Reinforcement Learning for Robots Using Neural Networks

Long-Ji Lin

January 6, 1993
CMU-CS-93-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A Dissertation
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was supported in part by Fujitsu Laboratories Ltd. and in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Fujitsu Laboratories Ltd. or the U.S. Government.

Keywords: Machine learning, reinforcement learning, artificial neural network, mobile robot, control, credit assignment, temporal difference, action model, teaching, hierarchical learning, abstraction, hidden state

Abstract

Reinforcement learning agents are adaptive, reactive, and self-supervised. The aim of this dissertation is to extend the state of the art of reinforcement learning and enable its applications to complex robot-learning problems. In particular, it focuses on two issues. First, learning from sparse and delayed reinforcement signals is hard and in general a slow process. Techniques for reducing learning time must be devised. Second, most existing reinforcement learning methods assume that the world is a Markov decision process. This assumption is too strong for many robot tasks of interest.

This dissertation demonstrates how we can possibly overcome the slow learning problem and tackle non-Markovian environments, making reinforcement learning more practical for realistic robot tasks:

- Reinforcement learning can be naturally integrated with artificial neural networks to obtain high-quality generalization, resulting in a significant learning speedup. Neural networks are used in this dissertation, and they generalize effectively even in the presence of noise and a large number of binary and real-valued inputs.

- Reinforcement learning agents can save many learning trials by using an action model, which can be learned on-line. With a model, an agent can mentally experience the effects of its actions without actually executing them. Experience replay is a simple technique that implements this idea, and is shown to be effective in reducing the number of action executions required.

- Reinforcement learning agents can take advantage of instructive training instances provided by human teachers, resulting in a significant learning speedup. Teaching can also help learning agents avoid local optima during the search for optimal control. Simulation experiments indicate that even a small amount of teaching can save agents many learning trials.

- Reinforcement learning agents can significantly reduce learning time by hierarchical learning— they first solve elementary learning problems and then combine solutions to the elementary problems to solve a complex problem. Simulation experiments indicate that a robot with hierarchical learning can solve a complex problem, which otherwise is hardly solvable within a reasonable time.

- Reinforcement learning agents can deal with a wide range of non-Markovian environments by having a memory of their past. Three memory architectures are discussed. They work reasonably well for a variety of simple problems. One of them is also successfully applied to a nontrivial non-Markovian robot task.

The results of this dissertation rely on computer simulation, including (1) an agent operating in a dynamic and hostile environment and (2) a mobile robot operating in a noisy and non-Markovian environment. The robot simulator is physically realistic. This dissertation concludes that it is possible to build artificial agents that can acquire complex control policies effectively by reinforcement learning.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Reinforcement Learning for Robots Using Neural Networks

LONG-JI LIN

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Tom M. Mitchell
THESIS COMMITTEE CHAIR

December 15, 1992
DATE

DEPARTMENT HEAD

12/15/92
DATE

APPROVED:

DEAN

12/15/92
DATE

Acknowledgements

I would like to thank my advisor, Tom Mitchell. Tom has not only patiently given me encouragement, advice and criticism for years, he has also taught me the way to become a good researcher. I am also grateful to Rich Sutton for showing interests to my work and providing me with his expertise in the field of reinforcement learning. Through several fruitful discussions, Rich helped me clarify some existing ideas as well as develop new ones for this dissertation.

I thank Tom Mitchell, Rich Sutton, Chuck Thorpe and Alex Waibel for their participation on my thesis committee and for their detailed and insightful comments on a draft of this dissertation. I also thank Nils Nilsson, Sebastian Thrun, Jeffrey Schlimmer, Lonnie Chrisman, Ryusuke Masuoka, and Chi-Ping Tsang for patiently reading drafts of my papers that become part of this dissertation. The valuable comments they provided have helped the clarity of this dissertation considerably. The feedback from the discussions with Ronald Williams, Chris Atkeson, Sven Koenig, Rich Caruana, and Ming Tan also has contributed to the success of this dissertation.

I am grateful to Reid *Simmons. Working with Reid on autonomous* mobile robots was one of my most enjoyable experiences during my graduate study. The knowledge I learned from working with him has been very valuable to my thesis research on robot learning.

I am especially grateful to my wife, Huey-Shing, for her constant love and support during the many years at Carnegie Mellon University. Finally, I thank my parents and parents-in-law for supporting me to pursue a PhD degree in the United States.

Contents

Abstract i

Acknowledgements iii

1 Introduction 1

1.1 The Thesis 2

1.2 Reinforcement Learning Paradigm 2

1.3 Scaling-Up Issues 5

1.4 Dissertation Overview and Main Results 7

2 Background and Related Work 11

2.1 Temporal Credit Assignment and Temporal Difference 11

2.2 Reinforcement Learning Algorithms 13

2.3 Generalization 14

2.4 Neural Networks 15

2.5 Exploration 17

2.6 Using Action Models 19

2.7 Learning Action Models 20

2.8 Active Perception 21

2.9 Other Related Work 21

2.10 Integration 22

3 Frameworks for Speeding Up Learning 23

3.1 Introduction 23

3.2 V-function and Q-function 24

3.3 Framework AHCON 25

3.4 Framework QCON 28

3.5 Experience Replay: AHCON-R and QCON-R 29

3.5.1 Replaying Only Policy Actions 30

3.5.2 Over-Replay 31

3.6 Using Action Models: AHCON-M and QCON-M 31

3.6.1 Framework AHCON-M 31

3.6.2 Framework QCON-M 33

3.7 Teaching: AHCON-T and QCON-T 34

3.7.1 Comparison with Supervised Learning 34

3.7.2 Over-Replay 35

3.8 A Note: AHC-learning vs. Q-learning 35

3.9 Summary 36

4 Scaling Up: Survival Task **37**

4.1 A Dynamic Environment 37

4.2 Learning Agents 39

4.2.1 Reinforcement Signals 39

4.2.2 Input Representations 39

4.2.3 Action Representations 41

4.2.4 Output Representations 41

4.2.5 Action Models 42

4.2.6 Active Exploration 42

4.2.7 Other Details 43

4.3 Experimental Results 43

4.3.1 Experimental Designs 44

4.3.2 Study 1: Using Global Action Representation 44

4.3.3 Study 2: Using Local Action Representation 44

4.3.4 Observations 46

4.4 Discussion 46

4.4.1 AHC-agents vs. Q-agents 48

4.4.2 Effects of Experience Replay 48
4.4.3 Effects of Using Action Models 49
4.4.4 Effects of Teaching 49
4.4.5 Experience Replay vs. Using Action Models 50
4.4.6 Why Not Perfect Performance? 51
4.5 Summary 52

5 Scaling Up: Mobile Robot Tasks 53
5.1 Using TD(λ) 53
5.1.1 Choice of λ 55
5.1.2 Training Modes and Conditions 58
5.2 A Mobile Robot Simulator 58
5.2.1 The Robot and Its Sensors 58
5.2.2 The Robot's Tasks 60
5.3 The Learning Robot 60
5.3.1 Reinforcement Functions 61
5.3.2 Input Representations 62
5.3.3 Q-nets 62
5.3.4 Exploration 63
5.3.5 Non-Policy Actions 63
5.4 Experimental Designs and Results 63
5.4.1 Experiment 1: Effects of Teaching 64
5.4.2 Experiment 2: Effects of TD(λ) 69
5.4.3 Experiment 3: Combining Supervised Learning and Reinforcement Learning 69
5.4.4 Experiment 4: Single-layer Q-nets 73
5.5 Discussion 73
5.6 Summary 75

6 Hierarchical Learning 77
6.1 Introduction 77
6.2 A Model and Complexity A^nalysis 79

6.2.1 Complexity of Non-Hierarchical Reinforcement Learning 79
6.2.2 A Model of Hierarchical Reinforcement Learning 80
6.2.3 Multiple Levels of Abstraction 84
6.2.4 Summary of the Analysis 85
6.3 Hierarchical Learning in Practice 87
6.3.1 Summary of the Technique 87
6.3.2 Reset Operators 87
6.4 The Learning Domain 88
6.4.1 Task Decomposition 88
6.4.2 Input Representation 90
6.4.3 Applicable Elementary Behaviors 90
6.5 Behaviors That Last Forever 91
6.5.1 Behavior-Based Exploration 94
6.6 Experimental Designs and Results 95
6.6.1 Experiment 1: Non-Hierarchical Learning 96
6.6.2 Experiment 2: Basics 97
6.6.3 Experiment 3: Persistence Exploration Rule 97
6.6.4 Experiment 4: Very Long Reward Delay 99
6.6.5 Experiment 5: Simultaneous Learning of Behaviors 101
6.7 Discussion 101
6.8 Related Work 103
6.9 Summary 104

7 Learning with Hidden States **107**
7.1 Introduction 107
7.2 Active Perception and Hidden State 108
7.3 Three Memory Architectures 109
7.4 OAON Network Architectures 112
7.5 Experimental Designs and Results 114
7.5.1 Task 1: 2-Cup Collection 115
7.5.2 Task 2: Task 1 With Random Features 119

7.5.3 Task 3: Task 1 With Control Errors 119

7.5.4 Task 4: Pole Balancing 121

7.6 Discussion 123

7.6.1 Problem Parameters 123

7.6.2 Architecture Characteristics 124

7.7 Related Work 126

7.8 Summary 128

8 Integration of Capabilities 131

8.1 The Learning Domain 131

8.2 Experiment 1: Memoryless Robot 132

8.3 Experiment 2: Robot with Recurrent Q-nets 132

8.4 Experiment 3: Unexpected Environmental Changes 134

8.5 Discussion 134

***9 Conclusion* 137**

9.1 Contributions 139

9.2 Future Work 140

9.3 Closing 141

A Algorithms Used in Chapter 4 143

A.1 Algorithm for Choosing Enemy Actions 143

A.2 Algorithm for Choosing Lessons for Replay 143

B Parameter Settings Used in Chapter 7 145

C The Pole Balancing Problem 147

Chapter 1

Introduction

This dissertation addresses the problem of automatically acquiring *control policies* by robots for task achieving. Much of the previous work on robot learning assumes the availability of either complete domain knowledge (for example, Mitchell's Theo-agent [45]) or teacher-provided training instances (for example, Pomerleau's autonomous vehicle based on neural networks [55]). It is often hard for humans to create a complete (or even sufficiently good) knowledge base for the robot due to the intractable complexity of the real world. Labeling training data is often an easier task than creating a knowledge base, but still can be a tremendous task for humans to do if the amount of data is great.

One way to shift the burden from humans to the robot itself is for the robot to acquire a model of the effects of its actions. With a predictive model, the robot can employ lookahead search techniques to make plans to achieve goals [47, 5, 13, 20]. Model learning in general is hard, however. If the environment has many features that are irrelevant to the robot's tasks, the robot may waste much time modeling the irrelevant aspects of the world. Moreover, if the environment is nondeterministic, the robot may need to model not only the state transitions caused by action executions but also the probability distributions of such transitions. It might be easier for robots to learn optimal control policies directly from interaction with its environment without learning a model. For instance, through experience a robot may figure out that the best action to execute in state x is action a rather than action b, simply because action a has been found empirically to lead to goal states sooner than action b. The robot does not need to know in detail how actions cause state changes.

Reinforcement learning is an optimization technique for control. It is self-supervised and does not rely on building models to obtain optimal control policies. Reinforcement learning can be viewed as a kind of compilation from trial-and-error experience into reactive control rules. A reinforcement learning agent may receive a reward or punishment after taking an action. In the most general case, rewards and punishments may be delayed; in other words, action executions that eventually lead to success or failure may not be rewarded or punished right away. Reinforcement learning is to construct a control policy that maximizes future rewards and minimizes future punishments.

1.1 The Thesis

Learning from delayed reinforcement in general is hard and can be a slow process if the agent begins with little knowledge of its environment, sensors, and actuators. Indeed, the problem of slow learning was often mentioned in previous studies on reinforcement learning. Slow learning is just one of the potential problems with reinforcement learning. For instance, most existing reinforcement learning methods assume that the world is a Markov decision process; that is, the agent can have direct access to the complete state information. For many robot problems of interest, this assumption does not hold, however. Because of these potential problems, whether reinforcement learning will ever be practical for real-world problems is questioned.

The thesis of this dissertation is:

> **We can scale up reinforcement learning in various ways, making it possible to build reinforcement learning agents that can effectively acquire complex control policies for realistic robot tasks.**

The aim of this dissertation is to present a series of scaling-up techniques and show empirical evidence to support this thesis statement. In particular, this dissertation develops various techniques to overcome the slow learning problem, and presents architectures for reinforcement learning in non-Markovian *environments.*

The results of this dissertation rely strongly on computer simulation, including (1) an agent operating in a dynamic and hostile environment and (2) a mobile robot operating in a noisy and non-Markovian environment. The robot simulator is physically realistic. For each issue studied here, various algorithms or ideas are tested and compared through a series of simulation experiments. Some of the algorithms or ideas are novel, and some are previously proposed by other researchers. Also, implications of simulation results are discussed, and possible performance differences between various methods (including some not studied in the simulation) are speculated.

1.2 Reinforcement Learning Paradigm

Figure 1.1 illustrates the reinforcement learning paradigm. In this paradigm, the learning agent has:

- *sensors*, from which it obtains information about the state of its environment,
- *effectors*, by which it can change its environment,
- *a control policy*, which maps situations to actions,
- *a reinforcement mechanism*, which generates a reinforcement signal after each action execution, and

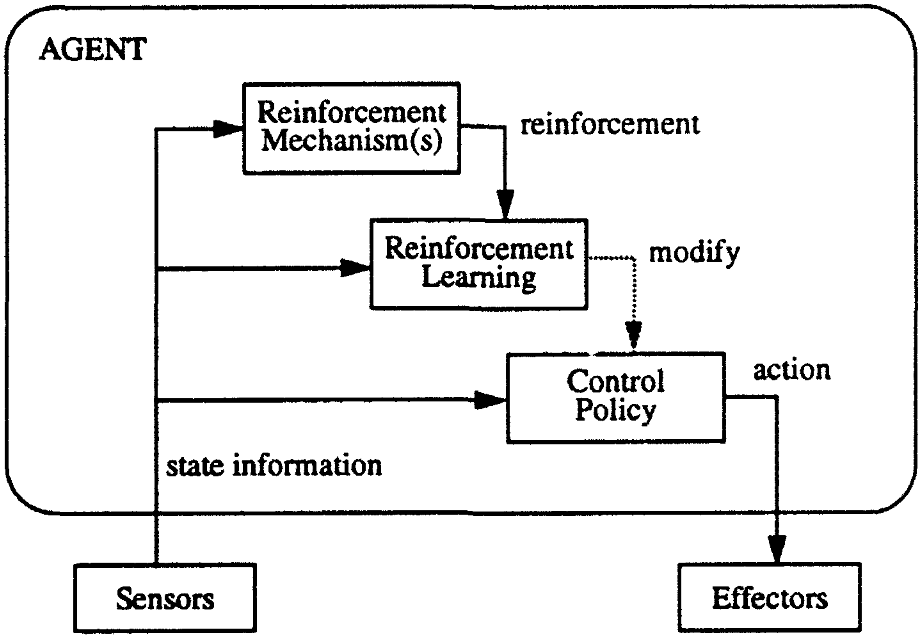


Figure 1.1: The reinforcement learning paradigm.

- ***a reinforcement learning component***, which takes into account the feedback from the environment (including the state information and the reinforcement signal) and adjusts the control policy.

The objective of learning is to construct a control policy (or simply policy) that maximizes the agent's performance. A natural measure of performance is the *discounted cumulative reinforcement* or, for short, *utility*:

$$V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1.1}$$

where V_t is the discounted cumulative reinforcement from time t throughout the future, r_t is the reinforcement received after the transition from time t to $t+1$, and $0 \leq \gamma \leq 1$ is a *discount factor*, which adjusts the importance of long-term consequences of actions. Note that a reinforcement signal can be positive (reward), negative (punishment), or 0.

In traditional reinforcement learning, we assume that there is a teacher to provide the learner with reinforcement signals. For example, a person may train an animal by rewarding it when it has done a good job. To build a truly autonomous learning agent, we can have a reinforcement mechanism built in to the agent. The reinforcement mechanism specifies what is desirable and what is undesirable for the agent. In essence, giving an agent a reinforcement mechanism is equivalent to giving it a goal. For example, a reinforcement mechanism may simply generate a reward signal when the agent reaches a goal state and zero reinforcement for all the other time. An agent may have multiple tasks, and thus may have multiple built-in reinforcement mechanisms; each one specifies a task.

Reinforcement learning is attractive. First of all, to program a reinforcement learning robot to carry out a task, we just need to design a reinforcement function specifying the goal of the task [1], which often takes much less human effort than explicitly programming the robot to accomplish the task. Second, reinforcement learning agents are:

- **adaptive**, in that they continually use feedback from the environment to improve their performance, and
- **reactive**, in that they compile experience directly into situation-action rules (the control policy), which can be evaluated in constant time.

In comparison with most other learning systems, reinforcement learning agents are also:

- **less dependent on prior domain knowledge**, because they learn inductively, and
- **less dependent on external teachers**, because they learn from reinforcement, which can be generated from a built-in reinforcement mechanism.

It is instructive to note the limitations of reinforcement learning:

- Not every learning task can be easily described in terms of rewards and punishments. For example, the task of building a map of an unknown environment may not be easily expressed as a reinforcement learning problem. But it appears that there is a wide range of tasks that can be expressed as reinforcement learning problems. At least, every problem that can be solved by search-based planning systems can also be described in terms of reinforcement.
- A reinforcement learning agent may have to learn a new control policy every time it is given a new goal. In contrast, traditional search-based planning systems offer the advantage of being more flexible in handling diverse goals, because they consist of a general-purpose problem solving mechanism, the planner. This problem, however, can be addressed in at least two ways. First, a reinforcement learning agent can learn an action model while it is learning to solve its first task. A new control policy still needs to be learned for a second task, but it is possible that the second task can be learned much more quickly than the first one, because the action model can be used and greatly

[1] Besides designing a reinforcement function, we may have to do extra work such as design an input representation, choose a network structure (if we use neural networks to implement the control policy), etc. But programming a robot in other approaches may require this extra work as well.

save the agent's action executions in the real world (Chapter 3). Second, learning skills hierarchically can greatly reduce the time for learning a new task (Chapter 6). For example, Task A and Task B consist of common subtasks. Once the agent learned to carry out Task A, learning to carry out Task B becomes easier, because the agent has already learned to carry out the common subtasks.

1.3 Scaling-Up Issues

Before reinforcement learning can become a practical learning technique for robots, several issues must be addressed:

Exploration

Consider the task of finding solution paths from any state to a goal state. To learn a policy for this task, first the learning agent must find some (possibly suboptimal) solution path from some initial state. Given no prior knowledge and no teachers, the agent has to find this first solution by trial and error. If it does this by naive random walks, finding the first solution may take exponential time in the solution length [79]. More clever exploration strategies will be needed if the agent is to be efficient. Finding any policy is not enough. Very often we want the agent to find the optimal policy. To do that, it has to find a diverse set of solution paths and determine the optimal one. An important issue is therefore how to explore the environment so that the optimal solution will not be missed and in the meantime the exploration cost will be kept minimal. For example, in order to collect information, the agent may want to move to an unexplored area of the state space, which, however, may result in damage to the agent. Basically, there is a tradeoff between acting to gain information and acting to gain reinforcement.

"Move to an unexplored state" is often a good exploration strategy. But implementation of this strategy may not be simple. First, if the state space is large, about all states will be unexplored. The agent has to determine whether a new state is worth exploration depending on its similarity to states visited before. This opens up the difficult question of how to measure the similarity of states. Second, if the agent does not have complete state information, it is hard for the agent to determine whether a state is new or not.

Delayed Reinforcement and Temporal Credit Assignment

Suppose a learning agent performs a sequence of actions and finally obtains certain rewards or penalties. The *temporal credit assignment problem* is to assign credit or blame to the situations and actions involved. Temporal credit assignment is often difficult when the reinforcement delay is long and the environment is nondeterministic.

Generalization

To be successful, a learning agent must generalize from its experience. In particular, when the state space is so large that exhaustive search is impractical, the agent must guess about new situations based on experience with similar situations. Again, this opens up the question of how to measure the similarity of states. Hamming distance, for example, may not be a good measure. For many nontrivial problems, the agent must develop high-order features for similarity measurement. The generalization problem is also known as *structural credit assignment problem*, because the task of generalization is to find out which of the many input features in the situations must take the credit or blame assigned by the temporal credit assignment process.

The generalization problem in the reinforcement learning paradigm is more difficult than that in the supervised learning paradigm, because supervised learning agents (e.g., Pomerleau's autonomous vehicle [55]) have access to a set of desired situation-action patterns, while reinforcement learning agents do not.

Using External Knowledge

Learning from trial-and-error can be very slow. For fast learning, reinforcement learning agents should be able to learn expertise not only from trial-and-error but also directly from other experts such as humans. External knowledge may come in various forms. For example, it may be an incomplete action model, a number of desirable stimulus-response patterns, or a few sample solutions to some task instances.

Hierarchical Learning

By hierarchical planning, a planning system can dramatically reduce the amount of search to solve a problem [32, 30]. Task decomposition and abstraction are the main ideas behind hierarchical planning. The same ideas can also apply to reinforcement learning to reduce learning time. Hierarchical learning can be performed in the following manner: First, a complex task is decomposed into multiple elementary tasks. Then, the agent learns control policies for accomplishing the elementary tasks. Finally, ignoring the low-level details, the agent can effectively learn a high-level policy which selects elementary policies to accomplish the original complex task.

Hidden state

A robot often cannot have direct access to complete state information due to sensory limitations. For example, visual sensors cannot identify objects hidden in a box. This problem of incomplete state information is known as the *hidden state problem*. When there are hidden states, the real world state is often difficult to determine, and so is the optimal action.

Active perception

A robot often cannot afford to sense everything in the world all the time. Fortunately, not all of the world features are always relevant. For example, a robot may only need sonar information to avoid obstacles and visual information to pick up an object; it does not need both kinds of information all the time. An important problem for robots is to choose the most economical set of sensing operations to apply so that all the needed information will be collected inexpensively.

Other Issues

Most of the reinforcement learning algorithms proposed to date assume discrete time and discrete actions, but we may like our robots to move continuously and smoothly. Real robots may have to control multiple sets of motors simultaneously. For example, a robot may want to run after a moving object and catch it. Doing that requires a proper coordination of two independent sets of actuators, the wheel and the arm.

1.4 Dissertation Overview and Main Results

Among the issues described in the previous section, this dissertation focuses on:

- effective and efficient temporal credit assignment,
- generalization (with artificial neural networks),
- learning from human teachers,
- hierarchical learning, and
- dealing with hidden states.

Whenever possible, the exploration problem is ignored in this dissertation. The problems of active perception, continuous time, continuous actions, etc, are postponed for future work.

Chapter 2 reviews previous algorithms and ideas that are directly or indirectly related to reinforcement learning.

Chapter 3 first describes two basic reinforcement learning methods in detail: *AHC-learning* [65] and *Q-learning* [74]. It then presents three extensions to the basic methods:

- *experience replay*— experiences are selectively replayed as if they were experienced again,
- learning an action model for (shallow) look-ahead planning, and
- *teaching*.

Each extension is expected to improve the learning speed. Detailed algorithms are given. These algorithms use artificial neural networks to obtain generalization, which is also expected to reduce learning time dramatically.

Chapter 4 presents the first simulation study in this dissertation. The learning task is for an agent to survive in a dynamic environment involving obstacles, food, and predators. The task is moderately complex, and is a good domain for a first study of scaling up reinforcement learning. The learning algorithms presented in the previous chapter are tested against this learning task. Some of the main results are as follows:

- Neural networks generalize effectively even in the presence of a large number of binary inputs.
- In general, the three extensions mentioned above all result in noticeable improvement on the learning speed.

Chapter 5 presents the second simulation study. The domain used here is a physically-realistic mobile-robot simulator. Control errors and sensing errors are included. This domain is more challenging than the previous one in the sense that the robot operates in a noisy environment with continuous state inputs. Some of the main results are as follows:

- Once again, neural networks generalize effectively even in the presence of noise and a large number of binary and real-valued inputs.
- A small amount of teaching can save the robot many learning trials.

This chapter also presents an improved version of the experience replay technique, and two possible ways of combining reinforcement learning and supervised learning.

Chapter 6 presents a novel approach to learning robot skills hierarchically within the reinforcement learning paradigm. It first presents a model of hierarchical reinforcement learning, followed by a complexity analysis for restricted domains. This analysis shows that hierarchical learning can reduce overall learning complexity under some assumptions. It then presents a simulation study using the robot simulator. Simulation experiments show that by hierarchical learning, the robot can solve a difficult learning problem, which as a monolithic problem is hardly solvable.

Chapter 7 presents three memory architectures that extend Q-learning and enable learning agents to tackle hidden states. These architectures are tested against several simple non-Markovian tasks with different characteristics. They all can solve those simple tasks, but have different shortcomings. This chapter also presents a categorization of reinforcement learning tasks, and speculates about which architecture might work best for what type of tasks.

Chapter 8 presents the last simulation study of this work, which is once again based on the robot simulator. The learning task is complex and involves hidden states. By combining hierarchical learning and one of the memory architectures mentioned above, the robot successfully develops skills to accomplish the task. Simulation experiments also indicate that the developed skills can adapt to small environmental changes without addition training. All of these simulation results together support the thesis statement; that is, it is possible for artificial agents to acquire complex skills effectively by reinforcement learning.

Chapter 9 summarizes the results, discusses the contributions of this dissertation, and points out areas for future research.

Chapter 2

Background and Related Work

Samuel's checker player [59], which was first published in 1959, perhaps is the earliest machine learning research that can be viewed as reinforcement learning. However, it seems that significant progress in reinforcement learning has only been taking place in the past decade, possibly due to the important inventions of algorithms for temporal credit assignment and for generalization. This chapter reviews the recent work that is directly or indirectly related to reinforcement learning. Detailed comparisons of my approaches with other previous approaches will be addressed in the coming chapters.

2.1 Temporal Credit Assignment and Temporal Difference

The best-understood learning procedure for temporal credit assignment is probably *temporal difference* (TD) methods [66]. TD methods have a solid mathematical foundation [66] [14], and can be related to dynamic programming [7]. A typical use of TD methods is to predict the probability of winning a game (such as checker) for each possible board situation. In reinforcement learning, TD methods can be used to predict the utility of each state (i.e., to predict the sum of future rewards received from a state throughout the future).

Traditional supervised learning methods for prediction learning are driven by the error between predicted and actual outcomes, while TD methods are driven by the error between *temporally successive predictions*. This may be easily explained by the following example. Consider playing a simple game such as tic-tac-toe. The learning problem is to predict, for each board situation, the probability of winning the game starting from that situation. By recording each game played, training instances can be collected for prediction learning. Consider the training instance shown in Figure 2.1. Suppose the estimated winning probabilities (hereafter, WP's) from States A, B, and C are currently 0.8, 0.3, and 0.6, respectively. The player started from State A, moved to B, to C, and finally won the game.

What a typical supervised learning method, such as Widrow and Hoff's *delta rule* [82],

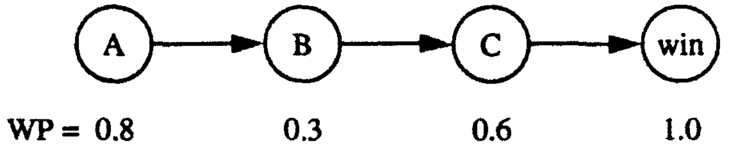


Figure 2.1: A sequence of state transitions in a game play. The WP for a state is the estimated probability of winning the game starting from that state.

would do regarding this training instance is the following: It increases the WP's from States A, B, and C by $(1-0.8)\alpha$, $(1-0.3)\alpha$, and $(1-0.6)\alpha$, where α is a learning rate. In essence, the learning changes are proportional to the differences between current predictions and the actual outcome (in this case 1). Given sufficient training data, repeated presentations of the data, and a small learning rate, this supervised learning method will eventually find the true WP from every state [82], assuming WP's are stored in a lookup table.

Unlike supervised learning methods, the simplest TD method, TD(0), attempts to minimize the error (called *TD error*) between successive predictions. Let Δ_S be the learning change to the WP from a state S. TD(0) will make the following learning changes:

$$\begin{aligned}\Delta_A &= (0.3-0.8)\alpha\\ \Delta_B &= (0.6-0.3)\alpha\\ \Delta_C &= (1.0-0.6)\alpha\end{aligned}$$

Sutton [66] has proved that TD(0) converges to the optimal estimates under three assumptions: sufficient training data, repeated presentations of training data, and the target system being linear (or a lookup table representation for the system to be predicted.) Sutton also argued and empirically showed that on multi-step prediction problems, TD(0) outperforms the kind of supervised learning discussed above.

There is in fact a family of TD methods called TD(λ), where λ is called the *recency factor* and $0 \leq \lambda \leq 1$. The TD error computed by TD(0) (called TD(0) error) is the difference between *two* successive predictions, while the TD error computed by TD(λ) is the sum of TD(0) errors exponentially weighted by λ. For instance, TD(λ) will make the following learning changes for the previous example:

$$\begin{aligned}\Delta_A &= [(0.3-0.8)+(0.6-0.3)\lambda+(1.0-0.6)\lambda^2]\alpha\\ \Delta_B &= [(0.6-0.3)+(1.0-0.6)\lambda]\alpha\\ \Delta_C &= [1.0-0.6]\alpha\end{aligned}$$

The convergence of TD(λ) has been proved by Dayan [14]. It is instructive to note that TD(1) makes the same learning changes as the supervised learning method. In fact, as λ approaches 1, TD(λ) behaves more like supervised learning. Because TD(λ) makes prediction changes based on not only the information from the immediately successive state but also the information from all future states, TD($\lambda > 0$) often results in faster learning than TD(0) at the early stage of learning when the initial predictions are meaningless. For detailed discussions on TD(λ) methods, see Chapter 5.

It is worthwhile to note that Samuel's famous checker-playing program [59] perhaps was the first system to use a kind of TD method. Samuel's program learned an evaluation function of board positions. The evaluation of a position was a likelihood estimate of winning the game from that position. The parameters of the evaluation function were adjusted based in part on the difference (i.e., TD error) between the evaluations of successive positions occurring in a game.

Holland's bucket brigade algorithm [25] is another learning procedure sharing the same idea of temporal difference. The bucket brigade is used to learn the priorities of rules for *classifier systems*, which are rule discovery systems based on genetic algorithms. In a classifier system, the invocation of a rule is determined by several factors. One of them is the *strength* associated with that rule. After a chain of rule invocations, the system may receive a payoff. The bucket brigade adjusts the strengths of rules so that rules leading to rewards will be more likely to fire when similar situations occur in the future. The process of strength adjustments is much like TD(0), except for a distinct difference: In the bucket brigade strengths are propagated back from rule to rule along the chain of rule invocations, while in TD(0) evaluation values are propagated back from state to state along the chain of state transitions. Grefenstette [21, 22] presented a credit assignment method called *profit sharing plan*, which is a variation of the bucket brigade.

2.2 Reinforcement Learning Algorithms

Two TD-based reinforcement learning algorithms have been proposed: *AHC-learning* due to Sutton [65] and *Q-learning* due to Watkins [74]. Both algorithms involve learning an evaluation function. AHC-learning learns an evaluation function of states, while Q-learning learns an evaluation function of state-action pairs. The convergence of Q-learning has been proved by Watkins and Dayan [74, 75]. In their proof, a set of conditions is needed to guarantee the convergence:

- the world is a Markov decision process,
- the evaluation function is represented by lookup tables,
- each state-action pair is tried infinitely often, and
- a proper scheduling of the learning rates.

Williams [83] presented a class of reinforcement learning algorithms that is quite different from those presented in this dissertation. He showed that his algorithms perform a gradient descent search for expected reinforcement, and can be naturally integrated with the error back-propagation algorithm [58].

2.3 Generalization

A common problem in robot learning is approximating a function, such as action models (mapping states and actions to next states), control policies (mapping situations to actions), evaluation functions (mapping states to utilities), etc. For example, Q-learning learns an evaluation function called the *Q-function*, which predicts the utility of states and actions. The Q-function can be represented by a lookup table, but this representation is feasible only when the learning problem has a small state space. For nontrivial problems, some kind of function approximator should be used to save memory and to generalize. A number of function approximators can be found in the literature, such as:

- Neural networks (see Section 2.4),
- Decision trees [56],
- Instance-based function approximation by nearest neighbor [47] or by locally weighted regression [5], and
- CMAC (Cerebellar Model Articulation Computer) [2],

In principle, all of these methods can be applied to approximate an evaluation function such as the Q-function. A potential difficulty, however, is that these methods were originally designed to be used in a supervised learning paradigm; in other words, for each training input, the corresponding desired output is directly known to the learning system. But this is not the case in the reinforcement learning paradigm. In Q-learning, for example, the real utilities of states and actions are never directly known to the learning agent, but can only be estimated through TD methods.

Among those function approximators mentioned above, neural networks appear most popular in the reinforcement learning community (for example, [3, 34, 36, 83, 70, 43, 6, 60, 78]). However, some previous attempts to combining the error back-propagation algorithm with TD methods did not seem very successful. For example, Anderson [3] combined back-propagation with AHC-learning in solving a pole balancing problem. While his program could solve the learning problem, it took 5000 trials to learn a good control policy— this is considered impractical in the real world. Other researches have also tried the combination and reported disappointing results [10, 29].

In contrast, the work reported here shows encouraging results on the combination of back-propagation and TD methods. This combination has successfully learned complex evaluation functions with a large number of binary and real-valued inputs. For example, Chapter 5 presents a simulated robot, which is able to learn to accomplish various tasks (such as wall following and door passing) within 150 learning trials. Recently, Tesauro presented interesting results on the use of the combination [70]. Tesauro's programs learned to play backgammon better than his previous supervised-learning approach and as well as world-class human players.

Besides neural networks, several other generalization approaches have also been studied for reinforcement learning. Mahadevan and Connell [41], Nilsson [52] and McCallum [42] all proposed approaches related to statistical clustering. The basic idea is to cluster the state space into ***regions*** based on some similarity measure. Each region of states is then treated as an abstract state and reinforcement learning algorithms are applied to this abstract state space. Another alternative (taken by Nilsson) is to construct a ***region graph*** showing the connectivity of regions by actions. With such a graph, solution paths can be found by graph search.

Chapman and Kaelbling [11] presented a *G algorithm* to approximate the Q-function. Like decision tree algorithms [56], the G algorithm incrementally builds tree-structured objects. It begins by assuming that the Q-function is constant over the entire state space and thereby collapsing the entire function into a single leaf node. When it collects sufficient statistical evidence to believe that an input bit plays an important role in the Q-function, the leaf node is split into two new leaf nodes corresponding to the two subspaces divided by that binary input bit. The new leaf nodes are candidates for further splitting. Ming Tan [69, 68] presented a learning method, which also stores evaluation values in decision trees.

Watkins [74] used a CMAC as a function approximator. Moore [48] used a method which was basically lookup tables but which had variable state resolution. Except for neural networks, all the generalization approaches mentioned above have only been tested for a few reinforcement learning tasks. Their utilities hence remain to be fully determined.

2.4 Neural Networks

Artificial neural networks are composed of simple processing elements called *units*. Each unit can pass its activation to another through a *connection*, which is associated with a *weight*. The activation of an input unit is simply the input signal. The activation of a non-input unit is computed as a function of (1) the weights of the incoming connections and (2) the activations transmitted through the connections. The function involved is called the *activation function* or *squashing function* of the unit. There are many types of artificial neural networks. Some use linear activation functions, and some nonlinear. Some have feed-back connections, and some do not.

Figure 2.2 illustrates a multi-layer network composed of input units, output units, and *hidden units*. Logistic units are a popular unit type used in networks with multiple layers. A logistic unit computes its activation as follows:

$$y = sigmoid(\sum_j w_j x_j) \tag{2.1}$$

where w_j is the weight of the jth incoming connection to the unit, x_j is the activation passed from the jth connection, and

$$sigmoid(x) = \frac{1}{1+e^{-x}}. \tag{2.2}$$

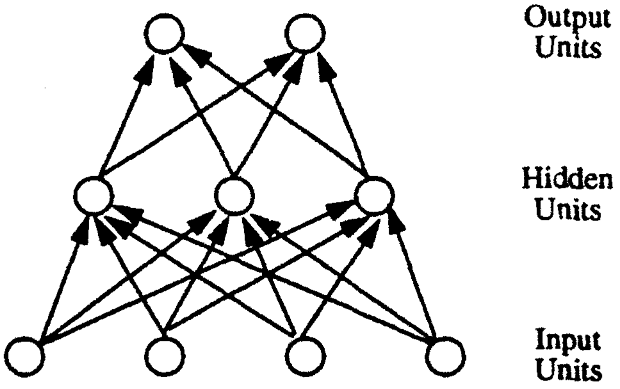


Figure 2.2: A simple multi-layer network with only feed-forward connections.

Stornetta and Huberman [64] advocate using a symmetric version of the sigmoid function:

$$symmetric\ sigmoid(x) = \frac{1}{1+e^{-x}} - 0.5. \tag{2.3}$$

By replacing Equation (2.2) with the symmetric one, they found speedups ranging from 10% to 50%. In this dissertation, I use this symmetric activation function.

The error back-propagation algorithm [58] is the most popular training algorithm for multilayer networks. Many other network training algorithms can be found in the literature [23]; most of them are variations of back-propagation, while others are new. For example, the quickprop algorithm [17] and the cascade-correlation algorithm [19] have been empirically shown to outperform the standard back-propagation algorithm. However, both algorithms in their current forms are not suited to the increment nature of learning evaluation functions. Both are specifically designed for batch-mode training— all the training patterns must be available at once for repeated presentations.

Networks that have feedback connections are called *recurrent networks*. Recurrent networks are useful for tasks in which the learning agent needs to recognize or predict temporal sequences of patterns. Figure 2.3 illustrates a simple type of recurrent networks called *Elman network*. The input layer of an Elman network is divided into two parts: the true input units and *context units*. The operation of the network is clocked. The context units hold feedback activations from the network state at a previous clock time. The context units function as short-term memory, so the output of the network depends on the past as well as on the current input.

The Elman network can be trained by a recurrent version of the back-propagation algorithm called *back-propagation through time* (BPTT) or *unfolding of time*. This BPTT training technique is based on the observation that any recurrent network spanning T steps can be converted into an equivalent feed-forward network by duplicating the network T times. For

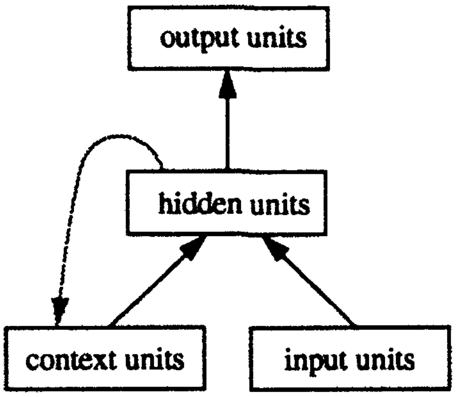


Figure 2.3: An Elman recurrent network.

example, the feed-forward network in Figure 2.4 and the recurrent network in Figure 2.3 behave identically for 3 time steps. Once a recurrent network is unfolded, back-propagation can be directly applied. (See also Chapter 7.) In addition to BPTT, researchers have developed other training procedures. For example, Williams and Zipser [84] have developed an algorithm called *real-time recurrent learning* (RTRL), which requires no duplication of networks. Fahlman [18] has developed an incremental construction architecture called the *recurrent cascade-correlation architecture*. Because Elman networks are conceptually simple and fit well the learning problems studied here, they are the type of recurrent networks used in this dissertation (see Chapter 7).

2.5 Exploration

The purpose of exploration is to acquire information for control learning. A reinforcement learning agent faces two opposing objectives: act to gain information and act to gain rewards. To improve its control policy, the agent sometimes has to take actions that appear suboptimal, because such actions may lead to an unexplored portion of the state space and eventually lead to the discovery of a better policy. On the other hand, the agent runs a risk of receiving negative reinforcement when it takes actions that are suboptimal or not taken before. How to control the tradeoff between gaining information and gaining reinforcement is an important issue.

Many reinforcement learning systems deal with the tradeoff mentioned above by taking random actions occasionally. For example, the learning agents presented in Chapter 4 choose actions randomly but favor to choose actions having high utilities. Meanwhile, action selection gets less random as learning proceeds. This approach is not very effective. Kaelbling [29]

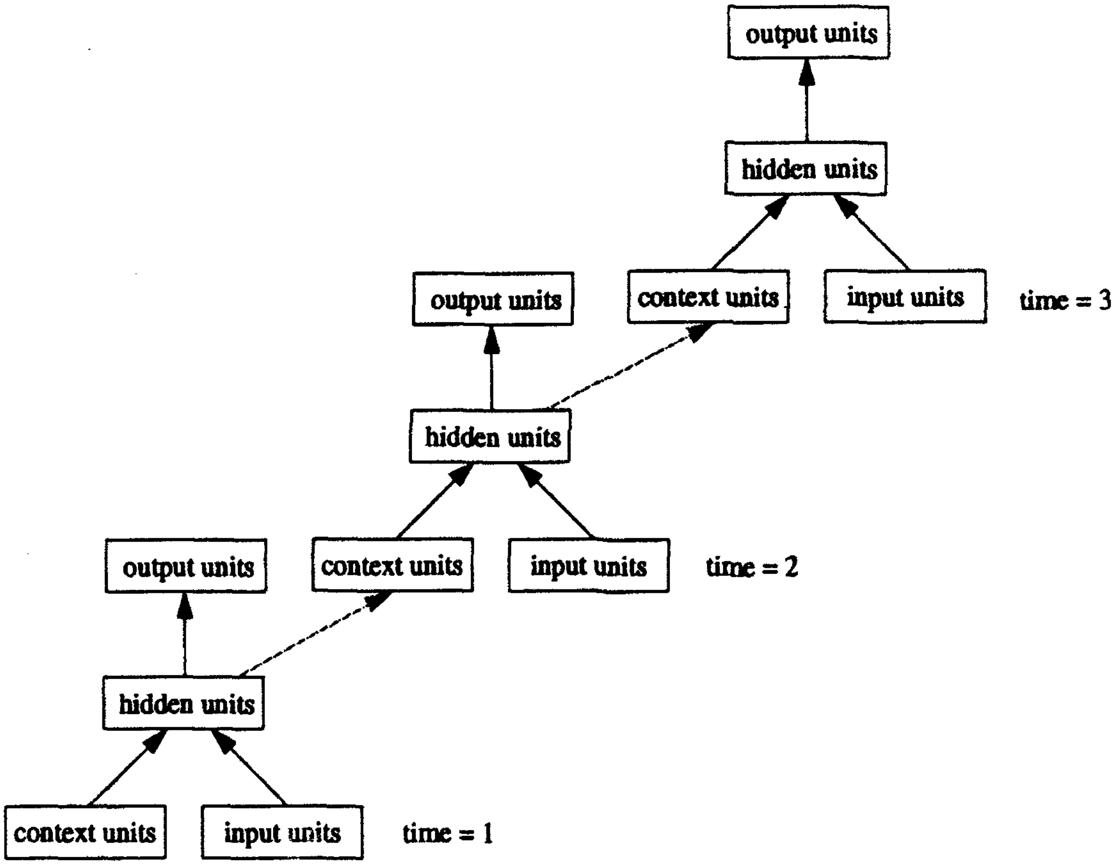


Figure 2.4: A feed-forward network which behaves the same as the Elman recurrent network for 3 time steps.

proposed a better approach called the *interval estimation* (IE) algorithm. The IE algorithm estimates not only the mean utility of an action but also the uncertainty about that mean. The agent then chooses the action that has the greatest combination of mean utility and uncertainty. Inspired by the IE algorithm, Sutton used a simple alternative in his DYNA architecture [67]: The DYNA agent assigns to each state-action pair a time duration indicating how long that state-action combination has not been attempted. The agent then chooses the action that has the greatest combination of utility and time duration.

Thrun [71] presented a systematic comparison among several exploration strategies, including:

- *counter-based exploration*— count the number of visits to each state and go to less explored states,
- *recency-based exploration*— similar to Sutton's method, and
- *counter-based exploration with decay*— counter-based exploration with counter values decaying over time.

Except for the random exploration strategy, all the approaches mentioned above allocate one counter for each state or state-action pair. To be practical for continuous or large state spaces, these approaches require some sort of generalization. For example, instead of one counter for each state, we have to *allocate a counter for a group of similar states* [52]. The problem of exploration becomes even more complicated when the agent's environment has hidden states. This is because we cannot count the number of times each state has been visited before if we can hardly determine the true world state.

2.6 Using Action Models

Learning in an unknown environment is risky especially when the agent is given no prior knowledge and must learn by trial and error. One way to reduce the risk is to utilize an action model, which may come from the agent's previous experience with solving a related task. By action projection with an action model, the agent can experience the consequence of an action without the risk of being hurt in the real world. As a matter of fact, if the agent has a perfect model, it can learn an optimal control policy just from hypothetical experience generated by the model. This way of using action models has been studied by Sutton in his DYNA architecture [67]. In Chapters 3 and 4, I present a similar way of using models and demonstrate that it may be beneficial to use a model even if the model is incomplete and only roughly correct.

Mitchell and Thrun [46] proposed another way of using action models, which they call *explanation-based neural network learning* (EBNN). Consider learning an evaluation function such as the Q-function. Given a sample point of the target function, EBNN estimates the slopes of the target function around that sample point and then trains a Q-function to fit not only the sample point but also the estimated slopes. The slope information is extracted from a differentiable action model, which in their case is represented by neural networks. Their experiments indicated that EBNN required significantly fewer training examples than standard inductive learning without using slopes. They also found that EBNN worked well even in the presence of some errors in the model.

Several approaches to control learning in the presence of hidden states are based on building an action model. The model maintains a short-term memory of the past, and will be able to uncover hidden states and determine the true world state unambiguously. Once the world state can be determined, traditional control-learning techniques, such as Q-learning, can be applied directly. Examples include the work by Schmidhuber [60], Thrun et al [72], Lin and Mitchell [37], Bachrach [6], and Chrisman [12]. The first four approaches all use recurrent networks for modeling.

Other ways of using action models can be found in Chapter 7.7.

2.7 Learning Action Models

Model learning is a problem that has been studied for decades. All of the function approximators mentioned in Section 2.3 can directly apply to learning a deterministic action model. In a nondeterministic environment, each action may have multiple outcomes, and a precise model has to predict all the possible outcomes of each action as well as the probability distribution of outcomes. Some work in modeling nondeterministic environments may be found in [13]. A review of many previous model learning approaches may be found in [4] and [6].

Model learning in general is a very hard problem. It is even more difficult when the environment has hidden states, in which case the hidden states need to be uncovered. The most famous and most successful algorithm for learning action models in the presence of hidden states is probably the *diversity-based inference* procedure proposed by Rivest and Schapire [57]. (This procedure cannot learn nondeterministic environments, however.) This learning procedure is based on a clever representation for finite-state environments called *update graphs*.

Mozer and Bachrach [51] proposed a connectionist implementation of the update graphs called the *SLUG* architecture. Although they found that SLUG failed to infer some finite-state environments that the Rivest and Schapire's symbolic algorithm could, it does have a useful property: Connectionist action models can continually make sensible predictions even before learning is complete. (In contrast, Rivest and Schapire's algorithm can only make sensible predictions after the update graph is complete.) This property is important, because it allows an agent to benefit from using a model even before it is perfectly learned. In Chapter 7.4, the SLUG architecture is studied and related to a family of network architectures called *OAON* (One Action One Network) architectures.

2.8 Active Perception

The goal of active perception is to acquire needed perceptual information in an economical way. Two active perception approaches for reinforcement learning agents have been developed by Whitehead and Ballard [81] and Tan [69, 68]. Both approaches rely on a technique to detect *perceptually aliased* states (i.e., states that cannot be distinguished by the immediate sensation). In Tan's approach, once an aliased state is detected, a new perceptual operation is added to the set of operations previously registered for that aliased state. Insertion of a new operation takes into account both the information gain and the cost incurred by applying the new operation. In Whitehead and Ballard's approach, once an aliased state is detected, a search is conducted to find an operation capable of disambiguating the aliased state. A common limitation of both approaches, however, is that they do not support generalization and nondeterministic environments. Relaxation of this limitation may be possible but will demand substantial efforts.

2.9 Other Related Work

Reinforcement learning has been studied mostly solving tasks with discrete time and discrete actions. While it seems possible to extend the idea of temporal difference (TD) to handle continuous time and continuous actions, this in general is an unexplored area. Millan and Torras [43] presented a reinforcement learning approach to path finding with continuous states and continuous actions. In their task, reinforcement was not delayed and thus there was no temporal credit assignment problem involved.

Ackley and Littman [1] presented a simulation study that combined two adaptation processes: reinforcement learning (adaptation of individuals) and *evolution* (adaptation of populations). The job of evolution is to discover a good reinforcement function that causes the simulated agents to survive in an environment involving food and predators. The job of reinforcement learning, as discussed before, is to discover a good control policy that maximizes reinforcement received with respect to a given reinforcement function. This strategy, which they call *evolutionary reinforcement learning*, displayed better performance than that employing only evolution or only reinforcement learning.

Singh [62] presented a modular reinforcement learning architecture, which could learn multiple elementary tasks and composite tasks efficiently. By sharing the solutions of elementary tasks across multiple composite tasks, significant savings of learning time can be obtained. Singh [63] [61] also proposed to use abstract action models to learn complex tasks efficiently. An abstract action model is a model for predicting the effects of abstract actions, which span multiple time steps. Chapter 6 gives more discussions on his work.

2.10 Integration

In parallel with this dissertation research, there have been many efforts on scaling and extending reinforcement learning. Some have proposed techniques for generalization, some for utilizing action models, some for modular learning, etc. But there have been few efforts attempting to integrate multiple extensions to reinforcement learning (but see [29]). This dissertation makes a step toward integration.

Chapter 3

Frameworks for Speeding Up Learning

This chapter presents four techniques to speed up reinforcement learning:

- generalization by neural networks,
- experience replay,
- using action models, and
- teaching.

These techniques are combined with AHC-learning and Q-learning, resulting in eight reinforcement learning frameworks. This chapter presents detailed algorithms for these frameworks, and the next chapter presents simulation experiments.

3.1 Introduction

Temporal difference (TD) [66] is a popular method for temporal credit assignment. For example, adaptive heuristic critic (AHC) learning [65] and Q-learning [74] are two well-known reinforcement learning methods based on TD methods. The error back-propagation algorithm [58] is a popular training algorithm for multi-layer neural networks. Variations of this training algorithm have been applied to deal with many real world problems that require a good generalization capability. Examples include speech recognition [73], autonomous vehicle control [55], and natural language parsing [27].

Both popular learning procedures (one for temporal credit assignment and one for generalization) in fact can be naturally integrated to solve hard reinforcement learning problems, as this dissertation will show (also see [70]).

Even when aided by a generalization mechanism, reinforcement learning still can be very slow. Temporal credit assignment in general is a hard problem, especially when credits must be propagated through a long action sequence. In addition to generalization, this chapter presents three extensions for speedup; two of them are novel. They are experience replay, using action models, and teaching. These extensions, neural networks, and the basic AHC-learning and Q-learning methods constitute eight different reinforcement learning frameworks:

- AHCON: connectionist AHC-learning
- AHCON-R: AHCON plus experience replay
- AHCON-M: AHCON plus using action models
- AHCON-T: AHCON plus experience replay plus teaching
- QCON: connectionist Q-learning
- QCON-R: QCON plus experience replay
- QCON-M: QCON plus using action models
- QCON-T: QCON plus experience replay plus teaching

This chapter describes these frameworks in detail. Empirical evaluation and comparisons of them are presented in the next chapter.

3.2 V-function and Q-function

As discussed in Chapter 1.2, the objective of reinforcement learning is to construct a control policy that maximizes an agent's performance. A natural measure of performance is the *discounted cumulative reinforcement* or *utility*, which is defined in Equation (1.1) and duplicated here for readers' convenience:

$$V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{3.1}$$

where V_t is the discounted cumulative reinforcement from time t throughout the future, r_t is the reinforcement received after the transition from time t to $t+1$, and $0 \leq \gamma \leq 1$ is a discount factor.

Both AHC-learning and Q-learning are based on the idea of learning an evaluation function to predict the discounted cumulative reinforcement to be received in the future. AHC-learning learns a V-function, while Q-learning learns a Q-function:

- $V(x)$ represents the expected discounted cumulative reinforcement that will be received starting from world state x, or simply the utility of state x;
- $Q(x, a)$ represents the expected discounted cumulative reinforcement that will be received after the execution of action a in response to world state x; or simply the utility of the state-action pair (x, a).

Note that it is only meaningful to estimate the V-function and the Q-function relative to some control policy, since different policies will result in different reinforcements received. Unless explicitly mentioned, both functions are assumed to be relative to the current policy.

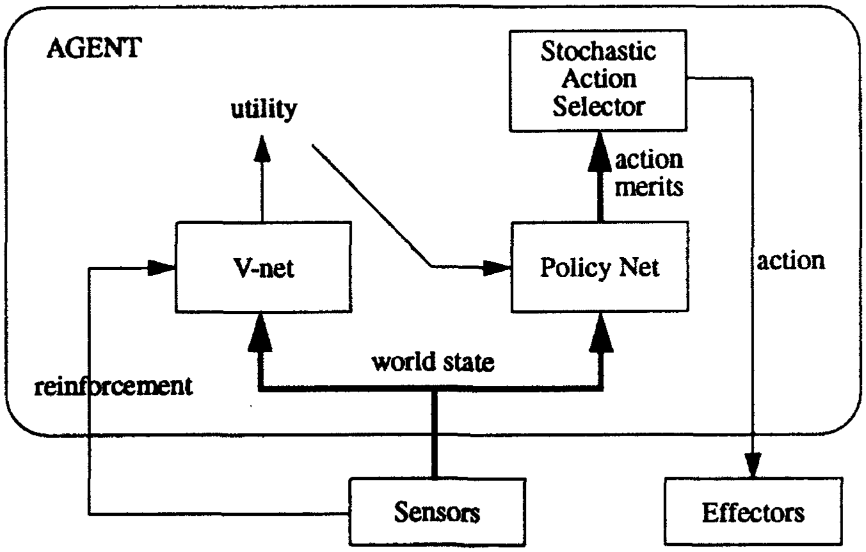


Figure 3.1: Framework AHCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

Note also that in a deterministic world, $Q(x,a)$ is equal to the immediate reinforcement r plus the utility of the next state y, discounted by γ:

$$Q(x,a) = r + \gamma \cdot V(y) \tag{3.2}$$

Equation (3.2) can be generalized to a nondeterministic world by taking into consideration the probabilities of multiple action outcomes [74].

For simplicity, I will assume the world is deterministic throughout the discussion of the learning algorithms in this chapter, although it is straightforward to extend the discussion to handle nondeterministic worlds [8, 74]. The reinforcement learning algorithms presented here work for both deterministic and nondeterministic worlds.

Both the V-function and Q-function can be represented by neural networks and learned by temporal difference and back-propagation. In essence, temporal difference computes the error between temporally successive predictions, and back-propagation minimizes the error by modifying the weights of the networks.

3.3 Framework AHCON

Framework AHCON (Figure 3.1) is a connectionist implementation of the adaptive heuristic critic learning architecture [65, 8]. It consists of three components: an evaluation network

```
1. $x \leftarrow$ current state;  $e \leftarrow V(x)$;
2. $a \leftarrow select(policy(x), T)$;
3. Perform action $a$;  $(y, r) \leftarrow$ new state and reinforcement;
4. $e' \leftarrow r + \gamma \cdot V(y)$;
5. Adjust the V-net by back-propagating TD error $(e' - e)$
   through it with input $x$;
6. Adjust the policy net by back-propagating error $\Delta$ through it
   with input $x$, where $\Delta_i = \begin{cases} e' - e & \text{if } i = a \quad (i \in \text{actions}) \\ 0 & \text{otherwise} \end{cases}$
7. go to 1.
```

Figure 3.2: The algorithm for Framework AHCON.

called *V-net*, a policy network called *policy net*, and a *stochastic action selector*. In essence, the framework decomposes reinforcement learning into two subtasks. The first subtask is to construct, using TD methods, a V-net that models the V-function. The policy net takes a world state [1] as inputs and assigns to each action a value (called *action merit*) indicating the relative merit of performing that action in response to that world state. Thus, the second subtask is to adjust the policy net so that it will assign higher merits to actions that result in higher utilities (as measured by the V-net).

Idealy, the outputs of the policy net will approach the extremes, for example, 1 for the best action(s) and -1 for the others. Given a policy net and a state, the agent's policy is to choose the action that has the highest merit. In order to learn an optimal policy effectively, the agent has to take actions that appear suboptimal once in a while so that the relative merits of actions can be assessed. A stochastic action selector, which favors actions having high merits, can be used for this purpose.

In the course of learning, both the V-net and policy net are adjusted incrementally. Figure 3.2 summarizes the learning algorithm, in which the simplest TD method, TD(0), is used. [2] To use TD methods to learn an evaluation function, the first step is to write down a recursive definition of the desired function. By Definition (3.1), the utility of a state x is the immediate payoff r plus the utility of the next state y, discounted by γ (assuming that the current policy is followed throughout the future). Therefore, the optimal V-function must satisfy Equation (3.3):

$$V(x) = r + \gamma \cdot V(y) \tag{3.3}$$

[1] Normally an agent's internal description of the world is obtained from a vector of sensory readings, which may or may not be pre-processed. Here it is assumed that the agent's input adequately represents the external world and is sufficient to determine the optimal actions. See Chapter 7 for relaxing this assumption.

[2] See Chapter 5 for using TD(λ) with general λ

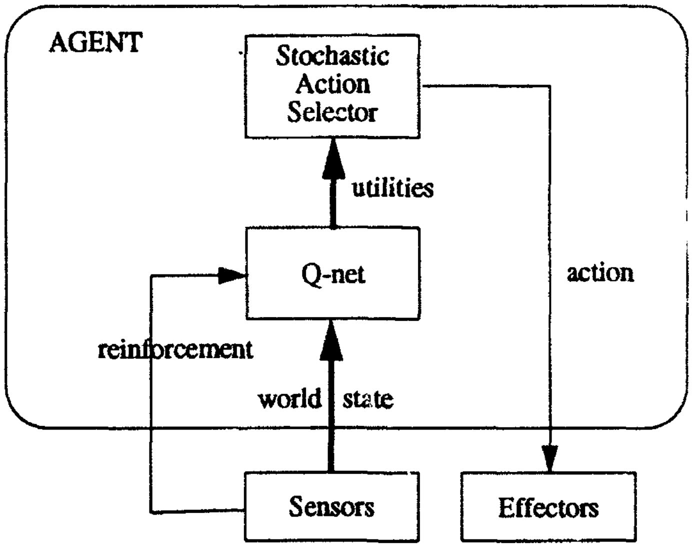


Figure 3.3: Framework QCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

During learning, the equation may not hold true. The difference between the two sides of the equation (called TD error) is reduced by adjusting the weights of the V-net using the back-propagation algorithm (Step 5).

The policy net is also adjusted (Step 6) according to the same TD error. The idea is as follows: If $e' > e$, meaning that action a is found to be better than previously expected, the policy is modified to increase the merit of the action. On the other hand, if $e' < e$, the policy is modified to decrease the merit. The policy net is not updated with respect to actions other than a, since from a single experience we know nothing about the merits of the other actions.

Du.ing learning, the stochastic action selector (i.e., the *select* function in the algorithm) chooses actions randomly according to a probability distribution determined by action merits. In this work, the probability of choosing action a_i is computed as follows:

$$Prob(a_i) = e^{m_i/T} / \sum_k e^{m_k/T} \tag{3.4}$$

where m_i is the merit of action a_i, and the *temperature* T adjusts the randomness of action selection.

```
1. $x \leftarrow$ current state;   for each action $i$, $U_i \leftarrow Q(x,i)$;
2. $a \leftarrow select(U,T)$;
3. Perform action $a$;   $(y,r) \leftarrow$ new state and reinforcement;
4. $u' \leftarrow r + \gamma \cdot Max\{Q(y,k) \mid k \in actions\}$;
5. Adjust the Q-net by back-propagating error $\Delta U$
     through it with input $x$, where $\Delta U_i = \begin{cases} u' - U_i & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$
6. go to 1.
```

Figure 3.4: The algorithm for Framework QCON.

3.4 Framework QCON

Framework QCON is a connectionist implementation of *Q-learning* [74]. QCON learns a network (called *Q-net*) that models the Q-function. Given a Q-net and a state x, the agent's policy is to choose the action a for which $Q(x,a)$ is maximal over all actions. Therefore, the Q-net is not only an evaluation function but also used as a policy.

The learning algorithm is depicted in Figure 3.4. In a deterministic world, the utility of an action a in response to a state x is equal to the immediate payoff r plus the best utility that can be obtained from the next state y, discounted by γ. Therefore, the optimal Q-function must satisfy Equation (3.5):

$$Q(x,a) = r + \gamma \cdot Max\{Q(y,k) \mid k \in actions\} \tag{3.5}$$

During learning, the difference between the two sides of the equation is minimized using the back-propagation algorithm (Step 5). Note that the network is not modified with respect to actions other than a, since from a single-experience we know nothing about the utilities of the other actions.

The Q-net can be implemented by a multi-layer network with multiple outputs; one for each action. This implementation, however, might not be desirable, because whenever this network is modified with respect to an action, no matter whether it is desired or not, the network is also modified with respect to the other actions as a result of shared hidden units between actions. An alternative is to use multiple networks to represent the Q-function; one network for each action and each with a single output. I call this network structure *OAON, One Action One Network* (see also Chapter 7.4). A similar argument can apply to the policy network of Framework AHCON. In this dissertation, I use OAON to implement Q-nets and policy nets.

Again, a stochastic action selector can be used to explore the consequences of different actions. In this work, the probability of choosing action a_i is computed as follows:

$$Prob(a_i) = e^{Q(x,a_i)/T} / \sum_k e^{Q(x,a_k)/T} \tag{3.6}$$

where x is the input state and T adjusts the randomness of action selection.

3.5 Experience Replay: AHCON-R and QCON-R

The basic AHC- and Q-learning algorithms described above are inefficient in that experiences obtained by trial-and-error are utilized to adjust the networks only once and then thrown away. This is wasteful, since some experiences may be rare and some (such as those involving damages) costly to obtain. Experiences should be reused in an effective way.

In this dissertation, two terms are often used:

- *Experience.* An *experience* is a quadruple, (x, a, y, r), meaning that the execution of an action a in a state x results in a new state y and reinforcement r.
- *Lesson.* A *lesson* is a temporal sequence of experiences starting from an initial state to a final state, where the goal may or may not be achieved.

The most straightforward way of reusing experiences is what I call *experience replay.* By experience replay, the learning agent remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The usefulness of doing this is two-fold:

- the process of credit/blame propagation is sped up, and
- an agent would get a chance to refresh what it has learned before.

During training a network, if an input pattern has not been presented for quite a while, the network typically will forget what it has learned for that pattern and thus need to re-learn it when that pattern is seen again later. This problem is called the *re-learning problem.* Consider the following scenario: The state space consists of two parts, A and B. The agent is now exploring part A, and has not visited part B for a long period of time. The agent may thus forget what it has learned for part B. By experience replay, the agent can mentally re-experience part B and refresh what it has learned.

It is important to note that a condition for experience replay to be useful is that the laws that govern the environment of the learning agent should not change over time (or at least not change rapidly), simply because if the laws have changed, past experiences may become irrelevant or even misleading.

Experience replay can be more effective in propagating credit/blame if a sequence of experiences is replayed in temporally **backward** order. It can be even more effective if TD(λ)

methods are used with $\lambda > 0$. With $\lambda > 0$, the TD error for adjusting network weights is determined by the discrepancy between not only two but multiple consecutive predictions. Chapter 5 presents a detailed algorithm and study of using backward replay and nonzero λ in a mouile robot domain. In this chapter and the next chapter, only backward replay and $\lambda = 0$ are used for simplicity reasons.

3.5.1 Replaying Only Policy Actions

The algorithm for AHCON-R is simply repeated presentation of past experiences to the algorithm for AHCON (Figure 3.2), except that *experiences involving non-policy actions (according to the current policy) are not presented.* By presenting an experience to the algorithm, I mean to bind the variables, (x, a, y, r), used in the algorithm with the experience. The reason for the exception is the following: AHCON estimates $V(x)$ (relative to the current policy) by sampling the current policy (i.e., executing some policy actions). Replaying past experiences, which is equivalent to sampling policies in the past, will disturb the sampling of the current policy, if the past policies are different from the current one. Consider an agent whose current policy chooses a very good action a in state x. If the agent changes to choose a very bad action b in the same state, the utility of state x, $V(x)$, will drop dramatically. Similarly, $V(x)$ will be underestimated if the bad action b are replayed a few times. For experience replay to be useful, the agent should only replay the experiences involving actions that still follow the current policy.

The algorithm for QCON-R is simply repeated presentation of past experiences to the algorithm for QCON (Figure 3.4). Just like AHCON-R but for a different reason, QCON-R should also replay only the actions that follow the current policy. Consider the previous example. Even if the agent changes from choosing good action a to choosing bad action b in state x, the values of both $Q(x, a)$ and $Q(x, b)$ will not change. As a matter of fact, if the Q-function is represented by look-up tables, Watkins and Dayan [74, 75] have shown that Q-learning is guaranteed to find the optimal policy as long as all state-action pairs are tried infinitely often (see Chapter 2.2). In other words, it is not harmful for the tabular version of Q-learning to replay bad actions. But this does not hold true for connectionist Q-learning, because whenever the back-propagation algorithm modifies the Q-net with respect to one input state, it also affects the network with respect to many or all input states. If the Q-net is trained on bad experiences many times consecutively, the network might come to underestimate the real utilities of some state-action pairs. If a bad action is never replayed, the Q-net may not have an accurate estimate of this action's utility. But this may be fine, since the agent has no need to know the exact utility of bad actions.

In short, AHCON-R and QCON-R should replay only policy actions. If a stochastic policy is used as suggested previously, all actions have more or less possibility of being chosen, and it is thus difficult to say whether an action is a current policy action or not. In this work, an action is considered as non-policy action, if its probability (see Equations (3.4) and (3.6)) of being chosen (according to the current policy) is lower than some threshold, P_l.

3.5.2 Over-Replay

To be efficient in terms of computational time and memory space, AHCON-R and QCON-R can keep and replay only recent experiences, because experiences in the far past typically involve many bad action choices and are thus less worth keeping than recent experiences.

Note that the correctness of Q-learning and AHC-learning relies on the assumption that *next states are visited in proportion to their probabilities.* Improper experience replay may invalidate this assumption, causing an *over-replay problem.* Consider an agent living in a nondeterministic world. Whenever the agent is in state x and performs action a, 80% of the time it will receive a great penalty, while 20% of the time it will receive no penalty. If the agent experienced only the situation with no penalty and is trained on this experience many times, the agent will come to believe that it is harmless to perform a in x, which is certainly wrong.

One way to prevent over-replay is to replay each experience roughly the same number of times except that bad actions need not be replayed.

3.6 Using Action Models: AHCON-M and QCON-M

Another way of reusing past experiences is to use experiences to build an *action model* and use it for planning and learning. This way of reusing experiences has been investigated in Sutton's DYNA architecture [67]. An action model is a function from a state and an action, (x, a), to the next state and the immediate reinforcement, (y, r). For stochastic worlds, each action may have multiple outcomes. To be precise, the action model may also need to model the probability distribution of outcomes— this issue is not addressed in this research, however.

An action model is intended to mimic the behaviors of the environment. Using an accurate action model (if it is available), the agent can experience the consequences of actions without participating in the real world. As a result, the agent will learn faster (if projecting is faster than acting) and more importantly, fewer mistakes will be committed in the real world. Frameworks AHCON-M and QCON-M to be discussed below are based on this idea.

3.6.1 Framework AHCON-M

Framework AHCON-M is very similar to Framework AHCON, but differs in that AHCON-M also learns an action model and uses it to do what Sutton called *relaxation planning* [67]. Relaxation planning is an incremental planning process that consists of a series of shallow (usually one-step look-ahead) searches and ultimately produces the same results as a conventional deep search. In Sutton's DYNA architecture, his learning algorithm (similar to the one for AHCON) is applied to real-world situations faced by the agent and also hypothetical situations randomly generated. In the former case the next state is obtained by executing an action, while in the latter case the next state is obtained by applying an action model. His approach has two ineffi-

```
1. $x \leftarrow$ current state;   $e \leftarrow V(x)$;
2. Select promising actions $S$ according to $policy(x)$;
3. If there is only one action in $S$, go to 8;
4. For $a \in S$ do
   4.a. Simulate action $a$;   $(y, r) \leftarrow$ predicted new state and reinforcement;
   4.b. $E_a \leftarrow r + \gamma \cdot V(y)$;
5. $\mu \leftarrow \sum_{a \in S} Prob(a) \cdot E_a$;   $max \leftarrow Max\{E_a \mid a \in S\}$;
6. Adjust the V-net by back-propagating error $(max - e)$
   through it with input $x$;
7. Adjust the policy net by back-propagating error $\Delta$ through it
   with input $x$, where $\Delta_a = \begin{cases} E_a - \mu & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$
8. exit.
```

Figure 3.5: Relaxation planning algorithm for AHCON-M. $Prob(\cdot)$ is the probability function for stochastic action selection.

ciencies. First, since hypothetical situations are randomly generated, the agent may spend too much effort planning what to do about hypothetical situations that will never happen in the :eal world at all. Second, it is not clear how his approach decides when relaxation planning is no longer necessary and to stop doing it.

The relaxation planning algorithm (see Figure 3.5) proposed here addresses both inefficiencies by projecting all actions from states actually visited, not from states chosen at random. Since all actions to a state are examined at the same time, the relative merits of actions can be more directly and effectively assessed than the kind of *policy iteration* [26] used in AHCON and the DYNA architecture. Compared with the DYNA architecture, the disadvantage of this algorithm, however, is that the number of hypothetical experiences t. at can be generated is limited by the number of states visitied.

In a deterministic world, the utility of a state x is equal to the immediate payoff r obtained from executing the best action a plus the discounted utility of the new state y:

$$V(x) = Max\{r + \gamma \cdot V(y) \mid a \in actions\} \qquad \text{(Note: } y \text{ is a function of } a\text{)} \tag{3.7}$$

Thus, if a correct action model is available, the utility of a state can be effectively estimated by looking ahead one step. During learning, the V-net is adjusted to minimize the difference between the two sides of Equation (3.7) (Step 6).

The policy net is updated in the following manner: First we compute the average utility of state x, μ, assuming that the current policy and stochastic action selector will be used throughout the future (Step 5). $Prob(\cdot)$ is the probability distribution function for action selection; it is

```
1. $x \leftarrow$ current state;   for each action $i$, $U_i \leftarrow Q(x,i)$;
2. Select promising actions $S$ according to $U$;
3. If there is only one action in $S$, go to 6;
4. For $a \in S$ do
   4.a. Simulate action $a$;  $(y,r) \leftarrow$ predicted new state and reinforcement;
   4.b. $U'_a \leftarrow r + \gamma \cdot Max\{Q(y,k) \mid k \in actions\}$;
5. Adjust the Q-net by back-propagating error $\Delta U$
   through it with input $x$, where $\Delta U_a = \begin{cases} U'_a - U_a & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$ .
6. exit.
```

Figure 3.6: Relaxation planning algorithm for QCON-M.

Equation (3.4) in this work. Next, the policy network is modified to increase/decrease the merits of actions which are above/below the average (Step 7).

The algorithm for Framework AHCON-M is similar to that for AHCON, except that relaxation planning may take place either before Step 2 or after Step 3 (in Figure 3.2)— in the latter case the agent will be more reactive, while in the former case the agent may make better action choices because it can benefit directly from the one-step look-ahead. To be efficient, relaxation planning is performed selectively (Steps 2 & 3 in Figure 3.5). The idea is this: For situations where the policy is very decisive about the best action, relaxation planning is not needed. If the policy cannot be very sure about which is the best action, relaxation planning is performed. In this way, at the beginning of learning, all actions are equally good and relaxation planning is performed frequently. As learning proceeds, relaxation planning is performed less and less often. (In this work, promising actions are those whose probability of being chosen is greater than 2%.)

3.6.2 Framework QCON-M

Framework QCON-M is Framework QCON plus using action models. The algorithm is similar to that in Figure 3.4. The main difference is that before Step 2 or after Step 3, the agent can perform relaxation planning by looking one step ahead (see Figure 3.6). For the same reason that it is harmful for QCON-R to replay non-policy actions (Section 3.5), experiencing bad actions with a model can be also harmful [3]. Therefore, relaxation planning must be performed selectively. For situations where the best action is obvious, no look-ahead planning is needed. If there are several promising actions, then these actions are tried with the

[3] AHCON-M does not have this problem, since at each planning step, only the most promising action is used to determine the amount of change to $V(x)$.

model. QCON-M is similar to Sutton's DYNA-Q architecture except that only the currently visited state is used to start hypothetical experiences. (Again, in this work, promising actions are those whose probability of being chosen is greater than 2%.)

3.7 Teaching: AHCON-T and QCON-T

Teaching plays a critical role in human learning. Very often teaching can shorten our learning time, and even turn intractable learning tasks into tractable ones. If learning can be viewed as a search problem [44], teaching, in some sense, can be viewed as external guidance for this search. Consider a reinforcement learning agent that attempts to reach a goal state in a large state space for the first time. If the agent begins with zero knowledge, it has to find a way to the goal state by trial-and-error, which may take an arbitrarily long time if the probability of achieving the goal by chance is arbitrarily small. This learning barrier will prevent agents from shortening learning time dramatically on their own. One way to overcome the barrier is to learn expertise directly from external experts.

Teaching is useful in three ways:

- Partial solutions can be readily available from teachers.
- Teaching can direct the learner to first explore the promising part of the search space which contains the goal states. This is important when the search space is large and thorough search is impractical.
- Teaching can help the learner avoid being stuck in local optima during the search for optimal control. Real examples can be found in Chapter 5.4.1.

Frameworks AHCON-T and QCON-T, which are AHCON-R and QCON-R plus teaching, use exactly the experience replay algorithms for AHCON-R and QCON-R, respectively. Teaching is conducted in the following manner: First, a teacher shows the learning agent how an instance of the target task can be accomplished from some initial state. The sequence of the shown actions as well as the state transitions and received reinforcements are recorded as a *taught lesson*. Several taught lessons can be collected and repeatedly replayed the same way *experienced* (i.e., self-generated) *lessons* are replayed. In this dissertation, the term *lesson* means both taught and experienced lessons.

3.7.1 Comparison with Supervised Learning

It is unnecessary that the teacher demonstrate only optimal solutions in order for the agent to learn an optimal policy. In fact, the agent can learn from both positive and negative examples. This property makes this approach to teaching different from supervised learning approaches such as [50, 54]. In the supervised learning paradigm, a learning agent tries to mimic a teacher by building a mapping from situations to the demonstrated actions and generalizing the mapping. The drawbacks of supervised learning are as follows:

1. The teacher is required to create many many training instances to cover most of the situations to be encountered by the agent.
2. When a new situation is encountered and the agent does not have a good strategy for it, a teacher must be available to give the solution.
3. If the teacher is not an expert, we generally do not expect the agent to become an expert.

The third drawback is often neglected by researchers, but it is important when humans want to build robots using supervised learning techniques. Since humans and robots have different sensors and hence see different things, an optimal action from a human's point of view may not be optimal for robots if we take into account the fact that robots may not be able to sense all the information that humans use to make decisions. Human teachers must teach in terms of what robots can sense— sometimes this is difficult to do.

Frameworks AHCON-T and QCON-T do not have the first and second drawbacks, because they can improve performance from reinforcement. They also do not have the third drawback, because they do not learn by mimicking the teacher. Instead, they determine the real utilities of the shown actions. On the other hand, an expert can give the learner more instructive training instances than a naive teacher.

3.7.2 Over-Replay

Like experienced lessons, taught lessons should be replayed selectively; in other words, only policy actions should be replayed. But if the taught actions are known to be optimal, all of them can be safely replayed.

Recall that improper experience replay may violate the correctness assumption that next states are visited in proportion to their probabilities, and cause an over-replay problem (Section 3.5.2). Similarly, learning from teaching experiences may also invalidate this assumption, if teaching experiences are replayed many times in a short period of time. Thus, it is important for agents not to be over-trained on teaching experiences.

3.8 A Note: AHC-learning vs. Q-learning

While the convergence of Q-learning has been proved by Watkins and Dayan [74, 75], it is still unclear whether AHC-learning will always converge and find the optimal control policy. In AHC-learning, there are two concurrent learning processes: learning a V-function and learning a policy. Both processes interact closely— a change to the policy will re-define the target

V-function, and an update to the V-function will cause the policy to change. It is possible that these two processes will interact to either prevent convergence or stablize to a suboptimal policy.

3.9 Summary

This chapter discussed two existing reinforcement learning methods; namely, AHC-learning and Q-learning. Both methods are based on temporal difference methods to learn an evaluation function of states or state-action pairs. The main idea is to write down a recursive definition of the target evaluation function and then incrementally construct a function to satisfy this definition. Here AHC-learning and Q-learning were extended to use neural networks as a generalization mechanism, which is expected to scale up reinforcement learning for problems with a large state space. As we have seen, the integration of neural networks and temporal difference methods is quite natural.

In addition to neural networks, three scaling-up extensions were also discussed:

- Experience replay. Sequences of experiences are stored and repeatedly replayed in temporally backward order. For good performance, experiences involving non-policy actions should not be replayed. For nondeterministic domains, the sampling of experiences for replay should reflect the actual probability distribution of multiple action outcomes.
- Using action models. Experiences are reused to build an action model. When the model is sufficiently good, it can replace the real world and allow an agent to mentally experience the effects of its actions without actually executing them.
- Teaching. Instructive training examples are generated by a human teacher, and replayed by the learning agent just like self-generated experiences.

Chapter 4

Scaling Up: Survival Task

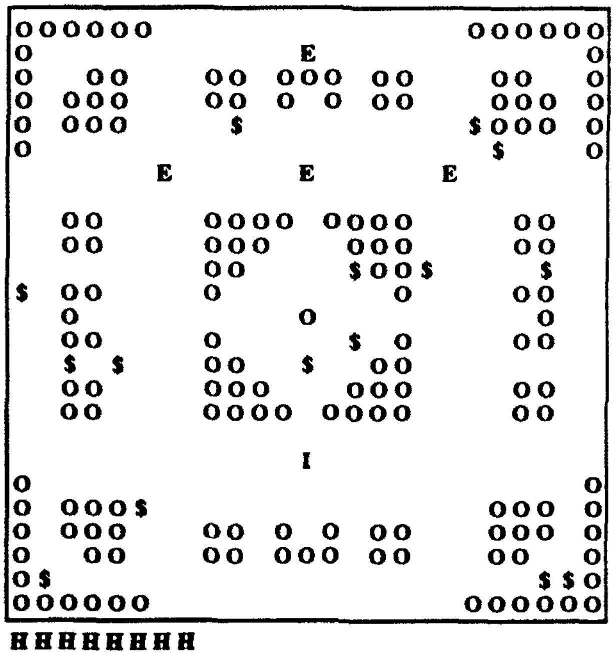
This chapter reports simulation experiments of the learning frameworks presented in the previous chapter. The learning task studied here is for an agent to survive in an unknown environment consisting of obstacles, food, and predators. The environment is nondeterministic and moderately complex. By testing the frameworks against such a hard learning task, we expect to see whether reinforcement learning can scale up and which framework may work better than another.

4.1 A Dynamic Environment

The learning domain used here to evaluate the various frameworks is a 25x25 cell world. A sample environment is shown in Figure 4.1. [1] There are four kinds of objects in the environment: the agent ("I"), food ("$"), enemies ("E"), and obstacles ("O"). The perimeter of the world is considered to be occupied by obstacles. The bottom of the figure is an energy indicator ("H"). At the start, the agent and four enemies are placed in their initial positions as shown in Figure 4.1, and fifteen pieces of food are randomly placed on unoccupied cells. On each move, the agent has four actions to choose from; it can walk to one of the four adjacent cells. If the agent attempts to walk into obstacles, it will remain at the same position.

After the agent moves, each of the enemies is allowed to stay or move to an adjacent cell that is not occupied by obstacles. To allow the agent to escape from chasing enemies, the enemy speed is limited to 80% of the full speed of the agent. The enemies move randomly, but tend to move toward the agent; the tendency becomes stronger as the agent gets closer. Appendix A.1 gives the algorithm for choosing enemy actions.

[1] It may appear that the learning task is too simplified by making the obstacles symmetric. That is not completely true, because there are also other objects in the world—enemies and food pieces are positioned randomly and with no symmetrical pattern. The number of different input patterns that the agent is likely to come across is estimated to be greater than 2^{50}.



```
OOOOOO             OOOOOO
O           E           O
O  OO   OO OOO OO   OO  O
O OOO   OO O O OO   OOO O
O OOO    $         $OOO O
O                   $   O
      E     E     E

  OO    OOOO OOOO    OO
  OO    OOO   OOO    OO
        OO    $OO$    $
$ OO    O       O    OO
  O         O         O
  OO    O     $ O    OO
  $ $   OO  $  OO
  OO    OOO   OOO    OO
  OO    OOOO OOOO    OO

            I
O                       O
O OOO$              OOO O
O OOO   OO O O OO   OOO O
O  OO   OO OOO OO   OO  O
O$                    $$O
OOOOOO             OOOOOO
HHHHHHHH
```

Figure 4.1: A dynamic environment involving an agent(I), enemies(E), food($), and obstacles(O). The H's indicate the agent's energy level.

The agent has two goals: to get as much food as possible and to avoid being caught by enemies. Note that the two goals conflict in some situations, and the agent must learn to arbitrate between them. A play ends when the agent gets all of the food or dies. The agent dies when it either collides with an enemy or runs out of energy. At the start of a new play, the agent is given 40 units of energy. Each piece of food provides the agent 15 units of additional energy, and each move costs the agent 1 unit. These parameter values were empirically chosen so that survival in the environment would not be too easy or too difficult.

To make the learning task more interesting and realistic, the agent is allowed to see only a local area surrounding it. From the agent's point of view, the world is nondeterministic, not only because the enemies behave randomly, but also because the world is only partially observable (thus, what will be seen after a move is not completely predictable). Although a human player with the same field of view can avoid the enemies and get all of the food most of the time, survival in this environment is not trivial. To survive, the agent must learn to (1) approach food, (2) escape from enemies, (3) avoid obstacles, (4) identify and stay away from certain dangerous places, (e.g., corridors), where it can be easily seized, and (5) seek food when food is out of sight.

4.2 Learning Agents

This section presents the implementation of four AHC-agents (i.e., Agents AHCON, AHCON-R, AHCON-M, and AHCON-T) and four Q-agents (i.e., Agents QCON, QCON-R, QCON-M, and QCON-T), which learn to survive in the environment. The agents are named after the learning frameworks on which they are based. In order to compare their performance, the agents use exactly the same reinforcement signals and sensory inputs as described below. Note that once the agents die, they get re-incarnated to try again, and in the meanwhile their learned networks are preserved.

All of the connectionist networks are trained using a symmetrical version of the error back-propagation algorithm; the squashing function is a variation of the sigmoid function: $f(x) = 1/(1 + e^{-x}) - 0.5$.

4.2.1 Reinforcement Signals

After each move, each learning agent receives one of the following reinforcement signals:

- -1.0 if the agent dies,
- 0.4 if the agent gets food,
- 0.0 otherwise.

Negative reinforcement is considered bad, and positive is good. The food reward is smaller than the penalty for being dead, since it is more important to stay alive. The food reward, 0.4, was chosen empirically; a few different values were tried for Agent AHCON, and 0.4 gave roughly the best performance. Theoretically speaking, the agent should learn to get food even without food rewards, because the agent is required to get food to stay alive. But the sooner good action decisions are rewarded, the sooner the agent should learn.

4.2.2 Input Representations

As described in Chapter 3, AHC-agents have a V-net and a policy net, while Q-agents have a Q-net. This subsection describes the input representation for the networks, while another subsection describes the output representation.

The V-net, Q-net, and policy net are all structurally similar; each of them is a feed-forward network, consisting of 145 input units, 1 output unit, and a layer of hidden units. (The number of hidden units is a parameter to be tuned for performance.) The networks are fully connected except that there are no connections between the input and output units. The input units of the networks can be divided into five groups: enemy map, food map, obstacle map, energy level, and some history information. Each of the maps shows a certain kind of object in a local region surrounding the agent and can be thought of as being obtained by an array of sensors fixed

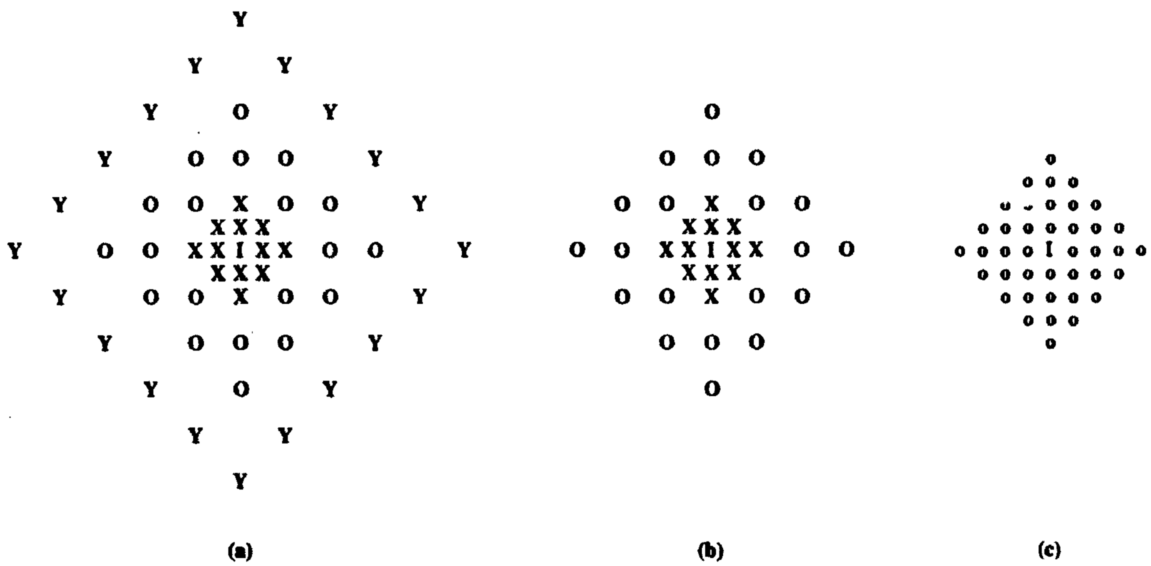


Figure 4.2: The (a) food, (b) enemy, and (c) obstacle sensor arrays.

on the agent. The sensor array moves as the agent moves. If the local action representation (see Section 4.2.3) is used, the array also rotates as the agent rotates. Figure 4.2 shows the configuration of the food, enemy and obstacle sensor arrays.

Each food sensor may be activated by several nearby food objects and each food object may activate several nearby food sensors. The food sensor array is composed of three different types of sensors, types "X", "O", and "Y". Different food sensor types have different resolution and different receptive fields— "X", "O", and "Y" sensors can be activated by food at any of the five, nine, and thirteen nearby cells, respectively. This technique of encoding spatial positions of objects is known as *coarse coding* [24]. Through the use of multiple resolution and coarse coding techniques, the food positions are effectively coded without loss of critical information.

The enemy sensor array has a similar layout of the food sensor array, except that it consists of only "X" and "O" types of sensors. The obstacle sensors are organized differently; there is only one type of obstacle sensor, and each obstacle sensor is only activated by an obstacle at the corresponding cell. The coarse coding technique is not used to encode obstacle positions, because it only works effectively when the features to be encoded are sparse [24], and this is not the case for obstacles.

The agent's energy level is, again, coarse-coded using sixteen input units. Each of the sixteen units represents a specific energy level and is activated when the agent's energy level is close to that specific level. Finally, four units are used to encode the agent's previous action choice and one unit to indicate whether or not the previous action resulted in a collision with an obstacle. These five units convey a kind of history information, which allows the agent to learn heuristics such as "moving back to previous position is generally bad", "when no interesting objects (e.g., food) are around, keep moving in the same direction until you see something interesting", etc.

4.2.3 Action Representations

The agent has four actions. This work experimented with two different representations of agent actions:

- **Global representation:** The four actions are to move to the north, south, west and east, regardless of the agent's current orientation in the environment.
- **Local representation:** The four actions are to move to the front, back, right and left relative to the agent's current orientation.

If the local representation is used, the agent begins with a random orientation for each new play, and its orientation is changed to the move direction after each step. For instance, if the agent takes four consecutive "right" moves and no collision occurs, it will end up being at the same cell with the same orientation as before.

4.2.4 Output Representations

The single output of the V-net represents the estimated utility of the input state. Corresponding to the action representations, there are two different output representations for the policy net and Q-net.

Global representation: With this action representation, the agent uses only one network to implement the policy net. The network has a single output representing the merit of moving to the "north". The merits of moving in other directions can be computed using the same network by rotating the state inputs (including food map, enemy map, obstacle map, and the 4 bits encoding the agent's previous action) by 90, 180, and 270 degrees. Similarly, the agent uses only one network to implement the Q-net. The network has a single output representing the utility of moving to the "north" in response to the input state. Again, the utilities of moving in other directions can be computed by rotating the state inputs appropriately. By taking advantage of action symmetry, the learning task is simplified, because whatever is learned for a situation is automatically carried over to situations which are more or less symmetric to it.

Local representation: This representation does not take advantage of action symmetry. Four networks (one for each action) are used to implement the policy net and Q-net, and each network has a single output.

All of the network outputs are between -1 and $+1$. By Definition (3.1), the V-function and Q-function may be greater than 1 when the agent is close to many pieces of food. In such (rare) cases, they are truncated to 1 before being used to compute TD errors. Also, the output units of the V-net and Q-net use mainly the linear part of the sigmoid function.

4.2.5 Action Models

Agents AHCON-M and QCON-M learn an action model. The action model is intended to model the input-output behavior of the dynamic environment. More specifically, given a world state and an action to be executed, the model is to predict what reinforcement signal will be received, where the food, enemies and obstacles will appear, and what the agent's energy level will be. In this nondeterministic environment, each action can have many possible outcomes. There are two alternatives to modeling the environment: the action model can generate either a list of outcomes associated with probabilities of happening or only the most likely outcome. The second alternative is adopted, because of its simplicity.

Since the food and obstacles do not move, their positions were found to be quite easy to predict using connectionist networks. So, to shorten simulation time without significantly simplifying the model-learning task, Agents AHCON-M and QCON-M were only required to learn *reinforcement networks* for predicting the immediate reinforcement signal and *enemy networks* for predicting the enemy positions. The reinforcement and enemy networks are single-layer networks (i.e., without hidden layers). The reinforcement networks use as inputs all of the 145 input bits mentioned before, while the enemy networks use only the enemy and obstacle maps. The single output of each enemy network is trained to predict whether a particular enemy sensor will be turned on or off after the agent moves.

The reinforcement and enemy networks are learned on-line just like the other networks. Learning these networks is a kind of supervised learning, and the encountered experiences (i.e., x, a, y, r) can be saved in a queue (of limited length) for repeated presentation to the networks. Because the enemies are nondeterministic and the agent does not know the exact enemy positions due to coarse coding, the prediction of enemy positions was found to be often incorrect even after the networks were well-trained.

To avoid using a completely senseless model, Agents AHCON-M and QCON-M in fact do not perform relaxation planning for the first 10 plays. Since it is also interesting to see what performance they will have if a perfect model can be learned quickly, AHCON-M and QCON-M are also provided with a "perfect" model, which is simply the environment simulator. Section 4.4.3 presents a performance comparison between using a perfect model and using a learned, potentially incorrect one.

4.2.6 Active Exploration

As described in Chapter 3.3 and 3.4, the learning agents use a stochastic action selector as a crude strategy for active exploration. The stochastic action selector uses a temperature parameter T to control the randomness of action selection. For better performance, a complementary strategy is also used: Each learning agent dead-reckons its trajectory and increases the temperature T whenever it finds itself stuck in a small area without getting food. A similar strategy is also used in Chapter 5. For both domains, this strategy was found good enough, although further improvement should be possible to obtain by using a better exploration strategy.

4.2.7 Other Details

Agents AHCON-R, QCON-R, AHCON-T, and QCON-T use the following heuristic strategies to control the amount of experience replay and to prevent the over-replay problem mentioned in Chapter 3.5.2 and 3.7.2:

- After each play, each agent replays n lessons chosen randomly from the most recent 100 experienced lessons, with recent lessons exponentially more likely to be chosen. n decreases linearly over time and is between 12 and 4. Appendix A.2 gives the algorithm for choosing experiences stochastically.
- After each play, Agents AHCON-T and QCON-T also stochastically choose taught lessons for replay. Each of the taught lessons is chosen with a decreasing probability between 0.5 and 0.1.

The numbers given above are not critical to the agents' performance. In fact, replaying more experiences may result in better performance, but it also demands more computational time.

Recall that only policy actions are replayed. Policy actions are those whose probability of being chosen is greater than P_l. For AHCON-R, $P_l = 0.2$ was found to give roughly the best performance for the survival task. For QCON-R, $P_l = 0.1, 0.01$ and 0.001 were tried and found to give similar performance. ($P_l = 0$ indeed gave poor performance.) In other words, AHCON-R is quite sensitive to replaying non-policy actions, while QCON-R is much less sensitive. The difference in sensitivity can be explained by the fact that replaying non-policy actions, in the case of using look-up tables, is bad for AHC-learning but is fine for Q-learning (see Chapter 3.5.1).

The experimental results reported below were obtained with $P_l = 0.2$ for AHC-agents and $P_l = 0.01$ for Q-agents. Since the taught lessons I gave to the agents were nearly optimal, P_l was in fact set to 0 during replaying taught lessons.

Recall also that Agents AHCON-M and QCON-M mentally experience only promising actions. In this simulation study, an action is considered promising if its probability of being chosen is greater than 2%.

4.3 Experimental Results

This section presents performance of the various learning agents. In the first study, the agents used the global action representation and took advantage of action symmetry. In the second study, the agents used the local action representation and did not exploit action symmetry.

4.3.1 Experimental Designs

Each study consisted of 7 experiments. (Each experiment took a Sparc Station two days to complete.) For each experiment, 300 training environments and 50 test environments were randomly generated. The agents were allowed to learn only from playing the 300 training environments. Each time an agent played 20 training environments, it was tested on the 50 test environments with learning turned off and the temperature T set to zero. The average number of food pieces obtained by the agent in the test environments was then plotted versus the number of training environments that the agent had played so far. The learning curves presented below show the mean performance over all of the 7 experiments.

For both studies, two taught lessons were provided for Agents AHCON-T and QCON-T to use. To generate a lesson, I pretended to be the agent trying to survive in a manually-chosen environment, which involved a few instructive situations. I got all the food in each of the lessons.

Each learning agent has several parameters that can be tuned for performance:

- γ: the discount factor (fixed to be 0.9);
- T: the temperature for the stochastic action selector;
- H_v, H_p and H_q: the number of hidden units of the V-nets, policy nets, and Q-nets;
- η_v, η_p and η_q: the learning rate of the back-propagation algorithm for the V-nets, policy nets, and Q-nets;
- the momentum factor of the back-propagation algorithm (fixed to be 0.9 for all networks);
- the range of the random initial weights of networks (fixed to be 0.1).

4.3.2 Study 1: Using Global Action Representation

In this study, the agents used the global action representation and exploited action symmetry. Table 4.1 shows the parameter settings used to generate the learning curves shown in Figure 4.3. Those parameter values were empirically chosen to give roughly the best performance for Agents AHCON and QCON. Little search was done for the other agents. AHC-agents used a temperature much higher than that used by Q-agents, because action merits are supposed to approach 1 for the best action(s) and -1 for the others, while utilities of state-action pairs are usually small numbers between 1 and -1. Cooling temperatures were used, although fixed low temperatures seemed to work as well. The learning agents with learning-speedup techniques used smaller learning rates than those used by the basic learning agents, because the former adjusted their networks more often.

4.3.3 Study 2: Using Local Action Representation

In this study, the agents used the local action representation and did not exploit action symmetry. The agents used the same parameter settings as in Study 1, except that all the learning rates

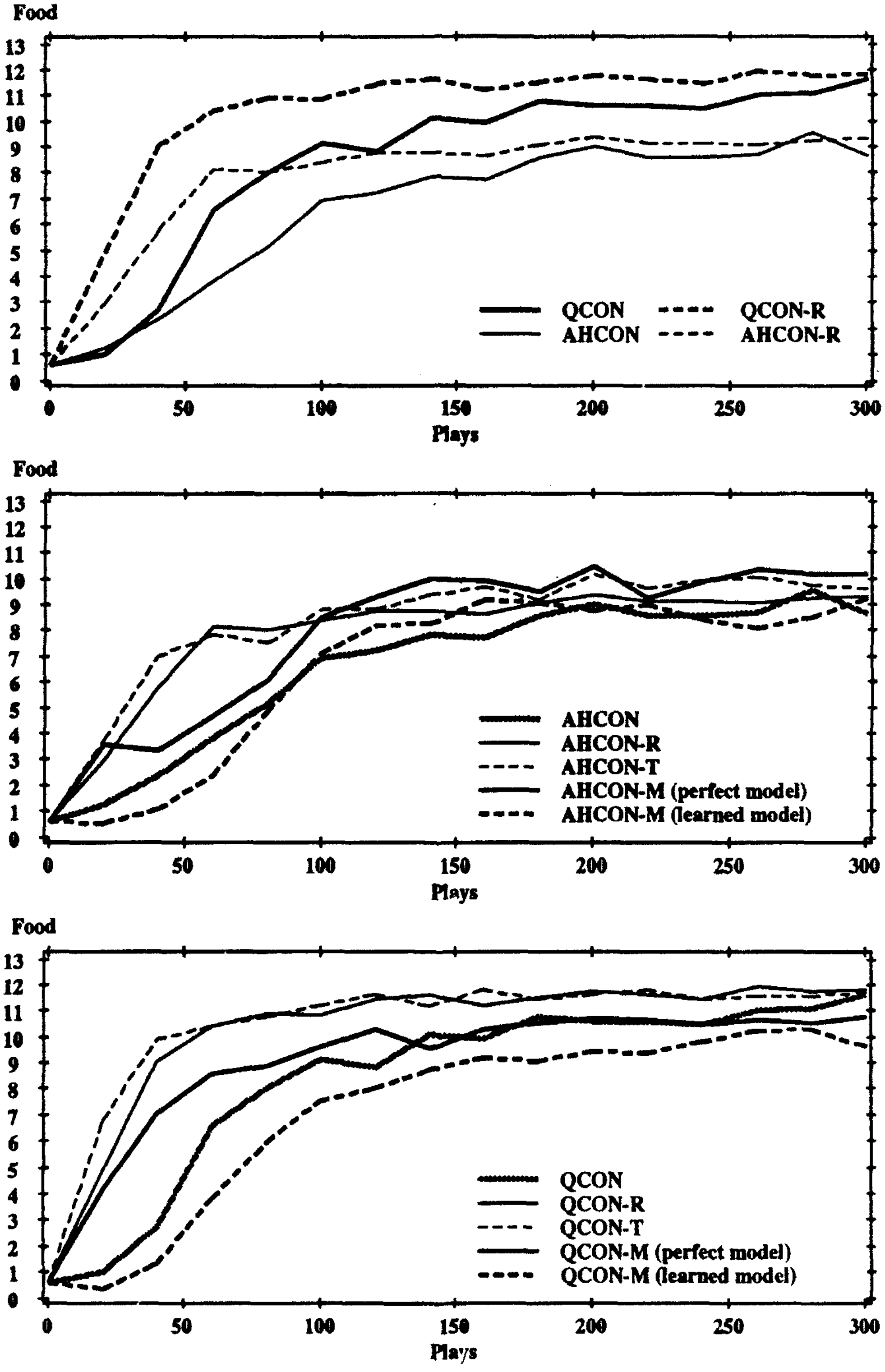


Figure 4.3: Learning curves of the agents using the global action representation.

Table 4.1: Parameter values used for Study 1, which used a global action representation

Agent	H	η	T
AHCON	$H_v = 30$	$\eta_v = 0.2$	$1/T = 2 \rightarrow 10$
	$H_p = 30$	$\eta_p = 0.4$	
other AHC-agents	$H_v = 30$	$\eta_v = 0.1$	$1/T = 2 \rightarrow 10$
	$H_p = 30$	$\eta_p = 0.2$	
QCON	$H_q = 30$	$\eta_q = 0.3$	$1/T = 20 \rightarrow 60$
other Q-agents	$H_q = 30$	$\eta_q = 0.15$	$1/T = 20 \rightarrow 60$

were doubled. Figure 4.4 shows the learning curves of the agents.

4.3.4 Observations

Among all the learning agents, QCON-T was roughly the best one. To see the absolute performance of the best agent after 300 plays, I tested QCON-T against 8000 randomly generated environments. The following table shows how often the agent got all the food, got killed, or ran out of energy.

got all food	got killed	ran out of energy
39.9%	31.9%	28.2%

In average the agent got 12.1 pieces of food in each play. The table below shows how often the agent got no food, 1 piece of food, more pieces, or all of the 15 pieces.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
%	0.1	0.3	0.8	1.8	2.2	2.9	4.0	4.1	3.8	3.7	3.4	4.1	5.4	8.2	15.2	39.9

As mentioned previously, the learning curves in Figures 4.3 and 4.4 show the mean performance over 7 experiments. The standard deviations of these curves, point by point, are roughly 2 food pieces in the beginning of learning and 1 food piece at the end of the experiments. Generally speaking, the agents who had better performance had smaller performance deviations between experiments. But an agent's performance difference between experiments did not imply much about the repeatability of experimental results, because each experiment used a different set of training and test environments. What really matters is the relative performance of agents, which was found consistent most of the time through each entire experiment.

4.4 Discussion

This section compares the learning agents by analyzing the experimental results in Section 4.3. In each of the coming subsections, I first compare the agents from the aspects of learning

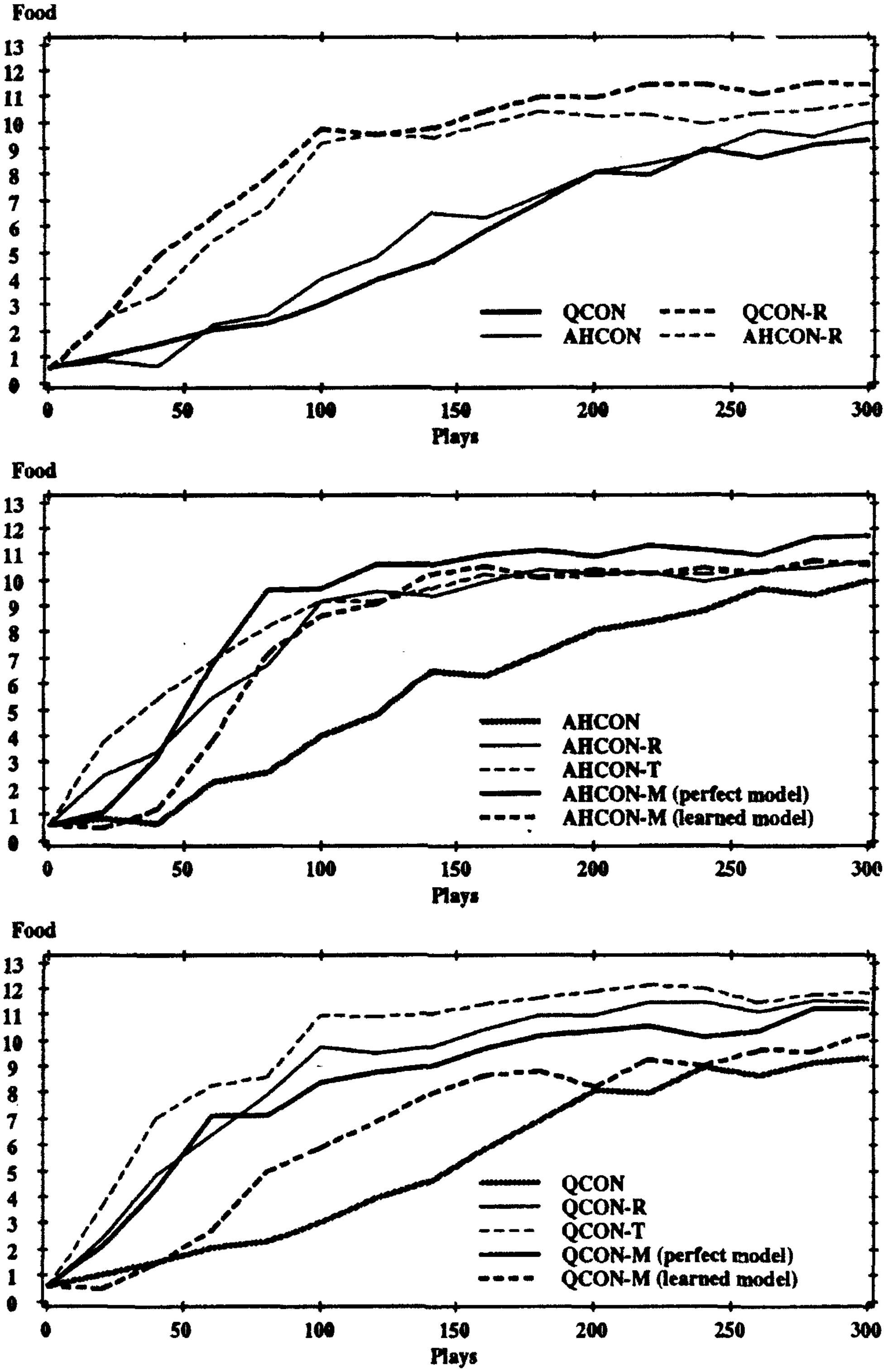


Figure 4.4: Learning curves of the agents using the local action representation.

speed and final performance at the end of the experiments, and then discuss implications of the results. In the comparison, I use $X > Y$ to mean agent X is better than agent Y. Similarly, $<$ means worse, $\approx$ means similar, $\geq$ means slightly better, and $\leq$ means slightly worse.

4.4.1 AHC-agents vs. Q-agents

	learning speed	final performance
Global	Q-agent > AHC-agent	Q-agent > AHC-agent
Local	Q-agent ≈ AHC-agent	Q-agent ≈ AHC-agent

In the first study with the global action representation, Q-agents performed better than AHC-agents. A number of previous studies [35, 67] also found the superiority of Q-learning over AHC-learning. But Study 2 with the local action representation did not confirm this superiority. It is still an open question whether Q-learning works better than AHC-learning in general.

Q-agents' final performance was basically unaffected by using different action representations, while for AHC-agents, the local action representation apparently worked better than the global action representation as far as the final performance is concerned. This result suggests that different action representations, as well as different input representations, may make a big difference in the performance of a learning system.

4.4.2 Effects of Experience Replay

	learning speed	final performance
Global	AHCON-R > AHCON	AHCON-R ≈ AHCON
	QCON-R > QCON	QCON-R ≈ QCON
Local	AHCON-R > AHCON	AHCON-R ≥ AHCON
	QCON-R > QCON	QCON-R > QCON

Experience replay indeed made learning converge faster, while the final performance was unchanged in the case of using the global representation. In general, asymptotic performance is expected to remain the same no matter whether experience replay is used or not.

4.4.3 Effects of Using Action Models

		learning speed	final performance
Global	perfect model	AHCON-M > AHCON	AHCON-M ≥ AHCON
		QCON-M > QCON	QCON-M ≈ QCON
	learned model	AHCON-M ≈ AHCON	AHCON-M ≈ AHCON
		QCON-M < QCON	QCON-M < QCON
Local	perfect model	AHCON-M > AHCON	AHCON-M > AHCON
		QCON-M > QCON	QCON-M > QCON
	learned model	AHCON-M > AHCON	AHCON-M ≥ AHCON
		QCON-M > QCON	QCON-M ≈ QCON

This result suggests that whether an action model helps depends on the relative difficulty of acquiring the model and learning a control policy directly. If a perfect model is available, it is always useful to use the model. If a sufficiently good model can be learned faster than a good control policy (such as Study 2 with the local action representation), it is worthwhile to learn the model. But if it is the opposite case (such as Study 1 with the global action representation), a potentially incorrect model can only be misleading and hamper the learning of a control policy in the early stage of learning. (In both studies, the model's prediction accuracy approached its asymptote after approximately 50 plays.)

The agents never acquired a nearly perfect model for this nondeterministic world, even after hundreds of plays. This suggests that *a model need not be perfect to be useful.*

As speculated in Chapter 3.6, relaxation planning should be needed less and less as learning proceeds. This speculation was confirmed in both studies. Figure 4.5 shows the average number of hypothetical actions that were taken on each step. (The curves show the mean results from the 7 experiments of Study 2.) The maximum is 4, since there are 4 actions to choose from on any step. As we can see from the figure, the amount of planning dropped very quickly as the learning curves leveled off. But it did not drop close to zero at the end, because in this task there are many situations where several actions are in fact almost equally good and therefore they all ended up being tried in simulation. For tasks where only a small portion of actions are relevant most of the time, the saving of relaxation planning should be more significant.

4.4.4 Effects of Teaching

	learning speed	final performance
Global	AHCON-T ≈ AHCON-R	AHCON-T ≈ AHCON-R
	QCON-T ≈ QCON-R	QCON-T ≈ QCON-R
Local	AHCON-T ≥ AHCON-R	AHCON-T ≈ AHCON-R
	QCON-T > QCON-R	QCON-T ≈ QCON-R

Recall that AHCON-T/QCON-T is AHCON-R/QCON-R plus replaying taught lessons. Because the task in the first study (with the global action representation) was not very difficult

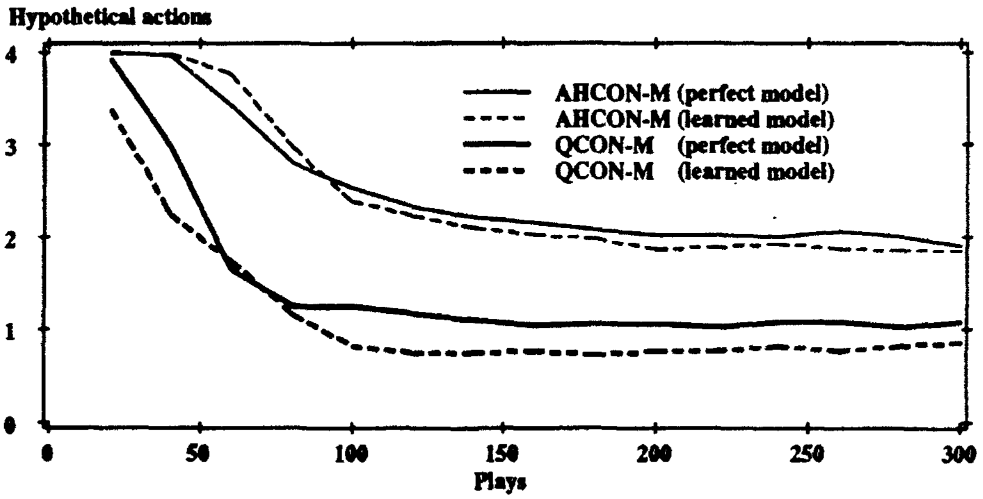


Figure 4.5: Average number of hypothetical actions taken on each step (from Study 2).

and experience replay alone had done a good job, there was not much improvement we could expect from teaching. In the second study, there was a noticeable speed advantage to using teaching experiences. This result suggests that the advantage of teaching should become more significant as the learning task gets more difficult. Indeed, Chapter 5 reports similar results, which are obtained from a simulated mobile robot.

4.4.5 Experience Replay vs. Using Action Models

		learning speed	final performance
Global	perfect model	AHCON-R ≥ AHCON-M	AHCON-R ≤ AHCON-M
		QCON-R > QCON-M	QCON-R ≥ QCON-M
	learned model	AHCON-R > AHCON-M	AHCON-R ≈ AHCON-M
		QCON-R > QCON-M	QCON-R > QCON-M
Local	perfect model	AHCON-R ≤ AHCON-M	AHCON-R ≤ AHCON-M
		QCON-R ≈ QCON-M	QCON-R ≈ QCON-M
	learned model	AHCON-R ≥ AHCON-M	AHCON-R ≈ AHCON-M
		QCON-R > QCON-M	QCON-R > QCON-M

When the agents had to learn a model themselves, there was clear, consistent superiority of experience replay over using action models. But when the agents were provided with a perfect action model, there was no clear, consistent superiority of one over the other. This result can be simply explained by the fact that it took some time for the agents to learn a sufficiently good model before the agents could start taking advantage of it.

Why couldn't relaxation planning with a perfect action model outperform experience replay? Two explanations:

1. The relaxation planning algorithms used here may not be the most effective way of using action models. For example, the number of hypothetical experiences that AHCON-M and QCON-M generated and used was limited by the the number of states actually visited.

2. *Experience replay is also a kind of relaxation planning.* By sampling past experiences, experience replay in effect uses an action model, but it need not explicitly build one (in the sense of doing curve-fitting, nearest neighbor, or the like). In essence, the collection of past experiences is a model. It represents not only explicitly the environment's input-output patterns but also implicitly the probability distributions of multiple outcomes of actions.

Experience replay is effective and easy to implement. The main cost of using it is the extra memory needed for storing experiences. So, is it of no use to learn an action model? The answer is unclear. If a model is learned *merely* for doing relaxation planning, perhaps it is *not* worthwhile to learn a model, since experience replay does the same thing as relaxation planning. One may argue that: (1) a model that generalizes can provide induced or interpolated experiences which are not seen before, and (2) the extra experiences will result in better learning of the evaluation functions. But if the evaluation functions also generalize (as it should be the case for non-toy tasks), it is unclear whether these extra experiences can actually do any extra good.

On the other hand, having an action model can be useful. In this work I only investigated how action models can be used for learning an evaluation function. There are other ways of using a model to improve performance. For example, we can use an evaluation function, an action model, and a look-ahead planning technique to help find the best actions [80, 72]. In a complex domain where an optimal policy may be hardly obtainable, by looking ahead a few steps (much as computer chess does), the non-optimality of a policy can be compensated, if an accurate action model is available. How to use action models effectively is an interesting issue and needs further study.

4.4.6 Why Not Perfect Performance?

Why did all the agents fail to reach the perfect performance (i.e., get all the food)? There are at least two reasons: (1) The local information used by the agents may be insufficient to determine the optimal actions, and (2) the perfect policy may be too complex to be represented by the connectionist networks used here. As a matter of fact, I also played against the simulator myself. Being allowed to see only objects in the local region as the learning agents do, I got all the food pieces most (but not all) of the time. I found that I often employed two techniques to play it so successfully: look-ahead planning (although very crude) and remembering food positions so that I could come back to get the food after I lost the sight of food because of chasing enemies. The learning agents did not use either of the two techniques.

4.5 Summary

In this chapter, AHC-learning, Q-learning, neural networks, experience replay, using action models, and teaching all have been tested against a moderately complex learning task—surviving in an unknown environment. The size of the task's state space is estimated to be greater than 2^{50}. The main simulation results are summarized as follows:

- Neural networks generalized effectively even with a large number (145) of binary inputs. Without generalization, it would take an agent a large number of plays, instead of just 300 plays, to develop a good survival strategy for such a large state space.
- In one case Q-learning outperformed AHC-learning, while in the other case both were similarly effective.
- Experience replay made learning converge faster, but would not be expected to affect the asymptotic performance.
- An action model need not be perfect to be useful, but a very poor model may mislead the learner and hamper, instead of improve, the learning speed. Whether it is useful to learn a model may depend on the relative difficulty of model-learning and modelless control-learning.
- Teaching alone was not found to result in a significant speedup here. Perhaps, the survival task is not difficult enough to demonstrate the actual utility of teaching.

To have a rough idea about how much speedup can be obtained from the extensions, here I take the Q-agents in Study 2 for example and show the obtained speedup in the following table. The second column of the table shows the number of plays needed by each agent to reach the performance of 8 (i.e., 8 food pieces per game), and the third column shows the speedup (compared with the agent without any extension). A speedup of x means that the number of plays needed is reduced to $1/x$ of the originally needed number.

	# of plays	speedup
no extensions	200	
using a learned model	140	1.4
using a perfect model	95	2.1
experience replay	80	2.5
experience replay + teaching	60	3.3

Chapter 5

Scaling Up: Mobile Robot Tasks

The simulation experiments reported in the previous chapter indicate that among the eight reinforcement learning frameworks presented in Chapter 3, QCON-T appears to work best. QCON-T integrates Q-learning, neural networks, experience replay, and teaching.

This chapter presents more simulation experiments with Framework QCON-T. The domain used here is a physically-realistic mobile-robot simulator. Control errors and sensing errors are simulated, and the robot operates in a continuous state space. The robot's tasks include wall following, door passing, and docking on a battery charger. The Q-functions to be learned are nonlinear. In other words, the robot must develop high-order state features in order to accomplish these learning tasks. Rewards in this domain in general are much less likely to be obtained by luck than those for the survival task (Chapter 4). This domain is more challenging than the survival task, allowing us to further demonstrate the utility of QCON-T and importance of teaching.

This chapter also extends the experience replay technique to use TD(λ) methods for general λ. Simulation experiments indicate that there is a speed advantage to using a nonzero λ value. Two possible ways of combining supervised learning and reinforcement learning are also discussed here, although their utility appears to be limited.

5.1 Using TD(λ)

This section extends the experience replay technique to use TD(λ) methods for general λ. For readers' convenience, two terms defined in Chapter 3 are repeated here:

- *Experience.* An *experience* is a quadruple, (x_t, a_t, x_{t+1}, r_t), meaning that at time t action a_t in response to state x_t results in the next state x_{t+1} and reinforcement r_t.
- *Lesson.* A *lesson* is a sequence of experiences starting from an initial state x_0 to a final state x_n where the goal is achieved or given up.

TD methods make learning changes in proportion to TD errors. The difference between TD(0) (the one used in the previous chapter) and TD($\lambda > 0$) is the way they compute TD errors. TD(0) errors are the difference between two successive predictions, while TD(λ) errors are the sum of TD(0) errors exponentially weighted by recency. A simple example illustrating the difference can be found in Chapter 2.1.

Let U_t be the predicted utility of state x_t. In other words, U_t is the discounted cumulative reinforcement that is expected to be received starting from state x_t. Consider the application of TD(λ) methods to the following sequence of state transitions:

$$(x_0, a_0, x_1, r_0) \cdots (x_n, a_n, x_{n+1}, r_n).$$

Let Δ_t^λ be the TD(λ) error in the prediction about the utility of state x_t, and let Δ_t^0 be the TD(0) error. Δ_t^λ and Δ_t^0 are computed as follows:

$$\begin{aligned} \Delta_t^0 &= r_t + \gamma \cdot U_{t+1} - U_t && (5.1) \\ \Delta_t^\lambda &= \sum_{k=0}^{n-t-1} (\gamma\lambda)^k \Delta_{t+k}^0 && (5.2) \end{aligned}$$

Note that the discount applied to TD(0) errors in (5.2) includes γ as well as λ. We now rewrite (5.2) as

$$\begin{aligned} \Delta_t^\lambda &= \Delta_t^0 + \sum_{k=1}^{n-t-1} (\gamma\lambda)^k \Delta_{t+k}^0 \\ &= [r_t + \gamma \cdot U_{t+1} + \sum_{k=1}^{n-t-1} (\gamma\lambda)^k \Delta_{t+k}^0] - U_t \end{aligned}$$

We define

$$R_t^\lambda = r_t + \gamma \cdot U_{t+1} + \sum_{k=1}^{n-t-1} (\gamma\lambda)^k \Delta_{t+k}^0 \qquad (5.3)$$

R_t^λ is called the *TD(λ) return*, the estimated utility from state x_t. The TD(λ) error is the difference between the TD(λ) return and the current prediction:

$$\Delta_t^\lambda = R_t^\lambda - U_t \qquad (5.4)$$

We can rewrite (5.3) as

$$\begin{aligned} R_t^\lambda &= r_t + \gamma \cdot U_{t+1} + \lambda\gamma\Delta_{t+1}^\lambda \\ &= r_t + \gamma \cdot U_{t+1} + \lambda\gamma(R_{t+1}^\lambda - U_{t+1}) \\ &= r_t + \gamma \cdot [(1-\lambda)U_{t+1} + \lambda R_{t+1}^\lambda] && (5.5) \end{aligned}$$

Equation (5.5) is the basis of the experience replay algorithms to be presented below. When $\lambda = 0$, the TD return computed by Equation (5.5) is the actual payoff r_t plus the predicted utility of the next state U_{t+1} weighted by γ. (This definition is consistent with the one described in Chapter 3.) When $\lambda = 1$ and backward replay is used, the TD return computed by Equation (5.5) is exactly the discounted cumulative reinforcement that is actually received in the (replayed) action sequence. When $0 < \lambda < 1$, the TD return is some aggregate of the predicted return (namely, TD(0) return) and the actual return (namely, TD(1) return).

Recall that TD methods train an evaluation function by minimizing TD errors. In other words, TD methods train the function using TD returns as target outputs. Thus, we can say TD(1) is a supervised learning method, because TD(1) trains an evaluation function using as target outputs the returns that are actually received from the environment.

Recall also that agents based on AHC-learning learn an evaluation of states, $V(x)$, while agents based on Q-learning learn an evaluation of states and actions, $Q(x, a)$. To compute TD returns using AHC-learning, we simply plug

$$U_{t+1} = V(x_{t+1})$$

into Equation (5.5). The TD(λ) error is thus the difference between Equation (5.5) and the current prediction, $V(x_t)$. If the V-function is implemented using neural networks, we can use the back-propagation algorithm to reduce the TD error.

Similarly, to compute TD returns using Q-learning, we simply plug

$$U_{t+1} = Max\{Q(x_{t+1}, a) | a \in actions\}$$

into Equation (5.5). The TD(λ) error is thus the difference between Equation (5.5) and the current prediction, $Q(x_t, a_t)$. Again, the back-propagation algorithm can be used to train Q-nets.

Figure 5.1 shows two experience replay algorithms; one for Q-learning and the other for AHC-learning. For the sake of reducing the number of experiments in this chapter and the coming chapters, only Q-learning is considered in the rest of this dissertation. Because of this, from now on, when I mention "***the experience replay algorithm***", I refer to the one using Q-learning.

5.1.1 Choice of λ

Recall Equation (5.2)— Δ_t^λ is some weighted sum of Δ_t^0, Δ_{t+1}^0, Δ_{t+2}^0, and so forth. In other words, Δ_t^0 contributes to the learning changes applied to states before time t; namely, x_t, x_{t-1}, x_{t-2}, and so forth. The contribution decays exponentially by the temporal distance between state x_t and the state to which the learning change is applied. Let D be the temporal distance for which this contribution is decayed to $(1/e) \approx 0.37$ of the original amount:

$$\frac{1}{e} = \lambda^D$$

```
Algorithm for Q-learning agents to replay $\{(x_0, a_0, x_1, r_0) \cdots (x_n, a_n, x_{n+1}, r_n)\}$:
1. $t \leftarrow n$;
2. $e_t \leftarrow Q(x_t, a_t)$;
3. $u_{t+1} \leftarrow Max\{Q(x_{t+1}, k) \mid k \in actions\}$;
4. $e'_t \leftarrow r_t + \gamma[(1-\lambda)u_{t+1} + \lambda e'_{t+1}]$;
5. Adjust the Q-net for action $a_t$ by back-propagating the error $(e'_t - e_t)$ through it
   with input $x_t$;
6. If $t = 0$ exit; else $t \leftarrow t - 1$; go to 2;
```

(a)

```
Algorithm for AHC-learning agents to replay $\{(x_0, a_0, x_1, r_0) \cdots
   (x_n, a_n, x_{n+1}, r_n)\}$:
1. $t \leftarrow n$;
2. $e_t \leftarrow V(x_t)$;
3. $u_{t+1} \leftarrow V(x_{t+1})$;
4. $e'_t \leftarrow r_t + \gamma[(1-\lambda)u_{t+1} + \lambda e'_{t+1}]$;
5. Adjust the V-net by back-propagating the error $(e'_t - e_t)$ through it with input
   $x_t$;
6. Adjust the policy net for action $a_t$ by back-propagating the error $(e'_t - e_t)$ through
   it with input $x_t$;
7. If $t = 0$ exit; else $t \leftarrow t - 1$; go to 2;
```

(b)

Figure 5.1: The experience replay algorithms using (a) Q-learning and (b) AHC-learning.

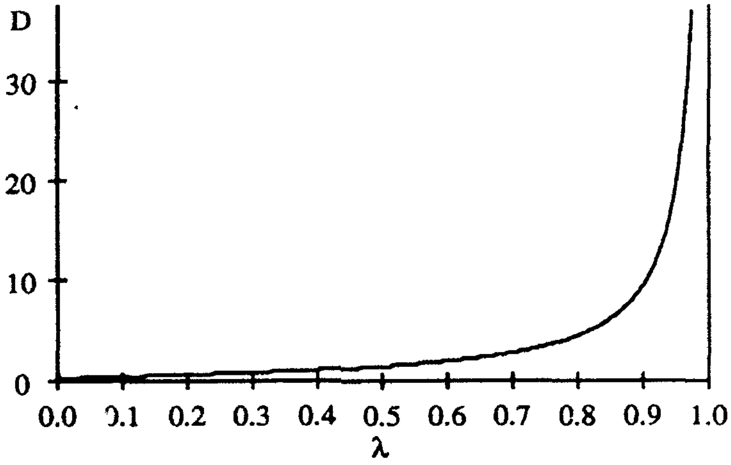


Figure 5.2: Relationship between D and λ.

In other words,

$$D = \frac{-1}{log_e(\lambda)}$$

Figure 5.2 shows the relationship between D and λ. As we can see, D increases gradually as λ increases toward 0.9, and increases rapidly as λ increases over 0.9.

How can one determine a proper λ value? Clearly, the larger the λ value, the farther credits can be propagated from goal states to preceding states at a time. But large λ values may not be desirable for certain reasons. The following is an informal analysis. Recall that the TD(λ) return computed in Equation (5.5) is some weighted sum of predicted return and actual return. Consider Q-learning. In the beginning of learning, the Q-function is normally far from correct, so the actual return is often a better estimate of the expected return than the return predicted by the Q-function. In this case, we may want to trust more the actual return by using a large λ value (say 1.0). On the other hand, as learning proceeds and the Q-function becomes more accurate, the Q-function provides a better prediction. In particular, when the replayed lesson is far back in the past and involves many bad choices of actions, the actual return (in that lesson) would be smaller than what really can be obtained using the current policy. In this case, we may want to trust more the predicted return by using a small λ value (say 0.0).

Thus, a good strategy might be to start with $\lambda = 1$ in the beginning of learning and gradually reduce it to 0 as learning goes on. In this dissertation, however, λ is kept constant and temporarily set to 0 in two situations (refer to Figure 5.1.a):

- when $t = n$, because e'_{t+1} in Step 4 is undefined, and
- when action a_{t+1} is not a policy action, because the actual return from executing this action would be an underestimate of the return obtainable by following the current (improved) policy.

As a matter of fact, non-policy actions will never be replayed as discussed in Chapter 3.5.

There were some empirical studies on the choice of λ. In one study, Sutton [66] found that $\lambda = 1.0$ gave the worst performance and that $\lambda = 0.3$ gave roughly the best performance. In another study, Tesauro [70] found that $\lambda = 0$ gave the best asymptotical performance, while there was a speed advantage to using a large λ value (say, 0.7).

5.1.2 Training Modes and Conditions

The experience replay algorithms can be used in batch mode or in incremental mode. In incremental mode the networks are adjusted after replaying each experience, while in batch mode the networks are adjusted only after replaying a whole lesson. The algorithms will do exactly what TD(λ) does only when they are used in batch mode. However, the advantage of backward replay is greatest when the incremental mode is used. Unless explicitly stated, the incremental mode was used in the experiments reported in this dissertation.

As mentioned in Chapter 3.5.2, the correctness of Q-learning and AHC-learning relies on the assumption that next states are visited in proportion to their probabilities. Experience replay may invalidate this assumption. Therefore, it is important to sample and replay experiences as uniformly as possible. It is also important not to be over-trained on teaching experiences.

5.2 A Mobile Robot Simulator

Many experiments in this dissertation are based on a mobile robot simulator called HEROINE (HEro RObot In a Novel Environment). HEROINE was intended to mimic a real robot called Hero [38, 39] and a real office environment. The simulated environment (Figure 5.3) consists of three rooms, A, B and C, and two narrow doorways connecting Room C to Rooms A and B. A battery charger is located in the left-bottom corner of Room A. A light is placed on top of the charger so that the robot can distinguish the charger from other obstacles.

5.2.1 The Robot and Its Sensors

The robot has 6 actions: turn ±15°, turn ±60°, and move ±12 inches. It has a sonar sensor and a light intensity sensor mounted on its top. Both sensors can rotate and collect 24 readings (separated by 15 degrees) per rotation. The sonar sensor returns a distance reading between 0 and 127 inches, with a resolution of 1 inch. In addition, the robot has a compass and four collision sensors placed around its body. The compass is not needed to execute the local navigation tasks studied in this chapter. Its use for global navigation will be discussed in Chapter 6.

The light intensity readings are thresholded to give 24 binary bits (called *light readings*);

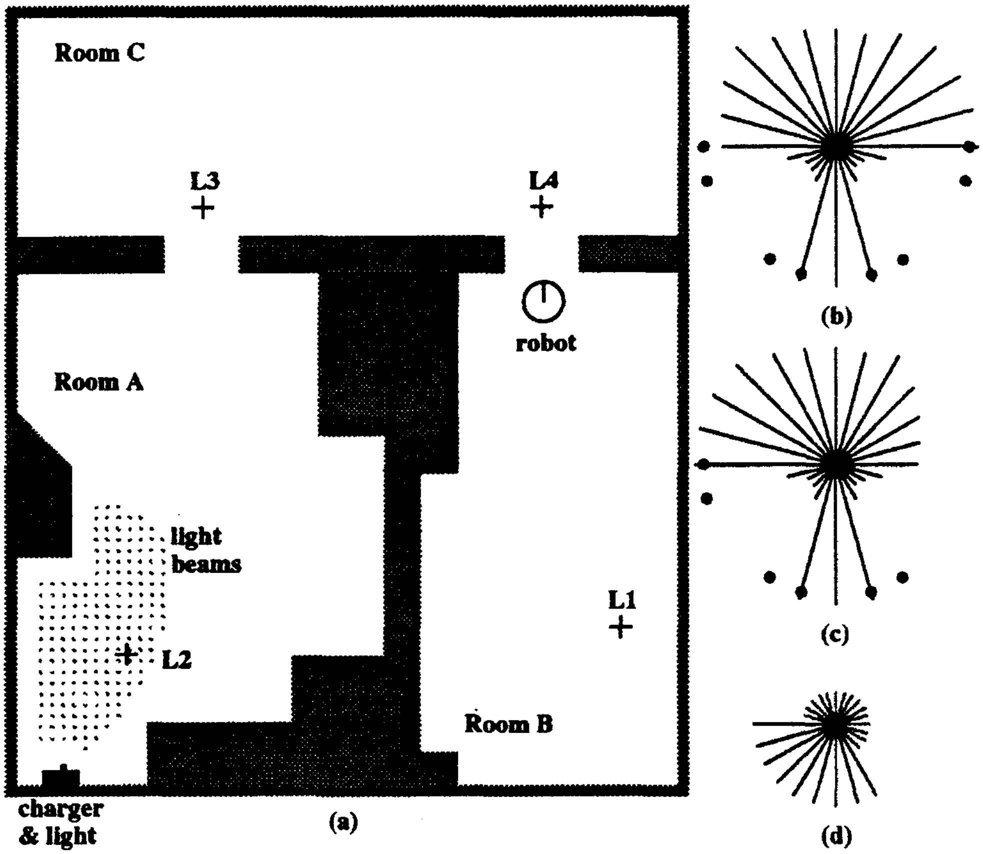


Figure 5.3: A Robot Simulator. (a) The robot and its environment. The environment consists of three rooms, two doors, and a battery charger with a light on its top. The radius of the robot is 12 inches. The dimensions of the environment are 360 inches by 420 inches. The robot can turn and rotate. It has sonar sensors, light intensity sensors, collision sensors, and a compass. The task is to find the battery charger and connect to it. The robot may start from any location with any orientation. It takes the robot approximately 100 steps to connect to the charger from Location L1. (b) Sonar readings obtained when the robot is at Location L3 and faces north. The dots indicate potential door edges. Four edges are detected here, but only two of them are real door edges. (c) Sonar readings obtained when the robot is at Location L4 and faces north. They are very similar to those in (b). (d) Binary light readings obtained when the robot is at Location L2 and faces north. A long line stands for the detection of light beams from that direction.

each indicates if there is light coming from a particular direction. The 24 sonar readings are also processed to give 24 binary bits (called *door edge readings*), each of which indicates if there is a potential door edge in a particular direction. The door edge feature is turned on when there is an abrupt change between neighboring sonar readings, The door edge readings are unreliable, because some obstacle edges may also look like door edges. See Figure 5.3 for an illustration.

To be realistic, about 8% control error has been added to the robot's actuators and about 10% sensing error added to the sensors. (For instance, let x be the distance between the robot and an object surface. The sonar reading will be uniformly distributed between $0.9x$ and $1.1x$) In the real world, sonar sometimes gets reflected by smooth surfaces or absorbed by soft objects, resulting in a maximum reading, 127. This sensing error is simulated by randomly selecting a sonar reading and setting it to 127 once in a while. In average, for every 4 sets of sonar readings the robot takes (24 readings per set), one of the readings is set to 127. Very often such a faulty reading looks like a door opening and causes faulty door edge readings.

5.2.2 The Robot's Tasks

The robot's ultimate task is to learn a control policy for finding the battery charger and connecting to it. It may start from any location with any orientation. This task, which is called the *battery recharging task* or CHG, is decomposed into four elementary tasks:

- following walls while keeping walls on the robot's right hand side (WFR),
- following walls while keeping walls on the robot's left hand side (WFL),
- passing a door starting from places near the door (DP), and
- docking on the battery charger starting from places where light beams are detected (DK).

This chapter focuses on the problem of learning skills (i.e., control policies) for carrying out the elementary tasks, while the next chapter presents a reinforcement learning approach to learning the coordination of previously learned skills. In that approach, the robot learns a high-level control policy, which chooses among the four skills to accomplish the battery recharging task. For instance, to connect to the battery charger from Location L1 (see Figure 5.3), the robot simply executes WFR to get to the vicinity of the right door, executes DP to get out of Room B, executes WFL to reach the left door, executes DP again to get into Room A, executes WFR again to get close to the charger, and finally executes DK to get connected to the battery charger.

5.3 The Learning Robot

This section describes the reinforcement functions, input representations, network structures, and parameter settings that the robot uses to learn the wall following, door passing, and docking tasks. The learning technique is based on the experience replay algorithm (Figure 5.1.a) and teaching.

5.3.1 Reinforcement Functions

The robot receives the following reinforcement signals for docking:

- 100 if it successfully docks, or
- −10 if a collision occurs, or
- 0 otherwise.

The robot receives the following reinforcement signals for door passing:

- 100 if the robot successfully passes a door, or
- −10 if a collision occurs, or
- 0 otherwise.

The robot receives the following reinforcement signals for wall following:

- −10 if a collision occurs, or
- −3 if the executed action is moving 12 inches backward, or
- 0 if the robot is too close to walls on either side, or
- 0 if the robot is too far from walls on the specified side, or
- 8, 9, or 10 depending on the alignment of the robot's body to walls (see below).

When executing WFL (or WFR), the robot is requested to maintain a clearance between its body and obstacles on the left (or right) hand side. Any clearance between 14 inches and 32 inches is acceptable. The robot receives a small penalty when moving backward. This penalty is needed to keep the robot moving forward, because otherwise the robot, in certain situations, may maximize rewards by moving back and forth.

When the executed action is moving forward and a desirable clearance is maintained, the robot receives a reward between 8 and 10. Comparing a few sonar readings from the right-hand side (or left-hand side in the case of WFL), the reinforcement mechanism determines how well the robot's body is parallel to walls. If they are almost parallel, the maximum reward (10) is received, or else a smaller reward is received. A constant reward, say 10, may be instead used, but the robot was found to learn a more desirable wall following skill with this variable reward than a constant reward.

The robot uses $\gamma = 0.9$ for all the tasks. The reinforcement values for each of the tasks were chosen in order to keep the Q-values between −100 and 100.

5.3.2 Input Representations

For the docking task, the robot uses 52 input units to encode the state information, including 24 sonar readings, 24 (binary) light readings, and 4 (binary) collision readings. A few different ways of encoding sonar data have been tested. They all gave reasonable performance. Among them, the best one was to encode readings ranging from 0 to 64 as values between 1 and 0 and to encode readings ranging from 64 to 127 as values between 0 and -0.25. (Linear scaling is used.) Smaller readings are weighted more heavily because the robot has to pay more attention to nearby obstacles than to obstacles far away. For all of the four robot tasks studied here, sonar readings are encoded in this way. As usual, binary inputs are represented by 0.5 and -0.5.

For the door passing task, the robot uses 52 input units to encode the state information, including 24 sonar readings, 24 (binary) door edge readings, and 4 collision readings.

For both wall following tasks, 24 units are used to encode sonar readings, and 4 units to encode collision readings. In addition, 2 real-valued units are used to indicate how well the robot's body is parallel to obstacles on both sides, and 6 binary bits to indicate whether the clearance between the robot's body and obstacles on either side is too small, too large, or just right. The last 8 bits of information are not critical to successful learning. Without them, the robot could learn to follow walls reasonably well. But it learned better and faster with them.

Note that the input representations used here are *robot-centered representations*. For example, the first element of the state vector always encodes the sonar reading coming from the direction in which the robot is facing. This kind of local-coordinate representations is desired, because the four tasks are local navigation tasks. For example, with this type of representation, once a wall following skill is learned to follow a wall going from north to south, it can apply equally well to following walls going from east to west or in any other direction.

5.3.3 Q-nets

The Q-function for each of the tasks is implemented by 6 feed-forward, fully-connected networks with one hidden layer. Each Q-net corresponds to one action. There are no connections between input and output layers. The inputs to the networks are the state information discussed previously. The single output of each Q-net represents the expected utility of the input state and the corresponding action. The number of hidden units of each Q-net is approximately 30% of the number of input units. It appeared similarly effective to use more hidden units (50%) or less (20%).

Each unit used a symmetric squashing function (see Equation (2.3)). The learning rate and momentum factor used by the back-propagation algorithm were fixed to 0.05 and 0.9, respectively. The network weights were randomly initialized to be between 0.5 and -0.5.

5.3.4 Exploration

The robot used the stochastic action selector described in Chapter 3.4 as a crude exploration strategy. The temperature T, which controls the randomness of action selection, was decreasing over time. It started from 0.05 and gradually cooled down to 0.02 after 300 trials. For better performance, another complementary strategy was also used together: The robot kept track of its X-Y position and orientation (relative to the start point) and temporarily increased the temperature whenever it found itself stuck in a small area without progress. This is still quite a naive exploration strategy. Using a more sophisticated strategy, the robot may acquire good skills faster than it did in the experiments reported below. But for the purposes of demonstrating the effects of teaching and TD(λ), this exploration strategy seems appropriate.

5.3.5 Non-Policy Actions

After each trial, the robot replayed 60 [1] (randomly selected) experienced lessons as well as all the taught lessons.

As discussed in Chapter 3.5.1, non-policy actions are not replayed. Here an action is considered a non-policy action if its probability of being chosen by the stochastic action selector is lower than $P_l = 2\%$. Also recall that λ is temporarily set to 0 when a non-policy action is encountered during replay.

5.4 Experimental Designs and Results

Four experiments are presented below. The purpose of the first experiment is to see how much performance improvement can be obtained from the use of teacher-provided training data. The second one experiments with various λ values. The purpose is to see whether there is an advantage to using TD(λ) over the simple TD(0). In the third experiment, two techniques for combining supervised learning and reinforcement learning are tested. Finally, in the fourth experiment, the Q-functions are represented by single-layer networks instead of multi-layer networks. The purpose is to see whether the Q-functions to be learned are nonlinear and whether temporal difference and back-propagation together can learn nonlinear Q-functions.

Each experiment consisted of 7 runs. Each run consisted of 300 trials. In each trial, the robot started with a random position and orientation near a door (in the case of DP), near the battery charger (in the case of DK), or anywhere (in the case of WFL and WFR). Each trial ended either when the goal was achieved, or else after 30 steps. (Since the wall following tasks do not have a final goal state, in this case each trial lasted for 30 steps.) To measure the robot's performance over time, after every 10 trials, the robot was tested against 50 randomly

[1] Fewer lessons were replayed in the beginning when there were not many lessons available.

generated test tasks. (The same set of test tasks was used throughout a run. Different runs used different sets.) Learning was turned off during tests. Again, each test lasted 30 steps at a maximum.

During learning, the robot selected actions semi-randomly using the exploration strategy mentioned in Section 5.3.4. During tests, the robot chose the best actions according to its current policy. The robot's performance was computed as the ratio between the total reinforcement received during solving the 50 test tasks and the total number of steps taken. In other words, the performance measure was the averaged reinforcement per step.

The robot's performance was then plotted versus the number of trials taken so far. The learning curves shown below describe the mean performance over 7 runs. Since WFR and WFL are similar tasks, only WFL was studied here. In some part of the experiments, the robot utilized taught lessons. To generate an instructive lesson, I placed the robot at a manually-chosen start position and did my best to maneuver the robot. (All the experiments used the same set of taught lessons.) The actions I showed to the robot were not all optimal, simply because it was difficult for me to tell what would be optimal from the robot's point of view. In particular, the optimal actions for wall following were rather hard to determine.

The parameter settings for the first three experiments are described in Section 5.3, while the parameter settings for the fourth experiment are slightly different as will be described later.

5.4.1 Experiment 1: Effects of Teaching

In this experiment, $\lambda = 0.7$ was used. But as described in Section 5.1.1, λ was selectively set to 0. Figure 5.4 shows the mean performance of the robot executing DP, DK, and WFL with or without teaching involved. One thing very clear from the figure is the usefulness of teaching. The following are observations from the experiments.

DP: Without a teacher, the robot often learned very little in the first 100 trials, because the robot was not lucky enough to achieve the door passing goal several times within 100 trials. Effective learning could take place only after a few successful experiences had been collected. In 1 out of 7 runs, the robot totally failed; it did not know how to gain rewards at all even if it was just in front of a door way. In contrast, with only 3 taught lessons, the robot learned significantly better than with no teaching involved. When provided with 10 positive examples, the robot acquired quite a good door passing skill after 100 trials.

DK: Once again, more teaching, the better performance. When no taught lessons were provided, the robot totally failed to learn the task in 2 out of 7 runs. [2] The high failure rate was due to the two contradictory goals of docking and obstacle avoidance. To get a tight connection to the battery charger, the robot has to position itself relative to the charger accurately and then drive to bump into it. But the robot might quickly learn to avoid obstacles in the very beginning,

[2] In my previous study [34], the robot never learned to perform docking without a teacher. Here with more randomness in action selection, it was able to succeed occasionally.

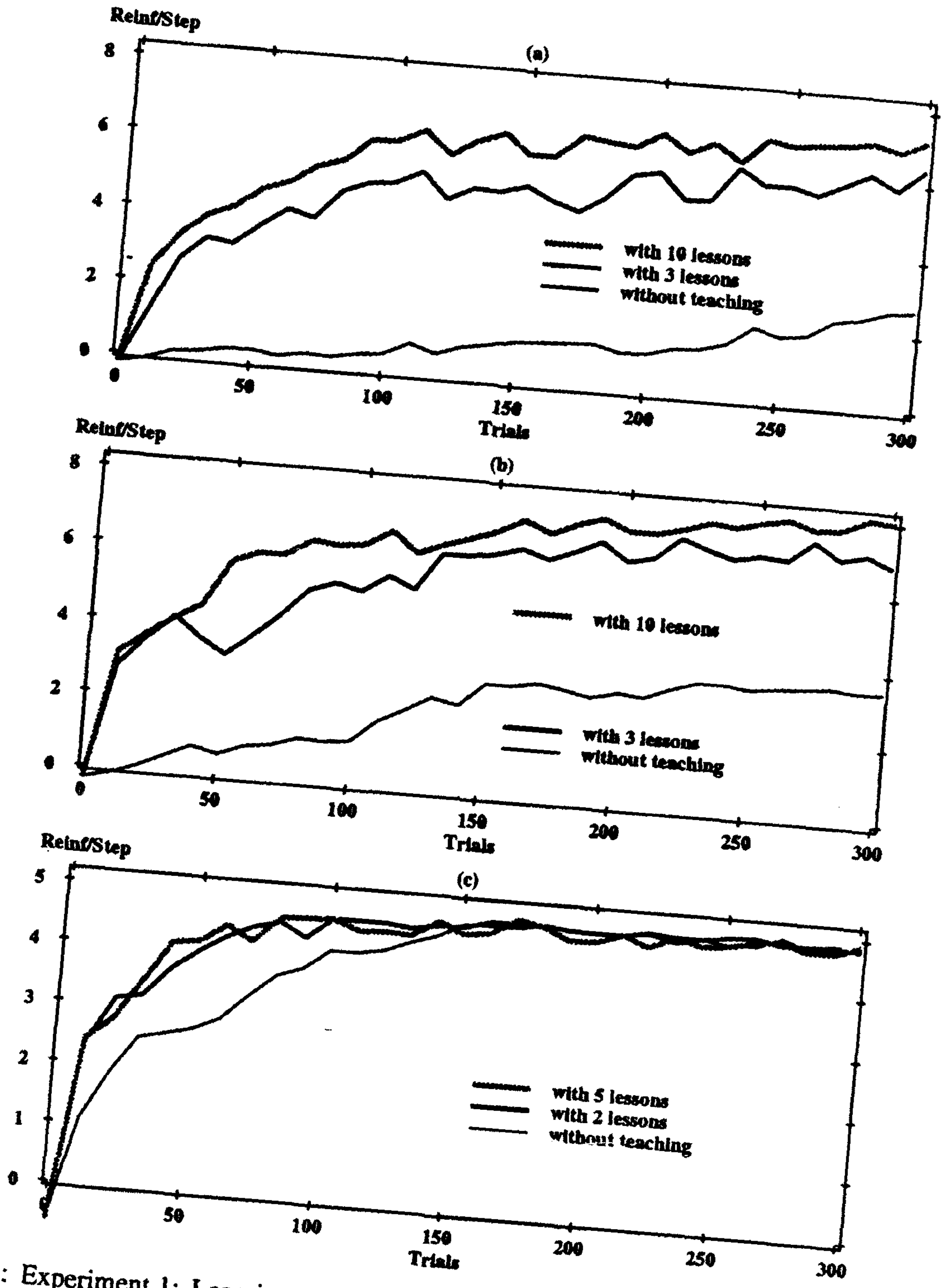


Figure 5.4: Experiment 1: Learning curves for (a) DP, (b) DK, and (c) WFL. The amount of teaching varied. λ was set to 0.7.

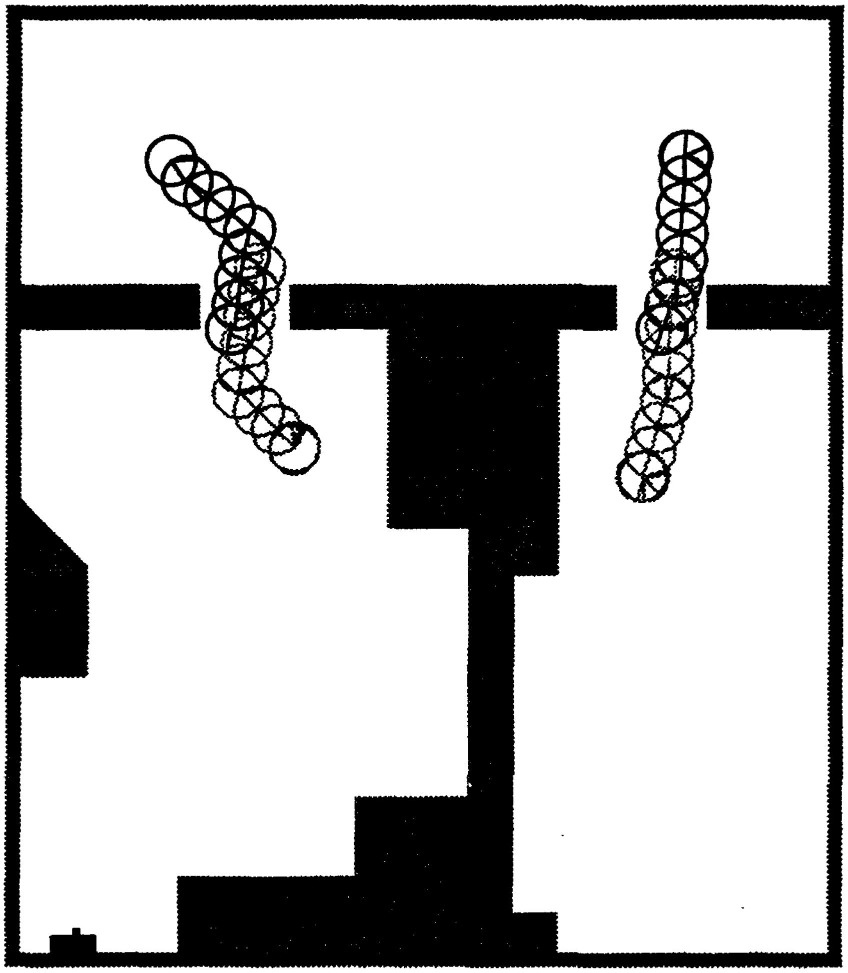


Figure 5.5: Four solution paths found for the door passing task.

and prevent itself from colliding with the charger later on. Once the robot experienced a few successful trials, effective learning suddenly took place and the final performance (after 300 trials) was often as good as that with teaching involved.

WFL: The wall following task is an easier task than DP and DK, because reinforcement is not far delayed. Once again, teaching had reduced the number of action executions needed to learn a good wall following skill. The impact of teaching, however, was not as significant as what we have seen for DP and DK. There was almost no difference in learning speed between the two cases with 2 and 5 taught lessons, possibly because the 2 lessons provided approximately the same amount of information as the 5 lessons.

Performance: Figures 5.5, 5.6, and 5.7 show some solution paths that the robot had learned after 300 learning trials with 10 (DK and DP) or 5 (WFL) taught lessons. Those solution paths are in fact nearly optimal.

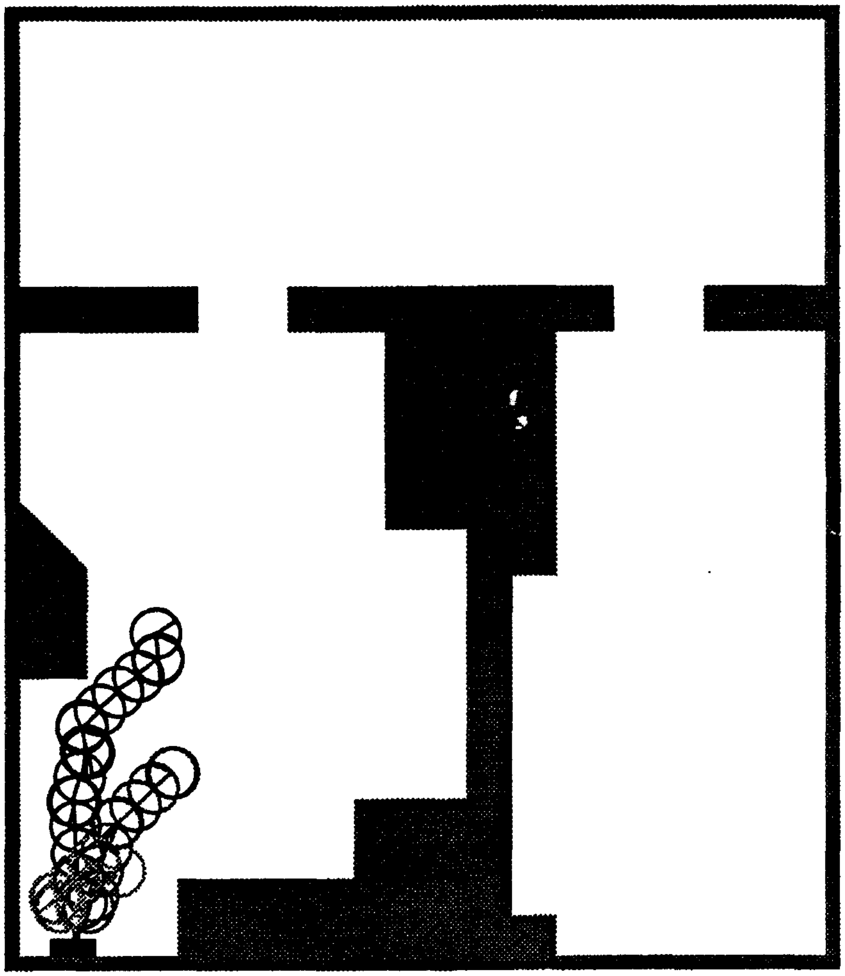


Figure 5.6: Three solution paths found for the docking task.

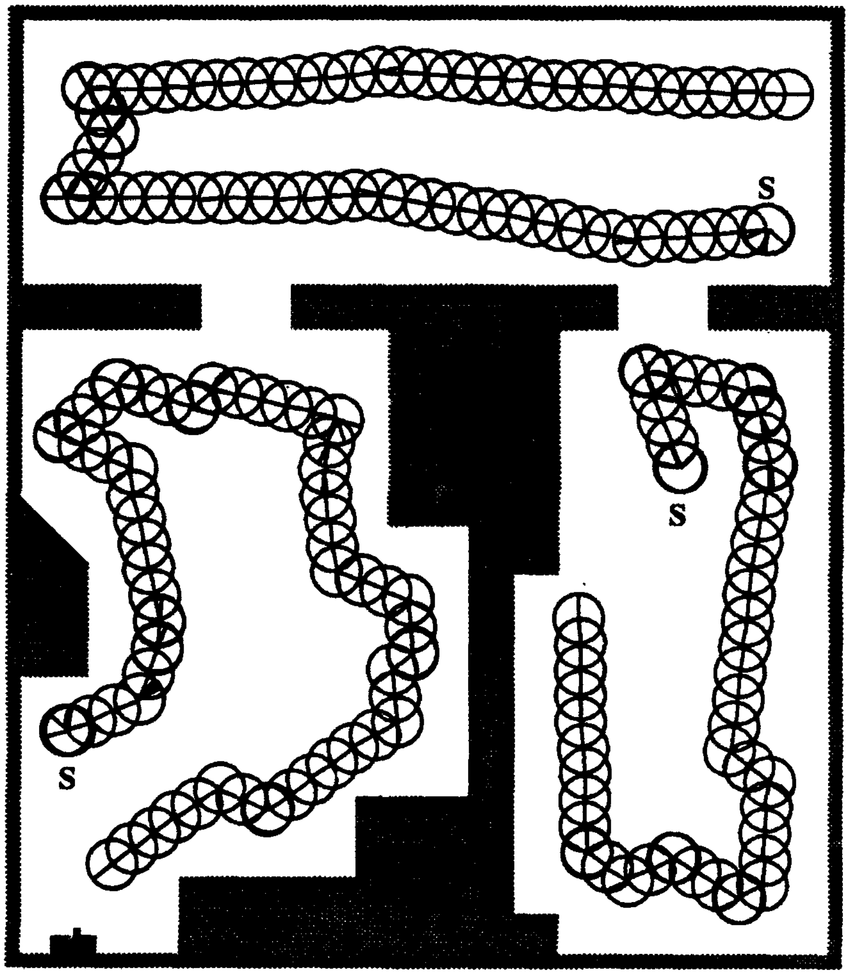


Figure 5.7: Three solution paths found for the left wall following (WFL) task. The start of each path is indicated by an "S".

Summary: Without a teacher, the robot's learning of DP and DK sometimes got stuck in poor local optima; in other words, the robot learned to avoid obstacles but was afraid to move close to the battery charger or the doors. In contrast, with the help of a teacher, the robot could easily avoid the local optima. If we let the robot keep high randomness in action selection and continue to explore for thousands of trials (maybe less if a clever exploration strategy is used), it is likely that the robot would reach the same level of competence regardless how much teaching is involved. But a robot programmer can save the robot lots of trial and error simply by giving the robot some demonstration.

5.4.2 Experiment 2: Effects of TD(λ)

In this experiment, the robot experimented with several different λ values. The number of taught lessons available to the robot was 10 in the case of DP and 0 in the case of WFL. (Only DP and WFL were studied.) Figure [illegible] shows the experimental data, which are summarized as follows:

- Among all the different λ values, $\lambda = 0$ clearly gave the worst performance.
- TD(0.9) and TD(1) were similarly effective, and more effective than TD(0.3).
- In the case of DP, TD(0.9) and TD(1) appeared slightly more effective than TD(0.7), while in the case of WFL, TD(0.7) seemed more effective than TD(0.9) and TD(1).

That TD(0) gave the worst performance can be explained by the fact that TD($\lambda > 0$) can propagate credits more effectively than TD(0). In his empirical study [66], Sutton found that TD methods worked the worst with $\lambda = 1$. His result seems inconsistent with the results obtained here. But this inconsistency may be explained by the following important difference between my experiments and his: In his case λ was kept constant throughout estimations, while in my case λ was selectively set to 0. The TD(1) used here switched between pure TD(1) and pure TD(0).

How can one determine the best λ value for a given task? Unfortunately, this experiment did not shed light on the answer. The best choice may depend on some characteristics of learning problems such as the length of reward delay and the nondeterminism of the domain.

5.4.3 Experiment 3: Combining Supervised Learning and Reinforcement Learning

The teaching technique studied in the previous experiments does not require the teacher to be an expert; the robot can learn from teachers who do not teach the optimal action. But this teaching technique may not be the best way of using valuable training instances provided by human experts. Suppose a teacher taught the robot to execute action a in response to state x. By using

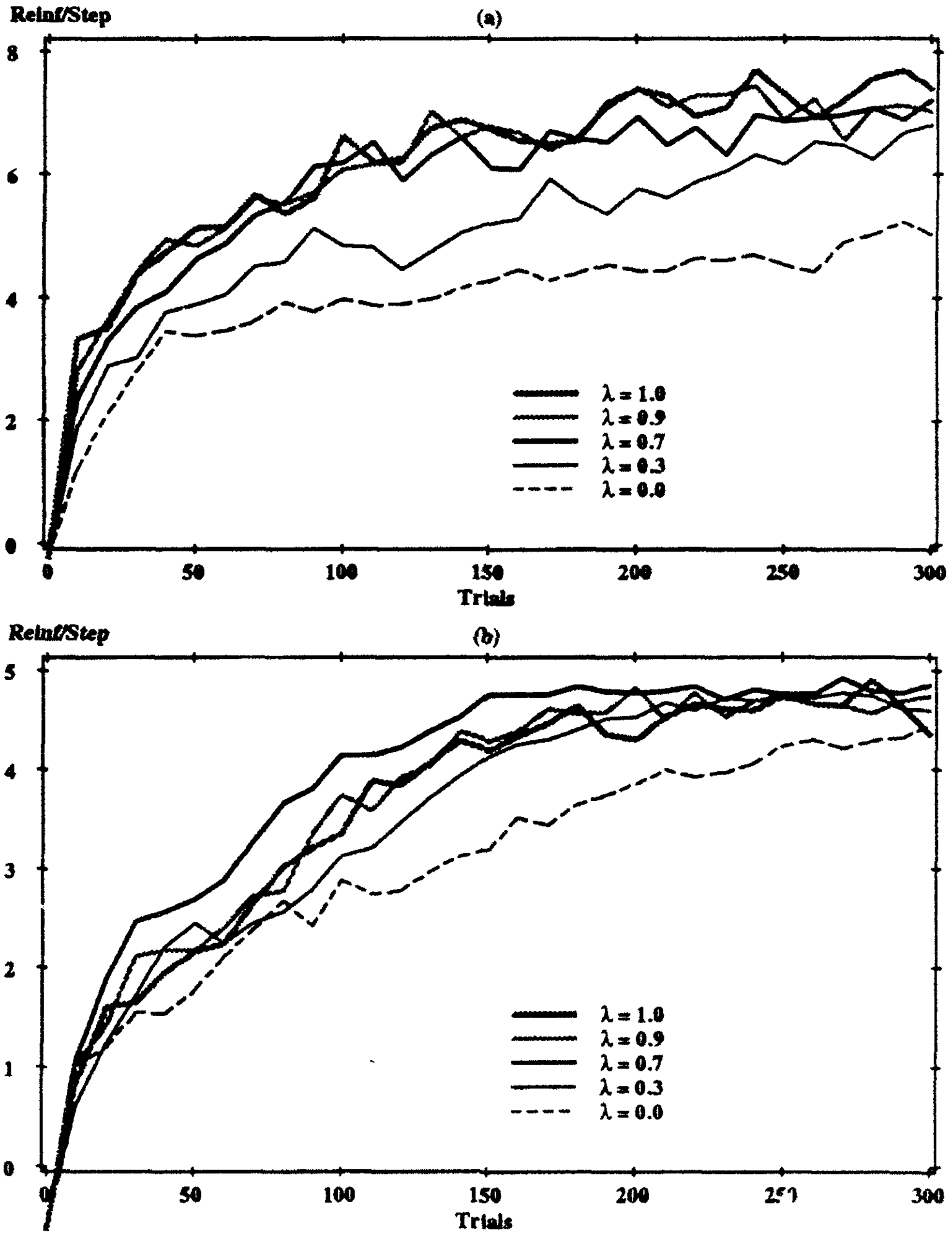


Figure 5.8: Experiment 2: Learning curves for (a) DP with 10 taught lessons and (b) WFL with no teaching involved. The purpose of this experiment was to compare the effects of various λ values.

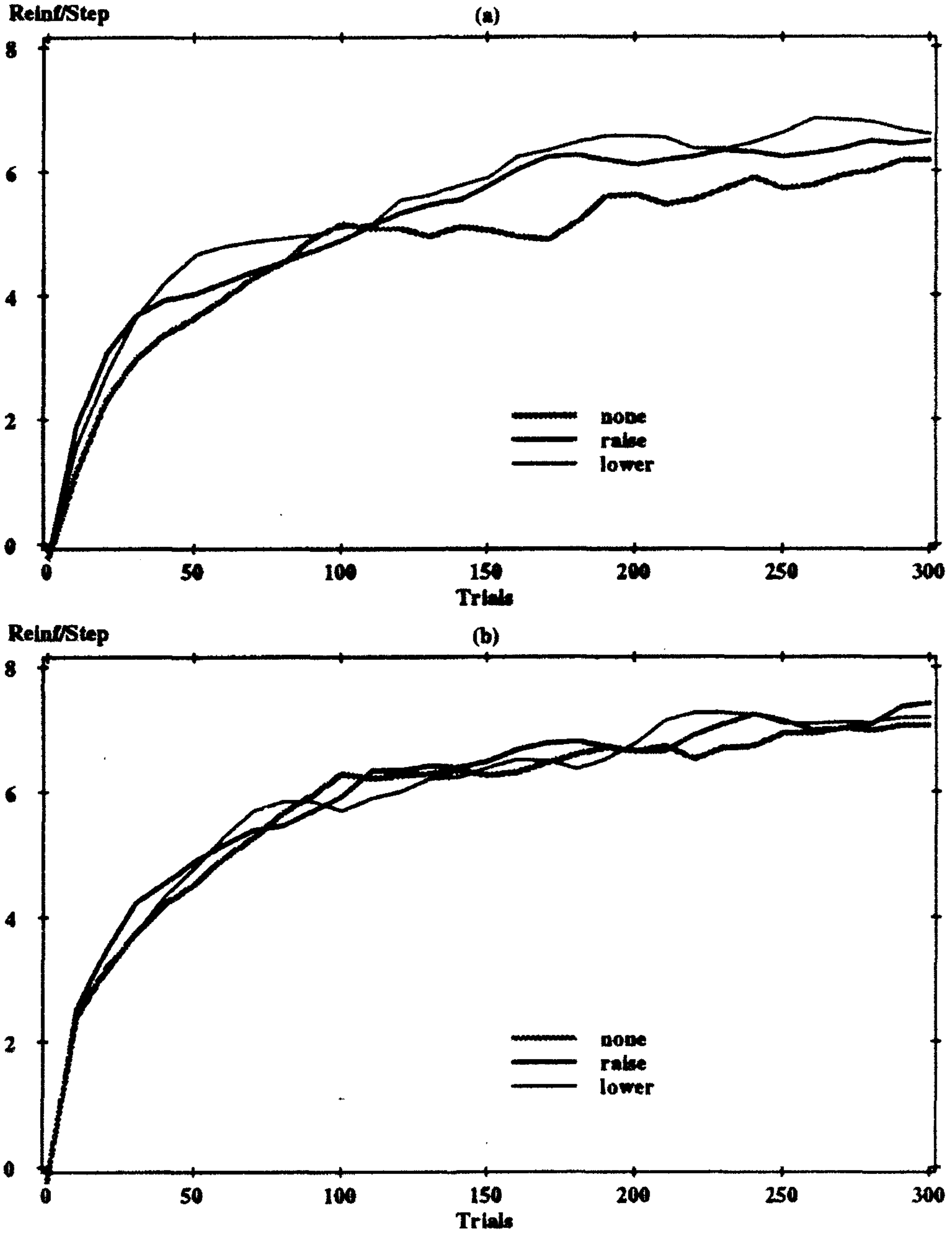


Figure 5.9: Experiment 3: Learning curves for (a) DP with 3 taught lessons and (b) DP with 10 taught lessons. The curves have been smoothed for ease of comprehension. λ was set to 0.7. The purpose of this experiment was to study two different strategies (*raise* and *lower*) for combining reinforcement learning and supervised learning. *none* means that neither of the strategies was used.

the teaching technique described before, what the robot learns from this piece of advice is only the utility of action a in response to state x. It makes no learning changes regarding the utilities of the other actions in response to the same state. Indeed, if the teacher is not trustworthy, this is about all that can be safely learned from this advice. But if the robot is told that action a is the optimal action in state x, how can the robot take advantage of the information that a is optimal? If a teacher always teaches the optimal action, supervised learning seems to be a more effective way of using training instances than reinforcement learning.

In fact, it is possible to combine supervised learning and reinforcement learning. Consider the following scenario: The robot has n actions, $a_1 \cdots a_n$, the taught action in state x is a_1, and a_1 is the optimal action in state x. An optimal Q-function should satisfy the following condition:

$$Q(x, a_1) \geq Q(x, a_i) \text{ for all } i \in [2..n]. \tag{5.6}$$

During learning, this condition may not hold true due to incorrect generalization. For example, the robot may examine its current Q-function and find that $Q(x, a_1) < Q(x, a_2)$. When this happens, two repairs are possible.

- *lower repair:* lower $Q(x, a_2)$ to make $Q(x, a_1) \geq Q(x, a_2)$, or

- *raise repair:* raise $Q(x, a_1)$ to make $Q(x, a_1) \geq Q(x, a_2)$.

The *lower* repair can be done by back-propagating the error $(Q(x, a_1) - Q(x, a_2))$ through the Q-net corresponding to action a_2. Similarly, the *raise* repair can be done by back-propagating the error $(Q(x, a_2) - Q(x, a_1))$ through the Q-net corresponding to action a_1.

In Experiment 3, the robot replayed taught lessons as usual. In addition, every time it replayed a taught lesson, it examined its Q-function to see if conditions like (5.6) had been violated. If a violation was detected, it made one of the repairs mentioned above. The experiment consisted of two parts, (a) and (b). For both parts, the door passing task was used as the test case, and λ was set to 0.7. In part (a) 3 taught lessons were available to the robot, and in part (b) 10 taught lessons. Figure 5.9 shows the learning curves, which have been smoothed for ease of comprehension. Smoothing is done by averaging every 3 successive original data points, except that the first two points are left untouched. In part (a) both repairs tended to improve the perfo.mance, while in part (b) the improvement, if any, was marginal.

Both techniques of utilizing taught lessons should be more effective than simple experience replay. But proper applications of the techniques are limited to situations where the teacher always demonstrates the optimal action. Indeed, the lessons I provided to the robot were not all optimal, and this may explain the experimental results of part (b), which showed no significant improvement gained by the use of the techniques.

5.4.4 Experiment 4: Single-layer Q-nets

In [11], Chapman and Kaelbling doubted whether the combination of temporal difference methods and connectionist back-propagation could learn *nonlinear* Q-functions. The purpose of this experiment is to test their doubt. The setup of this experiment was similar to that of Experiment 1 (Section 5.4.1) except that in this experiment:

- *single-layer* perceptrons were used to represent the Q-functions,
- binary inputs were represented by 1 and 0 (instead of 0.5 and −0.5),
- the learning rate was 0.04 for the DK and DP tasks, and 0.03 for the WFL task, and
- the momentum factor was 0.

The parameter settings were chosen to give roughly the best performance. Figure 5.10 shows the experimental results. For comparison, some of the learning curves from Experiment 1 are also included in the figure. Clearly, the robot could not accomplish the DP, DK and WFL tasks without using multi-layer networks to represent the Q-functions. Since single-layer perceptrons worked poorly for these tasks, the Q-functions are nonlinear. This result indicates that the combination of temporal difference and back-propagation really can learn nonlinear Q-functions.

5.5 Discussion

Based on a robot simulator, this chapter has demonstrated an approach to automatic robot programming. How likely is this approach to successfully apply to real robots in the real world? It seems that this approach should work for real robots, since reinforcement learning is inductive learning and does not rely on unrealistic assumptions such as prior domain knowledge. When applying reinforcement learning to real robots, we may face three problems:

- *Slow learning*. This problem can be handled by teaching, as I have already demonstrated. Another approach to reducing learning time is hierarchical learning, which is the topic of the next chapter.

- *Hazard*. Learning from trial-and-error could be hazardous in a hostile environment. A possible way to minimize damages to the robot during trial-and-error is again teaching. For example, we may teach the robot to avoid obstacles by carefully making up a collision situation (without hurting the robot), from which it would get the idea that moving forward is bad when the sonar readings from the front are small.

- *Noise*. Noise prevails in the real world. To be realistic, this robot simulator also simulates sensing and control errors. In this research, I found that the robot's performance in the noisy world was almost the same as that in a noise-free world. In other words, noise did not appear to be a problem.

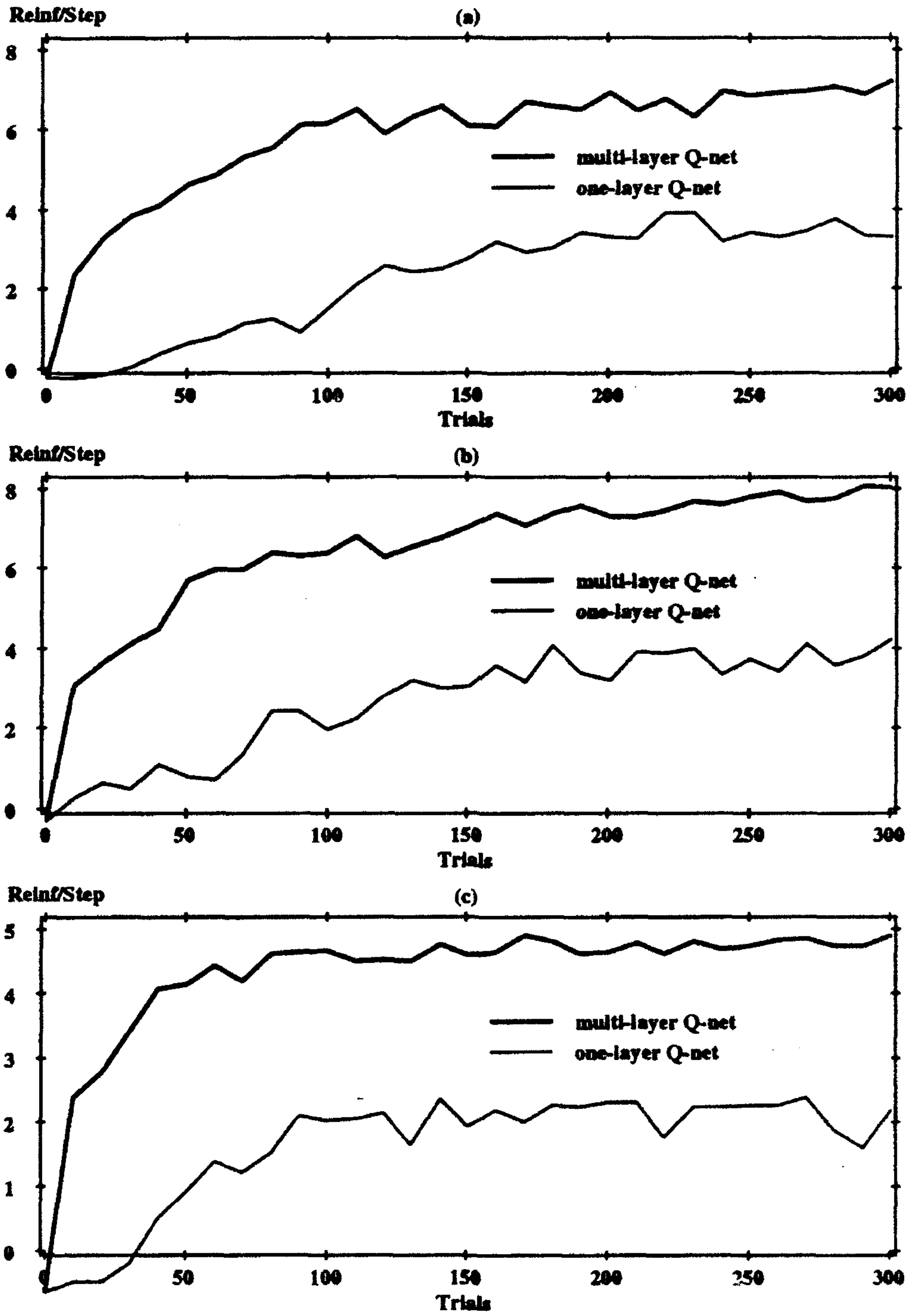


Figure 5.10: Experiment 4: Learning curves for (a) DP, (b) DK, and (c) WFL. λ was set to 0.7. The robot was provided with 10 taught lessons for learning each of the DP and DK tasks, and 5 taught lessons for learning the WFL task. The purpose of this experiment was to see whether single-layer networks are adequate to represent the Q-functions or not.

As a matter of fact, some preliminary experience with a real HERO robot has been presented in [34]. This learning approach was not directly tested on the real robot. Instead, the skills learned by a simulated robot were tested in the real world. (After many modifications, the simulator used in this dissertation is different from the simulator used in [34].) In other words, the real robot used the Q-nets learned in simulation to operate in the real world. Because that simulator mimicked the real robot and the real environment closely, the real robot indeed operated quite successfully in the real world. In simulation the robot achieved goals 96% of the time, while in the real world 84% of the time. This gives indirect evidence of the approach being practical for real world problems. I hope to apply the learning technique directly to real robots in the future.

Section 5.4.3 describes two possible ways of combining supervised learning and reinforcement learning. One is to lower the utility of a non-taught action if it has a higher utility than the optimal one. The other is to raise the utility of the taught action if it does not have a higher utility than all the others. The *lower* technique may be more appropriate than the *raise* technique. Because the robot has experience with the taught action and may not have experience with the other actions, it is likely that the robot would have a more accurate estimate in the utility of the taught action than in the utilities of non-taught actions. Therefore, it seems more appropriate to modify non-taught actions' utilities than to modify the taught action's. The utility of both techniques, however, was found limited.

Whitehead [79] describes a teaching technique called *learning with an external critic* (LEC). His idea is the following: While doing reinforcement learning, the learner is observed by a critic, who occasionally provides immediate feedback on the learner's performance. The feedback, for example, may be 1 if the executed action is optimal and -1 otherwise. The learner learns a Q-function as usual. In addition, it learns a *biasing function*, which represents the average feedback from the critic for each state-action pair. The control policy is to choose the action that has the greatest combination of Q-value and bias-value. It would be interesting to compare his approach and mine.

5.6 Summary

This chapter presented two experience replay algorithms for reinforcement learning; one uses Q-learning and the other AHC-learning. Both algorithms use backward replay and are efficient implementations of TD(λ) methods. The one using Q-learning has been tested here to learn nonlinear Q-functions, and found quite effective.

Two main simulation results were obtained:

- Teaching significantly improved the learning speed.
- TD(0.7) was significantly better than TD(0) regarding learning speed.

To have a rough idea about how much speedup can be obtained from teaching and from using non-zero λ values, here I take the door passing task for example and show the obtained speedup in the following two tables. In the first table, the second column shows the number of learning trials needed by the robot (using $\lambda = 0.7$) to reach the performance of 2.5 (i.e., 2.5 units of reward per step), and the third column shows the speedup (compared with the robot without teaching). In the second table, the second column shows the number of learning trials needed by the robot (using 10 taught lessons) to reach the performance of 5, and the third column shows the speedup (compared with TD(0)). A speedup of x means that the number of trials needed is reduced to $1/x$ of the originally needed number.

	# of trials	speedup
no teaching	280	
3 taught lessons	20	14
10 taught lessons	10	28

	# of trials	speedup
$\lambda = 0.0$	280	
$\lambda = 0.3$	140	2.0
$\lambda = 0.7$	65	4.3

Chapter 6

Hierarchical Learning

The previous chapters have discussed several techniques for improving the convergence speed of reinforcement learning, including using neural networks, teaching, using nonzero λ values, etc. These techniques alone, however, may not be sufficient for dealing with situations where the optimal control policy is very complex to represent and very difficult to learn.

This chapter presents *hierarchical learning* as another way to scale up reinforcement learning and enable its applications to very hard learning problems. Hierarchical learning is a divide-and-conquer technique— a complex learning problem is decomposed into small pieces so that they can be easily solved. This chapter presents a model of hierarchical reinforcement learning. A close examination into this model results in possible answers to questions such as (1) how hierarchical learning can be done and save learning time and (2) what domain knowledge may be needed to do it.

The robot simulator described in Chapter 5 is used here again to demonstrate the importance of hierarchical learning. The learning task studied in this chapter is so hard that the robot is unable to solve it as a monolithic learning problem within a reasonable time. In contrast, by hierarchical learning, the robot can effectively and efficiently solve it.

This chapter also addresses issues such as behavior-based exploration and efficient credit assignment in face of long reward delay.

6.1 Introduction

It is well known that no planning system can scale up well without hierarchical planning. Similarly, no learning agent can successfully cope with the complexity of the real world without hierarchical learning of skills. By hierarchical learning, an agent learns elementary skills for solving elementary problems first. After that, to learn a new skill for solving a complex problem, the agent only need learn to coordinate the elementary skills it has developed without worrying about low-level details. While new skills are being developed, the elementary skills may be polished further.

In this chapter, I will demonstrate that hierarchical learning of skills is possible within the reinforcement learning paradigm. In this chapter, four terms are used interchangeably: *skill*, *control policy*, *behavior*, and *abstract action*. This is because a control policy for a task is a skill for carrying out the task, and an agent exhibits a sort of behavior when following a control policy. A skill is also an abstract action, with which the agent can achieve a subgoal without considering the low-level details.

The primary virtue of hierarchical learning (and planning) is the great reduction in the complexity of problem solving. The essential idea is based on *problem decomposition* and the use of *abstraction*. To efficiently solve a complex learning problem, humans often decompose the problem into simpler subproblems. Once we have learned skills to solve each subproblem, solving the original problem becomes easier, because we now can concentrate on the high-level features of the problem and ignore the low-level details, which presumably will be filled in when the previously learned skills are actually applied. This idea can generalize to multiple levels of abstraction. At each level, we focus on a different level of details. Another virtue of hierarchical learning is that subproblems shared among high-level problems need be solved just once and their solutions can be reused.

Three main steps are involved in hierarchical learning:

1. design a hierarchy of skills,
2. learn elementary skills that select primitive actions to carry out elementary tasks, and
3. learn high-level skills that coordinate elementary skills to solve complex problems.

Among the three steps above, hierarchy design (Step 1) seems to be the most difficult part for machines to automate, because a good hierarchy design demands either lots of domain knowledge or lots of experience of solving related problems. In comparison, humans seem to come up a good hierarchy relatively quickly after being given a robot task. This work thus assumes that a correct hierarchy is given by human designers.

The problem of learning elementary skills (Step 2) has been the focus of the previous chapters. This chapter focuses on issues related to learning high-level skills (Step 3). It seems that learning high-level skills is similar to learning elementary skills, since elementary skills may be treated like primitive actions during learning high-level skills. While this is the essential idea of hierarchical learning, there is an important difference between primitive actions and elementary skills. Primitive actions are normally assumed to complete in a fixed period of time, while elementary skills may last forever. For example, a wall following skill, once activated, may never end.

The rest of this chapter is organized as follows: Section 6.2 presents a model of hierarchical reinforcement learning and a quantitative analysis of the learning efficiency gained by task decomposition and abstraction. Section 6.3 summarizes the model and addresses some practical issues involved. Section 6.4 describes the specific learning task investigated in this chapter. Section 6.5 discusses implications of never-ending behaviors and presents approaches to coordinating such behaviors. Section 6.6 presents simulation experiments and results. Based on the results, Section 6.7 makes comparisons between hierarchical learning and non-hierarchical learning. Finally, Section 6.8 discusses related work, and Section 6.9 summarizes the main results.

6.2 A Model and Complexity Analysis

This section presents a model of hierarchical reinforcement learning and a complexity analysis for deterministic domains. The analysis is based on previous results obtained by Koenig [31], who showed that the complexity of learning an optimal control policy in discrete and deterministic domains is $O(n^2)$, where n is the number of states (see below for details). The analysis here yields two main results. First, under some assumptions, learning problem solving skills hierarchically can reduce learning time from $O(n^2)$ to $O(n \cdot \log n)$. Second, once a learning problem has been solved, solving a second problem becomes easier, if both problems share the same subproblems.

Note that all the analyses mentioned in this section, including the one by Koenig, make the following *simplifying assumptions:*

- The environment is Markovian, deterministic, and discrete.
- The learning task is to find solution paths from every state to a single goal state.

6.2.1 Complexity of Non-Hierarchical Reinforcement Learning

Koenig [31] developed algorithms for real-time search. His algorithms can be modified for an agent to build a complete model of an unknown deterministic environment. (For any given state-action pair, the model predicts the next state and the immediate reinforcement.) The complexity of learning a complete model (i.e., the number of action executions required to build a model) can be shown to be $O(b \cdot n)$, where n is the number of states and b is the total number of actions available in all the states. If the number of actions in every state is bounded by a constant c, the complexity of learning a model is $O(c \cdot n^2)$. In d-dimensional grid worlds, for example, the number of actions in each state is dependent on d and independent of n. The complexity of learning a grid world is thus $O(n^2)$.

Once the agent has learned a model, it can construct the optimal control policy off-line by using the model to generate hypothetical experiences, much as Framework QCON-M or AHCON-M does (see Chapter 3.6). Consider using tabular Q-learning to build a control policy. For each experience generated by the model, the agent can apply the learning rule specified by Equation (3.5) to update the tabular Q-function. Since the domain is deterministic, a learning rate of 1 can be used without loss of optimality. After every state-action pair generated by the model has been tried, the agent has found the optimal Q-values for all the states that are 1-step away from the goal state. If the agent applies the learning rule to every state-action pair once again, it will find the optimal Q-values for all the states that are 2-step away from the goal state. Let p be the depth of the state space. [1] The agent can find the optimal Q-function by applying the learning rule to every state-action pair p times. The complexity of constructing an optimal control policy with a model is thus $O(b \cdot p)$. Since $p \leq n$, $O(b \cdot p) \leq O(b \cdot n)$. For grid worlds, the complexity is bounded by $O(n^2)$.

In the analysis to be presented below, I assume that the complexity of reinforcement learning for deterministic tasks is $O(n^2)$, where n is the number of states of the environment.

6.2.2 A Model of Hierarchical Reinforcement Learning

Definitions

Let us call the original state space the ***base space***, which consists of a set of ***base states***. Corresponding to the base space is a set of ***base actions***, which enables the agent to move from one base state to another. An ***abstract space*** is a subset of the base states called ***abstract states***. An ***abstract action***, which is composed of a sequence of base actions, enables the agent to move from a base state to an abstract state.

Abstract states and abstract actions are defined as follows: We divide the base space into k *regions* of approximately equal size and shape. We then choose the center state of each region as an abstract state. [2] Hence the abstract space consists of k abstract states. Abstract states serve as subgoals, and thus are also called ***subgoal states***. Each subgoal state has its corresponding abstract action, and vice versa. (Hence there are k abstract actions.) Each abstract action is a close-loop control policy (such as a Q-function), which finds solution paths from base states to its corresponding subgoal state. Each abstract action, however, is not required to find paths for every base state in the base space. Instead, it only need find paths for a subset of the base space called the ***application space*** of the abstract action. The application space is the set of base states from which the subgoal state is reachable within l steps, where l is the maximum distance between any two neighboring abstract states. This definition implies that abstract actions allow the agent to navigate among abstract states. This model of hierarchical learning is illustrated in Figure 6.1, in which a 2-D grid world is considered. In this example, $k = 16$ and $l = 3$.

To summarize, for a given base space, a number of base states are chosen as abstract state. For each abstract state, there is an abstract action. When executed, the abstract action will bring the agent to the corresponding abstract state, as long as the agent starts in the application

[1] The depth of a state space is the maximum distance from any state to the goal state.

[2] In grid worlds, the center state of a region can be chosen to be the geometrical center of the region. In general, a center state is chosen such that the distances (in terms of primitive actions) from the perimeter of the region to the center state are roughly equal.

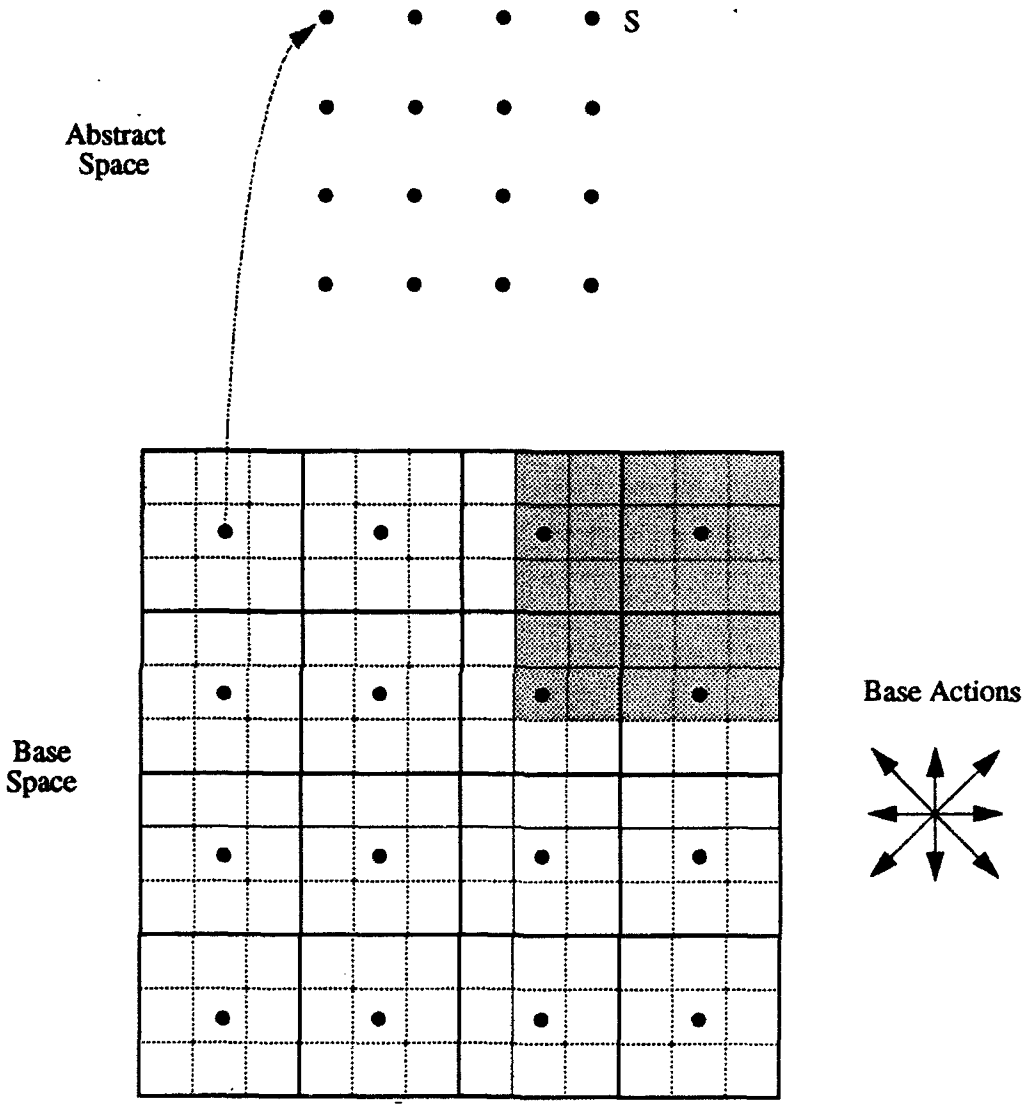


Figure 6.1: A hierarchy of abstraction for a 2-D grid world. Tiie grid world is divided into 16 regions. The center state of each region is selected as a subgoal (or abstract) state. Hence there are 16 subgoal states. There are also 16 abstract actions; each corresponds to a subgoal state. Each abstract action is a close-loop control policy, which defines paths from every state in its application space to its corresponding subgoal state. For example, the application space of the abstract action corresponding to the subgoal state S would be those states in the shaded area. If the current state is not in the application space, the agent may never reach state S by executing this abstract action.

space. The application space can be thought of as the *precondition* of the abstract action. An abstract action is *applicable*, if its precondition is met; in other words, if its application space covers the agent's current state. The application space is centered at the abstract state, and is large enough to cover the neighboring abstract states.

Learning Abstract Actions

Each abstract action can be learned by Q-learning. Without any restrictions, learning to reach a subgoal state has the same complexity as learning to reach a top-level goal state, and thus it is no use having hierarchical learning. For hierarchical learning to be useful, the following assumption is introduced:

> **Reset Assumption:** For each subgoal state, there is a corresponding *reset operator*. When used, the reset operator will randomly place the agent in a state which is at most l steps away from the subgoal state.

In other words, this reset operator will randomly place the agent in the application space. With such a reset operator, each abstract action can be effectively learned by repeating the following two steps until an optimal abstract action is learned:

1. apply the reset operator;
2. explore the environment until the abstract state is reached or else after l steps.

The basic idea is that during learning an abstract action, the agent uses the reset operator to confine its exploration to the vicinity of the abstract state, avoiding exhaustive exploration of the whole base space. Let us call the states to be explored in the above process the *exploration space* of the abstract action. Note that the exploration space covers the whole application space. According to the above exploration strategy, the radius of the application space is l, while the radius of the exploration space is $2l$. Let m be the number of base states in the exploration space. The complexity of learning an abstract action is thus $O(m^2)$.

Choosing Abstract Actions

An important question here is how the agent determines whether a given abstract action is applicable at the moment. In the learning phase, if the agent executes an abstract action whose application space does not cover the agent's current state, the agent may never reach the abstract state it intended to reach and thus get stuck forever. Depending on the reward function used, the agent may find the answer (i.e., whether a given abstract action is applicable) from the Q-function learned for the abstract action. If the reward function is 1 for reaching the goal state and 0 otherwise, from the Q-values the agent can figure out the distance from its current state to the abstract state. If the distance is not greater than l, the abstract action is applicable.

Otherwise, it is not. For general reward functions, the agent may not be able to figure out the answer from the Q-function. In such a case, the agent can learn another function (called *distance function*) to represent the distance from a state to the abstract state. This function can be learned simultaneously with the Q-function. Thus the complexity of learning an abstract action remains the same (i.e., $O(m^2)$).

Another question is how the agent finds all the applicable abstract actions and what is the complexity of finding them. In a straightforward way, the agent evaluates all the Q-functions (or distance functions) for the abstract actions and finds the applicable ones. All the functions can be evaluated in parallel and in a constant time, if the agent has sufficient hardware. Alternatively, the agent can record the applicable abstract actions for each state, and retrieve them in a constant time. Note that the number of applicable abstract actions in each state is a (small) constant. The number is generally dependent on the structure of the base space and independent of the size of the space. For example, in grid worlds, the number of applicable abstract actions is equal to the number of primitive actions. In short, the complexity of finding all the applicable abstract actions can be bounded by a constant.

Consider the problem of finding a solution path from a given start state s to a goal state g. Let us assume that g is one of the abstract states. If g is not, we simply add it to the abstract space. To find a path from s to g, the agent first finds all the applicable abstract actions and then determines which one can reach an abstract state at the least cost (namely, with the highest Q-value). It executes this abstract action and ends up in an abstract state i. Then, the agent finds a path from i to g in the abstract space. Note that solution paths found in this way may not be optimal, since they must go via subgoal states. When using abstraction, we generally have to trade off optimality for efficiency.

Total Complexity

The question now is how to find solution paths in the abstract space, and what is the number of action executions required. [3] Imagine that the agent is in an abstract state and is about to choose an experimental abstract action. It first finds all of the applicable abstract actions. Then it chooses one of them according to its exploration rule, executes it, and ends up in another abstract state. Essentially, the agent can navigate among abstract states by executing applicable abstract actions. Therefore, the number of abstract-action executions required to learn a Q-function in the abstract space is $O(k^2)$. It amounts to $O(l \cdot k^2)$ action executions,

[3] Depending on the reward functions used, it is possible that no more action executions are needed to find solution paths in the abstract space, once the agent has learned the abstract actions. Suppose the only concern of the agent is to find shortest paths between states. After learning the abstract actions, the agent already knows the distance between any two neighboring abstract states. Thus, the agent can simply use Dijkstra's shortest path algorithm to find the shortest paths between states in the abstract space. But in general, the agent may be concerned not only with shortest paths but also with avoidance of certain malicious states, which are unknown during learning the abstract actions. In this case, the agent must actually get involved in the world in order to find out which path is short and meanwhile away from malicious states. In the analysis here, I assume the general case.

because the distance between any two neighboring abstract states is bounded by l.

To summarize, the total number of action executions required to learn a hierarchy of control policies is $O(k \cdot m^2 + l \cdot k^2)$. The former is for learning k abstract actions. Without hierarchical learning, the learning complexity is $O(n^2)$.

l and m are dependent on the choice of k; the larger k, the smaller l and m. l and m are also dependent on the structure of the base space. For d-dimensional grid worlds, $l = (n/k)^{1/d}$ and $m = 4^d \cdot n/k$, assuming optimal dividing of the base space is possible. For 2-dimensional grid worlds, the learning complexity amounts to $O(4^4 \cdot n^2 \cdot k^{-1} + n^{0.5} \cdot k^{1.5})$. When $k = 7.8 \cdot n^{3/5}$, this complexity is minimum and equal to $(55 \cdot n^{1.4})$.

Suppose the agent has already learned a task. What is the expected number of action executions required to learn a new task? If the new goal state is already in the abstract space, then $O(l \cdot k^2)$ action executions will be required, since the agent only needs to find solution paths for a new goal in the abstract space. If not, the agent adds the new goal state to the abstract space, learns a new abstract action for this new abstract state, and finds solution paths in the abstract space. Doing all of these requires $O(m^2 + l \cdot (k+1)^2)$ action executions. For 2-dimensional grid worlds and $k = 7.8 \cdot n^{3/5}$, this is approximately equal to $(4 \cdot n^{0.8} + 22 \cdot n^{1.4})$ action executions.

Following the above analysis, we can show that for d-dimensional grid worlds where $d \geq 1$, the complexity of learning the first task is bounded by $O(n^{1.5})$, and the complexity of learning an additional task is also bounded by $O(n^{1.5})$, assuming an optimal value of k is chosen.

6.2.3 Multiple Levels of Abstraction

We now consider reinforcement learning with multiple levels of abstraction. Again, let n be the number of states in the base space. At each level of the hierarchy, states are divided into regions, whose center states form an abstract space for the next higher level. Abstract actions are defined and trained in the way described in Section 6.2.2, except that abstract states and abstract actions at one level will be treated as base states and base actions at the next higher level. To find a path from a base state s to a goal state g, the agent first finds the abstract action connecting s to the nearest abstract state s_1 in the abstraction space of the first level, then finds the abstract action connecting s_1 to the nearest abstract state in the abstract space of the second level, and so on, until the goal state is reached. Again, we assume that the goal state is included in the abstract spaces of all levels.

Let k_i be the number of states in the state space of Level i. (The base space corresponds to Level 0.) Let l_i be the distance (in terms of base actions) between two neighboring states at Level i. (Hence $l_0 = 1$.) Also let m_i be the number of states (in terms of states at Level $i-1$) in the exploration space during learning an abstract action at Level i. The number of action executions required to learn all the abstract actions at all levels is thus:

$$\sum_{i=1}^{h} k_i \cdot m_i^2 \cdot l_{i-1} \tag{6.1}$$

where h is the number of levels of the hierarchy and $k_h = 1$.

For the purpose of analysis, we make two more assumptions: First, the relative distribution of abstract states over the base space is the same for all levels. Second, the ratio between k_{i+1} and k_i is constant for all i. Also let k, l, and m be k_1, l_1, and m_1, respectively. Under the two assumptions, $k_i = n \cdot (k/n)^i$, $l_i = l^i$, and $m_i = m$ for all i. We thus can simplify (6.1) to:

$$k \cdot m^2 \cdot \sum_{i=1}^{h} p^{i-1} = k \cdot m^2 \cdot \frac{1-p^h}{1-p} \tag{6.2}$$

where $p = kl/n$. Because $k_h = n \cdot (k/n)^h = 1$, the abstraction hierarchy consists of $h = \log_{n/k} n$ levels. For 2-dimensional grid worlds, $l = (n/k)^{0.5}$, $m = 16 \cdot n/k$, and (6.2) further simplifies to:

$$4^4 \cdot \frac{n^2}{k} \cdot \frac{\sqrt{n}-1}{\sqrt{n}-\sqrt{k}} \tag{6.3}$$

When $k = \frac{4}{9}n$, (6.3) is minimum and equal to $1728n$.

What is the complexity of learning a new task? If the new goal state is already in the abstract spaces up to the jth level of the hierarchy, the expected number of action executions required is:

$$\sum_{i=j+1}^{h} (1+m_i)^2 \cdot l_{i-1} = (1+m)^2 \cdot \sum_{i=j+1}^{h} l^{i-1} \tag{6.4}$$

For 2-D grid worlds, (6.4) simplifies to:

$$(1 + 16 \cdot \frac{n}{k})^2 \cdot \sum_{i=j+1}^{h} (\frac{n}{k})^{(i-1)/2} \tag{6.5}$$

When $k = \frac{4}{9}n$ and when $j = 0$ (the worst case), (6.5) is equal to $2738\sqrt{n}$.

It can be shown that for 1-dimensional grid worlds, the complexity of learning the first task is $O(n \cdot \log n)$, and the complexity of learning an additional task is $O(n)$. For d-dimensional grid worlds where $d \geq 2$, the complexity of learning the first task is $O(n)$, and the complexity of learning an additional task is $O(n^{1/d})$. These complexities are obtained when (n/k) is a small constant.

6.2.4 Summary of the Analysis

Table 6.1 compares the complexity of reinforcement learning in grid worlds of arbitrary dimensions with and without abstraction. It is clear that hierarchical learning can reduce the

Table 6.1: Complexity of reinforcement learning in d-dimensional grid worlds where $d \geq 1$. n is the number of states.

	no abstraction	single-level abstraction	multi-ievel abstraction
learning the first task	$O(n^2)$	$O(n^{1.5})$	$O(n \cdot \log n)$ for $d = 1$ $O(n)$ for $d > 1$
learning an additional task	$O(n^2)$	$O(n^{1.5})$	$O(n^{1/d})$

Table 6.2: Complexity of planning provided that the planner has an action model. n is the number of states.

	no abstraction	single-level abstraction	multi-level abstraction
complexity	$O(n)$	$O(\sqrt{n})$	$O(\log n)$

number of action executions required to learn a task. It is important to emphasize that this reduction is not possible without the use of reset operators, because without using the operators to confine the space to be explored, learning to reach a subgoal state is just as difficult as learning to reach a high-level goal state. (Section 6.3.2 gives further discussions about reset operators.)

For 2-D grid worlds, the optimal hierarchy is obtained when the number of levels is $\log_{9/4} n$, which is about 11 for a 100 by 100 grid world. In general, such a deep hierarchy suggested by this analysis is not achievable and not practical. For example, solution paths found with a deep hierarchy may be too suboptimal to be acceptable, because many constraints are imposed on the solution paths regarding what intermediate states the paths should go via. Nevertheless, the analysis gives a sense about how much efficiency can be gained by the use of hierarchical learning.

Korf [32] presented a quantitative analysis of abstraction in planning systems. My analysis of hierarchical learning was in fact inspired by his work. His main results can be summarized in Table 6.2. With multiple levels of abstraction, the planning time can be dramatically reduced from $O(n)$ to $O(\log n)$, where n is the number of states. Such a dramatic reduction in complexity is not possible in reinforcement learning, because a planning system can change the system state from one to another in a single search step, while a reinforcement learning agent, which learns directly from the interactions with its environment, cannot jump from one state to another without actually going through all the intermediate states. Note that Korf assumed that an action model is available to a planning system, while here a reinforcement learning agent does not have a model to start with— this is why the planning complexity without abstraction is $O(n)$, while the learning complexity without abstraction is $O(n^2)$.

6.3 Hierarchical Learning in Practice

6.3.1 Summary of the Technique

According to the proposed model of hierarchical reinforcement learning, the following 5 steps are involved in doing hierarchical learning. (The rest of the chapter focuses on hierarchies with single-level abstraction.)

1. Select abstract states. A good heuristic might be to select as abstract states the states with special landmarks.

2. Define abstract actions— To define an abstract action, we need to specify two things:

   - a reinforcement function specifying the goal of the abstract action, and
   - an application space specifying the portion of the base space for which the abstract action will be trained.

   The main design criterion is that there must be abstract actions to take the agent from any base state to an abstract state and there must be abstract actions to allow the agent to navigate among abstract states.

3. Train each abstract action in its exploration space, which covers its application space.

4. Have a procedure to find applicable abstract actions. In some cases, this can be done by examining the Q-functions for the abstract actions.

5. Train a high-level policy to determine the optimal abstract action.

The 5 steps will become clearer as the simulaiion experiments are presented.

6.3.2 Reset Operators

The complexity analysis presented above assumes the availability of reset operators. (Note that the reset operators are needed during learning, but not needed afterwards.) How can we actually implement the reset operators, if learning takes place in the real-world rather than in simulation?

Humans spend several years in school learning to calculate. Step by step, we are taught to count, then to do addition and subtraction, and then to multiply and divide numbers. Teachers will not ask students to do multiplication before the students have learned addition. This is hierarchical learning, and the reset operators are in fact provided by the teachers, who carefully generate appropriate training tasks for the students.

Like students, robots may rely on human teachers to implement reset operators. Suppose we want to train a robot to dock on a battery charger whenever the charger is nearby and detected. One possibility is to have a human teacher physically carry the robot and place it at a random location close to the charger after each learning trial— this relocation has to be done each time the robot has either successfully docked or moved too far away from the charger. Alternatively, relocation may be performed by a software procedure that issues a random number of random action commands to the robot. A protection mechanism may be needed to prevent the robot from damages during random action executions. This alternative is normally more desirable than physically carrying the robot, since much less human labor is involved. However, generally speaking, this alternative works only when primitive actions are *invertible* (i.e., each action can be undone by another action).

6.4 The Learning Domain

The case study to be presented in this chapter is based on HEROINE, the mobile robot simulator described in Chapter 5. The specific learning task for the robot is battery recharging, which involves finding a battery charger and electrically connecting to it. The reinforcement function for this task is:

- 100 if the robot gets connected to the charger, and
- 0 otherwise.

6.4.1 Task Decomposition

According to Section 6.3.1, the first step to do hierarchical learning is to select a set of abstract states (i.e., subgoal states) for the robot. Figure 6.2 shows the 6 subgoal states I have picked for this study. Note that Subgoal 1 is in fact the goal state of the recharging task. To reach Subgoal 1 starting from Room B, the robot simply moves to Subgoals 6, 5, 4, 3, 2, and finally to Subgoal 1.

The second step to do hierarchical learning is to define abstract actions or elementary behaviors. For example, we may define the application space of Behavior 4 (corresponding to Subgoal 4) to be Room C and the vicinity of Subgoal 3. Similarly, we may define the application space of Behavior 6 to be Room B and the vicinity of Subgoal 5. A problem with this choice of behavior hierarchy is that it is very specific to this 3-room environment and may not apply to other environments.

Here I define the elementary behaviors in a slightly different way from the above. There are four elementary behaviors. A door passing behavior enables the robot to navigate between Subgoals 3 and 4, and between Subgoals 5 and 6. A docking behavior enables the robot to reach Subgoal 1 from places that surround Subgoal 1 and include Subgoal 2. And two wall following behaviors (wall following right and wall following left) allow the robot to navigate around in the same room. Detailed specifications of the four elementary behaviors can be found in Chapter 5. This choice of elementary behaviors is more appropriate than the one suggested

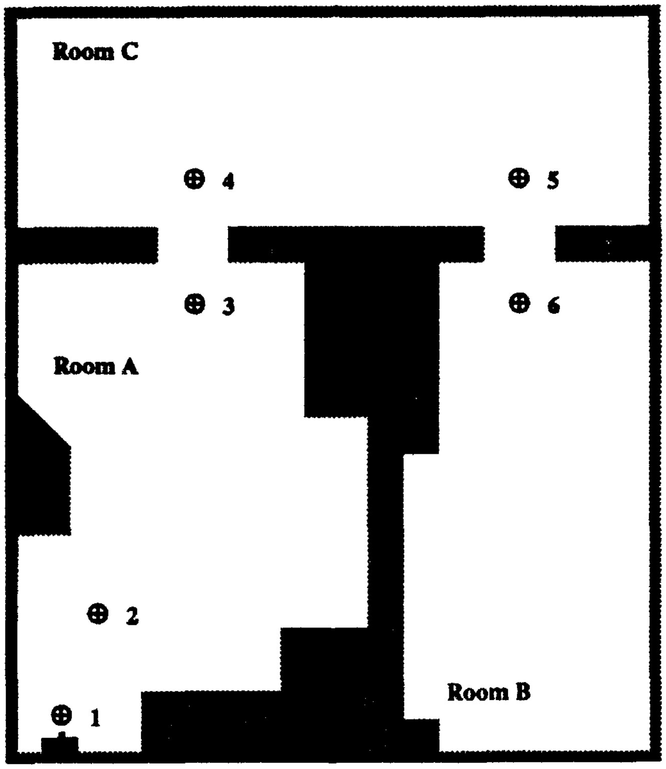


Figure 6.2: Six subgoal states selected for the recharging task.

in the previous paragraph, because it is *environment-independent*. For example, once the robot learns a door passing or wall following behavior in an environment, it can apply it to other environments as well.

The third step is to train the elementary behaviors, which has been discussed in great detail in Chapter 5. The rest of this chapter focuses on the last two steps: finding applicable abstract actions and learning a high-level control policy on top of the elementary behaviors.

6.4.2 Input Representation

There is an important difference between the input representation used for the recharging task and those used for the elementary tasks. As discussed in Chapter 5.3.2, the robot uses a local-coordinates input representation for the elementary tasks. Such a representation is appropriate, because those tasks are local navigation tasks. In contrast, the recharging task is a global navigation task. For this task, the robot uses a compass and a global-coordinates representation. To explain the difference, in the local-coordinates representation, the first element of the state vector encodes the sonar reading coming from the direction in which the robot is facing, while in the global-coordinates representation, the first element encodes the sonar reading coming from north regardless of the robot's orientation.

The state representation for the recharging task includes 16 input units:

- 8 real-valued units encoding 8 sonar readings averaged out of the 24 sonar readings,
- 3 binary units encoding the detection of collisions, light, and potential door edges,
- 3 binary units encoding the room which the robot is currently in, and
- 2 real-valued units encoding the robot's global X-Y position in the environment.

The last 2 units (X-Y position) are used here to remove the hidden state problem, which occurs when the robot is located at Subgoal 4 or Subgoal 5. As illustrated in Figure 5.3 (Chapter 5), both subgoal states display very similar sensory readings. But the optimal behaviors in both situations are different— one is to enter the doorway to Room A, while the other is not to enter the doorway to Room B. To concentrate on the issue of hierarchical learning, in this chapter I ignore the hidden state problem, which will be the focus of the next chapter. In Chapter 8, I will describe a learning robot, which has memory of its past and no longer needs the X-Y information.

6.4.3 Applicable Elementary Behaviors

As discussed previously (Section 6.2.2), an important question during learning is to determine whether the agent is currently in the application space of a given elementary behavior. For the wall following behaviors, the answer is quite simple— the application space covers the entire state space.

For the docking and door passing behaviors, which have final goal states, the answer can be found from the Q-functions that define the behaviors. Take the door passing behavior for example. There are basically two circumstances where the Q-values may be low: either the robot is far away from any door or the door passing behavior is unable to handle the given situation. In either case, we can say the agent is not in the application space. In contrast, high Q-values means that the robot is close to a door and the execution of the behavior will transfer the robot from its current room to another. Thus the agent is in the application space. Similarly, the applicability of the docking behavior can be determined in this way.

But, what Q-values should be considered high or low? In this work, a simple thresholding was used. A possible way to find a proper threshold for a behavior is to collect statistics on the Q-values of successful experiences in the course of learning. In this work, it was simply chosen off-line based on Q-value maps such as Figure 6.3. Figure 6.3 illustrates the Q-function that was used for door passing. According to the figure, Q-values greater than 30 can be considered high. Due to noise or imperfect learning, there are places where no door is nearby but the Q-values are abnormally high.[4] But this was not found to cause serious problems for the robot during learning the high-level task. Figure 6.4 illustrates the Q-function that was used for docking. The docking behavior was considered applicable when its Q-values were greater than 25.

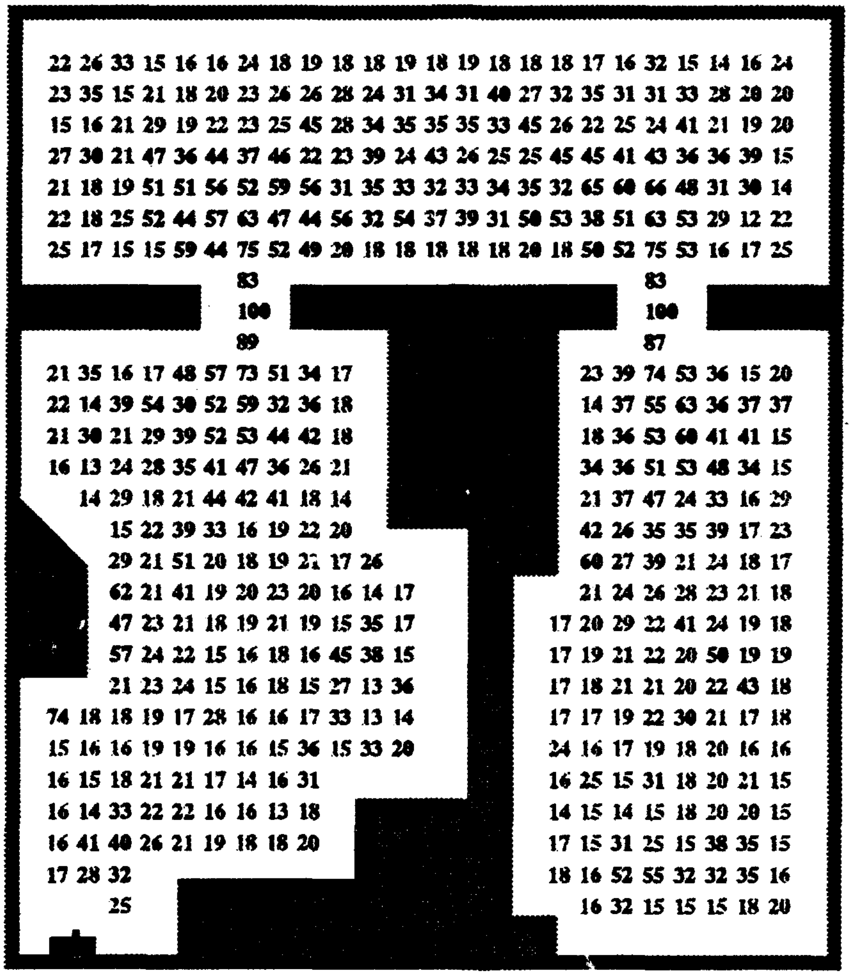
Note that because it is fine to activate an inappropriate behavior, the choice of the threshold was not found critical as long as it was not too high to rule out the optimal behavior.

6.5 Behaviors That Last Forever

In my proposed abstraction model, each abstract action has a subgoal state and ends automatically once the subgoal state is reached. Applying those abstract actions, the agent can navigate among abstract states without worrying about the low-level details of getting from one abstract state to another. Therefore, when learning a high-level control policy, the agent can naturally treat abstract states and abstract actions just like base states and base actions.

In general, however, robot behaviors may not always have this nice property of automatic termination. *Behaviors may last forever once they are activated.* In the robot simulator, the wall following behaviors have no termination condition at all. Even though the door passing behavior has a termination condition (i.e., getting into another room), it may also never stop

[4] If a neural network is never trained on an input pattern that is very different from the training patterns, we generally cannot guarantee that the network will display a reasonable output in response to this input pattern. Because of this, if the door passing behavior is only trained on the near-door locations, its Q-net may output abnormally high Q-values for situations it has not been trained on (such as near-charger locations). To minimize the problem, the door passing behavior was in fact occasionally trained on locations far away from the doors.



```
22 26 33 15 16 16 24 18 19 18 18 19 18 19 18 18 18 17 16 32 15 14 16 24
23 35 15 21 18 20 23 26 26 28 24 31 34 31 40 27 32 35 31 31 33 28 20 20
15 16 21 29 19 22 23 25 45 28 34 35 35 35 33 45 26 22 25 24 41 21 19 20
27 30 21 47 36 44 37 46 22 23 39 24 43 26 25 25 45 45 41 43 36 36 39 15
21 18 19 51 51 56 52 59 56 31 35 33 32 33 34 35 32 65 60 66 48 31 30 14
22 18 25 52 44 57 63 47 44 56 32 54 37 39 31 50 53 38 51 63 53 29 12 22
25 17 15 15 59 44 75 52 49 20 18 18 18 18 18 20 18 50 52 75 53 16 17 25
                  83                                    83
                  100                                   100
                  89                                    87
21 35 16 17 48 57 73 51 34 17                        23 39 74 53 36 15 20
22 14 39 54 30 52 59 32 36 18                        14 37 55 63 36 37 37
21 30 21 29 39 52 53 44 42 18                        18 36 53 60 41 41 15
16 13 24 28 35 41 47 36 26 21                        34 36 51 53 48 34 15
   14 29 18 21 44 42 41 18 14                        21 37 47 24 33 16 29
      15 22 39 33 16 19 22 20                        42 26 35 35 39 17 23
      29 21 51 20 18 19 21 17 26                     60 27 39 21 24 18 17
      62 21 41 19 20 23 20 16 14 17                  21 24 26 28 23 21 18
      47 23 21 18 19 21 19 15 35 17               17 20 29 22 41 24 19 18
      57 24 22 15 16 18 16 45 38 15               17 19 21 22 20 50 19 19
      21 23 24 15 16 18 15 27 13 36               17 18 21 21 20 22 43 18
74 18 18 19 17 28 16 16 17 33 13 14               17 17 19 22 30 21 17 18
15 16 16 19 19 16 16 15 36 15 33 20               24 16 17 19 18 20 16 16
16 15 18 21 21 17 14 16 31                        16 25 15 31 18 20 21 15
16 14 33 22 22 16 16 13 18                        14 15 14 15 18 20 20 15
16 41 40 26 21 19 18 18 20                        17 15 31 25 15 38 35 15
17 28 32                                          18 16 52 55 32 32 35 16
      25                                             16 32 15 15 15 18 20
```

Figure 6.3: Illustration of the Q-function that defines the door passing behavior. The Q-function can be thought of as an evaluation function of the robot's positions, orientations, and actions. Each number in the figure represents the maximum Q-value over all actions and 4 pre-selected orientations when the robot is on the corresponding location. The application space of the door passing behavior can be defined as the states where Q-values are greater than 30.

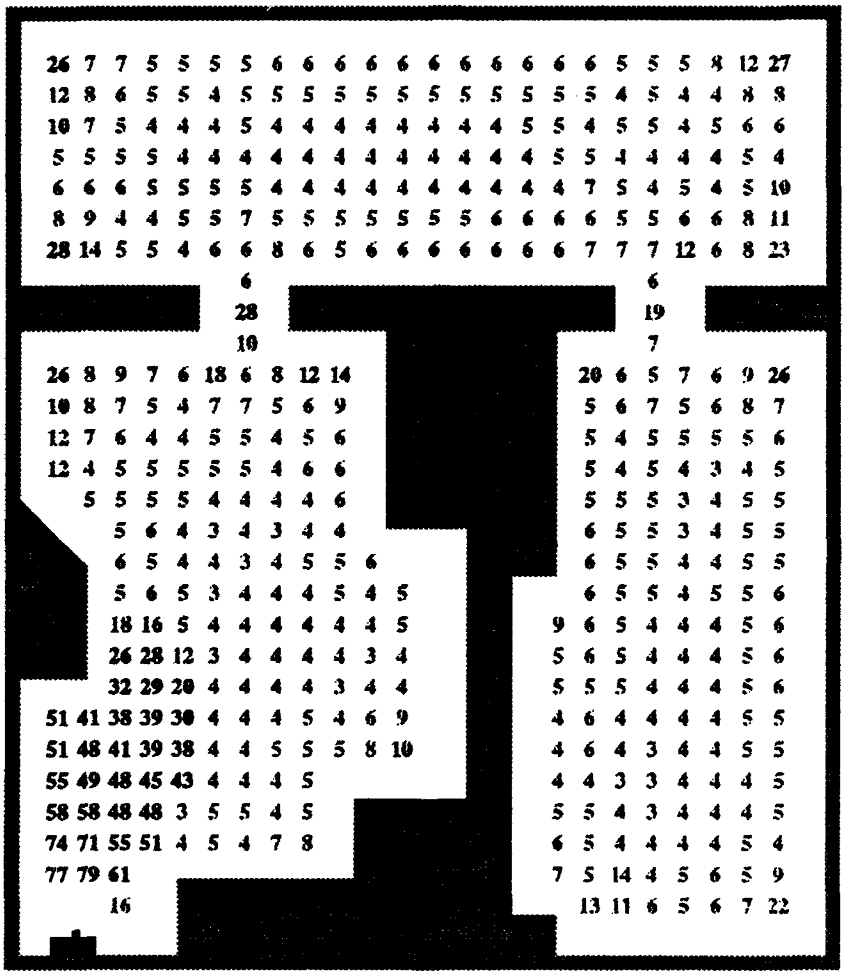


Figure 6.4: Illustration of the Q-functions that defines the docking behavior. Each number in the figure represents the maximum Q-value over all actions and 4 pre-selected orientations when the robot is on the corresponding location. The application space of the docking behavior can be defined as the states where Q-values are greater than 25.

under certain circumstances. For example, the robot's sensors may mistakenly detect a non-existing door (due to noise), causing an inappropriate activation of the behavior. Or the behavior may not have been trained well enough to deal with every situation that the behavior is supposed to take care of. Under both circumstances, the door passing behavior may never be able to achieve its goal and stop.

Since behaviors may last forever, the agent has to learn not only the *optimal sequence* of behaviors to reach a high-level goal state but also the *optimal timings* for switching behaviors. Because the optimal timings for switching one behavior to another may occur at any time, the learning agent must check [illegible] [illegible] [illegible] the optimal timing [illegible] For [illegible] frequent [illegible]

[illegible] robot chec[illegible] [illegible]ntly t[illegible] [illegible] that the behavior is no longer appr[illegible] [illegible] away.

This work takes an extreme— the [illegible] which behavior to apply is made by the robot on *each primitive step.* In other words, the robot checks to see which behavior is most appropriate; applies it for one single step; determines the next appropriate behavior; applies that behavior for another step; and so on. In the worst case, the robot needs a high-level control policy to decide the optimal behavior for each base state, not just for each abstract state. In other words, the agent needs to learn an evaluation function of $Q(state, behavior)$, where "*state*" here stands for the whole base space. In comparison, if hierarchical learning is not used, the robot has to learn $Q(state, action)$.

Is $Q(state, behavior)$ easier to learn than $Q(state, action)$? I have not been able to answer this question quantitatively in the way I have done in Section 6.2.2, but it appears that the answer is positive. This is because the former Q-function is less complex than the latter, which is easily seen by noting that the robot need switch actions much more frequently than switch behaviors.

6.5.1 Behavior-Based Exploration

To learn $Q(state, behavior)$ effectively, the robot needs a good strategy to explore the state space, which may be large. Fortunately, the robot has some knowledge (i.e., Q-functions) about the elementary behaviors it has learned previously. It can use this knowledge for effective exploration. First of all, not every behavior needs to be tried in a state. In fact, only applicable behaviors need to be tried (Section 6.4.3). Below I discuss an exploration strategy in the context of the battery recharging task. This exploration strategy seems quite general and applicable to many other domains as well:

> **Persistence Rule:** Keep following the same behavior unless some *significant* changes to the robot's situation has happened.

The follow[illegible] o changes can be considered significant: (1) the goal of the docking or door passing b[illegible] is achieved, and (2) a previously inapplicable behavior becomes applicable (for insta[illegible] oor opening suddenly comes into sight, making the door passing behavior become a[illegible] e). The basic idea behind this rule is the following: To make good progress, the robot s[illegible] avoid switching behaviors frequently. Imagine that the robot is close to a door. The door passing behavior and the wall following behaviors are all applicable at this moment. The robot can choose any of them for experimentation, but it should either keep following the wall until another doorway comes into sight or dedicate itself to door passing until the door is completely passed. If the robot keeps changing behaviors, it will often end up being stuck in the same place.

In this study, the robot chooses behaviors stochastically according to the following probability distribution:

$$Prob(b) = W(b) \cdot e^{Q(x,b)/T} / \sum_{k} W(b_k) \cdot e^{Q(x,b_k)/T} \tag{6.6}$$

[illegible]ere x is the input state, b is a behavior, $Q(\cdot)$ is the [illegible] is a temperature [illegible]meter, and [illegible] ght factor for each [illegible]

[illegible]ection [illegible] mulation experiments. Through the experim[illegible] [illegible] how we[illegible] idea of hierarchical reinforcement learning works and how well it [illegible] up. The task was to learn a control policy for the battery recharging task. In the first experiment, hierarchical learning was not used— the robot learned a monolithic Q-function, $Q(state, action)$, for the task. The other 4 experiments are about learning a high-level control policy, $Q(state, behavior)$, for the same task. Experiments 2, 3 and 4 assumed that the elementary behaviors had been learned beforehand, while Experiment 5 allowed high-level and elementary behaviors to be learned simultaneously.

Each of the experiments consisted of two interleaved phases: a learning phase and a test phase. In the learning phase, the robot explored its environment and incrementally learned a Q-function. In the test phase, learning was turned off and the robot always followed the best behavior according to its current policy. Each experiment was repeated 7 times, and the learning curves shown below describe the robot's mean performance. In both the learning phase and test phase, each robot's learning trial ended when the task was accomplished, or else after 200 primitive steps. The experience replay algorithm described in Chapter 5.1 was employed. Approximately 60 lessons were replayed after each trial.

Several learning parameters were used in the experiments:

- discount factor γ,

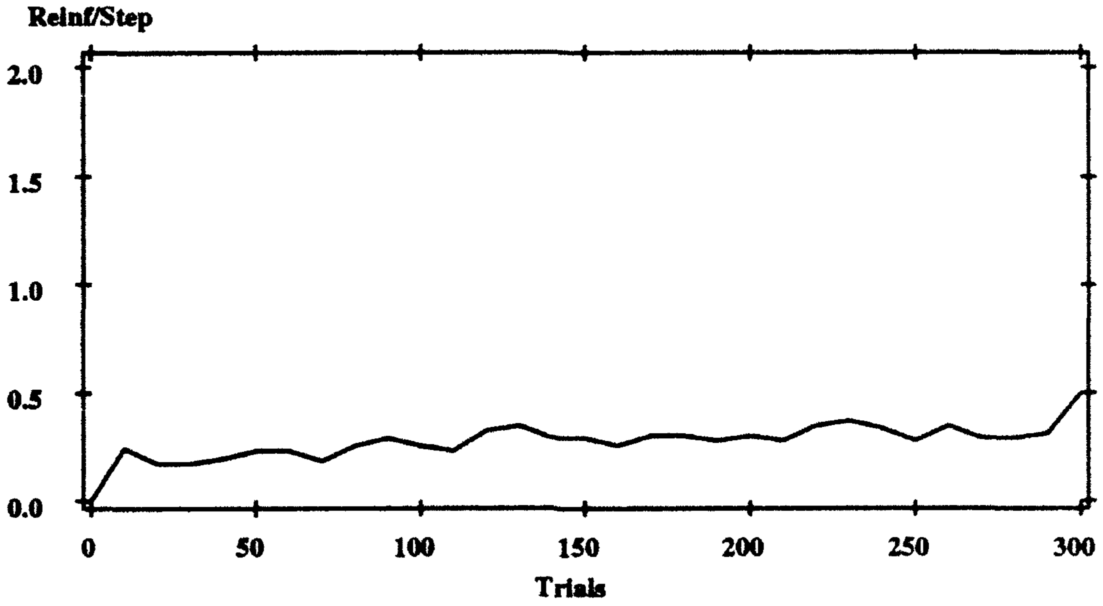


Figure 6.5: The robot's mean performance over 7 runs from Experiment 1. The robot did not use hierarchical learning in this experiment.

- recency factor λ,
- learning rate (fixed to 0.05),
- number of hidden units of Q-nets (fixed to 8 for Experiments 2–5),
- P_l, and
- temperature T (cooling down from 0.05 to 0.02).

Experiences involving bad behavior choices were not replayed. Bad behaviors were those whose probability of being chosen (see Equation 6.6) was less than P_l. P_l was dynamically set to be between 0 and 0.02, depending on the recency of the replayed lesson. ($P_l = 0$ was used for replaying the most recent lesson.)

6.6.1 Experiment 1: Non-Hierarchical Learning

Hierarchical learning was not used in this experiment. The robot was provided with 10 taught lessons to start with. The same learning technique used to train the elementary behaviors (see Chapter 5) was employed to train a recharging behavior. The state representation consisted of 102 input features, including all the input features used by the four elementary behaviors (Section 5.3.2) and the high-level recharging behavior (Section 6.4.2). The Q-nets used 30 hidden units. The parameter settings were $\gamma = 0.99$ and $\lambda = 0.8$.

Figure 6.5 shows the robot's mean performance over 7 runs. The Y axis indicates the average reinforcement per primitive step, and the X axis indicates the number of trials the robot had attempted so far. After 300 learning trials, the robot was generally unable to accomplish the task starting from Rooms B and C. Besides, it collided with obstacles very often (more than 5% of the time). In this experiment, the robot was not punished when it collided with obstacles. In some random tests, the robot received a small punishment (-5 or -10) for each collision. Slightly better performance was observed with this punishment, but it still performed poorly. The robot in fact never learned a systematic way to carry out the recharging task within 300 trials no matter what reward function was used.

6.6.2 Experiment 2: Basics

In this experiment and the rest, the robot was not provided with any teaching example except those for training the elementary behaviors. The state representation consisted 16 of input units, as mentioned in Section 6.4.2.

In this experiment, the robot did not use any special exploration rule. On each primitive step, the robot chose a behavior stochastically according to Equation (6.6). The W-function is computed as follows:

$$W(b) = \begin{cases} 0 & \text{if behavior } b \text{ is not applicable,} \\ 1 & \text{if behavior } b \text{ is applicable.} \end{cases}$$

The parameter settings were: $\gamma = 0.99$ and $\lambda = 0.8$. Figure 6.6 shows the robot's mean performance over 7 runs. This experiment demonstrates two points: On the one hand, the robot failed to learn a good control policy after 150 trials, because it switched behaviors too frequently, resulting in a considerable waste of exploration effort. On the other hand, when starting from Room A, the robot was generally able to accomplish the battery recharging task within as few as 10 learning trials. Within 50 trials, it was able to accomplish the task when it started from some portion of Room C.

6.6.3 Experiment 3: Persistence Exploration Rule

This experiment was similar to Experiment 2, and only differed in that it utilized the persistence exploration rule mentioned in Section 6.5.1. The rule was implemented by defining the W-function as follows:

$$W(b) = \begin{cases} 0 & \text{if behavior } b \text{ is not applicable,} \\ 100 & \text{if behavior } b \text{ is currently active and} \\ & \quad \text{the last reinforcement received by behavior } b \text{ is not 100 and} \\ & \quad \text{no previously inapplicable behavior becomes applicable,} \\ 1 & \text{otherwise.} \end{cases}$$

This experiment used the same parameter settings as Experiment 2. The learning curve is shown in Figure 6.6. Within 20 trials, the robot had developed a good high-level skill for

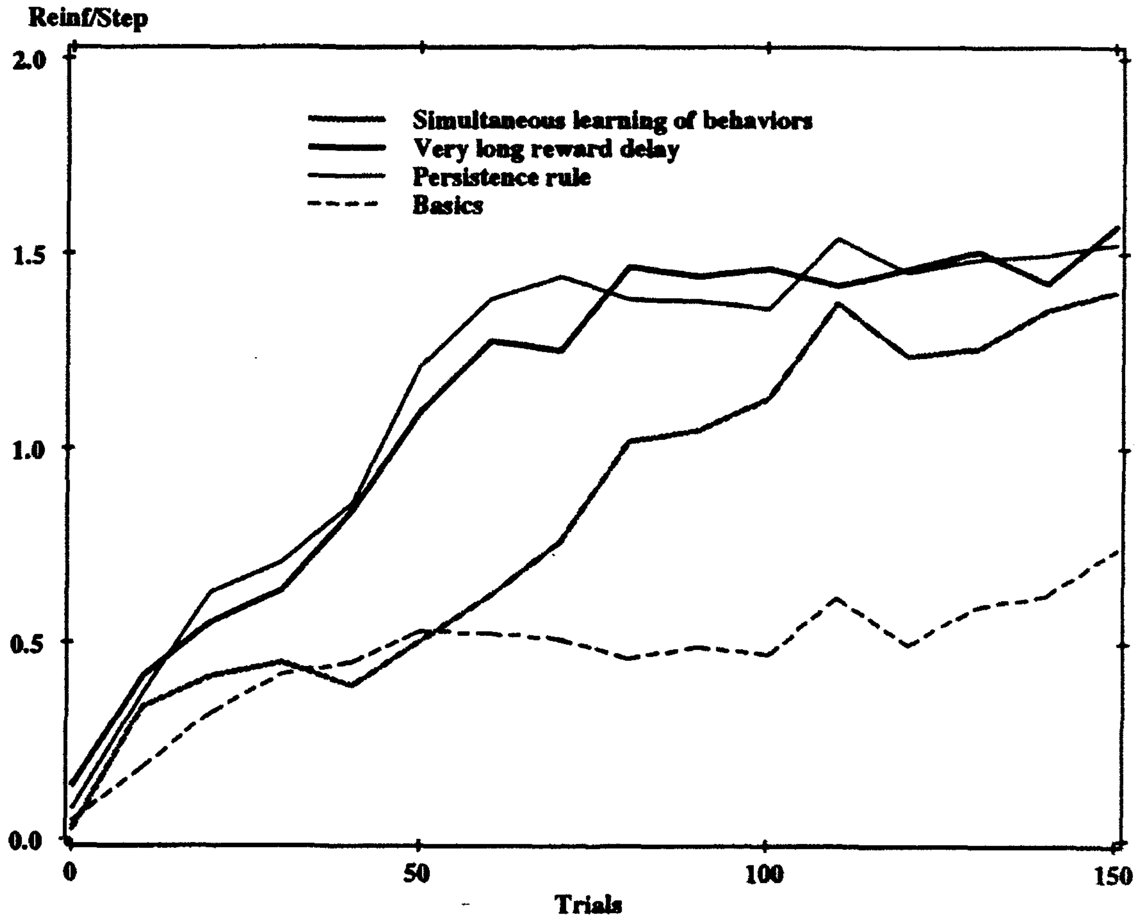


Figure 6.6: The learning curves obtained from the experiments. Each curve shows the robot's mean performance over 7 runs.

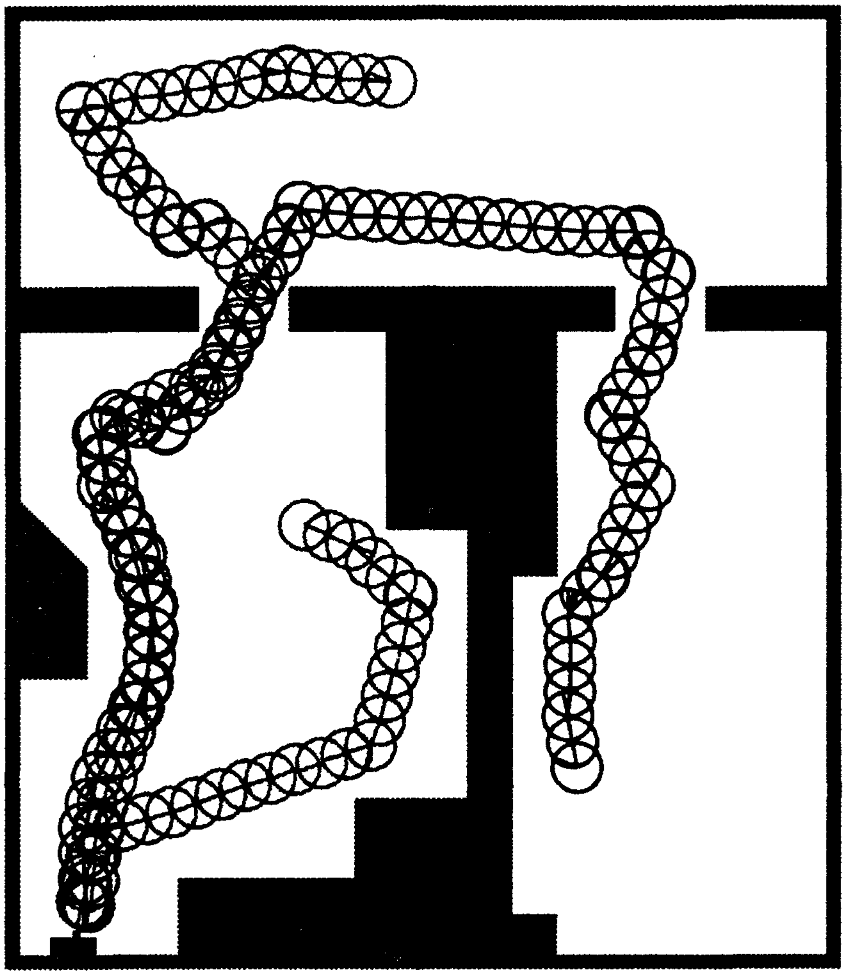


Figure 6.7: Three solution paths found for the recharging task.

connecting to the charger from most locations in Rooms A and C. A nearly optimal skill for all situations was generally learned within 60 trials. Figure 6.7 shows some solution paths found by the robot after 150 learning trials.

6.6.4 Experiment 4: Very Long Reward Delay

Experiment 4 was intended to test if the learning technique described here could deal with very lcng reward delay. Specifically, the reward delay in this experiment was made 5 times longer than that in Experiments 2 and 3. The experimental design was as follows. The step size of each primitive action was reduced to one fifth of its original size. For instance, in Experiment 3 the robot could move 12 inches forward in a single step, while here it could only move 2.4 inches at a time. Thus, the number of steps to reach the goal state was 5 times greater than before, and so was the reward delay. (Moving from Room B to the charger would require approximately 500 steps.) The robot explored its environment in a way similar to that in Experiment 3. The new timeout was 1000 steps instead of 200 steps in Experiment 3. For the purpose of performance comparison, the reward for accomplishing the task was set to 500, 5 times larger than before. In spite of these differences, the optimal high-level Q-functions in both Experiments 3 and 4 are very similar.

In the first attempt, the same learning technique used in Experiment 3 was used. The parameter settings were $\gamma = 0.998$ and $\lambda = 0.9$. The robot's performance was found poor—it was unable to accomplish the task when starting from Room B. I also tried various λ values (ranging from 0.8 to 0.97), but never obtained good performance.

Upon inspection, the main reason it failed seemed to be related to a *re-learning problem*. In supervised learning, if a network is not trained on some patterns for quite a while, the network is typically subject to forgetting what it has learned for those patterns. Re-learning is therefore needed. The same problem seemed to occur here. Consider replaying a lesson of 1000 steps long. The re-learning problem would occur simply because patterns not in the sequence will not be trained on until the entire 1000 patterns have been presented.

A *state grouping* technique is proposed here to deal with the re-learning problem and also to save computation time. The basic idea is the following: When some generalization method is employed, it would be unnecessary to replay an entire action sequence and adjust Q-values for every state-behavior pair along the sequence. Consider a behavior which takes the robot through five consecutive states (a, b, c, d, e). If states b, c, and d have similar state descriptions and similar Q-values, we may need to adjust Q-values for just one of them, since generalization will hopefully take care of the other two. One way to do this is to treat the three states as a single state, c', with c being its center [5]. Then the experience replay algorithm is applied to the new sequence (a, c', e). Note that doing this requires the discount factor, $gamma$, to be dynamically adjusted in order to reflect the fact that the actual distance from c' to a (and to e) is 2 steps, not 1 step. If the temporal distance between two states is n, the discount factor should be changed to γ^n, where γ is the normal discount factor.

In the second attempt, the state grouping technique was used, and the parameter settings were $\gamma = 0.998$ and $\lambda = 0.8$. Here two states were considered similar and grouped together when the Euclidean distance between the two input vectors was less than μ, where μ was a random number between 0.5 and 2.5. In average, every 11 states were grouped together. In general, Q-values of states should also be taken into account to determine whether two states are similar or not, because syntactically similar states may have very different utilities and should not be grouped together. This consideration was not found necessary here, however. The learning curve is shown in Figure 6.6. Clearly, the robot's performance was comparable to that of Experiment 3 in spite of the extended reward delay.

[5] For better performance, we may sometimes choose as the center state b, sometimes state c, and sometimes state d. This allows every pattern to be trained on.

6.6.5 Experiment 5: Simultaneous Learning of Behaviors

In Experiments 2–4, the elementary behaviors were trained separately and independently of the high-level behavior and of each other. There are advantages and disadvantages to doing this. The main advantage is stability. It is hard to imagine what may happen if all the high-level and elementary behaviors are trained simultaneously. It is possible that learning of each behavior would become unstable or easily fall into a poor local maximum due to unexpected interactions among changing behaviors. The disadvantage, on the other hand, is that it may be hard to determine when a behavior needs no further training and allows other behaviors to be learned. Furthermore, it is possible that the elementary behaviors will be over-trained for dealing with situations that occur rarely during solving the high-level task, and in the meantime they will also be under-trained for handling situations that occur frequently during solving the high-level task. An obvious solution to deal with this tradeoff is to train each elementary behavior separately to some acceptable degree and then polish them further as the robot proceeds to learn the high-level behavior. Experiment 5 was designed to demonstrate this point.

Experiment 5 was similar to Experiment 3; both used the same parameter settings. In Experiment 5, each elementary behavior was at first trained separately for approximately 300 primitive steps (about 10–13 trials). Teaching was employed during training the elementary behaviors. (The number of taught lessons used was 10 for door passing and docking, and was 5 for wall following.) As we can see from the learning curves in Figure 5.4 (Chapter 5.4.1), the performance of the elementary behaviors would be quite poor at this point. But the robot went on to learn the recharging behavior, anyway. As the robot was learning the recharging behavior, it further trained the four elementary behaviors.

The learning curve is shown in Figure 6.6. The robot was not making much progress in the first 50 trials, simpıy because its elementary behaviors had not been good enough yet. After 50 trials, the robot's performance started to improve rapidly. At the end of the experiment, it had learned good control policies for the elementary tasks and the recharging task, and was able to connect to the battery charger starting from any room. Although the performance here was slightly worse than that in Experiment 3, the robot did not have to wait for the elementary behaviors to be trained completely.

6.7 Discussion

This section compares the performance of the robot with and without hierarchical learning. More specifically, it compares the learning curve obtained from Experiment 1 (Figure 6.5) and that from Experiment 3 (Figure 6.6). Note that it is unfair to make a point-by-point comparison between the two learning curves, because in the case of hierarchical learning, the robot had already taken many action executions to learn the elementary behaviors before it started to learn the high-level behavior. But let us consider the following two facts:

- **Teaching steps.** In Experiment 1, the robot was provided with 10 taught lessons. In Experiment 3, it was provided with taught lessons for learning the four elementary behaviors. The numbers of teaching steps involved in both experiments were roughly the same and approximately 500.
- **Experimental steps.** When Experiment 1 and Experiment 3 ended, the robot executed approximately the same number of actions (about 40000) in both cases, taking into account the action executions during learning the elementary behaviors.

Given that the robot used approximately the same number of teaching steps and took approximately the same number of experimental steps in both experiments, the one using hierarchical learning was apparently superior to the one without hierarchical learning.

Why did the robot perform significantly better with hierarchical learning than without it? In additional to the theoretical complexity reduction mentioned in Section 6.2, hierarchical learning offers the following two practical advantages over non-hierarchical learning:

- **Better state representations.** Good input representations are crucial to efficient learning, because better representations support better generalization and thus faster learning. One criterion for being a good representation is that it includes only the information relevant to the task. When the recharging task was not decomposed, the robot had to use a large state representation that described the robot state in sufficient detail, even though not all of the detailed information would be needed at the same time during carrying out the task. Such a state representation is less desirable.

  When the task was decomposed, the robot only needed to use a small state representation during solving each subtask, resulting in efficient learning. For example, the light readings, which were needed for docking, would not be needed for door passing and thus could be excluded from the state representation for the door passing behavior. The high-level behavior also used a small state representation, which carried abstract information derived from the low-level sensory inputs. For example, the high-level state representation included 8 (instead of 24) sonar readings averaged out of the 24 sonar readings. Ignoring the features irrelevant to decision making at the moment, the robot was able to learn efficiently.
- **Easier-to-learn Q-nets.** It is a traditional wisdom that training multiple simple networks is often easier than training a single complex network [73]. Without task decomposition, the optimal Q-net for the recharging task is so complex that it could hardly be trained to the desired accuracy at all. With task decomposition, the optimal Q-net for each of the elementary and high-level tasks is simple and can be learned rapidly.

Note that being able to do hierarchical learning, the robot has been provided with additional domain-specific knowledge, including a proper selection of elementary tasks and a proper selection of a reward function and an application space for each elementary task. It seems inevitable that we need domain knowledge to buy great learning speed.

6.8 Related Work

Maes and Brooks [40] described a technique for an artificial insect robot to learn to coordinate its elementary behaviors. Each elementary behavior was pre-programmed and controlled the movement of one of the insect's 6 legs. The high-level control policy to be learned was to coordinate the movement of the 6 legs so that the insect could walk. Their task was less challenging, since an evaluation of the agent's performance was available on every step; no temporal credit assignment was involved.

Mahadevan and Connell [41] presented a box-pushing robot, which was able to learn elementary behaviors from delayed rewards. Their learning technique was reinforcement learning. Instead of using neural networks as I do here, they generalized Q-functions using statistical clustering techniques. Their work can be viewed as complementary to that of Maes and Brooks. In Maes and Brooks' case, the elementary behaviors were wired by humans; what to be learned was behavior coordination. In Mahadevan and Connell's case, the policy for behavior coordination was wired by humans beforehand; what to be learned was the elementary behaviors.

Lewis et al. [33] also studied a six legged insect robot, which successfully learned to walk. The robot was controlled by artificial neural networks with weights determined by genetic algorithms. The robot developed its walking skill in two stages: It developed first an oscillation behavior for each of the six legs, and then a walking behavior that coordinates the six oscillation behaviors. The robot might not have developed a successful walking skill without dividing the learning process into such two stages. Except that we used different optimization techniques (genetic algorithms vs. reinforcement learning), we have shared the same idea of hierarchical learning. Dorigo and Schnepf [15] also described a simulated behavior-based robot, which learned its behaviors hierarchically using genetic algorithms.

Singh [62] also addressed the issue of task decomposition within the reinforcement learning paradigm. He described an efficient way to construct a complex Q-function from elementary Q-functions. However, to use his technique requires two strong assumptions to be met: First, each elementary behavior must have a termination condition. Behaviors like wall following are not allowed. Second, for a given high-level task, there can be only one optimal sequence of behaviors to reach the goal state. For his technique to work properly for the robot task studied here, the recharging task must be divided into multiple tasks; each corresponding to a recharging task in which the robot starts from a different room of the environment. On the other hand, Singh has demonstrated for a simple case that his architecture could learn a new elementary behavior that was not explicitly specified. This suggests a possible way for robots to discover a behavior hierarchy on their own. However, no convincing results have been obtained yet.

Singh [61] also addressed the credit assignment problem in face of long reward delay. He proposed to learn abstract models for predicting the effects of elementary behaviors. Credit propagation can be sped up by doing many shallow lookahead search based upon the abstract models. Basically this extends the idea of one-step lookahead search proposed by Sutton [67] in his DYNA architecture (also studied in Chapter 3.6) to multi-step lookahead search. To use this idea, again, requires each behavior not to last forever, simply because it is impossible to predict the exact effects of a never-ending behavior. In fact, Singh's multi-step lookahead search can also be realized by experience replay and the state grouping technique described in Section 6.6.4. To do what Singh does, we simply group together all consecutive states linked by a single behavior execution, and then do credit assignment on groups of states. The advantage is that no explicit model needs to be constructed— the collection of experiences itself is a model.

Nilsson [52] and McCallum [42] also proposed ideas similar to state grouping. Their basic ideas are to divide the input space into regions based on syntactic similarity (and also locality in McCallum's case). In the meanwhile, a *region graph* is constructed and describes how actions may link regions together. With this graph, solution paths can be found by graph search at run-time. Alternatively, Q-learning can be applied to learn a control policy, whose input representation treats each region as a single state. For their methods to work properly, each region must correspond to a unique portion of the input space— this is often hard to do. Otherwise, a problem similar to hidden state will occur and degrade the performance. In contrast, my state grouping technique is based upon a weaker assumption: consecutive states with similar state descriptions should have similar Q-values and therefore can be grouped together.

6.9 Summary

This chapter presented a model of hierarchical reinforcement learning and analytically showed that a great saving in learning time can be obtained by task decomposition and the use of abstraction. The complexity analysis is restricted to deterministic domains with lookup-table representations; nevertheless, it provides a basis for understanding how hierarchical learning can be done and how it can save learning time.

This chapter also empirically demonstrated the importance of hierarchical learning— by hierarchical learning, a simulated robot was able to solve a complex problem, which otherwise was hardly solvable. The proposed model assumes that every elementary behavior has a termination condition, but in the simulation study this assumption was violated by the never-ending wall following behaviors. Using an appropriate exploration rule, the robot was able to coordinate never-ending behaviors in spite of long reward delay, which in one case was as large as 500 steps.

Another assumption made by the proposed model is the independent training of elementary behaviors and high-level behaviors. The purpose of doing this is mainly stability, but it may be hard to decide when a behavior requires no further training. In one experiment, the robot at first trained each elementary behavior separately for approximately 300 primitive steps, and then trained the elementary behaviors and the high-level behavior all simultaneously. In this way, an agent can start learning high-level behaviors without waiting for the elementary behaviors to be completely developed.

Hierarchical learning in this research relies on a careful decomposition of tasks. To decompose a task properly often requires some domain-specific knowledge. For example, as a designer of the robot studied here, I knew the four elementary behaviors would enable the robot to achieve the recharging task in an arbitrary multi-room environment. However, I need not know exactly how the behaviors can be combined to achieve goals, since there can be any number of rooms and the battery charger can be in any one of th^em. An extension to this work and great challenge for the machine learning community would be automatic discovery of a proper behavior hierarchy.

Chapter 7

Learning with Hidden States

The convergence of Q-learning relies on the assumption that any information needed to determine the optimal action is reflected i.ı the agent's state representation. If some important state features are missing (or hidden) from the state representation, true world states cannot be directly identified and optimal decisions cannot be made based on this state representation. This problem is known as the *hidden state problem.*

A possible solution to the hidden state problem is to use history information to help uncover the hidden states. This chapter presents three architectures that learn to use history to handle hidden states; namely, *window-Q*, *recurrent-Q*, and *recurrent-model.* Simulation experiments indicate that these architectures work to some extent for a variety of simple tasks involving hidden states.

This chapter also presents a categorization of reinforcement learning tasks, and discusses the relative strengths and weaknesses of the three learning architectures in dealing with tasks of various characteristics.

7.1 Introduction

Watkins and Dayan [74, 75] have shown that Q-learning will converge and find the optimal policy under a few conditions (see also Chapter 2.2). One of the assumptions is that the world is a Markov decision process; in other words, at any given time, the next state of the environment and the immediate reinforcement to be received are determined only by the current state and the action taken. In such environments, all information needed to determine the current optimal action is reflected in the current state representation.

Consider a reinforcement learning agent whose state representation is based on only its immediate sensation. When its sensors are not able to make essential distinctions among world states, the Markov assumption mentioned above is violated. Consider a packing task which involves 4 steps: open a closed box, put a gift into it, close it, and seal it. An agent driven only by its current visual percepts cannot accomplish this task, because when facing a closed box, the agent does not know if a gift is already in the box and therefore cannot decide whether to seal or to open the box. This problem is known as the *hidden state problem*, because some state features (in this example, *whether there is a gift in the box*) are hidden from the agent's sensory inputs. A possible solution to the hidden state problem is to use history information, such as whether a gift was ever put in the box, to help determine the current world state.

The rest of the chapter is organized as follows. Section 7.2 distinguishes two types of non-Markovian control problems: *active perception tasks* and *hidden state tasks*. This dissertation focuses on the second type of tasks, and Section 7.3 discusses three connectionist architectures for dealing with such tasks. Section 7.4 proposes OAON network architectures for model learning and Q-learning. Section 7.5 presents empirical study of these architectures, and Section 7.6 discusses their relative strengths and weaknesses. Finally, Section 7.7 discusses related work, and Section 7.8 summarizes the main lessons learned.

7.2 Active Perception and Hidden State

Two important types of control problems are inherently non-Markovian: *active perception tasks* and *hidden state tasks*. Active perception tasks are those in which the learning agent has to effectively manage its sensing operations in order to collect adequate state information for decision making. Hidden state tasks are those in which the learning agent has limited sensing capabilities and does not always obtain complete state information from its sensors.

Active perception tasks. Consider an agent with a set of sensing operations which allows the agent to make complete distinctions among world states. Because some sensing operations are very costly (due to either time constraints or physical resource constraints), the agent might choose to apply only some subset of its available sensing operations. When the choice of limited sensing operations is improper, the agent will fail to capture critical state information— this is why active perception tasks tend to be non-Markovian. Whitehead and Ballard [81] proposed a *lion* algorithm, with which a reinforcement learning agent can learn to focus perceptual attention on the right aspect of the environment during control. Tan [69, 68] proposed a *cost-sensitive* algorithm, with which a reinforcement learning agent can learn to choose minimal sensing operations to disambiguate world states. Both algorithms assume deterministic environments.

Hidden state tasks. The gift packing task mentioned previously is a hidden state task, because the agent's sensors are unable to determine whether there is a gift hidden in a closed box. One way to deal with hidden state tasks is to use history information to uncover hidden states. The problem, however, is that given a large amount of information in the past, how an agent can efficiently and effectively utilize its history information.

This dissertation focuses on hidden state tasks. Note that many tasks of interest involve both active perception and hidden state. Consider the gift-packing task again. Suppose after packing, the agent has to place the sealed box on a proper table according to the color of the

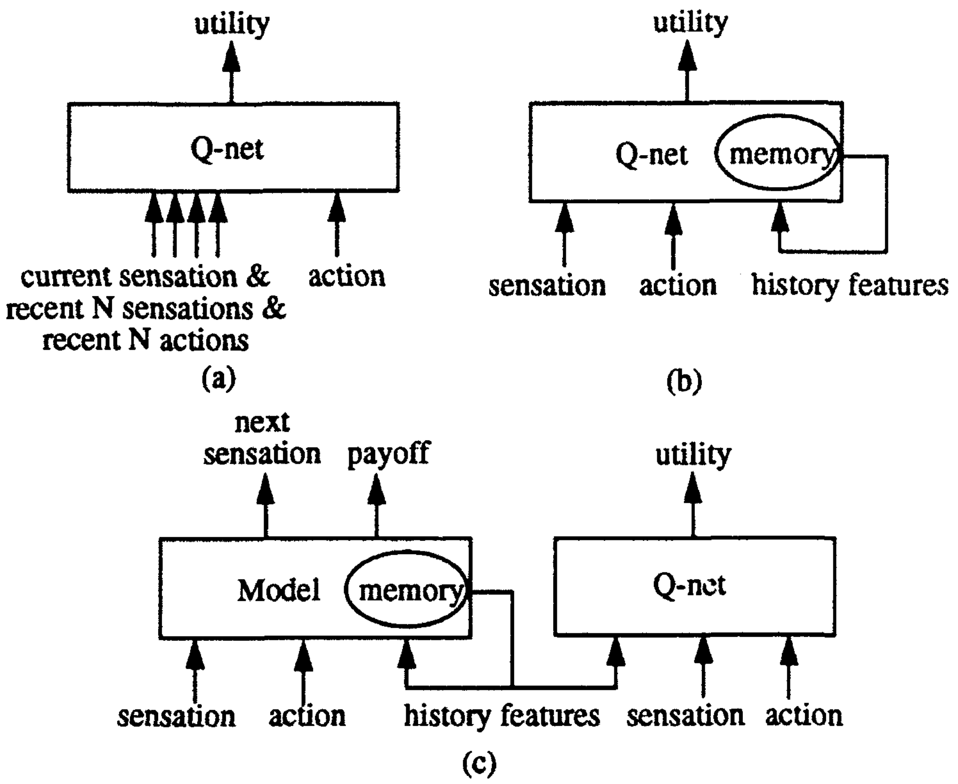


Figure 7.1: Three memory architectures for reinforcement learning with hidden states: (a) window-Q, (b) recurrent-Q, and (c) recurrent-model.

gift inside. Also suppose the agent has an expensive visual operation to identify object colors. To be successful, the agent should identify the color of a gift before it is packed in a box, and remember the color for later use. To be efficient, the agent should apply the expensive color-identification operation once for all.

7.3 Three Memory Architectures

Figure 7.1 depicts three memory architectures for reinforcement learning with hidden states. One is a type of indirect control, and the other two direct control. These architectures all learn a Q-function, which is represented by neural networks.

Instead of using just the current sensation as state representation, the *window-Q architecture* uses the current sensation, the N most recent sensations, and the N most recent actions taken all together as state representation. In other words, the window-Q architecture allows a direct

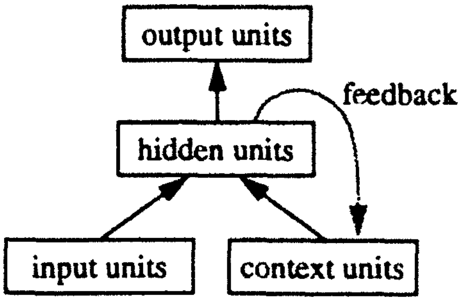


Figure 7.2: An Elman Network.

access to the information in the past through a sliding window. N is called the *window size*. The window-Q architecture is simple and straightforward, but the problem is that we may not know the right window size in advance. If the window size is chosen to be not large enough, there will be not enough history information to construct an optimal Q-function. Moreover, if the window size needs to be large, the large number of units in the input layer will require a lot of training patterns for successful learning and generalization. In spite of these problems, it is still worth study, since this kind of *time-delay neural networks* has been found quite successful in speech recognition [73] and several other domains.

The window-Q architecture is sort of a brute force approach to using history information. An alternative is to distill a (small) set of *history features* out of the large amount of information in the past. This set of features together with the current sensation become the agent's state representation. The optimal control actions then can be determined based on just this new state representation, if the set of history features reflects the information needed for optimal control. The *recurrent-Q* and *recurrent-model* architectures illustrated in Figure 7.1 are based on this same idea, but the ways they construct history features are different. Unlike the window-Q architecture, both architectures in principle can discover and utilize history features that depend on sensations arbitrarily deep in the past, although in practice it is difficult to achieve this.

Recurrent neural networks, such as Elman networks [16], provide a way to construct useful history features. As illustrated in Figure 7.2, the input layer of an Elman network is divided into two parts: the true input units and the *context units*. The operation of the network is clocked. The context units hold feedback activations coming from the network state at a previous clock time. The context units, which function as the memory in Figure 7.1, remember an aggregate of previous network states, so the output of the network depends on the past as well as on the current input.

The recurrent-Q architecture uses a recurrent network to model the Q-function. To predict utility values correctly, the recurrent network (called recurrent Q-net) will be forced to develop history features which enable the network to properly assign different utility values to states displaying the same sensation.

The recurrent-model architecture consists of two concurrent learning components: learning an *action model* and Q-learning. An action model is a function which maps a sensation and an action to the next sensation and the immediate payoff. To predict the behavior of the environment, the recurrent action model will be forced to develop a set of history features. Using the current sensation and this set of history features as state representation, we can in effect turn a non-Markovian task into Markovian one and solve it using the conventional Q-learning, if the action model is perfect. This is simply because at any given time, the next state of the environment and the immediate reinforcement now can be completely determined by this new state representation and the action taken.

Both the recurrent-Q and recurrent-model architectures learn history features using a gradient descent method (i.e., error back-propagation), but they differ in an important way. For model learning, the goal is to minimize the errors between actual and predicted sensations and payoffs. In essence, the environment provides all the needed training information, which is consistent over time as long as the environment does not change. For recurrent Q-learning, the goal is to minimize the errors between temporally successive predictions of utility values, as I have described in previous chapters. The error signals here are computed based partly on information from the environment and partly on the current approximation to the true Q-function. The latter (i.e., the learned Q-function) changes over time and carries no information at all in the beginning of learning. In other words, these error signals are in general weak, noisy and even inconsistent over time. Because of this, whether the recurrent-Q architecture will ever work in practice may be questioned.

In general the action model must be trained to predict not only the new sensation but also the immediate payoff. Consider another packing task which involves 3 steps: put a gift into an open box, seal the box so that it cannot be opened again, place the box in a proper bucket depending on the color of the gift in the box. A reward is given only when the box is placed in the right bucket. Note that the agent is never required to remember the gift color in order to predict future sensations, since the box cannot be opened once sealed. Therefore a model which only predicts sensations will not have the right features needed to accomplish this task. However, for each of the tasks described in Section 7.5, the action model does not need to predict the immediate payoff in order to discover history features needed for optimal control.

It is worthwhile to note that combinations of the three architectures are possible. For example, we can combine the first two architectures: the inputs to the recurrent Q-net could include not just the current sensation but also recent sensations. We can also combine the last two architectures: the memory is shared between a recurrent model and a recurrent Q-net, and history features are developed using the error signals coming from both the model and the Q-net. This chapter is only concerned with the three basic architectures. Further investigation is needed to see if these kinds of combination will result in better performance than the basic versions.

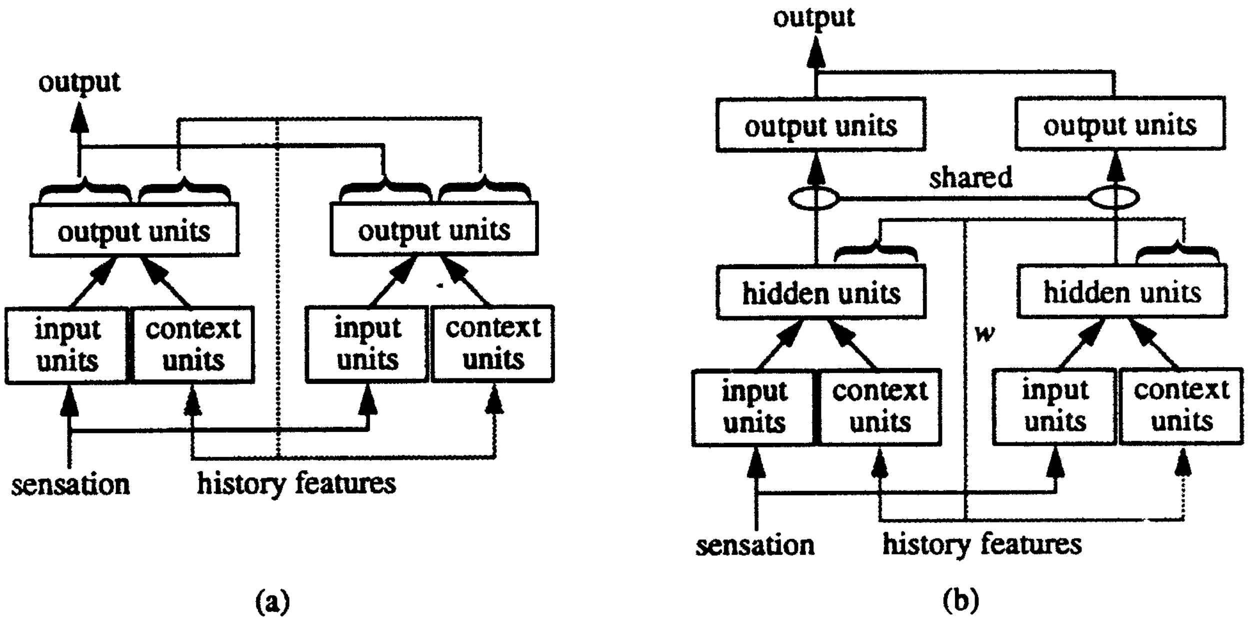


Figure 7.3: Two OAON architectures for recurrent models and recurrent Q-nets: (a) linear and (b) nonlinear. The linear one consists of multiple (here, 2) perceptrons, and the nonlinear one consists of (modified) Elman networks. In both cases, at any given time, only the network corresponding to the selected action is used to compute the next values of the output and context units.

7.4 OAON Network Architectures

Both the Q-function and action model take two kinds of inputs: sensation and action. There are two alternative network structures to realize them. One is a monolithic network whose input units encode both the sensation and the action. For domains with discrete actions, this structure is often undesirable, because the monolithic network is to be trained to model a highly nonlinear function— given the same sensation (and same history features), different actions may demand very different outputs of the network.

An alternative is what I call the OAON (*One Action One Network*) architectures, which use multiple networks (one for each action) to represent the Q-function and the model. Figure 7.3 illustrates two types of recurrent OAON architectures: linear and nonlinear. The linear OAON consists of single-layer perceptrons, and the nonlinear one consists of multilayer networks of units with a nonlinear squashing function. At any given time, only the network corresponding to the selected action is used to compute the next values of the output and context units, while the others can be ignored. It is important to note that we also have to ensure that the multiple networks will use the same (distributed) representation of history features. This is achieved by having the networks share activations of context units, and can be further reinforced (in the case of the nonlinear one) by having the networks share all connections emitting from the hidden layer. As found empirically, the sharing of connections does not seem necessary but tends to help.

As shown in Figure 7.3, the input layer of each network (either linear or nonlinear) is divided into two parts: the true input units and the context units, as we have seen in the Elman network (Figure 7.2). Note that each of the recurrent networks in Figure 7.3.b is a modified Elman network. There are three differences between the "standard" Elman network and my modified version.

- The Elman network does not use *back-propagation through time*, but mine does (see below for details).
- The whole hidden layer of the Elman network is fed back to the input layer, but here only a portion of hidden units is fed back. Consider an environment with just one hidden feature that needs to be discovered. In such a case, it makes sense to have many hidden units but just one context unit. From my limited experience, the Elman network tended to need more context units than this modified one. As a consequence, the former normally required more connections than the latter. Furthermore, since the context units are also part of the inputs to the Q-net, more context units require more training patterns and training time on the part of Q-learning.
- The third difference is in the constant feedback weight, the w in Figure 7.3.b. The Elman network uses $w = 1$, while it was found that w slightly greater than 1 (say, 1.5) tended to make the network converge sooner. The reason it might help is the following: In the early phase of training, the activations of context units are normally close to zero (I used a symmetric squashing function). Therefore, the context units appear unimportant compared with the inputs. By magnifying their activations, the back-propagation algorithm will pay more attention to the context units.

The linear OAON architecture shown in Figure 7.3.a is in fact the SLUG architecture proposed by Mozer and Bachrach [51], who have applied SLUG to learn to model finite state automata (FSA) and demonstrated its success in several domains. (They did not use SLUG to model a Q-function, however). They also found that the conventional recurrent networks, such as the Elman network, were "spectacularly unsuccessful" at modeling FSA's. This may be explained by the fact that their experiments took the monolithic approach (one network for all actions). In contrast, my experiments using the nonlinear OAON architecture were quite successful. It always learned perfectly the FSA's that SLUG was able to learn, although it seemed that more training data would be needed to acquire perfect models due to the fact that nonlinear networks often have more degrees of freedom (connections) than linear ones.[1]

[1] I must clarify what I mean by being able to learn a perfect model. Given any finite set of input-output patterns

To model FSA's, we probably do not need a nonlinear OAON; linear OAONs (SLUG) may suffice and outperform nonlinear OAONs. But for many applications, a nonlinear OAON is necessary and cannot be replaced by the linear one. For example, Q-functions are in general nonlinear. For a pole balancer to be discussed later, its action model is also nonlinear. For the sake of comparing simulation results, I used nonlinear OAONs for all the recurrent Q-nets and models in all the experiments to be presented below.

Both the hidden units and the output units (in the nonlinear case) used a symmetric sigmoid squashing function $y = 1/(1 + e^{-x}) - 0.5$. A slightly modified version of the back-propagation algorithm [58] was used to adjust network weights. *Back-propagation through time* (BPTT) [58] was used and found to be significant improvement over back-propagation without BPTT. To apply BPTT, the recurrent networks were completely unfolded in time, errors were then back-propagated through the whole chain of networks, and finally the weights were modified according to the cumulative gradients. (Readers may refer to Chapter 2.4 and Figure 2.4.) To restrict the influence of output errors at time t on the gradients computed at times much earlier than t, I applied a decay to the errors when they were propagated from the context units at time t to the hidden layer at time $t - 1$. Better performance was found with this decay than without it. The decay used here was 0.9. The context units were initialized to 0 at the start of each task instance.

7.5 Experimental Designs and Results

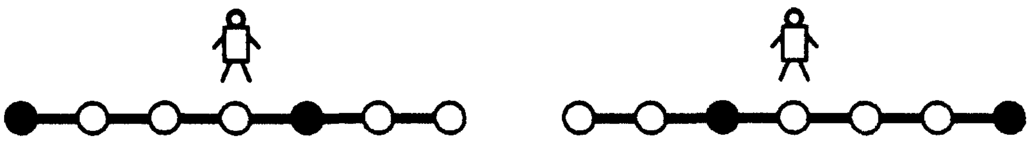
This section presents a study of the three architectures for solving various learning problems with different characteristics. Through the study, we expect to gain more insights into these architectures, such as whether these architectures work and which one may work best for which types of problems.

Several parameters were used in the experiments, such as the discount factor γ, the recency factor λ used in TD(λ) methods, the learning rate, the number of context units, the number of hidden units, the window size, etc. Several of them, however, were fixed for all of the experiments. Appendix B gives a detailed description of the parameter settings used here, which were chosen to give roughly the best performance.

The *experience replay* algorithm described in Chapter 5.1 was used to significantly reduce the number of trials needed to learn an optimal control policy. The experience replay algorithm can be used in batch mode or in incremental mode. Here non-recurrent Q-nets were trained in incremental mode, while recurrent Q-nets in batch mode to avoid possible instability. In all of

generated by an unknown finite state automaton (FSA), there are infinite number of FSA's which can fit perfectly the training data. Because of this, at any given moment, a model learning algorithm can never be sure whether it has learned exactly the same FSA as the one producing the training data, unless some characteristic (for example, the upper bound of size) of the target FSA is known in advance. Therefore, when I say an architecture can learn an FSA perfectly, I mean that it can predict an environment perfectly for a long period of time, say a thousand of steps.

2 possible initial states:



3 actions: walk left, walk right & pick up
4 binary inputs: left cup, right cup, left collision & right collision
reward: 1 when the last cup is picked up
0 otherwise

Figure 7.4: Task 1: A 2-cup collection task.

the experiments, only the experience from the most recent 70 trials would be replayed, while the action model was trained on all the experience in the past[2].

Each experiment consisted of two interleaved phases: a learning phase and a test phase. In the learning phase, the agent chose actions stochastically (see the stochastic action selector in Chapter 3.4). This randomness in action selection ensured sufficient exploration by the agent during learning. In the test phase, the agent always took the best actions according to its current policy. All of the experiments were repeated 5 times. The learning curves shown below describe the mean performance in the test phase.

7.5.1 Task 1: 2-Cup Collection

I started with a simple 2-cup collection task (Figure 7.4). This task requires the learning agent to pick up two cups located in a 1-D space. The agent has 3 actions: walking right one cell, walking left one cell, and pick-up. When the agent executes the pick-up action, it will pick up a cup if and only if the cup is located at the agent's current cell. The agent's sensation includes 4 binary bits: 2 bits indicating if there is a cup in the immediate left or right cell, and 2 bits indicating if the previous action results in a collision from the left or the right. An action attempting to move the agent out of the space will cause a collision.

The cup-collection problem is restricted such that there are only two possible initial states (Figure 7.4). In each trial, the agent starts with one of the two initial states. Because the agent can see only one of the two cups at the beginning of each trial, the location of the other cup can

[2] Previously, the action model was only trained on the experiences from the most recent 70 trials. The resulting model was found poor. As pointed out by Chris Atkeson, experiences from earlier trials in fact carry a better diversity of information than those from recent trials, because the agent is gradually converging to the optimal behavior as time goes by. Therefore, experiences from early trials should be kept for training the action model.

only be learned from previous trials. To collect the cups optimally, the agent must use history information, such as which initial state it starts with, to decide which way to go after picking up the first cup. Note that the reason for restricting the number of initial states is to simplify the task and to avoid ambiguity in the very beginning where no history information is available. This task is not trivial for several reasons: (1) The agent cannot sense the cup right in its current cell, (2) it gets no reward until both cups are picked up, and (3) it often operates with no cup in sight especially after picking up the first cup. The optimal policy requires 7 steps to pick up the 2 cups in each situation.

Figure 7.5 shows the learning curves for the three architectures. These curves show the mean performance over 5 runs. The Y axis indicates the number of steps to pick up the four cups in both task instances shown in Figure 7.4 before time out (i.e., 30 steps). The X axis indicates the number of trials the agent had attempted so far. In each trial the agent started with one of the two possible initial states, and stopped either when both cups were picked up, or else after time out.

I experimented with two different λ values. Two things are obvious from the learning curves. First, with $\lambda = 0.8$, all of these architectures successfully learned the optimal policy. Second, $\lambda = 0.8$ gave much better performance than $\lambda = 0$. Consider the situation in Figure 7.6, where A–E are five consecutive states and F is a bad state that displays the same sensation as State D. With $\lambda = 0$, the TD method estimates the utility of State C (for example) based on just the immediate payoff Rc and the utility of State D. Since State D displays the same sensation as the bad state F, the utility of State D will be much underestimated, until features are learned to distinguish States D and F. Before the features are learned, the utilities of State C and the preceding states will also be much underestimated. This problem is mitigated when $\lambda > 0$ is used, because the TD($\lambda > 0$) method estimates the utility of State C not only based on Rc and State D, but also based on all the following rewards (Rd, Re, etc) and states (State E, etc).

The following are some additional observations, which are not shown in Figure 7.5:

The window size N. The performance in Figure 7.5.a was obtained with $N = 5$. As a matter of fact, the optimal policy could be learned (but only occasionally) with $N = 2$. Imagine that the agent has picked up the first cup and is walking towards the second cup. For a few steps, the most recent 3 sensations (including the current one) in fact do not provide the agent any information at all (namely, all sensation bits are off). How could the agent ever learn the optimal policy with $N = 2$? The way it worked is the following: After picking up the first cup, the agent determines the right direction to move. Later on, the agent simply follows the general direction the previous actions has been headed for. In other words, *the agent's action choices are themselves used as a medium for passing information from the past to the present.*

Robustness. The policy learned by the window-Q architecture is not very "robust", because I can fool it in the following manner: Suppose the agent is following its learned optimal policy. I suddenly interrupt the agent and force it to move in the opposite direction for a few steps. The agent then cannot resume the cup collection task optimally, because its policy will choose to

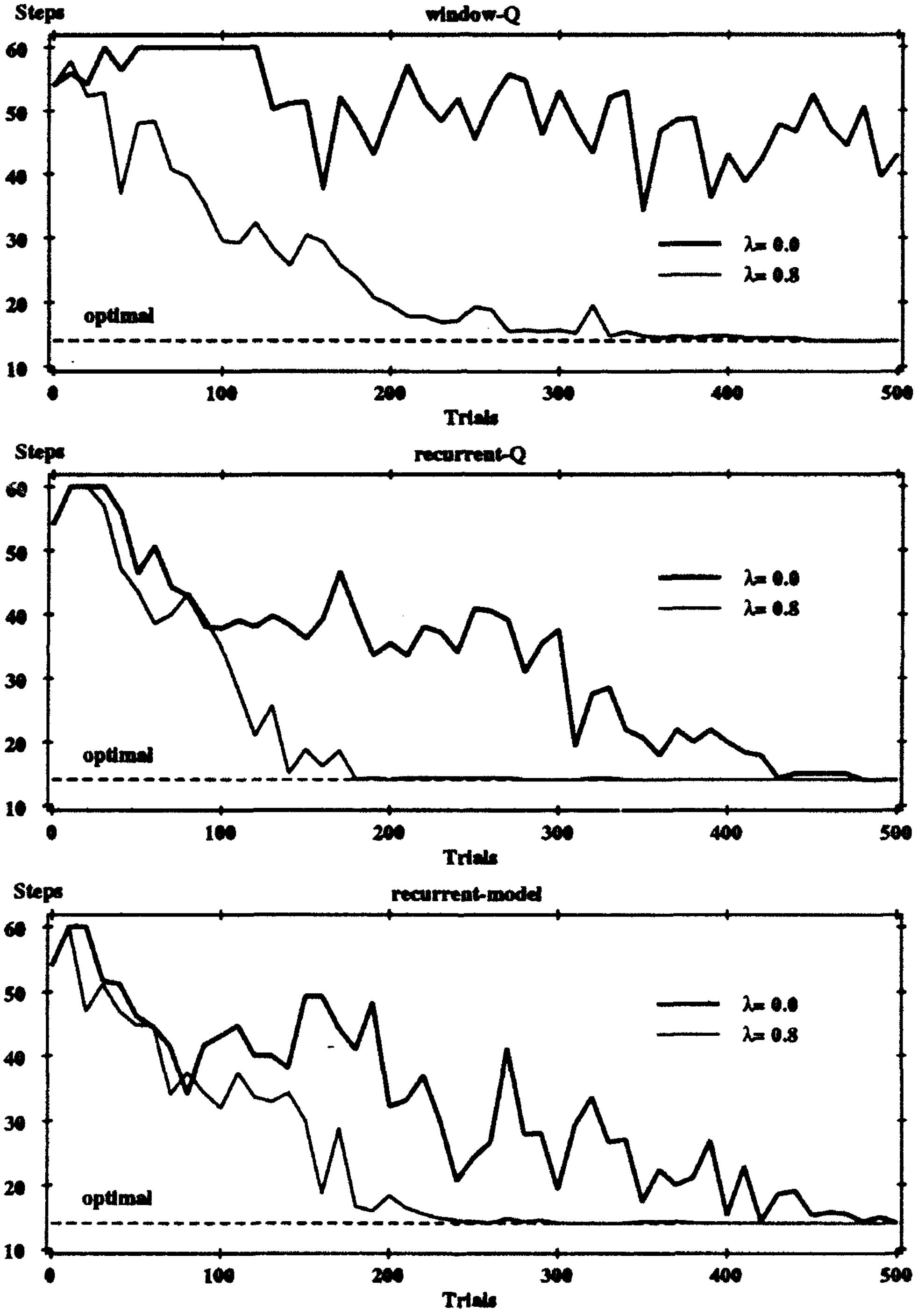


Figure 7.5: Performance for Task 1: (a) window-Q, (b) recurrent-Q, and (c) recurrent-model.

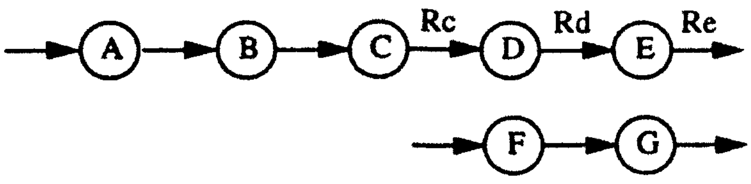


Figure 7.6: A difficult situation for TD(0). A–E are five consecutive states and F–G are bad states. Rc, Rd, and Re are the payoffs received on the state transitions. Also, States D and F display the same sensation.

continue on the new direction, which is opposite to the optimal direction (see the last paragraph). In contrast, the policy learned by the recurrent-model architecture was found more robust; it appeared to choose actions based on the actual world state rather than on the actions taken before. I also examined the policy learned by the recurrent-Q architecture— its robustness appeared to be somewhere between the above two; sometimes it appeared to choose actions based on the actual world state, and sometimes based on its recent action choices. It would be instructive to investigate what history features were learned in the model versus in the recurrent Q-net.

Imperfect model. The agent never learned a perfect action model within 500 trials. For instance, if the agent had not seen a cup for 10 steps or more, the model normally was unable to predict the appearance of the cup. But this imperfect model did not prevent Q-learning from learning an optimal policy. This reveals that *the recurrent-model architecture does not need to learn a perfect model in order to learn an optimal policy.* It needs to learn only the important aspects of the environment. But since it does not know in advance what are and what are not important to the task, it will end up attempting to learn everything including the details of the environment that are relevant to predicting the next sensation but unnecessary for optimal control.

Computation time. Both the recurrent-Q and recurrent-model architectures found the optimal policy after about the same number of trials, but the actual CPU time taken was very different. The former took 20 minutes to complete a run, while the latter took 80 minutes. This is because the recurrent-model architecture had to learn a model as well as a feed-forward Q-net and the model network was much bigger than the recurrent Q-net.

This experiment revealed two lessons:

- All of the three architectures worked for this simple cup-collection problem.
- For the recurrent-model architecture, just a partially correct model may provide sufficient history features for optimal control. This is good news, since a perfect model is often difficult to obtain.

7.5.2 Task 2: Task 1 With Random Features

Task 2 is simply Task 1 with two random bits in the agent's sensation. The random bits simulate two difficult-to-predict and irrelevant features accessible to the learning agent. In the real world, there are often many features which are difficult to predict but fortunately not relevant to the task to be solved. For example, predicting whether it is going to rain outside might be difficult, but it does not matter if the task is to pick up cups inside. The ability to handle difficult-to-predict but irrelevant features is important for a learning system to be practical.

Figure 7.7 shows the learning curves of the three architectures for this task. Again, these curves are the mean performance over 5 runs. As we can see, the two random features gave little impact on the performance of the window-Q architecture or the recurrent-Q architecture, while the negative impact on the recurrent-model architecture was noticeable.

The recurrent-model architecture in fact found the optimal policy many times after 300 trials. It just could not stablize on the optimal policy; it oscillated between the optimal policy and some sub-optimal policies. I also observed that the model tried in vain to reduce the prediction errors on the two random bits. There are two possible explanations for the poorer performance compared with that obtained when there are no random sensation bits. First, the model might fail to learn the history features needed to solve the task, because much of the effort was wasted on the random bits. Second, because the activations of the context units were shared between the model network and the Q-net, a change to the representation of history features on the model part could simply destabilize a well-trained Q-net, if the change is significant. The first explanation is ruled out, since the optimal policy indeed was found many times. To test the second explanation, I fixed the model at some point of learning and allowed only changes to the Q-net. In such a setup, the agent found the optimal policy and indeed stuck to it.

This experiment revealed two lessons:

- The recurrent-Q architecture is more economic than the recurrent-model architecture in the sense that the former will not try to learn a history feature if it does not appear to be relevant to predicting utilities.

- A potential problem with the recurrent-model architecture is that changes to the representation of history features on the model part may cause instability on the Q-net part.

7.5.3 Task 3: Task 1 With Control Errors

Noise and uncertainty prevail in the real world. To study the capability of these architectures to handle noise, I added 15% control errors to the agent's actuators, so that 15% of the time the executed action would not have any effect on the environment. (The 2 random bits were removed.) Figure 7.8 shows the mean performance of these architectures over 5 runs. Note

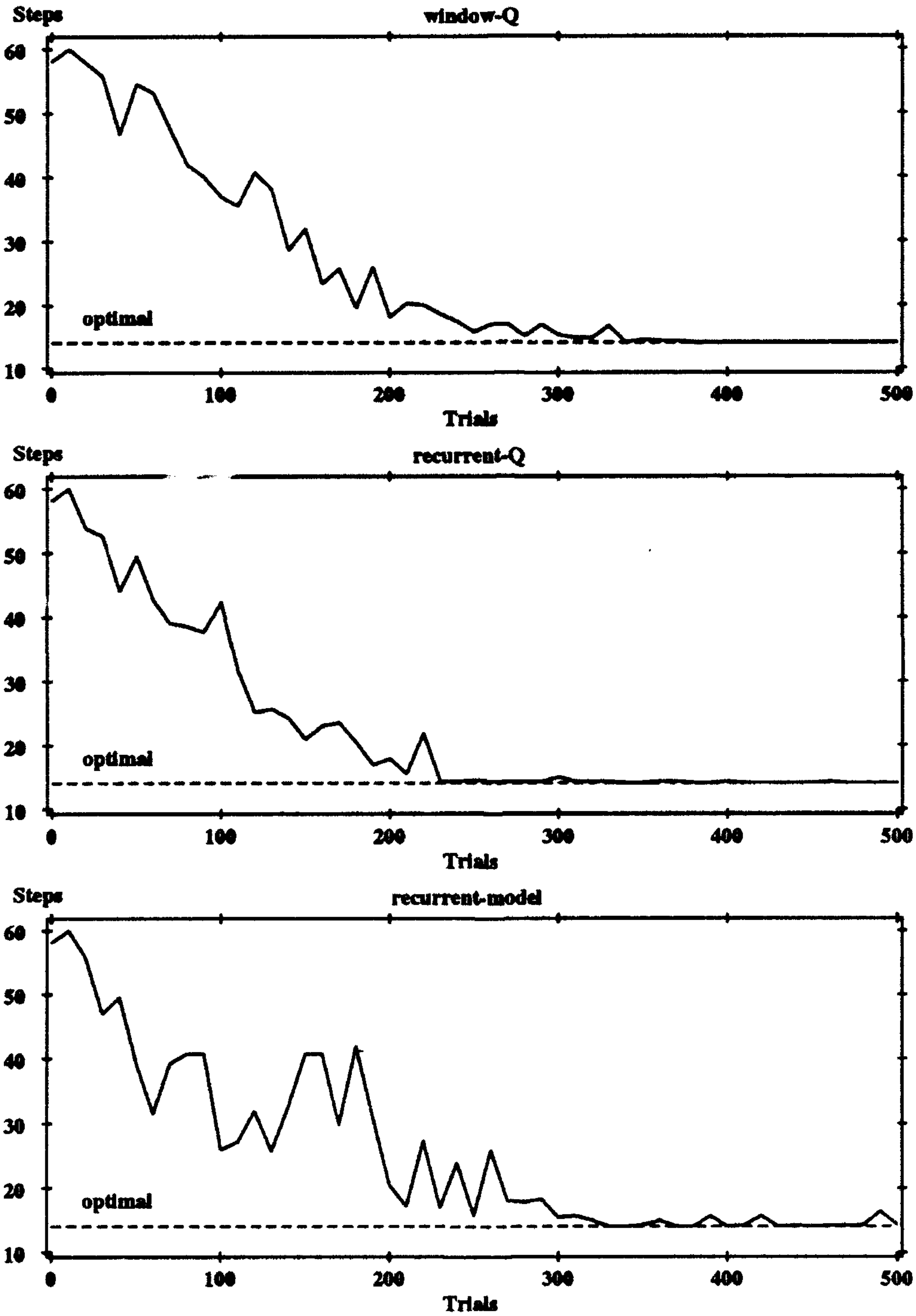


Figure 7.7: Performance for Task 2: (a) window-Q, (b) recurrent-Q, and (c) recurrent-model.

that I turned off the control errors when I tested the performance of the agent, so the optimal number of steps in the figure is still 14. [3]

In 3 out of the 5 runs, the window-Q architecture successfully found the optimal policy, while in the other two runs, it only found suboptimal policies. In spite of the control errors, the recurrent-Q architecture always learned the optimal policy (with little instability).

The recurrent-model architecture always found the optimal policy after 500 trials, but again its policy oscillated between the optimal one and some sub-optimal ones due to the changing representation of history features, much as happened in Task 2. If we can find some way to stablize the model (for example, by gradually decreasing the learning rate to 0 at the end), we should be able to obtain a stable and optimal policy.

In short, two lessons have been learned from this experiment:

- All of the three architectures can handle small control errors to some degree.
- Among the architectures, recurrent-Q seems to scale best in the presence of control errors.

7.5.4 Task 4: Pole Balancing

The traditional pole balancing problem is to balance a pole on a cart given the cart position, pole angle, cart velocity, and pole angular velocity. (See Appendix C for a detailed description of the problem.) This problem has been studied many times (e.g., [3]) as a nontrivial control problem due to sparse reinforcement signals, which are -1 when the pole falls past 12 degrees and 0 elsewhere. Task 4 is the same problem except that only the cart position and pole angle are given to the learning agent. To balance the pole, the agent must learn features like velocity. In this experiment, a policy was considered satisfactory whenever the pole could be balanced for over 5000 steps in each of the 7 test trials where the pole starts with an angle of 0, ± 1, ± 2, or ± 3 degrees. (The maximum initial pole angle with which the pole can be balanced indefinitely is about 3.3 degrees.) In the training phase, pole angles and cart positions were generated randomly. The initial cart velocity and pole velocity are always set to 0. $N = 1$ was used in this experiment.

The input representation used here was straightforward: one real-valued input unit for each of the pole angle and cart position. Table 7.1 shows the number of trials taken by each architecture before a satisfactory policy was learned. These numbers are the average of the results from the best 5 out of 6 runs. A satisfactory policy was not always found within 1000 trials. [4]

[3] In this chapter, I did not experiment with sensing errors, which will be future work. As pointed out by Mozer and Bachrach [51], to train a recurrent model properly in noisy environments, We may need to alter the learning procedure slightly. More precisely, current sensations cannot be 100% trusted for predicting future sensations. Instead, we should trust the model's predicted sensations more as the model gets better.

[4] Memoryless Q-learning took 119 trials to solve the traditional pole balancing problem, in which the agent is given the cart position, pole angle, cart velocity, and pole angular velocity.

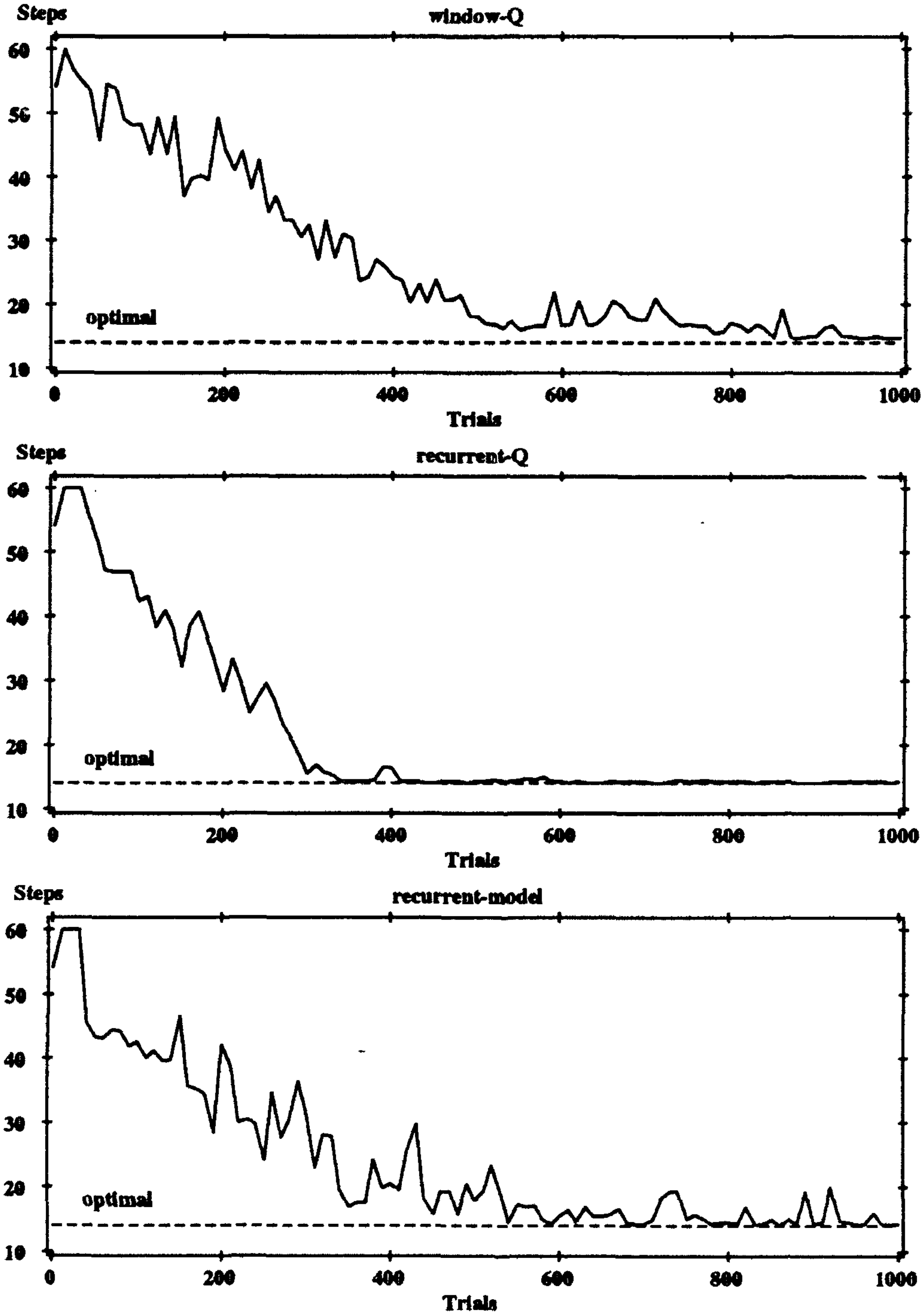


Figure 7.8: Performance for Task 3: (a) window-Q, (b) recurrent-Q, and (c) recurrent-model.

Table 7.1: Performance for the pole balancing task.

method	window-Q	recurrent-Q	recurrent-model
# of trials	206	552	247

A lesson has been learned from this experiment:

- While the recurrent-Q architecture was the most suitable architecture for the cup cc!lection tasks, it was outperformed by the other two architectures for the pole balancing task.

7.6 Discussion

The above experiments provide some insight into the performance of the three memory architectures. This section considers problem characteristics that determine which architecture is most appropriate to which task environments.

7.6.1 Prɔblem Parameters

The three architectures exhibit different advantages whose relative importance varies with problem parameters such as:

- **Memory depth.** One important problem parameter is the length of time over which the agent must remember previous inputs in order to represent an optimal control policy. For example, the memory depth for Task 1 is 2, as evidenced by the fact that the window-Q agent was able to obtain the optimal control based only on a window of size 2. The memory depth for the pole balancing task is 1. Note that learning an optimal policy may require a larger memory depth than that needed to represent the policy.

- **Payoff delay.** In cases where-the payoff is zero except for the goal state, we define the payoff delay of a problem to be the length of the optimal action sequence leading to the goal. This parameter is important because it influences the overall difficulty of Q-learning. As the payoff delay increases, learning an accurate Q-function becomes increasingly difficult due to the increasing difficulty of credit assignment.

- **Number of history features to be learned.** In general, the more hidden states an agent faces, the more history features the agent has to discover, and the more difficult the task becomes. In general, predicting sensations (i.e., a model) requires more history features than predicting utilities (i.e., a Q-net), which in turn requires more history features than representing optimal policies. Consider Task 1 for example. Only two binary history features are required to determine the optimal actions: "*is there a cup in front?*" and "*is the second cup on the right-hand side or left-hand side?*". But a perfect Q-function requires more features such as "*how many cups have been picked up so far?*" and "*how far is the second cup from here?*". A perfect model for this task requires the same features as the perfect Q-function. But a perfect model for Task 2 requires even more features such as "*what is the current state of the random number generator?*", while a perfect Q-function for Task 2 requires no extra features.

It is important to note that we do not need a perfect Q-function or a perfect model in order to obtain an optimal policy. A Q-function just needs to assign a value to each action in response to a given situation such that their relative values are in the right order, and a model just needs to provide sufficient features for constructing a good Q-function.

7.6.2 Architecture Characteristics

Given the above problem parameters, we would like to understand which of the three architectures is best suited to which types of problems. Here we consider the key advantages and disadvantages of each architecture, along with the problem parameters which influence the importance of these characteristics.

- **Recurrent-model architecture.** The key difference between this architecture and the recurrent-Q architecture is that its learning of history features is driven by learning an action model rather than the Q-function. One strength of this approach is that the agent can obtain better training data for the action model than it can for the Q-function, making this learning more reliable and efficient. In particular, training examples of the action model (<sensation, action, next-sensation, payoff> quadruples) are directly observable with each step the agent takes in its environment. In contrast, training examples of the Q-function (<sensation, action, utility> triples) are not directly observable since the agent must estimate the training utility values based on its own changing approximation to the true Q-function.

  The second strength of this approach is that the learned features are dependent on the environment and independent of the reward function (even though the action model may be trained to predict rewards as well as sensations), and therefore can be reused if the agent has several different reward functions, or goals, to learn to achieve.

- **Recurrent-Q architecture.** While this architecture suffers the relative disadvantage that it must learn from indirectly observable training examples, it has the offsetting advantage that it need only learn those history features that are *relevant* to the control problem. The history features needed to represent the optimal action model are a superset of those needed to represent the optimal Q-function. This is easily seen by noticing that the optimal control action can in principle be computed from the action model (by using look ahead search). Thus, in cases where only a few features are necessary for predicting utilities but many are needed to predict completely the next state, the number of history features that must be learned by the recurrent-Q architecture can be much smaller than the number needed by the recurrent-model architecture.

- **Window-Q architecture.** The primary advantage of this architecture is that it does not have to learn the state representation recursively (as do the other two recurrent network architectures). Recurrent networks typically take much longer to train than non-recurrent networks. This advantage is offset by the disadvantage that the history information it can use are limited to those features directly observable in its fixed window which captures only a bounded history. In contrast, the two recurrent network approaches can in principle represent history features that depend on sensations that are arbitrarily deep in the agent's history.

Given these competing advantages for the three architectures, one would imagine that each will be the preferred architecture for different types of problems:

- One would expect the advantage of the window-Q architecture to be greatest in tasks where the memory depths are the smallest (for example, the pole balancing task).
- One would expect the recurrent-model architecture's advantage of directly available training examples to be most important in tasks for which the payoff delay is the longest (for example, the pole balancing task). It is in these situations that the indirect estimation of training Q-values is most problematic for the recurrent-Q architecture.
- One would expect the advantage of the recurrent-Q architecture — that it need only learn those features relevant to control — to be most pronounced in tasks where the ratio between relevant and irrelevant history features is the lowest (for example, the cup collection task with two random features). Although the recurrent-model architecture can acquire the optimal policy as long as just the relevant features are learned, the drive to learning the irrelevant features may cause problems. First of all, representing the irrelevant features may use up many of the limited context units at the sacrifice of learning good relevant features. Secondly, as we have seen in the experiments, the recurrent-model architecture is also subject to instability due to changing representation of the history features— a change which improves the model is also likely to deteriorate the Q-function, which then needs to be re-learned.

The tapped delay line scheme, which the window-Q architecture uses, has been widely applied to speech recognition [73] and turned out to be quite a useful technique. However, I do not expect it to work as well for control tasks as it does for speech recognition, because of an important difference between these tasks. A major task of speech recognition is to find the temporal patterns that already exist in a given sequence of speech phonemes. While learning to control, the agent must look for temporal patterns generated by its own actions. If the actions are generated randomly as it is often the case during early learning, it is unlikely to find sensible temporal patterns within the action sequence so as to improve its action selection policy.

On the other hand, we may greatly improve the performance of the window-Q architecture by using more sophisticated time-delay neural networks (TDNN). The TDNN used here is quite primitive; it only has a fixed number of delays in the input layer. We can have delays in the hidden layer as well [73, 27]. Bodenhausen and Waibel [9] describe a TDNN with adaptive time delays. Using their TDNN, window-Q may be able to determine a proper window size automatically.

7.7 Related Work

In recent years, multilayer neural networks and recurrent networks have emerged and become important components for controlling nonlinear systems with or without hidden states. Figure 7.9 illustrates several control architectures, which can be found in the literature. Some of these architectures use recurrent networks, and some use just feed-forward networks. In principle, all these architectures can be modified to control systems with hidden states by introducing time-delay networks and/or recurrent networks into the architectures.

Note that there are two types of critic in Figure 7.9: *Q-type* and *V-type*. The Q-type critic is an evaluation function of state-action pairs, and therefore is equivalent to the Q-function except that the critic may have multiple outputs; for example, one output for discounted cumulative pain and another output for discounted cumulative pleasure. (The Q-nets in Figure 7.1 can also be modified to have multiple outputs.) The V-type critic is an evaluation function of only states. Temporal difference (TD) methods are often employed to learn both types of critic.

The *adaptive heuristic critic* (AHC) architecture (Figure 7.9.a) was first proposed by Sutton [65]. Each time an action is taken, the action is rewarded or punished based on whether it leads to better or worse results, as measured by the critic. At the same time, the critic is trained using TD methods. This architecture assumes discrete actions. This architecture was discussed and studied in Chapters 3 and 4.

The *back-propagated adaptive critic* (BAC) architecture (Figure 7.9.b) was proposed by Werbos [76, 78]. This architecture assumes the availability of the desired utility at any time. For example, the desired utility for the pole balancing system is 0 all the time; that is, the pole never falls. Under this assumption and the assumption that the critic and the action model have been learned already, the control policy can be trained by back-propagating the difference between the desired utility and actual critic output through the critic to the model and then to the policy network, as if the three networks formed one large feed-forward network. Here the gradients obtained in the policy network indicate how to change the policy so as to maximize the utility. Note that this architecture can handle continuous actions.

The architecture shown in Figure 7.9.c was described by Werbos [77] and Schmidhuber [60]. This architecture assumes the availability of the desired outputs from the system to be controlled. Under this assumption, the errors between actual outputs and desired outputs can be back-propagated through the model to the policy network. The gradients obtained in the

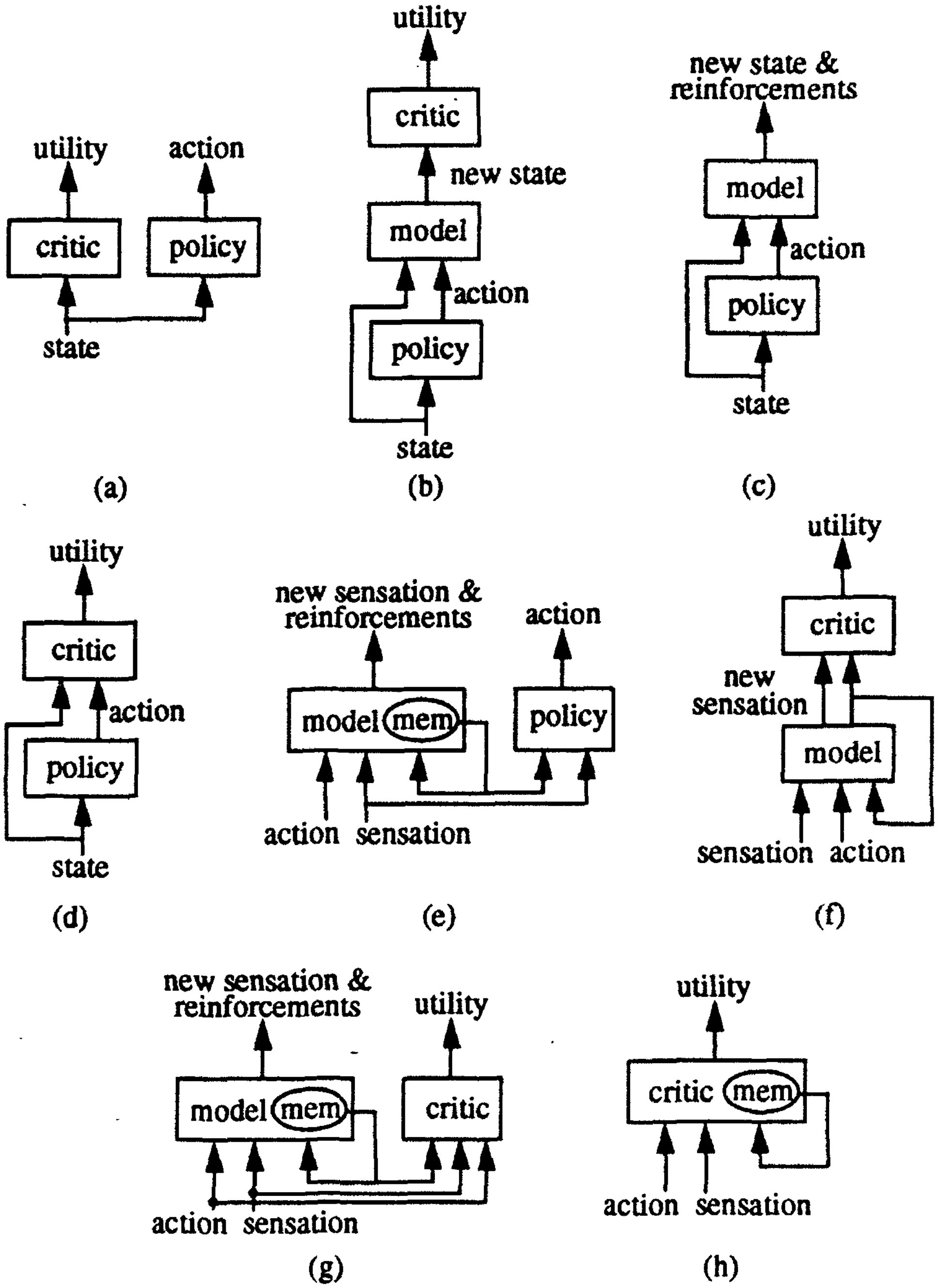


Figure 7.9: Control architectures.

policy network indicate how to change the policy so as to obtain the desired outputs from the system. Note that this architecture can handle continuous actions.

Jordan and Jacobs [28] proposed a control architecture (Figure 7.9.d) somewhat similar to the BAC architecture— instead of using a model and a V-type critic, it uses a Q-type critic. This architecture also assumes the availability of the desired utility, and can handle continuous actions.

The architecture shown in Figure 7.9.e was described by Thrun and Möller [72]. Given a well-trained model, this architecture performs multiple-step look-ahead planning to find out the best action for a given situation. Using the input situation and the action that is found best as a training pattern, the policy network is then trained to mimic the multiple-step planning process in one step. This architecture can handle continuous actions.

The architecture shown in Figure 7.9.f was proposed by Bachrach [6]. Assuming that the action model has been learned already, this architecture trains a V-type critic using TD methods. The control policy is simply to choose the action for which the critic output is maximal. This architecture assumes discrete actions.

My proposed recurrent-model architecture, which is shown in Figure 7.1.c and replicated in Figure 7.9.g, is similar to Bachrach's architecture (Figure 7.9.f). Both assume discrete actions and do not require the availability of the desired utility. Two differences between both are that: (1) mine uses a Q-type critic, while his a V-type critic, and (2) the Q-type critic uses the actual, current sensation as inputs, while the V-type critic uses the predicted, next sensation as inputs. Because correct predictions by the V-type critic strongly relies on correct predictions by the model, we may expect the recurrent-model architecture to outperform Bachrach's architecture, if the learning agent has rich sensations and is unable to learn a good action model.

The recurrent-model architecture is also similar to a control architecture proposed by Chrisman [12]. A main difference is that my model learning is based on gradient descent and his based on maximum likelihood estimation.

All of the architectures discussed in this section so far consist of two or more components. The recurrent-Q architecture (Figure 7.1.b and Figure 7.9.h), on the other hand, consists of only one component, a Q-type critic. The critic is directly used to choose actions, assuming that actions are enumerable and the number of actions is fin:'e. This architecture is interesting; it attempts to solve three difficult problems all by a single component: credit assignment, generalization, and hidden state.

7.8 Summary

This chapter presented three memory architectures for reinforcement learning with hidden states: window-Q, recurrent-Q, and recurrent-model. These architectures are all based on the idea of using history information to discriminate situations that are indistinguishable from immediate sensations. As shown already, these architectures are all capable of learning some tasks involving hidden states. They are also able to handle irrelevant features and small control errors to some degree. This chapter also discussed strengths and weaknesses of these architectures in solving tasks with various problem parameters.

The (good) performance reported in this chapter would not be obtainable without two techniques: experience replay with large λ values and back-propagation through time. The following is a general summary on each of these architectures:

1. For tasks where a small window size N is sufficient to remove hidden states, the window-Q architecture should work well. However, it is unable to represent an optimal control policy if the memory depth of the problem is greater than N.

2. Surprisingly, the recurrent-Q architecture seems to be much more promising than I thought before this study. I (as well as several other researchers) in fact did not expect this architecture to work at all. A potential problem with this architecture, however, is that it attempts to solve three difficult learning problems at the same time and within the same learning component. The three problems are credit assignment, generalization, and hidden state.

3. The recurrent-model architecture deals with the hidden state problem by learning an action model, and handles the credit assignment and generalization problems by learning a feed-forward Q-net. The recurrent-model approach may be quite costly to apply, because model learning often takes a lot more effort than what seems to be necessary. The difficulty of the general problem of model learning is well recognized. There are few methods that are truly successful. For example, the *diversity-based inference procedure* proposed by Rivest and Schapire [57] is restricted to deterministic, discrete domains. Instance-based approaches, such as the work by Moore [47], Atkeson [5], and Christiansen [13], are suitable for nondeterministic, continuous domains, but cannot learn history features. Chrisman [12] studied a model learning method based on maximum likelihood estimation. That method can handle nondeterminism and history features, but does not seem to scale well. I also do not think the architectures proposed here scale well to handle problems with very large memory depths. On the other hand, once a model is learned for a task, it may be re-usable for other tasks.

Finally, as mentioned before, combinations of these architectures are possible and may give better performance than the basic versions. Further study of this remains to be done.

Chapter 8

Integration of Capabilities

In Chapter 6, the hidden state problem was avoided by allowing the simulated robot to access its global X-Y position. In this chapter, the robot is no longer allowed to do that, and therefore faces the hidden state problem. To deal with this problem, the robot combines the hierarchical learning techniques developed in Chapter 6 and the recurrent-Q architecture developed in Chapter 7. This chapter demonstrates the success of this combination.

Moreover, the robot's learned control policies can adapt to certain unexpected environmental changes without additional training— this demonstrates the robustness of the learning techniques developed in this dissertation.

8.1 The Learning Domain

In this chapter, HEROINE (the robot simulator used in Chapters 5 and 6) is once again used as the learning domain. As described before, the simulated robot has 6 primitive actions and several kinds of sensors. The robot's sensors and actuators are not perfect and have errors. Its main task is to learn a control policy for electrically connecting to a battery charger in a three-room environment (see Figure 5.3). The battery recharging task is decomposed into four subtasks: right wall following (WFR), left wall following (WFL), docking (DK), and door passing (DP). Each subtask can be learned separately and the learning result is an elementary behavior for carrying out the subtask.

The specific task studied here is for the robot to learn an optimal control policy that accomplishes the recharging task by choosing among the four elementary behaviors. The state representation for this task includes sonar information (8 real-valued units), room information (3 binary units), and information about collisions, light, and door edges (3 binary units). This representation is in fact similar to what is described in Chapter 6.4.2. The only difference is that the robot is no longer allowed to access its X-Y position. The robot faces a small hidden state problem, because its sensory information is insufficient to make clear distinctions between the two doorways connecting Room C to Rooms A and B.

The experimental setup here is pretty much the same as that described in Chapter 6.6. The learning parameters include:

- discount factor γ (fixed to 0.99),
- recency factor λ (fixed to 0.8),

Other parameters such as learning rate will be given below. In the experiments presented below, the robot replayed approximately 100 lessons after each learning trial.

8.2 Experiment 1: Memoryless Robot

In this experiment, the robot did not use any mechanism to deal with hidden states. In other words, the same learning algorithm, the same exploration strategies, the same network structure, and the same learning parameters as used in Experiment 3 of Chapter 6.6.3 were used here. The learning curve is shown in Figure 8.1. As expected, the robot generally failed to accomplish the recharging task when it started from Room B. For comparison, the learning curve from Experiment 3 of Chapter 6.6.3 is also included in Figure 8.1. Recall that the robot did not have hidden states in that experiment.

8.3 Experiment 2: Robot with Recurrent Q-nets

Chapter 7 discussed three memory architectures for reinforcement learning agents to deal with hidden states: window-Q, recurrent-Q, and recurrent-model. Here I only experimented with the recurrent-Q architecture. The hidden state problem involved in this task is in fact not serious. The history features the robot needs to learn are mainly the features that can help make distinctions between the two doorways. Therefore, it does not seem worthwhile to learn a complicated recurrent model, which will inevitably attempt to model many detailed but unimportant aspects of the environment, such as where obstacles are located. The window-Q architecture was not tried, because it seemed that a large window size would be needed.

The parameter settings were as follows: The learning rate was 0.08, the number of hidden units was 12, and the number of context units was 8. To avoid instability, the experience replay algorithm was used in batch mode; in other words, the recurrent Q-nets were adjusted after an entire lesson was replayed. As we can see from the learning curve (shown in Figure 8.1), the robot based on recurrent-Q did significantly better than the memoryless robot described in Experiment 1. After 150 learning trials, the robot had successfully developed certain history features for uncover hidden states, and its control policy was acceptably good.

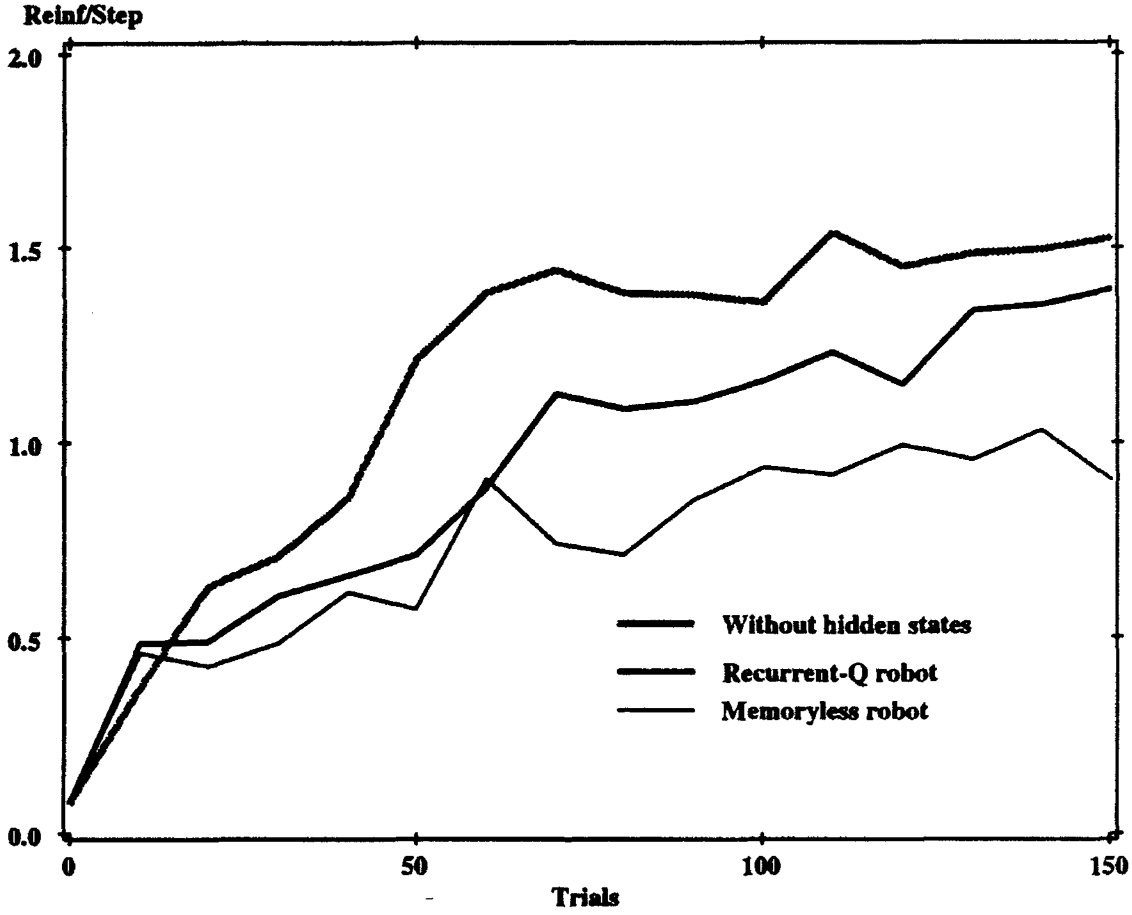


Figure 8.1: The learning curves obtained from the experiments. Each curve shows the robot's mean performance over 7 runs.

8.4 Experiment 3: Unexpected Environmental Changes

In this experiment, I modified the environment of the robot. The modifications (see Figure 8.2) included two new obstacles (one in Room A and one in Room B), movement of two existing obstacles (both in Room A), movement of the battery charger, and movement of both door openings. After the robot had successfully learned a recurrent control policy for battery recharging in the non-modified environment, it was placed in this new environment and learning was turned off. It was found that its control policy could adapt to these environmental changes reasonably well without additional training. Figure 8.2 illustrates three solution paths the robot came up with in the new environment. This result reveals the good generalization achieved by the robot. Being able to succeed in the new environment, the robot must have developed useful high-order features for both the elementary tasks and the high-level task.

8.5 Discussion

Up to this point of the dissertation, I have extended the state of the art of reinforcement learning in several directions, and have applied it to solve a complicated long-range robot navigation problem in a noisy environment with hidden states. The following table summarizes some of the simulation results:

	DK	DP	WFR	WFL	CHG	total
teaching steps	100	100	150	15C	0	500
experimental steps	2000	2000	1500	1500	15000	22000

The second row indicates the number of teaching steps that were provided to the robot for learning a task. The third row indicates the number of action executions that the robot took to learn a good policy for a task. [1] The robot may save several thousand steps if it simultaneously learns both the elementary behaviors and the high-level behavior, as I have shown in Chapter 6.6.5. The robot may save even more steps if more teaching is given. If each action execution takes 4 seconds in the real world, 22000 action executions will take about one day. Considering the complexity of the task, this performance is not bad.

What is important about this dissertation, however, is not in that it has solved a sonar-based navigation problem, but in that it has demonstrated a general learning approach to a wide range of control problems. Indeed, there are other approaches that seem to be more suitable than reinforcement learning for sonar-based navigation. For example, Moravec's sonar mapping technique [49] together with a path planner can enable a robot to navigate efficiently in an unknown environment. But Moravec's technique is specific to sonar-navigation, and cannot be applied to other control problems. In contrast, the reinforcement learning approach makes no assumptions regarding the robot's sensors or its environment, and is a general learning method.

[1] When the robot was allowed to access its X-Y position, it took only 7000 steps to solve the recharging task.

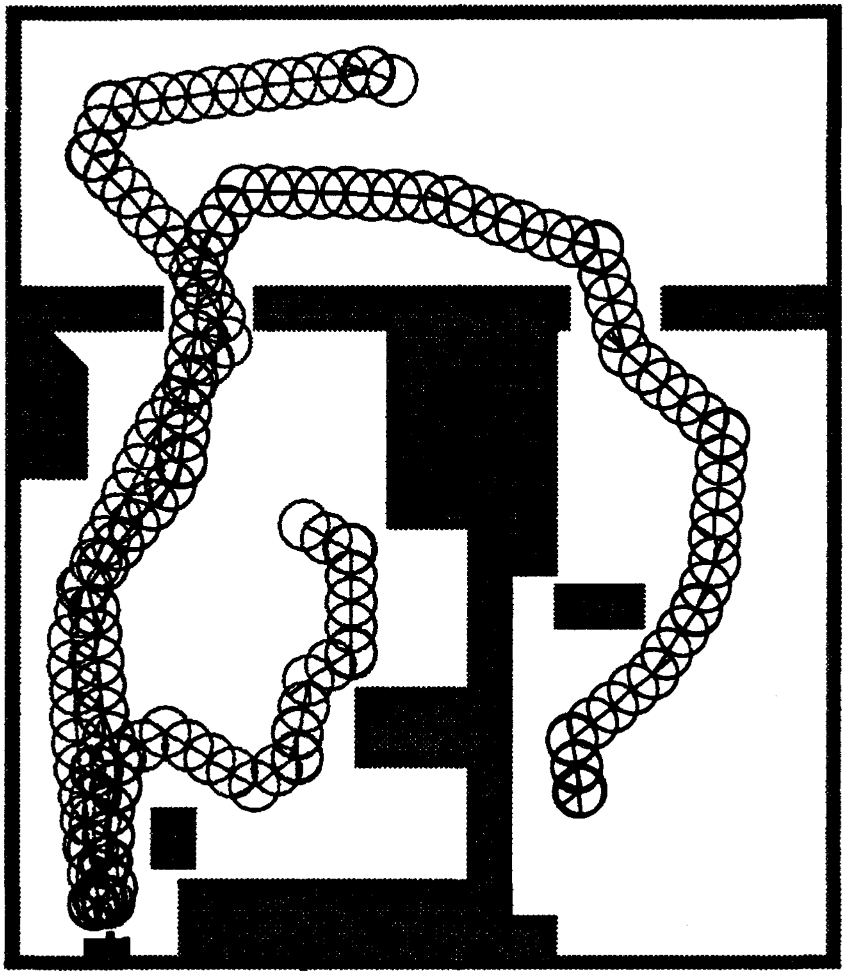


Figure 8.2: Three solution paths found for the recharging task. Recurrent Q-nets were used here to handle hidden states. In spite of many environmental changes, the robot was able to accomplish the task without additional training. This success suggests that the robot must have developed useful high-order features for its tasks.

Chapter 9

Conclusion

Reinforcement learning is a technique to build robot systems with much less involvement of human efforts than many other learning techniques. The main problem, however, is that existing reinforcement learning methods learn very slowly. This dissertation has presented a series of extensions to reinforcement learning, and each of the extension has been shown to result in either noticeable speedup or new capabilities of the learning agents. These extensions together with their simulation results are summarized as follows:

For any learning system to succeed, the ability to generalize is absolutely crucial. In this dissertation, multi-layer neural networks and back-propagation have been used to obtain generalization, and shown to be quite successful. For example, a simulated robot has successfully trained a set of neural networks to approximate a nonlinear evaluation function for a door passing task in a noisy environment. The inputs to the networks included 28 binary data and 24 real-valued sonar data. Given the fact that the back-propagation algorithm was originally designed for being used in a supervised learning manner, its successful applications to self-supervised reinforcement learning are particularly notable.

In this research, neural networks were found to handle control errors and sensing errors quite well. They also achieved good generalization, which can be seen by noting that the control policy reported in Chapter 8.4 could adapt to small environmental changes without additional training. On the other hand, neural networks seemed computationally expensive. They typically require many presentations of training data before they converge, and each presentation may take a lot of time depending on the size of the .ietworks. For example, the networks used in Experiment 1 of Chapter 6.6.1 have 102 input units, 30 hidden units and 1 output unit. For such large networks, each simulation run (consisting of 300 trials) took about 1.5 days to complete on a Sparc Station. Neural networks can be highly parallelized, however.

Generalization can result in significant speedup to reinforcement learning. Another way to obtain significant speedup is to utilize an action model, which can be incrementally built during trial and error. In essence, by action projection with a model, the learner can experience the consequence of an action without running risks in its environment. This dissertation proposed a technique called experience replay. This technique in effect takes advantage of a model, but does not have the difficult problem of building a model, because the model is simply the collection of past experiences.

The experience replay technique is simple and yet effective. For example, for a survival task in a dynamic environment, this technique in one case has resulted in a speedup of 2.5 (i.e., the learning time was reduced to 1/2.5 of the original one; see Chapter 4.5). (More significant speedup is generally expected for more difficult learning problems.) This speedup was obtained from using TD(0) methods. The experience replay technique was further extended to use TD(λ) methods, and has been shown to result in even more speedup. For example, with the use of a proper λ value, a speedup of 4.3 (compared with experience replay with TD(0)) was obtained for the door passing task mentioned above (see Chapter 5.6).

On the other hand, the experience replay technique did not seem to be a very efficient way of reusing experiences. In this research, each experience was typically replayed 60 times in order to obtain good performance. (Recall that non-policy actions were not replayed.) To be efficient, experience replay should be performed selectively. For example, we may estimate the information gain from replaying an experience, and allow only experiences with high information gain to be replayed (for example, see [53]).

Both generalization and experience replay effectively improve learning speed. However, there is a bottleneck that both techniques cannot help break through. The prerequisite of learning a good control policy is that the learning agent has reached its goal state once before. A learning agent starting with zero knowledge will rely on its luck in reaching the goal state for the first time, which may take an arbitrarily long time. Here a teacher can save the agent lots of trial and error by simply giving it a few successful examples. This research has shown that teaching can be naturally integrated into reinforcement learning and result in significant speedup. Take the door passing task for example. The robot obtained a speedup of 14 by taking advantage of 3 teacher-provided lessons, and a speedup of 28 with 10 lessons (see Chapter 5.6).

Another way to get great speedup is hierarchical learning. A model of hierarchical reinforcement learning has been proposed. A complexity analysis for discrete and deterministic domains has shown two main results. First, the complexity of reinforcement learning can be reduced from $O(n^2)$ to $O(n \cdot \log n)$ (n is the number of states), assuming that an optimal hierarchy is provided by a designer in advance (Chapter 6.2.4). Second, once a learning problem has been solved, solving a second problem becomes easier, if both problems share the same subproblems.

The idea of hierarchical reinforcement learning was tested in a robot simulator. By the teaching technique mentioned above, the simulated robot at first learned a set of elementary behaviors (i.e., control policies) such as wall following and door passing. Next, the robot learned a high-level control policy, which coordinated the elementary behaviors to accomplish a long-range navigation task. Without hierarchical learning, the robot could not learn a good policy for this navigation task within a reasonable time.

This dissertation also extended reinforcement learning to allow agents to learn with incomplete state information. Three architectures were presented: window-Q, recurrent-model, and recurrent-Q. They are all based on the idea of having a memory of the past, and thus allow agents to learn history-sensitive control policies. Window-Q allows agents to have direct access to a sliding window of its past. It is simple, but may not work effectively when the window size must be large. Based on recurrent neural networks, the recurrent-model architecture trains a recurrent action model, which will then help determine the actual world state. It was found that an agent might learn a perfect control policy without having learned a perfect model. The recurrent-Q architecture trains recurrent networks to approximate a history-sensitive evaluation function. Many researchers (including myself) did not expect recurrent-Q to work at all, but it turned out to work effectively for several problems.

Finally, Q-learning, neural networks, experience replay, teaching, hierarchical learning and the recurrent-Q architecture were combined to solve a complex robot problem in the face of a large and continuous state space, sparse reward signals, control errors, sensing errors and hidden states. All these results support the thesis of this dissertation: **We can scale up reinforcement learning in various ways, making it possible to build reinforcement learning agents that can effectively acquire complex control policies for realistic robot tasks.**

9.1 Contributions

This research has made a number of contributions to robot learning and learning from delayed rewards:

Successful integration of temporal difference and back-propagation: Temporal difference methods are often used to solve the temporal credit assignment problem in reinforcement learning. Neural networks and back-propagation are often used to solve the generalization problem. Previous attempts to integrating temporal difference and back-propagation, however, have not always been successful. In particular, most researchers did not believe that a straightforward integration of temporal difference and recurrent neural networks could ever learn history-sensitive evaluation functions in non-Markovian domains. This research has demonstrated the success of this integration in solving several nontrivial learning problems—some of them were non-Markovian.

Experience replay algorithms: The novel experience replay algorithms are simple and yet effective techniques for solving the temporal credit assignment problem. They use backward replay of experiences and are efficient implementation of TD(λ) methods for general λ.

Teaching: The novel teaching technique described in this dissertation allows a robot to learn control policies from a human teacher as well as from experimentation with its environment. This technique is appealing, not only because it is an effective way to speed up learning, but also because it is simple. What a human teacher needs to do is simply taking control over the robot and showing to the robot solutions to some sample task instances. Furthermore, the teacher is not required to demonstrate only the optimal action in order for the robot to learn an optimal control policy— it can learn from both positive and negative examples.

Hierarchical reinforcement learning: A model and an analysis of hierarchic al reinforcement learning have been presented. The analysis has shown that a great reduction in learning complexity can be obtained by the use of abstraction. The analysis is restricted to discrete and deterministic domains, and is based on a few assumptions; nevertheless, it sheds light on how hierarchical reinforcement learning can be done and how learning complexity can be reduced.

Hierarchical learning of robot behaviors: In addition to theoretical results, hierarchical reinforcement learning has been tested in a simulated mobile-robot domain. The robot was able to learn both the elementary behaviors and the coordination of them.

Learning with hidden states: Three memory architectures have been presented to allow reinforcement learning in the presence of hidden states. Two of them are novel. These architectures were empirically shown to be useful. Their relative strengths and weakness were also evaluated and discussed.

Integration: Previous studies of reinforcement learning have mostly been about solving simple learning problems. Previous efforts in scaling reinforcement learning have mostly focused on a single topic, for example, either on generalization or on utilizing models. This dissertation proposed a number of extensions to reinforcement learning, and furthermore integrated those extensions to solve a robot learning task, which is complex and physically realistic.

9.2 Future Work

Chapter 1 pointed out a list of issues involved in reinforcement learning. Some of them have been addressed in this dissertation, and some have not. For those studied and those not studied here, much work remains to be done. In particular, the following are some interesting areas to be pursued in the future:

- There has been significant progress in neural network learning. Improved training algorithms and new network structures have been proposed. Reinforcement learning may directly benefit from these new technologies. It remains to be investigated, however. It would also be interesting to explore alternative generalization approaches, such as those mentioned in Chapter 2.3.

- It seems that the learning techniques described here can be extended to handle actions with real-valued parameters (e.g., "*move x inches*"). For example, the Q-function can be represented by a network, whose input units encode both the state inputs and action parameters. Given such a Q-net, one can use error back-propagation to find the best action parame' . Initially a random parameter value is picked, and the network is evaluated to determine the utility (say, u) of the action with this parameter value. Suppose the maximum possible utility value is 1. We set the error of the network output to $(1 - u)$, and propagate this error back to the input units. Here the error in the input unit that encodes the action parameter indicates how we can change the parameter value so as to maximize the utility. We change the parameter value accordingly, and do the above once again with this new value as the start point. This process can be repeated many times until the error in the input unit becomes sufficiently small.

- We like robots to be able to accelerate learning by taking advantage of human inputs during learning. This dissertation presented one way to do it (i.e., teaching). Other approaches remain to be found.
- To prove the usefulness of the techniques described here, they must be tested in many domains. In particular, it would be interesting to see their applications to real-world robot problems.
- This research assumes that a correct behavior hierarchy is given by human designers. A great challenge is automatic discovery of a good hierarchy.
- It is possible that some combination of the three memory architectures may result in better performance than each individual one.
- Effective exploration is an important issue for learning agents. In particular, how can a learning agent explore efficiently in a continuous state space? Even more challenging, how can it explore an environment with hidden states?
- It is important that a robot intelligently manage its expensive sensory resources. There has been some work on applying reinforcement learning to solve this active perception problem. It is generally quite an open area, however.

9.3 Closing

Robot learning is generally a hard problem. Some robot learning problems, I believe, can be readily solved by the reinforcement learning techniques described in this dissertation. But for complicated real-world problems, reinforcement learning in its current stage is still not powerful enough. Nevertheless, reinforcement learning looks promising as an approach to robot learning, because it is not built on top of strong assumptions, such as deterministic worlds, availability of perfect (or sufficiently good) models, etc. Much progress in scaling reinforcement learning has been made in recent years. Many problems have been identified, and some possible solutions have been proposed and pursued. It would not be surprising to see in the near future that reinforcement learning is embedded in a robot system to continuously optimize and adapt the robot's skills of performing routine tasks.

Appendix A

Algorithms Used in Chapter 4

A.1 Algorithm for Choosing Enemy Actions

Each enemy in the dynamic environment behaves randomly. On each step, 20% of the time the enemy will not move, and 80% of the time it will choose one of the four actions, A_0 (east), A_1 (south), A_2 (west), and A_3 (north), according to the following probability distribution:

$$prob(A_i) = P_i/(P_0 + P_1 + P_2 + P_3)$$

where

$$P_i = \begin{cases} 0 & \text{if } A_i \text{ will result in a collision with obstacles} \\ exp(0.33 \cdot W(angle) \cdot T(dist)) & \text{otherwise} \end{cases}$$

$angle$ = angle between the direction of action A_i and
the direction from the enemy to the agent

$dist$ = distance between the enemy and the agent

$$W(angle) = (180 - |angle|)/180$$

$$T(dist) = \begin{cases} 15 - dist & \text{if } dist \leq 4 \\ 9 - dist/2 & \text{if } dist \leq 15 \\ 1 & \text{otherwise} \end{cases}$$

A.2 Algorithm for Choosing Lessons for Replay

The agents which replay experiences keep only the most recent 100 lessons in memory. Lessons are randomly chosen for replay; recent lessons are exponentially more likely to be chosen. The algorithm for choosing a lesson from memory is shown below.

Input: a sequence of lessons, $L_0, L_1, ..., L_{n-1}$, where L_{n-1} is the latest one.

Output: an integer $k, 0 \leq k < n$

Algorithm:

```
1. $w \leftarrow Min(3, 1 + 0.02 \cdot n)$
2. $r \leftarrow$ a random number between 0 and 1
3. $k \leftarrow n \cdot log(1 + r \cdot (e^w - 1))/w$
```

Appendix B

Parameter Settings Used in Chapter 7

The parameters used by the experiments in Chapter 7 included:

- discount factor γ (fixed to be 0.9),
- recency factor λ,
- range of the random initial weights of networks (fixed to be 0.5),
- momentum factor (fixed to be 0 for perceptrons and 0.9 for multilayer networks),
- window size (N),
- learning rate for Q-nets (η_q),
- learning rate for action models (η_m),
- number of hidden units for Q-nets (H_q),
- number of hidden units for action models (H_m),
- number of context units for Q-nets (C_q),
- number of context units for action models (C_m),
- temperature for controlling randomness of action selection (T), and
- probability threshold for determining policy actions for experience replay (P_l).

The parameter settings used are shown in Table B.1. Note that $H_q = 0$ means that the Q-nets are single-layer perceptrons. The value of P_l (see Chapter 3.5 or 5.3.5) was dynamically set to be between 0.01 and 0.00001 for the cup collection tasks and between 0.0001 and 0.0000001 for the pole balancing task, depending on the recency of the experience to be replayed.

Table B.1: Parameter Settings

	Task 1	Task 2	Task 3	Task 4
window-Q	$N = 5$ $\lambda = 0.8$ $H_q = 0$ $\eta_q = 0.02$ $1/T = 10 \rightarrow 30$	same as Task 1	same as Task 1 except $\eta_q = 0.01$	$N = 1$ $\lambda = 0.7$ $H_q = 6$ $\eta_q = 0.25$ $1/T = 20 \rightarrow 50$
recurrent-Q	$\lambda = 0.8$ $C_q = 8$ $H_q = 14$ $\eta_q = 0.04$ $1/T = 5 \rightarrow 30$	same as Task 1	$\lambda = 0.8$ $C_q = 12$ $H_q = 20$ $\eta_q = 0.02$ $1/T = 5 \rightarrow 30$	$\lambda = 0.7$ $C_q = 2$ $H_q = 10$ $\eta_q = 0.02$ $1/T = 20 \rightarrow 50$
recurrent-model	$\lambda = 0.8$ $C_m = 16$ $H_m = 24$ $\eta_m = 0.02$ $H_q = 10$ $\eta_q = 0.2$ $1/T = 5 \rightarrow 30$	$\lambda = 0.8$ $C_m = 18$ $H_m = 26$ $\eta_m = 0.02$ $H_q = 11$ $\eta_q = 0.2$ $1/T = 5 \rightarrow 30$	$\lambda = 0.8$ $C_m = 20$ $H_m = 30$ $\eta_m = 0.02$ $H_q = 12$ $\eta_q = 0.1$ $1/T = 5 \rightarrow 30$	$\lambda = 0.7$ $C_m = 4$ $H_m = 6$ $\eta_m = 0.005$ $H_q = 6$ $\eta_q = 0.5$ $1/T = 15 \rightarrow 50$

Appendix C

The Pole Balancing Problem

The dynamics of the cart-pole system (Figure C.1) are given by the following equations of motion [65] [3]:

$$
\begin{aligned}
x(t+1) &= x(t) + \tau\dot{x}(t) \\
\dot{x}(t+1) &= \dot{x}(t) + \tau\ddot{x}(t) \\
\theta(t+1) &= \theta(t) + \tau\dot{\theta}(t) \\
\dot{\theta}(t+1) &= \dot{\theta}(t) + \tau\ddot{\theta}(t) \\
\ddot{x}(t) &= \frac{F(t) + m_p l\left[\dot{\theta}^2(t)\sin\theta(t) - \ddot{\theta}(t)\cos\theta(t)\right] - \mu_c sgn(\dot{x}(t))}{m_c + m_p} \\
\ddot{\theta}(t) &= \frac{g\sin\theta(t) + \cos\theta(t)\left[\frac{-F(t) - m_p l\dot{\theta}^2(t)\sin\theta(t) + \mu_c sgn(\dot{x}(t))}{m_c+m_p}\right] - \frac{\mu_p\dot{\theta}(t)}{m_p l}}{l\left[\frac{4}{3} - \frac{m_p\cos^2\theta(t)}{m_c+m_p}\right]}
\end{aligned}
$$

where

g	=	$9.8 m/s^2$	=	acceleration due to gravity,
m_c	=	1.0 kg	=	mass of cart,
m_p	=	0.1 kg	=	mass of pole,
l	=	0.5 m	=	half pole length,
μ_c	=	0.0005	=	coefficient of friction of cart on track,
μ_p	=	0.000002	=	coefficient of friction of pole on cart,
$F(t)$	=	±10.0 newtons	=	force applied to cart's center of mass at time t,
τ	=	0.02 s	=	time in seconds corresponding to one simulation step,

$$
sgn(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases}
$$

The reinforcement function is defined as:

$$
r(t) = \begin{cases} -1 & \text{if } |\theta(t)| > 0.21 \text{ radian or } |x(t)| > 2.4 \text{ m}, \\ 0 & \text{otherwise.} \end{cases}
$$

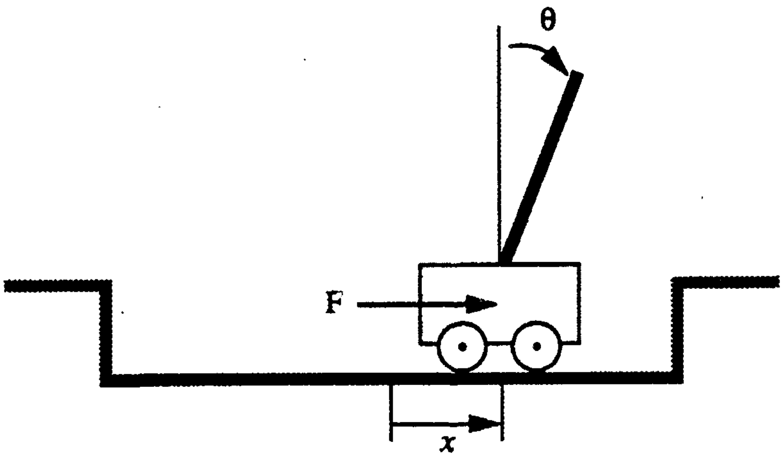


Figure C.1: The pole balancing problem.

Bibliography

[1] D.H. Ackley and M.L. Littman. Learning from natural selection in an artificial environment. In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 189–193. Lawrence Erlbaum Associates: Hillsdale, New Jersey, 1990.

[2] J.S. Albus. *Brains, Behaviour and Robotics.* BYTE Books, McGraw-Hill, 1981.

[3] C.W. Anderson. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114, Ann Arbor, Michigan, 1987. Morgan Kaufmann.

[4] C.G. Atkeson. Learning arm kinematics and dynamics. *Annual Review of Neuroscience*, 12:157–183, 1989.

[5] C.G. Atkeson. Using locally weighted regression for robot learning. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 958–963, Sacramento, California, 1991.

[6] J.R. Bachrach. *Connectionist Modeling and Control of Finite State Environments.* PhD thesis, University of Massachusetts, Department of Computer and Information Sciences, 1992.

[7] A.G. Barto, S.J. Bradtke, and S.P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, Computer Science Department, University of Massachusetts, 1991.

[8] A.G. Barto, R.S. Sutton, and C.J.C.H. Watkins. Learning and sequential decision making. In *Learning and Computational Neuroscience.* The MIT Press, 1990.

[9] U. Bodenhausen and A. Waibel. The Tempo 2 algorithm: Adjusting time-delays by supervised learning. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 3.* Morgan Kaufmann, 1991.

[10] D. Chapman. *Vision, Instruction, and Action.* PhD thesis, MIT, Artificial Intelligence Laboratory, 1990. Technical Report 1204.

[11] D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth Internation Joint Conference on Artificial Intelligence*, pages 726–731, Sydney, Australia, 1991.

[12] L Chrisman. Reinforcement learning with perceptual aliasing: The predictive distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183–188. AAAI Press/The MIT Press, 1992.

[13] A.D. Christiansen. *Automatic Acquisition of Task Theories for Robotic Manipulation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1992. Technical Report CMU-CS-92-111.

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

[15] M. Dorigo and U. Schnepf. Genetics-based machine learning and behavior-based robotics: A new synthesis. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6), 1992.

[16] J.L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[17] S.E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, 1988.

[18] S.E. Fahlman. The recurrent cascade-correlation architecture. Technical Report CMU-CS-91-100, School of Computer Science, Carnegie Mellon University, 1991.

[19] 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. Morgan Kaufmann, 1990.

[20] Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1992. Technical Report CMU-CS-92-175.

[21] J.J. Grefenstette. Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning*, 3:225–245, 1988.

[22] J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5:355–382, 1990.

[23] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, 1991.

[24] G.E. Hinton, J.L. McClelland, and D.E. Rumelhart. Distributed representations. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter 3. The MIT press, 1986.

[25] J.H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach, Volume 2*, pages 593–623. Morgan Kaufmann, 1986.

[26] R.A. Howard. *Dynamic Programming and Markov Processes*. Wiley, New York, 1960.

[27] A.N. Jain. *A Connectionist Learning Architecture for Parsing Spoken Language*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Technical Report CMU-CS-91-208.

[28] 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*, pages 324–331. Morgan Kaufmann, 1990.

[29] L.P. Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, Department of Computer Science, 1990.

[30] C.A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Technical Report CMU-CS-91-120.

[31] S. Koenig. The complexity of real-time search. Technical Report CMU-CS-92-145, School of Computer Science, Carnegie Mellon University, 1992.

[32] R.E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[33] M.A. Lewis, A.H. Fagg, and A. Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2618–2623, Nice, France, 1992.

[34] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786. AAAI Press/The MIT Press, 1991.

[35] Long-Ji Lin. Self-improving reactive agents: case studies of reinforcement learning frameworks. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 297–305, Paris, France, 1991. The MIT Press.

[36] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[37] Long-Ji Lin and T.M. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, School of Computer Science, Carnegie Mellon University, 1992.

[38] Long-Ji Lin, T.M. Mitchell, A. Phillips, and R. Simmons. A case study in autonomous robot behavior. Technical Report CMU-RI-89-1, Robotics Institute, Carnegie Mellon University, 1989.

[39] Long-Ji Lin, R. Simmons, and C. Fedor. Experience with a task control architecture for mobile robots. Technical Report CMU-RI-89-29, Robotics Institute, Carnegie Mellon University, 1989.

[40] P. Maes and R. Brooks. Learning to coordinate behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 796–802. AAAI Press/The MIT Press, 1990.

[41] S. Mahadevan and J. Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 328–332, Evanston, Illinois, 1991. Morgan Kaufmann.

[42] R.A. McCallum. Using transitional proximity for faster reinforcement learning. In *Proceedings of the Ninth International Conference on Machine Learning*, England, 1992. Morgan Kaufmann.

[43] J. del R. Millán and C. Torras. A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8:363–395, 1992.

[44] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[45] T.M. Mitchell. Becoming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1051–1059. AAAI Press/The MIT Press, 1990.

[46] T.M. Mitchell and S.B. Thrun. Explanation-based neural network learning for robot control. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, 1993. (To appear).

[47] A.W. Moore. *Efficient Memory-Based Learning for Robot Control*. PhD thesis, University of Cambridge, Computer Laboratory, 1990. Technical Report No. 209.

[48] A.W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 333–337, Evanston, Illinois, 1991. Morgan Kaufmann.

[49] H.P. Moravec and A.E. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 116–121, Washington D.C., 1985. IEEE Computer Society.

[50] M.C. Mozer. Rambot: A connectionist expert system that learns by example. Technical Report Institute for Cognitive Science Report 8610, University of California at San Diego, 1986.

[51] M.C Mozer and J.R. Bachrach. SLUG: A connectionist architecture for inferring the structure of finite-state environments. *Machine Learning*, 7:139–160, 1991.

[52] N. Nilsson. An approach to learning sequences of actions: Combining delayed reinforcement and input generalization. Unpublished draft notes, 1991.

[53] Jing Peng and R.J. Williams. Efficient learning and planning within the Dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. The MIT Press, 1993. (To appear).

[54] D.A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. Technical Report CMU-CS-89-107, Carnegie Mellon University, 1989.

[55] D.A. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1992. Technical Report CMU-CS-92-115.

[56] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[57] R.L. Rivest and R.E. Schapire. Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, 1987.

[58] 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*, volume 1, chapter 8. The MIT press, 1986.

[59] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959. Reprinted in E.A. Feigenbaum and J. Feldman (Eds.) *Computers and Thought*, 71-105, New York: McGraw-Hill, 1963.

[60] J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 3*, pages 500–506. Morgan Kaufmann, 1991.

[61] S.P. Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Conference on Machine Learning*, England, 1992. Morgan Kaufmann.

[62] S.P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[63] Singh. S.P. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207. AAAI Press/The MIT Press, 1992.

[64] W.S. Stornetta and B.A. Huberman. An improved three-layer back-propagation algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 637–644, San Diego, California, 1987.

[65] R.S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Department of Computer and Information Science, 1984.

[66] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[67] R.S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, Austin, Texas, 1990. Morgan Kaufmann.

[68] Ming Tan. *Cost-Sensitive Robot Learning*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Technical Report CMU-CS-91-134.

[69] Ming Tan. Learning a cost-sensitive internal representation for reinforcement learning. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 358–362, Evanston, Illinois, 1991. Morgan Kaufmann.

[70] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[71] S.B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, 1992.

[72] S.B. Thrun, K. Möller, and A. Linden. Planning with an adaptive world model. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991.

[73] A. Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39–46, 1989.

[74] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.

[75] C.J.C.H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 1992.

[76] 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*, 17:7–20, 1987.

[77] P.J. Werbos. Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[78] P.J. Werbos. A menu of designs for reinforcement learning over time. In *Neural Networks for Control*. The MIT Press, 1990.

[79] S.D. Whitehead. Complexity and cooperation in Q-learning. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 363–367, Evanston, Illinois, 1991. Morgan Kaufmann.

[80] S.D. Whitehead and D.H. Ballard. A role for anticipation in reactive systems that learn. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 354–357, Ithaca, New York, 1989. Morgan Kaufmann.

[81] S.D. Whitehead and D.H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7:45–83, 1991.

[82] B. Widrow and M.E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record, Part 4*, pages 96–104. New York: IRE, 1960.

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

[84] R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. Technical Report Institute for Cognitive Science Report 8805, University of California at San Diego, 1988.