LEARNING TO SOLVE MARKOVIAN DECISION PROCESSES

A Dissertation Presented

by

SATINDER P. SINGH

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1994

Department of Computer Science

LEARNING TO SOLVE MARKOVIAN DECISION PROCESSES

A Dissertation Presented

by

SATINDER P. SINGH

Approved as to style and content by:

Andrew G. Barto, Chair

Richard S. Sutton, Member

Paul E. Utgoff, Member

Roderic Grupen, Member

Donald Fisher, Member

W. Richards Adrion, Department Head
Department of Computer Science

Dedicated to

Mom and Dad,

who gave me

everything

ACKNOWLEDGMENTS

First I would like to thank my advisor, Andy Barto, for his continued guidance, inspiration and support, for sharing many of his insights, and for his constant effort to make me a better writer. I owe much to him for the freedom he gave me to pursue my own interests. Thanks to Rich Sutton for many inspiring conversations, for his insightful comments on my work, and for providing me with opportunities for presenting my work at GTE's Action Meetings. Thanks to Richard Yee for always being ready to listen and help. Numerous lunchtable (and elsewhere) conversations with Richard Yee helped shape my specific ideas and general academic interests.

Robbie Jacobs, Jonathan Bachrach, Vijaykumar Gullapalli and Steve Bradtke influenced my early ideas. Thanks to Rod Grupen and Chris Connolly for their contributions to the material presented in Chapter 8. Michael Jordan and Tommi Jaakkola were instrumental in completing the stochastic approximation connection to RL. I would also like to thank my committee members Rod Grupen, Paul Utgoff, and Don Fisher for their suggestions and advice. Many constructive debates with Brian Pinette, Mike Duff, Bob Crites, Sergio Guzman, Jay Buckingham, Peter Dayan, and Sebastian Thrun helped bring clarity to my ideas. Sushant Patnaik helped in many ways.

A special thanks to my parents whose sacrifices made my education possible and who inspired me to work hard. Their support and encouragement has been invaluable to me. To my brother and sister whose letters and telephone conversations have kept me going. Finally, a big thank you goes to my wife, Roohi, for the shared sacrifices and happiness of these past few years.

My dissertation work was supported by the Air Force Office of Scientific Research, Bolling AFB, under Grants 89-0526 and F49620-93-1-0269 to Andrew Barto and by the National Science Foundation Grant ECS-8912623 to Andrew Barto.

ABSTRACT

LEARNING TO SOLVE MARKOVIAN DECISION PROCESSES

FEBRUARY 1994

SATINDER P. SINGH

B.Tech., INDIAN INSTITUTE OF TECHNOLOGY NEW DELHI
M.S., UNIVERSITY OF MASSACHUSETTS AMHERST
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

This dissertation is about building learning control architectures for agents embedded in finite, stationary, and Markovian environments. Such architectures give embedded agents the ability to improve autonomously the efficiency with which they can achieve goals. Machine learning researchers have developed reinforcement learning (RL) algorithms based on dynamic programming (DP) that use the agent's experience in its environment to improve its decision policy incrementally. This is achieved by adapting an evaluation function in such a way that the decision policy that is "greedy" with respect to it improves with experience. This dissertation focuses on finite, stationary and Markovian environments for two reasons: it allows the development and use of a strong theory of RL, and there are many challenging real-world RL tasks that fall into this category.

This dissertation establishes a novel connection between stochastic approximation theory and RL that provides a uniform framework for understanding all the different RL algorithms that have been proposed to date. It also highlights a dimension that clearly separates all RL research from prior work on DP. Two other theoretical results showing how approximations affect performance in RL provide partial justification for the use of compact function approximators in RL. In addition, a new family of "soft" DP algorithms is presented. These algorithms converge to solutions that are more robust than the solutions found by classical DP algorithms.

Despite all of the theoretical progress, conventional RL architectures scale poorly enough to make them impractical for many real-world problems. This dissertation studies two aspects of the scaling issue: the need to accelerate RL, and the need to build RL architectures that can learn to solve multiple tasks. It presents three RL architectures, CQ-L, H-DYNA, and BB-RL, that accelerate learning by facilitating transfer of training from simple to complex tasks. Each architecture uses a different

method to achieve transfer of training; CQ-L uses the evaluation functions for simple tasks as building blocks to construct the evaluation function for complex tasks, H-DYNA uses the evaluation functions for simple tasks to build an abstract environment model, and BB-RL uses the decision policies found for the simple tasks as the primitive actions for the complex tasks. A mixture of theoretical and empirical results are presented to support the new RL architectures developed in this dissertation.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This dissertation is about building learning control architectures for agents embedded in finite, stationary, and Markovian environments. Such architectures give embedded agents the ability to improve autonomously the efficiency with which they can achieve goals. Machine learning researchers have developed architectures based on reinforcement learning (RL) methods that use the agent's experience in its environment to improve its decision policy incrementally. This dissertation presents a novel theory that provides a uniform framework for understanding and proving convergence for all the different RL algorithms that have been proposed to date. New theoretical results that lead to a better understanding of the strengths and limitations of conventional RL architectures are also developed. In addition, this dissertation presents new RL architectures that extend the range and complexity of applications to which RL algorithms can be applied in practice. These architectures use knowledge acquired in learning simple tasks to accelerate the learning of more complex tasks. A mixture of theoretical and empirical results are provided to validate the proposed architectures.

1.1 Learning and Autonomous Agents

An important long-term objective for artificial intelligence (AI) is to build intelligent agents capable of autonomously achieving goals in complex environments. Recently, some AI researchers have turned attention away from studying isolated aspects of intelligence and towards studying intelligent behavior in *complete* agents embedded in real-world environments (e.g., Agre [1], Brooks [24, 23], and Kaelbling [60]). Much of this research on building embedded agents has followed the approach of hand-coding the agent's behavior (Maes [68]). The success of such agents has depended heavily on their designers' prior knowledge of the dynamics of the interaction between the agent and its intended environment and on the careful choice of the agent's repertoire of behaviors. Such hand-coded agents lack flexibility and robustness.

To be able to deal autonomously with uncertainty due to the incompleteness of the designer's knowledge about complex environments, embedded agents will have to be able to learn. In addition, learning agents can determine solutions to new tasks more efficiently than hardwired agents, because the ability to learn can allow the agent to take advantage of unanticipated regularities in the environment. Although learning also becomes crucial if the environment changes over time, or if the agent's goals change over time, this dissertation will be focussed on the most basic advantage that learning provides to embedded agents: the ability to improve performance over time.

1.2 Why Finite, Stationary, and Markovian Environments?

This dissertation focuses on building learning control architectures for agents that are embedded in environments that have the following characteristics:

Finite: An environment is called finite if the number of different "situations" that the agent can encounter is finite. While many interesting tasks have infinite environments, there are many challenging real-world tasks that do have finite environments, e.g., games, many process control tasks, job scheduling. Besides, many tasks with infinite environments can be modeled as having finite environments by choosing an appropriate level of abstraction. The biggest advantage of focusing on finite environments is that it becomes possible to derive and use a general and uniform theory of learning for embedded agents. This dissertation presents such a theory based on RL that extends and builds upon previous research. Learning architectures developed for finite environments may also extend to infinite environments by the use of sampling techniques and function approximation methods that generalize to unsampled situations appropriately. Some empirical evidence for the last hypothesis is presented in this dissertation.

Stationary: An environment is called stationary if its dynamics are independent of time, i.e., if the outcome of executing an action in a particular situation is not a function of time. Note that stationary does not mean static. Studying stationary environments makes it possible to construct a general and simple theory of RL. At the appropriate level of abstraction a large variety of real-world environments, especially man-made environments, are stationary, or change very slowly over time. It is hoped that if the rate of change in the environment is small then with very minor modifications a RL architecture will be able to keep up with the changes in the environment.

Markovian: An agent's environment is Markovian if the information available to the agent in its current situation makes the future behavior of the environment independent of the past. The Markovian assumption plays a crucial role in all of the theory and the learning architectures presented in this dissertation. For many problem domains, specialists have already identified the minimal information that an agent needs to receive to make its environment Markovian. Researchers building learning control architectures for agents embedded in such environments can use that knowledge. In domains where such knowledge is not available, some researchers have used statistical estimation methods or other machine learning methods to convert non-Markovian problems to Markovian problems. Nevertheless, the Markovian assumption may be more limiting than the previous two assumptions, because RL methods developed for finite, stationary, and Markovian environments degrade more gracefully for small violations of the previous two assumptions relative to small violations of the Markovian assumption.

1.3 Why Reinforcement Learning?

A set of training examples is required if a problem is to be formulated as a supervised learning task (Duda and Hart [38]). For agents embedded in dynamic environments, actions executed in the short term can impact the long term dynamics of the environment. This makes it difficult, and sometimes impossible, to acquire even a small set of examples from the desired behavior of the agent without solving the entire task in the first place. Therefore, problems involving agents embedded in dynamic environments are difficult to formulate as supervised learning tasks.

On the other hand, it is often easy to evaluate the short term performance of the agent to provide an approximate (and perhaps noisy) scalar feedback, called a payoff or reinforcement signal. In the most difficult case, it can at least be determined if the agent succeeded or failed at the task, thereby providing a binary failure/success reinforcement signal. Therefore, tasks involving agents embedded in dynamic environments are naturally formulated as optimization tasks where the optimal behavior is not known in advance, but is defined to be the behavior that maximizes (or minimizes) some function of the agent's behavior over time. Such tasks are called *reinforcement learning tasks*.

The objective function maximized in RL tasks can incorporate many different types of performance criteria, such as minimum time, minimum cost, and minimum jerk. Therefore, a wide variety of tasks of interest in operations research, control theory, robotics, AI, can be formulated as RL tasks. Researchers within these diverse fields have developed a number of different methods under different names for solving RL tasks, e.g., dynamic programming (DP) algorithms (Bellman [16]), classifier systems (Holland *et al.* [51]), and reinforcement learning algorithms (Barto *et al.* [14], Werbos [121]).[1] The different algorithms assume different amounts of domain knowledge and work under different constraints, but they can all solve RL tasks and should perhaps all be called RL methods. However, for the purposes of this dissertation, it will be useful to distinguish between classical DP algorithms developed in the fields of operations research and control theory and the more recent RL algorithms developed in AI.

This dissertation studies two issues that arise when building autonomous agent architectures based on RL methods: the differences between RL algorithms and classical DP algorithms for solving RL tasks, and building RL architectures that can learn complex tasks more efficiently than conventional RL architectures. These issues are introduced briefly in the next two sections.

1.3.1 Reinforcement Learning Algorithms vis-a-vis Classical Dynamic Programming Algorithms

The problem of determining the optimal behavior for agents embedded in finite, stationary, and Markovian environments can be reduced to the problem of solving

[1] Combinatorial optimization methods, such as genetic algorithms (Goldberg [43]), can also be used to solve RL tasks (Grefenstette [45]). However, unlike DP and RL algorithms that use the agent's experience to adapt directly its architecture, genetic algorithms have to evaluate the fitness of a "population" of agents before making any changes to their architectures. Evaluating an agent embedded in a dynamic environment is in general a computationally expensive operation and it seems wasteful to ignore the "local" information acquired through that evaluation. This dissertation will focus on DP and RL algorithms. Nevertheless, it should be noted that no definitive comparison has yet been made between optimization methods based on genetic algorithms and RL or DP algorithms.

a system of nonlinear recursive equations (Ross [87], Bertsekas [17]). Dynamic programming (DP) is a set of iterative methods, developed in the classical literature on control theory and operations research, that are capable of solving such equations (Bellman [16]).[2] Control architectures that use DP algorithms require a model of the environment, either one that is known a priori, or one that is estimated on-line.

One of the main innovations in RL algorithms for solving problems traditionally solved by DP is that they are model-free because they do not *require* a model of the environment. Examples of such model-free RL algorithms are Sutton's [106] temporal differences (TD) algorithm and Watkins [118] Q-learning algorithm. RL algorithms and classical DP algorithms are related methods because they solve the same system of equations, and because RL algorithms estimate the same quantities that are computed by DP algorithms (see Watkins [118], Barto *et al.* [14], and Werbos [123, 124]). More recently, Barto [8] has identified the separate dimensions along which the different RL algorithms have weakened the strong constraints required by classical DP algorithms (see also Sutton [108]).

Despite all the progress in connecting DP and RL algorithms, the following question was unanswered: can TD and Q-learning be derived by the straightforward application of some classical method for solving systems of equations? Recently this author and others (Singh *et al.* [102], Jaakkola *et al.* [53], and Tsitsiklis [115]) have answered that question. In this dissertation it is shown that RL algorithms, such as TD and Q-learning, are instances of asynchronous stochastic approximation methods for solving the recursive system of equations associated with RL tasks. The stochastic approximation framework is also used to delineate the specific contributions made by RL algorithms, and to provide conditions under which RL architectures may be more efficient than architectures that use classical DP algorithms. Simulation studies are used to validate these conditions. The stochastic approximation framework leaves open several theoretical questions and this dissertation identifies and partially addresses some of them.

1.3.2 Learning Multiple Reinforcement Learning Tasks

Despite possessing several attractive properties, as outlined in Sections 1.3 and 1.3.1, RL algorithms have not been applied on-line to solve many complex problems.[3] One of the reasons is the widely held belief that RL algorithms are unacceptably slow for complex tasks. In fact, the common view is that RL algorithms can only be used as *weak* learning algorithms in the AI sense, i.e., they can use little domain knowledge, and hence like all weak learning algorithms are doomed to scale poorly to complex tasks (e.g., Mataric [72]).

[2] Within theoretical computer science, the term DP is applied to a general class of methods for efficiently solving recursive systems of equations for many different kinds of structured optimization problems (e.g., Cormen *et al.* [32]), not just the recursive equations derived for agents controlling external environments. In this dissertation, however, the term DP will be used exclusively to refer to algorithms for solving optimal control problems.

[3] While Tesauro's [112] backgammon player is certainly a complex application, it is not an on-line RL system.

However, the common view is misleading in two respects. The first misconception, as pointed out by Barto [8], is that while RL algorithms are indeed slow, there is little evidence that they are slower than any other method that can be applied with the same generality and under similar constraints. Indeed, there is some evidence that RL algorithms may be faster than their only known competitor that is applicable with the same level of generality, namely classical DP methods (Barto and Singh [12, 11], Moore and Atkeson [79], Gullapalli [48]).

The second misconception is the view that RL algorithms can only be used as weak methods. This misconception was perhaps generated inadvertently by the early developmental work on RL that used as illustrations applications with very little domain knowledge (Barto et.al. [13], Sutton [106]). However, RL architectures can easily incorporate many different kinds of domain knowledge. Indeed, a significant proportion of the current research on RL is about incorporating domain knowledge into RL architectures to alleviate some of their problems (Singh [99], Yee *et al.* [129], Mitchell and Thrun [75], Whitehead [125], Lin [66], Clouse and Utgoff [28]).

Despite the fact that under certain conditions RL algorithms may be the best available methods, conventional RL architectures are slow enough to make them impractical for many real-world problems. While some researchers are looking for faster learning algorithms, and others are investigating ways to improve computing technology in order to solve more complex tasks, this dissertation focuses on a fundamentally different way of tackling the scaling problem. This dissertation studies transfer of training in agents that have to solve multiple structured tasks. It presents RL architectures that use knowledge acquired in learning to solve simple tasks to accelerate the learning of solutions to more complex tasks.

Achieving transfer of training across an arbitrary set of tasks may be difficult, or even impossible. This dissertation explores three different ways of accelerating learning by transfer of training in a class of hierarchically-structured RL tasks. Chapter 6 presents a modular learning architecture that uses solutions for the simple tasks as building blocks for efficiently constructing solutions for more complex tasks. Chapter 7 presents a RL architecture that uses the solutions for the simple tasks to build abstract environment models. The abstract environment models accelerate the learning of solutions for complex tasks because they allow the agent to ignore unnecessary temporal detail. Finally, Chapter 8 presents a RL architecture that uses the solutions to the simple tasks to constrain the solution space for more complex tasks. Both theoretical and empirical support are provided for each of the three new RL architectures.

Transfer of training across tasks must play a crucial role in building autonomous embedded agents for complex real-world applications. Although studying architectures that solve multiple tasks is not a new idea (e.g. Korf [64], Jacobs [55]), achieving transfer of training within the RL framework requires the formulation of, and the solution to, several unique issues. To the best of my knowledge, this dissertation presents the first attempt to study transfer of training across tasks within the RL framework.

1.4 Organization

Chapter 2 presents the Markovian decision task (MDT) framework for formulating RL tasks. It also compares and contrasts the MDT framework for control with the AI state-space search framework for problem solving. The complementary aspects of the research in AI and control theory are emphasized. Chapter 2 concludes by formulating the two mathematical questions of prediction and control for embedded agents that will be addressed in this dissertation.

Chapter 3 presents the abstract framework of iterative relaxation algorithms that is common to both DP and RL. It presents a brief survey of classical DP algorithms for solving RL tasks. A new asynchronous DP algorithm is presented along with a proof of convergence. Chapter 4 presents RL algorithms as stochastic approximation algorithms for solving the problems of prediction and control in finite-state MDTs. Conditions under which RL algorithms may be more efficient than DP algorithms are derived and tested empirically. Several other smaller theoretical questions are identified and partially addressed. Detailed proofs of the theorems presented in Chapters 3 and 4 are presented in Appendices A and B.

Chapter 5 uses the abstract mathematical framework developed in the previous chapters to review prior work on scaling RL algorithms. The different approaches are divided into five abstract classes based on the particular aspect of the scaling problem that is central to each approach. Chapter 6 formulates the class of compositionally-structured MDTs in which complex MDTs are formed by sequencing a number of simpler MDTs. It presents a hierarchical, modular, connectionist architecture that addresses the scaling issue by achieving transfer of training across compositionally-structured MDTs. The modular architecture is tested empirically on a set of discrete navigation tasks, as well as on a set of more difficult, continuous-state, image-based navigation tasks. Theoretical support for the learning architecture is provided in Appendix D. Chapter 7 presents a RL architecture that builds a hierarchy of abstract environment models. It is also tested on compositionally-structured MDTs.

Chapter 8 focuses on a different aspect of the scaling problem for on-line RL architectures; that of maintaining acceptable performance while learning. This chapter is focused on solving motion planning problems. It presents a RL architecture that not only maintains an acceptable level of performance while solving motion planning problems, but is also more efficient than conventional RL architectures. The new architecture's departure from conventional RL architectures for solving motion planning problems is emphasized. Empirical results from two complex, continuous state, motion planning problems are presented. Finally, Chapter 9 presents a summary of the contributions of this dissertation to the theory and practice of building learning control architectures for agents embedded in dynamic environments.

CHAPTER 2

LEARNING FRAMEWORK

This chapter presents a formal framework for formulating tasks involving embedded agents. Embedded agents are being studied in a number of different disciplines, such as AI, robotics, systems engineering, control engineering, and theoretical computer science. This dissertation is focussed on embedded agents that use repeated experience at solving a task to become more skilled at that task. Accordingly, the framework adopted here abstracts the task to that of learning a behavior that approximates optimization of a preset objective functional defined over the space of possible behaviors of the agent. The framework presented here closely follows the work of Barto *et al.* [14, 10] and Sutton [108, 110].

2.1 Controlling Dynamic Environments

In general, the environment in which an agent is embedded can be dynamic, that is, can undergo transformations over time. An assumption and idealization that is often made by dynamical system theorists as well as by AI researchers is that all the information that is of interest about the environment depends only on a finite set of variables that are functions of time $x_1(t), x_2(t), x_3(t), \ldots, x_n(t)$. These variables are often called *state variables* and form the components of the n-dimensional state vector $x(t)$. Mathematical models of the transformation process, i.e., models of the dynamics of the environment, relate the time course of changes in the environment to the state of the environment.

If the agent has no control over the dynamics of the environment, the fundamental problem of interest is that of *prediction*. Solving the prediction problem requires ascertaining an approximation to the sequence $\{x(t)\}$, or more generally an approximation of some given function of $\{x(t)\}$. A more interesting situation arises when the agent can influence the environment's transformation over time. In such a case, the fundamental problem becomes that of *prescription* or *control*, the solution to which prescribes the actions that the agent should execute in order to bring about the desired transformations in the environment. We will return to these two issues of prediction and control throughout this dissertation.

For several decades control engineers have been designing controllers that are able to transform a variety of dynamical systems to a desired goal state or that can track a desired state trajectory over time (e.g., Goodwin and Sin [44]). Such tasks are called regulation and tracking tasks respectively. Similarly researchers in AI have developed problem solving methods for finding sequences of operators that will transform an initial problem (system) state into a desired problem state, often called a 'goal' state.

2.2 Problem Solving and Control

Despite underlying commonalities, the separate development of the theory of problem solving in AI and the theory of regulation and tracking in control engineering led to differences in terminology. For example, the transformation to be applied to the environment is variously called an operator, an action, or a control. The external system to be controlled is called an environment, a process, or a plant. In addition, in control engineering the agent is called a controller. The terms agent, action, and environment will be used in this dissertation.

There are also differences in the algorithms developed by the two communities because of the differing characteristics of the class of environments chosen for study. Traditionally, AI has focussed almost exclusively on deterministic, discrete state and time problems, whereas control theorists have embraced stochasticity and have included continuous state and time problems in their earliest efforts. Consequently, the emphasis in AI has been on *search control* with the aim of reducing the average proportion of states that have to be searched before finding a goal state, while in control theory the emphasis has been on ensuring *stability* by dealing robustly with disturbances and stochasticity.

The focus on deterministic environments within AI has led to the development of planning and heuristic search techniques that develop *open-loop* solutions, or plans. An open-loop solution is a sequence of actions that is executed without reference to the ensuing states of the environment. Any uncertainty or model mismatch can cause plan failure, which is usually handled by replanning.[1] On the other hand, most control design procedures within control theory have been developed to explicitly handle stochasticity and consequently compute a *closed-loop* solution that prescribes actions as a function of the environment's state and possibly of time. Note that forming closed-loop solutions confers no advantage in purely deterministic tasks, because for every start state the sequence of actions executed under the closed-loop solution is an open-loop solution for that start state.

A common feature of most of the early research in AI and in control theory was their focus on off-line design of solutions using environment models. Later, control theorists developed *indirect* and *direct* adaptive control methods for dealing with problems in which a model was unavailable (e.g., Astrom and Wittenmark [3]). Indirect methods estimate a model of the environment incrementally and use the estimated model to design a solution. The same interleaving procedure of system identification and off-line design can be followed in AI problem solving. Direct methods on the other hand directly estimate the parameters of a single, known, parametrized control solution without explicitly estimating a model. As stated before, part of the motivation for RL researchers has been to develop direct methods for solving learning tasks involving embedded agents.

[1] Some of the recent work on planning produces closed-loop plans by performing a cycle of sensing and open-loop planning (e.g., McDermott [74]).

2.3 Learning and Optimal Control

Of more relevance to the theory of learning agents embedded in dynamic environments is a class of control problems studied by optimal control theorists in which the desired state trajectory is not known in advance but is part of the solution to be determined by the control design process.[2] The desired trajectory is one that extremizes some external performance criteria, or some objective function, defined over the space of possible solutions. Such control problems are called *optimal control* problems and have also been studied for several decades. The optimal control perspective provides a suitable framework for learning tasks in which an embedded agent repeatedly tries to solve a task, caching the partial solution and other information garnered in such attempts, and reuses such information in subsequent attempts to improve performance with respect to the performance criteria.

This perspective of optimal control as search is the important common link to the view of problem solving as search developed within the AI community. For some optimal control problems, gradient-based search techniques, such as calculus of variations (e.g., Kirk [62]), can be used for finding the extrema. For other optimal control problems, where non-linearities or stochasticity make gradient-based search difficult, dynamic programming (DP) is the only known general class of algorithms for finding an optimal solution.

The current focus on embedded agents in AI has fortunately come at a time when a confluence of ideas from artificial intelligence, machine learning, robotics, and control engineering is taking place (Werbos [124], Barto [7], Barto *et al.* [10], Sutton *et al.* [108, 110], Dean and Wellman [36]). Part of the motivation behind this current research is to combine the complementary strengths of research on planning and problem solving in AI and of research on DP in optimal control to get the best of both worlds (e.g., Sutton [108], Barto *et al.* [10], and Moore and Atkeson [79]). For example, techniques for dealing with uncertainty and stochasticity developed in control theory are now of interest to AI researchers developing architectures for agents embedded in real-world environments. At the same time, techniques for reducing search in AI problem solving can play a role in making optimal control algorithms more efficient in their exploration of the solution space (Moore [77]).

Another feature of AI problem solving algorithms, e.g., A^* (Hart *et al.* [50], Nilsson [80]), that should be incorporated into optimal control algorithms is that of determining solutions only in parts of the problem space that matter. Algorithms from optimal control theory, such as DP, find complete optimal solutions that prescribe optimal actions to every possible state of the environment. At least in theory, there is no need to find optimal actions for states that are not on the set of optimal paths from the set of possible start states (see Korf [65], and Barto *et al.* [10]).

Following Sutton *et al.* [110] and Barto *et al.* [14], in this dissertation the adaptive optimal control framework is used to formulate tasks faced by autonomous embedded agents. The next section presents the specific formulation of optimal control tasks that is commonly used in machine learning research on building learning control architectures for embedded agents.

---

[2] Regulation and tracking tasks can also be defined using the optimal control framework.

2.4 Markovian Decision Tasks

Figure 2.1 shows a block diagram representation of a general class of tasks faced by embedded agents.[3] It shows an agent interacting with an external environment in a discrete time perception-action cycle. At each time step, the agent perceives its environment, executes an action and receives a payoff in return. Such tasks are called multi-stage decision tasks, or sequential decision tasks, in control theory and operations research. A simplifying assumption often made is that the task is *Markovian* which requires that at each time step the agent's immediate perception returns the *state* of the environment, i.e., provides all the information necessary to make the future perceptions and payoffs independent of the past perceptions. The action executed by the agent and the external disturbances determine the next state of the environment. Such multi-stage decision tasks are called Markovian decision tasks (MDTs).

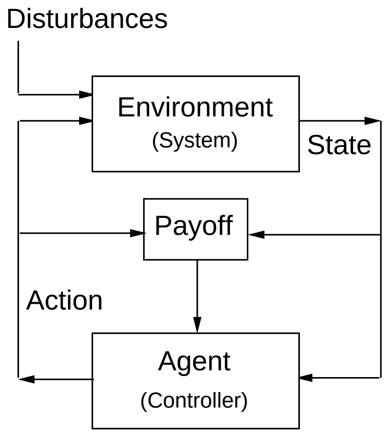

Figure 2.1 Markovian Decision Task. This figure shows a block diagram representation of an MDT. It shows an agent interacting with an external environment in a perception-action cycle. The agent perceives the state of the environment, executes an action, and gets a payoff in return. The action executed by the agent and the external disturbances change the state of the environment.

MDTs are discrete time tasks in which at each of a finite, or countably infinite, number of time steps the agent can choose an action to apply to the environment. Let X be the finite set of environmental states and A be the finite set of actions available to the agent.[4] At time step t, the agent observes the environment's current state, denoted $x_t \in X$, and executes action $a_t \in A$. As a result the environment makes

---

[3] Note that Figure 2.1 is just one possible block diagram representation; other researchers have used more complex block diagrams to capture some of the more subtle intricacies of tasks faced by embedded agents (e.g., Whitehead [125], Kaelbling [60]).

[4] For ease of exposition it is assumed that the same set of actions are available to the agent in each state of the environment. The extension to the case where different sets of actions are available in different states is straightforward.

a transition to state $x_{t+1} \in X$ with probability $P^{a_t}(x_t, x_{t+1})$, and the agent receives an expected payoff[5] $R^{a_t}(x_t) \in \mathbb{R}$. The process $\{x_t\}$ is called a Markovian decision process (MDP) in the operations research literature. The terms MDP and MDT will be used interchangeably throughout this dissertation.

The agent's *task* is to determine a policy for selecting actions that maximizes some cumulative measure of the payoffs received over time. Such a policy is called an *optimal policy.* The number of time steps over which the cumulative payoff is determined is called the *horizon* of the MDT. One commonly used measure for policies is the expected value of the discounted sum of payoffs over the time horizon of the agent as a function of the start state (see Barto *et al.* [14]). [6] This dissertation focuses on agents that have infinite life-times and therefore will have infinite horizons. Fortunately, infinite-horizon MDTs are simpler to solve than finite-horizon MDTs because with an infinite horizon there always exists a policy that is independent of time, called a *stationary* policy, that is optimal (see Ross [87]). Therefore, throughout this dissertation one need only consider stationary policies $\pi : X \rightarrow A$ that assign an action to each state. Mathematically, the measure for policy π as a function of the start state x_0 is

$$V^{\pi}(x_0) \quad = \quad E\left[\sum_{t=0}^{\infty} \gamma^t R^{\pi(x_t)}(x_t)\right], \tag{2.1}$$

where E indicates expected value, and $\pi(x_t)$ is the action prescribed by policy π for state x_t. The discount factor γ, where $0 \leq \gamma < 1$, allows the payoffs distant in time to be weighted less than more immediate payoffs. The function $V^{\pi} : X \rightarrow \mathbb{R}$ is called the *value function* for policy π. The symbol V^{π} is used to denote both the value function and the vector of values of size $|X|$. An optimal control policy, denoted π^*, maximizes the value of every state.

For MDTs that have a horizon of one, called *single-stage* MDTs, the search for an optimal policy can be conducted independently for each state because an optimal action in any state is simply an action that leads to the highest immediate payoff, i.e.,

$$\pi^*(x) = \text{argmax}_{a \in A} R^a(x) \tag{2.2}$$

MDTs with a horizon greater than one, or multi-stage MDTs, face a difficult temporal credit assignment problem (Sutton [105]) because actions executed in the short-term can have long-term consequences on the payoffs received by the agent. Hence, to search for an optimal action in a state it may be necessary to examine the consequences of all action sequences of length equal to the horizon of the MDT.

Most physical environments have infinite state sets and are continuous time systems. However, tasks faced by agents embedded in such environments can frequently

---

[5] All of the theory and the architectures developed in this dissertation extend to the formulation of MDTs in which payoffs are also a function of the next state, and are denoted $r^{a_t}(x_t, x_{t+1})$. In such a case, $R^{a_t}(x_t) \doteq E\{r^{a_t}(x_t, x_{t+1})\}$.

[6] The average payoff per time step received by an agent is another measure for policies that is used in the classical DP literature (Bertsekas [17]), and more recently in the RL literature (Schwartz [94], Singh [100]). This dissertation will only deal with the discounted measure for policies.

be modeled as MDTs by discretizing the state space and choosing actions at some fixed frequency. However, it is important to keep in mind that an MDT is only an abstraction of the physical task. Indeed, it may be possible to represent the same underlying physical task by several different MDTs, simply by varying the resolution of the state space and/or by varying the frequency of choosing actions. The choices made can impact the difficulty of solving the task. In general, the coarser the resolution in space and time, the easier it should be to find a solution. But at the same time better solutions may be found at finer resolutions. This tradeoff is a separate topic of research (e.g., Bertsekas [17]) and will only be partially addressed in this dissertation (see Chapter 8).

The MDT framework for control tasks is a natural extension to stochastic environments of the AI state-space search framework for problem solving tasks. A rich variety of learning tasks from diverse fields such as AI, robotics, control engineering, and operations research can be formulated as MDTs. However, some care has to be taken in applying the MDT framework because it makes the strong assumption that the agent's perception returns the state of the environment, an assumption that may not be satisfied in some real-world tasks with embedded agents (Chapter 9 discusses this is greater detail).

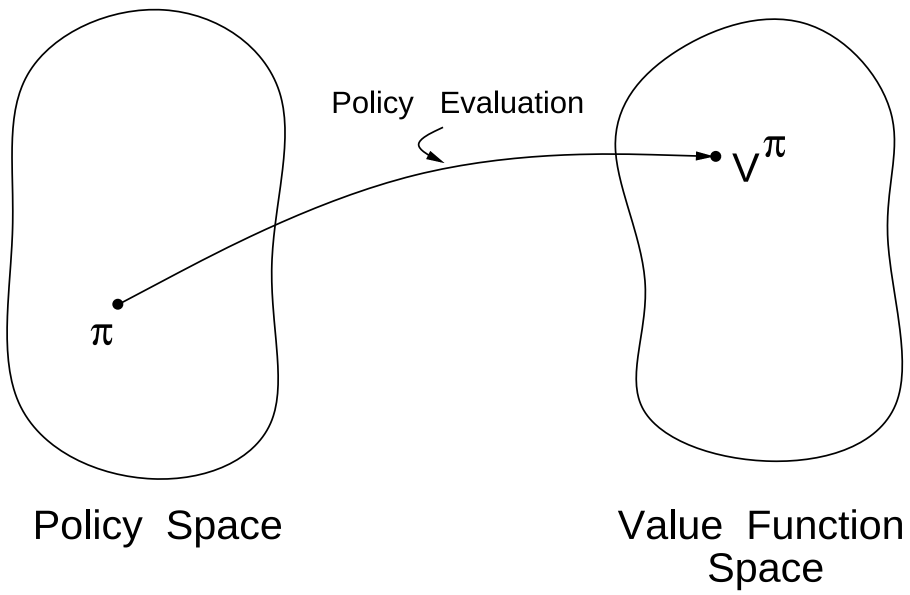

Figure 2.2 The policy evaluation problem. This figure shows the two spaces of interest in solving MDTs: the policy space and the value function space. Evaluating a policy π maps it into a vector of real numbers V^π. Each component of V^π is the infinite-horizon discounted sum of payoffs received by an agent when it follows that policy and starts in the state associated with that component.

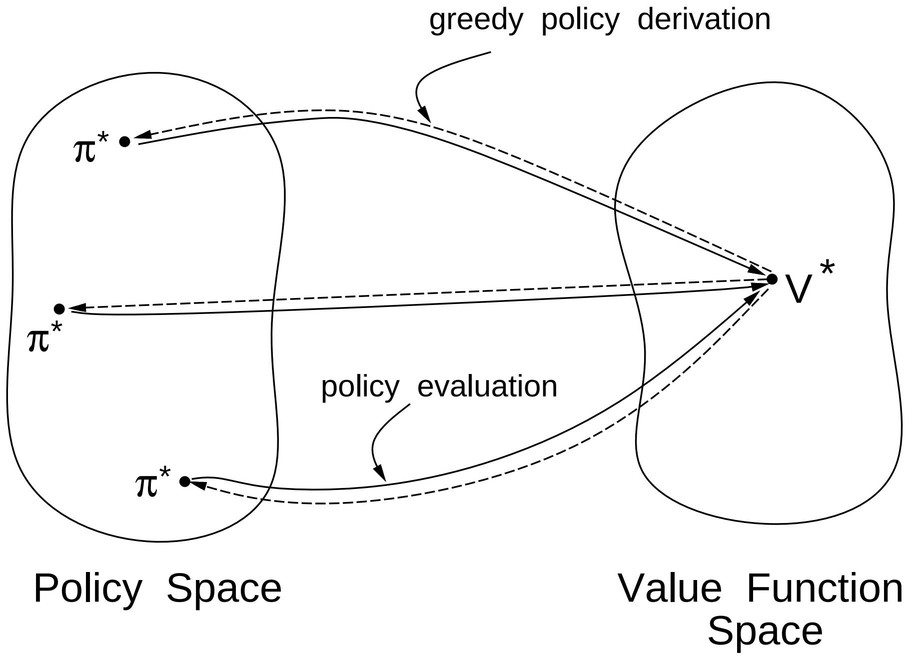

Figure 2.3 The Optimal Control Problem. Solving the optimal control problem requires finding a policy π^* that when evaluated maps onto a value function V^* that is componentwise larger than the value function of any other policy. The optimal policies are the only stationary policies that are greedy with respect to the unique optimal value function V^*.

2.4.1 Prediction and Control

For infinite-horizon MDTs the two fundamental questions of prediction and control can be reduced to that of solving fixed-point equations.

- **Policy Evaluation:** The prediction problem for an MDT, shown in Figure 2.2, is called *policy evaluation* and requires computing the vector V^π for a fixed policy π. Let R^π be the vector of payoffs under policy π and let $[P]^\pi$ be the transition probability matrix under policy π. It can be shown that the following system of linear fixed-point equations of size $|X|$, written in vector form:

$$V = R^\pi + \gamma [P]^\pi V \tag{2.3}$$

  always has a unique solution, and that the solution is V^π, under the assumption that R^π is finite (Ross [87]).

- **Optimal Control:** The control problem for an MDT, shown in Figure 2.3, is that of finding an optimal control policy π^*. The search for an optimal policy has to be conducted only in the set of stationary policies, denoted $\mathcal{P}$, that is of size $|A|^{|X|}$. The value function for an optimal policy π^* is called the optimal value function and is denoted V^*. There may be more than one optimal policy, but the optimal value function is always unique (Ross [87]). It is known that the following system of nonlinear fixed-point equations, $\forall x \in X$:

$$V(x) = \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y)), \tag{2.4}$$

of dimension $|X|$ always has a unique solution, and that the solution is V^*, under the assumption that all the payoff vectors are finite. The set of recurrence relations, $\forall x \in X$,

$$V^*(x) = \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V^*(y)),$$

is known in the DP literature as the Bellman equation for infinite-horizon MDTs (Bellman [16]).

A policy π is *greedy* with respect to any finite value function V if it prescribes to each state an action that maximizes the sum of the immediate payoff and the discounted expected value of the next state as determined by the value function V. Formally, π is greedy with respect to V iff $\forall x \in X$, and $\forall a \in A$:

$$R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y) \geq R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y).$$

Any policy that is greedy with respect to the optimal value function is optimal (see Figure 2.3). Therefore, once the optimal value function is known, an optimal policy for infinite-horizon MDTs can be determined by the following relatively straightforward computation:[7]

$$\pi^*(x) = \text{argmax}_{a \in A} \left[R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V^*(y) \right]. \tag{2.5}$$

In fact, solving the optimal control problem has come to mean solving for V^* with the implicit assumption that π^* is derived by using Equation 2.5.

2.5 Conclusion

The MDT framework offers many attractions for formulating learning tasks faced by embedded agents. It deals naturally with the perception-action cycle of embedded agents, it requires very little prior knowledge about an optimal solution, it can be used for stochastic and non-linear environments, and most importantly it comes with a great deal of theoretical and empirical results developed in the fields of control theory and operations research. Therefore, the MDT framework incorporates many, but not all, of the concerns of AI researchers as their emphasis shifts towards studying

[7]The problem of determining the optimal policy given V^* is a single-stage MDT with $(R^\pi + \gamma[P]^\pi V^*)$ playing the role of the immediate payoff function R^π (cf. Equation 2.2). If the size of the action set A is large, finding the best action in a state can itself become computationally expensive and is the subject of current research (e.g., Gullapalli [48]).

complete agents in real-life environments. This dissertation will deal exclusively with learning tasks that can be formulated as MDTs. In particular, the next two chapters will present theoretical results about the application of DP and RL algorithms to abstract MDTs without reference to any real application. Subsequent chapters will use applications to test new RL architectures that address more practical concerns.

CHAPTER 3

SOLVING MARKOVIAN DECISION TASKS: DYNAMIC PROGRAMMING

This chapter serves multiple purposes: it presents a framework for describing algorithms that solve Markovian decision tasks (MDTs), it uses that framework to survey classical dynamic programming (DP) algorithms, it presents a new asynchronous DP algorithm that is based on policy iteration, and it presents a new family of DP algorithms that find solutions that are more robust than the solutions found by conventional DP algorithms. Convergence proofs are also presented for the new DP algorithms. The framework developed in this chapter is also used in the next chapter to describe reinforcement learning (RL) algorithms and serves as a vehicle for highlighting the similarities and the differences between DP and RL. Section 3.2 is solely a review while Sections 3.3 and 3.4 present new algorithms and results obtained by this author.

3.1 Iterative Relaxation Algorithms

In Chapter 2 it was shown that the prediction and control problems for embedded agents can be reduced to the problem of solving the following systems of fixed-point equations, $\forall x \in X$:

$$\text{Policy Evaluation } (\pi): \quad V(x) = R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y)V(y) \tag{3.1}$$

$$\text{Optimal Control:} \quad V(x) = \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y)). \tag{3.2}$$

This chapter and the next focuses on algorithms that produce sequences of approximations to the solution value function — V^π for Equation 3.1 and V^* for Equation 3.2 — by iterating an "update" equation that takes the following general form:

$$\begin{aligned} \textit{new approximation} \;&=\; \textit{old approximation} + \textit{rate parameter} \,(\, \textit{new estimate} \\ &\qquad\qquad - \textit{old approximation}\,), \end{aligned} \tag{3.3}$$

where the *rate parameter* defines the proportion in which a *new estimate* of the solution value function and the *old approximation* are mixed together to produce a *new approximation*. This new approximation becomes the old approximation at the next iteration of the update equation. Such algorithms are called *iterative relaxation* algorithms.

The sequence of value functions produced by iterating Equation 3.3 is indexed by the iteration number and denoted $\{V_i\}$. Therefore the update equation can be written as follows:

$$V_{i+1} = V_i + \rho_i(U(V_i) - V_i) \tag{3.4}$$

where ρ_i is a relaxation parameter, V_i is the approximation of the solution value function after the $(i-1)^{st}$ iteration, and $U : V_i \rightarrow \mathcal{R}^{|X|}$ is an operator that produces a new estimate of the solution value function by using the approximate value function V_i.[1] In general, the value function is a vector, and at each iteration an arbitrary subset of its components can be updated. Therefore, it is useful to write down each component of the update equation as a function of the state associated with that component of the vector. Let the component of the operator U corresponding to state x be denoted $U_x : V_i \rightarrow \mathcal{R}$. The update equation for state x is

$$V_{i+1}(x) = V_i(x) + \rho_i(x)(U_x(V_i) - V_i(x)) \tag{3.5}$$

where the relaxation parameter is now a function of x.

In this chapter classical DP algorithms are derived as special cases of Equation 3.5. In the next chapter RL algorithms, such as Sutton's temporal differences (TD) and Watkins' Q-learning, will also be derived as special cases of Equation 3.5. The differences among the various iterative algorithms for solving MDTs are: 1) the definition of the operator U, and 2) the order in which the state-update equation is applied to the states of the MDT.

Following Barto [8], it is convenient to represent MDTs, both conceptually and pictorially, as directed stochastic graphs. Figure 3.1 shows the outgoing transitions for an arbitrary state x from a stochastic graph representation of an MDT that in turn is an abstraction of some real-world environment. The nodes represent states and the transitions represent possible outcomes of executing actions in a state. A transition is directed from a *predecessor* state, e.g., x, to a *successor* state, e.g., y (Figure 3.1). Because the problem can be stochastic, a set of outgoing transitions from a state can have the same action label. Each transition has a payoff and a probability attached to it (not shown in Figure 3.1). For any state-action pair the probabilities across all possible transitions sum to one. Figure 3.1 shows that there can be more than one transition between any two states.

In this chapter and the next, stochastic graphs resembling Figure 3.1 will be used to help describe the update equations, and in particular to describe the computation performed by the operator U. A common feature of all the algorithms presented in this dissertation is that the operator U is *local*, in that it produces a new estimate of the value of a state x by accessing information only about states that can be reached from x in one transition.[2]

---

[1] More generally the operator U could itself be a function of the iteration number i. However, for the algorithms discussed in this dissertation, U is assumed fixed for all iterations.

[2] One can define both DP and RL algorithms that use operators that do more than a one-step search and access information about states that are not one-step neighbors, e.g., the multi-step Q-learning of Watkins [118]. Most of the theoretical results stated in this dissertation will also hold for algorithms with multi-step operators.

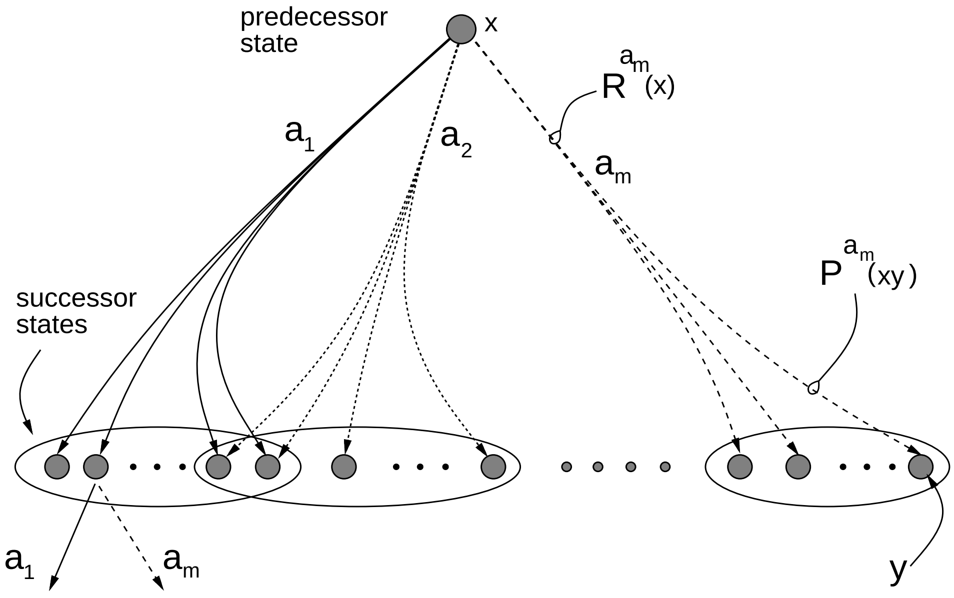

Figure 3.1 Directed Stochastic Graph Representation of an MDT. This figure shows a fragment of an MDT. The nodes represent states and the arcs represent transitions that are labeled with actions.

3.1.1 Terminology

This section presents terminology that will be used throughout this dissertation to describe iterative relaxation algorithms for solving MDTs. The state transition probabilities and the payoff function constitute a model of the environment.

- **Synchronous**: An algorithm is termed synchronous if in every $k|X|$ applications of the state-update equation the value of every state in set X is updated exactly k times. If the values of all the states are updated simultaneously, called Jacobi iteration, the algorithm given in Equation 3.5 can be written in the vector form of Equation 3.4. If the states are updated in some fixed order and the operator U always uses the most recent approximation to the value function, the algorithm is said to perform a Gauss-Sidel iteration.

- **Asynchronous:** Different researchers have used different models of asynchrony in iterative algorithms (e.g., Bertsekas and Tsitsiklis [18]). In this dissertation the term asynchronous is used for algorithms that place no constraints on the order in which the state-update equation is applied, except that in the limit the value of each state will be updated infinitely often. The set of states whose values are updated at iteration i is denoted S_i (as in Barto *et al.* [10]).

- **On-line** An on-line algorithm is one that not only learns a value function but also simultaneously controls a real environment. An on-line algorithm faces the *tradeoff between exploration and exploitation* because it has to choose between executing actions that allow it to improve its estimate of the value function and executing actions that return high payoffs.

- **Off-line** The term off-line implies that the algorithm is using simulated experience with a model of the environment. Off-line algorithms do not face the exploration versus exploitation tradeoff because they design the control solution before applying it to the real environment.

- **Model-Based** A model-based algorithm uses a model of the environment to update the value function, either a model that is given a priori, or one that is estimated on-line using a system identification procedure. Note that the model does not have to be correct, or complete. A model-based algorithm can select states in the model in ways that need not be constrained by the dynamics of the environment, or the actions of the agent in the real environment. Algorithms that estimate a model on-line and do model-based control design on the estimated model are also called *indirect* algorithms (see, e.g., Astrom and Wittenmark [3], and Barto *et al.* [14]).

- **Model-Free** A model-free algorithm does not use a model of the environment and therefore does not have access to the state transition matrices or the payoff function for the different policies. A model-free algorithm is limited to applying the state-update equation to the state of the real environment. Model-free algorithms for learning control are also referred to as *direct* algorithms.

It is not possible to devise algorithms that satisfy an arbitrary selection of the above characteristics; the constraints listed in Table 3.1 apply. For on-line algorithms

Table 3.1 Constraints on Iterative Relaxation Algorithms

Algorithm Type		*Characteristics*
off-line	$\Rightarrow$	model-based
synchronous	$\Rightarrow$	off-line, and therefore model-based
on-line	$\Rightarrow$	asynchronous
model-free	$\Rightarrow$	on-line, and therefore asynchronous

that are model-free it may be difficult to satisfy the conditions required for convergence of asynchronous algorithms because it may be difficult, or even impossible, to ensure that every state is visited infinitely often. In practice, either restrictive assumptions are placed on the nature of the MDT, such as ergodicity, or appropriate constraints are imposed on the control policy followed while learning, such as using probabilistic policies (Sutton [107]). A model-based relaxation algorithm can be either synchronous or asynchronous. An algorithm that does not require a model of the environment can always be applied to a task in which a model is available simply by using the model to simulate the real environment. In general, an algorithm that needs knowledge of the transition probabilities cannot be applied without a model of the environment.

3.2 Dynamic Programming

DP is a collection of algorithms based on Bellman's [16] powerful principle of optimality which states that "an optimal policy has the property that whatever the initial state and action are, the remaining actions must constitute an optimal policy with regard to the state resulting from the first action." The optimal control equation 3.2 can be derived directly from Bellman's principle. Part of the motivation for this section is to develop a systematic "recipe-like" format for describing DP-based algorithms, and the reader will notice its repeated use throughout this dissertation to describe both old and novel algorithms.

3.2.1 Policy Evaluation

As shown by Equation 3.1, evaluating a fixed stationary policy π requires solving a linear system of equations. Define the successive approximation *backup* operator, B^π, in vector form as follows:

$$B^\pi(V) \quad = \quad R^\pi + \gamma[P]^\pi V. \tag{3.6}$$

From Equation 3.1, V^π is the unique solution to the following vector-equation

$$V = B^\pi(V). \tag{3.7}$$

The backup operator for state x is

$$B_x^\pi(V) \quad = \quad R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y)V(y). \tag{3.8}$$

Operator B_x^π is called a backup operator because it "backs up" the value of the successor states (y's) to produce a new estimate of the value of the predecessor state x. Operator B^π requires a model because it requires knowledge of the state transition probabilities.

The computation involved in B_x^π can be explained with the help of Figure 3.2 which only shows the transitions for action $\pi(x) = a_1$ from state x. Operator B_x^π involves adding the immediate payoff to the expected value of the next state that can result from executing action $\pi(x)$ is state x. Operator B^π is a *full* backup operator because computing the expected value of the next state involves accessing information about *all of the possible next states* for state-action pair $(x, \pi(x))$. Operator B^π can be applied synchronously or asynchronously to yield the following algorithms:

(Jacobi) Synchronous successive approximation:

$$V_{i+1} \quad = \quad B^\pi(V_i), \qquad \text{and}$$

Asynchronous successive approximation:

$$V_{i+1}(x) = \begin{cases} B_x^\pi(V_i) & \forall x \in S_i. \\ V_i(x) & \forall x \in (X - S_i). \end{cases} \tag{3.9}$$

Equation 3.9 takes the form of the general iterative relaxation equation 3.5 with $\rho_x = 1$ and $U_x = B_x^\pi$.

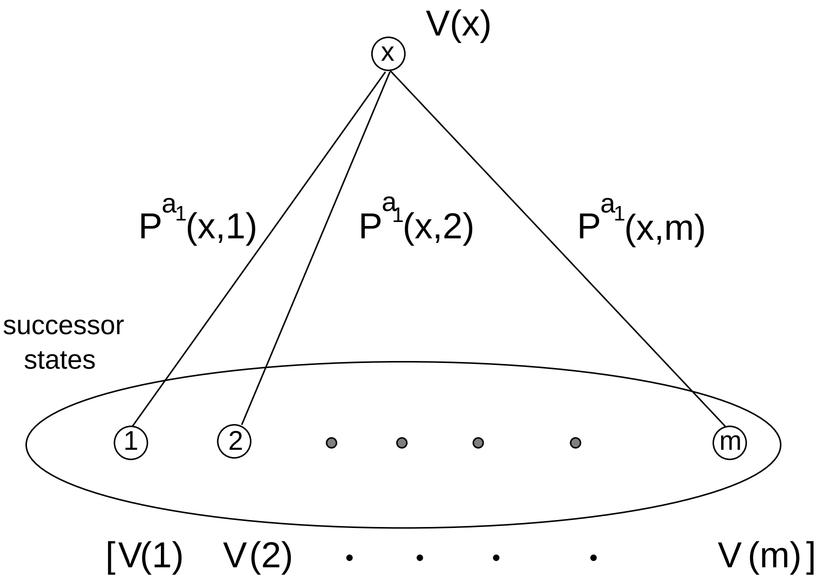

Figure 3.2 Full Policy Evaluation Backup. This figure shows the state transitions on executing action $\pi(x)$ in state x. Doing a full backup requires knowledge of the transition probabilities.

Convergence: If $\gamma < 1$, the operator B^π is a contraction operator because $\forall V \in \mathcal{R}^{|X|}$, $||B^\pi(V) - V^\pi||_\infty \leq \gamma||V - V^\pi||_\infty$, where $||.||_\infty$ is the l_∞ or max norm. Therefore, the synchronous successive approximation algorithm converges to V^π by the application of the contraction mapping theorem (see, e.g., Bertsekas and Tsitsiklis [18]). Convergence can be proven for asynchronous successive approximation by applying the asynchronous convergence theorem of Bertsekas and Tsitsiklis [18].

3.2.2 Optimal Control

Determining the optimal value function requires solving the nonlinear system of equations 3.2. Define the nonlinear value iteration *backup* operator, B, in vector-form as:

$$B(V) \;=\; \max_{\pi \in \mathcal{P}}(R^\pi + \gamma[P]^\pi V), \tag{3.10}$$

where throughout this dissertation the max over a set of vectors is defined to be the vector that results from a componentwise max. From Equation 3.2, the optimal value function V^* is the unique solution to the equation $V = B(V)$. The x-component of B is written as follows:

$$B_x(V) \;=\; \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y)). \tag{3.11}$$

Operator B also requires a model because it assumes knowledge of the state transition probabilities.

The computation involved in operator B can be explained with the help of Figure 3.3 which shows state x and its one-step neighbors. Operator B_x involves

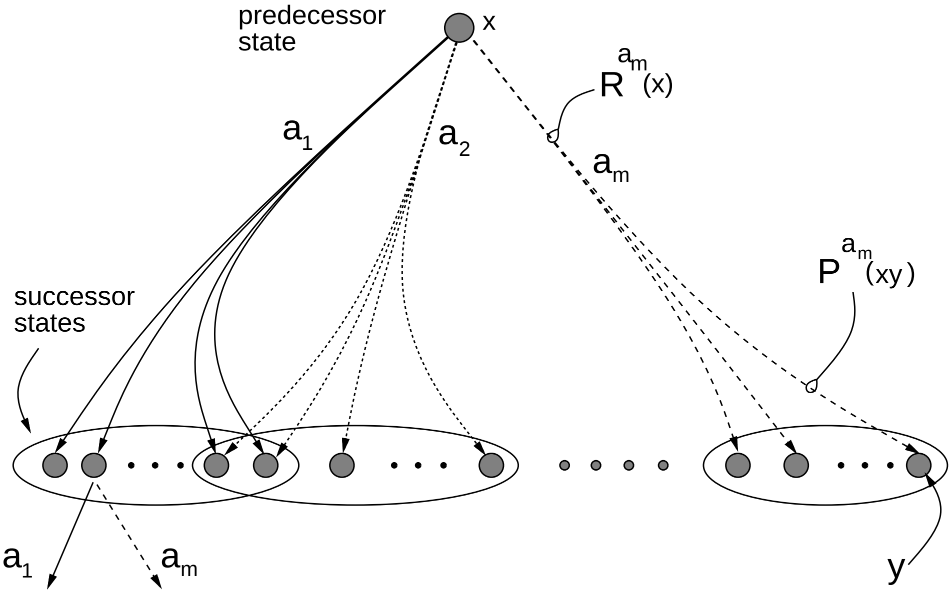

Figure 3.3 Full Value Iteration Backup. This figure shows all the actions in state x. Doing a full backup requires knowledge of the state transition probabilities.

computing the maximum value over all actions of the sum of the immediate payoff and the expected value of the next state for each action. Operator B_x is a *full* backup operator because it involves accessing all of the possible next states for all actions in state x. As in the policy evaluation case, the operator itself can be applied synchronously or asynchronously to yield the following two algorithms:

(Jacobi) Synchronous Value Iteration:

$$V_{i+1} = B(V_i), \qquad \text{and}$$

Asynchronous Value Iteration:

$$V_{i+1}(x) = \begin{cases} B_x(V_i) & \forall x \in S_i, \\ V_i(x) & \forall x \in (X - S_i). \end{cases} \tag{3.12}$$

Equation 3.12 takes the form of the general iterative relaxation equation 3.5 with $\rho_x = 1$ and $U_x = B_x$. The asynchronous value iteration algorithm allows the agent to sample the state space by randomly selecting the state to which the update equation is applied.

Convergence: For $\gamma < 1$, B is a contraction operator because $\forall V \in \mathcal{R}^{|X|}$, $||B(V) - V^*||_\infty \leq \gamma ||V - V^*||_\infty$. Therefore, the synchronous value iteration algorithm can be proven to converge to V^* by the application of the contraction mapping theorem. Convergence can be proven for asynchronous value iteration with $\gamma < 1$ by applying the asynchronous convergence theorem of Bertsekas and Tsitsiklis [18]. The rate of convergence is governed by γ and the second largest eigenvalue, λ_2, of the transition probability matrix for the optimal policy. The smaller the value of γ and the smaller the value of λ_2, the faster the convergence.

Stopping Conditions: For Jacobi value iteration, $V_{i+1} = B(V_i)$, it is possible to define the following error bounds (Bertsekas [17]):

$$
\begin{aligned}
\underline{h_i} &= \frac{\gamma}{1-\gamma}\min_{x\in X}[B_x(V_i) - V_i(x)], \\
\overline{h_i} &= \frac{\gamma}{1-\gamma}\max_{x\in X}[B_x(V_i) - V_i(x)].
\end{aligned}
$$

such that $\forall x \in X,\ B_x(V_i) + \underline{h_i} \leq V^*(x) \leq B_x(V_i) + \overline{h_i}$. The maximum and the minimum change in the value function at the i^{th} iteration bounds the max-norm distance between V_i and V^*. For asynchronous value iteration, these error bounds can be computed using the last visit to each state. Ensuring convergence to V^* may require an infinite number of iterations. In practice, value iteration can be terminated when $(\overline{h_i} - \underline{h_i})$ is small enough.

3.2.3 Discussion

This section presented a review of classical DP algorithms by casting them as iterative relaxation algorithms in a framework that allowed us to highlight the two aspects that differ across the various algorithms: the nature of the backup operator and the order in which it is applied to the states in the environment. The main difference between solving the policy evaluation and the optimal control problems is that in the first case a linear backup operator is employed while in the second a nonlinear backup operator is needed. In both problems the advantage of moving from synchronous to asynchronous is that the asynchronous algorithm can sample in predecessor-state space. The subject of the next section is a novel algorithm for solving the optimal control problem that is more finely asynchronous than the algorithms reviewed in this section because it can sample both in predecessor-state space and in action space.

3.3 A New Asynchronous Policy Iteration Algorithm

An alternative classical DP method for solving the optimal control problem that converges in a *finite number of iterations* is Howard's [52] *policy iteration* method.[3] In policy iteration one computes a sequence of control policies and value functions as follows:

$$
\pi_1 \overset{\mathrm{PE}}{\rightarrow} V^{\pi_1} \overset{\mathrm{greedy}}{\rightarrow} \pi_2 \overset{\mathrm{PE}}{\rightarrow} V^{\pi_2} \ldots \pi_n \overset{\mathrm{PE}}{\rightarrow} V^{\pi_n} \overset{\mathrm{greedy}}{\rightarrow} \pi_{n+1},
$$

were $\overset{\mathrm{PE}}{\rightarrow}$ is the policy evaluation operator that solves Equation 3.1 for the policy on the left-hand side of the operator. Notice that the $\overset{\mathrm{PE}}{\rightarrow}$ operator is not by itself a local operator, even though it can be solved by repeated application of local operators as

[3] Puterman and Brumelle [83] have shown that policy iteration is a Newton-Raphson method for solving for the Bellman equation.

shown in Section 3.2.1. The operator $\overset{\text{greedy}}{\rightarrow}$ computes the policy that is greedy with respect to the value function on the left-hand side.

Stopping Condition: The stopping condition for policy iteration is as follows: if at stage i, $\pi_{i-1} = \pi_i$, then $\pi_{i-1} = \pi_i = \pi^*$, and $V^{\pi_{i-1}} = V^*$, and no further iterations are required. The stopping condition assumes that in computing a new greedy policy ties are broken in favor of retaining the actions of the previous greedy policy.

3.3.1 Modified Policy Iteration

Despite the fact that policy iteration converges in a finite number of iterations, it is not suited to problems with large state spaces because each iteration requires evaluating a policy completely. Puterman and Shin [84] have shown that it is more appropriate to think of the two classical methods of policy iteration and value iteration as two extremes of a continuation of iterative methods, which they called *modified policy iteration* (M-PI). Like policy iteration, k-step M-PI is a synchronous algorithm but the crucial difference is that one evaluates a policy for only k steps before applying the greedy policy derivation operator. With $k = 0$, k-step M-PI becomes value iteration and with $k = \infty$ it becomes policy iteration. While both policy iteration and value iteration converge if the initial value function is finite, Puterman and Shin had to place strong restrictions on the initial value function to prove convergence of synchronous k-step M-PI (see Section 3.3.4).

The motivation behind this section is to derive an *asynchronous* version of k-step M-PI that can sample both in predecessor-state and action spaces, and at the same time converge under a set of initial conditions that are weaker than those required by k-step M-PI. The algorithm presented here is closely related to a set of asynchronous algorithms presented by Williams and Baird [127] that were later shown by Barto [6] to be a form of k-step M-PI.

3.3.2 Asynchronous Update Operators

For ease of exposition, let us denote the one-step backed-up value for state x under action a, given a value function V, by $Q^V(x, a)$. That is,

$$Q^V(x,a) = R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y). \tag{3.13}$$

The asynchronous policy iteration algorithm defined in this section takes the following general form:

$$(V_{k+1}, \pi_{k+1}) \quad = \quad U_k(V_k, \pi_k)$$

where (V_k, π_k) is the k^{th} estimate of (V^*, π^*), and U_k is the asynchronous *update* operator applied at iteration k. A significant difference between the algorithms based on value iteration defined in the previous section and the algorithms based on policy iteration presented in this section is the following: algorithms based on value iteration only estimate and update a value function; the optimal policy is derived after convergence of the value function, while algorithms based on policy iteration explicitly estimate and update a policy in addition to the value function.

Define the following asynchronous update operators (cf. Williams and Baird [127]):

1. A *single-sided policy evaluation* operator $T_x(V_k, \pi_k)$, that takes the current value function V_k and the current policy π_k and does one step of policy evaluation for state x. Formally, if $U_k = T_x$,

$$\begin{aligned} V_{k+1}(y) &= \begin{cases} \max(Q^{V_k}(y, \pi_k(y)),\ V_k(y)) & \text{if } y = x \\ V_k(y) & \text{otherwise, and} \end{cases} \\ \pi_{k+1} &= \pi_k. \end{aligned}$$

   The policy evaluation operator T_x is called single-sided because it never causes the value of a state to decrease.

2. A *single-action policy improvement* operator $L_x^a(V_k, \pi_k)$, that takes the current value function V_k and the current policy π_k and affects them as follows:

$$\begin{aligned} V_{k+1} &= V_k, \text{and} \\ \pi_{k+1}(y) &= \begin{cases} a & \text{if } y = x \text{ and } Q^{V_k}(y, a) > Q^{V_k}(y, \pi_k(y)) \\ \pi_k(y) & \text{otherwise.} \end{cases} \end{aligned}$$

   The policy improvement operator L_x^a is termed single-action because it only considers one action, a, in updating the policy for state x.

3. A *greedy policy improvement* operator $L_x(V_k, \pi_k)$ that corresponds to the sequential application of the operators $\{L_x^{a_1} L_x^{a_2}, \ldots, L_x^{a_{|A|}}\}$. Therefore, $(V_{k+1}, \pi_{k+1}) = L_x(V_k, \pi_k)$ implies that

$$\begin{aligned} V_{k+1} &= V_k, \text{and} \\ \pi_{k+1}(y) &= \begin{cases} \pi_k(y) & \text{if } y \neq x, \\ \text{argmax}_{a \in A} Q^{V_k}(x, a) & \text{otherwise.} \end{cases} \end{aligned}$$

   The operator $L_x(V, \pi)$ updates $\pi(x)$ to be the greedy action with respect to V.

3.3.3 Convergence Results

Initial Conditions: Let $\mathcal{V}_u$ be the set of non-overestimating value functions, $\{V \in \mathcal{R}^{|X|} | V \leq V^*\}$. The analysis of the algorithm presented in this section will be based on the assumption that the initial value-policy pair $(V_0, \pi_0) \in (\mathcal{V}_u \times \mathcal{P})$, where as before $\mathcal{P}$ is the set of stationary policies.

The **Single-Sided Asynchronous Policy Iteration (SS-API)** algorithm is defined as follows:

$$(V_{k+1}, \pi_{k+1}) \;=\; U_{k+1}(V_k, \pi_k),$$

where $U_{k+1} \in \{T_x \mid x \in X\} \bigcup \{L_x^a \mid x \in X, a \in A\}$.

Lemma 1: $\forall k\, V_{k+1} \geq V_k$.

Proof: By the definitions of operators T_x and L_x^a, the value of a state is never decreased.

Lemma 2: If (V_0, π_0) is such that $V_0 \in \mathcal{V}_u$, then $\forall k\, V_k \in \mathcal{V}_u$.

Lemma 2 implies that if the initial value function is non-overestimating, the sequence $\{V_k\}$ will be non-overestimating for all operator sequences $\{U_k\}$.

Proof: Lemma 2 is proved by induction. Given $V_0 \in \mathcal{V}_u$, assume that $\forall k \leq m$, $V_k \in \mathcal{V}_u$. There are only two possibilities for iteration $m+1$:

1. Operator $U_m = L_x^a$ for some arbitrary state-action pair in $X \times A$. Then, $V_{m+1} = V_m \leq V^*$.

2. Operator $U_m = T_x$, for some arbitrary state $x \in X$. Then U_m will only impact $V_{m+1}(x)$.

$$
\begin{aligned}
V_{m+1}(x) &= \max(Q^{V_m}(x, \pi_m(x)), V_m(x)) \\
&= \max([R^{\pi_m(x)}(x) + \gamma \sum_{y \in X} P^{\pi_m(x)}(x,y) V_m(y)], V_m(x)) \\
&\leq \max([R^{\pi_m(x)}(x) + \gamma \sum_{y \in X} P^{\pi_m(x)}(x,y) V^*(y)], V_m(x)) \\
&\leq \max([R^{\pi^*(x)}(x) + \gamma \sum_{y \in X} P^{\pi^*(x)}(x,y) V^*(y)], V_m(x)) \\
&\leq V^*(x).
\end{aligned}
$$

Hence $V_{m+1} \in \mathcal{V}_u$.

Q.E.D.

Theorem 1: Given a starting value-policy pair (V_0, π_0), such that $V_0 \in \mathcal{V}_{\mathbf{u}}$, the SS-API algorithm $(V_{k+1}, \pi_{k+1}) = U_k(V_k, \pi_k)$ converges to (V^*, π^*) under the following conditions:

$A1)$ $\forall x \in X$, T_x appears in $\{U_k\}$ infinitely often, and

$A2)$ $\forall (x, a) \in (X \times A)$, L_x^a appears in $\{U_k\}$ infinitely often.

Proof: It is possible to partition the sequence $\{U_k\}$ into disjoint subsequences of finite length in such a way that each partition itself contains a subsequence for each state that applies the local policy improvement operator for each action followed by the policy evaluation operator. Each such subsequence leads to a contraction in the max-norm of the error in the approximation to V^*. There are an infinity of such subsequences and that fact coupled with Lemma 2 and the contraction mapping theorem constitutes an informal proof. See Appendix A for a formal proof of convergence.

Corollary 1**A:** Given a starting value-policy pair (V_0, π_0), such that $V_0 \in \mathcal{V}_{\mathbf{u}}$, the iterative algorithm $(V_{k+1}, \pi_{k+1}) = U_k(V_k, \pi_k)$ where $U_k \in \{T_x \mid x \in X\} \bigcup \{L_x \mid x \in X\}$, converges to (V^*, π^*) provided for each $x \in X$, T_x and L_x appear infinitely often in the sequence $\{U_k\}$.

Proof: Each L_x can be replaced by a string of operators $L_x^{a_1} L_x^{a_2} \ldots L_x^{a_{|A|}}$. Then Theorem 1 applies.

Q.E.D.

Corollary 1B: Define the operator $T \doteq \{T_{x1}T_{x2}\dots T_{x_{|X|}}\}$ and the operator $L \doteq \{L_{x_1}L_{x_2}\dots L_{x_{|X|}}\}$. Let the sequence $\{U_k\} = (T^m L)^\infty$ consist of infinite repetitions of $m \geq 2$ applications of T operators followed by an L operator. Then, given a starting value-policy pair (V_0, π_0), such that $V_0 \in \mathcal{V}_{\mathbf{u}}$, the iterative algorithm $(V_{k+1}, \pi_{k+1}) = U_k(V_k, \pi_k)$ converges to (V^*, π^*).

Proof: Each T operator can be replaced by a string of operators $T_{x_1}T_{x_2}\dots T_{x_{|X|}}$, and each L operator by the string $L_{x_1}L_{x_2}\dots L_{x_{|X|}}$. Then Corollary 1A applies.

The algorithm presented in Corollary 1B does m steps of greedy policy evaluation followed by one step of single-sided policy improvement. It is virtually identical to m-step Gauss-Sidel M-PI except in that each component of the value function is updated only if it is increased as a result.

3.3.4 Discussion

While SS-API is as easily implemented as the M-PI algorithm of Puterman and Shin, it converges under a larger set of initial value functions and with no constraints on the initial policy. Modified policy iteration of Puterman and Shin requires that (V_0, π_0) be such that $V_0 \in \{V \in \mathcal{R}^{|X|} | \max_\pi(R^\pi + \gamma[P]^\pi V) \geq 0\}$, which is a strict subset of $\mathcal{V}_u$. Similarly the initial condition required by Williams and Baird is that $\forall x \in X$, $Q^{V_0}(x, \pi_0(x)) \geq V_0(x)$, which is again a proper subset of $(\mathcal{V}_u \times \mathcal{P})$.

The SS-API algorithm is more "finely" asynchronous than conventional asynchronous DP algorithms e.g., asynchronous value iteration (AVI), in two ways:

1. SS-API allows arbitrary sequences of policy evaluation and policy improvement operators as long as they satisfy the conditions stated in Theorem 1. AVI, on the other hand, is more coarsely asynchronous because it does not separate the two functions of policy improvement and policy evaluation. In effect AVI iterates a single operator that does greedy policy improvement followed immediately by one step of policy evaluation. Of course, the policy evaluation operator used by AVI is not single-sided.

2. Because AVI uses the greedy policy improvement operator, it has to consider all actions in the state being updated. SS-API on the other hand can sample a single action in each state to do a policy improvement step.

The policy evaluation and the policy improvement operators, T_x and L_x^a, were developed with the knowledge that V_0 would be non-overestimating. However, if V_0 is known to be non-*under*estimating, then it is easy to define analogous operators so that all of the results presented in this section still hold. The only difference for the non-underestimating case is that the max function in the definition of the single-sided policy evaluation operator T_x (see Section 3.3.2) is replaced by the min function.

Theorem 1 shows that if you start with a single-sided error in the estimated value function, then any arbitrary application of the single-sided policy evaluation and the single-action policy improvement operators defined in Section 3.3.2, with the only constraint that each be applied to all states infinitely often, will result in convergence to the optimal value function and an optimal policy.

3.4 Robust Dynamic Programming

In many optimal control problems, the optimal solution may be *brittle* in that it may not leave the controller any room for even the slightest non-stationarity or model-mismatch. For example, the minimum time solution in a navigation problem may take an expensive robot over a narrow ridge where the slightest error in executing the optimal action can lead to disaster. This section presents new DP algorithms that find solutions in which states that have many "good" actions to choose from are preferred over states that have a single good action choice. This robustness is achieved by potentially sacrificing optimality. Robustness can be particularly important if the there is mismatch between the model and the real physical system, or if the real system is non-stationary, or if availability of control actions varies with time.

In searching for the optimal control policy, DP-based algorithms employ the *max* operator that is a "hard" operator because it only considers the consequences of executing the best action in a state and ignores the fact that all the other actions could be disastrous (see Equation 3.12). This section introduces a family of iterative approximation algorithms constructed by replacing the hard max operator in DP-based algorithms by "soft" *generalized means* [49] of order p (e.g., a non-linearly weighted l_p norm). These soft DP algorithms converge to solutions that are more robust than those of classical DP. For each index $p \geq 1$, the corresponding iterative algorithm converges to a unique fixed point, and the approximation gets uniformly better as the index p is increased, converging in the limit ($p \rightarrow \infty$) to the DP solution. The main contribution of this section is the new family of approximation algorithms and their convergence results. The implications for neural network researchers are also discussed.

3.4.1 Some Facts about Generalized Means

This section defines generalized means and lists some of their properties that are useful in the convergence proofs for the soft DP algorithms that follow. Let $A = \{a_1, a_2, \ldots, a_n\}$, and $A' = \{a'_1, a'_2, \ldots, a'_n\}$. Define $A(\max) = \max\{a_1, a_2, \ldots, a_n\}$, and $||A||_\infty = \max\{|a_i|, |a_2|, \ldots, |a_n|\}$. Define $A(p) = [\frac{1}{n}\sum_{i=1}^n (a_i)^p]^{\frac{1}{p}}$, called a generalized mean of order p. The following facts are proved in Hardy *et al.* [49] under the conditions that $a_i, a'_i \in \mathcal{R}^+$ for all i, $1 \leq i \leq n$.

Fact 1. (Convergence) $\lim_{p\rightarrow\infty} A(p) = A(\max)$.

Fact 2. (Differentiability) While $\frac{\partial A(\max)}{\partial a_i}$ is not defined, $\frac{\partial A(p)}{\partial a_i} = \frac{1}{n}[\frac{a_i}{A(p)}]^{p-1}$ for $0 < p < \infty$.

Fact 3. (Uniform Improvement) $0 < p < q \Rightarrow A(p) \leq A(q) \leq A(\max)$; further if $\exists\, i, j$, s.t. $a_i \neq a_j$, then $0 < p < q \Rightarrow A(p) < A(q) < A(\max)$.

Fact 4. (Monotonicity) if $\forall\, i, a_i \leq a'_i$, then $A(p) \leq A'(p)$. In addition, if $\exists\, i$, s.t. $a_i < a'_i$, then $A(p) < A'(p)$.

Fact 5. (Boundedness) If $p \geq 1$, and if $||A - A'||_\infty \leq M$, i.e., the two different sequences of n numbers differ at most by M, then $|A(p) - A'(p)| \leq M$. In addition, if $p > 1$, and $A \neq A'$, then $||A - A'||_\infty \leq M \Rightarrow |A(p) - A'(p)| < M$.

3.4.2 Soft Iterative Dynamic Programming

A family of iterative improvement algorithms, indexed by a scalar parameter p, can be defined as $V_{t+1} = B(p)(V_t)$, where the *backup* operator $B(p) : (\Re^+)^{|X|} \rightarrow (\Re^+)^{|X|}$:

$$B_x(p)(V_t) \stackrel{\text{def}}{=} \left\{ \frac{1}{|A|} \sum_{a \in A} [R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V_t(y)]^p \right\}^{\frac{1}{p}}. \tag{3.14}$$

3.4.2.1 Convergence Results

Fact 7. By the "Convergence" (Fact 1) property of the generalized mean operator, $\lim_{p \rightarrow \infty} B(p) = B(\max) = B$, where B is the value iteration backup operator defined in Section 3.2.2.

Fact 8. For a discrete MDT the finite set of stationary policies form a partial order under the relation $>$: $\pi > \pi' \Rightarrow \forall\, x \in X,\ V^{\pi}(x) > V^{\pi'}(x)$. If $0 \leq \gamma < 1$, and a finite constant $\Delta \in \Re$ is added uniformly to all the payoffs, the partial order of the policies does not change.

The development of the convergence proofs closely follows Bertsekas and Tsitsiklis [18].

Condition 1. $0 \leq \gamma < 1$

Condition 2. $\forall\, x \in X, \forall\, a \in A,\ R^a(x) \geq 0$. This is not a restriction for MDTs with $0 \leq \gamma < 1$, because of Fact 8.

Throughout this section Conditions 1 and 2 are assumed true. Note that condition 2 guarantees that the optimal value function will be non-negative.

Proposition 1. For all $p \geq 1$, the following hold for the operator $B(p)$:

(a) (Monotonicity) $B(p)$ is monotone in the sense that $\forall\, V, V' \in (\Re^+)^{|X|}$:

$$V \leq V' \quad \Rightarrow \quad B(p)(V) \leq B(p)(V'),$$

Proof: Follows trivially from the monotonicity of the generalized mean (Fact 4).

(b) (Contraction Mapping) For all finite $V, V' \in (\Re^+)^{|X|}$,

$$||B(p)(V) - B(p)(V')||_{\infty} \quad \leq \quad \alpha||V - V'||_{\infty},$$

for some $\alpha < 1$.

Proof: Clearly, $\exists M$ such that $0 \leq M < \infty$ and $||V - V'||_\infty \leq M$. Then $\forall\, x \in X$, $-M \leq (V(x) - V'(x)) \leq M$. Substituting $V'(y) + M$ for $V(y)$ in Equation 3.14, we get

$$B_x(p)(V) \;\leq\; \left\{ \frac{1}{|A|} \sum_{a \in A} [R^a(x) + \gamma \sum_{y \in X} P^a(x,y)(V'(y) + M)]^p \right\}^{\frac{1}{p}},$$

and using the fact that P is a stochastic matrix,

$$B_x(p)(V) \;\leq\; \left\{ \frac{1}{|A|} \sum_{a \in A} [R^a(x) + \gamma M + \gamma \sum_{y \in X} P^a(x,y)V'(y)]^p \right\}^{\frac{1}{p}}. \qquad (3.15)$$

By symmetry, it is also true that:

$$B_x(p)(V') \;\leq\; \left\{ \frac{1}{|A|} \sum_{a \in A} [R^a(x) + \gamma M + \gamma \sum_{y \in X} P^a(x,y)V(y)]^p \right\}^{\frac{1}{p}}. \qquad (3.16)$$

Using the boundedness of the generalized mean (Fact 5), Equations 3.15 and 3.16 imply that $\forall\, x \in X$, $B_x(p)(V) \leq B_x(p)(V') + \gamma M$, and that $B_x(p)(V') \leq B_x(p)(V) + \gamma M$.
Q.E.D.
Theorem 2: Under Conditions 1 and 2, if the starting estimate $V_0 \in (\Re^+)^{|X|}$, then $\forall\, p \geq 1$, the iteration $V_{t+1} = B(p)(V_t)$ converges to a unique fixed point V_p^*.
Proof: Using Proposition 1 and the contraction mapping theorem, the iterative algorithm defined by Equation 3.14 converges to a unique fixed point.
Corollary 2A: Let V^* be the optimal value function. Then $\lim_{p \to \infty} V_p^* = V^*$.
Proof: Bertsekas and Tsitsiklis [18] show that the iteration $V_{t+1} = B(V_t)$, where the operator B is as defined in Equation 3.12, converges to the optimal value function V^*. Therefore, $\lim_{p \to \infty} B(p) = B \Rightarrow \lim_{p \to \infty} V_p^* = V^*$.
Theorem 3: $1 \leq p \leq q \Rightarrow V_p^* \leq V_q^* \leq V^*$.
Proof: Consider the iteration by iteration estimates for a parallel implementation of the two algorithms: $V_{p,t+1} = B(p)(V_{p,t})$ and $V_{q,t+1} = B_q(V_{q,t})$, where the successive estimates have been subscripted with the additional symbols p and q in order to distinguish between the two algorithms. Assume that $V_{p,0} = V_{q,0} \leq V_p^* \leq V^*$, and $V_{p,0} = V_{q,0} \leq V_q^* \leq V^*$.

$V_{p,0} = V_{q,0}$ by construction,
$V_{p,1} \leq V_{q,1}$ by Uniform Improvement (Fact 3); applied to each state,
$V_{p,2} \leq V_{q,2}$ by Monotonicity (Fact 4); applied to each state,
. . . .,
. . . .,
$V_{p,t} \leq V_{q,t}$ by Monotonicity (Fact 4); applied to each state.

It is known that $V_{p\infty} = V_p^*$, and $V_{q\infty} = V_q^*$. As shown above, $\exists\, V_0 \in (\Re^+)^{|X|}$, s.t. $\forall t > 0, V_{p,t} \leq V_{q,t} < V^*$. By Theorem 2, the fixed point is independent of V_0. Therefore, $V_p^* \leq V_q^* \leq V^*$ $(\forall\, V_0 \in (\Re^+)^{|X|})$.
Corollary 3A: For any $\epsilon > 0$, $\exists\, p \geq 1$, such that $\forall\, q > p$, $||V_q^* - V^*||_\infty < \epsilon$.
Proof: From Theorems 2 and 3 the sequence of vectors $\{V_p^*\}$ are bounded and converge to V^*. Corollary $3A$ is a property of bounded and convergent sequences.

3.4.3 How good are the approximations in policy space?

The operator sequence $\{B(p)\}$ defines a family of iterative approximation algorithms to the value iteration algorithm. As the index p is increased, the approximation to the optimal value function becomes uniformly better. However, the true measure of interest is not how closely the optimal value function is approximated, but how good is the greedy policy derived from the approximations.

Fact 9. For any given finite action MDT, $\exists\, \delta > 0$, such that $\forall\, \tilde{V} \in (\Re^{+})^{|X|}$, s.t. $||\tilde{V} - V^{*}||_{\infty} < \delta$, any policy that is greedy with respect to $\tilde{V}$ is optimal. See Section 4.5.3 for a proof.

Fact 9 implies that as long as the estimated value function is within δ of the optimal value function, the policy derived from the approximation will be optimal. Define Π_p to be the set of stationary policies that are greedy with respect to V_p^{*}. If $\pi \in \Pi_p$, then $\forall\, x \in X$, and $\forall a \in A$, the immediate payoff for executing action $\pi(x)$ summed with the expected discounted value of the next state is greater than or equal to that for any other action $a \in A$, i.e.,:

$$R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y) V_p^{*}(y) \quad \geq \quad R^{a}(x) + \gamma \sum_{y \in X} P^{a}(x,y) V_p^{*}(y).$$

Let Π^{*} be the set of stationary optimal policies. Define $I_p = \Pi_p \cap \Pi^{*}$.

Theorem 4: For any finite MDT, $\exists\, p$, where $1 \leq p < \infty$, s.t. $\forall\, q > p, \Pi_q \subset \Pi^{*}$.

Proof: Follows directly from Corollary $3A$ and Fact 9.

Theorem 4 implies that in practice there is no need for $p \uparrow \infty$ for the algorithm defined by $B(p)$ to yield optimal policies.

3.4.4 Discussion

DP-based learning algorithms defined using the max operator update the value of a state based on the estimate derived from the "best" action from that state. Algorithms based on the generalized mean, on the other hand, update the value of a state using some non-linearly weighted average of the estimates derived from all the actions available in that state. Thus, the latter will assign higher values to states that have many good actions over states that have just one good action, and conversely will also penalize states for having any bad action at all. Learning algorithms that increase the index p as more information accrues can smoothly interpolate between considering the estimates from all the actions to considering the estimate from the best action alone.

Another advantage of using the operator $B(p)$ instead of B is that unlike B, $B(p)$ is differentiable, which makes it possible to compute the following derivative for neighboring states x and y:

$$\frac{\partial V(x)}{\partial V(y)} \quad = \quad \frac{\gamma}{|A|} \sum_{a \in A} \left\{ \left[\frac{R^{a}(x) + \sum_{y' \in X} P^{a}(x,y') V(y')}{V(x)} \right]^{p-1} \sum_{z \in X} P^{a}(x,z) \frac{\partial V(z)}{\partial V(y)} \right\}.$$

Note that one can use the *chain rule* to compute the above derivatives for states that are not neighbors in the state graph of the MDT in much the same way as the backpropagation (Rumelhart *et al.* [88], Werbos [120]) algorithm for multi-layer connectionist networks. Being able to compute derivatives allows sensitivity analysis and may lead to some new ways of addressing the difficult exploration versus exploitation issue [107] in optimal control tasks. Indeed, the motivation for Rivest's work [85], which inspired the development of the algorithms presented in this section, was to use sensitivity analysis to address the analogous exploration issue in game tree search. Note that derivatives of the values of states with respect to the transition probabilities and the immediate payoffs can also be derived.

As discussed by Rivest [85], other forms of generalized means exist, e.g., for any continuous monotone increasing function, f, one can consider mean values of the form $f^{-1}(\frac{1}{n}\sum_{i=1}^{n} f(a_i))$. In particular, the exponential function can be used to derive an interesting alternative sequence of operators, $B(\lambda) = \frac{\ln(\frac{1}{n}\sum_{i=1}^{n} e^{\lambda a_i})}{\lambda}$, where $\lambda > 0$. As the parameter λ is increased the approximation to the max gets strictly better. Using $B(\lambda)$, a family of alternative iterative fixed point algorithms can be defined: $V_{t+1} = B(\lambda)(V_t)$. An advantage of using $B(\lambda)$ is that it requires less computation than the operator $B(p)$. The operator $B(\lambda)$ is similar in spirit to the "soft-max" function (Bridle [21]) used by Jacobs *et al.* [56] and may provide a probabilistic framework for action selection in DP-based algorithms.

Several researchers are investigating the advantages of combining nonlinear neural networks with traditional adaptive control techniques (e.g., [57, 42]). The algorithms presented in this section have the dual advantages of leading to more robust solutions and of employing a differentiable backup operator. It is hoped that these changes will pave the way for further progress in adapting DP algorithms and nonlinear neural network techniques for *learning* to solve optimal control tasks.

3.5 Conclusion

All of the algorithms presented in this chapter are model-based algorithms because they require a model of the environment to implement the update equations. The asynchronous algorithms based on value iteration move one step towards reducing the need for a model by sampling in predecessor-state space. The asynchronous policy iteration algorithm moved an additional step by sampling in action space as well. The final step towards developing model-free algorithms is to also sample in successor-state space and that is the subject of the next chapter.

CHAPTER 4

SOLVING MARKOVIAN DECISION TASKS: REINFORCEMENT LEARNING

The main contribution of this chapter is in establishing a hitherto unknown connection between stochastic approximation theory and reinforcement learning (RL). The stochastic approximation framework for RL provides a fairly complete theory of asymptotic convergence with lookup-table representations for some well known RL algorithms. A mixture of theory and empirical results are also provided to address partially the following question: which method, RL or dynamic programming (DP), should be applied to a particular problem if given a choice? The stochastic approximation framework leaves open several theoretical questions of great practical interest, and the second half of this chapter identifies and addresses some of them.

4.1 A Brief History of Reinforcement Learning

Early research in RL developed *non-associative* algorithms for solving single-stage Markovian decision tasks (MDTs) in environments with only one state, and had its roots in the work of psychologists working on mathematical learning theory (e.g., Bush and Mosteller [25]), and in the work of learning automata theorists (e.g., Narendra and Thatachar [63]). Later Barto *et al.* [9] developed an associative RL algorithm, they called the A_{RP} algorithm, that solves single-stage MDTs with multiple-state environments. Single-stage MDTs do not involve the temporal credit assignment problem. Therefore algorithms for solving single-stage MDTs are unrelated to DP algorithms (except in the trivial sense). In the early 1980s, in a landmark paper Barto *et al.* [13] described a technique for addressing the temporal credit assignment problem that culminated in Sutton's [106] paper on a class of techniques he called temporal difference (TD) methods (see also Sutton [105]).

In the late 1980s, Watkins [118] observed that the TD algorithm solves the linear policy evaluation problem for multi-stage MDTs (see, also Dayan [33], Barnard [5]). Further, Watkins developed the Q-learning algorithm for solving MDTs and noted the approximate relationship between TD, Q-learning, and DP algorithms (see also Barto *et al.* [15, 10], and Werbos [122]). In this chapter, the connection between RL and MDTs is made precise. But first, a brief detour has to be taken to explain the Robbins-Monro [86] stochastic approximation method for solving systems of equations. Stochastic approximation theory forms the basis for connecting RL and DP.

4.2 Stochastic Approximation for Solving Systems of Equations

Consider the problem of solving a scalar equation $G(V) = 0$, where V is a scalar variable. The classical Newton-Raphson method for finding roots of equations iterates the following recursion:

$$V_{k+1} = V_k - G(V_k)/G'(V_k), \tag{4.1}$$

where V_1 is some initial guess at the root, V_k is the approximation after $k-1$ iterations, and $G'(V_k)$ is the first derivative of the function evaluated at V_k. It is known that the sequence $\{V_k\}$ converges under certain conditions.

Now suppose that the function G is unknown and therefore its derivative cannot be computed. Further suppose that for any V we can observe $Y(V) = G(V) + \epsilon$, where ϵ represents some random error with mean zero and variance $\sigma^2 > 0$. Robbins and Monro [86] suggested using the following recursion

$$V_{k+1} = V_k - \rho_k Y(V_k), \tag{4.2}$$

where $\{\rho_k\}$ are positive constants, such that $\sum \rho_k^2 < \infty$, and $\sum \rho_k = \infty$, and proved convergence in probability for Equation 4.2 to the root of G under the conditions that $G(V) > 0$ if $V > 0$, and $G(V) < 0$ if $V < 0$. The condition that ϵ have zero mean implies that $E\{Y(V_n)\} = G(V_n)$, i.e., $Y(V_n)$ is an unbiased sample of the function G at V_n. It is critical to note that $Y(V_n)$ is not an unbiased estimate of the root of $G(V)$, but only an unbiased estimate of the value of function G at V_n, the current estimate of the root. Equation 4.2 will play an essential role in establishing the connection between RL and DP in Section 4.3.

Following Robbin and Monro's work, several authors extended their results to multi-dimensional equations and derived convergence with probability one under weaker conditions (Blum [19], Dvoretzky [39], Schmetterer [92]). Appendix B presents a theorem by Dvoretzky [39] that is more complex but more closely related to the material presented in the following sections.

4.3 Reinforcement Learning Algorithms

This section uses Equation 4.2 to derive stochastic approximation algorithms to solve the policy evaluation and optimal control problems. In addition, the general framework of iterative relaxation algorithms developed in Chapter 3 is used to highlight the similarities and differences between RL and DP. As in Chapter 3 the aspects of the algorithms that are noteworthy are 1) the definition of the backup operator, and 2) the order in which the states are updated.

4.3.1 Policy Evaluation

The value function for policy π, V^π, is the unique solution to the following policy evaluation equation:

$$V = R^\pi + \gamma[P]^\pi V \tag{4.3}$$

Define the multi-dimensional function $G(V) \doteq V - (R^\pi + \gamma[P]^\pi V)$. The dimension corresponding to state x of function $G(V)$ can be written as follows:

$$G_x(V) = V(x) - (R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y)V(y)).$$

The solution to the policy evaluation problem, V^π, is the unique root of $G(V)$.

Assume that the agent does not have access to a model of the environment and therefore cannot use the matrix $[P]^\pi$ in its calculations. Define $Y_x(V) = V(x) - (R^{\pi(x)}(x) + \gamma V(y))$, where the next state $y \in X$ occurs with probability $P^{\pi(x)}(x,y)$. Define the random matrix $[T]^\pi$ as an $|X| \times |X|$ matrix whose rows are unit vectors, with a 1 in column j of row i with probability $P^{\pi(i)}(i,j)$. Clearly, $E\{[T]^\pi\} = [P]^\pi$, and the vector form of $Y_x(V)$ can be written as follows: $Y(V) = V - (R^\pi + \gamma[T]^\pi V)$. Note that $E\{Y(V)\} = G(V)$.

A stochastic approximation algorithm to obtain the root of $G(V)$ can be derived from Equation 4.2 as follows:
(Jacobi) Synchronous Robbins-Monro Policy Evaluation:

$$\begin{aligned} V_{k+1} &= V_k - \rho_k Y(V_k) \\ &= V_k - \rho_k(V_k - (R^\pi + \gamma[T]^\pi V_k)), \end{aligned}$$

or for state x,

$$V_{k+1}(x) = V_k(x) - \rho_k(x)(V_k(x) - (R^{\pi(x)}(x) + \gamma V_k(y))) \tag{4.4}$$

where $\rho_k(x)$ is a relaxation parameter for state x. To compute $Y_x(V)$, the agent does not need to know the transition probabilities; as depicted in Figure 4.1 it can simply execute action $\pi(x)$ in the real environment and then observe the immediate payoff $R^{\pi(x)}(x)$ as well as the next state y. The value of the next state y can be retrieved from the data structure storing the V values.

The algorithm defined in Equation 4.4 is also an iterative relaxation algorithm of the form defined in Equation 3.5 with a backup operator $\mathcal{B}_x^\pi(V) = R^{\pi(x)}(x) + \gamma V(y)$. The full backup operator of the successive approximation (DP) algorithm: $B_x^\pi(V) = R^{\pi(x)}(x) + \gamma \sum_{y \in X} P^{\pi(x)}(x,y)V(y)$, computes the expected value of the next state to derive a new estimate of V^π. The operator $\mathcal{B}^\pi$, on the other hand, samples one next-state from the set of possible next states and uses the sampled state's value to compute a new estimate of V^π (shown in Figure 4.1). Therefore $\mathcal{B}^\pi$ is called the *sample* backup operator. Note that $\forall\, x \in X$, $E\{\mathcal{B}_x^\pi\}(V) = B_x^\pi(V)$. The RL algorithm represented in Equation 4.4 can be derived directly from the successive approximation (DP) algorithm (Chapter 3) by replacing the full backup operator, B^π, by a random, but unbiased, sample backup operator, $\mathcal{B}^\pi$.

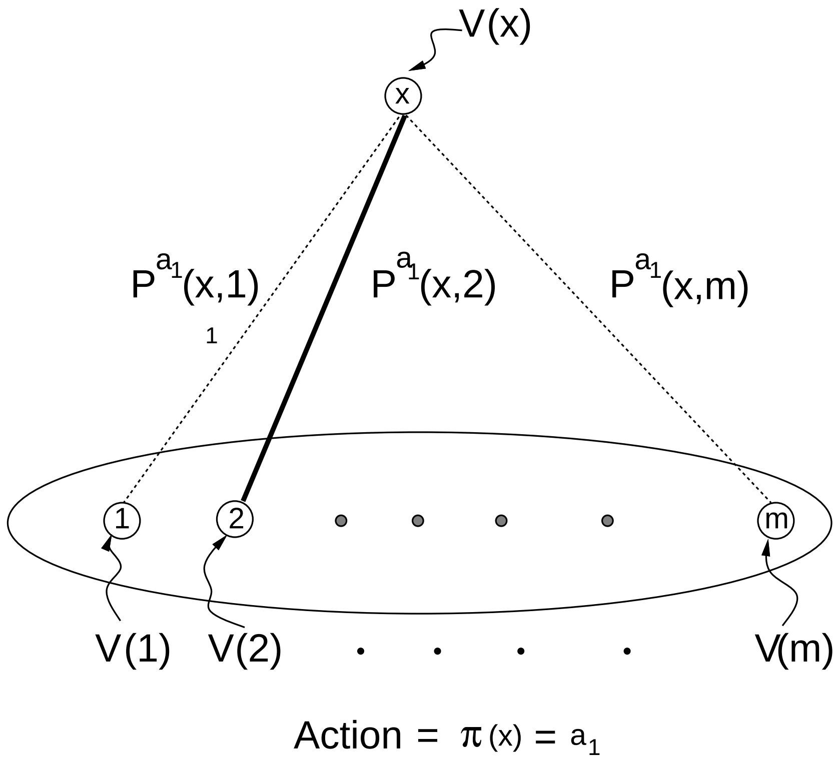

Figure 4.1 Policy Evaluation: Sample Backup. This figure shows the possible transitions for action $\pi(x)$ in state x. The transition marked with a solid line represents the actual transition when action $\pi(x)$ is executed once in state x. Therefore, the only information that a sample backup can consider is the payoff along that transition and the value of state 2.

Equation 4.4 is a Jacobi iteration, and its asynchronous version can be written as follows:
Asynchronous Robbins-Monro Policy Evaluation:

$$\begin{aligned} V_{k+1}(x) &= V_k(x) - \rho_k(V_k(x) - (R^{\pi(x)}(x) + \gamma V_k(y))); \qquad \text{for } x \in S_k \\ V_{k+1}(z) &= V_k(z); \qquad \text{for } z \in (X - S_k). \end{aligned} \tag{4.5}$$

The on-line version of Equation 4.5, i.e., where the sets $S_t = \{x_t\}$ contain only the current state of the environment is identical to the TD(0) algorithm commonly used in RL applications (Sutton [106]). An asymptotic convergence proof for the synchronous algorithm (Equation 4.4) can be easily derived from Dvoretzky's theorem presented in Appendix B. However, more recently, Jaakkola, Jordan and Singh [53] have derived the following theorem for the more general asynchronous case (of which the synchronous algorithm is a special case) by extending Dvoretzky's stochastic approximation results:
Theorem 5: (Jaakkola, Jordan and Singh [53]) For finite MDPs, the algorithm defined by Equation 4.5 converges with probability one to V^π under the following conditions:

1. every state in set X is updated infinitely often,

2. V_0 is finite, and

3. $\forall x \in X$; $\sum_{i=0}^{\infty} \rho_i(x) = \infty$ and $\sum_{i=0}^{\infty} \rho_i^2(x) < \infty$.

Sutton [106] has defined a family of temporal difference algorithms called TD(λ), where $0 \leq \lambda \leq 1$ is a scalar parameter. The TD algorithm discussed above is just one algorithm from that family, specifically the TD(0) algorithm. See Jaakkola *et al.* [53] for a discussion of the connection between stochastic approximation and the general class of TD(λ) algorithms. Also, see Section 5.1.4 in Chapter 5 of this dissertation for further discussion about TD(λ) algorithms. Hereafter, the name TD will be reserved for the TD(0) algorithm.

4.3.2 Optimal Control

The optimal value function, V^*, is the unique solution of the following system of equations: $\forall x$,

$$V(x) = \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y)). \tag{4.6}$$

Define the component corresponding to state x of a multi-dimensional function $G(V)$ as follows:

$$G_x(V) = V(x) - \max_{a \in A}(R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V(y))$$

The solution to the optimal control problem, V^*, is the unique root of the nonlinear function $G(V)$.

Again, as for policy evaluation, assume that the agent does not have access to a model of the environment and therefore cannot use the transition probabilities in its calculations. The attempt to define a stochastic approximation algorithm for solving the optimal control problem by following the procedure used in Section 4.3.1 fails because the function $G(V)$ is nonlinear. Specifically, if $Y_x(V) \doteq V(x) - \max_{a \in A}(R^a(x) + \gamma V(y))$, then $E\{Y_x(V)\} \neq G_x(V)$, because *the expectation and the* max *operators do not commute*. The solution to this impasse lies in a clever trick employed by Watkins in his Q-learning algorithm.

Watkins proposed rewriting Equation 4.6 in the following expanded notation: instead of keeping one value, $V(x)$, for each state $x \in X$, the agent stores $|A|$ values for each state, one for each action possible in that state. Watkins proposed the notation $Q(x,a)$ as the Q-value for state-action pair (x,a). Figure 4.2 shows the expanded representation. The optimal control equations can be written in Q-notation as:

$$Q(x,a) = R^a(x) + \sum_{y \in X} P^a(x,y)(\max_{a' \in A} Q(y,a')); \qquad \forall (x,a) \in (X \times A). \tag{4.7}$$

The optimal Q-values, denoted Q^*, are the unique solution to Equation 4.7, and further $V^*(x) = \max_{a \in A} Q^*(x,a)$.

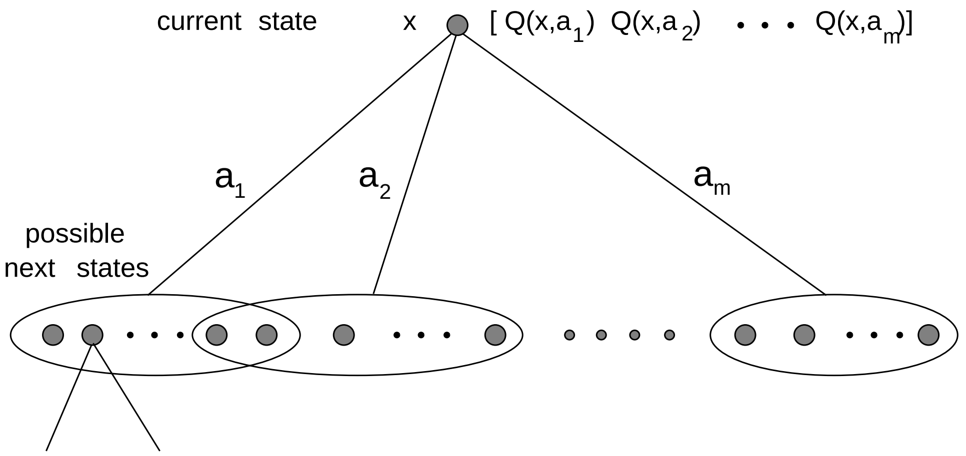

Figure 4.2 Q-values. This figure shows that instead of keeping just one value $V(x)$ for each state x, one keeps a set of Q-values, one for each action in a state.

An iterative relaxation DP algorithm can be derived in terms of the Q-values by defining a new local full backup operator,

$$B_x^a(Q) \quad = \quad R^a(x) + \gamma \sum_{y \in X} P^a(x,y)(\max_{a' \in A} Q(y,a')). \tag{4.8}$$

Operator $B_x^a(Q)$ requires a model of the environment because it involves computing the expected value of the maximum Q-values of all possible next states for state-action pair (x,a). Let $(X,A)_k \in (X \times A)$ be the set of state-action pairs updated at iteration k. An algorithm called Q-value iteration, by analogy to value iteration, can be defined by using the new operator $B(Q)$ as follows:

Asynchronous Q-value Iteration:

$$\begin{aligned} Q_{k+1}(x,a) &= B_x^a(Q_k) \\ &= R^a(x) + \gamma \sum_{y \in X} P^a(x,y)(\max_{a' \in A} Q_k(y,a')); \qquad \text{for } (x,a) \in (X,A)_k \\ Q_{k+1}(z,b) &= Q_k(z,b); \qquad \text{for } (z,b) \in ((X \times A) - (X,A)_k). \end{aligned} \tag{4.9}$$

One advantage of asynchronous Q-value iteration over asynchronous value iteration is that it allows the agent to select randomly both the state as well as the action to be updated (asynchronous value iteration only allows the agent to select the state randomly). A convergence proof for synchronous Q-value iteration takes the same form as a proof of convergence for synchronous value iteration because the operator $B(Q)$ is also a contraction (see Appendix C for proof). For the asynchronous case the proof requires the application of the asynchronous convergence theorem in Bertsekas and Tsitsiklis [18].

Another advantage of using Q-values is that it becomes possible to do stochastic approximation by sampling not only the state-action pair but the next state as well. Define $G_x^a(Q) = R^a(x) + \gamma \sum_{y \in X}(P^a(x,y) \max_{a' \in A} Q(y,a'))$. Further, define $Y_x^a(Q) = Q(x,a) - (R^a(x) + \gamma \max_{a' \in A} Q(y,a'))$, where the next state y occurs with probability

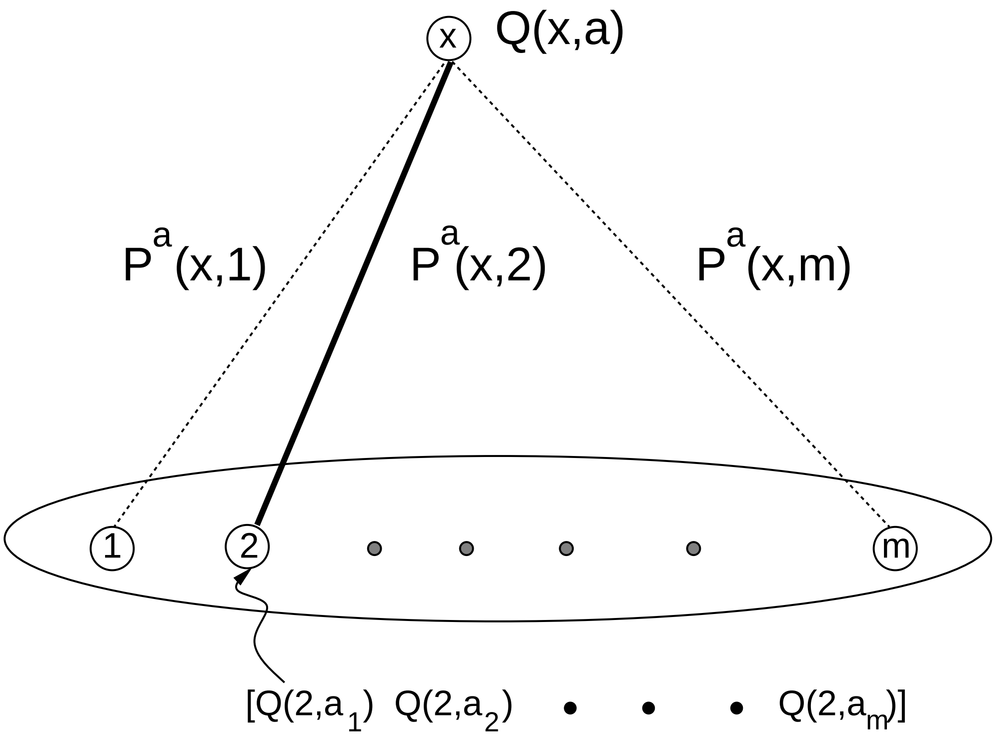

Figure 4.3 Sample Backup Operator. This figure shows the possible transitions for action a in state x. The transition marked in a solid line represents the actual transition that occurred on one occasion. The only information available for a sample backup then is the payoff along that transition and the Q-values stored for state 2.

$P^a(x, y)$. Note that $E\{Y_x^a(Q)\} = G_x^a(Q)$. Following the stochastic approximation recipe defined in Equation 4.2, the following algorithm can be defined:
Asynchronous Robbins-Monro Q-value Iteration:

$$\begin{aligned}
Q_{k+1}(x,a) &= Q_k(x,a) - \rho_k(x,a) Y_x^a(Q_k); \qquad \text{for } (x,a) \in (X,A)_k) \\
&= Q_k(x,a) - \rho_k(x,a)(Q_k(x,a) \\
&\qquad -(R^a(x) + \gamma \max_{a' \in A} Q_k(y,a'))), \\
Q_{k+1}(z,b) &= Q_k(z,b); \qquad \text{for } (z,b) \in ((X \times A) - (X,A)_k).
\end{aligned} \tag{4.10}$$

The agent does not need a model to compute $Y_x^a(Q)$ because it can just execute action a in state x and compute the maximum of the set of Q-values stored for the resulting state y (Figure 4.3).

Equation 4.10 is also an iterative relaxation algorithm of the form specified in Equation 3.5 with a random sample backup operator $\mathcal{B}_x^a(Q) = R^a(x) + \gamma \max_{a' \in A} Q(y, a')$ that is unbiased with respect to $B_x^a(Q)$. The sample backup operator $\mathcal{B}_x^a(Q)$ is shown pictorially in Figure 4.3, where instead of computing the expected value of the maximum Q-values of all possible next states, it samples a next state with the probability distribution defined by the state-action pair (x, a), and computes its maximum Q-value. Note that $E\{\mathcal{B}(\mathcal{Q})\} = B(Q)$.

The on-line version of Equation 4.10, i.e., where $(X, A)_{t+1} = \{(x_t, a_t)\}$ is the Q-learning algorithm invented by Watkins. As in the case of policy evaluation, an asymptotic convergence proof for the synchronous version of Equation 4.10 can be derived in a straightforward manner from Dvoretzky's results. However, more recently, Jaakkola, Jordan, and Singh [53] have proved convergence for the asynchronous algorithm that subsumes the synchronous algorithm:

Theorem 6: (Jaakkola, Jordan and Singh [53]) For finite MDPs, the algorithm defined in Equation 4.10 converge with probability one to Q^* if the following conditions hold true:

1. every state-action pair in $(X \times A)$ is updated infinitely often,
2. Q_0 is finite, and
3. $\forall (x, a) \in (X \times A)$; $\sum_{i=0}^{\infty} \rho_i(x, a) = \infty$ and $\sum_{i=0}^{\infty} \rho_i^2(x, a) < \infty$.

4.3.3 Discussion

Convergence proofs for TD and Q-learning that do not use stochastic approximation theory already existed in the RL literature (Sutton [106] and Dayan [33] for TD, and Watkins [118] and Watkins and Dayan [119] for Q-learning). But these proofs, especially the one for Q-learning, are based on special mathematical constructions that obscure the underlying simplicity of the algorithms. The connection to stochastic approximation provides a uniform framework for proving convergence of all the different RL algorithms. It also provides a conceptual framework that emphasizes the main innovation of RL algorithms, that of using unbiased-sample backups, relative to previously developed algorithms for solving MDTs.

Figure 4.4 graphs the relationship between iterative DP and RL algorithms that solve the policy evaluation problem along the following two dimensions: synchronous versus asynchronous, and full backups versus sample backups. Figure 4.5 does the same for DP and RL algorithms that solve the optimal control problem. All of the algorithms developed in the DP literature lie on the upper corners of the squares in Figures 4.4 and 4.5. By adding the "sample versus full backup" dimension, RL researchers have added the lower two corners to these pictures. Only the lower left-hand corner (asynchronous and sample backup) contains model-free algorithms; all the other corners must contain model-based algorithms (marked M).

Figure 4.6 shows a "constraint-diagram" representation of DP and RL algorithms that makes explicit the increased generality of application of RL algorithms. It is a Venn-diagram that shows the constraints required for applicability of the different classes of algorithms presented in Chapters 3 and 4. It shows that on-line RL algorithms are the "weakest" in the sense that they are applicable whenever any of the other algorithms are applicable. The next level of off-line RL algorithms require a model and can be applied whenever asynchronous DP and synchronous DP are applicable. Of course, it is not clear that one would ever want to do off-line RL instead of asynchronous DP. Section 4.4 studies that question. Asynchronous DP, the next level, requires full backups and can be applied whenever synchronous DP algorithms can be applied. Synchronous DP is the most restrictive class of algorithms.

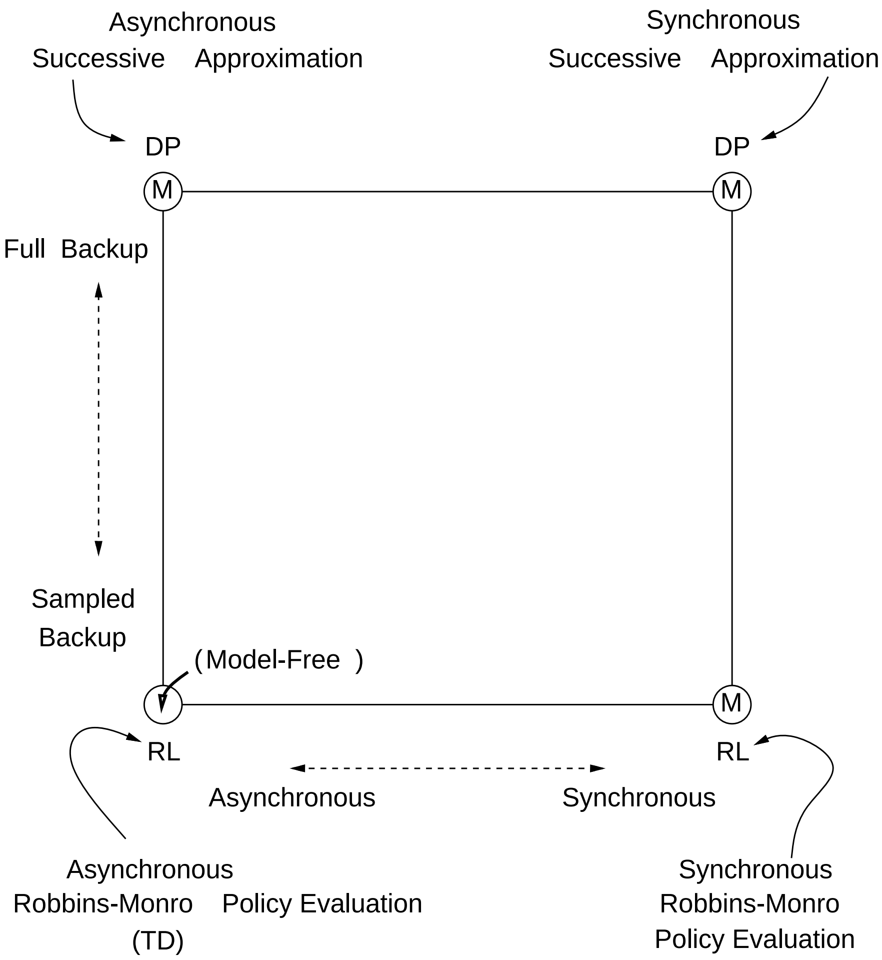

Figure 4.4 Iterative Algorithms for Policy Evaluation. This figure graphs the different DP and RL algorithms for solving the policy evaluation problem along two dimensions: full backup versus sample backup, and asynchronous versus synchronous.

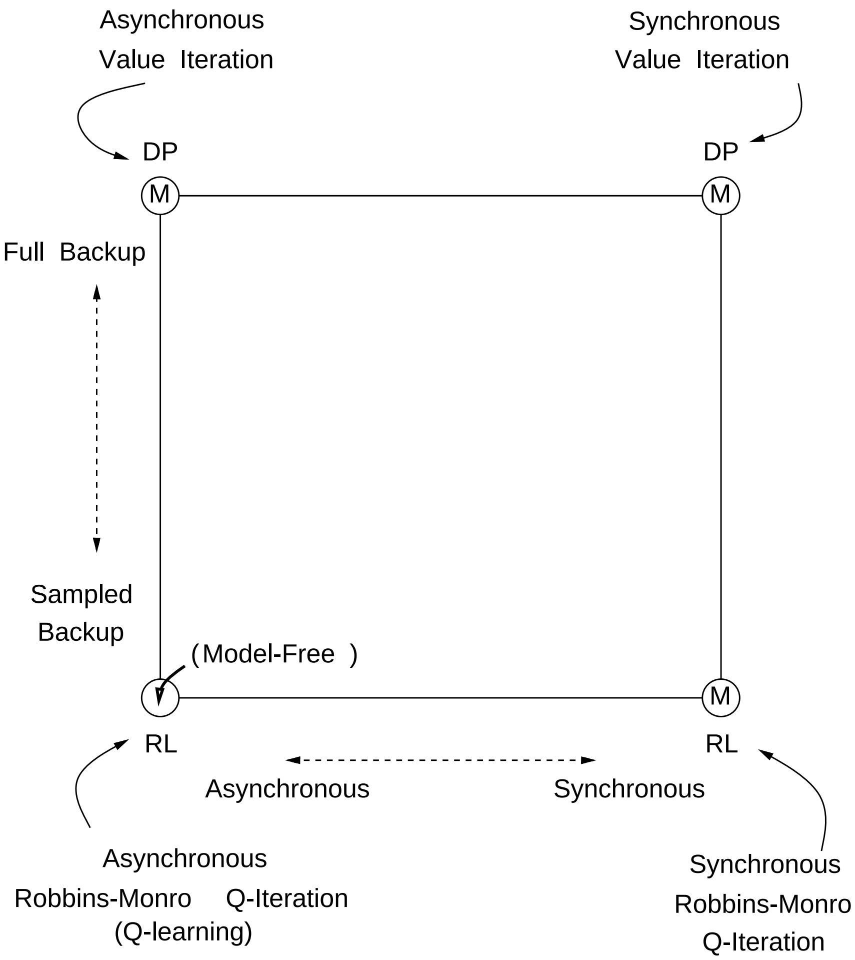

Figure 4.5 Iterative Algorithms for Solving the Optimal Control Problem. This figure graphs the different DP and RL algorithms for solving the optimal control problem along two dimensions: full backup versus sample backup, and asynchronous versus synchronous.

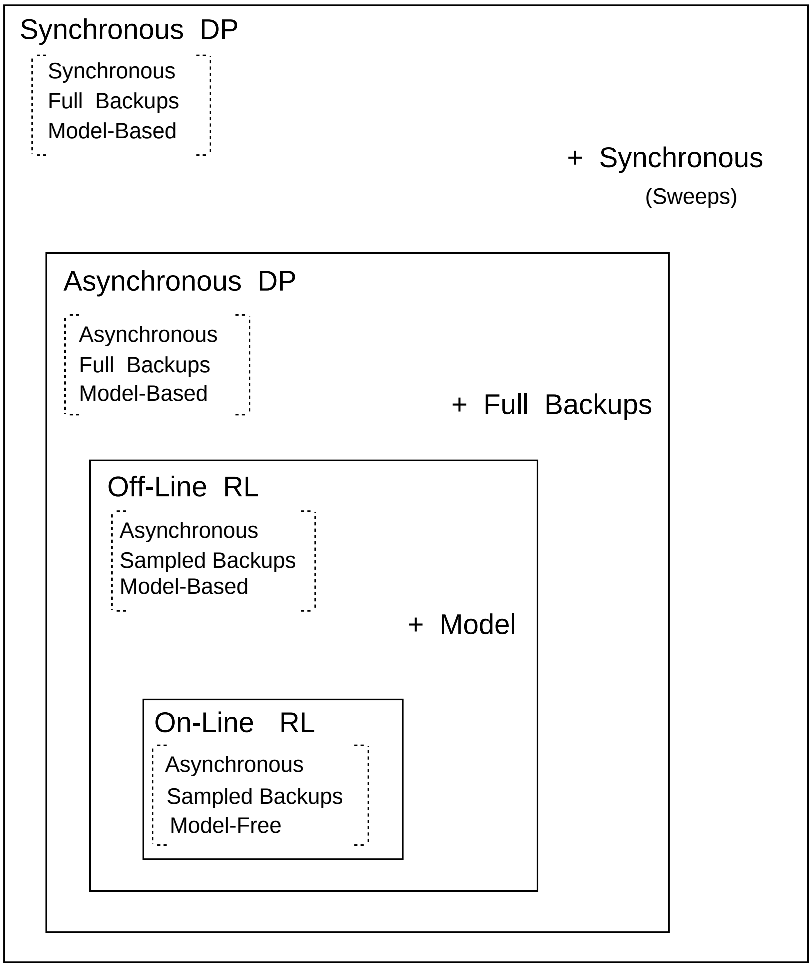

Figure 4.6 A Constraint-Diagram of Iterative Algorithms for Solving MDTs. This figure is a Venn-diagram like representation of the constraints required for all the different DP and RL algorithms reviewed in Chapters 3 and 4. Synchronous DP has the largest box because it places the most constraints.

Table 4.1 Tradeoff Between Sample Backup and Full Backup.

Sample Backup	Full Backup
Cheap	Expensive
Noisy	Informative

4.4 When to Use Sample Backups?

A sample backup provides a noisy and therefore less informative estimate than a full backup. A natural question that arises then is: are there conditions under which it can be computationally advantageous to use an iterative algorithm based on sample backups over an algorithm based on full backups to solve the optimal control problem? To simplify the argument somewhat, the basis of comparison between algorithms is assumed to be the number of state transitions, whether real or simulated, required for convergence. The number of state transitions is a fair measure because both sample and full backups require one multiplication operation for each state transition (see Equations 3.12, 4.10, 3.9 and 4.5).

The number of state transitions involved in a full backup at state-action pair (x, a) is equal to the number of possible next states, or the *branching factor* for action a. A sample backup always involves a single transition. In general, the number of possible next states could be as high as $|X|$, and therefore a sample backup can be as much as $|X|$ times cheaper than a full backup. In summary, the increased information provided by a full backup comes at the cost of increased computation, raising the possibility that for some MDTs doing $|X|$ sample backups may be more informative than a single full backup (see Table 4.1). Note that the estimate derived from a single sample backup $(\mathcal{B}(V))$ is unbiased with respect to the estimate derived from a single full backup $(B(V))$.

If multiple sample backups are performed without changing the Q-value function Q (or V), it would reduce the variance of the resulting estimate, and thereby make it a closer approximation to the estimate returned by a single full backup. However, in general the value function changes after every application of a sample backup, and therefore multiple updates with a sample backup could lead to a better approximation to the optimal value function than one update with a full backup. Let us consider this possibility in the following two situations separately: one where the agent is given an accurate environment model to start with, and the other where the agent is not provided with an environment model.

4.4.1 Agent is Provided With an Accurate Model

The relative speed of convergence of an algorithm based on sample backups versus an algorithm based on full backups will in general depend on the order in which the backups are applied to the state-action space. To keep things as even as possible, this section presents results comparing synchronous off-line Q-learning that employs sample backups and synchronous Q-value iteration that employs full backups. Therefore the only difference between the two algorithms is in the nature of the backup operator applied to produce new estimates; both algorithms apply the operators in sweeps through the state-action space.

Clearly if the MDT is deterministic, sample backups and full backups are identical, and therefore Q-learning and Q-value iteration are identical. For stochastic MDTs the variance of the estimate based on a sample backup, say $\mathcal{B}_x^a(Q_k)$, about the estimate returned by a full backup, $B_x^a(Q_k)$, is a function of the variance in the product $P^a(x,y)V_k(y)$, where the next state y is chosen randomly with uniform probability. For the empirical studies presented in this section, it is assumed that the effect of the value function on the variance is negligible and therefore only the effect of the non-uniformity in the transition probabilities is important.

The relative performance of synchronous off-line Q-learning and synchronous Q-value iteration was studied on artificially constructed MDTs that are nearly deterministic, but whose branching factor is exactly $|X|$ for every state-action pair. The MDTs were constructed so that for each state-action pair the transition probabilities form a Gaussian on some permutation of the next-state set X. Because the problems are artificial, the amount of determinism, or inversely the variance of the Gaussian transition probability distribution, could be controlled to study the effect of decreasing determinism on the relative speeds of convergence of the following two algorithms:

1. `Algorithm 1 (Gauss-Sidel Q-value iteration):`
   `for (`$i=0$`;` $i<$ `number-of-sweeps;` $i++$`)`
   `for each state-action pair`
   `do a full backup`

2. `Algorithm 2 (Gauss-Sidel off-line Q-learning):`
   `for (`$i=0$`;` $i<$ `(number-of-sweeps` $\times$ $|X|$`);` $i++$`)`
   `for each state-action pair`
   `do a sample backup`

Figures 4.7, 4.8, 4.9 and 4.10 show the relative performance of Algorithms 1 and 2 for MDTs with 50 states, 100 states, 150 states, and 200 states respectively. Each figure shows three graphs: the left-hand graph in the top panel shows the relative performance when the transition probability Gaussian has 95% of its probability mass concentrated on 3 states, the right-hand graph in the top panel shows the relative performance when the transition probability Gaussian has 95% of its mass concentrated on about 32% of the states, and the graph in the lower panel shows the relative performance when the variance of the transition probability Gaussian is designed so that 95% of the mass is concentrated on 65% of the states. For a fixed $|X|$, the increasing variance of the transition probability Gaussian is intended to reveal the decreasing advantage of Algorithm 2 over Algorithm 1 as the problem becomes less deterministic.

Each graph presents results averaged over 10 different runs with 10 different seeds for a random number generator. The x-axis shows the number of sweeps for Algorithm 1. For each sweep of Algorithm 1, $|X|$ sweeps of Algorithm 2 were performed. The performance of the two algorithms was computed as follows. After each sweep through the state space for Algorithm 1, and after every $|X|$ sweeps for Algorithm 2, the

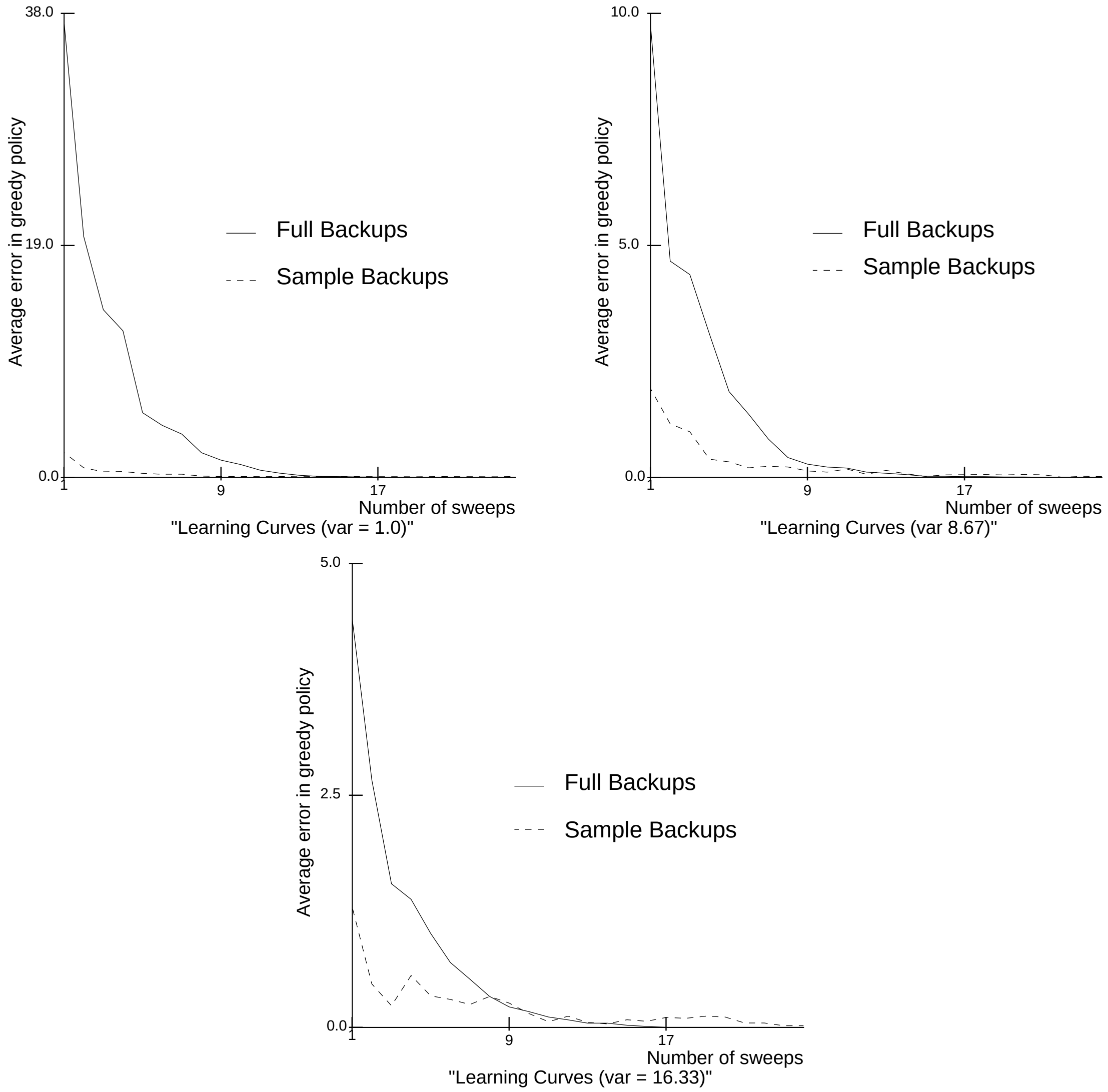

Figure 4.7 Full versus Sample Backups for 50 state MDTs. This figure shows three graphs: the upper left graph for MDTs that have 95% of the transition probability mass concentrated on 3 randomly chosen next states, the upper right graph for MDTs that have the transition probability mass concentrated on 32% of the states, and the lower graph for MDTs that have the transition probability mass concentrated on 65% of the states. The x-axis is the number of sweeps of Algorithm 1 (full backups). Note that for each sweep of Algorithm 1, 50 sweeps of Algorithm 2 (sample backups) are performed. The y-axis shows the average loss of the greedy policy. As expected the sample backup algorithm outperforms the full backup algorithm. Also as the amount of determinism is decreased the relative advantage of sample over full backups gets smaller.

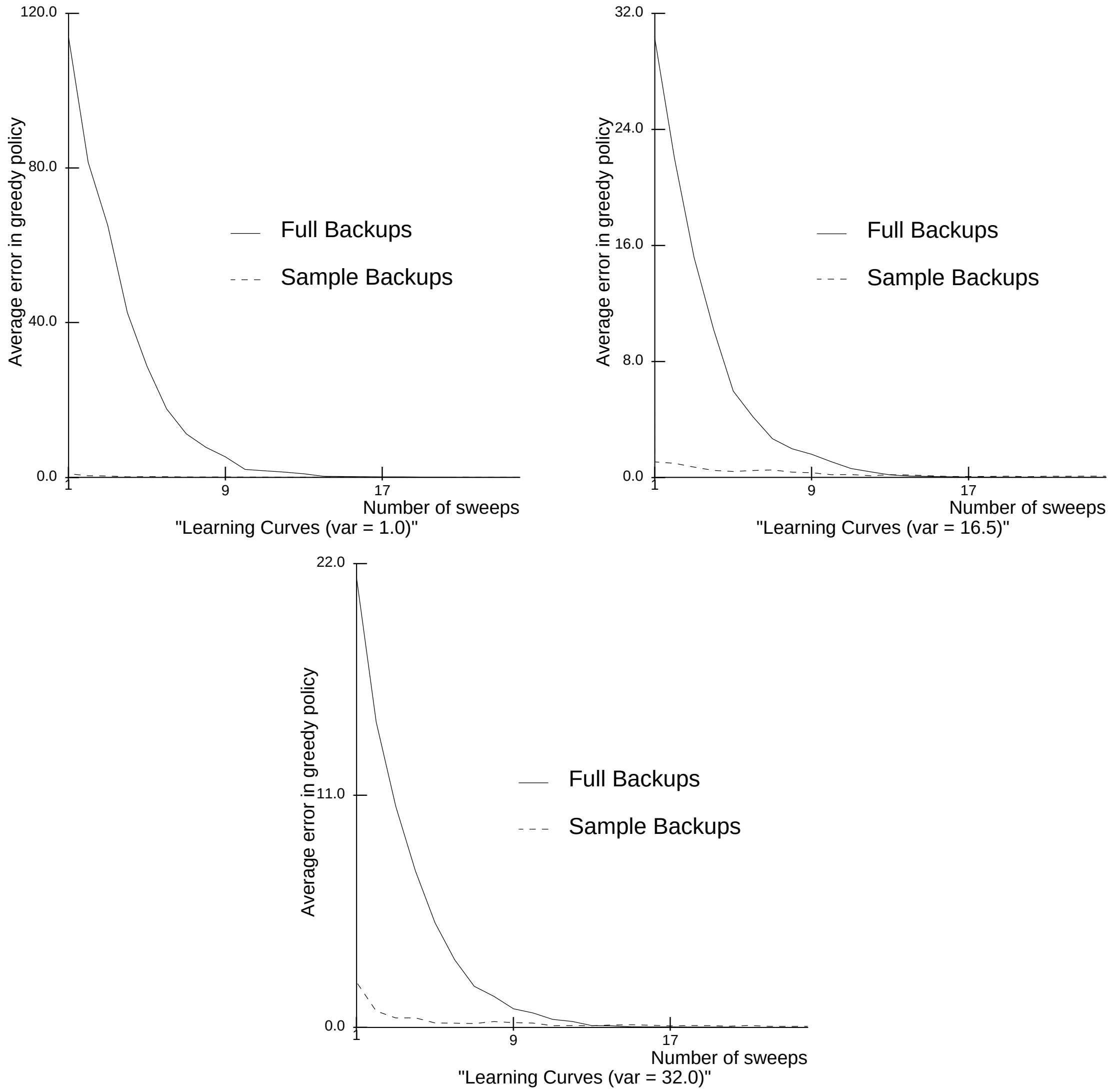

Figure 4.8 Full versus Sample Backups for 100 state MDTs. This figure shows three graphs: the upper left graph for MDTs that have 95% of the transition probability mass concentrated on 3 randomly chosen next states, the upper right graph for MDTs that have the transition probability mass concentrated on 32% of the states, and the lower graph for MDTs that have the transition probability mass concentrated on 65% of the states. The x-axis is the number of sweeps of Algorithm 1 (full backups). Note that for each sweep of Algorithm 1, 50 sweeps of Algorithm 2 (sample backups) are performed. The y-axis shows the average loss of the greedy policy. As expected the sample backup algorithm outperforms the full backup algorithm. Also as the amount of determinism is decreased the relative advantage of sample over full backups gets smaller.

greedy policy was derived and fully evaluated. The error value after each sweep is the cumulative absolute difference between the value function of the greedy policy and the pre-computed optimal policy, averaged over the 10 different runs. The y-axis shows the error value.

Figures 4.7 ,4.8, 4.9 and 4.10 each show that Algorithm 2, which uses sample backups, clearly outperforms Algorithm 1, which uses full backups, by a big margin. Another expected result is that as the variance in the Gaussian transition probabilities is increased, the relative advantage of Algorithm 2 reduces in each case. For example, the ratio of the error value after the first sweep in Algorithm 1 to the error value after $|X|$ sweeps in Algorithm 2 in the 100-state MDT is about 60.0 for the most deterministic problem, about 8.0 in the middle problem, and about 5.0 in the least deterministic problems. A similar decline is noticed in the other three sets of MDTs. Another effect to notice is that the relative advantage of Algorithm 2 over Algorithm 1 increases in ratio as the size of the MDT is increased, at least for the problem sizes studied here.

4.4.2 Agent is not Given a Model

If the agent is not given a model a priori, an algorithm that uses full backups would have to be adaptive, i.e., estimate a model on-line by using information about state transitions achieved in the real environment. In such a case, because an algorithm that uses full backups is still model-based, the relative advantages presented in the previous section in favor of algorithms that use sample backups will continue to hold. However, because the model is being estimated on-line, there are two additional reasons to prefer an algorithm that uses sample backups. First, an algorithm that uses sample backups can avoid the computational expense of building a model. Second, during the early stages of learning, the model will be highly inaccurate and can interfere with learning the value function by adding a "bias" to the full backup operator (see Barto and Singh [12, 11]).

Nevertheless, in general it is difficult to predict which of the two algorithms, one based on sample backups, and the other based on full backups on an estimated model, will outperform the other. The term adaptive full backup is used to denote a full backup performed on simulations from an estimated model. It is instructive to contrast both the sample backup algorithm and the adaptive full backup algorithm with a second non-adaptive full backup algorithm that is assumed to have access to the correct model, that is, to compare the following three backup operators:

$$\begin{aligned}
\text{Sample backup:} &= R^a(x) + \gamma \max_{a' \in A} Q(y, a') \\
\text{Adaptive full backup:} &= R^a(x) + \gamma \sum_{y \in X} [\hat{P}^a(x, y) \max_{a' \in A} Q(y, a')] \\
\text{Non-adaptive full backup:} &= R^a(x) + \gamma \sum_{y \in X} [P^a(x, y) \max_{a' \in A} Q(y, a')]
\end{aligned}$$

where $\hat{P}$ are the estimated transition probabilities that are learned on-line by the adaptive full backup algorithm.

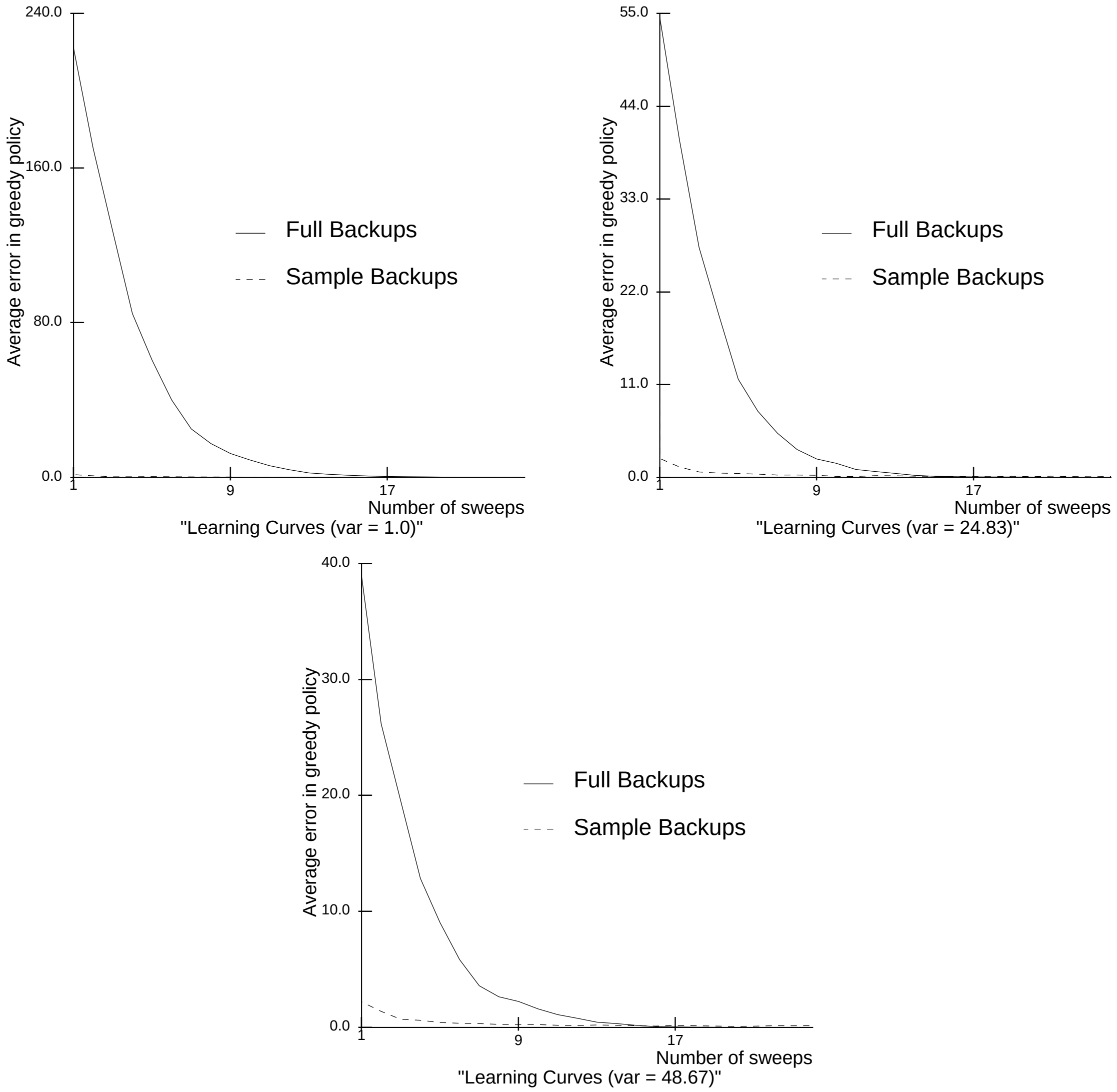

Figure 4.9 Full versus Sample Backups for 150 state MDTs. This figure shows three graphs: the upper left graph for MDTs that have 95% of the transition probability mass concentrated on 3 randomly chosen next states, the upper right graph for MDTs that have the transition probability mass concentrated on 32% of the states, and the lower graph for MDTs that have the transition probability mass concentrated on 65% of the states. The x-axis is the number of sweeps of Algorithm 1 (full backups). Note that for each sweep of Algorithm 1, 50 sweeps of Algorithm 2 (sample backups) are performed. The y-axis shows the average loss of the greedy policy. As expected the sample backup algorithm outperforms the full backup algorithm. Also as the amount of determinism is decreased the relative advantage of sample over full backups gets smaller.

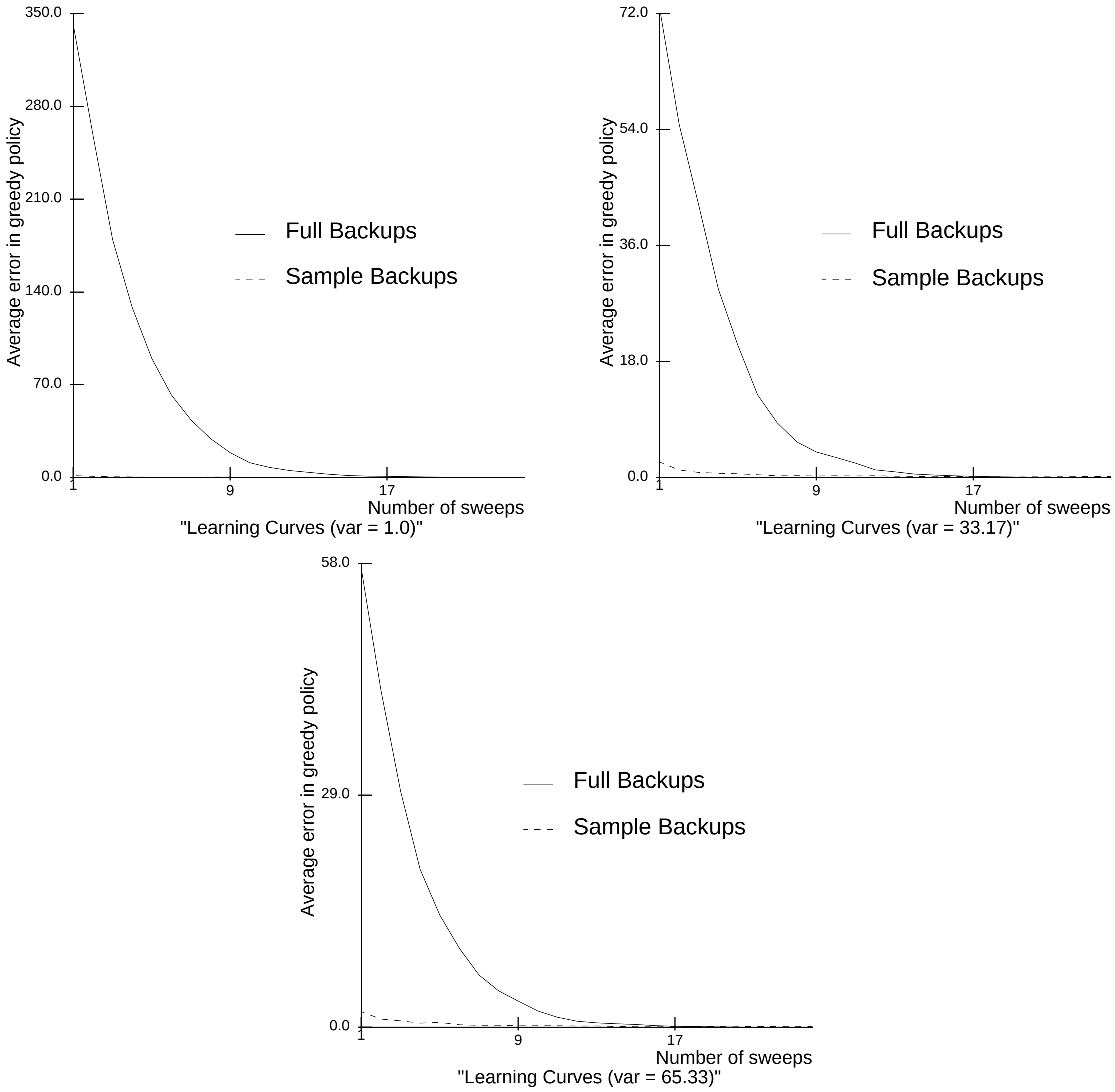

Figure 4.10 Full versus Sample Backups for 200 state MDTs. This figure shows three graphs: the upper left graph for MDTs that have 95% of the transition probability mass concentrated on 3 randomly chosen next states, the upper right graph for MDTs that have the transition probability mass concentrated on 32% of the states, and the lower graph for MDTs that have the transition probability mass concentrated on 65% of the states. The x-axis is the number of sweeps of Algorithm 1 (full backups). Note that for each sweep of Algorithm 1, 50 sweeps of Algorithm 2 (sample backups) are performed. The y-axis shows the average loss of the greedy policy. As expected the sample backup algorithm outperforms the full backup algorithm. Also as the amount of determinism is decreased the relative advantage of sample over full backups gets smaller.

Table 4.2 Bias-Variance Tradeoff in RL and Adaptive DP

Stage	*Sample backup*	*Adaptive full backup*
early	no bias low-high variance	high bias no variance
middle	no bias low-high variance	medium bias no variance
late	no bias low-high variance	low bias no variance

The following table summarizes the relative error in the estimates provided by the sample backup and the adaptive full backup with respect to a non-adaptive full backup as function of the stage of learning. Table 4.2 states that the sample backup is always unbiased with respect to the estimate provided by a full backup on a correct model. The variance in the sample backup can be low to high depending on the "skew" in the value function as well as in the transition probabilities. On the other hand, the adaptive full backup will have a high bias with respect to the non-adaptive case in the early stages of learning because of the large error in the estimated model, but as the estimate model becomes better over time, the bias will go away. Since the full backup operator computes expected values there is no variance in the estimate provided by the adaptive full backup operator.

The entries in Table 4.2 can lead to the conclusion that an algorithm that estimates a model, but uses sample backups during the early stages of learning, and then switches to doing full backups using simulated experience with the estimated model when its bias gets small enough, could get the best of both worlds. This conclusion is based on the fact that depending upon the bias and variance values it can be better to sample from a biased source with low variance than an unbiased source with high variance. However, it is unclear as to how the crossover point, i.e., the point at which to switch from sample backups to full backups, could be determined in practice, since the bias and the variances of the backup operators are not known. It may be possible to compute estimates of the bias and variance of the backup operators on-line to determine the crossover point, but in general it is unclear whether the potential savings will be more than the computational expense involved in doing so. In any case, the potential savings of such a hybrid method will be determined in large part by how early in the learning process the crossover point is reached.

4.4.3 Discussion

It was shown in this section that sample backup algorithms are likely to have significant advantage over full backup algorithms in MDTs that are nearly deterministic and yet have a large branching factor averaged over state-action pairs. While this does not conclude the ongoing debate over the relative merits of model-free versus model-based RL methods, it does provide additional evidence that there exist problems where model-free algorithms are more efficient than model-based ones.

4.5 Shortcomings of Asymptotic Convergence Results

A major limitation of most[1] current theoretical research on RL, including the research presented in this chapter, is its dependence on the following two assumptions:

1. that a lookup-table is used to represent the value (or Q-value) function,

2. that each state (or state-action pair) is updated infinitely often.

Both of these assumptions are unrealistic in practice.

Several researchers, including this author, have used function approximation methods other than lookup-tables, e.g., neural networks, to represent the value (or Q-value) function. With non lookup-table representations the following two factors could prevent convergence to V^* (or Q^*):

- V^* (or Q^*) may not be in the class of functions that the chosen function approximation architecture can represent. This is not possible to know in advance because V^* (and Q^*) is not known in advance. However, *constructive* function approximation approaches (e.g., Fahlman and Lebiere [40]) may be able to alleviate this problem.

- A more fundamental issue is that a non lookup-table function approximation method can generalize an update in such a way that the essential "contraction-based" convergence of DP-related algorithms may be thwarted.

This raises the following important question: if practical concerns dictate that value functions be approximated, how might performance be affected[2]? Is it possible that, despite some empirical evidence to the contrary (e.g., Barto *et al.* [13], Anderson [2], Tesauro [112]), small errors in approximations could result in arbitrarily bad performance in principle? If so, this could raise significant concerns about the use of function approximation in DP-based learning.

4.5.1 An Upper Bound on the Loss from Approximate Optimal-Value Functions

This section extends a result by Bertsekas [17] which guarantees that small errors in the approximation of a task's optimal value function cannot produce arbitrarily bad performance when actions are selected greedily. Specifically, the extension is in deriving an upper bound on performance loss which is slightly tighter than that

---

[1]Sutton [106] and Dayan [33] have proved convergence for the TD algorithm when linear networks are used to represent the value functions. Bradtke [20] adapted Q-learning to solve linear quadratic regulation (LQR) problems and proved convergence under the assumption that a linear-in-the-parameters network is used to store the Q-value function.

[2]Even with lookup-table representations, in practice it may be difficult to visit every state-action pair often enough to ensure convergence to the optimal value function. Thus the issue of how performance gets affected by using approximations to V^* is relevant even to lookup-tables.

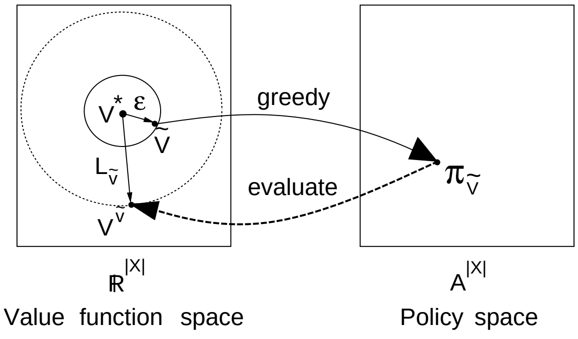

Figure 4.11 Given $\tilde{V}$, an approximation within $\epsilon > 0$ of V^*, derive the corresponding greedy policy $\pi_{\tilde{V}}$. The resulting loss in value, $L_{\tilde{V}} = V^* - V^{\tilde{V}}$, is bounded above by $(2\gamma\epsilon)/(1-\gamma)$.

in Bertsekas [17], and in deriving the corresponding upper bound for the newer Q-learning algorithm. These results also provide a theoretical justification for a practice that is common in RL. The material presented in this section was developed in collaboration with Yee and is reported in Singh and Yee [103].

Policies can be derived from value functions in a straightforward way. Given value function $\tilde{V}$, a *greedy policy* $\pi_{\tilde{V}}$ can be defined as

$$\pi_{\tilde{V}}(x) = \text{argmax}_{a \in A} \left[R^a(x) + \gamma \sum_{y \in X} P^a(x,y)\,\tilde{V}(y) \right],$$

where ties for the maximum action are broken arbitrarily. Figure 4.11 illustrates the relationship between the evaluation of policies and the derivation of greedy policies. A greed policy $\pi_{\tilde{V}}$ gives rise to its own value function $V^{\pi_{\tilde{V}}}$, denoted simply as $V^{\tilde{V}}$. In general, $\tilde{V} \neq V^{\tilde{V}}$, i.e., the value function used for deriving a greedy policy is different from the value function resulting from the policy's evaluation. Equality between $\tilde{V}$ and $V^{\tilde{V}}$ occurs if and only if $\tilde{V}$ is optimal, in which case any greedy policy will be optimal.

For a greedy policy $\pi_{\tilde{V}}$ derived from $\tilde{V}$, an approximation to V^*, define the *loss function* $L_{\tilde{V}}$ such that $\forall x \in X$,

$$L_{\tilde{V}}(x) = V^*(x) - V^{\tilde{V}}(x).$$

$L_{\tilde{V}}(x)$ is the expected loss in the value of state x resulting from the use of policy $\pi_{\tilde{V}}$ instead of an optimal policy. Note that $L_{\tilde{V}}(x) \geq 0$ because $V^*(x) \geq V_{\tilde{V}}(x)$. The following theorem gives an upper bound on the loss $L_{\tilde{V}}$.

Theorem 7: Let V^* be the optimal value function for a discrete-time MDT having finite states and actions and an infinite, geometrically discounted horizon, $\gamma \in [0,1)$. If $\tilde{V}$ is a function such that $\forall x \in X$, $\left| V^*(x) - \tilde{V}(x) \right| \leq \epsilon$, and $\pi_{\tilde{V}}$ is a greedy policy for $\tilde{V}$, then $\forall x$,

$$L_{\tilde{V}}(x) \leq \frac{2\gamma\epsilon}{1-\gamma}.$$

The upper bound of Theorem 7 is tighter than the result in Bertsekas [17] by a factor of γ (Cf. [p. 236, #14(c)]). One interpretation of this result is that if the approximation to the optimal value function is off by no more than ϵ, then the average worst-case loss per time step cannot be more than $2\gamma\epsilon$, under a greedy policy.

Proof: There exists a state z that achieves the maximum loss: $\exists z \in X$, $\forall x \in X$, $L_{\tilde{V}}(z) \geq L_{\tilde{V}}(x)$. For state z consider an optimal action, $a = \pi^*(z)$, and the action of $\pi_{\tilde{V}}$, $b = \pi_{\tilde{V}}(z)$. Because $\pi_{\tilde{V}}$ is a greedy policy for $\tilde{V}$, b must appear at least as good as a:

$$\begin{aligned} R^a(z) + \gamma \sum_{y \in X} P^a(z,y)\,\tilde{V}(y) &\leq R^b(z) + \gamma \sum_{y \in X} P^b(z,y)\,\tilde{V}(y) && (4.11)\\ R^a(z) - R^b(z) &\leq \gamma \sum_y \left[P^b(z,y)\,\left(V^*(y)+\epsilon\right) - P^a(z,y)\,\left(V^*(y)-\epsilon\right)\right] \\ R^a(z) - R^b(z) &\leq 2\gamma\epsilon + \gamma \sum_y \left[P^b(z,y)\,V^*(y) - P^a(z,y)\,V^*(y)\right]. && (4.12) \end{aligned}$$

The maximal loss is

$$\begin{aligned} L_{\tilde{V}}(z) &= V^*(z) - V^{\tilde{V}}(z) \\ &= R^a(z) - R^b(z) + \gamma \sum_y \left[P^a(z,y)V^*(y) - P^b(z,y)V^{\tilde{V}}(y)\right]. && (4.13) \end{aligned}$$

Substituting from (4.12) gives

$$\begin{aligned} L_{\tilde{V}}(z) &\leq 2\gamma\epsilon + \gamma \sum_y \Big[P^b(z,y)\,V^*(y) - P^a(z,y)\,V^*(y) \\ &\qquad\qquad + P^a(z,y)\,V^*(y) - P^b(z,y)\,V^{\tilde{V}}(y)\Big] \\ L_{\tilde{V}}(z) &\leq 2\gamma\epsilon + \gamma \sum_y P^b(z,y)\,\left[V^*(y) - V^{\tilde{V}}(y)\right] \\ L_{\tilde{V}}(z) &\leq 2\gamma\epsilon + \gamma \sum_y P^b(z,y)\,L_{\tilde{V}}(y) \end{aligned}$$

Because, by assumption, $\forall y \in X,\ L_{\tilde{V}}(z) \geq L_{\tilde{V}}(y)$,

$$\begin{aligned} L_{\tilde{V}}(z) &\leq 2\gamma\epsilon + \gamma \sum_y P^b(z,y)\,L_{\tilde{V}}(z) \\ L_{\tilde{V}}(z) &\leq \frac{2\gamma\epsilon}{1-\gamma} \end{aligned}$$

Q.E.D.

This result extends to a number of related cases.

4.5.1.1 Approximate payoffs

Theorem 7 assumes that the expected payoffs are known exactly. If the true expected payoffs $R^a(x)$ are approximated by $\tilde{R}^a(x)$, the upper bound on the loss is as follows.

Corollary $7A$**:** If $\forall x \in X$, $\left|V^*(x) - \tilde{V}(x)\right| \le \epsilon$ and $\forall a \in A$, $\left|R^a(x) - \tilde{R}^a(x)\right| \le \alpha$, then $\forall x$,

$$L_{\tilde{V}}(x) \le \frac{2\gamma\epsilon + 2\alpha}{1-\gamma},$$

where $\pi_{\tilde{V}}$ is the greedy policy for $\tilde{V}$.

Proof: Inequality (4.11) becomes

$$\tilde{R}^a(z) + \gamma \sum_{y \in X} P^a(z,y)\,\tilde{V}(y) \le \tilde{R}^b(z) + \gamma \sum_{y \in X} P^b(z,y)\,\tilde{V}(y),$$

and (4.12) becomes

$$R^a(z) - R^b(z) \le 2\gamma\epsilon + 2\alpha + \gamma \sum_y \left[P^b(z,y)\,V^*(y) - P^a(z,y)\,V^*(y)\right].$$

Substitution into (4.13) yields the bound. Q.E.D.

4.5.1.2 Q-learning

If neither the payoffs nor the state-transition probabilities are known, then the analogous bound for Q-learning is as follows. Evaluations are defined by

$$Q^\pi(x_t, a) = R^a(x_t) + \gamma E\left\{V_\pi(x_{t+1})\right\},$$

where $V_\pi(x) = \max_a Q^\pi(x,a)$. Given function $\tilde{Q}$, the greedy policy $\pi_{\tilde{Q}}$ is defined by

$$\pi_{\tilde{Q}}(x) = \mathrm{argmax}_{a \in A(x)}\,\tilde{Q}(x,a).$$

The loss is then expressed as

$$L_{\tilde{Q}}(x) = Q^*(x,\,\pi^*(x)) - \tilde{Q}(x,\,\pi_{\tilde{Q}}(x)).$$

Corollary $7B$**:** If $\forall x \in X$, $\forall a \in A(x)$, $\left|Q^*(x,a) - \tilde{Q}(x,a)\right| \le \epsilon$, then $\forall x$,

$$L_{\tilde{Q}}(x) \le \frac{2\epsilon}{1-\gamma}.$$

Proof: Inequality (4.11) becomes $\tilde{Q}(z,a) \le \tilde{Q}(z,b)$, which gives

$$\begin{aligned}
Q^*(z,a) - \epsilon &\le Q^*(z,b) + \epsilon \\
R^a(z) + \gamma \sum_y P^a(z,y)\,V^*(y) - \epsilon &\le R^b(z) + \gamma \sum_y P^b(z,y)\,V^*(y) + \epsilon \\
R^a(z) - R^b(z) &\le 2\epsilon + \gamma \sum_y \left[P^b(z,y)\,V^*(y) - P^a(z,y)\,V^*(y)\right].
\end{aligned}$$

Substitution into (4.13) yields the bound. Q.E.D.

4.5.2 Discussion

The bounds of Theorem 7 and its corollaries guarantee that the performance of DP-based learning approaches will not be far from optimal if (a) good approximations to optimal value functions are achieved, (b) a corresponding greedy policy is followed, and (c) the discount factor, γ, is not too close to 1.0. The analogous result holds for *indefinite* horizon, undiscounted MDTs (as defined in Barto *et al.*[10]). Although this result does not address convergence, it nevertheless helps to validate many practical approaches to DP-based learning that use approximations.

Theorem 7 does not directly address policy-derivation methods other than the indicated greedy one. For other methods, it is not clear, in general, what criteria to place on approximations of value functions because the criteria may depend upon the specifics of a derivation method. Greedy policy derivation allows one to specify an error-criterion on approximations, i.e., that they be within ϵ of optimal, under the max norm.

4.5.3 Stopping Criterion

Another unrealistic requirement to ensure convergence to V^* (or Q^*) is that each state (or state-action pair) be visited infinitely often. However, the real goal is not to converge to the optimal value function, but to derive an optimal policy. Indeed, as stated in the following theorem, for every finite MDT there is a spherical region (ball) around the optimal value function such that the greedy policy with respect to any value function in that ball is optimal.

Theorem 8: For any given finite action MDT, $\exists \epsilon > 0$, such that $\forall \tilde{V} \in (\Re^+)^{|X|}$, where $||\tilde{V} - V^*||_\infty < \epsilon$, any policy that is greedy with respect to $\tilde{V}$ is optimal.

Proof: For all $x \in X$, and $\forall a \in A$, define,

$$\begin{aligned} m(x,a) &= (R^{\pi^*(x)}(x) + \gamma \sum_{y \in X} P^{\pi^*(x)}(x,y)V^*(y)) - (R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V^*(y)), \\ &= Q^*(x, \pi^*(x)) - Q^*(x,a). \end{aligned}$$

Note that $m(x,a) \geq 0$ by definition. Let $\Pi^*(x)$ be the set of optimal actions in state x. Define

$$2\gamma\epsilon = \min_{x \in X}\{ \min_{a \in (A - \Pi^*(x))} \{m(x,a)\}\}. \tag{4.14}$$

Clearly ϵ will only be zero *iff* all stationary policies are optimal, in which case ϵ can be set to any value greater than zero. Therefore by the definition of ϵ, $\forall x \in X$, and $\forall a \in (A - \Pi^*(x))$, $Q^*(x, \pi^*(x)) - Q^*(x,a) \geq 2\gamma\epsilon > 0$. Let $\tilde{Q}$ be the Q-value function derived from the approximation $\tilde{V}$ that is assumed to satisfy the following: $||\tilde{V} - V^*||_\infty < \epsilon$. Therefore, for $a \in (A - \Pi^*(x))$:

$$\begin{aligned} \tilde{Q}(x,a) &= R^a(x) + \gamma \sum_{y \in X} P^a(x,y)\tilde{V}(y) \\ &< R^a(x) + \gamma \sum_{y \in X} P^a(x,y)(V^*(y) + \epsilon) \end{aligned}$$

$$
\begin{aligned}
&< \quad [R^a(x) + \gamma \sum_{y \in X} P^a(x,y)V^*(y)] + \gamma\epsilon \\
&< \quad Q^*(x,a) + \gamma\epsilon \\
&< \quad Q^*(x,\pi^*(x)) - \gamma\epsilon \\
&< \quad R^{\pi^*(x)}(x) + \gamma \sum_{y \in X} P^a(x,y)V^*(y) - \gamma\epsilon \\
&< \quad R^{\pi^*(x)}(x) + \gamma \sum_{y \in X} P^a(x,y)(\tilde{V}(y) + \epsilon) - \gamma\epsilon \\
&< \quad \tilde{Q}(x,\pi^*(x)),
\end{aligned}
$$

which implies that the greedy actions with respect to $\tilde{V}$ with ϵ defined in Equation 4.14 will be elements of the set $\Pi^*(x)$.
Q.E.D.

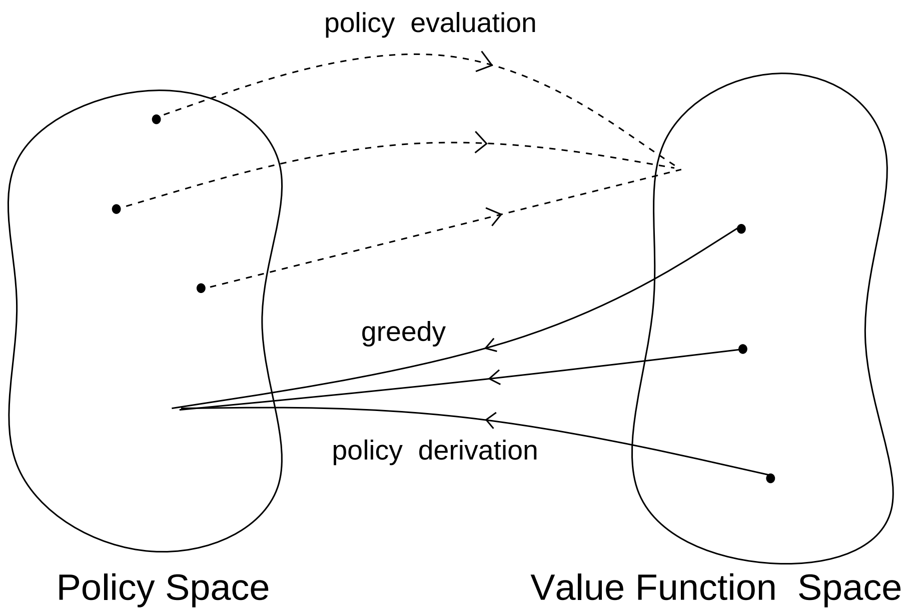

Figure 4.12 Mappings between Policy and Value Function Spaces. The greedy policy derivation mapping from value function space to policy space is many-to-one, nonlinear, and highly discontinuous. The policy evaluation mapping from policy space to value function space is many-to-one but linear. This makes it very difficult to discover stopping conditions.

It is possible that long before infinite visits to every state, the policy greedy with respect to the estimate of the optimal value function becomes optimal, e.g., when the estimated value function is the ball specified in Theorem 8. Unfortunately, in RL algorithms there is no general way to detect the iteration at which the greedy policy becomes optimal. It would be very beneficial to discover "stopping conditions" for RL algorithms which if met would indicate that with high probability the greedy policy derived from the current estimate of the value function would be optimal or close to optimal.

As seen in Section 4.5.1, the process of characterizing the possible loss from an approximation in value function space requires mapping the approximation into policy space and back into value function space (cf. Figures 4.11 and 4.12). This makes the discovery of stopping conditions hard because the mapping from the space of value functions to the space of stationary policies is many-to-one (Figure 4.12), and because it is non-linear and full of discontinuities. While the policy evaluation mapping from policy space to value function space is linear, it is also many-to-one. Another factor that complicates matters is that RL algorithms use sample backups, which are noisy and therefore only a probabilistic argument can be made in the first place.

4.6 Conclusion

This chapter establishes an important connection between stochastic approximation theory and RL and shows how RL researchers have added the sample backup versus full backup dimension to the class of iterative algorithms for solving MDTs. It provides empirical evidence that at least for certain MDTs, specifically those that are nearly deterministic and yet have a high branching factor, doing sample backups may be more efficient than doing full backups. Some preliminary theoretical work is presented on two of the pressing questions left unanswered by asymptotic convergence results: how do approximations affect convergence, and what are suitable stopping conditions. It is shown that minimizing max-norm distance to V^* is a good error criteria for function approximation, both because getting close to V^* will mean that the resulting greedy policy cannot be arbitrarily bad, and because there is a ball around V^* within which all greedy policies are optimal. The rest of this dissertation is focused on developing new RL architectures that address more practical concerns: learning multiple RL tasks efficiently, and ensuring safe performance while learning on-line.

CHAPTER 5

SCALING REINFORCEMENT LEARNING: PRIOR RESEARCH

While RL algorithms have many attractive properties, such as incremental learning, a strong theory, and proofs of convergence, conventional RL architectures are slow enough to make them impractical for many real-world applications. Much of the early developmental work in RL focused on establishing an abstract and general theoretical framework. In this early phase, RL architectures that used very little task-specific knowledge were developed and applied to simple and abstract problems to illustrate and help understand RL algorithms (e.g., Sutton [106], Watkins [118]).

In the 1990s RL research has moved out of the developmental phase and is increasingly focused on complex and diverse applications. As a result several researchers are investigating techniques for reducing the learning time of RL architectures to acceptable levels. Some of these acceleration techniques are based on incorporating task-specific knowledge into RL architectures. This chapter presents a brief survey of prior research on scaling RL algorithms followed by a preview of three approaches developed in this dissertation. The term "scaling RL" is used broadly to include any technique that extends the range of applications to which RL architectures can be applied in practice.

5.1 Previous Research on Scaling Reinforcement Learning

This section presents a review of research on scaling RL from the abstract perspective developed in Chapters 3 and 4. Specifically, the view of RL and DP algorithms as iterative relaxation algorithms allows us to focus on the properties of the update equation to the exclusion of all other detail. Five aspects of the prototypical update equation

$$V_{k+1}(x) = V_k(x) + \rho_k(x)(B_x(V_k) - V_k(x)),$$

are important for discussing the issue of scaling in lookuptable-based RL architectures. For the update equation, these five aspects are:

1. The "quality" of information provided by the backup B_x.

2. The order chosen in which the update equation is applied to the states x.

3. The fact that in each application of the update equation the value of only the predecessor state x (or state-action pair) is updated.

4. The fact that in each application of the update equation information is transferred only to a one-step neighbor (from the successor state to the predecessor state).[1]

5. The learning rate sequence $\{\rho_k(x)\}$.

The five aspects are also relevant for any algorithm that updates Q-values. The different approaches to scaling RL that are reviewed in this chapter are divided into five classes based on which of the five aspects listed above is addressed in each approach. Some of the algorithms reviewed in this section fall into more than one class.

5.1.1 Improving Backups

Researchers have studied at least three different ways of improving the quality of information provided by a backup: combining multiple estimators in a single backup, making the payoff function more informative, and learning models so that the full backup operator can be used instead of the sample backup operator. The DP and RL algorithms defined in Chapters 3 and 4 use one-step estimators in their backup operators. Sutton [106] extended the TD algorithm defined in Section 4.3.1 to a family of algorithms called TD(λ), where $0 \leq \lambda \leq 1$ is a scalar parameter. For $0 < \lambda < 1$, TD(λ) combines an infinite sequence of multi-step (n) estimators in a geometric sum (cf. Watkins [118]). In general, as one increases n in the n-step estimator, the value returned has a lower bias but may have a larger variance. Empirical results have shown that TD(λ) can outperform TD if λ is chosen carefully (Sutton [106]).

Whitehead [125] developed an approach he called *learning with an external critic* that assumes an external critic that knows the optimal policy in advance. The external critic occasionally rewards the agent when it executes an optimal action. This reward is in addition to the standard payoff function. The combined payoff function is more informative, and in fact it reduces the multi-stage MDT to a single-stage MDT. Whitehead demonstrated that this technique can greatly accelerate learning. Building RL architectures that are capable of using multiple sources of payoffs is an important research direction, particularly in light of the fact that human beings and animals can, and routinely do, use a rich variety of global and local evaluative feedback in addition to supervised training information.

Another technique for improving the quality of information returned by a backup is to use system identification methods to estimate a model on-line and to use algorithms that employ full backups on the estimated model. Several researchers have used such indirect control methods to solve RL tasks (e.g., Moore [78]) and it is also the usual method in classical Markov chain control (e.g., Sato *et al.* [91]). However as argued in Chapter 4 (Section 4.4.1), architectures that start by doing sample backups and switch to doing full backups using the estimated model at an appropriate time may be able to exploit the bias-variance trade off between sample and adaptive full backups.

---

[1] The TD(λ) algorithm is an exception.

5.1.2 The Order In Which The States Are Updated

The order in which the state-update equation is applied to the state space of an MDT does not effect asymptotic convergence of RL and asynchronous DP algorithms as long as every state (state-action pair) is updated infinitely often. The order in which the states are updated can, however, dramatically affect the *rate of* convergence. There are two separate cases to consider here: the model-based case and the model-free case. In the model-based case the agent can apply the update equation to an arbitrary state in the environment model. On the other hand, in the model-free case the agent is constrained to apply the update equation to the current state of the environment. At best, the agent can influence the future states of the environment by the actions it executes — the actual state trajectory will however also depend on chance and on the unknown transition probability distributions for the selected actions. These constraints make it difficult to optimize the order in which states should be visited in the model-free case.

5.1.2.1 Model-Free

Another complicating factor in the model-free case is that the agent is not only trying to approximate the optimal value function but also simultaneously controlling the real environment. The agent has to tradeoff the need to select non-greedy actions that could accelerate learning with the need to exploit the greedy solution to maximize current payoffs. Therefore, the agent cannot choose actions from the sole perspective of learning the value function in as few updates as possible. This dilemma is called the exploration versus exploitation tradeoff (Barto *et al.* [10], Thrun [113, 114]). The exploration strategy adopted by an agent determines the order in which the states are visited and updated.

Exploration:

A simple strategy adopted by many researchers is to execute a non-stationary and probabilistic policy defined by the Gibbs distribution over the Q-values (Watkins [118], Sutton [107]). The probability of executing action a in state x at time step t is: $P(a|x,t) = \frac{e^{\Gamma_t Q_t(x,a)}}{\sum_{a' \in A} e^{\Gamma_t Q_t(x,a')}}$, where Γ_t is the temperature at time index t. The algorithm starts with a low temperature and gradually increases the temperature over time. The ratio of the number of exploration steps to the number of exploitation steps is high in the beginning and falls off with increasing temperature. This author has found that the above simple strategy can be improved by implementing a version in which the temperature schedule is not preset, but adapted on-line based on the agent's experience.

Another simple, yet effective, approach in the model-free case has been to use optimistic initial value functions (Sutton [107], Kaelbling [60]). In such a case, parts of the state space that have not been visited will have higher values than those that have been visited often. The greedy policy will then automatically explore unvisited regions of the state space. Barto and Singh [12] developed an algorithm that keeps a frequency count of how often each action is executed in each state (see also Sato *et al.* [91]). If an action is neglected for too long its execution probability is increased. The effect is to ensure that every action is executed infinitely often in each state for ergodic MDTs. Kaelbling [60] has developed an algorithm based on interval-estimation techniques that maintains confidence intervals for all actions. The action with the highest upper bound on expected payoff is selected for execution.

Bias in Policy Selection:

Some model-free RL architectures control the order in which states are updated by using task-specific knowledge to bias the policy selection in such a way as to direct the state trajectories into a useful and informative part of the state space. Clouse and Utgoff [28] have developed an architecture in which a human expert monitors the performance of the RL agent. If the agent is performing badly, the expert replaces the agent's action by an optimal action. They demonstrated that if the human expert carefully selects the states in which to offer advice to the agent, very little supervised advice may be needed to improve dramatically the learning speed (see, also Utgoff and Clouse [116]).

Another technique for biasing the policy selection mechanism is to use a nominal controller that implements the best initial guess of the optimal control policy. Exploration can then be confined to small perturbations around the nominal trajectory thereby reducing the number of state-updates performed in regions of the state space that are unlikely to be part of the optimal solution. Whitehead [125] has developed an architecture called *learning by watching* where an agent observes the behavior of an expert agent and shares its experiences. The learning agent does state-updates on the states experienced by the expert. Lin's [66] architecture stores the experience of the agent from the start to the goal state and performs repeated state-updates on the stored states. The above algorithms focus state-updates on parts of the state space that are likely to be a part of the optimal solution.

5.1.2.2 Model-Based

There are two subcases of the model-based RL case: *a*) the agent estimates a model on-line, and *b*) the agent is provided with a complete and accurate model of the environment. In subcase *a*, the the agent must explore the real environment to construct the model efficiently and that can conflict with exploitation, raising a different kind of exploration versus exploitation tradeoff. Schmidhuber [93] and Thrun and Moller [114] have developed methods that explicitly estimate the accuracy of the learned model and execute actions taking the agent to those parts of the state space where the model has a low accuracy.

In both subcases *a* and *b* there is still the question of exploration, only unlike the model-free case the agent is not constrained by the dynamics of the environment. Moore and Atkeson [79] and Peng and Williams [82] independently developed an algorithm that estimates an inverse model and uses it to maintain a priority queue of states. The states in the priority queue are ordered by the estimated magnitude by which their value would change if the update equation were applied to those states. The agent uses whatever time it has between executing any two consecutive actions in the real environment to update the values of as many states as it can by using simulated experience from the estimated model. This architecture can significantly outperform conventional RL architectures because it focuses the updates on states where they have the most effect.

5.1.3 Structural Generalization

The third group of approaches for scaling RL address the structural generalization problem, i.e., the problem of generalizing the value estimates across the state space. Such approaches use the information derived in a single backup to update the value of a set of states that are "similar in structure" to the predecessor state. Several researchers have achieved this by using function approximation methods other than lookup-tables, e.g., neural networks, to store and maintain the value function (cf. Chapter 6, see, also Lin [66]). Supervised learning methods such as the backpropagation algorithm (Rumelhart *et al.* [88]) can be used to update the parameters of the function approximator. Because these function approximators have fewer free parameters than do lookup-tables, each application of the update equation affects the value of a set of states. The set will depend on the generalization bias of the function approximator. The main disadvantage of this approach is that there is no theory for picking a function approximator with the right generalization bias for the RL task at hand. The wrong generalization bias can prevent convergence to the optimal value function and lead to sub-optimal solutions.

Other researchers are exploring techniques for generalizing the values in state space in a way that takes the dynamics of the environment and the payoff function into account. Yee *et al.* [129] developed a method that uses a symbolic domain theory to do structural generalization. After every training episode, a form of explanation-based generalization (Mitchell *et al.* [76]) is used to determine a set of predecessor states that should have the same value. Any errors in generalization are handled via a mechanism for storing exceptions to concepts. Mitchell and Thrun [75] extended this approach to situations where a symbolic domain theory may be unavailable. They use on-line learning experiences to estimate a neural-network based environment model. Network inversion techniques are used to determine the slope of the value function in a local region around the predecessor state. The value function network is then trained to implement that slope. Both Yee *et al.* and Mitchell and Thrun demonstrate greatly accelerated learning.

Other researchers have developed approaches that start with a coarse resolution model of the environment and selectively increase the resolution where it is needed. Moore [78] uses the *trie* data structure to store a coarse environment model. A DP method is used to find a good solution to the abstract problem defined by the coarse model. The trajectory that the agent would follow if that solution were implemented is determined and the states around that trajectory are further subdivided. This two-step process is repeated until the agent finally finds a good solution for the underlying physical problem. Moore showed that his approach develops a model that has a high resolution around the optimal trajectory in state space and a coarse resolution elsewhere (see, also Yee [128]). Chapman and Kaelbling [26] developed an algorithm that builds a tree structured Q-table. Each node splits on one bit of the state representation, and that bit is chosen based on its relevance in predicting short-term and long-term payoffs. Relevance is measured via statistical tests. Both the above architectures can lead to a much smaller state space and therefore to great savings in the number of updates needed for deriving a good approximation to the optimal policy.

5.1.4 Temporal Generalization

The depth of an MDT can be defined as the average over the set of start states of the expected number of actions executed to reach a goal state when the agent follows an optimal policy. For problems that have a high "depth", conventional RL architectures can take too long to propagate information from the goal states backwards to states that are far from the goal states. The fourth scaling issue concerns the temporal generalization problem, i.e., the problem of doing backups that transfer information among states that are not one-step neighbors. Few researchers have developed RL architectures that explicitly address the need to do temporal generalization. Barto *et al.*'s [13] used eligibility traces to update states visited many steps in the past. Sutton [106] developed a family of algorithms called TD(λ) that use multi-step backup operators. Watkins [118] defined a multi-step version of Q-learning. All these approaches are model-free.

Dayan [34] and Sutton and Pinette [111] developed an algorithm that tackles the temporal generalization problem in policy evaluation problems by changing the agent's state representation. It learns to represent the i^{th} element of the state set by the i^{th} row of the matrix $(I - \gamma[P]^{\pi})^{-1}$. With this new representation of the state set, the value of a state under policy π is equal to the inner product of the vector representation of that state with the payoff vector because $V^{\pi} = (I - \gamma[P]^{\pi})^{-1}R^{\pi}$. This achieves "perfect" temporal abstraction because the depth of the new problem defined on the altered state representations is always one. Unfortunately, the above method is limited to the policy evaluation problem.

Several of the architectures that do structural generalization by building coarse environment models also implicitly do some temporal abstraction. One of the contributions of Chapter 7 of this dissertation is to separate the distinct but often confounded issues of structural and temporal abstraction. Chapter 7 presents a hierarchical architecture that achieves temporal abstraction without doing any structural abstraction.

5.1.5 Learning Rates

The fifth aspect of the update equation that affects rate of convergence is the learning rate sequence $\{\rho_k(x)\}$ for each state x. The only necessary conditions on the learning rate sequence for RL algorithms are those derived from the stochastic approximation theory, namely that $\forall x$, $\sum_k \rho_k(x) = \infty$, and that $\sum_k \rho_k^2(x) < \infty$ (cf. Section 4.3). Most researchers use $\rho_k(x) = f(\frac{1}{n_k(x)})$, where $f(\cdot)$ is a linear function, and $n_k(x)$ is the number of times state x got updated before the k^{th} iteration. Most researchers optimize the parameters of the function f by trial and error. Another approach would be to use the experience of the agent to adapt the learning rate on-line. Kesten's [61] method for accelerating stochastic approximation can be adapted to do on-line adaptation of individual learning rates. For each state, the sign of the last change in its value is stored. For each state, the learning rate is kept constant until the sign of the change in that state's value, flips, say at iteration i, at which point the learning rate is dropped to $f(\frac{1}{n_i(x)})$.[2] This author's experience has been that Kesten's method accelerates the rate of convergence of RL algorithms, and that it removes some of the burden of the trial and error search for the best learning rate parameters.

5.1.6 Discussion

Only the approaches to scaling RL that fell into one of the five abstract categories stated at the beginning of this section were reviewed above. By necessity, this chapter does not do justice to the algorithms mentioned, not is it an exhaustive survey of all the different approaches to scaling RL. There are other general approaches to scaling RL outside the categories considered here, e.g., using "shaping" techniques to guide the agent through a sequence of tasks of increasing difficulty culminating with the desired task (e.g., Gullapalli [48]), and hierarchical and modular RL architectures that use the principle of divide and conquer. Some of these approaches are discussed in later chapters because they are more closely related to the RL architectures developed in this dissertation.

5.2 Preview of the Next Three Chapters

The research presented in Chapters 6, 7 and 8 is motivated by two related concerns: the need to accelerate learning in RL architectures, and the need to develop RL architectures for agents that have to learn to solve multiple control tasks. The effort to build more sophisticated learning agents for operating in complex environments will require handling multiple complex tasks/goals. While building multi-task agent architectures may introduce new hurdles, it also offers the opportunity to use knowledge acquired while learning to solve the early tasks to accelerate the learning of solutions for later tasks. It is the thesis of the next three chapters that techniques allowing transfer of training across tasks will be indispensable for building sophisticated autonomous learning agents.

5.2.1 Transfer of Training Across Tasks

It is possible to build a RL agent that has to learn to solve multiple tasks by simply having a separate RL architecture learn the solution to each new task. However, given the fact that conventional RL architectures are too slow for many single complex tasks, it is unlikely that the "learn each task separately" architecture can learn multiple complex tasks fast enough to be of general use. Therefore, the ability to achieve transfer of training across tasks must play a crucial role in building useful multi-task

---

[2] Sutton [109] has adapted Kesten's method for adapting learning rates for individual parameters in a function approximator (see, also Jacobs [54]).

agent architectures based on RL. Multi-task agents can also achieve computational and monetary savings over multiple single-task agents simply by sharing hardware and software across tasks.

Transfer of training across tasks is different from the phenomenon of *generalization* as it is commonly studied in supervised learning tasks (Denker *et al.* [37]). Generalization refers to the ability of a learner to produce (hopefully correct) outputs when tested on inputs that are not part of the training set. *Typically* generalization is studied within a single task that can be thought of as a mapping from some input space to an output space. Transfer of training across multiple tasks refers to the ability of a learner to approximate the correct outputs for new tasks, i.e., new input-output mappings.[3] In the context of RL, transfer of training would enable an agent to use previous experience to better approximate optimal behavior in new RL tasks.

Building learning control architectures to achieve transfer of training across an arbitrary set of tasks is difficult, if not impossible. Consequently the approach taken in this dissertation is to focus on a constrained but useful class of RL tasks. All three modular architectures presented in the remainder of this dissertation transfer training from simple to complex tasks. Chapter 6 presents an architecture that uses the value functions of the simple tasks as building blocks for efficiently constructing value functions for more complex tasks. Chapter 7 uses the solutions of simple tasks to build abstract environment models. These abstract models can predict the consequences of executing multi-step actions and thereby achieve temporal generalization. Transfer of training is achieved by using these abstract models to learn the value functions for subsequent complex tasks. Finally, Chapter 8 uses the closed-loop policies found for the simpler tasks to redefine the actions for subsequent complex tasks. By suitably designing the simple tasks, the policy space for the complex tasks can be constrained to accelerate learning and to exclude "undesired" policies.

5.3 Conclusion

Most, if not all, the research work of other authors reviewed here was developed in the context of agents that have to learn to solve single tasks. The issue of achieving transfer of training is not even present in the single task context. In the multi-task context, transfer of training is orthogonal to the five abstract dimensions along which prior research on scaling RL was discussed. Therefore, many of the ideas behind the reviewed algorithms can also be used in multi-task agent architectures to derive the same benefits that they provide in the single task context.

[3]While "static" generalization as it is commonly studied can certainly be a mechanism for achieving transfer of training across tasks, the focus in this dissertation is on achieving transfer by constructing solutions to new tasks by "stringing together in time" pieces of solutions from other tasks.

CHAPTER 6

COMPOSITIONAL LEARNING

The subject of this chapter is a modular, multi-task, RL architecture that accelerates learning by achieving transfer of training across a set of hierarchically structured tasks. The architecture achieves transfer of training by constructing the value function for a complex task by computationally efficient modifications to the value functions of tasks that are lower in the hierarchy. The material presented in this chapter is also published in Singh [99, 96, 95].

6.1 Compositionally-Structured Markovian Decision Tasks

Much of everyday human activity involves multi-stage decision tasks that have *compositional* structure, i.e., complex tasks are built up in a systematic way from simpler subtasks. As an example consider the routine task of driving to work. It could involve many simpler tasks such as opening a door, walking down the stairs, walking to the car, opening the car door, driving, and opening the office door. Notice that the choice of "simpler" subtasks above is somewhat arbitrary because each of these subtasks can be decomposed into even simpler subtasks. However, the chosen subtasks are at a level of abstraction that suffices to illustrate that many of them are part of other complex tasks, such as driving to the grocery store, and going to the doctor's office.

Clearly we do not learn to solve the task of opening a door separately for all the above complex tasks. We are somehow able to piece together the solution to a new complex task from *parts* of solutions to other complex tasks, and perhaps by learning to solve some additional novel subtasks. Compositionally-structured tasks offer a precise and well-defined framework for studying the possibility of sharing knowledge across tasks that have common subtasks. While there may be many other interesting classes of tasks for studying transfer of training, achieving transfer of training across an arbitrary set of tasks is difficult, if not impossible. This chapter deals with the challenge of autonomously achieving transfer of training across a set of MDTs that have compositional structure.

To formulate the problem abstractly consider an agent that has to solve a set of simple and complex MDTs. Suppose that there are n simple MDTs labeled $T_1, T_2, \ldots, T_n$, that are called *elemental* MDTs because they are not decomposed into simpler MDTs. Further, suppose that there are m complex MDTs labeled $C_{n+1}, C_{n+2}, \ldots, C_{n+m}$, that are called *composite* MDTs because they are produced by temporally concatenating a number of elemental MDTs. For example, $C_j = [T(j,1)T(j,2)\cdots T(j,k)]$, is composite MDT j made up of k elemental MDTs that have to be performed in the order listed. For $1 \leq i \leq k$, subtask $T(j,i) \in \{T_1, T_2, \ldots, T_n\}$, is the i^{th} elemental MDT in the list for task C_j. Notice that the indices for composite tasks start at $n+1$. The sequence of elemental MDTs in a composite MDT will be called the *decomposition* of the composite MDT. Throughout this chapter it is assumed that the decomposition of a composite MDT is not made available to the learning agent.

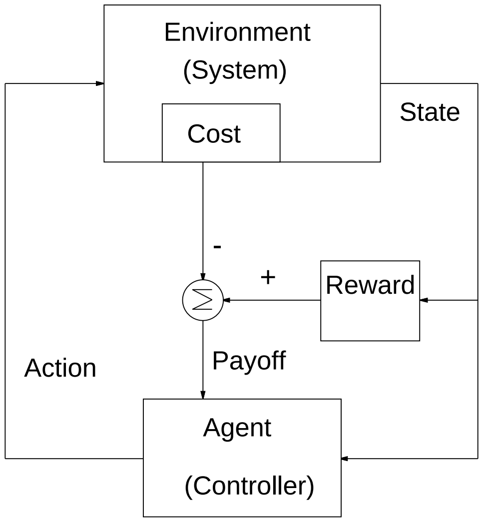

Figure 6.1 A single elemental MDT. This figure shows an agent interacting with an environment. The payoff function is divided into a cost function and the reward function.

6.1.1 Elemental and Composite Markovian Decision Tasks

In this chapter attention is restricted to the broad class of MDTs that have absorbing goal states, i.e., those requiring the agent to bring the environment to a desired final state. Figure 6.1 shows a block diagram representation of an elemental MDT. Note that the payoff function is assumed to be composed of two components: a cost function, c, where $c^a(x)$ is the cost of executing action a in state x, and a reward function r_j, where $r_j(x)$ is the reward associated with state x when executing task j. The payoff function for task j, $R_j^a(x) = E\{r_j(y) - c^a(x)\}$, where y is the state reached on executing action a in state x.

Several different elemental tasks can be defined in the same environment (Figure 6.2). Each elemental task has its own goal state. All elemental tasks are MDTs that have the same state set X, the same action set A, and the same transition probabilities P. The payoff function, however, can be different across the elemental tasks. It is assumed that the elemental tasks share the same cost function but have their own reward functions, i.e., the cost function is task independent, while the reward function is task dependent. As an example, consider a robot that has multiple goal locations in the same room — the energy or time cost of executing an action, say one-radius-north, is independent of whether the robot's goal is the door or the window.

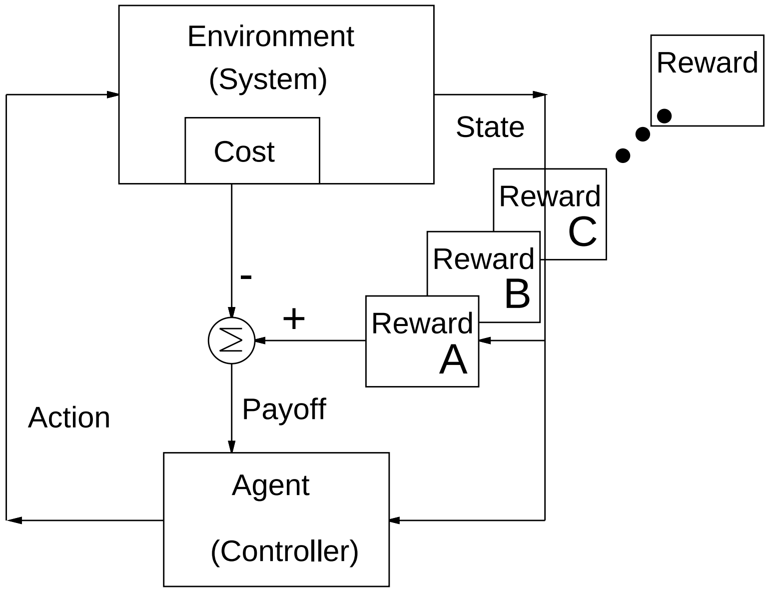

Figure 6.2 Multiple elemental MDTs defined in the same environment. Elemental MDTs, $A, B, C \ldots$, can be defined by simply switching in the associated reward function.

The reward however, say for entering state "next-to-door", will clearly depend on the goal.

A composite MDT is defined as an ordered sequence of elemental MDTs. The state set, X, of the elemental MDTs has to be augmented to define the state set for the composite MDTs because the payoff for a composite MDT is a function of which elemental task is being performed. The set X can be extended as follows: imagine a device that for each elemental task detects when the desired final state of that elemental task is visited for the first time and then remembers this fact. This device can be considered part of the learning system or equivalently as part of a new environment for the composite task. Formally, the new state set for a composite task, X', is formed by augmenting the elements of set X by n bits, one for each elemental task.[1] It is assumed that the number of elemental tasks is known in advance.

For each $x' \in X'$ the *projected state* $x \in X$ is defined as the state obtained by removing the augmenting bits from x'. The transition probabilities and the cost function for a composite task are defined by assigning to each $x' \in X'$ and $a \in A$ the transition probabilities and cost assigned to the projected state $x \in X$ and for the action[2] a. The reward function for composite task C_j, r_j, is defined as follows: $r_j(x') \geq 0$ if the projected state x is the final state of some elemental task in the

[1]The theory developed in this proposal does not depend on the particular extension of X to X' chosen in this chapter, as long as an appropriate mapping between the elements of X' and the elements of X can be defined.

[2]Deriving the transition probabilities for the composite tasks is a little more involved than that because care has to be taken in assigning the augmented bits to the next state.

decomposition of C_j, say task T_i, and if the augmenting bits of x' corresponding to elemental tasks coming earlier in the decomposition and including subtask T_i in the decomposition of C_j are one, and if the rest of the augmenting bits are zero; $r_j(x') = 0$ everywhere else.

6.1.2 Task Formulation

Consider a set of undiscounted ($\gamma = 1$) MDTs that have compositional structure and satisfy the following conditions:
(A1) Each elemental MDT has a single desired goal state.
(A2) For all elemental and composite MDTs, the optimal value function is finite for all states.
(A3) The cost associated with each state-action pair is independent of the task being accomplished.
(A4) For each elemental task T_i the reward function r_i is zero for all states except the desired final state for that task. For each composite task C_j, the reward function r_j is zero for all states except possibly for the final states of the elemental tasks in its decomposition.

The learning agent has to learn to solve a number of elemental and composite tasks in its environment. At any given time, the task faced by the agent is determined by a device that can be considered to be part of the environment or to be a part of the agent. As an example, consider a robot in a house-like environment. If the device is considered to be part of the environment it provides a task command to the agent, e.g., a human could command the agent to fetch water, or fetch food. On the other hand, if the device is part of the agent it provides a context or internal state for the agent. Such a case would arise if the agent has a "need" for food or water depending on whether it is thirsty or hungry. The above two views are formally equivalent; the crucial property is that they determine the reward function but do not affect the transition probabilities in the environment.

The representation used for the task command determines the difficulty of solving the *composition* problem, that is of learning which elemental subtasks compose a given composite task. At one extreme, using task-command representations that encode the decomposition of composite tasks in their representation can reduce the problem of solving the composition problem to that of "decoding" the task command. At the other extreme, unstructured task command representations force the system to learn the composition of each composite task separately. In this chapter unit-basis vector representations are used for task commands, thereby focusing on the issue of transfer of training by "sharing" solutions of elemental tasks across multiple composite tasks, and ignoring the possibilities that could arise from using richer task-command representations.

If the task command is considered to be part of the state description, the entire set of MDTs faced by an agent becomes one large unstructured MDT with a state set larger than any one MDT. While an optimal policy for the unstructured MDT can be learned by using Q-learning or any other DP-based learning algorithm, the structure inherent in the set of compositionally structured MDTs allows a more efficient solution, namely that of compositional learning.

6.2 Compositional Learning

Compositional learning involves solving a composite task by learning to compose the solutions of the elemental tasks in its decomposition. The technique presented in this chapter is to use a *modular* RL architecture that accomplishes the following:

1. Learns the solution to each elemental task in a separate RL module.
2. Learns which elemental-task modules to compose in what order to form solutions to composite tasks.

It should be emphasized that in the framework presented above the agent does not face the most general task decomposition problem. In particular, the agent does not face the difficult task of automatically discovering useful subtasks of arbitrary complex tasks. Instead the agent faces the composition problem whereby it has to discover which particular ordered subset of the set of elemental subtasks can be composed together to solve a composite task. Nevertheless, given the short-term, evaluative nature of the payoff from the environment (often the agent receives informative payoff only at the successful completion of the composite task), solving the composition problem remains a formidable task.

6.2.1 Compositional Q-learning

Compositional Q-learning (CQ-learning) is a method for computing the Q-values of a composite task from the Q-values of the elemental tasks in its decomposition. CQ-learning is advantageous because it takes significantly less effort than learning the Q-values of a composite task from scratch. The savings in computational effort arise from the special relationship that exists between the Q-values of a composite task and the Q-values of the elemental tasks in its decomposition. Let $Q^{T_i}(x, a)$ be the Q-value of state-action pair $(x, a) \in (X \times A)$ for elemental task T_i, and let $Q^{C_j}_{T_i}(x', a)$ be the Q-value of $(x', a) \in (X' \times A)$, for task T_i when performed as part of the composite task $C_j = [T(j, 1) \cdots T(j, k)]$. Let $T(j, l) = T_i$. Note that the superscript on Q refers to the task and the subscript refers to the elemental task currently being performed. The absence of a subscript implies that the task is elemental.

Proposition 2: For any elemental task T_i and for all composite tasks C_j containing elemental task T_i, the following holds for all $x' \in X'$ and $a \in A$:

$$Q^{C_j}_{T_i}(x', a) = Q^{T_i}(x, a) + K(C_j, l), \tag{6.1}$$

where $x \in X$ is the projected state (see Section 6.1.1), and $K(C_j, l)$ is a function of the composite task C_j and the position of elemental task T_i, l, in the decomposition of C_j.

A proof of Proposition 2 is given in Appendix D. Using Equation 6.1 to compute the Q-values of a composite task requires much less computation than computing them from scratch because $K(C_j, l)$ is independent of both the state and the action. Therefore, given solutions for the elemental tasks, learning the solution for a composite task with n elemental subtasks requires learning only the values of the function K

for the n different elemental subtasks. However, implementing Equation 6.1 requires knowledge of the decomposition of the composite tasks. In the next Section, a modular RL architecture is presented that simultaneously solves the composition problem for composite tasks and implements Equation 6.1.

6.2.1.1 CQ-L: The CQ-Learning Architecture

The compositional Q-learning architecture, or CQ-L, is a modification and extension of Jacobs et.al.'s associative Gaussian mixture model (GMM) architecture described in Jacobs *et al.* [56, 55]. GMM consists of several expert modules and a gating module that has an output for each expert module. When presented with training patterns (input-output pairs) from multiple tasks, the expert modules compete with each other to learn the training patterns, and this competition is mediated by the gating module. It has been shown empirically that when trained on a set containing input-output pairs from multiple tasks, different expert modules learn the different tasks, and that the gating module is able to activate the correct expert for each task. GMM models the training set as a mixture of associative parametrized Gaussians and learning is achieved by tuning the parameters by gradient descent in the log likelihood of generating the desired training patterns.

Only a brief and high level description of the details that are common to the GMM and CQ-L architectures is provided in this section[3]. In CQ-L, shown in Figure 6.3, the expert modules of the GMM architecture are replaced by Q-learning modules. The Q-modules receive state-action pairs as input. A bias module is added to learn the function K defined in Equation 6.1. The gating and bias modules (see Figure 6.3) receive as input the augmenting bits and the task command (see Section 6.1.1) used to encode the current task being performed by the architecture. The stochastic switch in Figure 6.3 uses the outputs of the gating module to select one Q-module at each time step. CQ-L's output is the output of the selected Q-module added to the output of the bias module.

At each time step, each Q-module competes with the other Q-modules to represent the value function of the current task at the current time step. The output of the selected Q-module at time $t+1$ is used to determine the *estimate* of the desired output at time t. Note that this is a crucial difference between GMM and CQ-L; in GMM the desired output is always available as part of the training set, while in CQ-L only an estimate of the desired output can be computed with a delay of one time step. The rest of the calculations are similar to those for the GMM architecture. Learning takes place by adjusting the parameters of each Q-module so as to reduce the error between its output and the estimated desired output in proportion to the probability of that Q-module having produced the desired output. Hence the Q-module whose output would have produced the least error is adjusted the most.

Simultaneously, the gating module is adjusted so that the *a priori* probability of selecting each Q-module becomes equal to the *a posteriori* probability of selecting that

[3]The interested reader is referred to the descriptions and derivations of GMM in Jacobs *et al.* [56], Nowlan [81], and Jordan and Jacobs [58].

Q-module, given the estimated desired output. Because of the different initial values of the free parameters in the different Q-modules, over time, different Q-modules start winning the competition for different elemental tasks, and the gating module learns to select the appropriate Q-module for each elemental task. For composite tasks, while performing a particular subtask, say T_i, the Q-module that has best learned task T_i will have smaller expected error than any other Q-module and will increasingly be selected by the gating module when that subtask is to be performed as part of the composite task. The bias module is also adjusted to reduce the error in the estimated Q-values.

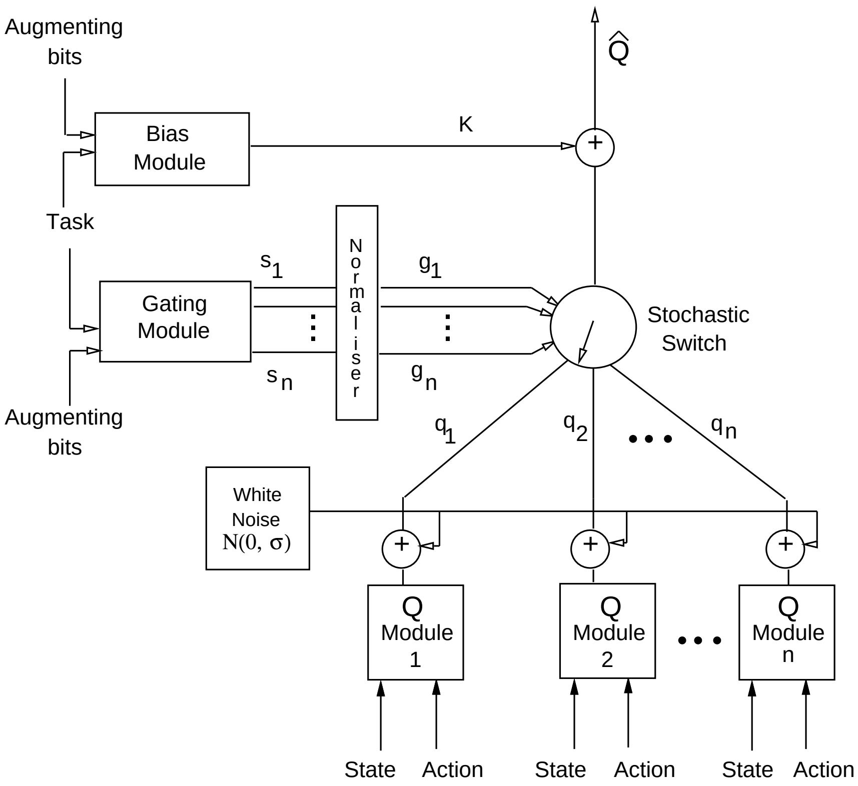

Figure 6.3 CQ-L: The CQ-Learning Architecture. This figure is adapted from Jacobs et al. [56]. The Q-modules learn the Q-values for elemental tasks. The gating module has an output for each Q-module and determines the probability of selecting a particular Q-module. The bias module learns the function K (see Equation 6.1).

6.2.1.2 Algorithmic details

At time step t, let the current state of the environment be x_t, let the output of the i^{th} Q-module be $q_i(t)$, and let the j^{th} output of the gating module be $s_j(t)$. The output of Q-module i, q_i, is treated as the mean of a Gaussian probability distribution with variance σ. The outputs of the gating module are normalized as follows: $g_j(t) = \frac{e^{s_j(t)}}{\sum_i e^{s_i(t)}}$, where $g_j(t)$ is the *prior* probability that Q-module j is selected at time step t by the stochastic switch. At each time step t the following steps are performed:

1. $\forall a \in A$, and $\forall i$, $q_i(x_t, a)$ is evaluated.

2. $\forall j$, $s_j(t)$ and then $g_j(t)$ is computed.

3. Using the probabilities defined by the function g, a single Q-module is selected. Let the label of the selected Q-module be $u(t)$.

4. Then an action to be executed in the real-world is selected using the Gibbs probability distribution: $P(a|x_t) = \frac{e^{\beta q_{u(t)}(x_t,a)}}{\sum_{a' \in A} e^{\beta q_{u(t)}(x_t,a')}}$. The action selected at time step t is denoted a_t.

5. The output of the bias module, $K(t)$ is computed.

6. The final output at time t is $Q(t) = q_{u(t)}(x_t, a_t) + K(t)$.

7. The estimate of the desired output at time $t-1$ is computed as follows: $D(t-1) = R(x_{t-1}, a_{t-1}) + Q(t)$. Note that $\gamma = 1$.

8. $D(t-1)$ is used to update the parameters of all the modules using Equations 6.2 developed below.

9. Go to Step 1 at time $t+1$.

The parameter β, used in Step 4, controls the probability of selecting a non-greedy action, and is increased over time so that eventually only the greedy actions are selected. The action chosen at time t is executed and the resulting next state is x_{t+1} and the payoff is $R(x_t, a_t)$. Note that the estimate of the desired output of the network at time $t-1$ only becomes available at time t. The probability that the Q-module i will have generated the desired output is

$$p_i(D(t-1)) \quad = \quad \frac{1}{N\sigma} e^{\frac{\|(D(t-1)-K(t-1))-q_i(t-1)\|^2}{2\sigma^2}},$$

where N is a normalizing constant. The *a posteriori* probability that Q-module i was selected at time $t-1$, given that the desired output is $D(t-1)$, is

$$p(i|D(t-1)) \quad = \quad \frac{g_i(t-1)p_i(D(t-1))}{\sum_j g_j(t-1)p_j(D(t-1))}.$$

The likelihood of producing the desired output, $L(D(t-1))$, is therefore given by $\sum_j g_j(t-1)p_j(D(t-1))$.

The objective of the architecture is to maximize the log likelihood of generating the desired Q-values for the current task. The partial derivative of the log likelihood with respect to the output of the Q-module j is

$$\frac{\partial \log L(D(t-1))}{\partial q_j(t-1)} = \frac{1}{\sigma^2} p(j|D(t-1))((D(t-1) - K(t-1)) - q_j(t-1)).$$

The partial derivative of the log likelihood with respect to the i^{th} output of the gating module simplifies to

$$\frac{\partial \log L(D(t-1))}{\partial s_i(t-1)} = (p(i|D(t-1)) - g_i(t-1)).$$

Using the above results, at time step t the update rules for Q-module j, the i^{th} output of the gating module, and the bias module[4] respectively are:

$$\begin{aligned} \Delta q_j(t) &= \alpha_Q \frac{\partial \log L(D(t-1))}{\partial q_j(t-1)}, \\ \Delta s_i(t) &= \alpha_g \frac{\partial \log L(D(t-1))}{\partial s_i(t-1)}, \text{and} \\ \Delta K(t) &= \alpha_b (D(t-1) - Q(t-1)), \end{aligned} \tag{6.2}$$

where α_Q, α_b and α_g are learning rate parameters.

CQ-L was tested empirically on compositionally structured tasks from two separate navigation domains: a simple discrete gridworld domain, and a more realistic continuous image-based navigation domain.

6.3 Gridroom Navigation Tasks

The first set of simulation results are from a discrete navigation domain, called the *grid-room*, and shown in Figure 6.4. The grid-room is an 8×8 room with three special locations designated A, B and C. The robot is shown as a circle, and the white squares represent fixed obstacles that the robot must avoid. In each state the robot has four actions: UP, DOWN, LEFT and RIGHT. Any action that would take the robot into an obstacle or boundary wall does not change the robot's location. There are three elemental tasks: "visit A", "visit B", and "visit C", labeled T_1, T_2 and T_3 respectively. Three composite tasks, C_1, C_2, and C_3 were constructed by temporally concatenating some subset of the elemental tasks (see Table 6.1).

The six tasks, along with their labels, are described in Table 6.1 and illustrated in Figure 6.5. For all $x \in X \cup X'$ and $a \in A$, $c^a(x) = -0.05$. The reward function is defined as follows: $r_i(x) = 1.0$, if $x \in X$ is the desired final state of elemental task T_i, or if $x \in X'$ is the final state of composite task C_i, and $r_i(x) = 0.0$ in all other states. Thus, for composite tasks no intermediate payoff was provided for successful completion of elemental subtasks. It is to be emphasized that the tasks defined in Table 6.1 are optimal control tasks and the optimal solutions define shortest paths through the sequence of special intermediate states to the final state.

[4]This assumes that the bias module is minimizing a mean square error criteria.

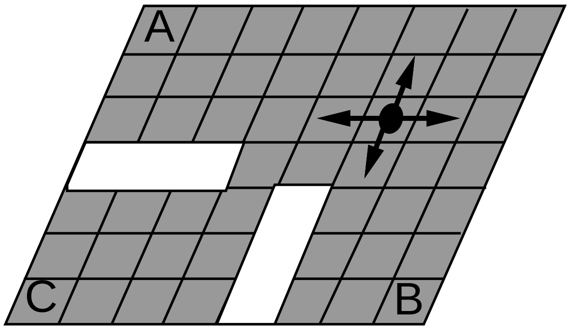

Figure 6.4 The Grid Room. The room is an 8×8 grid with three desired final locations designated A, B and C. The white squares represent obstacles. The robot is shown as a circle and has four actions available: UP, DOWN, RIGHT, and LEFT.

Table 6.1 Tasks. Tasks T_1, T_2, and T_3 are elemental tasks; tasks C_1, C_2, and C_3 are composite tasks. The last column describes the compositional structure of the tasks.

Label	*Command*	*Description*	*Decomposition*
T_1	000001	visit A	T_1
T_2	000010	visit B	T_2
T_3	000100	visit C	T_3
C_1	001000	visit A and then C	T_1T_3
C_2	010000	visit B and then C	T_2T_3
C_3	100000	visit A, then B and then C	$T_1T_2T_3$

6.3.1 Simulation Results

In the simulations described below, the performance of CQ-L is compared to the performance of a "one-for-one" architecture that learns to solve each MDT separately. The one-for-one architecture cannot achieve any transfer of learning because it has a pre-assigned distinct Q-learning module for each task. Each module of the one-for-one architecture was provided with the augmented state.

6.3.1.1 Simulation 1: Learning Multiple Elemental MDTs

Both CQ-L and the one-for-one architecture were separately trained on the three elemental MDTs T_1, T_2, and T_3 until they could perform the three tasks optimally. Training proceeded in trials. For each trial the task and the starting location of the robot were chosen randomly. Each trial ended when the robot reached the desired final location for that task. Both CQ-L and the one-for-one architecture contained three Q-learning modules. Figure 6.6 shows the number of actions taken by the robot to get to the desired final state. Each data point is an average over 50 trials. The one-for-one architecture converged to an optimal policy faster than CQ-L because it

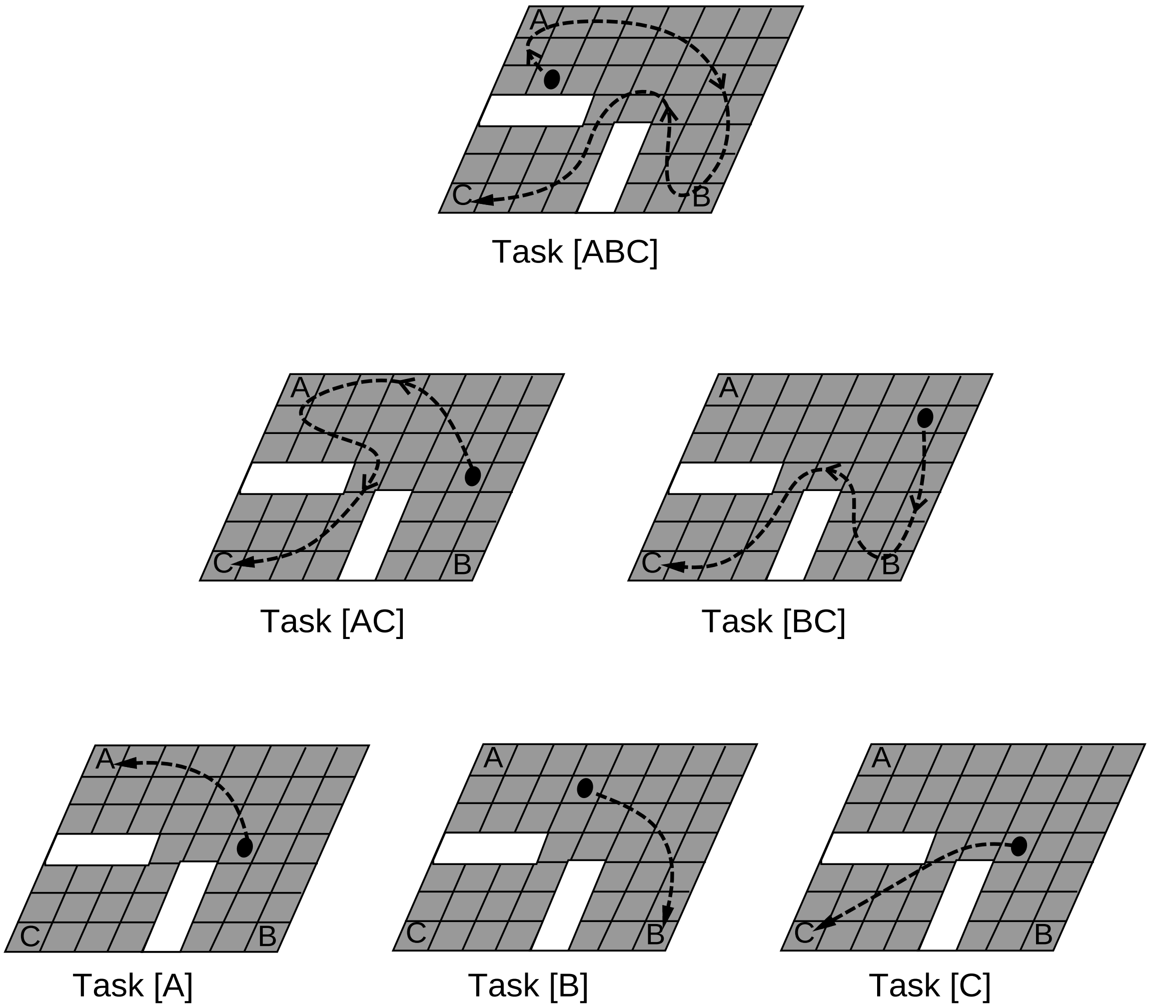

Figure 6.5 Gridroom Tasks: This figure shows the composition hierarchy. The lowest level shows the elemental tasks, the next level shows composite tasks of size two, and the uppermost level shows a composite task of size three.

took time for CQ-L's gating module's outputs to become approximately correct, at which point CQ-L learned rapidly.

Figures 6.7(**i**), 6.7(**ii**), and 6.7(**iii**) show the three normalized outputs of CQ-L's gating module for trials involving tasks T_1, T_2 and T_3 respectively. In each panel, the x-axis is the number of times the task associated with that panel occurred in the trial sequence. Each panel shows three curves, one for each of the three outputs of the gating module. For each curve the value plotted for each trial is the average prior probability in that trial for the associated Q-module. At the start of training each Q-module is selected with almost equal probability for all tasks. After training on approximately 100 trials a different Q-module is selected with probability one for each task. This simulation shows that CQ-L is able to partition its "across-trial" experience and learn to engage a distinct Q-module for each elemental task. This is similar in spirit to the simulations reported by Jacobs [55], except that he applies his

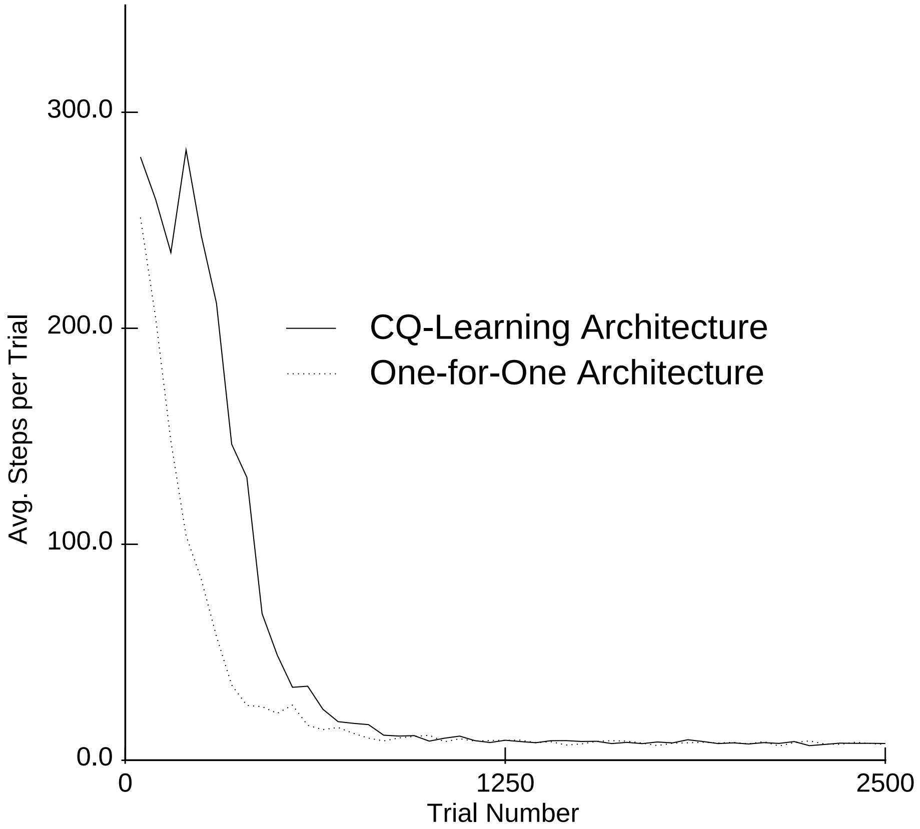

Figure 6.6 Learning Curve for Multiple Elemental tasks. Both CQ-L and one-for-one were trained on the intermixed trials of the three elemental tasks T_1, T_2, and T_3. Each data point is the average, taken over 50 trials, of the number of actions taken by the robot to get to the final state.

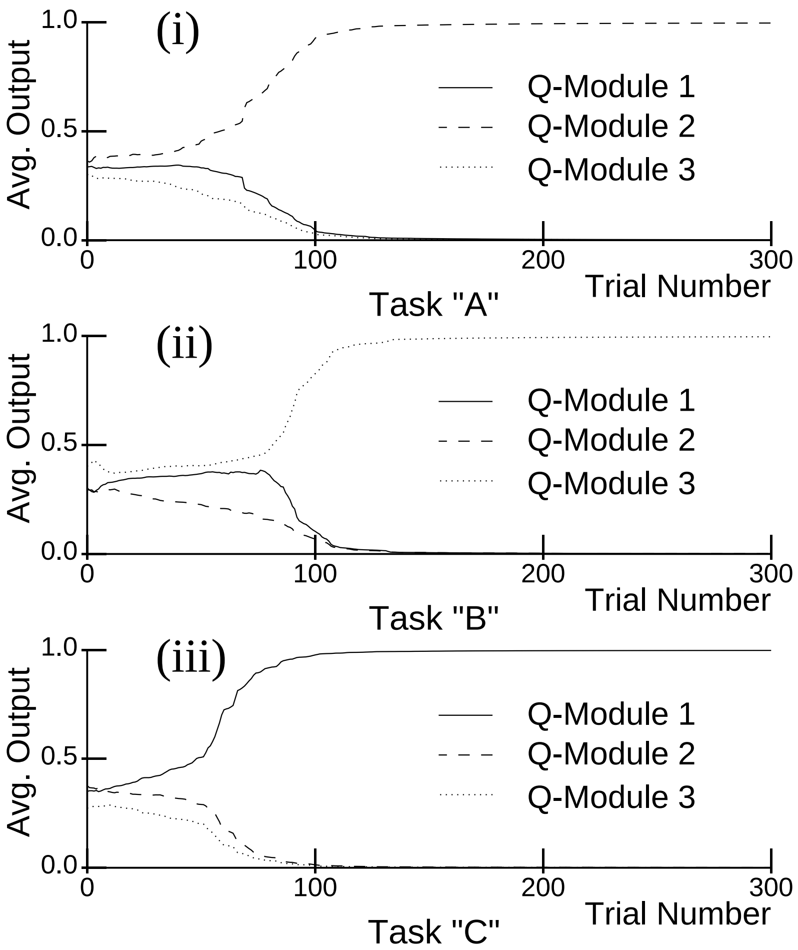

Figure 6.7 Both CQ-L and one-for-one were trained on intermixed trials of the three elemental tasks T_1, T_2 and T_3. This figure shows the prior probabilities of selecting the different Q-modules for each task. **(i)** Module Selection for Task T_1. The 3 normalized outputs of the gating module are shown averaged over each trial with task T_1. Initially the outputs were about 0.3 each, but as learning proceeded the gating module learned to select Q-module 2 for task T_1. **(ii)** Module Selection for Task T_2. Q-module 3 was selected. **(iii)** Module Selection for Task T_3. Q-module 1 was selected for task T_3.

architecture to supervised learning tasks. See Appendix D.1 for simulation details.

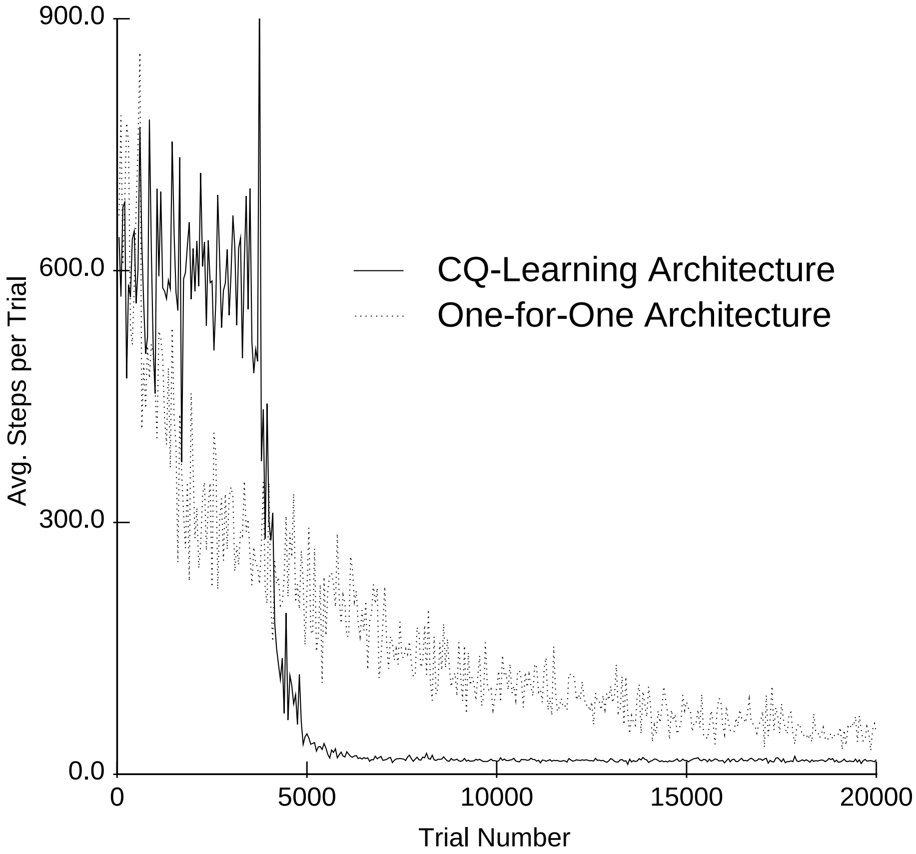

Figure 6.8 Learning Curve for a Set of Elemental and Composite Tasks. Both CQ-L and one-for-one were trained on intermixed trials of all the six tasks shown in Table 6.1. Each data point is the average over 50 trials of the time taken by the robot to reach the desired final state.

6.3.1.2 Simulation 2: Learning Elemental and Composite MDTs

Both CQ-L and the one-for-one architecture were separately trained on the six tasks T_1, T_2, T_3, C_1, C_2, and C_3 until they could perform the six tasks optimally. CQ-L contained four Q-modules, and the one-for-one architecture contained six Q-modules[5]. Training proceeded in trials. For each trial the task and the starting state of the robot were chosen randomly. Each trial ended when the robot reached the desired final state. Figure 6.8 shows the number of actions, averaged over 50 trials, taken by the robot to reach the desired final state. The one-for-one architecture performed better initially because it learned the three elemental tasks quickly, but learning

[5]In separate simulations CQ-L was given more Q-modules without any difference in training performance.

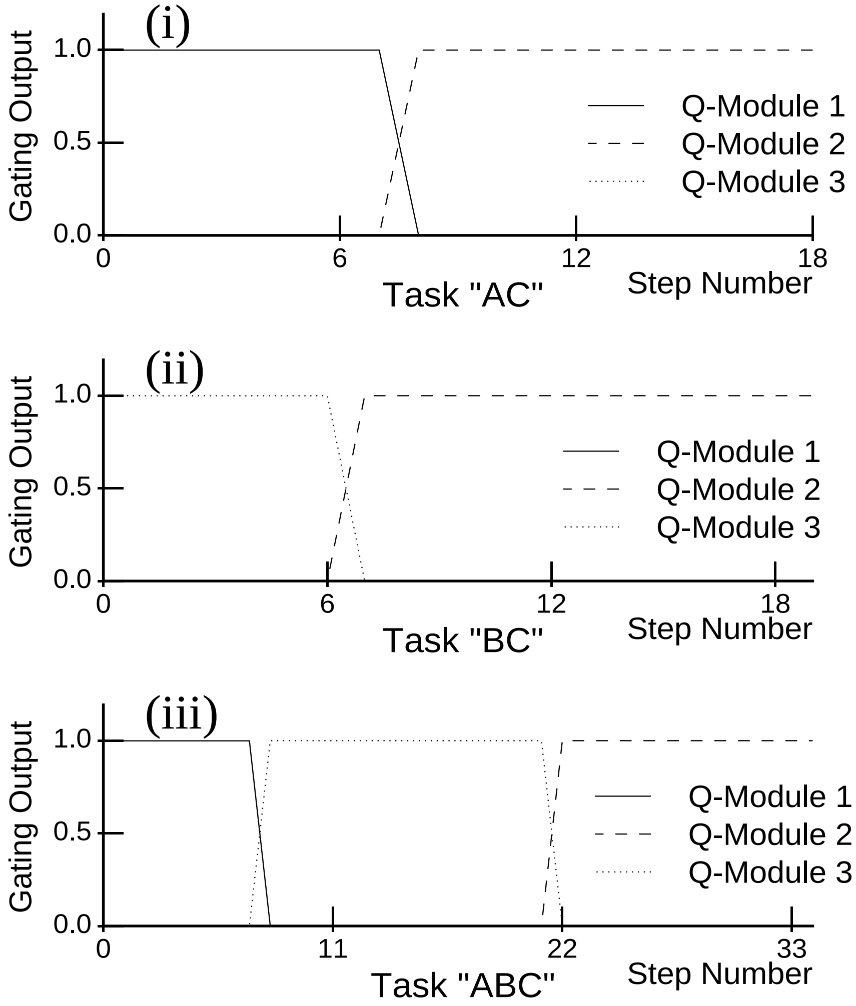

Figure 6.9 Both CQ-L and one-for-one were trained on intermixed trials of all the six tasks shown in Table 6.1. This figure shows the *intra*-trial selection of Q-modules for each composite task after learning. **(i)** Temporal Composition for Task C_1. After $10,000$ learning trials, the three outputs of the gating module during *one* trial of task C_1 are shown. Q-module 1 was turned on for the first seven actions to accomplish subtask T_1, and then Q-module 2 was turned on to accomplish subtask T_3. **(ii)** Temporal Composition for Task C_2. Q-module 3 was turned on for the first six actions to accomplish subtask T_2 and then Q-module 2 was turned on to accomplish task T_3. **(iii)** Temporal Composition for Task C_3. The three outputs of the gating module for one trial with task C_3 are shown. Q-modules 1, 3 and 2 were selected in that order to accomplish the composite task C_3.

the composite tasks took much longer due to the long action sequences required to accomplish the composite tasks. CQ-L performed worse initially, until the outputs of the gating module become approximately correct, at which point all six tasks were learned rapidly.

Figures 6.8(**i**), 6.8(**ii**), and 6.8(**iii**) respectively show the three normalized outputs of the gating module for three randomly chosen trials, one each for tasks C_1, C_2, and C_3. The trials shown were chosen after the robot had learned to do the tasks, specifically, after $10,000$ learning trials. The elemental tasks T_1, T_2, and T_3 respectively were learned by the Q-modules 1, 3 and 2. The graphs in each panel show that for each composite task the gating module learned to compose the outputs of the appropriate elemental Q-modules over time. This simulation shows that CQ-L is able to solve the composition problem for composite tasks, and that compositional learning, due to transfer of training across tasks, can be faster than learning each composite task separately. See Appendix D.1 for simulation details.

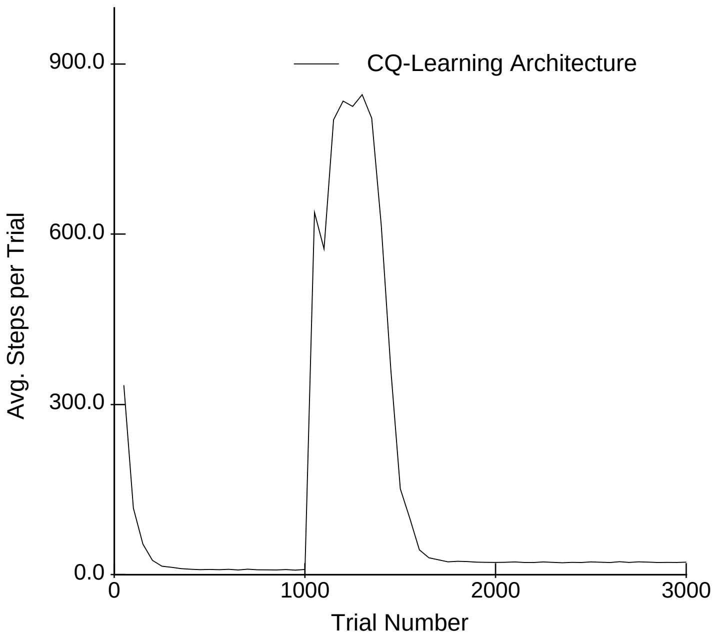

Figure 6.10 Shaping. The CQ-Learning architecture containing one Q-module was trained for 1000 trials on task T_3. Then another Q-module was added, and the architecture was trained to accomplish task C_2. The only task-dependent payoff for the task C_2 was on reaching the desired final location C. The graph shows the number of actions taken by the robot to reach the desired final state averaged over 50 trials.

6.3.1.3 Simulation 3: Shaping

One approach for training a robot on a composite task is to train the robot on multiple tasks: all the elemental tasks required in the composite task, and the composite task itself. Another approach is to train the robot on a succession of tasks, where each succeeding task requires some subset of the already learned elemental tasks, plus a new elemental task. This roughly corresponds to the "shaping" procedures used by psychologists to train animals to do complex motor tasks (see Skinner[104]).

A simple simulation to illustrate shaping was constructed by training CQ-L with one Q-module on one elemental task, T_3, for $1,000$ trials and then training on the composite task C_2. After the first $1,000$ trials, the learning was turned off for the first Q-module and a second Q-module was added for the composite task. Figure 6.10 shows the learning curve for task T_3 composed with the learning curve for task C_2. The number of actions taken by the robot to get to the desired final state, averaged over 50 trials, were plotted by concatenating the data points for the two tasks, T_3 and C_2. Figure 6.10 shows that the average number of actions required to reach the final state increases when the composite task was introduced, but eventually the gating module learned to decompose the task and the average decreased. The second Q-module learned the task T_2 without ever being explicitly exposed to it. See Appendix D.1 for simulation details.

6.3.2 Discussion

The simulations reported above show that CQ-L can construct the solution of a composite MDT by computationally inexpensive modifications to the solutions of its constituent elemental MDTs. It was shown that CQ-L can automatically learn the solutions to elemental tasks in separate modules, share the solutions to elemental tasks across multiple composite tasks, and can learn faster than a one-for-one architecture when trained on a set of compositionally structured MDTs.

Given a training set of composite and elemental MDTs, the sequence in which the learning agent receives training experiences on the different tasks determines the relative advantage of CQ-L over other architectures that learn the tasks separately. The simulation reported in Section 6.3.1.2 demonstrates that it is possible to train CQ-L on intermixed trials of elemental and composite tasks. Nevertheless, some training sequences on a set of tasks will result in faster learning of the set of tasks than other training sequences. The ability of CQ-L to scale well to complex sets of tasks is still dependent on the choice of the training sequence. Determining the *optimal training sequence* of subtasks is a meta-problem and is not considered in this dissertation.

6.4 Image Based Navigation Task

This section illustrates the utility of CQ-L on a set of simulated navigation tasks that are more "real-world" than the grid-room tasks simulated in the previous section. The new navigation tasks are in a navigational test bed that simulates a planar robot that can translate simultaneously and independently in both x and y directions (Figure 6.11). This testbed is similar to the one developed by Bachrach [4]. Figure 6.11 shows a display created by the navigation simulator. The bottom portion of the figure shows the robot's environment as seen from above. The circle represents the robot and the radial line inside the circle represents the robot's orientation. The

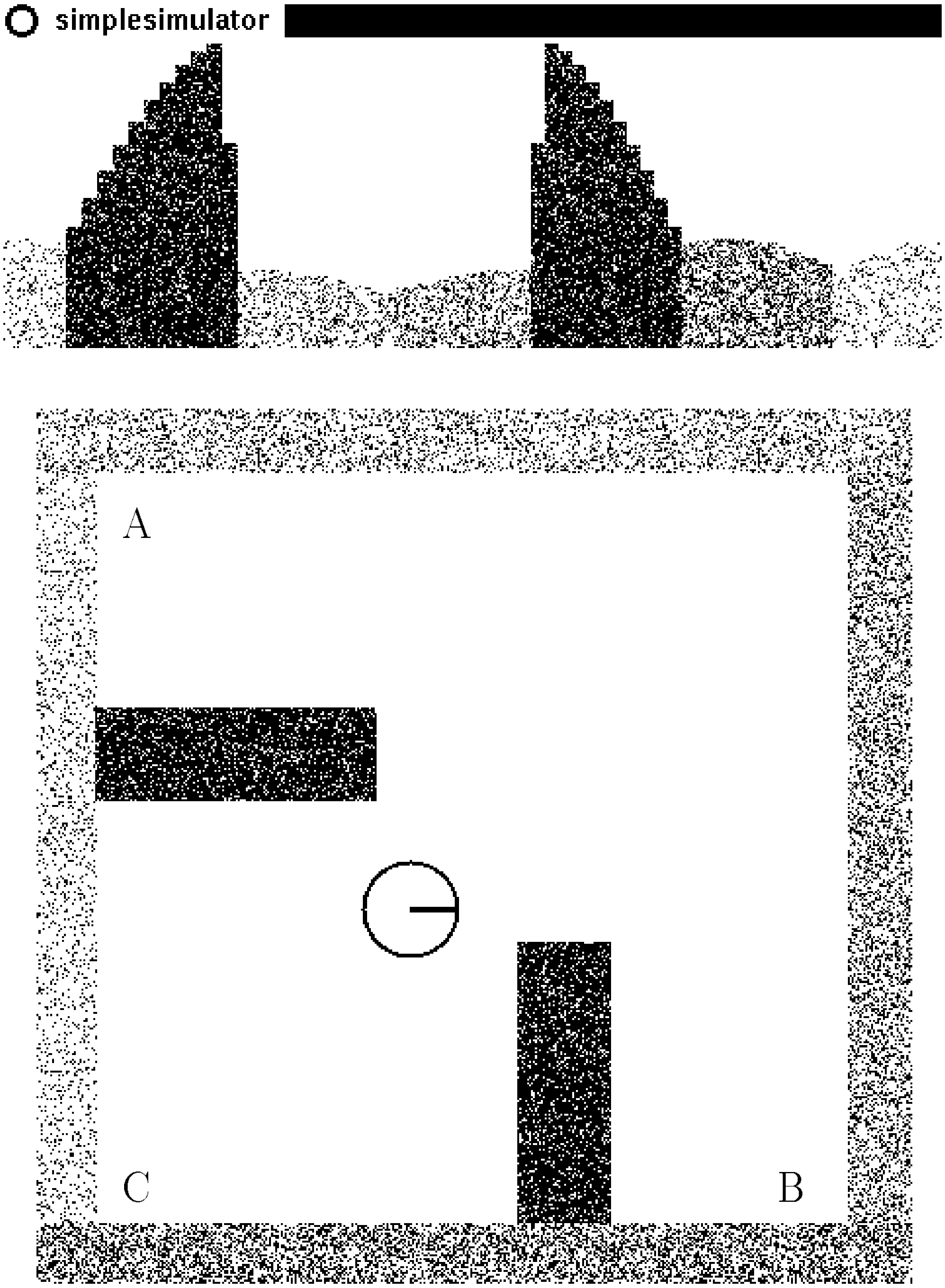

Figure 6.11 Simulated Image-Based Navigation Testbed. This lower panel shows a room with walls and obstacles that are painted in grayscale. The circular robot has 16 grayscale sensors and 16 distance sensors distributed evenly around its perimeter. The upper panel shows the robot's view. This testbed is identical to one developed by Bachrach [4]. See text for details.

walls of the simulated environment and the obstacles are shaded in grayscale. The robot can move one radius in any direction on each time step.

The robot has 8 distance sensors and 8 grayscale sensors evenly placed around its perimeter. These 16 values constitute the state vector. The upper panel shows the robot's view by drawing one column for each pair of distance and grayscale values. The central column corresponds to the sensors aligned with the robot's orientation. The grayscale value of each column corresponds to the reading of the grayscale sensor. The height of each column is the inverse of the reading of the corresponding distance sensor.

Tasks identical to the grid-room tasks can be defined in this environment. Three different goal locations, A, B, and C, are marked on the test bed. The set of tasks on which the robot is trained is shown in Table 6.1. The elemental tasks require the robot to go to the associated final location from a random starting location in minimum time. The composite tasks require the robot to go to the final location via the associated sequence of special locations. These navigation tasks are harder because the learning architecture has to deal with continuous state and actions and because the robot is not provided with a minimal state input.

6.4.1 CQ-L for Continuous States and Actions

To deal with the infinite states and actions, connectionist networks were used to implement the different modules in the CQ-L architecture. See Appendix F for a brief review on connectionist networks. Each Q-module was implemented as a feedforward connectionist network with a single hidden layer containing 128 radial basis units. The bias and gating modules were also feedforward networks with a single hidden layer containing sigmoid units. With continuous actions one cannot determine the Q-value for each action and therefore cannot use the Gibbs distribution to select actions. Instead, the action to be executed at time step t is computed by adding Gaussian noise to the estimated greedy action in the current state. The greedy action in state x_t is found by using a network inversion method (see Jordan and Rumelhart [59]). As learning proceeds the variance of the Gaussian noise is reduced over time so as to increase the likelihood of selecting the greedy action. Aside from this difference, the training algorithm for CQ-L is similar to the algorithm presented in Section 6.2.1.1. The weights of the networks are trained by using the backpropagation algorithm of Rumelhart *et al.* [88].

6.4.2 Simulation Results

As before, task commands were represented by standard unit basis vectors (Table 6.1), and thus the architecture could not "parse" the task command to solve the composition problem for a composite task. For all $x \in X \cup X'$ and $a \in A$, $c^a(x) = -0.05$. $r_i(x) = 1.0$ only if x is the desired final state of elemental task T_i, or if $x \in X'$ is the final state of composite task C_i; $r_i(x) = 0.0$ in all other states. Thus, for composite tasks no intermediate payoff for successful completion of subtasks was provided.

In the simulation described below, the performance of CQ-L is compared to the performance of a "one-for-one" architecture that implements the "learn-each-task-separately" strategy. The one-for-one architecture has a pre-assigned distinct network for each task, which prevents transfer of training. Each network of the one-for-one architecture was provided with the augmented state.

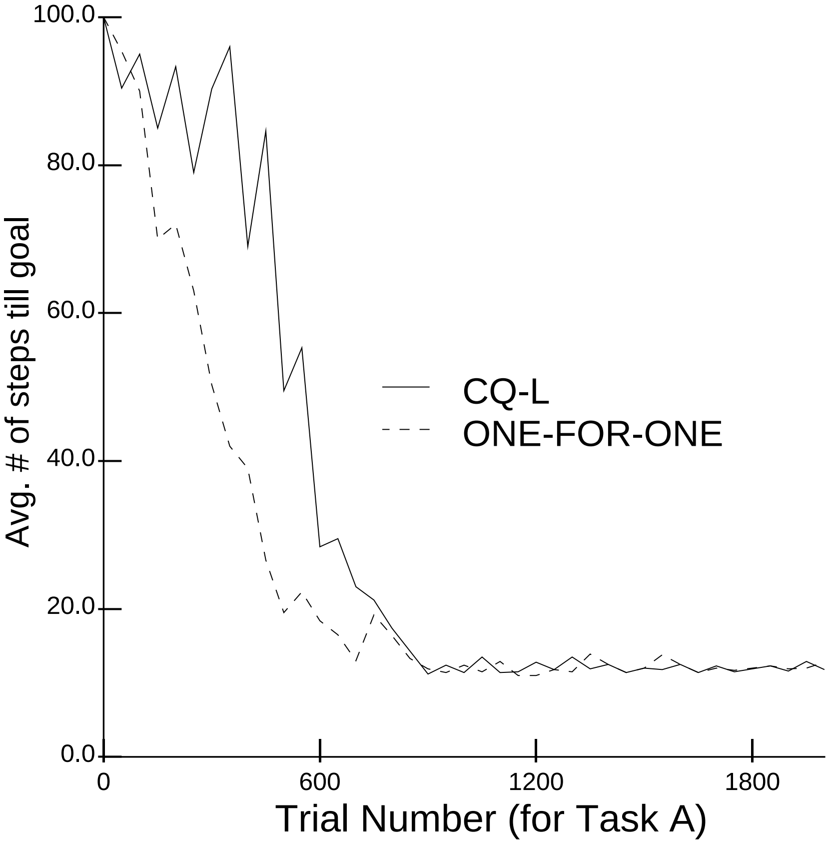

Figure 6.12 Learning Curve for task $[A]$. The horizontal axis shows the number of trials where the task was $[A]$, and the vertical axis shows the number of time steps to finish the trial. A trial finishes if the agent reaches the goal state or else if there is a time-out after 100 time steps.

Both CQ-L and the one-for-one architecture were separately trained on the six tasks T_1, T_2, T_3, C_1, C_2, and C_3 until they could perform the six tasks optimally. Training proceeded in trials. CQ-L contained three Q-networks, and the one-for-one architecture contained six Q-networks. For each trial the task and the starting location of the robot were chosen randomly. A trial ended when the robot reached the desired final location or when there was a time-out. The time-out period was 100 for the elemental tasks, 200 for C_1 and C_2, and 500 for task C_3[6]. The graphs in Figures 6.12, 6.13, and 6.14 show the number of actions executed per trial. The

[6]The time-out technique has been used by several authors to increase the likelihood of the robot's finding *any* path to the goal state with a random walk.

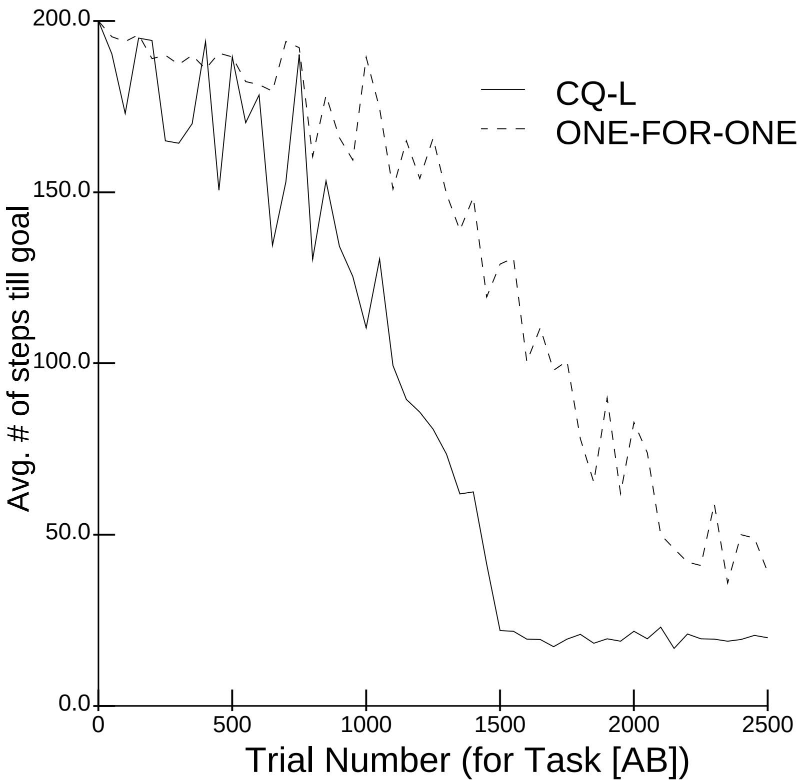

Figure 6.13 Learning Curves for task $[AB]$. The horizontal axis shows the number of trials where the task was $[AB]$, and the vertical axis shows the number of time steps to finish the trial. A trial finishes if the agent reaches state B after having traversed through state A, or else if there is a time-out after 200 time steps.

number of actions executed is equivalent to the time taken because each action is executed in a unit time step. Separate statistics were accumulated for each task.

Figure 6.12 graphs the performance of the two architectures on trials involving elemental task T_1. Not surprisingly, the one-for-one architecture learns more quickly than CQ-L because it does not have the overhead of figuring out which Q-network to train for task T_1. Figure 6.13 graphs the performance on task C_1 and shows that the CQ-L architecture is able to learn faster than the one-for-one architecture for a composite task containing just two elemental tasks. Figure 6.14 graphs the results for composite task C_3 and illustrates the main point of this chapter. The one-for-one architecture is unable to learn the task C_3, in fact it is unable to complete the task more than a couple of times due to the low probability of randomly performing the correct task sequence.

6.4.3 Discussion

As in Section 6.3, the simulations presented in Section 6.4.2 show that CQ-L is able to solve the composition problem for fairly complex composite tasks and that compositional learning, due to transfer of training across tasks, can be significantly

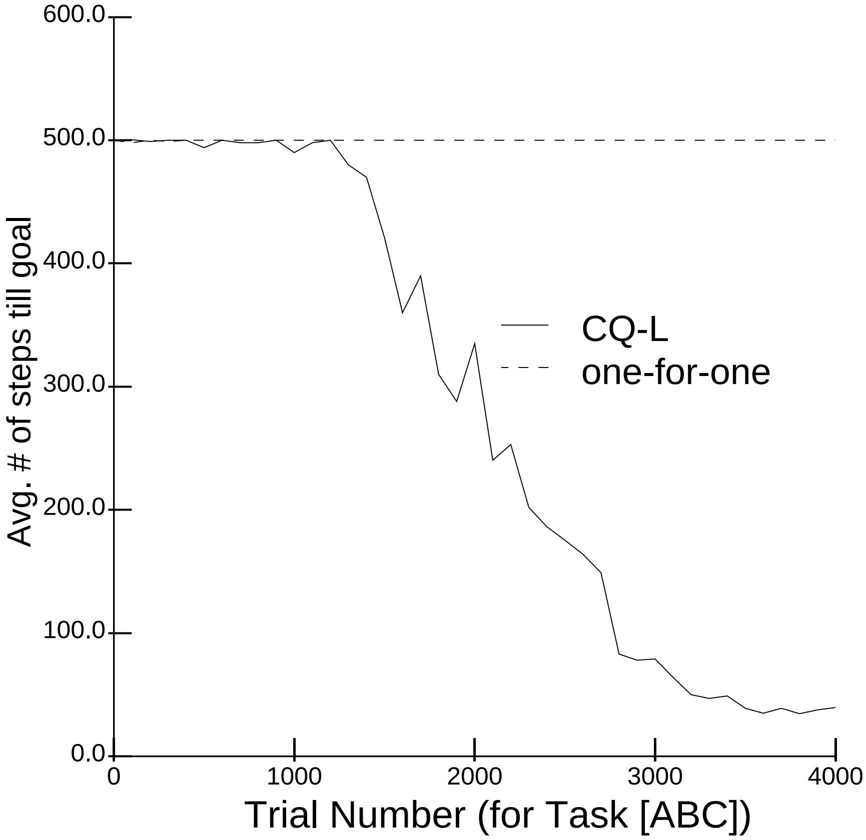

Figure 6.14 Learning Curves for task $[ABC]$. The horizontal axis shows the number of trials where the task was $[ABC]$, and the vertical axis shows the number of time steps to finish the trial. A trial finishes if the agent reaches state C after having traversed through states A and B in that order, or else if there is a time-out after 500 time steps.

faster than learning tasks separately. More importantly, CQ-L is able to learn to solve task $[ABC]$ which the more conventional application of Q-learning was unable to learn to solve. Although compositional Q-learning was illustrated using a set of navigational tasks, it is suitable for a number of different domains where multiple sequences from some set of elemental tasks need to be learned. CQ-L is a general mechanism whereby a "vocabulary" of elemental tasks can be learned in separate Q-modules, and arbitrary[7] temporal syntactic compositions of elemental tasks can be learned with the help of the bias and gating modules.

According to the definition used in this paper, composite tasks have only one decomposition and require the elemental tasks in their decomposition to be performed in a fixed order. A broader definition of a composite tasks allows it to be an unordered list of elemental tasks, or more generally, a disjunction of many ordered elemental task sequences. CQ-L should work with the broader definition for composite tasks without any modification because it should select the particular decomposition that is optimal with respect to its goal of maximizing expected returns. Further work is required to test this conjecture.

6.5 Related Work

An architecture similar to CQ-L is the subsumption architecture for autonomous intelligent agents (Brooks[22]), which is composed of several task-achieving modules along with precompiled switching circuitry that controls which module should be active at any time. In most implementations of the subsumption architecture both the task-achieving modules as well as the switching circuitry are hardwired by the agent designer. Maes and Brooks [69] showed how reinforcement learning can be used to learn the switching circuitry for a robot with hardwired task modules. Mahadevan and Connell [71], on the other hand, showed how Q-learning can be used to acquire behaviors that can then be controlled using a hardwired switching scheme. The simulations reported in this chapter show that at least for compositionally-structured MDTs, CQ-L combines the complementary objectives of Maes and Brooks's architecture with that of Mahadevan and Connell's architecture.

6.6 Conclusion

Learning to solve MDTs with large state sets is difficult due to the sparseness of the evaluative information and the low probability that a randomly selected sequence of actions will be optimal. Learning the long sequences of actions required to solve such tasks can be accelerated considerably if the agent has prior knowledge of useful subsequences. Such subsequences can be learned through experience in learning to solve other tasks. This chapter presented CQ-L, an architecture that combines the Q-learning algorithm of Watkins [118] and the modular architecture of Jacobs et al. [56] to achieve transfer of training by sharing the solutions of elemental tasks across multiple composite tasks.

---

[7] This assumes that the state representation is rich enough to distinguish repeated performances of the same elemental task.

CHAPTER 7

REINFORCEMENT LEARNING ON A HIERARCHY OF ENVIRONMENT MODELS

This chapter takes a familiar idea from artificial intelligence (AI), that of using an abstraction hierarchy to accelerate problem solving, and extends it to reinforcement learning (RL). Research on abstraction hierarchies in AI focused on deterministic domains and assumed that the problem solver was provided apriori with a hierarchy of state-space models of the problem environment (Sacerdoti [90]). The main contribution of this chapter is in extending the advantages of using a hierarchy of state-space models of the environment to RL agents that are embedded in stochastic environments and that learn a hierarchy of environment-models on-line. This chapter presents a RL agent architecture that uses the value functions learned for the simpler elemental tasks to build an abstract environment model. It is shown that doing backups in the abstract environment model can greatly accelerate the learning of value functions for composite tasks. Transfer of training is achieved by sharing the abstract environment model learned while solving the elemental tasks across multiple composite tasks. The material presented in this chapter is also published in Singh [98, 97].

7.1 Hierarchy of Environment Models

Building abstract models to speed up the process of learning the value function in RL tasks is not in itself a new idea. There is considerable work in building models that do structural abstraction, i.e., ignore structural details about the state that is perceived by the agent (see review in Chapter 5). In this chapter, however, the focus is on abstracting temporal detail by building an abstract model whose depth is much smaller than the depth of the real environment. As defined before, the depth of a RL problem is the average over the start states of the expected number of actions that are executed to get to a goal state when the agent follows an optimal policy (cf. Chapter 5).

For most problems there is a finest temporal grain at which the problem can be studied, determined usually by the highest sampling frequency and other hardware constraints. By limiting the backups to that fine a temporal scale, or alternatively to that high a temporal resolution, problems with large state sets and a large depth become intractable because of the many backups that have to be performed to learn the value function. There has been some research on overcoming the high temporal resolution problem without building a model by doing multi-step backups. For example, Sutton's TD(λ) algorithm can update the values of all states along a sampled trajectory based on the change in the value of the state at the head of the trajectory.[1] A model-based way to do backups at longer time scales requires a model that makes predictions at longer time scales, i.e., makes predictions for abstract actions that span many time steps in the real world.

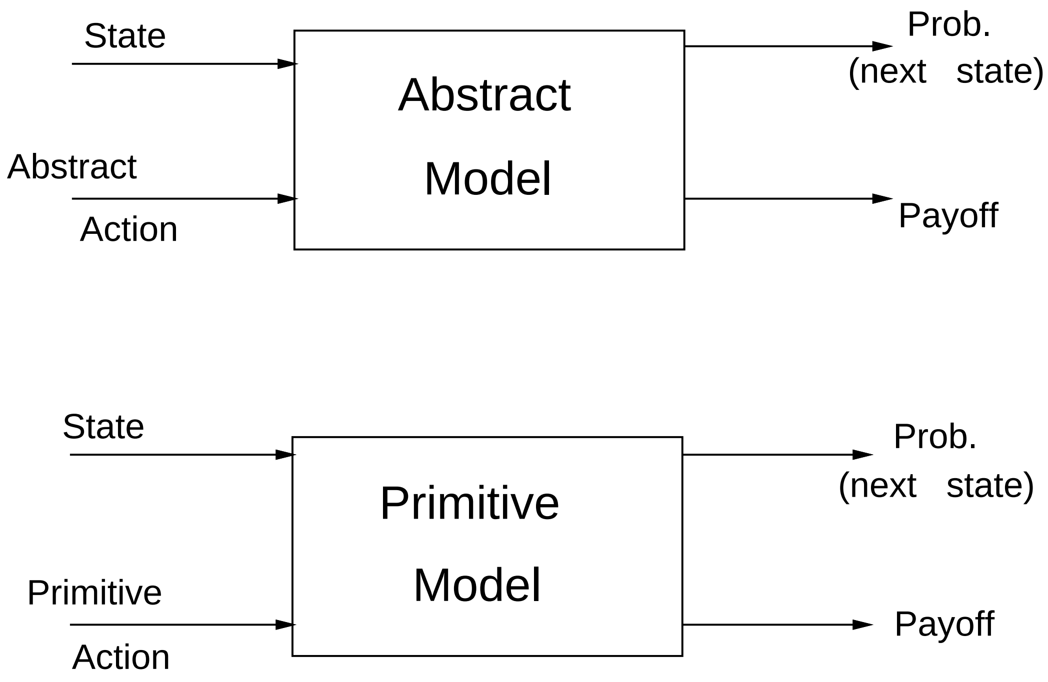

Figure 7.1 A Hierarchy of Environment Models. This figure shows a block diagram representation of two levels of a hierarchy of environment models. The lower level is the primitive model that stores the state transition probabilities and the payoff function for primitive actions. The upper level is the abstract model and stores the same two functions for abstract actions.

In the MDT framework there are two consequences of executing an action: the state of the environment changes, and the agent receives a payoff. Therefore, any environment model, whether primitive or abstract, has to store two functions of state-action pairs: the state transition probabilities, and the payoff function.[2] Figure 7.1 shows a two level hierarchy of environment models as block diagrams that take as input state-action pairs and output the payoff and the state transition probabilities for that state-action pair. The primitive model predicts the consequences of executing

[1]Alternatively, the controller can decrease the resolution by simply choosing not to change actions at some time steps – but this can only come at the expense of optimality. It also means that the agent may no longer be able to react appropriately to every state of the environment.

[2]Some researchers distinguish between a model of the state transition probabilities and a model of the payoff function by giving them distinct names, such as action model and payoff model respectively.

a primitive action, while the abstract model predicts the consequences of executing an abstract action.

7.2 Closed-loop Policies as Abstract Actions

The motivation behind building abstract models in the RL framework is the same as that of building abstract models for problem solving, that of achieving temporal abstraction. AI researchers have long used macro-operators, which are labels for useful sequences of operators/actions, to build abstract models of the problem environment (e.g., Fikes *et al.* [41]). The problem solver uses such abstract environment-models to plan in terms of the macro-operators instead of the primitive operators (Korf [64]). Planning in the abstract model achieves temporal abstraction because it allows the problem solver to ignore unnecessary detail. This chapter extends the familiar idea of macro-operators developed for problem solving in deterministic domains into the reinforcement learning (RL) framework for solving stochastic tasks.

Consider the motivation behind macro-operators in more detail. Imagine a room with just one door. No matter where one wants to go from inside the room to outside the room, one has to go through the door first. A path planning agent should not have to replan or relearn the skill of getting to the door separately for all the different destinations. A problem solving agent should be able to use the macro-operator "get-to-door" in planning for the optimal route to a new destination. Doing so will not only allow the agent to ignore unnecessary temporal detail but will also transfer knowledge across tasks which is the overall goal for this architecture.

The difficulty in building a macro-operator get-to-door in stochastic environments is that there may not be any finite open-loop sequence of actions that is guaranteed to get the agent to the door. The main innovation in this chapter is the idea that in many stochastic environments there may be a closed-loop policy that is guaranteed to get the agent to the door with probability one. Therefore, in stochastic environments it is possible to define abstract actions that are labels for closed-loop policies, instead of macro-operators, which are labels for open-loop policies. The architecture presented in this chapter builds abstract models for abstract actions that express the intention of achieving a "significant" state in the environment.

7.3 Building Abstract Models

In the above example of finding paths from a room to destinations outside the room, the door was obviously a significant state. The general problem of automatically finding significant states in arbitrary environments is difficult and is not addressed in this dissertation. Instead, a simple heuristic is used; the goal states of all the tasks faced by an agent in its lifetime in an environment are the significant states in that environment. This heuristic may be difficult to apply in environments where every state could become a goal state, but even in such cases, techniques for pruning the list of significant states over time could make the heuristic computationally feasible.

In this chapter the goal is to develop an abstract architecture for efficiently building and exploiting a hierarchy of environment models in the multi-MDT framework.

The focus is on agents that have to solve a set of MDTs that share the following properties:

1. Each MDT has a single absorbing goal state.
2. The payoff function for each MDT is decomposed into a non-positive cost function that is independent of the task being performed, and a non-negative reward function that is zero everywhere except the goal state.
3. The objective function to be maximized is the expected infinite-horizon sum of the *undiscounted* payoffs.

Note that the compositionally-structured MDTs of the previous chapter have the above properties in addition to the property that the composite MDTs are composed by sequencing the elemental MDTs. As before, it is assumed that there is an external agency that provides task commands to the agent that are not indicative of the decomposition of a task. In this chapter a generic task, elemental or composite, will be denoted simply as task i. Smaller case 'a' will be used for the primitive actions and the upper case 'A' will be used for abstract actions.

The agent simultaneously learns three things: the primitive model, the abstract model and the value function. For each task the agent faces, say task i, it adds an abstract action A_i to the abstract model and estimates its state transition probabilities and payoff function. Under the assumptions listed above about the nature of the MDTs faced by the agent, there is a very efficient way to build abstract models. Figure 7.2 shows how the value function learned for task i and the goal state of task i are equivalent to an abstract model for action A_i. The value function table for task i stores an estimate of $V_i^*(x)$, the optimal value of state x, but $V_i^*(x)$ is also the cost of executing abstract action A_i in state x. The next state after executing action A_i in state x is unique and it is the goal state of task i. Notice that even though the primitive model and the real environment may be stochastic, the abstract model is deterministic. In summary, no extra computation is needed to learn an abstract model for abstract action A_i; the information acquired while learning to solve task i is already an abstract model. Therefore, the abstract model explicitly stores only the state-independent goal state for each abstract action. The payoff function is not stored explicitly because it already exists in the value function table.

7.4 Hierarchical DYNA

Sutton [108] developed an on-line RL architecture called DYNA that uses the agent's experience to learn simultaneously a value function and build a primitive model of the environment. In the time interval between two actions in the real environment, DYNA updates the value function by doing backups on the primitive model. This chapter extends DYNA to hierarchical-DYNA, or H-DYNA, that not only builds a primitive model but also an abstract model. H-DYNA is an abstract architecture, in the tradition of Sutton's DYNA, and its main purpose is to illustrate the utility of building abstraction hierarchies.

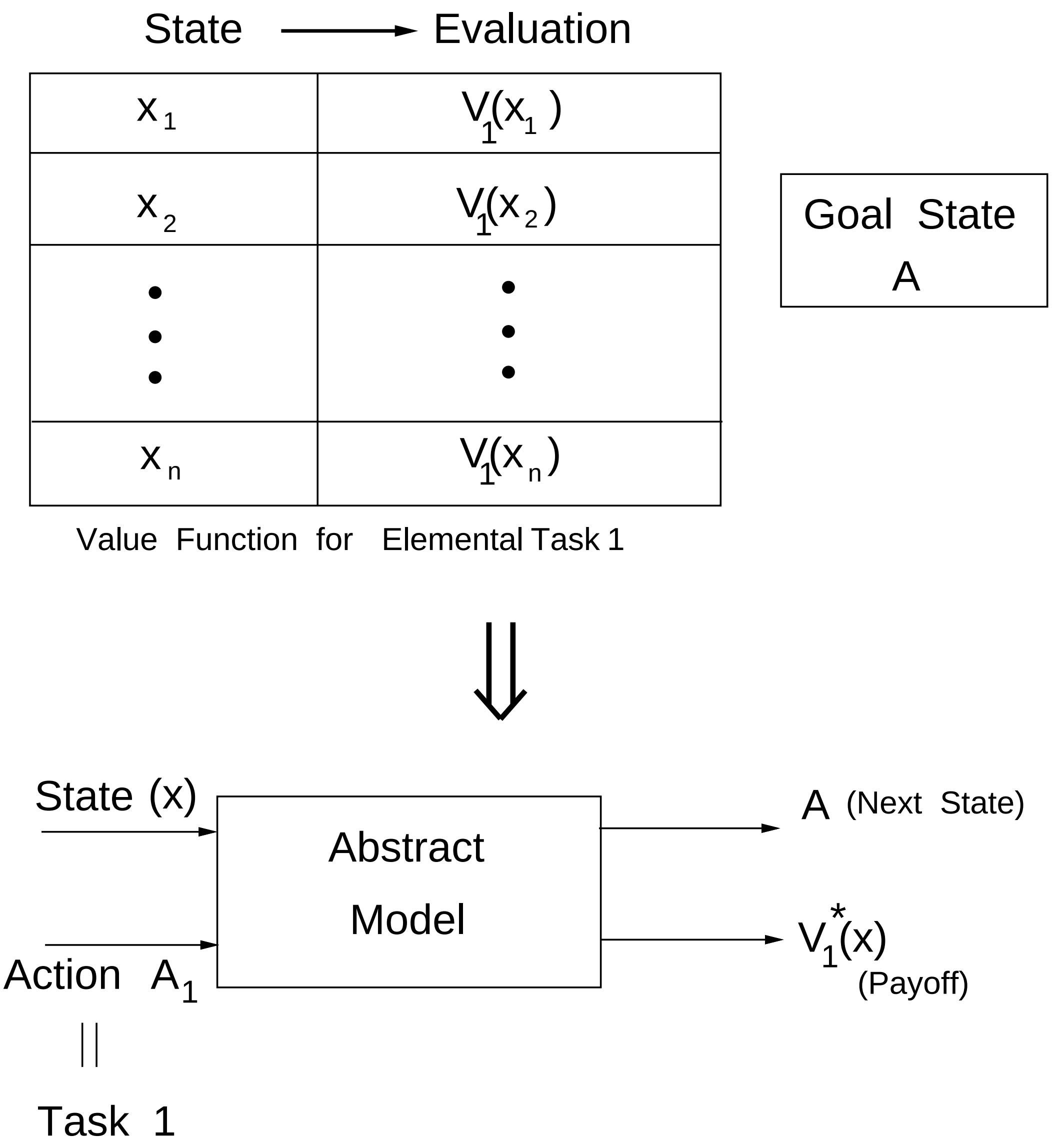

Figure 7.2 Building An Abstract Environment Model. This figure shows the abstract model for abstract action A_1 that is associated with Task 1. The goal state of Task 1 is labeled A. The payoff on executing action A_1 in a state x is $V_1^*(x)$, and state-independent next state is A. Therefore, learning the abstract model for Task 1 simply requires storage of the goal state A. The payoff function for action A_1 is already stored in the value function for Task 1.

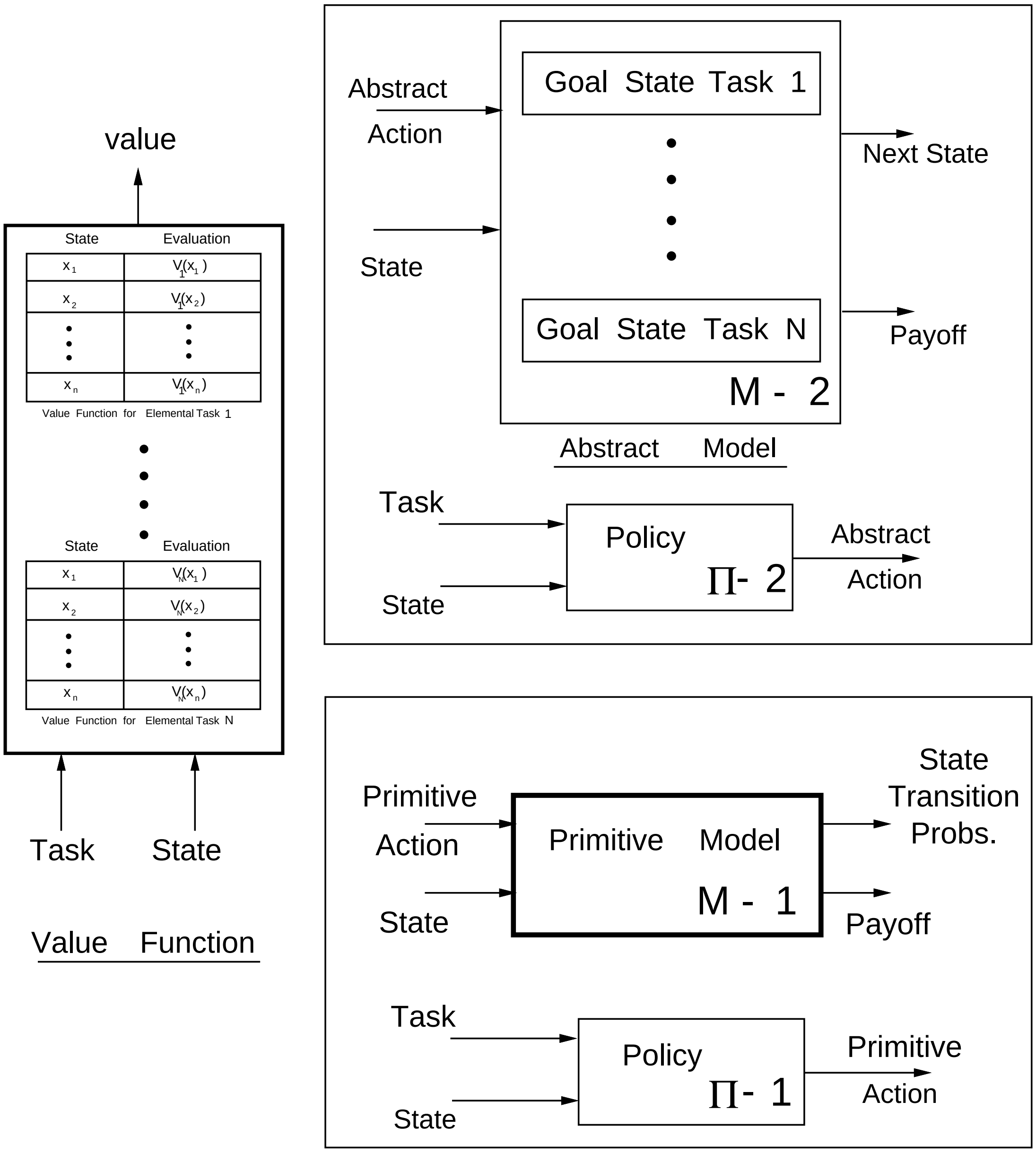

Figure 7.3 The Hierarchical Dyna (H-DYNA) architecture. The module on the left is the value function module that stores the value function V_i for each task i faced by the agent. The upper module on the right shows the abstract model M-2 and the abstract policy module Π-2. The abstract model stores the state-independent goal states for each task faced by the agent. The lower module in the right hand side shows the primitive model M-1 and the primitive policy module Π-1.

The essential elements of H-DYNA are shown in Figure 7.3: the value function module that stores the value function tables, one for each task, the primitive model (M-1), and the abstract model (M-2). There is a policy module Π-1 for M-1 that takes a task label and state as input and outputs a primitive action. Similarly there is a policy module Π-2 for model M-2.

Value Function Module: The value function module stores the value function V_i for each task i in a separate lookuptable. The value functions are updated using the TD algorithm. When the agent faces a new task, it adds a new lookuptable for that task to the value function module. The initial entries in the table are some default value, usually zero.

Primitive Model (M-1): The primitive actions and the dynamics of the environment do not change with the task. Therefore, learning the primitive model just involves keeping some statistics about transitions in the real environment independent of the task being solved. If $n^a(x, y)$ is the number of times the agent executed action a in state x and reached state y, then the estimated transition probability $\hat{P}^a(x, y)$ is $\frac{n^a(x,y)}{\sum_{y'} n^a(x,y')}$. The payoff function for state-action pair (x, a) is stored when action a is executed in state x for the first time.

Abstract Model (M-2): The abstract model stores the state-independent goal state for each abstract action. The payoff function is already stored in the value function module. When the agent faces a new task, say task i, it adds an abstract action A_i to the list of abstract actions and allocates memory for its goal state in the abstract model. When the goal state of task i is reached (for the first time), it is entered into the abstract model.

Primitive Policy Module (Π-1): It stores weights $w_i(x, a)$ for state-action pair (x, a) and task i. The probability of choosing action a in state x for task i is given by the Gibbs distribution, i.e., $P(a|x, i) = \frac{e^{\Gamma_i w_i(x,a)}}{\sum_{a'} e^{\Gamma_i w_i(x,a')}}$, where Γ_i is a temperature coefficient for task i. The weight values are updated by using TD (cf. Sutton's DYNA-Pi). The temperature is increased over time to increase the probability of taking a greedy action. A new table is allocated whenever the agent faces a new task. The initial weights are set to zero.

Abstract Policy Module (Π-2): It stores the weight $W_i(x, A)$ for state x, abstract action A, and task i. Actions are chosen using the Gibbs distribution just as in the primitive model. The weights are also updated using TD. A new table is allocated whenever the agent faces a new task. The initial weights are set to zero.

7.4.1 Learning Algorithm in H-DYNA

The operation of the architecture is a straightforward extension of Sutton's DYNA. Learning proceeds in trials as shown in Figure 7.4. Each trial starts with the external agency (experimenter) choosing a task for the agent, say task i, and ends when the task has been completed successfully. The trial proceeds in a loop. At the beginning of the loop it is determined if it is time for the agent to act in the real environment. If yes, then the agent selects a primitive action, say a, to execute in the current state x from the policy module Π-1, executes it in the real world, and then updates three things: the value function for state x and task i, the policy weights stored in Π-1 for state x, and the primitive model for abstract action a in state x.

At the beginning of the loop, if it is not time to act, the agent can do some model-based learning in which the agent is not constrained to update the value function for the current state in the current task. The agent can pick an arbitrary task j, an arbitrary state z, and the model M-1 or M-2 that it will use for the simulation. Assume, w.l.o.g., that it picks M-2. Then it can simulate the consequences of executing the action proposed by Π-2, say A, in state z for task j. The simulated experience is used to update two things: $V_j(z)$, the value of state z in task j, and the weights stored for state z in Π-2. Similarly, if the agent had chosen to simulate in M-1, $V_j(z)$ and the weights for state z in Π-1 would be updated. A trial ends when the goal state of the current task is reached.

If at the beginning of a trial, the agent receives a task command it has not seen before, then some new memory has to be allocated for that task. The agent adds a new value function table to the value function module, allocates memory for the goal state of that task in the abstract module, and allocates a table in Π-1 and Π-2 for that task. This new memory is filled with some default values. The rest of the loop remains the same.

7.5 Empirical Results

H-DYNA is a general architecture for building abstract environment models for abstract actions in RL tasks. The above description was presented for abstract actions that express intentions of achieving significant states in the environment. It was suggested that assuming the goal states of all the tasks faced by an agent in its environment to be significant states was a useful heuristic. In this section, this heuristic is tested in the context of compositionally-structured MDTs developed in Chapter 6. In particular H-DYNA is tested on the gridworld navigation tasks presented in Section 6.3.

7.6 Simulation 1

This simulation was designed to illustrate two things: first, that it is possible to solve a composite task by doing backups exclusively in the abstract model M-2, and second, that it takes fewer backups to learn the optimal value function by doing backups in the abstract model as compared to the number of backups it takes in the primitive model. H-DYNA was first trained on the three elemental tasks T_1, T_2 and T_3 (see Table 6.1). The system was trained until the primitive model had learned the expected payoffs for the primitive actions and abstract model had learned the expected payoffs for the three elemental tasks. This served as the starting point for two separate training runs for composite task C_3 that requires the agent to execute tasks T_1, T_2 and T_3 in that order.

For the first run, only M-1 was used to generate information for a backup. For the second run the same learning parameters were used, and only M-2 was used to do the backups. To make the conditions as similar as possible for the comparison, the

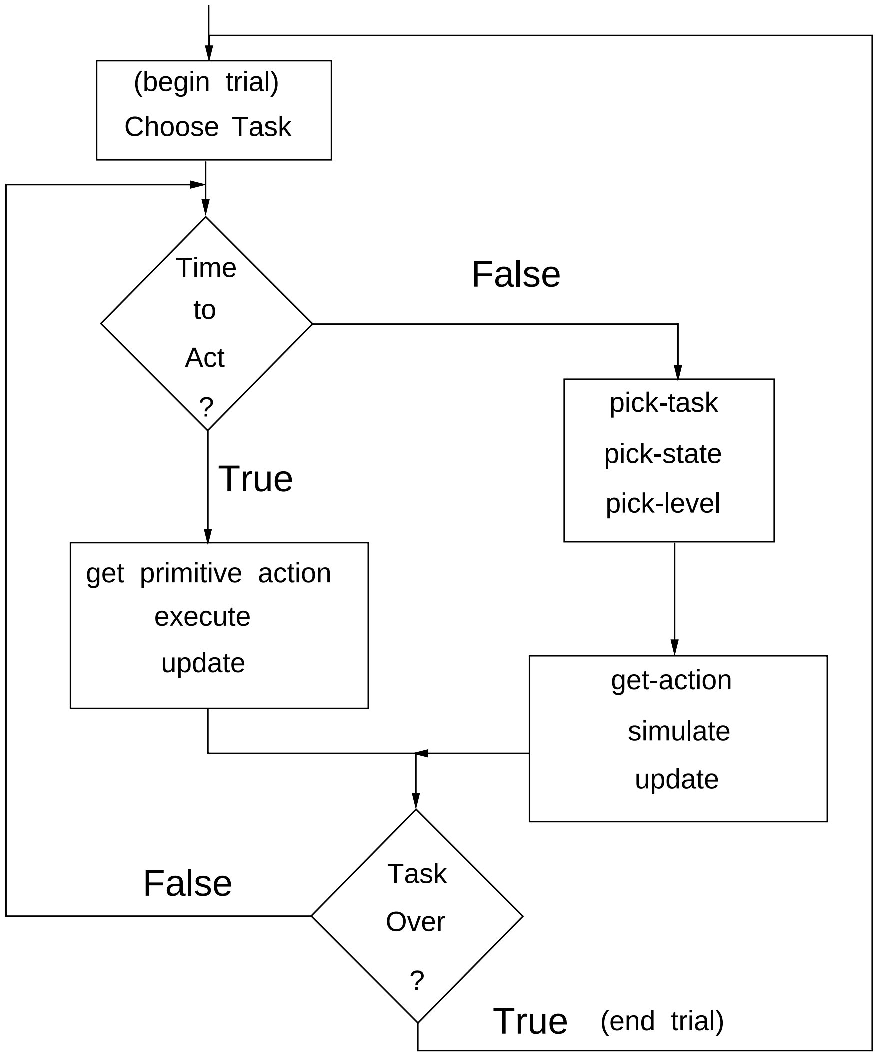

Figure 7.4 Anytime Learning Algorithm for H-DYNA. This figure shows the flowchart for the algorithm implemented by H-DYNA. It is a trial based algorithm. At the start of a trial a task is chosen for the agent. At any given moment, if it is time to act, a primitive action is chosen from Π-1 and executed in the real environment. All the modules are updated based on that real experience. If it is not time to act, the agent can update the value of any state for any task using simulated experience from M-1 or from M-2.

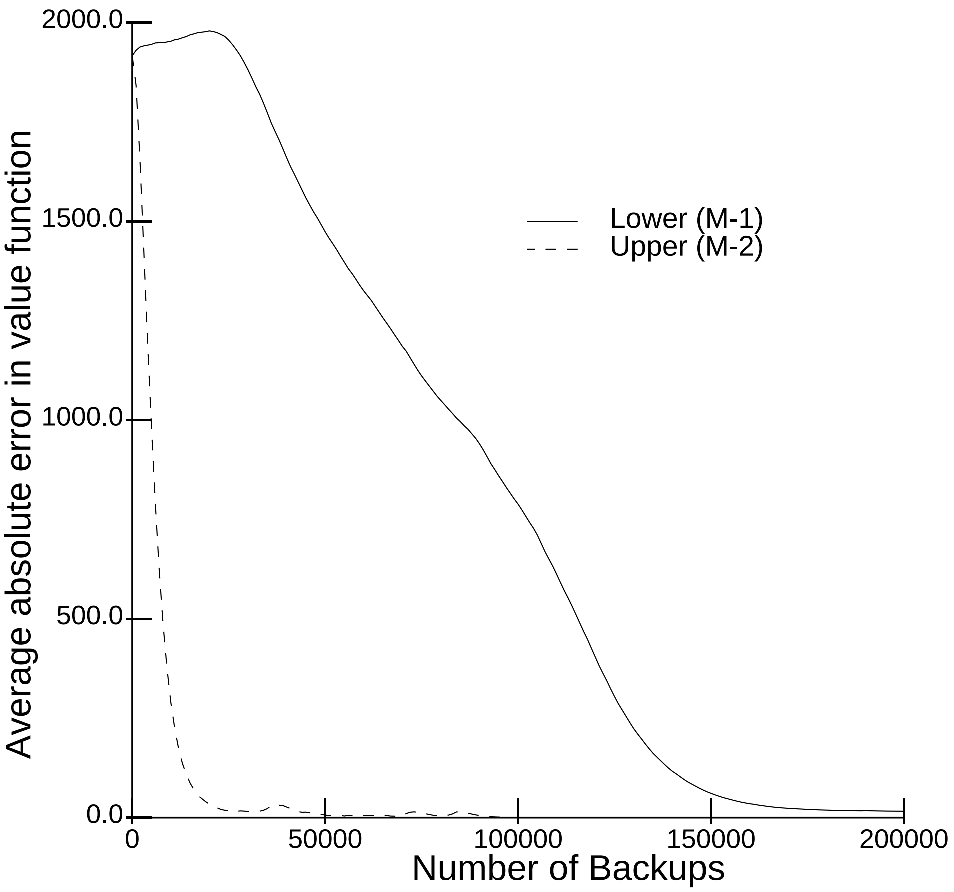

Figure 7.5 Rate of Convergence in M-1 versus M-2. This figure compares the rate of convergence of the value function for two algorithms: one does backups in the primitive model M-1, and the other does backups exclusively in the abstract environment model M-2. The task is C_3.

order in which the states were updated was kept the same for both runs by choosing predecessor states in a fixed order. After each backup, the absolute difference between the estimated value function and the previously computed optimal value function was determined. This absolute error was summed over all states for each backup and then averaged over 1000 backups to give a single data point. Figure 7.5 shows the learning curves for the two runs. The dashed line shows that the value function for the second run converges to the optimal value function. The two curves show that it takes far fewer backups in M-2 than M-1 for the value function to become very nearly-optimal.

7.7 Simulation 2

This simulation was conducted on-line to determine the effect of increasing the ratio of backups performed in M-2 to the backups performed in the real world. H-DYNA is first trained on the 3 elemental tasks T_1, T_2, and T_3, for 5000 trials. Each trial started with the agent at a randomly chosen location in the gridworld, and with a randomly selected elemental task. Each trial lasted until the agent had either successfully completed the task, or until 300 actions had been performed. After 5000 trials H-DYNA had achieved near-optimal performance on the three elemental tasks. Then the three composite tasks (see Table 6.1) were included in the task set. For each trial, one of the six tasks was chosen randomly, the agent started in a randomly chosen start state, and the trial continued until the task was accomplished or there was a time out. The tasks, C_1 and C_2 were timed out after 600 actions and the task C_3 after 800 actions.

For this simulation it is assumed that controlling the robot in real-time leaves enough time for the agent to do n backups in M-2. The purpose of this simulation is to show the effect of increasing n on the number of backups needed to learn the optimal value function. No backups were performed in M-1. The simulation was performed four times with the following values of n: 0, 1, 3 and 10. Figure 7.6 shows the results of the four different runs. Note that each backup performed in M-2 could potentially take much less time than a backup performed in the real world. Figure 7.6 displays the absolute error in value function plotted as a function of the number of backups performed. This results of this simulation show that even when used on-line, backups performed in M-2 are more effective in reducing the error in the value function than a backup in the real world.

7.8 Discussion

The previous chapter presented a model-free or direct approach for accelerating the learning of solutions for compositionally-structured MDTs that are defined in the same environment. The relative merits of adaptive model-based approaches versus model-free approaches in the single-task context are still debated and in the opinion of this author the answer is likely to be problem dependent. However, for agents that have to learn to solve multiple tasks defined in the same environment, it seems reasonable to assume that building an environment model will be useful. In the multi-task context, learning a model allows transfer of knowledge that is invariant

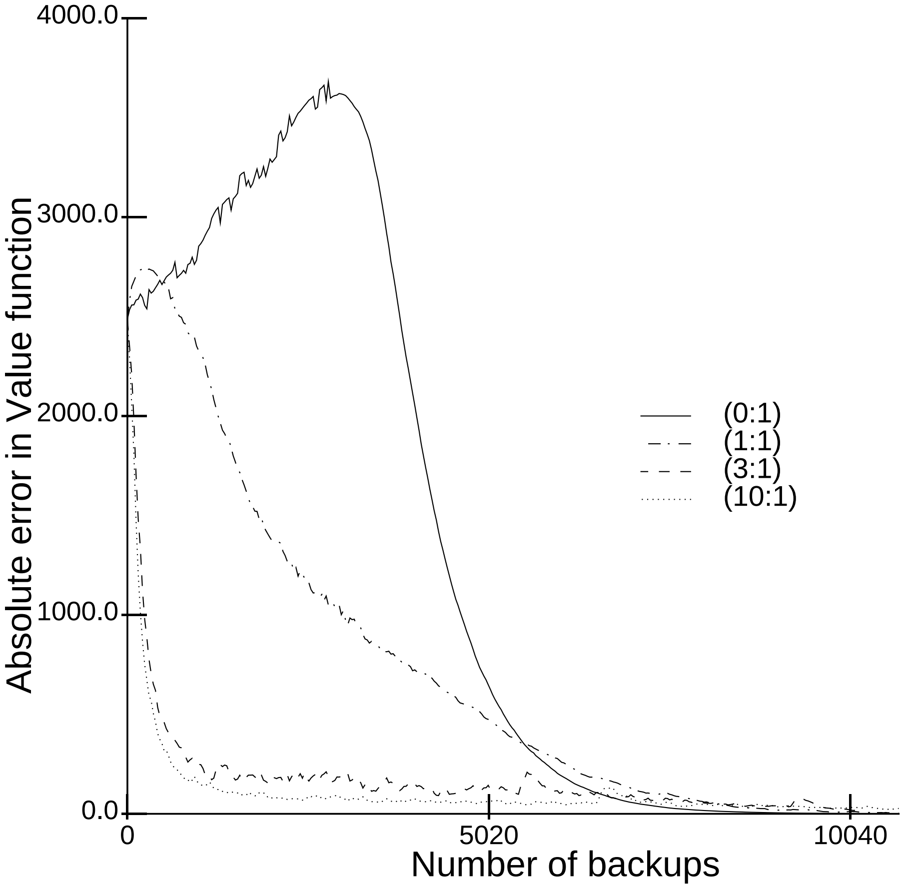

Figure 7.6 On-line Performance in H-DYNA. This figure shows the effect of increasing the ration of the number of backups in M-2 to the number of backups in the real environment.

across tasks (see, also Mahadevan [70]), and the computational expense of building the model can be amortized across the set of tasks.

In both the single and multiple task contexts, the nature of the model will play a role in determining its usefulness. As demonstrated in this chapter, it is possible to build models that do temporal abstraction by predicting the consequences for abstract actions. Solving new tasks by doing state-updates in abstract models can lead to low-depth solutions and that can accelerate convergence to the optimal value function.

7.8.1 Subsequent Related Work

After the publication of this author's work (Singh [95, 98, 97]) on CQ-L and H-DYNA, other authors have developed hierarchical RL architectures that make different assumptions and have different strengths and weaknesses. Lin [66] developed an architecture that first learns elementary tasks and then learns how to compose them to solve more complex tasks. Lin's work assumes deterministic environments but does not assume the compositional structure on tasks assumed in this authors work. It also does not build models of the elementary tasks. Dayan and Hinton [35] have developed a hierarchical architecture, called feudal RL, that also does not assume compositional structure on tasks. It consists of a predefined hierarchy of managers, each of whom has the power to set goals and payoffs for their immediate sub-managers. Each component of the hierarchy attempts to maximize its own expected long-term payoff. The system has learned when the goal of the top level manager is satisfied.

7.8.2 Future Extensions of H-DYNA

H-DYNA represents preliminary research on the topic of building abstract environment models. The structure of the abstract model made some assumptions about the class of tasks faced by the agent. One assumption that turns out not to be a limitation is that each task should have a single goal state. If a task has more than one goal state then all that will change is that the next-state function in the abstract model will no longer be state independent. The abstract model will have to store the goal state reached as a function of the start state. A second assumption that the cost function be task independent is somewhat more limiting. However, there are many classes of tasks where the cost function will be task independent, e.g., in many robotic tasks defined in the same environment.

Another potential drawback of H-DYNA is that in the worst case the number of abstract actions can equal the number of states in the environment. However, that is unlikely to be true in any realistic task setting. Nevertheless, as one increases the number of abstract actions, the advantage gained by a reduced depth is offset by the disadvantage of increasing branching factor. However, it should be possible to combine H-DYNA with some heuristics for pruning the abstract actions from time to time. Two obvious heuristics are: prune the least recently used abstract actions or the least frequently used abstract actions.

CHAPTER 8

ENSURING ACCEPTABLE BEHAVIOR DURING LEARNING

An agent using RL to solve an optimal control problem has to search, or explore, in order to avoid settling on suboptimal solutions. In off-line learning, exploration does not directly affect performance because the agent uses simulated experience derived from a model of the environment to compute a solution before applying it to the actual environment. However, in on-line learning, exploration can lead the agent to perform worse than is acceptable or safe in the real environment. For example, if the environment has catastrophic 'failure' states, exploration can lead to disaster despite the fact that the agent may already know a 'safe' but suboptimal solution.

Although the need for exploration cannot be removed entirely, this chapter presents a technique that constrains the solution space for a complex task to ensure that the exploration is conducted in a space composed mostly of acceptable solutions. The solution space is constrained by replacing the conventional primitive actions for the complex task by actions that engage closed-loop policies found for suitably-defined simpler tasks. This method also accomplishes transfer of training from simple to complex tasks. It is demonstrated in an optimal motion planning problem, a component of many problems in robotics. Empirical results are presented using a simulated dynamical robot in two different environments.

Constraining the solution space for a complex task in the manner described above also reduces the size of the solution space and thereby accelerates learning. One has to be careful, though, not to constrain the solution space so much as to exclude all good solutions. In the previous two chapters the terms simple and complex tasks were used to refer to elemental and composite MDTs that have a hierarchical relationship. The terms simple and complex tasks are used in a different sense here that will become clear later in the chapter.

8.1 Closed-loop policies as actions

To formulate a given control problem as an MDT one has to choose the state set and the actions available to the agent. In most attempts to apply RL, the actions of the agent are *primitive* in that they are the low-level, general-purpose actions that the agent can perform in most states, e.g., "rotate wheel by 90 degrees", or "close gripper". Primitive actions are assumed to have the following characteristics: 1) They are executed open loop, and 2) They last one time step. This is an arbitrary and self imposed restriction; in general the set of actions can have a much more abstract relationship to the problem being solved. Specifically, what are considered 'actions' by the RL algorithm can themselves be goal-directed closed-loop control policies.

RL algorithms search for optimal policies in a policy space defined by the actions available to the agent. Changing the set of actions available to an agent changes the policy space. RL should still find an optimal policy but only with respect to the changed policy space. The previous chapter used abstract actions that were closed-loop policies to build abstract environment models for achieving temporal abstraction. In this chapter, the motivation for considering abstract actions is to 1) constrain the policy space to satisfy external criteria, such as excluding unsafe policies, and 2) reduce the size of the policy space so that finding the optimal policy is easier. Of course, care has to be taken to ensure that the reduced policy space in fact does contain a policy that is close in evaluation to the policy that is optimal with respect to the largest policy space physically realizable by the agent.

The robustness and greatly accelerated learning resulting from the above factors can more than offset the cost of learning the abstract actions. The next section presents the optimal motion planning problem and a brief sketch of the harmonic function approach to path planning developed by Connolly [29] that is used to compute the abstract actions.

8.2 Motion Planning Problem

The motion planning problem arises from the need to give an autonomous robot the capability of planning its own motion, i.e., deciding what motions to execute in order to achieve a task specified by initial and goal spatial arrangements of physical objects. In the most basic problem, it is assumed that the robot is the only moving object, and the *dynamic properties of the robot are ignored.* Furthermore motion is restricted to non-contact motions so that mechanical interaction between the robot and its environment is also ignored. These assumptions transform the motion planning problem into a geometrical path planning problem. In the context of achieving transfer of training from simple to complex tasks, motion planning is the complex problem and geometric path planning is the simple problem.

Further simplification of the path planning problem is achieved by adopting the configuration space representation. The essential idea in configuration space is to represent the robot as a point in the robot's configuration space, and to map the obstacles into that space. This mapping transforms the problem of planning the motion of a Cartesian robot into the problem of planning the motion of a point in configuration space. The solution to the path planning problem is a path from every starting point that avoids obstacles and reaches the goal.

Both the motion planning problem and the path planning problem defined above are not optimal control problems because there is no objective function to optimize. To convert the motion planning problem into an optimal motion planning problem one requires the learner to find paths that minimize some objective function. The objective function adopted here is the time-to-goal objective function. Therefore the solution paths will be minimum time paths. The associated path planning problem, however, is still not an optimal control problem because time does not play a role.

The motivation behind the approach presented in this chapter is to use paths found for the (non-optimal) path planning problem to *a*) help reduce the number of failures and ensure acceptable performance, and *b*) to accelerate learning, in the more complex optimal motion planning problem. Even though it is possible to learn the solutions to the path planning problem, the simulations reported here simply compute the solutions off-line via an efficient procedure described in the next section.

8.2.1 Applying Harmonic Functions to Path Planning

A conventional approach to solving the geometric path planning problem is the potential field approach. Roughly, the point-robot in configuration space is treated as a particle in an artificial potential well generated by the goal configuration and the obstacles. Typically the goal generates an "attractive" potential which pulls the particle towards the goal, and the obstacles produce a "repulsive" potential that repels the particle away. The negative gradient of the total potential is treated as an artificial force applied to the robot. The problem with this approach is that it is not guaranteed that the particle will avoid spurious minima, i.e., minima not at goal location.

Harmonic functions have been proposed by Connolly *et al.* [31, 30] as an alternative to local potential functions. Harmonic functions are guaranteed to have no spurious local minima. The description presented here closely follows that of Connolly *et al.* [31]. A harmonic function ϕ on a domain $\Omega \subset \mathcal{R}^n$ is a function that satisfies Laplace's equation:

$$\nabla^2 \phi \;=\; \sum_{i=1}^{n} \frac{\partial^2 \phi}{\partial x_i^2} = 0. \tag{8.1}$$

In practice Equation 8.1 is solved numerically by finite difference methods. In a two-dimensional configuration space, let $\phi(x, y)$ be the solution to Laplace's equation, and let $u(x_i, y_j)$ represent a discrete regular sampling of ϕ on a grid. A Taylor series approximation to the second derivatives is used to derive the following linear system of equations:

$$\begin{aligned} h^2\phi(x_i, y_j) \;&=\; u(x_{i+1}, y_j) + u(x_{i-1}, y_j) \\ &\quad + u(x_i, y_{j+1}) + u(x_i, y_{j-1}) - 4u(x_i, y_j) \end{aligned} \tag{8.2}$$

where h is the grid spacing, henceforth set to be 1.0 without loss of generality. Equation 8.2 can be extended to higher dimensions in the obvious way.

Equation 8.2 can be solved by a relaxation equation much like successive approximation was used to solve the linear system of equations associated with policy evaluation. Successive approximations to ϕ are produced using the following iteration:

$$\begin{aligned} u_{k+1}(x_i, y_j) \;&=\; \frac{1}{4}(u_k(x_{i+1}, y_j) + u_k(x_{i-1}, y_j) \\ &\quad + u_k(x_i, y_{j+1}) + u_k(x_i, y_{j-1})). \end{aligned} \tag{8.3}$$

This iteration is performed for all non-boundary grid points, where the boundary of Ω, labeled $(\delta\Omega)$ consists of the boundaries of all obstacles and goals in a configuration-space representation.

At the grid points along the boundary of the configuration space the iteration depends on the nature of the boundary condition on ϕ. The following is called a Dirichlet boundary condition: $\phi|_{\delta\Omega} = c$, where c is some constant, and $\phi|_{\delta\Omega}$ is the value of the function ϕ at the boundary $\delta\Omega$. It amounts to holding the boundary to a fixed potential of c. Solutions derived with the Dirichlet boundary conditions are denoted ϕ_D. A second boundary condition is called the Neumann boundary condition defined as: $\frac{\delta\phi}{\delta\mathbf{n}}|_{\delta\Omega} = 0$, where $\mathbf{n}$ is the vector normal to the boundary, and $\frac{\delta\phi}{\delta\mathbf{n}}$ is the derivative of ϕ in the direction $\mathbf{n}$. The Neumann boundary condition constrains the gradient of ϕ at the boundary to be zero in the direction normal to the boundary. Solutions derived from the Neumann boundary condition are denoted ϕ_N.

8.2.2 Policy generation

The gradient of a harmonic function, $\nabla\phi$, defines streamlines, or paths, in configuration space that are guaranteed to be *a*) smooth, *b*) avoid all obstacles, and *c*) terminate in a goal state (Connolly *et al.* [30]). The paths generated by ϕ_D as well as by ϕ_N have these properties but are qualitatively very different from one other. The Dirichlet paths are perpendicular to the boundary, while the Neumann paths are parallel to the boundary. Examples of both types of paths are shown later in Figure 8.3, lower panel. These paths are solutions to the geometric path-planning problem. A controller to follow these paths can be obtained by using the gradient of the harmonic function as a velocity command for the robot controller. Depending on the type of the boundary condition, the controller would execute the following closed-loop control policy:

$$\begin{aligned} \pi_D(x) &= \nabla\phi_D|_x, \\ \pi_N(x) &= \nabla\phi_N|_x \end{aligned}$$

where x is some point in configuration space. Note that the actions of the robot are velocity commands for a velocity reference controller.

Note that the policies π_D and π_N are not solutions to an optimal control problem, i.e., they are not derived to optimize some objective criteria such as minimum time, or minimum jerk.[1] In the next section, a RL problem is defined that uses the Dirichlet and Neumann closed-loop control policies as abstract actions to define a policy space

8.2.3 RL with Dirichlet and Neumann control policies

The state space of the optimal motion planning problem for a robot is larger than the configuration space in which the harmonic functions are computed. For example, the state space in the motion planning problem for a robot in a planar environment is $\Re^4$ ($\{x, \dot{x}, y, \dot{y}\}$). The harmonic functions are computed by ignoring the dynamics,

[1]However the harmonic functions minimize a particular functional that is independent of the dynamics of the robot.

i.e., they are defined in two dimensional position space $\Re^2$ ($\{x, y\}$). One can define Dirichlet and Neumann policies (π_D and π_N) in state space as follows:

$$\begin{aligned}\pi_D(x) &= \nabla\phi_D|_{\hat{x}} \\ \pi_N(x) &= \nabla\phi_N|_{\hat{x}}\end{aligned}$$

where x is some point in the state space and $\hat{x}$ is its projection into configuration space. As before, the actions defined by policies π_D and π_N prescribe velocity commands for a velocity reference controller.

Instead of formulating the optimal motion planning problem as a RL task in which a control policy maps states into physical actions, consider the formulation in which a policy maps state x to a *mixing* parameter $k(x)$ that then defines the physical action as :

$$(1 - k(x))\pi_D(x) + k(x)\pi_N(x),$$

where $0 \leq k(x) \leq 1$. Appendix E presents conditions that guarantee that for a robot with no dynamical constraints, the solution space defined by the action $k(x)$ will not contain any unacceptable solutions. Although the guarantees stop holding as one adds dynamical constraints to the robot, the new formulation does serve to reduce the risk of hitting an obstacle.

8.2.4 Behavior-Based Reinforcement Learning

Behavior-based robotics is the name given to a body of relatively recent work in robotics that builds robots equipped with a set of useful "behaviors" and solves problems by switching these behaviors on and off appropriately (e.g., the subsumption architecture of Brooks [22]). The term behavior is often used loosely to include all kinds of open-loop, closed-loop, and mixed control policies. The closed-loop Dirichlet and Neumann policies are examples of goal-seeking behaviors. In most behavior-based architectures for robots the switching circuitry and the behaviors are designed by the roboticist. (see Mahadevan and Connell [71] for an exception that learns behaviors but has fixed switching circuitry).

The learning architecture presented in this section is called BB-RL, for behavior-based RL, because it uses RL to learn an optimal policy that maps states to a mixture of Dirichlet and Neumann behaviors. Maes and Brooks [69] used a simple form of RL to learn the switching circuitry for a walking robot with hardwired behavior modules. Their formulation of the RL problem was as a single stage decision task in which the learner's goal was to select at each time step the behavior that maximizes the immediate payoff. BB-RL extends Maes and Brooks' [69] system to multi-stage decision tasks and to policies that assign linear combinations of behaviors to states instead of a single behavior to each state. (see, also Gullapalli *et al.* [117]).

8.3 Simulation Results

Figure 8.1 shows the two simulation environments for which results are presented in this section. The environment in the top panel consists of two rooms connected by a corridor, and the environment in the lower panel is a horseshoe-shaped corridor. The robot is simulated as a unit-mass, and the only dynamical constraint is a bound on the acceleration.

8.3.1 Two-Room Environment

The learning task is to find a policy that minimize time to reach goal region. Q-learning [118] was used to learn the mixing function, k. Figure 8.2 shows the 2-layer neural network architecture used to store the Q-values (see Appendix F for a brief review of layered neural networks). Because both the states and the actions are continuous for these problems a network inversion technique was used to determine the best Q-value in a state as well as to determine the best action in a state (cf. Chapter 6). The robot was trained in a series of trials, with each trial starting with the robot placed at a randomly chosen state and ending when the robot entered the goal region. The points marked by stars in Figure 8.1 were the starting locations for which statistics were collected to produce learning curves.

Each panel in Figure 8.3 shows three robot trajectories from a randomly chosen start state; the black-filled circles mark the Dirichlet trajectory, the white-filled circles mark the Neumann trajectories, and the grey-filled circles mark the trajectories after learning. Trajectories are shown by taking snapshots of the robot at every time step; the velocity of the robot can be judged by the spacing between successive circles on the trajectory. The upper panel in Figure 8.4 shows the mixing function for zero-velocity states. The darker the region, the higher the proportion of the Neumann policy in the mixture. The agent learns to follow the Neumann policy in the room on the left-hand side, and to follow the Dirichlet policy in the room on the right-hand side. The lower panel in Figure 8.4 shows the average time to reach the goal state as a function of the number of trials. The solid-line curve shows the performance of the Q-learning algorithm. The horizontal lines show the average time to reach the goal for the designated unmixed policies. It is clear from the lower panel of Figure 8.4 that within the first hundred trials the RL architecture determines a mixing function that greatly outperforms both the unmixed policies.

8.3.2 Horseshoe Environment

Figures 8.5 and 8.6 present the results for the horseshoe environment. As above, the black-filled circles mark the Dirichlet trajectory, the white-filled circles mark the Neumann trajectories, and the grey-filled circles mark the trajectories after learning. Figure 8.5 shows sampled trajectories from two different start states. Notice that the Dirichlet trajectory seems to be better than the Neumann trajectory after the bend in the horseshoe and worse before the bend. The upper panel in Figure 8.6 presents the learned mixing function for zero-velocity states. The lower panel in Figure 8.6 shows the performance curve of the learned mixed policy versus the pure Dirichlet and Neumann policies. In this environment, the pure Neumann policy is quite good. Nonetheless the Q-learning agent finds a better solution within the first ten thousand trials.

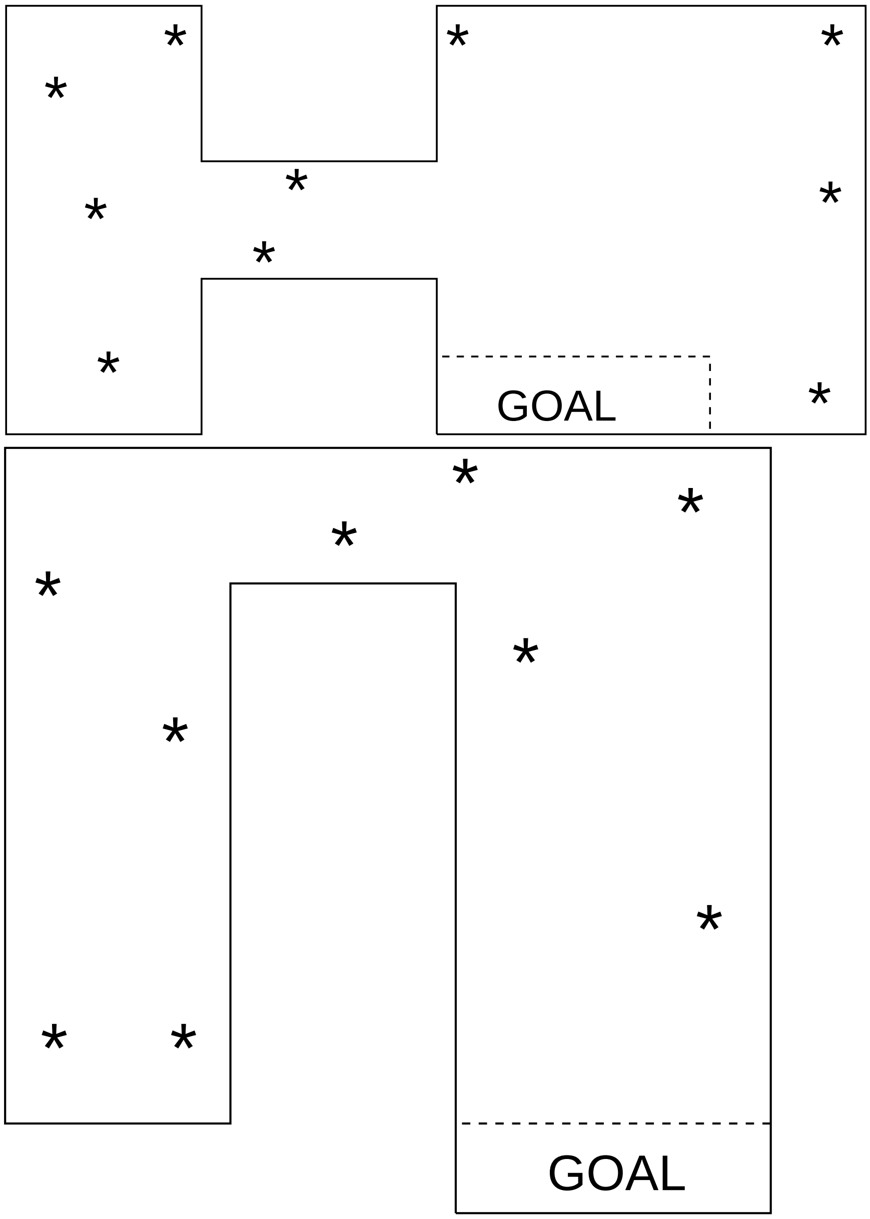

Figure 8.1 Simulated Motion Planning Environments. This figure shows the two environments for which results are presented in Section 8.3. The task is to find *minimum time* paths from every point in the workspace to the region marked as GOAL without hitting a boundary wall (shown by solid lines). The robot is trained in trials, each starting with the robot in a randomly chosen state and ending when the robot enters the goal region. The points marked with stars represent the starting locations for which statistics were kept to produce learning curves.

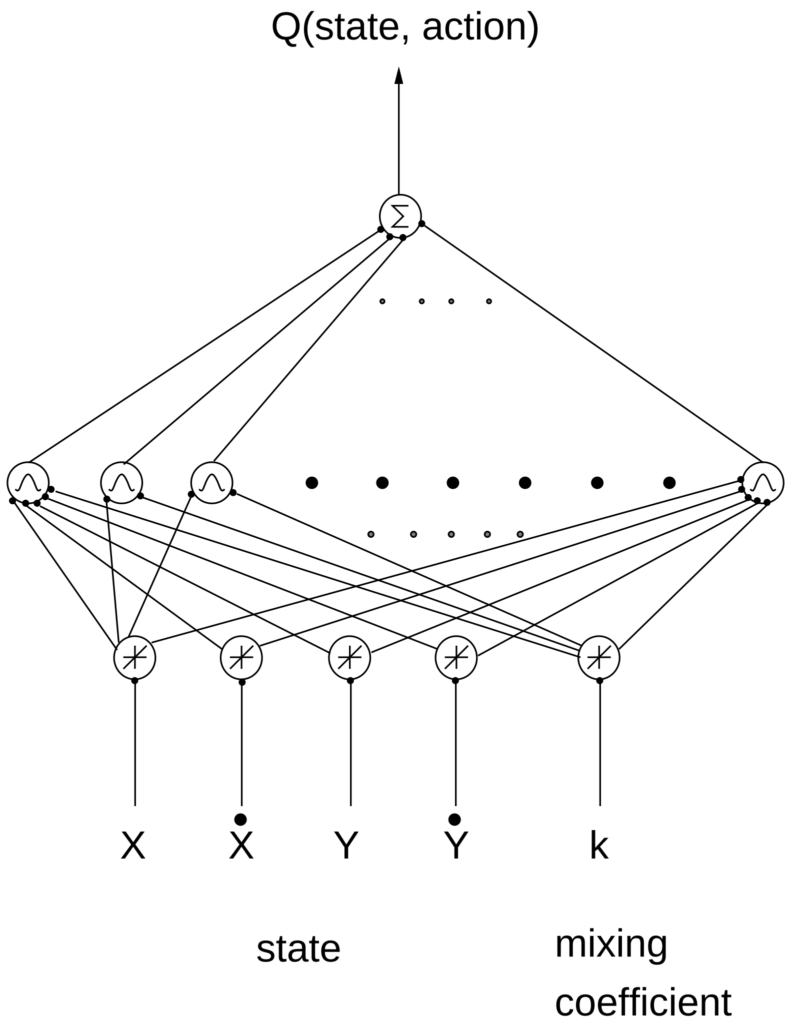

Figure 8.2 Neural Network for Learning Q-values. This figure shows a three-layered connectionist net, with the hidden layer composed of radial basis functions. The inputs to the net are the 4-dimensional state and the 1-dimensional action. The network was trained using backpropagation with target outputs determined by the Q-learning [118] algorithm.

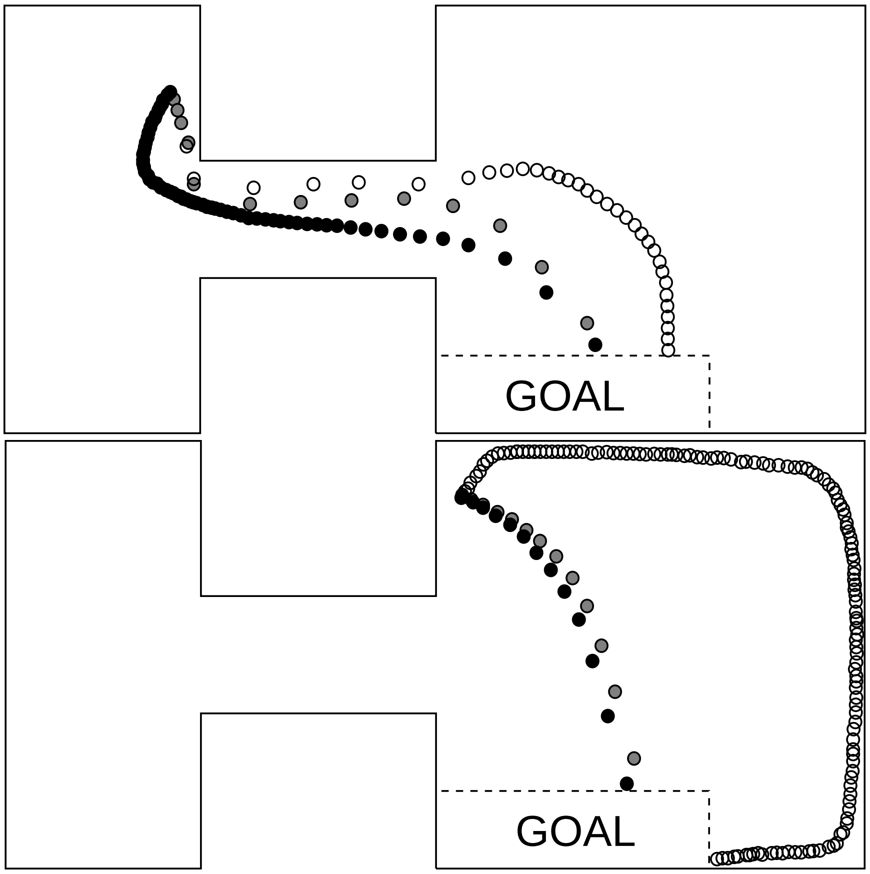

Figure 8.3 Sample Trajectories for Two-Room Environment. This figure shows the robot's trajectories from two different starting points. The black-filled circles mark the Dirichlet trajectory, white-filled circles mark the Neumann trajectory, and the grey-filled circles mark the trajectory that results from learning. Each trajectory is shown by showing the position of the robot after every time step. The velocity can be judged by the spacing between successive circles on a trajectory.

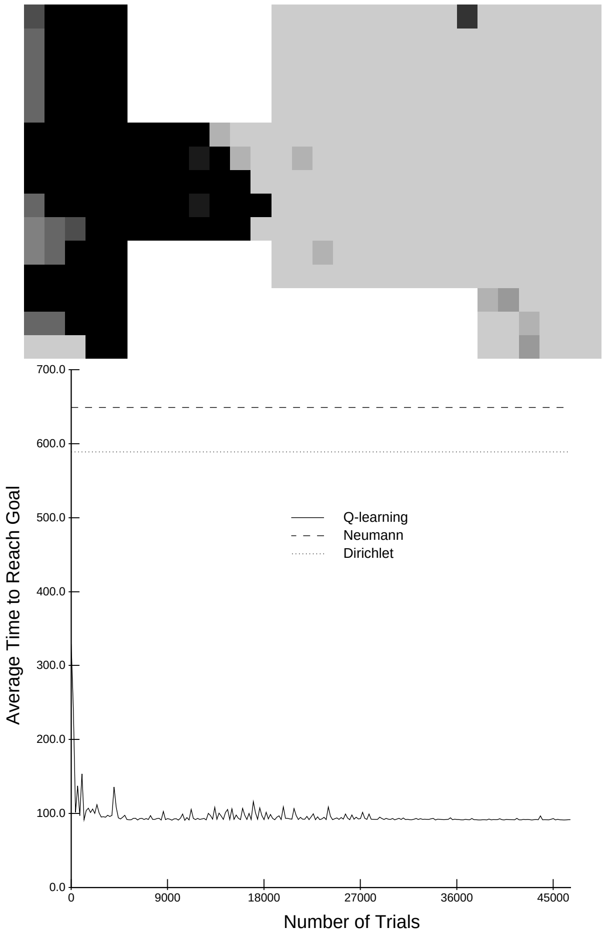

Figure 8.4 Learning Results for Two-Room Environment. The upper panel shows the mixing function for zero-velocity states after learning. The darker the region the higher the proportion of the Neumann policy in the action for that region. The lower panel compares the performance of the Q-learning robot relative to agents that follow un-mixed policies. The solid-line curve shows the incremental improvement over time that is achieved due to Q-learning. The horizontal lines represent the performance of the designated un-mixed policies.

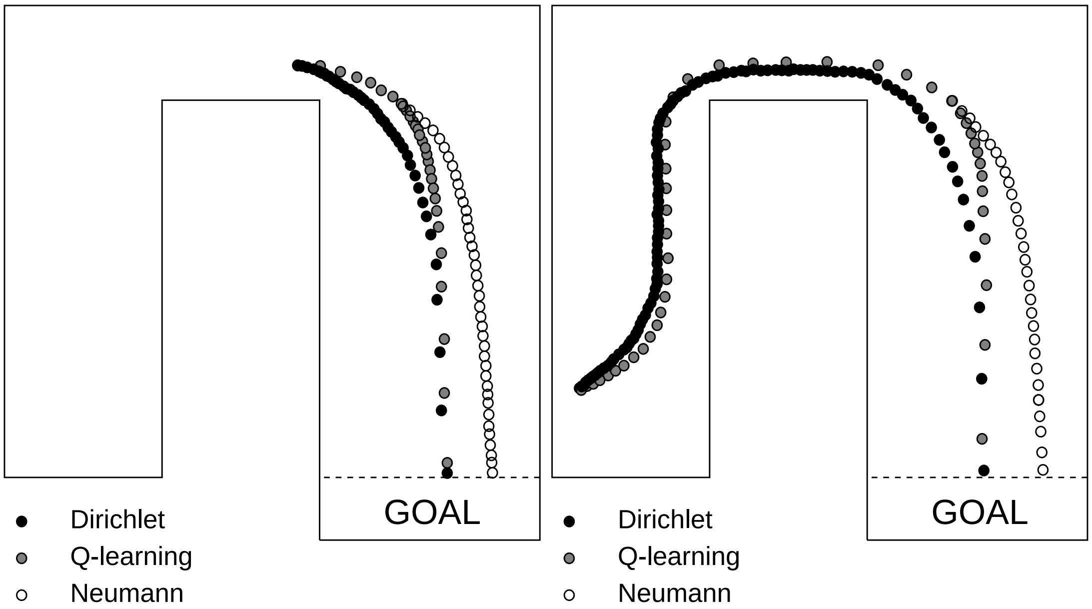

Figure 8.5 Sample Trajectories for the Horseshoe Environment. This figure shows the robot's trajectories from two different start states. The black-filled circles are used to show the Dirichlet trajectory, white-filled circles for Neumann trajectory, and the grey filled circles for the trajectory achieved after learning. The trajectory is shown by showing the position of the robot in position space after every second.

8.3.3 Comparison With a Conventional RL Architecture

The performance of BB-RL was compared with the performance of a conventional RL (C-RL) architecture that uses primitive actions to solve the optimal motion planning problem in the two-room and the horseshoe environments. The aim is to compare three things: the rate of convergence, the number of times the robot hits an obstacle, and the quality of the final solution. The primitive actions for the C-RL architecture are to choose an acceleration (magnitude and direction) for the robot. The magnitude of the acceleration is bounded from above. Notice that the action space for C-RL is two dimensional while the action space for BB-RL is one dimensional.

In the two-room environment the best C-RL architecture found by this author takes more than twenty thousand trials to achieve the performance achieved by the BB-RL architecture in roughly a hundred trials. Furthermore, the robot using the C-RL architecture collided with the boundary wall hundreds of times. The actual number of collisions varied with parameter settings and random number seeds. The final solution found by the C-RL architecture was 6% better than the final solution found by the BB-RL architecture. In the horseshoe environment, the C-RL architecture takes more than a hundred thousand trials to find a solution equivalent to a solution found by the BB-RL architecture in about ten thousand trials. The robot collides with a wall thousands of times, and the best solution is better by about 8%.

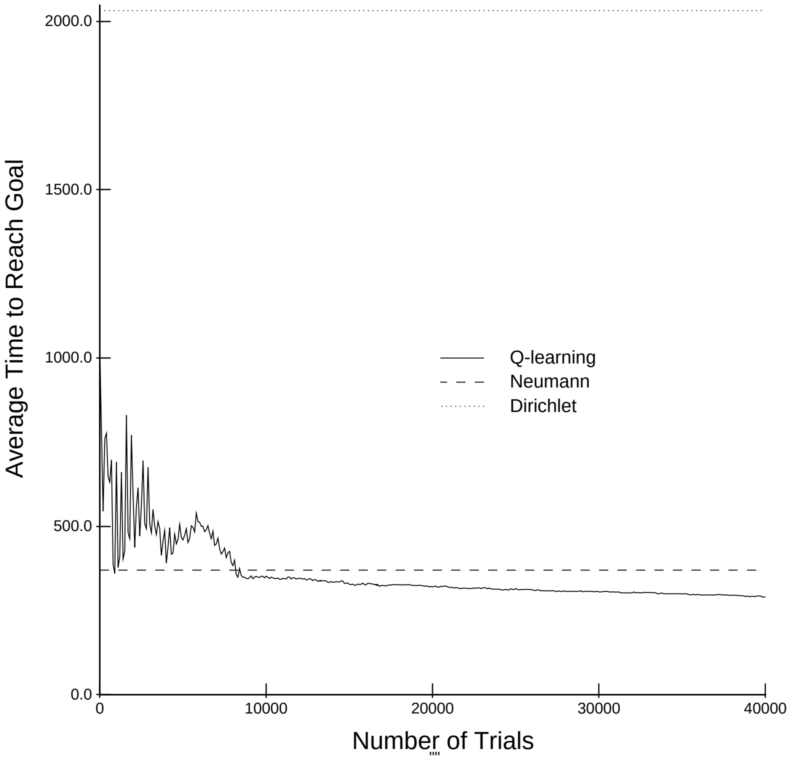

Figure 8.6 Learning Results for the Horseshoe Environment. The upper panel shows the mixing function for zero-velocity states. The darker the region, the higher the proportion of Neumann in the resulting action for that region. The lower panel compares the performance of the Q-learning agent relative to agents with fixed strategies. The solid-line curve shows the improving performance via Q-learning. The horizontal lines represent the performance of the fixed strategies shown on the line labels.

There are at least three reasons for the slower rate of learning in the C-RL architecture compared to the BB-RL architecture. The robot cannot generalize the concept of avoiding the wall; it has to learn it separately for each segment of the wall. Further, because the robot hits the wall so many times, the learning rate has to be very small to ensure that the early experience does not saturate the weights in the Q-learning network. The many collisions and the small learning rate slows down learning in C-RL. A third reason for the slow learning of C-RL is that the action space of C-RL is two dimensional compared to BB-RL's one-dimensional space.

In addition, C-RL was much more sensitive than BB-RL to the choice of learning rate and the neural network architecture used to implement Q-learning. In C-RL, a small increase in the learning rate prevented the robot from learning any solution at all. BB-RL is more robust because any choice of mixing function guarantees an acceptable level of performance.

8.4 Discussion

The BB-RL architecture kept the robot from colliding into a boundary wall, and it accelerated learning relative to a C-RL architecture. But BB-RL also has some disadvantages compared to C-RL. CR-L is ultimately able to find a better solution than BB-RL and so BB-RL's benefits are attained at the expense of optimality. BB-RL needs a map of the environment to solve the path planning tasks. Also, unlike C-RL, BB-RL has to expend the computational effort of computing the harmonic functions. However, as noted above, harmonic functions are cheaper to compute than value functions because they are computed in the lower dimensional configuration space, because they can be computed on a coarse grid over the environment, and because they do not involve any optimization. Connolly has also proposed a hardware resistive grid architecture that can compute harmonic functions very rapidly.

This chapter illustrated the idea of using the closed-loop solutions found for suitably designed simple tasks to constrain the policy space of a complex task in order to remove all/most undesired solutions. The difficult question of automatically designing suitable simple tasks is not addressed here. Instead, it is proposed that in some domains, especially in robotics, researchers have already identified sets of closed-loop behaviors that have desired properties. Determining stable behaviors and rules for composing them that are useful across a variety of complex tasks with multi-purpose robots is an active area of research (e.g., Grupen [47, 46]). RL architectures, such as the one described here, offer a technique for using existing closed-loop behaviors as primitives by learning mixing functions to solve new complex tasks.

CHAPTER 9

CONCLUSIONS

The field of reinforcement learning (RL) is at an exciting point in its history. A solid mathematical foundation has been developed for RL algorithms. There are asymptotic convergence proofs for lookup table implementations of all RL algorithms when applied to finite Markovian decision tasks (MDTs). There is now a common framework for understanding some heuristic search methods from artificial intelligence (AI), dynamic programming methods from optimal control, and RL methods from machine learning. This great synthesis of the different approaches and the clear understanding of the different assumptions behind them have come about as a result of work done by several researchers. The theoretical research presented in the first half of this dissertation has contributed to this understanding by providing a uniform framework based on classical stochastic approximation theory for understanding and proving convergence for the different RL algorithms.

As a result of its strong theoretical foundations, the field of RL is gaining acceptance not only among AI researchers interested in building systems that work in real environments, but also among researchers interested in incorporating RL into traditional control theoretic architectures. But along with the increasing acceptance has come the need to apply RL architectures to larger and larger applications. The research on scaling RL presented in the second half of this dissertation was motivated by the applications of tomorrow, e.g., adaptive household robots, multi-purpose software agents residing inside computer networks, adaptive multi-task space exploration robots, etc. Building multi-task, life-long learning agents raises some significant issues for RL. The architectures presented in Chapters 6 , 7 and 8 represent a first step in tackling the complex issues of achieving transfer of training across tasks and maintaining safe performance while learning that are crucial to the success of adaptive multi-task agents. Despite the preliminary nature of the architectures, some general ideas were developed that are likely to outlast the details of the specific architectures.

9.1 Contributions

The section presents a brief summary of the main contributions of this dissertation.

9.1.1 Theory of DP-based Learning

Soft Dynamic Programming: Classical DP algorithms use a backup operator that only takes into account the best action in each state. That can lead to solutions where the agent prefers states with only one very good action over states that have many good actions. This dissertation developed a family of soft backup operators that take into account all the actions available in a state. Soft DP can lead to solutions that are more robust in non-stationary environments than the solutions found by conventional DP. Proofs of convergence were also developed.

Single-Sided Asynchronous Policy Iteration: Classical asynchronous DP algorithms allow the agent to sample in predecessor state space but not in action space. This dissertation developed a new asynchronous algorithm that allows the agent to sample both in predecessor state space as well as in action space. It was shown that the new algorithm converges under more general initial conditions than the related algorithms of modified policy iteration (Puterman and Shin [84]) and the asynchronous algorithms developed by Williams and Baird [127].

Stochastic Approximation Framework for RL: A hitherto unknown connection between stochastic approximation theory and RL algorithms such as TD and Q-learning was developed. The stochastic approximation framework clearly delineates the contribution made by RL researchers to the entire class of algorithms for solving RL tasks.

Sample Backup Versus Full Backup: Sample backups are cheap to compute but return noisy estimates while full backups are expensive to compute but return more informative estimates. It was shown that algorithms using sample backups can be more efficient than algorithms using full backups for MDTs that are nearly deterministic but have a high average branching factor.

Impact of Approximations on Performance: Two separate results were derived that provide partial justification for using function approximation methods other than lookup tables to store and update value functions. The first result defined an upper bound on the worst-case average loss per action when the agent follows a policy greedy with respect to an approximation to the optimal value function. The second result shows that there is a region around the optimal value function such that any value function within that region will yield optimal policies.

9.1.2 Scaling RL: Transfer of Training from Simple to Complex Tasks

Compositional Q-learning: The constrained but useful class of compositionally-structured MDTs was defined. A modular connectionist architecture called CQ-L was developed that does compositional learning by composing the value functions for the elemental MDTs to build the value function for a composite MDT. Transfer of training is achieved by sharing the value functions learned for the elemental tasks across several composite tasks. CQ-L is able to solve automatically the composition problem, i.e., it can figure out which value functions to compose for a composite task without knowing the decomposition of that composite task.

Hierarchical-DYNA: An RL architecture was developed that learns value functions by doing backups in a hierarchy of environment models. The abstract environment model predicts the consequences of executing abstract actions that express intentions of achieving significant states in the environment. It was shown that if the agent is trained on compositionally-structured tasks, doing backups in the abstract model can speed up learning considerably. The definition of abstract actions as closed-loop policies for achieving significant states generalizes the definition of macro-operators as open-loop sequences of actions.

Closed-Loop Policies as Primitive Actions: An architecture that maintains acceptable performance while learning in motion planning problems was developed. The main innovation was in replacing the conventional primitive actions in motion planning problems by actions that select the proportion in which to mix two closed-loop policies found as solutions to two geometric path planning problems.

9.2 Future Work

A large proportion of RL research, including this dissertation, has focused on finite stationary MDTs for two reasons: it has allowed considerable progress in developing the theory of RL, and there are many interesting and challenging RL tasks that fall into that category. Nevertheless, as the range of real-world problems to which RL is applied is extended, both the theory and practice of RL will also have to be extended to deal with the following:

Continuous Domains: A common strategy of researchers currently applying RL to continuous state tasks is to replace the lookup tables of conventional RL architectures by function approximators such as neural networks and nearest neighbor methods. Some researchers handle continuous state spaces by partitioning the state space into a finite number of equivalence classes and then use conventional RL architectures on the reduced and finite state space. Of course, none of the theory developed for finite MDTs yet extends to general continuous state problems.

Non-Markovian Environments: In many real-world problems it is unrealistic to assume that the agent is complex enough, or that the environment is simple enough, to allow the agent's perceptions to return state information. Researchers are currently developing methods that attempt to build state information by either memorizing past perceptions, or by controlling the perceptual system of the agent to generate multiple perceptions (Whitehead and Ballard [126], Lin and Mitchell [67], Chrisman [27], and McCallum [73]). In both cases the hope is that techniques other than RL can be used to convert a non-Markovian problem into a Markovian one so that conventional RL can be applied. State estimation techniques developed in control engineering, e.g., Kalman filters, can also be used in conjunction with RL. Of course, in practice state estimation and control can also be done simultaneously.

Nonstationarity: The issue of changing, or nonstationary, environments has largely been ignored in RL research. One possible approach may be to build robust RL algorithms that find policies producing satisfactory, though perhaps suboptimal, performance in all possible environments. On the other hand, if the environment changes slowly enough, then RL methods, being incremental, can perhaps track those changes.

APPENDIX A

ASYNCHRONOUS POLICY ITERATION

The purpose of this appendix is to prove some Lemma's used in the proof of convergence of the single-sided asynchronous policy iteration (SS-API) algorithm developed in Chapter 3. The notation used in the following is developed in Chapter 3. Throughout we will use the shorthand $(V_{l+h}, \pi_{l+h}) = \{U_k\}_l^{l+h}(V_l, \pi_l)$ for

$$(V_{l+h}, \pi_{l+h}) = U_{l+h}U_{l+h-1}U_{l+h-2}\ldots U_l(V_l, \pi_l).$$

For ease of exposition, define the *identity* operator $I(V_k, \pi_k) = (V_k, \pi_k)$. Further define the operator $B_x = T_x L_x$, $\forall x \in X$.

Fact 1: Consider a sequence of operators $\{U_k\}_l^{l+h}$ such that for $l \leq k \leq (l+h)$, $U_k \in \{B_x \mid x \in X\}$, and $\forall x \in X$, $B_x \in \{U_k\}_l^{l+h}$. Then if $V_l \in \mathcal{V}_u$, $||V_{l+h} - V^*||_\infty \leq \gamma ||V_l - V^*||_\infty$.

Proof: This is a simple extension of a result in Bertsekas and Tsitsiklis [18].

Fact 2: Consider the following algorithm: $(V'_{k+1}, \pi_{k+1}) = U_k(V'_k, \pi_k)$, where $U_k \in \{B_x \mid x \in X\}$. If $V'_0 \in \mathcal{V}_u$, and $\forall x \in X$, $B_x \in \{U_k\}$ infinitely often, then $\lim_{k\to\infty} V'_k = V^*$.

Proof: This result follows from Fact 1 and the contraction mapping theorem. Note that the algorithm stated in Fact 2 is a modified version of asynchronous value iteration that updates both a value function as well as a policy.
Q.E.D.

Fact 3: Consider $(V_{k+1}, \pi_{k+1}) = B_x(V_k, \pi_k)$ and $(V'_{k+1}, \pi'_{k+1}) = B_x(V'_k, \pi'_k)$. If $V^* \geq V_k \geq V'_k$, then $\forall \pi_k, \pi'_k \in \mathcal{P}$, $V_{k+1} \geq V'_{k+1}$.

Proof:

$$\begin{aligned} V'_{k+1}(x) &= \max_{a\in A}\left[R^a(x) + \gamma \sum_{y\in S} P^a(x,y)V'_k(y)\right] \\ &= R^{a'}(x) + \gamma \sum_{y\in X} P^{a'}(x,y)V'_k(y) \qquad \text{for some } a' \in A. \end{aligned}$$

And,

$$\begin{aligned} V_{k+1}(x) = \max_{a\in A}&\left[R^a(x) + \gamma \sum_{y\in X} P^a(x,y)V_k(y)\right] \\ &\geq R^{a'}(x) + \gamma \sum_{y\in X} P^{a'}(x,y)V_k(y) \\ &\geq R^{a'}(x) + \gamma \sum_{y\in X} P^{a'}(x,y)V'_k(y) \\ &= V'_{k+1}(x) \end{aligned}$$

Q.E.D.

Lemma 3: Consider a sequence of operators $\{U_k\}_l^{l+h}$ such that for some arbitrary state x, $\forall l \leq k \leq l+h-1$, $U_k \in \{L_x^a \mid x \in X, a \in A\}$, and $\forall a \in A$, $L_x^a \in \{U_k\}_l^{l+h-1}$, and $U_{l+h} = T_x$. Let $V_l \in \mathcal{V}_u$, and let $(V', \pi') = B_x(V_l, \pi_l)$. Then, $V_{l+h} \geq V'$.

Proof: Let $(V_{l+h-1}, \pi_{l+h-1}) = \{U_k\}_l^{l+h-1}(V_l, \pi_l)$. Then $\forall k$, $l \leq k \leq (l+h-1)$, $V_k = V_l$ and therefore $\pi_{l+h-1}(x)$ will be a greedy action with respect to V_l, i.e., $\{U_k\}_l^{l+h}(V_l, \pi_l) = T_x(V_{l+h-1}, \pi_{l+h-1}) = (V', \pi_{l+h})$.
Q.E.D.

Define $\mathbf{W(x)}$ to be the set of *finite* length sequences of operators that satisfy the following properties:

1. each element $\{w_k\}_0^h \in \mathbf{W(x)}$ has a subsequence $\{w_k\}_0^d$, where $d < h$, and $\forall a \in A$, $L(x, a) \in \{w_k\}_0^d$,
2. $w_h = T(x)$.

Note, that for each state $x \in X$, there is a separate set $\mathbf{W(x)}$.
Lemma 4: In $ARPI$, for any arbitrary state x, consider a sequence $\{U_k\}_l^{l+h} \in \mathbf{W(x)}$. Let $(\hat{V}, \hat{\pi}) = B_x(V_l, \pi_l)$. If $V_l \in \mathcal{V}_u$, then $V_{l+h} \geq \hat{V}$.

Proof: Any element of $W(x)$ applies each element of the set $\{L_x^a | a \in A\}$ followed by one T_x. Intermediate applications of L_y^a where $y \neq x$ will not affect the policy for state x and intermediate applications of any T_y can only increase the value function. The above argument combined with Lemma 3 constitutes an informal proof. A formal proof follows. Let the sequence $\{U'_k\}_l^{l+h-1}$ be obtained by replacing all T operators in $\{U_k\}_l^{l+h-1}$ by the I operator, and let $U'_{l+h} = U_{l+h} = T_x$. The following can be concluded immediately:

1. $V_k \geq V'_k = V'_l = V_l$, for $l \leq k \leq (l+h-1)$, and
2. $V'_{l+h} = \hat{V}$

We will prove that

$$\begin{aligned} V_{l+h}(x) = \ & \max(Q^{V_{l+h-1}}(x, \pi_{l+h-1}(x)), V_{l+h-1}(x)) \\ & \geq V'_{l+h}(x) \\ = \ & \max(Q^{V'_{l+h-1}}(x, \pi_{l+h-1}(x)), V'_{l+h-1}(x)), \end{aligned}$$

thereby showing that $V_{l+h} \geq V'_{l+h} = \hat{V}$.

Because $V_k \geq V'_k$ for $l \leq k \leq l+h-1$, it suffices to show that $Q^{V_k}(x, \pi_k(x)) \geq Q^{V'_k}(x, \pi'_k(x))$, for $l \leq k \leq (l+h-1)$. We will show this by induction on k.
Base Case: By assumption $V_l = V'_l$, and $\pi_l = \pi'_l$. Therefore $Q^{V_l}(x, \pi_l(x)) = Q^{V'_l}(x, \pi'_l(x))$.

Induction Hypothesis: Assume that $Q^{V_m}(x, \pi_m(x)) \geq Q^{V'_m}(x, \pi'_m(x))$, for some m such that $l < m < (l+h-1)$.

We will show that the above inequality holds for $k = m+1$. There are three separate cases to study.

Case 1: $U_{m+1} = T(y)$ for some $y \in X$. Therefore $U'_{m+1} = I$. Then clearly $\pi_{m+1} = \pi_m$ and $\pi'_{m+1} = \pi'_m$. Also, $V_{m+1} \geq V_m = V'_m = V'_{m+1}$. Therefore,

$$Q^{V_{m+1}}(x, \pi_{m+1}(x)) = Q^{V_{m+1}}(x, \pi_m(x)) \geq Q^{V_m}(x, \pi_m(x)) \geq Q^{V'_m}(x, \pi'_m(x)) = Q^{V'_{m+1}}(x, \pi'_{m+1}(x)).$$

Case 2: $U_{m+1} = U'_{m+1} = L^a_y$ for some $y \neq x$. Then $V_{m+1} = V_m$ and $V'_{m+1} = V'_m$. Also $\pi_{m+1}(x) = \pi_m(x)$ and $\pi'_{m+1}(x) = \pi'_m(x)$. Therefore,

$$Q^{V_{m+1}}(x, \pi_{m+1}(x)) = Q^{V_m}(x, \pi_m(x)) \geq Q^{V'_m}(x, \pi'_m(x)) = Q^{V'_{m+1}}(x, \pi'_{m+1}(x)).$$

Case 3: $U_{m+1} = U'_{m+1} = L^a_x$ for some $a \in A$. Then $V_{m+1} = V_m$ and $V'_{m+1} = V'_m$. Therefore $Q^{V_{m+1}}(x, \pi_{m+1}(x)) = Q^{V_m}(x, \pi_{m+1}(x))$, and similarly $Q^{V'_{m+1}}(x, \pi'_{m+1}(x)) = Q^{V'_m}(x, \pi'_{m+1}(x))$. There are two subcases to consider:

1. $\pi'_{k+1} = a$. Then $Q^{V_{m+1}}(x, \pi_{m+1}(x)) = Q^{V_m}(x, \pi_{m+1}(x)) \geq Q^{V_m}(x, a) \geq Q^{V'_m}(x, a) = Q^{V'_{m+1}}(x, \pi'_{m+1}(x))$
2. $\pi'_{k+1} = \pi'_k$. Then $Q^{V_{m+1}}(x, \pi_{m+1}(x)) \geq Q^{V_{m+1}}(x, \pi_m(x)) = Q^{V_m}(x, \pi_m(x)) \geq Q^{V'_m}(x, \pi'_m(x)) = Q^{V'_{m+1}}(x, \pi'_{m+1}(x))$.

Q.E.D.

Define $\mathbf{W}$ to be a set of finite length sequences of operators that satisfy the following property: each element of $\mathbf{W}$ contains *disjoint* subsequences, such that at least one distinct subsequence is in $\mathbf{W}(\mathbf{x})$, $\forall x \in X$.

Lemma 5: Consider any sequence $\{U_k\}_l^{l+h} \in \mathbf{W}$. If $V_l \in \mathcal{V}_u$, then $||V_{l+h} - V^*||_\infty \leq \gamma||V_l - V^*||_\infty$.

Proof: From Lemma 4 and Facts 1 and 3 it can be seen that for any sequence of operators that is an element of $\mathbf{W}$ the result would be a contraction.

Theorem: Given a starting value-policy pair (V_0, π_0), such that $V_0 \in \mathcal{V}_\mathbf{u}$, the ARPI algorithm $(V_{k+1}, \pi_{k+1}) = U_k(V_k, \pi_k)$ converges to (V^*, π^*) provided for each $i \in X$, and for all state-action pairs, $(x, a) \in X \times A$, T_i and L^a_x appear infinitely often in the sequence $\{U_k\}$.

Proof: The infinite sequence $\{U_k\}$ has an infinity of disjoint subsequences that are elements of $\mathbf{W}$. The result that $\lim_{k\to\infty}(V_k, \pi_k) = (V^*, \pi_\infty)$ follows from Lemma 5 and the contraction mapping theorem. It is known that $\exists\epsilon > 0$ such that if $||V - V^*||_\infty \leq \epsilon$ then the greedy policy with respect to V is optimal (Singh [101]). Because the sequence $\{V_k\}$ is non-decreasing it can be concluded that $\pi_\infty \in \{\pi^*\}$.
Q.E.D.

APPENDIX B

DVORETZKY'S STOCHASTIC APPROXIMATION THEORY

After the initial Robbins-Monro paper on stochastic approximation algorithms that proved convergence in probability, several researchers derived stronger results of convergence with probability one and mean-square convergence under conditions that are more general than the ones proposed in the original Robbins-Monro paper. For the purposes of Chapter 4 a result derived by Dvoretzky is the most relevant. Dvoretzky [39] studied a problem more general than finding roots of equations. He studied convergent deterministic iterative processes of the form $V_{n+1} = T_n(V_n)$, where T_n is a deterministic operator. He derived conditions under which the stochastic process $V_{n+1} = T_n(V_n) + D_n$, where D_n is mean-zero noise, would converge to the fixed point of the original deterministic process.

The Robbins-Monro stochastic approximation iteration

$$V_{n+1} = V_n - \rho_n Y(V_n) \tag{B.1}$$

where $Y(V_n) = G(V_n) + \epsilon_n$, where $G(V)$ is deterministic and ϵ_n is mean-zero noise, can be rewritten in the form studied by Dvoretzky as follows:

$$V_{n+1} = (V_n - \rho_n G(V_n)) + \rho_n(G(V_n) - Y(V_n)), \tag{B.2}$$

where $T_n(V_n) = V_n - \rho_n G(V_n)$, and $D_n = G(V_n) - Y(V_n)$.

Let V_n be measurable random variables assuming values in a (finite or infinite dimensional) (real) Hilbert space $\mathcal{H}$. Let T_n be function for $\mathcal{H}$ to $\mathcal{H}$. Let $|.|$ denote the norm in $\mathcal{H}$.

Consider the basic recurrence scheme:

$$V_{n+1} = T_n(V_n) + D_n. \tag{B.3}$$

Theorem:. Let α_n, β_n and ϕ_n be non-negative real numbers satisfying

$$\begin{aligned} &\lim_{n=\infty} \alpha_n = 0 \\ &\sum_{i=1}^{\infty} \beta_i < \infty \\ &\sum_{i=1}^{\infty} \phi_i = \infty, \end{aligned} \tag{B.4}$$

Let V^* be a point of $\mathcal{H}$ and T_n satisfy the following equation

$$|T_n(V_n) - V^*| \leq \max\left[\alpha_n, (1 + \beta_n - \phi_n)|V_n - V^*|\right] \tag{B.5}$$

be satisfied for all $V_n \in \mathcal{H}$.

Then the iteration B.3 together with the conditions

$$E\{D_n|V_n\} = 0 \tag{B.6}$$

$\forall n$, and

$$\sum_{i=1}^{\infty} E\{D_n^2\} < \infty \tag{B.7}$$

imply

$$P\{\lim_{n=\infty} V_n = V^*\} = 1. \tag{B.8}$$

If moreover, $E\{V_1^2\} < \infty$ then also

$$\lim_{n=\infty} E(V_n - V^*)^2 = 0 \tag{B.9}$$

APPENDIX C

PROOF OF CONVERGENCE OF Q-VALUE ITERATION

This appendix proves that the full backup Q-value iteration operator, $B(Q)$, defined in Chapter 4 is a contraction operator. Therefore we have to show that if $Q_{k+1} = B(Q_k)$, then

$$\max_{x,a} |Q_{k+1}(x,a) - Q^*(x,a)| \leq \alpha \max_{x,a} |Q_k(x,a) - Q^*(x,a)|$$

where $0 < \alpha < 1$. Let $M = \max_{x,a} |Q_k(x,a) - Q^*(x,a)|$. Then $\forall (x,a)$, $Q^*(x,a) - M \leq Q_k(x,a) \leq Q^*(x,a) + M$. Therefore, $\forall (x,a)$

$$\begin{aligned}
Q_{k+1}(x,a) &= R^a(x) + \gamma \sum_{y \in X} P^a(x,y) V_k(y) \\
&\leq R^a(x) + \gamma \sum_{y \in X} P^a(x,y)(V_k^*(y) + M) \\
&\leq R^a(x) + \gamma M + \sum_{y \in X} P^a(x,y) V_k^*(y) \\
&\leq Q^*(x,a) + \gamma M.
\end{aligned}$$

Similarly one can show that $\forall (x,a)$, $Q_{k+1}(x,a) \geq Q^*(x,a) - \gamma M$.

APPENDIX D

PROOF OF PROPOSITION 2

This appendix proves Proposition 2 stated in Chapter 6. Consider an elemental deterministic Markovian decision task (MDT) T_i and let its final state be denoted x_g. Let π_i be the optimal policy for task T_i. The payoff function for task T_i is $R_i(x,a) = \sum_{y \in X} P_{xy}(a) r_i(y) - c(x,a)$, for all $x \in X$ and $a \in A$. By assumptions A1–A4 (Chapter 6) we know that the reward $r_i(x) = 0$ for all $x \neq x_g$. Thus, for any state $y \in X$ and action $a \in A$,

$$Q^{T_i}(y,a) \;\; = \;\; r_i(x_g) - c(y,a) - \Phi(y,x_g) + c(y,\pi_i(y)),$$

where $c(y, \pi_i(y))$ is the cost of executing the optimal action in state y, and $\Phi(y, x_g)$ is the expected cost of going from state y to x_g under policy π_i.

Consider a composite task C_j that satisfies assumptions A1–A4 given in Section 6.1.2 for Chapter 6 and w.l.o.g. assume that for $C_j = [T(j,1)T(j,2)\cdots T(j,k)]$, $\exists 1 \leq l \leq k$, such that $T(j,l) = T_i$. Let the set $L \subset X'$ be the set of all $x' \in X'$ that satisfy the property that the augmenting bits corresponding to the tasks before T_i in the decomposition of C_j are equal to one and the rest are equal to zero. Let $y' \in L$ be the state that has the projected state $y \in X$. Let $x'_g \in X'$ be the state formed from $x_g \in X$ by setting to one the augmenting bits corresponding to all the subtasks before and including subtask T_i in the decomposition of C_j, and setting the other augmenting bits to zero. Let π'_j be an optimal policy for task C_j. $r_j(x')$ is the reward for state $x' \in X'$ while performing task C_j. Then by assumptions A1–A4, we know that $r_j(x') = 0$ for all $x' \in L$ and that $c(x',a) = c(x,a)$.

By the definition of C_j, the agent has to navigate from state y' to state x'_g to accomplish subtask T_i. Let $\Phi'(y', x'_g)$ be the expected cost of going from state y' to state x'_g under policy π'_j. Then, given that $T(j,l) = T_i$,

$$Q^{C_j}_{T_i}(y',a) \;\; = \;\; Q^{C_j}_{T(j,l+1)}(x'_g, b) + r_j(x'_g) - c(y',a) - \Phi'(y',x'_g) + c(y',\pi'_j(y')),$$

where $b \in A$ is an optimal action for state x'_g while performing subtask $T(j, l+1)$ in task C_j. Clearly, $\Phi'(y', x'_g) = \Phi(y, x_g)$, for if it were not, either policy π_i would not be optimal for task T_i, or given the independence of the solutions of the subtasks the policy π'_j would not be optimal for task C_j. For the same reason, $c(y, \pi_i(y)) = c(y', \pi'_j(y'))$. Define $K(C_j, l) \equiv Q^{C_j}_{T(j,l+1)}(x'_g, b) + r_j(x'_g) - r_i(x_g)$. Then

$$Q^{C_j}_{T_i}(y',a) \;\; = \;\; Q^{T_i}(y,a) + K(C_j,l).$$

Q.E.D.

D.1 Parameter values for Simulations 1, 2 and 3

For all three simulations, the initial values for the lookup tables implementing the Q-modules were random values in the range 0.0–0.5, and the initial values for the gating module lookup table were random values in the range 0.0–0.4. For all three simulations, the variance of the Gaussian noise, σ, was 1.0 for all Q-modules.

For Simulation 1, the parameter values for the both CQ-L and the one-for-one architectures were: $\alpha_Q = 0.1$, $\alpha_b = 0.0$, $\alpha_g = 0.3$. The policy selection parameter, β, started at 1.0 and was incremented by 1.0 after every 100 trials.

For Simulation 2, the parameter values for CQ-L were: $\alpha_Q = 0.015$, $\alpha_b = 0.0001$, $\alpha_g = 0.025$, and β was incremented by 1.0 after every 200 trials. For the one-for-one architecture, the parameter values were: $\alpha_Q = 0.01$ and β was incremented by 1.0 after every 500 trials.[1]

For Simulation 3, the parameter values, $\alpha_Q = 0.1$, $\alpha_b = 0.0001$, and $\alpha_g = 0.01$ were used, and β was incremented by 1.0 every 25 trials.

[1] Incrementing the β values faster did not help the one-for-one architecture.

APPENDIX E

CONDITIONS FOR ROBUST REINFORCEMENT LEARNING IN MOTION PLANNING

Chapter 7 developed a RL architecture for solving the motion planning problem that learns to mix two closed-loop policies found for the simpler path-planning problem. Let ϕ_D be the Dirichlet closed-loop policy, ϕ_N be the Neumann closed-loop policy, and $k(x)$ be the mixing function learned by the RL algorithm. This appendix derives conditions under which the policy space of the RL algorithm developed in Chapter 7 is guaranteed to exclude all unsafe policies (see Chapter 7 for further detail).

Any closed-loop policy is a convex combination of the Dirichlet and Neumann policies derived from the Dirichlet and Neumann harmonic potential functions. Let L denote the surface whose gradients at any point are given by the closed-loop policy under consideration. Then for there to be no local minima in L, the gradient of L should not vanish in the workspace, i.e., $(1.0 - k(x))\nabla\phi_D(\hat{x}) + k(x)\nabla\phi_N(\hat{x}) \neq 0$. The only way that can happen is if $\forall i$

$$\frac{k(x)}{1.0 - k(x)} = \left[\frac{-\nabla\phi_D(\hat{x})}{\nabla\phi_N(\hat{x})}\right]_i, \tag{E.1}$$

where $[.]_i$ is the i^{th} component of vector $[.]$. The algorithm can explicitly check for that possibility and prevent it. Alternatively, note that due to the finite precision in any practical implementation, it is extremely unlikely that Equation E.1 will hold in any state. Also note that $\pi(s)$ for any point s on the boundary will always point away from the boundary because it is the convex sum of two vectors, one of which is normal to the boundary, and the other is parallel to the boundary.

APPENDIX F

BRIEF DESCRIPTION OF NEURAL NETWORKS

An artificial neural network or connectionist network is a computational model that is a directed graph composed of nodes and connections between nodes. Each connection is capable of transmitting a real number from the predecessor node to the successor node. Each node is capable of performing some simple, usually fixed, computation on the signals that come in on the incoming connections. The computation performed by a node is called its activation function and the result of the computation is called the activation of the node. The signal carried by a connection is the product of the activation of the predecessor node and a scalar parameter, called a weight, associated with each connection. In addition, each unit can also store additional parameters (weights) that are used in the activation function computation. The parameters of the network can be adapted. Several learning rules based on gradient descent have been developed for adapting the parameters. A good place for learning about artificial neural networks and about learning rules is Rumelhart *et al.*'s [89] 1986 book on parallel distributed processing.

A subset of nodes are called input nodes and their activations are set from the outside. Some special nodes are called output nodes and their activation can be read by the outside world. Only a special class of networks, called feedforward networks (Figure F.1) are used in this dissertation. In feedforward networks the nodes are arranged in layers, starting with the input layer and ending with an output layer. The layers inbetween the input and output layers are called hidden layers because they are not accessible to the outside world. Each node in a layer receives connections from all nodes in the layer below and sends connections to all nodes in the layer above.

For the purpose of this dissertation, neural networks can be thought of as function approximators for storing functions. A particular neural network with fixed weights implements a function that assigns an output vector to every input vector. The output vector is determined by setting the activations of the input units to the components of the input vector and forward propagating the activations and reading off the values at the output nodes.

Two types of activation functions are used in this dissertation: the linear function, and the radial basis function. A node that uses a linear activation function is called a linear unit. Similarly a node that uses the radial basis activation function is called a radial basis unit. A linear unit's activation is equal to the sum of the inputs on the incident connections. The radial basis unit's activation is equal to $e^{\frac{(x-w)^2}{2\sigma}}$, where x is the vector of inputs to the unit, w is the vector of weights of the incident connections, and σ is the vector of additional parameters stored in the unit.

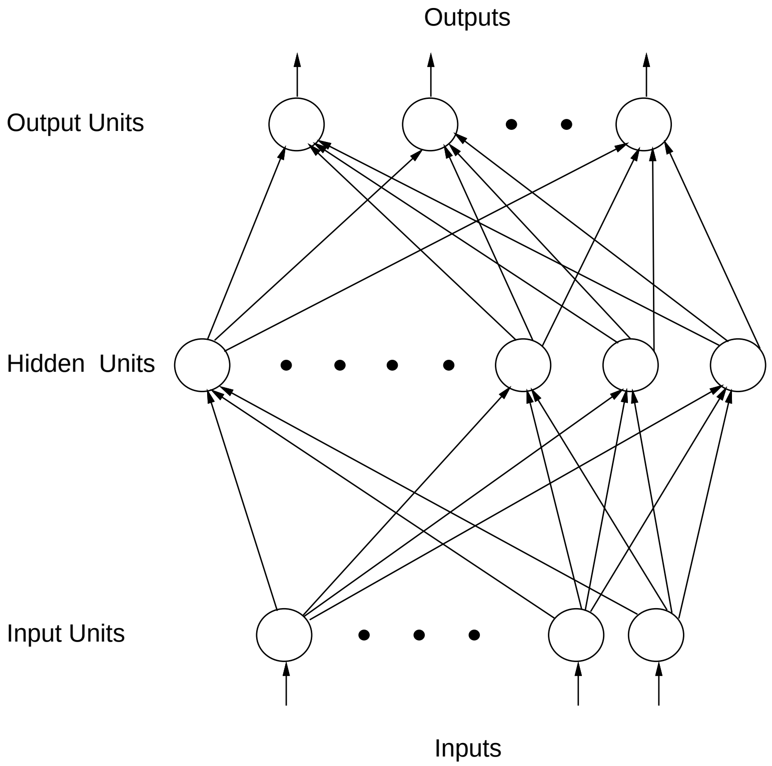

Figure F.1 A Feedforward Connectionist Network. The nodes represent neurons or units that compute simple functions of the inputs coming in on the edges. The edges in the graph store scalar parameters or weights that determine the function represented by the network.

BIBLIOGRAPHY

[1] P.E. Agre. *The Dynamic Structure of Everyday Life.* PhD thesis, M.I.T., 1988.

[2] C.W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems.* PhD thesis, University of Massachusetts, Amherst, MA, 1986.

[3] K.J. Astrom and B. Wittenmark. *Adaptive Control.* Addison-Wesley, 1989.

[4] J.R. Bachrach. A connectionist learning control architecture for navigation. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 457–463, San Mateo, CA, 1991. Morgan Kaufmann.

[5] E. Barnard. Temporal-difference methods and markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365, 1993.

[6] A.G. Barto. personal communication.

[7] A.G. Barto. Connectionist learning for control: An overview. In T. Miller, R.S. Sutton, and P.J. Werbos, editors, *Neural Networks for Control.* MIT Press, Cambridge, MA. To appear.

[8] A.G. Barto. Learning to act: A perspective from control theory, July 1992. AAAI invited talk.

[9] A.G. Barto and P. Anandan. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360–375, 1985.

[10] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. Technical Report 93-02, University of Massachusetts, Amherst, MA, 1993. Submitted to: AI Journal.

[11] A.G. Barto and S.P. Singh. Reinforcement learning and dynamic programming. In *Sixth Yale Workshop on Adaptive and Learning Systems*, pages 83–88, New Haven, CT, 1990.

[12] A.G. Barto and S.P. Singh. On the computational economics of reinforcement learning. In *Proceedings of the 1990 Connectionist Models Summer School*, San Mateo, CA, Nov. 1990. Morgan Kaufmann.

[13] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE SMC*, 13:835–846, 1983.

[14] A.G. Barto, R.S. Sutton, and C. Watkins. Learning and sequential decision making. In M. Gabriel and J.W. Moore, editors, *Learning and Computational Neuroscience.* MIT Press, Cambridge, MA, 1990.

[15] A.G. Barto, R.S. Sutton, and C. Watkins. Sequential decision problems and neural networks. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 686–693, San Mateo, CA, 1990. Morgan Kaufmann.

[16] R.E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[17] D.P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[18] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[19] J.R. Blum. Multidimensional stochastic approximation method. *Ann. of Math. Stat.*, 25, 1954.

[20] S.J. Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems 5*, pages 295–302, San Mateo, CA, 1993. Morgan Kaufmann.

[21] J. Bridle. *Probablistic interpretation of feedforward classification network outputs with relationships to statistical pattern recognition.* Springer-Verlag, New York, 1989.

[22] R.A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation*, 1:23–262, 1989.

[23] R.A. Brooks. Intelligence without reason. A.I. Memo. 1293, M.I.T., April 1991.

[24] R.A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[25] R.R. Bush and F. Mosteller. *Stochastic Models for Learning.* Wiley, New York, 1955.

[26] D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, 1991.

[27] L. Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI-92*, 1992.

[28] J.A. Clouse and P.E. Utgoff. A teaching method for reinforcement learning. In *Machine Learning: Proceedings of teh Ninth International Conference*, pages 92–101, San Mateo, CA, 1992. Morgan Kaufmann.

[29] C.I. Connolly. Applications of harmonic functions to robotics. In *The 1992 International Symposium on Intelligent Control.* IEEE, August 1992.

[30] C.I. Connolly and R.A. Grupen. Harmonic control. In *The 1992 International Symposium on Intelligent Control.* IEEE, August 1992.

[31] C.I. Connolly and R.A. Grupen. On the applications of harmonic functions to robotics. *Journal of Robotic Systems*, in press 1993.

[32] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[33] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8(3/4):341–362, May 1992.

[34] P. Dayan. Improving generalization for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, July 1993.

[35] P. Dayan and G.E. Hinton. Feudal reinforcement learning. In S.J. Hanson, J.D. Cowan, and C.L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan-Kaufmann, 1992.

[36] T.L. Dean and M.P. Wellman. *Planning and Control*. Morgan Kauffman, 1991.

[37] J. Denker, D. Schwartz, B. Wittner, S. Solla, R. Howard, L. Jackel, and J. Hopfield. Large automatic learning, rule extraction, and generalization. *Complex Systems*, 1:877–922, 1987.

[38] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[39] A. Dvoretzky. On stochastic approximation. In *Proceedings of the third Berkeley symposium on Mathematical Statisitics and Probability*, pages 39–55, 1956.

[40] S.E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 524–532, San Mateo, CA, 1990. IEEE, Morgan Kaufmann.

[41] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalised robot plans. *Artificial Inteligence*, 3:189–208, 1972.

[42] W. Fun and M.I. Jordan. The moving basin: Effective action-search in adaptive control, 1992. submitted to Neural Computation.

[43] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, 1989.

[44] G.C. Goodwin and K.W. Sin. *Adaptive Filtering Prediction and Control*. Englewood Cliffs, 1984.

[45] J.J. Grefenstette. Incremental learning of control strategies with genetic algorithms. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 340–344, Ithaca, New York, 1989. Morgan Kaufmann.

[46] R. Grupen. Planning grasp strategies for multifingered robot hands. In *Proceedings of the 1991 Conference on Robotics and Automation*, pages 646–651, Sacramento, CA, April 1991. IEEE.

[47] R. Grupen and R. Weiss. Integrated control for interpreting and manipulating the environment. *Robotica*, page accepted for publication, 1992.

[48] V. Gullapalli. *Reinforcement Learning and its application to control.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1992.

[49] G.H. Hardy, J.E. Littlewood, and G. Polya. *Inequalities.* University Press, Cambridge, England, 2 edition, 1952.

[50] P.E. Hart, N.J. Nilsson, and B. Rapahel. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Sys. Sci. Cybernet.*, SSC-4(2):100–107, 1968.

[51] J.H. Holland, K.J. Holoyak, R.E. Nisbet, and P.R. Thagard. *Induction: Processes of Inference, Learning and Discovery.* MIT Press, 1987.

[52] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[53] T. Jaakkola, M.I. Jordan, and S.P. Singh. Stochastic convergence of iterative DP algorithms, 1993. Submitted to Neural Computation.

[54] R.A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1, 1988.

[55] R.A. Jacobs. *Task decomposition through competition in a modular connectionist architecture.* PhD thesis, COINS dept Univ. of Massachusetts, Amherst, Mass. U.S.A., 1990.

[56] R.A. Jacobs, M.I. Jordan, S.J. Nowlan, and G.E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1), 1991.

[57] M.I. Jordan and R.A. Jacobs. Learning to control an unstable system with forward modeling. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufmann.

[58] M.I. Jordan and R.A. Jacobs. Hierarchies of adaptive experts. In J.E. Moody, S.J. Hanson, and R.P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 985–992. Morgan Kaufmann, 1992.

[59] M.I. Jordan and D.E. Rumelhart. Internal world models and supervised learning. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 70–74, San Mateo, CA, 1991. Morgan Kaufmann.

[60] L.P. Kaelbling. *Learning in Embedded Systems.* PhD thesis, Stanford University, Department of Computer Science, Stanford, CA, 1990. Technical Report TR-90-04.

[61] H. Kesten. Accelerated stochastic approximation. *Ann. Math. Statist.*, 29:41–59, 1958.

[62] D.E. Kirk. *Optimal control theory: an introduction.* Englewood Cliffs, 1970.

[63] K.Narendra and M.A.L. Thathachar. *Learning Automata: An Introduction.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[64] R.E. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* Pitman Publishers, Massachusetts, 1985.

[65] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[66] L.J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993.

[67] L.J. Lin and T.M. Mitchell. Reinforcement learning with hidden states. In *In Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1992.

[68] P. Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back.* MIT/Elsevier, 1991.

[69] P. Maes and R. Brooks. Learning to coordinate behaviours. In *Proceedings of the Eighth AAAI*, pages 796–802. Morgan Kaufmann, 1990.

[70] S. Mahadevan. Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In *Machine Learning: Proceedings of the Ninth International Conference*, pages 290–299. Morgan Kaufmann, 1992.

[71] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. Technical report, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, NY, 1990.

[72] M.J. Mataric. A comparative analysis of reinforcement learning methods. Technical report, M.I.T., 1991. A.I. Memo No.1322.

[73] R.A. McCallum. Overcoming incomplete perception with utile distinction memory. In P. Utgoff, editor, *Machine Learning: Proceedings of the Tenth International Conference*, pages 190–196. Morgan Kaufmann, 1993.

[74] D.V. McDermott. Planning and acting. *Cognitive Science*, 2, 1978.

[75] T.M. Mitchell and S.B. Thrun. Explanation-based neural network learning for robot control. In S.J. Hanson, J.D. Cowan, and C.L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 287–294. Morgan-Kaufmann, 1992.

[76] Tom M. Mitchell, Richard Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[77] A.W. Moore. personal communication.

[78] A.W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In L.A. Birnbaum and G.C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 333–337, San Mateo, CA, 1991. Morgan Kaufmann.

[79] A.W. Moore and C.G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13(1), October 1993.

[80] N.J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[81] S.J. Nowlan. Competing experts: An experimental investigation of associative mixture models. Technical Report CRG-TR-90-5, Department of Computer Sc., Univ. of Toronto, Toronto, Canada, 1990.

[82] J. Peng and R.J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, Spring 1993.

[83] M.L. Puterman and S.L. Brumelle. The analytic theory of policy iteration. In *Dynamic Programming and its Applications*, New York, 1978. Academic Press.

[84] M.L. Puterman and M.C. Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11), July 1978.

[85] R.L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34:77–96, 1988.

[86] H. Robbins and S. Monro. A stochastic approximation method. *Ann. Math. Stat.*, 22:400–407, 1951.

[87] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, 1983.

[88] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations.* Bradford Books/MIT Press, Cambridge, MA, 1986.

[89] D.E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol.1: Foundations, Vol. 2: Psychological and Biological models.* Bradford Books/MIT Press, Cambridge, MA, 1986.

[90] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[91] M. Sato, K. Abe, and H. Takeda. Learning control of finite markov chains with unknown transition probabilities. *IEEE Transactions on Automatic Control*, 27:502–505, 1982.

[92] L. Schmetterer. Stochastic approximation. In *Proceedings of the fourth Berkeley Symposium on Mathematics and Probability*, pages 587–609, 1960.

[93] J.H. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 222–227, Cambridge, MA, 1991. MIT Press.

[94] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth Machine Learning Conference*, 1993.

[95] S.P. Singh. Transfer of learning across compositions of sequential tasks. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 348–352, San Mateo, CA, 1991. Morgan Kaufmann.

[96] S.P. Singh. The efficient learning of multiple task sequences. In J.E. Moody, S.J. Hanson, and R.P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 251–258, San Mateo, CA, 1992. Morgan Kauffman.

[97] S.P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, San Jose,CA, July 1992. AAAI Press/MIT Press.

[98] S.P. Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth Machine Learning Conference*, pages 406–415, 1992.

[99] S.P. Singh. Transfer of learning by composing solutions for elemental sequential tasks. *Machine Learning*, 8(3/4):323–339, May 1992.

[100] S.P. Singh. New reinforcement learning algorithms for maximizing average payoff, 1993. Submitted.

[101] S.P. Singh. Soft dynamic programming algorithms: Convergence proofs, 1993. Poster at CLNL93.

[102] S.P. Singh, A.G. Barto, M.I. Jordan, and T. Jaakkolla. Understanding reinforcement learning. in preparation.

[103] S.P. Singh and R.C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*. to appear.

[104] B.F. Skinner. *The Behavior of Organisms: An experimental analysis*. D. Appleton Century, New York, 1938.

[105] R.S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.

[106] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[107] R.S. Sutton. Integrating architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. of the Seventh International Conference on Machine Learning*, pages 216–224, San Mateo, CA, 1990. Morgan Kaufmann.

[108] R.S. Sutton. Planning by incremental dynamic programming. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 353–357, San Mateo, CA, 1991. Morgan Kaufmann.

[109] R.S. Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose,CA, July 1992. AAAI Press/MIT Press.

[110] R.S. Sutton, A.G. Barto, and R.J. Williams. Reinforcement learning is direct adaptive optimal control. In *Proceedings of the American Control Conference*, pages 2143–2146, Boston, MA, 1991.

[111] R.S. Sutton and B. Pinette. The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, 1985.

[112] G.J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–277, May 1992.

[113] S.B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, 1992.

[114] S.B. Thrun and K.Möller. Active exploration in dynamic environments. In J.E. Moody, S.J. Hanson, and R.P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, San Mateo, CA, 1992. Morgan Kauffman.

[115] J. Tsitsiklis. Asynchronous stochastic approximation and Q-learning, February 1993. Submitted.

[116] P.E. Utgoff and J.A. Clouse. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth Annual Conference on Artificial Intelligence*, pages 596–600, San Mateo, CA, 1991. Morgan Kaufmann.

[117] R.A. Grupen V. Gullapalli, J. Coelho and A.G. Barto. Learning to grasp using a multi-fingered hand. In preparation.

[118] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge Univ., Cambridge, England, 1989.

[119] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3/4):279–292, May 1992.

[120] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[121] P.J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(1):7–20, 1987.

[122] P.J. Werbos. Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3(2):179–189, 1990.

[123] P.J. Werbos. Neurocontrol and related techniques. In A.J. Maren, editor, *Handbook of Neural Computer Applications*. Academic Press, 1990.

[124] P.J. Werbos. Approximate dynamic programming for real-time control and neural modelling. In D.A. White and D.A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 493–525. Van Nostrand Reinhold, 1992.

[125] S.D. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD thesis, University of Rochester, 1992.

[126] S.D. Whitehead and D.H. Ballard. Active perception and reinforcement learning. In *Proc. of the Seventh International Conference on Machine Learning*, Austin, TX, June 1990. M.

[127] R.J. Williams and L.C. Baird. A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pages 96–101, 1990.

[128] R.C. Yee. Abstraction in control learning. Technical Report COINS Technical Report 92-16, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, 1992. A dissertation proposal.

[129] R.C. Yee, S. Saxena, P.E. Utgoff, and A.G. Barto. Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 882–888, Cambridge, MA, 1990.