy, the probability $\pi(s,a)$ of taking action a in state s is computed as follows:

$$\pi(s,a) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{b \in A(s)} e^{\frac{Q(s,b)}{\tau}}} \tag{4.1}$$

where $Q(s,a)$ are action-values and τ is a positive temperature parameter that decreases the amount of randomness as it approaches zero.

The behavior of these undirected methods was analyzed theoretically in [Singh *et al.*, 2000], where it was shown that the on-policy reinforcement learning algorithm Sarsa(0) with tabular action-value function representation asymptotically converges to an optimal deterministic policy when the exploration rate (ϵ in the ϵ-greedy policy and τ in the Boltzman policy) decreases with time to zero in an appropriate manner.

In [Thrun, 1992], a comparison between the ϵ-greedy, uniform ($\epsilon = 1$) and Boltzman strategies was performed on a robot navigation goal-directed task. The results

showed that the best policies were found with the uniform exploration, demonstrating the necessity of a very extensive exploration for reinforcement learning. However ϵ-greedy strategies ($\epsilon < 1$) reduced significantly learning costs (i.e., reward loss due to the selection of suboptimal actions), allowing for exploitation of the acquired knowledge during learning. The Boltzman strategy demonstrated the worst performance both in terms of the found solution policies and learning costs[1]. Since action selection probabilities based on the Boltzman distribution depend on the actual values of all actions and not only on which action is best relative to others (as is the case with the ϵ-greedy strategy), inaccurate estimates of the action-value function bias the behavior towards wrong exploration and exploitation decisions much stronger than in the case of the ϵ-greedy strategy. On the other hand, the much coarser information about the value functions used by the ϵ-greedy method seems to be sufficient for finding good policies.

4.2.2. Directed Methods

Directed techniques attempt to render exploration more efficient. To do that, they use additional information about the learning process. These techniques are often concerned with a more uniform exploration of the state space and/or balancing the relative profits of discovering new information as opposed to exploiting current knowledge. Most directed techniques assume the availability of an MDP model or learn this model in parallel with the underlying reinforcement learning algorithm. Typically, they use an *evaluation function* that combines action-values and some kind of *exploration bonuses* in an additive manner:

$$N(s,a) = K_0 Q(s,a) + K_1 \delta_1(s,a) + ... + K_k \delta_k(s,a) \tag{4.2}$$

where each $\delta_i(s,a)$, $i = 1, ..., k$, is an exploration bonus that can be defined in various ways. Usually the action selection rule is deterministic: the action that attains the maximum of the evaluation function (4.2) is chosen on every step. In this case,

[1]Our own past experiments with the ϵ-greedy and Boltzman strategies, performed on random discrete MDPs with the Sarsa(0) algorithm produced similar results.

exploration is driven by the exploration bonuses that change over time with learning. The positive parameter K_0 indicates the desired balance of exploitation and exploration. The contribution of each of the exploration bonuses may also be weighted by parameters K_i, $i = 1, ..., k$. The evaluation function (4.2) is sensitive to the relative magnitudes of the action-values $Q(s, a)$ and the exploration bonuses. Thus, the corresponding weighting parameters K_0 and K_i have to be tuned to a particular tasks at hand. There are a number of techniques that define exploration bonuses in various ways, as will be discussed in the rest of this section.

In the *counter-based exploration strategy* [Sato *et al.*, 1990; Barto and Singh, 1990], the exploration bonus is based on the number of visitations to each state, $n(s)$, and is defined as $\delta(s, a) = \frac{n(s)}{E[n|s,a]}$, where $E[n|s, a] = \sum_{s' \in S} P^a_{ss'} n(s')$, i.e., the expected number of visits performed in the past to the successor states of the action a and the current state s. This strategy favors actions that lead to the least visited states, thus encouraging a more homogeneous state space exploration.

Recall that reinforcement learning methods are iterative in nature and gradually improve the estimates of the value functions. A heuristic based on the *recency of state visitations* accounts for the fact that value estimates may become outdated if not updated regularly. The recency information can be combined with the counter-based method by decaying counters on each time step with some fixed factor $\beta < 1$ [Thrun, 1992]. If two states have equal visitation counts, the one that was visited a longer time ago will be favored. Another way of using the recency information was introduced by Sutton in [Sutton, 1990], where the exploration bonus was defined as $\delta(s, a) = \sqrt{E[\rho|s, a]}$, where $\rho(s)$ is the number of actions performed since the last occurrence of the state s and $E[\rho|s, a] = \sum_{s' \in S} P^a_{ss'} \rho(s')$ is the expected value of the recency of the successor states of the action a and the state s. Recency-based exploration bonuses will reflect the accuracy or "up-to-date" status of the corresponding values in the states. This strategy also aims at a homogeneous exploration of the state space.

The *error-based exploration strategy* (or otherwise known as Prioritized Sweeping in the context of asynchronous dynamic programming) [Schmidhuber, 1991; Moore

and Atkeson, 1993] memorizes the change of the value estimates on the last value update in each state: $\Delta(s) = |Q(s,a) - [r^a_{ss'} + \gamma Q(s',a')]|$. The action with the largest expected change of the value in the next state can induce the largest change in the value of the current state. Thus, the exploration bonus is defined as the expected value of the last change for the next states: $\delta(s,a) = E[\Delta|s,a] = \sum_{s' \in S} P^a_{ss'} \Delta(s')$.

Experiments in [Thrun, 1992] with the robot navigation goal-directed task demonstrated a big advantage of directed techniques over undirected ones[2]. The best performance, both in terms of the solution quality and the learning costs, was demonstrated by the recency-based method, closely followed by the counter-based method with decay. The experiments showed that both of these methods were the slowest in obtaining accurate action-values, but still the fastest in finding good solutions and reducing the cost of learning. The sluggishness of the recency-based methods in improving the accuracy of the value functions can be due to the fact that values are updated using the most outdated value estimates in the next states.

Another approach for directed exploration is based on reasoning about the *uncertainty of the values that are being learned*, be it action-values, the model of the MDP or both. As previously mentioned, maintaining and using *exact* models of such uncertainty is computationally intractable, but there were several attempts to use approximations to such models or heuristics based on partial estimates of such uncertainty.

The method of Interval Estimation (IE) [Kaelbling, 1993] is a representative of such approaches. This strategy always chooses the action that maximizes the upper bound of the $100(1-\theta)\%$ confidence intervals of action-values, $Q(s,a)$, for some confidence coefficient $\theta \in (0,1)$. It is based on the assumption that the random variables $Q(s,a)$, samples of which we obtain from the agent's interaction with the environment, follow normal distributions with unknown means $\overline{Q(s,a)}$ and variances

[2]The experiments in [Thrun, 1992] were performed on one maze-type goal directed task, which is a type of MDPs that is expected to benefit most from the directed techniques discussed above. It would be interesting to have the comparisons of exploration techniques on a more versatile set of MDPs, however, most of the existing empirical work in exploration research concentrated on maze-like tasks.

$\sigma^2_{Q(s,a)}$. Then the upper bound of the confidence interval is

$$U_{Q(s,a)} = \overline{Q(s,a)} + \sigma_{Q(s,a)} \frac{z_{\theta/2}}{\sqrt{n(s,a)}} \tag{4.3}$$

where $n(s,a)$ is the number of visitis (samples) to the state-action pair (s,a) and $z_{\theta/2}$ is the value of Z statistic at $\theta/2$ confidence level (see, e.g., [Cohen, 1995]). In this case, action values $Q(s,a)$ are updated as usual by a chosen reinforcement learning algorithm, while at the same time, sample values of targets, e.g., $[r^a_{ss'} + \gamma Q(s',a')]$ with the Sarsa algorithm, are also used to compute the sample means $\overline{Q(s,a)}$. If the variances $\sigma^2_{Q(s,a)}$ are not known (which is typically the case), the sample standard deviation is also calculated from sample targets and the interval bounds are obtained with the values of the Student's t-function at the confidence level $\theta/2$ and with $(n(s,a)-1)$ degrees of freedom. The value $\sigma_{Q(s,a)} \frac{z_{\theta/2}}{\sqrt{(n(s,a)}}$ can be regarded as an exploration bonus used in the evaluation function (4.2). It represents a local measure of uncertainty about the estimated action-values. It can also be interpreted as an optimistic action selection, where the agent selects an action that still looks very promising with respect to the current upper bound on its value. The distribution of $Q(s,a)$ values changes as they get updated, so it is necessary to use a forgetting mechanism in calculating the involved statistics (i.e., the means $\overline{Q(s,a)}$ and the standard deviations). Typically, transition samples are stored in sliding windows for each state-action pair in order to implement such a forgetting mechanism, which is rather expensive.

In summary, this technique can be viewed as a *variance-based technique*: when two actions have been tried the same number of times and have the same estimated mean value, the one that delivers the most random samples will be chosen. It should be noted that the IE exploration strategy is concerned with action exploration and exploration-exploitation balancing but not with state space exploration, as it is the case with the counter- or recency-based methods.

IE and other previously discussed directed techniques are sometimes called *local* exploration policies, because they do not use exploration information related to

states and actions other than the current ones. The strategies that do use information about other states and actions are called *global*. Meuleau and Bourgine (1999) extended the IE method to perform global exploration. In their algorithm, the exploration bonuses $\delta(s,a) = \sigma_{Q(s,a)}\frac{z_{\theta/2}}{\sqrt{n(s,a)}}$, as in (4.3), are back-propagated through states together with action-values. The Q-learning-like updates are then performed on the unified evaluation function[3]:

$$N(s,a) = (1-\alpha)N(s,a) + \alpha[r^a_{ss'} + (1-\gamma)\delta(s,a) + \gamma \max_b N(s',b)] \tag{4.4}$$

The back-propagation of the exploration bonuses was previously suggested by Sutton (1990) and is known as Dyna-Q+ algorithm:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r^a_{ss'} + \beta\sqrt{\rho(s,a)} + \gamma \max_b Q(s',b)] \text{ , where } \beta \in [0,1) \tag{4.5}$$

The only difference between algorithms (4.4) and (4.5) is in the form of the propagated exploration bonus, where in the former case, it is the Interval Estimation bonus and in the latter case, it is the recency value. The empirical evaluation of the algorithm (4.4) showed an advantage of the global exploration over the undirected and local IE techniques. This algorithm was also used for comparison purposes in the study of Dearden *et.al* (1998), which found that on certain domains, the performance of this algorithm was worse than the performance of other techniques, including the local IE method and the Boltzman distribution. Thus, as is often the case with various learning and optimization methods, different domains may require different exploration approaches to improve the efficiency of learning.

In [Wiering and Schmidhuber, 1998; Wiering, 1999], the idea of variance- and optimism-based heuristic behind the IE algorithm of Kaelbling (1993) was used for the estimates of the transition probability function. In this approach, called *model-based interval estimation (MBIE)*, for each state-action pair, the upper bound of the $100(1-\theta)\%$ confidence interval is calculated for the transition probability of a successor state with the highest estimate of its state value. This upper bound (an optimistic

[3]This algorithm is heuristic and not theoretically founded as the original Q-learning.

probability estimate) is used for this successor state while all other transition probabilities are renormalized. Then asynchronous real-time dynamic programming (see Chapter 2) is applied to the MDP with such an optimistic model. Since dynamic programming is performed on the MDP model that reflects exploration preferences, the algorithm is also global, similar to the approach in [Meuleau and Bourgine, 1999]. Similar to the IE algorithm of Kaelbling (1993), the algorithm relies on the assumption of the normal (Gaussian) distribution to model the uncertainty about the estimated transition probabilities. In order to be able to use the Gaussian model, a sufficiently large number of samples need to be gathered for each state-action-successor triple first. Because of this, on the initial stages of learning, a counter-based exploration method is used to obtain a good estimated model, and only then the MBIE is employed. This prevents the agent from taking advantage of the MBIE approach early during learning.

The difficulty of the MBIE algorithm with using the Gaussian density, as explained above, was addressed in [Wyatt, 2001], where the Dirichlet distribution was used instead. This allowed incorporation of prior knowledge and taking advantage of the underlying idea from the outset of learning[4]. This algorithm was shown to produce good results both with respect to resolving the exploration-exploitation trade-off and finding good policies in a given amount of learning time. The algorithm compared favorably to the approach of [Meuleau and Bourgine, 1999] and [Wiering and Schmidhuber, 1998] on several test tasks. As remarked in [Wyatt, 2001], reasoning about the uncertainty in the model rather than in the value function can be advantageous in the case of knowledge transfer between similar tasks. It can also be easier, since parameters of the MDP model are stationary during learning while the value functions are not.

Another example of algorithms, which explicitly deal with the exploration-exploitation trade-off and the state space exploration, is a method known as E^3 (*Explicit Exploitation and Exploration*) [Kearns and Singh, 1998]. It reasons about the accuracy of the

[4]The algorithm requires an off-line calculation of a look-up table of values necessary to compute the estimates of the upper bounds of the confidence intervals.

learned model parameters based on the number of collected samples, with the state space divided into "known" and "unknown" parts. On every time step, the decision is taken on whether enough reward can be collected in the "known" part of the state space, or whether the "unknown" part should and can be efficiently explored to increase the chances of higher returns. This algorithm requires some prior knowledge about the MDP, for example, maximum attainable returns from each state, which constitutes one of the difficulties in its practical implementation[5]. This was the first general near-optimal tabular reinforcement learning algorithm with provably polynomial computational time.

The work in [Fiechter, 1994; 1997] also introduced a *probably approximately correct* model of reinforcement learning, where off- and on-line model-based algorithms were proposed, such that they guaranteed, with probability $1-\delta$, to produce ϵ-optimal policies in polynomial time. In this case, the polynomial time bounds were expressed in terms of a *size parameter* n that roughly measured the complexity of the environment. Among other things, for example, it was assumed that the complexity of the environment is such that the value of any policy is bounded by a polynomial in n, and no sequence of m trials takes time greater than some polynomial in m and n. However, in [Thrun, 1992], it was shown that for any size of the state space, there exists an MDP, in which an optimal policy takes an exponential expected number of steps to get to a goal state. Thus, in general, computational times can be exponential in the size of the MDP's state space with the algorithms of Fiechter (1994,1997) and Kearns and Singh (1998). It is not clear, however, whether real-world tasks could have such a "malicious" structure as in the result of Thrun (1992).

Another approach that relies on the assessment of the agent's uncertainty about the environment was developed by Dearden *et.al.* (1998,1999), both for the model-free Q-learning method and for model-based reinforcement learning. In these algorithms, the benefit of exploration is evaluated using a notion of Value of Information (VoI): an expected gain in the value of the best action, assuming that perfect information

[5]A slightly different and a more general variant of this algorithm, which is also easier for practical implementations, was introduced in [Brafman and Tennenhotlz, 2002].

about an exploratory action can be obtained. Since such perfect information cannot be obtained with on-line sampling, a myopic approximation to the VoI is computed. Some analogies can be drawn between the model-based method using the VoI and the E^3 algorithm, with one important conceptual difference: in E^3, states are classified into "known" and "unknown" in a binary manner, whereas the algorithm in [Dearden *et al.*, 1999] has a "smooth" model of the uncertainty about the parameters of the MDP model that is being learned. This allows exploitation even before the dynamics in "promising" parts of the state space is learned with high confidence. The disadvantage of these methods is that they are fairly complex and much more computationally expensive than the previously discussed heuristics. Empirical comparison in [Dearden *et al.*, 1998] showed that the VoI-based Q-learning performed better than Boltzman exploration, the IE technique and global IE algorithms of [Meuleau and Bourgine, 1999] on maze-like and loop-like domains, and was comparable to other techniques on chain-like tasks. Model-based learning with VoI exploration was compared to an error-based exploration on maze-like MDPs and showed superior performance in terms of learning speed and learning costs, but not in terms of the quality of the greedy solution strategy.

Another exploration method, suitable for maze-like tasks, was proposed in [Dayan and Sejnowski, 1996]. This algorithm performs model-based learning on the estimated means of the transition probabilities, while the posterior probabilities of the estimated MDP model are updated in a Bayesian manner after each observed transition. This approach implicitly favors actions that have been tried a long time ago, while also encouraging actions that have a potential of improving the solution that currently looks best.

All techniques discussed in this section were designed for finite MDPs and tabular reinforcement learning methods. To the best of our knowledge, only the error-based method was applied with reinforcement learning using function approximation in [Thrun and Moller, 1991].

4.3. Attribute-Based Exploration Strategy

Similarly to other directed exploration methods, the goal of our approach is to facilitate a more uniform visitation of the state space during learning, while also gathering quickly the samples most needed to estimate well the value functions. As the analysis in Chapters 3 suggested, several MDP attributes studied therein can be used in order to achieve this goal. In this chapter, we focus on using two attributes: the State Transition Entropy and the Forward Controllability. Both attributes can be computed for each state-action pair (s, a) based on the MDP model (if it is known) or they can be estimated based on sample transitions as discussed in Chapter 3. The basic idea of our strategy is to favor exploratory actions which exhibit high values of the State Transition Entropy, the Forward Controllability or both of these attributes. We will now explain the details of our approach.

4.3.1. Motivation for the Use of the State Transition Entropy

Recall that the State Transition Entropy measures the amount of stochasticity due to the environment's state dynamics. It can be computed using the definition in Equation (3.11). As discussed in Chapter 3, in environments with high State Transition Entropy values, the quality of the solutions obtained by on-line value-based algorithms can be affected by a trade-off between the positive effect of "natural" exploration and the negative effect of high variance in the value samples used by the algorithm.

A high value of $STE(s, a)$ means that there are many possible next states s' (with $P^a_{s,s'} \neq 0$) which occur with similar probabilities. If in some state s, actions a_1 and a_2 are such that $STE(s, a_1) > STE(s, a_2)$, the agent is more likely to encounter more different states by taking action a_1 than by taking action a_2. This means that giving preference to actions with higher State Transition Entropy values could achieve a more homogeneous exploration of the state space. Current literature contains several indications that this can be beneficial. Empirical evidence suggests that a homogeneous visitation of the state space can be helpful for different on-line reinforcement learning

algorithms. For instance, in [Sutton and Barto, 1998], the performance of Q-learning with the ϵ-greedy behavior policy was compared with the performance of Q-learning applied while picking states uniformly randomly. The experiments were performed on discrete random MDPs with different branching factors. Note that large branching factors mean high State Transition Entropy values. In these tasks, the ϵ-greedy updates resulted in better solutions and faster learning mainly for deterministic tasks (i.e., with branching factor 1). As the branching factor (and thus the State Transition Entropy) increased, performing action-value updates uniformly across the state space led to better solutions in a long run, and to a better learning speed. Also, as discussed in the previous section, the counter-based and recency-based directed approaches, which are intended for a similar purpose, demonstrated good empirical performance in the past (see e.g., [Thrun, 1992]). Similar indications can be found for approximate reinforcement learning methods (using non-tabular representations of the value functions). For example, in a theoretical analysis of the Approximate Policy Iteration in [Munos, 2003], the convergence result relies on a uniform state-space sampling. As indicated in [Munos, 2003], with the Policy Iteration algorithm, it is important to use a sample distribution that is sufficiently "close" to a uniform one, in order to ensure certain reliability of approximate value estimates uniformly over the state space. This is necessary for the correctness and efficiency of the policy improvement step.

Another potential consequence of a high value of $STE(s,a)$ is a large variance of the action-value estimates for (s,a). Recall that in on-line learning methods, such as Sarsa, the action-value of a state-action pair (s,a) is updated toward a target estimate obtained after taking action a:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha[\underbrace{r^a_{ss'} + \gamma Q(s',a')}_{\text{Target for } (s,a)}], \alpha \in (0,1)$$

These target estimates are drawn according to the probability distribution of the next states. If $STE(s,a)$ is high, there will be many possible next states, and consequently

the variance in the target estimates could be high. In order to get a good estimate of the value $Q(s,a)$ when it has a high variance, more samples are needed. By encouraging exploration of actions with high State Transition Entropy values, our strategy faciliates the agent's effort to collect enough samples. This idea is reminiscent of the IE directed exploration method [Kaelbling, 1993], as discussed in the previous section, but we do not rely on explicitly estimating the variance of the action-value samples, which would be much more expensive in terms of both storage and computation.

4.3.2. Motivation for the Use of the Forward Controllability

As defined in Chapter 3, the Controllability of a state s, $C(s)$, is a normalized measure of the information gain when predicting the next state based on knowledge of the action taken, as opposed to making the prediction before the action is chosen. It can be computed as in Equation (3.21). The Forward Controllability of a state-action pair is the expected Controllability of the next state: $FC(s,a) = \sum_{s' \in S} P^a_{s,s'} C(s')$.

Favoring actions with high Forward Controllability values will direct the reinforcement learning agent toward states in which its actions determine to a large extent the outcomes of state transitions. Having such control over the state dynamics enables the agent to reap higher returns in environments where some trajectories are more profitable than others. Correct decisions in highly controllable states are likely to have a greater impact on the agent's overall performance and thus are most beneficial if learned as quickly as possible.

At the same time, actions with high Forward Controllability values lead to successor states s', in which different actions a' have very different outcomes. Hence, from such states, the agent can experience richer exploration patterns by choosing different actions.

Finally, consider learning with the on-policy reinforcement learning algorithm Sarsa. Since different actions a', available in a highly controllable successor state s', lead to very different next states, one can expect a significant variation in the values of $Q(s',a')$ for different actions a'. Because of this, sample targets for $Q(s,a)$, that

is values $[r^a_{s,s'} + \gamma Q(s', a')]$, obtained with different exploration actions a', can have high variance. Thus, gathering more samples for state-action pairs with high values of $FC(s, a)$ can facilitate estimation of potentially high variance values.

4.3.3. Implementation of Attribute-Based Exploration

The idea of guiding exploration based on the values of the State Transition Entropy and Forward Controllability attributes can be incorporated easily into both undirected and directed exploration techniques. For instance, we can design an attribute-based ϵ-greedy exploration strategy in the following way. The greedy action is chosen with probability $(1 - \epsilon)$, as in the standard method. When a choice to explore is made (with probability ϵ), the exploratory action can be selected according to a probability distribution that depends on the values of the attributes. For instance, in our implementation, we use the following Boltzman distribution:

$$\pi(s, a) = \frac{e^{\frac{K_1 STE(s,a) + K_2 FC(s,a)}{\tau}}}{\sum_{b \in A} e^{\frac{K_1 STE(s,b) + K_2 FC(s,b)}{\tau}}} \tag{4.6}$$

where τ is the temperature parameter. The non-negative constants K_1 and K_2 can be used to adjust the relative contribution of each term.

An alternative to the Boltzman distribution would be a multinomial distribution that assigns action-selection probabilities proportionally to the values of the MDP attributes:

$$\begin{aligned} \hat{\pi}(s, a) &= \max\{[K_1 STE(s, a) + K_2 FC(s, a)], \epsilon_{min}\}, \forall a \in A \\ \pi(s, a) &= \frac{\hat{\pi}(s, a)}{\sum_{b \in A} \hat{\pi}(s, b)}, \forall a \in A \end{aligned} \tag{4.7}$$

Parameter $\epsilon_{min} \in (0, 1)$ represents a minimum probability of selecting each action regardless of the attribute values.

The State Transition Entropy and Forward Controllability attributes can be used as exploration bonuses in directed exploration, and hence can be easily combined with many existing methods. In this case, the behavior policy deterministically picks the

action maximizing the evaluation function:

$$N(s,a) = KQ(s,a) + K_1 STE(s,a) + K_2 FC(s,a) + \sum_{i=3}^{k} K_i \delta_i(s,a) \qquad (4.8)$$

where each $\delta_i(s,a)$, $i = 1,...,k$, can be any exploration bonus based on data about the learning process, e.g., the counter-based, recency-based, error-based or IE-based bonus. In this case, the trade-off between exploitation and exploration can be controlled by tuning the parameters K and K_i, $i = 1,...,k$, associated with each term.

Note that our exploration approach uses only characteristics of the environment, which are independent of the learning process. Thus, the information needed can be gathered prior to learning. This can be done if the transition model is known, or if the agent has an access to a simulator, with which it can interact to estimate the attributes from sampled state transitions. Note that even if the MDP model is known, it is often not feasible to apply standard synchronous dynamic programming methods and the issue of efficient exploration is still important. As suggested in [Wiatt, 1997; Fiechter, 1997], model-based exploration methods are in fact superior to model-free methods in many cases. Also, the values of the attributes can be carried over if the task changes slightly (e.g., in the case of goal-directed tasks, in which the goal location changes). Alternatively, the attributes can be estimated during learning based on observed state transitions. This can be done efficiently by incremental methods, as discussed in Chapter 3. In this case, only a small constant amount of extra computation per time step is needed. This is in contrast to most other directed exploration methods, which not only rely on estimation of the transition probabilities, but also require more computation to re-evaluate their exploration bonuses on every time step (see e.g., [Thrun, 1992; Kearns and Singh, 1998; Sutton, 1990; Kaelbling, 1993; Wiering and Schmidhuber, 1998; Dearden *et al.*, 1999]). In the case of incremental estimation of the attributes during learning, they should be initialized to high values to encourage good initial estimation of their true values for all state-action pairs.

We also note that the exploration-relevant information based on the learning history used in other directed techniques can carry the bias of previous (possibly unsuccessful) exploration decisions and value estimates. Our attribute-based exploration bonuses are less affected in this respect and provide a more independent criterion.

4.4. Empirical Evaluation of Attribute-Based Exploration

4.4.1. Exploration Strategies Tested

In order to assess empirically the merit of using the State Transition Entropy and the Forward Controllability as heuristics for guiding exploration, we incorporated these attributes into the ϵ-greedy exploration strategy (as a representative of undirected methods) and into the recency-based exploration strategy (as a representative of directed methods). We chose the recency-based method among directed exploration techniques because in previous experimental studies [Thrun, 1992], it compared favorably to other directed methods, while being less sensitive to variations in user-defined parameters. We used both exploration strategies with the Sarsa(0) algorithm.

We experimented with two ways of incorporating the attributes into the ϵ-greedy strategy: using the Boltzman distribution, as shown in Equation (4.6), and using the multinomial distribution, as shown in Equation (4.7). The recency-based technique was combined with the MDP attributes based on the idea of additive exploration bonuses, as shown in Equation (4.8), where we used the recency-based exploration bonus, $\delta_3(s,a)$, as detailed in Section 4.2, and the corresponding constant $K_3 = 1$. Both in the case of ϵ-greedy and recency-based exploration, we used parameter settings $K_1, K_2 \in \{0,1\}$, which corresponded to either not using or using the attribute(s). The settings of other parameters involved are specified in the description of each experiment below.

4.4.2. Experimental Results

Results on Random MDPs

First, we studied the performance of attribute-based exploration in a very general setting using discrete random MDPs as presented in Chapter 3. All random MDPs tested have two state variables, each with 25 possible values. There are three actions available in every state. The branching factor for these MDPs varies randomly between 1 and 20 across state-action pairs, which generates state-action pairs with different values of the State Transition Entropy and states with varying Controllability. These test MDPs are modeled as episodic with termination probability of 0.01 in each state.

The random MDPs were divided into two groups of five MDPs each. One group contained MDPs with "low" global values of the State Transition Entropy, that is in the interval $[0.9, 1.7)$, and the other group contained MDPs with "high" global State Transition Entropy values, that is in the interval $[1.7, 2.7]$. In the remainder of this section, we will call them low-STE and high-STE MDPs respectively. In low-STE MDPs, the values of the State Transition Entropy for state-action pairs are lower overall compared to high-STE MDPs. Also, low-STE environments have a smaller proportion of actions with relatively high State Transition Entropy values. The objective of considering two groups was to investigate whether the overall amount of stochasticity in the environment influences the effect of the attributes on exploration.

The attribute values were precomputed prior to learning. We simulated 500 state transitions for each state-action pair in order to estimate transition probabilities. Then, the attribute values were estimated using the definitions presented in Section 3.3.

We averaged performance measurements over the five MDPs in each group. In order to do this, we normalized the greedy returns with respect to the performance of the optimal policies of the corresponding MDPs on the test set of start states. Optimal policies were obtained through the Value Iteration algorithm using exact

RL algorithm	Sarsa(0)
Value-function representation	Table
Exploration strategy	ϵ-greedy, constant ϵ
Exploration parameter settings	$\epsilon \in \{0.1, 0.4, 0.9\}$
Learning rate schedule	Decreasing: $\alpha_N = \frac{\beta-1}{\beta^{N+1}-1}$
Learning rate settings	$\beta \in \{0.75, 0.9, 0.95, 0.98\}$
Number of learning trials per run	10000
Number of runs	20
Frequency of greedy policy evaluation	Every 50 trials
Number of simulated trajectories for greedy policy evaluation	30
Start state distribution	Uniform over the whole state space
Number of test states	50

TABLE 4.1. Experimental settings for ϵ-greedy exploration on discrete random MDPs.

MDP models. The normalization was performed in the same way as in the previous experiments described in Section 3.5.

The figures below present learning curves averaged over 20 learning runs and 5 tasks in each group, as well as bar-graphs for the Cumulative Weighted Penalty measure (introduced in Section 3.5).

Results for the ϵ-Greedy Strategy

Experimental settings for the experiments with ϵ-greedy exploration are summarized in Table 4.1. The attributes were incorporated into the ϵ-greedy strategy using the Boltzman distribution, as in Equation (4.6) with $\tau = 1$. The results of these experiments are presented in Figure 4.1 for the group of low-STE MDPs, and in Figure 4.2 for the group of high-STE MDPs. To produce these graphs, we selected the best setting of the learning rate parameter for each MDP, each setting of ϵ and each variant (using or not using the attributes).

As can be seen from figures 4.1 and 4.2, incorporating the MDP attributes into the ϵ-greedy strategy has a positive effect for both low-STE and high-STE MDPs, in all cases except $\epsilon = 0.1$ in high-STE environments.

The Forward Controllability exhibits a greater impact for high-STE MDPs (we can see larger differences between the performance of the Forward Controllability based variant and the attribute-free variant on Figure 4.2 than on Figure 4.1). High-STE environments tend to be less controllable, in general. In this case, it is important to identify those few states, where the agent can control the course of state transitions well and to focus exploration on these states.

On the other hand, the use of the State Transition Entropy has a greater impact for low-STE MDPs than for high-STE MDPs. As previously mentioned, our low-STE environments have a lower proportion of actions with high State Transition Entropy values compared to high-STE MDPs. For low-STE environments, using the State Transition Entropy attribute in the exploration strategy helps to identify these actions and to increase their selection probabilities significantly compared to other actions. Since the levels of the State Transition Entropy are lower in low-STE MDPs in general, increasing the frequency of exploring with actions that have high State Transition Entropy improves uniform visitation of the state space more significantly. Hence, the effect of using this attribute is more pronounced for low-STE MDPs.

As can be seen from figures 4.1 and 4.2, for both MDP groups, the quality of the greedy policies learned increases as the value of the exploration parameter ϵ increases, regardless of whether the attributes are used or not. Note also that the effect of using the attributes increases as the value of ϵ increases. Higher values of ϵ allow to exercise the attribute effect more frequently, hence, the effect is more pronounced. The only case in these experiments, where we see a degradation of performance with the use of the attributes is for the high-STE MDPs and with $\epsilon = 0.1$. In this case, using the State Transition Entropy produces a negative effect. As previously mentioned, high-STE environments have greater State Transition Entropy values overall and thus can have a large variance in the value samples. The agent using the State Transition Entropy attribute provokes most the variance for exploratory actions, but in order to deal with this variance successfully, more samples may be needed for these actions

overall. Thus, the performance of the strategy using the State Transition Entropy is not good for the low value of the exploration rate, but it improves as ϵ increases.

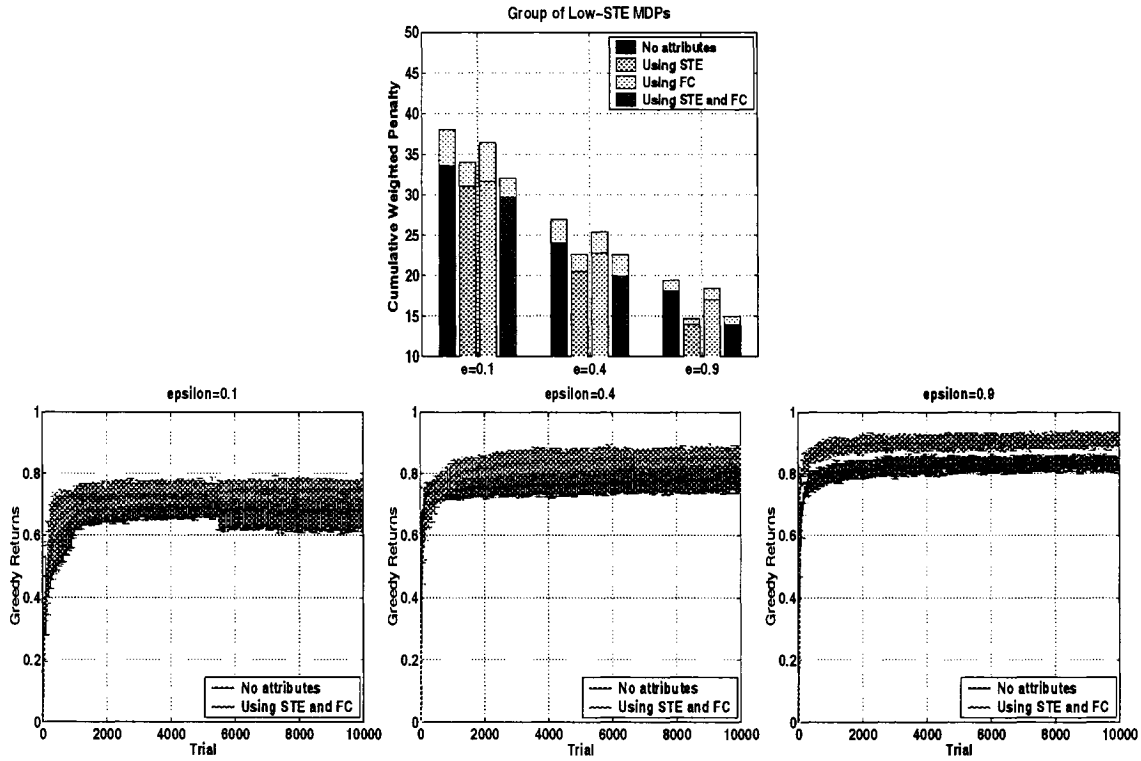


FIGURE 4.1. Performance of ϵ-greedy exploration (with and without the use of the MDP attributes) for low-STE discrete random MDPs.

We can see from figures 4.1 and 4.2 that using the two attributes together usually does not bring any performance improvement over using one of the attributes in isolation. The main reason is likely due to the fact that values of the State Transition Entropy are usually higher than values of the Forward Controllability and will tend to have a dominating influence. Also, the values of the two attributes can sometimes push exploration in different directions: when the State Transition Entropy is high and the Forward Controllability is low for some action relative to attribute values of other actions in a given state, their effects will be canceled out. Even if both attributes have relatively high values compared to other actions, when we take an action with a high Forward Controllability, it is likely that the next highly controllable state has low State Transition Entropies overall, and thus we reduce the chances for the State

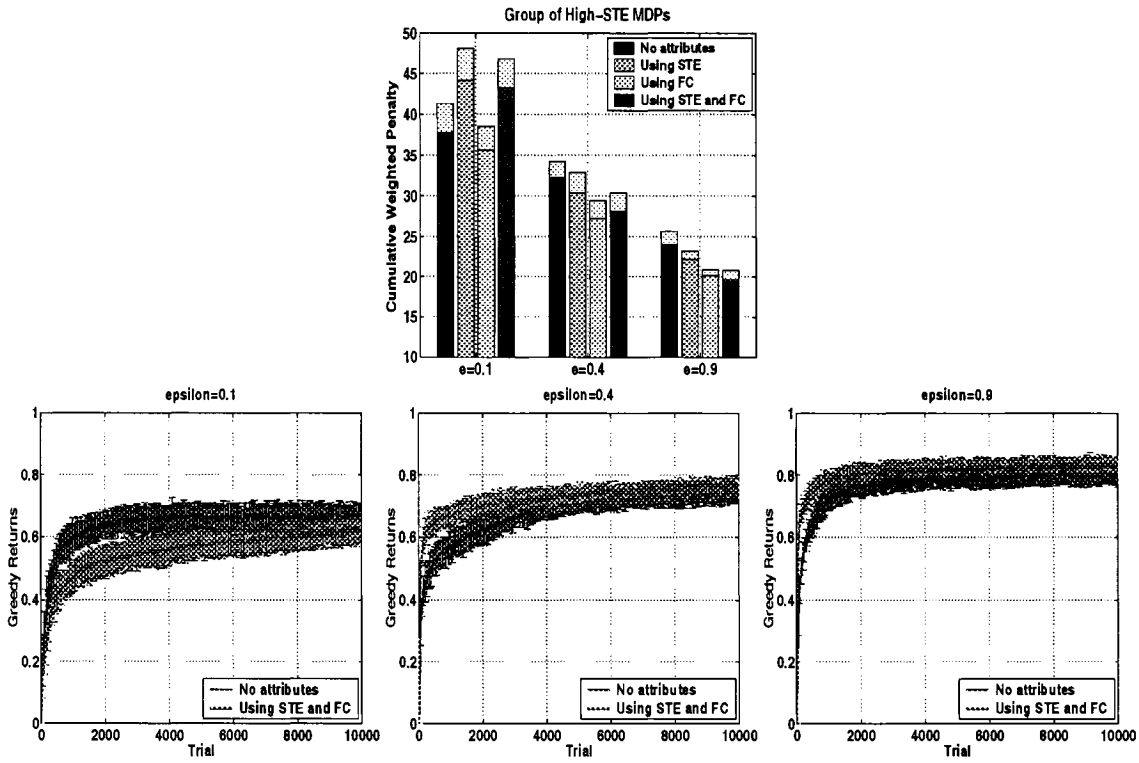


FIGURE 4.2. Performance of ϵ-greedy exploration (with and without the use of MDP attributes) for high-STE discrete random MDPs.

Transition Entropy to exercise its effect. It would be interesting to use weighting of the individual contribution of the two attributes when they are used together. We leave such experiments for future work.

We performed statistical tests to estimate the significance of differences in the performance of ϵ-greedy exploration with and without the MDP attributes. We performed two-way ANOVA for measurements of the Cumulative Weighted Penalty. In this analysis, one factor was the parameter ϵ and the other factor was the strategy variant (one variant without attributes, two variants using one of the attributes and one variant using both attributes together). This analysis was performed separately for the group of low-STE MDPs and the group of high-STE MDPs. Similar to the experiments presented in Section 3.5, each data point in this analysis represented the Cumulative Weighted Penalty measurement for one MDP. In this case, however, we have an experiment with repeated measures (see e.g., [Cohen, 1995;

Howell, 2002]), since each MDP is tested in all experimental conditions. The results of the tests are summarized in Figure 4.3. These tests show that the effect of using the attributes is statistically significant both for the low-STE and the high-STE groups at confidence level greater or equal than 0.98. The test for the high STE group showed a significant interaction effect, which is mainly due to the different performance pattern for $\epsilon = 0.1$ in this group.

	Low-STE MDPs	High-STE MDPs
$ndf_\epsilon = 2$; $ddf_\epsilon = 8$		
F_ϵ	8.64	23.54
$1 - p_\epsilon$	0.99	0.99
$ndf_{att} = 3$; $ddf_{att} = 12$		
F_{att}	4.81	13.23
$1 - p_{att}$	0.98	0.99
$ndf_{int} = 6$; $ddf_{int} = 6$		
F_{int}	1.02	8.46
$1 - p_{int}$	0.51	0.99

FIGURE 4.3. Two-way-ANOVA with repeated measures for ϵ-greedy exploration. Values of F and $1 - p$ represent F-statistic and the confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts ϵ and att refer to the effects of the exploration parameter ϵ and the effect of using the attributes respectively. Subscript int refers to the interaction effect between the two factors.

Results for the Recency-Based Strategy

Experimental settings for recency-based exploration are summarized in Table 4.2. The results of these experiments are presented in Figure 4.4 for the group of low-STE MDPs, and in Figure 4.5 for the group of high-STE MDPs. As before, for each MDP, each setting of K and each variant (using or not using the attributes), we selected the best setting of the learning rate parameter tested and used the corresponding performance measurements to produce the graphs.

As can be seen from figures 4.4 and 4.5, the recency-based method is significantly more robust with respect to the settings of the parameter K than the ϵ-greedy strategy is with respect to the settings of ϵ. With all considered settings of K, the performance of this strategy was very good. Nevertheless, using the MDP attributes

RL algorithm	Sarsa(0)
Value-function representation	Table
Exploration strategy	Recency-based
Exploration parameter settings	$K \in \{1, 10, 50\}$
Learning rate schedule	Decreasing: $\alpha_N = \frac{\beta-1}{\beta^{N+1}-1}$
Learning rate settings	$\beta \in \{0.75, 0.9, 0.95, 0.98\}$
Number of learning trials per run	10000
Number of runs	20
Frequency of greedy policy evaluation	Every 50 trials
Number of simulated trajectories for greedy policy evaluation	30
Start state distribution	Uniform over the whole state space
Number of test states	50

TABLE 4.2. Experimental settings for recency-based exploration on discrete random MDPs.

further improved the performance of the recency-based method, although the effects appear to be smaller than in the case of the ϵ-greedy method. The recency-based strategy itself, to a large extent, generates a more uniform state-space exploration. The attributes also contribute in this respect but they also help to sample actions with more action-value variance and they focus learning on more controllable states. In these experiments, we did not tune relative contributions of the recency-based exploration bonus and the attributes, which can also be responsible for a smaller effect. Similar to the case of ϵ-greedy exploration, the State Transition Entropy exhibits a stronger effect in the case of low-STE MDPs, whereas the Forward Controllability shows a stronger effect on high-STE MDPs. Also, as observed before, the use of the two attributes together does not lead to further performance improvements, compared to the case of their individual use.

We performed two-way ANOVA on the experimental results with recency-based exploration in the same manner as in the case of ϵ-greedy exploration. The results are summarized in Figure 4.6. This test shows that the effect of using the attributes is statistically significant both for the low-STE and the high-STE groups at the confidence level of 0.95.

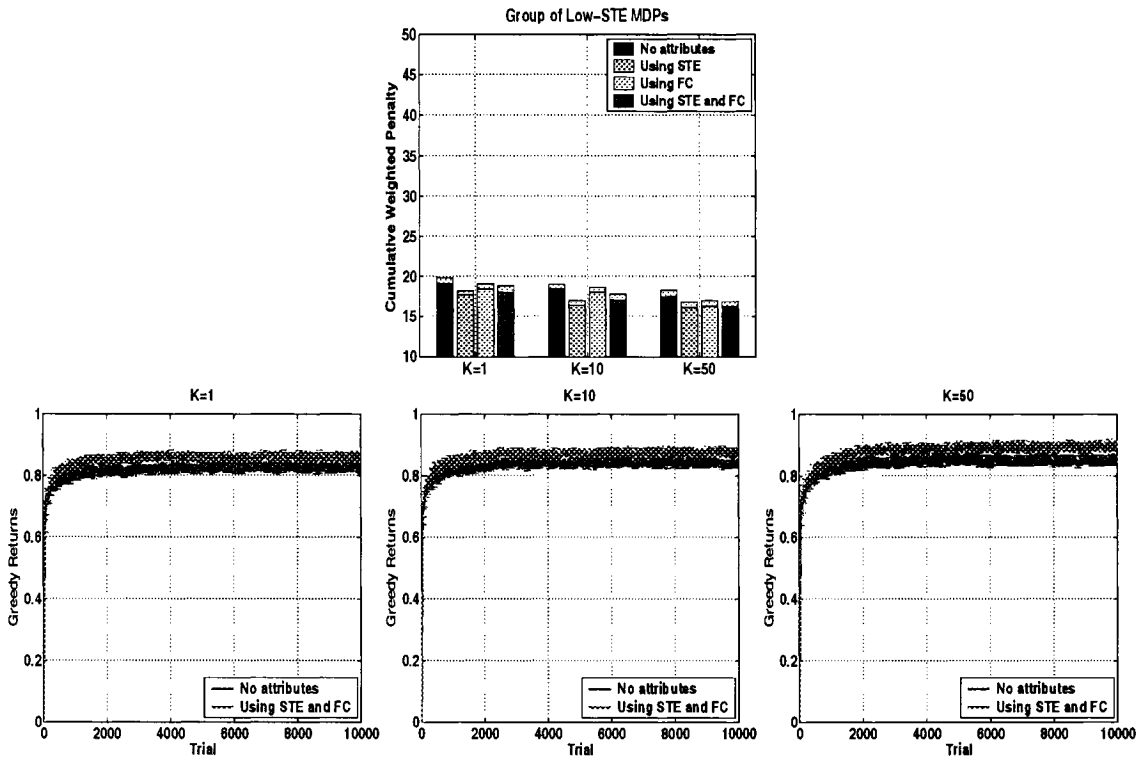


FIGURE 4.4. Performance of recency-based exploration (with and without the use of the MDP attributes) for low-STE discrete random MDPs.

We also performed experiments with two groups of MDPs which have similar global State Transition Entropy values as in the two groups discussed above. In this case, however, MDPs have similar values of the attributes across all states-action pairs. Here we would expect to see no effect of using the attributes, because exploration decisions in all states should be mostly unaffected by the attribute values. We performed these experiments to test the possibility of observing any effect "by chance". The results of these experiments did not reveal any effect of using the attributes with either ϵ-greedy or recency-based exploration. This reinforces our conclusion that the effects observed in the previous experiments are not spurious.

The experiments on random MDPs served as the first assessment of the potential of our strategy. It should be noted that the experiments were performed with uniformly randomly sampled start states, which already helps exploration significantly.

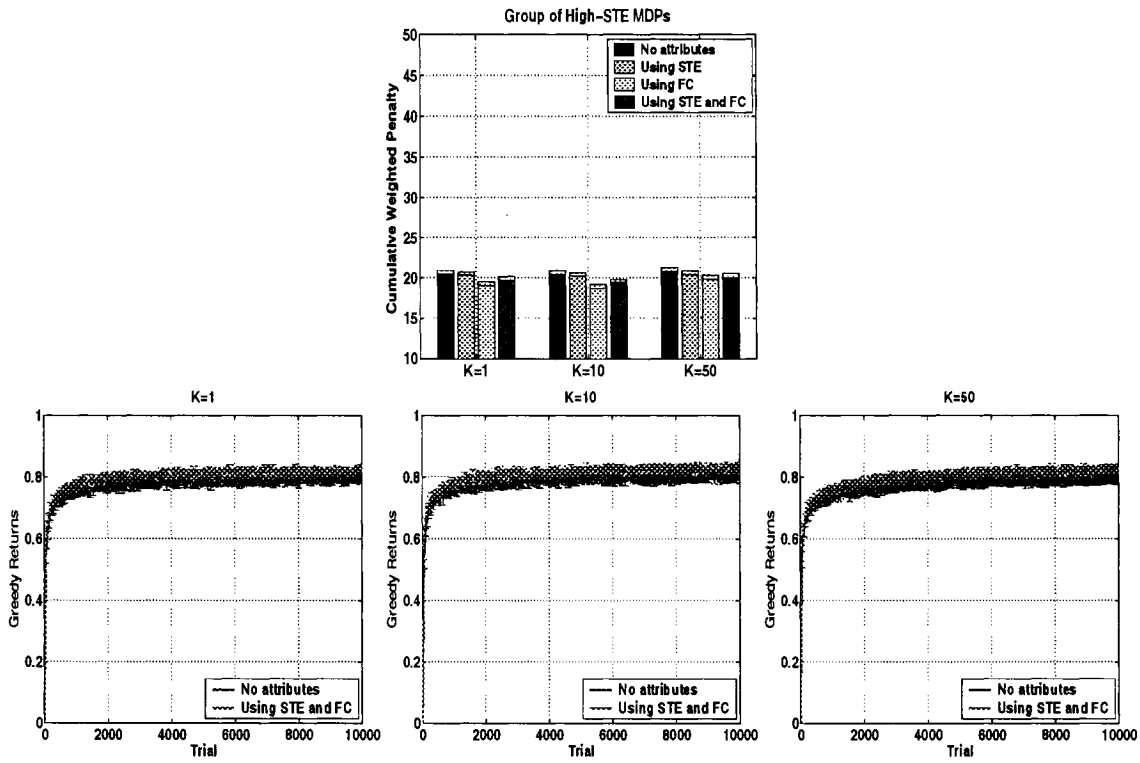


FIGURE 4.5. Performance of recency-based exploration (with and without the use of the MDP attributes) for high-STE discrete random MDPs.

However, even in this case, the recency-based strategy improves over ϵ-greedy exploration by ensuring more uniform state visitation on trajectories. Our approach improves the performance even further. Next, we present the results of similar experiments on a more structured grid-world domain.

	Low-STE MDPs	High-STE MDPs
$ndf_K = 2$; $ddf_K = 8$		
F_K	2.94	1.06
$1 - p_K$	0.89	0.61
$ndf_{att} = 3$; $ddf_{att} = 12$		
F_{att}	3.79	3.49
$1 - p_{att}$	0.96	0.95
$ndf_{int} = 6$; $ddf_{int} = 6$		
F_{int}	0.32	2.90
$1 - p_{int}$	0.10	0.89

FIGURE 4.6. Two-way-ANOVA with repeated measures for recency-based exploration. Values of F and $1 - p$ represent F-statistic and confidence level respectively. Values of "ndf" and "ddf" represent numerator and denominator degrees of freedom. Subscripts K and att refer to the effects of the parameter K and the effect of using the attributes respectively. Subscript int refers to the interaction effect between the two factors.

4.4.2.1. Results on Gridworld Tasks

In the second set of experiments, we examined the performance of attribute-based exploration on goal-directed tasks in grid-world environments with various obstacles. The structure of these MDPs is as follows.

The environment is a rectangular grid-world, where the agent has to reach a goal state. The agent has two kinds of actions: move one cell in one of the four directions (*OneNorth, OneWest, OneEast, OneSouth*) or move two cells in one of the four directions (*TwoNorth, TwoWest, TwoEast, TwoSouth*). We will call the first kind of actions one-cell movement actions and the second kind of actions - two-cell movement actions. The actions have the following outcomes. With the one-cell movement actions, the agent moves in the intended direction with probability 0.9 and stays in place with probability 0.1 (see the left panel of Figure 4.7). With the two-cell movement actions, the agent moves two cells in the intended direction with probability 0.7 and moves to one of the three adjacent cells in the intended direction with probabilities 0.1 respectively (see the right panel of Figure 4.7). The two-cell movement actions have a higher State Transition Entropy: the State Transition Entropy is equal to 0.46 for one-cell movement actions and is equal to 1.35 for two-cell movement actions.

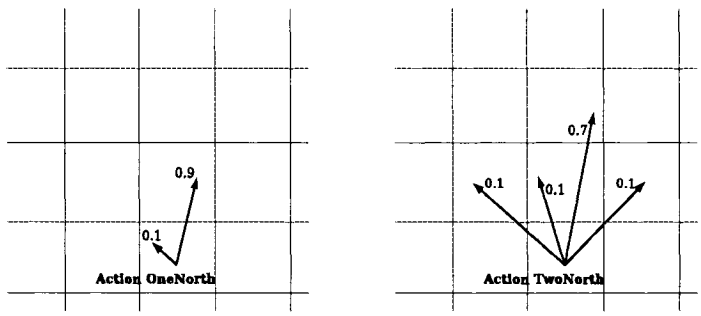


FIGURE 4.7. Selected transition probabilities in the grid-world domain.

Some of the states (cells on the grid) have obstacles along one or more of their sides (walls). Obstacles are always located around the perimeter of the grid-world, as well as at some internal cells. When the agent attempts to move into such an obstacle, it bounces back either to the same cell, with probability 0.5, or to the rear

cell, with probability 0.5, provided that there is no obstacle behind. The agent gets a penalty of -3 when it bounces.

In states with obstacles, the State Transition Entropy is almost the same for one-cell and two-cell movement actions that attempt to move into the obstacle (equal to 0.99 and 1 respectively). States with obstacles are less controllable. For example, a state with no obstacles has the Controllability equal to 0.73, whereas a state with one obstacle has the Controllability equal to 0.67.

There are parts of the grid-world with a slippery floor. When performing a two-cell movement action in a slippery state, the agent may "slip", with probability 0.2, and end up in any of the eight neighboring cells with equal probabilities. If the slippery state has obstacles along some of its sides, then the agent can slip only to those neighboring states that are accessible or stay in the same state. There is a -2 penalty for slipping. Slippery states are much less controllable. The two-cell movement actions in slippery states have, in general, a higher variance of value samples. It is introduced by the variance in the rewards as well as by the fact that a state transition may end up in any of the neighboring states. If a slippery state has different kinds of neighbors, e.g., both slippery and dry, with and without obstacles, then the variance is highest. On the other hand, a slip facilitates state space exploration. In addition to the penalties described above, the agent gets a penalty of -1 for every state transition until it reaches the goal state, at which time the episode terminates. Thus, the agent tries to minimize the time necessary to reach the goal.

Information about obstacles and slippery conditions is not part of a state description available to the agent. The agent's state representation consists of the two-dimensional coordinates of the state only.

For our experiments, we randomly generated a set of such grid-world MDPs by randomly placing obstacles and slippery parts. Our random generator attempts to arrange slippery states in groups and to put obstacles such that they resemble walls. An example of a grid-world MDP is shown in Figure 4.8.

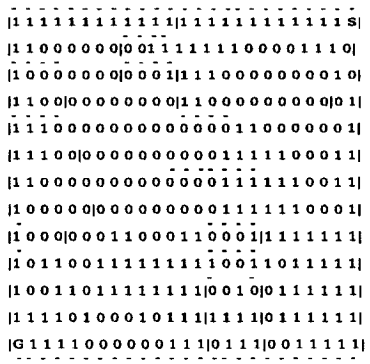


FIGURE 4.8. Example of a grid-world MDP. 1's represent slippery states and 0's represent dry states. Some states have walls on one or more of their sides. The goal state is marked with the letter G and the start state is marked with the letter S.

Group Number	Fixed Obstacles	% Slippery States
1	50	15
2	250	15
3	50	45
4	250	45
5	50	65
6	250	65

TABLE 4.3. Characteristics of the grid-world MDPs.

The goal state was always located at the lower left corner of the grid and the agent always started the episodes from the top right corner. This start state distribution is more difficult for exploration compared to the one used in the previous experiments, in which case start states were distributed uniformly randomly across the entire state space. Thus, directed exploration is even more important in this case.

We conducted our experiments on six groups of randomly generated grid-world MDPs (each defined over a 25 × 25 grid). Each group contained 5 MDPs with certain characteristics. In particular, we varied the number of fixed obstacles and the percentage of slippery states. These settings are summarized in Table 4.3.

As in the previous experiments, we normalized greedy returns with respect to the performance of the optimal policies of the corresponding MDPs from the start state. The optimal policies were obtained by the Value Iteration algorithm using exact MDP

RL algorithm	Sarsa(0)
Value-function representation	Table
Exploration strategy	ϵ-greedy, constant ϵ
Exploration parameter settings tested	$\epsilon \in \{0.01, 0.1, 0.4, 0.9\}$
Learning rate schedule	Decreasing: $\alpha_N = \frac{\beta-1}{\beta^{N+1}-1}$
Learning rate settings tested	$\beta \in \{0.75, 0.9, 0.95, 0.98\}$
Number of learning trials per run	10000
Number of runs	20
Frequency of greedy policy evaluation	Every 50 trials
Number of simulated trajectories for greedy policy evaluation	30

TABLE 4.4. Experimental settings for ϵ-greedy exploration on random grid-world MDPs.

models. The graphs below are based on performance measurements averaged over 20 learning runs and 5 MDPs in each group.

Results for the ϵ-Greedy Strategy

Experimental settings for ϵ-greedy exploration are summarized in Table 4.4. Attributes were incorporated into the ϵ-greedy strategy using the multinomial distribution in Equation (4.7). The corresponding results are presented in Figure 4.9. We selected the best settings for the exploration parameter ϵ and the learning rate parameter for each variant (with or without attributes) on each MDP and used the corresponding results to produce the graphs on Figure 4.9. This figure shows the results for the case, in which the attribute values were precomputed before learning, in the same way as described before for discrete random MDPs.

Figure 4.9 shows that using the MDP attributes improves the performance of the ϵ-greedy strategy for all groups of MDPs tested. By examining learning curves, we can see that the performance approaches asymptotically the optimal for all variants, but the variants using the attribute(s) exhibit faster learning. As can be observed from the bar-graphs showing the Cumulative Weighted Penalty measure, the variance is also smaller when using the MDP attributes. Overall, the performance of all three attribute-based variants is very similar. In the groups with 50 obstacles,

there is some tendency for an increased effect of the Forward Controllability as the percentage of slippery states increases, that is as the overall stochasticity increases. This trend is similar to the one observed for discrete random MDPs, although it is not as pronounced here.

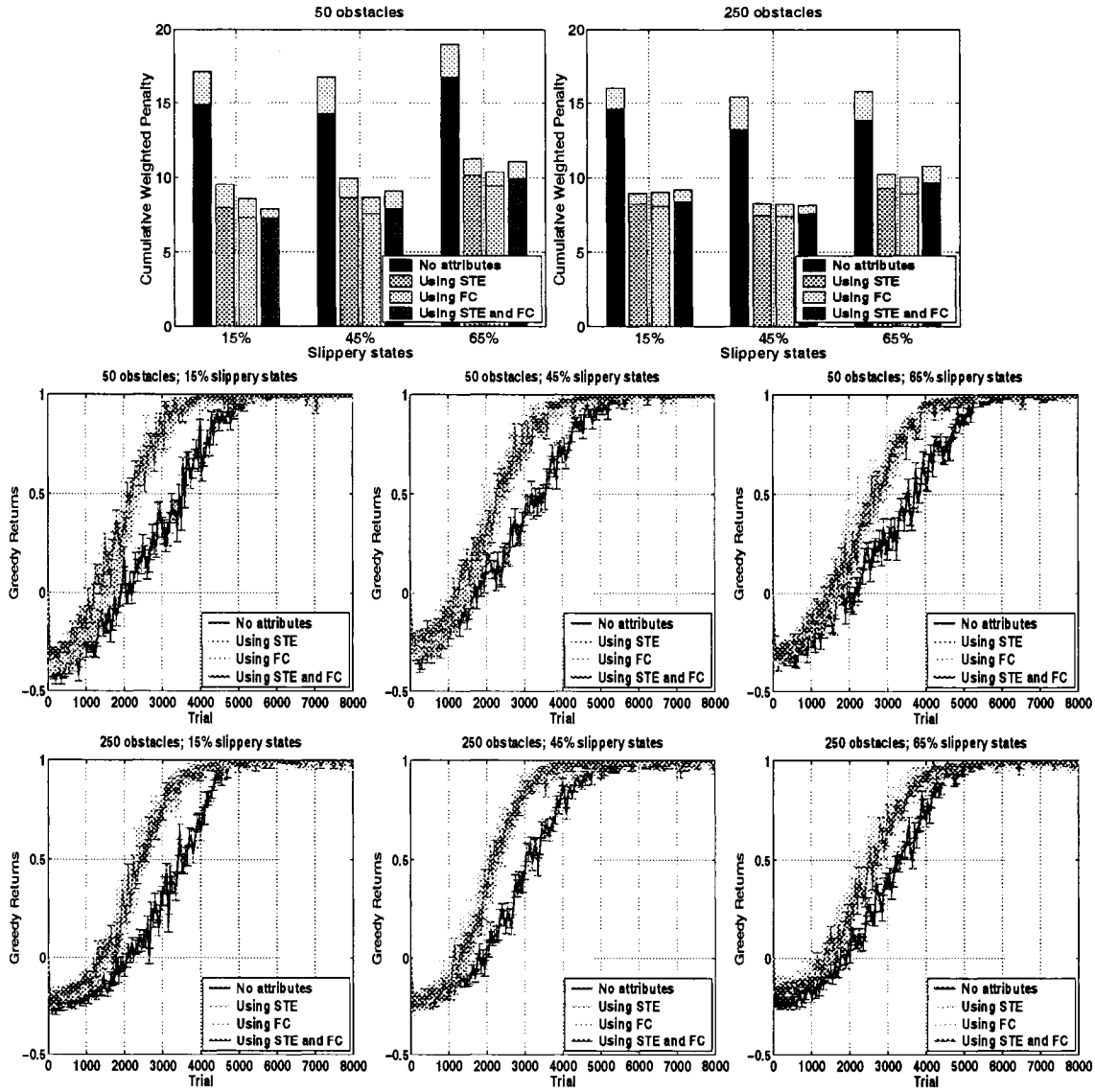


FIGURE 4.9. Performance of ϵ-greedy exploration with and without the use of the MDP attributes on grid-world MDPs. Attribute values were precomputed before the onset of learning.

We did not perform statistical tests for these experiments, since the learning curves and the bar-graphs clearly indicate differences in performance when using and not using the MDP attributes.

Figure 4.10 presents the results of similar experiments, however, in this case, attribute values were estimated during learning. Attribute values were initialized to high values at the beginning of learning (to the value of 3 for the State Transition Entropy and to the value of 1 for the Forward Controllability). Then incremental methods for attribute estimation, discussed in Section 3.4, were used to update attribute values after every state transition observed. Initializing attributes to high values results in encouraging equal exploration of all actions initially until attributes for some actions are estimated at lower values.

Figure 4.10 shows the same positive effect of using the attributes as in the previous experiments. Learning with the attributes is slightly slower in this case compared to the case where the attributes were precomputed before learning. We can see, for example, that in the beginning of learning (while the attribute values are estimated) the curves of the attribute-based variants are closer to the curves for learning without the attributes. Later on during learning, when the attribute estimates are better, performance trends are the same as in the previous experiments.

Results for the Recency-Based Strategy

Experimental settings for recency-based exploration on grid-world MDPs are summarized in Table 4.5. The results of these experiments are presented in Figure 4.11. The graphs were produced using the best combination of settings for the exploration parameter K and the learning rate parameter for each variant and each MDP. This figure shows the results for the case, in which the attribute values were precomputed before learning in the same way as described before.

As in the experiments with random MDPs, the performance of recency-based exploration is better overall than the performance of the ϵ-greedy strategy. Similar to the previous experiments, we can see from Figure 4.11 that using the MDP attributes

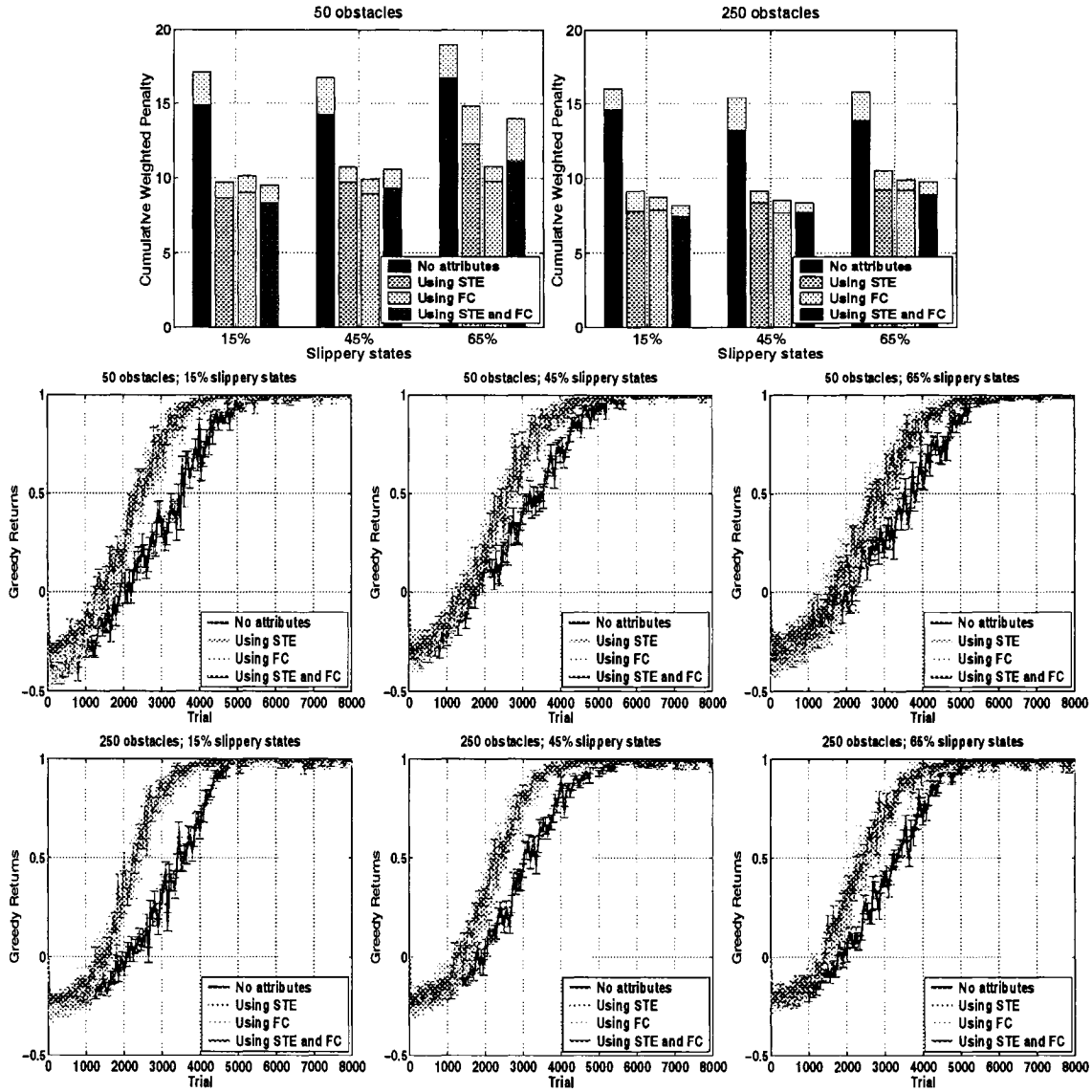


FIGURE 4.10. Performance of ϵ-greedy exploration with and without the use of the MDP attributes on grid-world MDPs. Attribute values were estimated during learning.

improves the performance of the recency-based method. In this case, for most groups, the greatest performance improvement is achieved by using the Forward Controllability. The effect of the State Transition Entropy decreases as the number of slippery states increases. This is consistent with the results on discrete random MDPs, where the strategy using the State Transition Entropy showed a greater impact on low-STE MDPs.

RL algorithm	Sarsa(0)
Value-function representation	Table
Exploration strategy	Recency-based
Exploration parameter settings tested	$K \in \{1, 10, 50\}$
Learning rate schedule	Decreasing: $\alpha_N = \frac{\beta-1}{\beta^{N+1}-1}$
Learning rate settings tested	$\beta \in \{0.75, 0.9, 0.95, 0.98\}$
Number of learning trials per run	10000
Number of runs	20
Frequency of greedy policy evaluation	Every 50 trials
Number of simulated trajectories for greedy policy evaluation	30

TABLE 4.5. Experimental settings for recency-based exploration on random grid-world MDPs.

In summary, the experiments on discrete random MDPs and on gridworld domains showed similar trends in the effect of using the MDP attributes for exploration. First of all, in both cases, incorporation of the attributes into the exploration strategies (both undirected and directed) had a positive effect on learning. We observed that the Forward Controllability has a greater impact in highly stochastic environments with fewer states that are well controllable. The State Transition Entropy exhibits a stronger effect in MDPs, in which state-action pairs with high values of this attribute are less numerous. Using the two attributes together, in the way that it is currently done, does not provide a significant advantage over using one of the attributes alone; future work will address the issue of finding better ways of combining the attributes. Comparing the results on random and gridworld MDPs, we can see that the effect of using the attributes is more pronounced in the latter case: the more structure there is in the environment (which would be expected in real applications) the more beneficial the use of the MDP attributes is.

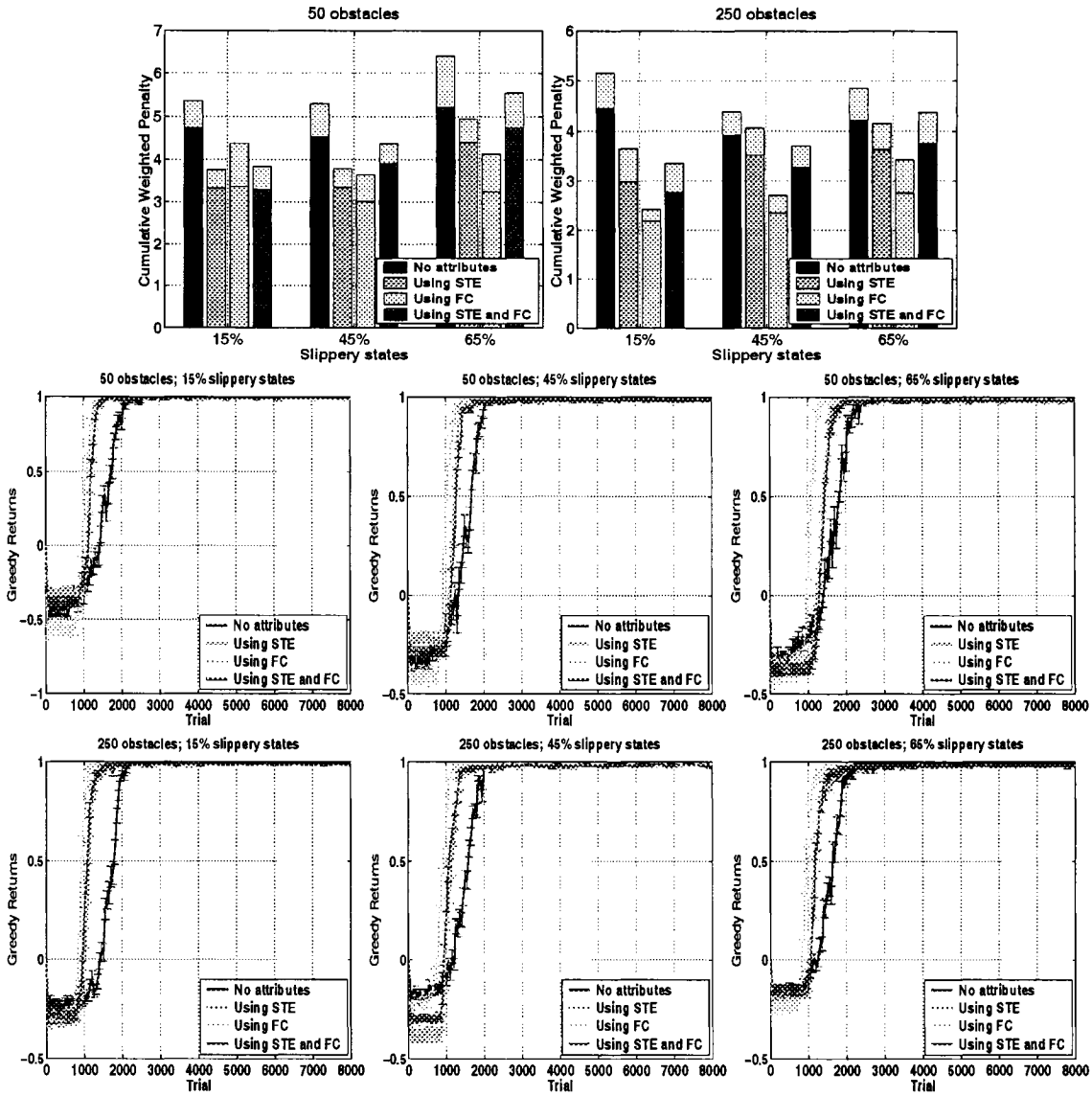


FIGURE 4.11. Performance of recency-based exploration with and without the the use of MDP attributes on grid-world MDPs. Attribute values were precomputed before the onset of learning.

4.5. Summary

In this chapter, we introduced a novel exploration approach, which relies on the State Transition Entropy and the Forward Controllability of state-action pairs in order to decide which states and actions to emphasize. We experimentally demonstrated the feasibility of our approach for improving the performance of both undirected (ϵ-greedy) and directed (recency-based) exploration. This result validates even further our hypothesis, tested in the experimental study presented in Chapter 3, that the State Transition Entropy and the Controllability have a statistically significant effect on learning performance. We conjectured that the use of the two attributes studied facilitates a more homogeneous exploration of the state space, a more extensive sampling of actions which potentially have a high variance of their value samples, and encourages the agent to focus on states where it has most control over the outcomes of its actions. However, more experiments will be performed in the future in order to investigate which of these three factors are responsible most for the improvement in performance and in what circumstances. This will allow us to understand even better how to leverage knowledge of the MDP properties most efficiently and how to combine the two attributes studied together as well as with other MDP related information, e.g., with other MDP attributes presented in Chapter 3, and with the influence of states on one another, as was studied in [Munos and Moore, 1999]. In this case, it will be important to evaluate the trade-off between additional computational requirements, necessary to estimate other properties, and benefits in terms of performance improvement.

Unlike other existing techniques for directed exploration, our method makes exploration decisions independently of the course of learning, based only on properties of the environment. Attribute values can be precomputed before the learning starts, or they can be estimated during learning. In the latter case, the amount of additional storage and computation is lesser compared to other directed techniques. It is also possible to reuse the attribute values for MDPs that differ only in terms of the rewards and not in terms of the state space and the state transition function.

Note that the fact that the MDP attributes are derived from the MDP model should not lead to a conclusion that using model-based methods (i.e., dynamic programming methods based on a known or estimated MDP model) would be preferable. As was discussed in Chapter 2, asynchronous dynamic programming is often used even in the case of complete knowledge of the MDP model in order to focus computational resources on states that are actually observed during the interaction with the environment. In this case, the choice of efficient behavior (exploration) policy is still relevant and important, and thus the use of the attributes can be still beneficial.

In the experiments presented in this chapter, we studied relatively small discrete MDPs. In the future, it would be very interesting to investigate how accurately the MDP attributes can be estimated from data during learning in large and continuous MDPs and what accuracy is necessary for them to be useful. Several results in the current literature (see e.g., [Kearns and Singh, 1999; Kakade, 2003]) indicate that the MDP model does not have to be estimated very accurately in order to be successfully used with model-based methods. We think that a similar conjecture can be made with respect to the MDP attributes and this would be an interesting avenue for future research. In particular, we think that it would be easier to estimate and use the attributes rather than model-based methods for continuous state spaces, especially if the attributes are estimated with respect to the features of a local function approximator used to represent the value function, as discussed in Chapter 3. In this case, it should be possible to extend directed exploration techniques to approximate reinforcement learning methods in a similar way as the use of eligibility traces is successfully extended to function approximation.

Finally, in the future, we would also like to compare the performance of our strategy with other methods, including methods that explicitly estimate the variance of the action-value estimates, for instance, Interval Estimation [Kaelbling, 1993] and its global variant [Meuleau and Bourgine, 1999].

CHAPTER 5

Sparse Distributed Memories for On-Line Value-Based Reinforcement Learning

Chapter Outline

This chapter presents a function approximation model based on Sparse Distributed Memories (SDMs) for use in on-line value-based reinforcement learning. The main contribution of this chapter is a new approach for allocation of the memory resources, which automatically adjusts the memory size and configuration.

5.1. Introduction

In this chapter, once again we focus on on-line value-based reinforcement learning algorithms. As already discussed, in domains with large or continuous state spaces, value functions can be represented by function approximators. The use of function approximators with value-based reinforcement learning algorithms is the subject of much recent research and presents important theoretical and practical challenges. Recall from the discussions in chapters 2 and 3 that reinforcement learning becomes more difficult when values of states or state-action pairs are approximated and generalized across the states. At the same time, the problem of function approximation also becomes more difficult in the context of reinforcement learning. One of the main reasons is in the fact that samples of the target function values do not come from the true optimal value function. They are "guesses" based on the current approximate

value estimates. Since these estimates evolve during learning, the target function is not stationary. Also, in on-line reinforcement learning, the agent typically follows a semi-greedy exploration policy. Its action choices and thus the states that it visits depend on the current value estimates, which change over time. Hence, the input distribution is also non-stationary. Moreover, during on-line learning, training samples are presented to the function approximator as they are encountered on trajectories and thus are not independent of each other. They are also correlated with the approximated values in the case of semi-greedy exploration strategies. The non-stationary nature of the data and the correlated sampling make the function approximation task particularly difficult for theoretical analysis and challenging for practical applications.

Linear, local function approximators are often preferred in value-based reinforcement learning. As was discussed in Chapter 2, convergence properties are best understood for linear approximators, which compute a state (or state-action) value as a linear combination of some features. Relevant results include the convergence of policy evaluation [Tsitsiklis and Van Roy, 1997], the convergence of approximate dynamic programming [Gordon, 1995; Tsitsiklis and Van Roy, 1996], and non-divergence of SARSA(λ) [Gordon, 2000] (see Section 5.3.2 for more details). The behavior of non-linear approximators is still poorly understood in theory, while practical evidence is inconsistent.

Some researchers (see, e.g., [Atkeson *et al.*, 1997a; Platt, 1991; Tham, 1995; Gorivensky and Connolly, 1994]) argue that local approximators are more suitable for reinforcement learning than global approximators. As discussed in Chapter 2, local approximation architectures allow only a few local parameters to be updated on every step, based on a distance from the current input. This is in contrast with global models (e.g., sigmoid neural networks), in which all parameters are updated on every step. Global function approximators, such as neural networks, are known to suffer from *catastrophic forgetting* (or *interference*) [McCloskey and Cohen, 1989; Weaver *et al.*, 1998]: the approximator "forgets" previously learned values of input patterns that are less common or that have not been revisited recently. This happens

because *all* of the architecture parameters get tuned to more recent and frequent training samples. However, in reinforcement learning (and in control in general), some rarely visited states may often be crucial for propagating and maintaining accurate estimates of the value function in other parts of the state space. It has been shown in [Narendra and Parthasarathy, 1991] that using neural networks in control requires a great care in the presentation of training samples, so that *persistent excitation*[1] of the system is ensured and catastrophic forgetting is prevented. Special measures, such as *rehearsal*, are often taken to avoid catastrophic forgetting (see, e.g., [Robins, 1995; Meeter, 2003]).

Local approximators do not suffer from catastrophic forgetting as much, since most parameters learned on previous distant samples are not influenced by the new training data due to the local nature of the parameter updates. At the same time, local approximators quickly incorporate new data, thus adjusting faster to the non-stationarity. In global architectures, on the other hand, a unit with a non-local response needs to undergo gradient descent on many samples for many iterations before a reasonable performance can be expected.

Many reinforcement learning applications have been built around local linear approximators, e.g., CMACs [Santamaria *et al.*, 1998; Sutton and Barto, 1998], soft state aggregation [Singh *et al.*, 1995], variable-resolution discretizations and regression trees [Munos and Moore, 2001; McCallum, 1996; Reynolds, 2000; Uther and Veloso, 1998; Wang and Dietterich, 1999] and memory-based methods [Atkeson *et al.*, 1997a; Forbes, 2002; McCallum, 1996; Santamaria *et al.*, 1998; Smart and Kaelbling, 2000]. Radial Basis Function Networks (RBFNs) with fixed centers and widths have been used much less [Anderson, 1993; Gordon, 1995; Sutton and Barto, 1998; Kretchmar and Anderson, 1997], the main difficulty being in the choice of parameters for the basis functions. Most of these methods, however, still face important difficulties when applied to on-line learning in large domains. For example, CMACs, soft state aggregation and variable-resolution discretization approaches do not scale well to

[1]Persistent excitation means that the set of training samples should *regularly* reveal all modes of the system behavior.

state spaces with high dimensionality; the methods in [Munos and Moore, 2001; Uther and Veloso, 1998; Wang and Dietterich, 1999] are intended for off-line learning; memory-based methods in [Atkeson *et al.*, 1997a; Smart and Kaelbling, 2000] do not address the issue of limiting the memory size, which can grow very big during on-line reinforcement learning.

Global and/or non-linear approximators, e.g., neural networks and Support Vector Machines (SVMs) scale better, in principle, with input dimensionality. However, as mentioned earlier, with on-line reinforcement learning, they have no convergence guarantees and are subject to some other practical problems. For example, as discussed above, neural networks suffer from catastrophic forgetting and are notoriously hard to tune in reinforcement learning. SVMs usually rely on batches of previously seen data [Dietterich and Wang, 2001; Engel *et al.*, 2002; Lagoudakis and Parr, 2003a; Martin, 2002; Ralaivola and d'Alche Buc, 2001] which can be problematic with on-line reinforcement learning due to the non-stationary data distribution. Only recently there was the first attempt to use a sparse incremental SVM method for value-function approximation [Jung and Uthmann, 2004].

In this chapter, we advocate the use of *Sparse Distributed Memories* (SDMs) [Kanerva, 1993] for action-value function approximation in on-line reinforcement learning. SDMs were originally designed for the case, in which a very large input (address) space has to be mapped into a much smaller physical memory. SDMs provide a linear, local architecture, which, due to the reasons discussed above, should be a good choice for on-line reinforcement learning.

In general, local architectures, SDMs included, can be subject to the curse of dimensionality, as an exponential number of local units may be required in order to approximate some target functions accurately across the entire input space. However, many researchers believe (see, e.g., [Atkeson *et al.*, 1997b]) that most decision-making systems need high accuracy only around low-dimensional manifolds of the state space or important state "highways". The SDM model and some related architectures, e.g., Radial Basis Function Networks, are flexible enough to take advantage of this

fact. Their local components can be positioned and shaped according to the needs of the application at hand. Related techniques have already been used in reinforcement learning (e.g., [Anderson, 1993; McCallum, 1996; Wiering, 1999; Smart, 2002; Glaubius and Smart, 2004]), and, in fact, are becoming a hot topic in the current literature.

There are no restrictions in the generic SDM model as to how the physical memory locations should be distributed in the input space. However, the performance of SDMs depends crucially on a good memory layout, which is determined by the distribution of the samples of the target function and its complexity. A number of methods have been developed for the automatic selection of the structure of SDMs and other local models, both in supervised and reinforcement learning. We review in detail such existing approaches in Section 5.4. We then propose a new approach for configuring SDMs, which attempts to avoid some of the pitfalls encountered by existing methods when used in the context of reinforcement learning.

In order to address the issue of finding appropriate memory structure, we turn to ideas underlying instance-based techniques [Atkeson *et al.*, 1997a]. In particular, similar to the instance-based methods, our approach does not require choosing the size or the structure of the approximator in advance, but shapes it on-line based on the distribution of observed data. In practice, our method exhibits robust behavior when used with on-line reinforcement learning, providing good performance as well as being quite efficient both in terms of the resulting memory size and computational time. It also remains close to the scope of existing theoretical convergence guarantees for linear approximators, as we will point out throughout the chapter.

The rest of the chapter is organized as follows. In Section 5.2, we summarize the standard SDM model as introduced in [Kanerva, 1993]. Then we present our implementation of the SDM model for the case of reinforcement learning in continuous state spaces with Least Mean Sqaures (LMS) training. We also discuss approximate reinforcement learning methods that do not use LMS training but with which the SDM model can be used. In Section 5.4, we review methods from the current literature for

automatic selection of structural parameters in SDMs and related models. In Section 5.5, we present the main contribution of this chapter, our approach for dynamic allocation and adjustment of resources in SDMs. Experimental results are presented in Section 5.6. Section 5.7 discusses the directions for future work. We end with conclusions in Section 5.8.

5.2. Sparse Distributed Memory

The Sparse Distributed Memory architecture was originally proposed in [Kanerva, 1988] for learning input-output associations between data drawn from a highly-dimensional binary space. The input can be viewed as a memory "address" and the output is the desired content to be stored at that address. The physical memory available is typically much smaller than the virtual space of all possible inputs, so the physical memory locations have to be distributed sparsely in the virtual address space.

In SDMs, a sample of addresses is chosen (in any suitable manner) and physical memory *locations* are associated only with these addresses. When some address $\mathbf{x} = \langle x_1, ..., x_n \rangle$ (where n is the dimensionality of the input space) has to be accessed, a set of *nearby* locations is activated (see Figure 5.1), as determined by some *similarity measure*. For instance, if addresses are binary, the similarity can be determined using the Hamming distance[2]. The original SDM model assumes that the data to be memorized, $f(\mathbf{x})$, consists of bit vectors (with 0s substituted by -1s). When such a vector $f(\mathbf{x})$ needs to be stored for input (address) $\mathbf{x}$, it is distributed between all the locations activated by $\mathbf{x}$, by performing bitwise addition to the existing content of the activated memory locations. This is in contrast to conventional memories, where addresses have to match exactly and the previous content of a memory location is simply replaced by the new one. When the value for input $\mathbf{x}$ is retrieved, the content of all active locations is combined by bitwise addition and thresholding. In the simplest

[2]The Hamming distance between two binary vectors $\langle x_1, ..., x_n \rangle$ and $\langle y_1, ..., y_n \rangle$ is defined as $HD(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \mathcal{I}(x_i, y_i)$, where $\mathcal{I}(x_i, y_i) = 1$ if $x_i \neq y_i$ and 0 otherwise.

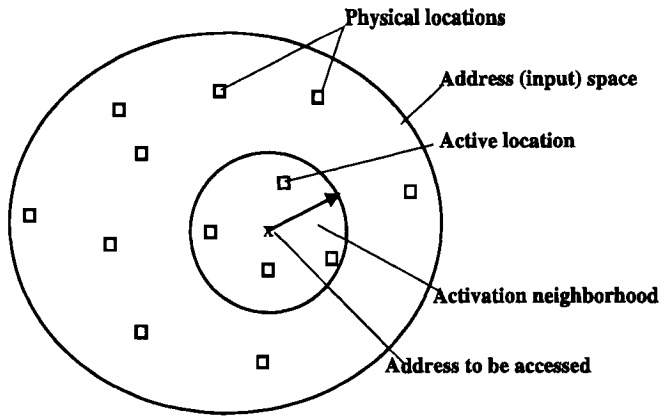


FIGURE 5.1. Sparse Distributed Memory: Generic Design

case, a fixed threshold is used across the entire address space, but more elaborate mechanisms have also been proposed, e.g., [Sjodin, 1995].

A vector $\hat{f}(\mathbf{x})$, retrieved some time after a vector $f(\mathbf{x})$ had been stored for address $\mathbf{x}$ in a distributed manner, should be very close to the original vector. Intuitively, when a vector $f(\mathbf{x})$ is stored, each of the activated locations receives a copy of the data. Subsequently, these active locations may be activated by other inputs $\mathbf{x}'$, and receive copies of other data vectors, but only if $\mathbf{x}$ and $\mathbf{x}'$ are relatively close. So, when we want to retrieve data corresponding to address $\mathbf{x}$, the sum of the contents of activated locations contains all the copies of the vector $f(\mathbf{x})$, originally stored for $\mathbf{x}$, plus copies of other vectors. However, if the memory locations are properly distributed in the input space, these extraneous copies are much fewer than the number of copies of the vector $f(\mathbf{x})$. This biases the sum vector in the direction of the vector $f(\mathbf{x})$ with high probability. This property of distributed storage has been rigorously studied based on signal-to-noise ratio analysis in [Kanerva, 1993].

In this chapter, we focus on the case where inputs are vectors of real values and the target values stored in memory are also real numbers. In other words, we are interested to approximate functions $f(\mathbf{x}) : \Re^n \rightarrow \Re$ with SDMs. In the remainder of this section, we present different aspects of the implementation of SDMs for this case.

5.2.1. Activation Mechanism

As mentioned above, the SDM activation mechanism relies on the similarity measure between the input address and the addresses of memory locations. In general, the choice of a similarity function depends on the input space and the application at hand. For example, the original SDM design for binary memories used a similarity mechanism based on thresholding the Hamming distance (HD) between binary vectors: a memory location $\mathbf{h} = \langle h_1, ..., h_n \rangle$ was considered activated by an input address $\mathbf{x} = \langle x_1, ..., x_n \rangle$ if $HD(\mathbf{h}, \mathbf{x}) > \eta$, where η is an integer threshold. The activation status of the memory locations is binary in this case.

A continuous activation mechanism was proposed for binary SDMs in [Hely *et al.*, 1997], which was modeled as if the input address would broadcast a signal in a radial manner. This signal loses strength by a certain percentage every time a new location is encountered. The activation status of each location is equal to the strength of the received signal. With this activation mechanism, the similarity is not directly related to the distance between inputs in some metric space, but is dependent on the local density and distribution of locations.

For presentation purposes and for our experiments, we chose a similarity measure based on symmetric triangular functions. This similarity between input vector $\mathbf{x} = \langle x_1, ..., x_n \rangle$ and location $\mathbf{h} = \langle h_1, ..., h_n \rangle$ is computed as follows:

$$\begin{aligned} \mu(\mathbf{h}, \mathbf{x}) &= \min_{i=1,...,n} \mu_i(\mathbf{h}, \mathbf{x}) \\ \mu_i(\mathbf{h}, \mathbf{x}) &= \begin{cases} 1 - \frac{|x_i - h_i|}{\beta_i} & \text{if } |x_i - h_i| \leq \beta_i \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{5.1}$$

Here, $\langle h_1, ..., h_n \rangle$ represents the *location address* and β_i are the *activation radii* in each dimension (see Figure 5.2). This similarity measure directly translates into the location's degree of activation, which, in this case, is continuous in the interval $[0, 1]$. This definition of the similarity function is factored, in the sense that the similarities in each dimension are combined in a "product", by the min operator in this case. This gives an immediate symbolic interpretation of the memory locations' semantics

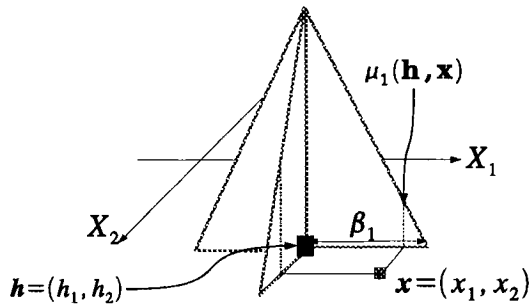


FIGURE 5.2. Similarity function for two dimentional space based on triangular symmetric uni-dimensional functions.

and, if needed, allows to extract and refine knowledge in terms of interpretable rules. From this point of view, our model resembles Factorized RBFNs [Tresp *et al.*, 1992]. It should be noted that taking the minimum similarity over *all* dimensions may be too demanding in highly dimensional input spaces with very sparse data. In this case, a non-zero similarity μ_i can be required only for a certain number of dimensions[3].

Of course, the similarity measure can be defined in many different ways; see, e.g., [Atkeson *et al.*, 1997a; Scholkopf, 2000] for different choices. For instance, popular Gaussian functions can be used instead of (5.1). Our choice of the similarity measure based on the triangular functions presented in Equation (5.1) was motivated mainly by the fact that a similarity function with a bounded support facilitates implementation of an efficient activation mechanism, as will be discussed below.

In the original SDM model, on every memory access, the similarity of the input address to all existing locations is computed in order to find which locations are activated. The same is usually true for the implementations of RBFNs. This step can be very computationally expensive for large architectures. A similar problem arises for instance-based learning methods, which also have to retrieve previously stored data samples that are close to the query point.

It is possible to implement SDMs and other related models efficiently so that isolating active locations does not require computing the similarity of a data point to all locations. For example, for the binary SDM model, a more efficient mechanism was

[3]We do not use such a definition in this thesis.

proposed in [Karlsson, 1995]. In this mechanism, memory locations are divided into a fixed number of blocks, equal to some predefined number of desired active locations. Each block has an associated filter, a subset of k address bits that are required to match, and each block consists of 2^k hard locations that have all combinations of 0s and 1s in the k specified address bits. On a memory access, the input values of the bits specified in the filters are used as indeces to the corresponding blocks of hard locations, thus speeding up the computation. The disadvantage of this approach is that the positions of hard locations are predetermined by the selection of filters, which may be problematic for many distributions of the data.

In the instance-based learning framework, *kd*-trees are often used [Freidman *et al.*, 1977; Deng and Moore, 1995; Smart and Kaelbling, 2000] to retrieve efficiently from a database the samples close to a query. A *kd*-tree can be used to divide the input space into hyper-rectangles corresponding to the leaves. Then the leaves of the tree contain pointers to the memory locations whose activation neighborhoods overlap with the corresponding hyper-rectangles. Another similar data structure, balltrees, can also be used in this context [Omohundro, 1989]. Voronoi diagrams, commonly used in computational geometry [Okabe *et al.*, 1992], can provide an efficient search mechanism. However, the size of the corresponding data structure is exponential in the dimensionality of the input space.

5.2.2. Reading from Memory: Prediction

Let $\mu^k(\mathbf{x}) = \mu(\mathbf{h}^k, \mathbf{x})$ be the similarity between input $\mathbf{x}$ and the k^{th} location $\mathbf{h}^k$, as in Equation (5.1). We denote by M the total number of memory locations in the SDM. To predict the value of the input $\mathbf{x}$, we first find the *set of active locations*, $H_\mathbf{x}$:

$$H_\mathbf{x} = \left\{ k \mid \mu^k(\mathbf{x}) = \mu(\mathbf{h}^k, \mathbf{x}) > 0 \right\}, k = 1, ..., M \tag{5.2}$$

Let $\mathbf{w} = (w_1, ..., w_M)$ be the values stored at the corresponding memory locations $\mathbf{h}^1, ..., \mathbf{h}^M$. Then, the predicted value of $\mathbf{x}$ is computed as follows:

$$f_{\mathbf{w}}(\mathbf{x}) = \frac{\sum_{k \in H_{\mathbf{x}}} \mu^k(\mathbf{x}) w_k}{\sum_{k \in H_{\mathbf{x}}} \mu^k(\mathbf{x})} \tag{5.3}$$

The normalized activations of the memory locations, $\frac{\mu^m(\mathbf{x})}{\sum_{k \in H_{\mathbf{x}}} \mu^k(\mathbf{x})}$, can be viewed as features of the input $\mathbf{x}$. Hence, the prediction is computed as a linear combination of local features.

The representation (5.3) of the approximate function, which we use for continuous input spaces, is equivalent to Normalized RBFNs (NRBFNs), which we briefly introduced in Chapter 2. The addresses of the SDM locations correspond to the centers of the RBFs and the activation radii - to the widths of the RBFs. The similarity measures of the SDMs correspond to the basis functions. The parameters of NRBFNs used to combine RBFs in a linear manner correspond to the values w_m stored in the memory locations.

Other linear local architectures can be related to this model as well. For example, recall from Chapter 2 the CMAC approximator. In CMACs, tiles correspond to the memory locations. As in the SDM model, several tiles get activated on every memory access, one tile in each tiling. The activation mechanism is, however, binary in the CMAC case. This is equivalent to having a rectangular-shaped similarity function, which is equal to one over an entire activation neighborhood (a region covered by the corresponding tile) and which drops to zero at the tile's borders. The parameters associated with each tile correspond to the values stored in the SDM locations.

Analogies can also be drawn with the manifold representation for function approximation, described in Chapter 2. In this case, memory locations correspond to the charts in the manifold atlas. The similarity measures correspond to the blend functions and the values stored at the memory locations are equivalent to using a constant approximating function within each chart.

An extensive review of various RBFN architectures and their relationships to various other models, including, among others, support vector machines, fuzzy systems, wavelet networks and instance-based learning, can be found in [Blanzieri, 2003]. The most important differences between such related models appear in the learning algorithms associated with them.

5.2.3. Storing in Memory: Least Mean Squares Learning for Continuous SDMs

Upon receiving a training sample $\langle \mathbf{x}, f(\mathbf{x}) \rangle$, the values stored in all active locations are updated. We perform the updates based on the standard incremental gradient descent algorithm for linear Least Mean Squares approximation [Widrow and Hoff, 1960]. As already discussed in Chapter 2, in this case, the goal is to minimize the Mean Squared Error:

$$MSE_{\mathbf{w}} = \frac{1}{2} \sum_{\mathbf{x}} P(\mathbf{x}) \left[f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x}) \right]^2 \tag{5.4}$$

where $f_{\mathbf{w}}(\mathbf{x})$ is the prediction for input $\mathbf{x}$, $P(\mathbf{x})$ is the probability or weight of sample $\mathbf{x}$ and the summation runs over the training samples received during learning.

During on-line learning, training samples are considered to be drawn from the distribution of interest, $P(\mathbf{x})$, and are processed one at a time. In this case, we consider the estimates of the MSE function at each training sample. In order to find the approximation $f_{\mathbf{w}}(\mathbf{x})$ that minimizes the MSE function, the parameters of the function approximator $\mathbf{w} = (w_1, ..., w_M)$ are updated in the direction of the negative gradient of the MSE estimate for the current sample $\langle \mathbf{x}, f(\mathbf{x}) \rangle$:

$$w_m := w_m - \alpha \left. \frac{\partial MSE_{\mathbf{w}}}{\partial w_m} \right|_{\mathbf{x}}, \; m = 1, ..., M \tag{5.5}$$

where $\alpha \in (0, 1)$ is the learning rate (which can vary from one iteration to the next). The partial derivatives of the MSE function with respect to the approximator's parameters w_m are computed as follows:

$$\begin{aligned} \left.\frac{\partial MSE_{\mathbf{w}}}{\partial w_m}\right|_{\mathbf{x}} &= -\left[f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x})\right] \left.\frac{\partial f_{\mathbf{w}}}{\partial w_m}\right|_{\mathbf{x}} \\ &= -\left[f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x})\right] \frac{\mu^m(\mathbf{x})}{\sum_{k \in H_{\mathbf{x}}} \mu^k(\mathbf{x})} \end{aligned} \tag{5.6}$$

Thus, from (5.5) and (5.6), the values stored in the SDM locations are updated as follows:

$$w_m := w_m + \alpha \left[f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x})\right] \frac{\mu^m(\mathbf{x})}{\sum_{k \in H_{\mathbf{x}}} \mu^k(\mathbf{x})}, \forall m \in H_{\mathbf{x}} \tag{5.7}$$

In Section 5.5.5, we will discuss how the addresses of the memory locations can be selected and updated. The memory allocation process will then proceed in parallel with learning the memory content.

5.3. SDMs in Reinforcement Learning

5.3.1. Least Mean Squares Training for SDMs in Reinforcement Learning

SDMs with the standard incremental LMS training, as explained above, can be incorporated into value-based reinforcement learning algorithms in a straightforward way. For instance, in order to combine SDMs with SARSA(λ), we can use a separate approximator to represent the action-value function $Q_{\mathbf{w}}(s, a)$ for each action a, as is typically done with other approximators [Sutton and Barto, 1998]. Let $w_m(a)$, $m = 1, ..., M_a$, denote the values stored in the locations of the SDM for action a. These values are updated after every observed transition $(s, a) \xrightarrow{r} (s', a')$. The current state-action pair (s, a) serves as the training input and the target value is obtained from the bootstrapped estimate $[r + \gamma Q_{\mathbf{w}}(s', a')]$, i.e., a sample of a "one-step-lookahead" based on the observed reward r, the next state s', the selected next action a' and the current prediction for the value of the next state-action pair (s', a'). The update rule, corresponding to the standard LMS update in Equation (5.7) for the case when

$\lambda = 0$ can be obtained as:

$$w_m(a) := w_m(a) + \alpha[r + \gamma Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a)] \frac{\mu^m(s, a)}{\sum_{k \in H_s} \mu^k(s, a)}, \; m = 1, ..., M_a \quad (5.8)$$

If eligibility traces are used, i.e., $\lambda > 0$, the training update will be as follows:

$$w_m(\bar{a}) := w_m(\bar{a}) + \alpha e_m(\bar{a})[r + \gamma Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a)], \; \forall \bar{a} \in A \text{ and } m = 1, ..., M_{\bar{a}} \quad (5.9)$$

Here, $e_m(\bar{a})$ are the eligibility traces associated with each location, similar to the tabular representation of action-value functions. In the tabular case, they accumulate the visitation counts for each state-action pair, whereas with function approximation, they accumulate gradients of the approximate function $Q_{\mathbf{w}}(s, a)$ with respect to the parameters. In our case, these gradients correspond to the terms $\frac{\mu^m(s,a)}{\sum_{k \in H_{\mathbf{x}}} \mu^k(s,a)}$ from equation (5.8). Similar to the tabular version of the Sarsa(λ) algorithm, these gradients represent how much the corresponding parameter values contributed to the approximation error $[r + \gamma Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a)]$. Recall from Chapter 2 that eligibility traces can be updated using either the replacing traces or accumulating traces method. For replacing traces, the trace update is as follows:

$$\begin{aligned} e_m(\bar{a}) &:= \lambda \gamma e_m(\bar{a}) && m = 1, ..., M_{\bar{a}} \text{ and } \forall \bar{a} \\ e_m(a) &:= \frac{\mu^m(s,a)}{\sum_{k \in H_s} \mu^k(s,a)} && \forall m \in H_{s,a} \\ e_m(\bar{a}) &:= 0 && \forall m \in H_{s,\bar{a}} \text{ and } \forall \bar{a} \neq a \end{aligned} \quad (5.10)$$

For accumulating traces, the eligibilities are updated as follows:

$$e_m(\bar{a}) := \lambda \gamma e_m(\bar{a}) + \begin{cases} \frac{\mu^m(s,a)}{\sum_{k \in H_{s,\bar{a}}} \mu^k(s,a)} & \text{if } m \in H_{s,\bar{a}} \text{ and } \bar{a} = a \\ 0 & \text{otherwise} \end{cases} \quad (5.11)$$

If the memory is big, a list of locations with traces greater than some threshold can be maintained in order to perform this update efficiently.

5.3.2. Alternatives to LMS Training with Non-Expansive Approximators

The convergence of reinforcement learning algorithms in the tabular case [Jaakkola *et al.*, 1994; Singh *et al.*, 2000] can be shown based on the contraction property of

the corresponding value update operators, (recall Equations (2.14), (2.17) and (2.28) from Chapter 2). This property cannot be guaranteed, in general, when reinforcement learning is used with arbitrary function approximators, both in the case of using LMS training and approximate dynamic programming. However, there are several studies in the current literature for various special cases. We will review some of them next.

Approximate Value Iteration with Averagers

The work in [Gordon, 1995] addressed the question of which function approximators can preserve the contraction property when combined with approximate dynamic programming. He analyzed a class of methods that became known as *averagers* in the reinforcement learning community. These methods have a non-expansion property, which means that the approximator cannot extrapolate off the range of the training *target* values[4]. A function represented by an averager is entirely contained within the vertical bounds of the target function. This property cannot be ensured when using the LMS rule, which can minimize the MSE while predicting values out of the range of the observed target values for some inputs.

Consider an approximation $\hat{V}(s)$ of the state-value function. As defined in [Gordon, 1995], a real-valued function approximation scheme is an averager if every approximated value $\hat{V}(s)$ is a weighted average of zero or more target values $V(s^m)$ and possibly some predetermined constant c:

$$\begin{aligned} \hat{V}(s) &= c\phi^0(s) + \sum_{m=1}^{M} \phi^m(s) V(s^m) \\ \phi^0(s) &+ \sum_{m=1}^{M} \phi^m(s) = 1 \end{aligned} \tag{5.12}$$

where $\langle s^m, V(s^m) \rangle$, $m = 1, ..., M$, are the training samples. The weights $\phi^m(s)$ involved in calculating the approximate value may depend on the input vector s but may not depend on the target values $V(s^m)$. For instance, these weights cannot be

[4]This notion of non-expansion is not to be confused with the notion of extrapolation off the range of training *inputs*.

determined by the LMS training rule. Usually these weights are determined by distances between the query input s and the training inputs s^m and are unaffected by the training target values.

One well-known method that operates in this way is the k-nearest neighbor, where weights $\phi^m(s)$ are often determined by a Gaussian kernel:

$$\phi^m(s) = \exp(\frac{-\|s - s^m\|_\xi^2}{\sigma^2}) \tag{5.13}$$

based on a weighted Euclidean distance $\|s - s^m\|_\xi^2 = \sum_{i=1}^n \xi_i(s_i^m - s_i)^2$, The weights ξ_i allow appropriate rescaling of the input dimensions and are usually specified by the user. The parameter σ is the kernel width, which determines how fast the weight $\phi^m(s)$ decreases with the increase in the Euclidean distance. In the k-nearest neighbor algorithm, the k training samples with the largest values of the weights $\phi^m(s)$ appear in the summation in Equation (5.12)[5]. Other methods that have been used in this context are bilinear interpolation [Gordon, 1995], interpolation based on multidimensional triangulations [Davies, 1996] and barycentric interpolation [Munos and Moore, 1999].

Note, that some analogies can be drawn between the averager and the SDM models. The weights $\phi^m(s)$ of the averager play the same role as the normalized similarity functions $\frac{\mu^m(s)}{\sum_{k\in H_s}\mu^k(s)}$ in the SDM model. In the averager model, training targets $V(s^m)$ play the same role as the values w_m stored in the SDM locations. If the SDMs were to be used in the averager framework, the addresses of the memory locations would be chosen exactly at the training inputs, i.e., $\mathbf{h}^m = s^m$, $m = 1, ..., M$. Also, memory content parameters w_m would be directly determined by the training targets: $w_m = V(s^m)$, $m = 1, ..., M$. No adaptive learning algorithm, like LMS iterative updates (5.7), would be performed to estimate the parameters w_m.

In [Gordon, 1995], averagers were used in combination with the Value Iteration algorithm as discussed in Chapter 2. This is possible when the model of the process is available or when transition samples can be generated at will from any state.

[5]Kernels of the nearest neighbors can be normalized so that they sum to one.

Recall that with approximate dynamic programming, a subset of prototype states $\bar{S} = \{s^1, ..., s^M\}$ is selected, and their values are arbitrarily initialized to some values $V^0(s^m)$, $m = 1, ..., M$. On every iteration t, the training set, consisting of samples $\langle s^m, V^t(s^m) \rangle$, $m = 1, ..., M$, is presented to an averager which generalizes the values of the states in $\bar{S}$ to the entire state space, as in Equation (5.12). Then, one iteration of the Value Iteration method is performed for the states in $\bar{S}$ only, and the next estimate of the value function, $V^{t+1}(s^m)$, is obtained as follows:

$$V^{t+1}(s^m) := \max_a \sum_{s'} P^a_{s^m,s'} \left[R^a_{s^m,s'} + \gamma V^t(s') \right] \ , m = 1, ..., M \tag{5.14}$$

Note that the successors s' may be outside of the set $\bar{S}$ of prototype states, in which case the current approximation, as given by Equation (5.12), is used to estimate their values. This method is practical only if the set of possible successors is reasonably small so that the summation in Equation (5.14) can be computed efficiently. Approximate Value Iteration combined with an averager for approximation of the state-value fucntion converges with probability one to a value function V^∞, such that

$$\max_{s \in S} |V^\infty(s) - V^*(s)| \leq \frac{2\gamma\eta}{1-\gamma} \tag{5.15}$$

where $\eta = \max_{s \in S} |V^A(s) - V^*(s)|$, and V^A is the best possible approximation to the optimal value function for the given averager scheme and the given set of prototype states $\bar{S}$.

Approximate Value Iteration with Linear Architectures

Another related result was obtained in [Tsitsiklis and Van Roy, 1996]. It proved convergence of the Value Iteration method, applied in the same manner as in [Gordon, 1995] (discussed above) with a general feature-based class of linear function approximators used to represent the optimal state-value function:

$$V_{\mathbf{w}}(s) = \sum_{m=1}^{M} w_m \phi^m(s) \tag{5.16}$$

where $\phi^m(s)$ are some features or basis functions. In this case, the Value Iteration algorithm operates on a set of preselected prototype states $\{s^1, ..., s^M\}$ as presented in Equation (5.14), similar to the approach of [Gordon, 1995]. The features have to satisfy the following conditions[6]:

$$\begin{aligned} &(i) \quad \text{The feature vectors } F(s^m) = \langle \phi^1(s^m), ..., \phi^M(s^m) \rangle,\ m = 1, ..., M \\ &\qquad \text{are linearly independent;} \\ &(ii) \quad \text{The feature space is contained in the convex hull of} \\ &\qquad \text{the vectors } \pm F(s^1), ..., \pm F(s^M), \text{ possibly with a small} \\ &\qquad \text{amount of slack to extend it slightly beyond that hull.} \end{aligned} \tag{5.17}$$

These conditions ensure that the extrema of the approximate value function, represented by the linear model (5.16) within a convex polyhedron, must be located at the corners:

$$\max_{s \in S} |V_{\mathbf{w}}(s)| < \frac{\gamma'}{\gamma} \max_{m=1,...,M} |V_{\mathbf{w}}(s^m)| \tag{5.18}$$

where the factor $\gamma' \in [\gamma, 1)$ allows for a small amount of slack in the bounds. Feature-based linear function approximators that satisfy the above conditions are non-expansions and preserve the contraction property of the Value Iteration method. They ensure convergence with error bounds comparable to those derived by Gordon (1995) (see Equation (5.15)).

Tsitsiklis and Van Roy (1996) analyzed a special case of linear approximators (5.16), in which features $\phi^m(s)$ are localized basis functions, e.g., Gaussians or triangular similarity functions (5.1) presented earlier for our SDM model. These basis functions are associated with particular prototype states that represent their centers. Under certain conditions, such approximators satisfy the non-expansion property with the Value Iteration method. More specifically, assume that we have basis functions $\phi^1(s), ..., \phi^M(s)$ centered at some prototype states $s^1, ..., s^M$. The required conditions

[6]See [Tsitsiklis and Van Roy, 1996] for exact technical definition.

are then as follows:

(i) For all $m \in \{1, ..., M\}$, $\phi^m(s^m) = 1$;

(ii) For all $m \in \{1, ..., M\}$, $\sum_{\substack{i=1 \\ i \neq m}}^{M} |\phi^i(s^m)| < 1$;

(iii) With δ defined by $\delta = \max_{m \in \{1,...,M\}} \sum_{\substack{i=1 \\ i \neq m}}^{M} |\phi^i(s^m)|$, (5.19)

there exists a constant $\gamma' \in [\gamma, 1)$, such that for all $s \in S$,

$$\sum_{m=1}^{M} |\phi^m(s)| \leq \frac{\gamma'}{\gamma}(1 - \delta)$$

Intuitively, condition (ii) bounds the influence of other basis functions on the value at the center of a particular basis function. Condition (iii) is necessary to ensure that the feature space defined by the basis functions is contained within a convex hull and that the approximate value function is always bounded as in Equation (5.18). Often a basis function centered at a prototype state s^m can be interpreted as providing a similarity between other states and the prototype state s^m (as is the case with the SDM model). In this case, given the maximum contribution δ of other basis functions at the center of any basis function, condition (iii) will establish a maximum contribution of all basis functions to the value at any state s.

Unfortunately, no constructive algorithm was ever suggested for choosing the centers and the widths of the basis functions so that these conditions are satisfied and the obtained approximator structure is suitable for the domain at hand[7].

As previously noted, the SDM architecture provides a linear local approximator, where the similarity functions can be viewed as basis functions. Thus, the SDM architecture can be configured to satisfy the conditions (5.19) so that the non-expansion property, required to guarantee convergence of the approximate Value Iteration, is

[7]Goodness of the obtained architecture for a particular task will be determined by the bounding factor $\eta = \max_{s \in S} |V_{\mathbf{w}^*}(s) - V^*(s)|$ where $V_{\mathbf{w}^*}(s)$ is the best possible approximation of the optimal state-value function, given the chosen architecture. This factor determines the error bound for the approximate value function estimate obtained by the approximate Value Iteration, similar as in Equation (5.15).

not violated. In Section 5.2.2, we suggested to use an SDM representation of the approximate function based on the normalized similarity measures $\frac{\mu^m(s)}{\sum_{k \in H_s} \mu^k(s)}$. In this case, conditions (i) and (ii) in (5.19) can be satisfied only if the address of any location is never covered by the activation neighborhood of any other location. In this case δ, as defined in (5.19), is equal to zero and the condition (iii) can always be satisfied. Thus, when building an SDM, we would have to ensure that a new location is added only if its address does not activate any other existing location. Hence, with such an architecture, there are points in the input space, whose values are determined by a single memory location. A set of such points will always contain points corresponding to addresses of the memory locations, plus possibly some nearby points, depending on the exact layout of all the activation neighborhoods. To some extent, this defies the idea of *distributing* the values stored in the memory over a number of locations, which is supposed to improve its robustness and noise tolerance.

Suppose that we would like to allow activation neighborhoods to cover the addresses of other locations, in order to better ensure the distribution of the memory content, while still trying to satisfy the rest of the conditions (5.19). In this case, we could use un-normalized similarities in the representation of the approximate function. In the case of the state-value function, the representation would be as follows:

$$V_{\mathbf{w}}(s) = \sum_{m \in H_s} \mu^m(s) w_m \tag{5.20}$$

Condition (i) in (5.19) would be satisfied with any similarity function μ with the range of $[0, 1]$. Then, we would need to find an appropriate threshold μ^* for the similarities (closeness) between any two memory locations, in order to satisfy other conditions in Equation (5.19).

It would be difficult to find analytically the design parameters so that conditions (5.19) are guaranteed to be satisfied. We can try to satisfy them approximately instead. For instance, we could sample the state space in some manner and incrementally add a new location with the address equal to the current input s whenever the total activation of the existing locations, $\sum_{m=1}^{M} \mu^m(s)$, $m = 1, ..., M$, does not exceed

some threshold $\delta < 1$. This would ensure satisfaction of condition (ii). To satisfy condition (iii) for *all* states s would be more difficult, though. As a rough approximation, we could check this condition for every sample state and, if it is violated, remove some of the existing locations. Effectively, condition (iii) will impose a limit on the number of memory locations that could be active in any given region of the state space.

The above raises a natural question of the choice of the sampling distribution that should be followed during the memory allocation process. The results in [Gordon, 1995] and [Tsitsiklis and Van Roy, 1996] discussed above were given for the Value Iteration algorithm, which is applied off-line by performing updates (5.14) synchronously to all preselected representative states $\{s^1, ..., s^M\}$ (in our case, they would correspond to the SDM locations). Thus, the architecture of the function approximator has to be fixed prior to learning. Ideally, we would be interested to allocate the memory under the distribution of an optimal policy, which, however, is not known. An obvious feasible alternative would be a uniformly random policy. In this case, we could aim at simply covering the entire state space with activation neighborhoods. However, it would not be possible to guarantee that the resulting memory layout is actually well suited for the target state-value function. Moreover, in very large, high-dimensional domains, it may not be possible to cover the entire state space. In this case, as mentioned previously, when using local function approximators, the hope is that it is not necessary to cover the entire state space, but only important regions. Unfortunately, it is not obvious how to identify such regions prior to learning, unless some domain knowledge is available. Alternatively, they could be identified while following on-policy exploration during learning.

On-line Algorithms with Non-Expansive Approximators

Gordon (1995) discussed the possibility of using an incremental version of the Value Iteration, as well as an algorithm analogous to the Value Iteration but based on the Bellman equation for the action-value function $Q(s, a)$. However, even these

incremental variants rely on the availability of the MDP model. When the MDP model is not known, the agent can use the on-line model-free Q-learning to update the values of prototype state-action pairs:

$$Q_{\mathbf{w}}(s^m, a) := Q_{\mathbf{w}}(s^m, a) + \alpha \left[r + \gamma \max_b Q_{\mathbf{w}}(s', b) \right] \tag{5.21}$$

while representing the action-value function with an averager:

$$Q_{\mathbf{w}}(s, a) = c\phi^0(s, a) + \sum_{m=1}^{M} \phi^m(s, a) Q_{\mathbf{w}}(s^m, a) \tag{5.22}$$

The convergence result of Gordon (1995) would still hold, if the updates to the values of prototype states s^m were performed conservatively only when these states are actually observed, that is when $s^m = s$ and s is the currently observed state. However, this may happen very rarely, especially in large stochastic environments, and thus learning will be very slow. The idea that was proposed in [Gordon, 1995] for this case is to pretend that the observed transition from some state s is actually from the "closest" prototype state s^m. Then, the update (5.21) is performed for the value of such state-action pair (s^m, a), where a is the current action.

A similar idea was used in [Ribeiro and Szepesvári, 1996; Szepesvári and Littman, 1999] in order to speedup tabular Q-learning in Eucledian state spaces, where after a state transition from some state s, an update was performed to multiple states $\bar{s}$, considered "close" to the observed state s. Closeness was determined according to some smoothening factor $z(\bar{s}, a, s)$, such that $z(\bar{s}, a, s) \leq z(\bar{s}, a, \bar{s})$ and such that it decays to zero as $\|\bar{s} - s\| \to \infty$. So, after observing a transition $(s, a) \xrightarrow{r} s'$, the update was be performed as follows:

$$Q(\bar{s}, a) = Q(\bar{s}, a) + \alpha z(\bar{s}, a, s) \left[r + \gamma \max_b Q(s', b) - Q(\bar{s}, a) \right] \text{, for all } \bar{s} \in S \tag{5.23}$$

In [Szepesvàri, 2001], the algorithm with such an update rule was analyzed with interpolative function approximation used to represent the action-value function. Interpolative function approximators can be seen as a special case of the feature-based

methods considered in [Tsitsiklis and Van Roy, 1996]. An interpolative function approximator for the action-value function $Q_{\mathbf{w}}(s,a)$, has a set of parameters $w_m(a)$, $m = 1, ..., M$, associated with a set of prototype states $\{s^1, ..., s^M\}$ (we assume that there is a separate approximator for each action a). The interpolative approximator satisfies the following property:

$$\begin{array}{ll} (i) & Q_{\mathbf{w}}(s,a) = \begin{cases} w_m(a) & \text{if } s = s^m \\ \sum_{m=1}^{M} w_m(a)\phi^m(s,a) & \text{otherwise} \end{cases} \\ (ii) & \phi^m(s,a) \geq 0,\ m = 1, ..., M \\ (iii) & \sum_{m=1}^{M} \phi^m(s,a) = 1 \end{array} \tag{5.24}$$

Note that the SDM architecture can be structured to satisfy the above interpolative property. In this case, the addresses of memory locations $\mathbf{h}^m$ would correspond to the prototype states s^m. Using normalized similarities in the representation of the approximate function, as in Equation (5.3), assures that condition (iii) in (5.24) is satisfied. In order to satisfy condition (i), it is necessary to ensure that the address of any memory location is not covered by an activation neighborhood of any other location, that is $\mu^i(\mathbf{h}^m) = 0$, $i \neq m$, $i, m \in \{1, ..., M\}$.

Consider using the same update rule as in Equation (5.23) in order to update the values of the prototype states $s^1, ..., s^M$ after every transition $(s,a) \xrightarrow{r} s'$ observed on-line. In that case, taking into account that $Q(s^m, a) = w_m(a)$, the update rule for the function approximation parameters $w_m(a)$ is easily obtained as follows:

$$w_m(a) := w_m(a) + \alpha z(s^m, a, s) \left[r + \gamma \max_b Q(s', b) - w_m(a) \right] ,\ m = 1, ..., M \tag{5.25}$$

That is, the values of several prototype states are updated depending on their similarity to the observed state, as defined by the factor $z(s^m, a, s)$. In the sequel, we will refer to this update rule as the *averager update*[8].

Note that here, the smoothening factor $z(s^m, a, s)$ does not necessarily have any relation to the features $\phi^m(s,a)$ used for interpolation. A similar approach was used

[8]This term was suggested in [Reynolds, 2002].

in [McCallum, 1996] with the k-nearest neighbor instance-based method. In that work, a binary factor was used in place of the smoothening factor $z(s^m, a, s)$, which indicated whether the corresponding prototype state s^m was used in the k-nearest neighbor calculation of the predicted action-value $Q_{\mathbf{w}}(s, a)$. The same approach was used in [Smart and Kaelbling, 2000] with instance-based learning and locally weighted regression.

The same update rule but with $z(s^m, a, s) = \phi^m(s, a)$ was also studied in [Reynolds, 2002]. It was shown that a linear interpolative function approximator, which is trained according to the rule (5.25) and with an arbitrary training distribution, cannot diverge with any one-step reinforcement learning algorithm (i.e., without eligibility traces). A stronger result for the interpolative approximation was obtained in [Szepesvàri, 2001], where the algorithm based on the update (5.25) was proved to converge with probability one provided that the set of prototype states $\{s^1, ..., s^M\}$ is fixed ahead of time and that the agent follows a fixed stochastic exploration strategy. This result was extended in [Szepesvári and Smart, 2004] to the case when the set of prototype states is adapted during learning under the same assumption of a fixed stochastic exploration policy. We will discuss this result in more detail in Section 5.5.

Although the averager update rule has been shown to have stronger theoretical convergence properties than LMS training for the on-line control problem, there is currently little empirical evidence for the practical performance of this method and how it compares to LMS training when the latter is stable. The experiments with the averager learning rule applied with memory-based learning in [Smart and Kaelbling, 2000] showed that using this rule without extra measures resulted in a very poor performance (e.g., learned strategies were only slightly better than the uniformly random policy on the Mountain Car domain). Good performance in [Smart and Kaelbling, 2000] was obtained only when on every prediction, two additional things were performed. First, it was checked whether the state for which the value had to be predicted was well covered by training samples stored in the memory (these samples were

required to form a convex hull around the query state). Second, in addition to updating the values of stored instances using the averager rule in Equation (5.25), locally weighted regression was performed for every prediction. The latter is actually quite expensive computationally: the number of predictions that have to be made during reinforcement learning is very high due to the fact that the values of *all* actions have to be computed on every time step to identify a greedy action. Essentially, it seemed that the averager rule ensured only that values stored in the memory for the training instances are kept relatively up-to-date in the face of non-stationarity of the target function. However, this rule did not provide an approximation of the action-value function that was ready-to-use at the time of prediction. As indicated in [Reynolds, 2002], the averager update in Equation (5.25) has a tendency to over-smooth the target function, which could be one of the reasons for the poor performance.

In summary, there is not enough evidence at present to prefer the averager update to the LMS update in practice, even though the former has theoretical convergence guarantees, while the latter does not have such guarantees.

Least-Squares Policy Iteration

Yet another alternative to standard incremental LMS training in reinforcement learning was introduced in [Lagoudakis and Parr, 2003b]. It is known as the Least Squares Policy Iteration (LSPI) method and is an extension of the Least Squares Temporal Difference method (LSTD) [Bradtke and Barto, 1996; Boyan, 1999] to the control problem. Being a variant of Policy Iteration, the LSPI algorithm alternates between two steps: policy evaluation and policy improvement. The approach used for performing the policy evaluation step relies on the fact that the action-value function Q^π is a fixed point of the Bellman operator:

$$\begin{aligned} & T_\pi Q^\pi = Q^\pi, \\ \text{where} \quad & T_\pi Q^\pi(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_b Q^\pi(s',b) \right] \end{aligned} \tag{5.26}$$

One way of finding a good approximation to Q^π is to find an approximate value function $Q^\pi_{\mathbf{w}}$, which is a fixed point under the corresponding Bellman operator: $T_\pi Q^\pi_{\mathbf{w}} \approx Q^\pi_{\mathbf{w}}$. This is possible if the fixed point lies in the space of approximate value functions determined by the chosen structure of the function approximator. Even when some $Q^\pi_{\mathbf{w}}$ lies in this space, the same is not guaranteed, however, for $T_\pi Q^\pi_{\mathbf{w}}$, and hence, $T_\pi Q^\pi_{\mathbf{w}}$ has to be projected back onto this space. Assuming that some orthogonal projection is used, the goal is to find an approximate value function $Q^\pi_{\mathbf{w}}$ that is invariant under the application of the Bellman operator T_π followed by the orthogonal projection.

In the LSPI approach, the entire learning process can be performed with a fixed batch of transition samples $D = \{(s^l, a^l) \xrightarrow{r^l} q^l \; ; \; l = 1, ..., L\}$, where $s_l, q_l \in S$, and which can be collected under an arbitrary distribution. The action-value function is assumed to be approximated by a linear combination of some basis functions, similar as in other approaches discussed above:

$$Q_{\mathbf{w}}(s, a) = \sum_{m=1}^{M} w_m \phi^m(s, a) \tag{5.27}$$

The policies evaluated on policy evaluation steps are greedy with respect to the current approximation $Q_{\mathbf{w}}$. The details of the LSTDQ algorithm used for policy evaluation may be found in [Boyan, 1999; Lagoudakis and Parr, 2003b]. In summary, this algorithm computes the following matrix $\mathbf{A}$ and vector $\mathbf{b}$:

$$\begin{aligned} \mathbf{A}_{ij} &= \sum_{l=1}^{L} \phi^i(s^l, a^l) \left[\phi^j(s^l, a^l) - \gamma \phi^j(q^l, \pi(q^l))\right] \; , \; i, j = 1, ..., M \\ \mathbf{b}_i &= \sum_{l=1}^{L} \phi^i(s^l, a^l) r^l \; , \; i = 1, ..., M \end{aligned} \tag{5.28}$$

The parameters $\mathbf{w} = (w_1, ..., w_M)$ of the function approximator presented in Equation (5.27) are calculated as follows[9]:

$$\mathbf{w} = \mathbf{A}^{-1}\mathbf{b} \tag{5.29}$$

Then on the policy improvement step, a new policy is adopted which is greedy with respect to the approximation of the action-value function corresponding to the new parameters $\mathbf{w}$. The policy evaluation and policy improvement steps are repeated until the parameter vector $\mathbf{w}$ does not change significantly in subsequent iterations.

In the limit, the LSPI algorithm is guaranteed either to converge or to oscillate between policies whose action-value functions are bounded away from the action-value function of the optimal policy by at most $\frac{2\gamma\eta}{(1-\gamma)^2}$, where η is the largest approximation error for the action-value function of any policy evaluated during Policy Iteration. The two factors that influence the approximation error η are the choice of the basis functions $\phi^m(s,a)$ and the distribution of the sample data set D.

The implementation of the LSPI algorithm sketched above is intended for off-line learning, in which case the algorithm can learn from a single data set that is reused for every policy evaluation step. An on-line version of the LSPI algorithm can also be obtained by performing the update in Equation (5.28) and computing the parameters as in Equation (5.29) for every sample. However, it would be quite computationally expensive to perform matrix invertion on every step. Another possibility would be to maintain a window of the most recent experiences and compute (5.28) and (5.29) only at some regular intervals. However, such approaches have not yet been explored in practice.

The LSPI algorithm has an interesting property that distinguishes it from other incremental algorithms: there is no learning rate involved in computing the approximator's parameters and hence, less tuning is required from the user. Most on-line

[9]In order to perform matrix inversion, it is necessary that the mattrix A be full rank. However, this cannot always be guaranteed in practice. See [Lagoudakis and Parr, 2003b] for some modifications to the way the matrix $\mathbf{A}$ is computed in Equation (5.28) to deal with the possibility that the resulting matrix $\mathbf{A}$ is singular.

methods discussed earlier in this section, such as LMS training presented in Equation (5.8), the residual gradient method in Equation (2.45) and the averager update in Equation (5.25), are stochastic approximation algorithms. Given that learning steps used for these algorithms are usually quite small, each sample causes only a very small parameter change before it is discarded. Hence, sample requirements of such algorithms can be very high. In contrast, the LSTDQ algorithm makes full use of all samples and is not sensitive to the order in which they are presented.

The LSPI algorithm does not require the set of the basis functions $\phi^m(s,a)$ used by the linear approximator as in Equation (5.27) to remain the same on every iteration. In fact, different representations may be best for policies evaluated on different policy evaluation steps. Recall that the error bound for the LSPI method relies on a good choice of basis functions. However, the issue of how such basis functions should be chosen is not addressed in [Lagoudakis and Parr, 2003b], while this question is identified as an important area for future research. In the experiments on the inverted pendulum problem presented in [Lagoudakis and Parr, 2003b], Gaussian basis functions were used and configured to cover uniformly the entire state space. The experiments on the bicycle balancing and riding problem presented in the same paper used polynomial features designed based on domain knowledge.

Since the SDM model is linear, it can be used in the framework of the LSPI approach. It should be noted that the cost of matrix inversion performed by the LSTDQ algorithm increases with the number of basis functions. Thus the architecture should contain the smallest number of basis functions possible, while ensuring that they cover appropriately the state space.

Self-Approximating Random Bellman Operator

All of the approaches discussed above use function approximation to represent value functions. Rust (1997) considered a randomized algorithm based on a *self-approximating* Bellman update (derived from the Value Iteration algorithm). This algorithm was analyzed for MDPs with continuous n-dimensional state spaces and

discrete finite sets of actions. In this case, explicit values are estimated only for a sample of states $\{s^1, ..., s^M\}$ by applying the following iterative algorithm:

$$V(s^m) := \max_a \sum_{k=1}^{M} P^a_{s^m,s^k} \left[R^a_{s^m,s^k} + \gamma V(s^k)\right] \ , \ m = 1, ..., M \tag{5.30}$$

Note that contrary to the update in Equation (5.14) for approximate Value Iteration, only the states that belong to the preselected sample $\{s^1, ..., s^M\}$ are used as successor states in the summation. Thus, no approximate representation of the value function is used and the update operator (5.30) is said to be self-approximating. This approach was motivated by the fact that multivariate function approximation, including interpolation, is intractable (not computable in polynomial time) even with randomized algorithms [Traub *et al.*, 1988]. Hence, the use of any function approximation was renounced in an attempt to find a polynomial-time approximate algorithm.

The iterative method in (5.30) aims at computing an η-close approximation to the true value function. It was proved to have a worst-case running time that is only polynomial in the number of state dimensions, thus "breaking the curse of dimensionality" in terms of computational time. The approximation error η scales proportionally to $\frac{1}{\sqrt{M}}$, where M is the number of sample states. The sample states need to be drawn independently and uniformly randomly from the entire state space. The reward and transition probability functions need to satisfy Lipschitz continuity conditions. When the values of the sample states, $V(s^m)$, are computed to a desired accuracy using the update rule in Equation (5.30), an optimal action in any state s can be obtained as follows:

$$\pi(s) = \arg\max_a \sum_{k=1}^{M} P^a_{s,s^k} \left[R^a_{s,s^k} + \gamma V(s^k)\right] \tag{5.31}$$

For this approach to be practically applicable, the model of the MDP needs to be known. The approach that we discuss next is similar, but removes this last assumption.

Kernel-Based Reinforcement Learning

In the kernel-based approach for reinforcement learning introduced in [Ormoneit and Sen, 2002], it is assumed that there is a fixed set of training samples $D(a) = \{(s^l, a) \xrightarrow{r^l} q^l \; ; \; l = 1, ..., L\}$ for every action a. The states q^l that appear as successor states in the transition samples of the data sets $D(a)$ are designated as prototype states. A smooth non-negative "mother-kernel" ψ is chosen (e.g., Gaussian), such that $\psi(s, a, q) = \psi(\frac{\|s-q\|}{b})$, where b is the width of the kernel. Then, an iterative algorithm in the style of Rust (1997) is applied in order to update the values of the prototype states:

$$\begin{aligned} V(q^m) &:= \max_a \sum_{l=1}^{L} \kappa(s^l, a, q^m) \left[r_l + \gamma V(q^l) \right] \quad , \text{for all } m = 1, ..., L \\ \kappa(s^l, a, q^m) &= \frac{\psi(s^l, a, q^m)}{\sum_{k=1}^{L} \psi(s^k, a, q^m)} \end{aligned} \tag{5.32}$$

Just like in the algorithm of Rust (1997), the operator (5.32) is self-approximating. Here the MDP model is not required, but learning is still performed off-line, based on a batch of data $D(a), \forall a \in A$, such that the values of all prototype states are updated synchronously.

The terms $\kappa(s^l, a, q^m)$ can be seen as replacing the transition probabilities in the algorithm of Rust (1997), Equation (5.30), by similarity kernels.

This algorithm may seem somewhat counterintuitive at first. It is estimating the values of the successor states q^m, that is the states *to* which the transitions are sampled. At the same time, the kernels $\kappa(s^l, a, q^m)$ in Equation (5.32) measure the similarity of these successor states q^m to the "start" states s^l of the transition samples. However, the algorithm operates under the assumption that the starting states s^l in the data sets $D(a)$ are sampled independently and uniformly randomly across the entire state space, in which case all prototype states q^m should be reasonably covered.

An optimal (greedy) action can be computed in any state s as follows:

$$\pi(s) = \arg\max_a \sum_{l=1}^{L} \kappa(s^l, a, s) \left[r^l + \gamma V(q^l)\right] \tag{5.33}$$

This approach is reminiscent of the case-based learning, in which case one would store a set of cases $\{\langle s^l, a, q^l, r^l, V(q^l)\rangle, l = 1, ..., L, \forall a \in A\}$ in a memory and combine them using Equation (5.33) based on the similarity of the query state s to the starting states s^l of the cases, whenever an action has to be chosen in some state s.

Contrary to most methods discussed above, the approach of Ormoneit and Sen (2002) provides a specific way in which prototype states should be selected based on the training data. Unfortunately, the assumption about the uniform training data distribution is quite restrictive for many practical applications. Also, an elegant extension of this method to on-line learning is not obvious.

As already indicated, the approaches in [Rust, 1997] and [Ormoneit and Sen, 2002] do not rely on any function approximation at all. We discussed them here mainly because the approach of Ormoneit and Sen (2002) may seem superficially similar to the approach in [Szepesvári and Smart, 2004] and to our SDM model, due to the use of the similarity kernel. Hence, we wanted to highlight the differences.

Summary

As previously indicated, the SDM architecture provides a linear approximator based on local basis functions. Basis functions $\phi^m(s,a) = \mu^m(s,a)$ can be defined either by similarity measures, associated with memory locations, or by the normalized similarity measures:

$$\phi^m(s,a) = \frac{\mu^m(s,a)}{\sum_{k \in H_s} \mu^k(s,a)} \ , \ m = 1, ..., M_a \tag{5.34}$$

We discussed above that under certain constraints imposed on the SDM layout, the SDM model can be configured to satisfy conditions in Equation (5.19) (or a special case of interpolative conditions in Equation (5.24)). In this case, the SDM approximator has the non-expansion property and can be used safely with such methods as

off-line approximate Value Iteration presented in Equation (5.14), LSPI presented in Equations (5.28)-(5.29) and the on-line averager update in Equation (5.25).

With the LMS training rule (5.8), unfortunately, there are no convergence guarantees for the control problem in the current literature. However, it does not require the model of the MDP, is not computationally costly and can be applied naturally on-line. It is most often used with on-policy exploration, which can be very important when trying to concentrate the available resources on important parts of the state space. Practical experiences with LMS training combined with linear local function approximators in value-based reinforcement learning are mostly positive (see, e.g., [Santamaria *et al.*, 1998]). Extensive comparative experimental studies would be useful to assess the relative performance of the methods discussed in this section. Unfortunately, little practical evidence of this kind exist in the current literature.

All of the methods discussed in this section rely on the selection of either prototype states, on which the algorithms focus their updates, or local basis functions centered around some states. The issue of how to choose such states and how to configure the structure of the function approximator has been gaining a lot of attention over the last few years in the reinforcement learning community. In the following section, we present an overview of existing approaches for this problem. Then in Section 5.5, we propose a new method for dynamic allocation of SDMs based on observed data.

5.4. Overview of the Resource Allocation Methods

In this section, we review different existing approaches for choosing parameters of the basis functions for local function approximators. In the rest of this chapter, we will sometimes refer to various building blocks of such architectures as *local units*, be it SDM memory locations, RBFs, instances in memory-based methods or discrete partitions of the input space.

In the current supervised and reinforcement learning literature, approaches for configuring local function approximators can be divided roughly into two categories.

One class of methods assumes that the number of local units is fixed ahead of time and that initially, all of them are either distributed uniformly randomly across the input space, or positioned by an unsupervised learning method, such as clustering. In the latter case, a batch of training data is assumed to exist from the beginning. Then during learning, these methods usually slowly tune the parameters of the local units based on data.

With the second class of methods, learning starts with no preallocated units or with just a few very coarse units. Then these methods dynamically add new units and refine existing units based on training data. Various (mostly heuristic) criteria are used to decide where the additional units are needed. When a maximum allowed number of local units is allocated, some methods remove previously added units (again based on various heuristics) and add new units elsewhere.

We will now review existing methods, belonging to both of the described categories. We focus on the methods that were developed for architectures based on the linear combination of local units. There is a big body of research addressing similar issues in multilayer Nural Networks (see, e.g., [Fahlman and Lebiere, 1989; Reed, 1993; Rivest and Precup, 2003; Thivierge *et al.*, 2003]), but it is beyond the scope of this thesis.

In our initial attempts to dynamically allocate and tune SDM locations in the context of on-line reinforcement learning, we explored some ideas from the current literature. We will mention these undertakings throughout this section. We will discuss the difficulties that we encountered, which led us to develop more robust heuristics, as will be presented in Section 5.5.

5.4.1. Methods for Tuning Local Units

In supervised learning, parameters of local units, such as their positions and widths, are often adjusted in an incremental manner based on training data. There are two main approaches used in this case: gradient descent learning and self-organizing learning in the style of Kohonen networks (see, e.g., [Haykin, 1994; Kohonen, 1984]).

Gradient-Based Tuning of Local Units

Parameters of local units can be optimized with LMS training, i.e., by gradient descent on some objective error function, e.g., the Mean Squared Error in Equation (5.4).

For instance, to optimize addresses of the SDM locations in this way, the MSE function would be considered to depend on the address parameters $\mathbf{L} = \{(h_1^m, ..., h_n^m), m = 1, ..., M\}$. In this case, the function approximation architecture is no longer linear in tunable parameters. Upon observing a training sample $\langle \mathbf{x}, f(\mathbf{x}) \rangle$, the addresses would be updated in the direction of the negative gradient of the MSE function with respect to the address parameters:

$$h_i^m := h_i^m - \alpha \left. \frac{\partial MSE_{\mathbf{w},\mathbf{L}}}{\partial h_i^m} \right|_{\mathbf{w},\mathbf{x},\mathbf{L}}, i \in \{1, ..., n\}; m \in H_{\mathbf{x}} \tag{5.35}$$

A similar update can be performed for tuning the activation widths $\mathbf{B} = \{(\beta_1^m, ..., \beta_n^m), m = 1, ..., M\}$ of the SDM locations. With this approach, local units are positioned and configured in a way that minimizes the approximation error.

Gradient-descent learning on the MSE criterion was used in [Flachs and Flynn, 1992] to tune the addresses of the memory locations in a binary SDM model. Intuitively, for a binary classification model, the memory locations that classify correctly the current training input, are moved closer to this input, while the locations that classify the current input incorrectly, are moved farther away. In [Flachs and Flynn, 1992], both the content parameters $\mathbf{w}$ and the address parameters $\mathbf{L}$ are updated for every training sample.

Gradient descent is also widely used for tuning parameters of basis functions in RBFNs in supervised learning (see, e.g., [Platt, 1991; Haykin, 1994; Schwenker *et al.*, 2001]). NRBFNs trained in this manner were applied in reinforcement learning in [Kretchmar and Anderson, 1997; Šter and Dobnikar, 2003]. However, using this method for learning a representation of the value function can be difficult. When the LMS training rule is applied with reinforcement learning, the error term takes the form of $[r + \gamma Q_{\mathbf{w},\mathbf{L},\mathbf{B}}(s', a') - Q_{\mathbf{w},\mathbf{L},\mathbf{B}}(s, a)]$. In this case, one implicitly minimizes

temporal difference (TD) errors over a set of representative state-action pairs (s, a) that appear as training inputs. It has been noted by many researchers in the past (see e.g., [Sutton and Barto, 1998; Baird, 1995; Kretchmar and Anderson, 1997] that minimizing the TD error *for a subset of states* can sidestep the ultimate goal of finding the optimal policy in the case of the control problem. The success of this approach depends, to a large extent, on the selection of the subset of state-action pairs for which TD errors are minimized (i.e., the distribution of the training inputs) as well as on the choice of the structure of the function approximator. In the case of on-policy semi-greedy exploration, the distribution of state-action pairs depends on the action-value estimates $Q_{\mathbf{w},\mathbf{L},\mathbf{B}}$. In this case, LMS value estimation may mislead the learning process: the agent can find a particular parameter setting, such that the corresponding policy visits mainly the states, for which it is easy to achieve small TD errors, while this policy fails to lead to truly rewarding states. Such behavior was observed, for instance, in [Kretchmar and Anderson, 1997] with NRBFNs. In their experiments, basis functions tended to migrate to the regions with small TD errors, while leaving other important parts of the state space underrepresented.

Finally, note that the TD errors observed during on-line learning may result from the environment's stochasticity and not only from the value approximation error. In a stochastic environment, where a state-action pair (s, a) can have many different successor states s', or in which immediate rewards are stochastic, the variance of the estimates $[r + \gamma Q_{\mathbf{w},\mathbf{L},\mathbf{B}}(s', a')]$ can be large and thus, observed TD errors can be large. In order to prevent state-action pairs with a high variance of their value samples from being unjustly abandoned as a result of LMS learning, information about the environment's stochasticity can potentially be used. Such information can be obtained, for example, by measuring the MDP attributes presented in Chapter 3, e.g., the State Transition Entropy and the Variance of Immediate Rewards.

Self-Organizing Tuning of Local Units

Function approximation architectures can also be adjusted using on-line, self-organizing learning methods. In [Rao and Fuentes, 1998], the SDMs were used for learning a fixed navigational behavior of a robot. The addresses of the SDM locations were trained with a soft competitive learning rule:

$$h_i^m := h_i^m + \alpha \bar{\mu}^m(\mathbf{x})(x_i - h_i^m) \ , \ i = 1, ..., n; \ m = 1, ..., M \tag{5.36}$$

where $\bar{\mu}^m(\mathbf{x})$ is defined as a Gaussian kernel, based on the Eucledian distance between the address of the location $\mathbf{h}^m$ and the input address $\mathbf{x}$. Thus, the addresses of memory locations are iteratively updated to match the distribution of training inputs. This rule is called competitive, because local units compete for getting closer to training inputs[10]. It should be noted that in this application, supervised training (training from demonstration) was performed in order to learn an appropriate navigational strategy. The agent was not operating in the setting of the standard control problem, but was learning the value of a fixed policy. Hence, the training data distribution was fixed and learning could slowly converge to an SDM configuration representative of the fixed distribution of the training inputs.

A very similar self-organizing update rule was used in [Wiering, 1999] for tuning the centers of basis functions in NRBFNs, as well as in [Millán *et al.*, 2002] for positioning localized receptive fields of an Incremental Topology Preserving Map (a linear local function approximator). In both cases, it was used in combination with other heuristics for dynamic addition of new local units, which we will discuss later in this section.

The approach introduced in [Sutton and Whitehead, 1993] for supervised learning with binary SDMs, also slowly moves existing memory locations toward the observed data. Their algorithm works as follows. If the number of active locations for a given training sample is too small (relative to some threshold specified by the user), an

[10]The Gaussian kernels used in [Rao and Fuentes, 1998] have a width parameter that decreased with time, so that the number of locations competing for each data point decreased over time. This, together with a decreasing learning step α, allowed the architecture to stabilize in the limit.

inactive location is selected at random and moved towards the current input in one address dimension, selected randomly. A symmetric adjustment is made if too many locations are active.

We implemented a version of this algorithm, where a pre-specified number of address dimensions (between 1 and n) can be adjusted towards or away from the current input proportionally to some learning rate α'. We tested this approach with on-line reinforcement learning using the Sarsa(0) algorithm and ϵ-greedy exploration on the Mountain Car domain. Unfortunately, this approach did not allow us to achieve any stable learning, despite a significant amount of tuning of all the user-defined parameters involved. We came to a conclusion that this algorithm was unable to track quickly the non-stationary data distribution produced by policies that can change abruptly with the on-line ϵ-greedy Sarsa(0) algorithm.

Other Optimization Approaches

In [Menache *et al.*, 2004], a global optimization approach was considered for tuning parameters of the basis functions in the RBFNs in the context of the policy evaluation problem. In this work, estimation of the linear weights of the RBFN and adjusting parameters of the basis functions were separated into two distinct phases, which were interleaved during learning. In one phase, linear parameters of the network (with the RBF parameters held fixed) were computed using the LSTD(λ) algorithm [Boyan, 1999]. Then, in the second phase, the RBF parameters were optimized in a batch mode. A novel approach that this work explored was based on the Cross Entropy (CE) method. It is a global optimization technique that iteratively improves the estimates of meta-parameters, which are the parameters of the probability density function assumed to generate parameters of the RBFs. This method was found to outperform the gradient descent approach for the policy evaluation task. The overall approach is, however, quite computationally expensive, as, on every iteration, it evaluates goodness of a *set* of RBF parameters in order to update the meta-parameters. For each such evaluation, the LSTD(λ) method is invoked on a batch of training data

(which involves matrix inversion) in order to re-estimate the setting for the linear network weights.

In [Sato and Ishii, 1998], a Normalized Gaussian Network (NGnet) was built with an on-line version of the EM algorithm, which is a stochastic approximation of the batch EM algorithm proposed for the NGnets in [Xu *et al.*, 1995]. In this approach, the NGnet is interpreted as a stochastic model (a mixture of experts) with hidden variables. In general, the EM algorithm alternates two steps: Estimation (E-step) and Maximization (M-step). In this algorithm, on the E-steps, the posterior probabilities of each unit being activated by a training sample is calculated according to the Bayes rule, given the current setting of the local units' parameters. On the M-step, network parameters are re-estimated to maximize the expected log-likelihood of the observed data, which uses the posterior probabilities estimated on the E-step. A general version of the algorithm proposed in [Sato and Ishii, 1998] assumes that a certain number of local units are preallocated from the onset of learning. However, it was observed in [Sato and Ishii, 1998] that, when the initial distribution of the local units was significantly different from the the distribution of the training data, the EM algorithm learned very slowly and could converge to a local optimum far from the desired solution. To overcome this difficulty, additional mechanisms for unit manipulations were used, which allowed unit additions, deletions and divisions conducted on-line, after observing each training sample. A new unit was added if the current input was too distant from the units of the current model. The distance criterion was evaluated based on the joint probabilities of seeing the current input and activating each of the existing units. These probabilities were obtained as a result of the E-steps of the EM algorithm. The deletion of a unit was based on the amount of activation enjoyed by the corresponding unit in the past. Unit division was performed based on the observed approximation error. In this case, when the error exceeded a certain threshold, the unit was substituted with two new units, each with the width half as large as the width of the original unit. Similar unit manipulation methods were used in many other approaches, as will be discussed below. In [Sato

and Ishii, 1998], a probabilistic interpretation of these heuristics was presented. The approach was tested with a non-stationary data distribution, which was gradually changing over time. To deal with the non-stationarity, the algorithm had a specific parameter, which controlled the rate, in which the on-line EM was forgetting past training samples and was giving more weight to the new ones. The approach was also applied to the data from a simulation of robotic arm movement, but in the context of a prediction problem, where the learning system was trained to predict certailn aspects of the system's dynamics.

5.4.2. Methods for Allocating Local Units

In the above discussion, most methods assumed that the total number of local units in the function approximation architecture remained fixed throughout learning while their positions and widths were adapted. Now, we will discuss techniques that also determine the size of the architecture during learning, by dynamically adding and possibly removing some units.

Perhaps, the most prominent example of methods that dynamically add local units to the function approximator are memory-based or instance-based approaches. As previously mentioned, they have already been used in reinforcement learning (see, e.g., [Atkeson *et al.*, 1997a; Forbes, 2002; McCallum, 1996; Santamaria *et al.*, 1998; Smart and Kaelbling, 2000]). The approaches in [Atkeson *et al.*, 1997a; Smart and Kaelbling, 2000] did not address the issue of a limited storage capacity, and there was no attempt to control the number of stored instances. In this case, the datasets of stored instances can get very large. For example, in [Smart and Kaelbling, 2000], for the two-dimensional Mountain Car domain, the datasets got as large as 300,000.

Various tree-based approaches, e.g., regression trees, the size of which is always determined by training data, were used in reinforcement learning to produce variable resolution discretizations of the input space (see, e.g., [McCallum, 1996; Uther and Veloso, 1998; Munos and Moore, 2001]. Techniques for dynamically growing function approximation architectures were also studied in the context of RBFNs. Well-known

approaches include Neural Gas [Martinetz and Schulten, 1991; Fritzke, 1994; Wiering, 1999] and Resource Allocating RBFNs [Blanzieri and Katenkamp, 1996; Platt, 1991; Anderson, 1993].

There are many heuristics to assess which parts of the input space need additional local units. Such heuristics can be roughly divided into three classes. First of all, some methods rely on certain distance measures between local units to control their density. Other methods rely on the prediction error in the local neighborhoods of the input space and add more local units in the regions of large approximation errors. Finally, the third class of methods make an assessment of the shape of the target function, in order to best match the topology of the target function with the shapes of the local units, e.g., piecewise constant or piecewise linear. We will now discuss such methods from the current supervised and reinforcement learning literature.

Input Similarity Based Local Unit Addition

In the applications of instance-based methods to reinforcement learning [Santamaria *et al.*, 1998; Forbes, 2002], training instances were only selectively added to memory in order to control the memory size. A heuristic approach used in these cases was formulated in the classical instance-based learning framework [Atkeson *et al.*, 1997a], which is based on the definition of two functions: a *distance metric* in the input space, e.g., the Euclidean distance, and a weighting function, or *kernel*, e.g., Gaussian. Kernel functions transform the distances into weights used in the locally weighted learning performed for predictions. They are related to the local basis functions in the linear architectures that we discussed so far. In [Santamaria *et al.*, 1998] and [Forbes, 2002], new instances were added to the memory if they were farther away from the existing instances than a specified threshold. Such threshold was defined in terms of the distance metric and was not related to the width of the kernel function. If this correspondence is not explicitly addressed, the obtained memory can be too sparse for the employed kernel functions. While it is easy to prevent this in the case of

a uniform and fixed width of the weighting functions, such formulation does not provide a robust generalization to varying widths. Adaptive kernel widths were claimed to be used in [Forbes, 2002], where the width update was performed based on the approximation error of the activated instances. Thus, such a width selection method would not properly correct for too large of a sparseness if no instances are activated. Unfortunately, no discussion was provided in [Forbes, 2002] for the practical behavior of this approach and its parameter settings.

The same distance-based addition heuristic was also used in [Platt, 1991], in combination with an error-based heuristic (discussed below), for the Resource-Allocating RBFNs. This approach also used variable widths of the local units, which were chosen to be proportional to the distance between the newly added unit and the closest existing one. Thus, this way of width selection did, indeed, reconcile the distances between the units' centers and their widths.

Another obvious way to measure similarity between local units is based on their *activations*. For instance, in the case of the SDM model, the activations are similarity measures, as in Equation (5.1); in the case of RBFNs, they are the values of the the RBFs; and in the case of the instance-based learning - kernel values. The distance of a unit u_1 to a unit u_2 would thus be the activation of the unit u_2 incurred by the center of the unit u_1. Note, that such distances are not necessarily symmetric, depending on the widths of the corresponding units. Several approaches in the current literature relied on this notion of the activation-based similarity to decide whether to add a new unit to the architecture, e.g., [Kondo and Ito, 2002; Anderson, 1993; Millán *et al.*, 2002]. In these cases, the new unit is added if the activations of all the existing units by the current input are below certain user-defined threshold. The new unit is then centered at the current input.

The method in [Kretchmar and Anderson, 1999] for configuring basis functions of a linear local function approximator (for state-value function representation) shaped each local unit so that it covered states based on their *temporal proximity*. More specifically, "temporal" activation neighborhoods of the local units grouped the states,

which experienced frequent intra-group transitions during on-line sampling. The method was suggested for the class of problems, where the immediate rewards on each step are small, in which case, the difference in the values of any two temporally adjacent states will also be small. The values within each such local unit were approximated by a constant. The algorithm interleaved the periods of tuning the weights of the linear combination of the local units with the periods of re-configuring the neighborhoods of the basis functions. In the case of discrete state spaces, each local unit had a vector indicating to which degree it could be activated by the states of the MDP. Thus, local units could potentially have irregular, non-convex shapes.

A similar idea was recently explored in [Glaubius and Smart, 2004] for the state-value function approximation with the manifold representations. A motivation behind the method developed in this work was the fact that most function approximation architectures rely on the assumption that the similarity between the states can *always* be related to their Eucledian distance. However, it is often not true. For example, consider an environment with obstacles, such as internal walls. In this case, imagine some state s_1 on one side of the wall and two other states s_2 and s_3 that are at the same Eucledian distance from s_1, but the state s_2 is on the opposite side of the wall, while the state s_3 is on the same side. The time that is needed to reach the state s_3 from the state s_1 can be much smaller than the time needed to reach the state s_2, since it is necessary to go around the wall. Thus the Euclidean distance cannot be used everywhere across the state space, while it can be meaningfully used within certain smaller regions of the state space. Thus, the approach attempted to cover the state space with *charts*, hyper-rectangular regions, within which Euclidean distance is a good indication of the distance between states. Within each chart, it was considered to be safe to use function approximators that rely on the Euclidean proximity between the states. The algorithm was then allocating charts in the state space, so that they established boundaries between the states that have different relative magnitudes of state transitions[11].

[11]Note, that contrary to the approach of [Kretchmar and Anderson, 1999], here, local units have a regular form, where it is easy to identify which of them are activated by any given state.

It should also be noted, that generalizing across states with different temporal transition distances is not always harmful. It may take longer to get from the state s_1 to state the s_2 than to the state s_3, but it may take the same amount of time to get to the goal from both s_1 and s_2, while it takes longer to get to the goal from the state s_3. In this case, these two states will be similar in terms of their values and, in fact, this is what ultimately matters. So, ideally, the similarity between states should be related to the similarities in their values, however, we cannot measure this precisely since the values are not known in the first place.

Nevertheless, there exist approaches in the current reinforcement learning literature that attempt to reason about the topology of target value-functions and to construct function approximators accordingly. We will discuss these approaches next.

Local Unit Addition Based on Target Topology

The choice of function approximation architecture imposes certain properties on the resulting approximate function, for example, its piece-wise constant or piece-wise linear nature. Thus, it would be desirable to determine whether the target function also possesses similar properties, so that it can be appropriately modeled by the chosen function approximator. A number of approaches attempt to assess the topology of the target function based on the training data and to adjust the approximator's structure accordingly. For example, if the target function appears to be non-constant within a particular region, while the approximation architecture is trying to fit a constant surface in that region, the approximator's constant piece can be broken into smaller surfaces in order to better explain the variation in the observed target values. In general, when using local function approximators, it would be useful to assess whether the target function has local properties similar to those implied by the existing local units. If this is not the case, more units should be added in order to be able to match a complex target topology with small simple building blocks. The differences in the topology of the target function across adjacent regions can be inferred from various indicators, such as the differences in the distributions of

the target values in those regions or a significant change of the gradient of the target function. Some methods in the current literature make an attempt to exploit such indicators while dynamically building approximation architectures. We will discuss some of them next.

Several approaches were studied in the reinforcement learning context that construct variable resolution discretizations for the action-value function approximation. An algorithm from [Chapman and Kaelbling, 1991], known as G-algorithm, was designed to build variable-resolution discretizations for the value-function representations in MDPs with discrete (binary) state spaces. The action-value functions were represented by decision trees. In this case, the leaves of the tree correspond to partitions of the state space. Then, throughout learning, partitions, induced by the decision tree, were considered for splitting. To assess whether a particular partition needed to be split, the method estimated a future reward distribution $D(s,a,r) = \sum_{k=0}^{\infty} \gamma^k P(r_{k+1} = r|s,a)$, where possible rewards r were considered to be drawn from a small discrete set R. The distributions D within new candidate partitions were tested for significant difference with a statistical test. If the difference was significant, the split was performed.

In the work of [Uther and Veloso, 1998], regression trees were also used to represent non-overlapping partitions of the state space[12]. Within each partition, the value function was approximated by a constant. Similar to the G-algorithm, this approach repeatedly considered whether the current partitions should be split. Learning was performed in an off-line manner by interleaving two phases. During one phase, the Value Iteration method was used to find the optimal action-value function $\hat{Q}(s,a)$ of the approximate discretized MDP, defined on the partitions induced by the current regression tree. Then, during the second phase, a batch of transition samples $(s,a) \xrightarrow{r} s'$ was collected by sampling with the policy, greedy with respect to the current approximation $\hat{Q}$ of the action-value function. For each sample state-action pair (s,a), its value was estimated by a one-step lookahead $Q(s,a) = r + \gamma \max_b \hat{Q}(s',b)$.

[12]As usual, a separate tree was used for each action.

One of the criteria used in [Uther and Veloso, 1998] for splitting partitions, induced by the regression tree, was based on the Kolmogorov-Smirnov statistical test measuring the difference between probability distributions. In this case, a split of an existing partition was performed if the distributions of the values of the state-action samples falling into two new candidate partitions were significantly different based on the performed test. Performing the Kolmogorov-Smirnov test takes time quadratic in the number of value samples.

The approach in [Munos and Moore, 2001] was devoted to building variable-resolution discretizations of the MDP state space, similar to the ones produced by the regression trees in the approach of [Uther and Veloso, 1998]. In this case, the approximate discretized MDP was defined on the vertices of the resulting partitions. Then, the state-value function of the original MDP was approximated, so that the values of the states within partitions were computed by barycentric interpolation on the corners of the hypercubes corresponding to the partitions. Similar to the approach of [Uther and Veloso, 1998] described above, the optimal state-value functions were computed for the discretized MDPs by the Value Iteration algorithm, and then each partition was considered for splitting. The work in [Munos and Moore, 2001], developed a number of different splitting criteria intended to assess the properties of the target function. The *corner-value difference* heuristic computes the average of the absolute differences of the state-values at the corners of the partition along each side of the corresponding hypercube. According to this criterion, the splits are performed for those partitions where the value function is least constant. The *value non-linearity* heuristic computes the variance of the absolute differences of the values at the corners on all the edges of the partition. Thus, according to this criterion, the partitions, within which the value function is least linear, are split. The *policy-disagreement* criterion attempts to determine the boundaries between the states where the optimal action changes. Such an estimate is based on a comparison between optimal actions, derived from the value function of the discretized MDP, and actions, derived from an

optimal feed-back control policy computed from the Hamilton-Jacobi-Bellman equation[13] based on the local gradient of the value function.

The approach in [Reynolds, 2002] also creates a variable resolution discretization, while it is designed for on-line training. It was based on the *decision boundary heuristic*, which, similar to the policy-disagreement criterion in [Munos and Moore, 2001], attempts to find boundaries in the state space that separate regions with different optimal actions. In order to detect such possible boundaries, the algorithm inspects all pairs of the adjacent partitions. Suppose, i and j are neighboring partitions, their corresponding action-values are $Q_i(a)$ and $Q_j(a)$, and the corresponding greedy actions are a_i and a_j. Then, $\Delta_i = |Q_i(a_i) - Q_i(a_j)|$ estimates a potential loss from taking the action a_j in the partition j, when hypothesizing that its values should be closer to those of partition i and thus have the greedy action a_i instead of a_j. Or, in other words, Δ_i estimates the loss of not extending the value of partition i farther into partition j. A similar loss value Δ_j is estimated symmetrically as $\Delta_j = |Q_j(a_j) - Q_j(a_i)|$. Both partitions are then considered for a split if at least one of the values, Δ_i or Δ_j, exceed a predefined loss threshold. The splits are actually performed if the action values of the considered partitions have been updated a sufficient number of times since their creation.

The approach in [Boutilier *et al.*, 2000] represents value functions as decision trees and seeks to increase the resolution of the state representation where there is evidence that the value is not constant withing the state partition. The value functions are computed with a modified (structured) policy iteration algorithm, where a Bayesian network is used to compactly represent a transition probability function.

As can be observed from the discussion of the topology-based approaches, some of them are rather computationally expensive and require batches of training samples to be accumulated many times over the course of learning. The heuristics, which seem to be the simplest in terms of the computational and memory requirements and which can potentially be used on-line, are the corner-value difference and the value

[13]Hamilton-Jacobi-Bellman equation is an equivalent of the Bellman equation for continuous state spaces and continuous time.

non-linearity heuristics from [Munos and Moore, 2001] and the decision boundary heuristic from [Reynolds, 2002].

The inability of a function approximation architecture to model a target function topology will also be directly reflected in high approximation errors. The approaches that we discuss next, rely precisely on this type of information.

Error-Based Local Unit Addition

The approaches to action-value function approximation based on decision and regression trees in [McCallum, 1996] and [Uther and Veloso, 1998] used an error-based heuristic for splitting the partitions induced by the trees. It was based on the estimated variance of the Q-values for the state-action samples that fall into the partition considered for a split. In the case of [McCallum, 1996], the data samples, needed to estimate this heuristic, were collected on-line in a continuing manner, while using the instance-based learning method combined with the averager training rule (5.25) to update the action-values of the stored instances. In the case of [Uther and Veloso, 1998], such samples were collected off-line, as described above for the method using the topology-based heuristic based on the Kolmogorov-Smirnov test. Then the variance of the action-values for the samples collected in every partition was computed. If it exceeded a predefined threshold in some partition, this partition was split. Evaluating such a heuristic takes time linear in the size of the sample data. The size of the data batch and the threshold for the variance-based splitting criterion are the parameters that the user has to tune for this algorithm. In the experiments performed in [Uther and Veloso, 1998], this heuristic and the one based on the Kolmogorov-Smirnov test, described earlier, performed equally well, while the latter is more computationally expensive.

In [Kondo and Ito, 2002], an approach for dynamically building NRBFNs (with Gaussian basis functions) was used in the context of reinforcement learning (with the Actor-Critic algorithm) for mobile robots applications. In this case, learning started with a network that had no basis functions. New RBFs were added using

a combination of the activation-based heuristic, as described above, and an error-based heuristic. According to the error-based criterion, a new basis function was added when the TD error for the current state-action pair exceeded a user-defined threshold, and provided that the activation-based criterion was also satisfied. After a unit had been added, it could also compete for survival based on an approach derived from evolutionary computation. The evolutionary fitness function was based on the error criterion as well.

A similar approach was used in several studies with the Resource Allocating RBFNs [Platt, 1991; Anderson, 1993; Blanzieri and Katenkamp, 1996]. There, new units were added when "novel" training samples were encountered. Novelty was determined based on two conditions: the Euclidean distance from the current sample to the center of the closest existing unit, and the error on the prediction for this sample. The prediction error was compared against a fixed, user-defined threshold parameter. If the current sample was found to be novel, based on these two criteria, a new unit was added and centered at the current input value.

The approach in [Millán *et al.*, 2002], developed for building a function approximator based on overlapping localized receptive fields, also used a form of the error-based criterion (in combination with the activation-based heuristic) in order to determine whether a new unit had to be added to the approximator. An unacceptable error was considered to be encountered when the agent experienced a serious failure (e.g., a robot collided with an obstacle), in which case, a new unit could be added.

In the later studies with similar approaches, it was suggested, that instantaneous errors may not be good indicators of the function approximator's inability to learn the observed target value with the existing resources. Instead, some time should be allowed for the current units to be trained, and only if this fails, a new unit should be added. For instance, the approach in [Fritzke, 1994; 1997], based on the prior research with RBFNs known as Neural Gas, accumulated errors in each unit over time and then a new unit was inserted near the unit with the maximum accumulated error.

In the work of Anderson (1993), a similar approach, involving NRBFNs, was applied to reinforcement learning, in particular, with the Q-learning algorithm. In this case, a fixed number of local units were used, but some of them were allowed to be reallocated to new positions (a procedure referred to as a "restart" in [Anderson, 1993]). The reallocation could be triggered by an unusually large approximation error. In order to detect such an event, the mean and standard deviation of the approximation errors were incrementally estimated. Then, if the error on the current input was larger than the estimated average error plus some factor proportional to the standard deviation, some unit was considered for reallocation to a position corresponding to the current input. The reallocation actually happened if the activation-based criterion, as described above, was satisfied for the current input with the threshold value of 0.5. The choice of the unit to be reallocated from its current position was based on its utility, which was related to the amount of its previous activations.

Another approach from the same framework was also used in [Wiering, 1999]. In this case, the NRBFNs (referred to as Neural Gas in [Wiering, 1999]) were used with the $Q(\lambda)$ algorithm. During learning, each unit accumulated a *Responsibility* variable, which was the sum of its activations, for all previous training samples. A new unit was added when an instantaneous prediction error exceeded a predefined threshold, provided that the Responsibility of the closest unit also exceeded another threshold. The latter condition indicated whether the close-by units already had a chance to be trained after they had been added to the network.

In the approach introduced in [Samejima and Omori, 1999], dynamic allocation of NRBFNs was also considered and applied in the context of reinforcement learning. In this case, learning started with a single basis function covering the entire state space. Then, throughout learning, existing RBFs were divided, that is substituted with smaller ones, based on the distribution of the TD errors in the state space. A basis function was divided when the ratio of the variance and the mean of the local TD error became too large, where the local TD error refers to the TD error weighted by the activation of the corresponding basis function. For each unit, the local mean

error $\overline{e_m}$ was estimated by a running average. The variance estimate was substituted by the running average of the squared local TD errors, $\overline{(e_m)^2}$. When both the ratio $\frac{\overline{(e_m)^2}}{\overline{e_m}}$ and $\overline{(e_m)^2}$ exceeded certain thresholds, the corresponding basis function was deemed unfit and was divided into two new basis functions. The division direction (centers of the new basis functions) was determined by a direction of positive local TD errors. Thus, more resolution was added where it seemed possible to improve currently underestimated state-action values. The widths and the initial linear weights associated with the new basis functions were selected so that to minimize the stress to the overall approximation caused by the division of the unfit basis function. A specific optimization criterion was introduced for this purpose. This approach was applied to a robot collision avoidance and a robot navigation domains in combination with the Actor-Critic architecture.

Another variation of the error-based unit addition heuristic was presented in [Šter and Dobnikar, 2003]. It extended the approach of [Samejima and Omori, 1999] for the NRBFNs, as discussed above, in that it used additional ways to decide when and how to allocate new basis functions. Contrary to the approach of [Samejima and Omori, 1999], learning started with empty networks. Similar to other previously discussed approaches, a new unit was added when the current TD error exceeded a pre-specified threshold and when the activations of the present units were lower than another threshold. Then, the method of [Samejima and Omori, 1999] was used to determine which of the existing RBFs were unfit and needed to be substituted with several other basis functions of smaller widths. However, the method of substitution was different in this case. When some RBF was established to be unfit, a two-class Learning Vector Quantization (LVQ) procedure [Kohonen, 1984] was initiated. In this case, $2X$ representative vectors were scattered in the input space and associated with the unfit RBF. The number X had to be specified by a user. Half of the representative vectors were designated to represent a spatial distribution of the positive TD errors and the other half - of the negative TD errors. Then, for a certain period of time, these vectors were allowed to adapt to the distribution of the corresponding errors, so that

they were located mainly in the areas of larger errors. At the end of this process, each unfit RBF was replaced with $2X$ smaller RBFs, centered at the associated representative vectors. The width of each new unit was set to half the distance from the nearest existing unit. In addition to the above dynamic addition/replacement processes, the centers and the widths of the existing units were also updated by the gradient descent procedure. This approach was applied in the reinforcement learning context to the problem of learning collision avoidance strategies for a simulated mobile robot. The experiments were performed with the Q-learning algorithm, using uniformly random action selection in one set of experiments and using Boltzman-based exploration in another set. In both cases, thus, the distribution of the training samples did not change drastically, as would usually be the case with the ϵ-greedy learning. Interestingly, when compared to the method that uses only the simpler addition heuristics, based on thresholding current TD errors and activations of the existing units, the approach, using the Learning Vector Qauntization based unit replacements, provided a rather modest improvement in the testing returns while providing a comparable number of local units. The LVQ-based approach is, at the same time, much more computationally demanding.

The approach in [Munos and Moore, 2001] for state-value function approximation with variable-resolution discretizations introduced two global criteria, the *influence* and the *variance*. They assess a long-distance influence of the state-value function accuracy at certain states on the accuracy of other states. The variance is defined as $\sigma^2(s) = E\left[(R(s) - V(s))^2\right]$, where $R(s)$ is the return in the state s and $V(s)$ is the value of state s, i.e., the expected return. As shown in [Munos and Moore, 2001], this variance is the solution to a Bellman-like equation, which can be solved with dynamic programming based on the MDP model. The influence of the state s_1 on the state s_2 is defined as $I(s_2|s_1) = \sum_{k=0}^{\infty} p_k(s_2, s_1)$, where $p_k(s_2, s_1)$ is the k-step discounted probability of reaching state s_1 after k steps starting from state s_2 and following a policy greedy with respect to the current value function approximation. Intuitively, the influence measures the amount of change in the value of state s_2 that would result

from a unit change of the value of state s_1, located down a possible trajectory. Note, that if the states with high influence values already have accurate value estimates, splitting the corresponding partitions is not necessary. Hence, the influence-based criterion is used in combination with the variance-based criterion and the policy-disagreement criterion (discussed above), so that a partition is split if the influence of its states on the states at the decision boundary is high and the variance of this partition is also high. The influence and the variance criteria are very principled ways of estimating the need for an increased resolution of the function approximator, as they are based on global effects of approximation accuracy. However, their computation is quite expensive: it is as costly as evaluating a policy with dynamic programming.

In general, it is a good idea to insert additional resources where the current layout cannot approximate the target function well enough. However, with on-line reinforcement learning, this approach may not work as well as in supervised learning applications. Indeed, such concerns were raised, for instance, in [Wiering, 1999], where the method described above for the Neural Gas architecture was applied with $Q(\lambda)$-learning to the Soccer domain.

When on-line incremental algorithms are used, such as Q-learning or Sarsa, the target function is constantly changing. Thus, the observed errors will often be due to the fact that the training distribution and the target function has just changed rather than due to the approximator's inability to learn this function given enough training. Methods that rely on instantaneous errors would be very likely to insert many new units wastefully. Even in the approach of [Wiering, 1999], where the Responsibility variables are used for each unit, it seems that after a certain time, all units will have big Responsibility values and thus additions will always be allowed. Some form of temporal discounting, similar to the one used for eligibility traces, could help to account for non-stationarity. The method in [Fritzke, 1994], for example, is doing a form of temporal smoothening. However, such an approach is likely to be sensitive to a temporal smoothening parameter, as the amount and rate of change in the target function is not uniform throughout learning and very much task dependent.

Given that the error-based heuristics, in general, rely on several other user-defined parameters, such as the error threshold and the distance threshold, this approach may be very hard to tune overall.

As was already mentioned, in the case of reinforcement learning, large errors may result from the environment's stochasticity. Thus, with error-based methods, more units are likely to be allocated in regions with high variance of the value-function sample estimates. This may result in an undesirable effect of overfitting the noise.

In general, in reinforcement learning, it is not clear whether it is crucial to maintain high approximation accuracy all the time throughout the learning process, while policies keep changing. Often, just rough estimates of the relative magnitudes of the action values are sufficient for finding good policies. Allocating new units frequently in order to match sudden drops in the accuracy can create unnecessary large architectures and may even become a destabilizing factor for the entire policy learning process, as was observed, for example, in [Wiering, 1999].

Discussion

Given the fact that the target function is non-stationary in the reinforcement learning context, it can happen that some intermediate target functions will require more resources than others. Since such high resource requirements can be only transient, periodic pruning of the architecture may be beneficial to free up wasteful resources. Some algorithms, in fact, do this. For example, some methods start to operate with all available resources from the beginning of learning and then redistribute them later on. Next, we discuss different heuristics from the current literature used for determining which units can be removed/reallocated.

Activation-Based Removals of Local Units

In the instance-based approach used in [McCallum, 1996], the issue of limited storage capacity was addressed by keeping the instances in a sliding window over time. Once the window was filled, the oldest instance was removed and a new one

added. This approach results in a function approximator whose structure does not stabilize but keeps changing constantly throughout learning.

In [Hely *et al.*, 1997], an approach for dynamic allocation of SDMs was proposed, which dynamically added and removed memory locations. At the beginning of learning, the SDM was empty, and new locations were added to the memory as new training samples arrived until the memory capacity limit was reached. The location contents were not learned until all the locations were set up. At that point, content learning started, while also memory locations were competing for "survival" based on their activation frequency. After some period of time, those locations that were activated rarely got terminated and some memory space was thus made available for allocation elsewhere. Then, the memory was filled with some new locations again. Learning, thus, proceeded in two interleaving phases: location setup followed by content learning and prunning.

Other approaches that we discussed above, for instance, [Anderson, 1993; Sato and Ishii, 1998] also used the amount of activation as an indication of the utility of a local unit. We experimented with such ideas in the past, where each memory location accumulated its activations (e.g., the similarity measures as in Equation (5.1)), discounted over time in order to account for the non-stationarity. We observed that the activation-based approach for unit removals was not reliable with reinforcement learning. The reason is that it often results in the removal of locations associated with very important states, such as goal states and catastrophic states, which may have relatively low activation frequencies for some time after they have been initially discovered. If they are removed and if the value information has not propagated back sufficiently, good partial strategies may be lost. A similar remark was given in [Millán *et al.*, 2002].

Error-Based Removals of Local Units

Another natural criterion for unit removals could be based on the approximation error, similar to the one used for unit additions. The heuristic in [Forbes, 2002], used

with the memory-based learning for removing instances in the case, when the memory capacity limit is reached, was based on an error criterion. It suggested to discard the instances whose removal introduced the least error in the prediction of the values of their neighbors.

Another method in a similar spirit was presented in [Fritzke, 1997] for RBFNs. In this case, two measures were incrementally estimated throughout learning for each unit: an average of the prediction error and an average utility. The utility estimate of a unit was computed on every step as the increase in the approximation error caused by the removal of this unit. Then, if the ratio of the largest error (over all local units) to the smallest utility exceeded some threshold, the unit with the smallest utility was removed and a new one was inserted near the unit with the highest error. This heuristic, thus, attempted to weight a loss of accuracy in one region relative to a potential gain in another.

Discussion

Some of the approaches discussed above used heuristics for both the dynamic additions and removals of local units. Often, however, the criteria used for removing units were different from the ones used for adding units (see, e.g., [Anderson, 1993; Forbes, 2002]). For instance, units were removed based on the amount of their previous activation, while added based on some error heuristic. Our experimental results, which will be presented in Section 5.6, will show how such discrepancies in the addition/removal decisions can lead to undesirable effects and can prevent the function approximator from stabilizing its structure.

5.5. Resource Allocation for SDMs: Our Approach

In the previous section, we reviewed different methods from the current literature, with which local units of the function approximators can be tuned, added or reallocated. We pointed out various difficulties that can be encountered when applying such methods in the reinforcement learning setting. In this section, we address the

question of choosing locations for SDMs in the context of on-line action-value function approximation. As previously mentioned, in our preliminary investigations, we experimented with a variant of the self-organizing approach of [Sutton and Whitehead, 1993] and with variants of the activation-based heuristics for reallocation of SDM locations. Unfortunately, we did not find these attempts successful in our empirical studies.

We designed a new approach, which is computationally cheap, robust in the face of the inherent non-stationarity with on-line reinforcement learning and allows coherent extensions and incorporation of new structure building criteria in the future. As the first step, we focused on the LMS learning with the Sarsa algorithm, although our approach can be used with other learning methods, as will be discussed later in Section 5.7.

In the remainder of this section, we describe our approach for determining automatically the SDM size and addresses of the memory locations based on the observed data. In this thesis, we assume that the activation radii of the memory locations are uniform and fixed by a user. The extensions of our approach for dynamically choosing these parameters are left for future work while some ideas are outlined in Section 5.7.

First of all, our algorithm presents a way to decide when and how to add new locations to the memory. This method is based on the ideas of the activation-based addition heuristic discussed earlier in the previous section. We also provide a method to decide which locations can be removed from their current positions and added elsewhere, in the case where the memory capacity limit is reached. This method resulted from our preliminary experiments, analysis and modifications of the self-organizing approach from [Sutton and Whitehead, 1993].

5.5.1. Algorithm for Adding SDM Locations

Our algorithm, which we will refer to as the *dynamic allocation* method, starts with an empty memory, and locations are added based on the observed data. Since the samples obtained during on-line reinforcement learning are highly correlated,

memorizing all samples until the memory is filled can create unnecessary densely populated areas, while leaving other parts of the state space uncovered. Hence, our goal is to add locations only if the memory is too sparse around the observed training samples.

Our algorithm relies on a user-defined parameter, denoted N, which is the minimum number of locations that we would like to see activated for a data sample. It is important to ensure that these locations are "evenly distributed" across their local regions, so that their activation neighborhoods do not overlap redundantly. Hence, we do not allow locations to be too close. More specifically, for any pair of locations $\mathbf{h}^i, \mathbf{h}^j$, we enforce the following condition on the similarity between them:

$$\mu(\mathbf{h}^i, \mathbf{h}^j) \leq \begin{cases} 1 - \frac{1}{N-1} & N \geq 3 \\ 0.5 & N = 2 \end{cases} \tag{5.37}$$

This condition implies that the fewer locations are required in a local neighborhood (the smaller N), the farther apart these locations should be.

A new location can be added upon observing any training sample $\langle (s,a), \bar{Q}(s,a) \rangle$, where $s = \langle s_1, ..., s_n \rangle$ serves as input to the SDM for the action-value function of action a, and $\bar{Q}(s,a)$ represents the target for the currently observed state-action pair (s,a). For example, $\bar{Q}(s,a) = r + \gamma Q_{\mathbf{w}}(s',a')$ in the case of the Sarsa algorithm. The following heuristic is aimed at ensuring a minimum of N active locations in the vicinity of s:

`Rule 1:` If fewer than N locations are activated by the input s, add a new location centered at s, if its addition does not violate condition (5.37) with respect to the existing locations. Store the current target value, $\bar{Q}(s,a)$, in this new location.

We also experimented with an option of storing in the newly added location the value currently predicted by the SDM, $Q_{\mathbf{w}}(s,a)$, instead of the training target $\bar{Q}(s,a)$. However, in our experiments, this variant performed slightly worse. Given that with

our heuristic, new locations are added in very sparse regions, their addition directly with the target values does not seem to disturb the approximation very much.

A pseudocode of the implementation of `Rule 1` is provided as Algorithm 1 in Appendix B.1.

In our experiments with various training scenarious, we noticed that if, during learning, there is not enough exploration to ensure a good spread of the visited states, the allocation using only the above heuristic rule proceeds very slowly. We observed situations, where, due to a small number of locations in some region of the state space (a small isolated group), successive updates of the action-value function would result in a chattering phenomenon, where the agent starts to oscillate between several policies that keep it trapped in the coresponding region. With a small exploration rate, the agent has a difficulty to break this cycle and to start exploring other parts of the state space. In this case, no new locations are added to the SDM and learning can be stalled for a long time. This can happen, for example, in domains, where distances between consecutive states are relatively small with respect to the activation widths of the memory locations, and where some actions are symmetric (e.g., "move left" and "move right"). To counteract this problem, we extended the above heuristic in the way that facilitates the agent's progress by setting up the memory resources faster. New locations are positioned slightly off the observed trajectory, but still relatively close to the actual training data samples. The new rule is formulated as follows:

`Rule 2`: If after applying `Rule 1`, the number of active locations is N' (including a location added by Rule 1, if any), and $N' < N$, then $(N - N')$ locations are randomly placed in the neighborhood of the current sample state s. The addresses of new locations are sampled uniformly randomly from the intervals $[s_i - \beta_i, s_i + \beta_i]$ in each dimension, while ensuring that they satisfy the condition (5.37). The value currently predicted by the memory for the corresponding address is stored in each such location.

A pseudocode of the implementation of the `Rule 2` is provided as Algorithm 2 in Appendix B.1.

The parameter N (minimum desired number of activated locations) in the above heuristics is reminiscent of the parameter k in the k-nearest-neighbor methods, which determines the number of instances that are used for locally weighted learning. Contrary to the classical instance-based approach, our method provides a way to selectively store training samples to obtain a good space coverage with respect to this parameter while controlling the memory size.

As previously mentioned, our method for adding new locations to the SDM architecture is activation-based (similar as those in [Kondo and Ito, 2002; Anderson, 1993; Šter and Dobnikar, 2003]), that is, it is directly related to the similarity measures used by the memory locations. It ensures that the memory locations are spread appropriately with respect to the radii of the employed similarity function and can easily accommodate the case of variable widths.

In our approach, the similarity (activation) threshold is implicitly derived from the condition (5.37) based on the parameter N. It should be noted that, in general, more than N training samples can satisfy the derived similarity threshold. Using the parameter N in combination with the corresponding similarity threshold, as it is done in `Rule 1` and `Rule 2`, provides a more conservative way to control the size of the memory, as will be illustrated by our experiments in Section 5.6. In the sequel, we will refer to our dynamic allocation method based on Rules 1 and 2 as the *N-based heuristic*, to distinguish it from the approach that employs the similarity (activation) threshold alone.

5.5.2. Algorithm for Reallocating SDM Locations

If the memory size limit is reached but we still encounter a data sample for which the number of active locations N' is smaller then the minimum desired number N, we also allow existing locations to move around. Unlike the approach described in [Sutton and Whitehead, 1993], we do not adjust the existing addresses slowly.

Instead, we apply the following rule. Suppose that L locations need to be added to the neighborhood of the current input state s based on `Rule 1` and `Rule 2`. Then we proceed as follows:

`Rule 3:` Pick at random and remove L inactive locations. When removing some location $\mathbf{h}$, find, among locations in the active set $H_{\mathbf{h}}(a)$, a location $\mathbf{h}'$ that is closest to $\mathbf{h}$. Then, replace $\mathbf{h}$ and $\mathbf{h}'$ by another location, $\mathbf{h}''$, placed midway between them. The value of $\mathbf{h}''$ is set to the average of the values of $\mathbf{h}$ and $\mathbf{h}'$. After removing L inactive locations in this manner, add the corresponding number of new locations to the neighborhood of the current state s using `Rule 1` (and optionally `Rule 2`).

The approach described above, which we call *randomized reallocation*, allows the memory to react quickly to the lack of resources in the regions visited under the current behavior policy. At the same time, because of the fact that there are sufficient locations in most of the previously visited regions, and because the choice of the location to be removed is random, removals do not dramatically affect any particular area of the input space. The method is cheap, both in terms of computation and space, since the choice of the locations to be removed is not based on any extra information, like in other algorithms, e.g., [Fritzke, 1997; Hely *et al.*, 1997; Kondo and Ito, 2002; Forbes, 2002].

A pseudocode of the implementation of `Rule 3` is provided as Algorithm 3 in Appendix B.1.

5.5.3. Putting It All Together

Resource allocation proceeds in parallel with learning the memory content. On each learning step, i.e., after observing the training sample $\langle (s,a), \bar{Q}(s,a) \rangle$, new locations are added or moved, if necessary, then the values stored in the memory are updated according to the LMS training rule as presented in Section 5.3. A pseudocode implementation of the entire learning step is provided in Algorithm 4 in Appendix B.1, assuming reinforcement learning without eligibility traces, e.g., Sarsa(0).

5.5.4. Adjusting the SDM Architecture on Prediction Steps

Another feature that we experimented with is to perform SDM structure adjustments (additions and reallocations) on prediction accesses to the memory in addition to the learning accesses. Recall that during reinforcement learning with a semi-greedy exploration strategy, on every step, predictions of values are performed for *all* actions in order to determine a greedy action, whereas a learning update is performed only for the performed action. In our experiments, allowing resourse adjustments on predictions proved to be very beneficial. It allows the SDMs for all actions to consistently adapt their layouts to the current state distribution, as opposed to adapting them only when the corresponding action is performed. The values of all actions are important for the current state distribution to properly determine greedy actions. This approach is particularly important with our randomized reallocation method. It prevents removing too many locations (so that there remain less than N of them) from the architecture of some action a even if the states in some region are not visited with that particular action, but their values are accessed to choose the greedy action. In other words, this ensures that the neighborhoods of important but not actually visited state-action pairs are not emptied. If a new location is added on a prediction access to the SDM, the value currently predicted for the corresponding input address is stored in it.

5.6. Experimental Results

In this section, we present the results of our experiments, where the SDMs were used for action-value function approximation. We used the Sarsa(0) algorithm with the ϵ-greedy exploration strategy. First of all, we provide our experimental results on the standard Mountain-Car benchmark, which, as previously mentioned, is commonly used in the reinforcement learning community. Then we also present the results on a variant of the hunter-prey domain. We experimented with instances of this domain with varying numbers of state dimensions, up to 11 state variables.

5.6.1. Mountain-Car Domain

Recall from the previous chapters that the Mountain-Car is an episodic goal-directed task with a two-dimensional continuous state space, where the agent has to learn to drive up the hill from a valley (see [Sutton and Barto, 1998] for exact specification of the environment's dynamics). In our experiments, episodes were terminated when the goal was reached, or after 1000 steps.

To obtain a baseline performance, with which we could compare performance of the SDMs, we used the popular CMAC (tile coding) approximator (see Chapter 2). This approximation model is known to be particularly successful on the Mountain-Car domain. Recall that CMACs are related to SDMs, but the CMAC layout is fixed a priori, with the locations (tiles) arranged in several superimposed tilings, which are regular discretization grids. Each input activates one tile in each tiling, and the activation mechanism is binary. Since CMACs rely on the discretization of the input space, their size scales exponentially with the input dimensionality.

In our experiments, we used CMACs and SDMs with the following correspondence in their resolutions: If a CMAC had T tilings (thus T tiles were activated on each input), we set the parameter N for our dynamic memory allocation with the N-based heuristic as $N = T$. The activation radii of the SDMs were set equal to the size of the CMAC tiles.

In addition to testing our dynamic allocation approach, we also tested a method, where the decision on whether to add a new location was based only on the similarity threshold, without checking whether the number of active locations already exceeds N. This approach is in the style of other activation-based methods from the current literature, e.g., [Kondo and Ito, 2002; Anderson, 1993; Millán *et al.*, 2002] (see Section 5.4). In other words, a new location was added whenever the similarity of the current data sample to all existing locations was below some threshold μ^*. We will refer to this method as the *threshold-based heuristic* in the sequel. In this case, we experimented with the similarity thresholds μ^* set to the values that would be derived from the condition in Equation (5.37) for the values of N and the activation radii used in

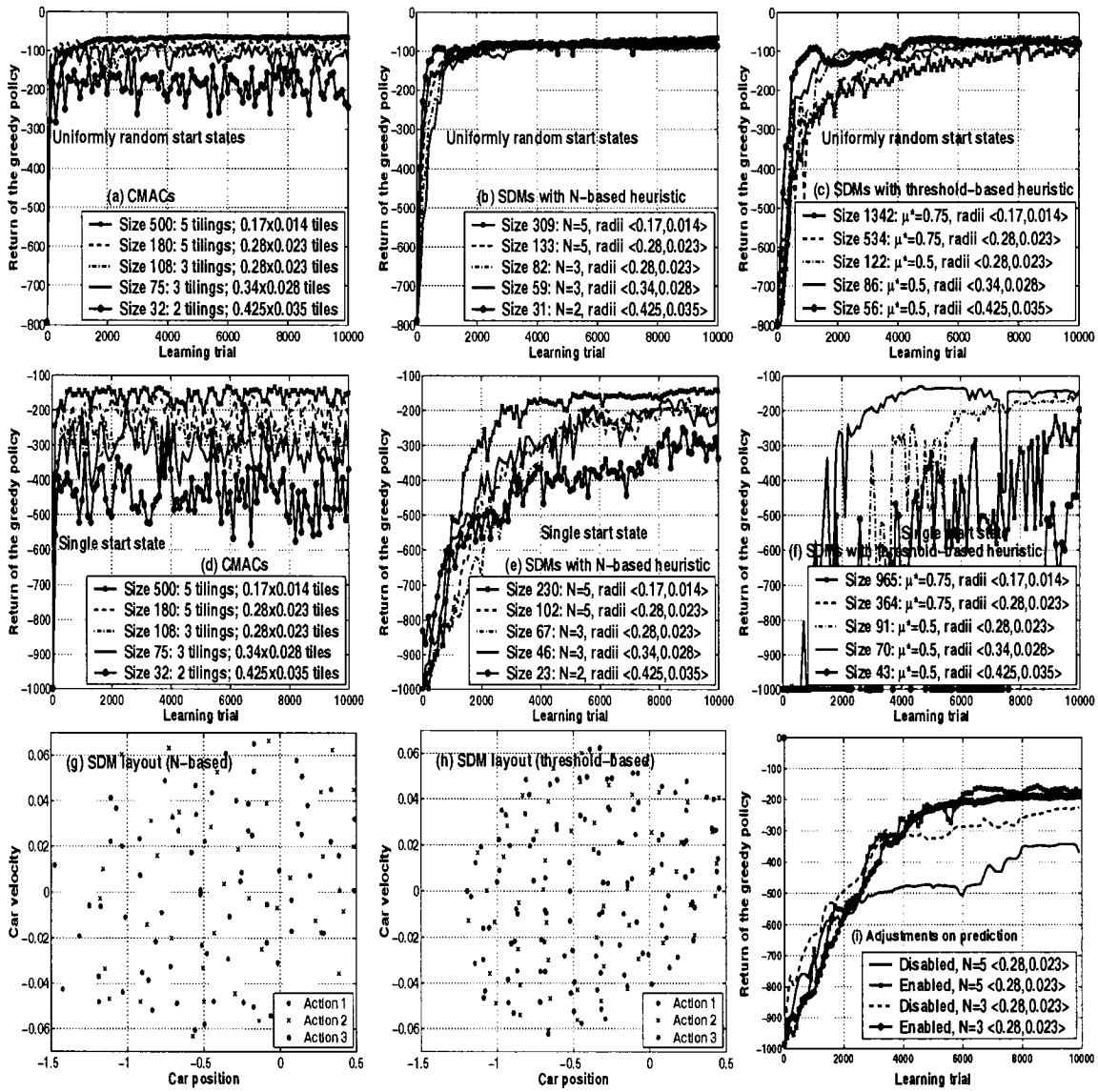


FIGURE 5.3. Dynamic allocation method. Returns of the greedy policies are averaged over 30 runs. On graphs (a)-(c), returns are also averaged over 50 fixed starting test states. SDM sizes represent maximum over 30 runs. The exploration parameter ϵ and the learning step α were optimized for each architecture. Graphs (g) and (h) are for SDMs with radii $\langle 0.34, 0.028 \rangle$, and $N = 5$ and $\mu^* = 0.5$ respectively.

the corresponding experiments with the N-based heuristic. The objective was to investigate the differences in the resulting memory sizes, layouts and the performance based on the two approaches.

We conducted two sets of experiments to investigate the performance under different exploration scenarios and under different training sample distributions. In the first set, the start state of each episode was chosen uniformly randomly across the entire state space. This is the most popular experimental setting. It presents an "easier" training scenario, because the starting distribution ensures, to a large extend, good (sufficient) exploration regardless of the exploration strategy followed during learning.

Graphs in the upper row of Figure 5.3 show the results in this training setting. The graphs (a), (b) and (c) present the returns of the greedy policies learned by CMACs, dynamically allocated SDMs with the N-based and the threshold-based heuristics respectively. In these experiments, the memory size limit was set sufficiently high to ensure that it would not be reached and we could test the dynamic allocation method alone.

As can be seen from these graphs, the performance of the SDMs is either the same or (in most cases) much better than that of CMACs. It degrades more gracefully with the decrease of resolution. Moreover, SDMs with the N-based heuristic (in this case using `Rule 1` only) always consume fewer resources, as shown in the legends of the graphs. The asymptotic performance of the SDMs with the threshold-based heuristic is similar to that of the N-based heuristic, however the learning is slower. The resulting memories are between 2 and 4 times larger with the threshold-based heuristic. This can slow down learning, because more training is required for larger architectures. As we anticipated, the N-based heuristic enables a better control over the amount of allocated resources. As the experiments show, it results in faster learning, while preserving the quality of the learned policies.

In the second set of experiments, we used a single start state for each episode. In this case, the car always starts at the bottom of the hill with zero velocity. In this setting, exploration is much more difficult, as it is always constrained by the behavior policy. Thus, there is a closed cycle where the exploration depends on

the learned action-value function, which, in turn, depends on the exploration policy. We specifically wanted to test the performance of SDMs in this setting, where the training samples are highly correlated and distributed non-uniformly. The corresponding results are shown in the middle row of Figure 5.3. In this case, SDMs were built with the N-based heuristic using both `Rule 1` and `Rule 2`. Without `Rule 2`, which allows adding locations close to but not exactly on trajectories, we observed the policy oscillation phenomenon, discussed in Section 5.5, due to poor exploration. SDMs generally learn better policies than CMACs and take advantage of the fact that not all the states are visited. The resulting memory sizes for SDMs (graph (e)) are roughly 30% smaller than in the previous experiment (graph (b)). SDMs with the threshold-based heuristic, however, were much slower and exhibited a much higher variance with this single start-state training, even though they had a large number of locations placed exactly along the followed trajectories. This demonstrates that, with the restricted exploration, `Rule 2` of our approach helps to build quickly a compact model with good generalization capabilities. Also, under limited exploration, smaller architectures (as obtained with the N-based heuristic) should be expected to learn better as they suffer less from overfitting[14].

Graphs (g) and (h) of Figure 5.3 show examples of SDM layouts obtained with the N-based and the threshold-based heuristics for these experiments. The SDMs obtained with the N-based heuristic are less dense and span the state space better.

Finally, graph (i) of Figure 5.3 shows the performance improvement achieved by allowing adjustments to the memory layout during predictions as well as during reinforcement learning updates. As can be seen from the graph, learning curves attain high levels of performance much faster with this feature enabled. Note that all the experiments with the SDMs (for both the N-based and the threshold-based heuristics) discussed above were performed with this option enabled.

[14]Overfitting happens when a model, that has a lot of parameters (large capacity) memorizes the training samples too much and does not generalize well to new samples.

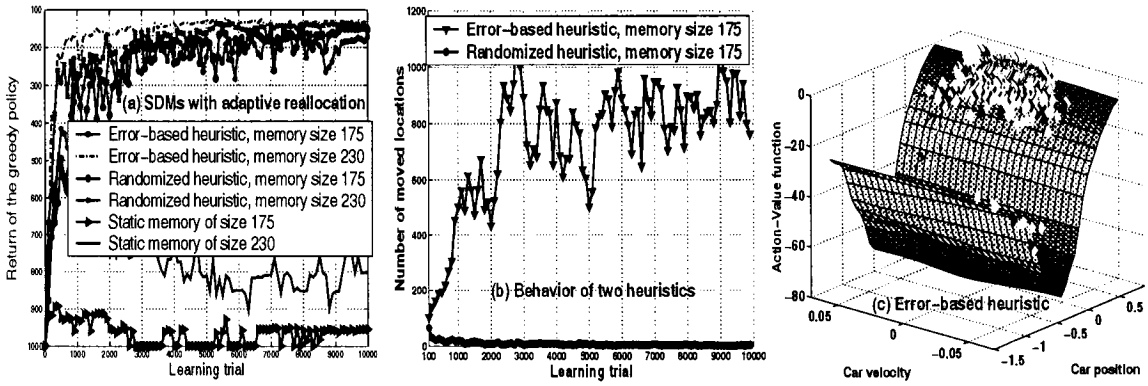


FIGURE 5.4. Randomized reallocation method. Each point on graph (b) represents the average over 100 trials and 30 runs. Graph (c) depicts an action-value function for action "positive throttle".

Most of the resourse allocation happens early in the learning process. In our experiments, both with the N-based and the threshold-based heuristics, $\sim 85\%$ memory locations where added in the first 200 trials.

Graph (a) of Figure 5.4 shows the performance of the adaptive reallocation method, which allows moving the existing locations when the memory size limit is reached. The experiments were performed for the single start-state training scenario on the Mountain-Car domain using the N-based heuristic for location additions. We tested two removal approaches: the randomized reallocation method, introduced in this thesis, and an error-based method suggested in [Forbes, 2002]. For each local unit, this method measures an average error in the prediction of the values at the centers of the neighboring units, introduced by the removal of this unit. Using our SDM notation, such an error can be computed as follows:

$$error^m = \frac{1}{|H_{\mathbf{h}^m}|} \sum_{k \in H_{\mathbf{h}^m}} |Q(\mathbf{h}^k, a) - Q_{-m}(\mathbf{h}^k, a)| \tag{5.38}$$

where $Q_{-m}(\mathbf{h}^k, a)$ is the prediction for input $\mathbf{h}^k$ (and action a) without the location $\mathbf{h}^m$.

Graph (a) shows experiments with memory parameters $N = 5, \beta = \langle 0.17, 0.014 \rangle$. The memory size limits were chosen to be equal to 230 and 175 which is 100% and 75% of the size obtained for the same memory resolution with the dynamic allocation

method and the N-based heuristic in the previous experiments. The SDMs were initialized with all locations distributed uniformly randomly across the state space and then allowed to be reallocated according to the heuristics used. Note that the static memories of the same sizes were not able to learn a good policy.

As can be seen from graph (a), both removal heuristics succeed in building SDMs that provide good performance. However, as shown on graph (b), the behavior of the two heuristics is quite different. This graph depicts the number of reallocations throughout learning. Each point on the curves represents the average number of reallocation over the last 100 learning trials. With the randomized heuristic, most reallocations happen at the beginning of learning and then their number decreases almost to zero. With the error-based heuristic the number of reallocations stays much higher until the end of the learning runs. Close examination of the logs of the learning runs provided us with an insight into a cause of such behavior. Our conclusion was that a high reallocation rate was maintained by the error-based removal heuristic because the objectives of the criteria for additions and removals of the memory locations were not "in agreement": addition heuristic was density-based and the removal heuristic was error-based. Graph (c) depicts 3000 location moves at the end of one training run, where removed locations are plotted with black dots and added locations - with white dots. A mixed black-and-white cloud in one region of the state space shows that most error-based removals happen in a particular region where the value function is relatively flat, but the same region is then visited and found to be too sparsely represented by the density-based addition heuristic. Thus, locations are added back. Apparently such a cycle repeats itself. As mentioned earlier, with the randomized reallocation heuristic, no specific area of the input space is affected by removals more than others, thus cyclic behavior is minimized.

It should also be noted that the error-based heuristic is more expensive computationally: it requires either to perform a complete memory sweep when reallocation is necessary, or to perform $(|H_{\mathbf{h}^m}| - 1)$ additional predictions on every memory access in

order to maintain (approximate) error estimates. The cost of the randomized heuristic, on the other hand, is that of generating a random number and applies only when a new location actually has to be added in an underrepresented region of the input space. Thus, the randomized reallocation method is computationally much cheaper while showing more stable behavior and providing good policies. The error-based removal heuristic can still be an interesting choice, provided that it is in tune with the addition heuristic.

5.6.2. Hunter-Prey Domain

The second task we used in our experiments is a variant of a hunter-prey domain often used for testing multi-agent systems. In this domain, H hunters team up with an objective to capture a prey. In our setting, reinforcement learning was used to learn how to control the prey, while the hunters behave according to fixed, heuristic strategies. We chose this task because we wanted to study the effect of increasing the dimensionality of the state space on the quality and speed of learning with SDMs, without increasing the number of actions available.

In this task, the state is described by $2H$ continuous variables, representing the position of each hunter relative to a polar coordinate system centered on the prey, and one integer variable, representing the number of hunters still alive. In order to capture the prey, C hunters have to approach it within a circle of radius equal to 5, and the angle between adjacent hunters has to be less than $(\frac{2\pi}{C} + 0.6)$ radians. However, if fewer than K hunters are within 5 units of the prey, the prey kills the closest one. The hunters start each episode at random positions inside a circle of radius 50 around the prey. A stochastic controller moves the hunters individually as follows: with probability 0.3, a hunter moves clockwise or counterclockwise 0.2 radians; with probability 0.7, the hunter moves in the radial direction, either towards the prey, if there is no danger to be killed, or otherwise away from the prey by 5 units. The episode ends when the prey is captured or when fewer than C hunters are alive (the minimum number of hunters necessary to capture the prey). The prey can move

north, south, east or west, 5 units per time step. The reward scheme is the following: the prey receives a reward of 1 if it kills a hunter, -200 if it is captured, and -1 per time step otherwise, thus encouraging the prey to learn an aggressive strategy and kill hunters as soon as possible.

Figure 5.5 presents the results of experiments with 2-, 3- and 5-hunter tasks (5, 7 and 11 state dimensions respectively). In each case, we used $C = H$ and $K = 2$. Thus, the complexity of the prey's task does not increase: it still has to kill one hunter to win the episode. However, the dimensionality of the state space increases with the increase of the number of hunters. Graph (a) shows the performance of the SDM and CMAC architectures on the 2-hunter task. The SDMs were trained with the dynamic allocation method using `Rule 1` only (as the start state distribution provides a good coverage of the state space). As shown in the graph, CMACs were not able to learn the task even with considerable memory sizes. Consequently, we experimented with the SDMs only on the task instances with 3 and 5 hunters. In these cases, the SDMs were empty at the beginning of learning. Locations were added then with the dynamic allocation method (using `Rule 1` only). Then, when the memory limit was reached, the randomized reallocation method was used.

Graph (b) shows the results on the 3-hunter task, illustrating the effect of changing the activation radii and the value of N. The choice of the activation radii seems to have a stronger impact on the overall performance compared to the N parameter. For large radii, smaller values of N seem to work better[15]. Graph (c) shows the average learning curve for the 5-hunter task, using the SDM resolution that seemed best in the 2- and 3-hunter tasks. Despite the higher dimensionality of the task, the performance achieved is the same as in the 2 and 3-hunter case, while the memory size was not allowed to increase significantly compared to the 3-hunter problem.

[15]The same trend appeared in the Mountain Car domain.

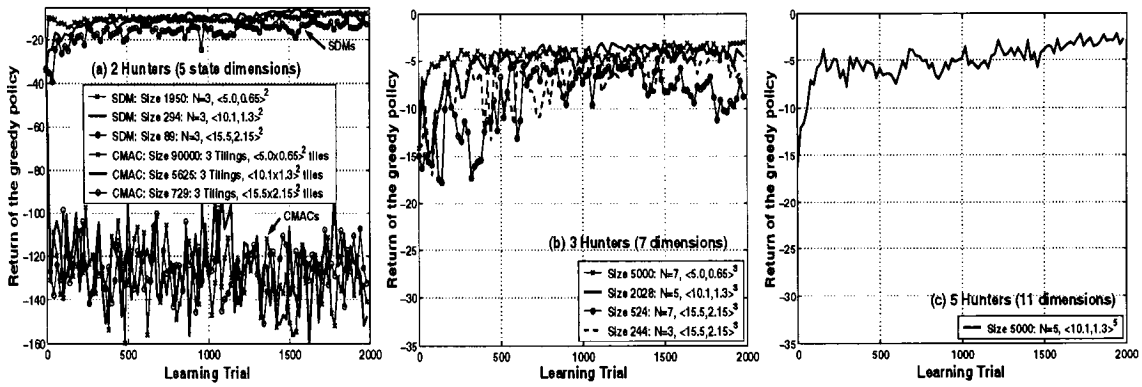


FIGURE 5.5. Hunter-prey domain. Returns are averaged over 20 runs and 100 fixed starting test states, sampled uniformly randomly. The exploration parameter $\epsilon = 0.05$ and the learning step α was optimized for each architecture and task.

5.7. Discussion and Future Work

Alternatives to LMS Learning

In our experiments, we performed on-line LMS training based on the Sarsa(0) algorithm with the ϵ-greedy exploration strategy. Recall from Chapter 2 and Section 5.3.2 that there are other ways in which approximate reinforcement learning can be performed. For instance, in Section 5.2.3, we discussed a number of approaches with stronger theoretical guarantees, which relied on the use of linear (in some cases local) function approximators. Thus, SDMs and the proposed approach for allocating memory resources can potentially be applied with these methods, as we will discuss next.

Recall the approaches in [Gordon, 1995] and [Tsitsiklis and Van Roy, 1996], which use the approximate Value Iteration algorithm with averagers in the former case and linear approximators in the latter case. In both cases, the function approximator's architecture has to be configured prior to learning. In Section 5.2.3, we already mentioned the possibility of doing this dynamically, while following some sampling strategy, e.g., a uniformly random policy, a policy based on the domain knowledge or simply using a set of previously collected samples. We sketched a possible procedure

for allocating SDMs so that the conditions (5.19), required for the algorithm's convergence, are satisfied. This procedure can be related to the approach we introduced in Section 5.5 for resourse allocation with SDMs.

Suppose, we would use unnormalized similarity measures in the representation of the approximate function, as in Equation (5.20), in order to allow activation neighborhoods of the memory locations to overlap with the centers of other locations (see discussion in Section 5.3.2). In order to satisfy the conditions (5.19), we could proceed with the allocation of new memory locations in a way similar to the `Rule 1` of our dynamic allocation approach. We would choose a setting of the parameter N, but then set the similarity threshold to some value $\mu^* < \frac{1}{N}$. According to `Rule 1`, we would add a new memory location at some address s only if less than N existing locations are activated by this address and the similarity with the closest location is less than μ^*. In this case, the condition (ii) in (5.19) is satisfied, since for every memory location $\mathbf{h}^m$, we have:

$$\sum_{i=1, i\neq m}^{M} \mu^i(\mathbf{h}^m) \leq (N-1)\mu^* < \frac{N-1}{N} < 1$$

The value of δ used in (5.19) can then be found as $\delta = (N-1)\mu^*$. As previously suggested, in order to satisfy condition (iii) in (5.19), we could check whether it is violated for every sample state, and if it is the case, remove some of the active locations. This would limit the maximum number of active locations in any given region of the input space. Note, that this method is likely to produce memories that are sparser than the ones resulting from our allocation approach introduced in Section 5.5. This is due to the choice of a smaller similarity threshold, $\mu^* < \frac{1}{N}$ instead of by Equation (5.37), and also due the attempts to satisfy the condition (iii), in which case some previously added location may be removed.

As previously noted, when we use normalized similarities in the representation of the approximate function and do not allow any memory location to activate other

locations, that is $\sum_{\substack{i=1 \\ i \neq m}}^{M} \mu^i(\mathbf{h}^m) = 0$, $m = 1, ..., M$, the architecture satisfies the interpolator property (5.24) and thus can be used with the averager update as in Equation (5.25). The convergence proof for this learning update extends to the case when a set of basis points (on which interpolator is defined) is dynamically adapted, as was shown in [Szepesvári and Smart, 2004]. The SDM locations can be viewed as such basis points. It is assumed in this case that updates to the set of basis points are performed in parallel with the updates (5.25) (similar as we conduct the resource allocation in parallel with the LMS learning). According to [Szepesvári and Smart, 2004], the updates to the set of basis points have to satisfy the following conditions: (i) the updates are performed based on the past observations; (ii) on any step, the maximum number of points added is bounded by some constant; (iii) the set of basis points cease to change after a finite amount of time. Moreover, if the target value function Q^* is Lipschitz[16], and the density of the set of basis points is bounded from below by some constant d_0, then the algorithm converges with probability 1 to the parameter vector w^* of the interpolative function approximator $f_{\mathbf{w}}$, such that $\|f_{w^*} - Q^*\| \leq O(\frac{d_0}{1-\gamma})$, where Q^* is the optimal action-value function. The work in [Szepesvári and Smart, 2004], however, does not provide any constructive algorithm for adaptively choosing the basis points.

Our algorithm for dynamically adding and reallocating memory locations can potentially be used in that framework. It satisfies conditions (i) and (ii). The condition (iii) will be satisfied automatically if the memory size is chosen large enough to cover the entire state space with respect to the radii of the similarity measure. In this case, our addition rules will stop adding new locations to the memory. However, memory size would scale exponentially with the input dimensionality in this case. The assumption of a fixed stochastic exploration strategy used in [Szepesvári

[16]A Lipschitz function is a function $f : \Re \rightarrow \mathbb{C}$, for which there exists a constant $c \in \Re$ such that for all $x, y \in \Re$, $|f(x) - f(y)| \leq c|x - y|$.

and Smart, 2004][17], is quite restrictive: as all of the state space will be repeatedly revisited throughout learning, a local function approximation architecture may need to cover the entire state space and not only a subset important to the learned policy.

It would be interesting in the future to compare the performance of the LMS training with the convergent approach of [Szepesvári and Smart, 2004], since at present, there is relatively little practical evidence for the performance of the latter.

Automatic Selection of the Activation Radii

One of the issues that we will address in the future is the automatic selection of the activation radii for the SDM locations while allowing them to vary across the state space. We believe that this is crucial in order to be able to further reduce the memory requirements and to optimize the memory layout. Several ideas from the current literature can be taken as a starting point in this case.

For instance, the work of Weaver *at. al.* (1998) presents a notion of *interference* of learning on two training samples. The interference is defined as a normalized dot product of the corresponding gradient vectors used in the gradient descent learning. This dot product can, thus, indicate whether two training samples cause parameter updates in opposing directions. In our case a similar idea can be used, where gradient vectors would be accumulated in each memory location. Then a dot product of these vectors can be considered for each pair of locations whose activation neighborhoods significantly overlap. If the dot product indicates that these two locations are often updated in very different directions, the overlap between them should be reduced by adjusting their activation widths.

The ideas of temporal neighborhoods, as were explored in [Kretchmar and Anderson, 1999] and [Glaubius and Smart, 2004] can also be further explored for the purpose of selecting the activation radii. In this case, memory locations can be allowed to dynamically stretch and shrink their activation neighborhoods based on the observed patterns of the state transitions.

[17]With an absence of any prior domain knowledge, a uniformly random exploration strategy would need to be used.

Incorporation of Other Allocation Criteria

Our current resource allocation mechanism is based only on the distribution of the inputs and is concerned with the density of the allocated memories. In the future, we also plan to explore other criteria that use information about the function topology, e.g., function linearity and decision boundaries, as in [Munos and Moore, 2001; Reynolds, 2002]. We would like to investigate how such criteria perform in an on-line setting and how they can be paired with the criteria for removal of the memory locations.

Learning with Progressive Refinement

Another idea that seems to be promising for obtaining better performance while minimizing the increase in the computational time, is that of a *progressive memory refinement*. In this case, a coarser memory resolution can be used at the beginning of learning (corresponding to large activation radii) and later a finer memory can be introduced. A small memory would first learn a coarse representation of the action-value function and then it would be refined where necessary. The fine-grained SDM can inherit locations from the coarse SDM, with their already reasonable values; hence, learning can proceed faster. Also, as certain parts of the state space would be visited less with a reasonable coarse policy, hopefully fewer new locations would be added to the finer SDM. In contrast to other approximators, e.g., neural networks, SDMs allow to change structural parameters, without loosing everything that have been previously learned.

The idea of the progressive refinement is reminiscent of the approach suggested in [Rust, 1997] for the random self-approximating Bellman operator (see Section 5.3.2), in which case sets of sample states of increasing size are used for successive runs of the corresponding approximate Value Iteration method. In this case, each run starts with initial values for the sampled states estimated by the value function computed on the previous run. This method allowed a significant reduction in the computational time in the theoretical analysis of Rust (1997).

A similar idea was used in [Platt, 1991], where the distance threshold for the distance-based addition heuristic was reduced over time. In this way, the architecture first allocated features, which were very broad and widely spread, and then progressively refined them.

Another related approach was developed in [Vollbrecht, 1999], where a *kd*-tree was used to fully partition the state space with respect to a given depth of the tree. Then action-values (and their confidence levels) were estimated by the Q-learning algorithm simultaneously at all levels of the tree (i.e., for different granularities of the state space representation). Computationally, this approach leveraged the fact that most of the finest-resolution levels were rarely visited. The algorithm decided at which level of the tree it had most confidence in the corresponding value estimates and used these values to determine a policy and values for the bootstrapped targets. This method can be applied only to MDPs with the state space dimensionality that allows to construct a full tree of the required depth, as the number of leaves (partitions) scales exponentially with the number of state dimensions.

Theoretical Analysis

Finally, we will investigate formal theoretical properties of the SDM model and the dynamic allocation approaches, for instance, in connection with the results in [Tsitsiklis and Van Roy, 1996] and [Szepesvári and Smart, 2004].

5.8. Summary

In this chapter, we investigated the use of the Sparse Distributed Memory model for action-value function approximation in on-line reinforcement learning. This model is local and linear, which is often preferred in reinforcement learning. It is also flexible enough to scale well with large and highly dimensional input spaces, if the underlying domain has properties that do not necessitate all of the state space to be (well) covered by the memory locations.

The main contribution of this chapter is a new approach for dynamic allocation and adaptation of the SDM resources. As can be seen from the literature review and various discussions presented in this chapter, the problem of automatic structure selection for local function approximators is very important for the practical applicability and scalability of reinforcement learning techniques and is being actively researched in the reinforcement learning community.

Our algorithm (as presented in Section 5.5) provides the ways to decide where new memory locations are necessary and which locations can be reallocated if the resources are limited. It provides a disciplined way to control the growth of the SDM size and its density based on the distribution of the training inputs. We empirically demonstrated that our approach is suited for on-line value-based reinforcement learning, where other related supervised learning techniques can become unreliable. We showed the importance of the agreement between the criteria used for adding and removing memory locations which have to be used in tandem when the memory limit is reached.

Our approach is very efficient computationally and does not introduce any additional storage requirements, contrary to many other related approaches from the current literature. Our method facilitates learning under constrained exploration conditions, which are likely in practical applications. We also proposed a new idea of adjusting memory resources on the prediction accesses and experimentally showed its positive effect on performance.

While we demonstrated a successful application of the proposed approach with the LMS training and on-policy exploration, we also discussed the possibility of using this approach with other approximate reinforcement learning methods, which opens new interesting directions for future research. The algorithm developed and the lessons learned here can be applied to other local function approximators. In this chapter, we discussed the relationships of some of them to the SDM model.

CHAPTER 6

Conclusions and Future Work

6.1. Contributions

This thesis is composed of two parts. One part addressed the issues of identifying, analyzing and measuring MDP characteristics that influence the performance of incremental value-based reinforcement learning algorithms, as well as using the corresponding measurements in practice to improve the efficiency of learning. The second part was devoted to the use of Sparse Distributed Memories as a function approximation model for representing value functions. In summary, this thesis contains the following contributions.

MDP Attributes. Most results for reinforcement learning techniques in the current literature are based on very general assumptions about the underlying MDPs. Very little research was devoted in the past to the analysis of specific properties of MDPs that can influence performance of reinforcement learning algorithms. One such study [Kirman, 1995] was devoted to dynamic programming algorithms for planning in MDPs and, indeed, demonstrated that MDP characteristics affect the performance of such algorithms under time constraints. To the best of our knowledge, no such analysis was performed in the past for incremental value-based reinforcement learning algorithms.

In this thesis, we discussed various challenges faced by on-line value based methods and analyzed how particular MDP properties relate to these challenges and how

they can influence the performance. We provided several hypotheses about the nature of the effects that these MDP characteristics can have on learning, which are related mainly to the sample variance, ease of exploration and the ability of the agent to influence its environment. Based on this analysis, we introduced five quantitative attributes for characterizing and measuring certain properties of MDPs: the State Transition Entropy, Variance of Immediate Rewards, Controllability, Reward Information Content and Risk Factor. They reflect mostly the amount and sources of stochasticity in an MDP as well as the amount of control that the agent can exercise over its environment. We presented ways in which these MDP attributes can be computed, either exactly if the MDP model is available, or estimated approximately based on experience (either before or during learning). For the case in which the attributes have to be estimated during learning, we discussed efficient incremental methods.

While analyzing the relationships between the MDP properties and the dynamics of on-line value-based reinforcement learning algorithms, we discussed how one can take advantage of knowledge about these properties to facilitate learning by (dynamically) tuning various parameters that typically need to be chosen by the user.

We presented the results of an empirical study, involving the two most important attributes, the State Transition Entropy and the Controllability. The study shows the existence of a statistically significant relationship between the measured properties and the performance of incremental value-based algorithms. We performed experiments both for tabular and for approximate methods and showed that the effect of the MDP characteristics is present in both cases.

In summary, this part of the thesis advances current understanding of what problem properties affect the difficulty of learning with incremental value-based algorithms and contributes new ideas as to how MDP characteristics can be defined precisely and used to improve the efficiency of learning.

Attribute-Based Exploration Strategy. Efficient exploration is one of the key factors in the success of reinforcement learning for practical applications. Considerable research in the current literature is devoted to developing exploration methods that direct the agent's experience gathering effort in a way that can speed up learning or optimize the agent's performance during learning.

We presented a new exploration strategy, developed based on the insights gained through the study of MDP characteristics. Our strategy directly uses measurements of two of the proposed MDP attributes, the State Transition Entropy and the Controllability, in order to improve the efficiency of exploration. It facilitates a homogeneous exploration of the state space, helps to provide sufficient sampling for actions with a potentially high variance in their value samples and focuses the agent's effort on states, in which the agent can influence most its environment. Our strategy differs from existing exploration techniques in that it is based exclusively on properties of the environment. Hence, it provides the means of guiding exploration based on information independent of the course of learning or current value estimates. This can be beneficial for reducing the influence of previous (possibly unsuccessful) exploration decisions on the decisions made on subsequent steps.

Our attribute-based approach can be used in combination with a variety of other existing exploration techniques. We provided an implementation and experimental results for combining our strategy with two existing methods, ϵ-greedy and recency-based exploration, belonging to the undirected and directed classes of exploration approaches respectively. We experimentally demonstrated that our method improves performance in both cases.

In summary, this work demonstrated feasibility of using the proposed MDP attributes in practice and the real benefits of doing so. Second, our strategy provides a simple yet effective way to boost the performance of existing exploration techniques, which is of great practical importance for the efficiency of learning. It equips the agent with the means of identifying the regions of the state and action spaces that

require extra exploration effort or that can potentially lead to most performance improvements.

Resource Allocation Method for Sparse Distributed Memories. The use of function approximation with value-based algorithms is essential for the applicability of reinforcement learning to large realistic domains. While this is currently an active area of research, there is still a considerable gap between the theoretical results available and state-of-the-art in practical applications of reinforcement learning.

In this thesis, we made progress on the idea of using Sparse Distributed Memories as a linear local function approximation model for value-function representation in reinforcement learning. The idea of using SDMs in value-based reinforcement learning was proposed several years ago with a conjecture that this model could be used to circumvent the curse of dimensionality in large domains [Sutton and Barto, 1998]. However, no formal study of its theoretical properties in the context of reinforcement learning has been performed, to the best of our knowledge. Equally, some important issues regarding the practical application of SDMs and related models with value-based methods have not been resolved in a satisfactory way in the past.

In this thesis, we discuss properties of SDMs that allow them to satisfy the assumptions of several existing theoretical results, which provide convergence guarantees for some approximate reinforcement learning methods [Tsitsiklis and Van Roy, 1996; 1997; Gordon, 1995; Szepesvári and Smart, 2004; Lagoudakis and Parr, 2003b]. We discuss how the SDM model can be used with these methods as well as with other popular reinforcement learning algorithms for which convergence properties are still not known, but which are nevertheless successfully applied in practice. The quality of the solutions obtained by all these techniques depends heavily on a good choice of the function approximator's structure. In this thesis, we developed a new approach for automatically choosing certain structural parameters of SDMs (namely, the addresses of the memory locations) based on training data. Related methods for structuring function approximators were developed for supervised learning in the past, and some of them have been used (with varying success) in reinforcement learning. We analyzed

the advantages and drawbacks of these methods in the context of on-line value-based reinforcement learning. Our approach succeeded at avoiding some of the difficulties encountered by other related methods. We empirically demonstrated that our approach provides good performance and is very efficient computationally.

In summary, this work advances state-of-the-art in the practical applicability of SDM function approximators in value-based reinforcement learning. The new technique for automatically choosing the structure of SDMs makes the use of this approximator much easier and opens up very interesting avenus for thoretical work, as discussed below.

6.2. Future Work

As already discussed in chapters 3, 4 and 5, this thesis opens new research avenues for the future, both in the line of research concerning MDP characteristics and that of value function approximation with SDMs. We highlight here a few main directions that would be interesting to pursue.

We investigated in detail the effect of two of the proposed MDP attributes on on-line value-based reinforcement learning, focusing on the attributes that we considered the most informative. In the future, we plan to conduct studies concerning the other attributes discussed in this thesis in order to verify our hypotheses about the effect of the corresponding MDP properties on learning. We would like to evaluate the practical benefits of using the information provided by these attributes for improving efficiency of learning, similarly to what we did with the exploration strategy based on State Transition Entropy and Controllability. We plan to work on using the MDP attributes for tuning other parameters of reinforcement learning algorithms, such as the eligibility trace decay parameter and the learning rate parameter.

We would like to further develop our attribute-based exploration strategy by incorporating other MDP attributes presented in this thesis. We also plan to work on improving the ways of combining several attributes and other exploration bonuses together.

The work related to the Sparse Distributed Memories in value-based reinforcement learning has many directions for future research. First, we plan to extend the approach presented in this thesis so that the activation radii of the memory locations are selected automatically along with the locations' addresses. This will fully automate the process of structure selection for SDMs during on-line learning. One of the ideas that can be explored is that of the progressive refinement of the memory resolution, as discussed in Chapter 5. We also plan to study and improve on existing heuristics developed for this purpose in the past, e.g., methods that attempt to assess the target function's topology [Munos and Moore, 2001], and ideas of avoiding parameter update interference from neural networks [Weaver *et al.*, 1998]. We are also currently investigating the theoretical properties of several reinforcement learning algorithms using SDMs. For instance, by analyzing the properties of SDMs obtained by various resource allocation methods based on sampling from the underlying MDP, we hope to obtain a model-based approximate method that has provable convergence based on the results in [Tsitsiklis and Van Roy, 1996], as well as a concrete way of configuring the function approximator's structure for the MDP at hand. With respect to model-free algorithms, one could study the SDM model and different dynamic allocation approaches in the context of the convergent averaging learning algorithm presented in [Szepesvári and Smart, 2004].

REFERENCES

[Albus, 1981] J.S. Albus. *Brain, Behaviour and Robotics.* Byte Books, 1981.

[Anderson, 1993] C. Anderson. Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems*, pages 81–88, 1993.

[Atkeson and Morimoto, 2002] C. Atkeson and J. Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In *Neural Information Processing Systems Conference*, pages 1611–1618, 2002.

[Atkeson *et al.*, 1997a] C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, pages 11–73, 1997.

[Atkeson *et al.*, 1997b] C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, pages 75–113, 1997.

[Bagnell and Schneider, 2001] J.A. Bagnell and J. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the International Conference on Robotics and Automation*, pages 1615–1620, 2001.

[Baird and Klopf, 1993] L. C. Baird and A. H. Klopf. Reinforcement learning with high-dimensional, continuous actions. Technical Report Technical Report WL-TR-93-1147, Wright-Patterson Air Force Base Ohio: Wright Laboratory. (Available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145, 1993.

[Baird and Moore, 1998] L. Baird and A. Moore. Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 968–974, 1998.

[Baird, 1995] L Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the International Conference on Machine Learning*, pages 30–37, 1995.

[Bartlett and Baxter, 2000] P. L. Bartlett and J. Baxter. Estimation and approximation bounds for gradient-based reinforcement learning. In *Proceedings of the Thirteenth Annual Conference on Computational Learning Theory*, pages 133–141, 2000.

[Barto and Singh, 1990] A.G. Barto and S.P. Singh. On the computational economics of reinforcement learning. In D.S. Touretzky et. al., editor, *Proceedings of the 1990 Summer School on Connectionist Models*, pages 35–44. 1990.

[Baxter and Bartlett, 1999] J. Baxter and P. L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Research School of Information Sciences and Engineering, Australian National University, 1999.

[Bellman and Dreyfus, 1959] R.E. Bellman and S.E. Dreyfus. Functional approximations and dynamic programming. *Mathematical tables and other aids to computation*, 13:247–251, 1959.

[Bellman, 1957] R.E. Bellman. A markov decision process. *Journal of Mathematical Mechanics*, 6:679–684, 1957.

[Bertsekas and Tsitsiklis, 1989] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Neumerical Methods.* Prentice-Hall,Englewwod Cliff, N.J., 1989.

[Bertsekas and Tsitsiklis, 1996] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[Bertsekas, 1995] D.P. Bertsekas. *Dynamic programming and optimal control.* Athena Scientific, Belmont, MA, 1995.

[Blanzieri and Katenkamp, 1996] E. Blanzieri and P. Katenkamp. Learning radial basis function networks on-line. In *Proceedings of the International Conference on Machine Learning*, pages 37–45, 1996.

[Blanzieri, 2003] E. Blanzieri. Theoretical interpretations and applications of radial basis function networks. Technical Report DIT-03-023, Department of Information and Communication Technology, University of Trento, Povo-Trento, Italy, 2003.

[Blondel and Tsitsiklis, 2000] V.D. Blondel and J.N. Tsitsiklis. A survey of computational complexity results in systems and control. *Automatica*, 36:1249–1274, 2000.

[Boothby, 1986] W.M. Boothby. *An introduction to differentiable manifolds and Riemannian geometry.* Academic Press, 1986.

[Boutilier *et al.*, 2000] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[Boyan and Littman, 1994] J. Boyan and M. L. Littman. Packet routing in dynamically changing networks: a reinforcement learning approach. In *Advances in Neural Information Processing Systems*, pages 671–678, 1994.

[Boyan and Moore, 1995] J.A. Boyan and A.W. Moore. Generalization in reinforcement learning: safely approximating the value function. In *Advances in Neural Information Processing Systems*, pages 369–376, 1995.

[Boyan and Moore, 1996] J.A. Boyan and A.W. Moore. Learning evaluation functions for large acyclic domains. In *Proceedings of the International Conference on Machine Learning*, pages 63–70, 1996.

[Boyan, 1999] J.A. Boyan. Least-squares temporal difference learning. In *Proceedings of the International Conference on Machine Learning*, pages 49–56, 1999.

[Bradtke and Barto, 1996] S.J. Bradtke and A.G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:336–57, 1996.

[Brafman and Tennenhotlz, 2002] R.I. Brafman and M. Tennenhotlz. R-max, a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, pages 213–231, 2002.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification And Regression Trees.* The Wadsworth Statistics/Probability Series. Wadsworth and Brooks/Cole Advanced Books and Software, 1984.

[Brown *et al.*, 1998] T.X. Brown, H. Tong, and S Singh. Optimizing admission control while ensuring quality of service in multimedia networks via reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 982–988, 1998.

[Brown, 2001] T. Brown. Switch packet arbitration via queue-learning. In *Advances in Neural Information Processing Systems*, pages 1337–1344, 2001.

[Chapman and Kaelbling, 1991] D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

[Cherkassky and Mulier, 1996] V. Cherkassky and F. Mulier. Comparison of adaptive methods for function estimation from samples. *IEEE Transactions on Neural Networks*, pages 969–984, 1996.

[Chow and Tsitsiklis, 1989] C.S. Chow and J.N. Tsitsiklis. The complexity of dynamic programming. *Journal of Complexity*, 5:466–488, 1989.

[Chow and Tsitsiklis, 1991] C.S. Chow and J.N. Tsitsiklis. An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE Transactions on Automatic Control*, 36:898–914, 1991.

[Chvatal, 1983] V. Chvatal. *Linear Programming.* WH Freeman and Company, New York, NY, 1983.

[Clark and Farley, 1955] W.A. Clark and B.G. Farley. Generalization of pattern recognition in self-organizing system. In *Proceedings of the Western Joint Computer Conference*, pages 86–91, 1955.

[Cohen, 1995] P.R. Cohen. *Empirical Methods for Artificial Intelligence.* Cambridge, MA: The MIT Press, 1995.

[Coraluppi and Markus, 1999] S. Coraluppi and S. Markus. Risk-sensitive and minimax control of discrete-time finite-state markov decision processes. *Automatica*, 35:301–309, 1999.

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

[Davies, 1996] S. Davies. Multidimensional trianguloation and interpolation for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1005–1011, 1996.

[Dayan and Sejnowski, 1996] P. Dayan and T.J. Sejnowski. Exploration bonuses and dual control. *Machine Learning*, pages 5–22, 1996.

[Dayan, 1992] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8:341–362, 1992.

[de Farias and Van Roy, 2000] D. P. de Farias and B Van Roy. On the existence of fixed points for approximate value iteration and temporal-difference learning. *Journal of Optimization Theory and Applications*, 105(3):589–608, 2000.

[de Farias, 2002] D. P. de Farias. *The Linear Programming Approach to Approximate Dynamic Programming: Theory and Application.* PhD thesis, Stanford University, 2002.

[Dean *et al.*, 1995] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, pages 35–74, 1995.

[Dearden *et al.*, 1998] R. Dearden, N. Friedman, and S. Russell. Bayesian Q-learning. In *Proceedings of the Americal Association for Artificial Intelligence Conference*, pages 761–768, 1998.

[Dearden *et al.*, 1999] R. Dearden, N. Friedman, and D. Andre. Model-based Bayesian exploration. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 150–159, 1999.

[Deng and Moore, 1995] K. Deng and A. Moore. Multiresolution instance-based learning. In *Proceedings of the International Joint Conference on Artificial Intellingence*, pages 1233–1239, 1995.

[Dietterich and Wang, 2001] T.G. Dietterich and X. Wang. Batch value function approximation via support vectors. In *Advances in Neural Information Processing Systems*, pages 444–450. Morgan Kaufmann, San Mateo, CA, 2001.

[Engel *et al.*, 2002] Y. Engel, S. Mannor, and R. Meir. Sparse on-line greedy support vector regression. In *Proceedings of the European Conference on Machine Learning*, pages 84–96, Helsinki, Finland, 2002.

[Even-Dar and Mansour, 2003] E. Even-Dar and Y. Mansour. Learning rates for q-learning. *Journal of Machine Learning Research*, 5:1–25, 2003.

[Fahlman and Lebiere, 1989] S.E. Fahlman and C. Lebiere. The cascade correlation learning architecture. In *Advances in Neural Information Processing Systems*, pages 525–532, 1989.

[Ferns *et al.*, 2004] N. Ferns, P. Panangaden, and D. Precup. Metrics for finite decision processes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 162–169, 2004.

[Fidelman and Stone, 2004] P. Fidelman and P. Stone. Learning ball acquisition on a physical robot. In *Proceedings of the International Symposium on Robotics and Automation*, 2004.

[Fiechter, 1994] C.-N. Fiechter. Efficient reinforcement learning. In *Proceedings of the Annual Conference on Computational Learning Theory*, pages 88 – 97, 1994.

[Fiechter, 1997] C.-N. Fiechter. Expected mistake bound model for on-line reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 116–124, 1997.

[Flachs and Flynn, 1992] B. Flachs and M. J. Flynn. Sparse adaptive memory. Technical Report 92-530, Computer Systems Laboratory, Dptm. Of Electrical Engineering and Computer Science, Stanford University, 1992.

[Forbes, 2002] J.R.N. Forbes. *Reinforcement Learning for Autonomous Vehicles.* PhD thesis, Computer Science Department, University of California at Berkeley, 2002.

[Freidman *et al.*, 1977] J.H. Freidman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, pages 209 – 226, 1977.

[Friedman, 1994] J.H. Friedman. An overview of predictive learning and function approximation. In V. Cherkassky, J.H. Friedman, and H. Weehsler, editors, *From statistics to neural networks: Theory and pattern recognition applications*, volume 136 of *NATO ASI Series F.* Springer-Verlag, 1994.

[Fritzke, 1994] B. Fritzke. Supervised learning with growing cell structures. In G. Tesuaro J. D. Cowan and J. Alspector, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, San Mateo, CA, 1994.

[Fritzke, 1997] B. Fritzke. A self-organizing network that can follow non-stationary distributions. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 613–618. Springer, 1997.

[Gaskell, 2003] C. Gaskell. Reinforcement learning under circumstances beyond its control. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation*, 2003.

[Geibel, 2001] P. Geibel. Reinforcement learning with bounded risk. In *Proceedings of the International Conference on Machine Learning*, pages 162–169, 2001.

[Geman *et al.*, 1992] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[Givan *et al.*, 2003] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, pages 163–223, 2003.

[Glaubius and Smart, 2004] R. Glaubius and W.D. Smart. Manifold representations for value-function approximation. Workshop on Learning and Planning in Markov Processes at the Americal Association for Artificial Intelligence Conference, 2004.

[Gordon, 1995] G. J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the International Conference on Machine Learning*, pages 261–268. Morgan Kaufman, 1995.

[Gordon, 2000] G.J. Gordon. Reinforcement learning with function approximation converges to a region. In *Advances in Neural Information Processing Systems*, pages 1040–1046, 2000.

[Gorivensky and Connolly, 1994] D. Gorivensky and T.H. Connolly. Comparison of some neural network and scattered data approximations: The inverse manipulator kinematics example. *Neural Conputation*, 6:521–542, 1994.

[Grimm and Hughes, 1995] C. M. Grimm and J. F. Hughes. Modeling surfaces of arbitrary topology using manifolds. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, volume 29, pages 359 – 368, 1995.

[Guestrin *et al.*, 2004] C. Guestrin, M. Hauskrecht, and B. Kveton. Solving factored MDPs with continuous and discrete variables. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 235–242, 2004.

[Guestrin, 2003] C. Guestrin. *Planning Under Uncertainty in Complex Structured Environments.* PhD thesis, Department of Computer Science, Stanford University, 2003.

[Hauskrecht and Kveton, 2004] M. Hauskrecht and B. Kveton. Linear program approximations for factored continuous-state Markov Decision Processes. In *In Advances in Neural Information Processing Systems*, 2004.

[Hauskrecht *et al.*, 2001] M. Hauskrecht, L. Ortiz, I. Tsochantaridis, and E. Upfal. Efficient methods for computing investment strategies for multi-market commodity trading. *Applied Artificial Intelligence*, 15:429–452, 2001.

[Haykin, 1994] S. Haykin. *Neural Networks: A Comprehensive Foundation.* McMillan, N.Y., 1994.

[Heger, 1994a] H. Heger. Risk and reinforcement learning. Technical report, Universität Bremen, Germany, 1994.

[Heger, 1994b] M. Heger. Consideration of risk in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 105–111, 1994.

[Hely *et al.*, 1997] T. Hely, D. J. Willshaw, and G. M. Hayes. A new approach to Kanerva's Sparse Distributed Memory. *Neural Networks*, 3:791–794, 1997.

[Henery, 1994] R.J. Henery. Methods for comparison. In D. Michie, D.J. Spilegelhalter, and C.C. Taylor, editors, *Machine Learning, Neural and Statistical Classification.* 1994.

[Hogg *et al.*, 1996] T. Hogg, B.A. Huberman, and C.P Williams. Phase transitions and the search problem. *Artificial Intelligence*, pages 1–16, 1996.

[Hoos and Stutzle, 2000] H.H. Hoos and T. Stutzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, pages 421–481, 2000.

[Howell, 2002] D. Howell. *Statistical Methods for Psychology.* Pacific Grove, CA : Duxbury/Thomson Learning, 2002.

[Jaakkola *et al.*, 1994] T. Jaakkola, M.I. Jordan, and S.P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, pages 1185–1201, 1994.

[Jerrum and Sinclair, 1988] M. Jerrum and A. Sinclair. Conductance and the rapid mixing property for markov chains: the approximation of permanent resolved. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 235–244, 1988.

[Jung and Uthmann, 2004] T. Jung and T. Uthmann. Experiments in value function approximation with sparse support vector regression. In *Proceedings of the European Conference on Machine Learning*, pages 180–191, 2004.

[Kaelbling, 1993] L.P. Kaelbling. *Learning in Embedded Systems.* Cambridge, MIT Press, 1993.

[Kakade, 2003] S.M. Kakade. *On the Sample Complexity of Reinforcement Learning.* PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

[Kanerva, 1988] P. Kanerva. *Sparse Distributed Memory.* Cambridge, MA: Bradford Books, 1988.

[Kanerva, 1993] P. Kanerva. Sparse distributed memory and related models. In M.H. Hassoun, editor, *Associative Neural Memories: Theory and Implementation*, pages 50–76. Oxford University Press, N.Y., 1993.

[Karlsson, 1995] R. Karlsson. Evaluation of a fast activation mechanism for the Kanerva SDM memory. Technical Report R95:10, RWCP Neuro SICS Lab, Sweden, 1995.

[Kearns and Singh, 1998] M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the International Conference on Machine Learning*, pages 260–268, 1998.

[Kearns and Singh, 1999] M. Kearns and S. Singh. Finite-sample rates of convergence for Q-learning and indirect methods. In *Proceedings of the Neural Information Processing Systems Conference*, pages 996–1002, 1999.

[Kearns and Singh, 2000] M. Kearns and S. Singh. Bias-variance error bounds for temporal difference updates. In *Proceedings of the Annual Conference on Learning Theory*, pages 142–147, 2000.

[Kirman, 1995] J. Kirman. *Predicting Real-Time Planner Performance by Domain Characterization.* PhD thesis, Brown University, 1995.

[Kohonen, 1984] T. Kohonen. *Self-Organization and Assocaitive Memory.* Berlin: Springer-Verlag, 1984.

[Kondo and Ito, 2002] T. Kondo and K. Ito. A reinforcement learning with adaptive state space recruitment strategy for real autonomous mobile robots. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages CD–ROM #393, Lausanne, Switzerland, 2002.

[Köpf *et al.*, 2000] C. Köpf, Taylor C., and J Keller. Meta-analysis: From data characterization for meta-learning to meta-regression. Proceedings of the PKDD-2000 Workshop on Data Mining, Decision Support, Meta-Learning and ILP, 2000.

[Kretchmar and Anderson, 1997] R. Kretchmar and C. Anderson. Comparison of CMACs and radial basis dunctions for local function approximators in reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 834–837, 1997.

[Kretchmar and Anderson, 1999] R.M. Kretchmar and C. Anderson. Using temporal neighborhoods to adapt function approximators in reinforcement learning. In *Proceedings of the International Work Conference on Artificial Intelligence and Neural Networks*, pages 488–496, 1999.

[Kumar, 1985] P.R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, pages 329–338, 1985.

[Lagoudakis and Littman, 2000] M. Lagoudakis and M.L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 511–518, 2000.

[Lagoudakis and Parr, 2003a] M. G. Lagoudakis and R. Parr. Reinforcement learning as classification: Leveraging modern classifiers. In *Proceedings of the International Conference on Machine Learning*, pages 424–431, Washington DC, 2003.

[Lagoudakis and Parr, 2003b] M.G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

[Lin, 1992] L.J. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, pages 293–321, 1992.

[Linder and Studer, 1999] G. Linder and R. Studer. AST: Support for algorithm selection with a CBR approach. In *Proceedings of the European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 15–18, 1999.

[Littman *et al.*, 1995] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 394–402, Montreal, Québec, Canada, 1995.

[Littman, 1996] M. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Brown University, Providence, RI, 1996.

[MacKay, 2003] D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[Mahadevan, 1996] S. Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1-3):159–195, 1996.

[Marbach *et al.*, 1998] P. Marbach, O. Mihatsch, M. Schulte, and J. N. Tsitsiklis. Reinforcement learning for call admission control and routing in integrated service networks. In *Advances in Neural Information Processing Systems*, pages 922 – 928, 1998.

[Martin, 2002] M. Martin. On-line support vector machine regression. In *Proceedings of the European Conference on Machine Learning*, pages 282–294, Helsinki, Finland, 2002.

[Martinetz and Schulten, 1991] T. Martinetz and K. Schulten. A "neural-gas" network learns topologies. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 397–402, 1991.

[McCallum, 1996] A. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, 1996.

[McCloskey and Cohen, 1989] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: the sequential learning problem. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, pages 109–164. New York: Academic Press, 1989.

[Meeter, 2003] M. Meeter. Control of consolidation in neural networks: avoiding runaway effects. *Connection Science*, 15:45–61, 2003.

[Menache *et al.*, 2004] I. Menache, S. Mannor, and N. Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Submitted to the Annals of Operations Research*, 2004.

[Meuleau and Bourgine, 1999] N. Meuleau and P. Bourgine. Exploration of multi-state environments: Local measures and back-propagation of uncertainty. *Machine Learning*, pages 117–154, 1999.

[Millán *et al.*, 2002] J.D.R. Millán, D. Posenato, and E. Dedieu. Continuous-action Q-learning. *Machine Learning*, 49:247–265, 2002.

[Mitchel, 1997] T. M. Mitchel. *Machine Learning.* McGraw-Hill, 1997.

[Moore and Atkeson, 1993] A.W. Moore and C.G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, pages 103–130, 1993.

[Munos and Moore, 1999] R. Munos and A. Moore. Barycentric interpolator for continuous space and time reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1024 – 1030, 1999.

[Munos and Moore, 2001] R. Munos and A.W. Moore. Variable resolution discretization in optimal control. *Machine learning*, (49):291–323, 2001.

[Munos, 2003] R. Munos. Error bounds for approximate policy iteration. In *Proceedings of the International Conference on Machine Learning*, pages 560–567, 2003.

[Munos, 2004] R. Munos. Error bounds for approximate value iteration. Technical Report RIN 527, Ecole Polytechnique, CMAP, 2004.

[Narendra and Parthasarathy, 1991] K. Narendra and K. Parthasarathy. Gradient methods for the optimization of dynamical systems containing neural networks. *IEEE Transactions on Neural Networks*, 2:252–262, 1991.

[Neuneier and Mihatsch, 1999] R. Neuneier and O. Mihatsch. Risk sensitive reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 11, pages 1031–1037, 1999.

[Ng *et al.*, 2004] A.Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Inverted autonomous helicopter flight via reinforcement learning. In *Proceedings of the International Symposium on Experimental Robotics*, 2004.

[Okabe *et al.*, 1992] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams.* Wiley, 1992.

[Omohundro, 1989] S.M. Omohundro. Bumptrees for efficient function, constraint, and classification learning. Technical Report TR-89-041, International Computer Science Institute, Berkeley, California, 1989.

[Ormoneit and Sen, 2002] D. Ormoneit and S. Sen. A resource-allocating network for function interpolation kernel-based reinforcement learning. *Machine Learning*, 49:161–178, 2002.

[Papadimitriou and Steiglitz, 1982] C.H. Papadimitriou and K Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[Papadimitriou and Tsitsiklis, 1987] C.H. Papadimitriou and J.N. Tsitsiklis. The complexity of markov chain decision processes. *Mathematics of Operations Research*, pages 441–450, 1987.

[Papadimitriou and Tsitsiklis, 1999] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of optimal queueing network control. *Mathematics of Operations Research*, 24(2):293–305, 1999.

[Peng and Williams, 1996] J. Peng and R. J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22, 1996.

[Peng *et al.*, 2002] Y. Peng, P.A. Flach, C. Soares, and P. Brazdil. Improved dataset characterisation for meta-learning. In *Proceedings of the International Conference on Discovery Science*, pages 141–152, 2002.

[Perkins and Precup, 2002] T.J. Perkins and D. Precup. A convergent form of approximate policy iteration. In *Advances in Neural Information Processing Systems*, pages 1595–1602, 2002.

[Platt, 1991] J. Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3:213–225, 1991.

[Precup *et al.*, 2001] D. Precup, R.S. Sutton, and S. Dasgupta. Off-policy temporal-difference learning with function approximation. In *Proceedings of the International Conference on Machine Learning*, pages 417–424, 2001.

[Puterman, 1994] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[Ralaivola and d'Alche Buc, 2001] L. Ralaivola and B. d'Alche Buc. Incremental support vector machine learning: a local approach. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 322–330. Springer, 2001.

[Rao and Fuentes, 1998] R. P. N. Rao and O. Fuentes. Hierarchical learning of navigational behaviors in an autonomous robot using a predictive sparse distributed memory. *Autonomous Robots*, 5:297–316, 1998.

[Ratitch and Precup, 2002] B. Ratitch and D. Precup. Characterizing markov decision processes. In *Proceedings of the European Conference on Machine Learning*, pages 391–404, 2002.

[Ratitch and Precup, 2003a] B. Ratitch and D. Precup. Exploration in RL using MDP characteristics. The 6th European Workshop on Reinforcement Learning, Nancy, France, 2003.

[Ratitch and Precup, 2003b] B. Ratitch and D. Precup. Using MDP characteristics to guide exploration in reinforcement learning. In *Proceedings of the European Conference on Machine Learning*, pages 313–324, 2003.

[Ratitch and Precup, 2004] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *Proceedings of the European Conference on Machine Learning*, 2004.

[Ratitch *et al.*, 2004] B. Ratitch, S. Mahadevan, and D. Precup. Sparse distributed memories as function approximators in value-based reinforcement learning: Case studies. Workshop on Learning and Planning in Markov Processes at the Americal Association for Artificial Intelligence Conference, 2004.

[Ravindran, 1996] B. Ravindran. Solution of delayed reinforcement learning problems having continuous action spaces. Master's thesis, Department of Computer Science and Automation, Indian Institute of Science,, Bangalore, India, 1996.

[Reed, 1993] R. Reed. Prunning algorithms - a survey. *Neural Networks*, 4:3–16, 1993.

[Reynolds, 2000] S. I. Reynolds. Decision boundary partitioning: Variable resolution model-free reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 783–790, 2000.

[Reynolds, 2002] S. I. Reynolds. *Reinforcement Learning with Exploration.* PhD thesis, School of Computer Science, The University of Birmingham, 2002.

[Ribeiro and Szepesvári, 1996] C. Ribeiro and C. Szepesvári. Q-learning combined with spreading: Convergence and results. In *Proceedings of the International Conference on Intelligent and Cognitive Systems, Neural Networks Symposium*, pages 32–36, 1996.

[Riedmiller and Merke, 2001] M. Riedmiller and A. Merke. Karlsruhe Brainstormers - a reinforcement learning approach to robotic soccer II. RoboCup-01: Robot Soccer World Cup V, LNCS, Springer, 2001.

[Riedmiller *et al.*, 2000] M. Riedmiller, A. Merke, D. Meier, A. Hoffmann, A. Sinner, O.Thate, and R. Ehrmann. Karlsruhe Brainstormers - a reinforcement learning approach to robotic soccer. RoboCup-00: Robot Soccer World Cup IV, LNCS, Springer, 2000.

[Rivest and Precup, 2003] F. Rivest and D. Precup. Combining TD-learning with cascade-correlation networks. In *Proceedings of the International Conference on Machine Learning*, pages 632–639, 2003.

[Robbins and Monro, 1951] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, pages 400–407, 1951.

[Robins, 1995] A. V. Robins. Catastrophic forgetting, rehearsal, and pseudorehearsal. *Connection Science*, 7:123–146, 1995.

[Rummery and Niranjan, 1994] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Engineering Department, Cambridhe University, 1994.

[Rust, 1997] J. Rust. Using randomization to break the curse of dimensionality. *Econometrica*, 65(3):487–516, 1997.

[Samejima and Omori, 1999] K. Samejima and T. Omori. Adaptive internal state space construction method for reinforcement learning of real-world agent. *Neural Networks*, 12:1143–1155, 1999.

[Samuel, 1959] A.L. Samuel. Some studies in machine learning using the game of chekers. *IBM Journal on Research and Development*, (3):211–229, 1959. Reprinted in E.A. Feigenbaum and J. Feldman (eds.) Computers and Thought, pp.71-105,1963.

[Samuel, 1967] A. L. Samuel. Some studies in machine learning using the game of chekers. ii-recent progress. *IBM Journal of Research and Development*, 11:601–617, 1967.

[Santamaria *et al.*, 1998] J. C. Santamaria, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6:163–218, 1998.

[Sato and Ishii, 1998] S. Sato and S. Ishii. On-line em algorithm for mixture of local experts. In *Proceedings of the Neural Information Processing Systems Conference*, pages 1397–1401, 1998.

[Sato *et al.*, 1990] M. Sato, K. Abe, and H. Takeda. Learning control of finite markov chains with explicit trade-off between estimation and control. In D.S. Touretzky et.al., editor, *Proceedings of the 1990 Summer School Connectionist Models*, pages 287–300. 1990.

[Schmidhuber, 1991] J.H. Schmidhuber. Adaptive confidence and adaptive curiosity. Technical Report FKI-149-91, Technische Universitat Munchen, 1991.

[Scholkopf, 2000] B. Scholkopf. The kernel trick for distances. In *Advances in Neural Information Processing Systems*, pages 301–307, 2000.

[Schwenker *et al.*, 2001] F. Schwenker, H. A. Kestler, and Palm. G. Three learning phases for radial-basis-function networks. *Neural Networks*, 14:439–458, 2001.

[Singh and Bertsekas, 1997] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*, pages 974–980, 1997.

[Singh and Yee, 1994] S.P. Singh and R.C. Yee. An upper bound on the loss from approximate optimal value functions. *Machine Learning*, pages 227–233, 1994.

[Singh *et al.*, 1995] S.P. Singh, T. Jaakkola, and M.I. Jordan. Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems*, pages 361–368, 1995.

[Singh *et al.*, 2000] S. Singh, T. Jaakkola, M.L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, pages 287–308, 2000.

[Sjodin, 1995] G. Sjodin. Improving the capacity of SDM. Technical Report R95:12, Swedish Inst. Comp. Sci., Stokholm, 1995.

[Smart and Kaelbling, 2000] W. Smart and L.P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the International Conference on Machine Learning*, pages 903–910, 2000.

[Smart, 2002] William D. Smart. *Making Reinforcement Learning Work on Real Robots.* PhD thesis, Department of Computer Science, Brown University, May 2002.

[Sondik, 1971] E. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes.* PhD thesis, Stanford University, 1971.

[Stone and Sutton, 2001] P. Stone and R. Sutton. Scaling reinforcement learning toward robocup soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544, 2001.

[Strösslin and Gerstner, 2003] Thomas Strösslin and Wulfram Gerstner. Reinforcement learning in continuous state and action space. In *Artificial Neural Networks - ICANN 2003*, 2003.

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto. *Reinforcement Learning. An Introduction.* The MIT Press, Cambridge, MA, 1998.

[Sutton and Whitehead, 1993] R. S. Sutton and S. D. Whitehead. Online learning with random representations. In *Proceedings of the International Conference On Machine Learning*, pages 314–321, 1993.

[Sutton *et al.*, 2000] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[Sutton, 1988] R.S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, pages 9–44, 1988.

[Sutton, 1990] R. Sutton. Integrated architecture for learning, planning and reacting based on approximating dynamic programming. In *Proceedings of the International Conference on Machine Learning*, pages 216–224, 1990.

[Szepesvári and Littman, 1999] C. Szepesvári and M. Littman. A unified analysis of value-function-based reinforcement learning algorithms. *Neural Computation*, 11:2017–2059, 1999.

[Szepesvári and Smart, 2004] C. Szepesvári and W. Smart. Interpolation-based Q-learning. In *Proceedings of the International Conference on Machine Learning*, 2004.

[Szepesvári, 1997] C. Szepesvári. The asymptotic convergence-rate of Q-learning. In *Advances in Neural Information Processing Systems*, pages 1064–1070, 1997.

[Szepesvàri, 2001] C. Szepesvàri. Efficient approximate planning in continuous space markovian decision problems. *AI Communications*, 13(3):163 - 176, 2001.

[Tadic, 2001] V. Tadic. On the convergence of temporal-difference learning with linear function approximation. *Machine Learning*, 42:241–267, 2001.

[Taha, 1987] H.A. Taha. *Operations Research: An Introduction.* Macmillan Publishing Company, New York, 1987.

[Tesauro, 1994] G.J. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, pages 215–219, 1994.

[Tham, 1995] C. K. Tham. Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems*, 15:247–274, 1995.

[Thivierge *et al.*, 2003] J.P. Thivierge, F. Rivest, and T.R. Shultz. A dual-phase technique for pruning constructive networks. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 559–564, 2003.

[Thrun and Moller, 1991] S.B. Thrun and K. Moller. Active exploration in dynamic environments. In *Advances in Neural Information Processing Systems*, pages 531–538, 1991.

[Thrun and Schwartz, 1993] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. Proceedings of the Fourth Connectionist Models Summer School, 1993.

[Thrun, 1992] S.B. Thrun. Efficient exploration in reinforcement learning. Technical Report MU-CS-92-102, School of Computer Science, Carnegie Mellon University, 1992.

[Traub *et al.*, 1988] J.F. Traub, G.W. Wasilkowski, and H. Wozniakiwski. *Information-Based Complexity.* Academic Press, 1988.

[Tresp *et al.*, 1992] V. Tresp, J. Hollatz, and S. Ahmad. Network structuring and training using rule-based knowledge. In *Advances in Neural Information Processing Systems*, pages 871–878, 1992.

[Tsitsiklis and Van Roy, 1996] J.N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, pages 59–94, 1996.

[Tsitsiklis and Van Roy, 1997] J Tsitsiklis and B. Van Roy. An analysis of temporal difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[Tsitsiklis, 1994] J.N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, pages 185–202, 1994.

[Tsitsiklis, 2002] J. N. Tsitsiklis. On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, pages 59–72, 2002.

[Uther and Veloso, 1998] W. T. B. Uther and M. M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the American Association for Artificial Intelligence Conference*, pages 769–774, 1998.

[Vignat and Bercher, 1999] C. Vignat and J.-F. Bercher. Un estimateur recursif de l'entropie. In *Proceedings of GRETSI Conference on Signal and Image Processing*, pages 701–704, 1999.

[Vollbrecht, 1999] H. Vollbrecht. *kd*-Q-learning with hierarchic generalization in state space. Technical Report SFB 527, Department of Neural Information Processing, University of Ulm, Germany, 1999.

[Šter and Dobnikar, 2003] B. Šter and A. Dobnikar. Adaptive radial basis decomposition by learning vector quantization. *Neural Processing Letters*, pages 17–27, 2003.

[Wang and Dietterich, 1999] X. Wang and T.G. Dietterich. Efficient value function approximation using regression trees. Proceedings of the International Joint Conference on Artificial Intelligence-99, Workshop on Statistical Machine Learning for Large-Scale Optimization, 1999.

[Wang and Dietterich, 2001] X. Wang and T.G. Dietterich. Stabilizing value function approximation with the BFBP algorithm. In *Advances in Neural Information Processing Systems*, pages 1587–1594, 2001.

[Watkins and Dayan, 1992] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, pages 279–292, 1992.

[Watkins, 1989] C.J.C.H Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[Weaver *et al.*, 1998] S. Weaver, L. Baird, and M. Polycarpou. Preventing unlearning during on-line training of feedforward networks. In *Proceedings of the International Symposium of Intelligent Control*, pages 359–364, 1998.

[Whitehead, 1991] S.D. Whitehead. A study of cooperative mechanisms for faster reinforcement learning. Technical Report 365, University of Rochester, Computer Science Department, 1991.

[Wiatt, 1997] J. Wiatt. *Exploration and Inference in Learning from Reinforcement*. PhD thesis, University of Edingburg, 1997.

[Widrow and Hoff, 1960] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *Western Electronic Show and Convention, Convention record*, volume 4. 1960. Reprentinted in J.A. Anderson and E. Rosenfeld, editors, *Neurocomputing: Foundations and Research*, MIT Press, 19988.

[Wiering and Schmidhuber, 1998] M.A. Wiering and J. Schmidhuber. Efficient model-based exploration. In *Proceedings of the International Conference on Simulation of Adaptive Behavior*, pages 223–228, 1998.

[Wiering *et al.*, 1999] M. Wiering, R. Salustowicz, and J. Schmidhuber. Reinforcement learning soccer teams with incomplete world models. *Journal of Autonomous Robots*, 7(1):77–88, 1999.

[Wiering, 1999] M. Wiering. *Exploration in Efficient Reinforcement Learning.* PhD thesis, Universiteit van Amsterdam, The Netherlands, 1999.

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

[Wyatt, 2001] J. Wyatt. Exploration control in reinforcement learning using optimistic model selection. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 593–600, 2001.

[Xu *et al.*, 1995] L. Xu, M.I. Jordan, and G.E. Hinton. An alternative model for mixtures of experts. In *Advances in Neural Information Processing Systems*, pages 633–640, 1995.

[Zhang and Dietterich, 1995] W. Zhang and T.G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.

APPENDICES

APPENDIX A

A.1. Bellman Equation for the Variance of the Return

We provide here the result from [Munos and Moore, 2001], which proves that the variance of the returns $R^\pi(s)$ for a particular policy π is solution to a Bellman-type equation.

The variance of the returns is defined as follows:

$$\sigma^2(s) = E\left\{[R^\pi(s) - V^\pi(s)]^2\right\} \tag{A.1}$$

where $V^\pi(s)$ is the state-value function, that is the expected return for the policy π.

This variance is solution to the following Bellman equation:

$$\sigma^2(s) = \gamma^2 \sum_{s' \in S} P_{ss'}^{\pi(s)} \sigma^2(s') + e(s) \tag{A.2}$$

where the term $e(s)$ is defined as follows:

$$e(s) = \sum_{s' \in S} P_{ss'}^{\pi(s)} \left[\gamma V^\pi(s') - V^\pi(s) + r_{ss'}^{\pi(s)}\right]^2 \tag{A.3}$$

Proof: The return obtained on a sequence of states starting from the state s satisfies: $R^\pi(s) = r_{ss'}^{\pi(s)} + \gamma R^\pi(s')$. Thus, the variance is:

$$\sigma^2(s) = E\left\{\left[\gamma R^\pi(s') - (V^\pi(s) - r_{ss'}^{\pi(s)})\right]^2\right\} \tag{A.4}$$

From the definition of the state-value function, $V^\pi(s) - r_{ss'}^{\pi(s)} = \gamma E\{R^\pi(s')\}$. Thus,

$$\sigma^2(s) = E\left\{(\gamma R^\pi(s'))^2 - \left(V^\pi(s) - r_{ss'}^{\pi(s)}\right)^2\right\} \tag{A.5}$$

We can decompose this expectation using average for all possible successor states s' weighted by their transition probabilities.

$$\begin{aligned}\sigma^2(s) &= \sum_{s' \in S} P_{ss'}^{\pi(s)} E\left\{(\gamma R^\pi(s'))^2 - \left(V^\pi(s) - r_{ss'}^{\pi(s)}\right)^2\right\} \\ &= \sum_{s' \in S} P_{ss'}^{\pi(s)} E\left\{(\gamma R^\pi(s'))^2 - (\gamma V^\pi(s'))^2\right\} \\ &+ \sum_{s' \in S} P_{ss'}^{\pi(s)} E\left\{(\gamma V^\pi(s'))^2 - \left(V^\pi(s) - r_{ss'}^{\pi(s)}\right)^2\right\}\end{aligned} \tag{A.6}$$

From the Bellman equation $V^\pi(s) = r_{ss'}^{\pi(s)} + \sum_{s' \in S} P_{ss'}^{\pi(s)} \gamma V^\pi(s')$, we deduce that:

$$\sum_{s' \in S} P_{ss'}^{\pi(s)} E\left\{(\gamma V^\pi(s'))^2 - \left(V^\pi(s) - r_{ss'}^{\pi(s)}\right)^2\right\} = e(s) \tag{A.7}$$

with $e(s)$ defined as in Equation (A.3). We also have:

$$E\left\{(\gamma R^\pi(s'))^2 - (\gamma V^\pi(s'))^2\right\} = \gamma^2 E\left\{(R^\pi(s') - V^\pi(s'))^2\right\} = \gamma^2 \sigma^2(s') \tag{A.8}$$

which combined with equations (A.6) and (A.7) gives Equation (A.2).

APPENDIX B

B.1. Resource Allocation for SDMs: Pseudocodes

Algorithm 1 *add_Rule*1(State s, Action a, Target $\bar{Q}$)

Assumes :

Predefined parameters:

N: minimum desired number of active locations

μ^*: similarity threshold, computed by (5.37)

Notation:

$SDM(a)$: locations in memory for action a

M_a: size of the $SDM(a)$

$w(\mathbf{h}^m, a)$: value stored at the location $\mathbf{h}^m$ of the $SDM(a)$

M: maximum memory capacity

Returns :

boolean Added : indicates whether a new location has been added

---

$Added := false$;

$H_s(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(s,a) > 0\}$; { Find a set of active locations for s}

$N' := size[H_s(a)]$ {Number of locations activated by s}

$\mu' := \max_{\mathbf{h}^m \in H_s(a)} \mu^m(s,a)$; { Similarity of s with the closest location}

if $(N' < N)$ and $(\mu' \leq \mu^*)$ and $(M_a < M)$ **then**

  $M_a := M_a + 1$;

  $\mathbf{h}^{M_a} := s$; { Place new location at s}

  $w(\mathbf{h}^{M_a}, a) := \bar{Q}$; {Store target value at the new location}

  $SDM(a) := SDM(a) \cup \mathbf{h}^{M_a}$;

  $Added := true$;

end if

return *Added*;

Algorithm 2 *add_Rule2*(State s, Action a, int $ToAdd$)

Assumes :

Predefined parameters:

N: minimum desired number of active locations

μ^*: similarity threshold, computed by (5.37)

$MaxTry$: a maximum number of times a random position is sampled

Notation:

$SDM(a)$: locations in memory for action a

$w(\mathbf{h}^m, a)$: value stored at the location $\mathbf{h}^m$ of the $SDM(a)$

M_a: size of the $SDM(a)$; M: maximum memory capacity

---

while $((ToAdd > 0)$ and $(M_a < M))$ **do**
  $AddAttempt := MaxTry$
  while $(AddAttempt > 0)$ **do**
    {Sample a potential new location $\mathbf{h} = (h_1, ..., h_n)$}
    for $i = 1$ to n **do**
      $h_i := random(s_i - \beta_i, s_i + \beta_i)$; {Random number in the specified interval}
    end for
    $H_{\mathbf{h}}(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(\mathbf{h}, a) > 0\}$; {Find a set of active locations for $\mathbf{h}$}
    $\mu' := \max_{\mathbf{h}^m \in H_{\mathbf{h}}(a)} \mu^m(\mathbf{h}, a)$; {Similarity of $\mathbf{h}$ with the closest location}
    if $(\mu' > \mu^*)$ { Sampled state $\mathbf{h}$ is too close to the existing locations} **then**
      $AddAttempt := AddAttempt - 1$;{Try to sample again}
    else
      $M_a := M_a + 1$;
      $\mathbf{h}^{M_a} := \mathbf{h}$; {Place a new location at the sampled state}
      {Store the value predicted by the memory in the new location}
      $w(\mathbf{h}^{M_a}, a) := \sum_{\mathbf{h}^m \in H_{\mathbf{h}}(a)} w(\mathbf{h}^m, a) \times \mu^m(\mathbf{h}, a) / \sum_{\mathbf{h}^m \in H_{\mathbf{h}}(a)} \mu^m(\mathbf{h}, a)$;
      $SDM(a) := SDM(a) \cup \mathbf{h}^{M_a}$;
      $AddAttempt := 0$;
    end if
  end while
  $ToAdd := ToAdd - 1$;
end while

Algorithm 3 *reallocate_Rule3*(State s, Action a, Target $\bar{Q}$, int $ToAdd$, boolean $useRule2$)

Assumes :

Predefined parameters:

N: minimum desired number of active locations

μ^*: similarity threshold, computed by (5.37)

$MaxTry$: a maximum number of times a random position is sampled

$ToAdd \geq 1$

Notation:

$SDM(a)$: locations in memory for action a

M_a: size of the $SDM(a)$; M: maximum memory capacity

$w(\mathbf{h}^m, a)$: value stored at the location $\mathbf{h}^m$ of the $SDM(a)$

Invoked when: $M_a + ToAdd > M$

$H_s(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(s,a) > 0\}$; {Find a set of active locations for s}
while ($(M_a + ToAdd) > M$) **do**
  $Removed := false$;
  while ($Removed == false$) **do**
    $k_1 := random\_int(1, M_a)$; {Random integer}
    if ($\mathbf{h}^{k_1} \notin H_s(a)$) **then**
      {Remove this location}
      $H_{\mathbf{h}^{k_1}}(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(\mathbf{h}^{k_1}, a) > 0\}$; {Find a set of active locations}
      $k_2 := \arg \max_{k \neq k_1; \mathbf{h}^k \in H_{\mathbf{h}^{k_1}}(a)} \mu^k(\mathbf{h}^{k_1}, a)$; { Find the closest active location}
      if ($k_2 \neq \oslash$) **then**
        {Create a new location $\mathbf{h} = (h_1, ..., h_n)$ midway between $\mathbf{h}^{k_1}$ and $\mathbf{h}^{k_2}$}
        for $i = 1$ to n **do**
          $h_i := (h_i^{k_1} + h_i^{k_2})/2$;
        end for
        $w(\mathbf{h}, a) := (w(\mathbf{h}^{k_1}, a) + w(\mathbf{h}^{k_2}, a))/2$;
        $SDM(a) := SDM(a) - \{\mathbf{h}^{k_1}, \mathbf{h}^{k_2}\}$;
        $SDM(a) := SDM(a) \cup \mathbf{h}$;
        $Removed := true$;
        $M_a := M_a - 1$;
      end if
    end if
  end while
end while
$Added = add\_Rule1(s, a, \bar{Q})$;
if ($Added == true$) **then**
  $ToAdd := ToAdd - 1$;
end if
if ($useRule2 == true$) **then**
  $add\_Rule2(s, a, ToAdd)$;
end if

Algorithm 4 *SDM_learning_step*(State s, Action a, Target $\bar{Q}$)

Assumes :
Predefined parameters:
N: minimum desired number of active locations
useRule2: indicates whether `Rule 2` is to be used
α: learning step parameter
Notation:
$SDM(a)$: locations in memory for action a
M_a: size of $SDM(a)$; M: maximum memory capacity
$w(\mathbf{h}^m, a)$: value stored at the location $\mathbf{h}^m$ of the $SDM(a)$

$H_s(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(s,a) > 0\}$; {Find a set of active locations for s}
$N' = size[H_s(a)]$; {Number of active locations}
{Add new locations if necessary}
if $(N' < N)$ **then**
  if $(useRule2 == true)$ **then**
    $ToAdd := N - N'$;
  else
    $ToAdd := 1$;
  end if
  if $(M_a + ToAdd) > M$ **then**
    $reallocate\_Rule3(s, a, \bar{Q}, ToAdd, useRule2)$
  else
    $Added := add\_Rule1(s, a, \bar{Q})$;
    if $(Added == true)$ **then**
      $ToAdd := ToAdd - 1$;
    end if
    $add\_Rule2(s, a, ToAdd)$;
  end if
  $H_s(a) := \{\mathbf{h}^m : \mathbf{h}^m \in SDM(a); \mu^m(s,a) > 0\}$; {Update the set of active locations}
end if
{Update the stored values}
$Q := \frac{\sum_{\mathbf{h}^m \in H_s(a)} w(\mathbf{h}^m, a) \times \mu^m(s,a)}{\sum_{\mathbf{h}^m \in H_s(a)} \mu^m(s,a)}$; {Compute current prediction for this training input}
for all $(\mathbf{h}^m \in H_s(a))$ **do**
  $w(\mathbf{h}^m, a) := w(\mathbf{h}^m, a) + \alpha[\bar{Q} - Q]\frac{\mu^m(s,a)}{\sum_{\mathbf{h}^k \in H_s(a)} \mu^k(s,a)}$; {Gradient descent update}
end for