Explorations in Efficient Reinforcement Learning

ACADEMISCH PROEFSCHRIFT

Ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van Rector Magnificus
Prof. dr. J.J.M. Franse
ten overstaan van een door het
college voor promoties ingestelde commissie
in het openbaar te verdedigen
in de Aula der Universiteit
op woensdag 17 februari 1999 te 13.00 uur

door
Marco Wiering
geboren te Schagen

Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde
Universiteit van Amsterdam

Cover: design and photography by Marco Wiering.

The work described in this thesis was performed at IDSIA: Istituto Dalle Molle di Studi sull'Intelligenza Artificiale in Lugano, Switzerland. It was made possible due to funding of IDSIA, and was supported in part by the Swiss National Fund (SNF) grant 2100-49'144.96 “Long Short-Term Memory”.

The more a man knows about himself
in relation to every kind of experience,
the greater his chance of suddenly,
one fine morning, realizing who in fact he is
or rather Who in Fact "he" Is

Aldous Huxley in *Island*

***** For You *****

Contents

1 Introduction **1**
1.1 Reinforcement Learning 4
1.2 Reinforcement Learning with Incomplete Information 7
1.3 Current Problems of RL 9
1.4 Goals of the Thesis 10
1.5 Outline and Principal Contributions 11

2 Markov Decision Processes **13**
2.1 Markov Decision Processes 13
2.1.1 Value Functions 14
2.1.2 Finite Horizon Problems 14
2.1.3 Infinite Horizon Problems 15
2.1.4 Action Evaluation Functions 17
2.1.5 Contraction 17
2.2 Dynamic Programming 18
2.2.1 Policy Iteration 18
2.2.2 Value Iteration 19
2.2.3 Linear Programming 20
2.3 Experiments 20
2.3.1 Description of the Maze Task 21
2.3.2 Scaling up Dynamic Programming 22
2.4 More Difficult Problems 25
2.4.1 Continuous State Spaces 25
2.4.2 Curse of Dimensionality 26
2.5 Conclusion 26

3 Reinforcement Learning **27**
3.1 Principles of Algorithms 28
3.1.1 Delayed Reward and Credit Assignment Problem 29
3.1.2 Markov Property 29
3.2 TD(λ) Learning 30
3.2.1 Temporal Difference Learning 31
3.2.2 Replacing Traces 34
3.3 Q-Learning 36
3.4 Q(λ)-learning 36
3.5 Fast Q(λ)-learning 38

3.6 Team Q(λ) 43
3.6.1 Extending the Eligibility Trace Framework 43
3.6.2 Combining Eligibility Traces 44
3.7 Experiments 47
3.7.1 Evaluating Fast Online Q(λ) 48
3.7.2 Experiments with Accumulating and Replacing Traces 50
3.7.3 Multi-agent Experiments 52
3.8 Conclusion 53

4 Learning World Models 55
4.1 Extracting a Model 56
4.2 Model-Based Q-learning 57
4.3 Prioritized Sweeping 59
4.4 Experiments 61
4.4.1 Comparison between Model-based and Model-free RL 62
4.4.2 Prioritized Sweeping: Sensitivity Analysis 66
4.4.3 Comparison between PS Methods 67
4.5 Discussion 68
4.6 Conclusion 69

5 Exploration 71
5.1 Undirected Exploration 72
5.1.1 Max-random Exploration Rule 72
5.1.2 Boltzmann Exploration Rule 73
5.1.3 Max-Boltzmann Exploration Rule 73
5.1.4 Initialize High Exploration 74
5.2 Directed Exploration 74
5.2.1 Reward Function 1: Frequency Based 75
5.2.2 Reward Function 2: Recency Based 75
5.2.3 Reward Function 3: Error Based 76
5.2.4 False Exploration Reward Rules 76
5.3 Learning Exploration Models 77
5.4 Model-Based Interval Estimation 78
5.5 Experiments 79
5.5.1 Exploration with Prioritized Sweeping 79
5.5.2 Exploitation/Exploration 81
5.5.3 Experiments with Suboptimal Goals 83
5.6 Discussion 85

6 Partially Observable MDPs 87
6.1 Optimal Algorithms 89
6.1.1 POMDP Specification 89
6.1.2 Belief States 89
6.1.3 Computing an Optimal Policy 90
6.2 HQ-learning 94
6.2.1 Memory in HQ 94
6.2.2 Learning Rules 95

6.2.3 Experiments . . . 99
6.2.4 Discussion . . . 107
6.3 Related Work . . . 108
6.4 Conclusion . . . 113

7 Function Approximation for RL 115
7.1 Function Approximation . . . 115
7.1.1 Linear Networks . . . 116
7.1.2 Local Function Approximators . . . 118
7.1.3 CMACs . . . 122
7.2 Function Approximation for Direct RL . . . 124
7.2.1 Extending Q(λ) to function approximators . . . 126
7.2.2 Notes on combining RL with FAs . . . 128
7.3 World Modeling with Function Approximators . . . 129
7.3.1 Linear Models . . . 130
7.3.2 Neural Gas Models . . . 132
7.3.3 CMAC Models . . . 133
7.4 A Soccer Case Study . . . 135
7.4.1 Soccer . . . 135
7.4.2 The Soccer Simulator . . . 136
7.4.3 Comparison 1: Q-lin, Q-gas and PIPE . . . 138
7.4.4 Comparison 2: CMACs vs PIPE . . . 142
7.4.5 Discussion . . . 147
7.5 Previous Work . . . 149
7.6 Conclusion . . . 151

8 Conclusion 153
8.1 Contributions . . . 153
8.1.1 Exact Algorithms for Markov Decision Problems . . . 153
8.1.2 Reinforcement Learning . . . 153
8.1.3 Model-based RL . . . 155
8.1.4 Exploration . . . 156
8.1.5 POMDPs . . . 157
8.1.6 Function Approximation . . . 158
8.2 Final Remarks . . . 158

A Markov Processes 161
A.1 Markov Property . . . 162
A.2 Generating a State Sequence . . . 162
A.3 State Occupancy Probabilities . . . 162
A.4 Stationary Distribution . . . 164
A.5 Properties of Markov Chains . . . 164
A.6 Counting Visits . . . 165
A.7 Examples . . . 165
A.8 Markov Order . . . 166
A.9 Higher Order Transition Functions . . . 167
A.10 Examples . . . 168

B Learning rate Annealing **171**

C Team Q(λ) Algorithm **173**

D PS for Function Approximators **177**
D.1 PS for Neural Gas 177
D.2 PS for CMACs 178
D.3 Non-Pessimistic Value Functions 178

Bibliography **181**

Index **196**

Summary **199**

Samenvatting **203**

Acknowledgments **208**

This thesis is based on the following publications:

Chapter 3

- M. A. Wiering and J. H. Schmidhuber, *Fast Online Q(λ)*, Machine Learning, ??, 1998.
- M. A. Wiering and J. H. Schmidhuber, *Speeding up Q(λ)-learning*, Proceedings of the Tenth European Conference on Machine Learning (ECML'98), 1998.

Chapter 5

- M. A. Wiering and J. H. Schmidhuber, *Efficient Model-Based Exploration*, Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB'98): From Animals to Animats, 223-228, J. A. Meyer and S. W. Wilson (eds.), MIT Press/ Bradford Books, 1998.

Chapter 6

- M. A. Wiering and J. H. Schmidhuber, *HQ-learning*, Adaptive Behavior 6(2), 219-246, 1997.

Chapter 7

- R. P. Sałustowicz, M. A. Wiering and J. H. Schmidhuber, *Learning Team Strategies: Soccer Case Studies*, Machine Learning, 33, (2/3), ???, M. Huhns and G. Weiss (eds.), 1998.
- R. P. Sałustowicz, M. A. Wiering, and J. H. Schmidhuber, *Evolving soccer strategies*, In Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97), pages 502–506. Springer-Verlag Singapore, 1997.
- M. A. Wiering, R. P. Sałustowicz, and J. H. Schmidhuber, *CMAC Models Learn to Play Soccer*, In Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN'98), L. Niklasson, M. Bodén and T. Ziemke (eds.), 443-448, 1998.

Other publications:

- M. A. Wiering and J. H. Schmidhuber, *Solving POMDPs with Levin search and EIRA*, Machine Learning: Proceedings of the Thirteenth International Conference, L. Saitta (ed.), pages 534-542. Morgan Kaufmann Publishers, San Francisco, CA, 1996.
- J. H. Schmidhuber, J. Zhao and M. A. Wiering, *Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement*, Machine Learning, 28, 105-130, 1997.
- M. A. Wiering and J. H. Schmidhuber, *Learning exploration policies with models*, In Conference on automated learning and discovery (CONALD'98): Robot Exploration and Learning, Carnegie Mellon University, Pittsburgh, 1998.

Chapter 1

Introduction

Suppose we want to use an intelligent *agent* (computer program or robot) for performing tasks for us, but we cannot or do not want to specify the precise task-operations. E.g. we may want to use a team of intelligent agents which can play soccer together. It is very complex to write a program which outputs the best movement for each agent in each situation. Rather, we want to let the agents play together and create the possibility for them to *learn* to improve their playing abilities. Learning could be done by trying out different soccer strategies and observing which strategy leads to the best game results.

Most real world problems require *sequential decision making*: the agent selects and executes an action and this causes some change in the state of the world. Given the new world state the agent selects a new action and so on. Examples of such problems are driving a car, cleaning an office, and playing soccer. In this thesis we are interested in developing intelligent agents which can learn to solve such tasks. There are two optimization methods which enable an agent to learn from the results of its interaction with the world in order to maximize its performance over time: the one we will use and describe in this thesis is *reinforcement learning* (RL), another possibility is to use evolutionary methods such as genetic algorithms (Holland, 1975).

An agent uses a *policy* for selecting actions. RL methods improve their policy by applying it in their environment and by monitoring and learning from the resulting experiences (experimental data). To learn as much as possible, it is important that the agent explores its environment. An exploring agent tries different actions even when it believes that one particular action is best. Without such *exploration*, the agent may always see the same world states and thus it cannot learn about new, possibly highly rewarding, situations.

Perception of the environment. An intelligent agent is situated in a particular environment and is equipped with sensors through which it receives inputs to describe the state of the environment. The world (environmental) state at a given moment is an exact description of the environment, including all objects present in the environment and their parameters such as position, rotation etc. The agent's inputs do not always describe the complete state of the environment, but may give a partial and noisy description. Figure 1.1 shows the Chinese wall with three doors in it. The agent sees a particular door, but since all doors look alike it cannot use its input to know exactly where it is.

Using these inputs, the agent selects and executes an action which affects the environment in some way. Take as an example the agent before the chinese wall. It can stand still, go forward, rotate, and open the door. If the agent opens the door, there is a clear impact on

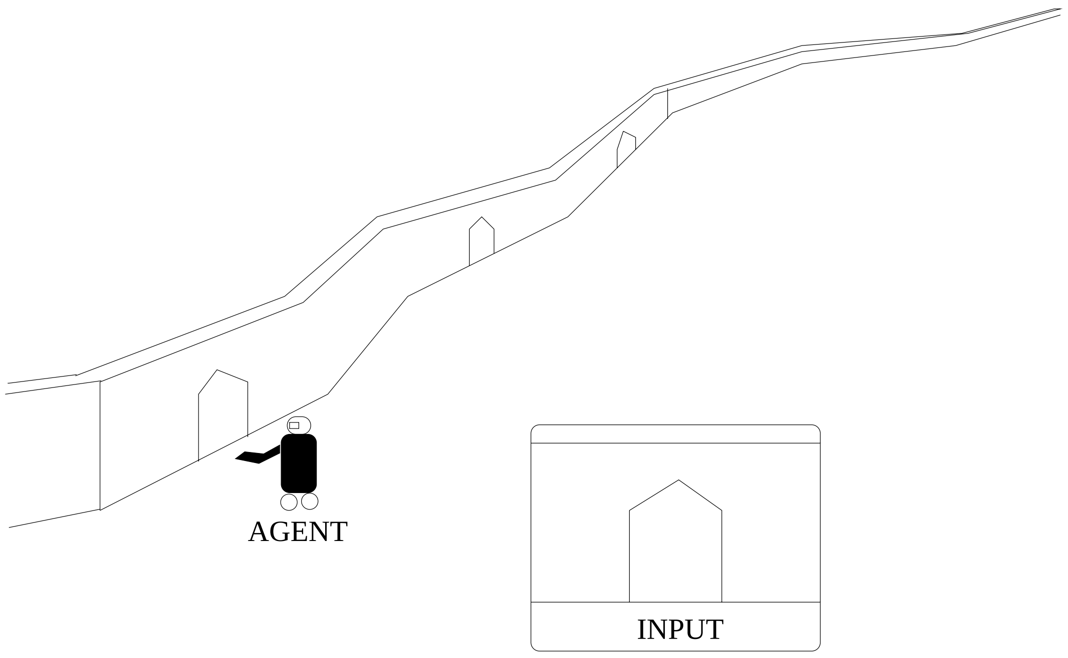


Figure 1.1: *An agent who is unable to distinct multiple doors in front of the chinese wall using its current sensory input only.*

the environment. But also when the agent goes forward or rotates, the environment changes since it includes the agent. The agent's action selection policy is determined by a function which maps sensory inputs to actions. This function forms the core of the agent's behavior and therefore is our main interest.

Modeling and control. Autonomous agents can be used for different goals. The goal on which we will focus in this thesis, is to learn to *control* an agent. Here we are interested in learning an action selection policy which solves a specific problem, where we use the term learning for adapting what has been experienced. For many real world problems it is necessary that we have an adaptive autonomous agent — it is a very costly and time consuming procedure to have a human programmer program a particular policy for an agent, and the resulting behavior often fails completely if the environment changes.

Another goal is to study an animal's or animat's (some artificial creature or system) behavior in order to better *understand* its behavior or reaction in particular situations. This can be done by designing a model of the creature and by simulating its behavior with a computer under a wide variety of circumstances. By carefully analyzing the observed behavior, the model may show discrepancies with the real or desired behavior and can be changed accordingly. Examples of interesting behaviors or phenomena to model are: the foraging behavior of a bee, fire spreading in forests or buildings, chewing lobsters, and the reaction of the human immune system to treatment with particular drugs.

Reward functions. When we want to use intelligent agents for solving a particular task, we must use evaluation methods which judge how well the agent performed. Analogous to research in animal biology (where people experiment with living animals to teach some behavior by giving rewards or punishments), our evaluation method will be a reward function assigning rewards and punishments to particular actions in particular environmental situations. These rewards serve to reinforce a particular behavior and hence comes the name reinforcement learning, the main topic of this thesis.

Reinforcement learning. In this thesis we consider reinforcement learning (RL) methods which optimize the policy of an agent by learning from trial and error. The goal is to select for all environmental (or world) states the action which leads to the maximal sum of future rewards. At each time step the agent receives input by observing the world state with its sensors, selects and executes an action based on this input, and receives a reward signal (see Figure 1.2). In order to learn a rewarding policy, the agent uses its experiences (experimental data) to learn a *value function* which estimates the expected future sum of rewards given that the agent is in a particular world state and selects a particular action. When we have an optimal value function, no planning is needed for selecting an action, and therefore we say that the agent follows a *reactive strategy.* Another advantage of learning is thus that no time-consuming planning is needed, although the accuracy of the value function should be extremely high for optimal decision making.

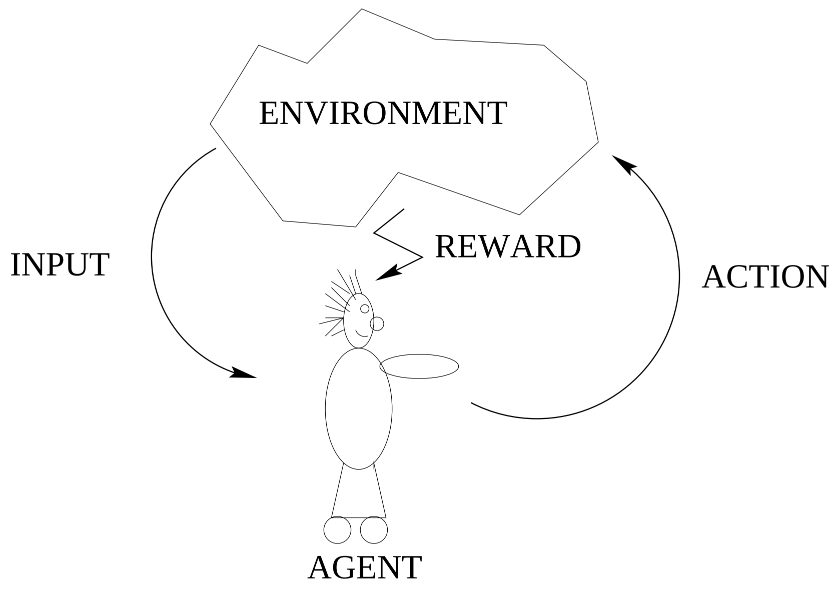


Figure 1.2: *The feedback loop of a RL agent. The agent receives environmental inputs through its sensors, selects an action which changes the state of the environment and receives a reward signal. Based on the rewards it receives, it adapts its policy.*

Applications of RL. The earliest success in RL is Samuel's checkers playing program (1959). His program uses a linear polynomial to evaluate a set of selected features. To make the evaluation of the current board position more precise, the program uses lookahead search. Then he trained the value function by letting the program play against itself, which resulted in the first program which was able to beat its programmer.

Probably the biggest success in RL is Tesauro's TD-Gammon (1992), a backgammon playing program consisting of a feedforward neural network trained by a variant of RL (Sutton, 1988) and self-play. TD-Gammon achieved world class level play and won some matches against some human world class players (Tesauro, 1995).

Other successful applications of RL to complex problems were reported by Zhang and Liu who used RL with time-delay neural networks (LeCun et al., 1989) for iteratively repairing job-shop schedules (Zhang and Liu, 1996), and by Crites who used a team of reinforcement learning agents to train a neural network for controlling multiple elevators in a simulated environment (Crites and Barto, 1996).

The potential of the practical applicability of RL approaches is huge, since it is often easy to set up a system which interacts with an environment and monitors its experiences. Using those generated experiences, the learning method can improve the policy. In backgammon for example, we only need to have a program which allows to play games and a learning architecture which can be used for improving the policy by using newly generated learning examples. Then we can let the computer play and learn for some time until it has succeeded in learning what we wanted it to learn. This is much simpler than designing a position evaluator by hand. E.g., Berliner's BKG backgammon program (1977) took many man years to construct, but its level of backgammon play was already much lower than that of TD-Gammon learning for 3 months on a RS6000 computer.

Complete vs incomplete information. In the following, we will make a distinction between decision making based on *complete state information* and decision making based on *incomplete state information* which causes uncertainty about the true state (remember the chinese wall where we cannot exactly infer where we are — the input allows for many different positions). An incomplete description of the world state is caused by sensory inputs which only partially reflect the true state. In the real world, such partially observable world states are very common, since sensors are imperfect or the exact state is not measurable with practical sensors because states may look exactly the same. Finally, it may be impossible to give a full description of the world state, since too many dynamic objects would need to be described.

The reason of making the distinction is that uncertainty in the state makes the task much more complex.[1] We first describe RL in fully observable environments and then describe RL in partially observable environments.

1.1 Reinforcement Learning

Complete state information. When the agent's sensors can be used to obtain a complete description of the world state, its decision making process is facilitated considerably. However, what does complete state information mean? Does it mean that the agent needs to describe all positions of objects and the values of all their attributes in the environment? The answer is no. There are two ways for obtaining a complete state description:

(1) If the agent has a local view of its environment, e.g. it is equipped with a camera, it would still receive a complete description if its sensory input provides it always with a unique view of the environment. Although most realistic agent's are equipped with such local sensors, they are in general unable to know the values of dynamic attributes. Therefore they can only obtain a complete description for static (unchanging) environments for which only their own position matters.

(2) The agent has a global view of the environment (e.g. by using multiple camera's on the ceilings of all different rooms) containing the values of all dynamic attributes, and its own

[1]Whereas there are efficient algorithms for finding solutions to problems with complete state information, no such algorithms exist for incomplete state information problems.

position in the environment. Values of static attributes are unimportant. E.g., suppose that the agent can go to a cupboard and open its drawer. Since the drawer can be open or closed, this fact may influence the effect of the action and hence the agent's decision making. If the position of the cupboard is constant, there is no need to use it as input information next to the position of the agent, since the effect of executing the *open drawer action* will just depend on the agent's position and the state of the drawer.

Markov decision processes. Many problems consist of making a sequence of decisions, where some decision in a specific world state causes some change of this state. Such problems can be modeled as Markov decision processes/problems (MDPs) and most work in RL has been done in finding solutions to such problems. Making decisions is done by using a policy which selects actions based on the world state. The goal is to find the policy which maximizes the expected cumulative future reward.

Markov property. A Markov decision process is a process where the transition probabilities to the possible next states only depend on the current state and the selected action of the agent. This means that all relevant historical information for determining the next state is present in the current state. Although physicists assume that this so-called Markov property holds in the universe, for decision making it is also important that the Markov property is satisfied from the agent's perspective (who may not always be able to perceive the exact state of the universe). This means that all information which may influence the future of the agent is at all times present in the agent's description of the state. Therefore, for MDPs, the agent is in principle able to exactly learn how the world evolves (although this may be indeterministically). This is of course impossible for many real world problems.

MDP characterization. A Markov decision problem is characterized by a model which consists of a complete transition model containing the transition-probabilities to all states given all state/action pairs, a reward model containing all rewards for all possible transitions and a discount factor which makes rewards received in the future less important than immediate rewards. The reward model is given by the designer and determines which agent's behavior is preferred. The discount factor has the effect that collecting immediate rewards is preferred to postponing collecting them for a while.

Dynamic programming. Algorithms based on dynamic programming (DP) (Bellman, 1961; Bertsekas and Tsitsiklis, 1996) may be used to compute optimal policies for MDPs when a complete model of the MDP is available. DP algorithms use this model for iteratively computing the value function. Unfortunately, a complete model is in general not available, so that DP on the *a priori* model often cannot be used. Furthermore, the number of states is sometimes so large that it becomes infeasible to store the whole model.

Heuristic DP. There exist a variety of RL methods which can be used to learn the policy without the need for an *a priori* model. Instead of creating a precise model (with all states, transition probabilities etc.), constructing a simulator or putting an agent in a real environment for generating experiences is much easier. Heuristic DP (RL) methods can be used as stochastic iterative approximation algorithms and learn from trials in which the agent's policy is used in the simulated or real environment to generate novel experiences. Then RL methods learn from these experiences to improve the approximation of the value function.

TD(λ)-learning. Temporal difference (TD) methods (Sutton, 1988) are a particular kind of RL method which can be used to learn to predict the outcome of a stochastic process given a set of state sequences generated by the process. TD methods are driven by the difference or TD-error between two successive state values and adapt the former state's value to decrease this difference. Sutton (1988) introduced the TD(λ) framework by creating a set of TD methods parameterized by a variable λ which weighs the degree of influence of state values in the distant future relative to the values of immediately successive states. This makes it possible to learn from effects of actions which show up after a long time. Although TD methods are essentially prediction methods, they can also be used for learning to control by extending the system with a model for simulating actions. Then actions can be tried one for one and the predictions of the resulting states can be used for selecting the action.

Q-learning. A well known RL algorithm is Q-learning (Watkins, 1989), which uses its policy to try out sequences of actions through state/action space and uses the rewards received from the environment to learn (estimate) the expected long-term reward for executing a specific state/action pair. Q-learning repeatedly performs a one-step lookahead backup, meaning that the Q-value [2] of the current state/action pair is adapted towards the immediately received reward plus the value of the next state. Thus, Q-learning is essentially a TD method. Q-learning works well for particular problems. It uses the feedback in a rather inefficient way as the feedback is only used for updating a single Q-value at a time. This makes it very slow for solving large problems with long delays until a reward signal is returned. Q-learning has been proven to converge to the optimal value function for MDPs (Watkins and Dayan, 1992), however, if all state/action pairs are tried infinitely many times.

Q(λ)-learning. Q(λ)-learning (Watkins, 1989; Peng and Williams, 1996) combines TD(λ) methods with Q-learning to propagate state/action updates back in time so that multiple state/action pairs (SAPs) which have occurred in the past can be updated. Sequences of SAPs often lead to the same future state/action trajectories and therefore multiple previous Q-values of executed SAPs depend on the current experience. That is probably why Q(λ)-learning has outperformed Q-learning in a number of experiments (Lin, 1993; Rummery and Niranjan, 1994). Q(λ) especially outperforms Q-learning for problems where (1) intermediate rewards are missing or not very informative, (2) the environment is not very stochastic, and (3) many actions have to be selected before a specific goal state has been reached. Larger values for λ increase the variance in the updates, however, since the range of the Q-values of possible state/action pairs far away in the future is usually much larger than Q-values of SAPs which occur immediately. Therefore, when the problem contains a lot of noise or when the variance in emitted rewards of state/action pairs is large, the advantages in using large values for λ is much smaller since the larger variance in the updates slows down converging.

Indirect RL. Q-learning and alternatives are *direct* RL learning methods which means that they directly estimate the Q-values from the results of the interaction with the world. This is in contrast with *indirect or model-based* RL methods which first estimate a model of the transitions and rewards and then apply dynamic programming techniques to compute policies. World models (WMs) integrate all observed experiences in a model so that the loss of useful information is minimal. Modeling approaches have several advantages besides their efficient usage of experiences. E.g. they are useful for efficiently exploring the environment, since they allow the agent to know what has not yet been learned. Since the quality of a computed policy depends on how well the WM fits the real unknown underlying Markov decision process, we can increase its accuracy in the interesting (e.g. unexplored or highly rewarding) areas until near optimal policies have been computed.

The combination of world modeling and dynamic programming tends to be time consuming due to the dynamic WMs which require that the slow DP algorithms compute a new

[2] Q-value stands for Quality value.

policy after each experience (when used in an online manner). More efficient methods partially recompute policies after new information comes along. A simple method, which is very fast, uses model-based Q-learning or Real-Time Dynamic Programming (RTDP) (Barto et al., 1995), which outperforms standard Q-learning due to the higher informed update steps. Even more efficient algorithms manage the cascade of updates which are computed by DP, by assigning priorities to the updates. A well known method for doing this is Prioritized Sweeping (PS) (Moore and Atkeson, 1993), which enables us to (partially) recompute the value function and policy at all times with few informative updates.

Exploration. RL uses multiple trials for resampling the utility of selecting particular SAPs for learning (near) optimal strategies. A policy needs to map all states to actions, and therefore finding the optimal policy needs all state/action pairs being tried out. Initially large parts of the state space are unknown and estimates of actions may be erroneous. To examine the effects of alternative (non-greedy) actions, actions are selected by an exploration rule. Good exploration rules use the policy and possibly additional information to select a promising alternative action about which the agent has gathered insufficient information (in theory each action could be the best possible action, but some are more likely to be optimal than others). Exploration rules "buy" information — the agent would probably obtain more immediate reward by selecting the action that currently looks most promising — in the hope that this information can be used to increase the reward intake in later life. Exploration rules can be split into (1) *undirected exploration* which uses pseudo-random generators to try their luck on generating novel interesting experiences, and (2) *directed exploration* which uses additional information for learning utilities for trying out exploration actions (Schmidhuber, 1991c; Thrun, 1992; Wiering and Schmidhuber, 1998a).

Function approximators. Most real world problems consist of a large or infinite amount of states due to continuous valued variables or a large number of state-variables.[3] To be able to store and learn value functions for such state spaces, we need to approximate the value function by function approximators. Function approximators consist of many adjustable parameters which map an inputvector to an output vector. The parameters of the function approximator are trained on learning examples so that the difference between the desired output and the output of the function approximator on the examples is minimized. Well known examples are feedforward neural networks (Rumelhart et al., 1986; Van de Smagt, 1995), linear or cubic spline interpolation (Press et al., 1988), and Kohonen networks (Kohonen, 1988).

1.2 Reinforcement Learning with Incomplete Information

When states are partially observable (due to e.g. imperfect sensors) the agent cannot always map its inputs to unique environmental states. This uncertainty about the real state makes choosing an optimal action very difficult. Problems with insufficient input information to distinguish between two different states are called *partially observable Markov decision problems* (POMDPs) (Sondik, 1971), and they are the hardest problems for RL — even solving deterministic POMDPs is a NP-complete problem (Littman, 1996).

Algorithms for solving POMDPs are based on using some kind of *internal state* which summarizes previous events. The state produced by combining the internal state and the current input can then be used by the agent's policy for selecting an action. Operations

[3]If we have 20 binary valued variables, our whole state space consists of 2^{20} ($> 1{,}000{,}000$) states.

research has developed several algorithms for finding optimal solutions to POMDPs. Most of these select an action on the basis of so-called *belief states*. Belief states describe the agent's knowledge about the world as a probability distribution over all possible states (Lovejoy, 1991). Since the number of belief states and candidate policies increase exponentially as we perform more planning steps, exact algorithms are computationally too expensive for problems containing more than 100 states (Littman et al., 1995a). Therefore, heuristic methods are needed to solve them in reasonable time. There are many possible heuristic algorithms, e.g., function approximators may be useful or we may keep the input representation small by only using short term memory of interesting events. Most real world problems are POMDPs, since it is usually not feasible to have perfect state information. Therefore the number of applications is huge. An example of a POMDP is the task of determining optimal dates for appointments and (often expensive) check-ups for patients with a serious disease (Peek, 1997).

Changing environments. Solving a task in a changing (non-stationary) environment is another complex problem. Changing environments could be: (1) environments where the outcome of executing an action changes over time, e.g. when an effector is damaged, or (2) environments where attributes of objects in the environment change over time, or (3) where the goal changes.

Although the agent could have complete state information, it cannot compute an optimal policy if it does not know the transition function or reward function in future time-steps. Therefore it has to find out what changes and this is very hard. Consider a vacuum-cleaning robot in a specific environment where chairs and tables move around and where reference points such as doors and windows may be opened and closed. If we need to know the exact state of the environment for computing the best vacuum cleaning trajectory, we would need to find out how the chairs and tables are moved around. Dealing with many dynamic variables is a challenging issue, since they cause an explosion of the state space. Fortunately, decision making in most situations only needs to use few variables and therefore huge reductions of the state space can be made by making the variables context dependent. E.g., an agent only needs to know whether a drawer is open or closed when it sees a cupboard and wants to take something out of it.

Multiple agents in RL. Recent research in RL focuses more on problems featuring multiple autonomous agents, e.g, (Mataric, 1994). When agents adapt their behavior, the environment of an agent (which contains the other agents) will change. Thus, multi-agent problems are changing environments from the perspective of each agent. To make such problems fully observable the planned actions or policy of other agents need to be given as part of the input, which again blows up the state space when there are many agents.

There are two kinds of multi-agent problems: (1) agents take part of a cooperative community and try to achieve the maximal reward for the entire group, and (2) agents try to optimize their own rewards and may compete (but also cooperate) with other agents to achieve their goals. Using RL for the first problem, i.e. for learning community policies, poses many, still unresolved, problems. These include among others: agent-organization (which subgroups are formed and which interactions will take place), task specialization (what is the function of each subgroup/individual), and communication languages to make interaction possible (Lindgren and Nordahl, 1994; Steels, 1997).

The second problem, i.e. of maximizing the reward intake of an individualistic rational agent in an environment with other agents, has been studied for the largest part by game theory (Myerson, 1991). A well known problem where agents have to choose between cooperation and competition is the prisoner's dilemma (Axelrod, 1984; Sandholm and Crites, 1995).

It is interesting to note that researchers from experimental economics have found their way to reinforcement learning algorithms for explaining rational human behavior (Roth and Erev, 1995).

An illustration of the increase of RL work in multi-agent problems is the major interest in learning robotic soccer teams, for which already world-championships are held (Kitano et al., 1997). A successful example in multi-agent RL in which agents have to cooperate is a problem studied by Crites and Barto (1996). They used RL to learn elevator controllers for a (simulated) building containing four elevators and were able to outperform even the best conventional elevator dispatchers.

1.3 Current Problems of RL

Although promising, the current state of the art of RL is not very well developed and much exciting work still remains to be done. TD-Gammon is a major success for reinforcement learning and in this section we will review the major challenges to obtain the same kind of success for other problems.

RL as a tool for building intelligent systems. RL provides us with useful tools which need to be combined with particular intelligent design methods to make it really work. One important factor which contributes to the success of a RL application is the utility of the chosen function approximator for representing the value function for the problem at hand. The feedforward neural network used in TD-Gammon was very well chosen since it allowed to generalize over the state space, thereby making it possible to select among possible moves in situations which had never occurred during training. The smoothness of the value function in a stochastic game like backgammon makes generalization much easier.

Although RL is a promising method, using it for attacking a complex problem is not straightforward. In this thesis, we describe the current state of the art in the research for RL methods and face the current problems associated with learning value functions:

- **Exploration.** Experimental data makes it possible to predict the consequences of particular behaviors. However, data can be costly to obtain. Then, how can we efficiently gather data to train the system?

- **Uncertainty.** In real world problems, information about the state of the environment is often incomplete. This makes the process of decision making under uncertainty an important issue. The problem is NP-complete (Littman, 1996), however, thus we would like to find heuristic algorithms which are useful for solving problems containing many states.

- **Generalization.** Function approximators can be used to generalize from the experiences with one situation to similar situations so that it is not necessary to store or encounter each possible situation.[4] Although function approximators are in theory a powerful method for storing input-output mappings and can be used potentially in combination with RL, there still needs to be done a lot of research before we can really exploit this combination. Often, function approximators are chosen on an ad hoc basis

[4]Even when it is possible to store all states, experiencing all of them would clearly be inefficient in terms of time and other costs.

resulting in unexpected successes or catastrophes. Furthermore, when applied to a particular task, some function approximators result in much faster training times or final performance levels than others. We would like to know which function approximators are most useful in combination with RL and on which task features this depends.

- **Model-based learning.** Model-based learning can significantly speed up reinforcement learning. We can use statistical world models to estimate the probability of making a transition to each possible successive state, given that some action has been executed in some state. These world models can be used to store experimental data much more efficiently, to detect what the agent does not know, i.e. has not experienced, to incorporate *a priori* knowledge about the problem, to explain what the agent has been doing all the time, to compute a new policy if the goal changes, and to overcome problems due to forgetting or interference of multiple problems. A lot of research has to be done, however, before we can efficiently combine model-based learning with function approximators.

- **Multi-agent RL.** Current RL methods have been designed for single agents. When multiple agents are used, many new challenging issues arise: e.g. if a group of agents is acting together, which agents were responsible for the given outcome? How can we let the agents cooperate in specific plans? How can we organize the agents? We also need to transform single agent RL methods so that they can be applied to multiple agents and that may not always be straightforward. Given their flexibility and power, one of the ultimate goals of RL would surely be to solve complex problems with multiple agents.

1.4 Goals of the Thesis

This thesis aims at describing all topics of interest in RL and proposing solutions to the current topics of interest. First of all, we aim to giving a clear description of the principles of reinforcement learning. Then, our goal is to develop new methods to (1) handle the exploration problem, (2) deal with uncertainty in the state description, (3) combine model-based RL with function approximators, and (4) apply RL to multi-agent problems. We hope that we have succeeded in explaining these topics in an understandable way to those readers who are not familiar with reinforcement learning, while still keeping the more professional reinforcement learning researcher interested by presenting many different and novel insights which found their way through this thesis.

All issues will be considered from a computer scientist's point of view: throughout the thesis, we will consider the algorithmic and computational issues for finding good solutions and use experiments to compare different methods.

Finally, we hope that with this thesis, current understanding of issues which are needed to be resolved to make RL useful for solving complex real world problems has increased significantly. Examples of problems which we would think could be solved with RL in the future include: controlling forest fires (Wiering and Dorigo, 1998), directing traffic flow, debugging computer programs, routing messages on the internet (Di Caro and Dorigo, 1998), and regulating combustion engines to reduce CO_2 emission.

1.5 Outline and Principal Contributions

First part

The first part of the dissertation describes the Markov decision process framework, dynamic programming, and reinforcement learning. The presented algorithms are described in the context of problems involving full state information.

Chapter 2 describes MDPs and dynamic programming. This will serve as a general framework and to introduce the notation used throughout this thesis.

Chapter 3 describes reinforcement learning together with the newly developed fast online Q(λ) approach and a novel team Q(λ)-learning algorithm.

Fast online Q(λ)-learning. Typical *online* Q(λ) implementations (Peng and Williams, 1996) based on lookup-tables are unnecessarily time-consuming. Their update complexity is bounded by the size of state/action space. We describe a novel online Q(λ) variant which is more efficient than others — it has average update complexity which is linear in the number of actions.

Team Q(λ)-learning. When dealing with multiple agents, it may happen that agent trajectories meet. Rather than only letting agents learn from their own experiences, we can make use of such crossing points so that agents can learn from the experiences of other agents. Our novel Team Q(λ) method is a new algorithm for linking agent trajectories.

Finally, at the end of Chapter 3 we describe the results of experiments with maze-tasks to evaluate existing algorithms and our novel RL algorithms.

Chapter 4 describes model-based learning. It shows how world models can be estimated and describes a model-based version for Q-learning (RTDP) (Barto et al., 1995). Then we review prioritized sweeping (PS) and introduce an alternative PS variant which manages updates in a more exact way. We also present an experimental comparison between the different model-based reinforcement learning approaches.

Chapter 5 describes exploration in reinforcement learning. Active data gathering or exploration (Fedorov, 1972; Schmidhuber, 1991a; Thrun and Möller, 1992; Cohn, 1994) is important for efficient learning, because otherwise lots of time is wasted on improving the approximation on what is already known, whereas large parts of the state space are badly fitted by the model. We have constructed a novel exploration approach which is based on model-based RL (MBRL).

Learning exploration models. We study the problem of collecting *useful* experiences for MBRL through exploration in stochastic environments. The novel method is based on learning an exploration value function and relies on maximizing exploration rewards for visits of states that promise information gain. We also combine MBRL and the Interval Estimation algorithm (Kaelbling, 1993). Results with maze experiments demonstrate the advantages of our approaches.

Second part

The second part of the dissertation describes generalizations of the MDP framework described above: partially observable MDPs, continuous MDPs and function approximation, and multi-agent learning.

Chapter 6 describes POMDPs in more detail, reviews approaches to solve them and presents our novel approach: HQ-learning.

HQ-learning attempts to exploit regularities in state space. Its divide-and-conquer strategy discovers a subgoal sequence decomposing a given POMDP into a sequence of *reactive policy problems* (RPPs). RPPs are problems for which each state which is mapped to an identical input by the sensors require the same optimal action. This makes it possible to store their solution in one representation. The only "critical" inputs are those corresponding to transitions from one RPP to the next. HQ is based on two cooperative temporal difference learning rules: the first learning rule learns a decomposition of the task into a temporal sequence of subtasks and the second rule learns policies to solve the subtasks. HQ does not need an *a priori* model of a POMDP and experiments show that it is able to find good solutions for large deterministic partially observable maze-tasks.

Chapter 7 describes function approximation methods which can be used in combination with Q(λ)-learning for learning the value function for large or continuous input spaces. We describe linear networks, a new variant of neural gas (Fritzke, 1994), and CMACs (Albus, 1975a). Then we will also show how these methods can be combined with world models.

CMAC models. We will in particular focus on CMAC models consisting of a set of independent filters which produce evaluations which are combined for selecting an action. The models are used to estimate the dynamics of each independent filter.

Multi-agent soccer. As a test environment for the function approximators we use simulated soccer. The task is to learn policies which outperform a fixed opponent. Results are shown for varying team sizes and different strengths of opponent teams.

Chapter 8 concludes this thesis, reviews what has been done, and outlines interesting directions for future work.

Chapter 2

Markov Decision Processes

This chapter describes dynamic programming (DP) in the context of Markov decision processes (MDPs). The reader who is not familiar with Markov processes (MPs) is adviced to read Appendix A first. MPs can be used to formalize stochastic dynamic processes. Section 2.1 formally describes Markov decision processes (MDPs). MDPs are controllable Markov processes and solving a MDP means finding the policy which maximizes some specific reward criterion. Section 2.2 describes DP algorithms which are able to compute the optimal policy for MDPs. Problems of DP are that they need a tabular representation for the state/input space and that they are computationally expensive for large numbers of states. In Section 2.3 we show results of DP algorithms on maze problems and evaluate their computational demands as maze sizes increase. In Section 2.4 we describe some of the difficulties of using the MDP framework directly for real world problem solving and describe some extensions for dealing with more general decision problems. Finally, in Section 2.5 the conclusions of this chapter are given.

2.1 Markov Decision Processes

A Markov decision process (MDP) is a controllable dynamic system whose state transitions depend on the previous state and the action selected by a policy. The policy is based on a reward function which assigns a scalar reward signal to each transition. The goal is to compute a policy for mapping states to actions which maximizes the expected long-term cumulative (discounted) reward, given an arbitrary initial state.

Basic set-up. We consider discrete Markov decision processes consisting of:

- A time counter $t = 0, 1, 2, 3, \ldots$
- A finite set of states $S = \{S_1, S_2, S_3, \ldots, S_N\}$. We denote the state at time t as s_t. To simplify notation, we will also use the variables i and $j \in S$ to refer to states.
- A finite set of actions $A = \{A_1, A_2, A_3, \ldots, A_M\}$. We denote the action at time t as a_t.
- A probabilistic transition function P. We use $P_{ij}(a) := P(s_{t+1} = j | s_t = i, a_t = a)$ for $i, j \in S$ and $a \in A$ to define the transition probability to the next state s_{t+1} given s_t and a_t.

- A reward function R maps a transition from state/action pair (SAP) (i, a) to state j to scalar reward signals $R(i, a, j) \in I\!R$. We assume that the reward function is deterministic, although it could in principle also be stochastic. We denote the reward at time t as r_t.

- The discount factor $\gamma \in [0, 1]$ is used to discount rewards received in the future. An agent is interested in accumulating as much future reward as possible. By exponentially decaying rewards with the number of time steps until the reward is received, immediate rewards are made more important than rewards received after a long time.

- P_{init} defines the initial probability distribution over all states. $P_{init}(s)$ is the probability the agent starts in state s.

2.1.1 Value Functions

Given the transition and reward functions, the goal is to find the policy Π^* which maps states to actions ($a^* = \Pi^*(s)$) with maximal expected discounted future reward. For defining how good a policy Π is, a value function V^Π is used.[1] The value $V^\Pi(s)$ is a prediction of the expected cumulative rewards received in the future given that the process is currently in state s and the policy Π is used throughout the future. Value functions are used to enable the evaluation of the available policies.

There are two common ways for dealing with the future: one method constraints the future until some finite horizon, whereas the other method allows the process to go on forever by using an infinite horizon.

2.1.2 Finite Horizon Problems

For many problems there is a bound on time (often called deadline). This means that the policy has to stop at some point and has to accumulate as much reward as possible during its limited “life-time”. In this case the value function V is time dependent and therefore we have to write V_t to account for the number of steps left in the future (t). Furthermore we have a non-stationary (time dependent) policy $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_T\}$, with horizon limit T. The horizon-bounded value function V_T^Π is allowed to make T decisions with policy Π and equals:

$$V_T^\Pi(i) = E(\sum_{t=0}^{T-1} R(s_t, \Pi(s_t), s_{t+1}) | s_0 = i)$$

Here we start with state s_0, and add all transition rewards, where a transition to state s_{t+1} is caused by selecting action $a_t = \Pi(s_t)$ in state s_t. The expectation operator E is used to account for the probability distribution over the state-trajectories.

Computing the optimal policy. Computing optimal solutions for deterministic shortest path problems can be efficiently done with Dijkstra's shortest path algorithm (Dijkstra, 1959). For general problems consisting of non-deterministic transition functions and arbitrary reward functions we need to use a more advanced and generalized algorithm. To find the optimal policy Π^*, we can proceed as follows: first we find the optimal action when only

[1]When it is clear which policy we are using, we may drop the policy-index and simply write V)

1 step can be made from each state, and compute the optimal value function V_1^*. Thus, first we compute for all states $i \in S$:

$$V_1^*(i) = \max_a \{ \sum_j P_{ij}(a) R(i,a,j) \}$$

and set the 1-step policy action in each state $i \in S$ to:

$$\Pi_1^*(i) = \arg\max_a \{ \sum_j P_{ij}(a) R(i,a,j) \}$$

Once we have calculated the optimal value function V_1^* we use it to compute V_2^* and so on for steps $1, \ldots, T$. The idea is that within t steps, the maximal reward we can obtain with the best action is equal to the immediate reward plus the cumulative reward obtained from making the optimal $t-1$ steps from the next state. Thus, we use the value function to accumulate the maximal attainable reward over more and more steps. Finally, the expected future cumulative reward over T steps is reflected in the value function V_T^*. Given V_t^*, we compute V_{t+1}^* as follows:

$$V_{t+1}^*(i) = \max_a \{ \sum_j P_{ij}(a)(R(i,a,j) + V_t^*(j)) \}$$

and we set the $t+1$-step policy actions to the action a which maximizes the value function:

$$\Pi_{t+1}^*(i) = \arg\max_a \{ \sum_j P_{ij}(a)(R(i,a,j) + V_t^*(j)) \}$$

2.1.3 Infinite Horizon Problems

Sometimes we do not want to pose a bound on the time counter t, since we would like the agent to continue "forever" or we do not have any *a priori* information about the duration of the problem we wish to solve. For such MDPs, we can use the infinite horizon case. For dealing with unlimited time-bounds, we cannot simply sum all future rewards: we must make sure that the summary operator over the future rewards returns a bounded (finite) value. This is usually done by using a discount factor γ which weighs future rewards in a way which assigns high weights to immediate rewards and weights which go to 0 for rewards which will be received far away in the future (Lin, 1993) [2]

Since we are dealing with an infinite horizon, we want to compute a stationary policy. Fortunately, the optimal policy for infinite horizon problems are always stationary since the specific time step becomes meaningless due to the infinite future. The value function V^Π for policy Π starting at state i is given by:

$$V^\Pi(i) = E(\sum_{t=0}^{\infty} \gamma^t R(s_t, \Pi(s_t), s_{t+1}) | s_0 = i)$$

Given a fixed policy Π, the actions in all states are fixed and the transition probability only depends on the current state. For a deterministic policy, the transition matrix P with

[2] Instead of using discounting, we may also use the average reward over all future steps (Van der Wal, 1981; Schwartz, 1993; Mahadevan, 1996).

transition probabilities P_{ij} is given by:[3]

$$P_{ij} = P_{ij}(\Pi(i))$$

Furthermore, we define the reward matrix R with rewards R_{ij} as follows:

$$R_{ij} = R(i, \Pi(i), j)$$

Now we can calculate the value function for policy Π as follows:

$$V^{\Pi} = \sum_{t=0}^{\infty} \gamma^t P^t Diag'(PR^T) \tag{2.1}$$

$Diag'(PR^T)$ returns the main diagonal of the matrix PR^T. This is a vector which contains the average single step reward for the different states. When we multiply this with the discounted average number of times that the states are visited in the future, we get the value function.

By making use of the fact that a state value equals the immediate reward plus the expected value of the subsequent state, Equation 2.1 can be rewritten as:

$$V^{\Pi} = \gamma P V^{\Pi} + Diag'(PR^T) \tag{2.2}$$

and we can solve this by computing:

$$V^{\Pi} = (I - \gamma P)^{-1} Diag'(PR^T)$$

Note that for ergodic Markov chains it holds that when $0 \leq \gamma < 1$, the inverse $(I - \gamma P)^{-1}$ can be computed, since $(I - \gamma P)$ is non-singular. However, there are better ways to calculate V^{Π} than by inverting a matrix, which may introduce rounding-errors due to an ill-conditioned matrix. We can compute the value function by iteration (Van der Wal, 1981; Bertsekas and Tsitsiklis, 1996). Each step improves the value function so we are sure that the computation will converge to the optimal value function. Thus, we iteratively compute for a certain policy Π, and for all states i:

$$V^{\Pi}(i) = \sum_j P_{ij}(R_{ij} + \gamma V^{\Pi}(j))$$

By updating the value function in this way, we essentially execute an additional lookahead step. The iteration will converge to the expected cumulative reward obtained by the policy.

For the optimal policy, we always select that action in a state which maximizes the value of that state. Thus, for the optimal value function V^* the following holds:

$$V^*(i) = \max_a \sum_j P_{ij}(a)(R(i,a,j) + \gamma V^*(j))$$

This equation is known as the Bellman equation (Bellman, 1961) and is the basis for dynamic programming. Every optimal policy satisfies the Bellman equation and every policy which satisfies the Bellman equation is optimal.

[3]In case of stochastic policies we can compute the transition and reward functions by taking the action probabilities into account.

2.1.4 Action Evaluation Functions

Q-functions evaluate actions given a specific state. They are functions of the form $Q^{\Pi}(i,a)$, and they return the expected future discounted reward when the agent is in state i, action a will be executed, and policy Π will be followed afterwards. For each policy Π, we can calculate the action evaluation function $Q^{\Pi}(i,a)$, $\forall a \in A$ and $\forall i \in S$:

$$Q^{\Pi}(i,a) = \sum_j P_{ij}(a)(R(i,a,j) + \gamma V^{\Pi}(j)) \tag{2.3}$$

A policy can exploit a given Q-function by selecting in each state the action which maximizes that Q-function. Thus, we have:

$$\Pi(i) = \arg\max_a \{Q^{\Pi}(i,a)\} \tag{2.4}$$

In general applying Equation 2.4 creates a new policy which will not be equal to the one before, and thus after computing a new policy we compute a new Q-function for it according to 2.3. By repeatedly alternating these computations, the Q-function converges to the optimal Q-function Q^*. The optimal policy Π^* selects the action which maximizes the optimal action value function $Q^*(i,a)$ for each state $i \in S$:

$$\Pi^*(i) = \arg\max_a \{Q^*(i,a)\}$$

Sometimes Q-values of different actions are equal. In such cases the agent is allowed to randomly select between the optimal actions.

2.1.5 Contraction

Although there may exist multiple optimal (deterministic) policies which differ in selecting an action for a state from actions with equal Q-values, there is only one optimal value or V-function: V^*, what we will show now. A V-function is a function of a state i and the actions of the policy. The optimal V-function is the one where the policy receives most future rewards for all states i:

$$V^*(i) = \max_{\Pi} \{V^{\Pi}(i)\}$$

Dynamic programming computes V^* by iteratively using a backup operator B. The operator B performs a one-step lookahead by evaluating all possible actions and selecting the optimal action:

$$B(V(i)) = \max_a \sum P_{ij}(a)(R(i,a,j) + \gamma V(j))$$

Thus, B may change actions in order to improve the value function or it may just compute a better estimate of the value function (without changing actions).

We will denote the use of the operator B on V as simply : BV. For the optimal value function V^*, the following must hold:

$$BV^* = V^*$$

This means that V^* is a fixed point in the value function space. B is a contraction operator, which means that each time we apply B on an arbitrary value function V, we get closer to the fixed point V^*.[4] This is a necessary condition for contraction and means that convergence will take place as we apply B infinitely many times (Bellman, 1961). To define a distance measure, we use the largest absolute difference between two values $V_1(i)$ and $V^*(i)$ for some specific state i (i.e. the max norm is used). This distance gets smaller when B is applied. First we define the max norm distance function:

$$||V^* - V_1||_\infty = \max_s |V^*(s) - V_1(s)|,$$

where $|a|$ denotes the absolute value of a. Then we show:

$$\begin{array}{ll}
||BV^* - BV_1||_\infty & = \\
\max_i |BV^*(i) - BV_1(i)| & = \\
\max_i |(\max_a \sum P_{ij}(a)(R(i,a,j) + \gamma V^*(j))) - (\max_b \sum P_{ij}(b)(R(i,b,j) + \gamma V_1(j)))| & \leq \\
\max_i \max_c |(\sum P_{ij}(c)(R(i,c,j) + \gamma V^*(j))) - (\sum P_{ij}(c)(R(i,c,j) + \gamma V_1(j)))| & = \\
\max_i \max_c |(\sum P_{ij}(c)(\gamma V^*(j) - \gamma V_1(j)))| & \leq \\
\gamma \max_j |V^*(j) - V_1(j)| & = \\
\gamma ||V^* - V_1||_\infty & \square
\end{array}$$

Since this holds for all starting value functions V_1, and V^* is a fixed point, there cannot be any local minima. Thus, we have shown that operator B applied on the value functions strictly brings the value functions closer when $\gamma < 1$.

2.2 Dynamic Programming

Dynamic programming (DP) can be used to compute optimal policies for (infinite horizon) MDPs (Bellman, 1961). There are three well known algorithms for computing the policy and value function: policy iteration, value (greedy) iteration, and linear programming. Policy iteration completely evaluates a policy by computing the value function after which the policy is changed so that always the actions are chosen with maximal Q-values. Value iteration changes $\Pi(s)$ whenever $Q(s,a)$, for all $a \in A$ have been computed. Linear programming (LP) maximizes the value function subject to a set of constraints (D'Epenoux, 1963; Littman, 1996). We will show the policy and value iteration algorithms, and solving a MDP as a linear programming problem.

2.2.1 Policy Iteration

Policy iteration computes optimal policies and always terminates in finite time (Littman, 1996; Bertsekas and Tsitsiklis, 1996). This can be easily seen, since there are $|A|^{|S|}$ policies, and policy iteration makes an improvement step at each iteration.[5] The algorithm consists of an iteration over two subroutines: policy evaluation and policy improvement.

The algorithm starts with an arbitrary policy and value function. The policy Π is evaluated by using the transition matrix P and the reward function R:

$$V^\Pi = (I - \gamma P)^{-1} Diag'(PR^T)$$

[4]Strictly speaking, the distance never increases, i.e. B is a non-expanding operator. However, we will see that if the discount factor is smaller than 1, the distance decreases.

[5]Policy iteration does not solve a MDP in *polynomial* time in the number of states and actions, however (Littman et al., 1995b).

Note, that instead of fully evaluating the policy, we may also repeatedly use the following equation for all states i to synchronously update V^{Π}:

$$V^{\Pi}(i) = \sum_j P_{ij}(\Pi)(R(i, \Pi(i), j) + \gamma V^{\Pi}(j))$$

and stop when the largest difference between both sides of the equation is smaller than some specific threshold ϵ. This may speed up policy evaluation significantly, although when ϵ is too large the resulting policy may not be optimal. In general ϵ is called the Bellman residual (the error which remains after iterating), and the resulting imperfect value function is bounded as follows (Williams and Baird, 1993; McDonald and Hingston, 1994; Singh and Yee, 1994):

$$V^{\Pi}(x) \geq V^*(x) - \frac{2\gamma\epsilon}{1-\gamma}$$

After evaluation we make a policy improvement step. First we compute the action evaluation function with elements $Q(i, a)$ which assigns the long term expected discounted reward when action a is selected in state i:

$$Q^{\Pi}(i, a) = \sum_j P_{ij}(a)(R(i, a, j) + \gamma V^{\Pi}(j))$$

Then, we improve policy Π by taking the action in each state which maximizes the action value function $Q^{\Pi}(s, a)$:

$$\Pi(s) = \arg\max_a \{Q^{\Pi}(s, a)\}$$

The policy evaluation and improvement steps should be repeated for a specific number of times until the policy is not changed anymore. Then the algorithm stops with the optimal value function V^* and an optimal policy Π^*.

The complexity of the algorithm mainly lies in the evaluation part. Inverting an $n \times n$ matrix costs $O(n^3)$ when no clever methods are used. We can speed this up to about $O(n^{2.807})$ (Press et al., 1988). This can be sped up further by using a stopping condition (minimal error requirement). The number of evaluate-improve repetitions is usually quite small, but depends on the problem complexity. In the worst case it may be $O(|A|^{|S|})$, the total number of policies, which may happen in case policy iteration always makes a single improvement step of the policy. For simple problems, e.g., when a goal state is accessible from all states and when all rewards are negative, the number of repetitions will be very small.

2.2.2 Value Iteration

Value iteration, also called greedy iteration, does not fully evaluate a policy before making policy improvements. The algorithm starts with an arbitrary policy Π and an arbitrary value function V and repeats the following steps for all $i \in S$. First, the Q-function for all actions in state i are computed:

$$Q(i, a) = \sum_j P_{ij}(a)(R(i, a, j) + \gamma V(j)) \tag{2.5}$$

Then, we calculate the new value function:

$$V(i) = \max_a \{Q(i, a)\}$$

Then, we change $\Pi(i)$ so that the action a which maximizes the value function in a state i will be chosen:

$$\Pi(i) = \arg\max_a \{Q(i,a)\}$$

The algorithm halts when the largest difference between two successive value functions is smaller than ϵ. Synchronous value iteration updates the values in step (2.5) by using the value function from the previous time step, whereas asynchronous value iteration immediately uses the already recomputed state values for calculating other state values.

Value iteration repeatedly performs a one-step lookahead. This is a big difference with policy iteration which evaluates the policy completely by looking ahead until no more changes can be seen. In contrast to policy iteration, value iteration is not guaranteed to find the optimal policy in a finite number of iterations (Littman, 1996).

When $M = \max_i \max_a |\sum_j P_{ij}(a)R(i,a,j)|$ and we start with a value function V^0 which is initialized with sufficiently small values, then the number of iterations t^* needed to execute the (synchronous) value iteration algorithm until $||V - V^*|| \leq \epsilon$, is (Littman, 1996):

$$t^* = \lceil \frac{log(M) + log(\frac{1}{\epsilon}) + log(\frac{1}{1-\gamma})}{log(\frac{1}{\gamma})} \rceil \tag{2.6}$$

For particular MDPs differences in value functions for different policies may only be seen when $\epsilon \to 0$. This means that value iteration may need many iterations; it will certainly need more iterations than policy iteration. However, value iteration does not need to evaluate the policy completely, and this may significantly speed up computation.

2.2.3 Linear Programming

We can also define a MDP as a linear programming problem. A linear program consists of a set of variables, a set of linear inequalities (constraints), and a linear objective function. In (D'Epenoux, 1963; Littman, 1996) a linear program is described with variables $V(i)$, for all $i \in S$. The goal is to minimize:

$$\sum_i V(i)$$

Under the following constraint for all $a \in A$ and all $i \in S$:

$$V(i) \geq \sum_j P_{ij}^a (R(i,a,j) + \gamma V(j))$$

The intuitive meaning is that we are searching for a value function for which all constraints hold. The one for which the values are minimized is the least upper bound of these value functions and therefore the true maximum. The linear programming problem consists of N variables and $N \times M$ constraints. For large N and M, the linear programming approach can be practical by using special large-scale LP methods (Bertsekas and Tsitsiklis, 1996).

2.3 Experiments

To compare the dynamic programming methods policy iteration and value iteration we have created a set of mazes. We do not use linear programming here since the problem contains (too) many states. The reasons for using mazes as case study are: optimal policies can easily be calculated, their complexities can easily be adapted, and many other researchers have used them so that they can be seen as a general testbed for algorithms solving (discrete) MDPs.

2.3.1 Description of the Maze Task

The story. Someone is living in a city which suffers from disasters. Therefore he has decided that he needs to look for a better place to stay. From inherited stories, he has found an old map of the environment which shows a place which is very rich, safe and contains lots of food. Using the map, he wants to find out what the best way is for going to the land. The environment consists of impassable mountains, open fields and dense forests. To cross open fields and dense forests costs a day, but it costs more effort to cross open fields than dense forests. The man also knows that there will be a specific probability that his actions are not executed as he wanted. Which method should the pilgrim use in order to plan his way to the promised land by the least effort?

More formally. We use a quantized map consisting of a number of states (25×25, 50×50, or 100×100). In each state the agent (pilgrim) can select one of the four actions: *go north, go east, go south, go west.* Sometimes an action gets replaced by another action due to the weather conditions which make it impossible to know in which direction the agent is heading. It is important to note that each area looks different to the agent since the agent may look at the flora and fauna and discovers that no area looks the same. This important property of the environment allows the agent to uniquely identify the state in which he is walking.

Figure 2.1 shows an example maze, where blocked (mountain) states are represented by black fields, and penalty (forest) states are represented by grey fields. Open fields are white. The starting state, indicated by the letter S, is located 1 field north/east of the south-west corner. The goal state (G) is located 1 field south/west of the north-east corner.

Reward functions. Actions leading to a mountain area (blocked field) are not executed and are punished by a reward of -2. Normal steps through open fields are punished by a reward of -1. Dense forests areas are difficult to cross, since they may harm the agent or involve some risk. Therefore by executing an action which leads to a forest field the agent receives a penalty of 10 points ($R = -10$). For finding the goal state, a reward of 1000 is returned. The discount factor γ is set to .99.[6]

Comments about the complexity. The mazes we use are somewhat more complicated than commonly used mazes, since the agents do not only have to find the shortest path to the goal, but also try to circumvent crossing punishing states. Furthermore, since we use noise in the execution of actions (we replace selected actions by random actions without informing the agent about which action was really executed), the agent wants to find the policy which collects most reward on average. This means the agent has to use the local rewards to find the globally most rewarding policy.

For each different maze-size we have generated 20 different randomly initialized mazes, all with a maximum of 20% blocked fields and 20% punishment fields (these are inserted randomly). We discarded mazes that could not be solved by Dijkstra's shortest path algorithm

[6]The discount factor can be seen as the probability that the process continues — instead of using a discount factor we can multiply all transition probabilities by γ and add transitions with probability (1-γ) from each state to an absorbing zero-value state. Thus by using a discount factor, we implicitly model the fact that an agent may halt before finding the goal.

(Dijkstra, 1959). One of the randomly generated mazes is shown in Figure 2.1. Selected actions are replaced by random actions with 10% probability (the noise in the execution of actions is 10%).

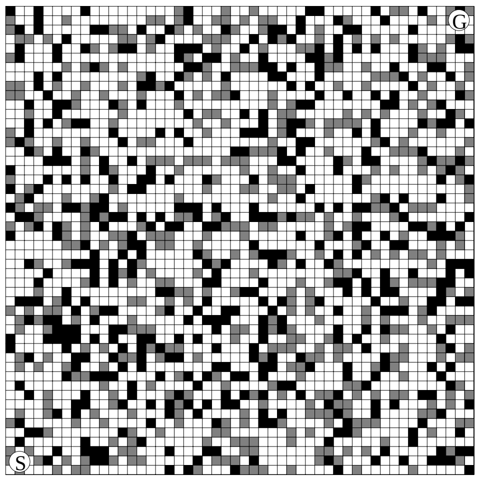


Figure 2.1: *A* 50×50 *maze used in the experiments. Black fields denote mountains (blocked fields), grey fields denote passable dense forests (penalty fields). The agent starts at S and has to search for the way of the least effort to the goal G.*

Traditional methods. We can solve planning problems by specifying them within the MDP framework. An advantage compared to more traditional planning problems is that MDPs allow stochasticity in the state transition function. E.g., for the problems considered above, A* planning (Nilsson, 1971; Trovato, 1996) would not work. It would select an action in a state and always expect to see the same successor state, which will often not be the case. Therefore it will not compute the optimal way. The same holds for Dijkstra's algorithm (1959) which is a method for computing the shortest path for deterministic problems only.

2.3.2 Scaling up Dynamic Programming

To examine how DP algorithms scale up, we executed policy iteration, synchronous value iteration and asynchronous value iteration on the *a priori* models of the 20 mazes of the different sizes. We used $\epsilon = 0.001$. For value iteration, we tried the following stopping conditions for the iteration:

(A) $\max_s\{|V_{t+1}(s) - V_t(s)|\} < \epsilon$, which is the commonly used stopping condition, and

(B) $\max_s\{V_{t+1}(s) - V_t(s)\} < \epsilon$. This stopping conditions makes sense if we know that value iteration approximates the optimal value function from below, i.e. if the initial value function is definitively smaller than the optimal value function, then value iteration will initially increase all values until the goal relevant information (i.e. the shortest path to the goal) has been backpropagated to all states. Since condition (B) is less strict than (A), the value iteration algorithm will stop earlier.

Results. The results of policy iteration (which uses the first stopping condition) on the different maze sizes are given in Table 2.1, and Tables 2.2 and 2.3 show the results with asynchronous and synchronous value iteration using the two different stopping conditions.

We can see a huge difference between policy evaluation and value iteration if we compare the number of times the Bellman backup operator was called for each state. Policy evaluation is an order of magnitude slower than value iteration (both using condition A). This difference can be explained by the fact that initial full policy evaluations are expensive since the policy implements many cycles with high probability through the state space. Therefore, there are many recurrences which need to be solved. Value iteration quickly learns policies which are more directed towards the goal state, and therefore only low probability cycles remain which do not cause large problems due to the fact that we cutoff low update steps. Asynchronous value iteration (B) performs the best, and uses the update information earliest — in a way it is more "online" than the other approaches.

Stopping condition (B) performs much better than (A). Using condition (B), asynchronous value iteration needs to iterate as many times as the length of the solution path. After this, stopping condition (A) will continue updating, since the value function does not yet reflect all cycles between states emitting negative rewards. Although this changes the value function, it hardly affects the policy. Value iteration with stopping condition (A) can cost the same number of iterations for different maze sizes. Note that the number of iterations is determined by the discount factor, transition function, and the reward function and not by the maze size. Compare this also with Littman's upperbound in equation 2.6 which computes $t^* = 1822$. In practice the algorithm takes less than 762 iterations.

Maze Size	Policy Iterations	no. Backups	Cum. Reward
25×25	11 ± 1	3040 ± 380	942 ± 12
50×50	19 ± 2	6450 ± 480	844 ± 24
100×100	34 ± 3	12650 ± 680	691 ± 28

Table 2.1: *Computational costs for policy iteration to compute* 0.001-*optimal policies for the different maze sizes. The number of policy iterations tracks the number of policy improvement steps, whereas the number of backups tracks how often the Bellman backup operator (to look one step more ahead) was called for each state. The last column indicates the cumulative rewards obtained by the computed policy during a single trial starting in S.*

Maze Size	no. Backups (A)	Cum. Reward (A)	no. Backups (B)	Cum. Reward (B)
25×25	669 ± 221	939 ± 14	43 ± 3	939 ± 13
50×50	758 ± 0	841 ± 27	77 ± 3	841 ± 27
100×100	758 ± 0	705 ± 31	142 ± 3	705 ± 31

Table 2.2: *Computational costs for asynchronous value iteration using two different stopping conditions to compute* 0.001-*optimal policies for the different maze sizes. Note that for value iteration, the number of iterations equals the number of evaluations.*

A typical solution path is shown in Figure 2.2. The path crosses a small number (6) of penalty states. Note that due to the discounting, and the large goal reward, ways round penalty states are often not preferred (this would decrease the discounted bonus of the final goal reward). Thus, we have to be careful of using discounting. The task has become more like finding the shortest path, which also crosses the least number of penalty states, instead of finding the least punishing path to the goal. Note that when the goal reward would have been 0, no penalty states would be crossed by an optimal policy.

Maze Size	Iterations (A)	Cum. Reward (A)	Iterations (B)	Cum. Reward (B)
25 × 25	668 ± 205	939 ± 14	75 ± 1	939 ± 14
50 × 50	759 ± 2	841 ± 27	138 ± 3	841 ± 27
100 × 100	762 ± 4	705 ± 31	264 ± 6	705 ± 31

Table 2.3: *Computational costs for synchronous value iteration to compute* 0.001-*optimal policies for the different maze sizes.*

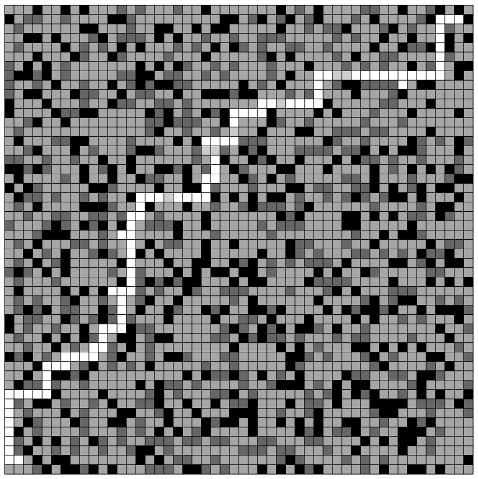


Figure 2.2: *A found solution in a* 50×50 *maze by asynchronous value iteration. The solution path is highlighted, penalty states which occurred on the path are made slightly darker. Note that the agent sometimes leaves the optimal path due to the stochasticity.*

We note that DP algorithms do not scale up badly, although found solutions are not always optimal (caused by early-stopping of the iteration). The number of iterations for computing a policy with value iteration using a cutoff for evaluating the policy does not depend on the size of the state space. Since iterations take more time for larger state spaces (we update each state value), the time needed to compute a policy scales up linearly with the size of the state space since for maze-worlds the number of transitions is linearly dependent on the number of states (although for fully connected state-spaces this would be quadratic).

The number of iterations for policy iteration using the same stopping condition (A) seems to depend on the size of the state-space, though. The computational time for policy iteration in our example is about 64 times longer for the 100 × 100 maze compared to the 25 × 25 maze. This implies that for this particular problem, the algorithms have complexity $O(n^{3/2})$ in which n is the number of states.

Comment. We also used 200 × 200 mazes, with discount factor $\gamma = .99$. However, the computed policies did not achieve to find a direct way to the goal. It seems that even although action punishment was used, the goal was so far away that its reward was "discounted away" and thus the policy was indifferent to different pathways. Using $\gamma = 0.999$ solved the problem, however. Thus, we have to analyse the problem before setting the discount factor — if pathways to the goal are very long, γ should be large or alternatively we could use the average reward criterion (Van der Wal, 1981; Schwartz, 1993; Mahadevan, 1996).[7] Asynchronous value iteration with $\gamma = 0.999$ and stopping condition (B) needed 290 ± 7 iterations and achieved a trial reward of 409 ± 30.

Conclusion. The experimental comparison shows evidence that value iteration outperforms policy iteration. This is to our knowledge an unknown result and is due to the fact that value iteration uses information earlier. Early policy updates cause less initial cyclic behavior of the policy, which makes the policy evaluation process hard. We also saw that we can stop the evaluation once the value function improvement is small, which makes DP much more efficient. Finally, we observed that DP does not scale up badly.

2.4 More Difficult Problems

In this chapter, Markov decision processes and dynamic programming were described. In this section, we describe some difficulties for the practical utility of MDPs and give a short description of some possible methods which allow to (at least partially) deal with these difficulties.

2.4.1 Continuous State Spaces

When the state space is continuous, we have to use a function approximator for representing the value function, since there is not a finite number of states which we can represent in a lookup table. A function approximator maps an input vector (which describes a state) to its value. Combining DP with function approximators was already discussed by Bellman (1961) who proposed using quantization and low-order polynomial interpolation for approximately representing a solution (Gordon, 1995a). There are two methods for combining function approximators with DP algorithms:

(1) We can just approximate the value function by the function approximator. Then we can use a finite set of sampling states for which we perform value iteration (Gordon, 1995b). After computing $B(V(i))$ with the Bellman backup operator for all sampling states i, we learn a new function approximation by fitting these new sample points. Gordon (1995) shows conditions under which the (approximate) value iteration algorithm converges when combined with function approximators.

(2) We can approximate the value function and the reward and transition functions by using state quantization and resampling. Given the probability density function of the state space and the positions of the quantized cells, we can use sampling techniques to estimate the transition function and the reward function. We may use Monte Carlo simulations to sample states from within each cell, select an action, use the MDP-model to find the successor state and map this back to the next active cell. By counting how often transitions occur, we can estimate the transition probabilities and likewise we can estimate the reward function. Then we can use DP on the approximate model.

[7]The average reward criterion (ARC) has other problems, however. E.g., consider an environment with many positive rewards. Using ARC, an agent may quickly go to a terminal state, neglecting larger reward sums obtainable in a trial.

2.4.2 Curse of Dimensionality

Completely evaluating a policy scales up with $O(n^3)$ when n is the number of states. Therefore when the number of states is very large, DP techniques become computationally infeasible.

The number of states scales with the number of dimensions d as $O(n^d)$, where n is the number of values a variable can have in each dimension. When the number of dimensions becomes large the state space will explode, and DP cannot be used anymore with tabular representations. This is known as Bellman's curse of dimensionality. One method to cope with this large number of states is to use function approximators as described above. Other ways to reduce the number of states are:

(1) Only store important states. When the state space is large, most states have probability zero of being visited, and therefore they do not need to be stored. Since the policy moves around on a lower dimensional manifold than the dimension of the state space would suggest (Landelius, 1997), we can just compute a policy for the states which are likely to be visited.

(2) Another way to deal with multiple variables, is to reduce the exponential number of states which result from combining all variables. This can be done in the following two ways:

(a) A state is usually described by all components, but often some components are not very useful. Instead we can decrease the number of variables by using principal component analysis, see e.g. (Jollife, 1986).

(b) We can break states into parts by making independency assumptions between variables and compute separate value functions for the different parts (blocks). For evaluating a complete state, we add the evaluations of the parts. This can be done by models based on Bayesian networks or probabilistic independence networks (Lauritzen and Wermuth, 1989). A similar approach based on CMACs (Albus, 1975a) will be used in Chapter 7.

2.5 Conclusion

In this chapter, we have described dynamic programming methods for computing optimal policies for MDPs. We also evaluated them on a set of maze tasks of different sizes. The experimental results showed the, to our knowledge, novel result that value iteration outperforms policy iteration. The reason for this is that policy iteration wastes a lot of time evaluating cyclic policies. Value iteration updates the policy while evaluating it and in this way uses the information earlier and more efficiently. We also saw that DP can quickly solve problems containing thousands of states and that we have to be careful to discount the future too much, especially in case of large goal rewards. Finally, we discussed some problems which limit the applicability of the MDP framework for real world problems: continuous state spaces and the curse of dimensionality. A final severe limitation of DP is that it needs a model of the MDP. In the next chapter we describe reinforcement learning methods which learn policies and do not need a model.

Chapter 3

Reinforcement Learning

In the previous chapter we showed how dynamic programming can be used to compute optimal policies for Markov decision problems. Although dynamic programming techniques are useful for solving a wide range of problems, they need a world model consisting of the reward and transition functions of the Markov decision problem as input. Since for real world tasks, world models are not *a priori* available, they have to be developed first. Engineering a model is not an easy task however: many interesting problems are very messy (complex and unspecified) (Vennix, 1996), and modeling them is a very complex and time consuming process. For most complex problems a complete model is not even needed, since it is unlikely that the agent (policy) will traverse all states — the agent's subjective world is only an approximation of the objective world.

Reinforcement Learning. Reinforcement learning (RL) provides us with a framework for training an agent by exploring an environment and learning from the outcomes of such trials (Samuel, 1959; Sutton, 1988). In reinforcement learning problems, an agent receives input from the environment, selects and executes an action, and receives reward which tells how good its last action was. The goal of the agent is to select in each state the action which leads to the largest future discounted cumulative rewards. To solve this, RL methods (Watkins, 1989; Bertsekas and Tsitsiklis, 1996; Kaelbling et al., 1996) try out different action sequences and learn how much long term reinforcement the agent receives on average by selecting a particular action in a particular state. These estimated values are stored in the Q-function which is used by the policy to select an action.

Direct vs. Indirect RL. This chapter describes *direct* RL methods which learn the Q-function directly from experienced interactions with the world. Direct methods are opposed to *indirect* RL approaches, which first estimate a world model from the experiences and then use DP-like methods to compute the Q-function using the imperfectly estimated model. This type of model-based approaches will be discussed in the following chapter.

TD(λ) and Q-learning. In the next sections, we will describe the most important RL methods: TD(λ) methods (Sutton, 1988) and Q-learning (Watkins, 1989). These methods are driven by the error or difference between temporally successive predictions or Q-values. There have been some successful applications of these methods: Tesauro showed impressive results with TD-methods by designing TD-Gammon, a program which learned to play backgammon by self-play (Tesauro, 1992). For the knowledge representation he used a feedforward neural network, which was able to generalize well, given the facts that there are about 10^{20} different backgammon positions and that the program reached human expert level after about 300,000 training games. Other successful applications of RL to complex problems were done by Zhang who used TD(λ) with time-delay neural networks (LeCun et al., 1989) for iteratively repairing job-shop schedules (Zhang and Liu, 1996), and by Crites who used a team of Q-learning agents to train a neural network for controlling multiple elevators in a simulated environment (Crites and Barto, 1996).

Outline. Section 3.1 describes the principles of RL. Section 3.2 describes TD(λ) methods (Sutton, 1988). It describes the distinctions between two existing algorithms for dealing with eligibility traces; replacing and accumulating traces (Singh and Sutton, 1996). Section 3.3 describes Q-Learning (Watkins, 1989), and Section 3.4 describes Q(λ)-learning (Peng and Williams, 1996) which combines Q-learning and TD-learning. In Section 3.5, we introduce a more efficient online Q(λ) algorithm (Wiering and Schmidhuber, 1998b), which improves the worst case update complexity for online Q(λ)-learning from O($|S||A|$) to O($|A|$) where $|S|$ is the number of states, and $|A|$ the number of actions.[1] In Section 3.6, we present Team TD(λ), a novel idea for combining trajectories generated by multiple agents, and we will also present the Team Q(λ) algorithm which combines this idea with our Q(λ) algorithm. In Section 3.7, we demonstrate the practical speed up of the novel online Q(λ) algorithm by comparing it to the previous one discussed in (Peng and Williams, 1996) on experiments with 100×100 mazes. Then we also show experimental results with the replace and accumulate traces algorithms, and with our Team Q(λ) algorithm. In Section 3.8, we finish with a short conclusion.

3.1 Principles of Algorithms

In reinforcement learning we have an agent and an environment, and we want to iteratively improve the agent's policy by letting it learn from its own experiences. This is done by testing the policy in the environment, i.e. we let the agent execute its behavior and observe which actions it executes in which states and where it obtains rewards or punishments. Using such experiences consisting of input, action and received reward, we adjust the agent's policy. Testing the policy has to be done with care. If we would simply test policies by always (greedily) selecting the action with the largest value, the agent might repeat the same behavior trial after trial. Although we would perfectly learn the Q-function for this behavior, the agent would not be able to learn from (completely) distinct experiences which may change the value function and the policy.[2] The reason why the agent might repeatedly select suboptimal actions is that alternative optimal actions may be underestimated due to the small number of experiences and unlucky results (biased in a negative way) when they were executed. Thus, we need a certain amount of exploration.

Exploration/Exploitation. For reinforcement learning we must use exploration which exploits the policy but selects non-policy actions as well. If we repeatedly test all actions in all states, we get sufficient experiences to estimate the real evaluation of each action. Exploration actions aim at trying out new experiences which can be used to improve the policy and is opposed to exploiting actions which bring the agent more immediate reward. The exploration/exploitation dilemma (Thrun, 1992) is to find a strategy for choosing between exploration and greedy actions in order to achieve most reward in the long run and will be

[1]This algorithm can also be applied to implement an O(1) algorithm for online TD(λ) learning with a tabular representation of the value function.

[2]For stochastic environments the agent may learn from novel experiences, although for small noise values the probability distribution over trajectories is still very peaked towards the greedy trajectory.

the topic of Chapter 5. In this chapter we assume that there is a given exploration rule X which relies on the policy for choosing the action. The simplest rule, which we use in this chapter is the Max-random exploration rule which selects the greedy action with probability P_{max} and selects a random action (all with equal probability) otherwise.

Online vs. offline RL. We distinguish *online* RL and *offline* RL. Online RL updates modifiable parameters after each visit of a state. Offline RL delays updates until a trial is finished, that is, until a goal has been reached or a time limit has expired. Without explicit trial boundaries offline RL does not make sense at all, because it would not be able to learn from the whole sequence. But even where applicable, offline RL tends to get outperformed by online RL which uses experience earlier and therefore more efficiently (Rummery and Niranjan, 1994). Online RL's advantage can be huge. For instance, online methods that punish actions (to prevent repetitive selection of identical actions) can discover certain environments' goal states in polynomial time (Koenig and Simmons, 1996), while offline RL requires exponential search time for them (Whitehead, 1992).

Now we will focus on different methods for learning from experiences. A reinforcement learning agent starts in a specific starting state (s_1) and repeats the following cycle throughout a trial:

$$\begin{array}{ll} a_t & = \text{Choose an action with policy } \Pi \text{ and exploration rule } X \text{ given } s_t \\ r_t & = \text{Execute the selected action, and receive a scalar reward} \\ s_{t+1} & = \text{Observe the new state} \\ \Pi & = \text{Use the experience to update the policy} \end{array}$$

The last step updates the policy after a new experience (a quadruple $< s_t, a_t, r_t, s_{t+1} >$) is known.

3.1.1 Delayed Reward and Credit Assignment Problem

Reinforcement learning algorithms have to be able to deal with *delayed reward* and the *credit assignment problem*. The problem of delayed reward is that the results of actions are often not immediately known, e.g. in some problems reward is only given after some goal has been reached. From this the problem of credit assignment follows: since we do not have a unique measure of goodness for each independent action, we have to split the final reward signal obtained by a whole sequence of actions in the contributions of single actions, and not all of them are likely to be equally responsible for the final outcome. For example if an agent follows a particular path and finally reaches a goal, it is not immediately clear which actions did and which actions did not bring the agent closer to the goal — see Figure 3.1. Even if there are many trajectories connecting a state and the goal, we cannot be sure which action is optimal in that state as long as the state is not connected to the goal by the optimal trajectory (which possibly consists of parts of all generated trajectories).

3.1.2 Markov Property

In this chapter we will assume that the environment is stationary, which means that the transition and reward functions do not change with time. Furthermore, we assume that the Markov property holds. The Markov property requires that the rewards and probabilities of making transitions to next states after executing an action in a state are fully determined by the

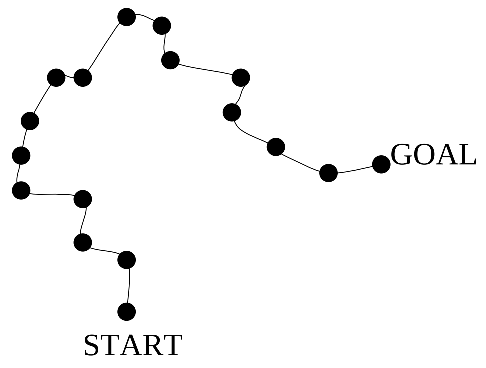


Figure 3.1: *A trajectory going from the start to the goal state. Which actions brought the agent closer to the goal?*

current state/action pair (SAP). More formally, this means that for all possible state/action sequences the following equation holds:

$$P(s_{t+1} = i|s_t, a_t) = P(s_{t+1} = i|s_1, a_1, \ldots, s_t, a_t),$$

where $P(s_{t+1} = i|.)$ is the conditional probability of being in state i at time $t+1$.

3.2 TD(λ) Learning

Temporal difference (TD) learning (Sutton, 1988) can be applied for learning to predict the outcome/results of a stochastic process.

Prediction problems. Suppose we observe a Markov chain (see Appendix A) which emits a reward on each transition and always reaches a terminal (e.g. goal) state. The goal of a prediction problem is to learn the expected total future cumulative reward from each state. More precisely, in prediction problems we receive training data in the form of histories (trajectories) of M trials $H_1, \ldots, H_M$ consisting of sequences of state-reward pairs: $H_i = \{(s_1, r_1), (s_2, r_2), \ldots, (s_{N_i}, r_{T_i})\}$ where s_t is the state at time t, r_t is the emitted reward at time t and T_i is the length of the i^{th} trial. The goal is to learn the expected cumulative future rewards (value) $V(s)$ for each state $s \in S$:

$$V(s) = E(\sum_{i=t}^{T} r_i | s_t = s)$$

where T is a random variable denoting the trial length, and the expectation operator E accounts for the stochasticity of the process including different times of occurrence of the state. Note that a state can be visited zero, one or multiple times during a single trial.

Monte Carlo sampling. We can learn about the outcomes of a Markov chain by using Monte Carlo (MC) sampling which uses multiple simulations of a random process to collect experimental data. Then we can compute the expected outcome by averaging the outcomes of the different independent trials. In what follows, $\eta_i^t(s)$ will denote the indicator function which returns 1 if s occurred at time t in the i^{th} trial, and 0 otherwise. We simply write

$\eta^t(s)$ if the trial number does not matter. In offline or batch-mode, MC sampling with the Widrow/Hoff algorithm (Widrow and Hoff, 1960) estimates the prediction $V(s)$ of the future emitted rewards given that the process is currently in state s as follows:

$$V(s) = \frac{\sum_{i=1}^{M} \sum_{t=1}^{T_i} \eta_i^t(s) \sum_{j=t}^{T_i} r_j^i}{\sum_{i=1}^{M} \sum_{t=1}^{T_i} \eta_i^t(s)}$$

where we wrote r_j^i to take the trial number (i) into account. We can also learn more incrementally (online) by updating the predictions after each trial. The online method learns to minimize the difference between the observed cumulative future reward in the current trial since state s was visited and the current prediction $V(s)$.[3] We minimize this difference by performing gradient descent on the squared error function E. We define the error of the visit of s at time step t, $E_t(s)$ as:

$$E_t(s) = \frac{1}{2}(V(s) - \sum_{i=t}^{T} r_i)^2|_{\eta^t(s)=1}$$

To minimize this error, we first differentiate $E_t(s)$ with respect to $V(s)$:

$$\frac{\partial E_t(s)}{\partial V(s)} = (V(s) - \sum_{i=t}^{T} r_i)|_{\eta^t(s)=1}$$

From this we construct the update rule which decreases this error using iterative steps with learning rate α_s. The learning rate influences the size of update steps and can be used to guarantee convergence. We will use $\alpha_s = 1/k$ where k is the number of occurrences of state s:

$$\Delta V(s) = -\alpha_s(V(s) - \sum_{i=t}^{T} r_i)|_{\eta^t(s)=1}$$

The problem of learning on the MC (Widrow/Hoff) estimates is that the variance in the updates can be large if the process is noisy, and therefore many trials are needed before the confidence intervals of the estimates become small (the variance goes down with the square-root of the number of state occurrences). The variance can be reduced by using information contained in the structure of the generated state sequence. E.g. suppose a visit of some state is always followed by a visit of one particular state (which can also follow other states). When the first state is visited 10 times and the second 100 times, we could compute the first state's value by using the value of the second state since our experiences tell us with certainty that we will make a transition to this state. .The second state's estimate is computed over 100 trials and thus a more reliable estimate than the MC estimate of the first state which is an average of the results of 10 trials.

3.2.1 Temporal Difference Learning

We can reduce the variance in the updates by using temporal difference (TD) learning. TD methods do not only use future rewards, but also the current predictions of successive states

[3]For online learning the values $V(s)$ are initialized before learning to some value such as 0.

for updating state predictions. Temporal difference methods such as Q-learning and TD(λ) methods enforce a time continuity assumption (Pineda, 1997): states which are visited in the same time interval should predict about the same outcomes.[4] In essence, TD methods try to minimize the difference between predictions of successive time steps.

TD(0) updating. Instead of adjusting a state's prediction on the cumulative reward received during the entire future, we can also just learn to minimize the difference between the current state's prediction and the reward which immediately follows the visit of the state plus the next state's prediction. The TD(0) algorithm, which is by the way very similar to Q-learning is exactly doing this. The learning algorithm uses temporal differences of successive state values to adapt the predictions. Let's define $V(s_t)$'s TD(0)-error e_t as:

$$e_t = (r_t + V(s_{t+1}) - V(s_t)) \tag{3.1}$$

This can be simply used to adjust $V(s_t)$:

$$\Delta V(s_t) = \alpha_{s_t} e_t$$

TD(0) has the largest bias inside the TD-family, since the learning signal is very much correlated with the current state. Its variance is small however, since in general we may expect to have only small TD(0) errors. Sutton showed (1988) that by repeatedly learning from sequences with TD(0), the estimates asymptotically converge to the true expected outcome.

TD(λ) methods. TD(λ) methods are parameterized by a value λ and although they use the entire future trajectory for updating state predictions, rewards and predictions further away in the future are weighted exponentially less according to their temporal distance by the parameter λ. TD(λ) methods have been proved to converge with probability 1 to the true prediction for all λ (Dayan, 1992; Dayan and Sejnowski, 1994; Jaakkola et al., 1994; Tsitsiklis, 1994; Pineda, 1997) provided each state is visited by infinitely many trajectories and the learning rates diminish to zero at a suitable rate (see Appendix B). TD(λ) uses $\lambda \in [0, 1]$ to discount TD-errors of future time steps. The TD(λ)-error e_t^λ is defined as:

$$e_t^\lambda = \sum_{i=0}^{T-t} \lambda^i e_{t+i}, \quad \text{and} \quad \Delta V(s_t) = \alpha_{s_t} e_t^\lambda \tag{3.2}$$

where we use e_{t+i} as defined in equation 3.1. It is important to set the λ parameter correctly for efficient learning, since it determines the tradeoff between the bias and the variance. If λ is larger, the variance of update steps is larger since the cumulative rewards computed over many steps will generally occupy a much larger range of outcomes than single-step rewards and the values of successive states.

The TD(λ) errors above cannot be computed as long as TD(0) errors of future time steps are not known. We can compute them incrementally, however, by using eligibility traces (Barto et al., 1983; Sutton 1984; Sutton 1988).

Eligibility traces. Given a new observation, we can immediately update all predictions of states visited previously in the same trial. In Figure 3.2 we show an example of a simulated Markov chain. After each new visit of a state, a new TD(0)-error becomes known. The figure shows how these TD-errors are backpropagated to previous states. To memorize which states

[4]This time continuity assumption holds for Markov processes and follows from the rules of causality and the direction of time. For non-Markov processes, the assumption does not hold and thus we should be careful using TD-learning for them.

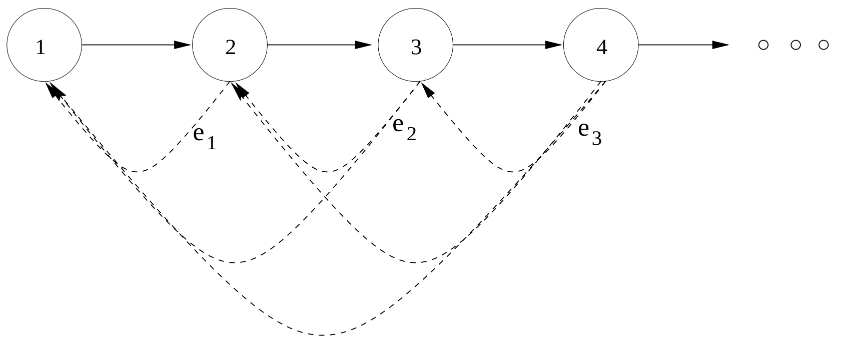


Figure 3.2: *A simulated Markov chain going through states 1,2,3,4,.... The figure shows how TD-errors are backpropagated to all previous states once new states are encountered.*

occurred and how long ago they occurred, eligibility traces are used (Barto et al., 1983). The eligibility of a state tells us how much a state is eligible to learn from current TD-errors. This value depends on λ, the recency of its occurrence and the frequency of its occurrence. Omitting the learning rate α_s for simplicity, $V(s)$'s increment for the complete trial is:

$$\begin{aligned}\Delta V(s) &= \sum_{t=1}^{T} e_t^{\lambda}\eta^t(s) \\ &= \sum_{t=1}^{T}\sum_{i=t}^{T} \lambda^{i-t} e_i \eta^t(s) \\ &= \sum_{t=1}^{T}\sum_{i=1}^{t} \lambda^{t-i} e_t \eta^i(s) \\ &= \sum_{t=1}^{T} e_t \sum_{i=1}^{t} \lambda^{t-i}\eta^i(s) \\ &= \sum_{t=1}^{T} e_t l_t(s) \qquad (3.3)\end{aligned}$$

Where at the last step we simplified the expression by introducing the symbol $l_t(s)$; the accumulating eligibility trace for state s:

$$l_t(s) = \sum_{i=1}^{t} \lambda^{t-i}\eta^i(s)$$

Computing this trace can be implemented using the following recursive form:

$$\begin{aligned} l_{t+1}(s) &\leftarrow \lambda l_t(s) && \text{if } s_t \neq s \\ l_{t+1}(s) &\leftarrow \lambda l_t(s) + 1 && \text{if } s_t = s \end{aligned}$$

Thus, at each time step eligibility traces of all states decay, whereas the currently visited state has its trace increased. This has the effect that only states which have been visited recently are made eligible to learn on future examples.

Now the online update at time t becomes (with learning rate α_s):

$$\forall(s,a) \in S \times A \;\; do: \;\; V(s) \leftarrow V(s) + \alpha_s e_t l_t(s)$$

The eligibility trace can be thought of as an attentional trace. As long as the eligibility trace of some state is "on" (larger than 0), this state will have its value adapted on TD errors of new experiences. The above method uses an accumulating eligibility trace: after each visit of a state its eligibility trace gets strengthened by adding 1 to it.

3.2.2 Replacing Traces

A different method is to replace traces when a state is revisited. Replacing traces (Singh and Sutton, 1996) resets the trace if a state is revisited:

$$\begin{aligned} l_{t+1}(s) \leftarrow &\; \lambda l_t(s) \quad \text{if} \;\; s_t \neq s \\ l_{t+1}(s) \leftarrow &\; 1 \quad \text{if} \;\; s_t = s \end{aligned}$$

The effect is that only recency determines the eligibility trace and that frequency does not matter. To show the difference between the accumulate and replace traces algorithms, we will look at the case $\lambda = 1$ and will analyze the bias of the estimators using a simple example following (Singh and Sutton, 1996).

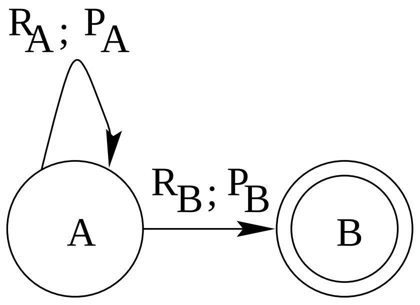


Figure 3.3: *A Markov chain consisting of two states (starting state A and terminal state B). Transition probabilities are denoted as P_A and P_B, transition rewards are denoted as R_A and R_B.*

Consider the Markov chain of Figure 3.3. It consists of two states; the starting state A and the terminal state B. In A, the transition to the next state is stochastic: a step is made to A and B with probability P_A and P_B, respectively. The chain starts at A, we assume $V(B) = 0$ and we are interested in computing the value $V(A)$.

Optimal value. The optimal value $V(A)$ can easily be computed according to:

$$V(A) = P_A(R_A + V(A)) + P_B(R_B + V(B))$$

which has the following solution:

$$V(A) = \frac{P_A}{P_B} R_A + R_B \tag{3.4}$$

Replacing traces/First visit MC. Singh and Sutton used $\lambda = 1$ for the analysis (which is the worst-case for accumulating traces — for $\lambda = 0$ both methods perform the same updates). For this value, it can be shown (Singh and Sutton, 1996) that replacing traces implements a first-visit Monte Carlo method given batch updates (offline) learning, and proper learning rate annealing. This method estimates the value of a state by simply summing the rewards received during a trial after the first time the state was visited. This means that in case of multiple occurrences of the state during a trial only the first one matters. For the Markov chain in Figure 3.3, the first visit MC algorithm (note that for this Markov chain, the replace traces algorithm would make the same updates for all values of λ) would compute $V_f(A)$ as follows given a single trial:

$$V_f(A) = R(A) + R(A) + \cdots + R(A) + R(B)$$

for which the expectancy over all possible trajectories is (Singh and Sutton, 1996):

$$E\{V_f(A)\} = R_B + P_B R_A \frac{P_A}{(1 - P_A)^2} = R_B + \frac{P_A}{P_B} R_A \tag{3.5}$$

When we compare equations 3.4 and 3.5, we can see that the first visit MC algorithm computes a correct (unbiased) estimate. The estimate over multiple trials is simply computed as the average of the estimates for the single trials and thus it is also an unbiased estimate.

Accumulating traces/Every visit MC. The every visit MC method computes the same updates as the accumulating trace algorithms for $\lambda = 1$ and proper learning rate annealing. It computes the value $V(a)$ for the Markov chain in Figure 3.3 as follows given a single (i^{th}) trial:

$$V_e(A) = \frac{R(A) + 2R(A) + \cdots + kR(A) + (k+1)R(B)}{k+1} = \frac{w_i}{k_i + 1} \tag{3.6}$$

Considering all possible trials and weighting them according to their probability of occurring, the estimate after one trial is:

$$E\{V_e(A)\} = R_B + R_A \frac{P(A)}{2P(B)}$$

Thus, the estimate for a single trial is biased, with $BIAS = E\{V_e(A)\} - V(A) = -R_A \frac{P(A)}{2P(B)}$. The estimate over multiple trials is computed as:

$$V_e(A) = \frac{\sum_{i=1}^{M} w_i}{\sum_{i=1}^{M}(k_i + 1)} \tag{3.7}$$

its bias after M trials is (Singh and Sutton, 1996):

$$-\frac{2}{M+1} \frac{P_A}{2P_B} R_A \tag{3.8}$$

and thus the bias goes to 0 when the number of trials M goes to ∞. At the first glance it may seem strange that the bias goes to 0. However, if we have multiple trials we will get many different outcomes, and due to reasons of combinatorics (e.g. there are 2 possibilities to have in one trial 2 and in another trial 3 revisits of state A, whereas there is only one possibility to have two times zero revisits) our estimate is less determined by a single outcome with an overestimated probability (immediately going to state B).

The analysis of Singh and Sutton furthermore shows that initially the mean squared error (Bias + Variance) is larger for the first visit MC algorithm, but that it will catch up in the long run. This implies that for difficult problems replacing traces is the better candidate.

3.3 Q-Learning

One of the simplest reinforcement learning algorithms for learning to control is Q-learning (Watkins, 1989; Watkins and Dayan, 1992). Q-learning enables an agent to learn a policy by repeatedly executing actions given the current state as input. At each time step the algorithm uses 1-step lookahead to update the currently selected state/action pair (SAP). Q-learning updates all SAPs along a solution-path a single time, thus moving the final reward for a trial one step back in the chain. Therefore it takes a lot of time before the goal-reward will be propagated back to the first SAP. E.g., for learning a solution-path costing 100 steps, naive Q-learning needs to traverse the path for at least 100 trials involving 10,000 SAP-updates. Although slow, Q-learning has been proven to converge to the optimal policy provided all state/action pairs are tried out infinitely many times and the learning rate is properly annealed (Watkins and Dayan, 1992). The Q-learning update-rule is as follows:

```
Conventional Q-learning($s_t$, $a_t$, $r_t$, $s_{t+1}$):
1) $e'_t \leftarrow (r_t + \gamma V(s_{t+1}) - Q(s_t, a_t))$
2) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(s, a) e'_t$
```

Here $V(i) = \max_a Q(i, a)$, γ is the discount factor which assigns preference to immediately obtaining reward instead of postponing it for a while, $\alpha(s, a)$ is the learning rate for the k^{th} update of SAP (s, a), and e'_t is the temporal difference or TD(0)-error, which tends to decrease over time.

Learning rate adaption. The learning rate is determined by $\alpha_k(s, a)$ which should be decreased on-line, so that it fulfills two specific conditions for stochastic iterative algorithms (Watkins and Dayan, 1992; Bertsekas and Tsitsiklis, 1996). The conditions on the learning rate $\alpha(s, a)$ are:

(1) $\sum_{k=1}^{\infty} \alpha_k(s, a) = \infty$, and

(2) $\sum_{k=1}^{\infty} \alpha_k^2(s, a) < \infty$.

Learning rate adaptions for which the conditions are satisfied may be of the form : $\alpha_k = \frac{1}{k^\beta}$, where k is a variable that counts the number of times a state/action pair has been updated. In Appendix B, we present a (to our knowledge novel) proof that for $\frac{1}{2} < \beta \leq 1$, both conditions hold.

3.4 Q(λ)-learning

Q(λ)-learning (Watkins, 1989; Peng and Williams, 1996) is an important reinforcement learning (RL) method. It combines Q-learning and TD(λ). Q(λ) is widely used — it is generally believed to outperform simple one-step Q-learning, since it uses *single* experiences to update evaluations of *multiple* state/action pairs (SAPs) that have occurred in the past.

Q's TD(λ) error. First note that Q-learning's update at time $t+1$ may change $V(s_{t+1})$ in the definition of $e'_t = (r_t + \gamma V(s_{t+1}) - Q(s_t, a_t))$. Following Peng and Williams (1996),[5] we define the TD(0)-error of $V(s_{t+1})$ as:

$$e_{t+1} = (r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1}))$$

[5]The difference with Watkins' Q(λ) (1989) is that he proposed $e_{t+1} = (r_{t+1} + \gamma V(s_{t+2}) - Q(s_{t+1}, a_{t+1}))$.

Q(λ) uses a factor $\lambda \in [0,1]$ to discount TD-errors of future time steps:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(s_t, a_t) e_t^\lambda,$$

where the TD(λ)-error e_t^λ is defined as:

$$e_t^\lambda = e_t' + \sum_{i=1}^{\infty} (\gamma\lambda)^i e_{t+i}$$

Here we add e_t' — the difference between the Q-value of the current state/action and immediate reward plus next V-value — to the TD(λ) error of subsequent steps (computed from the differences between a V-value and the immediate reward plus the V-value of the next state).

Eligibility traces. Again we make use of eligibility traces (Barto et al., 1983; Sutton 1988). In what follows, $\eta^t(s,a)$ will denote the indicator function which returns 1 if (s,a) occurred at time t, and 0 otherwise. Omitting the learning rate α for simplicity, $Q(s,a)$'s increment for the complete trial is:

$$\begin{aligned}
\Delta Q(s,a) &= \lim_{k\to\infty} \sum_{t=1}^{k} e_t^\lambda \eta^t(s,a) \\
&= \lim_{k\to\infty} \sum_{t=1}^{k} [e_t' \eta^t(s,a) + \sum_{i=t+1}^{k} (\gamma\lambda)^{i-t} e_i \eta^t(s,a)] \\
&= \lim_{k\to\infty} \sum_{t=1}^{k} [e_t' \eta^t(s,a) + \sum_{i=1}^{t-1} (\gamma\lambda)^{t-i} e_t \eta^i(s,a)] \\
&= \lim_{k\to\infty} \sum_{t=1}^{k} [e_t' \eta^t(s,a) + e_t \sum_{i=1}^{t-1} (\gamma\lambda)^{t-i} \eta^i(s,a)] \qquad (3.9)
\end{aligned}$$

To simplify this we use the symbol $l_t(s,a)$, the eligibility trace for SAP (s,a):

$$l_t(s,a) = \sum_{i=1}^{t-1} (\gamma\lambda)^{t-i} \eta^i(s,a),$$

and the online update at time t becomes:

$$\forall (s,a) \in S \times A \;\; do: \;\; Q(s,a) \leftarrow Q(s,a) + \alpha[e_t' \eta^t(s,a) + e_t l_t(s,a)]$$

We can see that this update is done for all SAPs. This makes online updating very slow. Fortunately, there are methods to circumvent having to update all SAPs. One of them only needs O($|A|$) updates and will be presented in the next section.

Online Q(λ). We will focus on Peng and Williams' algorithm (PW) (1996), although there are other possible variants, e.g, (Rummery and Niranjan, 1994). PW uses a list H of SAPs that have occurred at least once. SAPs with eligibility traces below $\epsilon \geq 0$ are removed from H. Boolean variables $visited(s,a)$ are used to make sure no two SAPs in H are identical.

```
PW's Q(λ)-update($s_t, a_t, r_t, s_{t+1}$) :
1) $e'_t \leftarrow (r_t + \gamma V(s_{t+1}) - Q(s_t, a_t))$
2) $e_t \leftarrow (r_t + \gamma V(s_{t+1}) - V(s_t))$
3) For each SAP $(s, a) \in H$ Do :
      3a) $l_t(s, a) \leftarrow \gamma\lambda l_{t-1}(s, a)$
      3b) $Q(s, a) \leftarrow Q(s, a) + \alpha e_t l_t(s, a)$
      3c) If $(l_t(s, a) < \epsilon)$
            3c-1) $H \leftarrow H \setminus (s, a)$
            3c-2) $visited(s, a) \leftarrow 0$
4) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha e'_t$
5) $l_t(s_t, a_t) \leftarrow l_t(s_t, a_t) + 1$
6) If $(visited(s_t, a_t) = 0)$
      6a) $visited(s_t, a_t) \leftarrow 1$
      6b) $H \leftarrow H \cup (s_t, a_t)$
```

Comments. 1. The SARSA algorithm (Rummery and Niranjan, 1994) replaces the right hand side in lines (1) and (2) by $(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$.

2. For "replacing eligibility traces" (Singh and Sutton, 1996), step 5 should be: $\forall a: \ l_t(s_t, a) \leftarrow 0; \, l_t(s_t, a_t) \leftarrow 1$.

3. Representing H by a doubly linked list and using direct pointers from each SAP to its position in H, the functions operating on H (deleting and adding elements — see lines (3c-1) and (6b)) cost $O(1)$.

Complexity. Deleting SAPs from H (step 3c-1) once their traces fall below a certain threshold may significantly speed up the algorithm. If $\gamma\lambda$ is sufficiently small, then this will keep the number of updates per time step manageable. For large $\gamma\lambda$ PW does not work as well: it needs a sweep (sequence of SAP updates) after each time step, and the update cost for such sweeps grows with $\gamma\lambda$. Let us consider worst-case behavior, which means that each SAP occurs just once. Near trial begin the number of updates increases linearly until at some time step t some SAPs get deleted from H. This happens as soon as $t \geq \frac{\log \epsilon}{\log(\gamma\lambda)}$. Since the number of updates is bounded from above by the number of SAPs, the total update complexity increases towards $O(|S||A|)$ per update for $\gamma\lambda \to 1$.

3.5 Fast Q(λ)-learning

There have been several ways to speed up Q(λ). Lin's *offline* Q(λ) (1993) creates an action-replay set of experiences after each trial. Computing the TD(λ)-returns can be done linearly in the length of the history. However, offline Q(λ)-learning is in general much less efficient than online Q(λ). Cichosz' *semi-online* method (1995) combines Lin's offline method and online learning. It needs fewer updates than Peng and Williams' online Q(λ), but postpones Q-updates until several subsequent experiences are available. Hence actions executed before the next Q-update are less informed than they could be. This may result in performance loss. For instance, suppose that the same state is visited twice in a row. If some hazardous action's Q-value does not reflect negative experience collected after the first visit then it may get selected again with higher probability than wanted.

Our proposed method. Previous methods are either not truly online and thus are likely to require more experiences, or their updates are less efficient than they could be and thus require more computation time. We will present a Q(λ) variant which is truly online and efficient: its update complexity does not depend on the number of states (Wiering and Schmidhuber, 1998b). The method can also be used for speeding up tabular TD(λ) to achieve O(1) update complexity. It uses "lazy learning" (introduced in memory-based learning, e.g., Atkeson, Moore and Schaal 1997) to postpone updates until they are needed. The algorithm is designed for $\lambda\gamma > 0$ — otherwise we can use simple Q-learning.

Main principle. The algorithm is based on the observation that the only Q-values needed at any given time are those for the possible actions given the current state. Hence, using "lazy learning", we can postpone updating Q-values until they are needed. Suppose some SAP (s,a) occurs at (not necessarily successive) time steps $t_1, t_2, t_3, \ldots$. Let us abbreviate $\eta^t = \eta^t(s,a)$, $\phi = \gamma\lambda$. First we unfold terms of expression (3.9):

$$\sum_{t=1}^{k}[e'_t\eta^t + e_t\sum_{i=1}^{t-1}\phi^{t-i}\eta^i] =$$

$$\sum_{t=1}^{t_1}[e'_t\eta^t + e_t\sum_{i=1}^{t-1}\phi^{t-i}\eta^i] + \sum_{t=t_1+1}^{t_2}[e'_t\eta^t + e_t\sum_{i=1}^{t-1}\phi^{t-i}\eta^i] + \sum_{t=t_2+1}^{t_3}[e'_t\eta^t + e_t\sum_{i=1}^{t-1}\phi^{t-i}\eta^i] + \ldots$$

Since η^t is 1 only for $t = t_1, t_2, t_3, \ldots$ and 0 otherwise, we can rewrite this as

$$e'_{t_1} + e'_{t_2} + \sum_{t=t_1+1}^{t_2} e_t\phi^{t-t_1} + e'_{t_3} + \sum_{t=t_2+1}^{t_3} e_t(\phi^{t-t_1} + \phi^{t-t_2}) + \ldots =$$

$$e'_{t_1} + e'_{t_2} + \frac{1}{\phi^{t_1}}\sum_{t=t_1+1}^{t_2} e_t\phi^t + e'_{t_3} + (\frac{1}{\phi^{t_1}} + \frac{1}{\phi^{t_2}})\sum_{t=t_2+1}^{t_3} e_t\phi^t + \ldots =$$

$$e'_{t_1} + e'_{t_2} + \frac{1}{\phi^{t_1}}(\sum_{t=1}^{t_2} e_t\phi^t - \sum_{t=1}^{t_1} e_t\phi^t) + e'_{t_3} + (\frac{1}{\phi^{t_1}} + \frac{1}{\phi^{t_2}})(\sum_{t=1}^{t_3} e_t\phi^t - \sum_{t=1}^{t_2} e_t\phi^t) + \ldots$$

Defining $\Delta_t = \sum_{i=1}^{t} e_i\phi^i$, this becomes:

$$e'_{t_1} + e'_{t_2} + \frac{1}{\phi^{t_1}}(\Delta_{t_2} - \Delta_{t_1}) + e'_{t_3} + (\frac{1}{\phi^{t_1}} + \frac{1}{\phi^{t_2}})(\Delta_{t_3} - \Delta_{t_2}) + \ldots \tag{3.10}$$

This will allow for constructing an efficient online Q(λ) algorithm. We define a local trace $l'_t(s,a) = \sum_{i=1}^{t}\frac{\eta^i(s,a)}{\phi^i}$, and use (3.10) to write the total update of $Q(s,a)$ during a trial as:

$$\Delta Q(s,a) = \lim_{k\to\infty}\sum_{t=1}^{k} e'_t\eta^t(s,a) + l'_t(s,a)(\Delta_{t+1} - \Delta_t) \tag{3.11}$$

To exploit this we introduce a global variable Δ which keeps track of the cumulative TD(λ) error since the start of the trial. As long as SAP (s,a) does not occur we postpone updating $Q(s,a)$. In the update below we need to subtract that part of Δ which has already been used (see equations 3.10 and 3.11). We use for each SAP (s,a) a local variable $\delta(s,a)$ which records the value of Δ at the moment of the last update, and a local trace variable $l'(s,a)$. Then, once $Q(s,a)$ needs to be known, we update $Q(s,a)$ by adding $l'(s,a)(\Delta - \delta(s,a))$. Figure 3.4 illustrates that the algorithm substitutes the varying eligibility trace $l(s,a)$ by multiplying a global trace ϕ^t by the local trace $l'(s,a)$. The value of ϕ^t changes all the time, but $l'(s,a)$ does not in intervals during which (s,a) does not occur.

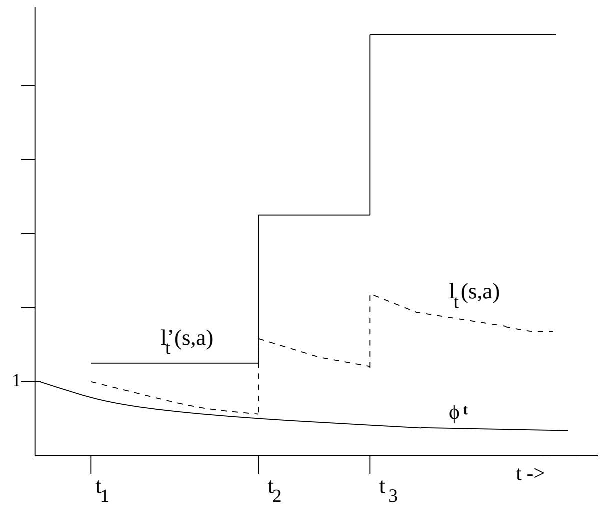


Figure 3.4: *SAP (s,a) occurs at times $t_1, t_2, t_3, \ldots$. The standard eligibility trace $l(s,a)$ equals the product of ϕ^t and $l'(s,a)$.*

Algorithm overview. The algorithm relies on two procedures: the *Local Update* procedure calculates exact Q-values once they are required; the *Global Update* procedure updates the global variables and the current Q-value. Initially we set the global variables $\phi^0 \leftarrow 1.0$ and $\Delta \leftarrow 0$. We also initialize the local variables $\delta(s,a) \leftarrow 0$ and $l'(s,a) \leftarrow 0$ for all SAPs.

Local updates. Q-values for all actions possible in a given state are updated before an action is selected and before a particular V-value is calculated. For each SAP (s,a) a variable $\delta(s,a)$ tracks changes since the last update:

```
**Local Update**($s_t, a_t$) :
1) $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(s_t,a_t)(\Delta - \delta(s_t,a_t))l'(s_t,a_t)$
2) $\delta(s_t,a_t) \leftarrow \Delta$
```

The global update procedure. After each executed action we invoke the procedure *Global Update*, which consists of three basic steps: (1) To calculate $V(s_{t+1})$ (which may have changed due to the most recent experience), it calls *Local Update* for the possible next SAPs. (2) It updates the global variables ϕ^t and Δ. (3) It updates (s_t, a_t)'s Q-value and trace variable and stores the current Δ value (in *Local Update*).

```
Global Update($s_t, a_t, r_t, s_{t+1}$) :
1) $\forall a \in A$ Do
      1a) $Local\ Update(s_{t+1}, a)$
2) $e'_t \leftarrow (r_t + \gamma V(s_{t+1}) - Q(s_t, a_t))$
3) $e_t \leftarrow (r_t + \gamma V(s_{t+1}) - V(s_t))$
4) $\phi^t \leftarrow \gamma\lambda\phi^{t-1}$
5) $\Delta \leftarrow \Delta + e_t\phi^t$
6) $Local\ Update(s_t, a_t)$
7) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(s_t, a_t)e'_t$
8) $l'(s_t, a_t) \leftarrow l'(s_t, a_t) + \frac{1}{\phi^t}$
```

For "replacing eligibility traces" (Singh and Sutton, 1996), step 8 should be changed as follows: $\forall a: \ l'(s_t, a) \leftarrow 0; \ l'(s_t, a_t) \leftarrow \frac{1}{\phi^t}$.

Machine precision problem and solution. Adding $e_t\phi^t$ to Δ in line 5 may create a problem due to limited machine precision: for large absolute values of Δ and small ϕ^t there may be significant rounding errors. More importantly, line 8 will quickly overflow any machine for $\gamma\lambda < 1$. The following addendum to the procedure *Global Update* detects when ϕ^t falls below machine precision ϵ_m, updates all SAPs which have occurred (again we make use of a list H), and removes SAPs with $l'(s, a) < \epsilon_m$ from H. Finally, Δ and ϕ^t are reset to their initial values.

```
Global Update : addendum
9) If ($visited(s_t, a_t) = 0$)
      9a) $H \leftarrow H \cup (s_t, a_t)$
      9b) $visited(s_t, a_t) \leftarrow 1$
10) If ($\phi^t < \epsilon_m$)
      10a) Do $\forall (s, a) \in H$
            10a-1) $Local\ Update(s, a)$
            10a-2) $l'(s, a) \leftarrow l'(s, a)\phi^t$
            10a-3) If ($l'(s, a) < \epsilon_m$)
                  10a-3-1) $H \leftarrow H \setminus (s, a)$
                  10a-3-2) $visited(s, a) \leftarrow 0$
            10a-4) $\delta(s, a) \leftarrow 0$
      10b) $\Delta \leftarrow 0$
      10c) $\phi^t \leftarrow 1.0$
```

Comments. Recall that *Local Update* sets $\delta(s, a) \leftarrow \Delta$, and update steps depend on $\Delta - \delta(s, a)$. Thus, after having updated all SAPs in H, we can set $\Delta \leftarrow 0$ and $\delta(s, a) \leftarrow 0$. Furthermore, we can simply set $l'(s, a) \leftarrow l'(s, a)\phi^t$ and $\phi^t \leftarrow 1.0$ without affecting the expression $l'(s, a)\phi^t$ used in future updates — this just rescales the variables. Note that if $\gamma\lambda = 1$ no sweeps through the history list will be necessary.

Complexity. The algorithm's most expensive part are the calls of *Local Update*, whose total cost is $O(|A|)$. This is not bad: even simple Q-learning's action selection procedure costs $O(|A|)$ if, say, the Boltzmann rule (Thrun, 1992; Caironi and Dorigo, 1994) is used. Concerning the occasional complete sweep through SAPs still in history list H: during each sweep the traces of SAPs in H are multiplied by $l < e_m$. SAPs are deleted from H once their trace falls below e_m. In the worst case one sweep per n time steps updates $2n$ SAPs and costs

O(1) on average. This means that there is an additional computational burden at certain time steps, but since this happens infrequently our method's total average update complexity stays $O(|A|)$.

The space complexity of the algorithm remains $O(|S||A|)$. We need to store the following variables for all SAPs: Q-values, eligibility traces, previous delta values, the "visited" bit, and three pointers to manage the history list (one from each SAP to its place in the history list, and two for the doubly linked list). Finally we need to store the two global variables.

Comparison to PW. Figure 3.5 illustrates the difference between theoretical worst-case behaviors of both methods for $|A| = 5$, $|S| = 1000$, and $\gamma = 1$. We plot updates per time step for $\lambda \in \{0.7, 0.9.0.99\}$. The accuracy parameter ϵ (used in PW) is set to 10^{-6} (in practice less precise values may be used, but this will not change matters much). The machine precision parameter ϵ_m is set to 10^{-16}. The spikes in the plot for fast Q(λ) reflect occasional full sweeps through the history list due to limited machine precision (the corresponding average number of updates, however, is very close to the value indicated by the horizontal solid line — as explained above, the spikes hardly effect the average). No sweep is necessary in fast Q(0.99)'s plot during the shown interval. Fast Q needs on average a total of 13 update steps: 5 in choose-action, 5 for calculating $V(s_{t+1})$, 1 for updating the chosen action, and 2 for taking into account the full sweeps.

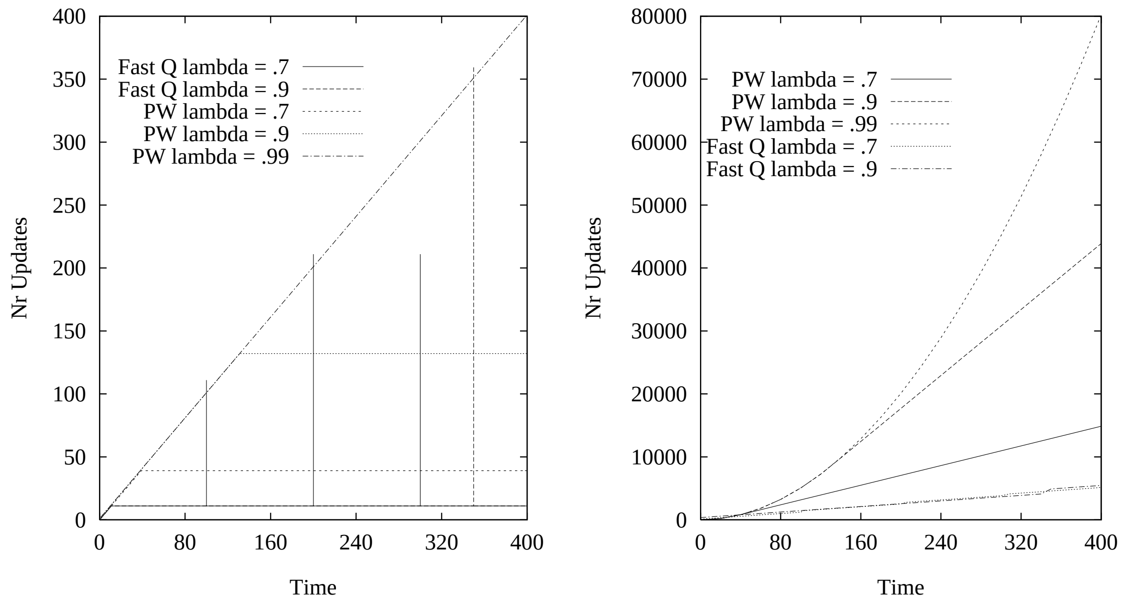


Figure 3.5: *Number of updates plotted against time: a worst case analysis for our method and Peng and Williams' (PW) for different values of λ. (A) Number of updates per time step. (B) Cumulative number of updates.*

Multiple Trials. We described a single-trial version of our algorithm. One might be tempted to think that in case of multiple trials all SAPs in the history list need to be updated and all eligibility traces reset after each trial. This is not necessary — we may use cross-trial learning as follows:

We introduce Δ^M variables, where index M stands for the M^{th} trial. Let N denote the current trial number, and let variable *visited*(s, a) represent the trial number of the most recent occurrence of SAP (s, a). Now we slightly change *Local Update*:

```
Local Update($s_t, a_t$) :
1) $M \leftarrow visited(s_t, a_t)$
2) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(s_t, a_t)(\Delta^M - \delta(s_t, a_t))l'(s_t, a_t)$
3) $\delta(s_t, a_t) \leftarrow \Delta^N$
4) If ($M < N$)
       4a) $l'(s_t, a_t) \leftarrow 0$
       4b) $visited(s_t, a_t) \leftarrow N$
```

Thus we update $Q(s, a)$ using the value Δ^M of the most recent trial M during which SAP (s, a) occurred and the corresponding values of $\delta(s_t, a_t)$ and $l'(s_t, a_t)$ (computed during the same trial). In case SAP (s, a) has not occurred during the current trial we reset the eligibility trace and set *visited*(s, a) to the current trial number. In *Global Update* we need to change lines 5 and 10b by adding trial subscripts to Δ, and we need to change line 9b in which we have to set *visited*$(s_t, a_t) \leftarrow N$. At trial end we reset ϕ^t to $\phi^0 = 1.0$, increment the trial counter N, and set $\Delta^N \leftarrow 0$. This allows for postponing certain updates until after the current trial's end.

3.6 Team Q(λ)

This section contains a novel idea for using multiple agents with eligibility traces. Consider that the value function is shared by a number of agents and that we generate multiple trajectories in parallel to learn the value function. Note that the shared value function does not make it possible for agents to specialize, since the team consists of homogeneous agents. When we immediately use TD(λ) for multiple agents, we will let the agents only learn from their own experiences. However, sometimes it would be a good thing to let agents learn from each other if they interact at some point. For example, in a soccer simulation we could reward an agent for passing the ball to another agent who makes a goal shortly afterwards. In principle, we could design value functions where the value for passing the ball would be determined by what happens with the ball afterwards. This makes a new kind of parallel learning possible in which simultaneously generated trajectories are combined to create new trajectories. In the following, we will not consider such general cases where agents can interact through passing objects or through communication, but will consider the case where two agents are said to interact if their trajectories cross each other. In the framework presented in this section, called Team TD(λ), we extend single agent TD(λ) by linking trajectories generated by multiple agents. Then, if an agent's trajectory traverses another agent's trajectory, previous steps of one agent will be updated on the future of the other agent.

3.6.1 Extending the Eligibility Trace Framework

The accumulate traces algorithm computes large eligibility traces for trials in which a state is visited multiple times. Suppose that some state is visited by two different trajectories. For one trajectory, the state may be visited a single time and for another the state may be visited many times. The accumulating traces algorithm will cause much larger state value adaptions for the second trajectory, even though both trajectories may be equiprobable. Therefore we should sometimes be careful in updating a state value.[6].

Another view on eligibility traces. In principle there are many ways for manipulating the traces and we have already seen a couple of them. In the following we want to extend our reasoning about eligibility traces. An eligibility trace is directly related to the temporal distance between the time step the state was visited and the time step another state is visited.

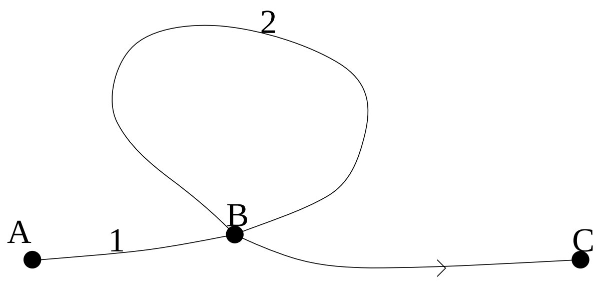


Figure 3.6: *An example trajectory through state space. The Markov chain consists of three (abstracted) states A, B and C. At B, the chain can return to B or go to C.*

Have a look at Figure 3.6. We are interested in the eligibility traces. As we have seen, these influence the amount with which a previous state learns on the current transition in the trajectory. Thus, let's just consider the sequence of states or temporally directed paths through state space. E.g., there can be sequences:

A - B - B - C

A - B - B - B - C

A - B - C

For these different sequences, replacing traces only effects the estimate of B and not of A compared to the accumulating traces. We could also reset the trace from A (to a larger value, e.g. in this case to λ) whenever B is revisited. We could also split a sequence $ABBBC$ into a tree combining the sequences: $ABC + BBB$ or $ABC + BB + BB$. In these ways we would still learn from the transitions numbered 1,2,2, and 3 shown in Figure 3.6. We should note that by transforming a sequence into a tree and then using TD(λ), the updates will in general not be the same for $\lambda > 0$. The second tree, however, is just a projection of the sequence unto a Markov chain, assigning a probability of $\frac{2}{3}$ to transition BB and $\frac{1}{3}$ to transition BC. Such different ways of dealing with eligibility traces could help to strengthen the relationship to full DP techniques (or full-fledged lookahead searches). The updates become more complex, however. Therefore although some improvement of the learning speed may be obtained, this advantage may become a disadvantage due to the larger computational requirements.

3.6.2 Combining Eligibility Traces

A way of extending the eligibility trace framework is to allow for multiple traces. Consider Figure 3.7. There are two trajectories which have been generated by the same underlying

[6]Optimally we use the probabilities of generating a trajectory. Storing probabilities explicitly is done in world models. Similar effects can be obtained by storing and learning from traces in different ways, however

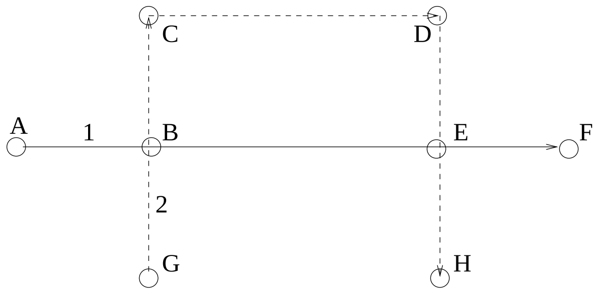


Figure 3.7: *Multiple trajectories through state space. Trajectory one visits states A, B, E, F in that order and trajectory two visits states G, B, C, D, E, and H.*

Markov chain and we want to update all state estimates based on both trajectories. The simplest is to let states visited by one trajectory only learn on future dynamics of that trajectory. In this way, we do not use all possible information, however. If two trajectories meet, there are multiple possible future paths and these can be used to adapt state values. E.g. in figure 3.7, there is a path between state A visited by trajectory 1 and transition $C-D$ visited by trajectory 2, and thus we can adapt state A's value on $C-D$.

The algorithm. The algorithm uses different eligibility traces and history lists for different trajectories and keeps track of the order in which states are visited (i.e. the most recently visited states are the first in the list — if a state is revisited its position in the list advances). We will first introduce some notation:

> $H_i = \{s_1^i, \ldots, s_n^i\}$ is the history list of trajectory i and is an ordered list of all its visited states.
>
> $U(s_i)$ lists all history lists of trajectories which have visited state s_i.
>
> $pred(s_i, H_i)$ lists all states which precede s_i in the history list H_i.

The general way of connecting traces is as follows. Once two trajectories have visited the same state s_m (i.e. they meet in s_m), we traverse the whole list of states which were visited by the trajectories before the meeting state to *connect* them. Thus, after trajectory i has made a step we perform the following operations:

> We insert the new state s_m at the end of trajectory i's history list:
>
> (1) If $s_m \notin H_i$, then $H_i = \text{insert}(s_m, H_i)$.
>
> We check which trajectories have also visited state s_m. The history lists of these trajectories are inserted in the lists $U(s_i)$ of all states s_i previously visited by trajectory i:
>
> (2) $\forall H \neq H_i \in U(s_m)$; $\forall s_i \in pred(s_m, H_i)$; if $H \notin U(s_i)$ then $U(s_i) = U(s_i) \cup \{H\}$.
>
> Finally, we check which other trajectories have visited state s_m. Then we put trajectory i in the lists $U(s)$ of all states s previously visited by one of the other trajectories:
>
> (3) $\forall H \neq H_i \in U(s_m)$; $\forall s \in pred(s_m, H)$; $U(s) = U(s) \cup \{H_i\}$.

Computing eligibility traces. Above we only described which states learn on which trajectories. We will now describe how their eligibility traces are computed. Each state s has an eligibility $l'_i(s)$ to learn on each trajectory i's future steps. In the beginning, we initialize all eligibility traces for all K trajectories and for all states:

$$\forall i, \forall s: \; l'_i(s) = 0$$

If a trajectory visits a particular state, this state's eligibility for that trajectory is updated as in the fast online Q(λ) algorithm. For connecting traces, we also compute new eligibility traces if we insert a trajectory in the list of visited trajectories of some state. Thus, after we perform the operation $U(s) = U(s) \cup \{H_j\}$, where $s \in pred(s_m, H_i)$, we also compute:

$$l'_j(s) = l'_j(s_m)\frac{l'_i(s)}{l'_i(s_m)}$$

Thus, all states visited by one trajectory H_i for which the eligibility trace is 0 for the other trajectory H_j, are assigned an eligibility trace to the trajectory H_j which is calculated by multiplying the eligibility trace of point s_m on the trajectory H_j by $(\lambda\gamma)^d$ where d is the temporal distance along trajectory H_i between state s and state s_m. In this way, they will start learning on the future dynamics from the other trajectory.

To make things more efficient, we traverse the lists backwards and stop once some state already has non zero eligibility trace on the other trajectory. If such a state is only visited once, we can be sure that all previous states also have a non-zero eligibility trace. If a state has been visited multiple times, however, it can be the case that it already has an eligibility on the other trajectory whereas the preceding states have not (see Figure 3.8). Therefore we only stop traversing the list backwards, if we find a state with non-zero eligibility which has not been revisited (see Appendix C for the complete algorithm). If states are revisited often, the procedure can still be quite expensive, since it is possible that we would traverse the whole list. The probability that one agent has revisited a large number of consecutive states is fortunately very low.

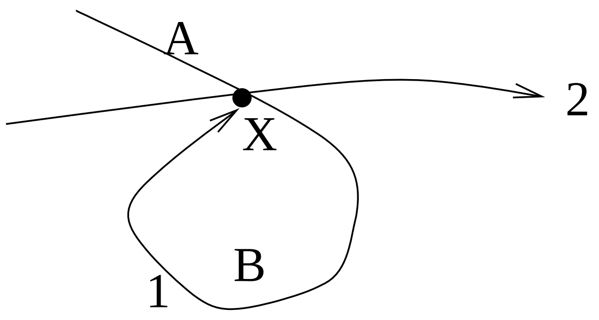


Figure 3.8: *An example of two trajectories crossing each other where we cannot stop tracing trajectory 1 back after noticing that state X already has a non zero eligibility trace on trajectory 2. At the moment that trajectory 1 has hit trajectory 2 in X for the first time, X and A got a non zero eligibility trace on trajectory 2, and therefore B would stay without eligibility if we would stop in X.*

Connecting multiple trajectories. If there are multiple trajectories, things get a little bit more complicated. Now, if trajectory 2 meets trajectory 3, and trajectory 2 has hit trajectory 1 before, we have to follow trajectory 2 and let trajectory 1 states also get eligibility on trajectory 3.

Again, we first insert the new state in the history list of trajectory i: $H_i = \text{insert}(s_m, H_i)$. We will now describe the recursive procedure $Connect(H_i, s_m, H_j)$ which connects all states visited by trajectory i before state s_m to trajectory j. Furthermore it also connects all states visited by another trajectory which traverses one of these states to trajectory j.

```
$Connect(H_i, s_m, H_j)$:
(A) $\forall s_i \in pred(s_m, H_i)$ ; $U(s_i) = U(s_i) \cup \{H_j\}$
(B) $\forall s_i \in pred(s_m, H_i)$ ; $\forall H_k \neq H_i \neq H_j \in U(s_i)$ $Connect(H_k, s_i, H_j)$
```

Then we call this procedure for all pairs which include the new trajectory H_i :
$\forall H \neq H_i \in U(s_m)$ $Connect(H_i, s_m, H)$ and $Connect(H, s_m, H_i)$.

Some limitations and their implications. There are two problems with the approach. (1) The Markov property is essential for the possibility to do multiple trace learning, since otherwise it may not be possible to "jump" from one trajectory to another. (2) The eligibility trace from a state visited by one trajectory on another trajectory is determined by the order in which the trajectories meet. However, the usual eligibility traces have the same problem — we cannot (efficiently) make eligibility traces invariant to the order in which experiences occur.

Comment. The approach resembles the replacing traces in the manner that it deals with multiple crossings of trajectories. For computing eligibility traces for single trajectories we can use the replacing traces algorithm or the accumulating traces. We have used this in an algorithm based on Team Q(λ)-learning which enables us to train a policy with multiple agents (see Appendix C).

Complexity of the algorithm. Note that the new complexity of the local update rule depends on the number of agents (trajectories), i.e. it becomes O(K) for a single update step, where K is the number of agents.

Stopping connecting the lists in the way described above may help to keep the average computational requirements low. When states are not revisited by the same trajectory, we could at most connect each state visited by 1 of the trajectories to all other trajectories. Thus, we could have made TK links after T steps, which makes an average update complexity of $O(K)$. In the worst case states are often revisited so that we need to traverse the whole list backwards. Then the worst case complexity would become $O(|H|K)$, where $|H|$ is the length of the history list. For most problems, agents will not revisit long sequences of states and therefore it may be expected that the update complexity scales up linearly with the number of trajectories. Computationally, we do therefore not expect a large difference in the update complexity by learning from many independent trials or by learning from combining the traces. However, if we plan to use multiple agents and generating trajectories is expensive, there may be a big gain in using the new algorithm, since we learn more on each experience.

3.7 Experiments

In this section, we will describe the following three experimental comparison studies: (1) We compare our novel fast Q(λ) to PW's Q(λ), (2) We compare replacing traces to accumulating traces, and (3) We compare Team Q(λ) to Independent Q(λ).

3.7.1 Evaluating Fast Online Q(λ)

To evaluate the practical performance improvement of the fast Q(λ) method over Peng and William's Q(λ) we use different mazes than in Chapter 2. We created a set of 20 different randomly initialized 100×100 mazes, each with about 20% blocked fields. All mazes share a fixed start state (S) and a fixed goal state (G) — we discarded mazes that could not be solved by Dijkstra's shortest path algorithm (1959). See figure 3.9 for an example maze.

In each field the agent can select one of the four actions: *go north, go east, go south, go west*. Actions that would lead into a blocked field are not executed. Once the agent finds the goal state it receives a reward of 1000 and is reset to the start state. The action execution is without noise, thus the transition function is deterministic (this makes it possible to use larger values for λ). All other steps are punished by a reward of -1. The discount factor γ is set to 0.99.

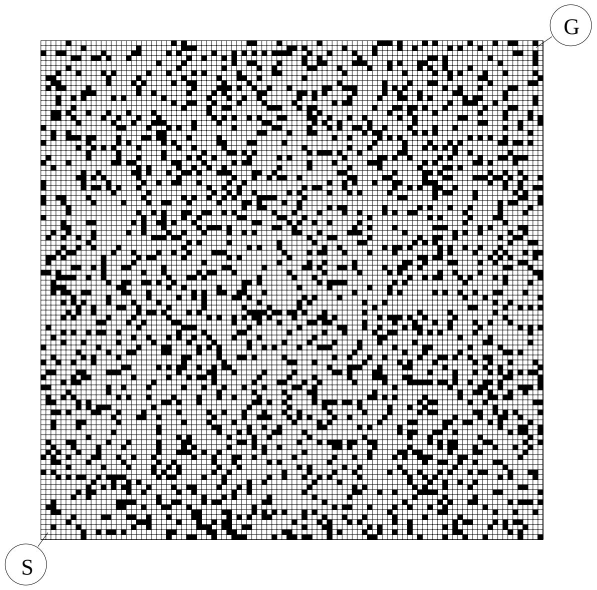


Figure 3.9: *A* 100×100 *maze used in the experiments. Black states denote blocked fields. White squares are accessible states. There are about 8000 accessible states. The agent's aim is to find the shortest path from starting state S to goal state G.*

Experimental set-up. A single run on one of the twenty mazes consisted of 5,000,000 steps. Every 10,000 steps the learner's performance was monitored by computing its cumulative reward so far. We must note that we measure performance as the cumulative reward sum, while the agent learns to maximize a discounted sum. Since all emitted rewards are equal, however, the evaluation criteria measure the same increase/decrease in performance. The optimal performance is about 41,500 = 41.5K reward points (this corresponds to 194-step paths). To select actions, we cannot use the Q-function in a deterministic way and by selecting the action with the largest Q-value, since this quickly leads to repetitive suboptimal behavior. Instead, we used the Max-random exploration rule, which selects an action with maximal Q-value with probability P_{max} and a random action with probability $1 - P_{max}$. Note that this probability P_{max} has nothing to do with the *noise* in the *execution* of actions which stems from the environment. During each run we linearly increased P_{max} from 0.5 (start) until 1.0 (end).

To show how learning performance depends on λ we set up multiple experiments with different values for λ and β (used for annealing the learning rate). As shown in Appendix B, the learning rate annealing parameter β must theoretically lie between 0.5 and 1.0, but may be lower in practice to increase the learning performance. If λ is large and β too small, however, then the Q-function may diverge.[7]

Parameters. We tried to choose the lowest β (and hence the largest learning rate) such that the Q-function did not diverge. Final parameter choices were: Q(0) and Q(0.5) with $\beta = 0$, Q(0.7) with $\beta = 0.01$, Q(0.9) with $\beta = 0.2$, Q(0.95) with $\beta = 0.3$, Q(0.99) with $\beta = 0.4$. PW's trace ϵ was set to 0.001. Larger values scored considerably less well; lower ones costed more time. Machine precision ϵ_m was set to 10^{-16}. Time costs were computed by measuring CPU time (including action selection and simulator time) on a 50 MHz Sun SPARC station.

Results. Learning performance for fast Q(λ) is plotted in Figure 1(A) (results with PW's Q(λ) are very similar). We observe that larger values of λ increase performance much faster than smaller values, although the final performances are best for standard Q-learning and Q(0.95). Figure 1(B), however, shows that fast Q(λ) is not much more time-consuming than standard Q-learning, whereas PW's Q(λ) consumes a lot of CPU time for large λ.

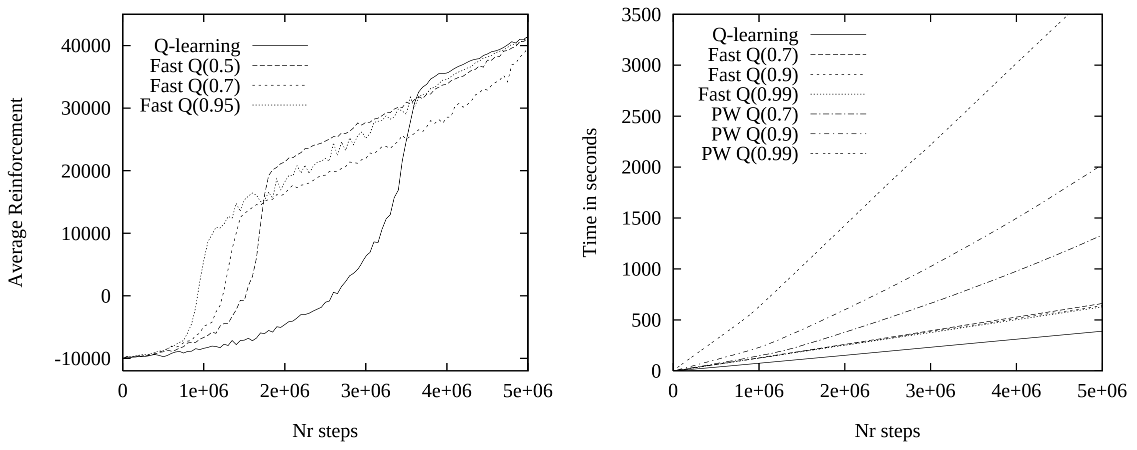


Figure 3.10: *(A) Learning performance of Q(λ)-learning with different values for λ. (B) CPU time plotted against the number of learning steps.*

Table 1 shows more detailed results. It shows that Q(0.95) led to the largest cumulative reward which indicates that its learning speed is fastest. Note that for this best value $\lambda = 0.95$, fast Q(λ) was more than four times faster than PW's Q(λ).

In the table we can also see that Q-learning achieved lowest cumulative reward, although it achieved (almost) optimal final performance. Most Q(λ) methods also achieved near optimal performance. An exception is fast Q(0.7) which performed slightly worse. After an analysis we found that the learning rate decay was not sufficient to keep the Q-function always from diverging. With a larger learning rate decay (0.05), Q(0.7) performed as well as most other Q(λ) methods.

[7]Note that we use a constant learning rate $\alpha = 1$ if we set $\beta = 0$. We can use such a learning rate if the environment is deterministic.

Although fast Q(λ) was 2 to 6 times faster than PW's Q(λ) for the tested values of λ, the difference could be much larger. PW's cut-off method works only when traces decay, which they did due to the chosen λ and γ parameters. For $\lambda\gamma = 1$ (worst case), however, PW would consume about 100 hours(!) whereas our method needs around 11 minutes.

Finally, we should remark that for stochastic environments, the best value for $\lambda = 0.5$. For this value, our algorithm was about 2 times faster than PW's.

3.7.2 Experiments with Accumulating and Replacing Traces

The accumulate traces algorithm is still used most often, although the replace traces algorithm (Singh and Sutton, 1996) may work better for difficult problems. Since Q(λ) is not exact when combined with exploration (Watkins, 1989; Lin, 1993; Peng and Williams, 1996), we also tested resetting λ (to 0.001) if an exploration action was selected which did not lead to an increase of the Q-value after a single step (i.e. if Q-learning would increase the Q-value of the SAP, we do not reset the trace). Thus, we have four methods for using eligibility traces: accumulate/replace with/without resetting traces for exploration actions.

Task set-up. We use the same 50×50 mazes as in the previous chapter. Again we use 20 different generated mazes. We also use noise in the execution of actions: we replace actions by random actions with 10% probability without informing the agent. We can view this noise as a change of the transition function: instead of having deterministic transitions, the transition function becomes stochastic. The new transition function consists of probabilities {0.925,0.025,0.025,0.025} of making a step towards the intended direction and each of the other directions. We have used policy iteration (PI) using the *a priori* model (the exact state-transition probabilities and reward function of the maze) to compute a solution. We stopped the policy evaluation once the maximum difference between state values between two subsequent evaluation steps was smaller than $\epsilon = 0.001$. The policy iteration ended once the policy remained unchanged after a policy improvement step. Computed over the 20 different mazes, the optimal policy receives a cumulative reinforcement of 78.6K $\pm$ 2.0K (1K

System	Final performance	Total performance	Time
Q-learning	41.5K ± 0.5K	4.5M ± 0.3M	390 ± 20
Fast Q(0.5)	41.2K ± 0.7K	8.9M ± 0.2M	660 ± 33
Fast Q(0.7)	39.6K ± 1.4K	7.9M ± 0.3M	660 ± 29
Fast Q(0.9)	40.9K ± 1.0K	9.2M ± 0.3M	640 ± 32
Fast Q(0.95)	41.5K ± 0.5K	9.8M ± 0.2M	610 ± 32
Fast Q(0.99)	40.0K ± 1.1K	8.3M ± 0.4M	630 ± 39
PW Q(0.5)	41.3K ± 0.8K	8.9M ± 0.3M	1300 ± 57
PW Q(0.7)	40.0K ± 0.7K	7.9M ± 0.3M	1330 ± 38
PW Q(0.9)	41.2K ± 0.7K	9.4M ± 0.3M	2030 ± 130
PW Q(0.95)	41.2K ± 0.9K	9.7M ± 0.3M	2700 ± 94
PW Q(0.99)	39.8K ± 1.4K	8.2M ± 0.4M	3810 ± 140

Table 3.1: *Average results for different online Q(λ) methods on twenty 100×100 mazes ($\pm$ 1 standard deviation). Final performance is the cumulative reward during the final 20,000 steps. Total performance is the cumulative reward during a simulation. Time is measured in CPU seconds.*

means 1000) within 10,000 steps. The standard deviation is caused by the stochasticity of the environment.

We tested the methods each 10,000 steps for 100,000 steps. Then we recorded the time step of the first testing trial in which the methods were able to collect 90% of what the PI-optimal policy collected in that particular maze. A total of 1,000,000 steps were allowed during a run. Parameters were as follows: We compare λ-values $\in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, α (learning rate) = 1.0, β (the learning rate decay) = 0.2. For accumulate traces with $\lambda = 0.9$, we used β (the learning rate decay) = 0.3, which gave the best results (we also tested other learning rates and decay rates, but these did not work better). We use Max-random exploration for all methods where the probability of selecting the greedy action was set to $P_{max} = 0.7$ during the entire run.

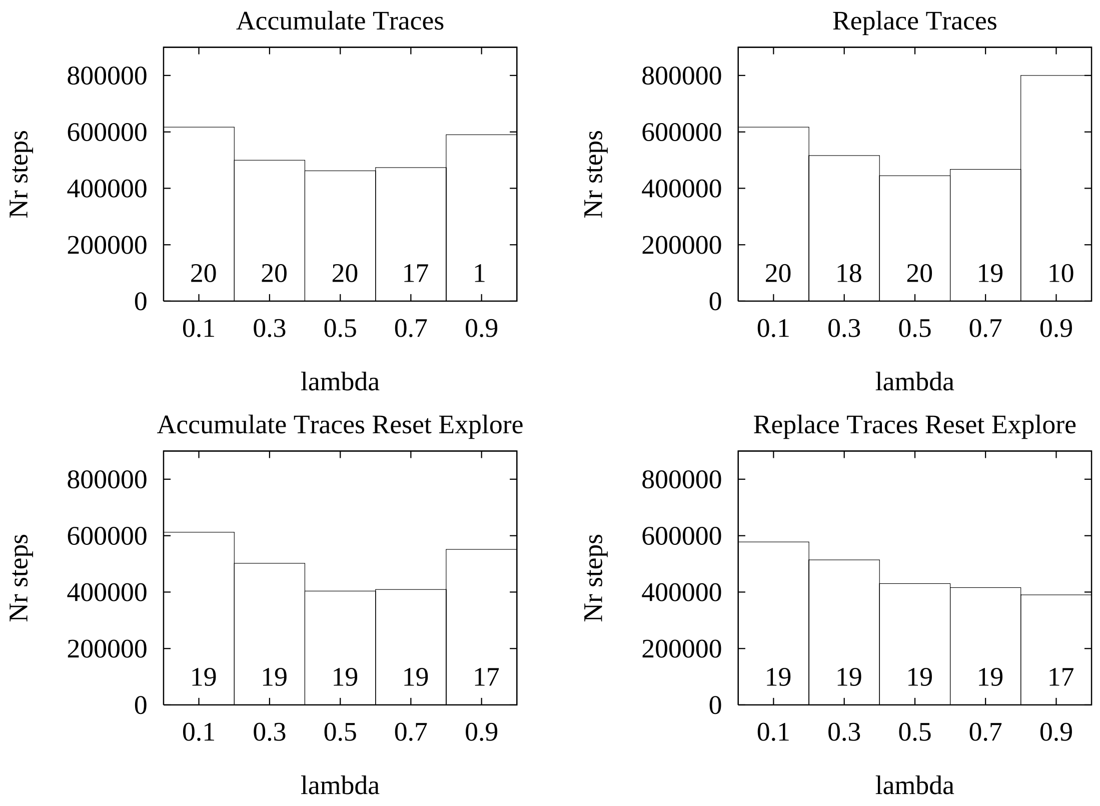


Figure 3.11: *A comparison between different Q(λ)-learning methods. Plots show the number of steps before the policy collected 90% of what the PI-optimal policy collected. Values in the boxes denote the number of successful simulations (out of 20).*

Results. Figure 3.11 shows the results. We observe that resetting traces when selecting exploration actions is useful for making large λ values perform better. Replace traces with resetting exploration traces achieves the most robust performance for the different λ values. $\lambda = 0.5$ gives the best overall performance. Due to the penalty states and noise we cannot use a large constant value for λ.

Conclusion. Managing the traces in different ways effects learning performance. By resetting traces once exploration actions have been executed we can use larger λ values which helps to make more effective use of the eligibility traces. The speed up gained by replacing/resetting traces are not so large, however. Eligibility traces are most useful for large environments in which the policy makes the same steps (or receives more or less the same rewards) over time so that λ can be large. For complex environments, some more efficient eligibility trace management seems necessary. We want to keep λ large, but this is only possible if experiences are not noisy, since otherwise the variance of updates gets very large, which causes large changes of the policy and that makes policy evaluation hard. More efficient managing would therefore try to reduce the noise in the TD-errors. This can be done by using sampling techniques from a single state (e.g. partial roll outs). In this way, we can return more valuable TD-updates, which allows us to send information back to many more states/action pairs. Another possibility is to use variable λ and to detect special landmark states which can be used to return reliable estimates of future rewards.

3.7.3 Multi-agent Experiments

The last experiments in this chapter validate the usefulness of the novel Team Q(λ) algorithm. We use one of the 50×50 mazes from the previous experiment and put multiple agents in this environment which share the same Q-function and policy. The agents select actions one after the other and after each action the global Q-function is updated. If an agent hits the goal, it is reset to the starting position whereas the other agents keep their position. We use 2, 4, 8 and 16 agents and compare the Team Q(λ) algorithm to using multiple agents which only learn from their own experiences (Independent Q(λ)). The Q(λ) method we use is replace traces with $\lambda = 0.9$ combined with resetting traces for exploration actions.

We perform 100 experiments for which we always use the same maze (to keep the variance in our results small). Again we use Max-random exploration and keep $P_{max} = 0.7$. Our objective is again to attain 90% of what the PI-policy attains.

Results. Table 3.2 shows the results of the experiments. We also performed experiments with a single agent.

System/Agents	1	2	4	8	16
Independent	98 (0.44M)	98 (0.42M)	99 (0.43M)	98 (0.46M)	99 (0.48M)
Team Q(λ)		100 (0.43M)	100 (0.43M)	99 (0.45M)	99 (0.50M)

Table 3.2: *(A) A comparison between Team Q(λ) and Independent multi-agent Q(λ). Results show for different numbers of agents (1, 2, 4, 8, and 16) how often a policy was learned which obtained 90% of the optimal policy (and the total number of actions after which it was found. M stands for Million). A total of 100 simulations were performed on the same maze.*

System	1	2	4	8	16
Independent	58 / 2.1M	60 / 2.1M	61 / 2.0M	65 / 1.7M	69 / 1.3M
Team Q(λ)		66 / 2.1M	70 / 2.0M	83 / 1.7M	106 / 1.2M

Table 3.3: *(A) A comparison between Team Q(λ) and Independent multi-agent Q(λ). Results show time requirements in CPU seconds / the cumulative rewards for an entire simulation.*

We note that there are no significant differences between the algorithms. The Team

Q(λ) algorithm attains the 90% level slightly more often, but needs on average more steps. Although Team Q(λ) learns more from agent experiences — for the two agent case it makes 73K additional connections with an average trace of 0.25, and for the 8 player case it makes 570K connections (which equals 57% of all experiences) with average trace 0.24 — this does not mean that these experiences are always used to improve the policy. A reason why the differences are so small is that we reset traces which means that only short parts of the trajectories will be connected.

Table 3.3 shows the time requirements of the multi-agent methods in CPU seconds on a Ultrasparc 170 MHz workstation. Both algorithms require additional time for multiple agents. The reason why Independent Q(λ) needs more time for more agents is due to the more time demanding local update rule. The computational cost of the Team Q(λ) algorithm scales up worse. The reason for this is that connecting traces consumes some time due to the loop in the procedure *connect*.

We can see, however, that using multiple agents with or without connecting traces works well if the number of agents is not too large, since the cumulative rewards obtained by the group of agents stays above 2.0M rewardpoints. The speed up of having multiple agents operating in parallel is linear in the number of experiences for small number of agents, but the gain becomes smaller if the number of agents is larger than 4 or 5, which is caused by the fact that goal-directed information is used later.

We also used the Team Q(λ) algorithm in the deterministic 100×100 maze with λ set to 0.95 and without resetting traces. Here it turned out that Team Q(λ) did perform slightly worse than the independent version. A reason could be that initial exploration was too large (0.5). Another problem with connecting traces is that later trajectories are connected to trajectories which were generated early in the simulation. This causes updates of the value function using old information (bad trajectories) and may therefore harm the policy.

Discussion. For the maze environment Team Q(λ) does not lead to improvements. One of its problems is that new trajectories can be connected to trajectories which were generated long ago by a completely different and possibly bad policy. Therefore some updates may be harmful. To update a specific state many different future trajectories are used. Some of which are created by better policies than others. It is hard to select on which trajectories we want to learn, since the influence of noise makes it impossible to learn only on the best possible future trajectories. This is different in model-based RL, which will be discussed in the next chapter, since in model-based RL we compute a new function based on single transitions which are always sampled from the underlying MDP. In the case of connecting traces, we use a complete trajectory which is not just sampled from the MDP, but uses the policy as well. Although there are ways to improve the algorithm, such as decaying the importance of learning from trajectories generated a longer time ago, the algorithm may in principle be better suited for tasks where the agents cooperatively have to solve a task and where one agent is helped by other agents in its way to reach a specific goal. For such tasks, Team Q(λ) may backpropagate reward to other agents which interacted with the agent in its way reaching the goal.

3.8 Conclusion

We have described the well known reinforcement learning methods temporal difference learning and Q-learning. These RL methods learn from generating trajectories through state/action space. For complex tasks in which many steps should be performed, TD(λ) methods are efficiently combined with Q-learning by our novel fast online Q(λ) algorithm. This method implements exact updates and in contrast to previous online methods, its update complexity is independent of the number of states. Experimental results indicate that single step lookahead methods like Q-learning learn slower than Q(λ) methods. This is especially the case in deterministic, large environments for which often the same large subtrajectories are generated. Therefore the fast online Q(λ) algorithm is very useful for such problems and the experimental results showed that it can significantly speed up learning.

The success of TD(λ) approaches relies heavily on the utility of the eligibility traces. For deterministic environments we can use large values of λ so that eligibility traces remain activated for a long time so that state values are updated using the whole future path. Thus, the estimates may more quickly converge to the true expected results of the Markov chain. For noisy environments or environments where a lot of exploration is needed, the traces are less effective, however. We have seen that in particular noisy maze environments, it is useful to reset traces whenever exploration steps are made. This makes it possible to use larger values of λ. We have also compared replacing traces to accumulating traces and found a small advantage in using replacing traces. This is mainly the case for large values of λ which can cause very large (undesirable) updates in the case of accumulating traces.

We have also developed a new framework for combining eligibility traces in the case of multi-agent Q(λ)-learning. The method makes it possible to adjust Q-values of state/action pairs which occurred in the trajectory of one agent on the future trajectory of another agent if there exists an interaction point between the trajectories. Experimental results on mazes did not show advantages of the Team Q(λ) algorithm compared to using multiple agents which all learn from their own trajectories. Although Team Q(λ) can use experiences more efficiently, it has the difficulty that it should find out which previously generated trajectories are worth to learn from. Some may be generated by bad policies and can only damage the current Q-function.

Another method of using eligibility traces was proposed by Sutton (1995). He introduced TD-methods for dealing with traces on different time scales, which is a step to approaches which learn by dealing with the future in different ways. The decaying λ-trace is also not necessary, we could use a variable λ as well. Thus, we could try to find landmark states which have small variance and let trajectories backpropagate the values of such landmark states. However it is considered to be quite difficult to measure variance of changing functions without a statistical model. Therefore we will forget about eligibility traces for the moment and turn our heads in the next chapter to indirect reinforcement learning methods which use experiences very efficiently by estimating a world model.

Chapter 4

Learning World Models

Although world models are usually not known *a priori*, it is possible to learn the world model from experiences. Learning world models by RL has wide applicability. If we first learn a model and then compute a Q-function, we can significantly improve the learning speed for particular environments. Furthermore, models are useful to improve exploration behavior, to explain causal relationships, to explain why the agent prefers some choice over another,[1] and to report what the agent has been doing most of its time.

This chapter describes how causal models can be learned by monitoring the agent in the environment and how they can be used to speed up learning. These causal models can be represented in different ways. We will concentrate on statistical maximum likelihood models which return a probability distribution over successor states given the current state and selected action. These models are approximations of the underlying transition and reward functions of the MDP and can be used by DP-like methods to compute Q-functions. The model-based RL agent has two goals: (1) to minimize the prediction error of the model so that optimal or near optimal policies may be computed, and (2) to spend most computational resources on the most promising parts of the state/action space to obtain high cumulative rewards with little computational effort.

Outline. We will assume finite MDPs. First we will explain how gathered experiences are integrated into a model. Dynamic programming (DP) techniques could immediately be applied to the estimated model, but online DP tends to be computationally very expensive. The estimated world model is updated after each experience and fully recomputing the policy is not practical for large state/action spaces. Hence, we will use methods which can change the policy much faster. These methods direct state value updates towards the most interesting parts of the space. The first method we will propose uses 1-step lookahead Q-learning (a member of the family of Real Time Dynamic Programming (Barto et al., 1995) algorithms). By using the model Q-learning is significantly improved, but the method does not effectively use the complete model. A second, very efficient algorithm is prioritized sweeping (PS) (Moore and Atkeson, 1993) which only updates SAPs for which there will be large update errors. Finally, we present an alternative prioritized sweeping algorithm, which is quite similar to Moore and Atkeson's PS, but manages the updates in a different, more exact, way.

Other methods, which we will shortly describe in the discussion are Dyna (Sutton, 1990)

[1]If an agent is acting in the real world and uses a world model, it can simulate possible scenario's resulting from a particular action, and use desired or undesired happenings inside these scenario's to explain why it prefers some action.

and Queue-Dyna (Peng and Williams, 1993). These methods learn action models by recording (frequently) observed experiences and replay these experiences to speed up directed reinforcement learning. These methods work well for deterministic environments, but for stochastic environments they have no principled way to deal with multiple successor states. E.g. if only one successor state is replayed, the Q-function will become biased towards it.

4.1 Extracting a Model

In the previous chapter, we updated the Q-function using generated trajectories and made use of traces to track state dependencies for weighing update steps. Now we want to use connections in the state/action model for updating the Q-function. When we store the complete model and we are given a new experience, we can propagate a change of a state value to all other state values using the directed probabilistic graph.

Given a set of examples, we have to choose a model and compute its parameters. To define which model Θ and parameters ζ reflect the experimental data d best, we can use a likelihood function. This likelihood function gives us the probability $P(\Theta;\zeta|d)$ that our model and parameters are the right one given the data. We can rewrite this with Bayes' rule as:

$$P(\Theta;\zeta|d) = \frac{P(d|\Theta;\zeta)P(\zeta|\Theta)P(\Theta)}{P(d)}$$

Where $P(d)$ is a normalizing constant and denotes the probability of generating the data (the complete set of generated experiences). One reason why the likelihood function is of importance is given by the likelihood principle. This tells us that assuming the model is correct, all the data has to tell us about the goodness of parameters is contained in the likelihood function (Box et al., 1994).

In our case, we know which model is the right one, namely a connected state/action transition graph consisting of transition probabilities and rewards. One simple way of inducing these parameters from a set of experiences is to count the frequency of the occurrence of experimental data, which are quadruples of the form (s_t, a_t, r_t, s_{t+1}) received during the interaction with the environment. For this, the agent uses the following variables:

C_{ij}^a = number of transitions from state i to state j after executing action a.
C_i^a = number of times the agent has executed action a in state i.
R_{ij}^a = sum of the rewards received by the agent by executing action a in state i after which a transition was made to state j.

The maximum likelihood model (MLM) consists of maximum likelihood estimates which maximize the likelihood function. We represent the MLM by matrices of transition probabilities and matrices of rewards and estimate the parameters of these matrices by computing the average probabilities over possible transitions and the average reward:

$$\hat{P}_{ij}(a) \leftarrow \frac{C_{ij}^a}{C_i^a} \tag{4.1}$$

$$\hat{R}(i,a,j) \leftarrow \frac{R_{ij}^a}{C_{ij}^a} \tag{4.2}$$

After each experience, the variables are adjusted and then the MLM model is updated. It is easy to see that these parameters maximize the probability of generating the data (single experiences).[2] If observations are without noise (the agent always perceives the real world state), we have a deterministic reward function and the estimated reward for a particular transition $\hat{R}(i,a,j)$ is known (and equal to $R(i,a,j)$) after a single experience. For estimating the transition probabilities, we need to have multiple occurrences of the transition in our experimental data, since there are multiple outcomes (in this case next states) given the occurrence of each state/action pair and we want to have a good estimate of the probability of each possible outcome. To see how many experiences we need to compute fairly precise models, we show the bias and variance of the estimates as a function of the number of times a state/action pair has occurred.

Bias. There is no bias. This is clear, since experiences are directly sampled from the underlying probability distribution and we use the correct model consisting of transition probability and reward matrices and so the estimator is unbiased. Note also that it is a maximum likelihood model. We do not use a prior on the model, but initialize all counters to 0 (this has the effect that initial policy changes can be very large though).

Variance. While updating, each transition probability changes in a stochastic manner. As a measure of the size of update steps, we can use the variance. The variance of $\hat{P}_{ij}(a)$ after n occurrences of the SAP (i,a) is:

$$Var(\hat{P}_{ij}(a)|n) = \sum_{k=0}^{n} (\frac{k}{n} - P_{ij}(a))^2 \binom{n}{k} (1 - P_{ij}(a))^{n-k} (P_{ij}(a))^k = \frac{P_{ij}(a)(1 - P_{ij}(a))}{n}$$

The variance goes to 0 as $n \to \infty$ and so in the limit we have the exact estimate. To give an indication of the number of needed occurrences, assume that $P_{ij}(a) = 0.5$. Then we have to try out the SAP (i,a) 100 times before the standard deviation $Stdev(\hat{P}_{ij}(a)) = 0.05$. For general problems, however, we do not need to know all transition probabilities as accurately, but use exploration to focus on some parts of the state space. Note also that since the policy is computed from the model, learning from new experiences may decrease the performance of the policy as long as the variance is significant. Therefore model-based learning is in principle a stochastic approximation algorithm.

4.2 Model-Based Q-learning

The MLM can be combined with the DP-algorithms from Section 2 to compute a policy. Although the resulting policy may make optimal use of the experiences generated so far, it usually costs a lot of computational time to use DP-algorithms in an online setting, especially when the state-space is quite large. A more efficient way to recompute the policy introduces some bias to choose which computations really need to be performed. One simple, but quite effective method is using one-step Q-learning on the model — which is the simplest version in the family of adaptive Real Time Dynamic Programming methods (Barto et al., 1995). This method updates the Q-value for the current state/action pair after it has been executed and the model has been updated by:

[2]Note that they do not maximize the probability of generating the complete state-trajectories, since many new (spurious) trajectories become possible.

$$Q(i,a) \leftarrow \sum_j \hat{P}_{ij}(a)(\hat{R}(i,a,j) + \gamma V(j)) \tag{4.3}$$

This update rule makes more efficient use of the experiences than standard Q-learning. We can see this as follows. For Q-learning, a particular update, let's say of $Q(i,a)$ (which in our case equals $V(i)$) will only be used to adapt other Q-values if afterwards there occurs another state/action pair (j,b) leading to state i. For model-based learning updates are already used if the model contains a transition from a state/action pair (j,b) to state i. This transition does not have to reoccur. A new occurrence of (j,b) will automatically take $V(i)$'s update into account. Therefore it may need significant fewer experiences, although single update steps are computationally more expensive (the update complexity depends on the number of possible outgoing transitions). Note, however, that for deterministic environments both methods are identical.

Variance of Model-based Q vs Q. We will now verify whether model-based Q-learning also reduces the variance in the updates. The expected size of the update of $Q(i,a)$ after its $(n+1)^{th}$ occurrence with model-based Q-learning is:

$$\sum_j P_{ij}(a)(\frac{C_{ij}(a)+1}{n+1}(R(i,a,j)+\gamma V(j)) + \sum_{k\neq j}\frac{C_{ik}(a)}{n+1}(R(i,a,k)+\gamma V(k)) - Q(i,a))^2$$

If all the neighbors have not been updated in the meanwhile, this can be simplified according to:

$$\begin{aligned}
&\sum_j P_{ij}(a)(\frac{C_{ij}(a)+1}{n+1}(R(i,a,j)+\gamma V(j)) &+& \sum_{k\neq j}\frac{C_{ik}(a)}{n+1}(R(i,a,k)+\gamma V(k)) \\
&\qquad -(\frac{C_{ij}(a)}{n}(R(i,a,j)+\gamma V(j)) &+& \sum_{k\neq j}\frac{C_{ik}(a)}{n}(R(i,a,k)+\gamma V(k))))^2 \\
=&\sum_j P_{ij}(a)(\frac{n-C_{ij}(a)}{n(n+1)}(R(i,a,j)+\gamma V(j)) &-& \sum_{k\neq j}\frac{C_{ik}(a)}{n(n+1)}(R(i,a,k)+\gamma V(k)))^2 \\
=&\sum_j P_{ij}(a)(\frac{1-\hat{P}_{ij}(a)}{n+1}(R(i,a,j)+\gamma V(j)) &-& \sum_{k\neq j}\frac{\hat{P}_{ik}(a)}{n+1}(R(i,a,k)+\gamma V(k)))^2 \\
=&\sum_j P_{ij}(a)(\frac{1}{n+1}(R(i,a,j)+\gamma V(j) &-& Q(i,a)))^2
\end{aligned}$$

For Q-learning where the learning rate anneals as $\frac{1}{n}$ the variance is:

$$\sum_j P_{ij}(a)((\frac{1}{n+1}(R(i,a,j)+\gamma V(j) - Q(i,a)))^2$$

Thus, in case of no updates of any neighbor's state value, the variances are equal. If some neighbors have had their values updated, it depends whether their values moved closer to the previous value of $Q(i,a)$ or not. Usually, in the initial phase all values in the same region of state space are increasing or decreasing together. For such environments, model-based learning is expected to make larger update steps, and thus will have larger variance. Note, however, that Q-learning usually does not anneal the learning rate with $\frac{1}{n}$ and thus may have a (much) larger variance.

4.3 Prioritized Sweeping

We want to use the directed probabilistic graph to efficiently propagate current state-value updates to other state-values. Although model-based Q-learning is a fast way for using the model, it only performs a one-step lookahead which makes information passing between distant states slow. When distant states are connected by high probability trajectories in a state space, a change in the value of the state at the end of the trajectory will cause a corresponding change in the other state. Since model-based Q-learning is not able to do such *deep* information passing quickly, it may need much more experiences than DP algorithms which will execute sufficient updates to propagate the change of a state's value to all other states. Although this always keeps the value function up-to-date, DP performs so many updates for each experience, that it becomes very slow. To speed up dynamic programming algorithms, some management of update-steps should be performed so that only the most useful updates are made. To find out which updates are important, we consider an experience as a newsflash: it brings information to a specific region which is in some way connected to it, but it does not need to be known everywhere (unless it is really big news).

An efficient management method which determines which updates have to be performed is prioritized sweeping (Moore and Atkeson, 1993). This method assigns priorities to updating the Q-values of different states according to a heuristic estimate of the size of the Q-values' updates. The algorithm keeps track of a backward model, which relates states to predecessor state/action pairs. After some update of a state value, the predecessors of this state are inserted in a priority queue. Then the priority queue is used for updating the Q-values for the actions of those states which have the highest priority. For the experiments, we will use a priority queue for which a promote/remove operation [3] takes $O(\log n)$ with n the number of states in the priority queue. Van Emde Boas, Kaas and Zijlstra (1977) describe a more efficient priority queue which only needs $O(\log \log n)$ for a promote/remove operation, but which only handles integer valued priorities in the interval $1, \ldots, n$.

Moore and Atkeson's PS uses a set of lists $Preds(j)$, where the list $Preds(i)$ contains all predecessor state/action pairs (j, a) of a state i. The priority of state i is stored in $\Delta(i)$. When state i has its value updated, the transition from (j, a) to i contributes to the update of $Q(j, a)$. The priority of a state j is the maximal value of such contributions. The algorithm is shown on the next page. The parameter U_{max} denotes the maximal number of updates which is allowed to be performed per update-sweep. The parameter ϵ controls the update accuracy. The algorithm pushes the current state/action pair on the top of the queue (1), then it repeats for all queue items: Remove the top state (step 2.1), update it (steps 2.2, 2.3 and 2.6), store the size of the update in the variable D (step 2.4), reset its priority (step 2.5), assign new priorities to all predecessors of the state (steps 2.7 and substeps), and insert them if their new priority is larger than the threshold ϵ (step 2.7.2.2.1).

[3] The operations are also called insert, delete (usually the top element), and heapify which means reinsert an element once it has been assigned a new priority.

Moore and Atkeson's Prioritized Sweeping:

```
1) Promote the most recent state to the top of the priority queue
2) While $n < U_{max}$ AND the priority queue is not empty
      2.1 Remove the top state $i$ from the priority queue
      2.2 $\forall$ $a$ do:
            2.2.1 $Q(i,a) \leftarrow \sum_j P_{ij}(a)(R(i,a,j) + \gamma V(j))$
      2.3 $V'(i) \leftarrow \max_a Q(i,a)$
      2.4 $D \leftarrow |V(i) - V'(i)|$
      2.5 $\Delta(i) \leftarrow 0$
      2.6 $V(i) \leftarrow V'(i)$
      2.7 $\forall$ $(j,a) \in Preds(i)$ do:
            2.7.1 $P \leftarrow P_{ji}(a)D$
            2.7.2 If $P > \Delta(j)$
                  2.7.2.1 $\Delta(j) \leftarrow P$
                  2.7.2.2 If $P > \epsilon$
                        2.7.2.2.1 Promote $j$ to new priority $\Delta(j)$
      2.8 $n \leftarrow n + 1$
```

Our Prioritized Sweeping. Our implementation of the algorithm uses a set of predecessor lists containing all predecessor states of a state. $|\Delta(i)|$ denotes the priority of state i. This priority is at all times equal to the true size of the update of value $V(i)$ since the last time it was processed by the priority queue. To calculate this, we constantly update all Q-values of predecessor states of currently processed states, and track changes of $V(i)$. Our PS looks as follows:

Our Prioritized Sweeping:

```
1) Update the most recent state $s$:  $\forall$ $a$ do:
      1.1 $Q(s,a) \leftarrow \sum_j P_{sj}(a)(R(s,a,j) + \gamma V(j))$
2) Promote the most recent state to the top of the priority queue
3) While $n < U_{max}$ AND the priority queue is not empty
      3.1 Remove the top state $s$ from the priority queue
      3.2 $\Delta(s) \leftarrow 0$
      3.3 $\forall$ Predecessor states $i$ of $s$ do:
            3.3.1 $V'(i) \leftarrow V(i)$
            3.3.2 $\forall$ $a$ do:
                  3.3.2.1 $Q(i,a) \leftarrow \sum_j P_{ij}(a)(R(i,a,j) + \gamma V(j))$
            3.3.3 $V(i) \leftarrow \max_a Q(i,a)$
            3.3.4 $\Delta(i) \leftarrow \Delta(i) + V(i) - V'(i)$
            3.3.5 If $|\Delta(i)| > \epsilon$
                  3.3.5.1 Promote $i$ to priority $|\Delta(i)|$
      3.4 $n \leftarrow n + 1$
4) Make priority queue empty, but keep the $\Delta(i)$ values
```

In Moore and Atkeson's PS (M+A's PS), states are inserted in the priority queue before their Q-values are updated. In our method Q-values of states are updated before the states are inserted. We do this to have an exact estimate of the update of the state value since it has been used for updating the priority of its predecessors. This makes the priorities exact, whereas M+A's PS calculates priorities based upon the largest single update of a successor state. This means that states are only inserted if at some specific time step their successor state makes a large update step (larger than ϵ). If the update-steps of successor states are smaller but more frequent it may happen that the states are never inserted in the priority queue, although the update could be quite large.[4]

Our version uses priority values of states which are calculated by summing the new change of the state value with the old change ($\Delta(i)$ in step 3.3.4). This has the advantage of being exact, but the disadvantage is that in our method Q-values of all actions for a particular predecessor state have to be updated which is more time consuming.

Clearing the priority queue. Another choice is whether the priority queue should be emptied or not after the update sweep (PS-call). In the algorithm shown above, our method empties the queue after the PS-call (step 4) while keeping the Δ-values of all states. Moore and Atkeson's PS does not empty the queue. The decision whether to empty the queue or not is independent of the selected method. Keeping the queue results in a quite different update sweep, if, for example, at some time step the size of the update of the current state value is small. The states still on the queue are then processed. These states have probably been inserted few time steps ago, but it is also possible that they were inserted while the agent visited different parts of the state space. Updating them costs time and since the updates did not belong to the most important U_{max} updates at the moment they were inserted in the queue, we can argue that we do not want to update them at all. Note that our method keeps the priority values, even when items are not in the priority queue. Then, if their priority is large enough, they will be inserted in the future anyway — being predecessor of some other visited state.

4.4 Experiments

We have evaluated the model-based RL methods described in this chapter by performing a series of maze experiments. The first experiments involve again the same 50×50 mazes used in Chapters 2 and 3. In all experiments selected actions are replaced by random actions with 10% probability. We have used policy iteration using the *a priori* model to compute a solution. We stopped the policy evaluation once the maximum difference between state values between two subsequent evaluation steps was smaller than $\epsilon = 0.001$. Policy iteration needed a total of 6450 ($\pm$ 480) sweeps for computing the policies. This means that about 13 million value update steps were performed. We will call this policy the optimal policy, although it is not guaranteed to be the optimal policy due to the use of the cutoff parameter ϵ (see Chapter 2). Stopping the evaluation prematurely speeds up the process significantly, however.

The optimal solution-paths cost on average a bit more than 100 steps. The optimal policy receives a cumulative reinforcement of 78.6K $\pm$ 2.0K (1K means 1000) within 10,000 steps. Within 2,000 steps, the cumulative reward is 15.7K $\pm$ 0.6K.

[4]This could be overcome by making ϵ small enough, although this would result in inserting many more (unimportant) states as well.

4.4.1 Comparison between Model-based and Model-free RL

We compare Q-learning, Q(λ)-learning, model-based Q-learning and (our) prioritized sweeping. For all Q(λ) experiments, we have used the fast online Q(λ) algorithm presented in Section 3.4.

Experimental setup. To compare the different RL methods, we executed them on 100 different mazes. For all methods we used the Max-random exploration rule, where we linearly increased P_{max} from 0.7 to 1.0 ($P_{max} = 0.7 \rightarrow 1.0$). Thus, in the beginning the agent selects the action with maximum Q-value with probability 70% and at the end of the simulation it always greedily selects the action with largest Q-value. We ran 1,000,000 actions during a trial and recorded the cumulative rewards during intervals of 10,000 (training) actions. After each 100 trials, we also performed test trials to compute the number of steps to reach the goal. Finally, we recorded the approximation of the learned value function by calculating the average distance of state values to the optimal V^*-function computed with policy iteration.

Parameters. For Q-learning, we used $\alpha = 1.0$ and $\beta = 0.1$. For Q(λ)-learning, we use the same learning rate parameters and set $\lambda = 0.5$. For model-based Q-learning there are no additional parameters to set. We have used our version of the prioritized sweeping method with clearing the queue. For PS the maximum number of updates $U_{max} = 100$, and $\epsilon = 1.0$.

Results. Figure 4.1(A) plots the learning performance of the different methods. It clearly shows the advantage of using model-based RL. Both model-based approaches converge much faster (the linear increase of cumulative reward is only due to the decreasing amount of exploration (stochasticity) of the tested policy) and reach much better final performance. The prioritized sweeping method needs the fewest experiences and reaches near-optimal performance levels (record that the optimal policy collects 78.6K $\pm$ 2.0K) points. Q(λ) improves its performance faster than Q-learning although the final performances are equal.

System	Q-learning	Q(λ)	model-based Q	PS
Performance	68K $\pm$ 10K	68K $\pm$ 15K	76K $\pm$ 4.4K	78.7K $\pm$ 2.0K
Time	78 $\pm$ 3	149 $\pm$ 14	277 $\pm$ 17	308 $\pm$ 8
Time for Solution	70	80	40	30

Table 4.1: *Final performance levels, average time needed by the different methods for a single simulation (1,000,000 steps), and the time needed to reach the interval of the highest performance.*

Table 4.1 shows the final performances and time costs of the methods. Time costs were computed by measuring CPU time for the entire simulation (including almost neglectable simulator time) on a 50 MHz Sun SPARC station. Note that the model-based approaches spend more time: their update-rules loop over all outgoing transitions. They could already be stopped much earlier, however, and the last row of the table indicates how much time it costs until the performance reaches the interval of the linear increase in the graphs (which is due to decaying exploration). Thus, we can see that our prioritized sweeping finds the best solution and finds this fastest. Prioritized sweeping only consumes more time during the first trials. When the goal state has been found for the first time, many updates are made. After the goal has been found a couple of times and most states have been visited, only few updates are made after each experience (most updates would be smaller than ϵ).

Approximation error. Figure 4.1(B) shows the approximation errors of the different learning algorithms. It is clear that there is not sufficient exploration to learn good value

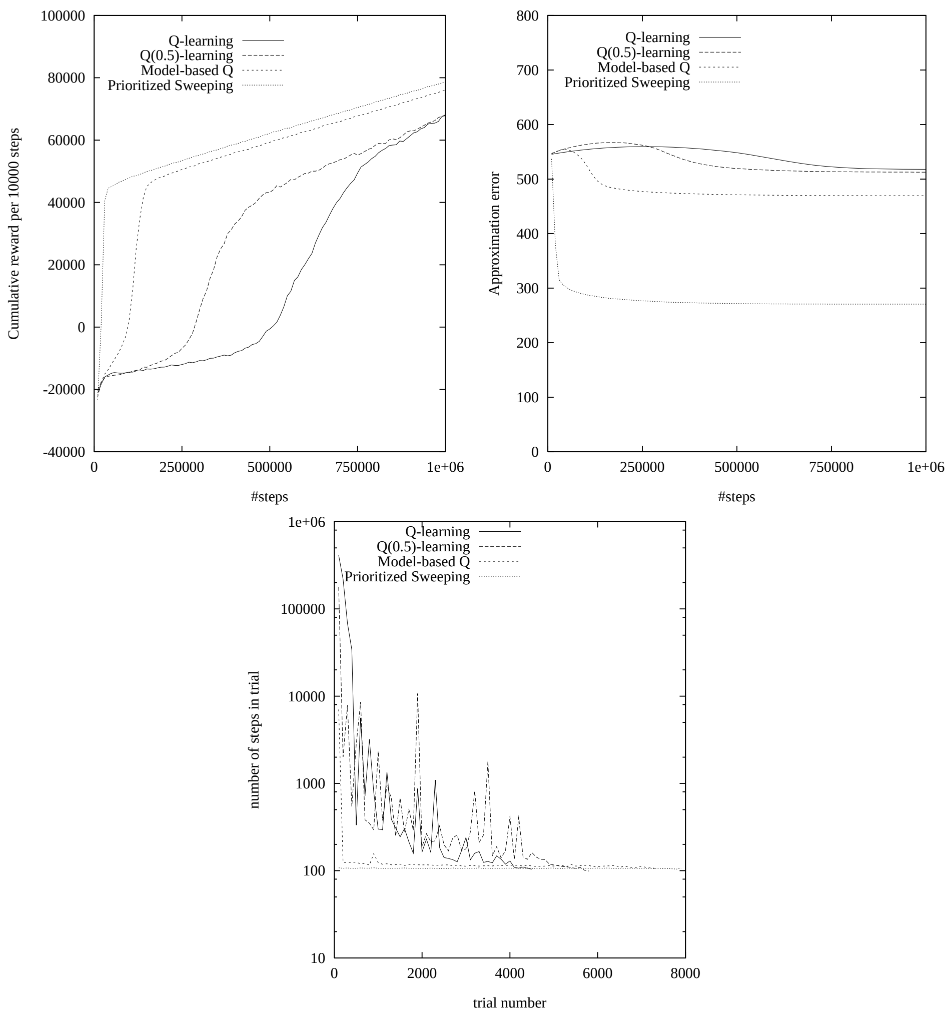


Figure 4.1: *(A) A comparison between the learning performance of the RL methods. The plot shows averages over 100 simulations. (B) The approximation error of the RL methods. The figure plots the average distance of state values to their optimal values against number of agent steps. (C) Average number of steps per test-trial. Note the logarithmic scaling of the y-axis. The peaks in the model-free approaches indicate unsuccessful and long trials.*

function approximations. Only PS learns a reasonable value function approximation. The direct (model-free) RL methods initially increase the approximation error, since they learn negative state values. After this they hardly improve the value function approximation. It may seem surprising that the methods were able to learn reasonable policies at all with such bad value function approximations. This is exactly why (real time) RL methods work: policies only learn to approximate values of states which they *frequently* visit. Thus, we conclude that approximating the performance of the optimal policy tends to be much easier than approximating the optimal value function.

Test trial results. We have performed test trials after each 100 learning trials. We used the greedy policy and measured the number of steps needed to reach the goal. The trials were stopped if the agent could not find the goal within 1,000,000 steps. In Figure 4.1(C) the average number of steps per test trial is displayed. The peaks in the model-free approaches indicate unsuccessful or long trials. The Q(λ) agent did not find the goal in 2 mazes. In 1 of these unsuccessful simulations, the goal had been found a couple of times, but the agent made some update step which ruined its policy. Hence, it could not reach the goal anymore from trial 1900 onwards — the policy must have been caught in cycles through a number of states from which it could not escape even though actions were selected/executed non-deterministically. The Q-learning agent failed for only 1 maze, although its average solutions for the other mazes were worse than for Q(λ) — which we can verify by considering that the average results are the same.

Comment. We also used value iteration to compute the value function while estimating a maximum likelihood model. As expected online VI is computationally very demanding: for simulations consisting of 100,000 steps it required 24,000 CPU seconds (using $\epsilon = 1.0$). Its performance was also not better than that of PS.

More noise. We also tested the effect of more noise in the execution of actions in the same mazes. For this we performed 20 simulations using 20 different mazes and 25% noise in the execution of actions. For Q(λ), the best exploration rate for the Max-random exploration rule is as follows: $P_{max} = 0.7 \rightarrow 1.0$. Figure 4.2(A) shows the results. For Q(λ), accumulate traces with the value 0.1 for λ worked best. Whereas the performance on the same 20 mazes with 10% noise was 7% off from maximum performance, with 25% noise the best method is 21% off. Thus the performance of Q(λ)-learning is quite sensitive to noise!

Figure 4.2(B) shows the results of model-based Q-learning with Max-random exploration with different exploration rates (indicated as (P_{exp}-0.0, where P_{exp} equals $1 - P_{max}$), and with an exploration method which initializes the Q-values to 1000). Model-based Q-learning does not seem to have any problems with the increased amount of noise. It reached stable performance after 140K steps and the final performance is close to optimal. The best performance of 59K is only 2% off optimal!

To study how well prioritized sweeping deals with more noise, we set up experiments with our PS method. We use 20 simulations during which 100K steps can be made. We used $\epsilon = 1.0$ and max-updates = 100. Figure 4.2(C) show the results, exploration rates are indicated as $(1 - P_{max})$ and are annealed to 0. It indicates that PS works very well, although the initial value of $P_{exp} = 1 - P_{max}$ should not be too small (0.1). The best result is attained with $P_{max} = 0.5$ which achieves a final result of 11.5K, which is 4% off the optimal performance (12.0K). We expect slightly worse, but in overall similar results for M+A's PS.

When we compare the three plots, we clearly see that model-based approaches work much better. This is due to the fact that they are much less sensitive to noise than direct RL approaches. Finally, we note that PS reaches its near-optimal performance levels much faster (about 5 times) than model-based Q-learning.

Discussion. Model-free approaches work worse for stochastic environments, since trajectories generated in different trials are not combined in an efficient way. Have a look at Figure 4.3, where we have designed two paths from the start (**S**) to goal (**G**). The solid line is the best known path and the dashed line is the current path. We want to combine the two paths to learn to go from **S** to **A** using the dashed line, and from **A** to **G** using the solid line. Given both lines, model-based RL would easily learn this. Q(λ)-methods, however, cannot learn this in a single pass.[5] The problem is that for the states just before state **A**, Q(λ)-learning

[5]Although learning from many trials solves the problem — single trials should make much larger improve-

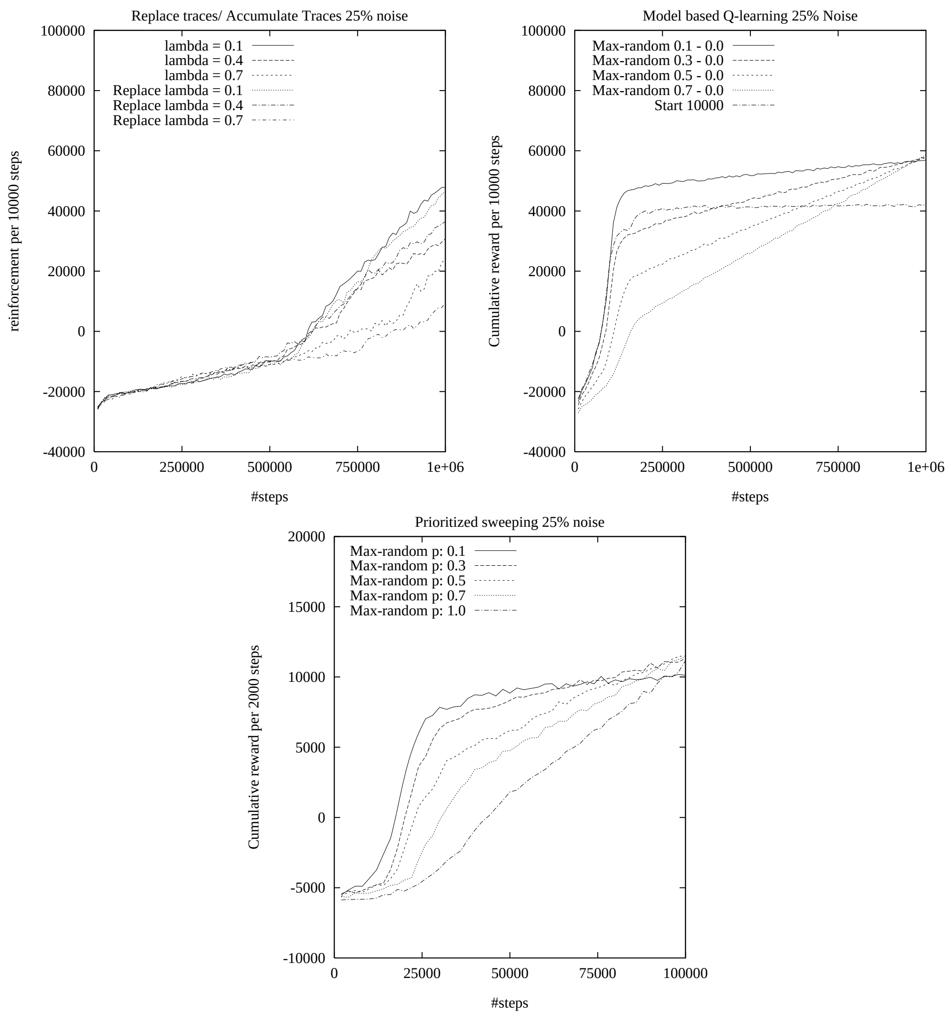


Figure 4.2: *(A) A comparison between different values for λ and the accumulate and replace traces algorithms for $Q(\lambda)$-learning on twenty 50×50 mazes with 25% noise in the execution of actions. (B) A comparison between different amounts of exploration for model-based Q-learning. (C) Different amounts of exploration with prioritized sweeping (note the different scalings of the X- and Y-axis).*

partially learns from **A**'s state value, and partially from the trajectory shown by the dashed line after **A**. When λ is large, the TD(λ)-return from **A** will reach point **S** with a lower value than the Q-value of the action leading to the solid path (since the complete dashed path is worse than the solid path). E.g. for $\lambda = 1.0$, we would compare at each state the entire future solid path to the dashed path and no changes would be made at all. If λ is small, the value from **A** would not be returned to **S**.[6] Prioritized sweeping does not suffer from this problem.

ment steps.

[6]Resetting λ at **A** would solve the problem, but this is difficult for online approaches, since we cannot be certain before the goal is reached whether we should learn from the current trial or not. If we choose to learn, the update steps cannot simply be undone.

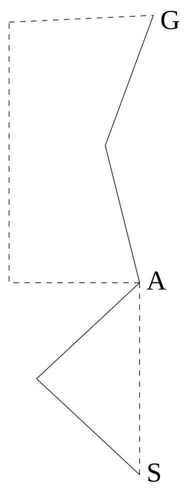


Figure 4.3: *The difficulty for Q(λ)-learning to improve. The solid line is the current best known path and the dashed line is the current path consisting of a good part from start S to point A and a bad part from A to goal G.*

The model-based approaches find near optimal performance levels and are able to learn from less experiences. Prioritized sweeping achieves excellent performance and is able to quickly solve large mazes.

4.4.2 Prioritized Sweeping: Sensitivity Analysis

In this sub-section, we shortly describe parameter setting for prioritized sweeping (PS). Two parameters have to be set, and we have analyzed the sensitivity of the learning behavior for these parameters.

Setting the accuracy parameter

The parameter ϵ controls the level of update accuracy. By setting ϵ to larger values, less updates have to be performed each time step so that the update speed is higher. Of course when ϵ is set to very large values, the value function will be badly fitted to the experiences and this will result in performance loss. We tried out values for ϵ out of the set $\{0.001, 0.01, 0.1, 1, 10, 100\}$. We found that the time requirements depends a lot on the values of ϵ, where values of 1.0 or larger result in very fast performance (below 40 seconds per simulation of 100,000 actions with Max-random exploration). Surprisingly, it is not true that a higher accuracy (lower ϵ) always results in better performance — this may result in "overfitting the value function". This overfitting happens when the model is not very accurate, but is used to compute an accurate value function. If we only have partial data, it can happen that some distant states are connected through high-probability paths, which do not reflect the real MDP very well. Therefore if we have partial data and make too many update steps, the value function can become very biased. Therefore we should use more exploration. The values 0.1 and 1.0 for ϵ

worked best in both performance and time requirements.

Tuning the maximal number of updates

Each update sweep makes a number of updates. In order to increase the update-speed, we have limited the maximal number U_{max} of updates which are allowed to be made per update sweep. Computational costs are less sensitive to the maximal number of updates than to the accuracy parameter, however. We may again "overfit the policy to a wrong model" — when we increase the number of updates the performance does not always improve, but may become worse. If we update less, we will update relatively more on nearby located states, and thus we will less likely update on a specific erroneously estimated trajectory which may introduce a large bias in the Q-estimates.

We found that using 100 updates resulted in the best performance for $\epsilon = 1.0$. For larger problems, having more updates may be necessary to keep the states connected — we may want to assure that each update could result in an update of any other state lying on the current shortest path.

4.4.3 Comparison between PS Methods

Our PS method is more selective in calculating the priorities, since it performs a full parallel backup for determining priorities. Moore and Atkeson's PS considers only one single output transition at a specific time-step for computing the priority. We have set up experiments to compare our PS method to Moore and Atkeson's PS. We test the policies each 1000 steps for 200,000 steps and record when they collect 90% and 95% of what the optimal PI-policy collects. One simulation ends when 100,000 actions have been executed. We combine all methods with Max-random exploration where P_{max} stays 0.7 during the run. We also measure the required CPU time in seconds on a Ultrasparc 170MHz Sun station. For these experiments $\epsilon = 0.1$, and $U_{max} = 1000$.

PS	Clear Q?	steps to 90%	steps to 95%	Final Reward	time
Our	Yes	19K ± 15K (20)	23K ± 16K (20)	157K ± 2K	80 ± 12
Our	No	15K ± 3K (19)	17K ± 3K (17)	155K ± 5K	82 ± 12
M+A	Yes	19K ± 13K (19)	21K ± 16K (17)	155K ± 5K	79 ± 13
M+A	No	16K ± 5K (19)	22K ± 9K (14)	153K ± 6K	98 ± 16

Table 4.2: *Required number of actions for PS methods to find policies for the 50 × 50 mazes which are 90% and 95% off the maximal obtainable cumulative reward. Between brackets we show the number of simulations in which they succeeded.*

Results. The results are shown in Table 4.2. For finding 90% optimal policies, the differences are not very large. However, unlike M+A's PS, our PS with clearing the queue found 95% optimal policies in all 20 mazes. Furthermore, this method also reaches the best final performance. We do not show here that the cumulative rewards for all methods are more or less the same, although there is a small advantage for not clearing the queue. This contradicts the final results (the cumulative reward over the final 20K steps), which shows better results for clearing the queue. The difference can be explained by the fact that not clearing the queue may speed up initial learning since more updates are performed, but as seen before many early updates may be harmful since it may lead to more initial bias in the value function.

Conclusion. Our PS method profits from exactness and therefore it will always work well. We expect that our PS method is currently the best model-based RL method for small to medium-sized discrete MDPs.

4.5 Discussion

Problems of model-based RL. One problem of the methods is that computational memory requirements can be huge: in the worst case we have to store a full model using $O(|S|^2|A|)$ space. On current machines, this becomes infeasible for $|S|$ larger than 5,000. However, for large problems not all transitions are experienced in the limited lifetime of the agent, so that only partial models need to be stored. However, sometimes it is a big problem. Take for example the game of backgammon. For some states, there may be many ($>> 1000$) successor states, because there are many possible actions. For such problems we should only store state transitions of actions which are frequently chosen by the policy or group transitions together if they lead to states with similar values. Of course this results in some loss of accuracy.

Statistical world models. In this chapter we used statistical world models. The use of statistical methods makes it possible to estimate models which can be used with dynamic programming like algorithms. Using statistical learning theory, Kearns and Singh (1998) proved that a variant of a model-based reinforcement learning method converges in the probably approximately correct (PAC) framework. That means that we probably succeed in learning an approximately correct value function after learning on a number of experiences which is bounded by the logarithm of the probability of success, a polynomial of the size of the state/action space, and the inverse of the approximation error.

Other world models include functional models consisting of a set of differential equations which are used in many sciences, e.g., in system dynamics (Vennix, 1996). Such models make cost assessment much harder, however, since we need to employ simulation techniques for predicting the results of actions. The advantage of such models is that they can be quite compact and based on physical properties of the problem. However, there is no systematic way of learning such models — although evolutionary methods (Holland, 1975; Rechenberg, 1971) could be used, there is no guarantee that these methods will find a correct model.

Other world models. Sutton's Dyna systems and relatives (Sutton, 1990; Peng and Williams, 1993) do not estimate a probabilistic transition function, but store trajectories consisting of sequences of experiences. During the course of learning the system may choose to learn from previously stored experiences instead of making real world steps. This is useful for learning in real worlds for which generating new experiences is often much more expensive than "replaying" old experiences. For deterministic environments, these systems work very well. They may be less suited for stochastic environments, however, since updates are less informed and biased to the experiences which are replayed. Instead the PS methods manage the update-sequences, updates are unbiased, and updates are more informed. An advantage of the Dyna approach is that it is easy to replay experiences for all kinds of function approximators.

Jordan and Jacobs (1990) and similarly (Nguyen and Widrow, 1989) describe an approach based on two models: one neural network (the world model) tries to predict the successor state, and another (the action model) is used for selecting actions. Given some discrepancy between the desired state and a predicted state, errors are backpropagated from the world model to the action model so that an action is selected which minimizes that discrepancy. Since it is a global model, however, only single successor states can be learned, but it can generalize well. Schmidhuber (1991) does the same kind of forward modeling with recurrent models and controllers.

Lin (1993) describes experience replay and action models for TD-learning with feedforward networks. For experience replay he records experiences and replays them in backward order. He only records policy actions and not exploration actions, since experiences generated by exploration actions do not reflect the true dynamics generated by the policy. Experience replay was compared to learning action models. For learning action models, he learned to predict the most likely reward and successor state with a feedforward neural network. He found that using experience replay led to larger speedups than learning an action model. He did not estimate a probability distribution over successor states in his action model, however. This would also be very difficult, because he used a partially observable changing environment for which there were hundreds of possible successor states for each state. Feedforward neural networks are not well suited for learning such models due to their static architecture, which makes it difficult to store transition probabilities of a variable number of transitions.

Planning. Planning methods build explicit search trees starting at the goal state as root note (backward planning) or at the start state (forward planning) in order to connect the goal and start states. Most planning mechanisms use heuristic functions and build search trees of a limited length after which they select the step leading to the leave node with the largest value. Davies, Ng and Moore (1998) discuss online search methods for improving a suboptimal value function and showed some improvements in performance at the cost of consuming more time. After the planning phase, they did not adapt the value function, however. It is also possible to use the planning phase for updating the values of states which were traversed during the planning process. This can be simply done by adapting state values to match the results of the planning process (a similar thing is done in (Baxter et al., 1997) which combine planning with TD(λ)). The problem of planning is that it is computationally expensive, especially for non-deterministic environments. We also tried (forward) planning together with adjusting the value function for the planning steps, but found that PS makes it easier to backpropagate goal-related information. Once the goal is found, PS makes many updates so that the agent immediately stores trajectories to the goal state from many other states. For planning, updates are only made if the planning process was successful, and thus if the goal is found by accident, the planning method will not learn how it got there. We may use backward planning instead, but then we already need to know where the goal is, and since the environment is stochastic, it is not clear how we should compute trajectories linking the goal state to the current state.

4.6 Conclusion

This chapter described RL methods which estimate a maximum likelihood model and compute the Q-function using the model. We used experiments to compare different RL methods described so far. We have seen that model-based approaches outperform the model-free RL approaches in finite stochastic mazes. They make more efficient use of experiences. Model-based approaches can profit from management techniques to keep computational requirements for making policy changes small. Prioritized sweeping's management rule: *Focus on the largest errors (problems)* is very effective.

We also showed that the model-based approaches were very resistant against noise. Using a lot of noise in executing actions hardly influenced the final results, whereas this was not true at all for direct RL methods, which suffered a lot from more noise. We consider our mazes to be a prototype of goal-directed stochastic finite MDPs, where the number of paths leading to the goal state, the number of possible choices at each step, the amount of noise in the transition function and the length of the path to the goal are the most important parameters which determine the problem's complexity. We think that for such problems, model-based RL methods will always outperform direct approaches. The difference will be larger if the noise in the transition function is larger and the length of the path to the goal is larger.

Model-based techniques rely more on the Markov property than Q(λ) methods, however, which can learn policies using Monte Carlo estimates for which the Markov property is not required.

Chapter 5

Exploration

A reinforcement learning agent only learns from what it experiences and therefore if it always sees the same world states since it always makes the same decisions, it will not increase its knowledge or performance. Only when an agent would be following an optimal policy, the agent does not need to explore. Otherwise there is always some need to select actions which look suboptimal to the agent. Such actions which deviate from what the agent believes is best (the agent's greedy actions) are called exploration actions. Even though it is true that most exploring actions are indeed not optimal, especially not if the agent already has acquired a lot of knowledge, exploring is always helpful if the agent has still a long future; changing something now may be costly, but will be beneficial for a much longer time and thus it will slowly pay back its costs.

Optimal experimental design (Fedorov, 1972; Dodge et al., 1988) and active learning (Cohn, 1994) try to gather those experiences (data) which are most useful for computing good approximative solutions. In reinforcement learning, the problem of selecting exploration actions is called exploration or dual control (Dayan and Hinton, 1993). Deviating from the current greedy policy (which always selects the action with the highest Q-value), however, usually causes some loss of immediate reinforcement intake. This is usually referred to as the exploration/exploitation dilemma. Therefore, in limited life scenario's the agent usually tries to maximize its cumulative reward over time, and it faces the problem of trying to spend as little time as possible on exploration while still being able to find a highly rewarding policy (Schmidhuber, 1991a; Schmidhuber, 1996). Another goal, however, could be to find the best possible policy in a fixed time or to find a near optimal policy in least time. For such problems, we only care about exploration and thus exploitation can be discarded completely.

Previous work. Thrun (1992) presents comparisons between different *directed* and *undirected* exploration methods. *Directed* exploration methods use special exploration specific knowledge to guide the search through alternative policies. *Undirected* exploration methods use randomized action selection methods to try their luck in generating useful novel experiences. Previous research has shown significant benefits for using directed exploration (see, e.g., Schmidhuber 1991, Thrun 1992, and Storck, Hochreiter and Schmidhuber 1995). Koenig and Simmons (1996) show how undirected exploration techniques can be improved by using the so called action-penalty rule. This rule penalizes actions which have been selected in particular states so that the unexplored actions for some state look more promising — this decreases the advantage of directed exploration.

When we want to solve the exploration/exploitation dilemma, it is also necessary to switch between exploration and exploitation. Thrun and Möller (1992) use a competence map which predicts the controller's accuracy, and their bistable system switches attention between exploration and exploitation depending on expected cost and knowledge gain.

Exploration is also very important for dealing with non-stationary (dynamic) environments. E.g. suppose that we have learned a policy for going from one room to another. Then, if a door between these rooms had always been closed, the policy could not have learned to pass through this door. If the door is opened afterwards, the agent should find this out in order to relearn a better policy for reaching the other room. Dayan and Sejnowski's dual control algorithm (1996) is designed for such environments. They slowly increase Q-values, so that actions which have not been tried out for some time will be selected again.

A completely different exploration approach is described in (Schmidhuber, 1997), where one agent gets reward if she is able to bring the other agent to regions which he does not know. Thus, a co-evolutive exploration behavior takes places, where the agent is presented with a continuous stream of unknown situations until it finally learned a policy for all of them.

The Interval Estimation (IE) algorithm (Kaelbling, 1993) uses second order statistics to detect whether certain actions have a potential of belonging to the optimal policy. IE computes confidence intervals of Q-values and always selects the action with largest upper interval boundary. Previous results (Kaelbling, 1993) show that IE works well for action selection in bandit problems (Berry and Fristedt, 1985). N-armed bandit problems are problems in which we can select between n arms, each one with different payoff probabilities and payoffs. Thus, we have to collect statistics in order to infer which arm to pull. Since pulling arms costs money as well, and we only have limited time, we want to gain information about the payoff ratio's of the arms and infer when we can best focus on the most promising one. Since we are dealing with stochastic problems, IE is interesting for us, and we will describe how we can combine IE with model-based RL.

Outline of this chapter. We first describe undirected exploration methods in Section 5.1. In Section 5.2, we describe directed exploration methods which are based on the design of an exploration reward function. In Section 5.3, we describe model-based exploration and a new exploration method which extends interval estimation (IE) to model-based RL. In Section 5.4, we present experimental comparisons between indirect/direct exploration methods. Section 5.5 concludes this chapter with a discussion.

5.1 Undirected Exploration

In this section we will summarize undirected exploration techniques. These methods usually rely on pseudo-random generators.

5.1.1 Max-random Exploration Rule

The Max-random, also known as pseudo-stochastic (Caironi and Dorigo, 1994) and ϵ-greedy (Sutton, 1996), exploration rule is the simplest exploration rule. The rule uses a single parameter P_{max} which denotes the probability of selecting the action with highest Q-value:

```
1) generate a number $r$ from the uniform distribution [0,1]
2) If $r \leq P_{max}$
```

2.1) select the action with highest Q-value.
2.2) Else select a random action $a \in \{A_1, A_2, \ldots, A_M\}$.

For step 2.1, it can happen that there are multiple actions which have the highest Q-value. In this case, we select one of them stochastically. The Max-random rule has the nice property that the time overhead for selecting an action can be small — in the optimal case where the max-action stays the same, it is O(1).[1] A good method for learning policies is to linearly increase P_{max} after each step during the trial: in the beginning of a simulation a lot of exploration makes it possible to compare many alternative policies, and at the end the probability of selecting greedy actions becomes 100%. Thus, exploration becomes more and more focused around the best found policy.

5.1.2 Boltzmann Exploration Rule

The Boltzmann-Gibbs rule is one of the most widely used exploration rules. The Boltzmann rule assigns probabilities to actions according to their Q-values. It uses a temperature variable T, which is used for annealing the amount of exploration. The Boltzmann exploration rule computes the probability $P(a|s)$ for selecting an action a given state s and Q-values $Q(s,i)$ for all $i \in A$ as follows:

$$P(a|s) = \frac{e^{Q(s,a)/T}}{\sum_i e^{Q(s,i)/T}} \tag{5.1}$$

The Boltzmann rule causes a lot of exploration in states where Q-values for different actions are almost equal, and little exploration in states where Q-values are very different. This is helpful for risk minimization purposes (Heger, 1994), for which we may prefer not to explore actions which look significantly worse than others. However, initially learned Q-estimates can be incorrect due to noise, so some exploration is still needed in case of large differences in Q-values. Using an annealing schedule for the temperature should get around this problem, but finding a good annealing schedule can be quite difficult and is reward function dependent.[2] Furthermore, since the temperature is a global variable, it may happen that some state is not visited for a long time, after which the amount of exploration in this state is very small. Although this last problem could be solved using local temperature variables, it is still a problem that the agent may concentrate on different trajectories which it already knows well.

5.1.3 Max-Boltzmann Exploration Rule

Max-random exploration may lead to bad results when particular actions lead to large negative rewards. It assigns equal probabilities to all non-optimal actions and therefore assigns too large probabilities to explore really bad actions. Boltzmann exploration has large problems focusing on the best actions while still being able to sometimes deviate from them. We may also combine the Max-random and Boltzmann exploration rules. The max-Boltzmann rule combines taking the "Max" action with the Boltzmann distribution:

[1]The action with maximal Q-value can be stored when learning the Q-values.

[2]If we change the reward function by e.g. scaling all rewards, we would interfere with the exploration behavior. This would not be the case for Max-random exploration.

```
1) generate a number $r$ from the uniform distribution [0,1]
2) If $r \leq P_{max}$
        2.1) select the action with highest Q-value.
        2.2) Else select an action according to the probabilities computed by Eq.
5.1.
```

Although the rule requires more parameters to set (the temperature T and P_{max}), there will in general be more good combinations. We will use this rule for exploration with ambiguous inputs (in Chapter 6), since there it is important to be almost deterministic for inputs which are not ambiguous, and to focus on a particular action for an ambiguous input while still sometimes exploring good alternative actions.

5.1.4 Initialize High Exploration

Another possible exploration rule is to start with high initial Q-values (the Q-function is initialized to the upperbound of the optimal Q-function) and to use the normal RL update-rule. Given the fact that action values will drop when they have been explored, taking the action with maximal Q-value will initially return that action which has been selected least times. The method is quite similar to counting the number of times particular state/action pairs have occurred, although it may happen that some actions which have occurred more often are still preferred above other actions since they are more rewarding.

Different kinds of exploration techniques based on this idea have been used used in e.g. (Koenig and Simmons, 1992; Prescott, 1994; Koenig and Simmons, 1996). Koenig (1996) presents a proof that finding a goal at all in particular environments by initializing high and penalizing actions will only take polynomial time, whereas (Whitehead, 1992) had shown that zero-rewarding actions may take exponential time for such worst-case environments. An example of such an environment consists of a simple sequence of states and two actions *go left* and *go right*, and where the action *go right* brings the agent to the successor state and the action *go left* brings the agent all the way back to the initial left-most state. To find the goal the agent has to select only *go right* actions. However, if we do not penalize actions, the probability of selecting a long sequence with only *go right* actions decreases exponentially with the length of the path. Thus, we have to record which actions have occurred to find the goal in polynomial time in the number of states.

5.2 Directed Exploration

For directed exploration, all we need to do is to create an exploration reward function which assigns rewards to trying out particular experiences. In this way, it determines which experience is interesting for gaining information (Storck et al., 1995). Directed exploration methods learn an *exploration value function* in the same way standard RL methods learn a problem-oriented value function. Therefore we may simply define an exploration reward function determining immediate exploration rewards and let the selected RL method learn *exploration Q-values*. We can use the same RL method for learning the exploration Q-function and the exploitation Q-function.

If we want to solve the exploration/exploitation problem, we have to find a method to choose which of the two Q-functions to use for action selection. There are different heuristics for switching from exploration to exploitation. We note that an optimal switching strategy is in general hard to compute, except for the most simple problems such as 2-armed bandit problems (Gittins, 1989; Kaelbling et al., 1996).

5.2.1 Reward Function 1: Frequency Based

Explore actions which have been executed least *frequently.* The (local) reward function is simply:

$$R^E(s, a, *) := -\frac{C_s(a)}{K_C} \tag{5.2}$$

Here $R^E(s, a, *)$ is the exploration reward assigned to selecting action a in state s. It is determined by dividing the local counter $C_s(a)$ by a scaling constant K_C. The asterisk stands for the don't-care symbol. In general the resulting policy will try to explore all state/action pairs uniformly, although the resulting policy still depends on the discount factor. If the discount factor is 1, the policy will try to follow paths leading to transitions with the minimal number of occurrences. Still, the resulting exploration behavior can be quite complex. Let's have a look at Figure 5.1. After a while, the exploration behavior will much more often execute the action *go-right* than *go-up* since the go-up action leads immediately to transitions which are well explored. Note that although this environment is deterministic, the same holds for stochastic environments. Thus, the exploration behavior depends strongly on the topology of the state space. Furthermore, the exploration behavior depends also on the discount factor.

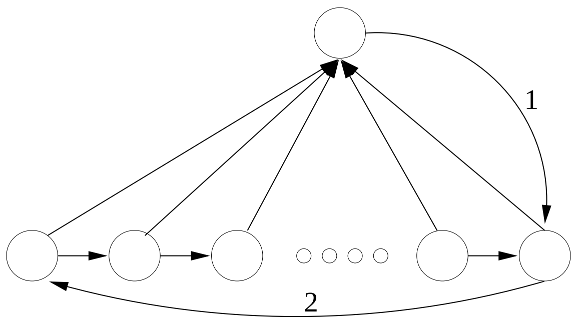


Figure 5.1: *A deterministic environment where the frequency based rule cannot explore all state/action pairs uniformly. The environment consists of a number of states and 2 actions. Actions lead up or right for most states with the exception of the top state for which both actions lead to the same successor. After trying out all up-actions a single time, the explorer becomes aware of the bottleneck transitions (1) and (2) which are crossed so often that up-actions immediately leading to them will get low exploration values and not be considered for a long time.*

5.2.2 Reward Function 2: Recency Based

Select the actions which have been selected least *recently.* The reward for exploring SAP (s, a) is:

$$R^E(s, a, *) = \frac{-t}{K_T}, \tag{5.3}$$

where K_T is a scaling constant and t the current time step. Note that the exploration reward is not entirely local, since it depends on the global time counter. This exploration reward rule makes a lot of sense for dealing with changing environments.

5.2.3 Reward Function 3: Error Based

The third reward rule is as follows:

$$R^E(s_t, a_t, s_{t+1}) = Q_{t+1}(s_t, a_t) - Q_t(s_t, a_t) - K_P \tag{5.4}$$

Here $Q_t(s_t, a_t)$ is the Q-value of the state/action pair (SAP) before the update, and $Q_{t+1}(s_t, a_t)$ is the Q-value of the SAP after the last update, which has been made before computing the exploration reward. The constant K_P ensures that all rewards are negative.[3] This reward rule prefers to keep on selecting state/action pairs which have strongly increasing Q-values. Initially it makes unexplored SAPs prime experiences, due to the negative rewards assigned to all explored steps.

Comments. (1) Schmidhuber (1991) used the absolute difference $|V_{t+1} - V_t|$ instead of the directional change. **(2)** Thrun (1992) combined the error based learning rule with the frequency based learning rule. He did not find any improvements over just using the frequency based rule, however.

5.2.4 False Exploration Reward Rules

We can of course construct all kinds of local exploration reward rules. However, we have to be careful: some rules may look quite good, but may lead to unwanted exploration behavior. To examine whether an exploration reward rule will work, we should analyze the following condition: if we continuously select exploration actions, we should never be absorbed in a subspace of the state space, unless we have gathered so much information that we can be sure that exploring different subspaces will not change our learned policy.

An example of a wrong exploration reward rule is the following: prefer state/action pairs for which the successor states are least predictable, i.e. the transition probabilities have high entropy:

$$R^E(s, a, *) = -\sum_k P_{sk}(a) \log P_{sk}(a) - c \tag{5.5}$$

Here c is a positive constant ensuring that exploration rewards are negative. That no preference should be given to exploring high entropy SAPs can be shown by considering the following counter example.

We have an environment which consists of a number of states connected to each other by a line. The starting state is in the middle of the line. The agent can select the action *go-left* or *go-right*. In 99% of the cases actions are executed properly and in 1% of the cases they are replaced by the opposite action. There is a special state located one step before the extreme right of the line. In this state the action *go-left* results in a completely random transition to the state left or right of it. If exploring with the high-entropy exploration rule, the agent may initially try all actions a couple of times. After a while, it would favor the state at the right, since the exploration reward is highest there. Therefore it will not sample correctly,

[3]We use negative rewards so that we can zero-initialize the exploration Q-function and guarantee that untried SAPs look more promising.

but continuously go to this state and learn its transition probabilities perfectly whereas other transition probabilities are approximated much worse. The exploration reward rule considers the complete randomness as being information carrying, but in this example it is not.

5.3 Learning Exploration Models

We can use all RL methods to learn exploration Q-values. Previous methods used Q-learning for learning where to explore (Schmidhuber, 1991a; Thrun, 1992; Storck et al., 1995). We propose to use prioritized sweeping instead. PS allows for quickly learning exploration models which may be useful for learning Q-values estimating global information gain, taking into account yet unexplored regions of the state-space.

Replacing reward. A nice option in using separate transition reward values for learning exploration functions is that all explorative transition rewards can be based entirely on the current reward. Suppose that an agent selects an action which has already been executed several times. Computing the exploration reward by averaging over all previous transition rewards would not result in the desired reward measure. For instance, with frequency based exploration we would receive rewards: $1, 2, \ldots, C_i(a)$ for the first, second, etc. occurrence of the state/action pair. Our estimated transition reward $\hat{R}^E(i,a,*)$ would be the average over these rewards instead of just equal to $C_i(a)$, the current reward value and therefore some undesired rescaling takes place. Using MBRL, we just replace the estimated reward $\hat{R}(i,a,j)$ by $R^E(i,a,j)$ for all j with $\hat{P}_{ij}(a) > 0$, that is, we update all rewards for outgoing transitions from the current state/action pair to take the latest available information into account.

Never-ending exploration. The exploration utilities continually change — there is not a stable, optimal exploration function. This is not a problem at all, since the goal of exploration is to search for alternative paths in order to find better and better policies, and therefore the exploration policy should never converge. Continuous exploration is useful when one wants to obtain a good final policy without caring for intermediate rewards. For particular problems, we may want to increase or switch to exploitation after some time, however. E.g. in limited lifetime scenarios (Schmidhuber et al., 1996) or bandit problems (Berry and Fristedt, 1985) we want to optimize the accumulative reward over the entire life time. There are a couple of simple ways to get around purely explorative behavior:

(1) Switch to the greedy policy once the value function (of the real task policy) is hardly changed. We could implement this as follows: during TS steps, compute the change of V. If this average change per step is smaller than η, this means that we hardly learn anymore so that we can switch to the greedy policy.

(2) Anneal the exploration rate P_{exp} from 1.0 in the beginning, (always select according to the exploration model) to 0.0 at the end. If the lifetime is unknown, we can try to predict it or repeatedly extend the horizon. The latter is sometimes done in game-playing programs which have fixed "thinking" time for playing an entire game, for which it is never sure how many moves will be played in total.

(3) Start with exploration and switch after a fixed amount of time to exploitation. This is useful for huge state spaces for which we can never visit all states.

5.4 Model-Based Interval Estimation

To explore efficiently, an agent should not repeatedly try out actions that certainly do not belong to the optimal policy. To reduce the set of optimal action candidates we extend the interval estimation (IE) algorithm (Kaelbling, 1993) to combine it with model-based RL.

Standard IE selects the action with the largest upper bound for its Q-value. To compute upper bounds it keeps track of the means and standard deviations of all Q-values. The standard deviation is caused by the stochastic transition function (although it could as well be caused by a stochastic reward function). Although IE seems promising it does not always outperform Q-learning with Boltzmann exploration due to problems of estimating the variance of a changing Q-function in the beginning of the learning phase (Kaelbling, 1993).

Model-Based Interval Estimation. MBIE uses the model to compute the upper bound of Q-values. Given a set of outgoing transitions from SAP (i,a), MBIE increases the probability of the best transition (the one which maximizes $\gamma V(j) + R(i,a,j)$), depending on its standard deviation. Then MBIE renormalizes the transition probabilities and uses the result for computing the Q-values. See Figure 5.2. The left-most transition is the best given the current state. For this transition we have computed the confidence interval, and set its probability to its upperbound (0.6) after which we renormalize the other probabilities.

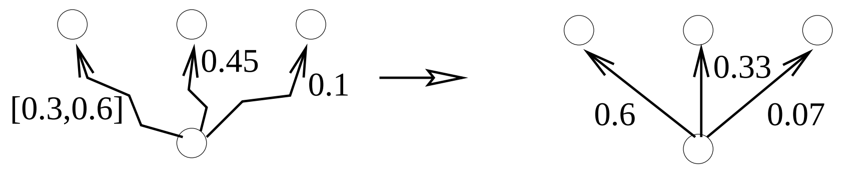


Figure 5.2: *Model-Based Interval Estimation changes the transition probability of the best transition from a state/action pair to its upper probability bound. After this, the other transition probabilities are renormalized.*

The following algorithm can be substituted for lines 1.1 and 3.3.2.1 in our PS algorithm (Section 4.3):

```
Model-Based Interval Estimation($i,a$):
a $m \leftarrow Argmax_{j:\hat{P}_{ij}(a)>0}\{R(i,a,j) + \gamma V(j)\}$
b $n \leftarrow C_i(a)$
c $P \leftarrow \hat{P}_{im}(a)$
d $P^{+}_{im}(a) \leftarrow (P + \frac{z_\alpha^2}{2n} + \frac{z_\alpha}{\sqrt{n}}\sqrt{P(1-P) + \frac{z_\alpha^2}{4n}})/(1 + \frac{z_\alpha^2}{n})$
e $\Delta_P \leftarrow P^{+}_{im}(a) - \hat{P}_{im}(a)$
f $\forall j \neq m$
      g $P^{+}_{ij}(a) \leftarrow \hat{P}_{ij}(a) - \frac{\Delta_P C_{ij}(a)}{C_i(a) - C_{im}(a)}$
h $Q(i,a) \leftarrow \sum_j P^{+}_{ij}(a)(\hat{R}(i,a,j) + \gamma V(j))$
```

Here z_α is a variable which determines the size of the confidence bounds. Step d elaborates on the commonly used $z_\alpha\sqrt{P(1-P)/n}$, and is designed to give better results for small values of n — see (Kaelbling, 1993) for details.

MBIE hybrids. Since MBIE also relies on initial statistics we propose to circumvent problems in estimating the variance by starting out with some other exploration method and switching to IE once some appropriate condition holds.

This is done as follows: we start with frequency based exploration and keep tracking the cumulative change of the problem-oriented value function. Once the average update of the V^{Π} function (computed over the most recent TS time-steps) falls below $\eta \in I\!R^+$, we (I) copy the rewards and Q-values from the problem-oriented model to the exploration model, and (II) switch to IE: we apply asynchronous value iteration to the model; the iteration procedure calls MBIE for computing Q-values and ends once the maximal change of some state value is less than $\epsilon \in I\!R^+$.

Simultaneous policy learning. The model-based learner simultaneously learns both exploration policy and problem-oriented policy. After each experience we update the model and use PS to recompute the value functions. If actions are only selected according to one policy, we can postpone recomputing the other value function and policy and use PS or value iteration once it is required. Note that we can use value iteration here, since we may expect many changes of the model and only need to recompute the value function once in a long while.

Comments. Note that we approximate the upper bound of the probability of making the best transition by assuming the normal distribution. We basically make the distinction between the best transition and any other transition. We could also compute the optimistic value function in a different way: first set all transition probabilities to their lower bounds and then iteratively set the best transitions to their upper bound as long as the probabilities do not sum to 1. See also (Givan et al., 1998) for a description of confidence bounded Markov decision problems or see (Campos et al., 1994) for mathematical methods for computing probability intervals.

5.5 Experiments

In the previous chapter we have used prioritized sweeping with Max-random exploration and saw that it reaches almost optimal performance levels. Still, the attained value function approximations were quite bad which indicates that the state space was badly explored. In this section we perform experiments to find out whether learning exploration models can speed up finding optimal or almost optimal policies. We will first describe experiments in which we compare using exploration models to undirected exploration. Here, the goal is to quickly find good policies. Then we will examine how well the different exploration methods handle the exploration/exploitation dilemma. Finally, in subsection 5.4.3 we will use the exploration models and MBIE for finding very close to optimal policies for a maze consisting of multiple goal states.

5.5.1 Exploration with Prioritized Sweeping

In order to study performance of the directed exploration methods, we combined them with PS and tested them on 20 different mazes of size 25×25, 50×50, and 100×100. The mazes are the same as in previous chapters and contain 10% noise in the action-execution. The 20 randomly generated mazes consist of about 20% blocked states and 20% penalty states.

Parameters and experimental setup. The accuracy parameter $\epsilon = 1.0$ and $U_{max} = 100$. The discount factor γ was set to 0.99 for learning the exploration reward function. We compare the following methods:

(1) Max-random with $P_{max} = 0.5$.

(2) Frequency based. The constant K_C is set to 50.

(3) Recency based. K_T is set to 1000.

(4) Error based. K_P is set to 1000. To get better results with the error based rule, we chose in 50% of the cases the Max-action.

We record how fast the methods are able to learn policies which collect 90% and 95% of what the optimal PI-policy collects. For this, we have tested the optimal policy for 100 times 20,000 steps, and computed the average over 20,000 steps. The learning methods are allowed 100,000 steps for the 25× 25 mazes and 500,000 steps for the other mazes. The agent is tested after each 1,000 steps for 10 times 20,000 steps.

Exploration Rule	25 × 25	50 × 50	100 ×100
Max random	4.6K ± 0.8K	26.2K ± 3.4K	221K ± 80K
Frequency based	3.9K ± 1.1K	17.7K ± 3.5K	88K ± 22K
Recency based	4.6K ± 1.8K	18.2K ± 2.1K	105K ± 12K*
Error based	3.6K ± 0.8K*	23K ± 23K	118K ± 40K

Table 5.1: *Learning exploration models with PS for the different maze sizes. The table shows the number of steps which were needed by the exploration rules to find policies which collect 90% of what the PI-policy collects. Explanation: * = 1 simulation did not meet the requirements.*

Exploration Rule	25 × 25	50 × 50	100 ×100
Max random	11K ± 26K	31K ± 12K	254K ± 108K*
Frequency based	4.7K ± 2.1K	21.0K ± 4.4K	99K ± 29K
Recency based	5.3K ± 1.6K	31K ± 21K	135K ± 48K*
Error based	4.8K ± 1.6K**	44K ± 69K**	161K ± 83K^{*5}

Table 5.2: *Learning exploration models for the different maze sizes. The table shows the number of steps which were needed by the exploration rules to find policies which collects 95% of what the PI-policy collects. Explanation: *, **, and ^{*5} = in 1, 2, and 5 simulation(s), respectively, the requirements were not met.*

Results. Tables 5.1 and 5.2 show the results. We can see that learning an exploration model with the frequency based reward rule works best. When state spaces become larger, the costs of undirected exploration methods such as Max-random exploration increases significantly. The costs of frequency based exploration seems to scale up almost linearly with the size of the state space, however.

Comments. (1). We used $\gamma = 0.99$ for learning the exploration model. This results in very global exploration behavior. We also tried out using lower values for γ. Those worked significantly worse. The entirely local policy with $\gamma = 0$ was only able to find 90%-optimal policies in 17 out of 20 simulations and took 60,000 steps for the 50 × 50 maze. Higher values worked better, although γ in the interval $[0.98, 0.99]$ gave the best results.

(2) Q-learning needed 69K steps to learn 90% optimal policies with Max-random exploration for the 25 × 25 maze. We also tried Q(λ) on 100 ×100 mazes. We found that $\lambda = 0.0$ performed best, although we have not tried replacing and resetting traces. Q-learning needed

14.4M steps before finding reasonable policies (compared to 360K for 50 $\times$50 mazes). Therefore as expected, we can say that Q-learning scales up much more poorly than model-based approaches.

(3) In our experimental results, Q(λ)-approaches did not benefit from learning exploration value functions. The results were more or less the same as using Max-random. It seems very difficult for Q-learning to learn useful exploration values. Since we penalize actions, the major gain in using exploration reward rules (to select alternative actions for a state) is lost.

Conclusion. Using exploration models can significantly speed up finding good policies. Using the frequency based reward rule we can compute 95%-policies by sampling all state/action pairs about 2.5 times. Although the variance in the estimated transition probabilities of the maximum likelihood model is still quite large, this does not seem to matter for finding good policies. Due to the randomly generated environment, there seem to be many policies which are quite good and it is not very difficult to learn one of them.

The other exploration techniques do not distribute the number of experiences fairly over the state/action space, and they have more difficulties finding good policies. The recency based reward rule might outperform frequency based exploration in changing environments. Finally, undirected exploration works well for the smallest problems, but consumes a lot of time for the more difficult problems. Thus, we need to make use of exploration models for solving large tasks.

5.5.2 Exploitation/Exploration

The previous subsection showed that we are able to quickly learn good policies by using exploration models. For limited lifetime problems involving expensive robots or experiments, we want to maximize the cumulative performance over time. This creates the exploitation/exploration problem. In this section we show how well the exploration methods maximize cumulative rewards.

We set up experiments with 50 $\times$ 50 mazes and use prioritized sweeping ($\epsilon = 1.0$ and $U_{max} = 100$) and let the agent learn for 100,000 steps during which we computed the cumulative and final rewards over intervals of 2,000 steps.

For the methods which learn exploration models, we increase P_{max} from 0.0 to 1.0 in order to exploit more and more. We also included a method which uses the frequency based rule to learn exploration models, but this system shifts to exploitation if the value function for the task is hardly changed. Each time the goal is found, the system checks whether a minimal of 2000 steps have been executed for which the average update of the V-function is lower than 1.0. We will call this system frequency based + stop explore. This system also uses $P_{max} = 0.0 \rightarrow 1.0$ (keeping the system fully explorative before switching to exploitation does not change things much). For Boltzmann exploration, we linearly anneal T from 10 to 0.1.

Results. Table 5.3 shows that although the value functions of the directed exploration methods are much better, the final performance for most exploration methods (except Boltzmann exploration) is more or less equal. The cumulative reward of using most exploration models is worse than using Max-random. Remember, however, that good policies are found earlier using directed exploration and that the difference in cumulative reinforcement is mainly because exploration policies lead the agent away from the greedy policies, thereby causing some loss in reward. The table shows that frequency based exploration and stop explore is able to accumulate a lot of reward. Figure 5.3(A) shows the learning curves for the different

System	End Rewards	Total Rewards	V-approx	Time
Max random (0.1 $\rightarrow$ 0.0)	15.0K $\pm$ 0.7K	560K $\pm$ 37K	310 $\pm$ 40	31 $\pm$ 3
Max random (0.3 $\rightarrow$ 0.0)	15.4K $\pm$ 0.6K	470K $\pm$ 37K	320 $\pm$ 39	32 $\pm$ 3
Max random (0.5 $\rightarrow$ 0.0)	15.3K $\pm$ 0.6K	390K $\pm$ 17K	280 $\pm$ 29	35 $\pm$ 2
Boltzmann	14K $\pm$ 4.0K	250K $\pm$ 120K	340 $\pm$ 72	41 $\pm$ 11
Error based	15.1K $\pm$ 0.7K	500K $\pm$ 120K	67 $\pm$ 61	260 $\pm$ 32
Recency based	15.2K $\pm$ 0.5K	250K $\pm$ 21K	30 $\pm$ 16	230 $\pm$ 16
Frequency based	15.2K $\pm$ 0.7K	130K $\pm$ 18K	21 $\pm$ 8	128 $\pm$ 3
Frequency based + Stop explore	15.4K $\pm$ 0.6K	560K $\pm$ 25K	65 $\pm$ 76	89 $\pm$ 9

Table 5.3: *Results for different exploration rules with prioritized sweeping.*

exploration methods. Positive Q-updates refers to the error based exploration rule. Figure 5.3(B) shows that the exploration models are very useful for learning good value function approximations. The recency and frequency based reward rules work best. They are able to learn quite good value function approximations. The fact that such good approximations are attained by sampling all state/actions only 2.5 times ($2.5 \times 4 \times 50 \times 50 = 25,000$ experiences) shows that the value function is not very sensitive to noise. The reason is that noisy actions take the agent to a field closeby so that even in case of strong bias due to noise in the 2.5 experiences for some state, the value function will not differ too much.

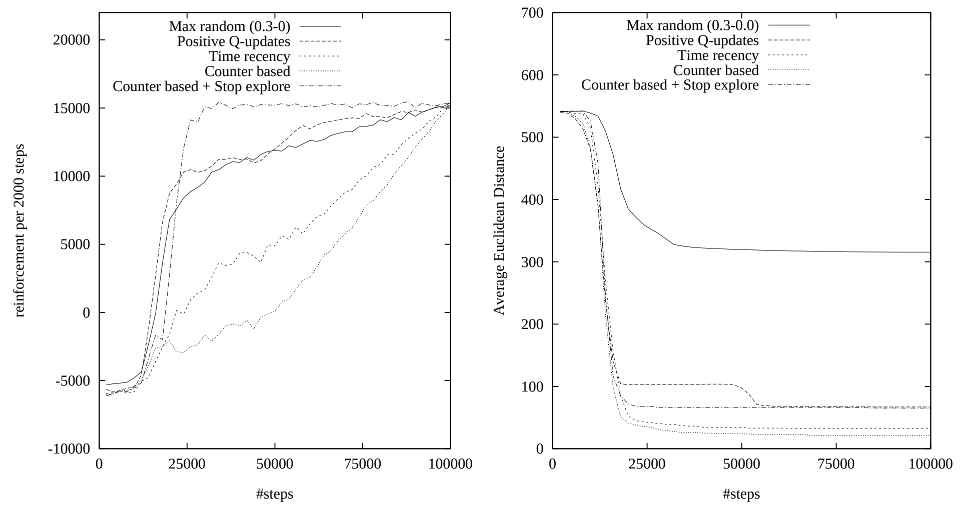


Figure 5.3: *(A) A comparison between different (learning) exploration rules with prioritized sweeping. The figure plots the cumulative rewards achieved over intervals of 2,000 steps. (B) The approximation error of the value function, using different exploration methods.*

The frequency based + stop explore system scores very high on all objective criteria: the final performance is very good, the cumulative performance is the best and the obtained value function is much better than that of indirect exploration methods.

Comments. (1) Initialize high exploration performed very poorly with PS. The reason for this is that exploration values can keep their large initial values for a long time. (2)

Error and recency based exploration cost more computational time than frequency based exploration. The reason is that reward values are bigger in size.

Conclusion. The exploration/exploitation dilemma is not handled very well by the fully explorative model-based exploration techniques. However, we can augment them simply by a heuristic switching rule which makes the agent switch from exploration to full exploitation. Our rule which switched once the value function did not change very much anymore made it possible to first quickly learn a quite good value function approximation which could afterwards be exploited for a long time. Although for the current problem simple undirected exploration methods also worked quite well, they were unable to learn a good value function.

We expect that for more general MDPs, Max-random exploration will perform worse, since for the current problems they did not have any problems finding the global terminal goal state. If there are multiple rewarding states, they may have larger problems and the advantage of using the directed exploration methods could be larger.

5.5.3 Experiments with Suboptimal Goals

The final experimental comparison shows results with directed exploration methods and MBIE on a maze with suboptimal goal-states. The maze is shown in Figure 5.4. The starting state (S) is located 1 field north/east of the south-west corner. There are three absorbing goal states, two of them are suboptimal. The optimal goal state (G) is located 1 field south/west of the north-east corner, the suboptimal goal states (F) are located in the north-west and south-east corners. Selected actions are replaced by random actions with 10% probability.

Reward function. Rewards and the discount factor are the same as before. For finding a suboptimal goal state the agent gets a reward of 500.

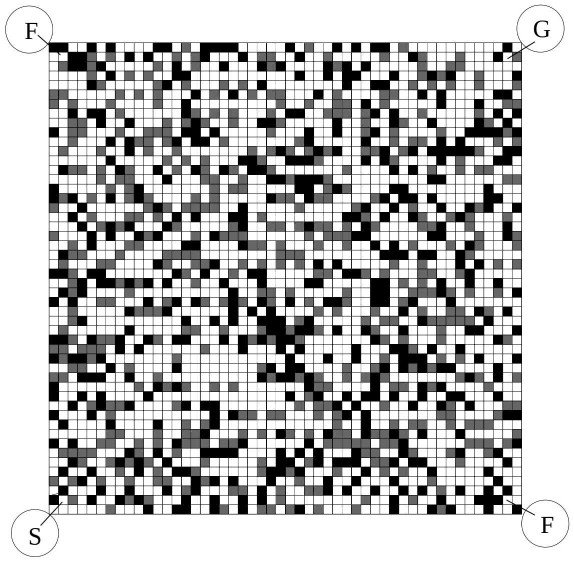


Figure 5.4: *The* 50×50 *maze used in the experiments. Black squares denote blocked fields, grey squares penalty fields. The agent starts at S and searches for a minimally punishing path to the optimal goal G. Good exploration is required to avoid focusing on suboptimal goal states (F).*

Comparison. We compare the following exploration methods: Max-random, directed model-based exploration techniques using frequency based and recency based reward rules, and MBIE. The latter starts out with model-based exploration using the frequency based reward rule, and switches to IE once the value function hardly changes any more.

The goal is to learn good policies as quickly as possible. We computed an optimal policy using value iteration (Bellman, 1961) and tested this optimal policy by executing it for 1,000 steps. We computed its average reinforcement intake by testing it 10,000 times, which resulted in 7590 $\pm$ 2 rewardpoints. For each method we conduct 20 runs of 100,000 learning steps. During each run we measure how quickly and how often the agent's policy collects 95%, 99% and 99.8% of what the optimal policy collects. This is done by averaging the results of 1000 test runs conducted every 2000 learning steps — each test run consists of executing the greedy policies (always selecting actions with maximal Q-value) for 1000 steps.

Parameters. We set the accuracy parameter $\epsilon = 0.1$, and $U_{max} = 100$ for both learning the problem-oriented and exploration value functions. The exploration reward's discount factor γ is set to 0.99 for frequency based and to 0.95 for recency based exploration. The constant K_C (used by the frequency based reward rules) is set to 50; K_T (used by the recency based reward rule) is set to 1000. We used two values for P_{max} (for Max-random exploration): 0.2 and 0.4. The value of z_α (for MBIE) is set to 1.96 (which corresponds to a confidence interval of 95%). The combination of MBIE and model-based exploration switches to MBIE once the value function has not changed by more than 0.1 (η) on average within the 1000 (TS) most recent steps.

Exploration Rule	95% (freq)	99% (freq)	99.8% (freq)
MBIE	20 (25K)	19 (42K)	18 (66K)
Frequency based	20 (24K)	16 (50K)	10 (66K)
Recency based	19 (27K)	18 (55K)	9 (69K)
Max-random 0.2	4 (43K)	4 (52K)	4 (68K)
Max-random 0.4	0 (—)	0 (—)	0 (—)

Table 5.4: *The number of runs of several exploration methods which found p-optimal policies (and how many steps were required for obtaining it).*

Exploration Rule	Best run result	Training Performance
MBIE	7.57K $\pm$ 0.05K	350K $\pm$ 40K
Frequency based	7.55K $\pm$ 0.06K	-45K $\pm$ 9K
Recency based	7.54K $\pm$ 0.11K	-120K $\pm$ 10K
Max-random 0.2	4.8K $\pm$ 1.4K	-190K $\pm$ 16K
Max-random 0.4	4.1K $\pm$ 0.3K	-62K $\pm$ 19K

Table 5.5: *Average and standard deviation of the best test result during a run and the total cumulative reward during training.*

Results. Table 5.4 shows significant improvements achieved by learning an exploration model. The undirected exploration methods focus too much on suboptimal goals (which are closer and therefore easier to find). This often prevents them from discovering near-optimal policies. On the other hand, model-based learning with directed exploration does favor paths

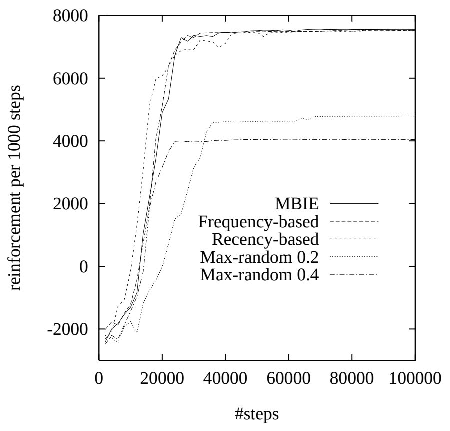


Figure 5.5: *Cumulative reward during test runs, averaged over 20 simulations.*

leading to the optimal goal. Using the frequency based reward rule by itself, the agent always finds the optimal goal although it fails to always find 99.8% optimal policies.

Figure 5.5 shows a large difference between learning performances of exploration models and Max-random exploration. It does not clearly show the distinction between frequency based or recency based exploration and the extension with MBIE. Switching to MBIE after some time (which happens between 35,000 and 55,000 steps), however, significantly improves matters. First of all, Table 5.4 shows that this strategy finds optimal or near-optimal policies in 90% of the cases, whereas the others fail in at least 50% of the cases. The second improvement with MBIE is shown in Table 5.5. MBIE collects much more reward during training than all other exploration methods, thereby effectively addressing the exploration/exploitation dilemma. In fact it is the only exploration rule leading to a positive cumulative reward score.

Conclusion. The results with multiple goals showed a large advantage of using directed exploration methods. Undirected methods are attracted to those rewarding states which come first on the path and therefore we cannot trust them for finding good policies. The best method, MBIE, combined directed exploration with confidence levels in order to prune away actions which almost certainly do not belong to the optimal policy. This results in the best performance, since it takes care that only actions which can belong to the optimal policy are selected. Finally, this also results in the highest cumulative performance. We expect that this method will perform best for most stochastic discrete MDPs, although a disadvantage of the method is that we need to set the right switching time.

5.6 Discussion

If we use undirected exploration methods, we rely on our random generator for selecting actions. However, if we learn where to explore, we can direct our exploration behavior so that more information is gained. We formulated a number of exploration reward rules which determine which information is interesting and which can be used for learning exploration value functions. Our experimental results with stochastic mazes showed that learning exploration policies can speed up finding good policies. Especially, when there are particular complex paths leading to much more rewards then other easier paths, good exploration is essential.

The frequency based reward rule performed best for the environments we have considered. We expect this to hold as well for other MDPs. For non-stationary problems, an exploration policy based upon the recency based reward rule may work best, since it is able to discover much earlier which fragments of the state space have changed.

Optimal exploration. When it comes to finding optimal policies, methods using external MDP rewards for focusing on actions with large probability of being optimal will often save some SAP visits over methods based solely on exploration reward rules. Furthermore, cumulative external MDP rewards obtained during training should be taken into account in attempts at approaching the exploit/explore dilemma. That is why we introduced MBIE, a method combining Kaelbling's interval estimation (IE) algorithm (Kaelbling, 1993) and model-based reinforcement learning. Since MBIE heavily relies on initial statistics, we switch it on only after an initial phase during which an exploration model is learned (according to, say, the frequency based exploration reward rule). In our experiments this approach almost always led to 99.8%-optimal policies. We expect MBIE to be advantageous for all kinds of MDPs, especially when the reward function is far from uniform. Other exploration methods do not work well for such problems since their exploration policies do not make distinctions between different actions leading to different payoffs.

Different topologies. For particular environments it may be a good idea to explore entire subregions before exploring other subregions. E.g., when there is a bottleneck connecting two subregions, both subregions may alternatingly become interesting to explore, but the bottleneck will become uninteresting after some time. Our exploration rules could detect such bottlenecks and would spend a lot of time exploring one subspace before going to the other, since going through the bottleneck brings a lot of negative exploration reward.

Function approximators. The ideas are presented for tabular representations, but can easily be extended to function approximators. For this we can use the same representation for the exploration value function as for the original value function. Exploration methods for large or continuous spaces are more complex, however, since it is infeasible to explore all states. Therefore we rely on the generalization abilities of the function approximator to determine the interestingness of experiences. Furthermore, the reward function gets a little bit more complicated since states may never be visited twice. Therefore the reward function should be based upon the distribution of previous experiences around a point for which we compute an exploration value. E.g. a Gaussian function can be used to weigh neighboring experiences according to their distance.

Chapter 6

Partially Observable MDPs

We have seen how we can efficiently compute policies for Markov decision processes (MDPs) consisting of a finite number of states and actions. MDPs require that all states are fully observable, which means that the agent is able to use perfect sensors to always know the exact state of the world. For real world problems this requirement is usually violated. Agents receive their information about the state of the world through imperfect sensors, which sometimes cause world states to be mapped to the same observations. This problem is called *perceptual aliasing* (Whitehead, 1992) and is the reason for the uncertainty of the agent about the state of the world.

As example, have a look at the environment depicted in Figure 6.1. There are a total of 6 states, but the agent can only see 2 different observations: a circle or a rectangle. Thus, solely based on the current observation the agent can never be certain in which state it is.

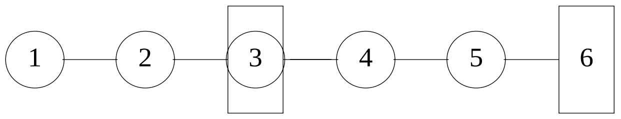


Figure 6.1: *An environment consisting of 6 states. Whenever the agent is in the first two states or in states 4 or 5, it always sees a circle as observation. In the third state it observes a circle or rectangle, both with probability 50%, and in the sixth state it always observes a rectangle.*

Markov decision problems for which the agent is uncertain about the true state of the world are called *partially observable Markov decision processes* (POMDPs, e.g., Littman, 1996). The simplest algorithms for solving them would directly map observations to actions, but since observations are received in different states which require different actions, such algorithms will not be able to find solutions. Better algorithms need to use previous observations and actions which were encountered on the current trajectory through the state space to disambiguate possible current states. This is not an easy task — even deterministic finite horizon POMDPs are NP-complete (Littman, 1996). Thus, general and exact algorithms are feasible only for small problems, which explains the interest in heuristic methods for finding good but not necessarily optimal solutions.

Hidden state. Since an observation conceals some part of the true underlying state, we say that some part of the state is *hidden*. The solution to the *hidden state problem*

(HSP) is to update an agent's *internal state* (IS) based on its previous observations and actions. This internal state summarizes the trace of previous events and is used to augment the information of the current observation in the hope that the state ambiguities are resolved. Let's have another look at Figure 6.1. Suppose that the agent sees a rectangle, steps left, sees a circle, steps right, and sees a circle. Suppose also that there is no noise in the execution of actions, then in which state is the agent? The answer is in state 3, since only state 3 can emit a rectangle and a circle symbol. Thus, the augmented state is uniquely defined and can be used by the policy for selecting the optimal action. Figure 6.2 shows how previous observations/actions of the environment are stored and used to create the IS. Then the IS is combined with the current observation to generate a description of the state which is used by the policy to select an action.

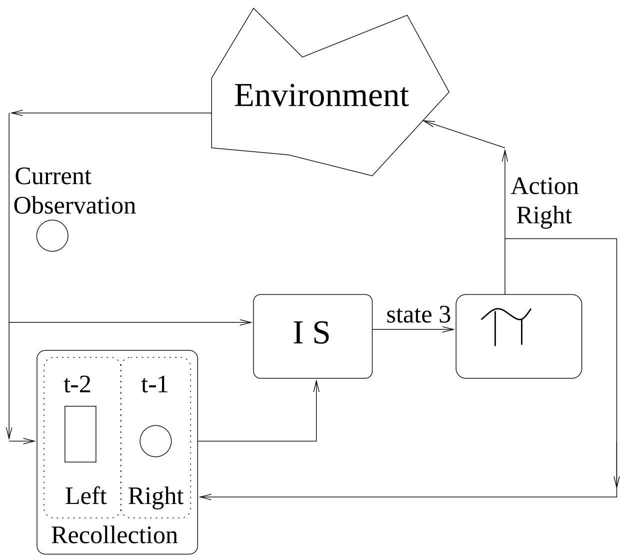


Figure 6.2: *The decision making loop of an agent in a partially observable environment. The agent receives an observation from the environment defined in Figure 6.1. The recollected observations and actions are used to create an internal state (IS). This internal state is then augmented with the current observation and the resulting state is presented to the policy. The policy then selects an action, which together with the current observation is memorized. Finally the action is executed and changes the environment.*

Outline of this chapter. In Section 6.1, we describe the POMDP framework and also optimal algorithms from operations research based on dynamic programming algorithms which allow computing optimal policies by explicitly representing the uncertainty of the state in a probabilistic description. In Section 6.2, we will describe our novel algorithm HQ-learning, and show its capability in solving a number of partially observable mazes. In Section 6.3, we summarize related work. In Section 6.4, we conclude with some final statements about the problems and algorithms.

6.1 Optimal Algorithms

Operations research has investigated optimal algorithms for solving POMDPs for some time already (see Lovejoy (1991) for an overview). They treat POMDPs as planning problems, which means that the state transition function, reward function, and observation function are given, and that the goal is to compute an optimal plan. Note that this plan is not represented by a single sequence of actions, since such a plan would probably not be carried out due to the stochasticity of the problem (in e.g. the transition function). Instead, the plan is again represented by a policy mapping states to actions. This is similar to the MDP framework of Chapter 2. The big difference with MDPs is that even when we know the model of the POMDP, computing an optimal policy is extremely hard. An agent may confuse different states due to the noise in the observation function and may be totally mistaken about the true state of the world. So how can it then compute an optimal policy? Well, it can by taking its uncertainty into account in its decision making. We will see that this allows for selecting information gathering actions and goal-directed actions based on maximizing its expected future discounted cumulative reward.

6.1.1 POMDP Specification

A POMDP is a MDP where the agent does not receive the state of the world as input, but an observation of this state. Thus, the agent should use multiple observations in its decision making policy. As for MDPs, the system's goal is to obtain maximal (discounted) cumulative reward. A POMDP is formally specified by:

- $t = 1, 2, 3, \ldots$ is a discrete time counter.
- S is a finite set of states.
- A is a finite set of actions.
- $P(s_{t+1}|s_t, a_t)$ denotes the probability of transition to state s_{t+1} given state s_t and action a_t.
- R maps state/action pairs to scalar reinforcement signals.
- O is a finite set of observations.
- P_0 is the probability distribution over initial states.
- $P(o|s)$ denotes the probability of observing a particular (ambiguous) input o in state s. We write $o_t = B(s_t)$ to denote the observation emitted at time t, while visiting state s_t.
- $0 \leq \gamma \leq 1$ is a discount factor which trades off immediate rewards against future rewards.

6.1.2 Belief States

To compute an optimal policy we need to use an optimal description of the world state. Due to the uncertainty in action outcomes (noise in the execution of actions) and the partial observability of world states, we are usually not certain that the agent is in a particular single environmental state. Taking the most probable state is possible, but this cannot lead to optimal policies for particular POMDPs, since it does not take into account that the agent can be in many different states. For computing optimal policies, we compute occupancy probabilities over all states. Such a probability distribution over possible world states is called a *belief state*. We will denote the belief state at time t as b_t, where $b_t(s)$ is the belief that we are in state s at time t. The belief state gives us all necessary information to decide upon the optimal action (Sondik, 1971) — additional information about previous states and actions is superfluous.

Computing belief states. At trial start up, the *a priori* probability of being in each state s is $P_0(s)$. Since the agent is "thrown" in the world at time $t = 1$, it also receives an observation (o_1). Therefore, we can compute the belief state at $t = 1$ using Bayes' rule:

$$b_1(s) = P(s|o_1, P_0) = \frac{P_0(s)P(o_1|s)}{\sum_i P_0(i)P(o_1|i)}$$

At time t-1, we know b_{t-1}, the selected action a_{t-1} and after making a new observation o_t, we can compute b_t as follows:

$$b_t(s) = P(s|o_t, b_{t-1}, a_{t-1}) = \frac{P(o_t|s)\sum_i b_{t-1}(i)P(s|i, a_{t-1})}{\sum_j P(o_t|j)\sum_i b_{t-1}(i)P(j|i, a_{t-1})} \tag{6.1}$$

In the nominator we sum the probabilities of entering state s over the whole ensemble of states. We can do this, since the state space is finite. The denominator equals $P(o_t|b_{t-1}, a_{t-1})$ and can be seen as a normalizing factor to keep the constraint $\sum_j b_t(j) = 1$ true.

Example of belief state dynamics. Consider again the environment of Figure 6.1. Suppose that the actions *go-left* and *go-right* are executed with 100% probability. If the initial probability distribution over states is $\{0.2, 0.2, 0.2, 0.2, 0.2, 0\}$ and the agent observes a circle, then $b_1 = \{0.22, 0.22, 0.11, 0.22, 0.22, 0\}$. Now suppose the agent executes action *go-right* and observes a circle. Then the belief state $b_2 = \{0, 0.33, 0.17, 0.17, 0.33, 0\}$. Again the agent goes right and observes a circle, then the belief state $b_3 = \{0, 0, 0.33, 0.33, 0.33, 0\}$, and again — $b_4 = \{0, 0, 0, 0.5, 0.5, 0\}$. Finally after 4 actions and 5 circles, the agent knows with certainty that it is in state 5 (thus it could also infer that it started in state 1). In this case, the actions were deterministic and the uncertainty decreased after each action. In general, the uncertainty can increase or decrease dependent on the belief state, and the transition and observation functions.[1] In such environments the agent can reduce state uncertainty by visiting particular states with exceptional observations. Such states are generally referred to as *landmark states*, which play an important role in designing algorithms for POMDPs.

6.1.3 Computing an Optimal Policy

The policy maps a belief state to an action:

$$a_t = \Pi(b_t)$$

Note that the belief state b_t has the dimension of the state space (to be precise its dimension is $|S| - 1$) and consists of continuous variables. Computing optimal policies for such spaces is extremely difficult, especially if there are many states. There are some special properties of the space setup by the belief states, however, which makes it possible to compute optimal

[1]Even deterministic actions do not imply that the uncertainty or entropy always decreases.

policies. The important point is that there are finitely many segments in the belief state space for which the same optimal action is required, when we consider a finite horizon of the POMDPs. This makes it possible for optimal algorithms to store for all crucial belief states (the corners of line segments in belief space), the optimal action and the Q-values.

There are two kinds of exact algorithms, one works with vectors in the belief state space, the other works with pre-generated policy trees which are then evaluated on different segments of belief space.

DP on vectors in belief state space. We can derive a DP algorithm which at each iteration computes all possible belief states given the previous set of belief states by using Equation 6.1 and iterating over all possible actions and observations. Then the algorithm computes their values — for some belief state b, we can compute its 1-step value as follows:

$$V_1^*(b) = \max_a \sum_s b(s)R(s,a) \tag{6.2}$$

Note that this value is a linear product of the belief state and the reward function. Then, the t-step value can be computed using value iteration as follows:

$$V_t^*(b) = \max_a \sum_s b(s)R(s,a) + \gamma \sum_s \sum_o \sum_i b(s)P(i|s,a)P(o|i)V_{t-1}^*(b(o,a)) \tag{6.3}$$

where $b(o,a)$ is defined as the belief state resulting from observing observation o after selecting action a in belief state b. Thus, we basically use transition probabilities between belief states and compute the value of a belief state in a very similar way as for MDPs.

One problem is that although the number of belief states is finite, it grows exponentially with the horizon of the planning process. Therefore, some pruning in belief state space is needed to make the computations feasible. We will not discuss those topics further. Interested readers are referred to Cassandra's thesis (1998).

Another problem is that we have assumed that we start with an initial belief state which is known *a priori* Now assume that we do not yet know the initial belief state when we want to compute the value function, although this belief state is given later on once we get a problem instance. It is still possible to store the value function since it is piecewise linear which means that there are finitely many facets (see (Cassandra, 1998) for a formal proof). We know that one policy is always best for a specific region of the belief state. Each policies' value function is linear in the belief state (see Equation 6.3). Thus, since we always take the maximum of the linear functions, the optimal value function is a piecewise linear convex function (Figure 6.3). We cannot compute the facets using the simple algorithm described above, however, since we would need to compute belief state intervals which are recursively refined. Then for each belief state interval we have to compute the optimal action. Therefore, we need to use more difficult algorithms for solving this more general problem.

Policy trees. For computing the optimal policy given an unknown *a priori* state, we can make use of *policy trees* (Littman, 1996; Cassandra et al., 1994; Kaelbling et al., 1995). Policy trees fully describe how the agent should act in the next t steps, and use the sequence of observations in determining correct action selection (see Figure 6.4).

Given each particular belief state, one t-step policy tree is optimal. We want to know, however, for which interval in belief state space some particular policy tree is optimal. To be able to compute this, we can generate all possible t-step policy trees and evaluate them on each state. Then we are able to compute the value function of a policy-tree as a linear function of the belief state.

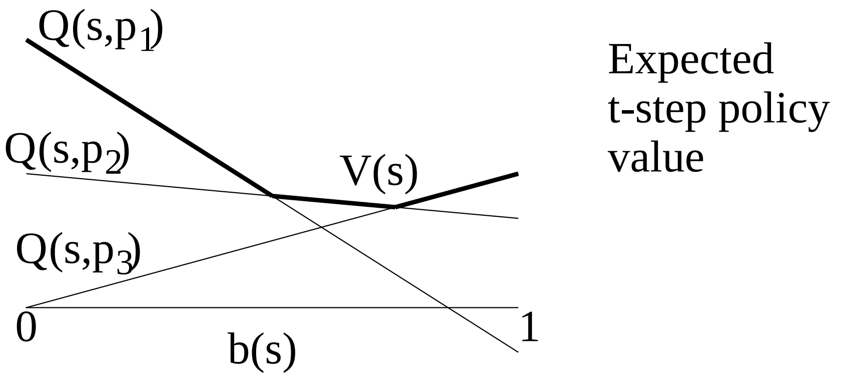


Figure 6.3: *A two state problem defined by the probability we are in state s. We show the optimal value function for this belief state space, together with 3 Q-functions of different policies. Dependent on the belief state, one policy is the best one. Since the Q-functions of policies are a linear function of the belief state, the optimal value function is a piece-wise linear convex function.*

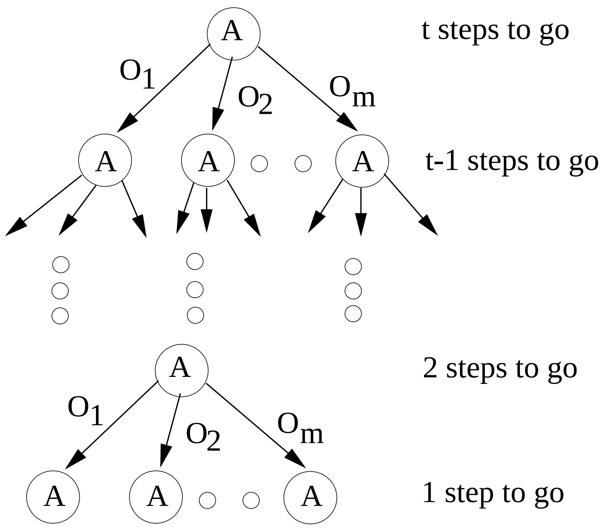


Figure 6.4: *A t-step policy tree. After each action an observation is made and this observation determines which branch in the tree is followed after which the rest of the policy tree is executed.*

Thus, again we start simple. If the agent can only select a single action, we say that the agent executes a 1-step policy tree. We denote the policy tree as p, and p's root action by $a(p)$. We compute the Q-value of executing the 1-step policy tree p in state s as:

$$Q(s,p) = R(s,a(p))$$

Where as usual $R(s,a(p))$ is the immediate reinforcement for selecting action $a(p)$ in state s.

Then, we construct a t-step policy tree p from a set of $t-1$-step policy trees $\{p_{t-1}^1, \ldots, p_{t-1}^{|O|}\}$

by defining which observations has lead to them (we denote $p_{t-1}(o)$ as the policy-tree p_{t-1} which follows observation o) and by merging these trees by a rootnode with action $a(p)$.

We can compute the value function of a t-step policy tree by combining the value functions of its t-1-step policy subtrees. The value of executing the t-step policy tree p_t in state s can be computed as:

$$Q(s, p_t) = R(s, a(p_t)) + \gamma \sum_{s'} P(s'|s, a) \sum_{o} P(o|s') Q(s', p_{t-1}(o))$$

Thus, we just add the immediate reward to the expected value of executing a particular selected policy subtree (which one depends on the observation) in the next state. We store all values $Q(s, p_t)$ of executing the policy tree p_t in state s.

Then, when given a belief state, we can simply weigh the different values according to their occupancy probabilities to obtain the evaluation of executing a policy tree on a belief state b:

$$Q(b, p) = \sum_{s} b(s) Q(s, p)$$

Now the t-step value of a particular belief state b is the maximal value over all policy trees:

$$V_t(b) = \max_{p_t} Q(b, p_t)$$

Complexity. The algorithm is intractable for all but small problems. Especially for long or infinite planning horizons the costs are infeasible. There are $|O^t|$ different t-step observation sequences, and $1 + |O| + |O^2| + \ldots |O^t| = (|O|^t + 1 - 1)/(|O| - 1)$ nodes in each t-step policy tree. Since at each node we can choose between $|A|$ different actions, the total number of policy trees is $|A|^{(|O|^t+1-1)/(|O|-1)}$ which grows superexponentially! Finally, we note that the number of facets of the optimal value function may be infinite for infinite horizon problems.

Faster optimal algorithms. In principle it is possible that each one of these policy trees can be optimal for some point in belief space. Fortunately, in practice many policy trees are completely dominated by other policy trees (a policy tree is dominated if for each possible belief state there exists another policy tree with larger or equal Q-value), so that they can be pruned away. This allows for significant reductions in computational time and storage space. Further reductions are made possible by using particular online generation algorithms, e.g. the Witness algorithm (Cassandra et al., 1994; Littman, 1996) or algorithms proposed by (Cassandra, 1998; Zhang and Liu, 1996) which only generate policy trees which have the possibility of being optimal for some points in belief space. These methods can significantly reduce computational requirements for practical problems. Still, computing optimal policies for general problems remains extremely expensive.

Littman et al. (1995) and Cassandra (1998) compare different POMDP algorithms using belief states. They report that "small POMDPs" (with less than 10 states and few actions) do not pose a very big problem for most methods. Larger POMDPs (50 to 100 states), however, can cause major problems. Thus, although the theory allows us to compute optimal solutions, the gap between theory and practice does not allow us to use these algorithms for real world problems. Therefore, we will focus on heuristic methods which can be used to find suboptimal solutions to much larger problems.

6.2 HQ-learning

We have seen that general and exact algorithms are feasible only for small problems. Furthermore, they require a model of the POMDP. That explains the interest in reinforcement learning methods for finding good but not necessarily optimal solutions. Examples are the use of recurrent networks (Schmidhuber, 1991d; Lin, 1993), hidden Markov models (McCallum, 1993), higher order neural networks (Ring, 1994), memory bits (Cliff and Ross, 1994), and stochastic policies (Jaakkola, Singh and Jordan, 1995). Unfortunately, however, most heuristic methods do not scale up very well (Littman, Cassandra and Kaelbling, 1995) and are only useful for particular POMDPs.

We will now describe HQ-learning (Wiering and Schmidhuber, 1997), a novel approach based on finite state memory implemented in a sequence of agents. HQ does not need a model and can solve large deterministic POMDPs.

6.2.1 Memory in HQ

To select the optimal next action it is often not necessary to memorize the entire past (in general, this would be infeasible since the necessary memory grows exponentially with the order of the Markov chain, see Appendix A). A few memories corresponding to important previously achieved subgoals can be sufficient. For instance, suppose your instructions for the way to the station are: "Follow this road to the traffic light, turn left, follow that road to the next traffic light, turn right, there you are.". While you are on your way, only a few memories are relevant, such as "I already passed the first traffic light". Between two such subgoals a memory-independent, *reactive policy* (RP) will carry you safely.

Overview. HQ-learning uses a divide-and-conquer strategy to decompose a given POMDP into a sequence of *reactive policy problems* (RPPs). RPPs can be solved by RPs: all states causing identical inputs require the same optimal action. The only "critical" points are those corresponding to transitions from one RP to the next.

To deal with such transitions HQ uses multiple RPP-solving subagents. Each agent's RP is an adaptive mapping from observations to actions. At a given time only one agent can be active, and the system's only type of short-term memory is embodied by a pointer indicating which one. Thus, the internal state (IS) of the system is just the pointer to the active agent.

RPs of different agents are combined in a way learned by the agents themselves. The first active agent uses a subgoal table (its *HQ-table*) to generate a subgoal for itself (subgoals are represented by desired inputs — if these inputs are unique in a specific region, we call them landmark states). Then it follows the policy embodied by its Q-function until it achieves its subgoal. Then control is passed to the next agent, and the procedure repeats itself. After the overall goal is achieved or a time limit is exceeded, each agent adjusts both its RP and its subgoal. This is done by two learning rules that interact without explicit communication: (1) Q-table adaptation is based on slight modifications of Q(λ)-learning. (2) HQ-table adaptation is based on tracing successful subgoal sequences by Q(λ)-learning on the higher (subgoal) level. Effectively, subgoal/RP combinations leading to higher rewards become more likely to be chosen.

POMDPs as RPP sequences. The optimal policy of any deterministic finite POMDP with fixed starting state and final goal state is decomposable into a finite sequence of optimal reactive memoryless policies for appropriate RPPs, along with subgoals determining transitions from one RPP to the next. The trivial decomposition consists of single-state RPPs and the corresponding subgoals. In general, POMDPs whose only decomposition is trivial are hard — there is no efficient algorithm for solving them. This is simple to see, if we consider an environment where there is only 1 observation (the agent is thus blind). To solve such problems, all T-step policies have to be tried out from the initial state, where T denotes the length of the optimal solution path, which results in $|A|^T$ possibilities. HQ, however, is aimed at situations that require few transitions between RPPs which means that each RPP can be used for many states.

Architecture. System life is separable into "trials". A trial consists of at most T_{max} discrete time steps $t = 1, 2, 3, \ldots, T$, where $T < T_{max}$ if the agent solves the problem in fewer than T_{max} time steps.

There is an ordered sequence of M agents $\mathcal{C}_1$, $\mathcal{C}_2$, ... $\mathcal{C}_M$, each equipped with a Q-table, an HQ-table, and a control transfer unit, except for $\mathcal{C}_M$, which only has a Q-table (see Figure 6.5). Each agent is responsible for learning part of the system's policy. Its Q-table represents its local policy for executing an action given an input. It is given by a matrix of size $|O| \times |A|$, where $|O|$ is the number of different possible observations and $|A|$ the number of possible actions. $Q_i(o_t, a_j)$ denotes $\mathcal{C}_i$'s Q-value (utility) of action a_j given observation o_t. The agent's current subgoal is generated with the help of its HQ-table, a vector with $|O|$ elements. For each possible observation there is an HQ-table entry representing its estimated value as a subgoal. $HQ_i(o_j)$ denotes $\mathcal{C}_i$'s HQ-value (utility) of selecting o_j as its subgoal.

The system's current policy is the policy of the currently *active* agent. If $\mathcal{C}_i$ is active at time step t, then we will denote this by $Active(t) = i$. The variable $Active(t)$ represents the only kind of short-term memory in the system.

Selecting a subgoal. In the beginning $\mathcal{C}_1$ is made active. Once $\mathcal{C}_i$ is active, its HQ-table is used to select a subgoal for $\mathcal{C}_i$. To explore different subgoal sequences we use the Max-random rule: the subgoal with maximal HQ_i value is selected with probability P_{max}, a random subgoal is selected with probability $1 - P_{max}$. Conflicts between multiple subgoals with maximal HQ_i-values are solved by randomly selecting one. $\hat{o}_i$ denotes the subgoal selected by agent $\mathcal{C}_i$. This subgoal is only used in transfer of control as defined below and should not be confused with an observation.

Selecting an action. $\mathcal{C}_i$'s action choice depends only on the current observation o_t. During learning, at time t, the active agent $\mathcal{C}_i$ will select actions according to the Max-Boltzmann distribution (see Chapter 5). The "temperature" T_i adjusts the degree of randomness involved in agent $\mathcal{C}_i$'s action selection in case the Boltzmann rule is used.

Transfer of control. Control is transferred from one active agent to the next as follows. Each time $\mathcal{C}_i$ has executed an action, its control transfer unit checks whether $\mathcal{C}_i$ has reached the goal. If not, it checks whether $\mathcal{C}_i$ has solved its subgoal to decide whether control should be passed on to $\mathcal{C}_{i+1}$. We let t_i denote the time at which agent $\mathcal{C}_i$ is made active (at system start-up, we set $t_1 \leftarrow 1$).

```
IF no goal state reached AND current subgoal = $\hat{o}_i$
AND $Active(t) < M$ AND $B(S_t) = \hat{o}_i$
THEN $Active(t+1) \leftarrow Active(t) + 1$ AND $t_{i+1} \leftarrow t + 1$
```

6.2.2 Learning Rules

We will use off-line learning for updating the tables — this means storing experiences and postponing learning until after trial end (no intra-trial parameter adaptation). In principle,

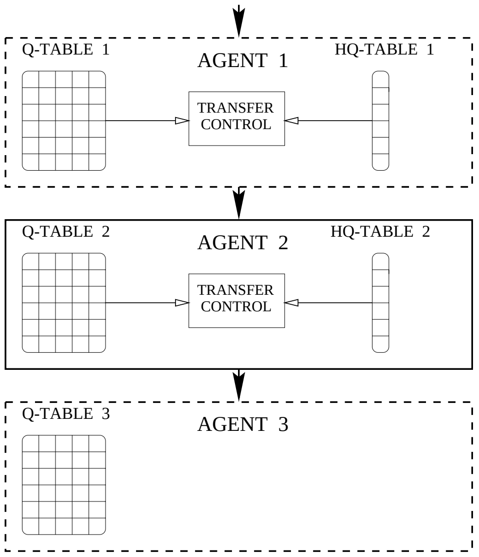


Figure 6.5: *Basic HQ-architecture. Three agents are connected in a sequential way. Each agent has a Q-table, an HQ-table, and a control transfer unit, except for the last agent which only has a Q-table. The Q-table stores estimates of actual observation/action values and is used to select the next action. The HQ-table stores estimated subgoal values and is used to generate a subgoal once the agent is made active. The solid box indicates that the second agent is the currently active agent. Once the agent has achieved its subgoal, the control transfer unit passes control to its successor.*

however, online learning is applicable as well (see below). We will describe two HQ variants, one based on Q-learning, the other on Q(λ)-learning — Q(λ) overcomes Q's inability to solve certain RPPs. The learning rules appear very similar to those of conventional Q and Q(λ). One major difference though is that each agent's prospects of achieving its subgoal tend to vary as various agents try various subgoals.

Learning the Q-values. We want $Q_i(o_t, a_t)$ to approximate the system's expected discounted future reward for executing action a_t, given o_t. In the one-step lookahead case we have

$$Q_i(o_t, a_t) = \sum_{s_j \in S} P(s_j|o_t, \Theta, i)(R(s_j, a_t) + \gamma \sum_{s_k \in S} P(s_k|s_j, a_t) V_{Active(t+1)}(B(s_k))),$$

where $P(s_j|o_t, \Theta, i)$ denotes the probability that the system is in state s_j at time t given observation o_t, all architecture parameters denoted Θ, and the information that $i = Active(t)$. HQ-learning does not depend on estimating this probability, although belief states or a world model might help to speed up learning. $V_i(o_t)$ is the utility of observation o_t according to agent $\mathcal{C}_i$, which is equal to the Q-value for taking the best action:
$V_i(o_t) = \max_{a_j \in A}\{Q_i(o_t, a_j)\}$.

Q-value updates are generated in two different situations ($T \leq T_{max}$ denotes the total number of executed actions during the current trial, and α_Q is the learning rate):

Q.1 Let $\mathcal{C}_i$ and $\mathcal{C}_j$ denote the agents active at times t and $t+1$ — possibly $i = j$. If $t < T$

then
$Q_i(o_t, a_t) \leftarrow (1 - \alpha_Q)Q_i(o_t, a_t) + \alpha_Q(R(s_t, a_t) + \gamma V_j(o_{t+1}))$

Q.2 If agent $\mathcal{C}_i$ is active at time T, and the final action a_T has been executed, then
$Q_i(o_T, a_T) \leftarrow (1 - \alpha_Q)Q_i(o_T, a_T) + \alpha_Q R(s_T, a_T)$

Note that $R(s_T, a_T)$ is the final reward for reaching a goal state if $T < T_{max}$. A main difference with standard one-step Q-learning is that agents can be trained on Q-values which are not their own (see [Q.1]).

Learning the HQ-values: intuition. Recall the introduction's traffic light task. The first traffic light is a good subgoal. We want our system to discover this by exploring (initially random) subgoals and learning their HQ-values. The traffic light's HQ-value, for instance, should converge to the expected (discounted) future cumulative reinforcement to be obtained after it has been chosen as a subgoal. How? Once the traffic light has been reached and the first agent passes control to the next, the latter's own expectation of future reward is used to update the first's HQ-values. Where do the latter's expectations originate? They reflect its own experience with final reward (to be obtained at the station).

More formally. In the optimal case we have

$$HQ_i(o_j) = E(R_i + \gamma^{t_{i+1}-t_i} HV_{i+1}),$$

where E denotes the average over all possible trajectories. $R_i = \sum_{t=t_i}^{t_{i+1}-1} \gamma^{t-t_i} R(s_t, a_t)$, $\mathcal{C}_i$'s discounted cumulative reinforcement during the time it will be active (note that this time interval and the states encountered by $\mathcal{C}_i$ depend on $\mathcal{C}_i$'s subtask). $HV_i = \max_{o_l \in O}\{HQ_i(o_l)\}$ is the estimated discounted cumulative reinforcement to be received by $\mathcal{C}_i$ and following agents.

We adjust only HQ-values of agents active before trial end (N denotes the number of agents active during the last trial, α_{HQ} denotes the learning rate, and $\hat{o}_i$ the chosen subgoal for agent $\mathcal{C}_i$):

HQ.1 If $\mathcal{C}_i$ is invoked before agent $\mathcal{C}_{N-1}$, then we update according to
$HQ_i(\hat{o}_i) \leftarrow (1 - \alpha_{HQ})HQ_i(\hat{o}_i) + \alpha_{HQ}(R_i + \gamma^{t_{i+1}-t_i} HV_{i+1})$

HQ.2 If $\mathcal{C}_i = \mathcal{C}_{N-1}$, then $HQ_i(\hat{o}_i) \leftarrow (1 - \alpha_{HQ})HQ_i(\hat{o}_i) + \alpha_{HQ}(R_i + \gamma^{t_N - t_i} R_N)$

HQ.3 If $\mathcal{C}_i = \mathcal{C}_N$, and $i < M$, then $HQ_i(\hat{o}_i) \leftarrow (1 - \alpha_{HQ})HQ_i(\hat{o}_i) + \alpha_{HQ} R_i$

The first and third rules resemble traditional Q-learning rules. The second rule is an additional improvement for cases in which agent $\mathcal{C}_N$ has learned a (possibly high) value for a subgoal that is unachievable due to subgoals selected by previous agents.

HQ(λ)-learning: motivation. Q-learning's lookahead capability is restricted to one step. It cannot solve all RPPs because it cannot properly assign credit to different actions leading to identical next states (Whitehead, 1992). For instance, suppose you walk along a wall that looks the same everywhere except in the middle where there is a picture. The goal is to reach the left corner where there is reward. This RPP is solvable by a RP. Given the "picture" input, however, Q-learning with one-step lookahead would assign equal values to actions "go left" and "go right" because they both yield identical "wall" observations.

Consequently HQ-learning may suffer from Q-learning's inability to solve certain RPPs. To overcome this problem, we augment HQ by TD(λ)-methods for evaluating and improving policies in a manner analogous to Lin's offline Q(λ)-method (1993). TD(λ)-methods can learn from long-term effects of actions and thus disambiguate identical short-term effects of different actions. Our experiments indicate that RPPs are solvable by Q(λ)-learning with sufficiently high λ.

Q(λ).1 For the Q-tables we first compute desired Q-values $Q'(o_t, a_j)$ for $t = T, \ldots, 1$:
$Q'(o_T, a_T) \leftarrow R(s_T, a_T)$
$Q'(o_t, a_t) \leftarrow R(s_t, a_t) + \gamma((1-\lambda)V_{Active(t+1)}(o_{t+1}) + \lambda Q'(o_{t+1}, a_{t+1}))$

Q(λ).2 Then we update Q-values, beginning with $Q_N(o_T, a_T)$ and ending with $Q_1(o_1, a_1)$, according to
$Q_i(o_t, a_t) \leftarrow (1-\alpha_Q)Q_i(o_t, a_t) + \alpha_Q Q'(o_t, a_t)$

HQ(λ).1 For the HQ-tables we also compute desired HQ-values $HQ'_i(\hat{o}_i)$ for $i = N, \ldots, 1$:
$HQ'_N(\hat{o}_N) \leftarrow R_N$
$HQ'_{N-1}(\hat{o}_{N-1}) \leftarrow R_{N-1} + \gamma^{t_N - t_i} R_N$
$HQ'_i(\hat{o}_i) \leftarrow R_i + \gamma^{t_{i+1}-t_i}((1-\lambda)HV_{i+1} + \lambda HQ'_{i+1}(\hat{o}_{i+1}))$

HQ(λ).2 Then we update the HQ-values for agents $\mathcal{C}_1, \ldots, \mathcal{C}_{Min(N,M-1)}$ according to
$HQ_i(\hat{o}_i) \leftarrow (1-\alpha_{HQ})HQ_i(\hat{o}_i) + \alpha_{HQ}HQ'_i(\hat{o}_i)$

In principle, online Q(λ) may be used as well. Online Q(λ) should not use "action-penalty" (Koenig and Simmons, 1996), however, because punishing varying actions in response to ambiguous inputs can trap the agent in cyclic behavior.

Combined dynamics. Q-table policies are *reactive* and learn to solve RPPs. HQ-table policies are *metastrategies* for composing RPP sequences. Although Q-tables and HQ-tables do not explicitly communicate they influence each other through simultaneous learning. Their cooperation results in complex dynamics quite different from those of conventional Q-learning.

Utilities of subgoals and RPs are estimated by tracking how often they are part of successful subgoal/RP combinations. Subgoals that never or rarely occur in solutions become less likely to be chosen, others become more likely. In a certain sense subtasks compete for being assigned to subagents, and the subgoal choices "co-evolve" with the RPs. Maximizing its own expected utility, each agent implicitly takes into account frequent decisions made by other agents. Each agent eventually settles down on a particular RPP solvable by its RP and ceases to adapt. This will be illustrated by Experiment 1 in Section 6.3.

Estimation of average reward for choosing a particular subgoal ignores dependencies on previous subgoals. This makes local minima possible. If several rewarding suboptimal subgoal sequences are "close" in subgoal space, then the optimal one may be less probable than suboptimal ones. We will show in the experiments that this actually happens.

Exploration issues. Initial choices of subgoals and RPs may influence the final result - there may be local minimum traps. Exploration is a partial remedy: it encourages alternative competitive strategies similar to the current one. Too little exploration may prevent the system from discovering the goal at all. Too much exploration, however, prevents reliable estimates of the current policy's quality and reuse of previous successful RPs. To avoid over-exploration we use the Max-Boltzmann (Max-random) distribution for Q-values (HQ-values). These distributions also make it easy to reduce the relative weight of exploration (as opposed to exploitation): to obtain a deterministic policy at the end of the learning process, we increase P_{max} during learning until it finally achieves a maximum value.

Selecting actions according to the traditional Boltzmann distribution causes the following problems: (1) It is hard to find good values for the temperature parameter. (2) The degree of exploration depends on the Q-values: actions with almost identical Q-values (given a certain input) will be executed equally often. For instance, suppose a sequence of 5 different states in a maze leads to observation sequence $O_1 - O_1 - O_1 - O_1 - O_1$, where O_1 represents a single observation. Now suppose there are almost equal Q-values for going west or east in response to O_1. Then the Q-updates will hardly change the differences between these Q-values. The resulting random walk behavior will cost a lot of simulation time.

For RP training we prefer the Max-Boltzmann rule instead. It focuses on the greedy policy and only explores actions competitive with the optimal actions. Subgoal exploration is less critical though. The Max-random subgoal exploration rule may be replaced by the Max-Boltzmann rule or others.

Using gate-values. The shown learning rules are almost the same as for conventional Q-learning, but since we use a multiple agent system, the task of one particular agent is usually not fixed during the learning process. As long as different subgoals of previous agents are tried out, the subtask of an agent changes. Since during the learning process subgoals of agents are generated stochastically, we must take care that an agent focuses on the subtask which it has to carry out most of the times. We can do this by using little exploration, but also by constructing a learning rule which uses the probabilities of the generated subgoals of previous agents to determine the magnitude (size) of parameter changes for each learning step. We can do this by computing gate-values g_i which stand for the probability that the current sequence of $i-1$ subgoals is selected:

$$g_1 \leftarrow 1.0,$$

because before agent 1 is made active, no subgoals are selected.

$$g_{i+1} \leftarrow g_i P^{HQ_i}(o_j),$$

which means that the gate-value for agent $i+1$ is equal to the gate-value for agent i multiplied with the probability that agent i has chosen its particular subgoal. For Max-random exploration $P^{HQ}(o_j) = P_{max} + \frac{1-P_{max}}{|O|}$ if subgoal o_j has maximal HQ-value (we add the probability of selecting the max-subgoal to the probability of randomly selecting the subgoal) and $P^{HQ}(o_j) = \frac{1-P_{max}}{|O|}$ otherwise. These gate-values are then multiplied with the learning rate to determine the size of each learning step. Note that we make a independency assumption among subgoals to keep the system simpler. It might be true that different sequences of subgoals lead to the same system-state, just like in normal Q-learning different sequences of actions might lead to the same environmental state, but in the general case it is impossible to calculate all the different combinations of the subgoals which result in the obtained systemstate. We will call HQ-learning with the gate-values HQG-learning.

6.2.3 Experiments

We tested our system on three tasks in partially observable environments. The first task is comparatively simple — it will serve to exemplify how HQ discovers and stabilizes appropriate subgoal combinations. It requires finding a path from start to goal in a partially observable 10×10-maze, and can be collectively solved by three or more agents. We study system performance as more agents are added. The second, quite complex task involves finding a

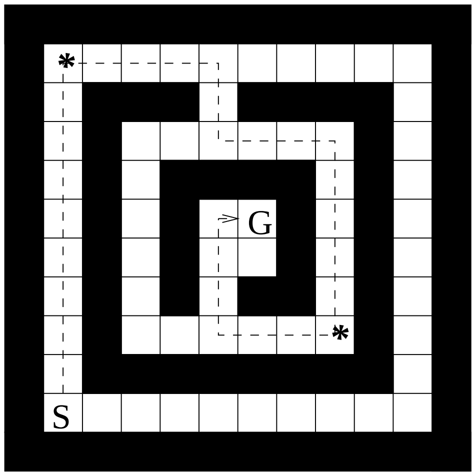


Figure 6.6: *A partially observable maze. The task is to find a path leading from start S to goal G. The optimal solution requires 28 steps and at least three reactive agents. The figure shows a possible suboptimal solution that costs 30 steps. Asterisks mark appropriate subgoals.*

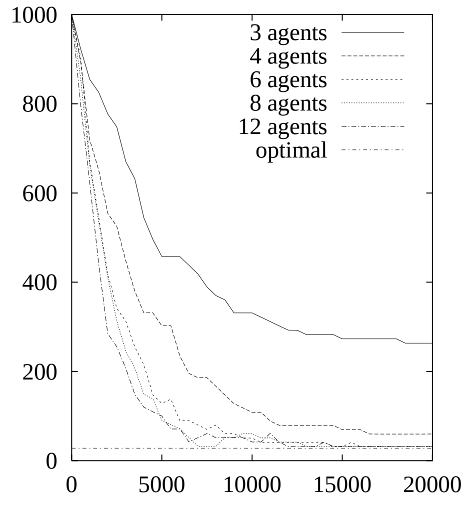
key which opens a door blocking the path to the goal. The optimal solution (which requires at least 3 agents) costs 83 steps. The third task shows that HQG-learning can be used for inductive transfer — problem solving knowledge acquired in one task can be used to speed-up solving another, similar, but more complex task.

Learning to Solve a Partially Observable Maze

Task. The first experiment involves the partially observable maze shown in Figure 6.6. The system has to discover a path leading from start position S to goal G. There are four actions with obvious semantics: *go west, go north, go east, go south.* 16 possible observations are computed by adding the "field values" of blocked fields next to the agent's position, where the field value of the west, north, east, and south field is 1, 2, 4, and 8, respectively — the agent can only "see" which of the 4 adjacent fields are blocked. Although there are 62 possible agent positions, there are only 9 highly ambiguous inputs. (Not all of the 16 possible observations can occur in this maze. This means that the system may occasionally generate unsolvable subgoals, such that control will never be transferred to another agent.) There is no deterministic, memory-free policy for solving this task. Stochastic memory-free policies (such as the one described in Jaakkola, Singh and Jordan, 1995) will also perform poorly. For instance, input 5 stands for "fields to the left and to the right of the agent are blocked". The optimal action in response to input 5 depends on the subtask: at the beginning of a trial, it is "go north", later "go south", near the end, "go north" again. Hence at least three reactive agents are necessary to solve this POMDP.

Reward function. Once the system hits the goal it receives a reward of 100. Otherwise the reward is zero. The discount factor $\gamma = 0.9$.

Parameters and experimental set-up. We compare systems with 3, 4, 6, 8, and 12 agents and noise-free actions. We also compare systems with 4, 8, and 12 agents whose actions selected during learning/testing are replaced by random actions with probability 10%. One experiment consists of 100 simulations of a given system. Each simulation consists of 20,000



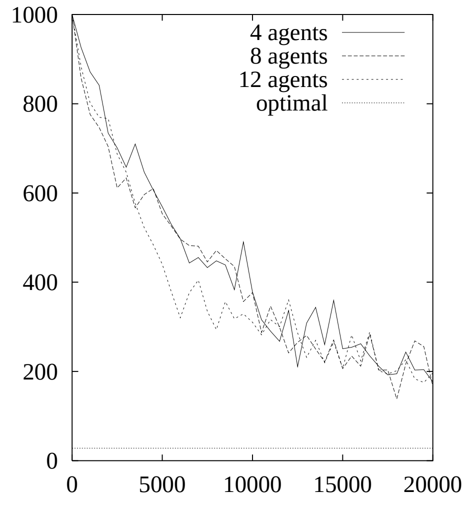


Figure 6.7: *left: HQ-learning results for the partially observable maze, for 3, 4, 6, 8, and 12 agents. We plot average test run length against trial numbers (means of 100 simulations). The system almost always converges to near-optimal solutions. Using more than the required 3 agents tends to improve performance. Right: results for 4, 8, and 12 agents whose actions are corrupted by 10% noise. In most cases they find the goal, although noisy actions decrease performance.*

trials. T_{max} is 1000. After every 500th trial there is a test run during which actions and subgoals with maximal table entries are selected (P_{max} is set to 1.0). If the system does not find the goal during a test run, then the trial's outcome is counted as 1000 steps.

After a coarse search through parameter space, we use the following parameters for all experiments: $\alpha_Q = .05$, $\alpha_{HQ} = .2$, $\forall i: \ T_i = .1$, $\lambda = .9$ for both HQ-tables and Q-tables. P_{max} is set to .9 and linearly increased to 1.0. All table entries are initialized with 0.

Results. Figure 6.7A plots average test run length against trial numbers. Within 20,000 trials most systems almost always find near-optimal deterministic policies. Consider Table 6.1. The largest systems are always able to decompose the POMDP into a sequence of RPPs. The average number of steps is close to optimal. In approximately 1 out of 8 cases, the optimal 28-step path is found. In most cases one of the 30-step solutions is found. Since the number of 30-step solutions is much larger than the number of 28-step solutions (there are many more appropriate subgoal combinations), this result is not surprising.

Systems with more than 3 agents are performing better — here the system profits from having more free parameters. More than 6 agents do not help though. All systems perform significantly better than the random system, which finds the goal in only 19% of all 1000 step trials.

In case of noisy actions (the probability of replacing a selected action by a random action is 10%), the systems still reach the goal in most of the simulations (see Figure 3B). In the final trial of each simulation, systems with 4, 8, and 12 agents find the goal with probabilities of 86, 87, and 84 percent, respectively. There is no significant difference between smaller and larger systems.

System	Av. steps	(%) Goal	Av. sol.	(%) Optimal
3 agents	263	76	30	3
4 agents	60	97	31	6
6 agents	31	100	31	14
8 agents	31	100	31	12
12 agents	32	100	32	6
4 agents 10% noise	177	86	43	2
8 agents 10% noise	166	87	41	2
12 agents 10% noise	196	84	43	0
Random	912	19	537	0

Table 6.1: *HQ-learning results for random actions replacing the selected actions with probability 0% and 10%. All table entries refer to the final test trial. The 2nd column lists average trial lengths. The 3rd column lists goal hit percentages. The 4th column lists average path lengths of solutions. The 5th column lists percentages of simulations during which the optimal path is found.*

We also studied how the system adds agents during the learning process. The 8-agent system found solutions using 3 (4, 5, 6, 7, 8) agents in 8 (19, 16, 17, 21, 19) simulations. Using more agents tends to make things easier. During the first few trials 3 agents were used on average, during the final trials 6. Less agents tend to give better results, however. Why? Systems that fail to solve the task with few subgoals start using more subgoals until they become successful. But the more subgoals there are, the more possibilities to compose paths, and the lower the probability of finding a shortest path in this maze.

Experimental analysis. How does the (3-agent) system discover and stabilize subgoal combinations (SCs)? The only optimal 28-step solution uses observation 2 as the first subgoal (5th top field) and observation 9 as the second (southwest inner corner). There are several 30-step solutions, however — e.g., SCs (3, 12), (2, 12), (10, 12).

Figure 6.8 shows how SCs evolve by plotting them every 10 trials (observation 16 stands for an unsolved subgoal). The first 10,000 SCs are quite random, and the second agent often is not able to achieve its subgoal at all. Later, however, the system gradually focuses on successful SCs. Although useful SCs are occasionally lost due to exploration of alternative SCs, near simulation end the system converges to SC (3, 12).

The goal is hardly ever found prior to trial 5200 (the Figure does not show this). Then there is a sudden jump in performance — most later trials cost just 30 steps. From this moment on observation 12 is used as second subgoal in more than 95% of all cases, and the goal is found in about 85%. The first subgoal tends to vary among observations 2, 3 and 10. Finally, around 16,000 trials, the first subgoal settles down on observation 3, although observation 2 would work as well.

The Key and the Door

Task. The second experiment involves the 26×23 maze shown in Figure 6.9. Starting at S, the system has to (1) fetch a key at position K, (2) move towards the "door" (the shaded area) which normally behaves like a wall and will open (disappear) only if the agent is in possession of the key, and (3) proceed to goal G. There are only 11 different, highly

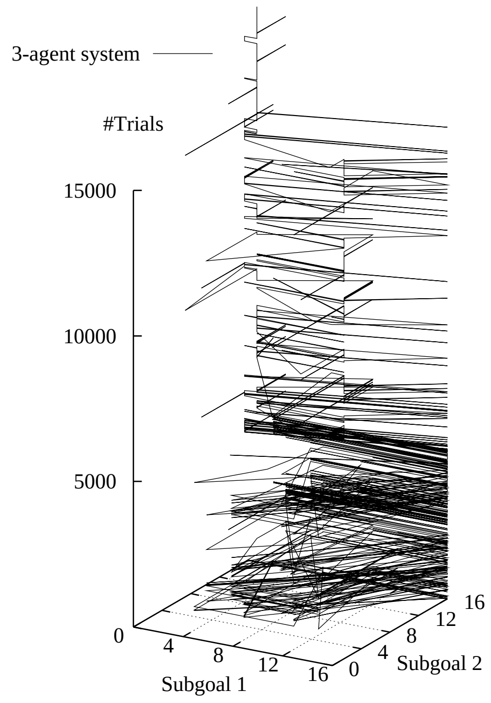


Figure 6.8: *Subgoal combinations (SCs) generated by a 3-agent system, sampled at intervals of 10 trials. Initially many different SCs are tried out. After 10,000 trials HQ explores less and less SCs until it finally converges to SC (3, 12).*

ambiguous inputs; the key (door) is observed as a free field (wall). The optimal path takes 83 steps.

Reward function. Once the system hits the goal, it receives a reward of 500. For all other actions there is a reward of -0.1. There is *no* additional, intermediate reward for taking the key or going through the door. The discount factor $\gamma = 1.0$.

Parameters. The experimental set-up is analogous to the one in section 3.1. We use systems with 3, 4, 6 and 8 agents, and systems with 8 agents whose actions are corrupted by different amounts of noise (5%, 10%, and 25%). $\alpha_Q = .05$, $\alpha_{HQ} = .01$ $\forall i : T_i = .2$. P_{max} is linearly increased from .4 to .8. Again, $\lambda = .9$ for both HQ-tables and Q-tables, and all table entries are zero-initialized. One simulation consists of 20,000 trials.

Results. We first ran 20,000 thousand-step trials of a system executing random actions. It never found the goal. Then we ran the random system for 3000 10,000 step trials. The shortest path ever found took 1,174 steps. We observe: goal discovery within 1000 steps (and without "action penalty" through negative reinforcement signals for each executed action) is very unlikely to happen.

Figure 6.10A and Table 6.2 show HQ-learning results for noise-free actions. Within 20,000

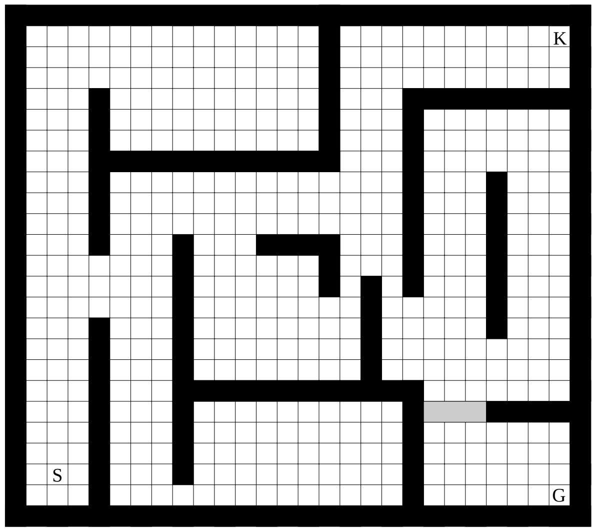
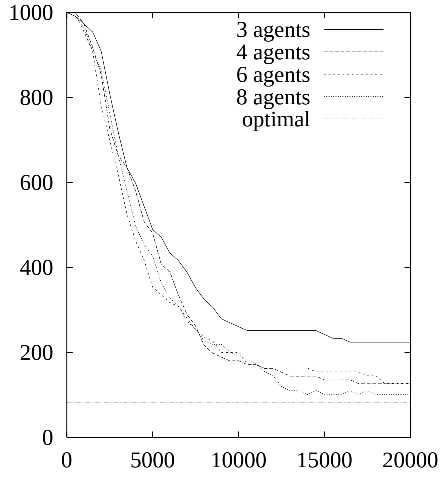


Figure 6.9: *A partially observable maze containing a key K and a door (grey area). Starting at S, the system first has to find the key to open the door, then proceed to the goal G. The shortest path costs 83 steps. This optimal solution requires at least three reactive agents. The number of possible world states is* 960.



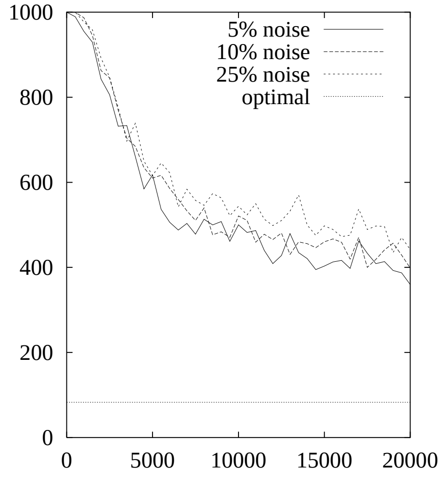


Figure 6.10: *left: HQ-learning results for the "key and door" problem. We plot average test run length against trial number (means of 100 simulations). Within 20,000 trials systems with 3 (4, 6 and 8) agents find good deterministic policies in 85% (96%, 96% and 99%) of the simulations. Right: HQ-learning results with an 8 agent system whose actions are replaced by random actions with probability 5%, 10%, and 25%.*

trials good, deterministic policies are found in almost all simulations. Optimal 83 step paths are found with 3 (4, 6, 8) agents in 8% (9%, 8%, 6%) of all simulations. During the few runs that did not lead to good solutions the goal was rarely found at all. This reflects a general

System	Av. steps	(%) Goal	Av. sol.	(%) Optimal
3 agents	224	85	87	8
4 agents	126	96	90	9
6 agents	127	96	91	8
8 agents	101	99	92	6
8 agents (5% noise)	360	92	304	0
8 agents (10% noise)	399	90	332	0
8 agents (25% noise)	442	84	336	0
Random	*9310	19	6370	0

Table 6.2: *Results of 100 HQ-learning simulations for the "key and door" task. All table entries refer to the final test trial. The second column lists average trial lengths. The third lists goal hit percentages. The fourth lists average path lengths of solutions. The fifth lists percentages of simulations during which the optimal 83 step path was found. HQ-learning could solve the task with a limit of 1000 steps per trial. Random search needed a 10,000 step limit.*

problem: in the POMDP case exploration issues are trickier than in the MDP case — much remains to be done to better understand them.

If random actions are taken in 5% (10%, 25%) of all cases, the 8 agent system still finds the goal in 92% (90%, 84%) of the final trials (see table 6.2). In many cases long paths (300 — 700 steps) are found. The best solutions use only 84 (91, 118) steps, though. Interestingly, a little noise (e.g. 5%) decreases performance a lot, but much more noise does not lead to much worse results.

Inductive transfer

Partial observability is not always a big problem. For example, it may facilitate generalization and knowledge transfer, if different world states leading to similar inputs require similar treatment. In the next experiment, we exploit this potential for incremental learning: the system learns to transfer knowledge from a relatively simple maze's solution to a more complex one's.

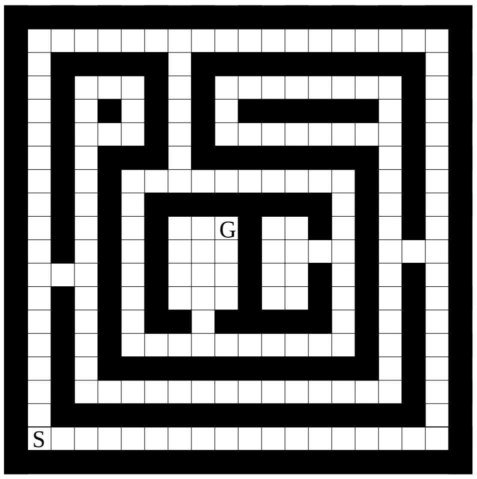
Task. First the system is trained to solve the maze from the first experiment (see Figure 6.6). After 20,000 trials, the system is placed in the more complex maze shown in Figure 6.11(A), without resetting its Q- and HQ-tables. Then we spend an additional 20,000 trials on solving the second maze. We compare the results to those obtained by learning from scratch (using only the more difficult maze for training).

Reward function. Once the system hits the goal, it receives a reward of 200. Once it bumps against a wall, the reward is -2.0. All other actions are penalized by negative reward -0.1. The discount factor γ is 1.0.

Parameters and experimental set-up. We test systems with 6 and 8 agents for both transfer learning and learning from scratch. We use HQG, the gate-based HQ system which leads to more stable learning performance which is an advantage for incremental learning, although the learning rate is slightly lower and it finds optimal policies less often.

One experiment consists of 100 simulations of a given system. The maximal number of steps per trial is 1000. Every 200 trials there is a test run.

After a coarse search through parameter space, we use the following parameters for all competitors: $\alpha^Q = 0.04$, $\alpha^{HQ} = 0.01$, $\forall i: T_i = 0.5$, $\lambda = 0.9$ for both HQ-tables and Q-tables. P_{max} is linearly increased from 0.6 in the beginning to 0.9 at the end of the first maze and then from 0.9 to 1.0 at the end of the simulation (now consisting of 40,000 trials, instead of 20,000). For systems which learn from scratch, P_{max} is linearly increased from 0.6 in the beginning to 1.0 at the end of a simulation (also consisting of 40,000 trials).



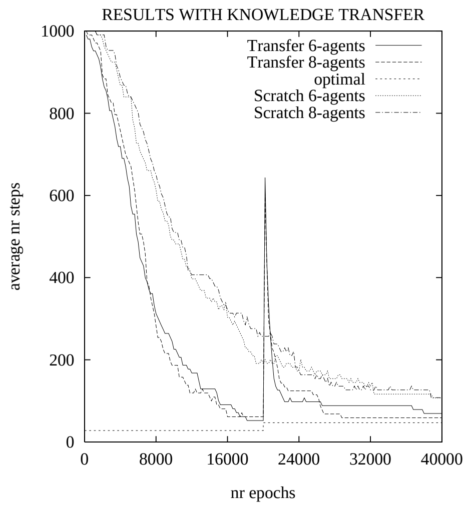


Figure 6.11: *(A) The system is first trained on the maze from Figure 6.6, then moved to the similar, but more complex, partially observable maze depicted above. There are 11 possible ambiguous observations. The optimal solution costs 47 steps. (B) HQ-learning results for the incremental learning task for 6 and 8 agents, with and without knowledge transfer. The transfer-based systems are trained for 20,000 trials on the maze from Figure 6.6. Then they are moved to the maze from Figure 6.11(A), for which they have to learn additional observation-action pairs. We plot average test run length against trial (epoch) numbers. After being primed by the first task, the system quickly learns good policies for the second maze.*

Results. Figure 6.11(B) plots average test run length against trial numbers, for systems with and without knowledge transfer. The plot shows performance on the more complex maze, except for the first 20,000 trials of transfer-based systems, which correspond to performance on the simpler maze. All systems almost always find near-optimal deterministic policies. Transfer-based systems with 6 and 8 agents find deterministic policies leading to the goal in 99%, and 100% of the simulations, respectively. Both systems that learn from scratch find good deterministic policies in 95% of the simulations. Obviously, knowledge transfer is beneficial: the system quickly refines solutions to the first partially observable maze to obtain good solutions for the second maze. For instance, after being moved to the second maze, the 8-agent system has found 75 (90, 100) times a good solution after 600 (1,800, 8,800) trials. Average trial length after 20,000 trials in the second maze is 59 steps, whereas the optimal solution requires 47 steps (the worst solution ever found requires 83 steps). Note that the system often does not immediately find the solution to the new maze. This is largely due to the new observation in the left corridor for which a new action (*go-north*) has to be learned.

Why does the system not always find optimal solutions, but sometimes gets stuck in "local maxima"? One reason may be that "local maxima" are close to global maxima: the cumulative discounted reward for a 47-step solution is 195.4 at the starting state, whereas the cumulative reward for a 83-step solution is only slightly less, namely 191.8.

6.2.4 Discussion

HQ's advantages.

1. Most POMDP algorithms need *a priori* information about the POMDP, such as the total number of environmental states, the observation function, or the action model. HQ does not. HQ only has to know which actions can be selected.

2. Unlike "history windows", HQ-learning can in principle handle arbitrary time lags between events worth memorizing. To focus this power on where it is really needed, short history windows may be included in the agent inputs to take care of the shorter time lags. This, however, is orthogonal to HQ's basic ideas.

3. To reduce memory requirements, HQ does not explicitly store all experiences with different subgoal combinations. Instead it estimates the average reward for choosing particular subgoal/RP combinations, and stores its experiences in a single sequence of Q- and HQ-tables. These are used to make successful subgoal/RP combinations more likely. HQ's approach is advantageous in case the POMDP exhibits certain regular structure: if one and the same agent tends to receive RPPs achievable by similar RPs then it can "reuse" previous RPP solutions.

4. HQ-learning can immediately generalize from solved POMDPs to "similar" POMDPs containing more states but requiring identical actions in response to inputs observed during subtasks (see the third experiment).

HQ's current limitations.

1. Like Q-learning, HQ-learning allows for representing RPs and subgoal evaluations by function approximators other than look-up tables. We have not implemented this combination, however.

2. An agent's current subgoal does not uniquely represent previous subgoal histories. This means that HQ-learning does not really get rid of the "hidden state problem" (HSP). HQ's HSP is not as bad as Q's, though. Q's is that it is impossible to build a Q-policy that reacts differently to identical observations, which may occur frequently. Appropriate HQ-policies, however, do exist.

   To deal with this HSP one might think of using subgoal trees instead of sequences. All possible subgoal sequences are representable by a policy tree whose branches are labeled with subgoals and whose nodes contain RPs for solving RPPs. Each node stands for a particular history of subgoals and previously solved subtasks — there is no HSP any more. Since the tree grows exponentially with the number of possible subgoals, however, it is practically infeasible in case of large scale POMDPs.

3. In case of noisy observations transfer of control may happen at inappropriate times. A remedy may be to use more reliable inputs combining successive observations.

4. In case of noisy actions, an inappropriate action may be executed right before or after passing control. The resulting new subtask may not be solvable by the next agent's RP. A remedy similar to the one mentioned above may be to represent subgoals as pairs of successive observations. Another possibility is to create an architecture which allows for returning control.

5. If there are many possible observations then subgoals will be tested infrequently. This may delay convergence. To overcome this problem one might either try function approximators instead of look-up tables or let each agent generate a set of multiple, alternative subgoals. In the latter instance, once a subgoal in the set is reached, control is transferred to the next agent.

6. HQ has a severe exploration problem. It relies on random walk behavior for finding the goal the first time, which can be quite expensive for some problems — in the worst case the agents needs exponential search time (Whitehead, 1992). Using action-penalty or counting methods for POMDPs may result in even worse, cyclic, behavior. Thus, better ways should be found for POMDP exploration. Possibilities include selecting a consistent action for the same observation so that e.g. counting can be used or using policy trees (as mentioned above) so that the HSP completely disappears. It remains a challenging issue to find exploration algorithms which work well for a large variety of POMDPs and algorithms solving them, however.

6.3 Related Work

Other authors proposed hierarchical reinforcement learning techniques, e.g., Schmidhuber (1991b), Singh (1992), Dayan and Hinton (1993), Tham (1995), Sutton (1995b), and Thrun and Schwartz (1995). Their methods, however, have been designed for MDPs. Since the current focus is on POMDPs, this section is limited to a summary of previous POMDP approaches and methods related to HQ due to their use of abstract behaviors instead of only single actions.

Recurrent neural networks. Schmidhuber (1991c) uses two interacting, gradient-based recurrent networks for solving POMDPs. The "model network" serves to model (predict) the environment, the other one uses the model net to compute gradients maximizing reinforcement predicted by the model (this extends ideas by Nguyen and B. Widrow, 1989; and Jordan and Rumelhart, 1990). To our knowledge this work presents the first successful reinforcement learning application to simple non-Markovian tasks (e.g., learning to be a flipflop). Lin (1993) also uses combinations of controllers and recurrent nets. He compares time-delay neural networks (TDNNs) and recurrent neural networks.

Despite their theoretical power, standard recurrent nets run into practical problems in case of long time lags between relevant input events. Although there are recent attempts at overcoming this problem for time series prediction (e.g., Schmidhuber 1992; Hihi and Bengio 1996; Hochreiter and Schmidhuber 1997; Lin, Horne and Giles 1996). In general, learning to control is also more complex, since the system does not only have to predict, but also make the right choices. Another important difference is that the system itself is responsible for the generated trajectories. Therefore if some trajectories need little memory, whereas other, better, trajectories need more memory, the latter may have much smaller probability of being found. Furthermore, exploration and system adaptions can cause previously acquired memory to become useless, since trajectories may change significantly.

Map learning. An altogether different approach for solving POMDPs is to learn a map of the environment. This approach is currently considered very attractive, and especially suitable for navigating mobile robots, where the agent has *a priori* information about the result of its actions (the odometry is known). Thrun (1998) successfully applies a map-learning algorithm to his robot Xavier, which first learns a grid-based map of an office building environment and then compiles this grid-based map into a topological map to speed up goal directed planning.

Map learning approaches may suffer when state uncertainty and odometry errors are large, however. Therefore, some mobile robots are equipped with multiple sensors, and inputs from these sensors are combined to decrease the uncertainty about the state. Van Dam (1998) describes techniques for fusing the data from different sources in order to learn environmental models.

Bayse, Dean and Vitter (1997) discuss the complexity of map learning in uncertain environments. If there is no uncertainty (in observations or actions) in the environment, then map learning is simple — we can just try all unexplored edges. In case of uncertainty in both actions and observations, they prove the existence of PAC algorithms, but restrict the environment in two ways: (1) the agent needs to know the direction in which it went after executing an action, although it does not have to know to which (unintended) state it went (this is needed for being able to find the way back), and (2) there should exist some set of landmarks (with unique observations).

Hidden Markov Models. McCallum's utile distinction memory (1993) is an extension of Chrisman's perceptual distinctions approach (1992), which combines Hidden Markov Models (HMMs) and Q-learning. The system is able to solve particular small, noisy POMDPs by splitting "inconsistent" HMM states whenever the agent fails to predict their utilities (but instead experiences quite different returns from these states). The approach cannot solve problems in which conjunctions of successive perceptions are useful for predicting reward while independent perceptions are irrelevant. HQ-learning does not have this problem — it deals with perceptive conjunctions by using multiple agents if necessary.

Belief state. Some authors have proposed speeding up computing solutions to POMDPs using belief states by using function approximators. Boutilier and Poole (1996) use Bayesian networks to represent POMDPs, and use these more compact models to accelerate policy computation. Parr and Russell (1995) use gradient descent methods on a continuous representation of the value function, although the use of a function approximator makes their method an heuristic algorithm. Their experiments show significant speed-ups on certain small problems. D'Ambrosio (1996) uses K-abstraction to discretize belief state space and then uses Q-learning to learn value functions. He shows how his algorithm can find solutions to diagnosis problems containing 256 states. A belief network is used for modeling the problem. The results show that the method can quickly improve its policy, although the success of the algorithm may heavily depend on the amount of uncertainty the agent has about the true state.

Memory bits. Littman (1994) uses branch-and-bound heuristics to find suboptimal memoryless policies extremely quickly. To handle mazes for which there is no safe, deterministic, memoryless policy, he replaces each conventional action by two actions, each having the additional effect of switching a "memory bit" on or off. Good results are obtained with a toy problem. Thus, the method directly searches in policy space. It may have problems if only few deterministic policies will lead to the goal. Therefore to better evaluate policies, Littman started the agent from each possible initial state.

Cliff and Ross (1994) use Wilson's (1994) classifier system (ZCS) for POMDPs. There are memory bits which can be set and reset by actions. ZCS is trained by bucket-brigade and genetic algorithms. The system is reported to work well on small problems but to become unstable in case of more than one memory bit. Also, it is usually not able to find optimal deterministic policies. Wilson (1995) recently described a more sophisticated classifier system which uses prediction accuracy for calculating fitness, and a genetic algorithm working in environmental niches. His study shows that this makes the classifiers more general and more accurate.

Martin (1998) also used memory bits in his system Candide, which could be put on or off independently by different actions. He used Q(1) and Monte Carlo experiments to evaluate actions, and was quite successful in solving large deterministic POMDPs. He used a clever exploration method which did not allow the agent to choose a different action for a specific observation if that observation is seen multiple times in an experiment.

One possible problem with memory bits is that tasks such as those in Section 6.2.3 require (1) switching on/off memory bits at precisely the right moment, and (2) keeping them switched on/off for long times. During learning and exploration, however, each memory bit will be very unstable and change all the time — algorithms based on incremental solution refinement will usually have great difficulties in finding out when to set or reset it. Even if the probability of changing a memory bit in response to a particular observation is low it will eventually change if the observation is made frequently. Like HQ-learning, Candide did not suffer a lot from such problems. These systems take care that the internal memory is not changed too often, which makes them much more stable.

Program evolution with memory cells. Certain techniques for automatic program synthesis based on evolutionary principles can be used to evolve short-term memorizing programs that read and write memory cells during runtime (e.g., Teller, 1994). A recent such method is Probabilistic Incremental Program Evolution (PIPE — Salustowicz and Schmidhuber, 1997). PIPE iteratively generates successive populations of functional programs according to an adaptive probability distribution over all possible programs. On each iteration it uses the best program to refine the distribution. Thus it stochastically generates better and better programs. A memory cell-based PIPE variant has been successfully used to find well performing stochastic policies for partially observable mazes. On serial machines, however, their evaluation tends to be computationally much more expensive than HQ.

Learning policy trees. Ring's system (1994) constructs a policy tree implemented in higher order neural networks. To disambiguate inconsistent states, new higher-level nodes are added to incorporate information hidden "deeper" in the past. The system is able to quickly solve certain non-Markovian maze problems but often is not able to generalize from previous experience without additional learning, even if the optimal policies for old and new task are identical. HQ-learning, however, can reuse the same policy and generalize well from previous to "similar" problems.

McCallum's U-tree (1996) is quite similar to Ring's system. It uses prediction suffix trees (see Ron, Singer and Tishby, 1994) in which the branches reflect decisions based on current or previous inputs/actions. Q-values are stored in the leaves. Statistical tests are used to decide whether groups of instances in a leave correspond to significantly different utility estimates. If so, the cluster is split. McCallum's recent experiments demonstrate the algorithm's ability to control a simulated car by learning to switch lanes on a highway-task involving a huge number of different inputs while still containing hidden state.

Another problem with Ring's and McCallum's approaches is that they have to store the whole sequence of observations and actions. Hence for large "time windows" used for sampling observations, the methods will suffer from space limitations.

Consistent Representations. Whitehead (1992) uses the "Consistent Representation (CR) Method" to deal with inconsistent internal states which result from "perceptual aliasing". CR uses an "identification stage" to execute perceptual actions which collect the information needed to define a consistent internal state. Once a consistent internal state has been identified, actions are generated to maximize future discounted reward. Both identifier and controller are adaptive. One limitation of his method is that the system has no means of remembering and using any information other than that immediately perceivable. HQ-learning, however, can profit from remembering previous events for very long time periods.

Levin Search. Wiering and Schmidhuber (1996) use Levin search (LS) through program space (Levin, 1973) to discover programs computing solutions for large POMDPs. LS is of interest because of its amazing theoretical properties: for a broad class of search problems, it has the optimal order of computational complexity. For instance, suppose there is an algorithm that solves a certain type of maze task in $O(n^3)$ steps, where n is a positive integer representing the problem size. Then LS will solve the same task in at most $O(n^3)$ steps. Wiering and Schmidhuber show that LS supplied with a set of conditional plans (abstract behaviors) can solve very large POMDPs for which the algorithmic complexity (Li and Vitányi, 1993) of the solutions is low.

Success-Story Algorithm. Wiering and Schmidhuber (1996) also extend LS to obtain an incremental method for generalizing from previous experience ("adaptive LS"). To guarantee that the lifelong history of policy changes corresponds to a lifelong history of reinforcement accelerations, they use the success-story algorithm (SSA, e.g., Schmidhuber, Zhao and Schraudolph 1997a; Zhao and Schmidhuber 1996, Schmidhuber, Zhao and Wiering 1997b). This can lead to further significant speed-ups. SSA is actually not LS-specific, but a general approach that allows for plugging in a great variety of learning algorithms. For instance, in additional experiments with a "self-referential" system that embeds its policy-modifying method within the policy itself, SSA is able to solve huge POMDPs using the proper initial bias (Schmidhuber et al. 1997a).

Using multiple abstract behaviors

Using abstract behaviors instead of single actions makes it possible to plan on a higher level and compress solutions, thereby facilitating policy construction considerably. Some early work in this direction has been performed by Singh (1992) who developed compositional Q-learning. Other early work was done by Thrun (1995) who developed Skills, a system which was able to learn and reuse particular reactive policies. HQ-learning went a step beyond these approaches, however, since it showed how abstract behaviors could directly learn from the Q-values of subsequent behaviors without taking single action values into account.

Like HQ-learning, Humphrys' W-learning (1996) uses multiple agents which use Q-learning to learn their own behaviors. A major difference is that his agents' skills are prewired — different agents focus on different input features and receive different rewards. "Good" reward functions are found by genetic algorithms. An important goal is to learn which agent to select for which part of the input space. Eight different learning methods implementing cooperative and competitive strategies are tested in a rather complex dynamic environment, and seem to lead to reasonable results.

Digney (1996) describes a nested Q-learning technique based on multiple agents learning independent, reusable skills. To generate quite arbitrary control hierarchies, simple actions and skills can be composed to form more complex skills. Learning rules for selecting skills and for selecting actions are the same, however. This may make it hard to deal with long reinforcement delays. In experiments the system reliably learns to solve a simple maze task.

Sutton, Precup and Singh (1998) and also Parr and Russel (1997) discuss approaches based on using abstract behaviors for speeding up finding solutions to Markov decision problems. Using Sutton's terminology here, these systems consist of options (abstract/macro actions) instead of single step actions, which are associated with a termination condition. Executing an option can take a variable amount of time, and that is why the problems are modeled as semi-Markov decision processes. Precup, Sutton and Singh (1998) prove that RL with options still converges, as long as we can guarantee that options stop. Results with Q-learning and abstract behaviors show that their methods significantly speed up policy construction. Although these approaches have not yet been used for solving POMDPs, the use of options could, just like HQ's subgoals, get rid of a lot of hidden state.

Martin (1998) developed Candide, a system which first uses Q(1)-learning to learn a set of behaviors, after which the system learns to hierarchically compose them to learn to solve more difficult problems. By first teaching the system a set of basic tasks, it was later able to learn to combine them in general policies for solving quite difficult block world problems.

Dietterich (1997) developed the MAXQ value function decomposition method for hierarchical reinforcement learning. It also consists of multiple abstract behaviors which are combined through learning Q-nodes in a policy tree. Each Q-node determines which branch of the tree is selected, and is followed by a primitive action (possibly an abstract behavior) and a descendant if the node is not a leave node. In this way, highly complex behaviors can be learned by first engineering (by hand) useful abstract policy trees and then learning the Q-values.

The difference of these approaches compared to HQ-learning, is that HQ-learning learns to decompose the task into subtasks and at the same time learns behaviors solving the subtasks without *a priori* knowledge about the tasks. If solutions do not work, HQ-learning will change subtasks and try to learn new behaviors solving them. Most other systems use either a prewired hierarchical decompositions or prewired skills which makes learning faster, but requires human engineering.

HQ-learning can also be seen as a system which combines learning a structure and the parameters of the structure. The structure is adapted to make learning the parameters easier. Learned parameters which work well ensure that the structure will remain more stable. In this way, the coupled system can have advantages compared to other one-sided learning systems. E.g. we also tried using a prewired subtask decomposition so that only policies needed to be learned. For this we used the first maze (Figure 6.6) and used the subgoals belonging to the decomposition of the optimal path (SCs 2 and 8). Surprisingly this resulted in worse training performance than learning the decomposition at the same time. This might be explained by the fact that the number of goal-finding solutions had become much smaller than that of the coupled system.. Furthermore, the agent was restricted to learning the optimal solution. Therefore its learning dynamics suffered from the additional constraints.

6.4 Conclusion

POMDPs are a difficult class of problems. Exact algorithms for solving them are infeasible for large problems, and most heuristic methods do not scale up very well. We developed a new algorithm, HQ-learning, which is based on learning to decompose a problem into a set of smaller problems solvable by different learning agents. This method has been shown to solve a number of semi-large, deterministic POMDPs. Since HQ scales up with the number of required agents and not with the number of states, it can in theory solve very large POMDPs for which there exist low complexity solutions. HQ does not solve all problems: in particular it has problems with uncertainty in the actions and observations, although it may still find competitive solutions for such problems. HQ-learning also supposes a fixed starting state which is another limitation.

POMDP algorithms are based on the construction of an internal state (IS) based on the sequence of observations. Efficient POMDP algorithms should therefore focus themselves on controlling the dynamics of their internal state. Based on recollected experiences, an algorithm could try to transform its IS using mental actions. If an algorithm could learn to control these mental actions, an agent might be able to learn the skill of reflection. The importance of reflection for intelligent behavior may be clear, but many problems remain to be solved until it becomes realistic. One important question is how mental states can be constructed as long as they are not automatically associated with obtaining reward.

Finally, there is a need for algorithms which can explore POMDPs more efficiently. Even map learning algorithms usually rely on random walk behavior for finding some landmark state (Bayse et al., 1997). As long as exploration issues are not resolved, we cannot expect to obtain efficient heuristic methods for a large number of POMDPs.

Reinforcement learning approaches using the construction of *a priori* policy trees or abstract behaviors can play an important role for solving more difficult RL problems. Current investigations in this direction are already being made on a variety of problems and show promising results, e.g., (Sutton et al., 1998; Dietterich, 1997). HQ is related to such methods as well as to exact policy tree construction methods (Kaelbling et al., 1995), since it allows for using policy trees where each node is represented as a behavior (reactive policy) and transitions between nodes are stored by subgoals (observations). Thus, HQ-learning can store the optimal policy for POMDPs given a unique initial belief state. Furthermore, HQ can allow for compression of policy trees, since often the same action for a particular observation may be optimal in a large region of the state space. Using policy nodes with M subgoals per policy (transitions from one node to the next), we might be able to decrease the complexity of (naive) exact algorithms from $O(|A|^{|O|^T})$ to $O(|A|^{|O|^{M^K}})$, where K is the length of the (longest) subgoal sequence. We can see the large gain in case the horizon (T) is large and the number of subgoals (M) and the length of subgoal sequences (K) is small. More investigation is needed to combine policy nodes with such exact algorithms, however.

Chapter 7

Function Approximation for RL

In the first part of this thesis we have described RL methods for finite state spaces for which it is possible to exactly store the optimal value function with lookup table representations. For high-dimensional or continuous state spaces, however, we need to employ function approximators for compactly representing an approximation of the value functions.

Function approximators (FAs) such as neural networks can be used to represent the value or Q-function by mapping an input vector to a Q-value. FAs consist of many adjustable parameters which are trained on learning examples to minimize the FA's approximation error of the target function. A very useful property of FAs is that they are able to generalize: if we train a FA on a set of training examples, the FA can interpolate between (or even extrapolate from) them to create a mapping from inputs to outputs for the complete input space. Learning a good approximation is very difficult (Judd, 1990), however. It can take a huge amount of time and even be unsuccessful. We call this the learning problem. However, sometimes it is possible to learn very useful value functions for huge state spaces after training the FAs on a relatively small number of training examples. This was demonstrated by the success of learning to play backgammon [1] with neural networks (Tesauro, 1992).

Outline of this chapter. In Section 7.1, we will describe three different function approximators: linear networks, neural gas (Martinetz and Schulten, 1991; Fritzke, 1994), and CMACs (Albus, 1975b; Albus, 1975a). Then in Section 7.2, we describe how they can be combined with direct RL methods. In Chapter 4, we have seen that the use of world models speeds up computing good value functions. Therefore, we would like to use world models in combination with function approximators as well. In Section 7.3, we will describe how we can combine models with the function approximators described in Section 7.1. Then, in Section 7.4, we will describe an experimental study on using function approximators in a challenging domain: learning to play soccer with multiple agents. Here we compare different methods by their ability to learn to beat a prewired soccer team. In Section 7.5, we conclude this chapter with some closing remarks.

7.1 Function Approximation

Principles of FAs. To use function approximators, we construct a u-dimensional input vector x based on selecting characteristic features for describing the state of the system or

[1] There are on the order of 10^{20} positions (states) on the backgammon board, whereas TD-Gammon had already learned a good policy after learning on about 10^7 examples.

agent. Constructing an input vector is quite important, since it provides the FA with all information to make its estimations from and therefore particular input designs can ease the burden of the learning problem. When we are constructing an input vector, it is important not to have ambiguities (see Chapter 6). More generally, we should try to circumvent having to deal with small distances between input vectors which must be mapped to very different outputs.[2] Furthermore, learning speed is largest if all input dimensions contribute something to the output (the output should be sensitive to all inputs), otherwise it is better to remove the non-contributing inputs. Usually we do not have *a priori* knowledge which enables us to decide which inputs are important and thus we should examine the output's sensitivity to an input's value after some amount of learning time.

FAs receive the input vector x, and compute a scalar output y (it is straightforward to extend the following presentation to using an output vector) using the mapping F:

$$y = F(x)$$

The FA F consists of a number of adjustable parameters (e.g. weights) and processing elements (nodes, neurons) structured in a specific *topology* (architecture). The topology can be static or dynamic. *Static topologies* are defined *a priori* by the engineer and during the training phase only the parameters in the FA are changed. *Dynamic topologies* develop and adjust themselves during the learning process. Although dynamic structures could make learning functions much easier with particular FAs (Baum, 1989), finding correct structures is generally considered to be a hard problem.

Local vs. Global FAs. There are many different function approximators and each kind of FA has its own advantages and disadvantages. One important characteristic for FAs is their degree of locality. We will call a FA a *local model* if the FA only uses a fraction (e.g. less than 20%) of all processing elements for computing the output given an input. Completely local models (such as lookup tables) only use a single processing element (e.g. a table entry) for computing the output. Examples of local models are: Kohonen networks (Kohonen, 1988), neural gas (Martinetz and Schulten, 1991; Fritzke, 1994), bumptrees (Omohundro, 1991; Landelius, 1997), decision trees, and CMACs (Albus, 1975b; Albus, 1975a). The counterpart of local models are *global models*, which use (almost) the entire representation for deriving the output. They usually interpolate more smoothly between training examples than local models, but also tend to overgeneralize more. Examples of global models are linear networks, sigmoidal feedforward neural networks trained with backpropagation (Werbos, 1974; Rumelhart et al., 1986), Hopfield Networks (Hopfield, 1982), and Boltzmann machines (Hinton and Sejnowski, 1983).

7.1.1 Linear Networks

Probably the simplest function approximator is a network which is linear in the components of x. The linear network can be written as:

$$y = w^T x + b$$

[2] If inputs which are located near to each other require very different outputs, we need to train the FA to represent steep slopes which is difficult and hinders successful generalization.

where w is the u-dimensional weight vector and the parameter b is the bias.[3] Figure 7.1(A) shows how a linear networks is used for regression tasks where we want to fit a straight u-dimensional hyperplane through a number of examples. Linear networks can only be used for learning linear functions, however, and that is why their expressive power is quite limited: e.g. for the regression problem demonstrated in Figure 7.1(B) linear networks cannot be used fruitfully.

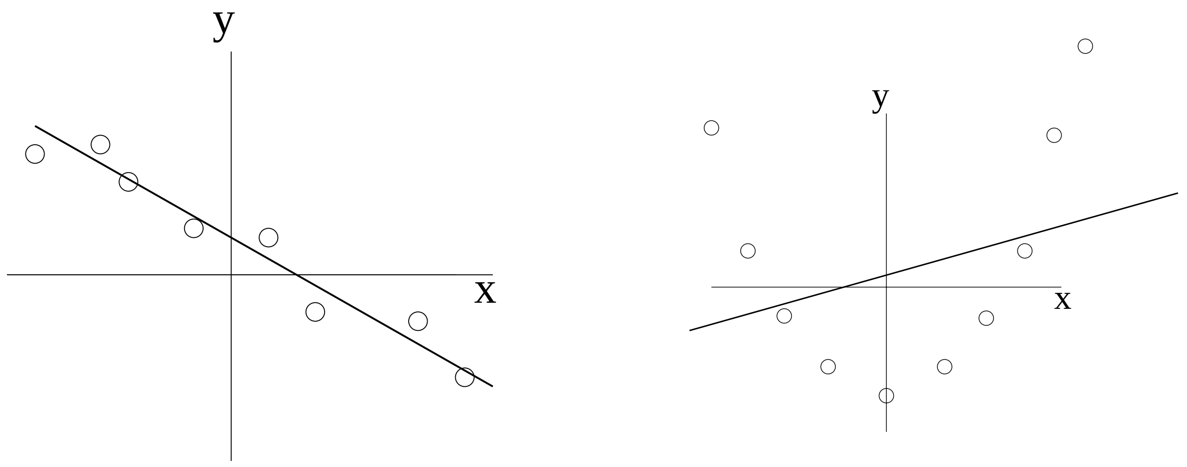


Figure 7.1: *(A) A linear network's approximation to a set of training examples (data points). (B) Training examples which cannot be represented well by a single linear hyperplane.*

Training the network. Given a set of training examples: input-output pairs (x_i, d_i), where $i \in \{1, \dots, M\}$ (M is the number of training examples), we want to minimize the mean squared error between the desired outputs d_i, and the network's outputs y_i. For online learning we derive the learning rule from the squared error function E^i:

$$E^i = \frac{1}{2}(d_i - y_i)^2 = \frac{1}{2}(d_i - w_T x_i)^2$$

Given a set of example transitions (training patterns) (x_i, d_i), we can compute w in closed form. First we collect all vectors x_t in the matrix X by putting all vectors as row vectors in the matrix. Thus, the vector X_k defines the k^{th} row of the m-by-u (m is the number of training pairs which should be larger or equal to u) matrix X. Then we compute the weight vector w as:

$$w = (X^T X)^{-1} X^T d$$

where d is the m-dimensional vector storing all desired outputs $(d_1, \dots, d_m)$, and $(X^T X)^{-1} X^T$ is the pseudo-inverse (Moore-Penrose inverse) of the matrix X (we assume that $(X^T X)^{-1}$ exists, otherwise we have to use $\lim_{a \to 0}(X^T X + aI)^{-1}$ which always has an inverse for $a > 0$. See (Rao and Mitra, 1971) for details).

We can also iteratively find a solution: after differentiating E^i with respect to w,[4] we get the delta-rule (Widrow and Hoff, 1960) which decreases the error function making updates of the weight vector with learning rate α_l:

[3]To simplify the following discussion, we put the additional parameter b in the weight vector and add an input to the inputvector which is always set to 1.

[4]We assume that the bias is part of the weight vector.

$$\Delta w = \alpha_l (d_i - y_i) x_i$$

Instead of linear networks, usually the much more powerful multi-layer neural networks are used. Such networks consist of (at least) one hidden layer with non-linear activation functions and are able to represent any Borel measurable function (Cybenko, 1989). They can be trained by using the backpropagation algorithm (Werbos, 1974; Rumelhart et al., 1986) which implements gradient descent on the error function. We refer to Kröse and v/d Smagt (1993) for a full description of these more powerful function approximators.

7.1.2 Local Function Approximators

Global function approximators such as linear or multi-layer neural networks compute the output using all adjustable parameters. Furthermore, they adapt all parameters on each example. This creates interference or forgetting problems: after we have trained the FA to learn a good approximation in some part of the input space and then train it to approximate different parts of the input space, the learning algorithm tries to use the same parameters to decrease the error over a new input distribution so that the FA "forgets" what it had learned before.

Local function approximators (LFAs) do not have this problem. They consist of many independent processing elements, which we will call processing cells or simply cells. Each cell has a centre in the input space (a vector) and an output value. Given some input, we compute cell activations based on the distance between cell centres and the input. For this, some weighting rule and a distance function is used which ensures that closeby located cells are activated most. Then, cell outputs are combined by first weighing them by the activities of the cells and then summing them. Finally, cell centres and outputs are adapted on the learning example where learning steps depend on cell activities. If we want to approximate the function in some unknown parts of the input space, most previously trained cells are not activated since the distance of their centres to the new inputs is too large. Therefore they are not adapted for these parts and the previously learned function approximation stays more or less the same.

Winner take all. We can look at such learning architectures as a competition between the cells for being active in different parts of the input space. Each cell tries to approximate the target function in some region best so that it will become the most activated cell there. LFAs can use hard decision surfaces by implementing a winner take all (WTA) algorithm. Using WTA, only the closest cell is activated [5] and this cell will determine the output of the FA. Examples of LFAs which employ the WTA strategy are 1-nearest neighbor, learning vector quantization (LVQ) (Kohonen, 1988), and rasters (grids) with fixed or variable resolution (Moore and Atkeson, 1995). When we use a nearest neighbor scheme with WTA cells to divide the input space into subregions, we say that we make a Voronoi tessellation of the input space (see Figure 7.2).

Once we have partitioned the input space, it becomes easy to learn output values. The output of each cell equals the average desired output of training examples falling inside a cell's region. However, learning a good partitioning is a hard problem, and if we change a partitioning (e.g. by adding or removing a cell), multiple output values of cells are not anymore up-to-date and need to be reestimated.

[5] In case of multiple closest neurons, we break ties arbitrarily.

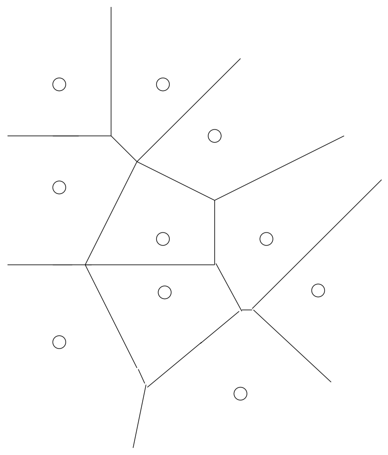


Figure 7.2: *(A) A Voronoi tesselation of the input space derived from using WTA and the locations of a set of cells (shown as circles). In between each two cells, there is a hard decision boundary. The constructed tesselation determines those regions of the input space for which a particular cell is used (is activated) for returning its output.*

Soft cell competition. Instead of WTA cells, we can also use a weighted combination of the outputs of particular closeby located cells as the final output, which results in a smoother function approximation. This is also referred to as soft competitive learning (Nowlan, 1991). Examples of smooth LFAs are k-nearest neighbor with $k > 1$, Delauney triangulizations with linear interpolations (Omohundro, 1988; Munos, 1996), locally weighted regression (Gordon, 1995; Atkeson, Moore and Schaal 1996), Kohonen networks (Kohonen, 1988), and neural gas (Fritzke, 1994).

Neural gas

We will now present a novel implementation of a smooth growing neural gas architecture. We use a set of Z neurons (cells): $\{n_1, \ldots, n_Z\}$ (initially $Z = Z_{init}$). They are placed in the input space by assigning to each a centre $w_k \in \mathbb{R}^u$ (a u-dimensional vector). Apart from the centre, each neuron n_k contains an output y_k.

We calculate the overall output of the FA by locally weighting the outputs of all neurons (although most will have a neglectable influence on the final output). We calculate the weighting factor g_k (gate) for each neuron n_k based on the cell centres and the environmental input x using:

$$g_k = \frac{e^{-\eta \; dist(w_k,x)}}{\sum_{j=1}^{Z} e^{-\eta \; dist(w_j,x)}}, \tag{7.1}$$

where $\eta \in \mathbb{R}^+$ is a user-defined constant which determines the smoothness of the FA and $dist(w, x)$ is an arbitrary distance function of the family of L_α-norms:[6]

$$L_\alpha(w,x) = (\sum_{i=1}^{u} (w(i) - x(i))^\alpha)^{\frac{1}{\alpha}}$$

[6]Weighing inputs differently for computing the distance by e.g. an additional parameter vector allows for more useful decompositions of the input space, although it also requires more parameters to be learned.

where $w(i)$ and $x(i)$ refer to the value of the i^{th} feature of the centervector and input, respectively. The overall output of the FA given input x is:

$$y = \sum_{j=1}^{Z} g_j y_j$$

Learning Rules. We want to maximize the probability of returning the correct output value. Since the output value depends on the locations of the neurons and on the output values of the neurons, we have to: (1) learn the network structure: move the neurons to locations where they help to minimize the overall error, and (2) learn correct output values: make individual neurons correctly evaluate the inputs for which they are used.

In order to let the FA converge, we will anneal the learning rate of neurons so that they will stabilize over time. For this, we keep track of the overall usage of a particular neuron in a responsibility variable C_k which is initially set to 0. After each example, each neuron n_k's responsibility variable C_k is adapted:

$$C_k \leftarrow C_k + g_k$$

Then, C_k is used to determine the size of n_k's learning rate (and also to determine conditions for adding a neuron).

(1) Learning Structure. The structure of the neurons is of great importance, since it determines which states are aggregated together and thus which family of functions can be represented. We should keep the following principles in mind: (a) neurons should be placed where they minimize the overall error, and (b) we want to have most neurons (a finer resolution) in regions with the largest fractions of the overall error.

There are a number of different ways for learning the structure of neural gas architectures, all based on using particular operations on the neurons. The most common operations are moving, adding and deleting (or merging) neurons.

Fritzke (1994) presents a particular method for growing cell structures. His method learns a topology by creating edges which link two neurons which together have been closest to a particular example. His learning rule moves the closest neuron and its neighborhood to the input. Furthermore, his method incrementally adds neurons, for which it keeps track of the total error of neurons when they have won the WTA competition. After a fixed number of example presentations a neuron is added near the neuron with the largest accumulated error. This neuron is initialized by interpolating from the nearest neurons and is assumed to be able to learn to decrease the error in that specific region of the input space.

Instead of using Fritzke's method, we have created a different method which does not accumulate error over time, but adds neuron's when the error on a single particular example is too large. The details are as follows: If the error $|d - y|$ of the system is larger than an error-threshold T_E, the number of neurons is less than Z_{max}, and the closest neuron's responsibility C_k exceeds the responsibility threshold T_C (C_k grows with the density of the input distribution around neuron n_k), then we add a new neuron n_{Z+1}. We set its location w_{Z+1} to x, and set the output value $y_{Z+1} \leftarrow d$. Finally we set $Z \leftarrow Z + 1$. Thus if the error on an example is large, we immediately copy the example to a novel neuron. The method is a good method to immediately decrease the error of difficult regions in the input space, but it can also suffer from noisy examples, which causes newly added neurons to approximate regions by a wrong output value (although they could still adjust themselves later on).

If no neuron is added, we calculate for each neuron n_k ($\forall k \in \{1, \ldots, Z\}$) a gate-value h_k, which reflects the posterior belief that neuron n_k evaluates the input best:

$$h_k = \frac{g_k e^{-(d-y_k)^2}}{\sum_{j=1}^{Z} g_j e^{-(d-y_j)^2}}$$

We then move each neuron n_k towards the example x according to:

$$w_k \leftarrow w_k + \alpha_k h_k^2 (x - w_k),$$

where $\alpha_k = \alpha_g (C_k)^{-\beta}$, α_g is the system learning rate and β is the learning rate decay factor. The effect of using the square of h_k instead of h_k is that neurons which are not a clear winner will not adapt themselves very fast. Thus, clear separations are made early on in the process, whereas finer divisions of the input space emerge only gradually.

(2) Learning Q-values. We update the output y_k of neuron n_k ($\forall k \in \{1, \ldots, Z\}$) by:

$$y_k \leftarrow y_k + \alpha_k h_k (d - y_k)$$

Note the similarities of our neural gas method with the hierarchical mixtures of experts (HME) architectures (Jacobs et al., 1991; Jordan and Jacobs, 1992).

Nearest neighbor search. An important issue in local FAs is to search for the set of neurons which are activated for returning the output. Even LFA's with smooth competition usually have only a small number of neurons which have non neglectable activations for returning the output.

The same problem, called the data-retrieval problem, is known in database systems. Given the input, finding the closest neuron(s)/data-item is usually implemented by computing the distance to all neurons and then selecting the closest one(s). The complexity of this is O(Zu), where Z is the number of neurons, and u the dimensionality of the input space. If we have thousands of neurons, the input dimensionality is large, and we have to return outputs for a large number of queries (inputs), then the system gets very slow. A solution to speed up the search is to use preprocessing and to store the neurons in a datastructure which allows for faster online search.

We can use datastructures and methods from computational geometry such as Voronoi diagrams (Okabe et al., 1990). These methods significantly speed up searching for the closest neuron, but the problem is that the size of such structures is exponential in the dimensionality of the input space.

K-d trees (Friedman et al., 1977; Preparate and Shamos, 1985; Moore, 1991) are another datastructure for speeding up nearest-neighbor search. To create a K-d tree, we recursively partition the population of neurons by dividing the individuals up along a single input dimension, which is chosen to keep the tree more or less balanced. The nearest neighbor is then found by descending the tree to find an initially best candidate, and then backtracking along the tree to examine whether all alternative siblings are not closer while discarding siblings which are surely not closer to the input than the current best one. K-d trees can be used to perform searches in expected time close to $min(a^u logZ, uZ)$, where a is a suitable constant > 1 (Moore, personal communication 1998). This means that searchtime is logarithmic in Z, but exponential in the input dimensionality. In practice the time requirements depends a lot on the distribution of the data (Omohundro, 1989; Moore, 1991). Constructing K-d trees takes $\theta(uZlogZ)$ preprocessing time (Preparate and Shamos, 1985).

A problem of the above methods is that they cannot be easily combined with neural gas; neurons are moved around and added to the system so that the data structures used for searching should change as well in an incremental way. This makes using these structures more expensive. Omohundro (1989) describes and experimentally evaluates some incremental methods for constructing balltrees, data structures very similar to K-d trees.

A quite similar idea, especially suitable for radial basis functions is to use a hierarchy of neurons, where the higher level neurons are used for clustering lower level neurons. Then we can descend from the top layer to lower layers and use leaf neurons for the final output. Although the primitive method is not guaranteed to return the closest neuron, [7] we could decide to use this neuron anyway or to use backtracking, e.g., used in bumptrees (Omohundro, 1988; Landelius, 1997), afterwards. Such methods can sometimes save a lot of time, see experimental results with bumptrees in (Omohundro, 1988; Landelius, 1997).

Another possibility which is an O(1) method is to restrict the search among a fixed set of neighbors which are allowed to be activated. E.g. we can construct (a limited number of) edges to link neurons which are activated after each other, or we can constrain the search by using an *a priori* temporal structure. Although this method may work well, it may be the case that many more neurons may be needed since different neurons may be implementing the same function for the same region (only the sequence of examples arriving at the region was different).

In our experiments we have used the simplest line search method, although it resulted sometimes in computationally very demanding simulations.

7.1.3 CMACs

The third kind of FAs are also local, but are different from the ones described previously, since they do not construct a tesselation based on the positions of a number of cells. They use a fixed *a priori* decomposition of the input space into subregions. Usually the subregions are structured in an ordered topology such as hypercubes which makes searching for active cells much faster compared to the methods described earlier, since random access (instant lookup) becomes possible. This increased time efficiency is (as usual) at the expense of a huge increase of the storage space, however. A well known FA which uses (for example) hypercubes to divide the input space into subregions is the CMAC (Albus, 1975a). A CMAC (Cerebellar Model Articulation Controller) uses *filters* mapping inputs to a set of activated cells. Each filter partitions the input space into subsections in a prewired way such that each (possibly multi-dimensional) subsection is represented by exactly one discrete cell of the filter. For example, a filter might consist of a finite number of cells representing an infinite set of colors represented by cubes with 3 dimensions red, blue and yellow, and activate the cell which encloses the current color input.

CMACs uses multiple filters consisting of a number of cells with associated output values. Applying the filters yields a set of activated cells (a discrete distributed representation of the input), and their output values are averaged to compute the overall output.

The difference with vector quantization methods is that the partitioning is designed *a priori* and that multiple cells are activated by using different views on the world.

General remarks on filter design. In principle the filters may yield arbitrary divisions of the state-space, such as hypercubes, but also more complex partitionings which make use of

[7]Higher level neurons are centered at the distribution of examples for which they are activated, but nothing is said about the positions of their lower level neurons.

preprocessing may be used. To avoid the curse of dimensionality which occurs when one uses all input-dimensions to construct hypercubes, one may use hashing to group a random set of inputs into an equivalence class, or use hyperslices omitting certain dimensions in particular filters (Sutton, 1996). Although hashing techniques may help to overcome storage problems, it collapses states together which can be widely apart. We expect that this will often not lead to good generalization performance. Therefore, we prefer hyperslices which group inputs by using *subsets* of all input dimensions. E.g. if we say that we see a green object which is larger than 5 meters, we could already infer that the object might be a tree, although there may be other options as well such as a green house or a mountain. If we add multiple descriptions (filters) of the object together, and all descriptions agree with the object, we may be certain which object we are examining.

Computing the output. The first step in computing the output is applying all filters to the input vector. Each filter uses particular input dimensions to create its universe, and partitions that universe using hyperboxes. Given an input, it projects the input vector in its universe and examines which of its cells encloses the input.

More formally: a filter is represented as a multi-dimensional array. Given an input x, the filter uses the relevant features to search for the cell (out of $\{c_1, \ldots, c_{n_c}\}$, where n_c is the number of cells) in which the input can lie. Applying all filters returns the active cells $\{f_1, \ldots, f_z\}$, where z is the number of filters.

The output value y given input x is calculated by averaging the outputs of all activated cells $y_k(f_k)$:

$$y = \sum_{k=1}^{z} y_k(f_k)/z,$$

where $y_k(f_k)$ is the output of the activated cell f_k of filter k.

Learning rule. Training CMACs is an extremely simple procedure: we just adapt the output values of all activated cells. Thus, for all activated cells we compute:

$$y_k(f_k) \leftarrow y_k(f_k) + \frac{\alpha}{z}(d - y_k(f_k)),$$

where α is the learning rate.

Filter design. The design of the filters is very important, since the construction of the filters determines what can be learned, and how fast that can be learned (more cells need more examples). There are two important matters in designing filters: (1) Which features to combine inside each filter. This introduces bias on which relationships can be learned. (2) How coarsely to divide each filter into cells. This determines the generalization behavior and the attainable accuracy of a filter. The possible choice of (1) and (2) is restricted by space limitations, since combining D input features each divided into n_c parts, would mean that our multi-dimensional cube consists of n_c^D cells.[8] Note that there is no necessity to have a fixed uniform decomposition. We could also use adaptive variable-resolution filters to optimize the decomposition of a filter into cells, or activate a variable number of filters for different inputs, which makes richer descriptions possible for objects which require more specification.

Example of effective filter design. Sometimes a particular filter design for CMACs can be very effective. E.g. consider Figure 7.3(a) which shows an empty maze with a starting state and a goal state. We can construct two filters for this maze, one slicing the maze up

[8]This is the worst case for which all cells can be made active, otherwise if some cells are never activated an implementation based on hashing tables could significantly decrease space requirements, while keeping the search for active cells efficient.

in vertical slices (Figure 7.3(b)) and one for slicing the maze up into horizontal slices (Figure 7.3(c)).

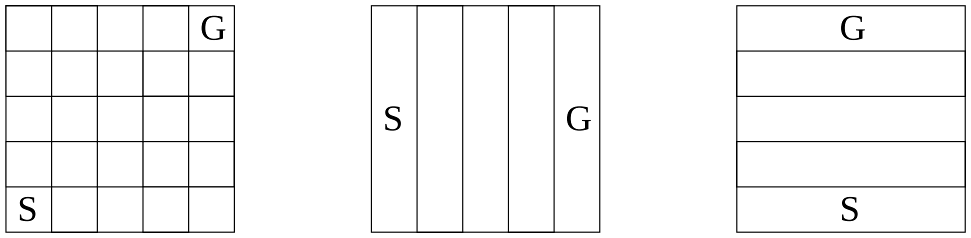


Figure 7.3: *(A) A maze with starting state S and goal state G. (B) One filter slices the maze up into vertical slices (the vertical position does not matter). (C) The second filter slices the maze up into horizontal slices. Together the filters can be used for efficiently learning a solution.*

These filters do not allow for accurately representing the optimal value function, but the approximated value function can be used for computing the optimal policy — horizontal filters (except for the last) will learn that *going north* is the best action, and vertical filters (except for the last) learn that *going east* is the best action. Good filter design also means that we have to learn less: for the maze in Figure 7.3(a) and the usual 4 actions, we would only have to learn 10×4 instead of 24×4 values.

Overgeneralization. If the maze would have blocked states, things get more complicated, since the filters would overgeneralize and filter away the exceptions. If there are few blocked states, we can still hope but not guarantee that the dynamics of the learning process will converge to a path which circumvents the blocked states. A more sophisticated approach to deal with few exceptions is to construct additional filters which are made active in the neighborhood of these exceptions and which add something to or override the values of the default filters. With many blocked states, we would store almost all exceptions, which would mean that we would in principle combine both features in a filter and this would of course be equivalent to the lookup-table representation.

Generalization by sharing features. CMACs also allow for another kind of generalization by using multiple filters sharing the same feature space. In Figure 7.4 we show how multiple filters on the same input creates a smoother generalization. Using more than the shown 2 filters would make this even more pronounced. Using many coarse filters leads to a dynamical refinement of the function — each filter adds something while still being very general.

7.2 Function Approximation for Direct RL

We have introduced a number of different FAs which can be used for approximating value functions. In this section we describe how they can be combined with direct RL methods and we analyse possible problems arising from such combinations. Even if we are well acquainted with a particular FA, we have to keep in mind that training a FA with RL instead of with supervised learning puts different demands on the FA. For instance, in RL examples are generated according to the policy and value function, which themselves are changing. Therefore

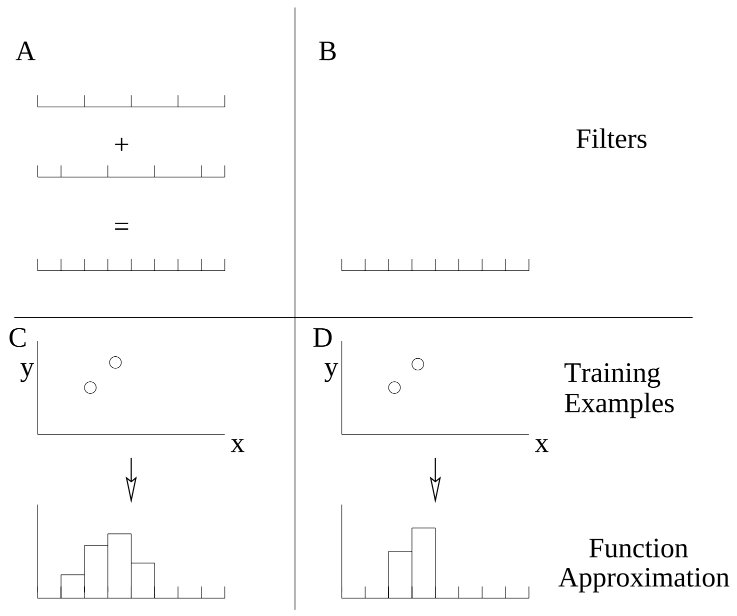


Figure 7.4: *(A) A design of two filters which splice a single input dimension in parts. When added, they allow for a partitioning into 8 different regions. (B) A single filter which partitions the input into 8 regions. (C) We examine what happens if we receive training examples (the circles) — the vertical distance to the line is the desired output. We can see that the combination of the two filters from (A) leads to some kind of generalization behavior. (D) With a single filter, there is no generalization across cells. The first filter design results in a smoother function due to its hierarchical approach — some example's output is dynamically refined if we add multiple filters.*

the function we try to learn is constantly changing, which makes the task very hard. On the other hand, in supervised learning we are usually interested in the most accurate approximation, whereas in RL we are much more concerned with the performance of the resulting policy. As we have seen in Chapter 4, we can have good policies with quite poor value function approximations. What is important, however, is that the preferential order between different actions is reflected in the value function.

Representational issues. We are interested in storing the Q-function with FAs. For this, we need to compute outputs $Q(x, a)$ for all actions $a \in A$. We will do this by using different functions for different actions $F_a(x)$:

$$Q(x, a) = F_a(x)$$

A different way would be to put the actions in the input, but this makes learning different mappings for different actions more difficult. For continuous actions, however, we cannot use one FA for one action and thus we would need to use different methods, dependent on the used FA. E.g. for linear networks we could use the action(s) as part of the input and for CMACs we could discretize the action space. We assume in the following that we have a finite set of actions.

Thus, with linear networks we will use $|A|$ networks, one for each action. For a multi-layer feedforward neural network, we can have different networks for each action or we can share the hidden units of one network to approximate all Q-values. Lin (1993) compared both possibilities and found that one network for one action worked better. For other tasks sharing the hidden layer may be worthwhile, however, since the hidden units need to settle down to extracting more characteristic features when they are used for describing the complete problem domain (shared by all actions) and that may improve generalization performance. E.g. Caruana (1996) added outputs for a prediction problem which puts an additional burden on the learning task, but lead to an improvement of the learned approximation for the requested output. Baxter (1995) presents a general theory of bias learning based upon the idea of learning multiple related tasks with a shared representation. Generally speaking, sharing parameters for learning multiple actions is fruitful if the tasks share mutual information.

The neural gas and CMAC architectures use for each neuron and filter/cell Q-functions of the form $Q(n_i, a_j)$ and $Q_f(c_i, a_j)$, respectively.

7.2.1 Extending Q(λ) to function approximators

We can combine TD(λ), Q-learning, and all Q(λ) methods with function approximators. One should be careful choosing a combination and learning parameters, however, since the resulting learning system may not always be stable. E.g. if we allow a FA to adapt itself quickly to maximize learning speed, it can happen that a specific "lucky" trajectory leads to a large increase of values of states visited by the current trajectory. This may change the overall value function dramatically and can lead to undesirable large changes of the policy. Although a large change of the approximation of a FA for supervised learning can also happen, we can still relearn a previous good approximation since the training set is stored. Since in RL the policy generates the examples, we may have much larger problems relearning "good" value functions. Furthermore, since in RL the Q-function changes and the learning examples are drawn from a noisy source with often a sequence of (highly) correlated examples, there can be many unreliable updates or drift of the value function. Thus, our FAs should be robust enough to handle such problems.

Offline Q(λ). To learn Q-values we monitor agent experiences during a trial in a history list with maximum size H_{max}. At trial end, the history list H is:

$$H = \{\{x_{t^1}, a_{t^1}, r_{t^1}, V(x_{t^1})\}, \ldots, \{x_{t^*}, a_{t^*}, r_{t^*}, V(x_{t^*})\}\}$$

Here x_t is the input seen at time t, a_t is the action selected at time t, r_t is the reward received at time t, $V(x_t) = Max_a\{Q(x_t, a)\}$, t^* denotes the end of the trial, and t^1 denotes the start of the history list: $t^1 \leftarrow 1$, if $t^* < H_{max}$, and $t^1 \leftarrow t^* - H_{max} + 1$ otherwise.

After each trial we calculate examples using offline Q(λ)-learning. For the history list H, we compute desired Q-values $Q^{new}(t)$ for selecting action a_t, given x_t ($t = t^1, \ldots, t^*$) as follows:

$$Q^{new}(t^*) \leftarrow r_{t^*}$$
$$Q^{new}(t) \leftarrow r_t + \gamma \cdot [\lambda \cdot Q^{new}(t+1) + (1-\lambda) \cdot V(x_{t+1})]$$

Once the agent has created a set of offline Q(λ) training examples, we train the FAs to minimize the Q(λ)-errors.

Offline multi-agent Q(λ). In Chapter 3, we have already discussed using multiple autonomous agents simultaneously and presented two algorithms for using online Q(λ) to learn a shared value function. We can also combine offline Q(λ) with multiple agents by using a different history-list for each different agent. In the experiments later in this chapter, we will use these history-lists independently. After having created multiple lists of training examples, we process the history-lists by dovetailing as follows: we train the FAs starting with the first history list entry of agent 1, then we take the first entry of agent 2, etc. Once all fist entries have been processed we start processing the second entries etc.

Online Q(λ) for function approximators. To combine online Q(λ) with FAs, we use eligibility traces. The eligibility trace $l_t(w_k, a)$ is used for weight (adjustable parameter) w_k and action a as follows:

$$l_t(w_k, a) = \sum_{i=1}^{t-1} (\gamma\lambda)^{t-i} \zeta^i(w_k, a),$$

where $\zeta^i(w_k, a)$ is the gradient of $Q(x_i, a)$ with respect to the weight w_k:

$$\zeta^i(w_k, a) = \frac{\partial Q(x_i, a)}{\partial w_k}$$

and the online update at time t becomes:

$$\forall w_k \;\; do: \;\; w_k \leftarrow w_k + \alpha[e'_t \zeta^t(w_k, a) + e_t l_t(w_k, a)]$$

where as usual $e'_t = r_t + \gamma V(x_{t+1}) - Q(x_t, a_t)$ and $e_t = r_t + \gamma V(x_{t+1}) - V(x_t)$.

Fast Q(λ) can also improve matters in case of LFAs. Suppose a LFA consists of $|S|$ possible state space "elements" (e.g. neurons or total filter cells). The Q-values of $z \leq |S|$ elements are combined to evaluate an input (query). Here the update complexity of fast Q(λ) equals $O(z|A|)$. This results in a speed-up of $\frac{|S|}{z}$ in comparison to conventional online Q(λ). For global approximators $z = |S|$, so the method is not helpful. For quantized state spaces with fine resolution, the gain can be quite large, however.

Online Q(λ)-learning for CMACs. We will show in detail how we combine fast Q(λ) with a LFA, in this case CMACs. Using a fixed partitioning, a continuous input x_t is transformed into a set of active features: $\{f_1^t, f_2^t, \ldots, f_z^t\}$, where z is the number of active features (tilings).

Before using $Q_k(f_k^t, a)$ we call the *Local Update* procedure for all features $(f_1^t, \ldots, f_z^t)$ so that their Q-values are up-to-date. *Local Update* stays almost the same (except that we use different variable names).

The *Global Update* procedure now looks as follows:

```
CMAC Global Update($x_t, a_t, r_t, x_{t+1}$) :
1) For $i$ is 1 to $z$, $\forall a \in A$ Do
       1a) $Local\ Update(f_i^{t+1}, a)$
2) $V(x_{t+1}) \leftarrow \max_a \sum_{i=1}^{z} Q_i(f_i^{t+1}, a)/z$
3) $e'_t \leftarrow (r_t + \gamma V(x_{t+1}) - Q(x_t, a_t))$
4) $e_t \leftarrow (r_t + \gamma V(x_{t+1}) - V(x_t))$
5) $\phi^t \leftarrow \gamma\lambda\phi^{t-1}$
6) $\Delta \leftarrow \Delta + e_t\phi^t$
7) For $i$ is 1 to $z$ Do
       7a) $Local\ Update(f_i^t, a_t)$
       7b) $Q_i(f_i^t, a_t) \leftarrow Q_i(f_i^t, a_t) + \alpha(f_i^t, a_t)e'_t$
       7c) $l'(f_i^t, a_t) \leftarrow l'(f_i^t, a_t) + \frac{1}{\phi^t z}$
```

Remarks about online multi-agent Q(λ). To use online Q(λ) with multiple agents, we could use the connecting traces approach from Chapter 3. We have not done so in the experiments in this chapter, however. We have used different eligibility traces for different agents, however, since that is a necessity for multi-agent learning.

7.2.2 Notes on combining RL with FAs

In the following we will shortly describe the representational and convergence issues for using our different FAs for RL.

Linear networks. We know that if the number of input features is small compared to the total number of (distinct) states, we strongly limit the kind of value functions which can be represented accurately by linear networks. However, we can use many inputs and higher order inputs as well, which makes linear networks more powerful. E.g. in the limit we use a weight for each state, and we have a tabular representation with known convergence results.

The networks try to learn a linear approximation of the Q-functions over the entire input space minimizing the squared error over all training examples. For many problems, there may be small differences between the Q-values of different actions. E.g. postponing the optimal action for one time step may only cause the expected discounted future reward (Q-value) to decay by the discount factor (if nothing changes, we may still select the optimal action at the next time step). That's why the learning dynamics may cause large policy changes which makes policy evaluation hard. Thus, we should be extremely careful setting the learning rate for linear nets.

Neural gas. The approximation of a neural gas representation may be very accurate, especially with a large number of neurons. Since RL becomes much slower when we use many neurons (retrieval time increases linearly and learning time even worse with the number of neurons), there exists a tradeoff between accuracy and learning time. Therefore we have to be careful choosing the number of neurons or alternatively the growthrate of a neural gas structure. Note that just a few neurons can already be sufficient for learning a good policy and make the learning problem a lot easier.

For the neural gas, we have to be careful changing the structure. If neurons move fast around, this may cause large policy changes which makes learning a (more or less fixed) value function very hard. If we add a neuron, we initialize the Q-value of the selected action according to the desired Q-value (computed by e.g. offline Q(λ) shown above). The Q-values of the other actions of the new neuron are computed by weighing the Q-values of existing neurons according to their posterior probabilities h_k. One difficulty is that the decision to add neurons is based on TD-errors, which themselves may be very noisy. Since it is difficult to decide whether the example is noisy or the FA is unable to learn a correct mapping, we just add neurons and hope they will shape themselves inside the structure by learning from future examples.

CMACs. Using CMACs with direct RL may work quite robustly, since it always maps the same state to the same set of activated cells. Furthermore, CMACs may approximate the value function arbitrarily well given a large number of filters and cells and is shown to converge to the least squared error approximation on a set of training data given proper learning rate annealing (Lin and Chiang, 1997). In practice a lot of CMAC's functionality depends on the grouping of states. If a set of states with highly different values is grouped together by a cell, then the learned mean value of these states may not be very useful and the cell's updates will have large variance. If each cell groups states together with similar

Q-values, then learning may be quite fast.

For RL this means that future trajectories of states grouped by a cell should be as similar as possible. In general this will not be the case, however. Furthermore cells may be biased or repelled from particular actions. This can cause problems for learning the policy. E.g., some actions which are only good in few states may never get selected. Such problems can be overcome by making the filters sufficiently specialized, however.

Convergence issues. Tsitsiklis and van Roy (1996) proof convergence of TD(λ) methods for function approximators using linear combinations of fixed basis functions (note that the linear networks and CMACs belong to this class, neural gas do not since the neurons are not fixed). In their proof, they stress the importance of sampling from the steady-state distribution of the Markov chains. This can be done by generating trajectories using the simulator/real environment. Learning by making single simulation steps starting in a fixed set of preselected input points may cause divergence (Boyan and Moore, 1995), however. Finally, Tsitsiklis and van Roy (1996) suggest that higher values of λ are likely to produce more accurate approximations of the optimal value function. This may imply as well that TD(λ) methods travel along lower mean squared error trajectories in parameter space for larger values of λ. Unfortunately, little understanding of the convergent behavior of changing policies as that of using Q(λ) with function approximators is available, however.

7.3 World Modeling with Function Approximators

Combining dynamic programming with function approximators was already discussed by Bellman (1961) who proposed using quantization and low-order polynomial interpolation for approximately representing a solution (Gordon, 1995a).

Transition modeling. We can use a function approximator to represent not only the value function, but also the world model. For this, it is best to use local function approximators. Global function approximators may be very useful for modeling deterministic processes, but have problems to model a variable number of outgoing transitions due to their rigid structure. Examples of using global FAs for world modeling are Lin's world modeling approaches (1993) for modeling a complex survival environment and explanation based neural network learning for learning a model of chess (Thrun, 1995). Although using these models helped speeding up learning a good policy, they cannot be easily combined with DP algorithms.

On the other hand, if we partition the space into a finite set of regions with local function approximators, we allow for a much higher variety of transitions and estimating transition probabilities becomes more precise and easier. Furthermore, when we use discrete states, we have the advantage of being able to apply DP-like algorithms to compute the value function.

Estimating transitions. The largest problem which has to solved before DP can be used fruitfully in combination with LFAs is that we need a good partitioning of the input space so that estimated transition and reward functions reflect the underlying model well. Given the positions of the cells (or neurons), we can use sampling techniques to estimate the transition function. We may use Monte Carlo simulations to sample a state, map this to an active cell, select an action, use the simulator or environment to make a transition to the successor state and map this back to the next active cell. By counting how often each transition occurs, we can estimate the transition probabilities between cells and similarly we can compute the reward function (see Chapter 4).

When we have a coarse partitioning, there may be many transitions from a cell which stay inside that cell. Those recurrent transitions make computing a correct value function harder. Only rewarding transitions or transitions going to neighboring cells are useful for computing the value function and policy, since they show differences between different actions. The finer the representation, the better we may model the true underlying dynamics, but the larger the problem of noise. Furthermore, with a fine representation we need much more experiences before we have learned reliable transition probabilities and rewards, and DP becomes much slower. Finally, if the partitioning is changing, we have to recompute transition probabilities and rewards, which may not always be easy.

Note that a model gives us the advantage of different splitting criteria: (1) If a (hypothetical) split will change the value function significantly, we keep it, and continue splitting; (2) If a current transition is unexpected according to the model, we can split the cell to be better able to predict the dynamics. Moore and Atkeson's partigame algorithm (1995) does exactly the latter: the algorithm greedily executes a goal directed behavior, but whenever executed actions in a cell do not make a transition to the next cell as expected, the algorithm splits the cell so that more accurate predictions become possible. A difficulty of this approach is that stochasticity in the transition function makes it difficult to know whether splitting the cell will make the ability to predict the successor more accurate or just increase the model's complexity. A solution to this may be to use hypothetical splits and collect more experimental data. Another problem is that single splits may not always be helpful to see any differences, so that multiple splits (e.g. in multiple dimensions) have to be tried out at the same time which is computationally quite expensive.

Sometimes, learning a model is not very difficult — if we have a deterministic MDP, discrete actions and discrete time, we could learn a perfect deterministic transition function between cells by making the coarseness of the representation sufficiently small, although for high-dimensional spaces the required space complexity may make this approach infeasible.

Learning imperfect models. Although learning an accurate model is hard, we can learn useful, but incomplete models. Such models can be used for more reliable and faster learning of good policies. E.g. consider a soccer environment in which we do not try to predict the positions of all players and the ball (which would be infeasible due to the huge amount of possible game constellations), but only try to estimate the position of the ball at each following time step. Such a model is much easier to learn and even although it may not be very accurately, it could be quite useful for planning action sequences.

First we will discuss learning linear neural network models and we will see that this leads to a very simple equation for estimating the value function. Then, we will discuss estimating models for neural gas and CMAC architectures after which DP or PS is used to compute value functions.

7.3.1 Linear Models

Linear models are the simplest models, but they can only be used to accurately model a deterministic evolution of the process. The goal is to estimate the linear model L [9] (a square matrix) which maps the input vector at time t, x_t to its successor x_{t+1}:

$$x_{t+1}^T = x_t^T L + \eta$$

[9] To simplify the following discussion, we leave action parameters out of the nomenclature.

where x^T denotes the transpose of x and η is a vector which accounts for white noise. We also want to model the reward function. For this, we use a linear model w_r to compute the expected reward r_t given that we see x_t:

$$r_t = x_t^T w_r$$

Note that we use a linear reward function which makes the following short introduction to linear models somewhat simpler, but for practical use it has some disadvantages. We should realize that for infinite state spaces, the agent can continuously increase its received rewards by making actions which follow the direction of the linear reward plane. Therefore optimal performance can never be reached. We can only use this linear reward function if the state space is bounded by the transition function. Many other researchers have used a quadratic reward function, written as $(r_t = x_t w_r x_t)$, which ensures that all positive rewards are bounded.

Computing the models. Given a set of example transitions (training patterns) (x_t, x_{t+1}), and emitted rewards (x_t, r_t), we can compute L and w_r. First we collect all vectors x_t in the matrix X and the vectors x_{t+1} in the matrix Y by putting the vectors as row vectors in the matrices. Then, we can compute L as follows:

$$L = (X^T X)^{-1} X^T Y \tag{7.2}$$

Note that if the vectors $(X_1, \ldots, X_m)$ are orthonormal,[10] then $(X^T X)^{-1}$ is identical to the identity matrix I, so that we can simplify Equation 7.2 to:

$$L = X^T Y$$

We compute the weight vector w_r for returning the immediate reward as:

$$w_r = (X^T X)^{-1} X^T r$$

where r is the m-dimensional vector which stores all immediately emitted rewards $(r_1, \ldots, r_m)$.

Computing the value function. We approximate the value of a state by using the linear network (see above):

$$V(x) = x^T w$$

where the vector w is the weight vector and for simplicity includes the bias parameter (an additional on-bit is included in the input vector).

Now we can rewrite the original DP Equation 2.2 from Chapter 2:

$$V = \gamma P V + Diag'(PR^T)$$

as

$$w = \gamma L w + w_r$$

for which we require that all absolute eigenvalues of L are smaller or equal to 1 (otherwise the value function is non-existent). This we can rewrite as:

$$w = (I - \gamma L)^{-1} w_r$$

[10]In practice this is not very likely to happen, although it could happen if, for example, we use input vectors with a single bit on to model different discrete states.

Which gives us the solution for a linear model.

Bradtke and Barto (1996) present an extensive analysis of linear least squared methods using a quadratic reward function combined with TD learning. They proof that if the input vectors are linearly independent and each input vector is of dimension equal to the number of states, the weights converge with probability 1 to the true value function weights after observing infinitely many transitions (examples).

These linear models are also called optimal linear associative memory (OLAM) filters (Kohonen, 1988). There also exist more complicated linear models such as linear quadratic regulation (LQR) models which extends OLAM by allowing the modeling of interactions between inputs (Landelius, 1997).

7.3.2 Neural Gas Models

In contrast to global FAs, local models can be simply combined with world models and can in principle model the underlying MDP arbitrary accurately — we can make a (fine-grained) partitioning of the input space into cells and compute transition probabilities by counting how often one cell has been activated after another.

Distributed/Factorial representation. Counting is easiest if we use WTA to create a set of discrete regions, since after each transition we only have to change three variables: the counters of the transitions and the variable which sums the transition rewards. If we would use smooth LFAs, we need to update all transitions between cells which were activated. This takes much longer: in the worst case of a full distributed probabilistic representation this takes $O(Z^2)$, where Z is the number of neurons. In this case we could just connect cells which are more activated than some threshold to speed things up.

Since WTA causes discontinuities at the borders of two subspaces, there may be advantages in smoothly combining the values of a set of states (Moore et al., 1997; Munos, 1996). Schneider (1997) uses multiple states to model uncertainty in model-based learning, and shows that dealing with uncertainty can be improved by interpolating over a larger neighborhood.

Gordon (1995) shows conditions under which the (approximate) value iteration algorithm converges when combined with function approximators which compute weighted averages of stored target values. An interesting insight which he offers is the following: if a sample point uses a cell with some (normalized) weight, then this weight can be seen as the probability of making a step to this cell from the sample point. This may be a good way of initializing transition probabilities after inserting a new cell.

Again we have to learn a neural gas structure and the Q-function. The latter is done by estimating the model and using our prioritized sweeping algorithm to manage the updates.

Learning structure. We used the same approach as before to learn the structure. Again, after each game we computed new Q-values according to $Q(\lambda)$, but now we used these values only to split states and not for computing a new Q-function. We also used the same conditions as before for adding neurons. Furthermore we initialize the transition variables to and from the new neuron to 0. Note that we may improve this initialization procedure by weighing the transitions from neighboring neurons according to their distances. Accurately initializing the transition probabilities and rewards is difficult, however, since we never know the exact dynamics at a new (unmodeled) region. Another way to learn the structure is to use hypothetical splits and model the dynamics at the new hypothetical cells without immediately using them for evaluating actions. Then we can estimate whether using them for real will significantly change the transition probabilities and the value function. If they would, we keep them, otherwise we try new splits. We have not tried this algorithm, however, although it has the advantage that newly introduced cells may be guaranteed to improve the model and we have good initial transition probabilities for them.

Estimating the model/Learning Q. To estimate the transition model for the i^{th} neuron, we count the transitions from the active neuron $n^t = i$ to the active neuron $n^{t+1} = j$ at the next time-step, given the selected action a. Thus, after each (i, a, j) transition we compute:

$$C_{ij}(a) \leftarrow C_{ij}(a) + 1 \quad \text{and} \quad C_i(a) \leftarrow C_i(a) + 1$$

We can also take all activated neurons (in the smooth competition case) into account by updating counters dependent on their gates. We do this by computing $\forall i, \forall j$:

$$C_{ij}(a) \leftarrow C_{ij}(a) + g_i^t g_j^{t+1} \quad \text{and} \quad C_i(a) \leftarrow C_i(a) + g_i^t$$

These counters are used (see Chapter 4) to estimate the transition probabilities $P(j|i,a) = P(n^{t+1} = j | n^t = i, a)$. For each transition we also compute in similar ways the average reward $R(i, a, j)$ by summing the immediate reinforcements over the transition from the active neuron i to the next active neuron j by selecting action a.

Our prioritized sweeping. We apply our prioritized sweeping to update the Q-values of the neuron/action pairs. After each time step, updates are made via the usual Bellman backup (Bellman, 1961):

$$Q(i,a) \leftarrow \sum_j P(j|i,a)(\gamma V(j) + R(i,a,j))$$

After each action we update the model and use PS to compute the new Q-function. Details of the algorithm are given in Appendix D.

7.3.3 CMAC Models

We now describe a world modeling approach for RL which uses CMACs. We know that learning accurate models estimating transitions between complete world states is very hard for complex tasks. E.g. the same world state may only appears once in an agent's life time. As an example consider a world state consisting of a number of input features for which the first feature would oscillate between two values: $x_1 = 1, 2, 1, 2, 1, \ldots$. A second feature could oscillate between 3 values, a third between 5 values and so on. If we would use one filter for each feature, we could easily learn the correct predictive model (in e.g. 2 to 13 steps). Estimating the transition function of the complete state vector with a single filter would take a lot of time (e.g. $2 * 3 * 5 * 7 * 11 * 13 = 30030$ steps), however, since a complete period in input space would cost so many steps.

Therefore, instead of estimating full transitions models, we can sometimes profit significantly from using a set of independent models and estimating their uncoupled internal dynamics. In this way, we can study the dynamics of an object by separating it from the rest of the world. If the separation holds, we can learn perfect models.

Our goals are to (1) create a set of filters which can be used for modeling the world, and (2) estimate parameters given the set of designed filters.

Creating filters. We have seen that with a completely observable discrete world and lookup tables, we can estimate a model and perfectly store the transition probabilities to successor states. This allows to compute answers to questions such as: "what is the expected probability of entering the goal state within N steps?" Furthermore, it is for static goal-directed environments always possible to select an action which is expected to increase the state value.

For CMAC models, we can in general not answer the question above, and it may not always be possible to select actions which lead to a larger value for each filter. If filters contradict each other, e.g., if one filter advises a particular action which is expected to decrease the value of another filter, learning problems may arise. Usually this happens in the initial phase, but such problems may be resolved due to learning, however.

Suppose actions are executed which change the world state, but keep the activated cell of a particular filter the same. Such actions will not be considered by that filter to improve the state value. After some time, an action is executed which changes the activated cell of the filter, which is now activated in a much better world state. Therefore, the filter learns a large preference to select that action immediately. This may lead to a policy change — in a new trial the new action may be tried out a number of times, but leads to failure. The result is that other filters will start penalizing that action in their currently activated cells and this will suppress that action from being selected immediately. Finally, the system may converge to cell/action values which postpone the action for a while after which it will be selected.

Difference to CMAC-Q. The CMAC Q-learning system uses the overall evaluation of the complete set of filters to update previous filter/cell/action triples, whereas CMAC models keep filters separated from each other. Thus, for the example above, Q-learning may learn Q-values of the filter which approve of the first action, even although the action does not change the filter's cell. However, once the other action needs to be selected, the filter itself may not have learned a large preference for that action. Thus, for Q-learning, the final Q-functions of filters may contain Q-values of different actions which are much closer to each other than those learned by the CMAC model (which may use large preferences and disapprovements).

Estimating the model. To estimate the transition model for the k^{th} filter, we count the transitions from activated cell f_k^t to activated cell f_k^{t+1} at the next time-step, given the selected action a. Thus, for all filters $k = 1, \ldots, Z$, we compute after each transition:

$$C_{f_k^t f_k^{t+1}}(a) \leftarrow C_{f_k^t f_k^{t+1}}(a) + 1$$

These counters are used to estimate the transition probabilities for the k^{th} filter $P_k(c_j|c_i, a) = P(f_k^{t+1} = c_j|f_k^t = c_i, a)$, where c_j and c_i are cells, and a is an action. For each transition we also compute the average reward $R_k(c_i, a, c_j)$ by summing the immediate reinforcements, given that we make a step from active cell c_i to cell c_j by selecting action a.

Prioritized sweeping (PS). Again we apply prioritized sweeping (PS) to compute the Q-function. This time we update the Q-value of the filter/cell/action triple with the largest size of the Q-value update before updating others. Each update is made via the usual Bellman backup (Bellman, 1961):

$$Q_f(c_i, a) \leftarrow \sum_j P_f(c_j|c_i, a)(\gamma V_f(c_j) + R_f(c_i, a, c_j))$$

After each agent action we update all filter models and use PS to compute the new Q-functions. Note that PS may use different numbers of updates for different filters, since some filters tend to make larger updates than others and the total number of updates per time step is limited. The complete PS algorithm is given in Appendix D.

Incrementally constructing filters. If there would be rare interactions between features, we could model them using filters based on their combination. This could mean that we may start out with a lot of parameters, however. A better way may be to incrementally construct and add filters. We can construct filters by making new combinations of inputs. Then we can add them in several ways: (1) we always use them, (2) filters can be only invoked if some condition holds, e.g. if some other particular filter/cell is activated, (3) filters may replace other filters if some condition holds.

The good thing of CMACs is that we can easily add filters without making large changes to the policy. Thus, adding filters may only improve the policy on the long term, although the improvement is at the expense of larger computational costs (evaluation costs are linear in the number of filters).

CMAC models are related to different kinds of probabilistic graphical models (Lauritzen and Wermuth, 1989). Both can be used to learn to estimate the dynamics of some state-variables. CMACs uses a committee of experts for this, whereas Bayesian networks fuse probabilistic dependencies according to Bayes' rule. This makes inference in CMAC models faster, although they will in general need more space to store an accurate model.

7.4 A Soccer Case Study

We use simulated soccer to study the performances of the function approximation methods. Soccer provides us with an interesting environment, since it features high-dimensional input spaces and also allows for studying multi-agent learning.

7.4.1 Soccer

Soccer has received attention by various researchers (Sahota, 1993; Asada et al., 1994; Littman, 1994a; Stone and Veloso, 1996; Matsubara et al., 1996). Most early research focused on physical coordination of soccer playing robots (Sahota, 1993; Asada et al., 1994). There also have been attempts at *learning* low-level cooperation tasks such as pass play (Stone and Veloso, 1996; Matsubara et al., 1996). Littman's (1994) used a 5×4 grid world with two single opponent players to examine the gain of using minimax strategies instead of using best average strategies. Learning complete soccer team strategies in more complex environments is described in (Luke et al., 1997; Stone and Veloso, 1998).

Our case study will involve simulations with continuous-valued inputs and actions, and up to 11 players (agents) on each team. We let team players (agents) share action set and policy, making them behave differently due to position-dependent inputs. All agents making up a team are rewarded or punished collectively in case of goals. We conduct simulations with varying team sizes and compare several learning algorithms: offline Q(λ)-learning with linear neural networks (Q-lin), offline Q(λ)-learning with neural gas (Q-gas), online Q(λ) with CMACs, and model-based CMACs. We want to find out how well these methods work — what are their strengths and weaknesses? This gives us the possibility to increase our understanding of why some function approximators work well in combination with RL for some tasks and why some do not.

Evolutionary computation vs RL. Finally, we will compare the performances of the RL methods to an evolutionary computation (EC) approach (Holland, 1975; Rechenberg, 1971) called Probabilistic Incremental Program Evolution (PIPE) introduced in (Sałustowicz and Schmidhuber, 1997). A PIPE alternative for searching program space would be genetic programming (GP) (Cramer, 1985; Koza, 1992), but PIPE compared favorably with Koza's GP variant in previous experiments (Sałustowicz and Schmidhuber, 1997). With the comparison to an EC method, we want to find out in what RL is particularly good or bad. When we look at backgammon, for which both approaches have been used before, the EC method described in (Pollack and Blair, 1996) was able to learn much faster in the initial phase, but did not lead to such a good final performance as that of TD-Gammon (Tesauro, 1992). Thus, it seems that EC can perform a faster coarse search, whereas getting high performance results in the long term is easier done with RL.

PIPE. PIPE searches for a program which achieves the performance. Each program is a tree composed of a set of functions such as {*sin, log, cos, +, exp*}, and terminal symbols (input variables and a generic random constant). The system generates programs according to *probabilistic prototype trees* (PPTs). The *PPTs* contain adaptive probability distributions over all programs that can be constructed from the predefined instruction set. The *PPTs* are initialized with probabilities for selecting a terminal symbol at a node and probabilities for selecting a function. If the former probabilities are set to high values, the generated programs are usually quite small (although learning can cause generated programs to grow in size). After generating a population of individuals, each individual is evaluated on the problem (e.g. by playing it one game against the opponent). Then, the best performing individual (the one which maximizes the score difference) is used for adapting the PPTs: all probabilities of functions used by the programs of the best individual are increased so that generating the winning individual gets more probable (e.g. after adapting the probabilities, the probability that a newly generated program would be the best program from the previous generation is 20%). Thus, the search always focuses around the best performing program. To limit the problem of ending up in local minima, a mutation rate is added to the system which randomly perturbates probabilities in the PPTs. A large difference with GP is that instead of individuals, a probability distribution over individuals is stored, and that no crossover is used. Therefore PIPE is closer related to Evolutionary Strategies (ES) (Rechenberg, 1989) and PBIL (Baluja, 1994).

7.4.2 The Soccer Simulator

We wrote our own soccer simulator, although there are also other, more sophisticated, simulators available.[11] Our discrete-time simulations feature two teams with either 1, 3 or 11 players per team. We use a two-dimensional continuous Cartesian coordinate system for the field. The field's southwest and northeast corners are at positions (0,0) and (4,2) respectively. Goal width is 0.4. As in indoor soccer the field is surrounded by impassable walls except for the two goals centered in the east and west walls. Only the ball or a player with ball can enter the goals. There are fixed initial positions for all players and the ball (see Figure 7.5).

Players/Ball. Players and ball are represented by solid circles consisting of a centre coordinate and a real-valued orientation (direction). Initial orientations are directed to the east (west) for the west (east) team. A player whose circle intersects the ball picks it up and owns it. The ball can be moved or shot by the player who owns it. When shot, the speed of the ball (v_b) gains an initial speed (0.12 units/time step) and then decreases over time due to friction: $v_b(t+1) = v_b(t) - 0.005$ until $v_b(t) = 0$. This makes shots possible of 1.5 units (37.5% of the length of the field).

[11]See e.g. RoboCup JavaSoccer from Tucker Balch, which follows Robocup rules, retrievable from http://www.cc.gatech.edu/grads/b/Tucker.Balch/JavaBots/EDU/gatech/cc/is/docs/index.html.

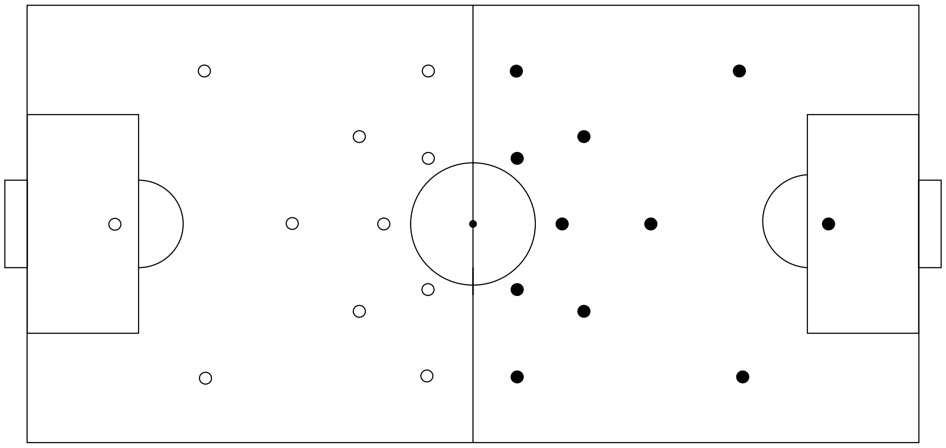


Figure 7.5: *22 players and ball (in the centre) in initial positions. Players of a 1 or 3 player team are those farthest in the back (defenders and/or goalkeepers).*

Players collide when their circles intersect. This causes the player who made the responsible action to bounce back to his previous positions at the previous time step. If one of the collision partners owned the ball prior to collision, the ball will change owners. There are four actions for each player:

- *go_forward:* move player 0.025 units in its current direction if without ball and $0.8 \cdot 0.025$ units if he owns the ball.
- *turn_to_ball:* change the player's orientation so that the player faces the ball.
- *turn_to_goal:* change the player's orientation so that the player faces the opponent's goal.
- *shoot:* If the player does not own the ball then do nothing. Otherwise, to allow for imperfect, noisy shots, turn the agent with an angle picked uniformly random from the interval $[-5°, 5°]$, and then shoot the ball according to the player's new orientation. The initial ball speed is: $v_b^{init} = 0.12$.

Action framework. A game lasts from time $t = 0$ to time $t_{end} = 5000$. The temporal order in which players execute their moves during each time step is chosen randomly. Once all players have selected a move, the ball moves according to its speed and direction, and we set $t \leftarrow t + 1$. If a team scores (or $t = t_{end}$), then all players and ball will be reset to their initial positions.

Input. At a given time t, player p's input vector consists of 14 basic features:

- Three boolean inputs that tell whether (1) the player has the ball; (2) a team member has the ball; (3) the opponent team has the ball.
- Polar coordinates (distance, angle) of both goals and the ball with respect to the player's orientation and position.
- Polar coordinates of both goals relative to the ball's orientation and position.

- Ball speed.

These 14 basic features do not provide information about the position of the other players including the opponents. Therefore, we designed also a more complex input consisting of 16 (1 player) or 24 (3 players) features by adding the following features:

- Polar coordinates (distance and angle) of all other players w.r.t. the player ordered by (a) teams and (b) distances to the player.

For Q-lin, Q-gas and PIPE, we change the input of distance d and angle ρ in $[-\Pi, \Pi]$, by applying the following functions to them before giving the input to the learning algorithms: $d \leftarrow \frac{5-d}{5}$ and $\rho \leftarrow e^{-20 \cdot \rho^2}$. This was mainly done to simplify Q-lin's function representation, so that it is able to focus its action selection on small distances and angles. For the other algorithms it should not matter much.

7.4.3 Comparison 1: Q-lin, Q-gas and PIPE

We first compare the following methods: linear networks and neural gas trained with offline Q(λ), and the evolutionary method (Rechenberg, 1971; Holland, 1975; Koza, 1992) PIPE (Sałustowicz and Schmidhuber, 1997). Results have been previously published in (Sałustowicz et al., 1998; Sałustowicz et al., 1997a). .

We have used an offline Q(λ) variant (using accumulating traces) for training the FAs. The reason for using offline Q(λ)-learning is that reinforcement is only given once a goal is scored, so that the largest reason of using online learning, improving initial exploration, does not apply, whereas offline learning is computationally cheaper.

Experimental set-up

Opponent. We train and test all learners against a "biased random opponent" *BRO*. *BRO* randomly executes actions, but due to the initial bias in the action set it performs a goal-directed behavior. If we let *BRO* play against a non-acting opponent *NO* (all *NO* can do is block) for twenty 5000 time step games then *BRO* always wins against *NO* with on average 72 to 0 goals for team size 1, 45 to 0 goals for team size 3, 109 to 1 goals for team size 11.

We also designed a simple but good team *GO* by hand. *GO* consists of players which move towards the ball as long as they do not own it, and shoot it straight at the opponent's goal otherwise. If we let *GO* play against *BRO* for twenty 5000 time step games then *GO* always wins with on average 417 to 0 goals for team size 1, 481 to 0 goals for team size 3, and 367 to 3 goals for team size 11. For this simulator, *GO* executes a very good single agent strategy. Note, however, that *GO* implements a non-cooperative strategy, this makes it suboptimal for larger team sizes.

Game duration and inputs. We play 3300 games of length $t_{end} = 5000$ for team sizes 1 and 11 (we will not show results with 3 players here, since they are very comparable to those of 1 and 11 players). Every 100 games we test current performance by playing 20 test games (no learning) against *BRO* and sum the score results.

For all methods we used the 14 basic input features (so without the additional information about other players).

Linear network set-up. After a coarse search through parameter space we used the following parameters for all Q-lin runs: α_l=0.0001, λ=0.9, H_{max}=100 (a small size of the history list worked best. Although in this case the FA is not trained on all possible examples, the most important ones leading to goals are learned.) All network weights are randomly initialized in $[-0.01, 0.01]$. We use Boltzmann exploration where during each run the Boltzmann-Gibbs rule's greediness parameter (the inverse of the temperature) is linearly increased from 0 to 60. The discount factor γ=0.99 to encourage quick goals (or a lasting defense against opponent goals), r_{t^*}, the reinforcement at trial end, is -1 if opponent team scores, 1 if own team scores, and 0 otherwise.

Neural gas set-up. For Q-gas we used: α_g=0.1, $\beta = 0.1$, λ=0.9, H_{max}=100, $\eta = 30$ (note that this large value makes the FAs very local), the initial number of neurons Z_{init}=10, and we constrain the maximum of neurons Z_{max}=100. We use Max-random exploration with P_{max} = 0.7. Parameters for adding neurons are: the required error T_E=0.5 and $T_C = 1000$. We used the Manhattan distance or L_1 norm, although some trial experiments using Euclidean distance resulted in similar performances. The components of the neuron centers w_k are randomly initialized in $[-1.0, 1.0]$. Q-values are zero-initialized. The discount factor γ=0.98. Again r_{t^*} is -1 if opponent team scores, 1 if own team scores, and 0 otherwise.

PIPE set-up. We compare our RL methods with the evolutionary search method PIPE (Sałustowicz and Schmidhuber, 1997). PIPE searches for the individual consisting of five programs (one for each action and one for determining the temperature for the used Boltzmann exploration rule) achieving the best score difference when tested against the opponent. The most interesting parameters for PIPE runs are set to: learning rate=0.2, mutation rate=0.2, and population size=10. During performance evaluations, we test the current best-of-generation program (except for the first evaluation where we test a random program). Note that this is an advantage for PIPE, since the program which is tested is known to outperform all other programs of the same generation.

Experimental results

We compare average score differences achieved during all test phases against *BRO*. Figure 7.6 shows results for PIPE, Q-lin and Q-gas. It plots goals scored by learner and opponent against number of games used for learning (averaged over 10 simulations). Larger teams score more frequently because some of their players start out closer to the ball and the opponent's goal.

PIPE learns fastest and always quickly finds an appropriate policy regardless of team size. Its score differences continually increase. Q-lin and Q-gas also improve, but in a less spectacular way. They are able to win on average and tend to increase score differences until they score roughly twice as many goals as in the beginning (when action selection is still random).

For the single agent case, Q-gas is able to learn good defensive policies, what can be seen from the strongly reduced opponent scores. For the multi-agent case, Q-gas slowly increases its score differences, although its learning performance shows many fluctuations. It seems that the system is changing its policy a lot and does not seem able to fix a good policy. Still it is almost always able to win.

For Q-lin the score differences start declining after a while — the linear neural networks cannot keep and improve useful value functions but tend to unlearn them instead. We will now describe a deeper investigation of the catastrophic performance breakdown in the 11 player Q-lin run followed by an explanation of neural gas' instability problems.

Q-lin's instability problems. Some runs of Q-lin led to good performance, although

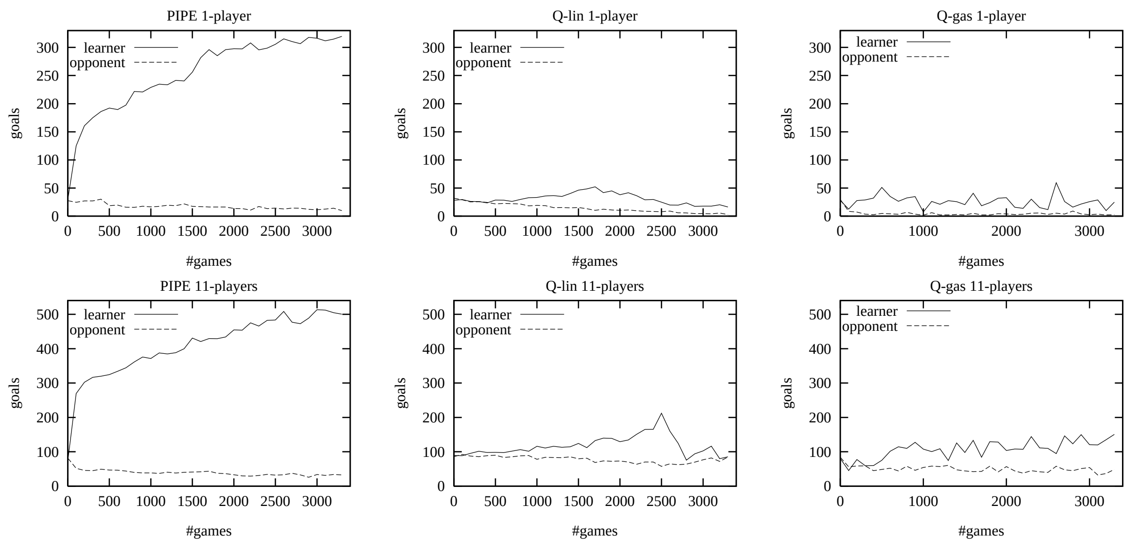


Figure 7.6: *Average number of goals scored during all test phases, for team sizes 1 and 11. Averages were computed over 10 simulations.*

sudden performance breakdowns hint at a lack of stability of good solutions. To understand Q-lin's problems in the 11 player case we saved a "good" network just before breakdown (after 2300 games). We performed 10 additional simulations, for which we continued training the saved network for 25 games, testing it after every training game by playing 20 test games. To achieve more pronounced score differences we set the temperature parameter in the Boltzmann exploration rule to $\frac{1}{90}$ — this leads to more deterministic behavior than the value round $\frac{1}{40}$ used by the saved network.

Figure 7.7(left) plots the average number of goals scored by Q-lin and *BRO* during all test games against the number of games. Although initial performance is very good (the score difference is 418 goals), the network fails to keep it up. To analyze a particular breakdown we focus on a single run. Figure 7.7(middle) shows the number of goals scored during the test phases of this run, and Figure 7.7(right) shows the relative frequencies of selected actions.

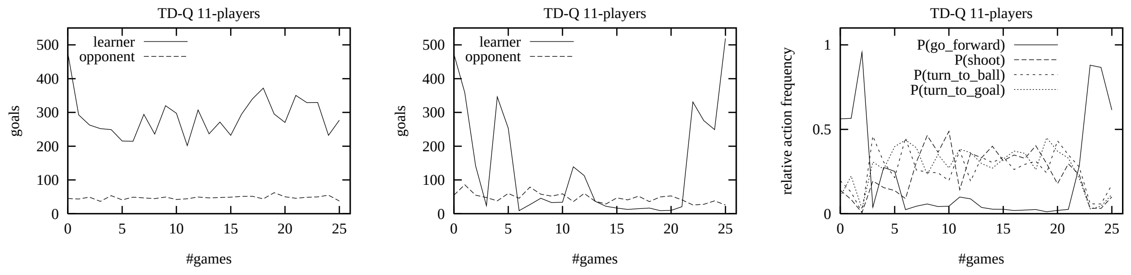


Figure 7.7: *Performance breakdown study. Left: average numbers (means of 10 runs) of goals scored against BRO by a team with 11 players starting out with a well-trained linear network. Middle: plot for a single run. Right: relative frequencies of actions selected during the single run.*

Figure 7.7(middle) shows a performance breakdown occurring within just a few games. It is accompanied by dramatic policy changes displayed in Figure 7.7(right). Analyzing the learning dynamics we found the following reasons for the instability:

(1) Since linear networks learn a global value function approximation, they compute an average expected reward for all game constellations. This makes the Q-values of many actions quite similar: their weight vectors differ only slightly in size and direction. Hence small updates can create large policy changes.

(2) Q-lin adapts some weight vectors more than others — the weight vector of the most frequent action of some rewarding (unrewarding) trial will grow (shrink) most. For example, according to Figures 7.7(middle) and 7.7(right), the action *go_forward* is selected most frequently by the initially good policy. Two games later, however, the behavior of the team is "dominated" by this action. This results in worse performance and provokes a strong "correction" in the third game. This suddenly makes the action unlikely even in game constellations where it should be selected, and leads to even worse performance.

For 11 player teams the effect of update steps on the policy is 11-fold and therefore instabilities due to outliers can easily become much more pronounced.

The question can be asked whether the RL method is the cause for the learning difficulties or whether it is the linear networks. Although the linear networks are not very powerful, they can still be used to reach high performance levels (even although the learned value function approximation may be very bad). Linear models could get rid of at least some of the instability problems. We have not tried them, however, since they are computationally much more expensive.

Neural gas's instability. Just like the linear network, neural gas is not able to constantly improve its policy. For the single agent case it quickly diminishes the opponents score to few goals and increases its own score, but does not keep good policies producing many goals. We also combined the neural gas with online Q(λ)-learning, but this did not improve the results. Analysing the system, we found one particularly important reason for neural gas' learning problems: adding neurons usually helps to improve the performance, but sometimes adding a neuron can ruin good policies. Noise can be responsible for placing a neuron which starts dominating some region, adds errors to its Q-function approximation, and selects a wrong action. Ways to solve this problem is to design more clever algorithms for adding neurons. If a neuron is added, the algorithm makes an unrealistically large update step of the value function. A way to get around this is to use local greediness values in Equation 7.1 for computing the neuron gate-values. Then we can initialize a greediness value of a newly added neuron to a very low value and gradually increase it so that it has time to adapt to its environment before being used fully by the system.

The lesson we have learned here is that (direct) RL methods require evaluating policies which do not change very fast. Each policy needs many experiences to be completely evaluated, and although we can update the value function before we have collected a very large number of experiences (which would make learning very slow), we should be careful not to make any large changes based on little experience, since that may cause havoc — previous "plans" or strategies suddenly may not work anymore. Thus, using state quantization methods for RL seems a powerful technique, but applying it may be quite difficult since they need lots of parameter twiddling and lack robust structures. We should try to gradually improve the structure and make small Q-value updates.

Neural gas models. We also tried using neural gas models, but these did not work successfully either. One of the main problems of this approach is that transition probabilities are estimated according to a particular positioning of the neurons, which constantly (and probably too fast) changes. A second problem was that neural gas models result in many outgoing transitions from each neuron, so that there is a lot of uncertainty about predicting the next state and Bellman backups consume a lot of time. Starting with a fixed structure (and connective neighborhood) may circumvent this problem, but choosing an *a priori* structure is a difficult design issue on its own. Finally, instability problems are caused by the used insertion procedure and initialization of new neurons.

Conclusion

We found that linear networks, despite of their lack of expressive power, can be used for representing and finding good policies. Since their function approximation of the true evaluation function is unstable, however, they quickly unlearn good policies once they are found.

The neural gas method showed faster initial learning behavior than the linear networks, but in our experiments the neural gas systems did not continuously learn to improve policies. A problem of the neural gas method is that although adding neurons is usually useful, sometimes it leads to bad interference with the learned approximation. Therefore more careful algorithms for growing cell structures should be used.

Both algorithms performed worse than the evolutionary method PIPE, which tests multiple different individuals and has a large probability of generating winning programs of the previous round again, thereby safeguarding the best found policies. Furthermore, PIPE was able to quickly find good policies, since it was biased to start searching for low complexity programs, and quickly discovered which features were important to use. Surprisingly, some low complexity programs implemented very good policies against the fixed soccer opponent.

Learning soccer strategies with a single player or with multiple players does not seem to be very different in our soccer environment. The only real difficulty with multiple players is that particular outliers get heavier weight when we use multiple players, thus leading to more learning instability and requiring more robustness from the FA.

From the current results, we conclude that value function based RL methods should be extended to make them also profit from (a) feature selection facilities, (b) existence of low-complexity solutions, (c) incremental search for more complex solutions where simple ones do not work, and (d) keeping and improving the best solution found sofar.

7.4.4 Comparison 2: CMACs vs PIPE

The previous function approximators were too unstable for reliably learning good soccer policies. CMACs possess different properties: while still being local, all cells have fixed locations. This could possibly make them more stable and therefore more suitable for RL. We will now compare CMACs trained with online Q(λ), model-based CMACs trained with PS, and PIPE. Results have been previously published in (Wiering et al., 1998).

Experimental set-up

Task. We train and test the learners against handmade programs of different strengths. We use a different set-up from the previous experiments to get some results against better opponents, since we expect that evolutionary computation may be faster when it comes to finding solutions to simpler problems. The opponent programs are mixtures of the program BRO which randomly executes actions and the program GO which moves players towards the ball as long as they do not own it, and shoots it straight at the opponent's goal otherwise.

Our five mixture programs, called *Opponent(P_r)*, use *BRO* for selecting an action with probability $P_r \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$, and otherwise they use *GO*. Thus, *Opponent*(1.0) equals *BRO* and *Opponent*(0.0) equals *GO*.

We test team sizes 1, and 3 (team sizes 3 and 11 do not show large differences in learning curves, although team size 11 consumes much more simulation time).

For all methods we used the extended input vectors (with the additional player dependent input features). We have observed that for the neural gas and linear networks there were minor advantages using extended inputs, although the overall results look more or less the same. Thus we use 16 inputs for the 1-player case and 24 inputs for the 3-player case.

CMAC-Q set-up. We play a total of 200 games. Every 20 games we test current performance by playing 20 test games against the opponent and sum the score results. The reward is +1 if the team scores and -1 if the opponent scores. The discount factor is set to 0.98. We used fast online Q(λ) with replacing traces: $\lambda = 0.8$ for the 1-player case, and $\lambda = 0.5$ for the 3-player case. Initial learning rate (lr) $\alpha_c = 1.0$, lr decay rate $\beta = 0.3$. We used Max-random exploration with $P_{max} = 0.7 \rightarrow 1.0$. We use 2 filters per input (total of 32 or 48 filters) and set the number of cells $n_c = 10$. Q-values are initially zero.

PIPE set-up. For PIPE we play a total of 1000 games. Every 50 games we test performance of the best program found during the most recent generation. Parameters for all PIPE runs are the same as in previous experiments (Sałustowicz et al., 1998).

CMAC model set-up

For the CMAC models, we have some additional special features which improved the learning performance.

Multiple restarts. CMAC models turn out to be very fast learners: often they converge to a particular policy after a small number of games. The method sometimes gets stuck with continually losing policies (also observed with our previous simulations based on linear networks and neural gas), however. We could not overcome this problem by adding standard exploration techniques. Instead we reset the Q-function and WM once the team has not scored for 5 successive games whereas the opponent scored during the most recent game (we check these conditions every 5 games). This multiple restarts combined with model-based CMACs (which behaves as a stochastic hillclimber) is a new way for searching policies in RL, but is also used in operations research (OR) for searching for solutions to combinatorial optimization problems. Note that it is no problem whatsoever to use multiple restarts with any learning algorithm for any problem, e.g., it could also be used for lifelong learning approaches.

Non-pessimistic value functions. Since we let multiple players share their policies, we “fuse” experiences of multiple different players inside a single representation. These experiences are, however, generated by different player histories (scenario’s) and therefore some experiences of one player would most probably never occur to another player. Thus, there is no straightforward way of combining experiences of different players. For instance, a value function may assign a low value to certain actions for all players due to previous unlucky experiences of one player. To overcome this problem we compute non-pessimistic value functions: we decrease the probability of the worst transition from each cell/action and renormalize the other probabilities. Then we use PS with the new probabilities. Thus, basically we do a similar thing as in Chapter 5 for computing optimistic value functions. The difference is that in Chapter 5, we made good experiences more important. Here we make bad experiences less important. Details are given in Appendix D.

Parameters. We play a total of 200 games. Every 10 games we test current performance by playing 20 test games against the opponent and summing the score results. The reward is +1 if the team scores and -1 if the opponent scores. The discount factor is set to 0.98. After a coarse search through parameter space we chose the following parameters. We use 2 filters per input (total of 32 or 48 filters) and set the number of cells $n_c = 20$, Q-values are initially zero. PS uses $\epsilon = 0.01$ and a maximum of 1000 updates per time step.

Experimental results

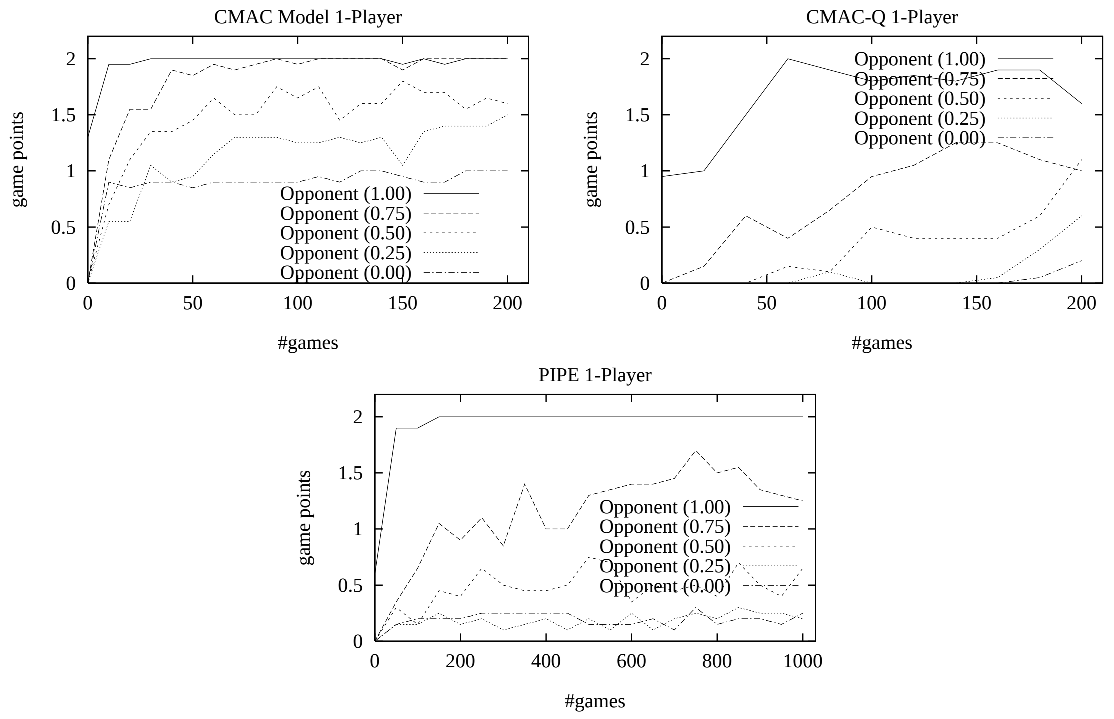


Figure 7.8: *Number of points (means of 20 simulations) during test phases for teams consisting of 1 player. Note the varying x-axis scalings.*

Results : 1-Player case. We plot number of points (2 for scoring more goals than the opponent during the 20 test games, 1 for ties, 0 for losses) against number of training games in Figure 7.8.

We observe that on average our CMAC model wins against almost all training programs. Only against the best 1-player team ($P_r = 0$) it wins as much as it loses. Against the worst two teams, CMAC model always finds winning strategies.

CMAC-Q(λ) finds programs that on average win against the random team, although they do not always win. It learns to play about as well as the 75% random and 50% random teams. CMAC-Q(λ) performs poorly against the best opponent, and although it seems that the performance jumps up at the end of the trial, longer trials do not lead to better performances.

PIPE is able to find programs beating the random team and quite often discovers programs that win against 75% random teams. It encounters great difficulties in learning good strategies against the better teams, though. Although PIPE may execute more games (1000 vs. 200), the probability of generating programs that perform well against the good opponents is very small. For this reason it tends to learn from the best of the losing programs. This in turn does not greatly facilitate the discovery of winning programs.

Results : 3-Player case. We plot number of points (2 for scoring more goals than the opponent during the 20 testgames) against number of training games in Figure 7.9.

Again, CMAC model always learns winning strategies against the worst 2 opponents. It loses on average against the best 3-player team (with $P_r = 0.25$) though. Note that this strategy mixture works better than always using the deterministic program ($P_r = 0$) against which CMAC model plays ties or even wins. In fact, the deterministic program tends to clutter agents such that they obstruct each other. The deterministic opponent's behavior also is easier to model. All of this makes the stochastic version a more difficult opponent.

CMAC-Q is clearly worse than CMAC model — it learns to win only against the worst opponent.

PIPE performs well only against random and 75% random opponents. Against the better opponents, it runs into the same problems as mentioned above.

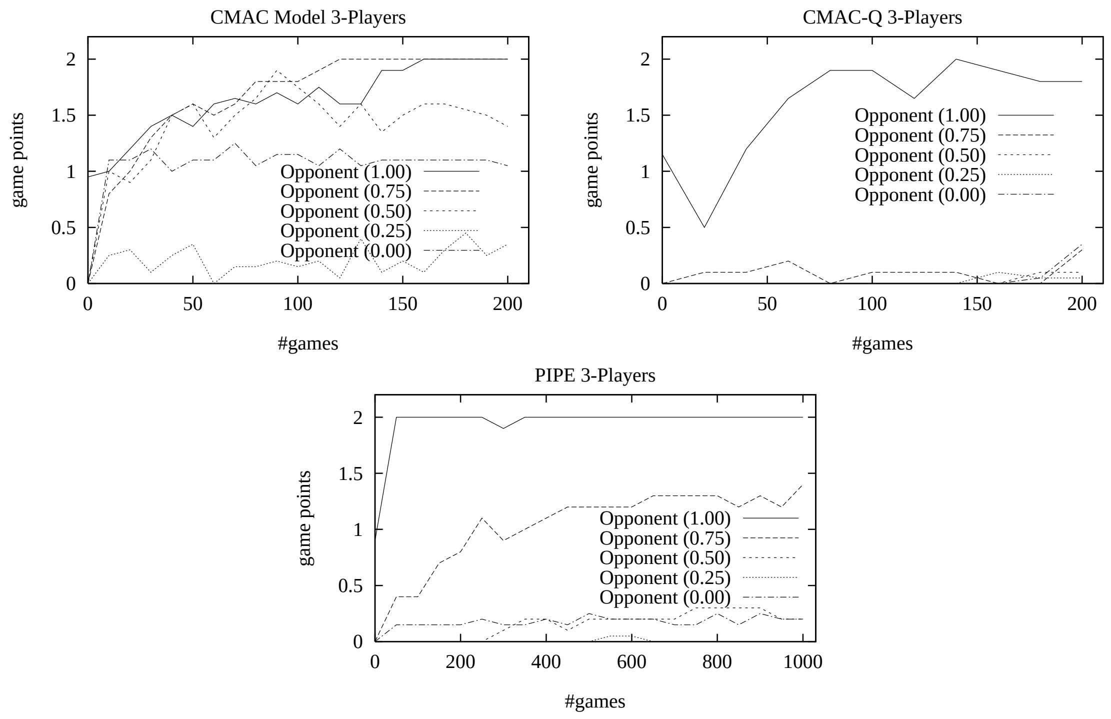


Figure 7.9: *Number of points (means of 20 simulations) during test phases for teams consisting of 3 players. Note the varying x-axis scalings.*

Score differences. We show the largest obtained score differences in Table 7.1 (1 player) and Table 7.2 (3 players). We should keep in mind that these score differences can have a large variance and may thus not convey too much information. E.g. in the case of CMAC models with 1 player, Opponent(0.0) may sometimes win 760-0 against CMAC models, since the CMAC model had not already found a good policy and its policy had just been resetted before testing. Although this only happens 2 or 3 times out of 20 simulations, these large scores have a tremendous impact on the overall averaged score results. PIPE has a small advantage there, since PIPE always uses the best found program during the last generation for testing, and thus effectively diminishes the probability to almost 0 of testing a really bad policy. Still, the tables show that although PIPE is able to score more against the bad opponents than CMAC-models or CMAC-Q, PIPE often cannot score against the good opponents. CMAC-models do score against the good opponents, and were able to find (at least 1 time) winning policies against all opponents. It is difficult to say whether winning 127-35 is better or worse than winning 68-3, though. We remark that PIPE performs better against the weakest opponent and CMAC models perform best against the best opponents.

Learning Algorithm	1.0	0.75	0.5	0.25	0.0
CMAC-models	85-2	68-3	27-1	6-15	1-146*
CMAC-Q	92-6	52-23	10-7	1-4	0-13
PIPE	225-18	127-35	19-13	0-3	0-10

Table 7.1: *Best average score differences for the different learning methods against different strengths of the 1-player opponent. * = Although CMAC-models were sometimes able to score 7 goals, they also sometimes lost 0-760.*

Learning Algorithm	1.0	0.75	0.5	0.25	0.0
CMAC-models	161-31	236-100	84-70	6-20	0.3-0
CMAC-Q	111-26	36-73	13-58	3-23	0-24
PIPE	297-18	163-64	30-31	0-11	0-21

Table 7.2: *Best average score differences for the different learning methods against different strengths of the 3-player opponent.*

Instead of showing score differences, we may also plot relative score differences. We compute relative scores as:

$$relative\_score = \frac{Player\_goals}{Player\_goals + Opponent\_goals}$$

This measure has as a drawback that the scores 1-0 and 10-0 are treated equally, although this makes comparing results against the strong opponents with results against the weak opponents easier possible. We plot relative scores against number of games in Figure 7.10. Here we can clearly see that relative scores against the 1-player team are larger than those against 3-player teams.

Comments

Filter design. For CMAC-models we used 20 cells for each single-input filter, whereas we used 10 cells for CMAC-Q. Changing the number of cells does not affect the results very much, although learning can become slower. Using 4 instead of 2 filters results in good performance as well, although the computational time is doubled. A single filter results in less stable learning and worse performance, however. Finally, different filter designs combining different inputs works good as well. Thus, it seems that the method is quite robust to the actual design of the filters.

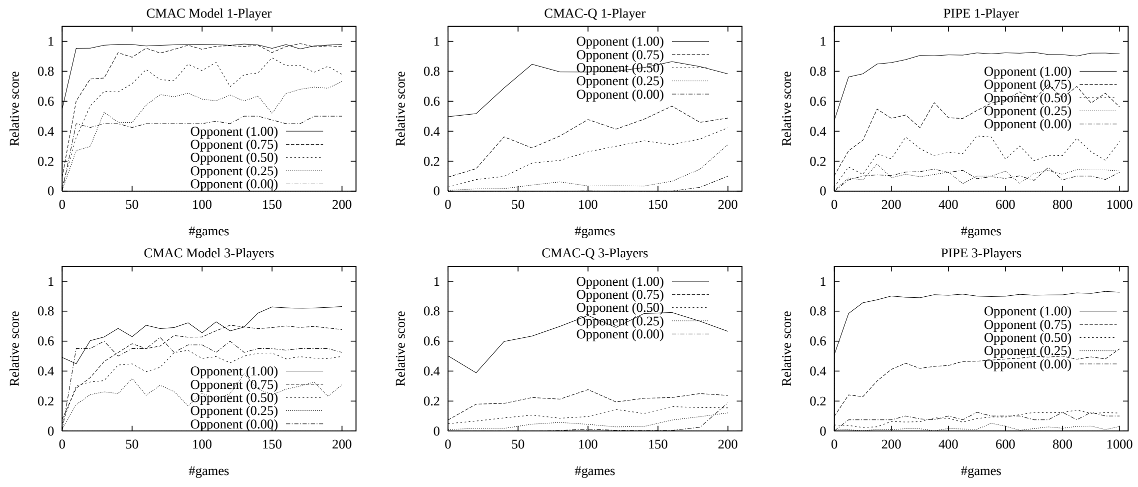


Figure 7.10: Relative scores (own goals / total goals) for the 3 methods for team sizes 1 and 3.

Non player-sharing policies. We also tried using different representations for the Q-functions of the 3-player teams. For this, we used CMAC-models and used an independent model for all players. We did not use the non-pessimistic value function approach, since the Q-functions were not shared. Against Opponent(1.0), the results were comparable with sharing policies, except that the CMAC-models now tended to score slightly more (the best result was 173-27), but did not always win (average points was 1.9). Against Opponent(0.25), the results were much worse: CMAC models lost on average (0-23), and gained 0 average points. Thus, it seems that policy sharing improves learning when tasks get more complicated.

Exploration in time critical problems. In certain problems, an agent has to reach a specific goal before some time criteria or time-horizon has elapsed. For such time critical problems, policies need to be good and almost deterministic (exploiting or greedy) to be successful since each bad/non-policy action will waste precious time. This makes exploration very difficult. Soccer is an example of such a time critical problem. If we play with a policy that contains too many bad actions against a very good opponent, then the opponent will score even before one of our own players has approached the ball. Since in each state all actions are tried about the same number of times, it is very hard to learn which actions are really bad. This problem is shown by the results of all learners against the 3-player *Opponent*(0.25) team. There are no good ways to circumvent this problem with RL. If the agent only receives reward at a specific goal state, and that goal state is only reachable in a small percentage of all trials, other ways for initially exploring the policy space are needed such as exhaustive search, means-end analysis, or *a priori* knowledge to be able to test policies which perform well enough so that the goal is found.

7.4.5 Discussion

Online vs Offline Q(λ). We used offline Q(λ) for the first two FAs and online Q(λ) for CMACs. We switched to online learning because we thought that it might improve learning performance. Although online learning improves things a little bit, the different function approximators are the main reason for the different learning performances. Although all FAs were often able to learn to beat the simplest fixed opponent, CMACs obtained the fastest learning performance.

RL vs EC. When we compare RL to evolutionary computation for the soccer task, then we observe that EC may be better in learning very good performances for the easier problems, whereas RL is better in finding good policies for more difficult problems. This confirms our expectations: RL may need more time (e.g. CPU or modeling), but may finally come up with better solutions for difficult problems.

CMAC models no 1. CMAC models are able to learn good, reactive soccer strategies against all opponents despite treating all features independently. They prefer actions that activate those cells of a filter which promise highest average reward. The use of a model stabilizes good strategies: given sufficient experiences, the policy will hardly change anymore.

The curse of filter dimensionality. First of all, a lot depends on the filter design. For particular problems, we would need to construct filters combining many context-dependent features. Such problems would need a lot of cells, which is especially a problem if we use model-based RL. Still, there may be ways cutting down the number of parameters. E.g. we could preprocess all "filter spaces" and transform them into lower-dimensional spaces using Independent Component Analysis, e.g., (Oja and Karhunen, 1995; Bell and Sejnowski, 1998).

Limitations of CMAC models. Note that if there are a huge number of possible actions, we cannot store world model transitions conditioned on the selected action. Instead we model only the effect of the best (selected) action. Then we select an action by simulating all possible actions with a model and evaluating the resulting states using the CMAC. For this we need to have a simulator or model mapping state/actions to subsequent states. Then we update the CMAC model according to the selected action.

Independent filter models. In chess there is so much contextual dependent information, that CMACs would probably not be the best representation for it. However, we can generalize our CMAC models to a set of independent filter models. E.g., we could use decision trees (DT) to save a lot of memory, whereas they are only slightly less time-efficient. DT-models could be used in the same way as CMAC models: we learn Q-values for leave nodes and we learn transition probabilities and rewards between leave nodes. Furthermore to allow for the power of the committee of experts, we can use multiple DT-filters, where each DT-model is based on different subsets of all inputs. Of course, instead of DT-filters, we could also use unsupervised learning in a preprocessing phase and use the resulting quantizations for constructing different filters. The thing to remember is that we are interested in using multiple filters based on different views on the universe so that each policy can be quickly learned and the combined policy works well for the many different (partial) universes.

Non-stationary filters. Finally, there may be problems if filters are only worthwhile during a particular stage of a process. After this stage, invoking them would just cost time. However, nothing forbids us to use temporal sequences of filter-sets or to use a higher-level decomposition of the process into subprocesses, all with their own filter design.

Learning filters. If we have constructed an architecture with a good set of filters, the method may be expected to work well. Still, sometimes we would like to learn the filters. This can be done by different techniques. Since Independent-filter models are quite robust, it offers many possibilities for changing the architecture without causing large policy changes. This may make them suitable for incremental policy refinement.

7.5 Previous Work

There have been quite a few combinations of function approximators and RL. We will give a sample of what has been done. Chapman and Kaelbling (1991) implemented the G-algorithm, a method based on incrementally computing decision trees by splitting nodes receiving inconsistent reinforcement signals. The algorithm used Q-learning for learning the Q-values, which were stored in the leave nodes. The system was successfully tested on an amazone environment, where the goal was to learn to shoot arrows at ghosts. One of the drawbacks of their (and similar) systems is that it could not handle multivariate splitting so that it is restricted to environments where singular inputs can show a difference. Another problem is that it may generate huge trees if all inputs are relevant.

Boyan (1992) used hierarchical neural networks to learn the games of Tic Tac Toe and Backgammon and showed performance gains in using hierarchical nets compared to monolithic networks. Wiering (1995) extended this work and was able to reach optimal performance levels on Tic Tac Toe using Q(λ)-learning. Furthermore, he showed which precision levels can be obtained by neural networks learning the evaluation function of the endgame of backgammon.

Santamaria, Sutton and Ram (1996) compare CMACs to memory-based function approximation with Sarsa (Rummery and Niranjan, 1994; Sutton, 1996) on the double integrator and pendulum swing up problems. They found that both methods profit from a non-uniform, variable resolution allocation of the cells/neurons, and that CMACs profitted most from this. They found an elegant way to change the input space by using a skewing function which increases the size of important regions such as those around start and goal states. The obtained results of both FAs were comparable.

Tham (1995) incorporated a set of CMACs in the hierarchical mixtures of experts (HME) architecture (Jacobs et al., 1991; Jordan and Jacobs, 1992). He used Singh's Compositional Q-learning (CQ-L) architecture (1992) which uses relevant task inputs to gate the different controllers. His system quickly learned good performance levels for different sequences of robot manipulator tasks.

Kröse and van Dam (1993) use RL with a neural gas architecture for collision free navigation. The system gets 8 distance measures as input and has to learn to steer a vehicle left or right in a simulated environment. Whenever there is a collision, neurons are added so that critical regions get high resolution. The system also removes neurons which have similar values as their neighbors. The experimental results a virtual environment show that the system can quickly find collision free paths.

Schraudolph, Dayan and Sejnowski (1994) used TD-learning to learn to play Go on a 9×9 board and showed that the system achieved significant levels of improvement. The program learned good opening strategies, but often made tactical mistakes later in the game. In later phases of the game many complex interactions between stones dominate position evaluation. This was (to our knowledge) the first time anybody used a whole grid of local reinforcement signals to evaluate each possible board position, as opposed to just a single scalar evaluation. This made a big difference in performance.

Thrun implemented neurochess (1995), a program which uses TD-learning and explanation based neural network learning for learning to play chess. He learned a model to predict likely sequences of gameplay and used this model for training the position evaluator. The resulting performance of the neural network was of limited success — a better level of play needs a more accurate value function which takes the precise board constellation into account.

Recently, Baxter, Tridgell and Weaver (1998) came up with a promising method for learning to play chess which is very similar to Samuel's approach in the sense that it combines lookahead search with reinforcement learning. Their system used a simple initially hand-crafted position evaluator which was improved by TD-learning by updating the values of those leave nodes which result from a lookahead search for the best move. Using lookahead makes the accuracy of the position evaluator much less important for good play and that makes learning a useful value function much easier. The system played on the internet chess server and was able to significantly improve its level of play after learning from only 200 games.

Tadepalli and Ok (1998) describe H-learning, a model-based RL method maximizing average reward per time step. They show that H-learning outperforms discounted model-based RL for particular tasks. Furthermore, they extend H-learning by introducing dynamic Bayesian networks to compactly store the model so that the system can cope with large state-spaces. The DBN for storing the model is then combined with local linear regression (LLR) for approximating the value function. The results of this combination on an automated guided vehicle domain show promising results.

Sampling & approximation

There also exists a different way of using models for computing a value function. We still represent the value function in a function approximator, but now we generate a set of state vectors, use the model (or simulator) to perform a one-step lookahead for all states to obtain more accurate values (possibly in combination with Monte Carlo simulations for stochastic problems), and train the function approximator to learn the new state values. The method is in principle a hybrid: we can use one representation for the model (e.g. Bayesian networks), and use another for the value function (e.g. neural networks). Once we have computed a new generation of training examples, we use a training method to adapt the function approximator. The simplest methods use gradient descent to minimize the Bellman error of the generated examples. This is called the direct method, but is has been shown that it may lead to learning instability (Boyan and Moore, 1995; Baird, 1995).

To overcome the problem of diverging value functions, Boyan and Moore (1995) used a method based on learning the value function backwards during which complete "roll-outs" are used for computing new value estimates while keeping a large batch set of training examples. This method showed promising results for deterministic problems, but is computationally expensive for stochastic problems.

Baird (1995) discussed residual gradient algorithms, where steepest descent on the mean squared error Bellman residual is performed. These methods converge to a local minimum on the Bellman error landscape, but tend to be slow. Therefore, Baird introduced residual algorithms which are a mixture between the direct method and the residual gradient method, combining the property of minimizing the Bellman residual at each step with fast learning. The method can be applied for stochastic problems by combining two independent successor states in a single learning example. This is called the two sample gradient method (Bertsekas and Tsitsiklis, 1996).

We have not used these methods here, because it may be difficult to select a set of target points. Furthermore the function can change dramatically if the environment is very noisy, since new target values may be quite different, dependent on the successor state we have sampled. Finally, for real world learning, we cannot sample from arbitrary points, so that the method requires engineering or learning a model first.

7.6 Conclusion

In this chapter we discussed one of the most interesting topics in RL: the application of function approximators to learn the value function. We described three different types of function approximators: linear networks, neural gas, and CMACs, as well as ways to combine them with direct RL and model-based RL. Then we compared them on a soccer learning task.

We found that linear networks had severe learning problems for the soccer task. Although the linear networks were sometimes able to learn good policies, small update steps of the weight vectors sometimes cause large policy changes after which good policies are unlearned.

A difficulty of reinforcement learning compared to supervised learning is that the output value of examples change and are taken from a non-stationary distribution due to policy changes and exploration behavior. Therefore, in order to receive sufficient data from the same input/output mapping, the policy should not change too fast, and the used function approximator should be robust enough so that noise will not affect its approximation very much. We should be careful in assessing whether we really make an improvement step of a particular policy, since it is easy to make changes which decrease the performance level of the policy. Especially if we want to obtain high performance levels, we should circumvent this from happening.

Approximators which use adaptive state quantization suffer from noise which may cause quick policy changes, and therefore they are not always very stable. One function approximator which discretizes the input space using an *a priori* decomposition, CMACs, turned to be quite stable, especially when combined with world models. CMAC models are able to capture lots of statistics, and makes fast and robust learning possible. One problem of CMACs is that filters combining different input features need to be designed by hand. Therefore, we would like to use methods which can learn the possible filters.

In this chapter we have not given any special attention to multi-agent RL. Future work on robotic soccer may profit by studying team strategies and the integration of perceptions of different agents. By mapping group states to team strategies, a useful abstraction can be made. Furthermore it may be easier to assign rewards to team strategies than to single player actions. Therefore it may be a good idea to define a set of team strategies involving reactive policies for each individual agent.

Chapter 8

Conclusion

In this final chapter we describe our contributions to different topics of interest in the reinforcement learning field. We also describe the limitations of the proposed methods and what can be done to overcome these limitations. Finally, we close with some final remarks.

8.1 Contributions

8.1.1 Exact Algorithms for Markov Decision Problems

Discussion

First of all, we compared two dynamic programming algorithms and found that value iteration is faster than policy iteration for computing policies for goal directed problems. The reason is that policy iteration completely evaluates policies before making policy updates and that policies in the initial iterations generate many cycles through the state space. Such cycles are certainly not optimal, but do cost a lot of computation time. On the other hand, value iteration uses its value function directly for greedily changing the policy whenever it needs to be changed. In this way it quickly computes goal-directed policies which generate few cycles and therefore it can stop evaluating policies much earlier. We conclude that methods profit from immediately using information so that policy changes are made earlier.

Limitations

Solving Markov decision problems with dynamic programming requires a model of transition probabilities and a reward function for computing policies which is often *a priori* not available. Furthermore, if there are many state variables, the computational costs become overwhelming. The first problem can be solved by using reinforcement learning. For solving the second problem we need to use function approximators which aggregate states.

8.1.2 Reinforcement Learning

Discussion

We developed a novel online Q(λ) algorithm which computes the same updates as previous (exact) online Q(λ) methods, but is able to make updates in time proportional to the number of actions and the number of active elements of the function approximator. In the case of local function approximators, e.g. lookup tables which activate a single element at each time step, the gain of the new method can be very large. The new algorithm is based on lazy learning which postpones computations until they are really needed. The novel algorithm was used in experiments with lookup tables for maze problems and with CMACs for learning to play soccer. Both applications resulted in a large speed up.

We developed an algorithm for online multi-agent Q(λ), called Team Q(λ), which allows updating the Q-value of some state visited by some agent on a change of the Q-value of another state visited by another agent. The only thing that is required is that there exists a directed path between the states, which now may go through some crossing point between the trajectories of the agents. With the new method, information is propagated along many more trajectories than would be the case if we would only update along single agent histories. Furthermore, the Team Q(λ) algorithm can be used for assigning credit in case of general interactions between agents, such as passing the ball, where we would like to evaluate the action on the basis of what happens to the ball in the future. Although we expect the method to be beneficial for solving real cooperative tasks for which agents work as a group, experimental results in a non-cooperative maze environment did not show that Team Q(λ) outperformed letting agents learn from their individual traces only.

Limitations

Direct RL methods such as TD(λ) and Q-learning have problems to backpropagate information over many time steps. Q-learning can only backpropagate information one step in the past, whereas TD(λ) can in principle be used to backpropagate information over many time steps, but this causes a large variance in the updates (especially for stochastic problems) which delays the convergence of value functions.

The Team Q(λ) is able to combine traces of multiple agents, but does not take into account when these traces have been generated. Therefore, very old traces generated by old policies are used to change the current value function, which may decrease the policy's performance. The algorithm is also not yet made amenable for different kinds of agent interactions — the only kind of interaction we have implemented so far is when different agents visit the same state. For more general applications, the algorithm would need a definition of interaction points between agents. An example interaction point is a player which decides to pass the ball to another player. This action can be evaluated on the trace of the other agent since the decision of passing the ball has been made, and may be better than to use the trace of the player itself, which may just be waiting around in the middle field after having passed the ball.

Open problems

We would like to have methods which diminish the variance of TD(λ)-updates for large values of λ. One way of doing this is to use variable λ. E.g. we can detect stable landmark states which have been visited so often, that their path to the goal has been optimized already and their value is quite stable. Then we can use large λ values before making a transition to them. We could also let the λ value depend on the standard deviation of some state value. It is a question whether and how much such methods improve TD(λ)'s abilities.

It is still unclear how we can extend the Team Q(λ) method to allow learning mostly from traces which are useful for improving the policy. For this, we would need to discard some traces, while keeping others. One option to do this is to only combine traces which have not been generated too long ago. Another possibility is to weigh traces according to their recency. An open question is whether using such more sophisticated methods can really significantly speed up multi-agent Q(λ) which uses separate traces.

A possible large problem of multi-agent RL is when traces connect in a 1-to-many way. E.g. if a mother is cooking for her children, and afterwards the children get a belly-ache, then we should be able to trace that back to an ingredient the mother used and punish using that ingredient or even buying it. The problem is that in this way, the action of cooking has multiple future paths, since we have to take all her children into account. Therefore the future branches and probabilities do not sum to 1, which makes credit assignment and computing value functions really hard. Thus, another open problem is to find neat methods which can combine such multiple futures into bounded value functions.

8.1.3 Model-based RL

Discussion

In Chapter 4, we described model-based RL approaches and introduced a novel implementation of the prioritized sweeping (PS) algorithm (Moore and Atkeson, 1993) which is more precise in calculating priorities for making different updates than the original PS. Experimental results on some maze problems showed that model-based RL significantly outperformed direct RL. Furthermore the new PS method is more reliable and leads to better final performances than the previous PS.

Open questions

The experimental result may not necessarily hold for other problems containing many possible outgoing transitions from each state. The price our PS has to pay for its exactness is that its computational efforts depend on the number of successor states, whereas the previous PS method is not. Therefore, it is computationally more expensive. Still, for problems with many successors we expect that the old PS will have much more problems in really making the most useful updates, since priorities are only based on a single outgoing transition.

Another question is whether even faster management methods can be constructed. Although this seems difficult, different methods could use other criteria or be more careful in updating states which leave the relative ordering of state values and therefore the policy unchanged.

Limitations

One problem of model-based RL, is that it heavily relies on the Markov assumption. TD(λ) approaches can much better evaluate a policy in case the Markov assumption is violated, since in case $\lambda = 1$, we just test the policy according to Monte Carlo sampling. One way for extending world models is to use TD(λ) world models which combine TD(λ) with model-based learning. Such a model may be useful for learning from learn term consequences of actions, although there may be many successor states so that it is uncertain whether we can efficiently use it for problems involving many states.

8.1.4 Exploration

Discussion

We described novel directed (information based) exploration methods which use a predefined exploration reward function for learning where to explore. Exploration reward functions are defined by the human engineer and determine how interesting a particular novel experience for an agent is. Examples of such reward functions assign penalties to state/actions which have occurred frequently or recently. Using such reward functions, the agent learns an exploration value function which allows the agent to select actions based on long term information gain. We used model-based RL for learning the exploration value functions and found large advantages compared to undirected, randomized exploration methods. Methods which only select actions according to information gain have as disadvantage that they do not take immediate problem specific rewards into account for selecting actions. To properly address the exploration/exploitation dilemma, we should only select actions which have some significant probability of belonging to the optimal policy. Therefore, we constructed model-based interval estimation (MBIE) which takes second order statistics into account for selecting actions. Since MBIE relies on initial statistics, we combined it with the directed exploration techniques discussed above. Experiments showed that MBIE was able to improve the directed model-based exploration methods in finding near-optimal policies, while collecting a much higher reward-sum during the training phase.

Open problems

There are still some unresolved problems in exploration. One problem is how to handle large or continuous state spaces for which we are never able to visit all states. For such problems we have to use function approximators and learn exploration functions. MBIE could then again be implemented by keeping track of the variance of all adjustable parameters of the chosen function approximator.

Another problem is how to explore efficiently in non-stationary environments. We could use the recency reward rule for such environments, but how could we use the model effectively? Clearly, we should discount early experiences, but it is in general difficult to set such discounting parameters. Furthermore, if the environment is partly changing, we should be able to find out how the environment is changing and thus our exploration policy should be tuned towards discovering specific changes of the environment.

Another problem in exploration is that in some problems it may be very difficult to achieve any reward at all. E.g. if only very few action sequences lead to the goal state and all others lead to some other terminal state, then how could we find the goal at all? We can use exhaustive search, trying out all action sequences, but there could be exponentially many of them! It may be that the only possibility is to use *a priori* goal-directed knowledge in order to be able to efficiently find the goal. One possibility is to implement a behavior which is guaranteed to find the goal, although it will most probably not find a very efficient solution. Such a behavior could be used to generate interesting initial experiences which can then be used to improve the policy. This process of designing and refining is also referred to as shaping (Dorigo and Colombetti, 1997).

8.1.5 POMDPs

Discussion

We described POMDPs in Chapter 6 and introduced a new algorithm, HQ-learning, for quickly solving some large deterministic POMDPs. HQ-learning is based on decomposing a problem into a sequence of reactive policy problems which are solvable by policies mapping observations to actions. Different policies are represented by different agents. Each agent learns HQ-values which determine for which observations control will be transferred to the next agent. HQ's memory is solely embodied in a pointer which determines which agent is active at the current time step. By using such a minimal representation for the short-term memory, HQ can quickly learn how to use it.

Limitations

HQ learning is designed for deterministic POMDPs featuring fixed start states. However, HQ could also be used in combination with map learning algorithms. E.g. we can learn maps for different regions, where transitions between regions are determined by agent transitions. We could even go beyond this and hierarchically learn a map where at the lowest level we have detailed observation or state transitions and at higher levels we have transitions between regions. The advantage of such an approach is that hierarchical planning becomes possible for solving large problems.

HQ-learning is a kind of hierarchical decomposition method, but does not learn complete policy-trees. It could be extended to allow for storing a whole tree containing policy nodes (instead of singular actions), and observation sequences as transitions. Then, a policy is followed for some time, after which a transition is made to another policy etc. In this way, ordinary policy trees of the kind described in (Kaelbling et al, 1995) may be compressed by allowing for collapsing multiple subsequent branches (or entire subtrees) inside a single reactive policy.

HQ suffered a lot from the limited exploration capabilities. In many trials, no solution was found at all, since the policy was generating cycles in state space. If, however, we allowed our undirected exploration methods to use more randomized actions, we continually execute a random walk behavior and do not really test the current policy. Such a random walk is not a very efficient exploration technique, and neither does it make learning the policy easier. Therefore we should construct directed exploration methods for POMDPs, which is difficult however, since the input used by the policy is usually highly ambiguous. A way to overcome this is to use more sophisticated representations for the internal state such as belief states, but these make policy construction hard in their own ways.

Open questions

A question remains whether we should not use different methods used for combinatorial optimization for solving POMDPs, since POMDPs are hard problems. E.g. we could use genetic algorithms (Holland, 1975), tabu search (Glover and Laguna, 1997) or the ant colony system (Dorigo, Maniezzo and Colorni, 1996; Dorigo and Gambardella, 1997; Gambardella, Taillard and Dorigo, 1997). Although currently multiple researchers start applying RL for combinatorial combination problems (e.g. Boyan, 1997), a real comparison between them would be very interesting. The ant colony system is in effect a RL method and is very successful solving a wide range of tasks (Dorigo and Gambardella, 1997). Since general RL methods are very good in dealing with stochasticity, they may finally outperform methods such as tabu search for dealing with complex stochastic problems. RL makes small changes which "on average" improve the policy. Local (greedy) search methods try out all changes and keep the best, but for stochastic problems they need many evaluations for testing which swap-move will result in the largest improvement.

8.1.6 Function Approximation

Discussion

We summarized the basics of a number of function approximators and introduced a new variant of the neural gas algorithm. We also described a novel combination of CMACs and world models and showed that CMAC-models were able to outperform a number of other methods (including the combination of Q(λ) with linear networks, neural gas, CMACs, and an evolutionary method called PIPE (Sałustowicz and Schmidhuber, 1997) at learning to play soccer. We noted that for efficient RL, the function approximator should be stable, thereby allowing a consistent policy evaluation. CMAC models are very stable and easily trained which makes them a powerful candidate for continuous state or action RL.

Limitations

CMAC models rely on an *a priori* defined structure which determines which relations can be learned between input features. It would be nice to learn the structure by reinforcement learning. E.g. HQ-learning could be combined with CMACs models, where HQ-learning tries structural changes and keeps track of the average performance of the function approximator given that particular input combinations are used. Thus, HQ can learn the structure which leads to the largest reinforcement intake.

We can also incrementally specialize filters if their predictive ability is small or if the variance of the Q-values of its cells is large. The advantage of learning a structure of a model such as CMACs is that it is easy to add a filter without changing the overall policy. This is in contrast with local models such as neural gas where adding neurons can result in large policy changes.

Other algorithms to learn structures are offline methods such as GA's and tabu search. Their problem, however, is that they need to test multiple different structures from which only few trained structures are used. This results in throwing away a lot of information. HQ-learning or an incremental approach can learn online and improve a single structure.

8.2 Final Remarks

Reinforcement learning can be used as a tool for designing many different intelligent computer systems — it can be used for improving simulation models, for designing autonomous vehicles, for controlling space missions, and for controlling forest fires. Especially for multi-agent environments, a lot of research needs to be done to make robust and well organized systems. Such environments often consist of multiple interacting elements which are adapted together to optimize a specific group criterion determining the desired group's behavior. RL can be used very well for solving such problems and in fact we want to use RL for controlling a team of forest fire fighting agents which have the goal to minimize the costs of forest fire fighting operations (Wiering and Dorigo, 1998).

The largest problem of applying reinforcement learning may ultimately be that constructing a reward function can be very hard. First of all, it may be difficult to optimize multiple evaluation criteria together, since it is difficult to weigh them. E.g. consider a driver of a racing-car who has the goal to minimize the risk of endangering his life while still maximizing the probability of winning the race. Which risk should he take or how much worth is his life?

The second problem of reward functions is that in the real world different agents may try to solve their own task, but doing this, they may interfere with each other. If we consider a large artificial world with many tasks performed by different RL agents, it is difficult to *a priori* construct reward functions which take the interactions into account. If we would just give each agent their own reward function, they would not care if they would be an obstacle for other agents performing their tasks and that may ultimately lead to a competition between agents. E.g. one agent which is hindered by another one, may learn to make it impossible for the other agent to become an obstacle by closing him in first. This is clearly not what we want. One solution is to weigh all tasks and design a group reward criterion, but this makes learning tasks for each agent much harder due to the agent credit assignment problem — how can we assign credit to each individual agent based on the group's outcome.

Thus, for complex multi-agent environments, there remain two options: organize the agents with clearly defined priorities and task-dependencies or let the agent evaluate each other and pass their evaluations as reward signals to other agents. For the first option, priorities could be swapped and the best organization maximizing the overall (colony's) reward kept. This could result in a set of complete team strategies or a set of interaction protocols which influence each agent's perception of the world state. Agents would not be able to change that organization.

For the second method, agent take social reward signals into account next to their task-oriented reward function. If we are unable to specify such (social) reward functions, agents should be able to learn ethical reward functions which improve the group's behavior. Learning such reward functions may be very difficult, but it opens up a huge variety of possibilities. Furthermore, if the task reward functions are well specified, slowly adapting the social reward functions based on the group's evaluation may only help.

Since the real world consists of many agents which adapt themselves according to the rewards they get, RL may be an important tool for studying what kind of "intelligent environments" work best. There are so many ways, so that we conclude by stating that there is no way of knowing which role RL will play in the future. Only time can tell...

Appendix A

Markov Processes

A Markov process (MP) is a model for capturing the probabilistic evolution of a system. A Markov process is used to model time sequences of discrete or continuous random variables (states) which satisfy the Markov property. The Markov property states that the transition to a new state only depends on the current state. Markov chains are a kind of discrete time Markov processes which have a finite set of states. The whole dynamics of the system is governed by the initial state probability distribution and the probabilistic transition function. For simplicity we consider Markov chains which consist of:

1. A discrete time counter $t = 1, 2, \ldots, T$, where T denotes the length of the process and may be infinite.

2. A set of states $S = \{S_1, S_2, \ldots, S_N\}$, where the integer N denotes the number of states. The active state at time t is denoted as $s_t = j$, with $j \in S$.

3. A probabilistic transition matrix P which determines the probability of the active state s_{t+1} at the next time step given s_t. The conditional probabilistic transition function $P_t(s_{t+1} = j|s_t = i)$ denotes the probability of making a transition to state j from state i at time t. We consider *stationary* probabilistic transition functions, which have the property that they are time independent. Therefore we will drop the time index from P_t and simply write P.

4. A probability distribution over initial states : o_1, where $o_1(j)$ denotes the probability that the sequence starts in state j.

We will use matrix notation for denoting the probabilistic transition function. The stochastic transition matrix P consisting of entries P_{ij} denotes the probability of making a transition from state i to state j:

$$P_{ij} = P(s_{t+1} = j|s_t = i).$$

The stochastic matrices we are interested in, have the following properties:

1. All $P_{ij} \geq 0$

2. $\forall i \in S$, $\sum_j P_{ij} \leq 1$.

Note that most authors require the last sum to be equal to 1. However, with our definition we allow the process to stop at some point with some probability. This will be useful in our study of Markov chains with terminal states.

A.1 Markov Property

For a Markov chain, the transition matrix is independent of previous states and depends only on the current state, that is for each $t = 1, 2, 3, \ldots$

$$p(s_{t+1} = j | s_t = i) = p(s_{t+1} = j | s_t = i, s_{t-1}, \ldots, s_1)$$

When this is true, we say that the *Markov property* holds.

A.2 Generating a State Sequence

The Markov chain is now the finite state process which is described by the tuple (S, P, o_1). Simulating Markov chains is fairly simple: first we select an initial state s_1 according to the probability distribution over initial states o_1, and then we just select a state s_{t+1} given s_t according to the probabilities $P(s_{t+1}|s_t)$. The resulting generated sequence of states is often called a state-trajectory. Each state-trajectory $H = (s_1, s_2, \ldots, s_n)$, has a specific probability which can be computed by:

$$P(H) = o_1(s_1) \prod_{i=2}^{n} P(s_i | s_{i-1})$$

If we could observe a huge amount of sufficiently long [1] state-trajectories like the one above, we would be able to compute an approximation to the initial state probabilities and the state transition function by counting how often initial states or state transitions have occurred and then averaging them.

A.3 State Occupancy Probabilities

We are interested in predicting the future given a current active state and our model of the Markov process. However, we cannot just predict a single state-trajectory, since each one of them may have a tiny probability of really occurring. Instead of state-trajectories, we predict the probability distribution that each state is active at a specific future time step. These probabilities are also called *state occupancy probabilities.* We will denote the state occupancy probabilities at time t by the vector $o_t = (p(s_t = S_1) \ldots p(s_t = S_n))^T$. Where o^T denotes the transpose of o.

The dynamics of the state occupancy probabilities over the chain are resulting from recursively applying the following equation:

$$o_{t+1}^T = o_t^T P$$

When we want to predict the future, we cannot be sure in which state s_t the process is at any time t, and therefore the uncertainty may be quite high. However, when it is possible to use intermediate results, e.g. when we can observe the state at a particular time-step t, this is useful for calculating o_{t+1}. The knowledge of being in a particular state s_t, helps to diminish the prediction error over the next state occupancy probabilities o_{t+1}, but does not always decrease the entropy of o_{t+1}. E.g. look at Figure A.1: we may have a case in which a state j

[1]We consider regular Markov chains, which means that there exists a number L so that all states are connected by a path of length at most L.

goes to two successors l and m with probability 0.5 each. If at time $t-1$, we know $s_{t-1} = i$, we may have a transition to $s_t = j$ or $s_t = k$ (both probability 0.5) where k goes to state m with probability 1.0. Then the entropy of the predicted state occupancy probabilities at time $t+1$ given that we are in state i at time $t-1$ is: $-0.75 log 0.75 - 0.25 log 0.25 = 0.81$ bits. After we know that we have stepped to state j at time t, and we predict again, the entropy is $-0.5 log(0.5) - 0.5 log(0.5) = 1$ bit. Thus, additional information may make our prediction about the future state less certain.

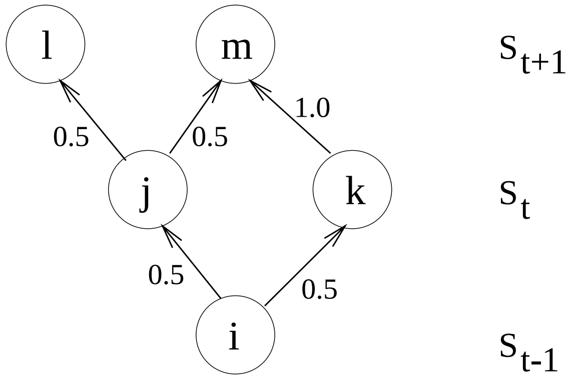


Figure A.1: *A Markov chain where entropy is not monotonically increasing.*

When we do not know $p(s_{t+k} = j)$, but we know s_t, we can compute the state occupancy probability vector o_{t+k} which stores the probability that the system is in state i (for $i = 1, \ldots, N$) at time-step $t+k$ as follows:

$$p(s_{t+k} = j | s_t = i) = P^k_{ij}$$

Where P^k is the k-step transition matrix. We may also calculate this by using:

$$p(s_{t+k} = j | s_t = i) = \sum_m P^l_{im} P^{k-l}_{mj}$$

with $0 \leq l \leq k$ which is the Chapman-Kolmogorov equation. The logic behind this equation is that the probability of going from one state i to another state j in k steps is equal to the probability of going from i to each possible in-between state m in l steps (probability P^l_{im}) and then going from m to j in $k-l$ steps (with probability P^{k-l}_{mj}). Note that the path which is taken from i to j does not matter due to the Markov property.

In particular, when k is a power of 2, we may use the following equation which uses recursion to efficiently calculate the probability:

$$p(s_{t+k} = j | s_t = i) = (P^{\frac{k}{2}}_{ij})^2$$

Given o_t, we can calculate o_{t+k} by:

$$o^T_{t+k} = o^T_t P^k$$

A.4 Stationary Distribution

The stationary (steady-state) distribution x has the following property: $x^T = x^T P$. This means that the probabilities will not change anymore by looking one step further in time, i.e. the dynamics of the state occupancy probabilities becomes 0 (a fixed point has been reached). We can find x by solving $x^T = x^T P$ directly (by iteration), or as follows:

$$x^T = \vec{1}^T (I + ONE - P)^{-1},$$

where ONE is a matrix containing a 1 on each place, and $\vec{1}$ is a vector containing only 1's. I is the identity matrix. (Resnick, 1992).

A.5 Properties of Markov Chains

There are some properties of Markov chains which are useful for classifying states, see also (Resnick, 1992; Bertsekas and Tsitsiklis, 1996).

- When it is possible to go from state i to state j, i.e. $\exists k$ so that $P^k_{ij} > 0$, we say that j is *accessible* from i.

- If for two states i and j, it holds that i is accessible from j and j is accessible from i, (that means that there is a path from one to the other and v.v.) then we say that i and j *communicate.*

- A Markov chain is *irreducible*, if there is a path between each pair of states, that is all states in S communicate with each other.

- When $P_{ii} = 1$ for some i, we say i is an *absorbing* state.

- When $P_{ij} = 0$ for all $j \in S$, we say that i is a *terminal state.* Note that this definition is different from that of most other authors who use the same definition for terminal as for absorbing states.

- When there exists a set of states $C \subset S$, with the property that for each $i \in C$, and each $j \in S - C$, $P^k_{ij} = 0$, for all k, and all states in C communicate we say that C is a *recurrent class.* This means that the process will not leave C, once it has entered a state $i \in C$.

- States that do not belong to any recurrent class are called *transient.* For transient states, we have $P^k_{ii} = 0$, as $k \to \infty$. That means that the probability of returning to a transient state goes to 0 as the number of steps after the visit goes to infinity, which means that $o_t(i) \to 0$ for $t \to \infty$.

- A chain is called *ergodic*, if it is irreducible, the complete set of states S is recurrent and a stationary distribution exists.

A.6 Counting Visits

Sometimes it is useful to know the average number of times the process will be in a particular state j starting in the current state i until T steps in the future. The expected number of visits K_{ij} of a state j starting in i until T steps in the future can be calculated by:

$$K_{ij} = \sum_{l=0}^{T} P_{ij}^{l}$$

For $T \to \infty$, we can calculate the complete counting matrix K with elements K_{ij} by:

$$K = \sum_{l=0}^{\infty} P^{l} = (I - P)^{-1},$$

where I is the identity matrix, and Q^{-1} denotes the inverse of Q. Note that in this case the process should terminate, since otherwise K_{ij} goes to ∞ for at least one i, j pair. Termination may be assured by including a terminal state which is accessible from all other states.

A.7 Examples

Example 1: Look at the following example. The set of states is $S = \{1, 2, 3\}$ and the probabilistic transition matrix P is given by:

$$\begin{vmatrix} 0.6 & 0.4 & 0.0 \\ 0.3 & 0.2 & 0.5 \\ 0.0 & 0.6 & 0.4 \end{vmatrix}$$

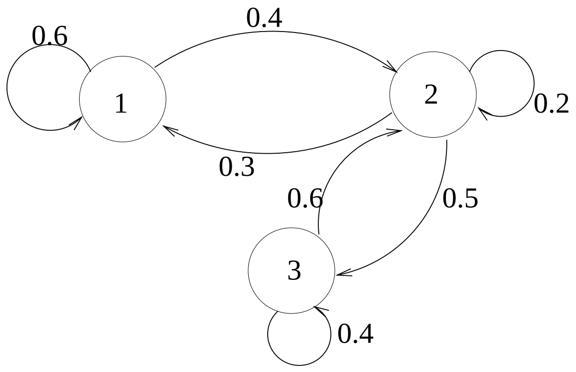


Figure A.2: *The Markov chain of Example 1.*

This Markov chain in shown in Figure A.2. The chain is irreducible since all states communicate. There are no transient states. The stationary distribution x can be calculated as follows:

$$\begin{array}{ccccccc} 0.6x_1 & + & 0.3x_2 & & & = & x_1 \\ 0.4x_1 & + & 0.2x_2 & + & 0.5x_3 & = & x_2 \\ 0.5x_2 & + & 0.4x_3 & & & = & x_3 \end{array}$$

From this follows: $x_1 = \frac{3}{4}x_2$ and $x_3 = \frac{5}{6}x_2$. Since $x_1 + x_2 + x_3 = 1$, we get $\frac{31}{12}x_2 = 1$, and it follows that $x_1 = \frac{9}{31}$, $x_2 = \frac{12}{31}$ and $x_3 = \frac{10}{31}$. The chain is ergodic. Note that (I - P) is singular, therefore we cannot compute the counting matrix for the infinite case (in this case all states will be visited infinitely many times).

Example 2: Now have a look at the following example. Again $S = \{1, 2, 3\}$. Now P is given by:

$$\begin{vmatrix} 0.6 & 0.4 & 0.0 \\ 0.0 & 0.8 & 0.2 \\ 0.0 & 0.4 & 0.6 \end{vmatrix}$$

State 1, is a transient state, since once the process has left state 1 it will not go back to it anymore. States 2 and 3 form a recurrent class. The stationary distribution is given by: $x^T = (0 \;\; \frac{2}{3} \;\; \frac{1}{3})$. Note that a transient state has probability 0 in the stationary distribution. The chain is also not ergodic.

Example 3: Now have a look at the following example. $S = \{1, 2, 3\}$, and P is given by:

$$\begin{vmatrix} 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.0 & 0.0 & 0.0 \end{vmatrix}$$

Here, state 3 is a terminal state. Since the terminal state is accessible from all states, this means that the process will always stop. This also implies that all non-terminal states are transient states. The chain is not ergodic. We can calculate the future visits matrix $K = (I - P)^{-1}$. This gives:

$$\begin{vmatrix} 5 & 5 & 1 \\ 2.5 & 5 & 1 \\ 0 & 0 & 1 \end{vmatrix}$$

We can see that the terminal state is reached one time from all states, and that the expected number of transitions between state 1 and 2 is bounded.

A.8 Markov Order

When we consider Markov chains, we make the requirement that the previous states are not allowed to have any influence on the current transition which is therefore solely based on the current state. When a transition matrix has this property, we call it memoryless. However, sometimes states are not uniquely perceived. Instead we may be in the posession of a partial description of a state. This partial state or observation may not always contain all information needed to find a perfect prediction of the subsequent state. In such cases previous states may be used to improve the prediction abilities. Therefore, we will shortly describe Markov chains where previous visits of states influence the current dynamics.

The **Markov order** mo of a process is defined as the minimal number of previous steps which influence the transition function. When $\forall i \in S$

$$p(s_{t+1} = i) = p(s_{t+1} = i | s_t, \ldots, s_1),$$

the Markov order is $mo = 0$ and we see that the current state and the previous states do not influence the next transition. This means that there is a fixed probability of visiting each state.

We will use a recursive definition for the Markov order. The Markov order is mo (for $mo > 0$) when it is not $mo - 1$ and when the following holds $\forall i \in S$:

$$p(s_{t+1} = i | s_t, \ldots, s_{t+1-mo}) = p(s_{t+1} = i | s_t, \ldots, s_1) \tag{A.1}$$

The equation says that no more than mo previous states can be important for determining the transition probabilities. There are two special cases, the first is when $mo = 1$, which we have already seen, since it is a requirement for Markov chains (the Markov property holds). When $mo = t$, and goes to ∞ as $t \to \infty$, we say it is a process of indefinite Markov order.

Dealing with processes with definite Markov order higher than one, which we will call higher order Markov processes, can be done as follows: we construct new states which are Cartesian products of the last mo steps. The new space of higher order states, contains N^{mo} elements (although usually only a small fraction of all elements do occur). In this new space, however, we can consider Markov chains of order 1.

A.9 Higher Order Transition Functions

When we consider Markov processes of higher order ($mo > 1$), and we want to predict the dynamics of the states, we have to use higher order states i^{mo} and mo^{th} order probabilistic transition functions. For this we introduce higher-order states i^{mo} which contain all information about the last mo steps (the history of the state trajectory). The state i^{mo} is defined as follows. Given: $s_t = j_0, s_{t-1} = j_1, s_{t-2} = j_2, \ldots, s_{t+1-mo} = j_{mo-1}$, we compute i^{mo} as follows:

$$i^{mo} = \sum_{k=0}^{mo-1} s_{t-k} N^{mo-k-1}. \tag{A.2}$$

That means that i^{mo} assigns a number to uniquely describe each different sequence of states. It is useful to number the (first order states) as follows: $S = \{0, 1, 2, \ldots, N-1\}$, which gives the following numbering for i^{mo} : $i^{mo} \in HS = \{0, 1, 2, \ldots, N^{mo} - 1\}$. HS is the resulting set of higher order states. Note that when $mo = 1$, $i^{mo} = s_t$ and $HS = S$.

For defining higher order transitions of order mo, we use:

$$p_{mo}(j^{mo} | i^{mo}) = p(s_{t+1} = j, s_t = j_0, \ldots s_{t+2-mo} = j_{mo-2} | s_t = j_0, \ldots, s_{t+1-mo} = j_{mo-1}),$$

with i^{mo} and j^{mo} defined by equation A.2.

$p_{mo}(j^{mo}|i^{mo})$ denotes the probability of going from i^{mo} to j^{mo} in 1 step, based on the last mo states which are covered in i^{mo}. We can again use matrix notations. We construct the higher order matrix HP with size $N^{mo} \times N^{mo}$ where we define

$$HP_{i^{mo} j^{mo}} = p_{mo}(j^{mo} | i^{mo}),$$

Note that each row of HP contains only n nonzero elements, since only the first state can be changed (and thus we could represent the matrix more efficiently). The Markov order is the longest sequence of previous states needed for determining the probability of all transitions from each possible history. For most states i^{mo}, the Markov order will be lower, which means that we do not have to know $p_{mo}(j^{mo}|i^{mo})$, but can rely on $p_l(j^l|i^l)$ with $l < mo$, and therefore $i^l < i^{mo}$. In such cases we can represent the transition function more compactly by a tree.

We can calculate $p^n_{mo}(j^{mo}|i^{mo})$ that is the probability of going from i^{mo} to j^{mo} in n steps using the previous mo steps as follows:

$$p^n_{mo}(j^{mo}|i^{mo}) = HP^n_{i^{mo}j^{mo}}$$

A.10 Examples

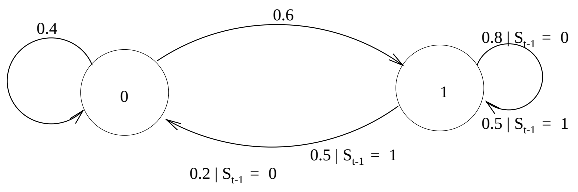


Figure A.3: *Second order Markov chain before introducing higher order states.*

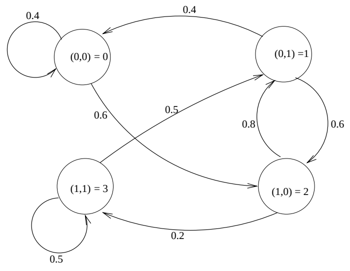


Figure A.4: *The resulting Markov chain of figure 2 after converting the chain to higher order states and transitions.*

Example 4: Given the 2^{th} order ($mo = 2$) Markov chain in Figure A.3. The states are $\{0, 1\}$ and in state 1 the next transition depends on the previous state. We construct the higher order states : $HS = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ or equivalently: $HS = \{0, 1, 2, 3\}$. HP is given by:

$From/To$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$(0,0)$	0.4	0	0.6	0
$(0,1)$	0.4	0	0.6	0
$(1,0)$	0	0.2	0	0.8
$(1,1)$	0	0.5	0	0.5

Figure A.4 shows the resulting Markov chain. The resulting higher order transition matrix is irreducible, does not contain absorbing states or transient states. The stationary occupancy probabilities are given by $x^T = (\frac{10}{64}\ \frac{15}{64}\ \frac{15}{64}\ \frac{24}{64})$. The chain is ergodic. Finally, for the stationary distribution over the states in Figure A.3, we have probability $p(0) = \frac{25}{64}$ that we observe a 0, and $p(1) = \frac{39}{64}$.

Example 5: Given the 2^{th} order Markov chain in Figure A.5, we can of course again define HS and HP in the same way, but now HP would be a 9×9 matrix. A better way in this case is to define HS with four states as follows (not all 9 states are needed) $HS = \{0, 1, 2, 3\}$, state 0 is the set of 2^{th} order states : $\{(0, 0), (0, 1)\}$, state 1 $= \{(1, 1), (1, 2)\}$, state 2 $= \{(2, 1), (2, 2)\}$, and finally state 3 $= \{(1, 0)\}$. Note that we split the original state 1, which needs the previous state for calculating its transition probabilities, into a set of two states $\{1, 3\}$. The resulting Markov chains is shown in Figure A.6. The transition probability matrix now looks as follows:

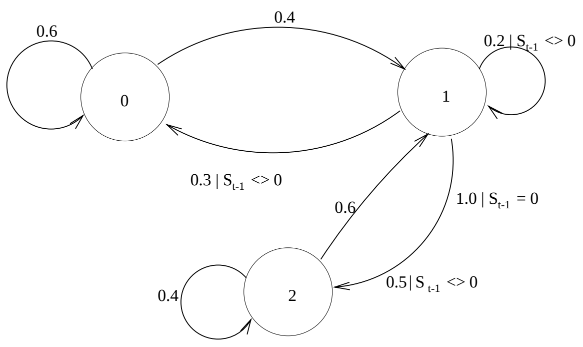


Figure A.5: *Example 5. Higher order Markov chain before introducing higher order states.*

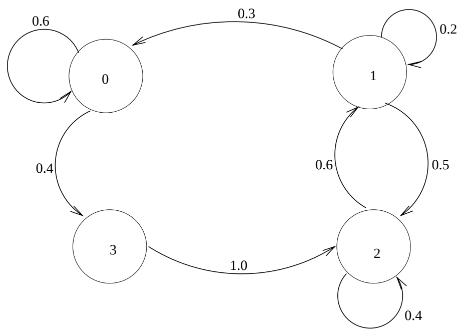


Figure A.6: *Markov chain with splitted higher order states.*

$From/To$	(0)	(1)	(2)	(3)
(0)	0.6	0	0	0.4
(1)	0.3	0.2	0.5	0
(2)	0	0.6	0.4	0
(3)	0	0	1.0	0

Note that this matrix is invertible (just check $Pv = 0$, which means that all $v_i = 0$). When we would have used the full 9×9 higher order matrix, then it would not have been invertible. This is because some rows would be the same. Therefore an invertible matrix contains all necessary information and we should not try to make matrices unnecessarily large by introducing redundant information (as was the case in Example 4).

Now we can calculate the stationary occupancy probabilities: $x = (\frac{45}{203}\ \frac{60}{203}\ \frac{80}{203}\ \frac{18}{203})^T$. From these we can calculate the probabilities that we will see each of the states 0, 1, and 2 when the process has settled down in the steady-state distribution by just summing up the stationary occupancy probabilities of the splitted state 1: $(\frac{45}{203}\ \frac{78}{203}\ \frac{80}{203})$.

Appendix B

Learning rate Annealing

Consider the following conditions on the learning rate α_t which depends on t. Here we use t to denote the number of visits of a specific state/action pair.

1. $\sum_{t=1}^{\infty} \alpha_t = \infty$
2. $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$

The reason for the first condition is that by updating with learning rate α_t, each (finite) distance between the optimal parameter-setting and the initial parameters can be overcome. The reason for the second condition is that the variance goes to zero, so that convergence in the limit will take place (otherwise our solution will be inside a ball with the size of the variance containing the optimal parameter-setting, but the true single-point solution will be unknown). We want to choose a function $f(t)$ and set $\alpha_t = f(t)$. In (Bertsekas and Tsitsiklis, 1996) $f(t) = \frac{1}{t}$ is proposed. Here we propose to use more general functions of type : $f(t) = \frac{1}{t^\beta}$, with $\beta > 0$, and will proof that for $\frac{1}{2} < \beta \leq 1$, the two conditions on the learning rate are satisfied.

The function is a step function, since a sum is used. In the following we use a continuous function in order to make it easy to compute β. Since there is a mismatch between the continuous function and the step function, we use two functions, one which returns a smaller value than the sum (the lower bound function), and another which returns a larger value (the higher bound function).

For the first condition we use a lower bound on the sum:

$$\begin{aligned}\sum_{t=1}^{\infty} \frac{1}{t^\beta} &\geq \int_{t=1}^{\infty} \frac{1}{t^\beta} dt \\ &= \left[\frac{t^{1-\beta}}{1-\beta}\right]_1^\infty \\ &= \infty\end{aligned}$$

From this (and from dealing with the special case $\beta = 1$ separately) follows : $\beta \leq 1$.

For the second condition we use a higher bound on the sum:

$$\sum_{t=1}^{\infty} \frac{1}{t^{2\beta}} \leq 1 + \int_{t=2}^{\infty} \frac{1}{(t-1)^{2\beta}} dt$$

$$\begin{aligned} &= 1 + \left[\frac{(t-1)^{1-2\beta}}{1-2\beta} \right]_2^{\infty} \\ &< \infty \end{aligned}$$

From this follows : $\beta > \frac{1}{2}$.

Hence we can use the following family of methods for decreasing the learning rate: $f(t) = \frac{1}{t^\beta}$ with $\frac{1}{2} < \beta \leq 1$. Since in general we want to choose the learning rate as high as possible, β should be chosen close to $\frac{1}{2}$. In practical simulations often constant learning rates are used (i.e. $\beta = 0$), but this may prohibit convergence.

Appendix C

Team Q(λ) Algorithm

We have extended the Fast Q(λ) algorithm to the multi-agent case. The complete Team Q(λ) algorithm will be described here. We will first introduce additional indices for the different agents (histories) and use the following variables:

K = Number of agents.

H_i = History list of agent i.

$l'_i(s,a)$ = Eligibility trace of agent i.

Δ_i = Global delta trace of agent i.

ϕ_i^t = Global eligibility trace of agent i.

$\delta_i(s,a)$ = Local delta trace for SAP (s,a) of agent i.

$self\_visited_i(s,a)$ = Boolean which is true if SAP (s,a) occurred in the trajectory of agent i.

$visited_i(s,a)$ = Last trial number in which SAP (s,a) occurred in the trajectory of agent j, where j may, or may not be equal to i. If it is not, there should be a connected path between SAP (s,a) to the trajectory of agent i.

$revisited_i(s,a)$ = Boolean which indicates whether SAP (s,a) has been visited multiple times by agent i.

First we adapt the previous procedures *Global Update* and *Local Update* to take into account that we have to specify the current agent. These procedures should always be used for the multi-agent Q(λ) where the value function is shared by the agents, independent of the choice whether we connect traces or not. The new *Local Update* simply adds the contributions from all agent trajectories to update a Q-value:

```
**Local Update($s_t, a_t$) :**
1) $D \leftarrow 0$
2) For $i = 1$ to $K$ Do
      2a) If $(visited_i(s_t, a_t) > 0)$
            2a.1) $M \leftarrow visited_i(s_t, a_t)$
            2a.2) $D \leftarrow D + (\Delta_i^M - \delta_i(s_t, a_t)) l'_i(s_t, a_t)$
            2a.3) $\delta_i(s_t, a_t) \leftarrow \Delta_i^N$
            2a.4) If $(M < N)$
                  2a.4.1) $l'_i(s_t, a_t) \leftarrow 0$
                  2a.4.2) $visited_i(s_t, a_t) \leftarrow N$
3) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_k(s_t, a_t) D$
```

For *Global Update* we first have to change the variable-indices to take the active agent (the agent which made the last step) into account (this is straightforward to do, so we do not show the new procedure here). In the procedure *Global Update's addendum*, we change the first lines (line 9, 9a and 9b), to make sure that we insert revisited states in the beginning of the history list. Here the union operator $\cup$ is implemented as an insertion at the head of a list.

```
**Global Update($s_t, a_t, i$) :**
9) If $(visited(s_t, a_t) = 0)$
      9a) $H_i \leftarrow H_i \cup (s_t, a_t)$
      9b) $visited_i(s_t, a_t) \leftarrow N$
      9c) $self\_visited_i(s_t, a_t) \leftarrow 1$
      9d) $revisited_i(s_t, a_t) \leftarrow 0$
9') Else
      9e) $H_i \leftarrow H_i / (s_t, a_t)$
      9f) $H_i \leftarrow H_i \cup (s_t, a_t)$
      9g) $self\_visited_i(s_t, a_t) \leftarrow 1$
      9h) $revisited_i(s_t, a_t) \leftarrow 1$
```

Finally, *Global addendum* should be updated so that we reset the variables $revisited_i(s, a)$ and $self\_visited_i(s, a)$ in lines 10a-3.3 and 10a-3.4.

Connecting traces algorithm. Now we have adapted the existing procedures for the value function sharing multi-agent case. We will call the multi-agent Q(λ) method without connecting traces: *Independent Q(λ)*. If we want to connect traces, all we have to do is to check if an agent executes a SAP which has been executed already by another agent and call the procedure *Connect* as shown below. Note that in our algorithm we compare whether the same *SAP* has occurred in two agent trajectories, another method would be to check whether an agent visits the same *state* as another agent. Although this would cause more links between trajectories, the second agent may have chosen a different action, since the action selected by the first agent turned out not to be very good, and therefore learning on that future trajectory may be more harmful than if we only learn if the same SAP has occurred.

In the procedure *Connect*, we have the meeting point (s, a) and want to let SAPs from the trajectory of agent i learn on the future dynamics of trajectory $Goal_A$. The variables $Delta$ and $trace2$ store local values of (s, a) for trajectory $Goal_A$. $Trace1$ is the eligibility trace of the current SAP $Prev$ which we insert.

We traverse the trajectory of agent i by going backwards in the chain. The variables S_{Prev} and A_{prev} denote the state and action of the history element $Prev$ of agent i. Whenever SAPs ($Prev$) from i have not been visited by the trajectory called $Goal_A$, we insert them in the list and update the trial, eligibility traces, and δ-values. After inserting 1 SAP we take its predecessor and check in which agent trajectories the preceding SAP occurred, so that they may also start learning on the trajectory of $Goal_A$. This is implemented by the lines starting from (3f) below. Essentially it implements a recursive call which stores SAPs from all trajectories in the history list of $Goal_A$ if there is a directed path to SAP (s, a) to the list of $Goal_A$. If we "hop" over to another trajectory, we recompute the trace variables. Finally line 4 implements the case in which the SAP $Prev$ already has a non-zero eligibility trace on the trajectory of $Goal_A$, but since it has been revisited, its predecessor may still not have eligibility.

```
Connect($s, a, Prev, i, Goal_A, delta, trace1, trace2, trial$) :
1) If ($Prev$ = NULL) return
2) $l_{new} \leftarrow trace2(l'_i(s_{Prev}, a_{Prev})/trace1)$
3) If (($visited_{Goal_A}(s_{Prev}, a_{Prev})$ = 0) AND $self\_visited_i(s_{Prev}, a_{Prev})$ AND
       ($l_{new} > \epsilon_m$) AND $visited_i(s_{Prev}, a_{Prev} = trial)$
    3a) $visited_{Goal_A}(s_{Prev}, a_{Prev}) \leftarrow visited_{Goal_A}(s, a)$
    3b) $l'_{Goal_A}(s_{Prev}, a_{Prev}) \leftarrow l_{new}$
    3c) $\delta_{Goal_A}(s_{Prev}, a_{Prev}) \leftarrow delta$
    3d) $H_{Goal_A} \leftarrow insert\_at\_end\_of\_list(H_{Goal_A}, (s_{Prev}, a_{Prev}))$
    3e) $visited_{Goal_A}(s_{Prev}, a_{Prev}) \leftarrow visited_{Goal_A}(s, a)$
    3f) For $j = 1$ to $K$
      3f.1) If (($j <> Goal_A$) AND ($self\_visited_j(s_{prev}, a_{prev})$))
        3f.1.a) If ($j <> i$)
          3f.1.a.1) $new\_trace1 \leftarrow l'_j(s_{prev}, a_{prev})$
          3f.1.a.2) $new\_trace2 \leftarrow l_{new}$
          3f.1.a.3) $trial \leftarrow visited_j(s_{prev}, a_{prev})$
        3f.1.b) else
          3f.1.b.1) $new\_trace1 \leftarrow trace1$
          3f.1.b.2) $new\_trace2 \leftarrow trace2$
          3f.1.b.3) $trial \leftarrow visited_i(s_{prev}, a_{prev})$
        3f.1.c) $Prev \leftarrow Prev\_history\_elt(H_j, (s_{Prev}, a_{Prev}))$
        3f.1.d) If (($trace1 > \epsilon_m$) AND ($trace2 > \epsilon_m$))
          3f.1.d.1) $Connect(s, a, Prev, j, Goal_A, delta, new\_trace1, new\_trace2, trial)$
4) Else If ($revisited_i(s_{prev}, a_{prev})$ AND $self\_visited_i(s_{prev}, a_{prev})$)
    4a) $Prev \leftarrow Prev\_history\_elt(H_i, (s_{Prev}, a_{Prev}))$
    4b) $Connect(s, a, Prev, i, Goal_A, delta, trace1, trace2, trial)$
```

Finally, we have to call the procedure connect. This we do if we observe that the current SAP (s, a) has also occurred in the trajectory of another agent. There are two ways of connecting trajectories: (1) SAPs are inserted in the history list of the active agent so that they will start learning on what the active agent does in the future. This is done in the first part of the procedure below (2a). (2) SAPs are inserted in the history list of the other agent so that SAPs which occurred in the list of the active agent will start learning on the trajectory of the other agent (3a). Below, we present the complete *Team Global Update* procedure. It first calls *Global Update* and then tries to connect trajectories. We call the active agent *Active*.

```
Team Global Update($s_t, a_t, Active$) :
1) $Global\ Update(s_t, a_t, Active)$
2) For $i = 1$ To $K$
    2a) If (($i <> Active$) AND ($self\_visited_i(s, a)$))
       2a.1) $Prev \leftarrow Prev\_history\_elt(H_i(s, a))$
       2a.2) $delta \leftarrow \delta_{Active}(s, a)$
       2a.3) $trace1 \leftarrow l'_i(s, a)$
       2a.4) $trace2 \leftarrow l'_{Active}(s, a)$
       2a.5) $trial \leftarrow visited_i(s, a)$
       2a.6) if (($trace1 > \epsilon_m$) AND ($trace2 > \epsilon_m$))
          2a.6.1) $Connect(s, a, Prev, i, Active, delta, trace1, trace2, trial)$
3) For $i = 1$ To $K$
    3a) If (($i <> Active$) AND ($self\_visited_i(s, a)$))
       3a.1) $Prev \leftarrow Prev\_history\_elt(H_{Active}(s, a))$
       3a.2) $delta \leftarrow \delta_i(s, a)$
       3a.3) $trace1 \leftarrow l'_{Active}(s, a)$
       3a.4) $trace2 \leftarrow l'_i(s, a)$
       3a.5) $trial \leftarrow visited_{Active}(s, a)$
       3a.6) if (($trace1 > \epsilon_m$) AND ($trace2 > \epsilon_m$))
          3a.6.1) $Connect(s, a, Prev, Active, i, delta, trace1, trace2, trial)$
```

Appendix D

PS for Function Approximators

D.1 PS for Neural Gas

The model-based update of the Q-value $Q(n_i, a)$, **Q-update(n_i, a)** looks as follows:

$$Q(n_i, a) \leftarrow \sum_j P_{n_i j}(a)(R(n_i, a, j) + \gamma V(j)),$$

where $P_{n_i j}(a) = P(j|n_i, a)$. The details of our PS look as follows:

```
Our-Prioritized-Sweeping-NG($x$):
1) Compute active neurons:  $c_1, \ldots, c_z$
2) For $k$ = 1 to $z$ do:
      2a) Update $c_k$ --- $\forall a$ do:
            2a.1) Q-update($c_k, a$)
      2b) Set $|\Delta(c_k)|$ to $\infty$
      2c) Promote $c_k$ to top of queue
3) While ($n < U_{max}$ & queue $\neq$ nil)
      3a) Remove top $c_k$ from the queue
      3b) $\Delta(c_k) \leftarrow 0$
      3c) $\forall$ Predecessor neurons $i$ of $c_k$ do:
            3c.1) $V'(i) \leftarrow V(i)$
            3c.2) $\forall a$ do:
                  3c.2.1) Q-update($i, a$)
            3c.3) $V(i) \leftarrow \max_a Q(i, a)$
            3c.4) $\Delta(i) \leftarrow \Delta(i) + V(i) - V'(i)$
            3c.5) If $|\Delta(i)| > \epsilon$
                  3c.5.1) Insert $i$ at priority $|\Delta(i)|$
      3d) $n \leftarrow n + 1$
4) Empty queue, but keep $\Delta(i)$ values
```

Here U_{max} is the maximal number of updates to be performed per update-sweep. The parameter $\epsilon \in I\!R^+$ controls update accuracy.

D.2 PS for CMACs

The model-based update of the Q-value $Q_k(c, a)$, **Q-update(k, c, a)** looks as follows:

$$Q_k(c,a) \leftarrow \sum_j P^k_{cj}(a)(R_k(c,a,j) + \gamma V_k(j)),$$

where $P^k_{cj}(a) = P_k(j|c,a)$. The details of our PS look as follows:

Our-Prioritized-Sweeping-CMAC(x):

```
1) Compute active cells:  $f_1, \ldots, f_z$
2) For $k$ = 1 to $z$ do:
      2a) Update $f_k$ --- $\forall a$ do:
            2a.1) Q-update($k, f_k, a$)
      2b) Set $|\Delta_k(f_k)|$ to $\infty$
      2c) Promote ($k, f_k$) to top of queue
3) While ($n < U_{max}$ & queue $\neq$ nil)
      3a) Remove top ($k,c$) from the queue
      3b) $\Delta_k(c) \leftarrow 0$
      3c) $\forall$ Predecessor cells $k,i$ of $k,c$ do:
            3c.1) $V'_k(i) \leftarrow V_k(i)$
            3c.2) $\forall a$ do:
                  3c.2.1) Q-update($k, i, a$)
            3c.3) $V_k(i) \leftarrow \max_a Q_k(i,a)$
            3c.4) $\Delta_k(i) \leftarrow \Delta_k(i) + V_k(i) - V'_k(i)$
            3c.5) If $|\Delta_k(i)| > \epsilon$
                  3c.5.1) Insert $i$ at priority $|\Delta_k(i)|$
      3d) $n \leftarrow n + 1$
4) Empty queue, but keep $\Delta_k(i)$ values
```

D.3 Non-Pessimistic Value Functions

To compute non-pessimistic value functions for multi-agent CMACs, we decrease the probability of the worst transition from each filter/cell/action and then renormalize the other probabilities. Then we use the adjusted probabilities to compute the Q-functions. In the following $C^k_{ij}(a)$ counts the number of transitions of cell i to j in filter k after selecting action a and $C^k_i(a)$ counts the number of times action a was selected and cell i of filter k was activated. We obtain $\hat{P}^k_{ij}(a)$, the estimated transition probability, by dividing them. Then we substitute the following for **Q-update(k, c, a)**:

```
**Compute Non-Pessimistic Q-value**$(k, i, a)$:
1) $m \leftarrow \arg\min_j \{R_k(i,a,j) + \gamma V_k(j)\}$
2) $n \leftarrow C_i^k(a)$
3) $P \leftarrow \hat{P}_{im}^k(a)$
4) $P_{im}^k(a) \leftarrow \frac{(P - \frac{z_\alpha^2}{2n} + \frac{z_\alpha}{\sqrt{n}}\sqrt{P(1-P) + \frac{z_\alpha^2}{4n}})}{1 + \frac{z_\alpha^2}{n}}$
5) $\Delta_P \leftarrow P_{im}^k(a) - \hat{P}_{im}^k(a)$
6) $\forall j \neq m$
        6a) $P_{ij}^k(a) \leftarrow \hat{P}_{ij}^k(a) - \frac{\Delta_P C_{ij}^k(a)}{C_i^k(a) - C_{im}^k(a)}$
7) **Q-update**$(k, i, a)$
```

The variable z_α which determines the step size for decreasing worst transition probabilities. To select the worst transition in step 1, we only compare existing transitions (we check whether $\hat{P}_{ij}(a) > 0$ holds). See Figure D.1 for a plot of the function. Note that if there is only one transition for a given filter/cell/action triplet then there will not be any renormalization. Hence the "probabilities" may not sum up to 1. Consequentially , if some filter/cell/action with deterministic dynamics has not occurred frequently then it will contribute just a comparatively small Q-value and thus have less impact on the computation of the overall Q-value.

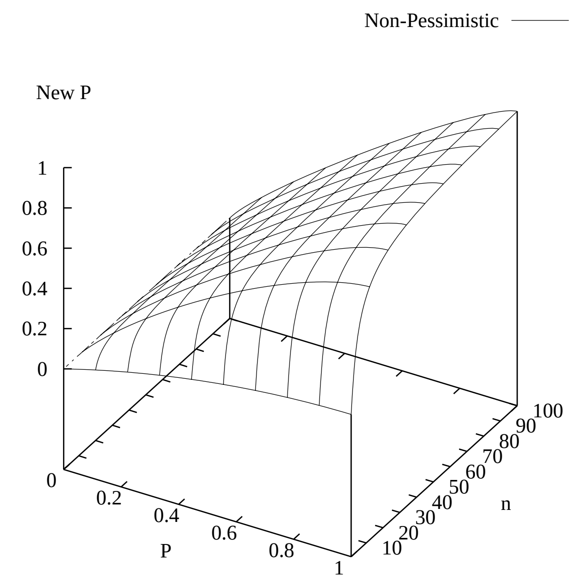


Figure D.1: *The non-pessimistic function (new P) which decreases the probability of the worst transition decreased as a function of the (maximum likelihood) probability P and the number of occurrences n.*

Bibliography

Albus, J. S. (1975a). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Dynamic Systems, Measurement and Control*, pages 220–227.

Albus, J. S. (1975b). A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61.

Asada, M., Uchibe, E., Noda, S., Tawaratsumida, S., and Hosoda, K. (1994). A vision-based reinforcement learning for coordination of soccer playing behaviors. In *Proceedings of AAAI-94 Workshop on AI and A-life and Entertainment*, pages 16–21.

Atkeson, C. G., Schaal, S. A., and Moore, A. W. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11:11–73.

Axelrod, R. (1984). *The evolution of cooperation.* Basic Books, New York, NY.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 30–37. Morgan Kaufmann Publishers, San Francisco, CA.

Baluja, S. (1994). Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University.

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846.

Baum, E. (1989). A proposal for more powerful learning algortihms. *Neural Computation*, 1(2):201–207.

Baxter, J., Tridgell, A., and Weaver, L. (1997). Knightcap: A chess program that learns by combining TD(λ) with minimax search. Technical report, Australian National University, Canberra.

Bayse, K., Dean, T., and Vitter, J. (1997). Coping with uncertainty in map learning. *Machine Learning*, 29(1):65–88.

Bell, A. and Sejnowski, T. (1998). The "independent components" of natural scenes are edge filters. *Vision Research.* To appear.

Bellman, R. (1961). *Adaptive Control Processes.* Princeton University Press.

Berliner, H. (1977). Experiences in evaluation with BKG - a program that plays backgammon. In *Proceedings of IJCAI*, pages 428–433.

Berry, D. and Fristedt, B. (1985). *Bandit Problems: sequential allocation of experiments.* Chapman and Hall, London/New York.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-dynamic Programming.* Athena Scientific, Belmont, MA.

Boutilier, C. and Poole, D. (1996). Computing optimal policies for partially observable decision processes using compact representations. In *AAAI-1996: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1168–1175, Portland, OR.

Box, G., Jenkins, G. M., and Reinsel, H. C. (1994). *Time series analysis: forecasting and control.* Prentice Hall.

Boyan, J. A. (1992). Modular neural networks for learning context-dependent game strategies. Master's thesis, University of Chicago.

Boyan, J. A. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 369–376. MIT Press, Cambridge MA.

Boyan, J. A. and Moore, A. W. (1997). Using prediction to improve combinatorial optimization search. In *Proceedings of the Sixth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, page 14.

Bradtke, S. J. and Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57.

Caironi, P. V. C. and Dorigo, M. (1994). Training Q-agents. Technical Report IRIDIA-94-14, Université Libre de Bruxelles.

Campos, L. M. D., Huete, J. P., and Moral, S. (1994). Probability intervals: A tool for uncertain reasoning. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2 (2):167–196.

Cassandra, A. R. (1998). *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, Brown University, Providence, RI.

Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (April 1994). Acting optimally in partially observable stochastic domains. Technical Report CS-94-20, Brown University, Providence RI.

Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 726–731. Morgan Kaufman.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 183–188. AAAI Press, San Jose, California.

Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning. *Journal on Artificial Intelligence*, 2:287–318.

Cliff, D. and Ross, S. (1994). Adding temporary memory to ZCS. *Adaptive Behavior*, 3:101–150.

Cohn, D. A. (1994). Neural network exploration using optimal experiment design. In Cowan, J., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 679–686. San Mateo, CA: Morgan Kaufmann.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, Hillsdale NJ. Lawrence Erlbaum Associates.

Crites, R. and Barto, A. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA. MIT Press.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Math. Control Signals Systems*, 2:303–314.

D'Ambrosio, B. (1989). POMDP learning using qualitative belief spaces. Technical report, Oregon State University, Corvallis.

Davies, S., Ng, A. Y., and Moore, A. W. (1998). Applying online search techniques to continuous-state reinforcement learning. In *Proceedings of the AAAI'98.*

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

Dayan, P. and Hinton, G. (1993). Feudal reinforcement learning. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 5*, pages 271–278. San Mateo, CA: Morgan Kaufmann.

Dayan, P. and Sejnowski, T. (1994). TD(λ): Convergence with probability 1. *Machine Learning*, 14:295–301.

Dayan, P. and Sejnowski, T. J. (1996). Exploration bonuses and dual control. *Machine Learning*, 25:5–22.

D'Epenoux, F. (1963). A probabilisitc production and inventory problem. *Management Science*, 10:98–108.

Di Caro, G. and Dorigo, M. (1998). An adaptive multi-agent routing algorithm inspired by ants behavior. In *Proceedings of PART98 - Fifth Annual Australasian Conference on Parallel and Real-Time Systems.*

Dietterich, T. (1997). Hierarchical reinforcement learning with the MAXQ value function decomposition. Technical report, Oregon State University.

Digney, B. (1996). Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 363–372. MIT Press, Bradford Books.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Dodge, Y., Fedorov, V. V., and Wynn, H. P., editors (1988). *Optimal Design and Analysis of Experiments: Proceedings of First International Conference on Optimal Design and Analysis of Experiments.* Elsevier Publishers.

Dorigo, M. and Colombetti, M. (1997). *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press/Bradford Books. in press.

Dorigo, M. and Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation*, 1(1):53–66.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29–41.

Fedorov, V. V. (1972). *Theory of optimal experiments.* Academic Press.

Friedman, J., Bentley, J., and Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *AMC Transactions on Mathematical Software*, 3(3):209–226.

Fritzke, B. (1994). Supervised learning with growing cell structures. In Cowan, J., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 255–262. San Mateo, CA: Morgan Kaufmann.

Gambardella, L. M., Taillard, E., and Dorigo, M. (1997). Ant colonies for the QAP. Technical Report IDSIA-4-97, IDSIA, Lugano, Switzerland. Submitted to: Journal of the Operational Research Society.

Gittins, J. C. (1989). *Multi-armed Bandit Allocation Indices.* Wiley, Chichester, NJ.

Givan, R., Leach, S., and Dean, T. (1998). Bounded parameter Markov decision processes. Technical report. Retrievable from http://www.cs.brown.edu/people/tld/home.html.

Glover, F. and Laguna, M. (1997). *Tabu Search.* Kluwer Academic Publishers.

Gordon, G. (1995a). Stable function approximation in dynamic programming. Technical Report CMU-CS-95-103, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Gordon, G. (1995b). Stable function approximation in dynamic programming. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 261–268. Morgan Kaufmann Publishers, San Francisco, CA.

Heger, M. (1994). Consideration of risk in reinforcement learning. In *Machine Learning: Proceedings of the 11th International Conference*, pages 105–111. Morgan Kaufmann Publishers, San Francisco, CA.

Hihi, S. E. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 493–499. MIT Press, Cambridge MA.

Hinton, G. and Sejnowski, T. (1983). Optimal perceptual inference. In *Proceedings of the 1983 IEEE Conference on Computer Vision and Pattern Recognition*, pages 448–453. New york: IEEE.

Hochreiter, S. and Schmidhuber, J. H. (1997). Long short-term memory. *Neural Computation*, 9:1681–1726.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558.

Humphrys, M. (1996). Action selection methods using reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 135–144. MIT Press, Bradford Books.

Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201.

Jaakkola, T., Singh, S. P., and Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 345–352. MIT Press, Cambridge MA.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87.

Jollife, I. T. (1986). *Principal Component Analysis.* New York: Springer Verlag.

Jordan, M. I. and Jacobs, R. A. (1992). Hierarchies of adaptive experts. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 985–993. Morgan Kauffmann.

Jordan, M. I. and Rumelhart, D. E. (1990). Supervised learning with a distal teacher. Technical Report Occasional Paper #40, Center for Cognitive Science, Massachusetts Institute of Technology.

Judd, J. (1990). *Neural Network Design and the Complexity of Learning.* The MIT press, Cambridge.

Kaelbling, L. P. (1993). *Learning in Embedded Systems.* MIT Press.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1995). Planning and acting in partially observable stochastic domains. Unpublished report.

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

Kearns, M. and Singh, S. P. (1998). Near-optimal performance for reinforcement learning in polynomial time. Retrievable from http://www.research.att.com/~mkearns/.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). Robocup: The robot world cup initiative. In *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*. The ACM Press.

Koenig, S. and Simmons, R. G. (1992). Complexity analysis of real-time exploration learning applied to finding shortest paths in deterministic domains. Technical Report CMU-CS-93-106, School of Computer Science, Carnegie Mellon University.

Koenig, S. and Simmons, R. G. (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement learning algorithms. *Machine Learning*, 22:228–250.

Kohonen, T. (1988). *Self-Organization and Associative Memory*. Springer, second edition.

Koza, J. R. (1992). Genetic evolution and co-evolution of computer programs. In Langton, C., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 313–324. Addison Wesley Publishing Company.

Kröse, B. J. A. and van Dam, J. W. M. (1992). Adaptive state space quantisation : Adding and removing neurons. In Aleksander, I. and Taylor, J., editors, *Artificial Neural Networks, 2*, pages 619–624. North-Holland/Elsevier Science Publishers, Amsterdam.

Kröse, B. J. A. and Van de Smagt, P. (1993). An introduction to neural networks. Autonomous Systems, University of Amsterdam.

Landelius, T. (1997). *Reinforcement Learning and distributed Local Model Synthesis*. PhD thesis, Linköping Universtity, Sweden.

Lauritzen, S. and Wermuth, N. (1989). Graphical models for associations between variables some of which are qualitative and some quantitative. *Annals of Statistics*, 17:31–57.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

Li, M. and Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications*. Springer.

Lin, C.-S. and Chiang, C.-T. (1997). Learning convergence of CMAC technique. *IEEE Transactions on Neural Networks*, 8(6):1281–1292.

Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh.

Lin, T., Horne, B., and Giles, C. (1996). How embedded memory in recurrent neural network architectures helps learning long-term temporal dependencies. Technical Report CS-TR-3626 and UMIACS-TR-96-28, University of Maryland, College Park MD 20712.

Lindgren, K. and Nordahl, M. G. (1994). Cooperation and community structure in artificial ecosystems. *Artificial Life*, 1(1/2):15–37.

Littman, M. L. (1994a). Markov games as a framework for multi-agent reinforcement learning. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 157–163. Morgan Kaufmann Publishers, San Francisco, CA.

Littman, M. L. (1994b). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J. A., and Wilson, S. W., editors, *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pages 297–305. MIT Press/Bradford Books.

Littman, M. L. (1996). *Algorithms for Sequential Decision Making.* PhD thesis, Brown University.

Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995a). Learning policies for partially observable environments: Scaling up. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 362–370. Morgan Kaufmann Publishers, San Francisco, CA.

Littman, M. L., Dean, T. L., and Kaelbling, L. P. (1995b). On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI-95).*

Lovejoy, W. S. (1991). A survey of algorithms methods for partially observable Markov decision processes. *Annals of Operations Research*, 28:47–66.

Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. (1997). Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence (IJCAI-97).*

Mahadevan, S. (1996). Sensitive discount optimality: Unifying discounted and average reward reinforcement learning. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 328–336. Morgan Kaufmann Publishers, San Francisco, CA.

Martin, M. (1998). *Reinforcement Learning for Embedded Agents facing Complex tasks.* PhD thesis, Universitat Politecnica de Catalunya, Barcelona.

Martinetz, T. and Schulten, K. (1991). A "neural-gas" network learns topologies. In Kohonen, T., Mäkisara, K., Simula, O., and Kangas, J., editors, *Artificial Neural Networks*, pages 397–402. Elsevier Science Publishers B.V., North-Holland.

Mataric, M. J. (1994). *Interaction and Intelligent Behavior.* PhD thesis, Massacusetts institute of Technology.

Matsubara, H., Noda, I., and Hiraki, K. (1996). Learning of cooperative actions in multi-agent systems: a case study of pass play in soccer. In Sen, S., editor, *Working Notes for the AAAI-96 Spring Symposium on Adaptation, Coevolution and Learning in Multi-agent Systems*, pages 63–67.

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Machine Learning: Proceedings of the Tenth International Conference*, pages 190–196. Morgan Kaufmann, Amherst, MA.

McCallum, R. A. (1996). Learning to use selective attention and short-term memory in sequential tasks. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 315–324. MIT Press, Bradford Books.

McDonald, M. A. F. and Hingston, P. (1994). Approximate discounted dynamic programming is unreliable. Technical Report 94/6, Department of Computer Science, The University of Western Australia, Crawley, WA.

Moore, A. W. (1991). *Efficient Memory-based Learning for Robot Control.* PhD thesis, University of Cambridge.

Moore, A. W. (1998). Personal communication.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130.

Moore, A. W. and Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21:3:199–233.

Moore, A. W., Atkeson, C. G., and Schaal, S. A. (1997). Locally weighted learning for control. *Artificial Intelligence Review*, 11:75–113.

Munos, R. (1996). A convergent reinforcement learning scheme in the continuous case: the finite element reinforcement learning. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 337–345. Morgan Kaufmann Publishers, San Francisco, CA.

Myerson, R. (1991). *Game Theory.* Harvard University Press.

Nguyen and Widrow, B. (1989). The truck backer-upper: An example of self learning in neural networks. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D.C.*, volume 1, pages 357–364.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill.

Nowlan, S. (1991). *Soft Competitive Adaption: Neural Network Learning Algorithms based on Fitting Statistical Mixtures.* PhD thesis, Carnegie Mellon University, Pittsburgh.

Oja, E. and Karhunen, J. (1995). Signal separation by nonlinear hebbian learning. In Palaniswami, M., Attikiouzel, Y., Marks II, R., Fogel, D., and Fukuda, T., editors, *Computational Intelligence - a Dynamic System Perspective*, pages 83–97. IEEE Press, New York.

Okabe, A., Boots, B., and Sugihara, K. (1990). *Spatial Tesselations - Concepts and applications of Voronoi diagrams.* Wiley and Sons, New York.

Omohundro, S. M. (1988). Foundations of geometric learning. Technical Report UIUCDCS-R-88-1 408, University of Illinois, Department of Computer Science.

Omohundro, S. M. (1989). Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA.

Omohundro, S. M. (1991). Bumptrees for efficient function, constraint, and classification learning. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 693–699. San Mateo, CA: Morgan Kaufmann.

Parr, R. and Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1088–1094. Morgan Kaufmann.

Parr, R. and Russell, S. (1997). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 11.*

Peek, N. B. (1997). Predictive probabilistic models for treatment planning in paediatric cardiology. In *Proceedings of CESA'98 IMACS Multiconference (Computational Engineering in Systems Applications), Symposium on Signal Processing and Cybernetics.*

Peng, J. and Williams, R. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22:283–290.

Peng, J. and Williams, R. J. (1993). Efficient learning and planning with the DYNA framework. *Adaptive Behavior*, 1:437–454.

Pineda, F. (1997). Mean-field theory for batched TD(λ). *Neural Computation*, 9(7):1404–1419.

Pollack, J. and Blair, A. (1996). Why did TD-Gammon work. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*, pages 10–16, Cambridge MA. MIT Press.

Precup, D. and Sutton, R. (1998). Theoretical results on reinforcement learning with temporally abstract options. In *Proceedings of the Tenth European Conference on Machine Learning (ECML'98).*

Preparate, F. P. and Shamos, M. I. (1985). *Computational Geometry: an Introduction.* Springer Verlag, New York.

Prescott, T. (1994). *Exploration in Reinforcement and Model-based Learning.* PhD thesis, University of Sheffield.

Press, W., Teukolsky, S., Vettering, W., and Flannery, B. (1988). *Numerical recipes in C.* Cambridge University Press.

Rao, C. and Mitra, S. (1971). *Generalized Inverse of Matrices and Its Applications.* Wiley, New York.

Rechenberg, I. (1971). Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation. Published 1973 by Fromman-Holzboog.

Rechenberg, I. (1989). Evolution strategy: Nature's way of optimization. In Bergmann, editor, *Methods and Applications, Possibilities and Limitations*, pages 106–126. Lecture notes in Engineering.

Resnick, S. (1992). *Adventures in stochastic processes.* Birkhaeuser Verlag.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments.* PhD thesis, University of Texas, Austin, Texas.

Ron, D., Singer, Y., and Tishby, N. (1994). Learning probabilistic automata with variable memory length. In Aleksander, I. and Taylor, J., editors, *Proceedings Computational Learning Theory.* ACM Press.

Roth, A. and Erev, I. (1995). Learning in extensive-form games: Experimental data and simple dynamic models in the intermediate term. *Games and Economic Behavior*, 8:164–212. Special issue: Nobel Symposium.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press.

Rummery, G. and Niranjan, M. (1994). On-line Q-learning using connectionist sytems. Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK.

Sahota, M. (1993). Real-time intelligent behaviour in dynamic environments: Soccer-playing robots. Master's thesis, University of British Columbia.

Sałustowicz, R. P. and Schmidhuber, J. H. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. H. (1997a). Evolving soccer strategies. In *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97)*, pages 502–506. Springer-Verlag Singapore.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. H. (1997b). On learning soccer strategies. In Gerstner, W., Germond, A., Hasler, M., and Nicoud, J.-D., editors, *Proceedings of the Seventh International Conference on Artificial Neural Networks (ICANN'97)*, volume 1327 of *Lecture Notes in Computer Science*, pages 769–774. Springer-Verlag Berlin Heidelberg.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. H. (1998). Learning team strategies: Soccer case studies. *Machine Learning*, 33(2/3).

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229.

Sandholm, T. W. and Crites, R. H. (1995). On multiagent Q-learning in a semi-competitive domain. In Weiss, G. and Sen, S., editors, *IJCAI'95 Workshop: Adaption and Learning in Multi-Agent Systems*, pages 164–176. Springer-Verlag.

Santamaria, J. C., Sutton, R. S., and Ram, A. (1996). Experiments with reinforcement learning in problems with continuous state and action spaces. Technical Report COINS 96-088, Georgia Institute of Technology, Atlanta.

Schmidhuber, J. H. (1991a). Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE.

Schmidhuber, J. H. (1991b). Learning to generate sub-goals for action sequences. In Kohonen, T., Mäkisara, K., Simula, O., and Kangas, J., editors, *Artificial Neural Networks*, pages 967–972. Elsevier Science Publishers B.V., North-Holland.

Schmidhuber, J. H. (1991c). A possibility for implementing curiosity and boredom in model-building neural controllers. In Meyer, J. A. and Wilson, S. W., editors, *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227. MIT Press/Bradford Books.

Schmidhuber, J. H. (1991d). Reinforcement learning in Markovian and non-Markovian environments. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 500–506. San Mateo, CA: Morgan Kaufmann.

Schmidhuber, J. H. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242.

Schmidhuber, J. H. (1996). A general method for incremental self-improvement and multi-agent learning in unrestricted environments. In Yao, X., editor, *Evolutionary Computation: Theory and Applications*. Scientific Publ. Co., Singapore.

Schmidhuber, J. H. (1997). What's interesting? Technical Report IDSIA-35-97, IDSIA.

Schmidhuber, J. H., Zhao, J., and Schraudolph, N. N. (1997a). Reinforcement learning with self-modifying policies. In Thrun, S. and Pratt, L., editors, *Learning to learn*. Kluwer.

Schmidhuber, J. H., Zhao, J., and Wiering, M. A. (1996). Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA.

Schmidhuber, J. H., Zhao, J., and Wiering, M. A. (1997b). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130.

Schneider, J. G. (1997). Exploiting model uncertainty estimates for safe dynamic control learning. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 1047–1053. MIT Press/Bradford Books, Cambridge.

Schraudolph, N. N., Dayan, P., and Sejnowski, T. J. (1994). Temporal difference learning of position evaluation in the game of go. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann, San Francisco.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Machine Learning: Proceedings of the Tenth International Conference*, pages 298–305. Morgan Kaufmann, Amherst, MA.

Singh, S. P. (1992). The efficient learning of multiple task sequences. In Moody, J., Hanson, S., and Lippman, R., editors, *Advances in Neural Information Processing Systems 4*, pages 251–258, San Mateo, CA. Morgan Kaufmann.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing elibibility traces. *Machine Learning*, 22:123–158.

Singh, S. P. and Yee, R. C. (1994). An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Decision Processes.* PhD thesis, Standford, California.

Steels, L. (1997). Constructing and sharing perceptual distinctions. In van Someren, M. and Widmer, G., editors, *Machine Learning: Proceedings of the ninth European Conference*, pages 4–13. Springer-Verlag, Berlin Heidelberg.

Stone, P. and Veloso, M. (1996). Beating a defender in robotic soccer: Memory-based learning of a continuous function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 8*, pages 896–902. MIT Press, Cambridge MA.

Stone, P. and Veloso, M. (1998). Team-partitioned opaque-transition reinforcement learning. In *Proceedings of the Conference on automated learning and discovery (CONALD'98): Robot Exploration and Learning.* Carnegie Mellon University, Pittsburgh.

Storck, J., Hochreiter, S., and Schmidhuber, J. H. (1995). Reinforcement driven information acquisition in nondeterministic environments. In *Proceedings of the International Conference on Artificial Neural Networks*, volume 2, pages 159–164. EC2 & Cie, Paris.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci.

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

Sutton, R. S. (1990). Integrated architectures for learning, planning and reacting based on dynamic programming. In *Machine Learning: Proceedings of the Seventh International Workshop.*

Sutton, R. S. (1995). TD models: Modeling the world at a mixture of time scales. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 531–539. Morgan Kaufmann Publishers, San Francisco, CA.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1045. MIT Press, Cambridge MA.

Sutton, R. S., Precup, D., and Singh, S. P. (1998). Between MDPs and semi-MDPs: Learning, planning, learning and sequential decision making. Technical Report COINS 89-95, University of Massachusetts, Amherst.

Teller, A. (1994). The evolution of mental models. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, pages 199–219. MIT Press.

Tesauro, G. (1992). Practical issues in temporal difference learning. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 259–266. San Mateo, CA: Morgan Kaufmann.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68.

Tham, C. (1995). Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems*, 15(4):247–274.

Thrun, S. (1992). Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie-Mellon University.

Thrun, S. (1995). Learning to play the game of chess. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. San Fransisco, CA: Morgan Kaufmann.

Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence Journal*, 99(1):21–71.

Thrun, S. and Möller, K. (1992). Active exploration in dynamic environments. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 531–538. San Mateo, CA: Morgan Kaufmann.

Thrun, S. and Schwartz, A. (1995). Finding structure in reinforcement learning. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 385–392. MIT Press, Cambridge MA.

Trovato, K. (1996). *A* Planning in Discrete Configuration Spaces of Autonomous Systems*. PhD thesis, University of Amsterdam.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202.

Tsitsiklis, J. N. and Van Roy, B. (1996). An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, Cambrdge,MA: MIT Laboratory for Information and Decision Systems.

Van Dam, J. W. M. (1998). *Environmental Modelling for Mobile Robots: Neural Learning for Sensor Fusion*. PhD thesis, University of Amsterdam, The Netherlands.

Van de Smagt, P. (1995). *Visual robot arm guidance using neural networks.* PhD thesis, University of Amsterdam, The Netherlands.

Van der Wal, J. (1981). *Stochastic Dynamic Programming.* Number 139 in Mathematical Centre tracts. Mathematisch Centrum, Amsterdam.

Van Emde Boas, P., Kaas, R., and Zijlstra, E. (1977). Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127.

Vennix, J. A. M. (1996). Systeemdynamica methode for strategie-ontwikkeling. Technical report, Faculteit der Beleidswetenschappen, Katholieke Universiteit Nijmegen.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, England.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University.

Whitehead, S. (1992). *Reinforcement Learning for the adaptive control of perception and action.* PhD thesis, University of Rochester.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. *1960 IRE WESCON Convention Record*, 4:96–104. New York: IRE. Reprinted in Anderson and Rosenfeld [1988].

Wiering, M. A. (1995). *TD Learning of Game Evaluation Functions with Hierarchical Neural Architectures.* Master's thesis, Department of Computer Systems, University of Amsterdam.

Wiering, M. A. and Dorigo, M. (1998). Learning to control forest fires. In Haasis, H.-D. and Ranze, K. C., editors, *Proceedings of the 12th international Symposium on "Computer Science for Environmental Protection"*, volume 18 of *Umweltinformatik Aktuell*, pages 378–388, Marburg. Metropolis Verlag.

Wiering, M. A., Sałustowicz, R. P., and Schmidhuber, J. H. (1998). CMAC models learn to play soccer. In Niklasson, L., Bodén, M., and Ziemke, T., editors, *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN'98)*, volume 1, pages 443–448. Springer-Verlag, London.

Wiering, M. A. and Schmidhuber, J. H. (1996). Solving POMDPs with Levin search and EIRA. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 534–542. Morgan Kaufmann Publishers, San Francisco, CA.

Wiering, M. A. and Schmidhuber, J. H. (1997). HQ-learning. *Adaptive Behavior*, 6(2):219–246.

Wiering, M. A. and Schmidhuber, J. H. (1998a). Efficient model-based exploration. In Meyer, J. A. and Wilson, S. W., editors, *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 6*, pages 223–228. MIT Press/Bradford Books.

Wiering, M. A. and Schmidhuber, J. H. (1998b). Fast online Q(λ). *Machine Learning Journal.*

Williams, R. J. and Baird, L. C. (1993). Tight performance bounds on greedy policies based on imperfect value function. Technical Report NU-CCS-93-14, College of Computer Science, Northeastern University, Boston, MA.

Wilson, S. (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2:1–18.

Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175.

Zhang, N. L. and Liu, W. (1996). Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology.

Zhao, J. and Schmidhuber, J. H. (1996). Incremental self-improvement for life-time multi-agent reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 516–525. MIT Press, Bradford Books.

Index

BRO, 138
DP, 5
DT, 148
EC, 135
FA, 115
GO, 138
GP, 135
HMM, 109
IE, 72
IS, 88
LFA, 118
LP, 18 LS, 111
MBIE, 78
MC. 30
MDP, 5
MLM, 56
MP, 13
NO, 138
PAC, 68
PI, 50
PIPE, 135
POMDP, 7
PPT, 136
PS, 7
PW, 37
RL, 1
RP, 94
RPP, 12
RTDP, 7
SAP, 6
SC, 102
SSA, 111
TD, 5
WM, 6
WTA, 118
action evaluation function, 16
active learning, 71
agent, 1
belief state, 8, 89, 109
Chapman-Kolmogorov equation, 163
classifier system, 110
CMAC, 122
  model, 133
complete state information, 4
consistent representation, 111
contraction, 17
control, 2
credit assignment problem, 29
data-retrieval problem, 121
decision tree, 148
delayed reward, 29
discount factor, 14
dual control, 71
Dyna, 68
dynamic programming, 5, 18
  and function approximation, 25
  real time, 7
eligibility traces, 32, 37
  accumulating, 34
  another view, 44
  connecting, 45
  multiple, 44
  replacing, 34
environment, 8
  changing, 8
  non-stationary, 8
evolutionary computation, 135
exploitation, 28
exploration, 1, 7, 28, 71
  Boltzmann, 73
  directed, 7, 71
  initialize high, 74
  Max-Boltzmann, 73
  Max-random, 29, 72
  reward function, 74
    error based, 75
    frequency based, 75
    recency based, 75

undirected, 7, 71, 72
exploration/exploitation dilemma, 28, 71
fixed point, 17
function approximator, 7, 115
  and Q(λ), 127
  and world modelling, 129
  dynamic topology, 116
  local, 118
  static topology, 116
genetic programming, 135
global model, 116
hidden Markov model, 109
hidden state, 87
  problem, 87
  specification, 89
HQ-learning, 93
incomplete state information, 4
independent filter model, 148
internal state, 7, 88
interval estimation, 72, 78
  model-based, 78
landmark state, 90
learn, 1, 2
Levin search, 111
linear models, 130
linear network, 116
linear programming, 20
local model, 116
K-d tree, 121
map learning, 109
Markov chain, 161
  higher order, 167
  properties, 164
Markov decision process, 5, 13
  finite horizon, 14
  infinite horizon, 15
Markov order, 166
Markov process, 13, 161
Markov property, 5, 29, 161
maximum likelihood model, 56
MAXQ, 112
memory bits, 109
model-based, 6
  interval estimation, 78
  Q-learning, 7, 57
Monte Carlo sampling, 30
  every visit, 34
  first visit, 34
multi-agent RL, 8
nearest neighbor search, 121
neural gas, 119
  model, 132
occupancy probabilities, 162
optimal experiment design, 71
optimal linear associative memory, 132
partially observable MDP, 7, 87
  specification, 89
perceptual aliasing, 87
policy, 1
  optimal, 14
  reactive, 94
  shared, 43
policy iteration, 18
policy tree, 91
prediction problems, 30
prioritized sweeping, 7, 59
probabilistic incremental program evolution, 135
probabilistic program tree, 136
probably approximately correct framework, 68
program evolution, 110
Q-function, 17
Q-learning, 6, 35
  model-based, 7, 57
Q(λ)-learning, 6, 36
  Fast Q(λ), 38
  offline, 126
  Semi-online, 38
  Team, 43
  with function approximators, 127
reactive policy problem, 12, 94
reactive strategy, 3
recurrent neural network, 108
reinforcement learning, 1, 3
  direct, 6, 27
  indirect, 6, 27
  model-based, 6
  multi-agent, 8
  offline, 29
    and function approximation, 124
  online, 29
  vs. evolutionary computation, 148
reward function, 2, 13

for exploration, 74
robotic soccer, 8, 135
simulator, 136
sampling and approximation, 150
Sarsa, 38
sequential decision making, 1
soft cell competition, 118
subgoal combination, 102
success-story algorithm, 111
TD-learning, 5, 30
TD(0), 32
TD(λ), 32
team Q(λ)-learning, 43
temporal differences, 5, 30
transition function, 13
value function, 3
optimal, 15
shared, 43
value iteration, 19
Voronoi diagram, 121
winner take all, 118
world state, 1

Summary

This thesis describes reinforcement learning (RL) methods which can solve sequential decision making problems by learning from trial and error. Sequential decision making problems are problems in which an artificial agent interacts with a specific environment through its sensors (to get inputs) and effectors (to make actions). To measure the goodness of some agent's behavior, a reward function is used which determines how much an agent is rewarded or penalized for performing particular actions in particular environmental states. The goal is to find an action selection policy for the agent which maximizes the cumulative reward collected in the future.

In RL, an agent's policy maps sensor-based inputs to actions. To evaluate a policy, a value function is learned which returns for each possible state the future cumulative reward collected by following the current policy. Given a value function, we can simply select the action with the largest value. In order to learn a value function for a specific problem, reinforcement learning methods simulate a policy and use the resulting agent's experiences consisting of <state,action,reward,next-state> quadruples.

There are different RL problems and different RL methods for solving them. We describe different categories of problems and introduce new methods for solving them.

Markov Decision Problems

If the agent's inputs allow for determining the state of the environment with certainty and the environment is stationary, we can model the decision making problem as a Markov Decision Problem (MDP). In a MDP, the probabilities and expected rewards of transitions to possible next states only depend on the current state and action and not on any previous events. Given an exact model of a MDP, we can use Dynamic Programming (DP) methods for computing optimal policies. Such methods iterate over each state and constantly recompute the value function by looking ahead one step at each time. There are two principal DP methods: value iteration and policy iteration. Value iteration reupdates the policy after each improvement of the value function by immediately mapping a state to the action with the largest value, whereas policy iteration first completely evaluates a policy by using sufficient iterations, after which a policy update step is performed. Although in theory policy iteration always converges in finitely many iterations whereas value iteration does not, our experiments on a set of maze-tasks indicate that value iteration converges much faster to a solution. The main reason for this is that policy iteration, postponing policy updates, wastes a lot of time evaluating policies which generate cycles in the state space.

Reinforcement Learning

Dynamic programming methods can only be used if we have an *a priori* model of the MDP. Therefore their practical utility is severely limited. In case we do not possess such a model, we can use reinforcement learning methods which learn policies by interacting with a real or simulated environment. For this we use trials during which agents generate trajectories through the state space and let agents learn from the resulting experiences.

RL methods such as Q-learning learn a value function by minimizing temporal differences between the current state's value and the reward which immediately follows plus the next state's value. Q-learning updates values of state/action pairs along a solution-path by looking ahead only one single step in a trial, and therefore it takes many trials before a goal-reward will be backpropagated to the start. TD(λ) methods are parameterized by a variable λ en use the entire future for updating the value of a state, although the degree of influence of state values in the distant future is less than values of immediately successive states. This makes it possible to learn from effects of actions which show up after a long time. Q(λ)-learning combines Q-learning and TD(λ) and uses single experiences to update values of multiple state/action pairs that have occurred in the past. Although Q(λ) methods can sometimes learn policies from much less experiences than Q-learning, naive implementations of the online Q(λ) method suffer from a large computational overhead. We have introduced a novel implementation of the algorithm which makes it possible to efficiently perform the updates.

If multiple agents are used simultaneously in an environment, they can share the policy to collect more experiences at each time step which may speed-up learning a policy. Using "policy-sharing", agents select actions according to the same policy, but their actions may differ since they receive different situation specific inputs. In such cases we can use each agent's generated trajectory individually to update the shared value function. However, sometimes there exists an interaction point (IP) between agent's trajectories such as a state visited by both of them. In such cases we can connect the trajectories and update values of states visited by one agent before the IP using values of states visited by the other agent after the IP. In this way we collect much more experiences to learn from, which may increase learning speed. We describe an implementation of this Team Q(λ) algorithm and evaluate it in a non-cooperative maze-task using different numbers of agents. The results do not show that the method improves learning speed. One reason for this is that our implementation allows for connecting a state-trajectory of one agent with another state-trajectory which may be generated a long time ago by a possibly much worse policy and learning from such a trajectory may degenerate the policy. Therefore we need to study the trade-off between the quantity and quality of learning examples.

Model-based Reinforcement Learning

Q-learning and Q(λ) methods are direct memoryless RL methods which directly estimate the value function from generated state-trajectories. Indirect or model-based RL methods first learn a model of the MDP and then apply DP algorithms to compute the value function. When used in an online manner, DP algorithms are very slow since they recompute the value function after each experience by iterating over all states. To speed up dynamic programming algorithms, some algorithms manage which update-steps should be performed after each experience so that only the most useful updates are made. One of the most efficient management methods is Prioritized Sweeping (PS) which assigns priorities to updating the values of different states according to a heuristic estimate of the size of the values' updates. The original PS algorithm by Moore and Atkeson (1993) calculates state-priorities based upon the largest single update of one of the successor states. We describe an alternative PS method which uses the exact update sizes of state values to compute priorities. This may overcome PS' problems in case there are many tiny state value updates resulting in a large update, which remains undetected by PS.

We compare model-based RL methods to direct RL methods on a set of maze-tasks and observe that model-based RL results in much faster learning performances and better final policies. Finally, we observe that our PS algorithm achieves better results than the original PS algorithm.

Exploration

If we test an agent's policy by always selecting the action with the largest value, we may end up with a suboptimal policy which has (almost) never visited a large part of the state space. Therefore it is important that the agent also selects exploration actions and tries to increase its knowledge of the environment. Usually, however, exploration actions cause some loss of immediate reward intake. Therefore, if an agent wants to maximize its cumulative reward during its limited life-time, it faces the problem of trying to spend as little time as possible on exploration while still being able to find a highly rewarding policy (the exploration/exploitation dilemma). However, if we are interested in learning a policy achieving some minimal performance level as soon as possible, we do not care about intermediate rewards and thus only exploration is important.

Exploration methods can be split into undirected exploration methods which use pseudo-random generators to try their luck on generating novel interesting experiences, and directed exploration methods which use additional information such as the knowledge of what the agent has seen or learned in order to send the agent to interesting unknown regions.

Directed exploration methods can be constructed by constructing an exploration reward rule which makes it possible to learn an exploration value function for estimating global information gain for selecting particular action-sequences. Although we can use any RL method for learning an exploration value function, we propose to use model-based RL for this. Finally, in order to focus only on state/action pairs which can belong to the optimal policy, we introduce MBIE, an algorithm which uses the standard deviance for computing optimistic value functions. The results on experiments with maze-tasks featuring multiple goals show that directed exploration using model-based RL can significantly outperform indirect exploration. Furthermore, the results show that MBIE can be used to almost always obtain near-optimal performance levels and that MBIE efficiently deals with the exploration/exploitation dilemma.

Partially Observable Problems

In the real world the agent's sensory inputs may not always convey all information needed to infer the current environmental state. Therefore the agent will sometimes be uncertain about the real state and that makes the decision problem much harder. Such problems are usually called Partially Observable Markov Decision Problems (POMDPs) and even solving deterministic POMDPs in NP-hard. To solve such problems, the agent needs to use previous events or some kind of short-term memory to disambiguate inputs. Exact methods make use of a belief state, a vector representing the probabilities that the real environmental state is each of the possible states. Then they use DP algorithms to compute optimal solutions in the belief state space. Exact methods are computationally infeasible if there are many states, since they are too slow. Therefore we are interested in heuristic algorithms which can be used for larger problems, but do not necessarily find optimal solutions. We describe a novel algorithm, HQ-learning, which is able to learn good policies for large deterministic POMDPs. HQ automatically decomposes POMDPs into sequences of simpler subtasks that can be solved by memoryless policies learnable by reactive subagents. Decomposing the task into subtasks and learning policies for each subtask is done by two cooperating Q(λ)-learning rules. Experiments, on (among other) a large maze-task consisting of 960 states and only 11 different, highly ambiguous inputs, show that HQ-learning can quickly find good or even optimal solutions for difficult problems.

Function Approximation in RL

An important topic in RL is the application of function approximation to learn value functions for continuous or very large state spaces. We describe three different function approximators: linear networks, neural gas, and CMACs, and describe how they can be combined with direct and model-based RL methods. Then we describe a simulated multi-agent soccer environment which we use to test the learning capabilities and problems of the function approximators and RL methods. The goal is to learn good soccer strategies against a fixed programmed opponent. The results show that the linear networks and neural gas architecture trained with Q(λ) had problems to steadily improve their learned policies. Sometimes, the performance broke down due to catastrophic learning interference. CMACs was more stable, although CMAC-models, the combination of CMACs and model-based RL was the only method which found really good performances and was able to beat some good opponents.

Conclusions

We have shown in this thesis that RL can be used to quickly find good solutions for different kinds of problems. The methods described in this thesis do not need a model of the world, but learn a policy for an agent by interacting with the (real) environment. Especially model-based RL can be very efficient since it is able to use all information. Therefore the practical utility of (model-based) reinforcement learning is very large and we expect the research field to grow a lot in importance in the forthcoming years.

Samenvatting

Dit proefschrift beschrijft reinforcement leermethoden (RL-methoden) welke sequentiële besluitvormings problemen op kunnen oplossen door het leren van proberen en fouten maken. Sequentiële besluitsvormings problemen zijn problemen waarin een kunstmatige agent interacteert met een bepaalde omgeving door middel van haar/zijn sensoren (om input te verkrijgen) en effectoren (om acties te verrichten). Voor het meten van de goedheid van het gedrag van een specifieke agent, maken we gebruik van een beloningsfunctie welke bepaalt in hoeverre een agent beloond of gestraft moet worden voor het verrichten van bepaalde handelingen in bepaalde situaties. Het doel is om een actie-selecteer handelsprocedure voor de agent te vinden welke de totale som der beloningen verkregen in de toekomst maximaliseert.

In een RL-setting worden inputs geprojecteerd op acties door de handelsprocedure van de agent. Voor het evalueren van een handelsprocedure, leren we een waarde-functie welke voor elke mogelijke toestand van de wereld de som der beloningen teruggeeft welke in de toekomst wordt verkregen door het volgen van de huidige handelsprocedure. Als we eenmaal een waarde-functie hebben verkregen, kunnen we gewoon de actie selecteren met de hoogste waarde. Om een waarde-functie te leren voor een bepaald probleem, simuleren reinforcement leermethoden een handelsprocedure en gebruiken ze de resulterende ervaringen van de agent welke bestaan uit <toestand, actie, beloning, volgende-toestand> quadruples om de toekomstige beloningssom te schatten beginnende in elke mogelijke toestand.

Er zijn verschillende RL-problemen en verschillende RL-methoden om ze op te lossen. We beschrijven verschillende probleem-categorieën en introduceren nieuwe methoden om ze op te lossen.

Markov Besluits Problemen

Als de input van een agent toestaat om de toestand van de omgeving met zekerheid te bepalen en de causale wetten die in de omgeving opgaan zijn onveranderlijk, dan kunnen we het besluitvorming probleem modelleren als een Markov Besluits Probleem (MBP). In een MBP hangen de kansen en beloningen van overgangen naar mogelijke opvolgende toestanden enkel af van de huidige toestand en gekozen actie en niet van vorige gebeurtenissen. Gegeven een exact model van een MBP, kunnen we Dynamisch Programmeer (DP) methoden gebruiken om optimale handelsprocedures te berekenen. Zulke methoden itereren over elke toestand en verbeteren de waarde-functie voortdurend door steeds een stapje verder vooruit te kijken. Er zijn twee voorname DP methoden: waarde iteratie en handelsprocedure iteratie. Waarde iteratie verbetert de handelsprocedure na elke verbetering van de waarde-functie door onmiddellijk elke toestand te projecteren op de actie met de hoogste waarde, terwijl

besluitsprocedure iteratie eerst de handelsprocedure volledig evalueert om zo een exact evaluatie van de handelsprocedure te verkrijgen, waarna een verbetering van de handelsprocedure wordt gemaakt. Hoewel handelsprocedure iteratie in theorie altijd in eindig veel iteraties convergeert en waarde iteratie niet, wijzen onze experimenten erop dat waarde iteratie veel sneller een oplossing kan berekenen. De hoofdreden hiervoor is dat handelsprocedure iteratie verbeteringen in de besluitsprocedure uitstelt en dusdanig veel tijd verliest door het volledig evalueren van handelsprocedures welke wederkerende paden in de toestandruimte genereren.

Reinforcement Leren

Dynamisch programmeer methoden kunnen enkel gebruikt worden als we een *a priori* model van de MBP tot onze beschikking hebben. Daarom is hun praktische bruikbaarheid erg beperkt. In het geval dat we niet over zo'n model beschikken, kunnen we reinforcement leermethoden gebruiken welke handelsprocedures leren door te interacteren met een werkelijke of gesimuleerde omgeving. RL-methoden gebruiken experimenten waarin de agent toestands-paden door de toestandsruimte genereert waarvan geleerd kan worden. In het optimale geval leert de agent dusdanig optimale paden te genereren.

RL-methoden zoals Q-leren leren een waarde-functie door tijdelijke verschillen tussen de waarde van de huidige toestand en de beloning welke onmiddellijk volgt plus de waarde van de volgende toestand te minimaliseren. Q-leren past waarden van toestand/actie paren welke op een oplossings-pad liggen aan door in elk opvolgend experiment slechts één stap vooruit te kijken en daarom zijn er veel experimenten nodig voordat een beloning welke verkregen wordt bij de doel-toestand teruggepropageerd wordt naar het begin. TD(λ) methoden gebruiken de hele toekomst voor het aanpassen van de waarde van een toestand, hoewel waarden van toestanden welke ver in de toekomst liggen minder zwaar wegen dan waarden van onmiddellijk opvolgende toestanden. Dit maakt het mogelijk om van de gevolgen van acties te leren welke pas na een lange tijd zichtbaar worden. Q(λ) combineert Q-leren met TD(λ) en gebruikt ervaringen om waarden van meerdere toestand/actie paren welke in het verleden zijn opgetreden aan te passen. Hoewel Q(λ) methoden soms handelsprocedures kunnen leren van veel minder ervaringen dan Q-leren, lijden naïve implementaties van online Q(λ) methoden onder een grote computationele overhead. Wij hebben een nieuwe implementatie van het algoritme geïntroduceerd welke het mogelijk maakt om de aanpassingen efficiënt te verrichten.

Als meerdere agenten gelijktijdig gebruikt worden in een omgeving, dan kunnen ze de handelsprocedure delen om zo meer ervaringen in elke tijdstap te verzamelen welke het leren van een handelsprocedure kan versnellen. Als we gebruik maken van dit "één handelsprocedure voor allen", dan selecteren agenten acties volgens dezelfde handelsprocedure, maar hun acties kunnen verschillen omdat ze verschillende situatie-specifieke inputs verkrijgen. Voor zulke gevallen kunnen we paden gegenereerd door verschillende agenten individueel gebruiken om de gedeelde waarde-functie aan te passen. Soms, echter, bestaat er een interactie-punt (IP) tussen paden van agenten zoals een toestand welke door beiden bezocht is. In zulke gevallen kunnen we paden verbinden en waarden van toestanden welke door één agent voor het IP zijn bezocht, aanpassen door gebruik te maken van waarden van toestanden welke door de andere agent bezocht zijn na het IP. Op deze manier verzamelen we veel meer ervaringen om van te leren, hetgeen de leersnelheid wellicht groter maakt. We beschrijven een implementatie van dit Team Q(λ) algoritme en evalueren het in een niet-coöperatieve doolhof-taak waarin we gebruik maken van verschillende aantallen agenten. De resultaten konden niet aantonen dat de methode de leersnelheid verbetert. Een reden hiervoor is dat onze implementatie verbindingen toestaat tussen een toestand-pad van één agent met een ander toestand-pad welke al een hele lange tijd geleden gegenereerd is door een mogelijkerwijze veel slechtere handelsprocedure waardoor aanpassingen in de waarde-functie onterecht kunnen zijn. Dus is er een trade-off tussen de kwantiteit en kwaliteit van leervoorbeelden.

Model-gebaseerd Reinforcement Leren

Q-leren en Q(λ) methoden zijn directe RL methoden welke de waarde-functie direct schatten aan de hand van gegenereerde toestand-paden. Indirecte of model-gebaseerde RL methoden leren eerst een model van het MBP en gebruiken dan DP algoritmes om de waarde-functie te berekenen. Wanneer DP algoritmes op een online manier gebruikt worden, zijn ze erg langzaam omdat ze de waarde-functie na elke ervaring herberekenen en dat doen door over alle toestanden te itereren. Om DP algoritmes te versnellen, zijn er algoritmes die regelen welke aanpassings-stappen verricht moeten worden na elke ervaring zodat enkel de meest bruikbare aanpassingen gemaakt worden. Eén van de meest efficiënte methoden is Prioritized Sweeping (PS) welke prioriteiten aanwijst voor het aanpassen van waarden van verschillende toestanden. Deze prioriteiten worden bepaald aan de hand van een schatting van de grootte van de aanpassingen. Het oorspronkelijke PS algoritme van Moore en Atkeson (1993) berekent toestand-prioriteiten gebaseerd op de grootste enkelvoudige aanpassing van één van de opvolgende toestanden. Wij beschrijven een alternatieve PS methode welke de exacte groottes van aanpassingen gebruikt om prioriteiten te bereken. Dit kan het probleem van PS verhelpen als er vele kleine aanpassingen verricht worden welke tot een grote aanpassing van de waarde van een toestand leiden, maar welke niet door PS gedetecteerd wordt.

We vergelijken model-gebaseerde methoden met directe RL methoden op een verzameling doolhof-taken en observeren dat model-gebaseerde RL in veel sneller leergedrag resulteert en veel betere handelsprocedures oplevert. Tenslotte observeren we dat ons PS algoritme betere resultaten behaalt dan het originele PS algoritme.

Exploratie

Als we de handelsprocedure van een agent testen door altijd de actie met de grootste waarde te selecteren, dan kan dat uiteindelijk leiden tot een suboptimale handelsprocedure welke een groot deel van de toestandruimte nog nooit of nauwelijks bezocht heeft. Daarom is het belangrijk dat de agent ook exploratie acties selecteert en haar/zijn kennis van de omgeving probeert te verbeteren. Gewoonlijk kosten exploratie acties een bepaalde som aan onmiddellijke beloningen. Daarom, als een agent haar/zijn som der beloningen welke in haar/zijn beperkte levensduur verkregen wordt wil maximaliseren, staat het voor het probleem om zo weinig mogelijk tijd aan exploratie te verspillen welke echter wel voldoende is om een goed belonende handelsprocedure te vinden (het exploratie/exploitatie dilemma). Indien we echter meer geïnteresseerd zijn in het zo snel mogelijk leren van een handelsprocedure welke een bepaald prestatie-nivo levert, dan geven we niks om de beloningen welke in de tussentijd verkregen worden en dus is enkel exploratie belangrijk.

Exploratie methoden kunnen ingedeeld worden in *niet-gerichte* exploratie methoden welke pseudo-random generatoren gebruiken om zo te proberen nieuwe interessante ervaringen te genereren en *gerichte* exploratie methoden welke extra informatie gebruiken zoals de kennis wat de agent al gezien of geleerd heeft om zodoende de agent te sturen naar interessante onbekende gebieden.

Gerichte exploratie methoden kunnen geconstrueerd worden door een exploratie belonings functie te construeren waarmee we een exploratie waarde-functie kunnen leren voor het schatten van globale informatie verdiensten voor het verrichten van bepaalde acties. Hoewel we alle RL-methoden kunnen gebruiken voor het leren van een exploratie waarde-functie, stellen we voor om model-gebaseerd RL hiervoor te gebruiken. Tenslotte, om enkel op toestand/actie paren te concentreren welke deel uit kunnen maken van de optimale handelsprocedure, introduceren we MBIE, een algoritme welke gebruik maakt van de standaard deviatie voor het berekenen van optimistische waarde-functies. De resultaten van experimenten met doolhof-taken bestaande uit meerdere doeltoestanden tonen dat gerichte exploratie met model-gebaseerd RL ongerichte exploratie duidelijk overtreft. Voorts demonstreren de resultaten dat MBIE gebruikt kan worden om bijna altijd haast optimale prestatie nivo's te behalen en verder efficiënt omgaat met het exploratie/exploitatie dilemma.

Gedeeltelijk Waarneembare Problemen

In de wereld dragen de inputs van de sensoren van de agent niet altijd genoeg informatie over om de exacte huidige toestand van de omgeving af te leiden. Daarom zal de agent soms onzeker zijn over de daadwerkelijke toestand en dat maakt het besluitsprobleem veel moeilijker. Zulke problemen worden gewoonlijk Gedeeltelijk Waarneembare Markov Besluits Problemen (GWMBP'en) genoemd en zelfs het oplossen van deterministische GWMBP'en is NP-moeilijk. Voor het oplossen van zulke problemen moet de agent vorige gebeurtenissen en een bepaalde vorm van korte-termijn geheugen gebruiken om inputs van de ambiguïteiten te ontdoen. Exacte methoden gebruiken een geloofs toestand; een vector welke de kansen representeert dat de echte toestand van de omgeving gelijk aan elk van de mogelijke toestanden is. Daarna gebruiken ze DP algoritmes om optimale oplossingen in de geloofs toestandruimte te berekenen. Exacte methoden zijn computationeel onbruikbaar als er teveel toestanden zijn, omdat ze te langzaam zijn. Daarom zijn we geïnteresseerd in heuristieke algoritmes welke gebruikt kunnen worden om grotere problemen op te lossen, maar welke niet zeker optimale oplossingen vinden. We beschrijven een nieuw algoritme, HQ-leren, welke goede handelsprocedures kan leren voor grote deterministische GWMBP'en. HQ deelt GWMBP'en op in een opeenvolging van makkelijkere deelproblemen welke opgelost kunnen worden door geheugenloze handelsprocedures welke leerbaar zijn door reactieve subagenten. De decompositie van de taak in subtaken en het leren van handelsprocedures voor elke subtaak wordt gedaan door twee samenwerkende Q(λ)-leerregels. Experimenten op onder andere een grote doolhof-taak bestaande uit 960 toestanden en slechts 11 verschillende ambigue inputs, tonen dat HQ-leren snel goede of zelfs optimale oplossingen kan vinden voor bepaalde moeilijke problemen.

Functie Approximatie in RL

Een belangrijk thema in RL is de toepassing van functie approximatie voor het leren van waarde-functies voor continue of zeer grote toestandruimtes. We beschrijven drie verschillende functie approximatoren: lineaire netwerken, neuraal gas, en CMACs en beschrijven hoe ze gecombineerd kunnen worden met directe en model-gebaseerde RL-methoden. Dan beschrijven we een gesimuleerde voetbal omgeving welke we gebruiken om de leercapaciteiten te testen van de verschillende functie approximatoren en RL methoden. Het doel is om goede voetbal strategieën te leren tegen een voorgeprogrammeerde tegenstander. De resultaten tonen dat de lineaire netwerken en neurale gas architecturen getraind met Q(λ) problemen hadden om voortdurend hun handelsprocedures te verbeteren. Soms stortte de prestatie in door een catastrofale leerinterferentie. CMACs was meer stabiel, hoewel CMAC-modellen, de combinatie van CMACs en model-gebaseerd RL de enige methode was welke daadwerkelijk goede prestaties leverde en in staat was enkele goede tegenstanders te kloppen.

Conclusies

In dit proefschrift hebben we aangetoond dat RL gebruikt kan worden om snel goede oplossingen voor verschillende soorten problemen te vinden. De beschreven methoden hebben geen model van de wereld nodig, maar leren een handelsprocedure door te interacteren met de (echte) omgeving. Model-gebaseerde is de meest efficiënte leermethode omdat het de door de experimenten verkregen data geheel gebruikt. We bevestigen dus dat het praktisch nut van reinforcement leren erg groot is en verwachten dat het onderzoeksveld aanzienlijk aan belang zal toenemen in de komende jaren.

Acknowledgments

This thesis would never have seen the light if my promotor Frans Groen would not have gone through so many efforts to put things in order. I am very thankful for this and have greatly appreciated watching and listening to his sharp hawk-eye views flying over the scientific fields.

My co-promotor, supervisor and boss Jürgen Schmidhuber has always given me the opportunity to explore interesting ideas which autonomously created themselves and have been astonishing my mind and computer. Without Jürgen's careful editing, continuous stimulation and pragmatic reasoning, these ideas would never have crystallized themselves and find their way to our respected scientific audience. I'm therefore very grateful to Jürgen. Maybe different from many other Ph.D students, I've never felt like a slave. Instead, I am very fortunate that I could participate in Jürgen's very interesting and enjoyable research group.

I have had a great time in Lugano and in IDSIA. I very much enjoyed the parties presented by my good friend Rafal Salustowicz. There are few friends willing to give away their last food, especially if you are very hungry and the next day all shops are closed. I was fortunate to know Rafal and his lovely wife Malgorza who are like that. Furthermore my discussions with Rafal during coffee breaks were often so stimulating that my research has highly profited from them, since they created new challenging views which I then carefully transmitted to computer land, resulting in a positive research feedback loop.

I also want to thank Marcuso Hoffman, without whom life in IDSIA would never have been so pleasant. Marcuso is one of the few people who are always in a good mood. He even stays reasonable when the cappuccino machine blows up! I will certainly miss Marcuso's presence, Italian pasta, and number one cappuccio.

Nic Schraudolph has always been a very interesting and nice guy to talk to. He can tell you everything you always wanted to know about bugs, viruses, internet worms, neural networks, and so much more. He reminds me of a story of a scientist who had a theory which told that large brains imply greater intelligence. Once this scientist received five brains of German professors (after they had died) and found out that their brains were of moderate size. His conclusion was that the professors were not so intelligent after all. I have to say that this will not happen to Nic (since he sure has a large brain).

Felix Gers and his girlfriend Mara also helped creating a blissful time in Lugano. I really liked Felix' curiosity, way of talking, and great laugh. Hitting the road with Felix and Mara have always lead to a lot of joy!

Thanks to Marco Dorigo for his comments on my thesis and for helping me develop new scientific interests. I also want to thank the other Idsiani, thousand times thanks for the Cure to Monaldo, grazie mille for all his kindness to Marco Z., thanks to Gianluca and Cristina for interesting discussions, thanks to Giuseppe for putting robots in space, thanks a lot to Jieyu for his remarkable presence and sharing many research interests, thanks to Eric for cursing his computer and not mine, thanks to Andreas for his interesting stories about kangaroos playing cricket (although I could not always follow them exactly since they were spoken in fast Italian), thanks to Fred for the great poker night(s), and thanks to Nicky, Sandro and Giovanni for everything. Finally thanks to Luca for his humor and fruitful efforts to make a great lab from IDSIA, thanks to Carlo for his patience, support, and efforts to keep things going. Last, but not least thanks to Monica, for her endless help and devotion to settle all important issues outside the field of research.

I want to thank Ben Kröse for his everlasting help starting many years back while I was his student. I am indebted to him for putting me on the road of science and happily look back to the pleasant times enjoyed in his company. I'm also grateful to Stephen and Nikos for our friendly discussions in the university of A'dam. Thanks also to Maria who helped me to see life in a very positive way and the happy times we shared together. She helped me becoming less artificial and much more artistic. Thanks to Flower for our everlasting friendship and all good times we have shared together. Many thanks go to my parents for their everlasting support and love.

Finally, I want to express my warm feelings to René Wiering for his great friendliness and hospitality. I have always enjoyed and profited from our mind breaking discussions pursuing the goal to better understand ourselves and our life. I really hope these discussions will never stop, but continuously evolve and become truth.