Reinforcement Learning under Space and Time Constraints

Harm van Seijen

The cover pictures a robot, representing a reinforcement learning agent, bounded by a rectangle, representing the computational time and memory space constraints.
Concept: Harm van Seijen
Drawings (front and back): Robert-Jan van Seijen

ISBN 978-90-5986-394-1

Reinforcement Learning under Space and Time Constraints

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 8 december 2011, te 10:00 uur

door

Harm Hendrik van Seijen

geboren te Leeuwarden

Promotiecommissie

Promotor:	Prof. dr. ir. F. C. A. Groen
Co-promotoren:	Dr. S. Whiteson
	Dr. ir. L. J. H. M. Kester
Overige leden:	Prof. dr. P. Adriaans
	Prof. dr. R. Babuska
	Prof. dr. P. M. A. Sloot
	Prof. dr. R. S. Sutton
	Dr. M. A. Wiering

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

This research has been performed at the Distributed Sensor Systems group of TNO and the Intelligent Autonomous Systems group of the University of Amsterdam and is part of the enabling technology program Adaptive Multi Sensor Networks (AMSN), supported by TNO, and the Interactive Collaborative Information Systems (ICIS) project, supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024.

La Clairvoyance - by *René Magritte*

"Man had always assumed that he was more intelligent than dolphins because he had achieved so much — the wheel, New York, wars, and so on — whilst all the dolphins had ever done was muck about in the water having a good time. But conversely, the dolphins had always believed that they were far more intelligent than man — for precisely the same reasons."

Excerpt from 'The Hitchhiker's Guide to the Galaxy' - *by Douglas Adams*

Contents

CHAPTER 1

Introduction

Artificial Intelligence (AI) is the research field that is concerned with understanding and building intelligent systems. Compared to classic fields like physics or mathematics it is a very young field. While ancient philosophers already laid some of the groundwork in their attempts to describe the process of human thinking, it was not until the early 40s of the previous century, with the invention of the digital computer, that the field really took off. The name 'artificial intelligence' was coined in 1956, at a conference on the campus of Dartmouth College. Despite its relative young age, AI is a very broad field, with a large variety of subfields. Examples are reasoning, knowledge representation, natural language processing, computer vision, planning and machine learning.

The topic of this thesis falls within the subfield of machine learning. Informally, a computer program is said to learn from experience if its performance over some set of tasks improves with experience, according to some performance measure. Machine learning can be divided into several subfields, such as supervised learning, unsupervised learning and reinforcement learning. The goal of supervised learning (Vapnik, 1995; Bishop, 2006) is to learn an input-output relation, given a set of training examples, consisting of input data with an output label attached to them. A supervised learning algorithm needs to generalize from these examples in order to correctly predict the label of data not present in the training set. An example is the classification of handwritten text (Schomaker, 1993). The goal of unsupervised learning (Hartigan, 1975; Barlow, 1989) is to find certain patterns in unlabeled data, for example to achieve dimensionality reduction or clustering. The purpose of this can be to efficiently communicate the inputs, predict future inputs or build a representation for decision making. The goal of reinforcement learning (RL) (Bertsekas and Tsitsiklis, 1996; Kaelbling et al., 1996; Sutton and Barto, 1998), the topic of this thesis, is to learn control behavior for a sequential decision task with unknown dynamics. As with supervised learning, there is a feedback signal used to improve the behavior. However, in contrast to supervised learning, an example of correct behavior is never given. Instead, single decisions result in a positive or negative reward signal and by a trial and error process behavior is learned that maximizes the total received reward. Many problems can be modeled as an RL problem. Successful applications of RL include building a backgammon playing agent (Tesauro, 1994), robotics (Lin, 1993) and elevator control (Crites and Barto, 1998).

1.1 Reinforcement Learning

In this section, we provide a brief overview of some important elements of the reinforcement learning problem and its solution strategies. The purpose is mainly to provide the

necessary background for understanding the different topics of this thesis, summarized in Section 1.2.1. In Chapter 2, we provide a more detailed overview of reinforcement learning.

1.1.1 The Reinforcement Learning Problem

An RL problem (Kaelbling et al., 1996; Sutton and Barto, 1998) is a task that can be described in terms of an *agent* interacting with its *environment*. At discrete timesteps, the agent selects an *action* and observes the resulting new environment *state* as well as a *reward*. In the general case, the resulting reward and next state are uncertain, that is, they are drawn from a probability distribution. Typically, the goal of an RL agent is to improve the expected *return*, which is the (discounted) sum of rewards over the different timesteps.

A key aspect of an RL problem is the immediate versus delayed reward consideration. To obtain a high return, not just the immediate reward of an action has to considered, but also the next state, since this determines what future rewards can be obtained.

Another key aspect is that the effect of each action, i.e., the associated probability distribution over rewards and next states, is initially unknown. Therefore, the agent needs to interact with the environment in order to learn which actions are best. This leads to a practical dilemma often referred to as the *exploration-exploitation* dilemma: the agent can either exploit its current knowledge by taking the action that predicts the highest expected return, or explore by taking a different action in order to improve the accuracy of the prediction for that action, and so improve its future action selections for the current state.

An RL problem is said to obey the *Markov property* if the probability distribution for the reward and next state of an action only depends on the current state and not on the history. If this is the case, the RL problem can be modeled as a *Markov decision Process* (MDP) (Puterman, 1994). An MDP formally defines an RL problem by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the set of all states, $\mathcal{A}$ is the set of all actions, $\mathcal{P}$ gives the transition probability from state $s \in \mathcal{S}$ to state s' for each action $a \in \mathcal{A}$, $\mathcal{R}$ is the reward function, giving the expected reward when action a is taken in state s and γ is the discount factor, which specifies how future rewards should be weighted with respect to the immediate reward.

For an MDP, the *policy* of an agent, which defines its behavior, can be expressed as a mapping from each state it may encounter to a probability distribution over the available actions. Each MDP contains at least one *optimal policy*, which is a policy whose expected return is maximal. We refer to the set of all possible policies that can be defined for an MDP as the *policy space* associated with that MDP.

1.1.2 Solution Strategies

In this thesis we focus on *value-function based RL methods*, which use *value functions* (Bellman, 1956) to improve their policies. The action-value, or Q-value function $Q^\pi(s, a)$ of a policy π gives the expected return when the agent takes action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ and follows policy π thereafter. Many value-function methods try to approximate the optimal Q-value function, which is the Q-value function corresponding to an optimal policy, by

iteratively improving an estimate of this function. Once the optimal Q-value function has been determined, an optimal policy can be constructed by taking actions that are greedy with respect to this function.

Value-function methods can be divided into *model-free* and *model-based* methods. Model-based methods (Sutton, 1990; Moore and Atkeson, 1993; Brafman and Tennenholtz, 2002; Kearns and Singh, 2002; Strehl and Littman, 2005; Diuk et al., 2009) use the experience samples obtained from interaction with the environment to update an estimate of the environment model, i.e., the functions $\mathcal{P}$ and $\mathcal{R}$. Using this model, off-line techniques, such as dynamic programming (Bellman, 1957), are then used to determine an estimate of the optimal Q-value function. On the other hand, model-free methods (Sutton, 1988; Watkins, 1989; Rummery and Niranjan, 1994; Sutton, 1996; Strehl et al., 2006) use the experience samples to directly update a Q-value function. The space requirements of model-based methods are typically a lot higher than that of model-free methods, since they require storage of the model. However, the advantage is that experience can be re-used, which improves the sample complexity, i.e., the number of environment samples required to obtain a good policy.

Besides the model-free/model-based categorization, value-function methods can either be *on-policy* or *off-policy*. For on-policy methods the *behavior policy*, i.e, the policy that generates the experience samples, is equal to the *estimation policy*, i.e., the policy whose Q-value function is being estimated and improved. For off-policy methods, on the other hand, the behavior policy is different from the estimation policy. Both method types have their advantages and disadvantages (see Chapter 2 for details).

A classic off-policy, model-free method is *Q-learning* (Watkins, 1989), which is based on the common model-free update rule

$$Q_{t+1}(s_t, a_t) \leftarrow (1-\alpha) Q_t(s_t, a_t) + \alpha \, \upsilon_t \,, \tag{1.1}$$

where s_t is the state visited at timestep t, a_t is the action taken at that timestep, α is the learning rate and υ_t is the update target. For Q-learning, this update target is

$$\upsilon_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') \,,$$

where r_{t+1} is the reward received after taking action a_t in s_t, and s_{t+1} is the next state observed.

A classic on-policy method is *Sarsa* (Rummery and Niranjan, 1994; Sutton, 1996). Sarsa is also based on Equation 1.1, however the update target for Sarsa is

$$\upsilon_t = r_{t+1} + Q_t(s_{t+1}, a_{t+1}) \,,$$

where a_{t+1} is the action taken at timestep $t+1$.

For methods like Q-learning and Sarsa, new experience only affects the Q-value of the state-action pair that generated this experience. This can cause slow learning, especially in sparse reward tasks. *Eligibility traces* (Sutton, 1988; Watkins, 1989) is a popular technique that can be combined with Q-learning and Sarsa to propagate new information faster through an MDP, improving performance. It achieves this by not only updating the

Q-value of the state-action pair that generated the experience sample, but also recently visited state-action pairs, in proportion to a trace parameter. This trace parameter is decayed according to the trace decay parameter λ, causing the effect to be larger for recently visited state-action pairs.

1.2 Focus of this Thesis

An RL agent needs to interact with the environment in order to gain knowledge about it and improve its policy. A typical performance measure is the average return an agent accumulates during learning. Using this measure, agents that learn fast, i.e., that require only a small number of environment interactions to obtain a good policy, will have a high performance.

Besides the rate of policy improvement with respect to the number of environment interactions, there are two additional performance parameters that play an important role when evaluating a method. These are the computational cost for processing a newly obtained environment sample and the space requirements of a method, i.e., the physical memory (RAM) needed to store data.

The computational cost is important, since the most interesting RL problems often require a high frequency of action selection, limiting the computation that can be done in between actions. Think for example of the task of dynamic robot walking or balancing an inverted pendulum in real-time. Clearly, in these domains a fast reaction cycle is essential. To effectively operate in these domains, an RL agent should process each new sample as efficiently and effectively as possible. The time constraint in the title of this thesis reflects this idea; it refers to the computational time available in between observing a sample and selecting the next action, i.e., the time available to process a sample.

Besides this time constraint, the agent faces a space constraint, i.e., a constraint on the size of the physical memory (RAM) that is available to the agent to store data. When memory is abundant, the agent can store a full model estimate of the environment, enabling full re-use of data, which in general improves performance . However, the (memory) space complexity this requires is quadratic in the size of the state space. Therefore, for large tasks, storing the full model is often infeasible and choices have to be made about which data to store.

In the thesis, we use the term ‘performance’ exclusively to indicate the return per episode (or average reward per environment interaction), while the computational power and memory space we treat as resources. The general focus of this thesis is on optimal exploitation of these resources, that is, on methods that get the best performance under certain space and time constraints. We perform research on several topics related to this. The next section discusses these topics.

1.2.1 Topics

Below, we discuss the six different topics related to RL under space and time constraints that are addressed in this thesis.

Analysis of Expected Sarsa

The optimal learning rate α of a method based on Equation 1.1 is the learning rate that yields the highest performance. This optimal value is a trade-off between two processes. On the one hand, a high learning rate means the effect of updates is larger, resulting in faster policy improvements. On the other hand, a low learning rate means that update targets are better averaged, hence more accurate value estimates can be obtained.

In contrast to Q-learning, the on-policy method Sarsa performs updates using an update target based on the action that is selected for execution at the next state. This causes additional variance in the update target, since the selection policy is in general stochastic. By using a variation of the Sarsa update rule that uses the expectation over all actions instead of the selected action, a lower variance in the update targets can be achieved. This allows for higher learning rates and thus faster learning.

Though the variation of Sarsa using this update rule, which we call Expected Sarsa, appears to have better theoretical properties and is mentioned several times in the literature (Rummery, 1995; Sutton and Barto, 1998), it is not widely used in practise and no systematic study of it can be found. For this reason, we perform an extensive theoretical and empirical analysis of Expected Sarsa to assess its merits.

Just-In-Time Learning

Methods like Q-learning and (expected) Sarsa use a sample immediately after it is observed to update the Q-value of the corresponding state-action pair. However, storing the sample and postponing the corresponding update can potentially improve performance, due to a more accurate update target, as the value estimates of other states or state-action pairs involved in the update may have improved in the meantime. Postponing the update for too long can have a negative overall effect though, since the action selection process of other updates based on this Q-value might use an outdated value. If an update is postponed till just before it is needed, the negative effects are avoided, while the positive effects due to more accurate update targets are still present. We call this type of learning *just-in-time learning* and perform an empirical and theoretical analysis of it.

Eligibility Traces Improvements

For Sarsa the variance due to policy stochasticity can be fully removed by using the expectation over actions in the update target. There is no straightforward extension of this principle to eligibility traces. In other words, when Sarsa is combined with eligibility traces, the extra variance due to policy stochasticity results in a lower optimal learning rate, reducing the performance advantage due to faster information propagation. On the other hand, the combination of eligibility traces with Q-learning is also not ideal. For the off-policy implementation (Watkins, 1989) the traces have to be reset, whenever a non-greedy action is taken, limiting the propagation of information; for the implementation that does not reset the traces (Peng and Williams, 1996), the same variance issues occur as with the Sarsa implementation of eligibility traces. We investigate whether variants of Sarsa and/or Q-learning can be constructed that exploit the same principle behind eligibility traces, but do so at a lower variance and without resetting traces. These variants can potentially result in higher performance, since they enable the use of higher learning rates.

Integrating Model-Free and Model-Based Learning

When performing value-function based RL, two major classes of methods are model-free and model-based learning. The difference in space requirements between these two classes can be huge. While typical model-free methods have a space complexity that is linear in the size of the state space, model-based methods are bounded by a space complexity that is quadratic in the state space size. This huge difference forms a disadvantage, since in a practical situation an RL agent gets assigned a certain amount of resources - in terms of computation time and memory space - and ideally should fully exploit these resources to get the maximum performance. When the agent has to resort to model-free methods when there is not enough space available for storing the full model, it does not optimally exploit its space resources and misses out on an opportunity for a better performance.

A practical approach that is sometimes used to address the gap in space requirements between model-free and model-based methods is to use sparse, approximate models that require only a fraction of the space used by full model-based methods. However, this is not ideal, since the performance of such methods is bounded by the quality of the model approximation (Kearns and Singh, 1999). Furthermore, since the models may remain incorrect regardless of how much sample experience is gathered, such methods are not guaranteed to find optimal policies even in the limit. We investigate whether it is possible to combine model-free with model-based learning in such a way that value-function methods can be constructed that provably converge to the optimal Q-values, and that have a space complexity anywhere in between that of model-free and model-based methods.

Representation Selection

In a *factored MDP* (Boutilier et al., 1995), wherein each state is described by a set of state feature values, prior knowledge about independence between such features can be expressed using dynamic Bayesian networks (DBNs) (Dean and Kanazawa, 1989). In learning problems, DBNs enable near-optimal performance using only samples and computation polynomial in the number of parameters of the DBN, which may be exponentially smaller than the number of states (Kearns and Koller, 1999).

Unfortunately, in many real-world problems, the DBN structure is not known in advance and must also be learned. Doing so is also possible in a sample-efficient way, given prior knowledge of the maximum degree of the DBN (Li et al., 2008; Diuk et al., 2009; Kroon and Whiteson, 2009). However, the space and time requirements for such methods is linear in the number of states, making them impractical for large problems.

In this thesis, we propose an alternative approach for exploiting structure in MDPs. Rather than learning the structure and parameter values of a DBN, our approach learns which representation among a set of *candidate representations* yields the highest expected return. Each candidate representation consists of a subset of the available state features. In general, the number of candidate representations can be prohibitively large. However, in many real-world settings, prior knowledge about the task can be used to deduce a small set of candidate representations.

Exploiting Policy Restrictions

A natural form of prior knowledge is knowledge in the form of *policy restrictions*, i.e., re-

strictions on the policy set that should be considered when searching for an optimal policy. In case of policy-search RL, that is, RL methods that search for the optimal policy directly in the policy space (Moriarty et al., 1999; Stanley and Miikkulainen, 2002), the advantage of a smaller policy set is obvious. However, policy-search method excel in different task domains than value-function methods (Whiteson et al., 2010; Kalyanakrishnan and Stone, 2011). Therefore, for task domains where value-function methods are superior, we would like to use value-function methods but still be able to exploit prior knowledge about policy restrictions. We investigate how prior knowledge about policy restrictions can be efficiently represented and exploited in value-function based RL.

1.2.2 Research Questions

For each of the six topics discussed in the previous section, we formulate a central research question, listed below, which guides the research on the corresponding topic. At the end of this thesis we come back to these questions.

- **Analysis of Expected Sarsa**: Under which settings does Expected Sarsa outperform regular Sarsa?
- **Just-In-Time Learning**: Does just-in-time Q-learning have a guaranteed performance improvement over regular Q-learning?
- **Eligibility Traces Improvements**: Is it possible to construct a strategy with similar space and time requirements to those of eligibility traces that consistently outperforms it?
- **Integrating Model-Free and Model-Based Learning**: Is it possible to construct methods with a space complexity between $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ that provably converge to the optimal Q-values?
- **Representation Selection**: Under which conditions can convergence to the optimal Q-values be guaranteed, when representation selection is applied?
- **Exploiting Policy Restrictions**: How can a reduced policy space be exploited in value-function based RL?

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 is a background section, in which relevant theory about reinforcement learning is explained. In Chapter 3, Expected Sarsa is evaluated and just-in-time learning is introduced. Chapter 4 discusses *best-match learning*, a new type of learning that trades off the sample efficiency of model-based methods with the space efficiency of model-free methods. In Chapter 5, problem size reduction by representation selection is discussed, while Chapter 6 discusses how policy restrictions can reduce the problem size. Finally, in Chapter 7, the research questions formulated in Section 1.2.2 are addressed and the three most promising avenues of future work are discussed.

CHAPTER 2

Background

In this chapter we provide the relevant background for this thesis. Section 2.1 discusses the reinforcement learning problem, while Section 2.2 discusses solution strategies.

2.1 The Reinforcement Learning Problem

A reinforcement learning (RL) problem (Kaelbling et al., 1996; Sutton and Barto, 1998) is a task that can be described in terms of an *agent* interacting with an (initially) unkown *environment*. While in the general case this interaction can be of a continuous nature, most problems can be effectively described using a discrete time scale, in which case interaction occurs at timesteps $t = 0, 1, 2, ...$ At timestep t, the agent executes action a_t in environment state s_t, and observes, at the next timestep, the resulting reward r_{t+1} and next state s_{t+1}. The reward and next state are generally uncertain, that is, they are drawn from a probability distribution. Typically, the goal of an RL agent is to improve the expected *return*, which is the sum of rewards over the different timesteps. Usually, this is a weighted sum, in which rewards further away in the future are given a lower weight than immediate rewards.

The simplest example of a reinforcement learning problem is the multi-armed bandit problem, which we discuss next.

2.1.1 The (Contextual) Multi-Armed Bandit Problem

A *multi-armed bandit problem* (Lai and Robbins, 1985; Auer et al., 2002) is a task in which repeatedly a choice has to be made between the same set of actions. The action produces a reward drawn from an unknown probability distribution corresponding to that action, and the goal is to maximize the total reward over a series of action selections. The term 'multi-armed bandit' is derived from the analogy to a slot machine (traditionally called a 'one-armed bandit') with multiple arms instead of one.

The fact that the distribution over rewards for each action is initially unknown leads to a dilemma that is typical to reinforcement learning and is often referred to as the *exploration-exploitation* dilemma. To illustrate this dilemma, consider the task of optimizing the total reward over a sequence of 100 action selections. If the average reward for each action would be known in advance, the best strategy would simply be to always take the action with the highest average reward. However, since this is not the case, the agent has to choose each time to either exploit its current knowledge and select the action that is optimal according to its current average reward estimates, or to explore and select a different action in order to improve the estimate for that action and hence future action selections.

A *contextual* multi-armed bandit problem is an extension of the multi-armed bandit problem for which the reward distribution of the bandit arms is correlated with observed context information. This context information is generated by a fixed probability distribution and changes after each arm pull. This problem maps to an RL problem by interpreting the arms as actions and the context information as states. The task basically boils down to learning the average reward of each arm conditioned on the context information. The optimal policy is simply a policy that selects at each moment the arm with the highest expected reward given the current context information.

A real-life example of a contextual multi-armed bandit problem is the task of placing ads on web pages (Langford and Zhang, 2007; Langford et al., 2008). Since companies that serve ads are typically paid per click, the goal is to select the ads that maximize the chance of being clicked. This task can be modeled as a contextual bandit problem wherein available ads are actions, web pages are states, and rewards are payments for clicked ads.

For both the multi-armed bandit and the contextual multi-armed bandit problem the agent does not have to worry about the next state when selecting an action. To maximize its total (expected) reward, the agent simply has to take the action with the highest expected reward at each timestep. This changes when the probability distribution for the next state also depends on the action that is taken. If this is the case, an agent optimizing the sum of rewards cannot simply take the action with the highest immediate reward. Instead, it should also consider the next state when selecting an action, since this determines what future rewards the agent might receive. In the next section, we introduce a mathematical framework for describing an important class of such *sequential decision problems*.

2.1.2 Markov Decision Processes

If the agent's actions not only affect the immediate reward, but also the next state, the RL problem becomes a *sequential decision problem*. This is a much more complicated problem than a bandit problem, since the agent now has to take into account the next state as well, when determining the best action. A real-life example of such a problem is the task of minimizing the time people have to wait for an elevator (Crites and Barto, 1998).

In this thesis, we focus on sequential decision problems for which the *Markov property* holds. This property states that the probability distribution for the reward and next state of an action a_t, only depends on the current state s_t, and not on the history. More formally stated:

$$P(s_{t+1} = s', r_{t+1} = r | s_t, a_t) = \\ P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0) \,. \quad (2.1)$$

This is a very useful property, since it allows the RL problem to be modeled as a *Markov Decision Process* (MDPs) (Puterman, 1994), a particularly effective and compact representation. An RL problem that does not have the Markov property can often be transformed in one that does by using a different state definition.

An MDP can be described by a tuple of the form $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ consisting of

- $\mathcal{S}$, the set of all states.

- $\mathcal{A}$, the set of all actions.
- $\mathcal{P}_{sa}^{s'} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, the transition probability from state $s \in \mathcal{S}$ to state s' when action $a \in \mathcal{A}$ is taken.
- $\mathcal{R}_{sa} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, the reward function giving the expected reward r when action a is taken in state s.
- $\gamma \in [0, 1]$, the discount factor, controlling the weight of future rewards versus that of the immediate reward.

In this thesis we mainly focus on RL problems for which $\mathcal{S}$ and $\mathcal{A}$ are discrete and finite. For an RL problem, the environment dynamics and reward function, i.e., $\mathcal{P}_{sa}^{s'}$ and $\mathcal{R}_{sa}$, are initially unknown.

The agent selects its actions at discrete timesteps $t = 0, 1, 2, ...$ according to a *policy* π. A *stochastic* policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ defines for every action the selection probability conditioned on the state. A *deterministic* policy is a special case of a stochastic policy, where for each state there is one action with selection probability 1, while the other actions have probability 0. For a deterministic policy we use the function definition $\pi : \mathcal{S} \to \mathcal{A}$, which maps every state to a single action.

The goal of RL is to improve the agent's policy in order to increase the *return* R received by the agent, which is the discounted cumulative reward

$$R_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + ... = \sum_{k=1}^{\infty} \gamma^{k-1}\, r_{t+k}\,, \tag{2.2}$$

where r_{t+1} is the reward received after taking action a_t in state s_t at timestep t.

In this thesis, we only consider *stationary* MDP problems with *infinite horizon*. This means that the MDP the agent interacts with does not change during learning and that the agent's interaction is not terminated after some fixed number of timesteps (which would make the optimal policy time-dependent). We do consider MDPs with *terminal states*, which divide the agent's environment interaction into *episodes*. When a terminal state is reached, the current episode ends and a new one is started by resetting the environment to the initial state. If the agent starts at state s_t and reaches the terminal state at timestep T, then the total return for s_t is defined as:

$$R_t = \sum_{k=1}^{T-t} \gamma^{k-1}\, r_{t+k} \tag{2.3}$$

This finite sum can be related to the infinite sum of Equation 2.2 by interpreting terminal states as states with only a single action with zero reward that points to itself.

A (contextual) multi-armed bandit problem can be viewed as a special case of an episodic MDP problem, for which each episode ends after only a single action. For a regular multi-armed bandit problem the initial state is always the same, while for a contextual multi-armed bandit problem, the initial state is drawn from a (fixed) probability distribution over different states. Note that the Markov property always holds in case of a (contextual) multi-armed bandit problem, since there is no history to consider.

An example of an episodic MDP task that is used several times in this thesis is the Dyna maze task (Sutton, 1990), a navigation task in which the agent has to move as fast as possible from a start location to the goal location (see Figure 2.1). The agent can choose at each timestep between four actions: up, down, left and right. Each action deterministically moves the agent one square in the corresponding direction, unless the agent hits a wall or the edge of the maze, in which case it stays at the same position. The reward received after each action is 0, except when the goal location is reached, which results in a reward of +1 (and terminates the episode). This discount factor γ is 0.95.

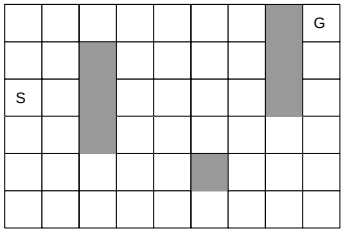

Figure 2.1: The Dyna maze task, in which the agent must travel from S to G. The grey areas represent walls. The reward is +1 when the goal state is reached and 0 otherwise.

This task clearly demonstrates the purpose of the discount factor in the return. Without discounting future rewards (i.e., without setting $\gamma < 1$), there is no incentive for the agent to search for the shortest path, since a random policy (i.e., a policy that selects a random action at each timestep) will eventually also result in a reward of +1. On the other hand, with $\gamma < 1$, an agent aiming for maximum expected return will try to reach the goal location as fast as possible.

There can be different reward functions that lead to the same optimal policy. In case of the Dyna maze task, a reward of -1 for each action (including the action that results in the agent reaching the goal location) results in the same optimal policy. With this reward function the agent also tries to reach the goal location as fast as possible, since reaching the goal location terminates the episode and hence stops the accumulation of negative rewards. With this reward function even a discount factor of 1 is possible.

A stochastic variation of the Dyna maze task can be made, by specifying that, in response to an action, the agent moves with a probability of 10% in an arbitrary direction, instead of the direction corresponding with the action.

2.1.3 Value Functions and the Bellman Equations

The reinforcement learning methods discussed in this thesis are so-called value-function methods, which derive their policy from a value function. Each policy π has a *state-value function* $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ associated with it that gives the expected return from state s, when policy π is followed:

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} \tag{2.4}$$

where $E_X\{\}$ denotes the expected value of a variable conditioned on X. Similarly, there is an *action-value function* $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ associated with it that gives the expected return when action a is taken in state s and policy π is followed thereafter.

$$Q^\pi(s,a) = E_\pi\{R_t | s_t = s, a_t = a\} \tag{2.5}$$

The action-value function is related to the state-value function through the relation:

$$V^\pi(s) = \sum_a \pi(s,a)\, Q^\pi(s,a) \tag{2.6}$$

A fundamental relation that is exploited by many reinforcement learning methods is the *Bellman equation*, which relates the value of a state to the value of its successor states:

$$V^\pi(s) = \sum_a \pi(s,a) \left[\mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V^\pi(s') \right] \tag{2.7}$$

We now discuss the concept of optimality. A policy π is better than or equal to another policy π' if for every state the expected return under π is higher or equal than the expected return under π'. In other words, if $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$. The *optimal policy* π^* is a policy that is better or equal than all other possible policies. A property of an MDP is that there always exists such a policy, i.e., there is always a policy that provides the maximum expected return for *every* state. The value function associated with this policy is the *optimal value function* V^*. Similarly, the *optimal action-value function* Q^* is the action-value function associated with π^*. The Bellman equation relating the optimal value of a state to the optimal value of its successor state is called the *Bellman optimality equation for* V^*:

$$V^*(s) = \max_a \left[\mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V^*(s') \right] \tag{2.8}$$

On the other hand, the *Bellman optimality equation for* Q^* is:

$$Q^*(s,a) = \mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} \max_{a'} Q^*(s',a') \tag{2.9}$$

While there is only a single optimal value function and optimal action-value function for each MDP, there can be multiple optimal policies. The reason is that a state can have multiple actions resulting in the same expected return. If that expected return is the optimal expected return, then an optimal policy can use either of these actions (or some stochastic selection between them). When the optimal action-value function is known, an optimal policy can easily be constructed by always taking the greedy action with respect to Q^*.

The main focus of this thesis is on problems with discrete, finite state spaces. In this case, value functions can be stored in a table, with one entry per state or state-action pair. In the general case, the state space can be continuous, in which case tabular values cannot be used for obvious reasons. In this case, the value function can be approximated with some parameterized function, and only the function parameters have to learned and stored. An advantage of this approach is that experience is generalized. A disadvantage is that finding a good parameterized function for approximation of the value function can be very hard.

2.2 Solution Strategies

In this section, we discuss the basic solution strategies for solving an RL problem modeled as an MDP. All discussed methods are *value-function methods*, meaning they use value functions to improve the policy. We start with describing dynamic programming, which is a *planning* strategy, i.e., the full MDP description $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ is known by the agent. In this case, no interaction with the environment is required. Instead, the optimal policy can be directly computed from the MDP description. The relevance of dynamic programming to reinforcement learning, which assumes the environment model is (initially) unknown, is that it forms the core of many model-based learning methods. These methods update an estimate of the model at each timestep and use dynamic programming to compute a policy based on this model estimate.

2.2.1 Dynamic Programming

Dynamic programming refers to a class of methods that compute the optimal policy of an MDP, given its full description $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. In other words, the agent has perfect knowledge of the environment dynamics.

A popular dynamic programming method is *value iteration* (see Algorithm 1). This method maintains an estimate V_k of the optimal value function V^* and iteratively improves this estimate by performing updates based on the Bellman optimality equation for V (Equation 2.8):

$$V_{k+1}(s) \leftarrow \max_a \left[\mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V_k(s') \right], \qquad \text{for all } s \,.$$

It can easily be proven that in the limit the following holds:

$$\lim_{k \to \infty} V_k = V^* \,.$$

Algorithm 1 Value iteration

initialize $V(s)$ arbitrarily for all s
repeat
  $\Delta \leftarrow 0$
  for all $s \in \mathcal{S}$ **do**
    $V_{old} \leftarrow V(s)$
    $V(s) \leftarrow \max_a \left[\mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V(s') \right]$
    $\Delta \leftarrow \max(\Delta, |V_{old} - V(s)|)$
until $\Delta <$ some small threshold value

2.2.2 Model-Based and Model-Free Learning

In a learning setting, the agent does not know the environment model, i.e., $\mathcal{P}$ and $\mathcal{R}$, and needs to interact with the environment in order to learn about the effect of its actions and improve its policy. We assume in this thesis trajectory-based interactions, where the agent

starts at some initial state and then moves through the environment according to the next states its actions result in.

Two tasks can be distinguished in a learning setting. The first task is referred to as *evaluation*, and involves assessing how good a specific policy π is by determining its value function V^π. The second task is referred to as *control*, and involves (often simultaneously) evaluating and improving some *estimation* policy. The *behavior policy* is the policy that controls the agent, i.e., the policy that generates the samples. A method that uses the estimation policy to control the agent is called an *on-policy* method. In contrast, a method for which the behavior policy is different than the estimation policy is called an *off-policy* method. In the latter case, the behavior policy is often derived from the action-value function of the estimation policy. In Section 2.2.3, we discuss the advantages and disadvantages of on-policy and off-policy methods using two concrete methods.

Every control method needs to have some strategy to deal with the *exploration-exploitation* dilemma. In this thesis, we mainly resort to ε*-greedy* behavior policies to ensure sufficient exploration. These are policies that take with a probability of ε a random action, while the greedy action, i.e., the action with the maximum estimated action value, is taken with a probability of $1 - \varepsilon$. Note that random selection of an action can also result in the greedy action, hence the total selection probability of the greedy action is $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|}$.[1]

Value-function methods can be divided into *model-free* and *model-based* methods. Model-based methods (Sutton, 1990; Moore and Atkeson, 1993; Brafman and Tennenholtz, 2002; Kearns and Singh, 2002; Strehl and Littman, 2005; Diuk et al., 2009) use the experience samples obtained from interaction with the environment to update an estimate of the environment model. Using this model, off-line techniques, such as dynamic programming (Bellman, 1957), are then used to determine an estimate of the optimal Q-value function (or state value function). On the other hand, model-free methods (Sutton, 1988; Watkins, 1989; Rummery and Niranjan, 1994; Sutton, 1996; Strehl et al., 2006) use the experience samples to directly update a Q-value function. The space requirements of model-based methods are typically a lot higher than those of model-free methods, since they require storage of the model. However, the advantage is that experience can be re-used, which improves the sample complexity, i.e., the number of environment samples required to obtain a good policy.

2.2.3 Temporal-Difference Learning

A popular model-free approach is *temporal-difference* (TD) learning (Sutton, 1988), which bootstraps value estimates from other values. Two classic TD methods are Q-learning (Watkins, 1989) and Sarsa (Rummery and Niranjan, 1994; Sutton, 1996). Both methods use a sample immediately after it is observed to update the Q-value of the state-action pair that generated the sample, using the common TD update rule:

$$Q_{t+1}(s_t, a_t) \leftarrow (1 - \alpha) Q_t(s_t, a_t) + \alpha \, v_t \tag{2.10}$$

[1] This assumes there is only one action with a maximum value. If there are multiple actions with a maximum value, the $1 - \varepsilon$ probability for greedy action selection is divided among these actions.

where α is the learning rate (or step size) and v_t is the update target. An alternative formulation of this update rule is

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha\, \delta_t\,, \tag{2.11}$$

where $\delta_t = v_t - Q_t(s_t, a_t)$ is called the *TD error*. For Q-learning, the update target is

$$v_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a)\,, \tag{2.12}$$

while for Sarsa it is

$$v_t = r_{t+1} + \gamma\, Q_t(s_{t+1}, a_{t+1})\,. \tag{2.13}$$

The pseudocode for Q-learning and Sarsa is shown in Algorithm 2 and Algorithm 3, respectively. Note that Sarsa uses a_{t+1}, the action selected and executed at timestep $t+1$, in its update target.[2] This creates a small disadvantage compared to Q-learning for problems with returning actions, i.e., actions for which $s_{t+1} = s_t$, since Sarsa's action selection for timestep $t+1$ does not take into account the update of $Q(s_t, a_t)$ that occurs at this timestep.

Algorithm 2 Q-Learning

initialize $Q(s, a)$ arbitrarily for all s,a
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    select action a, based on $Q(s, \cdot)$
    take action a, observe r and s'
    $Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha\,[r + \gamma\, max_{a'}\, Q(s', a')]$
    $s \leftarrow s'$
  until s is terminal

Algorithm 3 Sarsa

initialize $Q(s, a)$ arbitrarily for all s,a
loop {over episodes}
  initialize s
  select action a, based on $Q(s, \cdot)$
  repeat {for each step in the episode}
    take action a, observe r and s'
    select action a', based on $Q(s', \cdot)$
    $Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha\,[r + \gamma\, Q(s', a')]$
    $s \leftarrow s'$
  until s is terminal

Besides the small disadvantage for returning actions, the main difference between Q-learning and Sarsa is that Q-learning is an off-policy method and Sarsa an on-policy

[2]The name Sarsa is in fact derived from the five components employed in its update rule: the current state and action s_t and a_t, the immediate reward r_{t+1}, and the next state and action s_{t+1} and a_{t+1}.

method. Therefore, Q-learning can employ a stochastic behavior policy (for example a policy that is ε-greedy with respect to the Q-values), ensuring sufficient exploration, while its estimation policy is greedy. This enables that the Q-values of Q-learning converge to the optimal Q-values (under some mild conditions). A disadvantage of Q-learning is that it aims to improve the estimation policy, while the performance criterium for an RL agent is typically related to the return, generated by the behavior policy. This can lead for certain domains to a disadvantage compared to on-policy methods, such as Sarsa (see for example the Cliff walking task in Section 3.1.4.1). A disadvantage of Sarsa is that its Q-values do not directly converge to the optimal Q-values. Therefore, when the goal is to find an optimal policy, additional measures have to be taken, like annealing ε for ε-greedy policies, which can make it more difficult to guarantee sufficient exploration.

2.2.4 Eligibility Traces

For methods like Q-learning and Sarsa, new experience only affects the Q-value of the state-action pair that generated this experience. This can cause slow learning, especially in sparse reward tasks. *Eligibility traces* (Sutton, 1988; Watkins, 1989) is a popular technique that can be combined with Q-learning and Sarsa to propagate new information faster through an MDP, improving performance.

The idea behind eligibility traces is that state-action pairs visited in the past are responsible for the agent being at the current state. Therefore, they are also partly eligible for undergoing learning changes. The more recent a state-action pair is visited, the more eligible it is, that is, the larger its Q-value correction should be.

This idea is implemented by maintaining a variable for each state-action pair, its *eligibility trace* $e(s, a) \geq 0$. This variable is increased when a state-action pair (s, a) is visited (i.e., action a is taken in state s), and decayed by $\gamma\lambda$ at the other timesteps, where $\lambda \in [0, 1]$ is called the *trace-decay parameter*.

There are two common types of traces, *accumulating* traces and *replacing* traces, each using a different way to update the eligibility trace of the current state-action pair. For accumulating traces, all traces are decayed by $\gamma\lambda$, but the current state-action pair gets an additional value increment of 1:

$$e_t(s, a) = \begin{cases} \gamma\lambda\, e_{t-1}(s, a) + 1 & \text{if } (s, a) = (s_t, a_t) \\ \gamma\lambda\, e_{t-1}(s, a) & \text{otherwise}. \end{cases}$$

While accumulating traces in general work well, for certain domains with many revisits of states they can cause problems, since the value of an eligibility trace can grow unbounded. For these domains, replacing traces is a better choice, which reset the value of the current state-action pair to 1:

$$e_t(s, a) = \begin{cases} 1 & \text{if } (s, a) = (s_t, a_t) \\ \gamma\lambda\, e_{t-1}(s, a) & \text{otherwise}. \end{cases}$$

In Algorithm 4 we shows the pseudocode for Sarsa(λ), the combination of eligibility traces with Sarsa. Note, that Sarsa(λ) reduces to regular Sarsa (Algorithm 3), when λ is set to 0.

Algorithm 4 Sarsa(λ)

initialize $Q(s, a)$ arbitrarily for all s,a
loop {over episodes}
  initialize $e(s, a) = 0$ for all s,a
  initialize s
  select action a, based on $Q(s, \cdot)$
  repeat {for each step in the episode}
    take action a, observe r and s'
    select action a', based on $Q(s', \cdot)$
    $\delta \leftarrow r + \gamma\, Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow e(s, a) + 1$ (accumulating traces)
    or $e(s, a) \leftarrow 1$ (replacing traces)
    for all s,a **do**
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta\, e(s, a)$
      $e(s, a) \leftarrow \gamma\lambda\, e(s, a)$
    $s \leftarrow s'; a \leftarrow a'$
  until s is terminal

For Q(λ), the combination of eligibility traces with Q-learning, two implementations are proposed. The implementation of Watkins (1989) results in a fully off-policy method. The disadvantage of this implementation is that all traces are reset to 0, when a non-greedy action is selected, limiting the propagation of new information. Peng and Williams (1996) propose a different implementation, which does not rely on resetting traces. However, the implementation is more complicated and the resulting method is neither off-policy nor on-policy, but rather some kind of mixture between the two.

For evaluation methods based on eligibility traces good convergence results exist for the tabular case (Jaakkola et al., 1994) as well as the function approximation case (Maei and Sutton, 2010). However, for control methods, including Sarsa(λ) and Q(λ), such results do not exist.

CHAPTER 3

Maximizing Performance under Severe Space and Time Constraints

When it comes to space and time efficiency, it is hard to imagine value-function based learning methods that are more efficient than the classical methods Q-learning and Sarsa. Both these methods update a Q-value right after a new experience sample from the corresponding state-action pair is observed. Hence, they perform a single update at every timestep.

In this chapter, we focus on strategies that can improve the performance of Q-learning and Sarsa without deteriorating their low space requirements and computational efficiency. In Section 3.1, we discuss a variation on Sarsa that performs updates with lower variance. Though this variation, which we call Expected Sarsa, appears to have better theoretical properties and is mentioned several times in the literature (Rummery, 1995; Sutton and Barto, 1998), it is not widely used in practise and no systematic study of it can be found. For this reason, we perform an extensive theoretical and empirical analysis of Expected Sarsa to assess its merits.

In Section 3.2 and 3.3, we present just-in-time learning, which postpones the update of a value until it is needed. By postponing the update, the update can become more accurate, since the values on which the update target is based may have improved in the meantime. In Section 3.2 we evaluate a just-in-time version of Q-learning; in Section 3.3 we evaluate just-in-time versions of Sarsa and Expected Sarsa.

3.1 Expected Sarsa

Since Sarsa's convergence guarantee requires that every state is visited infinitely often, the behavior policy is typically stochastic so as to ensure sufficient exploration. Due to the on-policy nature of Sarsa, the estimation policy is stochastic as well in this case. As a result, there can be substantial variance in Sarsa updates, since a_{t+1}, used in the update target (see Equation 2.13), is not selected deterministically.

Of course, variance can occur in updates for any TD method because the environment can introduce stochasticity through $\mathcal{P}$ and $\mathcal{R}$. Since the environment model is unknown, there is little the agent can do about this stochasticity, except employ a suitably low α. However, the additional variance introduced by Sarsa stems from the policy stochasticity, which is known to the agent.

Expected Sarsa is a variation of Sarsa that exploits knowledge about the policy stochasticity to prevent this stochasticity from further increasing variance in the updates. It does

so by basing the update not on $Q(s_{t+1}, a_{t+1})$, but on its expected value $E\{Q(s_{t+1}, a_{t+1})\}$. The resulting update target is:

$$v_t = r_{t+1} + \gamma \sum_a \pi(s_{t+1}, a)\, Q(s_{t+1}, a) \tag{3.1}$$

Using this update target reduces the variance in the update, as we show formally in Section 3.1.2. Lower variance means that in practice α can often be increased in order to speed learning, as we demonstrate empirically in Section 3.1.4. In fact, when the environment is deterministic, Expected Sarsa can employ $\alpha = 1$, while Sarsa still requires $\alpha < 1$ to cope with policy stochasticity.

Algorithm 5 shows the complete Expected Sarsa algorithm. Because the update rule of Expected Sarsa, unlike Sarsa, does not make use of the action taken in s_{t+1}, action selection can occur *after* the update. Doing so creates an additional advantageous in problems containing states with returning actions. When $s_{t+1} = s_t$, performing an update of $Q(s_t, a_t)$, also updates $Q(s_{t+1}, a_t)$, yielding a better estimate before action selection for state s_{t+1} occurs.

Algorithm 5 Expected Sarsa

```
Initialize Q(s,a) arbitrarily for all s,a
loop {over episodes}
    Initialize s
    repeat {for each step in the episode}
        choose a from s using policy π derived from Q
        take action a, observe r and s'
        V_s' = Σ_a π(s',a) · Q(s',a)
        Q(s,a) ← (1 − α) Q(s,a) + α [r + γ V_s']
        s ← s'
    until s is terminal
```

Instead of a low-variance version of Sarsa, Expected Sarsa can also be viewed as an on-policy version of Q-learning. Note the similarity between the expectation value $E\{Q(s_{t+1}, a_{t+1})\}$ used by Expected Sarsa and Equation 2.6, relating $V^\pi(s)$ to $Q^\pi(s, a)$. Since $Q(s, a)$ is an estimate of $Q^\pi(s, a)$, its expectation value can be seen as the estimate $V(s)$ for $V^\pi(s)$ using the relation:

$$V(s) = \sum_a \pi(s, a)\, Q(s, a) \tag{3.2}$$

If the policy π is greedy, $\pi(s, a) = 0$ for all a except for the action for which Q has its maximal value. Therefore, in the case of a greedy policy, (3.2) simplifies to

$$V(s) = \max_a Q(s, a) \tag{3.3}$$

Thus, Q-learning's update target (Equation 2.12) is just a special case of Expected Sarsa's update target (Equation 3.1), for which the estimation policy is greedy. Nonetheless, the Expected Sarsa algorithm is different from the Q-learning algorithm because the former is on-policy and the latter is off-policy.

3.1.1 Convergence

In this section, we prove that Expected Sarsa converges to the optimal policy under some straightforward conditions given below. We make use of the following Lemma, which was also used to prove convergence of Sarsa (Singh et al., 2000):

Lemma 1. *Consider a stochastic process* (ζ_t, Δ_t, F_t)*, where* $\zeta_t, \Delta_t, F_t : X \to \mathbb{R}$ *satisfy the equations*

$$\Delta_{t+1}(x_t) = (1 - \zeta_t(x_t))\Delta_t(x_t) + \zeta_t(x_t)F_t(x_t) \ ,$$

where $x_t \in X$ *and* $t = 0, 1, 2, \ldots$*. Let* P_t *be a sequence of increasing* σ*-fields such that* ζ_0 *and* Δ_0 *are* P_0*-measurable and* ζ_t, Δ_t *and* F_{t-1} *are* P_t*-measurable,* $t \geq 1$*. Assume that the following hold:*

1. *the set X is finite,*
2. $\zeta_t(x_t) \in [0, 1]$, $\sum_t \zeta_t(x_t) = \infty$, $\sum_t (\zeta_t(x_t))^2 < \infty$ *w.p.1 and* $\forall x \neq x_t : \zeta_t(x) = 0$*,*
3. $||E\{F_t|P_t\}|| \leq \kappa||\Delta_t|| + c_t$*, where* $\kappa \in [0, 1)$ *and* c_t *converges to zero w.p.1,*
4. $Var\{F_t(x_t)|P_t\} \leq K(1 + \kappa||\Delta_t||)^2$*, where* K *is some constant,*

where $||\cdot||$ *denotes a maximum norm. Then* Δ_t *converges to zero with probability one.*

The idea is to apply Lemma 1 with $X = S \times A$, $P_t = \{Q_0, s_0, a_0, r_0, \alpha_0, s_1, a_1, \ldots, s_t, a_t\}$, $x_t = (s_t, a_t)$, $\zeta_t(x_t) = \alpha_t(s_t, a_t)$ and $\Delta_t(x_t) = Q_t(s_t, a_t) - Q^*(s_t, a_t)$. If we can then prove that Δ_t converges to zero with probability one, we have convergence of the Q values to the optimal values. The maximum norm specified in the lemma can then be understood as satisfying the following equation:

$$||\Delta_t|| = \max_s \max_a |Q_t(s, a) - Q^*(s, a)| \tag{3.4}$$

Theorem 1. *Expected Sarsa converges to the optimal value function whenever the following assumptions hold:*

1. S *and* A *are finite,*
2. $\alpha_t(s_t, a_t) \in [0, 1]$, $\sum_t \alpha_t(s_t, a_t) = \infty$, $\sum_t (\alpha_t(s_t, a_t))^2 < \infty$ *w.p.1 and* $\forall (s, a) \neq (s_t, a_t) : \alpha_t(s, a) = 0$*,*
3. *The policy is greedy in the limit with infinite exploration,*
4. *The reward function is bounded.*

Proof. To prove this theorem, we simply check that all the conditions of Lemma 1 are fulfilled. The first, second and fourth conditions of this lemma correspond to the first, second and fourth assumptions of the theorem. Below, we will show the third condition of the lemma holds.

We can derive the value of F_t as follows:

$$\begin{aligned} F_t &= \frac{1}{\alpha_t}\Big(\Delta_{t+1} - (1-\alpha_t)\Delta_t\Big) \ , \\ &= r_t + \gamma \sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a) - Q^*(s_t, a_t) \ , \end{aligned}$$

where all the values are taken over the state action pair (s_t, a_t), except when specified differently.

If we can show that $||E\{F_t\}|| \leq \kappa ||\Delta_t|| + c_t$, where $\kappa \in [0, 1)$ and c_t converges to zero, all the conditions of the lemma can be fulfilled and we have convergence of Δ_t to zero and therefore convergence of Q_t to Q^*. We derive this as follows:

$$\begin{aligned} & ||E\{F_t\}|| \\ = \ & ||E\{r_t + \gamma \sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a) - Q^*(s_t, a_t)\}|| \\ \leq \ & ||E\{r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q^*(s_t, a_t)\}|| + \\ & \gamma ||E\{\sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a) - \max_a Q_t(s_{t+1}, a)\}|| \\ \leq \ & \gamma \max_s \Big| \max_a Q_t(s, a) - \max_a Q^*(s, a) \Big| + \\ & \gamma \max_s \Big| \sum_a \pi_t(s, a) Q_t(s, a) - \max_a Q_t(s, a) \Big| \\ \leq \ & \gamma ||\Delta_t|| + \\ & \gamma \max_s \Big| \sum_a \pi_t(s, a) Q_t(s, a) - \max_a Q_t(s, a) \Big| \ , \end{aligned}$$

where the second inequality results from the definition of Q^* and the fact that the maximal difference in value over all states is always at least as large as a difference between values corresponding to a state s_{t+1}. The third inequality follows directly from (3.4). The other (in)equalities are based on algebraic rewriting or definitions.

We identify $c_t = \gamma \max_s |\sum_a \pi_t(s, a) Q_t(s, a) - \max_a Q_t(s, a)|$ and $\kappa = \gamma$. Clearly, c_t converges to zero for policies that are greedy in the limit. Therefore, if $\gamma < 1$, all of the conditions of Lemma 1 follow from the assumptions in the present theorem and we can apply the lemma to prove convergence of Q_t to Q^*. □

3.1.2 Variance Analysis

Section 3.1.1 shows that Expected-Sarsa converges to the optimal policy under the same conditions as Sarsa. In this section, we further analyze the behavior of the two methods to show theoretically under what conditions Expected-Sarsa will in some sense perform better. Specifically, we show that both algorithms have the same bias and that the variance of Expected-Sarsa is lower. Finally, we describe which factors affect this difference in variance. In this section, we use $v_t = r_t + \gamma \sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a)$ and $\hat{v}_t = r_t + \gamma Q_t(s_{t+1}, a_{t+1})$ to denote the target of Expected-Sarsa and Sarsa, respectively.

The bias of the updates of both algorithms under a certain policy π is given by the following equation:

$$Bias(s,a) = Q^{\pi}(s,a) - E\{X_t\} \tag{3.5}$$

where X_t is either v_t or $\hat{v}_t$. Both algorithms have the same bias, since $E\{v_t\} = E\{\hat{v}_t\}$. The variance is then given by:

$$Var(s,a) = E\{(X_t)^2\} - (E\{X_t\})^2 \tag{3.6}$$

We first calculate this variance for Sarsa:

$$\begin{aligned} Var(s,a) &= \sum_{s'} T_{sa}^{s'} \Big(\gamma^2 \sum_{a'} \pi_{s'a'} (Q_t(s',a'))^2 + (R_{sa}^{s'})^2 \\ &\quad + 2\gamma R_{sa}^{s'} \sum_{a'} \pi_{s'a'} Q_t(s',a') \Big) - (E\{\hat{v}_t\})^2 \ . \end{aligned}$$

Similarly, for Expected-Sarsa we get:

$$\begin{aligned} Var(s,a) &= \sum_{s'} T_{sa}^{s'} \Big(\gamma^2 (\sum_{a'} \pi_{s'a'} Q_t(s',a'))^2 + (R_{sa}^{s'})^2 \\ &\quad + 2\gamma R_{sa}^{s'} \sum_{a'} \pi_{s'a'} Q_t(s',a') \Big) - (E\{\hat{v}_t\})^2 \ . \end{aligned}$$

Since $E\{v_t\} = E\{\hat{v}_t\}$, the difference between the two variances simplifies to the following:

$$\gamma^2 \sum_{s'} T_{sa}^{s'} \Big(\sum_{a'} \pi_{s'a'} (Q_t(s',a'))^2 - (\sum_{a'} \pi_{s'a'} Q_t(s',a'))^2 \Big) \ .$$

The inner term is of the form:

$$\sum_i w_i x_i^2 - (\sum_i w_i x_i)^2 \ , \tag{3.7}$$

where the w and x correspond to the π and Q values. When $w_i \geq 0$ for all i and $\sum_i w_i = 1$, we can give an unbiased estimate of the variance of the weighed values $w_i x_i$ as follows:

$$\frac{\sum_i w_i (x_i - \bar{x})^2}{1 - \sum_i w_i^2} \ , \tag{3.8}$$

where $\bar{x}$ is the weighted mean $\sum_i w_i x_i$. Taking the numerator of this fraction and rewriting this gives us:

$$\begin{aligned} \sum_i w_i (x_i - \bar{x})^2 &= \sum_i w_i x_i^2 - 2 \sum_i w_i x_i \bar{x} + \sum_i w_i \bar{x}^2 \\ &= \sum_i w_i x_i^2 - 2\bar{x}^2 + \bar{x}^2 \\ &= \sum_i w_i x_i^2 - \bar{x}^2 \ , \end{aligned}$$

which is exactly the same quantity as given in (3.7). This shows that this quantity is closely related to the weighted variance of the $w_i x_i$. Therefore, the more the x_i deviate from the weighted mean $\sum_i w_i x_i$, the larger this quantity will be. In our context this occurs in settings where there is a large difference between the Q values of different actions and there is much exploration. In case of a greedy policy or when all Q values have the same value, this quantity is 0.

3.1.3 Hypotheses

In this section, we formulate specific hypotheses about when Expected Sarsa will outperform Q-learning and Sarsa. These hypotheses are based on the central differences between Expected Sarsa and these two alternatives: 1) unlike Q-learning, Expected Sarsa is on-policy and 2) Expected Sarsa has lower variance than Sarsa.

For simplicity, we restrict our attention to the case where exploration is performed using ε-soft behavior policies, i.e., the agent takes a random action with probability ε and uses the estimation policy otherwise. Using such exploration, off-policy methods can sometimes perform quite differently than on-policy methods. For example, in the cliff-walking task (detailed in Section 3.1.4), some actions can have disastrous consequences in certain states, e.g., when near a cliff. Off-policy methods try to estimate the optimal way to behave *without exploration* and then merely employ an ε-soft version of the resulting policy. Consequently, they may never learn to avoid such catastrophic actions. By contrast, on-policy methods try to estimate the optimal way to behave *given the exploration that is occurring*. Therefore, they can learn policies that are qualitatively different from the optimal policy without exploration but that avoid catastrophic actions in the presence of exploration, e.g., by staying further away from the cliff. Based on this difference we can define two different types of problems:

1. Problems where the optimal ε-soft policy is better than the ε-soft policy based on $Q^*(s, a)$.

2. Problems where the optimal ε-soft policy is equal to the ε-soft policy based on $Q^*(s, a)$.

Because Expected Sarsa is on-policy and Q-learning is off-policy, we state the following hypothesis:

Hypothesis 1. *Expected Sarsa will outperform Q-learning for problems of Type 1.*

Section 3.1.2 demonstrated that the variance in the update target for Sarsa is larger than for Expected Sarsa, especially when the policy stochasticity is large and when there is a large spread in Q values of the actions of a state. Based on these facts, we can formulate a second hypothesis, one about the performance difference between Expected Sarsa and Sarsa.

Hypothesis 2. *Expected Sarsa will outperform Sarsa on problems of both Type 1 and Type 2. The size of the performance difference depends primarily on two factors:*

1. *When environment stochasticity is high, performance difference will be small.*

2. *When policy stochasticity is high, performance difference will be large.*

3.1.4 Empirical Results

In this section we present a series of experiments to compare the online performance of Expected Sarsa to that of Sarsa and Q-learning in order to test the hypotheses described in the previous section. We start with the cliff walking problem. This is an example of a problem where an exploration policy based on the optimal action values $Q^*(s, a)$ is not equal to the optimal policy with exploration added. Sutton and Barto (1998) showed that Sarsa outperforms Q-learning on this problem. We show that Expected Sarsa outperforms Q-learning as well as Sarsa, confirming Hypothesis 1 and providing some evidence for Hypothesis 2.

We then present results on two versions of the windy grid world problem, one with a deterministic environment and one with a stochastic environment. We do so in order to evaluate the influence of environment stochasticity on the performance difference between Expected Sarsa and Sarsa and confirm the first part of Hypothesis 2. We then present results for different amounts of policy stochasticity to confirm the second part of Hypothesis 2. For completeness, we also show the performance of Q-learning on this problem. Finally, we present results in other domains verifying the advantages of Expected Sarsa in a broader setting. All results presented below are averaged over numerous independent trials such that the standard error becomes negligible.

3.1.4.1 Cliff Walking

We begin by testing Hypothesis 1 using the cliff walking task, an undiscounted, episodic navigation task in which the agent has to find its way from start to goal in a deterministic grid world. Along the edge of the grid world is a cliff (see Figure 3.1). The agent can take any of four movement actions: *up*, *down*, *left* and *right*, each of which moves the agent one square in the corresponding direction. Each step results in a reward of -1, except when the agent steps into the cliff area, which results in a reward of -100 and an immediate return to the start state. The episode ends upon reaching the goal state.

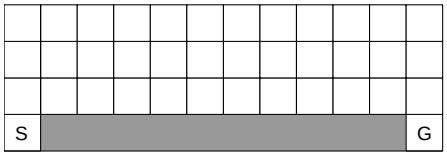

Figure 3.1: The cliff walking task. The agent has to move from the start [S] to the goal [G], while avoiding stepping into the cliff (grey area).

We evaluated the performance over the first n episodes as a function of the learning rate α using an ε-greedy policy with ε = 0.1. Figure 3.2 shows the result for $n = 100$ and $n = 100,000$. We averaged the results over 50,000 runs and 10 runs, respectively.

Expected Sarsa outperforms Q-learning and Sarsa for all learning rate values, confirming Hypothesis 1 and providing some evidence for Hypothesis 2. The optimal α value of Expected Sarsa for $n = 100$ is 1, while for Sarsa it is lower, as expected for a deterministic problem. That the optimal value of Q-learning is also lower than 1 is surprising, since Q-learning also has no stochasticity in its updates in a deterministic environment. Our explanation is that Q-learning first learns policies that are sub-optimal in the greedy sense, i.e. walking towards the goal with a detour further from the cliff. Q-learning iteratively optimizes these early policies, resulting in a path more closely along the cliff. However, although this path is better in the off-line sense, in terms of on-line performance it is worse. A large value of α ensures the goal is reached quickly, but a value somewhat lower than 1 ensures that the agent does not try to walk right on the edge of the cliff immediately, resulting in a slightly better on-line performance.

For $n = 100,000$, the average return is equal for all α values in case of Expected Sarsa and Q-learning. This indicates that the algorithms have converged long before the end of the run for all α values, since we do not see any effect of the initial learning phase. For Sarsa the performance comes close to the performance of Expected Sarsa only for $\alpha = 0.1$, while for large α, the performance for $n = 100,000$ even drops below the performance for $n = 100$. The reason is that for large values of α the Q values of Sarsa diverge. Although the policy is still improved over the initial random policy during the early stages of learning, divergence causes the policy to get worse in the long run.

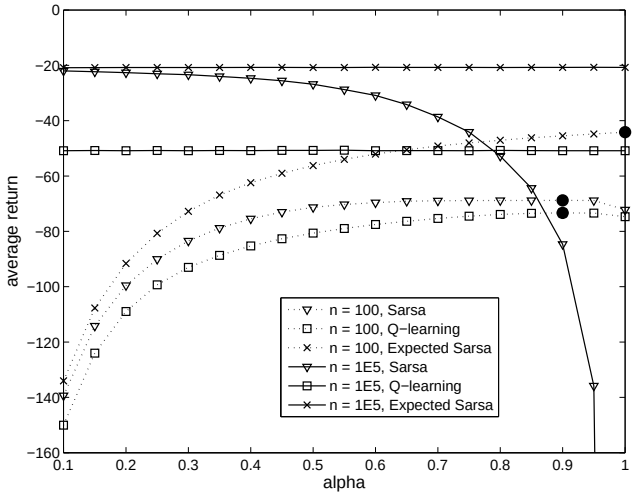

Figure 3.2: Average return on the cliff walking task over the first n episodes for $n = 100$ and $n = 100,000$ using an ε-greedy policy with $\varepsilon = 0.1$. The big dots indicate the maximal values.

3.1.4.2 Windy Grid World

We turn to the windy grid world task to further test Hypothesis 2. The windy grid world task is another navigation task, where the agent has to find its way from start to goal. The grid has a height of 7 and a width of 10 squares. There is a wind blowing in the 'up'

direction in the middle part of the grid, with a strength of 1 or 2 depending on the column. Figure 3.3 shows the grid world with a number below each column indicating the wind strength. Again, the agent can choose between four movement actions: *up*, *down*, *left* and *right*, each resulting in a reward of -1. The result of an action is a movement of 1 square in the corresponding direction plus an additional movement in the 'up' direction, corresponding with the wind strength. For example, when the agent is in the square right of the goal and takes a 'left' action, it ends up in the square just above the goal.

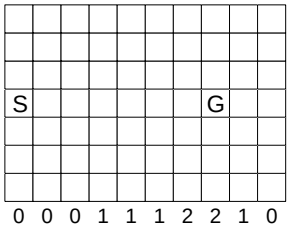

Figure 3.3: The windy grid world task. The agent has to move from start [S] to goal [G]. The numbers under the grid indicate the wind strength in the column above.

Deterministic Environment

We first consider a deterministic environment. As in the cliff walking task, we use an ε-greedy policy with $\varepsilon = 0.1$. Figure 3.4 shows the performance as a function of the learning rate α over the first n episodes for $n = 100$ and $n = 100,000$. For $n = 100$ the results are averaged over $10,000$ independent runs, for $n = 100,000$ over 10 independent runs.

For the deterministic windy grid world task the performance of Q-learning and Expected Sarsa is essentially equal. The fact that for $n = 100,000$ the average return is equal indicates that the behavior policies of Expected Sarsa and Q-learning are equal in the limit for this task, i.e., the optimal policy among the ε-greedy policies (Expected Sarsa) is equal to the policy that is ε-greedy with respect to $Q^*(s,a)$ (Q-learning). The optimal α is 1 for Expected Sarsa as well as Q-learning. Sarsa again has a lower optimal α. As in the cliff walking task we observed divergence of Q values for high α values in the case of Sarsa. The performance difference for $n = 100$ between Expected Sarsa and Sarsa at their optimal values is $(-45.0) - (-58.3) = 13.3$ in favor of Expected Sarsa.

Environment Stochasticity

We also consider a stochastic variation of the windy grid world problem and compare results to the performance difference in the deterministic case in order to evaluate the first part of Hypothesis 2. We added stochasticity to the environment by moving the agent with a probability of 20% in a random direction instead of the direction corresponding to the action. The performance as function of the learning rate is presented in Figure 3.5 for $n = 100$ and $n = 100,000$. Again, we averaged the results over 10,000 runs and 10 runs respectively.

As expected, the optimal α for Expected Sarsa and Q-learning in case of $n = 100$ drops

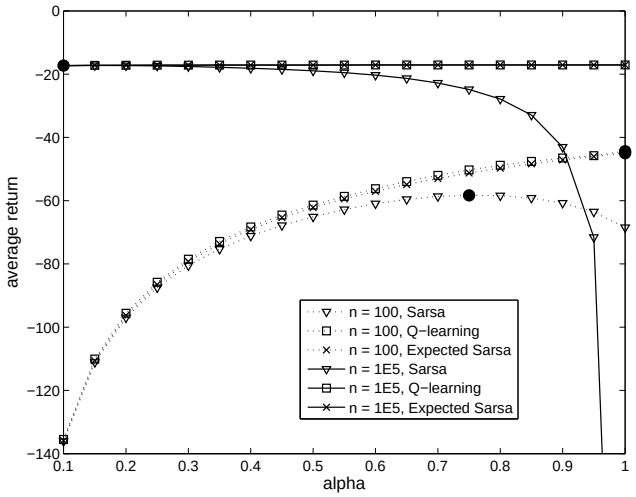

Figure 3.4: Average return on the windy grid world task over the first n episodes for $n = 100$ and $n = 100,000$ and an ε-greedy policy with ε = 0.1 in a deterministic environment. The big dots indicate maximal values.

considerably in comparison to the deterministic case, to a value of 0.6. The optimal α value of Sarsa also decreases, to 0.55. From the $n = 100,000$ case, we can see that the policy no longer converges for Expected Sarsa and Q-learning for all α values. Although not stable for high α values, the average policy is better for Expected Sarsa than for Q-learning, which is likely due to the on-policy nature of Expected Sarsa. On the other hand, For $n = 100$, Q-learning slightly outperforms Expected Sarsa because it benefits more from *optimistic initialization*, i.e., initially overestimating the Q values to increase exploration during early learning. Since Q-learning uses the maximal Q value of the next state in its update, it takes longer for the Q values to decrease.

The performance difference for $n = 100$ between Expected Sarsa and Sarsa at their optimal values is $(-93.7) - (-98.3) = 4.6$ in favor of Expected Sarsa. The performance difference is less than half that of the deterministic case, confirming the first part of Hypothesis 2.

Policy Stochasticity

To confirm the second part of Hypothesis 2, we repeat the stochastic windy grid world experiment but with higher policy stochasticity, using an ε of 0.3 instead of 0.1. Figure 3.6 shows the results.

For $n = 100$ the optimal α for Sarsa drops from 0.55 to 0.45 and the optimal α for Q-learning decreases slightly, though for Expected Sarsa it stays the same. Furthermore, the performance difference between Q-learning and Expected Sarsa increases. The performance difference between Sarsa and Expected Sarsa also increases for $n = 100$ and is now $(-121.0) - (-136.4) = 15.4$, confirming the second part of Hypothesis 2. Other experiments, not shown in this thesis, confirmed that also the opposite is true: when policy

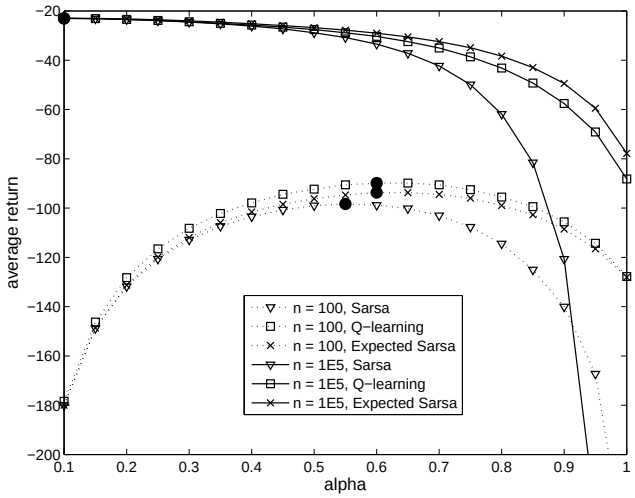

Figure 3.5: Average return on the windy grid world task over the first n episodes for $n = 100$ and $n = 100,000$ using a ε-greedy policy with ε = 0.1 in a stochastic environment. The big dots indicate maximal values.

stochasticity is low, i.e. using an ε-greedy policy with $\varepsilon = 0.01$ there is practically no performance difference between Sarsa and Expected Sarsa.

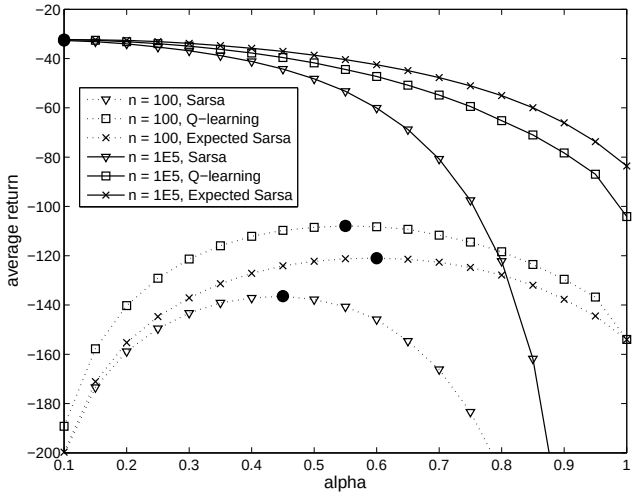

Figure 3.6: Average return on the windy grid world task over the first n episodes for $n = 100$ and $n = 100,000$ using a ε-greedy policy with ε = 0.3 in a stochastic environment. The big dots indicate maximal values.

To demonstrate that the advantage of Expected Sarsa holds more generally, we also tested in two other domains.

3.1.4.3 Maze

We compared Expected Sarsa to Sarsa and Q-learning on the maze problem shown in Figure 3.7. The goal of the agent is to find a path from start to goal, while avoiding hitting the walls. The reward for arriving at the goal is 100. When the agent bumps into a wall or border of the environment it stays at the same position, but receives a reward of -2. For all other steps a reward of -0.1 is received. The environment is stochastic and moves the agent with a probability of 10% in a random direction instead of the direction corresponding to the action. The discount factor γ is set to 0.997. A trial is finished after the agent reaches the goal or 10,000 actions have been performed. An ε-greedy behavior policy is used with $\varepsilon = 0.05$ and we initialized the Q values to 0.

We optimized α for each method such that the average reward over the first $2 * 10^6$ timesteps is maximized. The optimal values were 0.24, 0.28 and 0.27 for Sarsa, Q-learning and Expected Sarsa respectively. We then plotted the reward as function of the number of timesteps for these optimal α values to get a more detailed look at performance. Figure 3.8 shows the results, which are averaged over 100 trials.

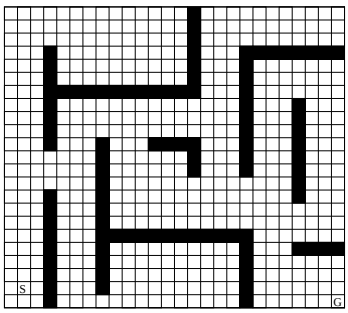

Figure 3.7: The maze problem. The starting position is indicated by [S] and the goal position is indicated by [G].

Although Expected Sarsa and Q-learning perform equally, Sarsa's performance is lower and not monotonically increasing. It shows a drop in performance after $0.2 * 10^6$ timesteps, before it slowly increases again. This drop occurs in all one hundred runs.

Although this is a clear demonstration of the possibility that Sarsa can be unstable in certain cases, we have not observed this phenomenon in previous research, and it is remarkable because the value function is represented in a table, without the complications of function approximation. We explain this temporary performance drop of Sarsa as follows: since in our implementation we initialized all Q values to 0, while their real value is higher, all values start to increase in the beginning. However, the values of the best actions increase faster because they have a shorter propagation path to the final reward of 100. Therefore, initially Sarsa learns well. However, because of the high discount factor of 0.997, all action-values in a state start to get very close to each other. This makes it possible that after a bad exploration step, some values are updated in a way that makes the policy worse. After a while Sarsa finds a policy that is not optimal, but that is robust against such value updates. The same drop in performance also happens when using a learning rate

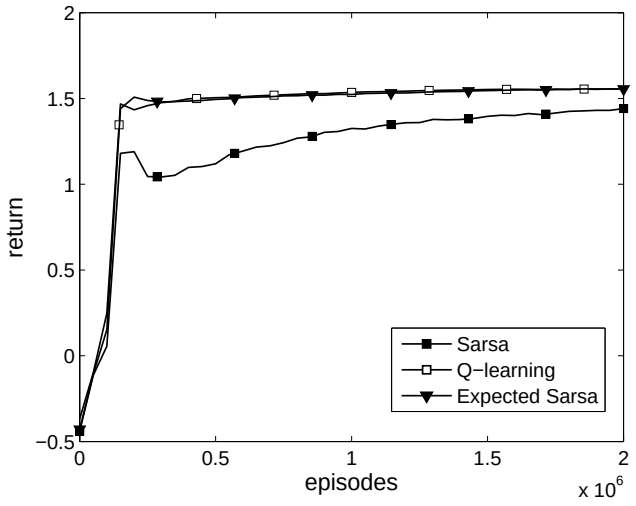

Figure 3.8: The on-line performance of the different methods on the maze problem. The results are averaged over 100 runs.

of 0.04 for Sarsa, although initial learning performance was slower and the drop occurred later. The update targets of Expected Sarsa and Q-learning are not effected by the action selected in the next state and are therefore more robust towards performance drops.

3.1.4.4 Cart Pole

As a final comparison, we test the on-line performance of Expected Sarsa, Sarsa and Q-learning on a cart-pole task. The goal was to balance a 1 m long pole, weighing 0.1 kg, on a cart that weighs 1.0 kg. The possible actions were all integer amounts between -10 and 10 Newton, where positive and negative forces correspond to pushing the cart right and left, respectively. An action was performed every 0.02 s. If the cart was pushed further than 2.4 m from the center of the track or if the pole drops further than 12 degrees to either side, the algorithm would receive a -1 reward and the cart would be reset to the center with the pole at a random angle between -3 and 3 degrees. A neural network with 15 sigmoidal hidden units was used to approximate the Q values. The input vector consisted of the position and velocity of the cart and the angle and angular velocity of the pole, all normalized to [-1,1]. The value of ε was 0.05 and γ was 0.95. Figure 3.9 shows the average reward during learning at optimized α values of 0.12, 0.16 and 0.16 for Sarsa, Q-learning and Expected Sarsa respectively.

We see again that Expected Sarsa and Q-learning perform similar, while Sarsa is less stable and shows lower performance. This demonstrates that the results extend to the case of function approximation.

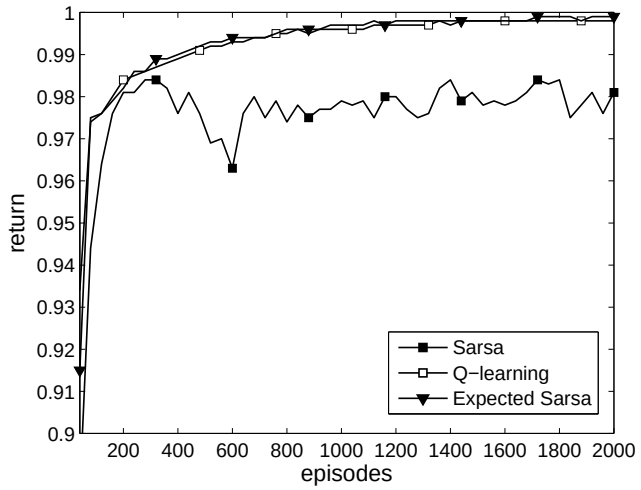

Figure 3.9: The learning performance of the different methods on the cart pole. The results are averaged over 200 simulations.

3.2 Just-In-Time Q-learning

In this section we present just-in-time (JIT) Q-learning. Like other *lazy learning* methods, e.g., (Atkeson et al., 1997), JIT Q-learning postpones updates until they are needed. Wiering and Schmidhuber (1998) showed that by postponing updates a computationally efficient version of Q(λ) can be constructed that does not rely on placing a bound on the trace length. We prove that by postponing Q-learning updates until a state is revisited, the update targets involved receive in general more updates, while the total number of updates of the current state stays the same. Empirically, we demonstrate that this leads to a performance gain under a range of settings at similar computational cost.

When a Q-learning update is postponed, the values on which the update target is based are from a more recent timestep. This is advantageous, since Q-learning updates cause the expected error in the values to decrease over time (Watkins and Dayan, 1992) and therefore more recent values will be on average more accurate. However, postponing the update of a value for too long can negatively affect performance, since a value that has not been updated might be used for action selection or for bootstrapping other values. We start by showing that updates can be postponed until their corresponding states are revisited, without negatively affecting performance.

Consider the state-action sequence in Figure 3.10. State s_A is visited at timestep 0 and revisited at timestep 4. With the regular Q-learning update, the Q-value of state-action pair (s_A, a_0) gets updated at timestep 1:

$$Q_1(s_A, a_0) = (1 - \alpha)Q_0(s_A, a_0) + \alpha \left[r_1 + \gamma \max_a Q_0(s_B, a)\right]$$

while at timesteps $2 - 4$ no update of (s_A, a_0) occurs, and therefore $Q_4(s_A, a_0) = Q_1(s_A, a_0)$. The update of the Q-value of (s_A, a_0) at timestep 1 can be considered premature, since the earliest use of its value is in the update target for (s_D, a_3), which uses

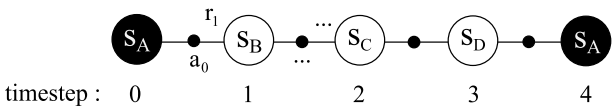

Figure 3.10: A state transition sequence in which the initial state s_A is revisited at timestep 4. The small black dots in between states represent actions.

$Q_3(s_A, a_0)$. Therefore, the update of the Q-value of (s_A, a_0) can be postponed until at least timestep 3 without negatively affecting the update target for (s_D, a_3). When the update of (s_D, a_3) is also postponed, the earliest use of the Q-value of (s_A, a_0) occurs at timestep 4, where it is used for action selection. Thus, if we postpone the update of all state-action pairs, the update of the Q-value of (s_A, a_0) can be postponed until the timestep of its revisit, without causing dependent state values or the action selection procedure to use a value of (s_A, a_0) that has not been updated. We call this type of update a *just-in-time update*, since the update is postponed until just before the updated value is needed.

To denote the Q-values resulting from just-in-time updates we use $\tilde{Q}$ throughout this section. With just-in-time updates, no updates of (s_A, a_0) occur at timesteps 1-3, so $\tilde{Q}_3(s_A, a_0) = \tilde{Q}_0(s_A, a_0)$. Instead, an update occurs when s_A is revisited:

$$\tilde{Q}_4(s_A, a_0) = (1-\alpha)\tilde{Q}_3(s_A, a_0) + \alpha\left[r_1 + \gamma \max_a \tilde{Q}_3(s_B, a)\right]$$

The regular and just-in-time update for (s_A, a_0) can be written in a more similar form by expressing the value at timestep 4 in terms of the value at timestep 0:

$$Q_4(s_A, a_0) = (1-\alpha)Q_0(s_A, a_0) + \alpha[r_1 + \gamma \max_a Q_0(s_B, a)] \quad (3.9)$$

$$\tilde{Q}_4(s_A, a_0) = (1-\alpha)\tilde{Q}_0(s_A, a_0) + \alpha[r_1 + \gamma \max_a \tilde{Q}_3(s_B, a)] \quad (3.10)$$

This formulation highlights the difference between the two update types. At timestep 4, under both update schemes, the Q-value of (s_A, a_0) has received one update based on the same experience sample. However, a just-in-time update uses the most recent value of the Q-values of s_B, while a regular update uses the value at the timestep of the initial visit of s_A. By defining t^* as the timestep of the previous visit of state s_t, we can write the two update types more generally as:

$$Q_t(s_t, a_{t^*}) = (1-\alpha)Q_{t^*}(s_t, a_{t^*}) + \alpha[r_{t^*+1} + \gamma \max_a Q_{t^*}(s_{t^*+1}, a)] \quad (3.11)$$

$$\tilde{Q}_t(s_t, a_{t^*}) = (1-\alpha)\tilde{Q}_{t^*}(s_t, a_{t^*}) + \alpha[r_{t^*+1} + \gamma \max_a \tilde{Q}_{t-1}(s_{t^*+1}, a)] \quad (3.12)$$

Note that we express the update target using only values from the past, making an implementation easier to interpret. Note also that while $s_t = s_{t^*}$ per definition (because s_t is revisited), s_{t^*+1} does not have to be equal to s_{t+1}, since the state transition from s_t can be stochastic. Also, a_{t^*} is in general not equal to a_t.

When comparing the two update targets in more detail, two cases can be distinguished. See Figure 3.11 for an example of each case. In the first case, state s_B is not revisited

before the revisit of state s_A. In this case, neither update type makes use of an updated Q-value for s_B in the update target for s_A. The regular update does not since it uses the values of s_B at timestep t^*, and the just-in-time update does not since s_B is not revisited and therefore no update has occurred yet at timestep $t-1$. In the second case, state s_B has been revisited before the revisit of s_A. The regular update still uses the value of s_B from timestep t^* and therefore does not use an updated value. The just-in-time update on the other hand does use an updated value, since this update occurred at the revisit of s_B. Note that for a returning action ($t^* = t-1$), both update types have exactly the same form and this can therefore be treated as an example of case 1. From these two cases, we can deduce the following theorem, which is proven in Appendix C.1.

Theorem 2. *Given the same experience sequence, each Q-value from the current state has received the same number of updates using JIT updates (Equation 3.12) as using regular updates (Equation 3.11). However, each Q-value in the update target of a JIT update has received an equal or greater number of updates as in the update target of the corresponding regular update.*

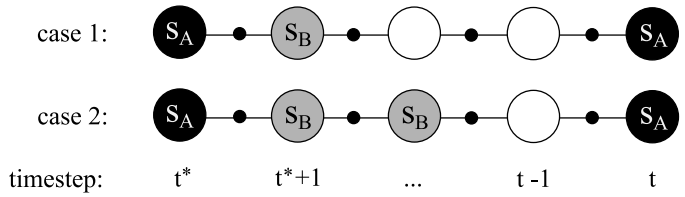

Figure 3.11: Two cases in which state s_A is revisited. In the first case, neither a regular update nor a just-in-time update make use of an updated value for s_B in the update target of s_A, while in the second case a just-in-time update does.

Algorithm 6 shows pseudocode for the implementation of just-in-time (JIT) Q-learning. The agent stores the reward and transition state received upon the last visit of a state, i.e., the *last-visit sample*, in $R'(s)$ and $S'(s)$ respectively, while the action taken at the last visit of a state is stored in $A(s)$. If $S'(s) = \emptyset$, state s has not been visited yet and no update can be performed. Note that the last-visit sample is not reset at the end of an episode, but maintained across episodes.

Because JIT Q-learning uses more recent values in its update targets than regular Q-learning, we expect a performance improvement over regular Q-learning. We test this hypothesis by comparing the performance of JIT Q-learning with regular Q-learning on the Dyna maze task (Sutton, 1990). In this navigation task, depicted in Figure 3.12, the agent has to find its way from start to goal. The agent can choose between four movement actions: up, down, left and right. All actions result in 0 reward, except for when the goal is reached, which results in a reward of +1. The discount factor γ is set to 0.95. We use a deterministic as well as a stochastic environment to test the generality of the hypothesis. In the stochastic version, we employ a probabilistic transition function: with a 20% probability, the agent moves in an arbitrary direction instead of the direction corresponding to the action.

Algorithm 6 JIT Q-Learning

initialize $Q(s,a)$ arbitrarily for all s,a
initialize $S'(s) = \emptyset$ for all s
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    if $S'(s) \neq \emptyset$ **then**
      $Q(s,\bar{a}) \leftarrow (1-\alpha^{s\bar{a}}) \cdot Q(s,\bar{a}) + \alpha^{s\bar{a}} \left[R'(s) + \gamma \, max_{a'} \, Q(S'(s),a')\right]$ $// \, \bar{a} = A(s)$
    select action a, based on $Q(s,\cdot)$
    take action a, observe r and s'
    $S'(s) \leftarrow s'$; $R'(s) \leftarrow r$; $A(s) \leftarrow a$
    $s \leftarrow s'$
  until s is terminal

To compare performance, we measure the average return each method accrues from the start state during the first 100 episodes in the deterministic case, averaged over 5000 independent runs per method. For the stochastic version, we measure the return during the first 200 episodes. Each method uses ε-greedy action selection with $\varepsilon = 0.1$. In the deterministic case, we use a constant learning rate of 1, while in the stochastic case we use an initial learning rate α_0 of 1 that is decayed in the following manner[1]

$$\alpha^{sa} = \frac{\alpha_0}{d \cdot [n(s,a) - 1] + 1} \tag{3.13}$$

where $n(s,a)$ is the total number of times action a has been selected in state s. Note that for $d = 0$, $\alpha^{sa} = \alpha_0$, while for $d = 1$, $\alpha^{sa} = \frac{\alpha_0}{n(s,a)}$. We optimize the learning rate decay d between 0 and 1 by taking the decay rate with the maximum average return over the measured number of episodes. We use two different initialization schemes for the Q-values to determine whether the performance difference depends on initialization. We use optimistic initialization, by initializing the Q-values to 20, and pessimistic initialization, by setting the Q-values to 0.

Table 3.1: The performance of JIT Q-learning and regular Q-learning on the Dyna maze task and the optimal learning rate decay d.

	deterministic - 100 eps.			stochastic - 200 eps.		
	d	average return	standard error	d	average return	standard error
Q-learning, $Q_0 = 0$	0	0.3506	0.0004	1.0	0.3039	0.0003
JIT Q-learning, $Q_0 = 0$	0	0.3628	0.0004	1.0	0.3083	0.0003
Q-learning, $Q_0 = 20$	0	0.3438	0.0002	0.005	0.2562	0.0002
JIT Q-learning, $Q_0 = 20$	0	0.3714	0.0002	0.010	0.2674	0.0002

[1]This decay is similar to the more common form $\frac{c_1}{c_2+n(s,a)}$, but with the free parameters re-arranged.

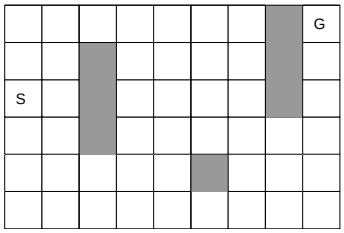

Figure 3.12: The Dyna maze task, in which the agent must travel from S to G. The reward is +1 when the goal state is reached and 0 otherwise.

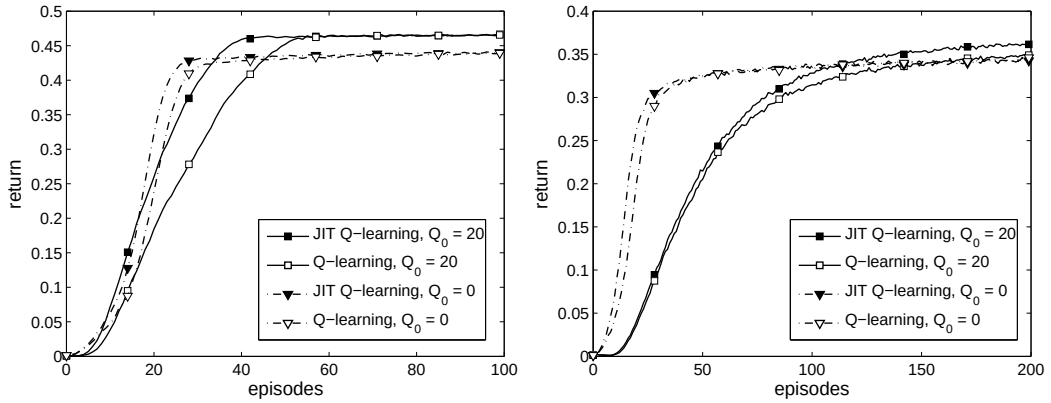

Figure 3.13: Comparison of the performance of JIT Q-learning and regular Q-learning on the deterministic (left) and stochastic (right) Dyna maze task for two different initialization schemes.

Figure 3.13 plots the return as a function of the number of episodes, while Table 3.1 shows the average return and optimal learning rate. The computation time for both methods was similar. JIT Q-learning outperforms regular Q-learning in the deterministic as well as the stochastic environment and for both types of initialization.[2]

The results confirm our intuition that, since JIT Q-learning uses values from a later time which are in general more accurate, a performance advantage is obtained with respect to regular Q-learning for a broad range of settings. Although this performance advantage is not for all settings substantial, JIT Q-learning performs consistently better at no extra computational cost, making it overall a better choice. In the next section we apply the just-in-time principle to Sarsa and Expected Sarsa.

[2]The performance benefit in the deterministic case can be explained by exploration, which causes the order in which states are visited to change despite the deterministic state transitions.

3.3 Just-In-Time (Expected) Sarsa

Just-in-time learning is not limited to Q-learning, but can be applied to other methods as well. In this section, we evaluate a JIT variant of Sarsa and of Expected Sarsa by comparing their performance with their regular counterparts on the cliff walking task (see Figure 3.1).

Algorithm 7 shows the pseudo-code for JIT Sarsa as well as JIT Expected Sarsa. Note that JIT Sarsa performs twice per timestep an action selection, once to select an action for the update target and once to select an action for control. Although an alternative version of JIT Sarsa can be made that only select an action once per timestep, the advantage of the version shown in Algorithm 7 is that the pseudo-code is less complicated. In addition, this version has a performance edge when dealing with MDPs with returning actions, i.e., actions that produce a next state that is equal to the current state. The reason is that when the update target is based on the actual action executed at the next timestep, this action has to be selected one timestep before it is needed. Therefore, the Q-values used to select this action, miss out on the last update, which can hurt performance if the next state is equal as the current state. The version of JIT Sarsa shown in Algorithm 7 avoids this.

Algorithm 7 JIT (Expected) Sarsa

initialize $Q(s,a)$ arbitrarily for all s,a
initialize $S'(s) = \emptyset$ for all s
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    if $S'(s) \neq \emptyset$ **then**
      if Sarsa **then**
        select action a', based on $Q(S'(s), \cdot)$
        $\upsilon = R'(s) + \gamma\, Q(S'(s), a')$
      if Expected Sarsa **then**
        $V' = \sum_a' \pi(S'(s), a') \cdot Q(S'(s), a')$
        $\upsilon = R'(s) + \gamma\, V'$
      $Q(s, \bar{a}) \leftarrow (1 - \alpha^{s\bar{a}}) \cdot Q(s, \bar{a}) + \alpha^{s\bar{a}}\, \upsilon$ // $\bar{a} = A(s)$
    select action a, based on $Q(s, \cdot)$
    take action a, observe r and s'
    $S'(s) \leftarrow s'$; $R'(s) \leftarrow r$; $A(s) \leftarrow a$
    $s \leftarrow s'$
  until s is terminal

We compare the average return over the first 50 episodes for Sarsa, Expected Sarsa and Q-learning and their just-in-time counterparts on the cliff walking task from Figure 3.1. We average the results over 10.000 independent runs. Each method uses ε-greedy action selection with $\varepsilon = 0.05$. The learning rates have an initial value of 1 and are decayed according to Equation 3.13. We optimize the decay parameter d in the range from 0 to 1 with steps of 0.01. The initial Q-values are set to 0, so optimistic initialization is used.

Figure 3.14 plots the return as a function of the number of episodes, while Table 3.2

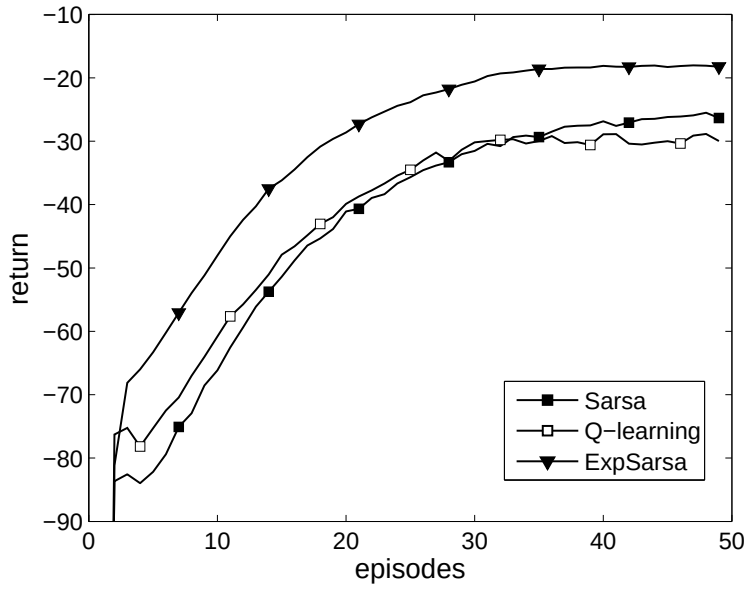

Figure 3.14: Comparison of the performance of JIT Q-learning, JIT Sarsa and JIT Expected Sarsa on the cliff walking task.

shows the average return and optimal learning rate decay. The just-in-time versions outperform their regular counterparts for all three methods. In case of Q-learning and Expected Sarsa the performance difference is about 5%, while for Sarsa the difference is abut 10%. The larger performance gap for Sarsa can be explained by the improved strategy for returning actions employed by JIT Sarsa. Surprisingly, the performance of (regular) Sarsa is worse than that of Q-learning, despite the fact that Sarsa converges in the limit to a higher value (see Section 3.1.4.1). The reason is that, when considering the 50 first episodes, in the early stages of learning Q-learning still has a performance advantage over Sarsa, which outweighs the performance advantage of Sarsa for the later episodes (see Figure 3.14).

Table 3.2: The performance of Q-learning, Sarsa and Expected Sarsa and their just-in-time counterparts on the cliff walking task shown in Figure 3.1.

	regular			just-in-time		
	d	average return	standard error	d	average return	standard error
Q-learning	0.01	-74.20	0.06	0.01	-70.89	0.06
Sarsa	0.01	-76.97	0.06	0.01	-69.88	0.07
Exp. Sarsa	0	-62.07	0.03	0	-58.84	0.03

Overall, these results demonstrate that the benefit of just-in-time learning is not limited to Q-learning, but applies to other methods as well.

3.4 Conclusion

In this chapter, we examined Expected Sarsa, a variation on the Sarsa algorithm intended to decrease the variance in the update rule. In addition, we presented just-in-time learning, a learning strategy that postpones updates until the moment the updated values are needed.

We proved that Expected Sarsa converges under the same conditions as Sarsa. We also proved that the variance in the update rule of Expected Sarsa is smaller than the variance for Sarsa and that the difference in variance is largest when there is a high amount of exploration and a large spread in Q-values of the actions of a specific state. We showed that in practise this translates in a performance advantage of Expected Sarsa compared to Sarsa and that the difference in performance is relatively high when the policy stochasticity is high (for example in case of an ε-greedy policy with $\varepsilon > 0.1$) and the environment stochasticity is low.

Just-in-time Q-learning is a variation on Q-learning that postpones Q-value updates until the moment these values are needed. We proved that by postponing Q-learning updates until a state is revisited, the update targets involved receive in general more updates, while the total number of updates for the actions of the current state stays the same. We demonstrated empirically that this leads to a performance improvement under a range of settings at similar computational cost. In addition, we empirically demonstrated that just-in-time learning causes also a performance improvement when combined with Sarsa or Expected Sarsa.

CHAPTER 4

Trading Space and Time for Performance

When performing value-function based RL, two major classes of methods are model-free and model-based learning. The difference in space requirements between these two classes can be huge. While typical model-free methods have a space complexity that is linear in the size of the state space, model-based methods are bounded by a space complexity that is quadratic in the state space size. This huge difference forms a disadvantage, since in a practical situation an RL agent gets assigned a certain amount of resources - in terms of computation time and memory space - and ideally should fully exploit these resources to get the maximum performance. When the agent has to resort to model-free methods when there is not enough space available for storing the full model, it does not optimally exploit its space resources and misses out on an opportunity for a better performance.

To avoid this limitation, methods can learn smaller, approximate models that require only a fraction of the space used by full model-based methods. Kearns and Singh (1999) show that, when using such sparse models, it is still possible to learn probably approximately correct policies. However, the performance of such methods is bounded by the quality of the model approximation. Furthermore, since the models may remain incorrect regardless of how much sample experience is gathered, such methods are not guaranteed to find optimal policies even in the limit.

In this chapter, we present and evaluate *best-match learning*, a new approach for trading off the strengths of model-based and model-free methods. Best-match learning works by approximating the solution to a set of *best-match equations*, which combine a sparse model with a model-free Q-value function constructed from samples not used by the model. We prove that, unlike regular sparse model-based methods, best-match learning is guaranteed to converge to the optimal policy in the tabular case. This guarantee holds even when using a *last-visit model* (LVM), which stores only the last observed reward and transition state for each state-action pair.

In addition, we present an extensive empirical analysis, comparing the performance of best-match learning to several algorithms with similar space requirements. These results demonstrate that best-match learning can outperform regular sparse model-based methods, as well as several model-free methods that strive to improve the sample efficiency of traditional TD methods. These include *eligibility traces* (Sutton, 1988; Watkins, 1989), which update recently visited states in proportion to a trace parameter; *experience replay* (Lin, 1992), which stores experience sequences and uses them for repeated TD updates; and *delayed Q-learning* (Strehl et al., 2006), which uses optimistic Q-value estimates to follow an approximately correct policy except for $\mathcal{O}(|\mathcal{S}||\mathcal{A}|\log(|\mathcal{S}||\mathcal{A}|))$ timesteps.

The remainder of this chapter is organized as follows. Section 4.1 extends the idea of just-in-time learning, introduced in Section 3.2, to best-match learning with an LVM, in which updates are continually revised such that the update targets constructed from them are more accurate. We show that best-match LVM evaluation is related to eligibility traces, by proving that under certain conditions they compute the same values. However, we also show that in arbitrary MDPs best-match LVM evaluation, unlike eligibility traces, performs updates that are unbiased with respect to initial state values. We demonstrate empirically that, as a result, it can substantially outperform TD(λ) despite using similar space and computation.

Section 4.1 also addresses the control case. We propose an efficient best-match LVM algorithm that uses *prioritized sweeping* (Moore and Atkeson, 1993), a well-known technique for prioritizing model-based updates, to trade off extra computation for improved performance. We prove that, despite the use of a sparse model, this approach converges to the optimal Q-values under the same conditions as Q-learning. In addition, we demonstrate empirically that it can substantially outperform competitors with similar space requirements.

Section 4.2 proposes a best-match learning algorithm that uses an *n-transition model* (NTM), which maintains an estimate of the transition probability for n transition states per state action pair. By tuning n, the space requirements can be controlled. We prove that the algorithm converges to the optimal Q-values for any value of n. We demonstrate empirically the resulting performance improvement over regular sparse model-based methods with equal space requirements, whose performance is bounded by the quality of the model approximation.

Section 4.3 proposes *best-match function approximation*, which demonstrates that best-match learning is useful beyond the tabular case. In particular, we combine best-match learning with gradient-descent function approximation and show empirically that it can outperform Sarsa(λ) and experience replay with linear function approximation while using similar computation.

Section 4.4 discusses the theoretical and empirical results, Section 4.5 outlines future work, and Section 4.6 concludes.

4.1 Best-Match Last-Visit Model

In this section, we demonstrate that updates can be postponed much further than is done by JIT Q-learning (see Section 3.2), without negatively affecting other updates, when *best-match updates* are performed. Best-match updates are updates that can correct previous updates when more recent information becomes available. This insight leads to the derivation of the *best-match last-visit model equations*, which combine a *last-visit model* (LVM), consisting of the last experienced reward and transition state for each state-action pair, with *model-free Q-values*, constructed from model-free updates of all observed samples, except the ones stored in the LVM. We present an evaluation as well as a control algorithm based on solving these equations and empirically demonstrate that these methods can outperform competitors with similar space requirements.

4.1.1 Best-Match LVM Equations

In the example presented in Section 3.2, the update of $Q(s_A, a_0)$ is postponed until state s_A is revisited. In this section, we demonstrate that the update can be postponed even further in the case that a different action is selected upon revisit. Since we will consider multiple updates per timestep in this section, we denote the Q-value function using two iteration indices: t and i. Each time an update occurs, i is increased, while each time an action is taken, t is increased and i is reset to 0. Therefore, if I denotes the total number of updates that occurs at time t, by definition $Q_{t,I} = Q_{t+1,0}$. Action selection at time t is based on $Q_{t,I}$. Using this convention, the regular Q-learning update can be written as

$$Q_{t+1,1}(s_t, a_t) = (1-\alpha)Q_{t+1,0}(s_t, a_t) + \alpha[r_{t+1} + \max_{a'} Q_{t+1,0}(s_{t+1}, a')] \,.$$

Now consider the example shown in Figure 4.1, which extends Figure 3.10 to include a second revisit of s_0 at timestep $t = 7$. Suppose that a different action is selected on the first revisit, that is, $a_4 \neq a_0$. Using just-in-time updates, the Q-value of state-action pair (s_A, a_0) gets updated at time $t = 4$. Using the two indices convention we can rewrite Equation 3.10 as[1]

$$Q_{4,1}(s_A, a_0) = (1-\alpha)Q_{1,0}(s_A, a_0) + \alpha[r_1 + \gamma \max_a Q_{4,0}(s_B, a)] \,. \tag{4.1}$$

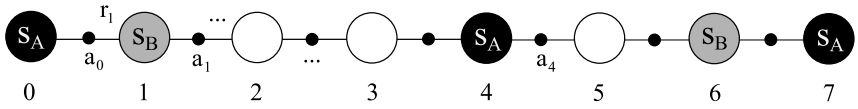

Figure 4.1: A state transition sequence in which best-match updates can enable further postponing. Timesteps are shown below each state.

To perform this update, the experience set (r_1, s_B) resulting from taking action a_0 in s_A is temporarily stored. With JIT Q-learning, this experience is stored per state. If the state is revisited and a new action is taken, the previous experience is overwritten and lost. However, if the experience is stored per state-action pair, then the previous experience is not overwritten until the same action is selected again. If the same action is not selected upon revisit, the experience can be used again to redo the update at a later time, using more recent values for the next state. In the example from Figure 4.1, the update of (s_A, a_0) can be redone at timestep 7:

$$Q_{7,1}(s_A, a_0) = (1-\alpha)Q_{1,0}(s_A, a_0) + \alpha[r_1 + \gamma \max_a Q_{7,0}(s_B, a)] \,. \tag{4.2}$$

Since state s_B is revisited at timestep 6, (s_B, a_1) has received an extra update and therefore $Q_{7,0}(s_B, a_1)$ is likely to be more accurate than $Q_{4,0}(s_B, a_1)$.

[1] We use Q now instead of $\tilde{Q}$, since the only purpose of the tilde was to distinguish it from the Q-values of regular Q-learning.

Equation 4.2 is not equivalent to a (postponed) Q-learning update, in contrast to Equation 4.1, since $Q_{1,0}(s_A, a_0)$ is not equal to $Q_{7,0}(s_A, a_0)$ due to the update at timestep 4. Equation 4.2 corrects the update from timestep 4, by redoing it using the most recent Q-values for the update target. We call this update a *best-match update* (this name will be explained later in the section), while we call $Q_{1,0}(s_A, a_0)$ the *model-free Q-value* of (s_A, a_0).

Before formally defining a best-match update, we define the last-visit experience and the model-free Q-values.

Definition 1. *The last-visit experience of state-action pair (s, a) denotes the last-visit reward, $R'_t(s, a)$, that is, the reward received upon the last visit of (s, a), and the last-visit transition state, $S'_t(s, a)$, that is, the state transitioned to upon the last visit of (s, a). For a state-action pair that has not yet been visited, we define $R'_t(s, a) = \emptyset$ and $S'_t(s, a) = \emptyset$.*

The LVM consists of the last-visit experience from all state-action pairs.

Definition 2. *The model-free Q-value of a state-action pair (s, a), $Q^{mf}_t(s, a)$, is a Q-value that has received updates from all observed samples except those stored in the LVM, that is, $R'_t(s, a)$ and $S'_t(s, a)$. For a state-action pair that has not yet been visited, we define $Q^{mf}_t(s, a) = Q_{0,0}(s, a)$.*

While Q can be updated multiple times per timestep, Q^{mf} is updated only once per timestep. Therefore, it uses a single time index t. We define a best-match update as:

Definition 3. *A best-match update combines the model-free Q-value of a state-action pair with its last-visit experience from the same timestep according to*

$$Q_{t,i+1}(s, a) = (1 - \alpha)Q^{mf}_t(s, a) + \alpha[R'_t(s, a) + \gamma \max_{a'} Q_{t,i}(S'_t(s, a), a')] \,.$$

Using best-match updates to extend the postponing period of a sample update requires additional computation, as the agent typically performs multiple best-match updates per timestep. In the example, at timestep 7 the agent redoes the update of $Q(s_A, a_0)$, but also performs an update of $Q(s_A, a_4)$.

The model-free Q-value function is updated only once per timestep. Specifically, at timestep $t + 1$ Q^{mf} is updated according to

$$Q^{mf}_{t+1}(s_t, a_t) = Q_{t+1,0}(s_t, a_t) \,. \tag{4.3}$$

Assuming (s_t, a_t) has received a best-match update at timestep t, Equation 4.3 is equivalent to the update

$$Q^{mf}_{t+1}(s_t, a_t) = (1 - \alpha)Q^{mf}_t(s_t, a_t) + \alpha[R'_t(s_t, a_t) + \gamma \max_{a'} Q_{t,i}(S'_t(s_t, a_t), a')] \,,$$

where the value of i depends on the order of best-match updates at timestep t. After Q^{mf} has been updated, the last-visit experience for (s_t, a_t) is overwritten with the new experience

$$\begin{aligned} R'_{t+1}(s_t, a_t) &= r_{t+1} \,, \\ S'_{t+1}(s_t, a_t) &= s_{t+1} \,. \end{aligned}$$

In the approach described above, best-match updates are used to postpone the update from a sample without negatively affecting other updates or the action selection process. However, best-match updates can be exploited far beyond simply avoiding these negative effects. As an example, consider the state-action sequence in Figure 4.2. s_B is not revisited before the revisit of s_A. With the update strategy described above, best-match updates occur only when a state is revisited. Consequently, the experience from (s_B, a_1) is not used in the update target of (s_A, a_0). However, it is not necessary to wait for a revisit of s_B to perform a best-match update. Instead, it can be performed at the moment it is needed: when s_A is revisited. Thus, if at timestep 3 the agent performs a best-match update of $Q(s_B, a_1)$, before updating $Q(s_A, s_0)$, the latter update will exploit more recent Q-values for s_B, just as if s_B had been revisited.

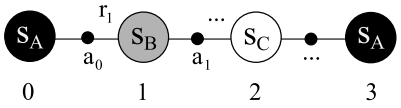

Figure 4.2: A state transition sequence in which s_B is not revisited. Timesteps are shown below each state.

Taking this idea further, the agent can first update the Q-values of s_C before updating the Q-values of s_B. In other words, the agent uses the Q-values of s_A to perform a best-match update of s_C, then performs a best-match update of s_B and finally updates s_A. However, once the Q-values of s_A have changed, it is possible to further improve the Q-values of s_C by performing a new best-match update. The new Q-values of s_C can then be used to redo the update of s_B, which in turn can be used to re-update s_A. This process can repeat until the Q-values reach a fixed point, which is the solution to a system of $|\mathcal{S}||\mathcal{A}|$ *best-match LVM equations*. We call this solution the *best-match Q-value function*, Q^B, which forms the best match between the LVM and the model-free Q-values.

Definition 4. *The best-match LVM equations at timestep t are defined as*

$$Q_t^B(s,a) = \begin{cases} (1-\alpha_t^{sa})\, Q_t^{mf}(s,a) + \\ \qquad \alpha_t^{sa}\ [R'_t(s,a) + \gamma\, \max_c Q_t^B(S'_t(s,a),c)] & \text{if } S'_t(s,a) \neq \emptyset \\ Q_t^{mf}(s,a) & \text{if } S'_t(s,a) = \emptyset\,. \end{cases}$$

There are different ways to look at these equations. One way is to see them as the limit case of redoing updates using (in general) increasingly more accurate update targets. Another way is to see them as Bellman optimality equations based on an induced model. For state-action pair (s,a) this induced model can be described as a transition with probability α to state $S'(s,a)$ with a reward of $R'(s,a)$ and a transition with probability $1-\alpha$ to a terminal state s_T (with a value of 0) and a reward of $Q^{mf}(s,a)$ (see Figure 4.3).[2]

[2]We assume $S'_t(s,a) \neq \emptyset$ for (s,a) in this case.

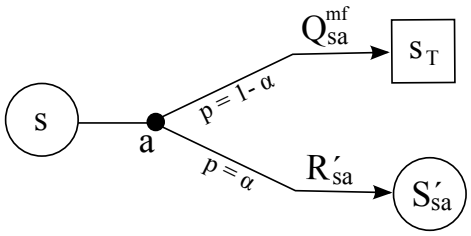

Figure 4.3: Illustration of the induced model for state-action pair (s, a) corresponding with the best-match LVM equations. The small black dot represents the stochastic action a leading with probability α to state $S'(s, a)$ and with probability 1-α to state s_T.

The advantage of solving the Bellman optimality equations for this induced model, compared to solving it using only the LVM, is that the bias towards the samples in the LVM can be controlled using the learning rates. With annealing learning rates, the transition probability to $S'_t(s, a)$ is decreased over time in favor of transition to the terminal state. On the other hand, when using only the LVM, the solution of the Bellman equations depends only on the samples of the LVM and does not take into account any previous samples. Clearly, in a stochastic environment, this will lead to a sub-optimal policy. Also when the solution is not computed exactly, but approximated by only performing a finite number of updates at each timestep (which is the case for any practical algorithm), using the induced model leads to a better performance, because of the strong bias towards the most recent samples that occurs when using only the LVM.

Section 4.1.3 discusses how to solve the best-match equations. However, we first discuss the policy evaluation case, for which analogous equations can be defined.

Definition 5. *The best-match LVM equations for state values at time* t *are*

$$V_t^B(s) = \begin{cases} (1-\alpha_t^s)V_t^{mf}(s) + \alpha_t^s \left[R'_t(s) + \gamma\, V_t^B(S'_t(s))\right] & \text{if } S'_t(s) \neq \emptyset \\ V_t^{mf}(s) & \text{if } S'_t(s) = \emptyset\,. \end{cases}$$

The model-free state values are updated according to $V^{mf}_{t+1}(s_t) = V_{t+1,0}(s_t)$.While in general the value function V can be seen as a special case of the action-value function Q (with all states only having a single action), V has a linear set of best-match equations, in contrast to Q, a property we exploit in best-match LVM evaluation.

4.1.2 Best-Match LVM Evaluation

In the evaluation case, the best-match LVM equations form a linear set that can be solved exactly. This section proposes an algorithm that does so in a computationally efficient way, using updates that are unbiased with respect to the initial state values.

The algorithm is based on two observations. First, not all $|\mathcal{S}|$ best-match equations necessarily depend on each other. The subset of equations needed to compute the best-match value for s_t can be found by iterating through the sequence of last-visit transition

states, starting with $S'(s_t)$. The corresponding N best-match equations form the linear set of equations to solve. For readability, we write s_t as $s_{[0]}$ and use the notation $s_{[n]} = S'(s_{[n-1]})$ and $r_{[n]} = R'(s_{[n-1]})$ for the subsequent transition state and reward. In addition, we use $\alpha^{[n]}$ for $\alpha^{s_{[n]}}$. The equations can now be written as

$$V^B(s_{[n]}) = (1-\alpha^{[n]})V^{mf}(s_{[n]})+\alpha^{[n]}\left[r_{[n+1]} + \gamma V^B(s_{[n+1]})\right] , \quad \text{for all } n \in [0, N-1] .$$

Second, the last state of this sequence, $s_{[N]}$, is always either a terminal state or the current state. Furthermore, none of the intermediate states can appear twice, making the N equations independent. This can be proven by contradiction. First, assume that the sequence has a dead-end, that is, ends with a state for which $S' = \emptyset$. This is impossible because it would cause the agent to get stuck in this state, preventing it from reaching the current state. Since last-visit information is maintained across episodes, $s_{[N]}$ is a terminal state if the path followed after the previous visit of s_t led to a terminal state. Next, assume the sequence contains the same intermediate state twice. After the second visit of this intermediate state, the subsequent sequence would be the same as after the first visit, since there is only a single last-visit next state defined per state. This would create an infinite sequence of next states, also preventing the agent from reaching the current state.

The set of equations can be solved by backwards substituting the equations, that is, substituting the equation for $V^B(s_{[n+1]})$ in the one for $V^B(s_{[n]})$ and so on until a single equation for $V^B(s_{[0]})$ remains of the form

$$V^B(s_{[0]}) = c_A + c_B\, V^B(s_{[N]}) ,$$

with c_A and c_B defined as

$$c_A = \sum_{i=0}^{N-1} \left((1-\alpha^{[i]})V^{mf}(s_{[i]}) + \alpha^{[i]} r_{[i+1]}\right) \prod_{k=0}^{i-1} \gamma\alpha^{[k]} , \tag{4.4}$$

$$c_B = \prod_{i=0}^{N-1} \gamma\alpha^{[i]} . \tag{4.5}$$

If $s_{[N]}$ is a terminal state, its value is 0 and $V^B(s_t) = c_A$. On the other hand, if $s_{[N]} = s_t$ then $V^B(s_t) = c_A/(1-c_B)$.

Algorithm 8 shows pseudocode of the on-line policy evaluation algorithm, which computes the best-match value of the current state at each timestep. Lines 7–11 compute the values of c_A and c_B in a forward, incremental way by going from one next state to the other. Note that it is not necessary to store V^{mf} and R' separately, since they are always used in the same combination, $(1-\alpha)\, V^{mf}(s)+\alpha\, R'(s)$, which is stored in a single variable, V_r^{mf}, saving space and computation. Line 17 combines the assignments $V^{mf}(s_t) = V(s_t)$, $R'(s_t) = r_{t+1}$ and the computation of V_r^{mf} in a single update. Note that the algorithm makes use of the just-in-time learning principle, that is, updating states at the moment of their revisit. In JIT Q-learning, it is used to improve the performance without increasing the computation cost, while in the best-match evaluation algorithm it is used to efficiently compute the best-match values.

Algorithm 8 Best-Match LVM Evaluation

initialize $V(s)$ arbitrarily for all s
initialize $S'(s) = \emptyset$ for all s
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    if $S'(s) \neq \emptyset$ **then**
      $c_A \leftarrow V_r^{mf}(s);\ c_B \leftarrow \gamma\,\alpha^s;\ s' \leftarrow S'(s)$
      while $s' \neq s \wedge s'$ is not terminal **do**
        $c_A \leftarrow c_A + c_B \cdot V_r^{mf}(s')$
        $c_B \leftarrow c_B \cdot \gamma\,\alpha^{s'}$
        $s' \leftarrow S'(s')$
      if $s' = s$ **then**
        $V(s) \leftarrow c_A/(1 - c_B)$
      else
        $V(s) \leftarrow c_A$
    take action $\pi(s)$, observe r and s'
    $V_r^{mf}(s) \leftarrow (1 - \alpha^s)V(s) + \alpha^s \cdot r$
    $S'(s) \leftarrow s';\ s \leftarrow s'$
  until s is terminal

Algorithm 8 is an on-line algorithm that computes at each timestep the best-match value of the current state. We define the off-line version as one that computes at the end of each episode the best-match values of the states that were visited during that episode. This off-line algorithm is related to off-line TD(λ), as demonstrated by the following theorem. We prove this theorem in Appendix A.

Theorem 3. *For an episodic, acyclic, evaluation task, off-line best-match LVM evaluation computes the same values as off-line TD(λ) with $\lambda_t = \alpha_t(s_t)$.*

For acyclic tasks, that is, episodic tasks with no revisits of states within an episode, $TD(\lambda)$ with $\lambda_t = \alpha_t(s_t)$ can perform TD updates that are unbiased with respect to the initial values (Sutton and Singh, 1994). Because of Theorem 3, this also holds for best-match LVM evaluation. However, in contrast to $TD(\lambda)$, best-match LVM evaluation can perform unbiased updates for any MDP, as we demonstrate with the following theorem, also proven in Appendix A.

Theorem 4. *The state values computed by the on-line best-match LVM evaluation algorithm (Algorithm 8) are unbiased with respect to the initial state values, when the initial learning rates $\alpha_0(s)$ are set to 1 for all s.*

Because best-match LVM evaluation can perform unbiased updates for any MDP, it can often substantially outperform TD(λ) while requiring similar space and computation. We demonstrate this empirically using the two tasks shown in Figure 4.4. Besides comparing against TD(λ), we also compare against experience replay (Lin, 1992), which stores the n last experience samples and uses them for repeated TD updates.

Task A features a small circular network consisting of four identical states, each having a deterministic transition to a neighbor. The reward received after each transition is +1. Task B is a stochastic variation on the first task, with stochastic transitions and a reward drawn from a normal distribution with mean 1 and standard deviation 0.5. The discount factor is 0.95, resulting in a state value of 20 on both tasks for all states. We compare the RMS error of the current state value $V_t(s_t)$ for all three methods. For experience replay, we performed a TD update for each of the last 4 samples at every timestep, resulting in a computation time similar to best-match LVM and TD(λ). In addition, we implemented a version where all observed samples are stored and updated at each timestep. The learning rate is initialized to 1 and decayed according to

$$\alpha^s = \frac{\alpha_0}{d \cdot [n(s) - 1] + 1} \,.$$

where $n(s)$ is the total number of times state s has been visited. We optimize d as well as λ between 0 and 1. Results are averaged over 5000 runs.

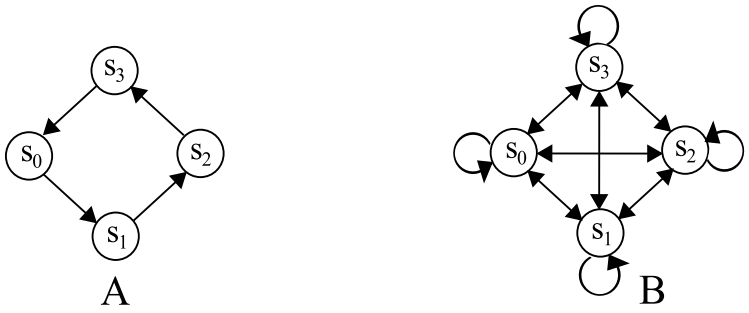

Figure 4.4: Two tasks for policy evaluation. Task A has deterministic state transitions and a deterministic reward of +1, while task B has stochastic transitions and a reward drawn from a normal distribution with mean +1 and standard deviation 0.5.

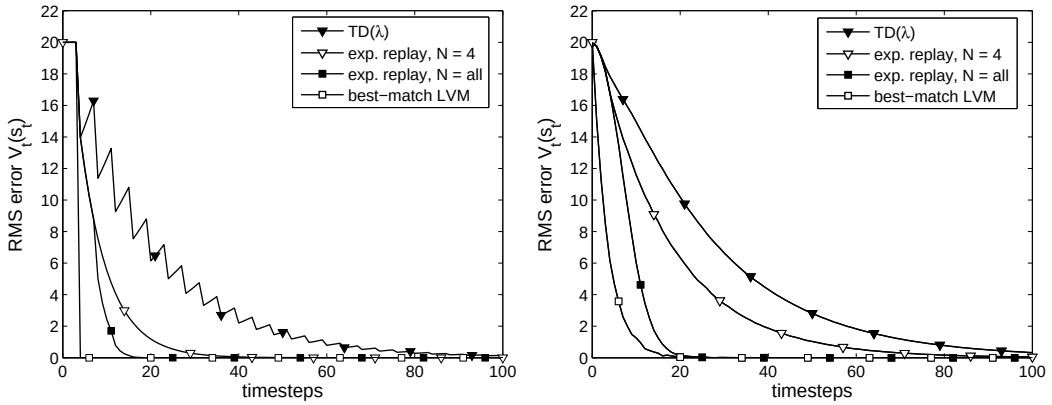

Figure 4.5: Comparison of the performance of best-match LVM, TD(λ) and experience replay on tasks A (left) and task B (right) of Figure 4.4.

Figure 4.5 shows the experimental results in these tasks. In task A, at timestep 4 the start state is revisited and the RMS error for best-match LVM drops to 0. The reason is that in the deterministic case the last-visit model is equal to the full model once every state has been visited. Furthermore, with learning rates of 1, the best-match LVM equations reduce to the Bellman optimality equations. Therefore best-match LVM effectively performs model-based learning. TD(λ), on the other hand, has to incrementally improve upon the initial values of 0. The spiky behavior of TD(λ) is caused by the combination of a λ of 1, with zero learning rate decay (which were the optimal settings in this case). Experience replay has a performance in between best-match LVM and TD(λ). In task B, the RMS error drops more smoothly. Best-match LVM again substantially outperforms TD(λ) and experience replay, even when all samples are stored and updated. The total computation time for the 5000 runs was marginally higher for experience replay with N=4, which has to maintain a queue of recent samples, than for best-match LVM and TD(λ): on task A, around 90 ms compared to 80 ms for both best-match LVM and TD(λ). Experience replay with all samples updated had a computation time of 280 ms. On task B, all methods were about 10 ms slower.

4.1.3 Best-Match LVM Control

The best-match LVM equations for the control case form a nonlinear set. Therefore, it is in general not possible to compute the exact best-match Q-values at each timestep. However, they can be approximated to arbitrary accuracy via update sweeps through the state-action space, in a manner similar to value iteration, as we prove in the following lemma.

Lemma 2. *For the best-match Q-values the following equation holds for all (s,a):*

$$Q_t^B(s,a) = \lim_{i\to\infty} Q_{t,i}(s,a)\,,$$

where $Q_{t,i}$ is initialized arbitrarily for $i = 0$ and is defined for $i > 0$ as

$$Q_{t,i}(s,a) = \begin{cases} (1-\alpha)\, Q_t^{mf}(s,a) + \\ \qquad \alpha\,[R'_t(s,a) + \gamma\, \max_{a'} Q_{t,i-i}(S'_t(s,a),a')] & \text{if } S'_t(s,a) \neq \emptyset \\ Q_t^{mf}(s,a) & \text{if } S'_t(s,a) = \emptyset\,. \end{cases}$$

Proof. For state-action pairs (s,a) with $S'_t(s,a) = \emptyset$ the proof follows directly from the definition of Q_t^B and $Q_{t,i}$. For (s,a) with $S'_t(s,a) \neq \emptyset$, the absolute difference between $Q_{t,i}(s,a)$ and $Q_t^B(s,a)$ can be written as

$$\begin{aligned} |Q_{t,i}(s,a) - Q_t^B(s,a)| &= \alpha\gamma\, |\max_c Q_{t,i-i}(S'_t(s,a),c) - \max_c Q_t^B(S'_t(s,a),c)| \\ &\leq \alpha\gamma \max_c |Q_{t,i-i}(S'_t(s,a),c) - Q_t^B(S'_t(s,a),c)| \\ &\leq \alpha\gamma\, ||Q_{t,i-i} - Q_t^B||\,. \end{aligned}$$

From this it follows that

$$||Q_{t,i} - Q_t^B|| \leq \alpha\gamma\, ||Q_{t,i-i} - Q_t^B||\,.$$

For $\alpha\,\gamma < 1$, it follows that for $i \to \infty$, $Q_{t,i} \to Q_t^B$. □

Lemma 2 shows that Q_t^B can be approximated to arbitrary accuracy with a finite number of best-match updates.

Algorithm 9 shows the pseudocode for a general class of algorithms that approximate the best-match Q-values by performing best-match updates.[3] Lines 9–12 perform a series of best-match updates. Note that while only a single Q^{mf} value is updated per timestep, many Q-values can be updated at the same timestep. By varying the way state-action pairs are selected for updating (line 10) and changing the stopping criterion (line 12), a whole range of algorithms can be constructed that trade off computation cost per timestep for better approximations of the best-match Q-values. Note that JIT Q-learning and even regular Q-learning are members of this general class of algorithms. If the state-action pair selection criterion is the state-action pair visited at the previous timestep and the stopping criterion allows only a single update, the algorithm reduces to the regular Q-learning algorithm. Thus, Q-learning is a form of best-match control with a simplistic approximation of the best-match Q-values. However, we reserve the term 'best-match learning' for algorithms that use the same sample multiple times to redo updates.

Algorithm 9 General Best-Match LVM Control

1: initialize $Q(s,a)$ arbitrarily for all s,a
2: initialize $S'(s,a) = \emptyset$ for all s,a
3: **loop** {over episodes}
4: initialize s
5: **repeat** {for each step in the episode}
6: select action a, based on $Q(s,\cdot)$
7: take action a, observe r and s'
8: $Q^{mf}(s,a) \leftarrow Q(s,a); S'(s,a) \leftarrow s'; R'(s,a) \leftarrow r$
9: **repeat**
10: select some $(\bar{s},\bar{a})$ pair with $S'(\bar{s},\bar{a}) \neq \emptyset$ {each pair is selected at least once before its revisit}
11: $Q(\bar{s},\bar{a}) \leftarrow (1-\alpha^{\bar{s}\bar{a}})Q^{mf}(\bar{s},\bar{a}) + \alpha^{\bar{s}\bar{a}}\ [R'(\bar{s},\bar{a}) + \gamma\ \max_c Q(S'(\bar{s},\bar{a}),c)]$
12: **until** some stopping criterion has been met
13: $s \leftarrow s'$
14: **until** s is terminal

The following theorem states that, for any member of the best-match LVM control class, the Q-values converge to the optimal Q-values.

Theorem 5. *The Q-values of a member of the best-match LVM control class, shown in Algorithm 9, converge to Q^* if the following conditions are satisfied:*

1. *S and A are finite.*
2. *$\alpha_t(s,a) \in [0,1]$, $\sum_t \alpha_t(s,a) = \infty$, $\sum_t(\alpha_t(s,a))^2 < \infty$ w.p.1 and $\alpha_t(s,a) = 0$ unless $(s,a) = (s_t,a_t)$.*

[3] Similar to the variable V_r^{mf} of Algorithm 8, a variable Q_r^{mf} can be defined that combines the variables Q^{mf} and R', saving space and computation. For readability we do not show this for Algorithm 9.

3. $Var\{R(s,a,s')\} < \infty$.

4. $\gamma < 1$.

We prove this theorem in Appendix C.3.

4.1.4 Best-Match LVM Prioritized Sweeping

A wide range of methods can be constructed within the general class of best-match LVM control algorithms that trade off increased computation time for better approximation of the best-match Q-values in different ways. This section proposes one method that performs this trade-off with a strategy based on *prioritized sweeping* (PS) (Moore and Atkeson, 1993).

PS makes the planning step of model-based RL more efficient by focusing on the updates expected to have the largest effect on the Q-value function. The algorithm maintains a priority queue of state-action pairs in consideration for updating. When a state-action pair (s,a) is updated, all predecessors (i.e., those state-action pairs whose estimated transition probabilities to s are greater than 0) are added to the queue according to a heuristic estimating the impact of the update. At each timestep, the top N state-action pairs from this queue are updated, with N depending on the available computation time. Because PS maintains a full model, it requires $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ space.

This same idea can be applied to the best-match equations for efficient approximation of the best-match values. A priority queue of state-action pairs is maintained whose corresponding best-match updates have the largest expected effect on the best-match Q-value estimates. When a state-action pair has received an update, all state-action pairs whose last-visit transition state equals the state from the updated state-action pair are placed into the priority queue with a priority equal to the absolute change an update would cause in its Q-value. Since this approach uses only an LVM, it requires only $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ space.

Algorithm 10 shows the pseudocode of this algorithm, which we call *best-match LVM prioritized sweeping* (BM-LVM). By always putting the state-action pair from the previous timestep on top of the priority queue (line 10), the requirement that each visited state-action pair receives at least one best-match update is fulfilled, guaranteeing convergence in the limit.

On the surface, this algorithm resembles *deterministic prioritized sweeping* (DPS) (Sutton and Barto, 1998), a simpler variation that learns only a deterministic model, uses a slightly different priority heuristic, and performs Q-learning updates to its Q-values. While clearly designed for deterministic tasks, it can also be applied to stochastic tasks, in which case updates are based on an LVM.

However, there is a crucial difference between DPS and BM-LVM. By performing updates with respect to Q^{mf} instead of Q, BM-LVM corrects previous updates instead of performing multiple updates based on the same sample. This ensures proper averaging of experience and enables convergence to the optimal Q-values using only an LVM, even in stochastic environments. This is not guaranteed for DPS since if some samples are used more often than others a bias towards these samples is created, which can prevent convergence to the optimal Q-values.

Algorithm 10 Best-Match LVM Prioritized Sweeping (BM-LVM)

initialize $Q(s,a)$ arbitrarily for all s,a
initialize $S'(s,a) = \emptyset$ for all s,a
initialize PQueue as an empty queue
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    select action a, based on $Q(s,\cdot)$
    Take action a, observe r and s'
    $S'(s,a) \leftarrow s'; R'(s,a) \leftarrow r; Q^{mf}(s,a) \leftarrow Q(s,a)$
    promote (s,a) to top of priority queue
    $n \leftarrow 0$
    while $(n < N) \wedge$ (*PQueue* is not empty) **do**
      $s_1, a_1 \leftarrow first(PQueue)$
      $Q(s_1,a_1) \leftarrow (1 - \alpha^{s_1 a_1})\, Q^{mf}(s_1,a_1) + \alpha^{s_1 a_1} [R'(s_1,a_1) + \gamma \max_c Q(S'(s_1,a_1),c)]$
      $V_{s_1} \leftarrow max_{a'} Q(s_1,a')$
      for all $(\bar{s},\bar{a})$ with $S'(\bar{s},\bar{a}) = s_1$ **do**
        $p \leftarrow |(1 - \alpha^{\bar{s}\bar{a}})\, Q^{mf}(\bar{s},\bar{a}) + \alpha^{\bar{s}\bar{a}} [R'(\bar{s},\bar{a}) + \gamma\, V_{s_1}] - Q(\bar{s},\bar{a})|$
        if $p > \theta$ **then**
          insert $(\bar{s},\bar{a})$ into *PQueue* with priority p
      $n \leftarrow n + 1$
    $s \leftarrow s'$
  until s is terminal

We compare the performance of PS, DPS, and BM-LVM on the deterministic and stochastic variation of the Dyna maze task shown in Figure 3.12. In addition, we also compare to Q(λ) as described by (Watkins, 1989). This is an off-policy control version of eligibility traces. We also tried Sarsa(λ), the on-policy version, since it can sometimes outperform Q(λ) considerably, but saw no significant difference for these experiments and present only the Q(λ) results. Note that when a greedy behavior policy is used, as in the deterministic experiment, Q(λ) computes exactly the same values as Sarsa(λ). As in Section 4.1.2, we also compare to experience replay.

Finally, we compare to delayed Q-learning (Strehl et al., 2006), a model-free method that, like some model-based methods (Brafman and Tennenholtz, 2002; Kearns and Singh, 2002; Strehl and Littman, 2005), is proven to be *probably approximately correct* (PAC), that is, its sample complexity is polynomial with high probability. Delayed Q-learning initializes its Q-values optimistically and ensures that value estimates are not reduced until the corresponding state-action pairs have been sufficiently explored. Because it does not maintain a model, it has the same $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ space requirements as best-match prioritized sweeping. However, to our knowledge, its empirical performance has never been evaluated before.

For each method, the free parameters are optimized within a certain range. In the

deterministic case, for Q(λ) we optimized the λ value in the range from 0 to 1, and the learning rate decay d (using Equation 3.13) in the range from 0 to 1, while α_0 was set to 1. We also optimized the (unbounded) trace type (replacing versus accumulating). For delayed Q-learning we optimized m in the range from 1 to 5 with steps of 1 and e_1 in the range 0 to 0.020 with steps of 0.001. For DPS and BM-LVM, we did not optimize any parameters in the deterministic case, but simply used a constant α of 1. In the stochastic case, we also optimized the learning rate decay d for DPS and BM-LVM.

For all methods, we used optimistic initialization with $Q_0 = 20$ in order to get a fair comparison with delayed Q-learning, for which initialization to $R_{max}/(1-\gamma)$ is part of the algorithm.[4]

In the deterministic case we used a greedy behavior policy, while we used an ε-greedy policy with $\varepsilon = 0.1$ in the stochastic variant. For all prioritized-sweeping algorithms we performed a maximum of 20 updates per timestep (i.e., N = 20). For experience replay we used the last 20 samples, which also results in 20 updates per timestep. Results are averaged over 1000 independent runs.

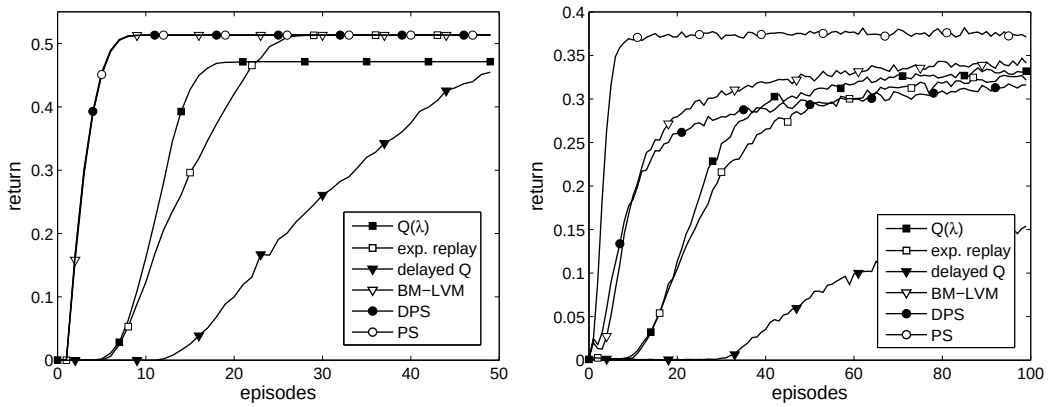

Figure 4.6: Comparison of the performance of BM-LVM and several competitors on the deterministic (left) and stochastic (right) Dyna maze task.

Figure 4.6 shows the return as a function of the number of episodes, while Tables 4.1 and 4.2 show the average return over the measured episodes and the optimal parameter values. In the deterministic experiment, we see that the performance of PS, DPS, and BM-LVM is exactly equal, as expected when $\alpha = 1$, since the last-visit experience is equal to the model of the environment. Q(λ) performs considerably worse than the prioritized sweeping methods and does not converge to the optimal policy. In contrast, the combination of a greedy behavior policy with optimistic initialization enables the prioritized sweeping methods to converge to the optimal policy in a deterministic environment. Experience replay performs similarly to Q(λ), though it does converge to the optimal policy.

[4]For this task $r = R_{max}$ only when the exit is reached and 0 otherwise. Thus, the Q-values can never be higher than 1 and $Q_0 = 20$ is overly optimistic. However, since realizing that an initialization of 1 is possible would require extra prior knowledge, we initialize to 20.

	deterministic - 50 eps.			
	optimal parameters	average return	standard error	time per step $(\cdot 10^{-6} s)$
Q(λ)	λ: 0.8, d: 0	0.3606	0.0007	0.68
exp. replay	d: 0	0.3602	0.0004	0.37
delayed Q	m: 1, $e_1 = 0$	0.1878	0.0004	0.11
BM-LVM	d: 0	0.4769	0.0002	0.88
DPS	d: 0	0.4774	0.0002	0.85
PS	-	0.4772	0.0002	0.95

Table 4.1: Average return and optimal parameters (d = α decay rate) of best-match prioritized sweeping and several competitors on the deterministic Dyna maze task.

	stochastic - 100 eps.			
	optimal parameters	average return	standard error	time per step $(\cdot 10^{-6} s)$
Q(λ)	λ: 0.9, d: 0.03	0.2417	0.0007	0.59
exp. replay	d: 0.18	0.2272	0.0006	0.43
delayed Q	m: 2, e_1:0.015	0.0668	0.0004	0.12
BM-LVM	d: 0.02	0.2911	0.0006	3.2
DPS	d: 0.30	0.2683	0.0008	3.7
PS	-	0.3603	0.0004	4.7

Table 4.2: Average return and optimal parameters (d = α decay rate) of best-match prioritized sweeping and several competitors on the stochastic Dyna maze task.

Delayed Q-learning also converges to the optimal policy, as predicted by the theory, but does so much more slowly.

In the stochastic experiment, PS has a clear performance advantage. However, the goal of BM-LVM is not to match or even come close to the performance of PS. It cannot match this performance in general, since PS takes advantage of its higher space complexity. Instead, the goal of BM-LVM is to optimally perform at a space complexity of $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$. The results confirm that BM-LVM is considerably better than the other methods with this space complexity, like Q(λ) and DPS. DPS initially performs well, but cannot keep up with BM-LVM after about 10 episodes, even though BM-LVM has similar space and computation costs per timestep. Experience replay performs slightly worse than Q(λ). We tested whether doubling the size of the stored experience sequence improves the performance of experience replay, but this led to no significant performance increase. Delayed Q-learning also performs poorly in the stochastic case, despite its PAC bounds.

The computation time of BM-LVM, DPS and PS is in the deterministic experiment considerably lower than in the stochastic case. The reason for this is that while in both cases the maximum number of updates per timestep is 20, in the deterministic case the priority queue often has fewer than 20 samples, so fewer updates occur. The computation time of Q(λ) is slightly better than that of BM-LVM, while experience replay is about twice

as fast as BM-LVM.

In the stochastic experiment, the computation time of Q(λ) is much better than that of any of the prioritized sweeping algorithms, which could suggest that Q(λ) is a better choice than BM-LVM when computation power is scarce. To test this hypothesis, we performed additional experiments with smaller values of N. The computation time for BM-LVM for $N = 4$ ($0.61 \cdot 10^{-6}$ s) was similar to that of Q(λ). The average return of BM-LVM dropped to 0.2598 in this case, which is still considerably better than the average return of Q(λ). This demonstrates that BM-LVM is a better choice than Q(λ) even under severe computational constraints.

Together, these results clearly demonstrate the strength of best-match learning, since BM-LVM outperforms several competitors with similar space complexity. However, the results also show that the performance gap with full model-based learning can be considerable. Therefore, if more space is available, a better approximate model would be preferred. We address this need in the next section by applying best-match learning to an n-transition model, which estimates the transition function for n next states per state-action pair, allowing increased space requirements to be traded for improved performance.

4.2 Best-Match n-Transition Model

The best-match LVM equations described above combine model-free Q-values with the last-visit model. When state-action pairs have only a small number of possible next states, the last-visit model can effectively approximate the full model. In other cases, however, the last-visit model captures only a fraction of the full model and the effect of the best-match updates will be small. In this section, we combine best-match learning with the n-transition model, which estimates the transition probability for n possible next states of each state-action pair. By tuning n, increased space requirements can be traded for improved performance.

4.2.1 Generalized Best-Match Equations

Best-match LVM learning takes the idea of using more accurate update targets to the extreme by continuously revising update targets with best-match updates. For a specific sample, the update target is revised until the moment of revisit of the corresponding state-action pair, since at that moment the sample is overwritten with the newly collected sample. However, if space allows, the new sample can be stored along with the old sample instead of overwriting it, allowing the update target from the new as well as the old sample to be further improved. We explain with an example how this changes the best-match equations.

Consider the state-action sequence from Figure 4.7 and assume the best-match Q-values are computed at each timestep. At the revisit of s_A, action a_0 is retaken. Therefore, when using the LVM, at timestep 5 the old experience sample is overwritten with the new experience. Before this occurs, the old experience is used in a final update of Q^{mf}. Let v_y^x indicate the update target from the sample collected at timestep x based on the best-match

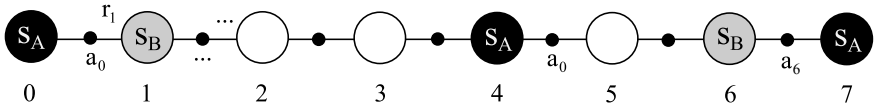

Figure 4.7: A state transition sequence in which best-match updates can enable further postponing. Timesteps are shown below each state.

Q-value of timestep y: $v_y^x = r_x + \gamma \max_a Q_y^B(s_x, a)$. Using this convention the update of Q^{mf} at timestep 5 becomes

$$Q_5^{mf}(s_A, a_0) = (1-\alpha) Q_0^{mf}(s_A, a_0) + \alpha v_4^1 \,.$$

At timestep 7, the best-match LVM equation for (s_A, a_0) can be written as

$$\begin{aligned} Q_7^B(s_A, a_0) &= (1-\alpha) Q_7^{mf}(s_A, a_0) + \alpha v_7^5 \\ &= (1-\alpha) Q_5^{mf}(s_A, a_0) + \alpha v_7^5 \\ &= (1-\alpha)^2 Q_0^{mf}(s_A, a_0) + \alpha\,(1-\alpha) v_4^1 + \alpha v_7^5 \,. \end{aligned}$$

Thus, the best-match Q-value of (s_A, a_0) at timestep 7 is equal to a weighted average of Q_0^{mf}, v_4^1 and v_7^5. On the other hand, if both the old and the new sample are stored, Q-values from timestep 7 could also be used for the update target of the old sample, yielding

$$Q_7^B(s_A, a_0) = (1-\alpha)^2 Q_0^{mf}(s_A, a_0) + \alpha\,(1-\alpha) v_7^1 + \alpha v_7^5 \,. \tag{4.6}$$

For the state-sequence from Figure 4.7 this means that the experience resulting from (s_B, a_6) is also taken into account in the update target for (s_A, a_0).

The above example shows how the best-match LVM equations can be naturally extended to two samples per state-action pair. Following the same pattern, we can define best-match equations given an arbitrary set of samples. Consider the set of samples X of size N_X, where a sample $x \in X$ has the form $\{s, a, r, s'\}$. These samples can be grouped according to their state-action pairs. We define X_{sa} as the subset of X containing all samples belonging to state-action pair (s, a) and N_{sa}^x as the size of X_{sa}. Without loss of generality, we index the samples from X_{sa} as x_k^{sa} for $1 \le k \le N_{sa}^x$. In addition, we define W_{sa} as a set consisting of $N_{sa}^x + 1$ weights $w_k^{sa} \in \mathbb{R}$ such that $0 \le w_k^{sa} \le 1$ for $0 \le k \le N_{sa}^x$ and $\sum_{k=0}^{N_{sa}^x} w_k^{sa} = 1$. We define W as the union of the weight sets from all state-action pairs.

Definition 6. *The generalized best-match equations with respect to Q_t^{mf}, X and W are*

$$Q_t^B(s,a) = w_0^{sa}\, Q_t^{mf}(s,a) + w_1^{sa} v_1^{sa} + w_2^{sa} v_2^{sa} + \ldots + w_{N_{sa}^x}^{sa} v_{N_{sa}^x} \,, \qquad \textit{for all } s, a \,, \tag{4.7}$$

where $v_k^{sa} = r + \gamma\, \max_c Q_t^B(s', c)\ | r, s' \in x_k^{sa}$.

Note that Equation 4.7 reduces to $Q_t^B(s,a) = Q_t^{mf}(s,a)$ for state-action pairs with no samples in X.

Within this context, Q^{mf} is defined as a model-free Q-value constructed from all observed samples except those in X. Consequently, when a sample is removed from X, it is used for a model-free update of Q^{mf}.

Using Definition 6, a range of algorithms can be constructed based on different sets of samples X and weights W. When the samples are combined by incremental Q-learning updates, like in Equation 4.6, the weights have the values

$$w_0^{sa} = \prod_{i=1}^{N_{sa}^x} (1 - \alpha_i^{sa}), \tag{4.8}$$

$$w_k^{sa} = \alpha_k^{sa} \prod_{i=k+1}^{N_{sa}^x} (1 - \alpha_i^{sa}), \qquad \text{for } 1 \leq k \leq N_{sa}^x. \tag{4.9}$$

With this weight distribution, the update targets from older samples have lower weights than more recent samples. In Q-learning, more recent samples in general have more accurate update targets so giving them higher weight makes sense. However, in best-match learning the update targets from all stored samples have the same time index so there is no reason to use different weights for them. A better weight distribution gives all samples the same weights:

$$w_k^{sa} = (1 - w_0^{sa})/N_{sa}^x, \qquad \text{for } 1 \leq k \leq N_{sa}^x,$$

for some value of w_0^{sa}.

The last-visit model, storing one sample for each state-action pair, is one possible sample set. A straightforward extension is to store n samples per state-action pair. In the following section, however, we propose a different sample set, called the n-transition model, which can be stored more compactly.

4.2.2 Best-Match Learning based on the n-transition Model

While BM-LVM outperforms model-free methods with the same space complexity, it does not perform as well as PS, which stores a full model. This is symptomatic of an important limitation of BM-LVM: it offers only a single trade-off between space and performance. When there is not enough space available to store the full model, but more than enough to store the LVM, a more sophisticated method is needed to make maximal use of the available space. Using the generalized best-match equations, we can construct such a method.

An obvious approach is to store n samples per state-action pair. However, obtaining an accurate model often requires a large n, even when the number of next states per state-action pair is small. A more space-efficient solution is to group together samples that have the same next state. If we store the size of such a group in $N_{sas'}^x$ and give each sample a weight of $1/N_{sa}$, where N_{sa} is the total number of times state-action pair (s, a) is visited, then we can rewrite the contribution from all samples of X_{sa} to the best-match equations as

$$\sum_{k=1}^{N_{sa}^x} w_k v_k = \frac{1}{N_{sa}} \left[\sum_X r_{sa} + \gamma \sum_{s'} N_{sas'}^x \max_{a'} Q^B(s', a') \right],$$

where $\sum_X r_{sa}$ is the sum of the rewards from all samples in the sample set belonging to (s, a). Using $w_0^{sa} = 1 - N_{sa}^x/N_{sa}$, $\hat{\mathcal{P}}_{sa}^{s'} = N_{sas'}^x/N_{sa}^x$ and $\hat{\mathcal{R}}_{sa} = \sum_X r_{sa}/N_{sa}^x$, the generalized best-match equations can now be rewritten as

$$Q^B(s,a) = w_0^{sa} Q^{mf}(s,a) + (1 - w_0^{sa}) \left[\hat{\mathcal{R}}_{sa} + \gamma \sum_{s'} \hat{\mathcal{P}}_{sa}^{s'} \max_{a'} Q^B(s', a') \right], \quad \text{for all } s, a\,.$$

In these equations, $\hat{\mathcal{P}}$ and $\hat{\mathcal{R}}$ constitute a sparse, approximate model, whose size can be controlled by limiting the number of next states per state-action pair for which $\hat{\mathcal{P}}$ is estimated. w_0^{sa} is the fraction of all samples belonging to (s, a) not used by the sparse model. We define an *n-transition model* (NTM) to be one that estimates the transition probability $\hat{\mathcal{P}}$ for n next states per state action pair. Once a sample enters the model, that is, is used to update $\hat{\mathcal{P}}$, it stays in the model. Each sample not used to update the model is used for a model-free update of Q^{mf}. Different strategies can be used to determine which samples enter the model. A simple approach is to use the first n unique next states that are encountered for a specific state-action pair.

Algorithm 11 shows general pseudocode for best-match NTM learning. The algorithm presents two trade-offs. First, the space complexity can be traded off with performance by selecting n. Second, the computation time per simulation step can be traded off with performance by controlling the number of best-match updates performed per timestep.

Based on this general control algorithm, various specific algorithms can be constructed using different stopping criteria and strategies for selecting state-action pairs to receive best-match updates. The following theorem states that, for any member of this class, the Q-values converge to the optimal Q-values. We prove this theorem in Appendix C.4.

Theorem 6. *The Q-values of a member of the best-match NTM control class, shown in Algorithm 11, converge to Q^* if the following conditions are satisfied:*

1. *S and A are finite.*
2. *$\alpha_t(s,a) \in [0,1]$, $\sum_t \alpha_t(s,a) = \infty$, $\sum_t (\alpha_t(s,a))^2 < \infty$ w.p.1 and $\alpha_t(s,a) = 0$ unless $(s,a) = (s_t, a_t)$ and $s_{t+1} \notin NTM(s_t, a_t)$.*
3. *$Var\{R(s,a,s')\} < \infty$.*
4. *$\gamma < 1$.*

4.2.3 Experimental Results

As in BM-LVM, prioritized sweeping can be used to trade off computation time and performance in Algorithm 11, yielding a method we call BM-NTM. We compare its performance to BM-LVM, Q-learning, and a sparse model-based method that combines prioritized sweeping with an NTM without best-match updates, which we call PS-NTM. While BM-NTM uses the samples that are not part of the NTM to update Q^{mf}, PS-NTM ignores these samples. The priority of a state-action pair (s, a) for BM-NTM is defined as

$$p = (1 - w_0^{sa}) \hat{\mathcal{P}}_{sa}^{s_1} \cdot |\Delta V(s_1)|\,,$$

Algorithm 11 General Best-Match NTM Control

initialize $Q(s,a) = Q^{mf}(s,a)$ arbitrarily for all s,a
initialize $N_{sa}, N^x_{sa}, R^{sum}_{sa}$ to 0 for all s,a
initialize $N^x_{sas'}$ to 0 for all s,a and $s' \in NTM(s,a)$
initialize w_0^{sa} to 1 for all s,a
loop {over episodes}
  initialize s
  repeat {for each step in the episode}
    select action a, based on $Q(s,\cdot)$
    take action a, observe r and s'
    if $s' \in NTM(s,a)$ **then**
      $N^x_{sa} = N^x_{sa} + 1; \quad N^x_{sas'} = N^x_{sas'} + 1; \quad R^{sum}_{sa} = R^{sum}_{sa} + r$
      $\hat{\mathcal{P}}^{s'}_{sa} = N^x_{sas'}/N^x_{sa}; \quad \hat{\mathcal{R}}_{sa} = R^{sum}_{sa}/N^x_{sa}$
    else
      $Q^{mf}(s,a) \leftarrow (1-\alpha^{sa})Q^{mf}(s,a) + \alpha^{sa}\left[r + \gamma \max_c Q(s',c)\right]$
    $N_{sa} = N_{sa} + 1$
    $w_0^{sa} = 1 - N^x_{sa}/N_{sa}$
    repeat
      select some $(\bar{s},\bar{a})$ pair with $N_{\bar{s}\bar{a}} > 0$ {each pair is selected at least once before its revisit}
      $Q(\bar{s},\bar{a}) \leftarrow w_0^{\bar{s}\bar{a}} Q^{mf}(\bar{s},\bar{a}) + (1 - w_0^{\bar{s}\bar{a}}) \left[\hat{\mathcal{R}}_{\bar{s}\bar{a}} + \gamma \sum_{s'} \hat{\mathcal{P}}^{s'}_{\bar{s}\bar{a}} \max_c Q(s',c)\right]$
    until some stopping criterion has been met
    $s \leftarrow s'$
  until s is terminal

where $\Delta V(s_1)$ is the difference in the state value of s_1 before and after the best-match update of one of the Q-values of s_1. For PS-NTM, the priority is defined similarly:

$$p = \hat{\mathcal{P}}^{s_1}_{sa} \cdot |\Delta V(s_1)| \,.$$

The NTM we use for BM-NTM and PS-NTM is defined by the first n unique next states that are encountered for a specific state-action pair. Although more sophisticated models could be used (e.g., by estimating the n most likely transition states), this model is sufficient for our experimental setting since most transition states have similar transition probabilities.

We consider the large maze task shown at the left in Figure 4.8. For this maze, the reward received by the agent is -0.1 at each timestep, while reaching the goal state results in a reward of +100. The discount factor is 0.99. The agent can take four actions, 'north','south','east' and 'west'. The action outcomes are made very stochastic, in order to compare different model sizes. The right side of Figure 4.8 shows the relative action outcome for a 'north' action. In free space, there are 15 possible next states, each with equal transition probability. On the other hand, walls prevent not only the transition to the square the wall is located on, but also any squares behind the wall. Therefore, close to a wall the number of possible next states is less than 15. When transition to a square is

blocked by a wall, the transition probability of that square is added to the transition probability of the square in front of the wall. In order to make reaching the goal feasible despite the stochastic actions, we use a goal area consisting of four goal states.

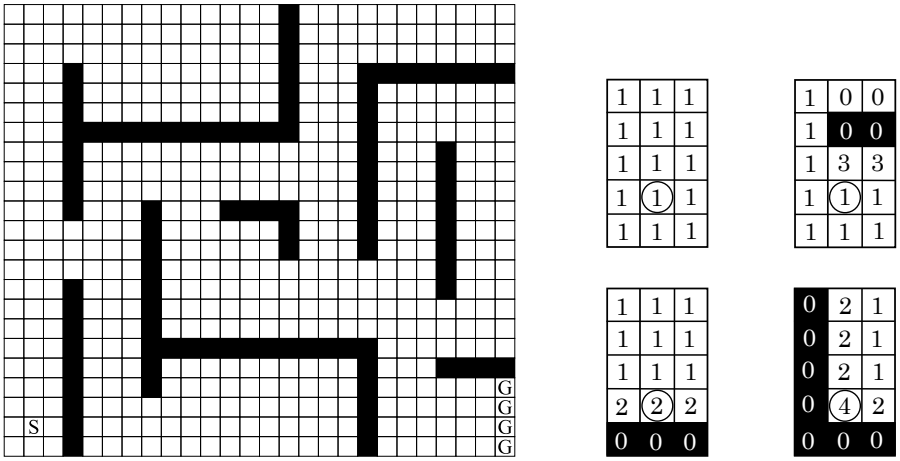

Figure 4.8: Left, the large maze task, in which the agent must travel from S to one of the G's. Right, transition probabilities ($\cdot\frac{1}{15}$) of a 'north' action for different positions of the agent (indicated by the circle) with respect to the walls (black squares). When the transition to a square is blocked by a wall, its transition probability is added to that of the square in front of the wall.

To compare performance, we measure the average return for each method over the first 500 episodes. For all methods, we use an ε-greedy policy with $\varepsilon = 0.05$ and initialize Q-values to 0. BM-NTM, PS-NTM and BM-LVM perform a maximum of 5 updates per timestep. For all learning rate based methods, we use an initial learning rate of 1 and decay the learning rate according to Equation 3.13, while optimizing the decay rate d. Results are averaged over 200 independent runs. An episode is stopped prematurely if the goal is not reached within 500 steps.

Table 4.3 presents the results, including the average return, optimal parameters, and computation time per simulation step. The model sizes used are N = 1, 3, 5, and 15. For $N = 15$, all samples enter the model. Therefore, BM-NTM has no decay rate in this case. The model weight indicates the fraction of samples that entered the model. BM-NTM has in general a slightly higher weight than PS-NTM, indicating the agent spends less time in open spaces and more time close to a wall.

For model sizes $N = 1$ and $N = 3$, the average return of BM-NTM is much better than that of PS-NTM, despite the fact that for $N = 3$ more than a third of the samples are stored in the model. For $N = 1$, the average return of PS-NTM is even worse than that of Q-learning. Figure 4.9 shows the return as a function of the number of episodes for BM-NTM and PS-NTM with $N = 1$ and $N = 3$. Unlike BM-NTM, the asymptotic performance for PS-NTM is clearly bounded by the size of the model. Thus, PS-NTM can match the performance of BM-NTM only when the space reduction over the full model is quite small (i.e., less than a factor of 2).

Interestingly, when $N = 1$, BM-LVM outperforms BM-NTM despite having the same

	model size	model weight	optimal parameters	average return	standard error	time per step ($\cdot 10^{-6}$ s)
PS-NTM	1	0.12	-	-16.9	0.4	0.21
	3	0.36	-	9.8	0.3	1.5
	5	0.57	-	22.6	0.2	2.1
	15	1.00	-	28.9	0.2	3.1
BM-NTM	1	0.14	d = 0.04	15.4	0.3	0.85
	3	0.40	d = 0.09	19.6	0.2	1.7
	5	0.60	d = 0.06	22.3	0.2	2.2
	15	1.00	-	29.3	0.2	3.1
BM-LVM	-	-	d = 0.09	17.4	0.3	1.5
Q-learning	-	-	d = 0.03	2.4	0.2	0.09

Table 4.3: Average return over the first 500 episodes, optimal parameters (d: α decay rate) and computation time per simulation step on the Large Maze task.

space complexity. Thus, when space is scarce, BM-LVM is a good option. In contrast, BM-NTM can exploit larger models to further improve performance. The computation time per simulation step for BM-NTM is comparable to that of PS-NTM, with the exception of $N = 1$, for which it is four times larger. The reason is that the priority queue of PS-NTM is often close to empty in this case and thus the 5 updates per timestep are often not reached.

Overall, these results clearly demonstrate the strength of best-match NTM learning. When a significant space reduction over storing the full model is required, BM-NTM performs dramatically better than PS-NTM at similar computational cost.

4.3 Best-Match Function Approximation

The BM-NTM method described in the previous section has a space complexity of $\mathcal{O}(n|\mathcal{S}||\mathcal{A}|)$ compared to $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ for full model-based methods. However, in problems with large state spaces, this space complexity may be prohibitive even when $n = 1$. In addition, BM-NTM cannot be applied in problems with continuous state spaces. To address these limitations, this section demonstrates that the principles behind best-match learning can also be applied to function approximation. We show that the resulting algorithm, which combines the N most recent samples with the model-free Q-value function, outperforms both linear Sarsa(λ) and linear experience replay on the mountain car task. We start by describing best-match learning based on the N most recent samples for the tabular case, and then we show how this can be extended to the function approximation case.

4.3.1 Tabular Sequence Based Best-Match Learning

The generalized best-match equations are defined for an arbitrary set of samples (see Definition 6), which can be stored in a model or as an explicit set. To combine best-match principles with function approximation, we employ an explicit set consisting of the last N observed samples, an approach we call *sequence based best-match learning*. In this section

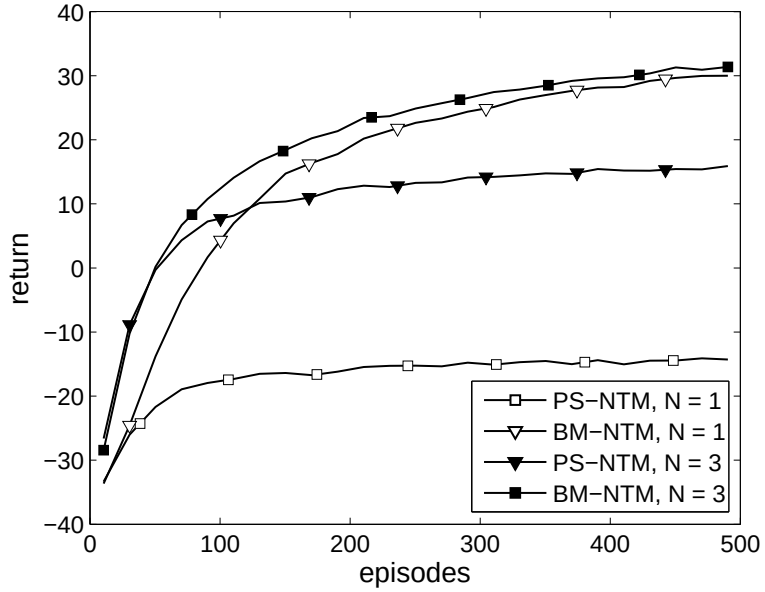

Figure 4.9: Performance of BM-NTM and PS-NTM on the large maze task.

we describe sequence based best-match learning for the tabular case and its advantage over experience replay, which also exploits a set of recent samples. In the next section, we extend the tabular version of sequence based best-match learning to function approximation.

Assume that a queue of the last N samples is maintained. When the queue is full and a new sample is added to the back of the queue, the sample at the front of the queue is removed and used to perform a model-free update of $Q^{mf}(s,a)$. The queue may contain multiple samples that belong to the same state-action pair. If there are N_{sa}^x samples belonging to state-action pair (s,a), then the best-match update based on these samples is

$$Q_{t,i+1}(s,a) = w_0^{sa}\, Q_t^{mf}(s,a) + w_1^{sa} \upsilon_1^{sa} + w_2^{sa} \upsilon_2^{sa} + ... + w_{N_{sa}^x}^{sa} \upsilon_{N_{sa}^x}\,, \tag{4.10}$$

where $\upsilon_k^{sa} = r + \gamma \max_c Q_{t,i}(s',c) \;|r, s' \in x_k^{sa}$. When the weights are defined according to Equations 4.8 and 4.9, this update can be implemented incrementally by performing N_{sa}^x Q-learning updates:

$$Q_{<k>}(s,a) = (1-\alpha) Q_{<k-1>}(s,a) + \alpha \left[r_k + \gamma \max_{a'} Q_{t,i}(s'_k, a') \right], \quad \text{for } 1 \le k \le N_{sa}^x\,,$$

with $Q_{<0>}(s,a) = Q_t^{mf}(s,a)$ and $Q_{t,i+1}(s,a) = Q_{<N_{sa}^x>}(s,a)$.

By stepping through the queue from front to back and using each sample to perform an incremental Q-learning update, all state-action pairs with samples in the queue receive one full best-match update, according to Equation 4.10. By storing the intermediate $Q_{<k>}$ values at the same location as the final Q-value, $Q_{<N_{sa}^x>}$ automatically becomes $Q_{t,i+1}$ after all incremental updates have been performed. This implementation requires that the Q-values from the state-action pairs with samples in the queue are set equal to $Q_{<0>}$, that is, to Q_t^{mf}, before the update sweep begins. Before resetting these Q-values, the update targets of the samples must be recomputed.

Despite a superficial resemblance, sequence based best-match learning is fundamentally different from experience replay. Best-match learning uses the stored samples to correct previous updates based on those samples, whereas experience replay performs additional updates with the same sample. To illustrate the effect of this difference, suppose that sample (s, a, r, s') is observed at timestep $t = 1$ and used for an update n timesteps in a row. For simplicity, assume there are no other samples belonging to (s, a) in the sample queue and that the learning rate α is constant. We indicate the update target of the sample with $\bar{v}_i$, where i corresponds to the timestep at which the update is performed. Therefore, $\bar{v}_{i+1}$ is likely to be more accurate than $\bar{v}_i$ since it uses more recent Q-values for s'. Since experience replay performs additional updates we can express $Q_n(s, a)$, the Q-value of (s, a) at timestep n, in terms of $Q_0(s, a)$ and the update targets from the different timesteps as follows:

$$Q_n(s, a) = w_0\, Q_0(s, a) + w_1\bar{v}_1 + w_2\bar{v}_2 + ... + w_n\bar{v}_n\,,$$

with $w_0 = \prod_{i=1}^{n}(1-\alpha)$ and $w_k = \alpha\prod_{i=k+1}^{n}(1-\alpha)$ for $k > 0$. If $\alpha \ll 1$, the weights can be accurately described with first-order approximations in α, yielding $w_0 \approx 1 - n\alpha$ and $w_k \approx \alpha$ for $k > 0$. Using these approximations, we can write for $Q_n(s, a)$:

$$Q_n(s, a) \approx (1-\beta)Q_0(s, a) + \beta\,\frac{\sum_{i=1}^{n}\bar{v}_i}{n}\,, \tag{4.11}$$

with $\beta = n\alpha$. On the other hand, best-match learning uses the sample for best-match updates, that is, $Q_n(s, a) = (1-\alpha)Q_n^{mf}(s, a) + \alpha\,\bar{v}_n$. However, since $Q_i^{mf}(s, a)$ gets updated only when a sample is removed from the queue, $Q_n^{mf}(s, a) = Q_0(s, a)$ in this case. Therefore, the following holds for best-match learning:

$$Q_n(s, a) = (1-\alpha)Q_0(s, a) + \alpha\,\bar{v}_n\,. \tag{4.12}$$

The difference between Equation 4.11 and Equation 4.12 illustrates the fundamental advantage of sequence based best-match learning, for which Q_n can be seen as an update with sample (s, a, r, s') using the most recent update target. In contrast, experience replay effectively performs an update using an update target that is an average of the update targets from the different timesteps. Therefore, the older, less accurate update targets still have an effect on Q_n.

4.3.2 Best-Match Gradient Descent Learning

Since tabular sequence based best-match learning can be implemented by incremental Q-learning updates, it can be easily extended to function approximation by combining it with the general gradient descent update for Q-values (Sutton and Barto, 1998)

$$\theta_{t+1} = \theta_t + \alpha\,[v_t - Q_t(s_t, a_t)]\,\nabla_{\theta_t}Q_t(s_t, a_t)\,, \tag{4.13}$$

where θ_t is a weight vector corresponding to the basis functions of the approximation.

Algorithm 12 shows pseudocode for general gradient descent best-match function approximation. Note that a learning rate and the most recent update target are stored per sample. The updates of θ and θ^{mf} are based on Equation 4.13.

Algorithm 12 General Gradient-Descent Best-Match

set N, γ
initialize θ, α and set $\theta^{mf} = \theta$
initialize $SampleQueue$ to empty
loop {over episodes}
  initialize s
  while $s \neq$ terminal state **do**
    select action a, based on θ
    take action a, observe s',r
    if size $SampleQueue = N$ **then**
      pop sample x from front of the $SampleQueue$
      update θ^{mf} using x
    decay α; $\upsilon = \emptyset$
    push new sample $\{s, a, r, s', \alpha, \upsilon\}$ to back of $SampleQueue$
    for all samples x update $\upsilon_x \leftarrow r_x + \gamma \cdot V_{s'_x}$ using θ
    for all samples x **do**
      for all features from x: $\theta \leftarrow \theta^{mf}$
    for all samples x (from front to back of $SampleQueue$) **do**
      update θ using υ_x
    $s \leftarrow s'$

We evaluate a linear version of the best-match gradient descent algorithm by comparing its performance with linear Sarsa(λ) as well as a linear version of experience replay on the mountain car task (Boyan and Moore, 1995; Sutton, 1996; Sutton and Barto, 1998) using the settings as described by Sutton and Barto (1998). This involves tile coding with ten 9x9 tilings, a discount factor of 1, an exploration parameter $\varepsilon = 0$, and Q-values optimistically initialized to 0. Additionally, to bound the run-time of an experiment, an episode is stopped prematurely if the goal is not reached within 1000 steps. Linear Sarsa(λ) is known for its good performance on this task (Sutton and Barto, 1998) and is therefore a good benchmark test. For Sarsa(λ), we use the settings that showed the best performance over the first 20 episodes: $\alpha = 0.14$ and $\lambda = 0.9$ with replacing traces. We tested whether decaying the learning rate improves the performance for a number of different α values around 0.14 but did not find a significant improvement. To make Sarsa(λ) more computationally efficient, traces are cut-off for state-action pairs that were visited longer than 20 timesteps ago. For best-match and experience replay, a queue of the 20 most recent samples is used and a single update sweep through this sample set is performed at every timestep. We optimize the initial learning rate α_0 and the learning rate decay d (see Equation 3.13). Results are averaged over 5000 independent runs.

Table 4.4 shows the average return over the first 20 episodes, the optimal parameters, and the computation time per simulation step for the 5000 runs. Figure 4.10 shows the return as a function of the number of episodes. For trace length/N = 20, the performance of linear best-match is about 27% better than that of linear Sarsa(λ).[5] On the other hand,

[5]The linear Sarsa(λ) performance is in accordance with the performance found by several other researchers

Sarsa(λ) is about twice as fast.

	optimal parameters	average return	standard error	time per step $(\cdot 10^{-6} s)$
best-match, N=20	$\alpha_0 = 0.10$, d = 0.09	-170.1	0.4	3.0
exp. replay, N=20	$\alpha_0 = 0.10$, d = 0.16	-195.1	0.4	2.5
Sarsa(λ), trace=20	$\lambda = 0.9$, $\alpha_0 = 0.14$, d = 0.0	-231.9	0.4	1.5
best-match, N=15	$\alpha_0 = 0.10$, d = 0.03	-176.3	0.4	2.5
best-match, N= 5	$\alpha_0 = 0.10$, d = 0.03	-215.1	0.4	1.5
Sarsa(λ), trace=∞	$\lambda = 0.9$, $\alpha_0 = 0.14$, d = 0.0	-228.2	0.4	6.7

Table 4.4: Average performance over the first 20 episodes and the computation time per simulation step on the Mountain Car task ('trace' indicates trace length)

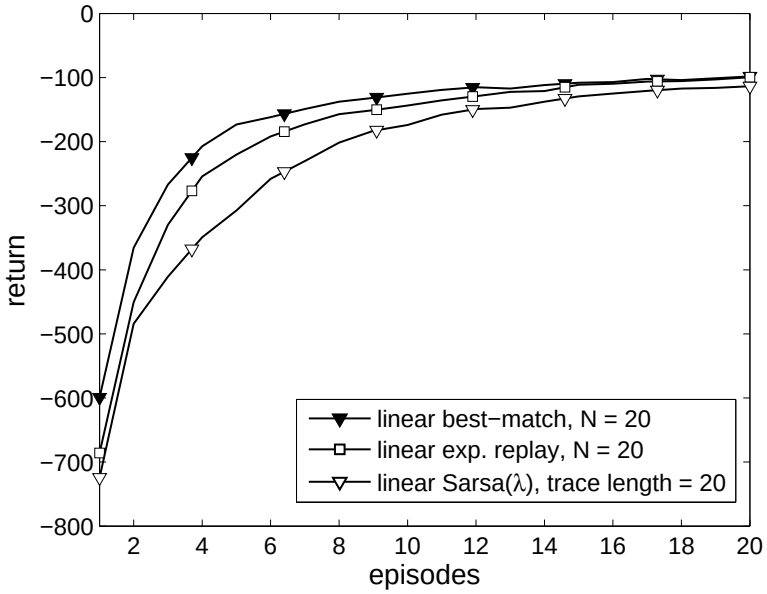

Figure 4.10: Performance of linear best-match, experience replay and linear Sarsa(λ) on the Mountain Car task using the 20 most recent samples.

Surprisingly, while experience replay performed comparably to Sarsa(λ) in the tabular case, in the mountain car task it performs 16% better than linear Sarsa(λ). However, as expected, it performs worse than linear best-match. Thus, a substantial portion of the performance improvement linear best-match offers over Sarsa(λ) is due to the use of best-match principles, not simply the reuse of data.

Besides a comparison with equal number of samples/updates, it is interesting to make a comparison with equal computation time. To achieve this, we can either increase the

(`http://webdocs.cs.ualberta.ca/~sutton/book/errata.html`).

sample set size used by experience replay and Sarsa(λ), or decrease the sample set size used by linear best-match, in such a way that the computation times approximately match. We chose to decrease the sample set size of linear best-match. Using $N = 15$ and $N = 5$ resulted in a computation time matching that of experience replay and Sarsa(λ), respectively. Table 4.4 shows that the performance of linear best-match is also better with equal amount of computation time. In addition, we performed an experiment with Sarsa(λ) without bound on the trace length. This resulted in an average return of -228.2, demonstrating that the performance of Sarsa(λ) cannot be improved significantly by increasing the trace length.

Overall, these results show that best-match learning can be successfully applied to function approximation. Furthermore, they demonstrate that using samples to correct previous updates can lead to better performance that using them to perform additional updates.

4.4 Discussion

The methods presented in this chapter approximate solutions to different instantiations of the generalized best-match equations (Definition 6). These best-match equations provide a theoretical foundation for combining model-free learning (through updates of Q^{mf}) with model-based learning (through updates of Q). The resulting methods offer two trade-offs. First, the selection of a sparse, approximate model provides a trade-off between space and performance. Second, the number of best-match updates performed per timestep provides a trade-off between computation cost per timestep and performance. The performance gain offered by best-match learning can be explained from the perspective of the update targets. By performing best-match updates, the update targets from the samples stored in the model are continually recomputed and the Q-values are updated to incorporate any resulting changes.

In the case of best-match LVM, this produces an evaluation method that leads to the same values as TD(λ) with $\lambda_t = \alpha_t(s_t)$ for acyclic tasks, as proven in Theorem 3. This equivalence arises from the fact that both best-match LVM learning and eligibility traces outperform 1-step methods by correcting previous updates with newly obtained samples. However, our theoretical and empirical results suggest that the best-match LVM equations provide a much stronger basis for exploiting this principle.

Theorem 4 proves that best-match LVM evaluation can perform updates that are unbiased with respect to the initial values for an arbitrary MDP, while for TD(λ) this can only be achieved for acyclic tasks. In the control case, Theorem 5 proves convergence in the limit to the optimal Q-values for a general class of best-match LVM control algorithms. Similar converge guarantees do not exist for eligibility traces. In addition, best-match LVM learning avoids the need to choose between different trace types (accumulating or replacing) and does not require an extra λ parameter. Furthermore, in deterministic problems, best-match LVM learning, reduces to model-based learning, as one would expect for an algorithm that makes optimal use of the $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ space complexity.

Our empirical results show that best-match LVM evaluation substantially outperforms TD(λ) and experience replay (Figure 4.5), despite having similar computational costs. For

the control case, we show that BM-LVM, which uses prioritized sweeping to trade-off computation cost with performance, substantially outperforms not only Q(λ), but also other methods with a space complexity of $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ (Figure 4.6). These results illustrate how best-match LVM learning efficiently exploits its stored samples.

Alternatively, best-match learning can be combined with an n-transition model, yielding space complexity between $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$. Without using best-match learning, the performance of an NTM is bounded by the quality of the model approximation. In contrast, Theorem 6 proves that BM-NTM converges in the limit to the optimal Q-values. Empirically, we demonstrate that, for any significant space reduction compared to the full model, BM-NTM performs much better than using only the NTM (Figure 4.9).

Finally, our results demonstrate that the ideas behind best-match learning can be successfully extended to function approximation by combining sequence based best-match learning with gradient descent updates (Algorithm 12). In particular, a linear implementation outperforms Sarsa(λ) and experience replay on a benchmark task (Figure 4.10).

4.5 Future Work

Several avenues of future research are suggested by the work presented in this chapter. For example, in Section 4.1.2 we proved that the best-match LVM evaluation algorithm can eliminate bias with respect to the initial values. It may be possible to extend this result to the control case. One approach would be to define a state value as the maximum of the Q-values over previously taken actions instead of the maximum over all available actions. However, a potential problem is that the control algorithms compute an approximation of the best-match Q-values, instead of the exact values. It is an open question whether efficient approximations exist that are also unbiased. A second potential problem is that many exploration schemes, such as optimistic initialization, depend on the Q-values and might not work as well when updates are unbiased.

The convergence results for the tabular best-match methods are similar to those of Q-learning: convergence in the limit to the optimal policy. It may be possible, however, to construct best-match methods that are probably approximately correct (PAC). Since Strehl et al. (2006) showed that a full model is not required for a method to be PAC, we are optimistic that such methods exist.

Finally, it may be possible to develop novel combinations of best-match function approximation with other sample-based approaches such as fitted Q-iteration (Ernst et al., 2005) or LSPI (Lagoudakis and Parr, 2003). By combining the strengths of each approach, such methods could yield even better on-line performance. Fitted Q-iteration, for example, is an off-line algorithm that computes a policy based on a large set of samples, by performing iterative update sweeps through the sample set. For a good approximation, the number of samples should be much larger than the number of parameters of the approximation. By using a combination between a model-free Q-value function and a sample set, a smaller sample set might be possible, reducing the requirements with respect to space and computation, and potentially producing an efficient on-line version of fitted Q-iteration.

4.6 Conclusion

This chapter introduced best-match learning, a reinforcement learning approach that combines model-free and model-based learning by using some samples to update a sparse model and the rest to update a model-free Q-value. The final Q-values are computed from best-match updates that combine the model-free Q-values with the sparse model. By controlling which samples enter the model, the size of the model, and hence the space requirements, can be controlled. In the tabular case, the combination with the model-free Q-values ensures convergence to the optimal Q-values for a variety of model approximations.

Our empirical results demonstrate that in the tabular case, when there is not enough space available to store the full model, methods that exploit the best-match equations perform substantially better than methods based on only model-free learning or sparse model-based methods. This suggests that best-match learning should be the strategy of choice when limited space is available.

In addition, we demonstrated that best-match learning can be successfully extended to the function approximation domain, where the sparse model is replaced by an explicit set of samples. An interesting result in this domain is that best-match learning, which uses the sample set to correct previous updates, outperforms experience replay, which uses the same sample set but performs additional updates.

Overall, we believe that best-match learning provides an important missing link between model-free and model-based learning and that the methods introduced in this chapter constitute a new benchmark for reinforcement learning algorithms that are efficient with respect to both space and computation.

CHAPTER 5

Reducing the Problem Size by Representation Selection

Learning in MDPs is challenging because of the *curse of dimensionality*: the size of the state space grows exponentially with respect to the number of problem parameters. Consequently, finding a good policy can require prohibitive amounts of memory, computation time, and/or sample experience (i.e., interactions with the environment). Fortunately, many real-world problems have internal structure that can be exploited to dramatically speed learning.

In a *factored MDP* (Boutilier et al., 1995), wherein each state is described by a set of state feature values, independence between such features can be expressed using *dynamic Bayesian networks* (DBNs) (Dean and Kanazawa, 1989). In *planning* problems, i.e., when the MDP is known in advance, DBNs enable efficient solution methods that do not require explicit enumeration of the state space (Boutilier et al., 1995). Similarly, in learning problems, DBNs enable near-optimal performance using only samples and computation polynomial in the number of parameters of the DBN, which may be exponentially smaller than the number of states (Kearns and Koller, 1999). However, achieving this performance requires as input a complete description of the DBN's structure (but not its parameter values).

Unfortunately, in many real-world problems, the DBN structure is not known in advance and must also be learned. Doing so is also possible in a sample-efficient way, given prior knowledge of the maximum degree of the DBN (Li et al., 2008; Diuk et al., 2009; Kroon and Whiteson, 2009). However, the memory and computation requirements for such methods is linear in the number of states, making them impractical for large problems.

In this chapter, we propose an alternative approach for exploiting structure in MDPs. Rather than learning the structure and parameter values of a DBN, our approach learns which representation among a set of *candidate representations* yields the highest expected return. Each candidate representation consists of a subset of the available state features. In general, the number of candidate representations can be prohibitively large. However, in many real-world settings, prior knowledge about the task can be used to deduce a small set of candidate representations.

For example, consider a predator-prey scenario in which the prey must make optimal use of its sensors to quickly detect and evade predators. It can, e.g., choose to scan the sky for flying predators, or it can focus on nearby trees to determine if predators are hiding behind them. These different strategies involve relying on different sensors (or sensor settings) and thus different candidate representations. Given human expertise or previous experience on similar tasks, it may be easy to deduce what candidate representations are

worth trying, whereas specifying the structure or even the maximum degree of the corresponding DBN would be infeasible.

In this chapter, we demonstrate that an MDP in which the agent can choose between a set of candidate representation can be transformed into a *derived task* containing a single representation and internal *switch actions* that select which candidate representation to use for external action selection. In particular, we prove that under certain conditions this derived task forms an MDP whose solution yields both the optimal representation for the original MDP and the optimal policy under that representation.

We show that, because the derived task obeys the Markov property, it can be solved with standard RL methods. The computation time and memory required for doing so depends on the size of the derived MDP's state space, which can be exponentially smaller than that of the original MDP. However, we also demonstrate how the unique structure of the derived task can be exploited to further speed learning. In particular, the agent can construct *parallel experience sequences* that allow it to simultaneously learn about multiple candidate representations.

The remainder of this chapter is organized as follows. In Section 5.1 we define different types of features and representations. In Sections 5.2, 5.3, and 5.4 we propose and evaluate representation-selection methods for contextual bandit problems, MDPs, and MDPs with context-specific candidate representations, respectively. Section 5.5 discusses the empirical results, Section 5.6 reviews related work, and Section 5.7 concludes.

5.1 Representations

Using the full feature set to describe the environment can lead to prohibitively large state spaces. To circumvent this, the agent can choose to ignore certain features, e.g., those that it knows are irrelevant. By ignoring features, the agent effectively interacts with a different task that has a smaller state space. Depending on the features that are ignored, this task may also obey the Markov property, i.e., form an MDP by itself, in which case standard RL methods can be used to solve it.

We refer to the set of state features used by the agent as its *representation*. When a representation results in a task that obeys the Markov property, we call the representation *valid*. In this section we discuss how valid representations can be constructed by removing certain feature types from the full feature set. But we start with the definition of a factored MDP.

5.1.1 Factored MDPs

In a factored MDP (Boutilier et al., 1995), each state is described using a set of state variables or *features*: $\mathbf{X} = \{X_1, ..., X_N\}$ where each X_i takes on values in some domain $Dom(X_i)$. A state $\mathbf{x}$ defines a value $x_i \in Dom(X_i)$ for each variable X_i. Unless specified otherwise, we use upper case letters (e.g., X) to denote random variables, and lower case (e.g., x) to denote their values. We use boldface to denote vectors of variables (e.g., $\mathbf{X}$) or their values (e.g., $\mathbf{x}$). The domain of a vector of variables, $Dom(\mathbf{X})$, is the set of all value

assignments of **x** that have a probablity > 0. For the domain size of $\mathbf{X} = \{X_1, ..., X_N\}$ the following holds:

$$|Dom(\mathbf{X})| \leq \prod_{i=1}^{N} |Dom(X_i)|$$

The domain size of a vector of variables can be smaller than the product of its variable sizes when variables are correlated. For example, for two identical features X_1 and X_2 the domain size relation is $|Dom(\{X_1, X_2\})| = |Dom(X_1)| = |Dom(X_2)|$. For an instantiation $\mathbf{y} \in Dom(\mathbf{Y})$ and a subset of these variables $\mathbf{Z} \subset \mathbf{Y}$ we use $\mathbf{y}[\mathbf{Z}]$ to denote the value of the variables $\mathbf{Z}$ in the instantiation $\mathbf{y}$.

5.1.2 Feature Types

In this section, we define several feature types, where a feature is an element of the total feature set $\mathbf{X} = \{X_1, ...X_N\}$, belonging to some MDP. We use the DBN shown in Figure 5.1 as a running example to illustrate feature types.

Definition 7. *A feature $X_i \in \mathbf{X}$ is irrelevant with respect to $\mathbf{Y}$ if the following holds for all $\mathbf{x}_{t+1}, r_{t+1}, \mathbf{x}_t$ and a_t:*

$$P(\mathbf{y}^-_{t+1}, r_{t+1} | \mathbf{y}^-_t, a_t) = P(\mathbf{y}^-_{t+1}, r_{t+1} | \mathbf{y}^+_t, a_t) \tag{5.1}$$

with

$$\begin{aligned} \mathbf{y}^+_t &= \mathbf{x}_t[\mathbf{Y} \cup X_i] \\ \mathbf{y}^-_t &= \mathbf{x}_t[\mathbf{Y} \setminus X_i] \end{aligned}$$

Informally, an irrelevant feature is a feature whose value affects neither the next value of any feature from $\mathbf{Y}$ (except potentially its own value), nor the reward received. In Figure 5.1, X_1 is irrelevant with respect to $\{X_1, X_3, X_4\}$ because it affects neither those features nor reward. Similarly, X_3 is irrelevant with respect to $\mathbf{X}$ because it affects only itself. The complement class is the class of relevant features:

Definition 8. *A feature $X_i \in \mathbf{X}$ is relevant with respect to $\mathbf{Y}$ if it is not irrelevant with respect to $\mathbf{Y}$.*

In Figure 5.1, X_1 and X_2 are relevant with respect to $\mathbf{X}$ because they affect features other than themselves. Furthermore, X_4 is relevant with respect to each $\mathbf{Y} \subseteq \mathbf{X}$ because it affects reward.

We can divide the irrelevant feature class into three subclasses: constant, empty and redundant features.

Definition 9. *A constant feature $X_i \in \mathbf{X}$ is a feature with $|Dom(X_i)| = 1$.*

A constant feature is a feature that never changes value. It is therefore irrelevant w.r.t. all subsets $\mathbf{Y} \subseteq \mathbf{X}$. Note that features that stay constant during an episode, but change values between episodes are not constant. We exclude such features from the definition because they can still be relevant.

An irrelevant feature can also be empty:

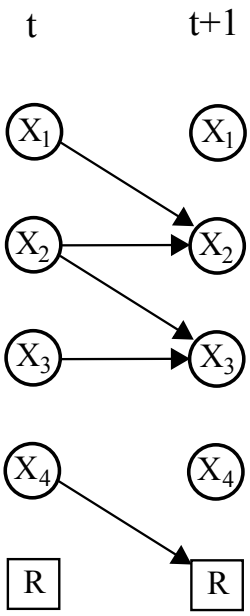

Figure 5.1: A DBN of the transition dynamics of an MDP with features $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$, illustrating various feature types. Circles correspond to state features, squares to reward, and arrows to probabilistic dependencies. The left column and right columns correspond to timesteps t and $t+1$ respectively.

Definition 10. *An* empty *feature $X_i \in \mathbf{X}$ is a non-constant feature that is irrelevant with respect to all subsets $\mathbf{Y} \subseteq \mathbf{X}$.*

In Figure 5.1, X_3 is an example of either a constant or an empty feature (depending on whether the value stays constant or not). Finally, an irrelevant feature can be redundant:

Definition 11. *A feature $X_i \in \mathbf{X}$ is redundant with respect to $\mathbf{Y}$ if it irrelevant with respect to $\mathbf{Y}$ and non-empty.*

A redundant feature X_i either affects the feature value of a feature not part of $\mathbf{Y}$ or affects a feature from $\mathbf{Y}$, but X_i is fully correlated with some other feature(s) from $\mathbf{Y}$, hence removing feature X_i from $\mathbf{Y}$ does not affect the predictions of the next state. In this last case, by removing some feature set $\mathbf{Z}$ from $\mathbf{Y}$, feature X_i could become a relevant feature with respect to $\mathbf{Y} \setminus \mathbf{Z}$.

Apart from the relevant/irrelevant classification, we define another classification: dependent and independent features.

Definition 12. *A feature $X_i \in \mathbf{X}$ is independent if for all $\mathbf{x}_{t+1}$, $\mathbf{x}_t$, r_{t+1} and a_t the following holds:*

$$P(x^i_{t+1}) = P(x^i_{t+1} | r_{t+1}, \mathbf{x}_t, a_t) \tag{5.2}$$

with

$$x^i_{t+1} \quad = \quad \mathbf{x}_{t+1}[\,X_i]$$

Thus, the value of an independent feature does not depend on the previous state features or the reward just received. Note that an independent feature *can* affect the next value of other features or the next reward. In Figure 5.1, X_1 and X_4 are independent because no variables affect them.

As we prove in the next subsection, an independent feature is unique in the sense that it can contain relevant information, but omitting it still give a Markov representation. Therefore, normal reinforcement learning methods still converge when using such a representation, though the resulting policy is not optimal in $\mathbf{X}$. However, since we are primarily interested in the best online performance instead of the optimal policy, omitting independent features can play an important role in finding an efficient representation.

For completeness, we also define the counterpart of an independent feature:

Definition 13. *A feature $X_i \in \mathbf{X}$ is* dependent *if it is not independent.*

In Figure 5.1, X_2 and X_3 are dependent because their values at timestep $t+1$ depend on the value of variables at timestep t.

5.1.3 Valid Representations

While the full feature set yields a Markov task, using a subset of features will not always produce a Markov task. We call a representation *valid* if it does yield a Markov task, in which case standard methods can be used to solve it.

Definition 14. *Consider the MDP $M = \langle \mathbf{X}, A, T, R \rangle$. A subset of features $\mathbf{Y} \subseteq \mathbf{X}$ is a valid representation if the Markov property applies to it, i.e., if the following condition holds for all $\mathbf{x}_{t+1}, r_{t+1}$ and all possible histories $\mathbf{x}_t, a_t, r_t, ..., r_1, \mathbf{x}_0, a_0$:*

$$P(\mathbf{y}_{t+1}, r_{t+1} | \mathbf{y}_t, a_t) = P(\mathbf{y}_{t+1}, r_{t+1} | \mathbf{y}_t, a_t, r_t, \mathbf{y}_{t-1}, a_{t-1}, ..., r_1, \mathbf{y}_0, a_0) \tag{5.3}$$

with $\mathbf{y}_t = \mathbf{x}_t[\mathbf{Y}]$.

The following theorem shows how a valid representation $\mathbf{Y}$ can be constructed from the full feature set $\mathbf{X}$. We prove this theorem in Appendix A.

Theorem 7. *Consider the MDP $M = \langle \mathbf{X}, A, T, R \rangle$. A subset of features $\mathbf{Y} \subseteq \mathbf{X}$ is a valid representation if for all $X_i \in \mathbf{X}$ the following holds:*

$$\textit{if } X_i \notin \mathbf{Y} \textit{ then } X_i \textit{ is irrelevant w.r.t. } \mathbf{Y} \textit{ or } X_i \textit{ is an independent feature.} \tag{5.4}$$

5.1.4 Context-Specific Representations

Sometimes the structure of a problem dictates that a feature is irrelevant depending on the value of other features. To exploit this type of structure, an agent can employ a *context-specific* representation (Boutilier et al., 1995; Zhang and Poole, 1999; Guestrin et al., 2003) in which different features are used to describe different states.

We define a context-specific representation as a mapping H_{cs} that maps each state $\mathbf{x} \in Dom(\mathbf{X})$ to the subset of features (i.e., to an element of the powerset of $\mathbf{X}$) used to represent that state:

$$H_{cs} : Dom(\mathbf{X}) \rightarrow \mathcal{P}(\mathbf{X})$$

The notion of validity can also be extended to context-specific representations, as shown in the following definition.

Definition 15. *The context-specific representation H_{cs} is a valid representation for MDP $M = \langle \mathbf{X}, A, T, \mathcal{R} \rangle$ if the following condition holds for all $\mathbf{x}_{t+1}, r_{t+1}$ and all possible histories $\mathbf{x}_t, a_t, r_t, ..., r_1, \mathbf{x}_0, a_0$:*

$$P(\mathbf{y}_{t+1}, r_{t+1} | \mathbf{y}_t, a_t) = P(\mathbf{y}_{t+1}, r_{t+1} | \mathbf{y}_t, a_t, r_t, \mathbf{y}_{t-1}, ..., r_1, \mathbf{y}_0, a_0) \tag{5.5}$$

where $\mathbf{y}_t = \mathbf{x}_t[\mathbf{Y}_t]$ and $\mathbf{Y}_t = H_{cs}(\mathbf{x}_t)$.

5.2 Representation Selection for Contextual Bandit Problems

In this section, we introduce and evaluate representation-selection algorithms for a *contextual bandit problem* (Wang et al., 2005; Pandey et al., 2007), a special case of an MDP that consists of only single-action episodes. Contextual bandit problems are a useful way to model many real-world tasks, e.g., selecting ads to place alongside web pages (Langford and Zhang, 2007; Langford et al., 2008). In Section 5.3, we extend our representation selection approach to general MDPs.

5.2.1 Contextual Bandit Problems

In a *multi-armed bandit problem* (Lai and Robbins, 1985; Auer et al., 2002), an agent faces a slot machine with multiple arms. Each arm has a distribution governing the stochastic reward received for pulling that arm. The goal of the agent is to maximize its reward over iterative pulls. This setting can be viewed as a special case of an MDP wherein the arms correspond to actions and $|\mathbf{X}| = 1$, i.e., there is only one state. Since the agent does not initially know the distribution over rewards for each arm, it faces a reinforcement learning problem in which the primary challenge is balancing exploration and exploitation.

The contextual bandit problem is an extension of the multi-armed bandit problem in which the reward distribution of the arms is correlated with context information observed by the agent. This setting can also be viewed as a special case of an MDP. As before, the arms correspond to actions. In addition, the context corresponds to the state, i.e., $|\mathbf{X}| > 1$. However, unlike in a general MDP, each action results in a terminal state, hence all episodes have length 1. Each new episode has a new context drawn according to some probability distribution. Thus, contextual bandit problems form a middle ground between multi-armed bandit problems and general MDPs. Like multi-armed bandit problems, the agent's actions affect only its immediate reward. However, like general MDPs, there are many possible states and the reward an action produces depends on that state.

5.2.2 Representation Selection

To see the potential benefits of representation selection, consider the ad-placement problem mentioned above. Since companies that serve ads are typically paid per click, the goal is to select the ads that maximize the chance of being clicked. This task can be modeled as a contextual bandit problem wherein available ads are actions, web pages are states, and rewards are payments for clicked ads.

Typically, the web page is described using a set of features. These can include the frequency of each term in the web page, a categorization of the page (e.g., news, entertainment, shopping), the number of incoming or outgoing links, the length of the URL, etc. Since the size of the state space depends critically on the number of features used, selecting a good representation is essential. The chosen subset of features must be rich enough to allow the system to determine what ad to place and yet be small enough to make learning feasible.

Suppose that ten features are available, each of which can take on five values. This yields $5^{10} \approx 10^7$ states. One option would be for an agent to try to learn an accurate action-value function for each state-action pair. However, doing so for such a large state space would be immensely challenging. Instead, the system designer could try to select the most useful features. For example, if three features are chosen, the size of the state space is only 125. However, due to the unpredictability of user behavior, selecting the right features would require enormous domain expertise.

In this section, we propose a method for automatically determining which representation to select for a contextual bandit problem. Our approach works by trying out the various candidate representations in the task, learning with them, and measuring the average reward accrued. To do so, we construct a *derived task* containing internal *switch actions* that correspond to selecting a candidate representation. Given some minimal prior knowledge about what representations to consider, this approach can efficiently find the best representation by solving the derived task. For example, given only the information that three features suffice to describe a web page, we can construct a derived task for the ad-placement contextual bandit problem that contains two orders of magnitude fewer states.[1]

In the following subsections, we formally describe the derived task, prove that it obeys the Markov property, provide a concrete example, and describe a model-free algorithm for learning in the derived task.

5.2.2.1 Derived Tasks

Consider the contextual bandit problem described by the factored state space $\mathbf{X}_{gr}$, action set A_{gr} and reward function $\mathcal{R}_{gr}$ and assume that the agent interacting with it can choose between K different representations: $\mathbf{X}^1, ..., \mathbf{X}^K$, where each $\mathbf{X}^k \subset \mathbf{X}_{gr}$. We call these the *candidate representations*.

Intuitively, our approach to such a task is to try out different representations over time and measure how well they perform. As the agent becomes more confident about which

[1] There are $\binom{20}{3} = 1140$ candidate representations with three features and 125 states per candidate representation and $1140 * 125 \approx 1.4 \times 10^5$.

representation is the best, it can use this representation more often, thus boosting its expected reward.

The main insight behind our method is that an agent trying out representations in a task faces an exploration/exploitation dilemma just like that of an agent choosing ordinary actions in such a task. As a result, the choice of which representation to use can be modeled as an action internal to the agent. We call these *switch actions*, to distinguish them from the ordinary actions in A_{gr}, i.e., the *ground actions*.

In the resulting *derived task*, the agent must select two actions in each timestep. First, it must choose a switch action, which selects a particular candidate representation. Then it must choose a ground action, based on the current Q-values of the selected candidate representation. The action set of the derived task, A_{dr}, consists of the action set of the original contextual bandit problem augmented with the switch action set $A_{sw} = \{a_1^{sw}, ...a_K^{sw}\}$. There is one switch action for each candidate representation.

$$A_{dr} = A_{sw} \cup A_{gr} \tag{5.6}$$

The feature set of the derived task contains, in addition to the features from the contextual bandit problem, one extra feature $X_{rep} = \{x_0^{rep}, ..., x_K^{rep}\}$ that specifies which candidate representation is selected. This feature has $K + 1$ values, one for each candidate representation plus one extra value, x_0^{rep}, that indicates no candidate representation is currently selected.

$$\mathbf{X}_{dr} = X_{rep} \cup \mathbf{X}_{gr} \tag{5.7}$$

The initial value of feature X_{rep} is x_0^{rep}, the value that corresponds with no selected candidate representation. The action set A_{dr} is state dependent according to:

$$A_{dr}(\mathbf{x}) = \begin{cases} A_{sw} & \text{if } \mathbf{x}[X_{rep}] = x_0^{rep} \\ A_{gr} & \text{otherwise} \end{cases} \tag{5.8}$$

From this equation and the initial value of X_{rep} it follows that the agent's first action is always a switch action.

The reward received after taking a switch action is zero, since it is an internal action that does not generate a reward from the environment. Therefore, the reward function of the derived task is, for all $\mathbf{x} \in Dom(\mathbf{X}_{dr})$:

$$\mathcal{R}_{dr}(\mathbf{x}, a) = \begin{cases} 0 & \text{if } a \in A_{sw} \\ \mathcal{R}_{gr}(\mathbf{x}[\mathbf{X}_{gr}], a) & \text{if } a \in A_{gr} \end{cases} \tag{5.9}$$

The transition function of the derived task, $T_{dr}(\mathbf{x}, a, \mathbf{x}') = P_{dr}(\mathbf{x}'|\mathbf{x}, a)$, is defined as follows. We split the transition function up as $P_{dr}(\mathbf{x}'|\mathbf{x}, a) = P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a) \cdot P_{dr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}, a)$. We start by defining the transition function for the switch actions set A_{sw}. The switch action a_k^{sw} sets the value of feature X_{rep} to x_k^{rep} which corresponds to the selection of candidate representation $\mathbf{X}^k$:

$$P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a_n^{sw}) = \begin{cases} 1 & \text{if } \mathbf{x}'[X_{rep}] = x_k^{rep} \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

Note that there is no action a_0^{sw} that sets X_{rep} to x_0^{rep}. Therefore, the second action the agent takes is always a ground action. The values of the other features are left unchanged by the switch actions:

$$P_{dr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}, a_n^{sw}) = \begin{cases} 1 & \text{if } \mathbf{x}'[\mathbf{X}_{gr}] = \mathbf{x}[\mathbf{X}_{gr}] \\ 0 & \text{otherwise} \end{cases} \tag{5.11}$$

The ground actions always result in a terminal state:

$$P_{dr}(\mathbf{x}'|\mathbf{x}, a^{gr}) = \begin{cases} 1 & \text{if } \mathbf{x}' \text{ is a terminal state} \\ 0 & \text{otherwise} \end{cases} \tag{5.12}$$

These following definition gives the complete definition of the derived task.

Definition 16. *The derived task of a contextual bandit with representation selection is formed by the four-tuple $\langle \mathbf{X}_{dr}, A_{dr}, T_{dr}, \mathcal{R}_{dr} \rangle$ defined by Equations 5.6 through 5.12 and the context-specific representation H_{cr} defined as follows:*

$$H_{cr}(\mathbf{x}) = \begin{cases} \{X_{rep}\} & \textit{if } \mathbf{x}[X_{rep}] = x_0^{rep} \\ X_{rep} \cup \mathbf{X}^k & \textit{if } \mathbf{x}[X_{rep}] = x_k^{rep} \,|\, 1 \leq k \leq K \end{cases} \tag{5.13}$$

The optimal policy of the derived task yields the best switch action, i.e., the best representation, and the best ground actions, i.e., the optimal policy for each representation. The following theorem proves that the derived task is in fact Markov, allowing the use of standard RL methods for solving it.

Theorem 8. *The derived task of a contextual bandit problem with representation selection (Definition 16) obeys the Markov property.*

Proof. The Markov property states that the transition probability and expected reward for a state-action pair should be independent of all possible histories. An episode of the derived task consists of two actions: a switch action followed by a ground action. The switch action always obeys the Markov property since there is no history yet. Since the history of the ground action is always the same for a given state $\mathbf{x}$, this action always obeys the Markov property. □

5.2.2.2 Example

Consider a simple contextual bandit problem with two actions, a_0 and a_1, and four states, described by the binary independent features X_1 and X_2. The probability of each feature being true is 0.5. Action a_0 always produces a reward of 0, while the average reward of action a_1 depends on the state, as shown in Table 5.1. From this table, the optimal policy can be easily deduced: action a_1 should be taken in states $\{X_1 = true, X_2 = true\}$ and $\{X_1 = true, X_2 = false\}$ and action a_0 should be taken in the other states. The expected reward of this policy is $\sum_{x_1,x_2} P_0(x_1, x_2) \cdot \max_a R(a, x_1, x_2) = 1.5$. Although

Table 5.1: Rewards and initial state probability P_0 using representation $\{X_1, X_2\}$

X_1	X_2	$P_0(x_1, x_2)$	$R(a_0, x_1, x_2)$	$R(a_1, x_1, x_2)$
true	true	0.25	0	+4.0
true	false	0.25	0	+2.0
false	true	0.25	0	-2.0
false	false	0.25	0	-4.0

the state space of this task is small enough that learning using both features is feasible, for illustrative purposes we assume that the agent must choose between using feature X_1 or feature X_2 as a representation.

The action set of the derived task is augmented with the switch actions a_{X_1} and a_{X_2}, which correspond to selecting candidate representations X_1 and X_2, respectively. The state space is derived with feature $X_{rep} = \{\text{`}\emptyset\text{'}, \text{`}X_1\text{'}, \text{`}X_2\text{'}\}$, whose values refer to the candidate representation that is selected (`$\emptyset$' means no candidate representation is selected).[2] Thus:

$$A_{dr} = \{a_{X_1}, a_{X_2}, a_0, a_1\} \tag{5.14}$$

$$\mathbf{X}_{dr} = \{X_{rep}, X_1, X_2\} \tag{5.15}$$

The derived task makes use of the context-specific representation defined by

$$H_{cr}(\mathbf{x}) = \begin{cases} \{X_{rep}\} & \text{if } \mathbf{x}[X_{rep}] = \text{`}\emptyset\text{'} \\ \{X_{rep}, X_1\} & \text{if } \mathbf{x}[X_{rep}] = \text{`}X_1\text{'} \\ \{X_{rep}, X_2\} & \text{if } \mathbf{x}[X_{rep}] = \text{`}X_2\text{'} \end{cases} \tag{5.16}$$

for all $\mathbf{x} \in Dom(\mathbf{X}_{dr})$. The transition dynamics of this dervied task are summarized in Figure 5.2.

The transition probabilities of the switch actions a_{X_1} and a_{X_2} can be directly deduced from Table 5.1 and are shown in in Tables 5.2 and 5.3, respectively. From these tables it follows that the expected reward is 1.5 for the optimal policy of representation X_1, and 0.5 for that of representation X_2. Thus, after the optimal policy for both representation has been learned, the agent should use representation X_1.

Table 5.2: Rewards and transition probabilities for representation X_1.

X_1	$P(x\|\{\emptyset\}, a_{X_1})$	$R(a_0, x_1)$	$R(a_1, x_1)$
true	0.5	0	+3.0
false	0.5	0	-3.0

[2] We use accolades to distinguish the value `X_1' from the feature X_1.

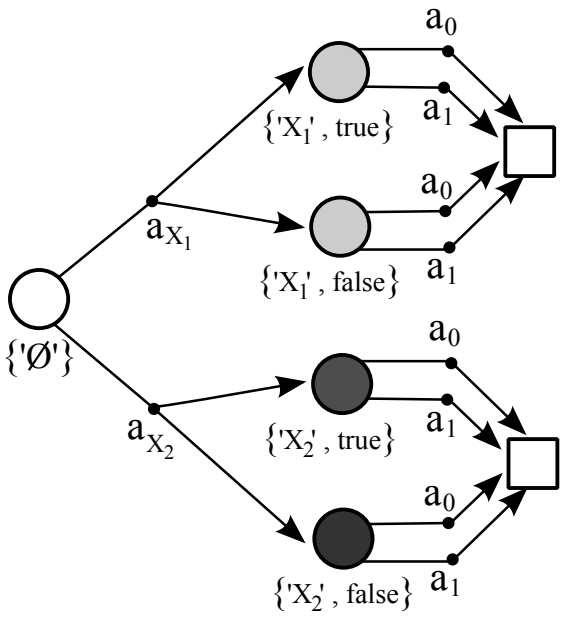

Figure 5.2: Derived MDP for the contextual bandit problem with a choice between two representations, each consisting of a single, binary feature. Circles indicate states and their colors indicate the corresponding representation. Below each state the feature values are shown. Squares indicates terminal states and the black dots indicate actions. Actions with stochastic transitions have multiple arrows.

Table 5.3: Rewards and transition probabilities for representation X_2.

X_2	$P(x_2\|\{\emptyset\}, a_{X_2})$	$R(a_0, x_2)$	$R(a_1, x_2)$
true	0.5	0	+1.0
false	0.5	0	-1.0

5.2.3 Model-Free Updating

In this section, we present an update scheme for the derived task of a contextual bandit problem with representation selection. Since the state space of the derived task can be exponentially smaller than that of the original MDP, using the derived task can greatly reduce the computational, memory, and sample requirements of learning. Furthermore, since the derived task obeys the Markov property, it is itself an MDP. Consequently, in principle, standard TD methods can be used to solve it.[3]

However, in this section we also show how the special properties of the derived MDP can be exploited to speed learning even further. In particular, since the agent can observe all features, even those that are not part of the currently selected representation, it can construct a *parallel experience sequence* for each representation that is not selected. This parallel

[3]Model-based methods could also be applied to the derived MDP. For simplicity, we focus on model-free updates in this chapter.

sequence can be used, under certain conditions, for more efficient *off-policy* updating.

5.2.3.1 Updating Switch Actions

An episode of the derived task consists of a switch action followed by a ground action. Since a ground action always results in a terminal state, its value in a given state, i.e., the Q-value, is simply the expected immediate reward, which can be trivially estimated by averaging the corresponding observed sample rewards. However, estimating the Q-values of switch actions is less straightforward.

Before describing how to do so, we first illustrate why learning such Q-values is necessary. After all, since switch actions are internal, the agent already knows what the next state will be before it takes a switch action. Thus, it may seem that learning Q-values for switch actions is unnecessary since they could simply be inferred from the next state. To see why such a strategy fails, consider the example presented in Section 5.2.2.2. If the current state is $\{X_1 = false, X_2 = true\}$, representation X_1 predicts action a_0 is best (see Table 5.2), yielding an average reward of 0. On the other hand, representation X_2 predicts action a_1 is best (see Table 5.3), yielding an average reward of +1. If representations selection was done merely on the basis of the state values of each representation, then representation X_2 would be chosen, since it predicts a higher reward. However, a_0 is actually the best action (see Table 5.1). The problem is that this strategy implicitly uses both features to select the switch action while the representation consists only of a single feature, yielding a non-Markov task.

Thus, learning separate Q-values for the switch actions is essential. However, the best way to do so is not obvious. Using a simple Q-learning update is not ideal because it bootstraps the value of the switch action with values of the corresponding candidate representation. If this candidate representation is large and has optimistically initialized Q-values, then the Q-value of the switch action will have a positive bias for a long time, even if the representation itself is poor. To get an estimate of how good a candidate representation is more quickly, a Monte Carlo update can be used, which updates a Q-value with the complete observed return and is therefore not biased by values of the candidate representation.

The regular Monte Carlo update is an *on-policy update*: the policy the agent follows (the *behavior policy*) is the same as the policy whose Q-values are being estimated (the *estimation policy*). Alternatively, an *off-policy update* can be performed, in which case the behavior and estimation policies are different. For example, Sutton and Barto (1998) present an off-policy MC update that evaluates the greedy policy, while following an ε-greedy behavior policy (i.e, one that selects the greedy action with $1 - \varepsilon$ probability, while selecting a random action otherwise).

The main disadvantage of off-policy Monte Carlo updates is that learning is inefficient for long episodes because a state-action pair can be updated only when all subsequent actions are greedy, which occurs only rarely given an ε-greedy behavior policy. Fortunately, the derived task of a contextual bandit problem has episodes that are only 2 actions long, reducing the off-policy MC update to a particular simple form. In Appendix B, we show that for the derived task of a contextual bandit problem, the off-policy MC update reduces to one based on the immediate reward of the ground action under the condition that the

ground action is optimal (to ensure the update is unbiased):

$$Q(\mathbf{x}_{t=0}, a_{sw}) = (1-\alpha)\, Q(\mathbf{x}_{t=0}, a_{sw}) + \alpha\, r \qquad \text{if } a_{gr} \text{ is optimal} \tag{5.17}$$

where $\mathbf{x}_{t=0}$ is the start state and r is the reward after the ground action.

5.2.3.2 Off-Policy Updating Ground Actions

The concept of off-policy updating can be used for more than merely evaluating the greedy policy while using an ε-greedy behavior policy. In fact, we can also use off-policy updating to simultaneously update the state-action pairs of candidate representations that are not currently selected, which can speed learning considerably.

Since the agent observes all the available features, it can construct a parallel experience sequence for each candidate representation that was not selected. For example, consider the derived task from Section 5.2.2.2 when the current ground state is $\{X_1 = true, X_2 = false\}$ and the agent takes switch action a_{X_1}. The experience sequence of the derived MDP observed by the agent is:

$$\{\emptyset\} \rightarrow a_{X_1} \rightarrow \{\text{`}X_1\text{'}, true\} \rightarrow a \rightarrow r \tag{5.18}$$

Even though the agent did not take action a_{X_2}, it can deduce what state would result, since this state is simply constructed from feature X_2, which it can also observe. The agent can therefore construct a parallel experience sequence corresponding to representation X_2:

$$\{\emptyset\} \rightarrow a_{X_2} \rightarrow \{\text{`}X_2\text{'}, false\} \rightarrow a \rightarrow r \tag{5.19}$$

This parallel sequence can be used to update the ground action a for representation X_2 as well as the switch action a_{X_2}.

However, the updates performed using such a sequence may be biased, since ground actions for representation X_2 are updated but action selection is based on representation X_1. This bias is a consequence of the fact that state $\{\text{`}X_2\text{'}, true\}$ is actually an aggregation of two states from the ground feature set: $\{X_1 = false, X_2 = true\}$ and $\{X_1 = true, X_2 = true\}$. The expected reward for action $a_1$ in state $\{\text{`}X_2\text{'}, true\}$ is a weighted average of the expected rewards of these two underlying states. Since representation X_1 aggregates the states from the full feature set in a different way, these grounds states correspond to two different states in representation X_1. Therefore, if the selection probability of a_1 is different for those states, the rewards are not properly weighted to correctly estimate the expected reward.

To avoid this problem, the agent can perform such off policy updates only under conditions in which no bias will result. The following theorem establishes those conditions. Note that the theorem is stated in terms of an MDP, of which a contextual bandit problem is a special case.

Theorem 9. *Consider the ground MDP with feature set* $\mathbf{X}$ *and valid candidate representations* $\mathbf{Y} \subseteq \mathbf{X}$ *and* $\mathbf{Z} \subseteq \mathbf{X}$. *Assume the agent selects representation* $\mathbf{Y}$ *to determine its ground action. An unbiased, single-step update of representation* $\mathbf{Z}$ *can be performed, based an the parallel experience sequence, if one of the following two conditions hold:*

1. $\mathbf{Y} \subseteq \mathbf{Z}$
2. *If the action was an exploratory action under an exploration scheme that does not depend upon the specific state, e.g., an ε-greedy exploration scheme.*

Proof. Bias is introduced if a state in $\mathbf{Z}$ is represented by multiple states in $\mathbf{Y}$, while the action outcomes of these multiple states are incorrectly weighted. Under the first condition, a single state of $\mathbf{Z}$ always corresponds to a single state of $\mathbf{Y}$, and therefore no incorrect weighting can occur. For the second condition, recall that since $\mathbf{Z}$ is a valid representation, features that are in $\mathbf{Y}$ but not in $\mathbf{Z}$ are either independent features or features that are irrelevant w.r.t. $\mathbf{Z}$. Features that are irrelevant w.r.t. $\mathbf{Z}$ result in states with equal one-step models, therefore the action outcomes can never be incorrectly weighted. States based on different values of an independent feature occur with a probability that scales with the feature value probability, since the agent has no control over the value of independent features. Therefore, if the exploration scheme does not depend upon the state, a particular action will also be selected with a probability that scales with the feature value probability, hence correctly weighting the action outcomes. □

Thus, when one of the conditions of Theorem 9 holds, the reward can be used to update the ground action of the unselected representation as if it were the selected representation.

5.2.3.3 Off-Policy Updating Switch Actions

To be able to perform an unbiased Monte Carlo update based on the parallel experience sequence, all transitions involved must be unbiased. In the previous section, we showed that an update of the ground action is unbiased if one of the conditions of Theorem 9 holds. However, the switch action update is always unbiased because the state to which it belongs is the same for all switch actions. Therefore, when one of the conditions of Theorem 9 holds, an unbiased Monte Carlo update can also be performed for the corresponding switch action. In other words, both the switch and ground actions of a candidate representation can be updated without selecting that representation. As a result, in contextual bandit problems, representations can be fully evaluated in an off-policy manner. An immediate consequence is that the agent can use greedy action selection for the switch actions, since exploring the switch actions is unnecessary.

Table 5.4 summarizes our action selection and update strategy for the derived task of the contextual bandit problem. In the next section, we illustrate the performance improvements that this update scheme makes possible.

5.2.4 Experimental Results

In the section, we present two experiments involving representation selection for a contextual bandit problem. For both experiments we use the action selection and update strategy described in Table 5.4.

In the first experiment, we consider two versions of a contextual bandit problem, one with three features and the other with four. In both cases, the agent has the prior knowledge

Table 5.4: Action selection and update strategy for the derived task of a contextual bandit problem with representation selection. The “if subset/on explore" condition refers to the two conditions of Theorem 9.

	action selection	update, current rep	update, other reps
ground action	ε-greedy	average	if subset/on explore: average
switch action	greedy	MC update	if subset/on explore: MC update

that only one of the available features is relevant, while the others are empty. However, the agent does not know in advance which feature is relevant. Each feature has eight feature values, which are initialized randomly after each arm pull. The problem has two arms with opposite expected reward: depending on the context, one has an expected reward of +1, while the other has -1. The reward is drawn from a normal distribution with a standard deviation of 2. For half of the feature values of the relevant feature, the first arm has the +1 expected reward. Therefore, when this feature is ignored, the expected reward is zero.

The switching method uses one candidate representation for each feature, a learning rate of 0.01 for the switch action, and an ε of 0.2 for the ε-greedy selection of the ground action. To kick-start the switch method, we use an ε of 1.0 for the first 50 episodes. Since all candidate representations are updated during this exploration phase, this has a positive effect on the total reward. We compare the performance of the switching method against two alternatives that represent upper and lower bounds on performance. The lower bound is achieved by a naïve method that ignores the prior knowledge about the candidate representations and simply learns on the original task using the full representation. The upper bound is achieved by informing the agent in advance which candidate representation is correct and letting it learn using only this representation from the start.

Figure 5.3 shows the results, averaged over 10,000 independent runs and smoothed. The performance of the switch method illustrates how effectively it can exploit prior knowledge about the set of candidate representations. While the size of the full representation grows exponentially with the number of features (512 states for a set of 3 features and 4096 for a set of 4 features), the total number of states for the switching methods grows only linearly (24 states for a set of 3 features and 32 states for a set of 4 features). The simultaneous updating of the representations increases performance even further, making it nearly indistinguishable from the agent that knows in advance which candidate representation is correct.

The second experiment is a variation on the first experiment. There are three features in total (X_1, X_2 and X_3), each with eight feature values, which are selected randomly after each arm pull. This time, however, there is not a single relevant feature; instead, all three features contain some relevant information. From Table 5.5, which shows the corresponding rewards and probabilities, it follows that the expected reward of the optimal policy for this representation is +1. The size of the state-space of this representation is $8^3 = 512$.

Table 5.6 is deduced from Table 5.5 and shows the rewards conditioned on only feature X_1. While all features are necessary to achieve an expected reward of +1, using only feature

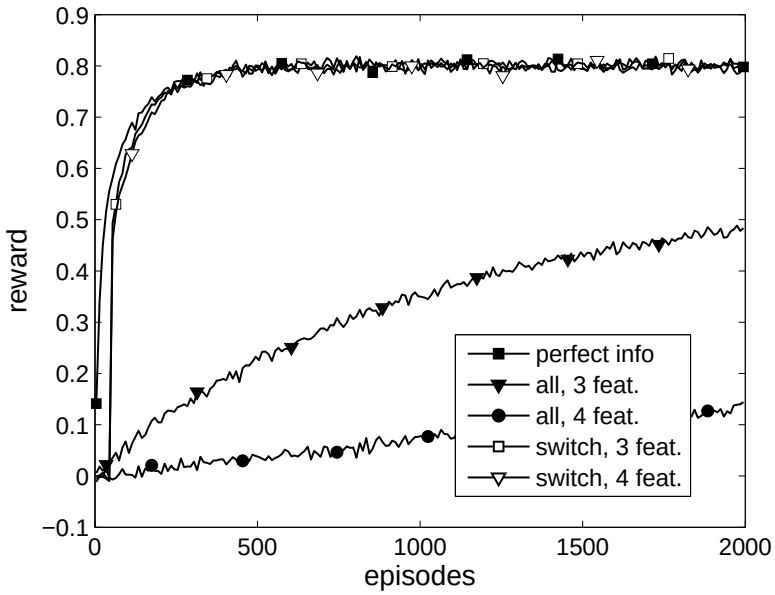

Figure 5.3: The performance of the switch method on contextual bandit problems with either three or four features, compared to a naïve method that uses all the available features ('full') and one that knows in advance which candidate representation is correct ('perfect info').

Table 5.5: Rewards and initial state probability P_0 using representation $\{X_1, X_2, X_3\}$. The true/false labels are derived from the real feature values: for X_1 and X_2, 4 of the 8 values correspond to a 'true' label, while the other values correspond to a 'false' label. For X_3, 6 of the 8 features correspond to a 'true' label, while the other values correspond to a 'false' label.

X_1	X_2	X_3	$P_0(x_1, x_2, x_3)$	$R(a_0, x_1, x_2, x_3)$	$R(a_1, x_1, x_2, x_3)$
true	true	true	0.1875	+1	-1
true	true	false	0.0625	-1	+1
true	false	true	0.1875	+1	-1
true	false	false	0.0625	+1	-1
false	true	true	0.1875	-1	+1
false	true	false	0.0625	+1	-1
false	false	true	0.1875	-1	+1
false	false	false	0.0625	-1	+1

X_1, results in an expected reward that is only slightly less (0.75), while the state-space size is considerably smaller (8 states). We compare the performance of 1) learning with a representation containing only feature X_1 (REP-SMALL), 2) learning with a representation containing all three relevant features (REP-LARGE), and 3) using the switch method given both representations as candidates (SWITCH). We use a learning rate of 0.001 for the switch

action and an ε of 0.2 for the ε-greedy selection of the ground action for all methods. The switch method uses an ε of 1.0 for the first 100 episodes.

Table 5.6: Rewards and initial state probability P_0 when conditioned on only X_1

X_1	$P_0(x_1)$	$R(a_0, x_1)$	$R(a_1, x_1)$
true	0.5	0.75	-0.75
false	0.5	-0.75	0.75

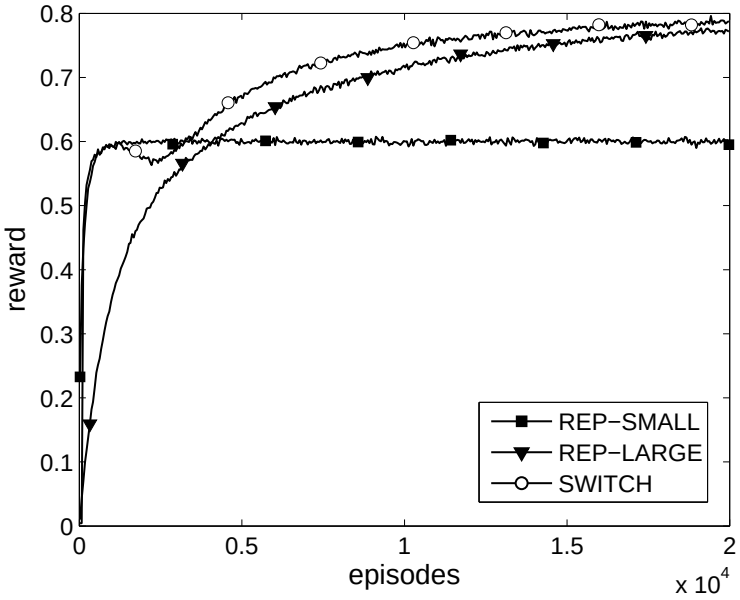

Figure 5.4: Average reward for a contextual bandit when switching between a large and a small representation.

Figure 5.4 shows the average reward for the first 20,000 episodes, averaged over 10,000 independent runs and smoothed. For approximately the first 5000 episodes, REP-SMALL outperforms REP-LARGE since it learns more quickly. However, since its representation does not contain all relevant information, it plateaus below REP-LARGE, which ultimately performs much better. After the initial exploration phase, the switch method quickly catches up with REP-SMALL, after which the performance shows a small dip before climbing above that of REP-LARGE.

We explain this dip as follows: while the small representation is quickly recognized as the better one initially, the agent uses off-policy updating to keep improving the Q-values of both the large representation and the switch actions. In some runs, once the Q-value of the selection action for the large representation approaches that of the small representation, the estimates will prematurely indicate that the large one is better, causing a small dip in the performance.

After the dip, the Q-values of of the large representation improve further, exceeding

those of the small representation and causing the second climb in performance. Interestingly, the switch method outperforms REP-LARGE at each point during learning. The reason is that the exact point where REP-LARGE outperforms REP-SMALL is slightly different for each run. Since the switch method uses an up-to-date estimate of the expected reward for each representation, it simply makes longer use of the small representation for those runs where the Q-values of the large representation improve more slowly than average. Therefore, once the Q-values of the small representation have been properly learned, the performance of REP-SMALL forms a lower bound for the remaining episodes. This lower bound is not present for REP-LARGE, whose performance is bounded only by zero, leading to lower average performance.

The results of this experiment underscore an important advantage of the switching method. Because it evaluates candidate representations on-line, it can automatically identify, not only the best representation to use in the long run, but also the best one to use *during learning*. For example, the small representation, although ultimately inferior to the large one, is preferable early in learning when insufficient data is available for the large representation to be effective. The switch method gets the best of both worlds, mimicking the performance of REP-SMALL early in learning and that of REP-LARGE later on.

5.3 Representation Selection for MDPs

In the previous section, we showed that a contextual bandit problem with representation selection can be modeled as a derived task with a single, context-specific representation. The solution of this derived task not only yields the best candidate representation for the contextual bandit problem, but also the optimal policy of that representation. In this section, we discuss representation selection for an MDP. As in a contextual bandit problem, representation selection for an MDP can be modeled as a derived task. In this case, after taking a switch action to select a representation, the agent takes not one but a series of ground actions until a terminal state is reached. The ground actions are chosen based on the Q-values of the selected representation.

The definition of the derived task of an MDP with representation selection is similar to that of a contextual bandit problem. However, we show that the conditions under which this derived task obeys the Markov property, i.e., is a derived MDP, are more restrictive. In particular, we prove that when all candidate representations are valid with respect to the ground MDP, the derived task obeys the Markov property, in which case standard RL methods can be used to solve it.

5.3.1 Derived Tasks

Consider the ground MDP $M_{gr} = \langle \mathbf{X}_{gr}, A_{gr}, T_{gr}, \mathcal{R}_{gr} \rangle$ with K candidate representations: $\mathbf{X}^1, ..., \mathbf{X}^K$, where each $\mathbf{X}^k \subset \mathbf{X}_{gr}$. In the derived task for this MDP, the agent first takes a switch action, selecting a representation, and then a series of ground actions, based on the Q-values of the selected representation, until a terminal state is reached. The construction of the derived task for this MDP is similar to that of the contextual bandit problem (see Section 5.2.2.1). In fact, Equations 5.6 through 5.11, defining the different components of

the derived task, are still valid for the MDP case. The only difference is that now a ground action can result not only in a terminal state, but also in a different state belonging to the same candidate representation. Therefore, the transition function of the ground actions, $P_{dr}(\mathbf{x}'|\mathbf{x}, a^{gr})$, is defined, not by Equation 5.12, but as follows:

$$P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a^{gr}) = \begin{cases} 1 & \text{if } \mathbf{x}'[X_{rep}] = \mathbf{x}[X_{rep}] \\ 0 & \text{otherwise} \end{cases} \tag{5.20}$$

In other words, since the candidate representation stays the same, the ground action has no effect on the value of X_{rep}. Furthermore:

$$P_{dr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}, a^{gr}) = P_{gr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}[\mathbf{X}_{gr}], a^{gr}) \tag{5.21}$$

That is, the effect of the ground action on the other features is the same as in the ground MDP.

The context-specific representation for the derived task of an MDP is equal to that of the contextual bandit problem, as can shown by the following definition:

Definition 17. *The derived task of an MDP with representation selection is formed by the four-tuple $\langle \mathbf{X}_{dr}, A_{dr}, T_{dr}, \mathcal{R}_{dr} \rangle$ defined by Equations 5.6 through 5.11, Equations 5.20 and 5.21 and the context-specific representation H_{cr} defined as following:*

$$H_{cr}(\mathbf{x}) = \begin{cases} \{X_{rep}\} & \textit{if } \mathbf{x}[X_{rep}] = x_0^{rep} \\ X_{rep} \cup \mathbf{X}^k & \textit{if } \mathbf{x}[X_{rep}] = x_n^{rep} \,|\, 1 \leq k \leq K \end{cases}$$

In contrast to the derived task of a contextual bandit problem, the derived task of an MDP is not always Markov. The following theorem specifies a sufficient condition for guaranteeing that the derived task obeys the Markov property, making it possible to apply standard RL methods to solve it.

Theorem 10. *The derived task of an MDP M with representation selection (Definition 17) obeys the Markov property if all candidate representations are valid w.r.t. M.*

Proof. An episode of the derived task of an MDP with representation selection starts with a switch action, selecting a candidate representation, and then a series of ground actions until a terminal state is reached. The switch action always obeys the Markov property since there is no history yet. The history of the first ground action is always the same for a given state $\mathbf{x}$. Therefore, the first ground action also always obeys the Markov property. For the ground actions after that, the Markov property follows from the fact that the selected candidate representation is valid. □

5.3.2 Model-Free Updating

In this section, we discuss two different update strategies. As before, for simplicity, we restrict ourselves to model-free learning. The first strategy adapts the update scheme used

for contextual bandit problems to the full MDP setting. For this strategy, the ground actions are updated with Q-learning updates. As with contextual bandit problems, a parallel experience sequence is created for each candidate representation. In the MDP setting, this parallel sequence is used to perform off-policy Q-learning updates. These updates are applied to the ground actions of all unselected candidate representations for which at least one of the conditions of Theorem 9 holds, guaranteeing that the updates are unbiased.

As before, a Monte Carlo update is used for the switch action. However, the off-policy MC update described in Section 5.2.3.1 is inefficient for the full MDP case, since it can only be performed when all actions following the switch action are greedy. Therefore, we use an on-policy MC update instead. Updating the switch actions of other representations with an MC update requires that all actions are unbiased, which is not the case in general. Therefore, such updates are not performed. A consequence is that each switch action needs to be explored in order to obtain accurate Q-value predication for that action. We call the resulting update scheme, summarized in Table 5.7, the *Monte Carlo update scheme* since a Monte Carlo update is used for the switch action.

Table 5.7: Monte Carlo update scheme. The “if subset/on explore" condition refers to the two conditions of Theorem 9.

	action selection	update, current rep	update, other reps
ground action	ε_{gr}-greedy	Q-learning	if subset/on explore: Q-learning
switch action	ε_{sw}-greedy	MC-update	-

The second strategy uses a Q-learning update for the ground actions as well as the switch actions. The update of a switch action is performed for *all* candidate representations *before* selecting the switch action, since the agent can already observe the result of a switch action before taking it. Performing the update before selecting the action is preferable, since the selection is then based on more accurate Q-values. Since the ground actions as well as the switch action of a candidate representation can be updated without selecting it, exploring the switch actions is unnecessary and thus the agent can always choose the greedy switch action.

We call the resulting update scheme, summarized in Table 5.8, the *Q-learning update scheme* since a Q-learning update is used for the switch action.

Table 5.8: Q-learning update scheme. The “if subset/on explore" condition refers to the two conditions of Theorem 9.

	action selection	update, current rep	update, other reps
ground action	ε-greedy	Q-learning	if subset/on explore: Q-learning
switch action	greedy	early Q-learning	early Q-learning

Each scheme has its advantages and disadvantages. On the one hand, the MC update scheme has switch actions that are not biased by the values of the candidate representa-

tions; therefore, it is expected to more quickly learn an accurate estimate of a representation's value. On the other hand, it does not update the switch actions off-policy; therefore, more exploration is required. The Q-learning update scheme can perform off-policy updates of the ground actions as well as the switch actions, allowing greedy selection of the switch action. However, the values of the switch actions are bootstrapped from the representations. Thus, each candidate representation needs to be sufficiently explored before its switch action has an accurate value. In the next section, we compare both update schemes experimentally.

5.3.3 Experimental Results

In this section, we present experimental results evaluating both the Monte Carlo and Q-learning update schemes described above. These experiments are conducted on an MDP we call the *Mars rover task*. Suppose a rover on a Mars mission must frequently navigate between its home base and a research site. The area it must cross can be described by a 15×15 square grid, with the home base and research site in opposite corners. The rover observes its current position and has four movement actions: *north*, *south*, *east* and *west*, which, on a regular surface, cause a movement of one square in the corresponding direction. However, on the sandy soil of Mars, the rover's action outcomes are heavily distorted. The effect of the distortion can be modeled as an additional (clockwise) rotation of either 0, 90, 180 or 270 degrees applied before the directional movement. A *north* action can for example lead to an *east* movement if the distortion is 90 degrees.

The distortion is affected by the local sand structure of the rover's current location, which consists of a number of ditches of different sizes. These ditches are described by a set of structural features that the agent observes along with its position (see Figure 5.5). Each structural feature corresponds to a different ditch size, while its value indicates the number of ditches of that size on the local square. The structure is not static, but changes each time the rover enters a square, according to some probability distribution, due to the interaction between the rover and the sand.

By learning the relationship between the structural properties and the distortion for a certain square, the rover can compensate for the distortion. However, learning with all structural features can be prohibitively slow. Fortunately, experiments on Earth showed that only ditches of one size cause the distortion, i.e., only one structural feature is relevant. Which feature this is depends on the grain size of the sand, which cannot be observed. Therefore the rover must learn which of the available structural features is the relevant one.

In our experimental setup, we assume the local sand structure can be described by 5 features, each having 4 different feature values. The features are independent and all feature values have equal probability. Each of the 4 values of the relevant feature corresponds to a different distortion values (0, 90, 180 or 270 degrees). In addition to the distortion caused by the local sand structure, there is a 10% chance on an additional distortion, modeled as another rotation of 0, 90, 180 or 270 degrees (each value has equal probability).

Under these settings, the unconditional transition probabilities (i.e., ignoring the relevant structural feature) are the same for all actions. Consequently, without considering the structural feature, the agent cannot learn an effective policy, since all actions have the

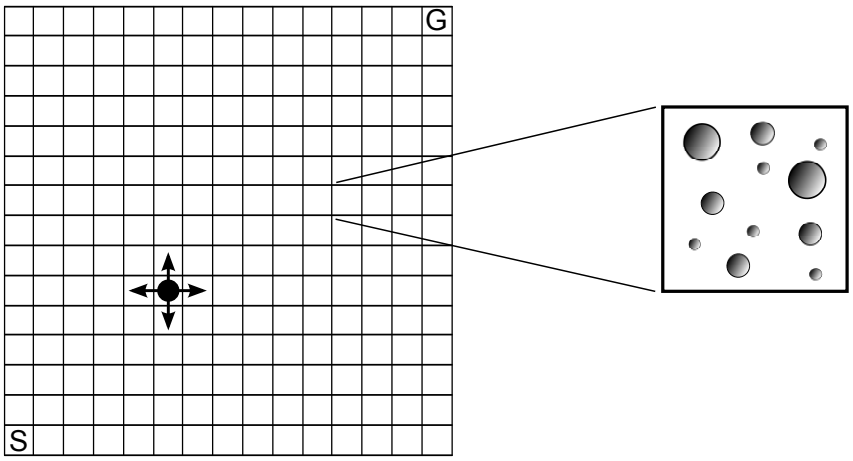

Figure 5.5: Mars rover task: the agent must move from S to G using four directional actions. The local sand structure, consisting of ditches of different sizes, causes a distortion of the regular action outcome. If the agent learns the relationship between the local sand structure and the distortion, it can compensate for it and thus reach the goal location more quickly.

same effect and the agent moves randomly through the grid. When the agent does consider the relevant feature, it can learn to compensate for the distortion caused by the sand structure, causing near-deterministic action outcomes (the additional distortion cannot be compensated for).

Our experiments compare the performance of five different algorithms. The first two are the Monte Carlo and Q-learning representation selection algorithms described in Section 5.3.2. The third algorithm uses perfect information, i.e., only the position feature and the relevant structural feature. As in the experiments presented in Section 5.2.4, this algorithm provides an upper bound on performance. The fourth and fifth algorithms provide lower bounds on performance by ignoring the prior knowledge that only one of the structural features is necessary. The fourth algorithm algorithm uses all the available features: both the position feature and all 5 structural features. The fifth algorithm does not use any structural features, relying only on position.

For each algorithm, we measure the average return over the first 1000 episodes. All algorithms use ε-greedy ground action selection with $\varepsilon = 0.1$ and a learning rate with an initial value α_0 of 1.0 that is decayed according to:[4]

$$\alpha(\mathbf{x}, a) = \alpha_0 \, d^{n(\mathbf{x},a)} \tag{5.22}$$

where $n(\mathbf{x}, a)$ is the number of times action a was previously selected in state $\mathbf{x}$. We optimize the decay rate d for each method. For the switch methods, the extra parameters are also optimized. The range for which parameters are optimized is determined by performing

[4]This type of decay does not meet the requirements for convergence in the limit of many TD algorithms (Jaakkola et al., 1994; Singh et al., 2000). However it gives good results in practice and is very easy to implement.

some initial experiments to find roughly the settings with the best performance. For the Q-learning update scheme we use full exploration for the first s_ε episodes (the value of s_ε is optimized). We do not do this for the Monte Carlo update scheme since the initial experiments showed that this extra parameter causes negligible performance improvement. All results are averaged over 200 independent runs and smoothed.

Table 5.9 shows the average performance of the different methods over the first 1000 episodes together with the optimal parameters, while Figure 5.6 plots the average return over time for these optimal parameters. As predicted, when the structural features are ignored, no learning occurs since all actions have the same expected outcome. The agent moves randomly through the environment, generating large negative reward. In contrast, learning does occur when all structural features are used. However, because of the size of the resulting state space, learning is slow and the performance improvement is marginal.

Both of the switching methods perform much better, generating up to 9 times less negative reward than when using the full feature set. Comparing the two switching methods with each other reveals that the Q-learning update scheme outperforms the Monte Carlo update scheme, generating 1.3 times less negative reward. Apparently, the advantage of the Q-learning update scheme (more off-policy updates resulting in less exploration) outweighs its disadvantage (bootstrapping from the representations values).

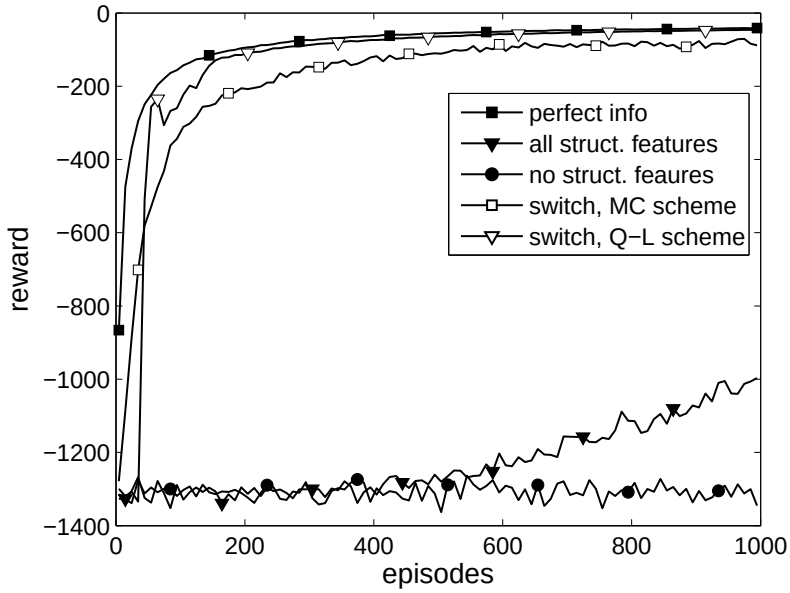

Figure 5.6: Performance as a function of number of episodes on the Mars rover task.

These results show that tasks with a limited number of candidate representations can be efficiently solved using our approach of a derived MDP with a context-specific representation. In the next section, we analyze what happens when the candidate representations themselves are context-specific.

Table 5.9: Average performance over the first 1000 episodes and optimal parameters on the Mars rover task.

	optimal parameters	average return	standard error
perfect info	d = 1%	-87.75	0.05
all struct. features	d = 1%	-1226	1
no struct. features	(no learning occurs)	-1308	2
switch, MC scheme	$\varepsilon_{sw} = 0.04$, $d_{sw} = 1\%$, $d_{ex} = 2\%$	-180	7
switch, Q-L scheme	$s_{\varepsilon} = 40$, $d_{sw} = 7\%$, $d_{ex} = 2\%$	-138	1

5.4 Representation Selection for MDPs with Context-Specific Structure

In the previous two sections, we considered tasks for which the best representation had globally the same features. However, many real-world problems have a context-specific structure. Consider for example an extension of the Mars rover task from Section 5.3.3, where the rover encounters different types of sand along its journey. The relevant feature can be different for each of these types, resulting in a context-specific representation. To find the best representation for this task, the candidate representations must also be context-specific.

The solution strategy developed in Section 5.3 can in principle also be applied when the candidate representations are context-specific. A problem however is that the number of candidate representations can increase exponentially when context-specific structure is introduced. Consider the extended Mars rover task described above with 6 types of sand and choice between 5 different features. If the rover can distinguish between the different sand types, then there are $6^5 = 7776$ possible assignments of relevant features to sand types possible, resulting in 7776 candidate representations. When the agent cannot distinguish between the different sand types, things get even worse since the agent must learn the relevant feature for each position value. With 225 positions (the number of positions from the original Mars rover task), there are $225^5 \approx 6 \cdot 10^{11}$ candidate representations. Thus, the strategy from Section 5.3, which results in a derived task whose state space size is linear in the number of candidate representations, is no longer feasible.

In this section, we introduce a strategy for context-specific candidate representations that results in a derived task with an exponentially smaller state space. With this strategy, the $6 \cdot 10^{11}$ context-specific candidate representations from the extended Mars rover task can be evaluated without explicit enumeration of each context-specific candidate representation. In short, our strategy allows the agent to switch between representations in the middle of an episode, instead of just selecting one at the start of an episode. By doing so, a small number of regular representations (5 in the case of the extended Mars rover task) can be used to represent and evaluate a large number of context-specific representations.

5.4.1 Candidate Context Representations

Before explaining how the derived task is constructed, we start with the problem description. Specifically, we explain how available prior knowledge about the structure of a problem is expressed.

In Section 5.1.4, we defined a context-specific representation by the mapping $H_{cs} : Dom(\mathbf{X}) \rightarrow \mathcal{P}(\mathbf{X})$, which maps each state $\mathbf{x}$ to a subset of features. Although problems with context-specific structure use different features for different states, in practice, many states share the same set of features. To represent prior knowledge about states sharing the same features, we use a refined formulation based on *contexts*. Contexts are aggregations of states that share the same state features. The feature C is the set of all contexts. We call the state features that are used within a context the *context representation*. The mapping H_{cs} can now be decomposed as $H_{cs}(\mathbf{x}) = H_2(H_1(\mathbf{x}))$ where $H_1 : Dom(\mathbf{X}) \rightarrow C$ maps a state $\mathbf{x}$ to a context $c \in C$ and $H_2 : C \rightarrow \mathcal{P}(\mathbf{X})$ maps a context c to a context representation $\mathbf{X}^c$.

H_1 (and consequently H_2) is not uniquely defined: there are many possible aggregations of states into contexts. As long as the relevant features within each aggregation are the same, H_1 is a legitimate mapping. Therefore, an H_1 mapping can be defined even without prior knowledge (by mapping each state to a separate context). Thus, we assume H_1 is known to the agent, while H_2 needs to be learned.

Note that with the H_1/H_2 decomposition, two types of prior knowledge related to the context-specific structure can be represented. Knowledge about states sharing the same features can be represented by H_1, by mapping states with the same relevant features to the same context. On the other hand, prior knowledge about relevant features is represented by sets of *candidate context representations* for H_2, giving the agent the choice, for each context, between a set of different context representations.

5.4.2 Derived Tasks

Consider the ground MDP $M_{gr} = \langle \mathbf{X}_{gr}, A_{gr}, T_{gr}, \mathcal{R}_{gr} \rangle$ which has a context-specific structure. The set of context-specific candidate representations is implicitly defined by H_1, which maps each state $\mathbf{x}$ to a value of the context feature $C = \{c_1, ...c_J\}$ and the set of K_j candidate context representations: $\{\mathbf{X}_1^{c_j}, ...\mathbf{X}_{K_j}^{c_j}\}$ for each context $c_j \in C$.

Our strategy is to keep the representation the same as long as the agent stays in the same context but let the agent choose a context representation whenever it enters a new context. The agent chooses a context representation by taking the corresponding switch action. Like in Section 5.3.1, the X_{rep} feature is added to the feature set of the derived task to specify what candidate representation is currently selected:

$$\mathbf{X}_{dr} = X_{rep} \cup \mathbf{X}_{gr} \tag{5.23}$$

The feature values of X_{rep} are:

$$X_{rep} = \{x_0^{rep}, x_1^{c_1}, ...x_{K_1}^{c_1}, ..., x_1^{c_J}, ...x_{K_J}^{c_J}\} \tag{5.24}$$

where the value $x_k^{c_j}$ corresponds to the kth candidate context representation of the jth context, $\mathbf{X}_k^{c_j}$, while x_0^{rep} is the value that indicates no representation is currently selected.

Switch actions can be taken only when $X_{rep} = x_0^{rep}$. In the derived task of Section 5.3.1, x_0^{rep} is the initial value of X_{rep}. After the first timestep, x_0^{rep} was never selected again. This is no longer the case. Instead, since the agent must select a context representation each time it changes context, ground actions that cause a change of the context set the value of X_{rep} equal to x_0^{rep}. In other words, if $H_1(\mathbf{x}') \neq H_1(\mathbf{x})$, the transition function for X_{rep} is:

$$P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a^{gr}) = \begin{cases} 1 & \text{if } \mathbf{x}'[X_{rep}] = \mathbf{x}_0^{rep} \\ 0 & \text{otherwise} \end{cases} \tag{5.25}$$

On the other hand, when the context stays the same, so does the value of X_{rep}. So, if $H_1(\mathbf{x}') = H_1(\mathbf{x})$:

$$P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a^{gr}) = \begin{cases} 1 & \text{if } \mathbf{x}'[X_{rep}] = \mathbf{x}[X_{rep}] \\ 0 & \text{otherwise} \end{cases} \tag{5.26}$$

The effect of the ground action on the value of the other features is the same as under the ground MDP:

$$P_{dr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}, a^{gr}) = P_{gr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}[\mathbf{X}_{gr}], a^{gr}) \tag{5.27}$$

Whenever $X_{rep} = x_0^{rep}$ the agent must choose a representation by selecting a switch action. The switch actions the agent can choose from depend on the context. There is a switch action set $A_{sw}^{c_j} = \{a_1^{c_j}, ..., a_{K_j}^{c_j}\}$ associated with each context $c_j \in C$, whose member actions correspond to the different candidate context representations of c_j. The effect of the switch action $a_k^{c_j}$ is that the value of X_{rep} is set to $x_k^{c_j}$, which corresponds to candidate context representation $\mathbf{X}_k^{c_j}$:

$$P_{dr}(\mathbf{x}'[X_{rep}]|\mathbf{x}, a_k^{c_j}) = \begin{cases} 1 & \text{if } \mathbf{x}'[X_{rep}] = x_k^{c_j} \\ 0 & \text{otherwise} \end{cases} \tag{5.28}$$

The values of the other features are left unchanged by the switch actions:

$$P_{dr}(\mathbf{x}'[\mathbf{X}_{gr}]|\mathbf{x}, a_k^{sw}) = \begin{cases} 1 & \text{if } \mathbf{x}'[\mathbf{X}_{gr}] = \mathbf{x}[\mathbf{X}_{gr}] \\ 0 & \text{otherwise} \end{cases} \tag{5.29}$$

The total action set of the derived task is:

$$A_{dr} = A_{gr} \cup A_{sw}^{c_1} \cup ... \cup A_{sw}^{c_J} \tag{5.30}$$

while the actions available in a particular state $\mathbf{x} \in \mathbf{X}_{dr}$ are:

$$A_{dr}(\mathbf{x}) = \begin{cases} A_{sw}^{c_j} & \text{if } \mathbf{x}[X_{rep}] = x_0^{rep} \\ A_{gr} & \text{otherwise} \end{cases} \tag{5.31}$$

where $c_j = H_1(\mathbf{x})$.

The reward received after taking a switch action is zero, since it is an internal action. Therefore, the reward function of the derived task is, for all $\mathbf{x} \in Dom(\mathbf{X}_{dr})$:

$$\mathcal{R}_{dr}(\mathbf{x}, a) = \begin{cases} \mathcal{R}_{gr}(\mathbf{x}[\mathbf{X}_{gr}], a) & \text{if } a \in A_{gr} \\ 0 & \text{otherwise} \end{cases} \tag{5.32}$$

The full definition of the derived task of an MDP with context representation selection is as follows:

Definition 18. *The derived task of an MDP with context representation selection is formed by the four-tuple $\langle \mathbf{X}_{dr}, A_{dr}, T_{dr}, \mathcal{R}_{dr} \rangle$ defined by Equations 5.23 through 5.32 and the context-specific representation H_{cr} defined as follows:*

$$H_{cr}(\mathbf{x}) = \begin{cases} X_{rep} \cup \mathbf{X}_{sw} & \text{if } \mathbf{x}[X_{rep}] = x_0^{rep} \\ X_{rep} \cup \mathbf{X}_k^{c_j} & \text{if } \mathbf{x}[X_{rep}] = x_k^{c_j} \mid 1 \leq j \leq J \end{cases} \tag{5.33}$$

where $\mathbf{X}_{sw} \subseteq \mathbf{X}_{gr}$.

Note that the definition of the context-specific representation is similar to that of Section 5.3.1, except for the addition of the feature set $\mathbf{X}_{sw}$ when $X_{rep} = x_0^{rep}$. We call $\mathbf{X}_{sw}$ the *switch representation*. These extra features are necessary to ensure that convergence of the standard RL algorithms can be obtained.

This derived task is Markov only under strict conditions. However, we can relax these conditions by requiring a slightly weaker form of the Markov property, which we call the *Markov property with respect to relevant features*. This property holds if the values of the relevant features and the reward depend only on the current state and action.

Definition 19. *A task is Markov with respect to relevant features if the following equation holds for all $\mathbf{x}_{t+1}, r_{t+1}$ and all possible values of $\mathbf{x}_t, a_t, r_t, ..., r_1, \mathbf{x}_0, a_0$.*

$$P(\mathbf{x}_{t+1}[\mathbf{Y}], r_{t+1} | \mathbf{x}_t, a_t) = P(\mathbf{x}_{t+1}[\mathbf{Y}], r_{t+1} | \mathbf{x}_t, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, ..., r_1, \mathbf{x}_0, a_0) \tag{5.34}$$

where $\mathbf{Y} = \mathbf{X} \setminus \mathbf{X}_{irr}$ and $\mathbf{X}_{irr}$ is a subset of features that are irrelevant w.r.t. $\mathbf{Y}$.

Since irrelevant features do not affect the values of other features or the reward, states that differ from each other only in their irrelevant feature values have the same state value. Therefore, the standard RL algorithms converge when a task is Markov with respect to the relevant features.

The following theorem specifies a set of conditions that guarantees the derived task defined by Definition 18 obeys the Markov property with respect to relevant features:

Theorem 11. *The derived task of an MDP M with context representation selection (Definition 18) is Markov with respect to relevant features if the following two conditions hold:*

1. *$\mathbf{X}_{sw}$ is valid w.r.t. M, and*
2. *$\mathbf{X}_{sw} \subseteq \mathbf{X}_k^{c_j}$ for all values of k and j.*

Proof. Let $\mathbf{Z}_t$ be the representation at timestep t and $\mathbf{Z}_{t+1}$ be the representation at timestep $t+1$. Since $\mathbf{X}_{sw} \subseteq \mathbf{Z}_{t+1}$ and since $\mathbf{X}_{sw}$ is a valid representation (i.e., all the features not in $\mathbf{X}_{sw}$ are either independent or irrelevant w.r.t $\mathbf{X}_{sw}$) $\mathbf{Z}_{t+1}$ can be decomposed as $\mathbf{Z}_{t+1} = \mathbf{X}_{sw} \cup \mathbf{X}_{ind} \cup \mathbf{X}_{irr}$, where $\mathbf{X}_{ind}$ is the set of independent features of $\mathbf{Z}_{t+1}$ and $\mathbf{X}_{irr}$ is the set of features that are irrelevant w.r.t. $\mathbf{X}_{sw}$ that are part of $\mathbf{Z}_{t+1}$.

Let $\mathbf{Y} = \mathbf{X}_{sw} \cup \mathbf{X}_{ind}$. To prove the theorem we need to prove that Equation 5.34 holds, i.e. we need to prove that the feature values of $\mathbf{Y}$ only depend on the feature values of $\mathbf{Z}_t$ and not on the history. The features $\mathbf{X}_{ind}$ are independent, so these features do not depend upon the history by definition. On the other hand, the feature values from $\mathbf{X}_{sw}$ do not depend upon the history, since the set $\mathbf{X}_{sw}$ forms a valid representation. □

5.4.3 Model-Free Updating

The derived task described in the previous subsection can be solved using update schemes similar to those for the derived task of an MDP with regular candidate representations (see Section 5.3.2). The only difference arises from the fact that now a state with switch actions can be revisited multiple times before a terminal state is reached. Therefore, using a Monte Carlo update for the switch actions is less suitable, since it is an off-line update that misses out on the opportunity to update the Q-values during the episode. Instead, we use an *n-step update*, which accumulates the reward received after taking the switch action until the agent re-enters a state with switch actions, i.e., until the context changes. This n-step update still has the advantage of not bootstrapping from a representation's Q-values. However, it is also an on-line update, i.e., it is performed *during* an episode. We will refer to this update scheme as the *n-step update scheme*. Besides this update scheme, the *Q-learning update scheme* from Section 5.3.2 can be applied. This update scheme can be applied without making any modifications.

We summarize the update schemes for updating the Q-values of the derived task of an MDP with context representation selection in Table 5.10 and Table 5.11.

Table 5.10: n-step update scheme.

	action selection	update, current rep	update, other reps
ground action	ε_{ex}-greedy	Q-learning	if subset/on explore: Q-learning
switch action	ε_{sw}-greedy	n-step update	-

Table 5.11: Q-learning update scheme.

	action selection	update, current rep	update, other reps
ground action	ε-greedy	Q-learning	if subset/on explore: Q-learning
switch action	greedy	early Q-learning	early Q-learning

In the next subsection, we compare these update schemes experimentally on an extension of the Mars rover task.

5.4.4 Experimental Results

In this subsection, we consider an extension of the Mars rover task of Section 5.3.3. In this extension, the area the rover has to cross to get from the start state to the goal state is

divided into 4 types of sand (see Figure 5.7). Positions with the same sand type have the same relevant structural feature, but the relevant feature can differ for each sand type.

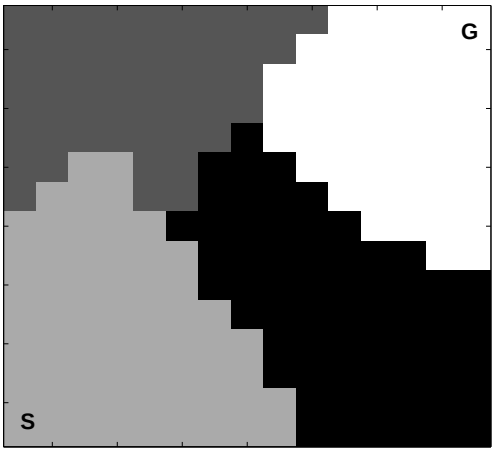

Figure 5.7: The extended Mars rover task: the rover must move from *S* to *G* while crossing different types of sand, each of which can have a different relevant structural feature.

We consider two scenarios: in the first, the agent observes the different sand types (e.g., by observing the sand color) while in the second scenario it does not. When the agent observes the sand types, it select a new representation only when it crosses the border between sand types. Otherwise, the agent selects a new representation after each ground action. We compare the two switching strategies from Section 5.4.3 on both these scenarios.

All switch methods use five different candidate context representations, each consisting of two features: the position feature and one of the five structural features, while $\mathbf{X}_{sw}$ consists only of the position feature. We compare these switch methods with having perfect information, using all structural features, and using no structural features at all. This results in the following list of methods:

- Perfect Info - Single, context-specific, representation consisting of only the position feature and the locally relevant structural feature.
- All structural features - Single representation taking into account the position feature and all structural features.
- No structural features - Single representation taking into account only the position feature.
- Switch, n-step scheme, border - Representation selection with n-step update scheme and switching only when the border between sand types is crossed.
- Switch, n-step scheme, always - Representation selection with n-step update scheme and switching after each ground action.
- Switch, Q-L scheme, border - Representation selection with Q-learning update scheme and switching only when the border between sand types is crossed.

- Switch, Q-L scheme, always - Representation selection with Q-learning update scheme and switching after each ground action.

We measure the average return over the first 1000 episodes for these seven methods. All methods use ε-greedy action selection for the external action with $\varepsilon = 0.1$ and decaying learning rates (according to Equation 5.22) with initial values of 1 and an optimized decay rate, d_{ex}. For the switch algorithms, the extra parameters are also optimized. All results are averaged over 200 independent runs and smoothed. At the start of each run, a random assignment of relevant features to sand types is made.

Table 5.12 shows the average performance of the different methods over the first 1000 episodes together with the optimal parameters, while Figure 5.8 plots the return as a function of the number of episodes for these optimal parameters. Note that the perfect info method, the method that uses all structural features, and the method using no structural features all have the same performance when compared to the regular Mars rover task (see Figure 5.6). The reason is different for each method. For the perfect info method, it does not matter which structural feature is relevant, since it uses the right one by definition. For the method using all structural features, it does not matter since the relevant is always observed and the total number of candidate representations is still five. For the method using no structural features, the performance is the same since it never uses the relevant feature.

All switch methods show a large performance improvement compared to the full representation. Surprisingly, the n-step method with switching after each ground action performs remarkably well. It yields an average reward of -123.6, while the other switch methods yield rewards between -240 to -280.[5] This result is remarkable because this method does not require the agent to distinguish the different sand types. The methods that do require this extra knowledge (the switch methods that only let the agent select a new representation when the border between sand types is crossed) perform worse.

Apparently, the combination of the n-step update scheme with a low value of n (in our case, $n = 2$) is quite powerful. This can be explained as follows. The n-step update scheme has the same advantage as the MC update scheme from Section 5.3.2: the update targets for the switch actions are not biased by the Q-values of the candidate representations. A disadvantage of an MC update or an n-step update with high values of n is that the variance of the update target is very high, since many different state-action sequences can cause the update. When n is small, the variance of the update target is smaller, yielding more accurate updates. Thus, the property of not bootstrapping from the representations Q-value seems powerful, an effect that was obscured in our previous experiments by the high variance of the MC update targets.

[5] Around episode 300, the n-step method even yields a higher return than the perfect info graph. However, note that the overall performance is worse at each point in time, since the return during the first 200 episodes is much lower.

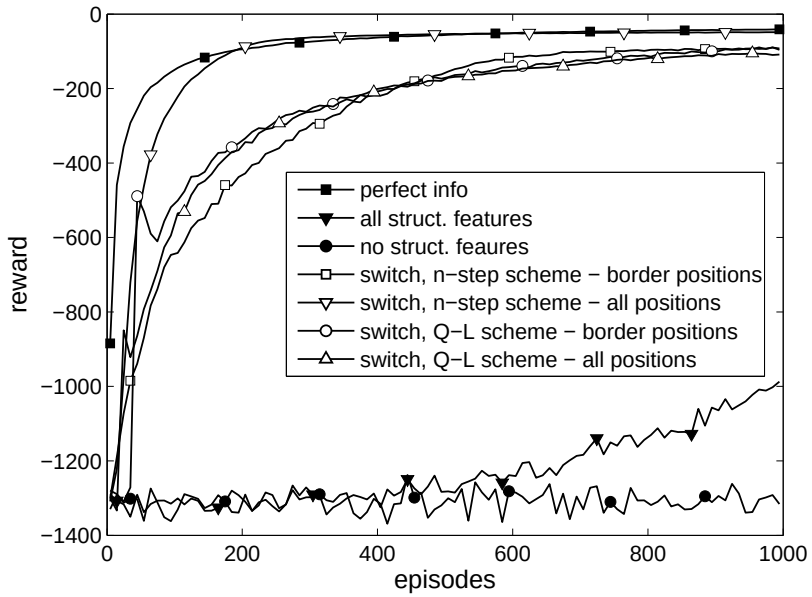

Figure 5.8: Performance as a function of number of episodes on the extended Mars rover task.

5.5 Discussion and Future Work

Taken together, the empirical results presented in Sections 5.2, 5.3, and 5.4 provide substantial evidence of the value of prior knowledge about the set of candidate representations in reinforcement learning. Furthermore, they consistently validate the benefit of using derived tasks to exploit such prior knowledge. In contextual bandit problems, MDPs, and MDPs with context-specific structure, the methods we propose perform much better than alternatives that do not exploit such prior knowledge. In addition, they often perform nearly as well as methods that are given the optimal representation in advance.

For MDPs with context-specific structure, our methods can also exploit a second form

Table 5.12: Average performance over the first 1000 episodes and optimal parameters on the extended Mars rover task.

	optimal parameters	average return	standard error
perfect info	$d_{ex} = 1.0\%$	-87.83	0.06
switch, n-step scheme, border	$\varepsilon_{sw} = 0.5$, $d_{sw} = 0.4\%$, $d_{ex} = 0.8\%$	-278.9	0.6
switch, n-step scheme, always	$\varepsilon_{sw} = 0.2$, $d_{sw} = 0.4\%$, $d_{ex} = 0.8\%$	-123.6	0.2
switch, Q-L scheme, border	$s_{\varepsilon} = 40$, $d_{sw} = 0.6\%$, $d_{ex} = 0.4\%$	-242.3	0.2
switch, Q-L scheme, always	$s_{\varepsilon} = 20$, $d_{sw} = 0.6\%$, $d_{ex} = 0.2\%$	-270.0	0.2
all struct. features	$d_{ex} = 1.0\%$	-1227	1
no struct. features	(no learning occurs)	-1309	2

of prior knowledge. In addition to knowledge about the set of candidate representations, they also rely on knowledge about which states have the same relevant features. This knowledge is represented by aggregating such states into a context, i.e., by the definition of H_1. Surprisingly, the extended Mars rover experiment presented in Section 5.4.4 shows that, when the different contexts have the same set of candidate context representations, this additional prior knowledge does not further improve performance. On the contrary, Figure 5.8 shows that the best performance is obtained by switching at all positions, i.e., ignoring the context. However, such knowledge clearly would improve performance in cases where the set of candidate representations is different in each context. For example, suppose each sand type in the extended Mars rover task had a different set of 5 candidate representations. In this case, an agent that is ignorant of the sand type would have to explore $5 \times 4 = 20$ candidate representations at each position, instead of 5.

Since the best performance in the extended Mars rover task occurs when switching at all positions, these results also demonstrate that it can be beneficial to switch representations even when the context does not change. While these experiments evaluate only MDPs with context-specific structure, the conclusion can be applied to all MDPs, since those without context-specific structure are just a special case. Consequently, we hypothesize that further performance improvements could be obtained by using the derived task presented in Section 5.4.2 in combination with the n-step update scheme (Table 5.10) on the regular Mars rover task from Section 5.3.3. In fact, performance on this task should be the same as on the extended Mars rover task, since the two tasks are actually identical from an algorithmic point of view. In both cases, the agent must learn which feature out of a set of 5 features is relevant for each position. We intend to investigate such variations in future work.

For simplicity, all of the methods considered in this chapter take a model-free approach. Perhaps the most promising direction for future work is to extend these ideas to model-based methods. In particular, models of the derived tasks presented in Sections 5.2, 5.3, and 5.4 could be learned from the agent's interactions with its environment. Dynamic programming methods could then be used to compute value functions and policies that are optimal with respect to the learned models. We hypothesize that with such model-based methods, knowledge about which states have the same relevant features would still not lead to performance improvements when the candidate context representations are the same for each context, as the size of the derived task would remain independent of H_1. We also hypothesize that switching at all positions would still lead to performance improvements, since the extra switch states would allow for better exploration.

In addition to making it possible to test these hypotheses, developing model-based methods based on derived tasks would open the door to meaningful empirical comparisons with approaches that learn DBN structure and weights. Such methods can be used to learn a DBN representation of the model given prior knowledge about the maximum degree of the DBN (Li et al., 2008; Diuk et al., 2009; Kroon and Whiteson, 2009). Thus, the advent of model-based switching methods would enable controlled experiments assessing the relative advantages and disadvantages of exploiting these different forms of prior knowledge.

5.6 Related Work

In addition to the DBN structure learning methods mentioned above, a large body of work on finding and exploiting *state abstractions* is related to the work presented in this chapter. In a planning context, state abstraction techniques can be roughly divided between exact and approximate methods. Exact methods aggregate states for which the transition and reward functions are equal (Givan et al., 2003; Boutilier et al., 2000; Ravindran and Barto, 2003). In contrast, approximate methods aggregate states for which the transition and reward functions are similar according to some metric (Dean et al., 1997; Ferns et al., 2004). These methods differ from ours in that they focus on planning and thereby assume complete knowledge of the transition dynamics, whereas we focus on learning, in which such knowledge is absent.

More closely related are other state abstraction methods that are also designed for the learning setting. For example, Chapman and Kaelbling (1991) propose a method for on-line state abstraction of states with the same reward and Q-value for each action. Similarly, McCallum (1995) describes a method that can learn to aggregate states that have the same optimal action and similar Q-values for these actions. Jong and Stone (2005) propose a method that can learn to aggregate states with the same optimal action. A common feature of these methods is the reliance on statistical methods to identify the irrelevant features. Since such approaches typically require large amounts of data, their practical application is largely limited to *transfer learning* (Taylor and Stone, 2009), where abstractions learned in one task can be used to speed learning in other, related tasks. In contrast, our methods aim to determine which representation to use on-line, within a single task. Thus, we do not rely on statistical tests but instead exploit prior knowledge about the set of candidate representations to construct a derived task that can be efficiently solved.

Finally, the use of derived tasks makes our approach similar to *temporal abstractions* such as *options* (Sutton et al., 1999), in which each option uses a different state aggregation (e.g., as in (Jong and Stone, 2005)). However, our methods are different in that the value of the terminal states of the options are not zero, but instead have a value derived from that of states at the top of the hierarchy. This avoids limiting our approach to *recursively optimal* solutions (Dietterich, 2000), i.e., those that are optimal on each subtask, but not necessarily globally optimal.

5.7 Conclusion

This chapter presented a new strategy for on-line representation selection for factored MDPs. The proposed approach addresses a special case of the structure learning problem in which prior knowledge can be used to restrict the set of candidate representations that must be considered. The problem of representation selection was formalized by defining a derived task that extends the action set with internal switch actions that select the representation to be used for external action selection. We proved this derived task is Markov or Markov with respect to relevant features under various conditions related to the type of specific features. This result enables the use of resource-efficient model-free learning

methods. In addition, we demonstrated that learning speed can be further improved by constructing parallel experience sequences corresponding to candidate representations that were not selected. These parallel sequences can be used for off-policy updating of the Q-values of these representations.

We demonstrated the validity of the approach via experiments on a contextual bandit task, an MDP task with regular candidate representations, and an MDP task with context-specific candidate representations. In all three domains, a large performance improvement was achieved by automatically discovering the best candidate representations. Furthermore, we demonstrated that our approach can automatically switch between a set of relevant features and a subset of these features and, in so doing, can perform even better than either of these individual feature sets, since doing so combines the learning speed of the small representation with the high asymptotic performance of the large representation.

CHAPTER 6

Reducing the Problem Size by Policy Space Reduction

In a realistic RL task the learning agent often has some knowledge about the task, as well as knowledge about the interpretation of some of the state features it observes. Consider for example a robot navigating through an unknown building in search for a (ground-level) power outlet to recharge its batteries. This robot knows it has to reach a specific location and will most likely have access to a number of sensors that tell it something about its (local) surroundings, like an infra-red sensor for detecting close range obstacles, and a camera sensor that, combined with certain image processing software, can recognize power outlets.

In this chapter, we present a strategy to exploit such knowledge by using *policy restrictions*. Policy restrictions remove policies from the *policy space*, the set of all policies that can be defined for a particular MDP, while keeping (at least) one optimal policy.

The main contribution of this chapter consists of the *policy restriction set*, a compact and effective way to model and exploit a wide variety of policy restrictions with value-function based reinforcement learning methods. While options (Sutton et al., 1999; Precup, 2000; Stolle and Precup, 2002) could also be used to model certain policy restrictions, the range of restrictions that can be effectively described using the policy restrictions set is much larger, allowing exploitation of new forms of prior knowledge.

Combining an MDP with prior knowledge encoded by a policy restriction set results in a derived MDP whose policy space is equal to the subset of the policy space defined by the policy restrictions. We demonstrate in this chapter that using this derived MDP can result in large performance improvements.

The remainder of this chapter is organized as follows. In Section 6.1, we describe how policy restrictions can be effectively modeled with the policy restriction set. In Section 6.2, we discuss the relation with options and shaping. In Section 6.3, we present three learning methods that use the policy restriction set as prior knowledge. In Section 6.4, we compare these 3 methods on a deterministic variant of the large maze task from Section 4.2.3. In Section 6.5 we discuss the results as well as future work. Section 6.6 contains the conclusion of this chapter.

6.1 Policy Restrictions

In this section we discuss how prior knowledge about policy restrictions can result in an improved performance, and demonstrate how subsets of the policy space can be effectively described with the policy restriction set.

6.1.1 Restrictions in the Policy Space

In many RL tasks, the goal of an agent is to find an optimal (or near-optimal) policy. An optimal policy is 'optimal' with respect to the set of all policies that can be defined for the MDP. For a general RL task a policy can take into account the complete history from the moment it was initiated up to the current state for the selection of an action. However, in case of an MDP, an agent seeking optimality can restrict its attention to the subset of *Markov policies*, which only take into account the current state when choosing an action,[1] since there is at least one optimal policy within this set. In fact, the agent can refine its search space even further by only focusing on *deterministic Markov policies*, which map each state to a single action, since there exists also at least one optimal policy within the set of deterministic Markov policies. In this section, we analyze the relation between an MDP and the corresponding set of deterministic Markov policies that can be defined for it. In this chapter, we refer to this set simply as the *policy space*, although strictly speaking the policy space is the set of *all* possible policies that can be defined for an MDP.

The *state-action space* of an MDP refers to the set of all state-action pairs. For an MDP with a finite state-action space, the policy space is also finite, although it is in general exponentially larger. Consider an MDP $M = \langle \mathbf{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ with a finite state-action space, where $\mathcal{A}$ is a function mapping each state $\mathbf{x} \in Dom(\mathbf{X})$ to a subset of the total action set A_{all}, so[2]

$$\mathcal{A} : \mathbf{X} \rightarrow \mathcal{P}(\mathcal{A}_{all}) \tag{6.1}$$

The size of the state-action space of M is:

$$\text{size state-action space} = \sum_{\mathbf{x} \in Dom(\mathbf{X})} |\mathcal{A}(\mathbf{x})| \tag{6.2}$$

On the other hand, the size of the policy space of M is:

$$\text{size policy space} = \prod_{\mathbf{x} \in Dom(\mathbf{X})} |\mathcal{A}(\mathbf{x})| \tag{6.3}$$

The prior knowledge we wish to encode in this chapter is knowledge that reduces the set of policies to consider even further, by specifying a subset of the policy space that is guaranteed to have at least one optimal policy. This is done by defining *policy restrictions*, which specify policies that are excluded from the policy space.

To illustrate the effect of policy restrictions, consider the small deterministic network shown in Figure 6.1(a). It consists of only 6 states, 2 of them being terminal states. In states x_0 and x_3 the agent can choose between two different actions; in states x_1 and x_2 only a single action is available. State x_0 is the initial state. The state-action space size if 6; the corresponding policy space, shown in Figure 6.1(b), has a size of 4. In a deterministic environment a policy can be visualized as a single trajectory through the state-action space

[1] These are the policies we consider so far in this thesis. For simplicity, we never mentioned more general policies and never used the term 'Markov policies' before. Note, however, that the behavior policy is in general a policy that depends on the complete history, since it is typically based on Q-values that are updated during learning.

[2] $\mathcal{P}(\mathcal{A}_{all})$ refers to the powerset of $\mathcal{A}_{all}$.

starting at the initial state. In general, only a fraction of the states are visited by a single policy trajectory. The policy for states that are not visited are irrelevant and therefore not shown.

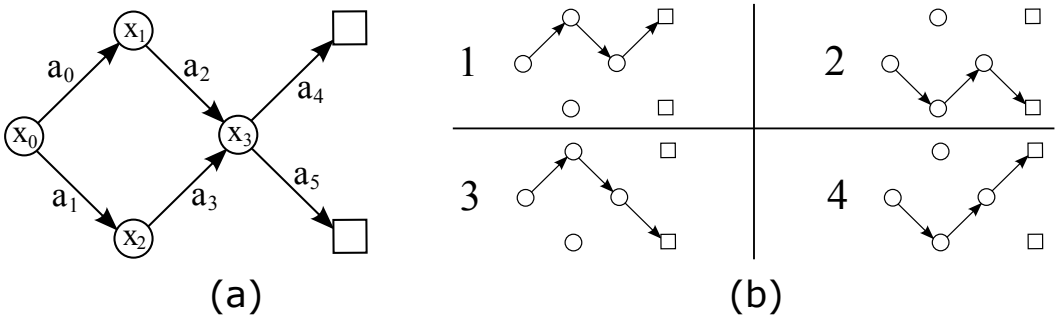

Figure 6.1: MDP network (a) and the corresponding policy space (b). In a deterministic environment a policy can be visualized as a single trajectory through the state space starting from the initial state.

Assume that the agent knows that either policy 1 or policy 4 is optimal. This knowledge can be translated into an action set reduction by removing action a_5 from the MDP network. Removing this action can be interpreted as the definition of a derived MDP whose policy space consists of only policy 1 and policy 4.

However, not all subsets of the policy space can be represented by simply removing actions. For example, if the agent knows that either policy 1 or 2 is optimal, it cannot remove any action, since all actions are contained within these two policies. To construct a derived MDP with a policy space consisting of only policies 1 and 2, the agent has to take into account the policy it followed before arriving at state x_3 when selecting an action in state x_3. This can be achieved by extending the state feature set of the original MDP with *policy features*, which can contain information about actions taken before or state feature values observed before. The action set of the derived MDP can then be defined in terms of this extended feature set. We name the resulting derived MDP a *policy-restricted MDP*.

In Figure 6.2, we show a policy-restricted MDP, whose feature set is extended with feature Y, consisting of values y_0, y_1 and y_2. Feature value y_0 corresponds with the statement that either feature value x_0 was observed at the previous timestep, or the current timestep is the initial timestep (i.e., there is no history yet). The interpretation of y_1 is that feature value x_1 was observed the previous timestep, while y_2 means feature value x_2 was observed the previous timestep.

The policy space of this derived MDP consists of only policies 1 and 2.[3] Note that the total state-action space size of this derived MDP is equal to that of the original MDP. So, while the policy space is reduced, the state-action space is still equal in size. The smaller

[3]Strictly speaking, the policy space of the derived MDP is not equal to the set consisting of policies 1 and 2, since the policies for the derived MDP are defined in terms of an extended feature set. However, there is a straightforward mapping between them.

policy space of the derived MDP has resulted however in a simpler structure for the derived MDP. With the right methods, this can be exploited and result in an improved performance.

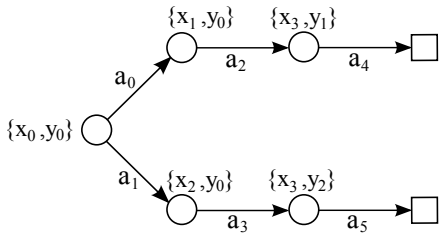

Figure 6.2: History-extended MDP corresponding to policy 1 and 2 from Figure 6.1(b).

In this section, we specified the policy restrictions by enumerating the policies from the remaining policy subset. Clearly, this is infeasible for all but the most simple MDPs. In the next section, we show how policy restriction can be compactly modeled for a general MDP using the *policy restriction set*.

6.1.2 The Policy Restriction Set

Formally, policy restrictions define an action set restriction that depends on all state-action pairs visited before. Let the state-action history h_t be the sequence of all state-action pairs visited from the start of an episode up to the current state $\mathbf{x}_t$

$$h_t = \langle \mathbf{x}_0, a_0, \mathbf{x}_1, a_1, ..., \mathbf{x}_{t-1}, a_{t-1}, \mathbf{x}_t \rangle \tag{6.4}$$

We denote by $\mathcal{H}$ the set of all possible state-action histories. In addition, let h_t^- be the same state-action history as h_t minus the last state $\mathbf{x}_t$, and let $\mathcal{H}^-$ be the set of all possible state-action histories that leave out the last state. Let $\mathcal{A}$ be the (original) state-dependent action set as defined by Equation 6.1. With these definitions, policy restrictions can be captured by a function $\mathcal{A}'$, defined as

$$\mathcal{A}' : \mathcal{H} \rightarrow \mathcal{P}(\mathcal{A}_{all}) \,, \tag{6.5}$$

with the following property:

$$\mathcal{A}'(h) \subseteq \mathcal{A}(\mathbf{x}) \qquad \text{for all } \mathbf{x} \text{ and } h \in \mathcal{H} \text{ ending in } \mathbf{x}. \tag{6.6}$$

The expressiveness of this function is extremely large. However, specifying it explicitly is infeasible in practise, since the set $\mathcal{H}$ is infinite in the general case. Therefore, instead of specifying $\mathcal{A}'$ as function of $\mathcal{H}$, we specify it as a function of $\mathbf{X}$ and $\mathbf{Y}$, an abstract feature set that only takes into account the relevant history information. We denote this function by A_Y, to distinguish it from $\mathcal{A}'$, the action set restriction as function of $\mathcal{H}$.

Formally, the feature set $\mathbf{Y}$ is an aggregation of the set $\mathcal{H}^-$. In other words, there exists a function $\mathcal{F} : \mathcal{H}^- \rightarrow \mathbf{Y}$ that maps each h^- to a value $\mathbf{y} \in Dom\{\mathbf{Y}\}$. Note that this mapping, as well as the feature set $\mathbf{Y}$, is not unique; there can be multiple ways to

aggregate $\mathcal{H}^-$, resulting in different feature sets $\mathbf{Y}$. In practise, prior knowledge will be directly translated into a relevant feature set $\mathbf{Y}$ (see Section 6.1.3 for an example), instead of first defining $\mathcal{A}'$ and then constructing a mapping function $\mathcal{F}$.

For completeness, we define the aggregation condition that should hold for the function $\mathcal{F}$. Informally, two different state-action histories h_1^- and h_2^- can be aggregated if they do not result in different action subsets now or in the future. Formally, we can define this condition as follows. Let $\langle a, b\rangle$ be a sequence concatenating sequence a and sequence b, and let h^+ be a state-action sequence $\langle \mathbf{x}_k, a_k, ..., \mathbf{x}_{k+n-1}, a_{k+n-1}, \mathbf{x}_{k+n}\rangle$, starting in an arbitrarily state $\mathbf{x}_k$ and of arbitrarily length (including length 1, i.e., $h^+ = \langle \mathbf{x}_k\rangle$). Let h_1^- and h_2^- be elements of $\mathcal{H}^-$. h_1^- and h_2^- can be aggregated, i.e., $\mathcal{F}(h_1^-) = \mathcal{F}(h_2^-)$ is allowed, if for all $\langle h_1^-, h^+\rangle \in \mathcal{H}$ the following holds: $\langle h_2^-, h^+\rangle \in \mathcal{H}$ and

$$A'(\langle h_1^-, h^+\rangle) = \mathcal{A}'(\langle h_2^-, h^+\rangle) \tag{6.7}$$

It follows from this condition that if $\mathcal{F}(h_1^-) = \mathcal{F}(h_2^-)$, for all $\langle h_1^-, \mathbf{x}, a\rangle \in \mathcal{H}^-$ the following holds: $\langle h_2^-, \mathbf{x}, a\rangle \in \mathcal{H}^-$ and

$$\mathcal{F}(\langle h_1^-, \mathbf{x}, a\rangle) = \mathcal{F}(\langle h_2^-, \mathbf{x}, a\rangle) \tag{6.8}$$

Because of Equation 6.8 a transition function $T_Y : \mathbf{X} \times \mathbf{Y} \rightarrow \mathbf{Y}$ can be defined that transforms each current state, $\mathbf{x}_t$, current policy value $\mathbf{y}_t$ and action a_t into a new policy value $\mathbf{y}_{t+1}$. With an initial value $\mathbf{y}_0$ corresponding to an empty state-action history (i.e., $\mathcal{F}(\langle\rangle) = \mathbf{y}_0$), the action set restriction $\mathcal{A}'(h_t)$ can be computed at each timestep using the *policy restriction set* $\{\mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y\}$. To summarize, its four elements are:

- $\mathbf{Y}$: a set of abstract policy features
- $\mathbf{y}_0 \in Dom(\mathbf{Y})$: the initial value of $\mathbf{Y}$.
- $\mathcal{A}_Y : \mathbf{X} \times \mathbf{Y} \rightarrow \mathcal{A}_Y \subset \mathcal{A}$: a policy-restricted action set. This encodes the actual policy space subset.
- $T_Y : \mathbf{X} \times \mathbf{Y} \times \mathcal{A}_Y \rightarrow \mathbf{Y}$: a transition function, describing the next policy feature values based on the current state feature values, the current policy feature values and the action.

The combination of the policy restriction set and the original MDP implicitly defines a policy restricted MDP, as shown in the previous section. The state space of this MDP is spanned by the feature sets $\mathbf{X}$ and $\mathbf{Y}$, hence it is a factor $|\mathbf{Y}|$ larger than the state space of the original MDP. However, the action space of this MDP, defined by $\mathcal{A}_Y$, is smaller than that of the original MDP. Therefore, the effective state-action space (i.e., the set of all state-action pairs that can be visited by the agent) of the policy restricted MDP can be smaller than that of the original MDP (in the next section we show an example of this).

The feature set $\mathbf{Y}$ can simply encode state features from the previous timestep or the previous action, but it can also encode more complicated history elements. For example, it could consist of a set of boolean statements that say something about a specific event (or events) that happened in the past. Note that while in theory all functions A' can be

transformed to a policy restriction set, not all A' will lead to a compact representation. If $|\mathbf{Y}|$ is very large, the increase in state space is also large. Because of this, policy restrictions will not in all cases result in a performance advantage.

In the next section, an example is given of a task with policy restrictions that lead to a huge reduction of the (effective) state-action space.

6.1.3 Illustrative Example

In this section we give a concrete example of a task and a policy restriction set that encodes prior knowledge about it. In Section 6.4 we empirically test the performance of the three methods presented in Section 6.3 on this example task.

Consider a variation of the large maze task (Section 4.2.3, Figure 4.8) that has only one goal state (instead of four), which is located at the lower right of the grid. In addition, assume a deterministic environment. So, the agent has four movement actions, each corresponding with a single step in one of the directions 'north', 'east', 'south' or 'west', unless the action moves the agent into a wall, in which case it remains at the same position.

Assume the agent has general knowledge of the task it faces, that is, it knows it interacts with a deterministic navigation task and that it should find the shortest path towards the goal (however, the location of the goal is unknown as well as the maze layout). In addition, assume the state feature set $\mathbf{X}$ consists of the following features:

- 1 position feature (X_{pos}), indicating the position of the agent in the 2-dimensional grid.
- 4 wall-distance features ($X_{[dir]\_wall\_dist}$) corresponding with the number of squares between the agent and the wall/border, for each direction. For example, if there is a wall just next to the agent on its east side, $X_{east\_wall\_dist} = 0$.
- 4 binary features indicating whether the agent has an unblocked view on the goal, for each direction ($X_{[dir]\_goal\_visible}$).

Using this feature set and the prior knowledge of the task, the following policy restrictions can be defined.

- The agent is not allowed to take an action that points directly towards a wall.
- The agent is not allowed to take an action that is perpendicular to its previous action (e.g., an 'east' action cannot follow a 'north' action), unless the distance to the wall in that direction increased with respect to the previous timestep (i.e, a wall-opening appeared), or if it can see the goal location along this direction (i.e., its view is not blocked by any wall).
- The agent is not allowed to take an action opposite of the action it just took (e.g., a 'south' action cannot follow a 'north' action), unless it faces a wall in that direction.

The subset of the policy space defined by these restrictions contains at least one optimal policy.

These policy restriction can be encoded using the following policy restriction set $\{\mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y\}$. $\mathbf{Y}$ contains 5 features:

- 1 previous action feature (Y_{prev_action}), indicating the action taken at the previous timestep.
- 4 previous wall distance features ($Y_{prev\_[dir]\_wall\_dist}$), indicating the distance to the wall at the previous timestep for the four compass directions.

We define $\mathbf{y}_0$, the feature values of $\mathbf{Y}$ at the initial timestep as

$$\mathbf{y}_0 = \{\text{'none'}, -1, -1, -1, -1\}\,.$$

The function T_Y follows directly from the definition of the feature set $\mathbf{Y}$:

$$\begin{aligned} y_{prev\_action,t+1} &= a_t \\ y_{prev\_[dir]\_wall\_dist,t+1} &= x_{[dir]\_wall\_dist,t} \end{aligned}$$

Finally, the function A_Y, encoding the actual policy restrictions can be defined using if-then statements. For example, it can be implemented by splitting A_Y up in four boolean functions $A_{Y,[dir]} : \mathbf{X} \times \mathbf{Y} \rightarrow \{TRUE, FALSE\}$, each defining for one compass direction whether the corresponding action is an element of the action set $A_Y(\mathbf{X}, \mathbf{Y})$. Algorithm 13 shows pseudocode for $A_{Y,east}$. The other compass directions have similar pseudocode.

Algorithm 13 $A_{Y,east} : \mathbf{X} \times \mathbf{Y} \rightarrow \{TRUE, FALSE\}$

Input: current values of features $\mathbf{X}$ and $\mathbf{Y}$
Output: *east-valid* (TRUE if 'east' action is available, FALSE otherwise)

```
if Y_prev_action = 'none' then
    east-valid = TRUE
else if Y_prev_action = 'east' then
    if X_east_wall_dist > 0 then
        east-valid = TRUE
    else
        east-valid = FALSE
else if Y_prev_action = 'west' then
    if X_west_wall_dist = 0 then
        east-valid = TRUE
    else
        east-valid = FALSE
else
    if (Y_prev_east_wall_dist > X_east_wall_dist) ∨ X_east_goal_visible then
        east-valid = TRUE
    else
        east-valid = FALSE
```

The policy restriction set discussed above is independent of the maze layout and goal location. Therefore, once it has been defined, it can be reused each time the agent encounters a similar deterministic maze task with features $X_{[dir]\_wall\_dist}$ and $X_{[dir]\_goal\_visible}$.

Note that the features $X_{[dir]\_wall\_dist}$ and $X_{[dir]\_goal\_visible}$ are redundant (as defined by Definition 11) with respect to the feature X_{pos}, that is, each location feature value corresponds with a single set of values for features $X_{[dir]\_wall\_dist}$ and $X_{[dir]\_goal\_visible}$. In addition, the features $Y_{prev\_[dir]\_wall\_dist}$ are redundant with respect to the feature set $\{X_{pos}, Y_{prev\_action}\}$. Therefore, these features do not increase the size of the state space and the Q-values of the policy-restricted MDP corresponding to this policy restriction set can be stored as function of only the feature set $\{X_{pos}, Y_{prev\_action}\}$, ignoring the other features.

In Section 6.4, we measure the performance of three different methods (presented in Section 6.3) that exploit the policy restriction set above, as well as regular Q-learning. We use three versions of this task, corresponding to resolutions 1, 3 and 7. A resolution of 1 refers to a grid of 23 by 26 squares, which is the grid size of the original task (Figure 4.8). A resolution of 3 divides the grid into 23*3 by 26*3 squares, and a resolution of 7 divides the grid into 23*7 by 26*7 squares. Figure 6.3 shows the same area around the start state for the three different resolutions. The reward received after an action depends on the resolution. For a resolution of 1, the reward is -1 for each action. For a resolution of 3 it is -1/3, and for a resolution of 7 it is -1/7, while γ equals 1 in all cases. This way, if the agent covers the same distance, it receives the same return for all three resolutions.

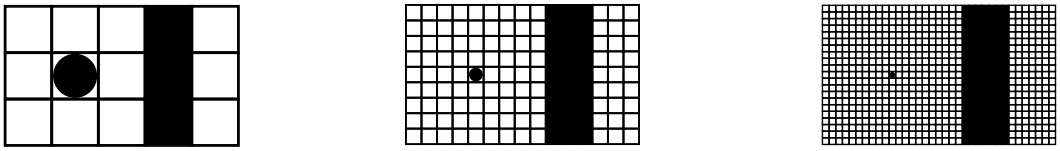

Figure 6.3: Area around the start state of the large maze task for three different resolutions. Note that the wall thickness and location is not effected by the resolution.

In the next section, we explain the advantages of the policy restricted MDP in comparison to the original MDP using the task and policy restriction set defined in this section.

6.1.4 Advantages of Policy Restrictions

There are two reasons why using the policy-restricted MDP to find the optimal policy can be advantageous: a smaller state-action space and single-action states.

To illustrate the huge state-action space reduction that can be achieved, we let a random agent (i.e., an agent that select randomly among the available actions) interact with the large maze task at resolution 3 under the specified policy restrictions (for a large number of episodes), while keeping track of the visited positions (i.e., the observed values for X_{pos}). Figure 6.4 shows the result (because of the density of the grid, we do not show individual squares). The grey lines are the positions visited by the random agent. This figure very clearly shows that, under the given policy restrictions, only a fraction of the total maze is visited by the agent. This fraction is resolution dependent: the higher the

resolution, the smaller this fraction is and hence the larger the performance benefit of the policy restrictions. Note as well that an optimal policy can be constructed, even when the behavior of the agent is restricted to the grey lines.

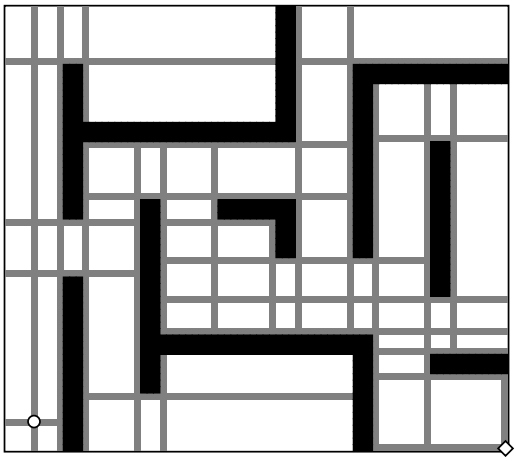

Figure 6.4: The positions an agent can reach (grey lines) when following the policy restrictions in the large maze task at resolution 3. The white circle in the lower-left corner is the start position; the white diamond in the lower-right corner is the goal position.

The second reason why using policy restrictions can result in higher performance is less obvious, but can contribute just as much. To understand why, note that a single state from the original MDP is in general mapped to multiple states in the policy-restricted MDP. For example, state $\mathbf{x}$ of the original MDP can be mapped to states $\{\mathbf{x}, \mathbf{y}_1\}$ and $\{\mathbf{x}, \mathbf{y}_2\}$ of the policy-restricted MDP. By definition, the action sets of these states are subsets of the action set of the original state: $\mathcal{A}_Y(\mathbf{x}, \mathbf{y}_1) \subseteq \mathcal{A}(\mathbf{x})$ and $\mathcal{A}_Y(\mathbf{x}, \mathbf{y}_2) \subseteq \mathcal{A}(\mathbf{x})$. Because of this action subset, it can occur that the action set of states in the policy restricted MDP contain only a single action. We name such states *single-action states*. Single-action states are special, since they do not require Q-values. Obviously, for action selection this is not required, but also its use in update targets for other state-action pairs can be avoided by using multi-step updates whenever single-action states are encountered (see Section 6.3.4 for a detailed explanation). So, while formally single-action states do not reduce the state-action space, they do reduce the number of Q-values that need to be learned. This reduction in Q-values results in improved efficiency, or, under given space and time constraints, improved performance, as we demonstrate in Section 6.4.

For the large maze task, the policy-restricted MDP corresponding to the restrictions described in the previous section contains a large number of these single-action states. In fact, all states except those corresponding with locations where the grey lines in Figure X cross are single-action states. Therefore, a huge reduction in the required Q-values is obtained. In fact, the number of these cross locations is resolution independent and hence the number of required Q-values as well. So, while the number of required Q-values of the original MDP increases quadratically with the resolution, this number remains the same

when using policy restrictions. This results in a huge performance advantage for methods using the policy restriction set.

This clearly demonstrates the advantage of policy restrictions compared to using no restrictions at all, but how do policy restrictions compare against plain action restrictions?[4] In other words, what is the advantage of using a function $\mathcal{A}' : \mathcal{H} \rightarrow \mathcal{P}(\mathcal{A}_{all})$ for defining restrictions compared to using a function $\mathcal{A}' : \mathbf{X} \rightarrow \mathcal{P}(\mathcal{A}_{all})$? Clearly, action restrictions are a special case of policy restrictions, so some knowledge can be represented by it. But how much of the performance advantage would be lost if only the prior knowledge that can be represented by action restrictions would be exploited? By analyzing the relation between policy restrictions and action restrictions in more detail, three types of prior knowledge can be distinguished.

First, there is the knowledge that can also be represented with action restrictions. For this type of knowledge, application of it to the MDP does not lead to different action sets for the same state. An example is the knowledge that the agent should not take an action that points directly towards a wall.

Second, there is knowledge that can never be represented by action restrictions, since application of it to the MDP leads to different action sets for the same state (depending of the state-action history). An example is the knowledge that the agent should not take an action opposite of the action it just took. This knowledge does not prohibit an action under all circumstances, but only for certain state-action histories. Hence, it cannot be represented by action restrictions. If on the large maze task action restrictions would be used instead of policy restrictions, all states along the grey lines would be 'multiple-action states' and the large performance benefit due to single-action states would be lost, as well as the property of maze tasks that the number of required Q-values is resolution independent.

The third type of knowledge consists of knowledge that cannot be represented by action restrictions, however, application of it to the MDP does *not* lead to different action sets for the same state. Therefore, there exists an action set restriction function that is equivalent to this knowledge in the sense that they share the same policy space. The problem is that the available prior knowledge cannot be directly translated into this function. As an example, consider a maze task, where the wall openings have a width of only 1 square and assume we want to encode the knowledge that the agent should not take actions pointing towards a wall as well the knowledge that the agent should not take an action that is perpendicular to its previous action, unless the distance to the wall in that direction increased with respect to the previous timestep. In this case, states corresponding to locations where wall openings are visible (along one of the four movement directions) have always the same action set. However, these locations are not part of the prior knowledge, and therefore the action set restriction function cannot be specified in advance.

While this third type of prior knowledge cannot be directly translated into action restrictions, the action restrictions that are equivalent to this knowledge can be learned from environment interaction. Although learning these action restrictions is more complicated than the simple example above suggests. In Section 6.3.3 we present a method that can

[4] In principle, policy restrictions can also be viewed as action restrictions (conditioned on the state-action history). However, in this chapter we use the term 'action restrictions' exclusively to indicate restrictions on the state-dependent action set.

learn these action restrictions (P-PR, Algorithm 16) and in the Section 6.4 we empirically compare it against methods that rely on policy restrictions on the large maze task described in Section 6.1.3.

In the next section, we discuss how the policy-restriction set differs from options and shaping, two other methods that can control the behavior of the agent.

6.2 Related Work

An alternative way to achieve a reduction of the policy space is by replacing primitive actions by options (Sutton et al., 1999; Precup, 2000; Stolle and Precup, 2002). However, the range of restrictions that can be effectively described using the policy restrictions set is much larger.

The main reason is that an option represents only a single policy, while a policy restriction set represents a subset of the policy space. Although any subset of the policy space can in theory be represented by a set of options (enumerating every single policy), this clearly is a very inefficient way to model such a policy subset and infeasible for all but the most simple MDPs.

Extensions of options, where the policy of the option is not fully specified but has to be learned given an option dependent action set, effectively also defines a set of options. However, the option action set only depends on the current state, limiting the set of policies that can be represented.

With the policy restriction set we have extended the policy subset that can be effectively modeled, allowing for new types of prior knowledge to be exploited.

There is also a relation between using policy restrictions and reward shaping (Ng et al., 1999; Wiewiora, 2003; Konidaris and Barto, 2006; Grzes and Kudenko, 2009; Snel and Whiteson, 2010, 2011), a strategy in which additional rewards are supplied to the agent to guide its learning process. Both methods have in common that they aim to improve the convergence speed of a method, while preserving the optimal policy. For shaping, optimal policies are preserved, when potential-based shaping rewards are used (Ng et al., 1999).

While both approaches have essentially the same goal, they differ in how they try to reach this goal. With the policy restriction approach certain actions are never taken, reducing the state-action space. On the hand, with shaping, the additional shaping reward tries to steer the exploration in the right direction. However, to guarantee optimality, still the full state-action space has to be explored.

They also exploit different prior knowledge. The policy restriction approach requires prior knowledge about suboptimal policy behavior. This concerns usually very local behavior, like ‘never take an action opposite of the action at the previous timestep’. In contrast, shaping usually requires knowledge about global (sub)goals, for example, the agent could receive an additional reward if the distance towards the goal is decreased in some navigational task.

In conclusion, shaping and policy restrictions can be considered orthogonal approaches to improve the converge rate. In fact, they can be combined very easily. Note that shaping rewards only effect the reward function; the policy space remains the same. Therefore, the

policy restriction subset defined for the original MDP can just as easily be applied to the derived MDP based on the modified reward function.

6.3 Methods

In this section, we present three methods that exploit the policy restrictions defined by the policy restriction set. The first two methods, PR and A-PR, obey the policy restrictions when interacting with the environment. The third method, P-PR, learns action restrictions using the prior knowledge captured by the policy restriction set.

6.3.1 Q-learning with Policy Restrictions (PR)

The policy restriction set $\langle \mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y \rangle$ implicitly defines a policy-restricted MDP. The agent can interact with this derived MDP by taking actions in the original MDP, observing the next state feature values, $\mathbf{x}'$, and combining this with $\mathbf{y}'$, the policy feature values resulting from the transition function T_Y, to form the new state $(\mathbf{x}', \mathbf{y}')$ of the policy-restricted MDP. Algorithm 14 shows pseudo-code for a Q-learning implementation based on the policy-extended MDP. Note that the learned Q-values correspond with the Q-values from the policy-restricted MDP.

Algorithm 14 Q-learning with Policy Restrictions (PR)

define $\langle \mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y \rangle$
initialize $Q(\mathbf{x}, \mathbf{y}, a)$ arbitrarily for all $\mathbf{x}, \mathbf{y}, a$
loop {over episodes}
  initialize $\mathbf{x}, \mathbf{y}$
  while $\mathbf{x}$ not terminal **do**
    select action $a \in \mathcal{A}_Y(\mathbf{x}, \mathbf{y})$, based on $Q(\mathbf{x}, \mathbf{y}, \cdot)$
    take action a, observe $\mathbf{x}'$ and r
    $\mathbf{y}' \leftarrow T_Y(\mathbf{x}, \mathbf{y}, a)$
    $Q(\mathbf{x}, \mathbf{y}, a) \leftarrow (1 - \alpha)\, Q(\mathbf{x}, \mathbf{y}, a) + [r + \gamma \max_{a' \in \mathcal{A}_Y(\mathbf{x}', \mathbf{y}')} Q(\mathbf{x}', \mathbf{y}', a')]$
    $\mathbf{x} \leftarrow \mathbf{x}', \mathbf{y} \leftarrow \mathbf{y}'$

This implementation ignores the fact that the next state is partly known, since $\mathbf{y}'$ can be computed before the action is taken. In the next section, we present a variation of Algorithm 14 that exploits this.

6.3.2 Q-learning with Policy Restrictions and Aggregation (A-PR)

In general, different mappings $\mathcal{F}$ can be defined, resulting in different feature sets $\mathbf{Y}$. But even with the most compact aggregation of state-action histories, the state-action space of the policy-restricted MDP can be larger than that of the original MDP. Although the size of the state-action space is not the only factor that determines the performance, the number of Q-values that need to be learned can often be reduced in case of a policy-restricted MDP,

without changing its policy space. In this section, we present a variation of Algorithm 14 that can achieve this.

The variation exploits the fact that for a derived MDP constructed from a policy restriction set, in contrast to a regular MDP, the next state is partially known in advance. Specifically, $\mathbf{y}'$ is known in advance, since it is determined by the current state $(\mathbf{x}, \mathbf{y})$, the action a selected for the current state and the transition function T_Y, which is part of the prior knowledge. Knowledge of $\mathbf{y}'$ can be exploited by aggregating state-action pairs, that is, by using a single Q-value table entry for multiple state-action pairs. Note that this is different from regular state abstraction, which usually involves aggregation of states that share (approximately) the same reward and transition functions for all their actions into a single abstract state. State-action aggregation generalizes state aggregation in that it allows aggregation even if only a single action of some state shares the same reward and transition function with a different state-action pair.

To see how state-action aggregation works, consider once more the MDP from Figure 6.1(a) and assume we want a policy-restricted MDP whose policy space consists of policies 1, 2 and 3 from Figure 6.1(b). In Figure 6.5 such a policy-restricted MDP is shown. It contains 7 state-action pairs, one more than the original MDP. States $\{x_3, y_0\}$ and $\{x_3, y_1\}$ cannot be aggregated since the available actions are different for the two states. However, action a_5 leads to exactly the same state, $\{x_T, y_0\}$, and yields the same reward, independent if its taken from state $\{x_3, y_0\}$ or state $\{x_3, y_1\}$. Hence, their optimal Q-value is the same. Moreover, that this is the case, can be deduced before the actions are taken. The reward is the same, since it is generated by the original MDP and $\mathbf{x}$ is the same for the two states. For the same reason, $\mathbf{x}'$ is the same, so the next states can only differ in their policy feature values. However, the agent can check this since it knows T_Y, yielding y_0 for (x_3, y_0, a_5) as well as (x_3, y_1, a_5). So, the agent knows the reward and transition of a_5 is the same for states $\{x_3, y_0\}$ and $\{x_3, y_1\}$. More generally, if action a is available in states $\{\mathbf{x}, \mathbf{y}_1\}$ and $\{\mathbf{x}, \mathbf{y}_2\}$, they share the same reward and transition function if $T_Y(\mathbf{x}, \mathbf{y}_1, a) = T_Y(\mathbf{x}, \mathbf{y}_2, a)$. These state-action pairs can be aggregated by storing the Q-values of the policy-extended MDP as function of $\mathbf{x}$, a and $\mathbf{y}' = T_Y(\mathbf{x}, \mathbf{y})$.

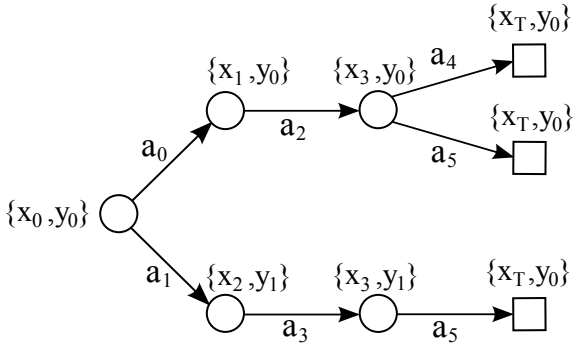

Figure 6.5: Policy-restricted MDP corresponding to policy 1, 2 and 3 from Figure 6.1(b).

Applying this type of abstraction to the policy-extended MDP shown in Figure 6.5

reduces the state-action space size to 6, the same size as the original state-action space. Algorithm 15 shows pseudocode for Q-learning based on a policy-restricted MDP with state-action aggregation.

Algorithm 15 Q-learning with State-Action Aggregated Policy Restrictions (A-PR)

define $\langle \mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y \rangle$
initialize $Q(\mathbf{x}, \mathbf{y}, a)$ arbitrarily for all $\mathbf{x}, \mathbf{y}', a$
loop {over episodes}
  initialize $\mathbf{x}, \mathbf{y}$
  while $\mathbf{x}$ not terminal **do**
    select action $a \in \mathcal{A}_Y(\mathbf{x}, \mathbf{y})$, based on $Q(\mathbf{x}, T_Y(\mathbf{x}, \mathbf{y}, \cdot), \cdot)$
    take action a, observe $\mathbf{x}'$ and r
    $\mathbf{y}' \leftarrow T_Y(\mathbf{x}, \mathbf{y}, a)$
    $Q(\mathbf{x}, \mathbf{y}', a) \leftarrow (1-\alpha) Q(\mathbf{x}, \mathbf{y}', a) +$
      $[r + \gamma \max_{a' \in \mathcal{A}_Y(\mathbf{x}', \mathbf{y}')} Q(\mathbf{x}', T_Y(\mathbf{x}', \mathbf{y}', a'), a')]$
    $\mathbf{x} \leftarrow \mathbf{x}', \mathbf{y} \leftarrow \mathbf{y}'$

6.3.3 Q-learning with Projected Policy Restrictions (P-PR)

Even when a policy-restricted MDP with state-action aggregation is used, its state-action space size can be larger than that of the original MDP. In this section, we present a method that maps the policy restriction set to a reduced action set for the original MDP. In general, the resulting state-action space is smaller than that of the policy-restricted MDP (and of the original MDP), however, the corresponding policy space is in general larger than the policy space of the policy-restricted MDP.

As an example, consider the network shown in Figure 6.6(a). To represent the two policies shown in Figure 6.6(b), the policy-restricted MDP shown in Figure 6.7 is used. This MDP has a total of 8 state-action pairs, 1 more than the original MDP, since action a_4 is present twice: it can be taken from state $\{x_3, y_1\}$ as well as from state $\{x_3, y_2\}$. In this case, state-action pairs (x_3, y_1, a_4) and (x_3, y_2, a_4) cannot be aggregated, since the Q-value of a_4 can be different for the two policies because the action taken after a_4 is different and hence the return can be different as well.

To avoid that the state-space size grows with respect to the original size when applying policy restrictions, instead of using the policy-restricted MDP, the agent can choose to learn the projection of the action set $\mathcal{A}_Y$ onto the original state feature set $\mathbf{X}$:

$$\mathcal{A}_X(\mathbf{x}) = \bigcup_{\mathbf{y} \in \mathcal{Y}(\mathbf{x})} \mathcal{A}_Y(\mathbf{x}, \mathbf{y}) \tag{6.9}$$

where $\mathcal{Y}(\mathbf{x})$ is the set of all policy feature values $\mathbf{y}$ that $\mathbf{x}$ is associated with in the policy-restricted MDP. For the policy-restricted MDP of Figure 6.7, $\mathcal{Y}(x_0) = \mathcal{Y}(x_1) = \mathcal{Y}(x_2) = \{y_0\}$, while $\mathcal{Y}(x_3) = \mathcal{Y}(x_4) = \{y_1, y_2\}$. $\mathcal{A}_X$ is in this case equal to $\mathcal{A}$, the action set of the original MDP, so no reduction of the state-action space occurs. However, in general, the state-action space can be reduced considerably. The state-action space of the derived

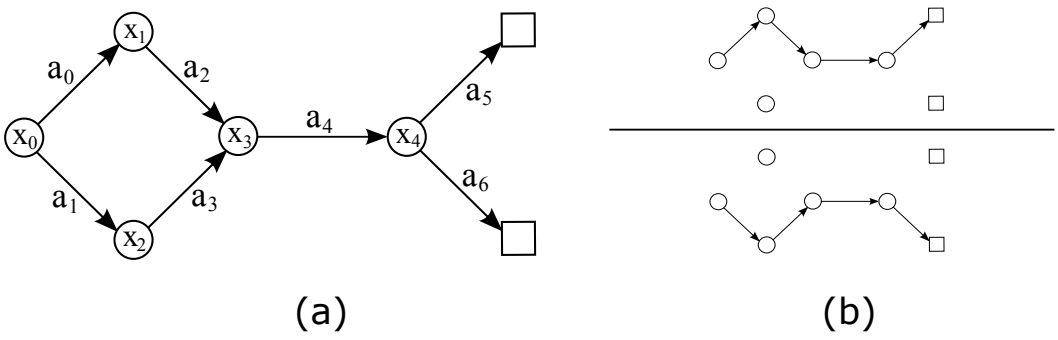

Figure 6.6: MDP network (a) and a subset of its policy space (b).

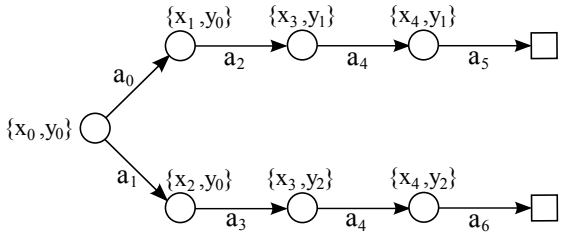

Figure 6.7: Policy-restricted MDP corresponding to the two policies from Figure 6.6(b).

MDP based on $\mathcal{A}_X$ is never larger than the original state-action space. The downside is, that the policy space of this derived MDP is, in general, larger than defined by the policy restriction set.

The agent cannot determine $\mathcal{A}_X(\mathbf{x})$ directly from its policy restriction set, since it does not know $\mathcal{Y}(\mathbf{x})$ in advance. Instead, $A_X(\mathbf{x})$ has to be constructed iteratively. Initially $\mathcal{A}_X(\mathbf{x}) = \emptyset$. Each time a new policy feature $\mathbf{y}$ is determined for state $\mathbf{x}$, the action set $\mathcal{A}_X(\mathbf{x})$ is expanded with $\mathcal{A}_Y(\mathbf{x}, \mathbf{y})$.

By determining $\mathcal{A}_X \subset \mathcal{A}$ the agent can interact with a derived MDP with a smaller state-action space than the original MDP. However, since the action set $\mathcal{A}_X$ is expanded over time, special care has to be taken to ensure proper exploration. In a deterministic environment, a very powerful exploration scheme is using optimistic initialization of Q-values combined with a greedy behavior policy. For this scheme to work, the Q-value of each state-action pair should be an overestimate of its optimal Q-value at all times. With a regular deterministic MDP network, using optimistic initial values is enough to ensure this condition holds. However, in the case of an expanding action set, the value of a state can increase if extra actions become available in this state. An action that was updated with a state value based on a smaller action set can be an underestimate with respect to the new state value. To avoid these underestimates, the Q-value of a state-action pair $(\mathbf{x}, a)$ needs to be re-initialized to an optimistic value if a new $\mathbf{y}'$ is determined for $(\mathbf{x}, a)$, since this could result in an expanded action set of the next state, potentially increasing the Q-value

of $(\mathbf{x}, a)$.[5]

Algorithm 16 shows pseudo-code of an implementation based on projected policy restrictions (P-PR). Note that, instead of storing $\mathcal{Y}(\mathbf{x})$ for each state, $\mathcal{Y}'(\mathbf{x}, a)$ is stored for each state-action pair, which is the set of all policy feature value sets $\mathbf{y}'$ that have so far been encountered for $(\mathbf{x}, a)$. If a new $\mathbf{y}'$ is observed for $(\mathbf{x}, a)$, the Q-value of $(\mathbf{x}, a)$ is re-initialized and $\mathcal{Y}_{new}$ is set to 'true', indicating $\mathcal{Y}'(\mathbf{x}, a)$ has been expanded and therefore, the next time this state-action pair is taken, $\mathcal{A}_X$ of the next state should be updated.

Algorithm 16 Q-learning with Projected Policy Restrictions (P-PR)

define $\langle \mathbf{Y}, \mathbf{y}_0, \mathcal{A}_Y, T_Y \rangle$, Q_{init}, x_0
initialize $\mathcal{A}(\mathbf{x}) \leftarrow \emptyset$, for all $\mathbf{x}$; $Q(\mathbf{x}, a) \leftarrow Q_{init}$, $\mathcal{Y}'(\mathbf{x}, a) \leftarrow \emptyset$ for all $\mathbf{x}$ and a
$\mathcal{Y}'(\mathbf{x}_0, a) \leftarrow T_Y(\mathbf{x}_0, \mathbf{y}_0, a)$, $\mathcal{Y}'_{new}(\mathbf{x}_0, a) \leftarrow true$ for all $a \in \mathcal{A}_Y(\mathbf{x}_0, \mathbf{y}_0)$
$\mathcal{A}(\mathbf{x}_0) \leftarrow \mathcal{A}_Y(\mathbf{x}_0, \mathbf{y}_0)$
loop {over episodes}
  $\mathbf{x} \leftarrow \mathbf{x}_0$, $\mathbf{y} \leftarrow \mathbf{y}_0$
  while $\mathbf{x}$ not terminal **do**
    select action $a \in \mathcal{A}(\mathbf{x})$, based on $Q(\mathbf{x}, \cdot)$
    take action a, observe $\mathbf{x}'$ and r
    if $\mathcal{Y}'_{new}(\mathbf{x}, a) = true$ **then**
      for all $\mathbf{y}' \in \mathcal{Y}'(\mathbf{x}, a)$ **do**
        $\mathcal{A}(\mathbf{x}') \leftarrow \mathcal{A}(\mathbf{x}') \cup \mathcal{A}_Y(\mathbf{x}', \mathbf{y}')$
        for all $\bar{a} \in \mathcal{A}_Y(\mathbf{x}', \mathbf{y}')$ **do**
          $\mathbf{y}'' \leftarrow T_Y(\mathbf{x}', \mathbf{y}', \bar{a})$
          if $\mathbf{y}'' \notin \mathcal{Y}'(\mathbf{x}', \bar{a})$ **then**
            $\mathcal{Y}'(\mathbf{x}', \bar{a}) \leftarrow \mathcal{Y}'(\mathbf{x}', \bar{a}) \cup \mathbf{y}''$
            $\mathcal{Y}'_{new}(\mathbf{x}', \bar{a}) \leftarrow true$; $Q(\mathbf{x}', \bar{a}) \leftarrow Q_{init}$
      $\mathcal{Y}'_{new}(\mathbf{x}, a) \leftarrow false$
    $Q(\mathbf{x}, a) \leftarrow (1 - \alpha)\, Q(\mathbf{x}, a) + [r + \gamma \max_{a' \in \mathcal{A}(\mathbf{x}')} Q(\mathbf{x}', a')]$
    $\mathbf{x} \leftarrow \mathbf{x}'$, $\mathbf{y} \leftarrow \mathbf{y}'$

6.3.4 Multi-Step Variants (PR$^+$, A-PR$^+$, P-PR$^+$)

The methods PR, A-PR and P-PR all use at their core an update derived from the single-step Q-learning update. For state-action pair (s_0, a_0), resulting in reward r_1 and next state s_1 , this update has the form

$$Q(\mathbf{x}_0, a_0) \leftarrow (1 - \alpha) Q(\mathbf{x}_0, a_0) + \alpha\, \upsilon\,,$$

with

$$\upsilon = r_1 + \gamma \max_{a'} Q(\mathbf{x}_1, a')\,. \tag{6.10}$$

[5] Technically, this assures $Q(\mathbf{x}, a) \geq Q(\mathbf{x}, \mathbf{y}, a)$ for all $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$, which guarantees the policy after convergence, when applying a greedy behavior policy, is at least as good as the optimal policy of the policy-restricted MDP. If the policy space of the policy-restricted MDP contains the optimal policy of the original MDP, this is the policy found after convergence.

A policy-restricted MDP can contain a lot of *single-action states*, i.e., states whose action set consists of only a single action. Clearly, for action selection, such state-action pairs do not require a Q-value. In fact, by using multi-step updates (i.e., updates based on the n-step return) whenever such single-action states are encountered, their Q-values are also not required for bootstrapping other Q-values, and hence can be ignored altogether. The advantage of this is faster information propagation and increased time efficiency, since single-action states do not require Q-value updates.

If action a_0 is taken in (multi-action) state $\mathbf{x}_0$ and followed by single-action states $\mathbf{x}_1, ..\mathbf{x}_{n-1}$ before multi-action state $\mathbf{x}_n$ is reached, the multi-step update target υ_n for $(\mathbf{x}_0, a_0)$ that ignores the Q-values of these single-action states is:

$$\upsilon_n = \sum_{k=0}^{n-1} \gamma^k r_{k+1} + \gamma^n \max_{a'} Q(\mathbf{x}_n, a') \tag{6.11}$$

where r_{k+1} is the reward following state-action pair $(\mathbf{x}_k, a_k)$. Note that for $n = 1$ (i.e., when $(\mathbf{x}_0, a_0)$ is followed by a multi-action state) Equation 6.11 reduces to Equation 6.10.

We indicate the variants of PR, A-PR and P-PR that use multi-step updates whenever single-action states are encountered by a '+' superscript, i.e., by PR$^+$, A-PR$^+$ and P-PR$^+$, respectively. When an MDP contains no single-action states, these multi-step variants will compute exactly the same Q-values as their regular counterparts. When an MDP does have single-action states, the multi-step variants will have in general a higher performance at equal or less computational cost.

6.4 Empirical Results

In this section, we compare the methods PR, A-PR and P-PR on the variation of the large maze and with the policy restriction set discussed in Section 6.1.3. For reference, we also compare against regular Q-learning.

We start by measuring the average return over the first 50 episodes of PR, A-PR, P-PR as well as regular Q-learning on the large maze task at resolution 1. Besides the average return, we measure the number of state-actions pairs whose Q-value got updated (by measuring, at the end of the 50 episodes, the number of Q-values with a value different than the initial value). Each method uses a greedy behavior policy with optimistically initialized Q-values of -0.01. The learning rate has a fixed value of 1. Results are averaged over 1000 independent runs.

Table 6.1 shows the average return over the 50 episodes, and the number of updated Q-values. The standard error on the average return value is not shown, but it is lower than 1 for all four methods. Figure 6.8 shows the return as function of the episode number. As expected, all three methods based on policy restrictions perform substantially better than regular Q-learning. A-PR outperforms PR. This can be expected, since they use the same policy-restricted MDP, but A-PR exploits additional structure resulting in a reduced state-action space size. A-PR and P-PR have the same number of updated Q-values, but P-PR has a better performance. It is hard to explain exactly why P-PR has a better performance, but that they have a different performance is not surprising, since they have a different

behavior policy (A-PR uses a greedy policy with respect to $\mathcal{A}_Y$, while P-PR uses a greedy policy with respect to $\mathcal{A}_X$). Apparently, in this case, the exploration performed by P-PR is more effective.

	resolution 1	
	average return	Q-values updated
Q-learning	-797.5	2008
PR	-350.8	797
A-PR	322.5	675
P-PR	-302.8	675

Table 6.1: Average return, standard error, and the number of updated Q-values for regular Q-learning and the policy restricted methods on the large maze task at resolution 1.

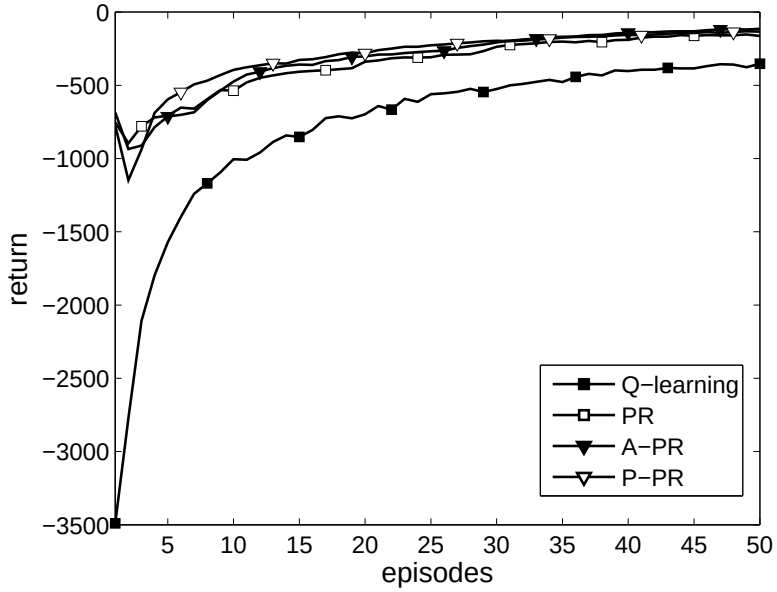

Figure 6.8: Performance of Q-learning and the policy restriction methods on the large maze task at resolution 1 for the first 50 episodes.

Based on these first results, it appears there is no advantage in trying to achieve a small policy space, since the best performance is obtained by P-PR, which aims to reduce the state-action space size over the policy space size. However, the policy-restricted MDP contains a lot of single-action states. Therefore, the performance of PR and A-PR (methods that learn Q-values for the policy-restricted MDP) improves substantially, when multi-step versions of these methods are used.

In the next experiment we apply multi-step updates to regular Q-learning as well as PR, A-PR and P-PR and compare their performance on the large maze task at resolutions

1, 3 and 7. We average the results again over 1000 independent runs. Table 6.2 shows the average return over the first 50 episodes as well as the number of state-action pairs that got their Q-value updated. Figure 6.9 shows the return as function of the episode number at resolution 3. The '+' subscript that is added to the method names indicates that multi-step updates are applied . The standard error on the average return value is lower than 1 for PR^+, P-PR^+ and A-PR^+ at all three resolutions. For regular Q-learning it is lower than 10 at all three resolutions.

	resolution 1		resolution 3		resolution 7	
	average return	Q-values updated	average return	Q-values updated	average return	Q-values updated
Q-learning$^+$	-796.7	2008	-2693.2	18104	-6606.8	98584
PR^+	-132.8	209	-137.1	214	-138.0	214
A-PR^+	-118.1	179	-121.7	184	-122.2	184
P-PR^+	-303.2	643	-669.7	2091	-1234.6	4987

Table 6.2: Average return over the first 50 episodes as well as the number of state-action pairs that got their Q-value updated on the large maze task at resolutions 1, 3 and 7. The '+' subscript indicates that multi-step updates are applied when single-action states are encountered.

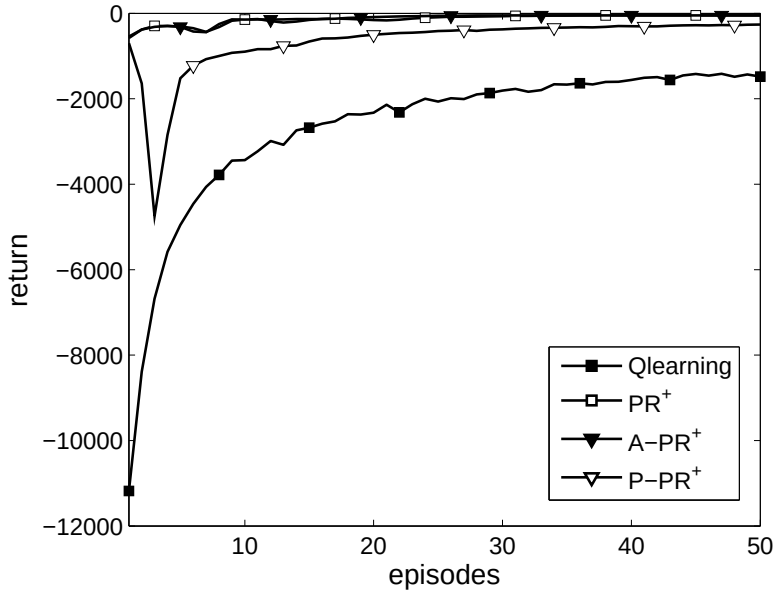

Figure 6.9: Performance of Q-learning and the policy restricted methods (with multi-step updates) on the large maze task at resolution 3.

The most striking result from this experiment lies in the number of state-action pairs that got their Q-values updated. For regular Q-learning this number scales quadratically

(by approximation) with the resolution. For P-PR$^+$ it scales linearly, and for PR$^+$ and A-PR$^+$ the number of different state-action pairs visited is approximately independent from the resolution. The huge difference in updated Q-values is reflected in the performance. At resolution 7, the performance of PR$^+$ and A-PR$^+$ is a factor 10 better than P-PR$^+$ and close to a factor 60 better than regular Q-learning.

This result can be explained by examination of Figure 6.4. The policy restrictions bound the agent to certain lines. If the resolution is increased, the number of states on such a line increases, but the 'thickness' of a line remains one square. Therefore, while the total state-action space depends quadratically on the resolution, the subset of the state-action space defined by the policy restrictions depends linearly on the resolution. This explains why the number of updated Q-values scales linearly with the resolution for P-PR$^+$. The multi-step implementation results in a slightly decreased number (643 instead of 675 at resolution 1), but this causes no significant performance increase.

In contrast, at resolution 1, for PR$^+$ and A-PR$^+$ a huge difference in updated state-action pairs occurs (for A-PR$^+$ 179 instead of 675). The reason is that along the trajectory lines of Figure 6.4 only single-state action pairs occur in the policy-restricted MDP, except when a line intersects another line. In other words, using the policy-restricted MDP, the agent only needs to make a choice between different actions at line intersections. The number of line intersections is resolution independent, and therefore the number of updated Q-values, as well as the performance, is independent of the resolution. The small performance differences that do occur for PR$^+$ and A-PR$^+$ across the different resolutions are due to subtle task differences caused by the resolution difference. For example, the distance the agent needs to travel to go from one wall to the opposite wall increases slightly when a higher resolution is used (basically, the agents size decreases with higher resolution, so the agent can move closer with its center point to a wall).

For the large maze task, the reduction in the number of Q-values achieved by A-PR with respect to PR is only small. Therefore, the performance difference of A-PR and PR is also only small, although A-PR consistently outperforms PR with 5% - 10%.

6.5 Discussion and Future Work

The large maze experiments (Table 6.1 and Table 6.2) clearly demonstrate the power of policy restrictions. The performance improvement of a factor 60 for A-PR$^+$ compared to regular Q-learning speaks for itself. In addition, it demonstrates the generality of the knowledge that is exploited: the policy restriction set only needs to be defined once, and can then be applied to any maze task, independent of its size or wall locations.

A performance advantage of a policy-restricted method over regular Q-learning can have two causes. First, the state-action space size of the derived MDP the agent interacts with can be smaller. Second, the derived MDP can contain single-action states, which can result in a higher efficiency, or, alternatively, under given space and time constraints, a higher performance. In our experiments, we used multi-step versions of the methods to exploit these single-action states.

The experiments show a consistent performance advantage of A-PR over PR (see Table

6.1, Table 6.2). This performance advantage can be fully attributed to the smaller state-action space of A-PR, since apart from that the methods are equal. They use, for example, exactly the same policy space subset. The experiments demonstrate that the size of the performance difference depends on the state-action space reduction that is achieved by the state-action aggregation.

The relative performance of P-PR with respect to A-PR and PR depends on a trade-off between two opposite effects. On the one hand, the state-action space of P-PR is in general smaller, causing an advantage over A-PR and PR; on the other hand, the structure of the derived MDP for P-PR is in general more complex than the structure of the policy-restricted MDP, causing a disadvantage. In the second large maze experiment (Table 6.2, Figure 6.9), multi-step updates are employed to exploit single-action states. While these updates do not decrease the state-action pairs visited by the agent, they do reduce the number of Q-values that need to be learned, resulting in much more effective learning. In fact, the number of Q-values that need to be learned for A-PR and PR in this case is (approximately) invariant with respect to the resolution, causing a huge performance advantage over P-PR and PR.

For the experiments performed, there is no clear case where P-PR substantially outperformed A-PR. However, since they are based on different principles, it is very likely that such tasks exist. For example, if the derived MDP does not contain any single-action states.

This observation also suggests that for certain tasks, methods might be preferred that trade-off the principles behind A-PR and P-PR in a different way. Currently, A-PR aims to create a derived MDP with the smallest policy space, whereas the goal of P-PR is to create the smallest state-action space. Crossovers between these two methods can be created that aim for a more balanced trade-off of these two goals.

An additional venue for future work is to combine the approaches with best-match learning. While in principle there is no reason why best-match learning could not be applied, it is not immediately clear which best-match methods work best. For example, the prioritized sweeping heuristic might cause problems in the case of state-action aggregation.

Finally, the obvious extension is to stochastic environments. The developed formalism and methods can be applied without adaptation to a stochastic environment. However, constructing a useful policy restriction set might be less obvious. In addition, while the policy restricted MDP will have a simpler structure, it might still be too complex to exploit this.

6.6 Conclusion

In this chapter, we presented a formalism to efficiently encode very general knowledge about suboptimal behavior that cannot be represented by a plain action set reduction. This prior knowledge can be interpreted as the removal of policies from the policy space corresponding to an MDP. We presented three value-function methods that use this prior knowledge in different ways. Methods PR and A-PR use the prior knowledge to interact with a derived MDP, whose policy space corresponds with the subset of the original policy space that is implicitly defined by the policy restrictions. Method P-PR uses the policy restrictions to learn a reduced action set for the original MDP. While its corresponding policy space is

in general larger than that of PR or A-PR, its state-action space is in general smaller.

Empirically, we showed that by defining general policy restrictions for a maze task, the performance of PR and A-PR on a maze task becomes independent of the resolution of the maze, whereas for Q-learning the performance scales quadratically with the resolution. For P-PR the performance depends linearly on the resolution. These results clearly demonstrate the huge impact that policy restrictions can have on performance.

CHAPTER 7

Conclusions and Future Work

In this thesis we presented an approach to fill the gap between model-free and model-based learning. Best-match learning unifies these two forms of learning in the sense that a model-free method (Q-learning) and a model-based method (value interaction based on a maximum-likelihood model) are both special cases of best-match learning. Using best-match learning, methods can be constructed that can trade both space and time requirements for an improved performance.

We also presented a novel strategy for on-line representation selection for tasks where the agent can choose among a set of different candidate representations. This strategy transforms the original MDP and the set of candidate representations into a derived MDP with a single representation whose solution yields both the optimal representation for the original MDP and the optimal policy under that representation. The time and space requirements depend on the size of the derived MDP's state space, which can be exponentially smaller than that of the original MDP.

Finally, we demonstrated how prior knowledge about a task can often be naturally described in terms of policy restrictions and presented a straightforward way to exploit these restrictions. We empirically showed that for a maze task with policy restrictions the performance only depends on the general layout of the maze and not on the scale of the maze.

Overall, we believe that the theory and strategies presented in this thesis provide a valuable extension to the toolkit of reinforcement learning researchers that strive to optimize the performance of their systems under space and time constraints.

In the next section, we evaluate the six research questions formulated in Section 1.2.2. In Section 7.2, we discuss the three most promising avenues of future work.

7.1 Evaluation of Research Questions

Question: Under which settings does Expected Sarsa outperform regular Sarsa?
Answer: Tasks with low environment stochasticity (for example deterministic environments) combined with a highly stochastic policy (for example ε-greedy with $\varepsilon \geq 0.1$).
Explanation: Variance in an update target slows learning, since it forces averaging over multiple update targets to get accurate estimates. The downside of this is that the effect of a single update is smaller, and hence the number of updates necessary to obtain (near-) optimal Q-values increases. The variance in a Sarsa update has two sources: environment stochasticity and policy stochasticity. Expected Sarsa is based on an update target that uses an expectation over the action selected at the next timestep, and by doing so it fully

removes the variance due to the policy stochasticity. This makes less averaging possible, which improves the speed of learning.

Based on this difference between Sarsa and Expected Sarsa, the performance difference between these two methods is obviously larger for high policy stochasticity. For example, when an ε-greedy policy is used, more exploration, i.e, higher values of ε, cause a higher performance advantage of Expected Sarsa with respect to Sarsa. When environment stochasticity is high, the relative impact of the variance due to policy stochasticity is smaller, and hence the performance advantage of Expected Sarsa decreases. The performance advantage of Expected Sarsa is maximum when the environment is deterministic and the policy stochasticity, i.e., the exploration, is high.

Question: Does just-in-time Q-learning have a guaranteed performance improvement over regular Q-learning?
Answer: No, it is possible to design problems for which Q-learning is better than just-in-time Q-learning. However, on most problems, just-in-time Q-learning outperforms regular Q-learning since its update targets receive more updates.
Explanation: The intuitive idea behind just-in-time (JIT) learning is very simple. As long as a Q-value is not used, it does not have to be updated. Postponing the update until just before it is needed, can result in more accurate updates (and hence improved performance), since the Q-values used in the update target may have improved in the meantime.

We proved that, given the same experience sequence, each Q-value from the current state has received the same number of updates using JIT updates as using regular updates. However, each Q-value in the update target of a JIT update has received an equal or greater number of updates as in the update target of the corresponding regular update.

Despite this strong result, guaranteeing that JIT Q-learning outperforms Q-learning for all MDPs, and for all possible initializations of the Q-values is simply not possible. It is always possible to design some 'freak cases' with specific Q-value initialization for which update targets with more updates (temporarily) cause a performance disadvantage. In general, however, just-in-time Q-learning outperforms regular Q-learning. We demonstrated this empirically by showing that JIT Q-learning consistently outperforms regular Q-learning under a wide variety of settings.

Question: Is it possible to construct a strategy with similar space and time requirements to those of eligibility traces that consistently outperforms it?
Answer: Yes, this can be achieved by using best-match learning based on a last-visit model (LVM).
Explanation: Best-match LVM learning is strongly related to eligibility traces, however, it improves it in two important ways. The first improvement is related to how revisits of states (or state-action pairs) are handled. From the theory behind eligibility traces it is a little unclear what the best way is to deal with revisits of states, resulting in different traces types and no hard rules on when to use which one. With best-match learning treatment of revisits follows in a natural way from the theory and is fundamentally different from

eligibility traces. The better way of dealing with revisits of states is the main reason of the huge performance difference between best-match learning and eligibility traces shown in Figure 4.5.

The second improvement is related to the fact that eligibility traces can be viewed (roughly) as a best-match method that approximates the best-match values based on the followed trajectory. However, this is not the best, nor most efficient way to approximate the best-match values. For example, with prioritized sweeping (as in BM-LVM, see Algorithm 10) a much better trade-off between computation time and quality of approximation is obtained.

Because of these two improvements, best-match methods can be constructed (for example BM-LVM) that have similar space and time requirements as methods based on eligibility traces, but consistently outperform it.

Question: Is it possible to construct methods with a space complexity between $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ that provably converge to the optimal Q-values?
Answer: Yes, this can be achieved by using best-match learning based on the n-transition model (NTM).
Explanation: By using a partial model that only stores a fraction of the observed samples, a space complexity between $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ can be obtained. However, because only a fraction of the samples is stored, updates based on only this partial model cause an unfair bias in the Q-values towards samples in this model, preventing convergence. Best-match learning gets rid of this bias by combining the partial model with a partial Q-value function, constructed from samples not stored in the model. This enables convergence for a wide variety of different models, including NTMs. An NTM estimates the transition probability for n possible next states of each state-action pair. By tuning n, the space complexity can be set anywhere between $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$.

Question: Under which conditions can convergence to the optimal Q-values be guaranteed, when representation selection is applied?
Answer: If all candidate representations are valid representations.
Explanation: A candidate representation is a feature set that is a subset of the total feature set of an MDP. A candidate representation is valid if the features that the candidate representation leaves out are either independent features (see Definition 12) or features that are irrelevant with respect to the candidate representation (see Definition 7).

We showed that the original MDP and the set of candidate representations can be transformed into a derived task with a single representation whose solution yields both the optimal representation for the original MDP and the optimal policy under that representation. We proved that if all the candidate representations are valid this derived task obeys the Markov property, i.e., it forms a derived MDP. If this is the case, standard RL methods can be used to solve it.

Question: How can a reduced policy space be exploited in value-function based RL?
Answer: By describing it using a policy restriction set.
Explanation: A wide variety of policy restrictions can be compactly modeled using a policy restriction set. Combining this policy restriction set with the original MDP results in a derived MDP. The optimal policy of this derived MDP is the same as the optimal policy of the original MDP. However, learning with the derived MDP can be faster, due to a smaller state-action space and/or the existence of single-action states, states that do not require Q-values since their action set consists of only a single state, resulting in an overall improved performance. We demonstrated this empirically for a maze problem.

7.2 Future Work

In this section, we discuss the three most promising avenues of future work (in random order) that came out of the research described in this thesis.

- **Approximating the solution of the best-match equations by simulation-based search**
  In Chapter 4 the main strategy for selecting state-action pairs for updating is prioritized sweeping. While very effective, the overhead involved in maintaining a priority queue makes best-match methods based upon prioritized sweeping less suitable when the time constraints are really tight. On the other hand, in case of function approximation, prioritized sweeping cannot be used. However, the performance of the best-match method based upon a list of the most recent samples was only slightly better than the performance of experience replay based upon these samples. We think that both these issues can be solved by using an alternative strategy for selecting state-action pairs for updating, based upon simulation-based search (Rust, 1997; Coulom, 2006; Kocsis and Szepesvári, 2006).

  Simulation-based search is a planning technique that creates simulated experience sequences starting from the current state and uses these sequences for updating in order to get a better estimate of the values of the available actions in the current state.

  Silver (2009) investigated *temporal-difference search*, a simulation-based search method that uses temporal difference learning to update a value function based on the simulated experience sequence. We propose to use best-match updates instead of temporal-difference updates and let the partial model of best-match learning generate the simulated experience sequences.

  We expect that for tabular Q-values this strategy can rival prioritized sweeping in terms of efficiency and performance in most settings and can outperform it substantially under tight time constraints. In addition, after some slight modifications, we expect that this strategy can also be used for best-match function approximation. What enables this combination is the unique property of best-match learning to perform updates that are unbiased with respect to the update selection strategy.

We expect that by using simulation-based search new efficient function approximation methods can be constructed that can substantially outperform existing (efficient) function approximation methods.

- **Best-Match learning combined with the Sarsa update rule**
  The best-match equations discussed in Chapter 4 are based on the Q-learning update rule. However, it is also possible to construct a set of best-match equations based on the Sarsa or Expected Sarsa update rule.

  For best-match methods based on the Expected Sarsa update rule we do not expect a huge performance advantage, although the on-policy nature of methods based on Expected Sarsa will likely give them a slight performance edge over regular, off-policy best-match methods on certain domains.

  A far more interesting combination is the combination with the Sarsa update rule. The reason is that while best-match equations based on the Q-learning or Expected Sarsa update rule are non-linear, the best-match equations based on the Sarsa update rule form a linear set. From the evaluation case (Section 4.1.2) we know that this can result in a very efficient computation of the exact solution plus updates that are unbiased with respect to the initial values. We expect that these properties can be extended to the control case if best-match equations based on the Sarsa update rule are used.

  However, note that the solution of best-match equations based on the Sarsa rule is different from the solution of the equations based on the Q-learning rule. We expect this solution to be worse, due to the extra variance introduced by the Sarsa update rule (among others). It would be interesting to see how the trade-off between the advantages and the disadvantages of using best-match learning based on the Sarsa update rule plays out in practise for a variety of tasks.

- **Combining Policy Restrictions with Dynamic Programming**
  With dynamic programming, the full MDP is known in advance. Also in this case policy restrictions can be useful. Although, in this setting, the added value of the policy restriction set is not that it represents prior knowledge not otherwise available to the agent, since the agent already has full knowledge of the task. Instead, the added value is that it represents knowledge that might be deeply hidden inside the MDP and that cannot easily be extracted. In addition, it offers a way to exploit this knowledge.

  The combination of policy restrictions with dynamic programming provides new research opportunities not available in reinforcement learning. The ability to represent prior knowledge about a task depends strongly on the available state features. In reinforcement learning, the available feature set cannot be modified and hence forms a constraint in the ability to represent knowledge. This constraint does not exist in dynamic programming, since the agent has full knowledge of the MDP and can therefore construct an arbitrary feature set. This can result in very advanced features capable of modeling very specific knowledge.

In addition, the knowledge encoded by the policy restrictions does not have to be specified in advance. Instead, a separate reasoning process can be applied to extract policy restrictions from the full MDP. This can lead to an interesting trade-off between the amount of time spent on extracting policy restrictions versus the amount of time spent on solving the resulting derived MDP.

That said, we admit that we are not fully up-to-date with the state of the art in dynamic programming, so any research along this path would have to start with an extensive literature study to check for similar existing approaches for dynamic programming.

APPENDIX A

Relationship between Best-Match LVM and TD(λ)

Sutton and Singh (1994) showed that it is possible to perform TD updates that are unbiased with respect to the initial values, by using TD(λ) where λ is made time-dependent and set equal to $\alpha_t(s_t)$. However, TD(λ) can be made unbiased only for acyclic tasks, that is, episodic tasks with no revisits of states within an episode. In this appendix, we prove that best-match LVM evaluation and TD(λ) can lead to the same values for acyclic tasks and that best-match LVM evaluation can perform unbiased updates for all MDPs.

A.1 Background on TD(λ)

The forward view of TD(λ) relates it to the λ-*return* Watkins (1989); Jaakkola et al. (1994), defined by

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} ,$$

where $R_t^{(n)}$ indicates the *n-step return*, defined by

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

The λ-*return algorithm* updates state s_t with R_t^λ. It can only be implemented off-line, since it makes use of values from timesteps larger than t for the update of state s_t. While the off-line version of TD(λ) computes the same state values as the λ-return algorithm Sutton and Barto (1998), TD(λ) can also be implemented on-line, since it does not rely on values from the future. On-line TD(λ) can lead to more accurate updates than off-line TD(λ), although the interpretation as an incremental implementation of the λ-return holds only by approximation for the on-line case Sutton and Barto (1998).

The backward view of TD(λ) interprets λ as the trace decay parameter of an eligibility trace. Each sample is used to update, not just the current state, but all states, proportional to their *trace* parameter. At each timestep the trace of the current state is increased, while the other traces are decreased by $\gamma\lambda$. *Accumulating traces* increase the trace parameter of a visited state by 1, while *replacing traces* set it equal to 1.

Sutton and Singh (1994) proposed several ways for setting α and λ that eliminate bias towards initial state values, normally inherent to temporal-difference methods. One of these is to use TD(λ) where $\lambda_t = \alpha_t(s_t)$ and $\alpha_0(s) = 1$ for all s. This produces the same values as processing a state backwards with TD(0). All the proposed methods eliminate the bias only for acyclic tasks.

The equation for the λ-return with time-dependent λ is Sutton and Barto (1998)

$$\begin{aligned} R_t^{\lambda_t} &= \sum_{n=1}^{\infty} R_t^{(n)}(1-\lambda_{t+n}) \prod_{i=1}^{n-1} \lambda_{t+i} \\ &= \sum_{n=1}^{T-t-1} R_t^{(n)}(1-\lambda_{t+n}) \prod_{i=1}^{n-1} \lambda_{t+i} + R_t \prod_{i=1}^{T-t-1} \lambda_{t+i}\,, \end{aligned} \tag{A.1}$$

where T is the last timestep of the episode and R_t is the complete return. Note that $R_t = R_t^{(T-t)}$.

A.2 Forward View Best-Match LVM Values

The λ-return is based on the experience sequence encountered by the agent when interacting with the environment. We can define for each visited state a *last-visit experience sequence* based on the LVM by going through the transition states defined in the LVM. Using this sequence we define a last-visit version of the n-step return and of a special version of the λ-return.

Definition 20. *The last-visit experience sequence for state s is*

$$s_{[0]}, r_{[1]}, s_{[1]}, r_{[2]}, s_{[2]}, ..., r_{[N]}, s_{[N]}\,,$$

where $s_{[0]} = s$, $s_{[n]} = S'(s_{[n-1]})$ for $n > 0$ and $r_{[n]} = R'(s_{[n-1]})$. The sequence ends when a state is encountered that is terminal, equal to $s_{[0]}$ or that has no transition state. We call $s_{[N]}$ the last-visit sequence end state.

Using this definition, we define a last-visit version of the n-step return.

Definition 21. *The last-visit n-step return of s is the n-step return applied to the last-visit experience sequence of s:*

$$\breve{R}_s^{(n)} = r_{[1]} + \gamma\, r_{[2]} + \gamma^2\, r_{[3]} + ... + \gamma^{n-1}\, r_{[n]} + \gamma^n\, V^{mf}(s_{[n]})\,. \tag{A.2}$$

We can now define a special version of the λ-return, which we call the *last-visit α-return*: a last-visit version of the time dependent λ-return (Equation A.1) with $\lambda_t = \alpha_t(s_t)$.

Definition 22. *The last-visit α-return of s is*

$$\breve{R}_s^{\alpha} = \sum_{n=1}^{N-1} \breve{R}_s^{(n)}(1-\alpha^{[n]}) \prod_{i=1}^{n-1} \alpha^{[i]} + \breve{R}_s^{(N)} \prod_{i=1}^{N-1} \alpha^{[i]}\,, \tag{A.3}$$

where $\alpha^{[k]}$ is shorthand for $\alpha(s_{[k]})$, $s_{[k]}$ is the k^{th} state from the last-visit experience sequence of s and N is the index of the last-visit sequence end state.

The following lemma relates the last-visit α-return of s to the best-match value of s. The lemma is proven in Appendix C.2.

Lemma 3. *If the last-visit sequence end state of s is a terminal state, the following equation holds for the best-match value of s:*

$$V^B(s) = (1 - \alpha^s)\, V^{mf}(s) + \alpha^s \breve{R}^{\alpha}_s \,.$$

This lemma forms the basis for the proof of the following theorem.

Theorem 3 *For an episodic, acyclic, evaluation task, off-line best-match LVM evaluation computes the same values as off-line TD(λ) with $\lambda_t = \alpha_t(s_t)$.*

Proof. Let V_k be the state value function after the off-line updates at the end of episode k. For all states that are visited during an episode, V is updated according to Lemma 3, since the last-visit sequence end state is a terminal state for all these visited states. For the off-line algorithm, before $V_k(s)$ is computed, the update $V^{mf}_k(s) = V_{k-1}(s)$ is performed for all visited states. Therefore, the value updates of the visited states can be written as

$$V_k(s) = (1 - \alpha^s)\, V_{k-1}(s) + \alpha^s \breve{R}^{\alpha}_s \,.$$

If the task is acyclic, the last-visit experience sequence of a visited state s is equal to the experience sequence followed by the agent from this state to the terminal state. Therefore, $\breve{R}^{\alpha}_s = R_t^{\lambda=\alpha_t(s_t)}$. Finally, since the values computed by off-line TD(λ) are equal to the values computed by the λ-return algorithm, off-line TD(λ) with $\lambda_t = \alpha_t(s_t)$ performs the same updates as off-line best-match LVM evaluation. □

It follows from Theorem 3 that best-match evaluation can also eliminate the bias for acyclic tasks. The next theorem extends this property to a general MDP.

Theorem 4 *The state values computed by the on-line best-match LVM evaluation algorithm (Algorithm 8) are unbiased with respect to the initial state values, when the initial learning rates $\alpha_0(s)$ are set to 1 for all s.*

Proof. Algorithm 8 computes at each timestep the best-match value of the current state. We will prove that if the best-match values of visited states computed at timesteps smaller than t are unbiased with respect to the initial state values, then so is the best-match value computed at timestep t. Since for $t = 0$ there are no visited states, it follows by induction that the values computed for all timesteps t are unbiased.

The best-match values are computed using $V^B(s_{[0]}) = c_A + c_B\, V^B(s_{[N]})$ with c_A and c_B defined as in (4.4) and (4.5) respectively. In Section 4.1.2 we showed that for the current state, $s_{[N]}$ is either a terminal state or equal to $s_{[0]}$. If $s_{[N]}$ is a terminal state, $V^B(s_{[0]}) = c_A$, while if $s_{[0]} = s_{[N]}$, then $V^B(s_{[0]}) = c_A/(1 - c_B)$. In either case, the computed best-match value depends only on the variables in c_A and c_B, which consists of the learning rates, $V^{mf}(s_{[i]})$, $s_{[i]}$ and $r_{[i]}$ for $0 \leq i \leq N$. Clearly, only $V^{mf}(s_{[i]})$ can be affected by the initial state values. $s_{[i]}$ has been visited at least once, else it would not appear in the last-visit experience sequence. If $s_{[i]}$ has been visited once, $V^{mf}(s_{[i]})$ is equal to the initial value $V_0(s_{[i]})$. However, since we assumed initial learning rates of 1, this value of $V^{mf}(s_{[i]})$ is removed from c_A. If $s_{[i]}$ has been visited more than once, it is equal to the

best-match value of $s_{[i]}$ computed at a timestep smaller than t. From this it follows that if the best-match values computed at timesteps smaller than t are unbiased with respect to the initial values, then so is the best-match value computed at timestep t. $\square$

APPENDIX B

Off-Policy Monte Carlo Update

In this appendix, we deduce the equation for the off-policy Monte Carlo update of the switch actions for the derived task of a contextual bandit problem with representation selection. While in general, off-policy Monte Carlo updates are very inefficient, in this specific case a particular simple and efficient equation is obtained. In this appendix, we use $\pi(x)$ to refer to the action given by a deterministic policy, while we use $\pi(x, a)$ to refer to the action selection probability of a stochastic policy.

The experience sequence of the derived task of a contextual bandit problem consists of two actions: first a switch actions, a_{sw}, and then a ground action, a_{gr}:

$$x_{t=0} \rightarrow a_{sw} \rightarrow x_{t=1} \rightarrow a_{gr} \rightarrow r$$

A Monte Carlo update is an update with the complete return, i.e., the (discounted) cumulative reward. To understand the difference with regular (on-policy) Monte Carlo updates consider that we determine the Q-value of a state-action pair (x, a) by simply taking the average of all returns seen so far:

$$Q(x, a) = \frac{\sum_{i=1}^{N} R_i(x, a)}{N} \tag{B.1}$$

where $R_i(x, a)$ is the return followed by the i-th visit of state-action pair (x, a) and N is the total number of returns observed for (x, a). A similar off-policy version can than be made by taking the weighted average:

$$Q(x, a) = \frac{\sum_{i=1}^{N} w_i(x, a) \cdot R_i(x, a)}{\sum_{i=1}^{N} w_i(x, a)} \tag{B.2}$$

where $w_i(x, a)$ is the weight assigned to the i-th return for (x, a). The value of $w_i(x, a)$ is computed as follows. Let $p(x, a)$ be the probability of the state action sequence following (x, a) occurring under the estimation policy π and $p'(x, a)$ be the probability of it occurring under the behavior policy π'. Than the weight $w_i(x, a)$ is equal to the relative probability of the observed experience-sequence of occurring under π and π', i.e. by $p(x, a)/p'(x, a)$. These probabilities can be expressed in their policy probabilities by:

$$w(x_t, a_t) = \frac{p(x_t, a_t)}{p'(x_t, a_t)} = \prod_{k=t+1}^{T-1} \frac{\pi(x_k, a_k)}{\pi'(x_k, a_k)} \tag{B.3}$$

For a deterministic evaluation policy π the weight w is non-zero only when all actions taken under π' match the action that would have been taken under π. If this is the case, the

above equation simplifies to:

$$w(x_t, a_t) = \prod_{k=t+1}^{T-1} \frac{1}{\pi'(x_k, a_k)} \qquad \text{if } \pi(x_k) = a_k \text{ for all } k \geq t+1 \tag{B.4}$$

where $\pi(x_k)$ refers to the action the agent would take at timestep k (with probability 1) when following this deterministic policy.

Since in our case, the state-action pair that requires the off-policy Monte Carlo update, $(x_{t=0}, a_{sw})$, is followed by only a single action (a_{gr}), the weight expression is reduced even further to

$$w = \frac{1}{\pi'(x_{t=0}, a_{sw})} \qquad \text{if } \pi(x_{t=1}) = a_{gr} \tag{B.5}$$

Given that we use an ε-greedy behavior policy and the condition that $\pi(x_{t=1}) = a_{gr}$, the weight w is a fixed value and can therefore be scaled to 1. Now, the off-policy Monte Carlo update of the switch action is reduced to the form:

$$Q(x_{t=0}, a_{sw}) = (1 - \alpha)\, Q(x_{t=0}, a_{sw}) + \alpha \cdot r \qquad \text{if } a_{gr} \text{ is optimal} \tag{B.6}$$

APPENDIX C

Proofs

C.1 Theorem 2

Theorem 2 *Given the same experience sequence, each Q-value from the current state has received the same number of updates using JIT updates (Equation 3.12) as using regular updates (Equation 3.11). However, each Q-value in the update target of a JIT update has received an equal or greater number of updates as in the update target of the corresponding regular update.*

Proof. To prove the theorem, we need to prove

$$U[\tilde{Q}_t(s_t, a)] = U[Q_t(s_t, a)], \quad \text{for all } a, \tag{C.1}$$

$$U[\tilde{Q}_{t-1}(s_{t^*+1}, a)] \geq U[Q_{t^*}(s_{t^*+1}, a)], \quad \text{for all } a, \tag{C.2}$$

where $U[Q_k]$ is the total number of updates a Q-value has received at time k. From Equation 3.11 and 3.12 it follows that for both update types (s_t, a_{t^*}) is updated once between timestep t^* and timestep t, while the Q-values of the other actions of s_t are not updated during this period. Since this applies to all visits and $U[\tilde{Q}_0(s, a)] = U[Q_0(s, a)] = 0$ for all s and a, the total number of updates for a state-action pair is always equal for just-in-time updates and regular updates, when the state is the current state, proving (C.1).

To prove (C.2), first assume that a_{t^*} is a returning action, that is, $t-1 = t^*$. In this case clearly (C.2) is true. Now, assume a_{t^*} is not a returning action, that is, $t - 1 > t^*$. From (C.1) it follows that $U[\tilde{Q}_{t^*+1}(s_{t^*+1}, a)] = U[Q_{t^*+1}(s_{t^*+1}, a)]$. Since $t - 1 \geq t^* + 1$ and $U[\tilde{Q}]$ increases monotonically over time, it follows that (C.2) is true. When state s_{t^*+1} is revisited before t, an extra update is performed and there is at least one action a, for which $U[\tilde{Q}_{t-1}(s_{t^*+1}, a)] > U[Q_{t^*}(s_{t^*+1}, a)]$. □

C.2 Lemma 3

For the sake of brevity, we present only the proof of Lemma 3 for constant α. The proof for state dependent α follows the same pattern.

Lemma 3 *If the last-visit sequence end state of s is a terminal state, the following equation holds for the best-match value of s:*

$$V^B(s) = (1 - \alpha^s)\, V^{mf}(s) + \alpha^s \breve{R}^{\alpha}_s .$$

Proof. The best-match values in case of an LVM are defined as the solution of the set of best-match LVM equations (Definition 5). In Section 4.1.2 we showed that by backward substitution of best-match equations we can express the best-match value of $s_{[0]}$ in terms of the best-match value of $s_{[N]}$. If $s_{[N]}$ is a terminal state, $V^B(s_{[N]}) = 0$ and $V^B(s_{[0]})$ is equal to c_A defined as in (4.4). This yields

$$\begin{aligned} V^B(s_{[0]}) &= \sum_{i=0}^{N-1} \Big((1-\alpha) V^{mf}(s_{[i]}) + \alpha\, r_{[i+1]} \Big) \prod_{k=0}^{i-1} \gamma\alpha\,, \\ &= \alpha \sum_{k=1}^{N} (\alpha\,\gamma)^{k-1}\, r_{[k]} + (1-\alpha) \sum_{k=0}^{N-1} (\alpha\,\gamma)^k\, V^{mf}(s_{[k]})\,. \end{aligned} \tag{C.3}$$

On the other hand, by substituting the definitions of the last-visit α-return (A.3) and the last-visit n-step return (A.2) into the lemma, the following equation for $V^B(s_{[0]})$ appears:

$$\begin{aligned} V^B(s_{[0]}) &= (1-\alpha)\, V^{mf}(s_{[0]}) + \alpha \Bigg[(1-\alpha) \sum_{k=1}^{N-1} \alpha^{k-1} \Big(\sum_{p=1}^{k} \gamma^{p-1} r_{[p]} + \gamma^k\, V^{mf}(s_{[k]}) \Big) \\ &\quad + \alpha^{N-1} \sum_{p=1}^{N} \gamma^{p-1} r_{[p]} \Bigg]\,. \end{aligned} \tag{C.4}$$

The rest of this proof shows that (C.3) is equal to (C.4).

We start by separating (C.4) into its state value components (V^c) and its reward components (R^c). We then simplify these components separately:

$$\begin{aligned} V^c &= (1-\alpha)\, V^{mf}(s_{[0]}) + \alpha\,(1-\alpha) \sum_{k=1}^{N-1} \alpha^{k-1}\, \gamma^k\, V^{mf}(s_{[k]}) \\ &= (1-\alpha) \Big(V^{mf}(s_{[0]}) + \sum_{k=1}^{N-1} (\alpha\,\gamma)^k\, V^{mf}(s_{[k]}) \Big) \\ &= (1-\alpha) \sum_{k=0}^{N-1} (\alpha\,\gamma)^k\, V^{mf}(s_{[k]})\,, \end{aligned}$$

$$
\begin{aligned}
R^c &= (1-\alpha)\sum_{k=1}^{N-1}\sum_{p=1}^{k}\alpha^k\,\gamma^{p-1}r_{[p]} + \alpha^N\sum_{p=1}^{N}\gamma^{p-1}r_{[p]} \\
&= (1-\alpha)\sum_{p=1}^{N-1}\sum_{k=p}^{N-1}\alpha^k\,\gamma^{p-1}r_{[p]} + \alpha^N\sum_{p=1}^{N-1}\gamma^{p-1}r_{[p]} + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \sum_{p=1}^{N-1}\left[(1-\alpha)\sum_{k=p}^{N-1}\alpha^k\,\gamma^{p-1}r_{[p]} + \alpha^N\,\gamma^{p-1}r_{[p]}\right] + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \sum_{p=1}^{N-1}\left[\left(\sum_{k=p}^{N-1}\alpha^k - \sum_{k=p}^{N-1}\alpha^{k+1} + \alpha^N\right)\gamma^{p-1}\,r_{[p]}\right] + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \sum_{p=1}^{N-1}\left[\left(\sum_{k=p}^{N}\alpha^k - \sum_{k=p}^{N-1}\alpha^{k+1}\right)\gamma^{p-1}\,r_{[p]}\right] + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \sum_{p=1}^{N-1}\left[\left(\sum_{j=p-1}^{N-1}\alpha^{j+1} - \sum_{k=p}^{N-1}\alpha^{k+1}\right)\gamma^{p-1}\,r_{[p]}\right] + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \sum_{p=1}^{N-1}\left[\alpha^p\,\gamma^{p-1}\,r_{[p]}\right] + \alpha^N\,\gamma^{N-1}\,r_{[N]} \\
&= \alpha\sum_{p=1}^{N}(\alpha\,\gamma)^{p-1}\,r_{[p]}\,.
\end{aligned}
$$

Adding these simplified components back together yields Equation C.3. □

C.3 Theorem 5

Theorem 5 *The Q-values of a member of the best-match LVM control class, shown in Algorithm 9, converge to Q^* if the following conditions are satisfied:*

1. *S and A are finite.*
2. *$\alpha_t(s,a) \in [0,1]$, $\sum_t \alpha_t(s,a) = \infty$, $\sum_t(\alpha_t(s,a))^2 < \infty$ with probability 1 (w.p.1) and $\alpha_t(s,a) = 0$ unless $(s,a) = (s_t, a_t)$.*
3. *$Var\{R(s,a,s')\} < \infty$.*
4. *$\gamma < 1$.*

Proof. We prove that the Q-values of an arbitrary instantiation of Algorithm 9 converge in the limit w.p.1 to those of the regular Q-learning algorithm. Because the algorithm requires that each visited state action pair is updated at least once before its revisit, the following equation holds

$$
Q^{mf}_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))\,Q^{mf}_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t^*+1} + \max_{a'} Q_{\tau,i}(s_{t^*+1},a')\right)\,,
$$

where t^* is the timestep of the previous visit of (s_t, a_t) and $Q_{\tau,i}$ is the Q-value of s_{t^*+1} that is used in the update target of the last best-match update of (s_t, a_t), at timestep τ. Note that $t^* + 1 \leq \tau \leq t$. Assume that Q-learning is applied to the same state-action sequence produced by the given instantiation of Algorithm 9. We denote the Q-values from Q-learning by $\tilde{Q}$. Subtracting the update equation for Q-learning at time $t^* + 1$ using learning rate $\alpha_t(s_t, a_t)$ and defining $\Delta_t(s, a) = Q_t^{mf}(s, a) - \tilde{Q}_{t^*}(s, a)$ yields

$$\Delta_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))\Delta_t(s_t, a_t) + \alpha_t(s_t, a_t)F_t(s_t, a_t)\,, \tag{C.5}$$

where $F_t(s_t, a_t) = \gamma \left(\max_c Q_{\tau,i}(s_{t^*+1}, c) - \max_c \tilde{Q}_{t^*}(s_{t^*+1}, c) \right)$.

We now prove that Q_t^{mf} and Q_{t^*} converge in the limit to each other using the same lemma used to prove the convergence of Sarsa Singh et al. (2000):

Lemma 4. *Consider a stochastic process $(\alpha_t, \Delta_t, F_t)$, $t \geq 0$, where $\alpha_t, \Delta_t, F_t : X \to \mathbb{R}$ satisfy the equations:*

$$\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \alpha_t(x)F_t(x)\ \ ,$$

where $x \in X$ and $t = 0, 1, 2, \ldots$. Let P_t be a sequence of increasing σ-fields such that α_0 and Δ_0 are P_0-measurable and ζ_t, Δ_t and F_{t-1} are P_t-measurable, $t = 1, 2, \ldots$. Assume that the following conditions hold:

1. *The set X is finite.*
2. *$\alpha_t(x) \in [0, 1]$, $\sum_t \alpha_t(x) = \infty$, $\sum_t (\alpha_t(x))^2 < \infty$ w.p.1.*
3. *$\|E\{F_t|P_t\}\| \leq \kappa\|\Delta_t\| + c_t$, where $\kappa \in [0, 1)$ and c_t converges to zero w.p.1, and*
4. *$Var\{F_t(x_t)|P_t\} \leq K(1 + \kappa\|\Delta_t\|)^2$, where K is some constant,*

where $\|\cdot\|$ denotes a maximum norm. Then Δ_t converges to zero with probability one.

The correspondence of (C.5) to Lemma 4 follows from associating X with the set of state-action pairs (s, a) and $\alpha_t(x)$ with $\alpha_t(s, a)$. We now prove that the 4 conditions hold.

The first two conditions follow from the first two conditions of Theorem 5. We define P_t as the set $\{Q_0, \alpha_0, a_0, s_0, ..., r_{t-1}, \alpha_t, a_t, s_t\}$. With this definition, $\text{Var}\{F_t(s_t, a_t)|P_t\} = 0$, satisfying condition 4, and $E\{F_t(s_t, a_t)|P_t\} = F_t(s_t, a_t)$. For $|F_t(s_t, a_t)|$ the following holds:

$$\begin{aligned}
|F_t(s_t, a_t)| &= \gamma\,|\max_b Q_{\tau,i}(s_{t^*+1}, b) - \max_b \tilde{Q}_{t^*}(s_{t^*+1}, b)| \\
&\leq \gamma||Q_{\tau,i}(u, b) - \tilde{Q}_{t^*}(u, b)|| \\
&= \gamma||\Delta_t(u, b) + Q_{\tau,i}(u, b) - Q_t^{mf}(u, b)|| \\
&\leq \gamma||\Delta_t|| + ||Q_{\tau,i}(u, b) - Q_t^{mf}(u, b)||\,.
\end{aligned}$$

We further define $F_t(s, a) = 0$ if $(s, a) \neq (s_t, a_t)$. Therefore, $||F_t(s, a)|| = |F_t(s_t, a_t)| \leq \gamma||\Delta_t|| + C_t$, where $C_t = ||Q_{\tau,i}(u, b) - Q_t^{mf}(u, b)||$. We now show that C_t converges to zero w.p.1. For C_t, the following holds:

$$C_t \leq ||Q_{\tau,i}(u, b) - Q_{\tau^*}^{mf}(u, b)|| + ||Q_{\tau^*}^{mf}(u, b) - Q_t^{mf}(u, b)||\,,$$

where τ^* is the timestep of the last visit of (u, b) before timestep τ. $Q_{\tau,i}(u, b)$ is the result of a best-match update of $Q^{mf}_{\tau^*}(u, b)$ or is equal to it if no best-match update has been performed yet. In the latter case, the first term is zero; in the former case it is

$$Q_{\tau,i}(u, b) = (1 - \alpha_\tau(u, b))Q^{mf}_{\tau^*}(u, b) + \alpha_\tau(u, b)\upsilon^{ub}_\tau .$$

Because of condition 2 of Theorem 5, $\alpha_\tau(u, b)$ converges to 0 w.p.1 and $Q_{\tau,i}(u, b)$ converges to $Q^{mf}_{\tau^*}(u, b)$ w.p.1. Therefore, the first term converges to 0 w.p.1. For the same reason, the second term converges to zero.

Thus, the third condition of the lemma also holds and $Q^{mf}(s, a)$ converges to $\tilde{Q}(s, a)$, the Q-values from Q-learning. Because of the convergence guarantee of Q-learning, $Q^{mf}(s, a)$ also converges to $Q^*(s, a)$. Finally, since the Q-values of the given instantiation are a best-match update of $Q^{mf}(s, a)$ and because $\alpha_t(s, a)$ converges to zero w.p.1, this also proves that the Q-values of the instantiation converge to Q^*. □

C.4 Theorem 6

Theorem 6 *The Q-values of a member of the best-match NTM control class, shown in Algorithm 11, converge to Q^* if the following conditions are satisfied:*

1. *S and A are finite.*
2. *$\alpha^{sa}_t \in [0, 1]$, $\sum_t \alpha^{sa}_t = \infty$, $\sum_t (\alpha^{sa}_t)^2 < \infty$ with probability 1 (w.p.1), and $\alpha^{sa}_t = 0$ unless $(s, a) = (s_t, a_t)$ and $s_{t+1} \notin NTM(s_t, a_t)$.*
3. *$Var\{R(s, a, s')\} < \infty$.*
4. *$\gamma < 1$.*

C.4.1 Preliminaries

In this proof, we indicate the NTM by $\mathcal{M}$. Also, we indicate the model-free Q-value, Q^{mf}, by $\breve{Q}$. In addition, we use a single iteration index j for Q as well as $\breve{Q}$. This global index is increased each time an update (of either $\breve{Q}$ or Q) occurs. Thus, j is equal to the total number of model-free updates plus best-match updates that have occurred since the start of an episode. Clearly, $t \to \infty$ implies $j \to \infty$.

By denoting the state-action pair that gets updated by the j-th update as (s_j, a_j), we can write the model-free (mf) update as

$$\breve{Q}_{j+1}(s_j, a_j) = (1 - \alpha^{s_j a_j})\breve{Q}_j(s_j, a_j) + \alpha^{s_j a_j} \left[r_{j+1} + \gamma \max_{a'} Q_j(s'_{j+1}, a')\right], \qquad \text{(C.6)}$$

where r_{j+1} and s'_{j+1} are the reward and transition state from the sample $(s_t, a_t, r_{t+1}, s_{t+1})$ corresponding to (s_j, a_j). We use s'_{j+1} instead of s_{j+1}, since s'_{j+1}, the transition state for s_j, is in general not equal to s_{j+1}, the state that receives an update at iteration step $j + 1$. The best-match (bm) update is

$$Q_{j+1}(s_j, a_j) = w_0^{s_j, a_j}\breve{Q}_j(s_j, a_j) + (1 - w_0^{s_j a_j}) \left[\hat{\mathcal{R}}_{s_j a_j} + \gamma \sum_{s'} \hat{\mathcal{P}}^{s'}_{s_j a_j} \max_{a'} Q_j(s', a')\right] .$$

Note that there is no specific sample corresponding to a best-match update, since the update is based on the model estimate and can occur multiple times per timestep.

Let $\mathcal{P}^{\mathcal{M}}_{sa} = \sum_{s' \in \mathcal{M}} \mathcal{P}^{s'}_{sa}$. If $\mathcal{P}^{\mathcal{M}}_{sa} = 0$, w_0^{sa} will always be 1 and the best-match update reduces to $Q_{j+1}(s_j, a_j) = \breve{Q}_j(s_j, a_j)$. We make this explicit by the following equation:

$$Q_{j+1}(s_j, a_j) = \begin{cases} \breve{Q}_j(s_j, a_j) & \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0 \\ Y_j(s_j, a_j) & \text{if } \mathcal{P}^{\mathcal{M}}_{sa} > 0\,, \end{cases} \tag{C.7}$$

with

$$Y_j(s_j, a_j) = w_0^{s_j, a_j} \breve{Q}_j(s_j, a_j) + (1 - w_0^{s_j a_j}) \left[\hat{\mathcal{R}}_{s_j a_j} + \gamma \sum_{s'} \hat{\mathcal{P}}^{s'}_{s_j a_j} \max_{a'} Q_j(s', a') \right].$$

Each time a sample is observed by the algorithm, w_0 gets updated. In addition, when the sample is part of $\mathcal{M}$, $\hat{\mathcal{R}}$ and $\hat{\mathcal{P}}$ get updated. Therefore, the values of these variables can change between iteration steps. However, for readability, we omit the j subscript for these variables. From the definition of w_0, $\hat{\mathcal{R}}$ and $\hat{\mathcal{P}}$, and the law of large numbers, it follows that in the limit the following holds:[1]

$$\lim_{j \to \infty} w_0^{sa} = 1 - \mathcal{P}^{\mathcal{M}}_{sa}\,, \tag{C.8}$$

$$\lim_{j \to \infty} \hat{\mathcal{R}}_{sa} = \sum_{s' \in \mathcal{M}} \mathcal{P}^{s'}_{sa} \mathcal{R}^{s'}_{sa} / \mathcal{P}^{\mathcal{M}}_{sa}\,, \tag{C.9}$$

$$\lim_{j \to \infty} \hat{\mathcal{P}}^{s'}_{sa} = \mathcal{P}^{s'}_{sa} / \mathcal{P}^{\mathcal{M}}_{sa}\,. \tag{C.10}$$

In general, the model-free Q-values, $\breve{Q}$, will not converge to Q^*, since they do not receive updates from samples corresponding to the next states stored by the NTM. However, as part of the proof, we show that the model-free Q-values converge to an alternative value, which we indicate by $\breve{Q}^*$. This value is defined as[2]

$$\breve{Q}^*(s, a) = \sum_{s' \notin \mathcal{M}} \mathcal{P}^{s'}_{sa} \left[\mathcal{R}^{s'}_{sa} + \gamma \max_{a'} Q^*(s', a') \right] / (1 - \mathcal{P}^{\mathcal{M}}_{sa})\,. \tag{C.11}$$

Using this equation, we can express Q^* as

$$\begin{aligned} Q^*(s, a) &= \sum_{s' \notin \mathcal{M}} \mathcal{P}^{s'}_{sa} [\mathcal{R}^{s'}_{sa} + \gamma \max_{a'} Q^*(s', a')] + \sum_{s' \in \mathcal{M}} \mathcal{P}^{s'}_{sa} [\mathcal{R}^{s'}_{sa} + \gamma \max_{a'} Q^*(s', a')] \\ &= (1 - \mathcal{P}^{\mathcal{M}}_{sa}) \breve{Q}^*(sa) + \sum_{s' \in \mathcal{M}} \mathcal{P}^{s'}_{sa} [\mathcal{R}^{s'}_{sa} + \gamma \max_{a'} Q^*(s', a')]\,. \end{aligned} \tag{C.12}$$

Note that it follows from (C.12), that

$$Q^*(s, a) = \breve{Q}^*(s, a)\,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0\,. \tag{C.13}$$

[1] Note that $\hat{\mathcal{R}}_{sa}$ and $\hat{\mathcal{P}}^{s'}_{sa}$ do not converge to $\mathcal{R}_{sa}$ and $\mathcal{P}^{s'}_{sa}$, but to normalized values of these variables.

[2] For $\mathcal{P}^{\mathcal{M}}_{sa} = 1$, that is, when all samples are stored by the NTM, $\breve{Q}^*(s, a)$ is not defined. However, in this case, $\breve{Q}(s, a)$ does not receive any updates, nor is it used by any other update. Therefore, we can safely ignore the value $\breve{Q}(s, a)$, and consequently $\breve{Q}^*(s, a)$, if $\mathcal{P}^{\mathcal{M}}_{sa} = 1$.

Convergence of Q_j to Q^* requires convergence of $\breve{Q}_j$ to $\breve{Q}^*$, and vice versa. To deal with this mutual dependence relation, we simultaneously prove their convergence. To achieve this, we define a function $U : \mathcal{S} \times \mathcal{A} \times \mathcal{B} \rightarrow \mathbb{R}$ that encompasses both functions Q and $\breve{Q}$. $\mathcal{B}$ is a set consisting of only two elements: 'mf' and 'bm', which indicate the *Q-value type*. We define U_j as

$$U_j(s,a,b) = \begin{cases} \breve{Q}_j(s,a) & \text{if b = 'mf'} \\ Q_j(s,a) & \text{if b = 'bm'} . \end{cases} \tag{C.14}$$

Both updates (C.6) and (C.7) can now be interpreted as updates of $U_j(s_j, a_j, b_j)$. It follows from (C.14) that when the model-free update is performed, b_j = 'mf', while for the best-match update b_j = 'bm'.

We will prove convergence of U_j to U_j^*, defined as

$$U^*(s,a,b) = \begin{cases} \breve{Q}^*(s,a) & \text{if b = 'mf'} \\ Q^*(s,a) & \text{if b = 'bm'} . \end{cases}$$

The difficulty with this proof is that we cannot simply apply Lemma 4 (or similar stochastic approximation lemmas), used to prove convergence of BM-LVM, since the $\sum_t (\alpha_t(x_t))^2 < \infty$ condition of Lemma 4 is not met for b = 'bm'. On the other hand, a related lemma can be deduced (see Appendix C.5), that does not require $\sum_t (\alpha_t(x_t))^2 < \infty$, however, it requires that the contraction condition holds for the value of F_t, instead of its expected value. Hence, also this lemma cannot be directly applied.

To deal with this, we define a related function U'_j, that does comply with the $\sum_t (\alpha_t(x_t))^2 < \infty$ condition, hence we can prove convergence of it to U^* using Lemma 4. On the other hand, the difference between U'_j and U_j complies with all the conditions of Lemma 7, hence we can prove that U_j converges to U'_j using Lemma 7. Adding these two results together, proves the theorem.

We define U'_j as

$$U'_j(s,a,b) = \begin{cases} \breve{Q}'(s,a) & \text{if b = 'mf'} \\ Q'(s,a) & \text{if b = 'bm'} . \end{cases}$$

$\breve{Q}'$ and Q' are updated using the same sample sequence as used for $\breve{Q}$ and Q. The update for $\breve{Q}'$ is

$$\breve{Q}'_{j+1}(s_j,a_j) = (1-\alpha^{s_j a_j})\breve{Q}'_j(s_j,a_j) + \alpha^{s_j a_j}\left[r_{j+1} + \gamma \max_{a'} Q'_j(s'_{j+1},a')\right],$$

while the update for Q' is

$$Q'_{j+1}(s_j,a_j) = \begin{cases} \breve{Q}'_j(s_j,a_j) & \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0 \\ (1-\beta^{s_j a_j})Q'_j(s_j,a_j) + \beta^{s_j a_j} Y'_j(s_j,a_j) & \text{if } \mathcal{P}^{\mathcal{M}}_{sa} > 0 , \end{cases} \tag{C.15}$$

with

$$Y'_j(s_j,a_j) = w_0^{s_j,a_j}\breve{Q}'_j(s_j,a_j) + (1-w_0^{s_j a_j})\left[\hat{\mathcal{R}}_{s_j a_j} + \gamma \sum_{s'} \hat{\mathcal{P}}^{s'}_{s_j a_j} \max_{a'} Q'_j(s',a')\right].$$

Note that the only difference with the updates of Q and $\breve{Q}$ is the way Q' is updated for $\mathcal{P}^{\mathcal{M}}_{sa} > 0$. Instead of setting $Q'_{j+1}(s_j, a_j)$ equal to $Y'_j(s_j, a_j)$, it is set equal to a weighted average of $Y'_j(s_j, a_j)$ and $Q'_j(s_j, a_j)$. The weighting is controlled by β_j, which is an arbitrary learning rate with properties $\beta^{sa}_j \in [0, 1]$, $\sum_j \beta^{sa}_j = \infty$, $\sum_j (\beta^{sa}_j)^2 < \infty$ w.p.1., and $\beta^{sa}_j = 0$ unless $(s, a) = (s_j, a_j)$ and $b_j =$ 'bm'.[3] Because of this learning rate, Lemma 4 can be used to prove convergence of U'_j to U^*.

C.4.2 Convergence of U'_j to U^*

Lemma 5. *$U'_j(s, a, b)$ converges in the limit to $U^*(s, a, b)$ w.p.1.*

Proof. We define $\Delta'(s, a, b) = U'_j(s, a, b) - U^*_j(s, a, b)$ and will prove that $\Delta'(s, a, b)$ converges to 0 using Lemma 4. For $b_j =$ 'bm', we use the contraction factor κ^{sa}, defined as

$$\kappa^{sa} = (1 - \mathcal{P}^{\mathcal{M}}_{sa}) + \gamma\, \mathcal{P}^{\mathcal{M}}_{sa}\,. \tag{C.16}$$

To ensure that $\kappa^{sa} < 1$, $\mathcal{P}^{\mathcal{M}}_{sa}$ has to be larger than 0. Therefore, we exclude (s, a, b) triples for which $b =$ 'bm' $\wedge\, \mathcal{P}^{\mathcal{M}}_{sa} = 0$ from the domain of Δ'. This can be done, because Algorithm 11 states that at least one best-match update occurs in between two model-free updates. Therefore, if $\mathcal{P}^{\mathcal{M}}_{sa} = 0$, $Q'_j(s, a)$ is either equal to $\breve{Q}'_j(s, a)$ or one (model-free) update apart. Since α^{sa}_j converges to 0, it follows that $Q'_j(s, a)$ converges in the limit to $\breve{Q}'_j(s, a)$. Alternatively, we can say

$$Q'_j(s, a) = \breve{Q}'_j(s, a) + c'_j(s, a)\,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0\,, \tag{C.17}$$

with $c'_j(s, a)$ converging to 0 w.p.1.[4] Combining this with (C.13), the following holds:

$$\lim_{j\to\infty} \breve{Q}'_j(s, a) = \breve{Q}^*(s, a) \quad \Rightarrow \quad \lim_{j\to\infty} Q'_j(s, a) = Q^*(s, a)\,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0\,. \tag{C.18}$$

Note, $\|\breve{Q}'_j - \breve{Q}^*\| \leq \|\Delta'_j\|$. However, because of the exclusion of $(s, a,$ 'bm'$)$ triples with $\mathcal{P}^{\mathcal{M}}_{sa} = 0$, $\|Q'_j - Q^*\| \leq \|\Delta_j\|$ does not hold in general. Instead, the following holds:

$$\begin{aligned}
\|Q'_j - Q^*\| &= \max(\|Q'_j - Q^*\|_{P^{\mathcal{M}}_{sa}>0}, \|Q'_j - Q^*\|_{P^{\mathcal{M}}_{sa}=0}) \\
&\leq \max(\|Q'_j - Q^*\|_{P^{\mathcal{M}}_{sa}>0}, \|\breve{Q}'_j - \breve{Q}^*\|_{P^{\mathcal{M}}_{sa}=0} + \|c'_j\|) \\
&\leq \max(\|U'_j - U^*\|, \|U'_j - U^*\| + \|c'_j\|) \\
&= \|U'_j - U^*\| + \|c'_j\| \\
&= \|\Delta'\| + \|c'_j\|\,.
\end{aligned}$$

Because of the exclusion of the (s, a, b) triples mentioned above, for all $(s, a,$ 'bm'$)$ triples in the domain of Δ'_j, $\mathcal{P}^{\mathcal{M}}_{sa} > 0$.

Δ'_j is updated according to

$$\Delta'_{j+1}(s, a, b) = (1 - \zeta'_j(s, a, b))\Delta'_j(s, a, b) + \zeta'_j(s, a, b)\, F'_j(s, a, b)\,.$$

[3] Note that such a β always exists.

[4] We use the notational convention to indicate variables that converge to 0 with probability 1 with lowercase, Latin, letters: c, d, e, ...

For $(s,a,b) \neq (s_j,a_j,b_j)$, $\zeta'_j(s,a,b) = 0$ and $F'_j(s,a,b) = 0$. For (s_j,a_j,b_j) the following holds:

$$\zeta'_j(s_j,a_j,b_j) = \begin{cases} \alpha_j^{s_j a_j} & \text{if } b_j = \text{`mf'} \\ \beta_j^{s_j a_j} & \text{if } b_j = \text{`bm'} \end{cases},$$

$$F'_j(s_j,a_j,b_j) = \begin{cases} r_{j+1} + \gamma \max_{a'} Q'_j(s'_{j+1},a') - \breve{Q}^*(s_j,a_j) & \text{if } b_j = \text{`mf'} \\ Y'_j(s_j,a_j) - Q^*(s_j,a_j) & \text{if } b_j = \text{`bm'} \end{cases}.$$

We now prove that Δ'_j converges to zero, by showing the conditions for Lemma 4 hold, using the σ-field P_j, defined as[5]

$$\begin{aligned} P_0 &= \{Q'_0, \breve{Q}'_0, \zeta_0, w_{0,0}, \breve{\mathcal{P}}_0, \breve{\mathcal{R}}_0, s_0, a_0\}, \\ P_j &= P_{j-1} \cap \{r_j, s'_j, \zeta_j, w_{0,j}, \breve{\mathcal{P}}_j, \breve{\mathcal{R}}_j, s_j, a_j\}. \end{aligned}$$

Conditions 1, 2 and 4 of the Lemma 4 follow from conditions 1,2, and 3 of Theorem 6 and the conditions that hold for β_j^{sa}. Condition 3 of the lemma, we prove below.

For b_j = 'mf', using (C.11), the following holds:

$$\begin{aligned} |E\{F'_j(s_j,a_j,\text{`mf'})|P_j\}| &= \Big| \sum_{s' \notin \mathcal{M}} \mathcal{P}_{s_j a_j}^{s'} [\mathcal{R}_{s_j a_j}^{s'} + \gamma \max_{a'} Q'_j(s',a')]/(1 - \mathcal{P}_{s_j a_j}^{\mathcal{M}}) - \breve{Q}^*(s_j,a_j) \Big| \\ &= \gamma \sum_{s' \notin \mathcal{M}} \mathcal{P}_{s_j a_j}^{s'} \Big| \max_{a'} Q'_j(s',a') - \max_{a'} Q^*(s',a') \Big| / (1 - \mathcal{P}_{s_j a_j}^{\mathcal{M}}) \\ &\leq \gamma \|Q'_j - Q^*\| \\ &\leq \gamma \|\Delta'_j\| + \gamma \|c'_j\|. \end{aligned} \tag{C.19}$$

For b_j = 'bm', using (C.12), we can write

$$\begin{aligned} |F'_j(s_j,a_j,\text{`bm'})| &= |Y'_j(s_j,a_j) - Q^*(s_j,a_j)| \\ &\leq \Big| (1 - \mathcal{P}_{s_j a_j}^{\mathcal{M}})(\breve{Q}'_j(s_j,a_j) - \breve{Q}^*(s_j,a_j)) \\ &\quad + \gamma \sum_{s' \in \mathcal{M}} \mathcal{P}_{s_j a_j}^{s'} [\max_{a'} Q'_j(s',a') - \max_{a'} Q^*(s',a')] \Big| \\ &+ \Big| \Big[w_0^{s_j a_j} - (1 - \mathcal{P}_{s_j a_j}^{\mathcal{M}}) \Big] \cdot \breve{Q}'_j(s_j,a_j) \Big| \\ &+ \Big| (1 - w_0^{s_j a_j}) \hat{\mathcal{R}}_{s_j a_j} - \sum_{s' \in \mathcal{M}} \mathcal{P}_{s_j a_j}^{s'} \mathcal{R}_{s_j a_j}^{s'} \Big| \\ &+ \gamma \Big| \sum_{s' \in \mathcal{M}} \Big[(1 - w_0^{s_j a_j}) \hat{\mathcal{P}}_{s_j a_j}^{s'} - \mathcal{P}_{s_j a_j}^{s'} \Big] \cdot \max_{a'} Q'_j(s',a') \Big|. \end{aligned}$$

The sum of the last three terms we call $d_j(s_j,a_j)$. By substituting (C.8), (C.9) and (C.10) in these three terms, it follows that $\lim_{j\to\infty} d_j(s_j,a_j) = 0$. We can further bound

[5] There is no explicit sample related to a best-match update. For consistency, we define $r_j = \emptyset$ and $s'_j = \emptyset$ if $b_{j-1} = \text{`}bm\text{'}$.

$|F'_j(s_j, a_j, \text{'bm'})|$ as follows:

$$\begin{aligned}
|F'_j(s_j, a_j, \text{'bm'})| &\leq (1 - \mathcal{P}^{\mathcal{M}}_{s_j a_j}) \|\breve{Q}_j - \breve{Q}^*\| + \gamma \mathcal{P}^{\mathcal{M}}_{s_j a_j} \|Q_j - Q^*\| + d_j(s_j, a_j) \\
&\leq (1 - \mathcal{P}^{\mathcal{M}}_{s_j a_j}) \|\Delta'_j\| + \gamma \mathcal{P}^{\mathcal{M}}_{s_j a_j} \left(\|\Delta'_j\| + \|c_j\| \right) + d_j(s_j, a_j) \\
&\leq \left((1 - \mathcal{P}^{\mathcal{M}}_{s_j a_j}) + \gamma \mathcal{P}^{\mathcal{M}}_{s_j a_j} \right) \cdot \|\Delta'_j\| + \gamma \mathcal{P}^{\mathcal{M}}_{s_j a_j} \|c'_j\| + d_j(s_j, a_j) \\
&= \kappa^{s_j a_j} \|\Delta'_j\| + \gamma \mathcal{P}^{\mathcal{M}}_{s_j a_j} \|c'_j\| + d_j(s_j, a_j) \,. \qquad \text{(C.20)}
\end{aligned}$$

Note $\|c'_j\|$, as well as $d_j(s_j, a_j)$, converge to 0. Note also that $\kappa^{s_j a_j} < 1$, since $\mathcal{P}^{\mathcal{M}}_{s_j a_j} > 0$ and $\gamma < 1$. From (C.19) and (C.20) it follows that the third condition of Lemma 4 is also satisfied. Hence, all conditions hold and Δ'_j converges to 0 w.p.1. Combining this with (C.18), proves Lemma 5.

□

C.4.3 Convergence of U_j to U'_j

Lemma 6. *$U_j(s, a, b)$ converges in the limit to $U'_j(s, a, b)$ w.p.1.*

Proof. We define $\Delta(s, a, b) = U'_j(s, a, b) - U_j(s, a, b)$ and will prove that $\Delta(s, a, b)$ converges to 0 using Lemma 7. We exclude $(s, a, \text{'bm'})$ triples for which $\mathcal{P}^{\mathcal{M}}_{sa} = 0$ from the domain of Δ. Similar to the reasoning behind (C.18) and (C.17), we can deduce

$$Q_j(s, a) = \breve{Q}_j(s, a) + c_j(s, a) \,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0 \,,$$

with $c_j(s, a)$ converging to 0 in the limit, as well as

$$\lim_{j \to \infty} \left(\breve{Q}'_j(s, a) - \breve{Q}_j(s, a) \right) = 0 \quad \Rightarrow \quad \lim_{j \to \infty} \left(Q'_j(s, a) - Q_j(s, a) \right) = 0 \,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{sa} = 0 \,. \qquad \text{(C.21)}$$

Note, $\|\breve{Q}'_j - \breve{Q}_j\| \leq \|\Delta_j\|$. However, $\|Q'_j - Q_j\| \leq \|\Delta_j\|$ does not hold in general, because of the exclusion of (s,a,'bm') triples with $\mathcal{P}^{\mathcal{M}}_{sa} = 0$ from the domain of Δ_j. Instead, the following holds:

$$\begin{aligned}
\|Q'_j - Q_j\| &= \max(\|Q'_j - Q_j\|_{P^{\mathcal{M}}_{sa} > 0}, \|Q'_j - Q_j\|_{P^{\mathcal{M}}_{sa} = 0}) \\
&\leq \max(\|Q'_j - Q_j\|_{P^{\mathcal{M}}_{sa} > 0}, \|\breve{Q}'_j - \breve{Q}_j\|_{P^{\mathcal{M}}_{sa} = 0} + \|c_j\| + \|c'_j\|) \\
&\leq \max(\|U'_j - U_j\|, \|U'_j - U^*\| + \|c_j\| + \|c'_j\|) \\
&= \|U'_j - U_j\| + \|c_j\| + \|c'_j\| \\
&= \|\Delta'_j\| + c''_j \,,
\end{aligned}$$

with $c''_j = \|c_j\| + \|c'_j\|$ converging to 0 w.p.1.

For $\mathcal{P}^{\mathcal{M}}_{sa} > 0$ we can rewrite (C.15) as

$$\begin{aligned}
Q'_{j+1}(s_j, a_j) &= (1 - \beta^{s_j a_j}) Q'_j(s_j, a_j) + \beta^{s_j a_j} Y'_j(s_j, a_j) \\
&= Y'_j(s_j, a_j) + (1 - \beta^{s_j a_j}) [Q'_j(s_j, a_j) - Y'_j(s_j, a_j)] \,.
\end{aligned}$$

In Section C.4.2 we proved that $\Delta'_j(s, a, \text{'bm'}) = Q'(s, a) - Q^*(s, a)_j$ converges to 0 w.p.1. On the other hand, it follows from (C.20), that $F'_j(s_j, a_j, \text{'bm'})$, which is equal

to $Y'_j(s_j,a_j) - Q^*(s_j,a_j)$, also converges to 0 w.p.1. Therefore, both $Q'_j(s_j,a_j)$ and $Y'_j(s_j,a_j)$ converge to the same value, so we can write

$$Q'_{j+1}(s_j,a_j) = Y'_j(s_j,a_j) + e_j(s_j,a_j)\,, \qquad \text{if } \mathcal{P}^{\mathcal{M}}_{s_ja_j} > 0\,,$$

with $e_j(s_j,a_j)$ converging to 0 w.p.1.

Δ_j is updated according to

$$\Delta_{j+1}(s,a,b) = (1-\zeta_j(s,a,b))\Delta_j(s,a,b) + \zeta_j(s,a,b)\,F_j(s,a,b)\,.$$

For $(s,a,b) \neq (s_j,a_j,b_j)$, $\zeta_j(s,a,b) = 0$ and $F_j(s,a,b) = 0$. While for (s_j,a_j,b_j) the following holds:

$$\zeta_j(s_j,a_j,b_j) = \begin{cases} \alpha_j^{s_ja_j} & \text{if } b_j = \text{`mf'} \\ 1 & \text{if } b_j = \text{`bm'}\,, \end{cases}$$

and

$$F_j(s_j,a_j,b_j) = \begin{cases} \gamma \max_{a'} Q'_j(s'_{j+1},a') - \gamma \max_{a'} Q_j(s'_{j+1},a') & \text{if } b_j = \text{`mf'} \\ Y'_j(s_j,a_j) - Y_j(s_j,a_j,b_j) + e_j(s_j,a_j) & \text{if } b_j = \text{`bm'}\,. \end{cases}$$

We now check the three conditions of Lemma 7. Conditions 1 and 2 from the lemma follow from conditions 1 and 2 of Theorem 6. Condition 3, we prove below.

For b_j = 'mf', the following holds:

$$\begin{aligned} |F_j(s_j,a_j,\text{`mf'})| &= \gamma \left| \max_{a'} Q'_j(s'_{j+1},a') - \max_{a'} Q_j(s'_{j+1},a') \right| \\ &\leq \gamma \|Q'_j - Q_j\| \\ &\leq \gamma \|\Delta_j\| + \gamma\, c''_j\,, \end{aligned} \tag{C.22}$$

while for b_j = 'bm', we can write

$$\begin{aligned} |F_j(s_j,a_j,\text{`bm'})| &= |Y'_j(s_j,a_j) - Y_j(s_j,a_j) + e_j(s_j,a_j,b_j)| \\ &\leq w_0^{s_j,a_j}\, |\breve{Q}'_j(s_j,a_j) - \breve{Q}_j(s_j,a_j)| + |e_j(s_j,a_j)| + \\ &\qquad \gamma(1-w_0^{s_ja_j}) \sum_{s'} \hat{\mathcal{P}}^{s'}_{s_ja_j} \left| \max_{a'} Q'_j(s',a') - \max_{a'} Q_j(s',a') \right| \\ &\leq w_0^{s_j,a_j} \|\Delta_j\| + \gamma(1-w_0^{s_ja_j})\|\Delta_j\| + |e_j(s_j,a_j)| + \gamma(1-w_0^{s_ja_j})\, c''_j \\ &= \left((1-\mathcal{P}^{\mathcal{M}}_{s_ja_j}) + \gamma\, \mathcal{P}^{\mathcal{M}}_{s_ja_j} \right) \|\Delta_j\| + |e_j(s_j,a_j)| + \gamma(1-w_0^{s_ja_j})\, c''_j + \\ &\qquad \left(w_0^{s_j,a_j} + \gamma(1-w_0^{s_ja_j}) - (1-\mathcal{P}^{\mathcal{M}}_{s_ja_j}) - \gamma\, \mathcal{P}^{\mathcal{M}}_{s_ja_j} \right) \|\Delta_j\|\,. \end{aligned}$$

We define

$$\begin{aligned} f_j(s_j,a_j) &= \left(w_0^{s_j,a_j} + \gamma(1-w_0^{s_ja_j}) - (1-\mathcal{P}^{\mathcal{M}}_{s_ja_j}) - \gamma\, \mathcal{P}^{\mathcal{M}}_{s_ja_j} \right) \|\Delta_j\| \\ &\qquad + |e_j(s_j,a_j)| + \gamma(1-w_0^{s_ja_j})\, c''_j\,. \end{aligned}$$

Note that $\lim_{j\to\infty} f_j = 0$, since e_j and c''_j converge to 0 and $w_0^{s_j,a_j}$ converges to $1 - \mathcal{P}^{\mathcal{M}}_{s_j a_j}$. Using this definition and (C.16), we can write

$$|F_j(s_j, a_j, \text{'bm'})| \leq \kappa^{s_j a_j} \|\Delta_j\| + f_j(s_j, a_j)\,. \tag{C.23}$$

Note that $\kappa^{s_j a_j} < 1$. From (C.22) and (C.23) it follows that the third condition of Lemma 7 is also satisfied. Hence, all conditions hold and Δ_j converges to 0 w.p.1.Combining this with (C.21), proves Lemma 6. □

C.4.4 Proof of Theorem 6

Because U'_j converges to U^* (Lemma 5) and U_j converges to U'_j (Lemma 6), it follows that also U_j converges to U^*. From this it follows that Q converges to Q^*, proving Theorem 6.

C.5 Lemma 7

Lemma 7. *Consider a stochastic process $(\alpha_t, \Delta_t, F_t)$, $t \geq 0$, where $\alpha_t, \Delta_t, F_t : X \to \mathbb{R}$ satisfy the equations:*

$$\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \alpha_t(x)F_t(x)\,,$$

where $x \in X$ and $t = 0, 1, 2, \ldots$. Assume that the following conditions hold:

1. *The set X is finite.*
2. *$\alpha_t(x) = [0, 1]$, $\sum_t \alpha_t(x) = \infty$.*
3. *$\|F_t\| \leq \kappa\|\Delta_t\| + c_t$, where $\kappa \in [0, 1)$ and c_t converges to zero w.p. 1 ,*

where $\|\cdot\|$ denotes a maximum norm. Then Δ_t converges to zero with probability one.

Note that this lemma is similar to Lemma 4, but the conditions for the learning rates are less strict ($\sum_t(\alpha_t(x_t))^2 < \infty$ is missing), while the condition for F_t is more strict (condition 3 uses the value of F_t instead of its expected value).

Proof. The outline of this proof is that we define a related process Δ'_t that converges to 0 and show that $\|\Delta_t\| \leq \|\Delta'_t\|$ for all t. We will ignore c_t in this proof. This can be safely done, since c_t converges to zero, $\kappa < 1$ and $\sum_t \alpha_t(x) = \infty$ for all x. Therefore, this term is asymptotically unimportant.

We define $\Delta'_0(x) = \|\Delta_0\|$ for all x. For $t > 0$, $\Delta'_t(x)$ is defined as

$$\Delta'_{t+1}(x) = (1 - \beta_t(x))\Delta'_t(x) + \beta_t(x)\kappa\|\Delta'_t\|\,, \tag{C.24}$$

with $\beta_t(x) \leq \alpha_t(x)$ and $\beta_t(x) \in [0, 1]$, $\sum_t \beta_t(x) = \infty$, $\sum_t(\beta_t(x))^2 < \infty$ w.p.1. It follows from (C.24) that $\|\Delta'_{t+1}\| \leq \|\Delta'_t\|$. It also follows that if $\Delta'_t(x) \geq \kappa\|\Delta'_t\|$ then $\Delta'_{t+1}(x) \geq \kappa\|\Delta'_t\| \geq \kappa\|\Delta'_{t+1}\|$. And since $\Delta'_0(x) \geq \kappa\|\Delta'_0\|$ it follows that

$$\Delta'_t(x) \geq \kappa\|\Delta'_t\|\,, \qquad \text{for all } t\,. \tag{C.25}$$

Using Lemma 17, it can easily be shown that Δ' converges in the limit to 0 w.p.1.

We now prove that $\|\Delta_t\| \leq \|\Delta'_t\|$ for all t. We start by proving

$$|\Delta_t(x)| \leq \Delta'_t(x) \quad \text{for all } x \quad \Rightarrow \quad |\Delta_{t+1}(x)| \leq \Delta'_{t+1}(x) \quad \text{for all } x\,. \tag{C.26}$$

Assuming the left part of (C.26), for $|\Delta_{t+1}(x)|$ the following holds:

$$\begin{aligned} |\Delta_{t+1}(x)| &\leq (1-\alpha_t(x))|\Delta_t(x)| + \alpha_t(x)\,\kappa\|\Delta_t\| \\ &\leq (1-\alpha_t(x))\Delta'_t(x) + \alpha_t(x)\,\kappa\|\Delta'_t\|\,. \end{aligned}$$

Since (C.25) and $\beta_t(x) \leq \alpha_t(x)$, we can continue as

$$\begin{aligned} |\Delta_{t+1}(x)| &\leq (1-\beta_t(x))\Delta'_t(x) + \beta_t(x)\,\kappa\|\Delta'_t\| \\ &\leq \Delta'_{t+1}(x)\,. \end{aligned}$$

This proves (C.26). And since $|\Delta_0(x)| \leq \Delta'_0(x)$, it follows that $|\Delta_t(x)| \leq \Delta'_t(x)$ holds for all t, and hence, $\|\Delta_t\| \leq \|\Delta'_t\|$ proving the lemma. □

C.6 Theorem 7

Theorem 7 Consider the MDP $M = \langle \mathbf{X}, A, T, R \rangle$. A subset of features $\mathbf{Y} \subset \mathbf{X}$ is valid w.r.t. M if for the set of missing features the following holds

$$\forall X_i \notin \mathbf{Y} : X_i \text{ is irrelevant w.r.t. } \mathbf{Y} \text{ or } X_i \text{ is an independent feature.} \tag{C.27}$$

Proof. For the proof we will make use the following formulas that can be easily deduced from the Bayesian statistics rules:

$$P(a|b,c) \cdot P(b|c) = P(a,b|c) \tag{C.28}$$

$$P(a|b) = P(a|b,c,d) \quad \Rightarrow \quad P(a|b) = P(a|b,c) \tag{C.29}$$

$$P(a,b_i|c) = P(a,b_i|c,d) \quad \text{for all } i \quad \Rightarrow \quad P(a|c) = P(a|c,d) \tag{C.30}$$

with $P(b_i,b_j) = 0$ for all $i \neq j$ and $\sum_i P(b_i) = 1$.

Let $\mathbf{X}_{ind} \subset \mathbf{X}$ be the set of all independent features that are not in $\mathbf{Y}$ and $\mathbf{X}_{irr} \subset \mathbf{X}$ be the set of all irrelevant features w.r.t. $\mathbf{Y}$ that are not in $\mathbf{Y}$. We start by proving that the Markov property holds for $\mathbf{Z} = \mathbf{X} \setminus \mathbf{X}_{ind}$ and then prove it holds for $\mathbf{Y} = \mathbf{Z} \setminus \mathbf{X}_{irr}$.

Let $X_i \in \mathbf{X}_{ind}$ be an independent feature and let $\mathbf{X}^- = \mathbf{X} \setminus X_i$. We start with the Markov property for $\mathbf{X}$:

$$P(\mathbf{x}_{t+1}, r_{t+1}|\mathbf{x}_t, a_t) = P(\mathbf{x}_{t+1}, r_{t+1}|\mathbf{x}_t, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, \ldots, r_1, \mathbf{x}_0, a_0)$$

$$P(\mathbf{x}^-_{t+1}, r_{t+1}|\mathbf{x}_t, a_t) = P(\mathbf{x}^-_{t+1}, r_{t+1}|\mathbf{x}_t, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, \ldots, r_1, \mathbf{x}_0, a_0) \quad \text{using (C.30)}$$

We now multiply both sides with $P(x_t^i|\mathbf{x}_t^-, a_t)$, where $x_t^i \in X_i$, and rewrite the left part as

$$P(\mathbf{x}_{t+1}^-, r_{t+1}|\mathbf{x}_t, a_t) \cdot P(x_t^i|\mathbf{x}_t^-, a_t) = P(\mathbf{x}_{t+1}^-, r_{t+1}, x_t^i|\mathbf{x}_t^-, a_t) \quad \text{using (C.28)}$$

For the right part, we rewrite $P(x_t^i|\mathbf{x}_t^-, a_t)$ as

$$P(x_t^k|\mathbf{x}_t^-, a_t) = P(x_t^i|\mathbf{x}_t^-, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, ..., r_1, \mathbf{x}_0, a_0) \quad \text{using (5.2)}$$

Multiply this with the right part and rewriting it using (C.28) gives:

$$P(\mathbf{x}_{t+1}^-, r_{t+1}, x_t^i|\mathbf{x}_t^-, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, ..., r_1, \mathbf{x}_0, a_0)$$

Combining now the left and right part again gives:

$$\begin{aligned}
&P(\mathbf{x}_{t+1}^-, r_{t+1}, x_t^i|\mathbf{x}_t^-, a_t) = P(\mathbf{x}_{t+1}^-, r_{t+1}, x_t^i|\mathbf{x}^-{}_t, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, ..., r_1, \mathbf{x}_0, a_0)\\
&P(\mathbf{x}_{t+1}^-, r_{t+1}|\mathbf{x}_t^-, a_t) = P(\mathbf{x}_{t+1}^-, r_{t+1}|\mathbf{x}^-{}_t, a_t, r_t, \mathbf{x}_{t-1}, a_{t-1}, ..., r_1, \mathbf{x}_0, a_0) \text{ using (C.30)}\\
&P(\mathbf{x}_{t+1}^-, r_{t+1}|\mathbf{x}_t^-, a_t) = P(\mathbf{x}_{t+1}^-, r_{t+1}|\mathbf{x}_t^-, a_t, r_t, \mathbf{x}_{t-1}^-, a_{t-1}, ..., r_1, \mathbf{x}_0^-, a_0) \text{ using (C.29)}
\end{aligned}$$

By repeating this for each independent feature in $\mathbf{X}_i$, the following equation is obtained.

$$P(\mathbf{z}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t) = P(\mathbf{z}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t, r_t, \mathbf{z}_{t-1}, a_{t-1}, \ldots, r_1, \mathbf{z}_0, a_0) \qquad \text{(C.31)}$$

where $\mathbf{z}_t = \mathbf{x}_t[\mathbf{Z}]$ and $\mathbf{Z} = \mathbf{X} \setminus \mathbf{X}_i$.

Starting from Equation C.31 we can prove the Markov property holds for $\mathbf{Y} = \mathbf{Z} \setminus \mathbf{X}_{irr}$ as following:

$$\begin{aligned}
&P(\mathbf{z}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t) = P(\mathbf{z}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t, r_t, \mathbf{z}_{t-1}, a_{t-1}, ..., r_1, \mathbf{z}_0, a_0)\\
&P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t) = P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t, r_t, \mathbf{z}_{t-1}, a_{t-1}, ..., r_1, \mathbf{z}_0, a_0) && \text{using (C.30)}\\
&P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{y}_t, a_t) = P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{z}_t, a_t, r_t, \mathbf{z}_{t-1}, a_{t-1}, ..., r_1, \mathbf{z}_0, a_0) && \text{using (5.1)}\\
&P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{y}_t, a_t) = P(\mathbf{y}_{t+1}, r_{t+1}|\mathbf{y}_t, a_t, r_t, \mathbf{y}_{t-1}, a_{t-1}, ..., r_1, \mathbf{y}_0, a_0) && \text{using (C.29)}
\end{aligned}$$

This last equation proves that $\mathbf{Y}$ is a valid representation (see Definition 14). □

Publications by the Author

Almost all research described in this thesis is based on previous publications by the author, with the exception of the research on policy restrictions (Chapter 6), which has not been submitted yet to any conference or journal. Below we relate the different research topics to their corresponding publications.

The theoretical and empirical evaluation of Expected Sarsa (Section 3.1), has been published at the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (van Seijen et al., 2009).

Just-in-time Q-learning (Section 3.2) together with best-match learning (Chapter 4) has been published by the Journal of Machine Learning Research (JMLR) (van Seijen et al., 2011). The combination of just-in-time learning with (Expected) Sarsa (Section 3.3) is new material. An early version of the work on just-in-time learning and best-match learning was published at the International Conference on Intelligent Systems Design and Applications (van Seijen and Whiteson, 2009). In this paper just-in-time learning was referred to as 'basic postponing', while the BM-LVM method (Algorithm 10) was called 'model-free prioritized sweeping'. The JMLR article extended this research in a major way by introducing the generalized best-match equations (see Definition 6) from which BM-NTM (Algorithm 11) is derived, which can tune its space requirements, as well as gradient-descent best-match learning (Algorithm 12).

The research on representation selection (Chapter 5) has a long history. Initial work was presented at a NIPS workshop and later published at the Artificial Intelligence and Applications Conference (van Seijen et al., 2008). This paper presented representation selection informally and showed promising empirical results on two specific tasks. However, extending these results to a general class of tasks turned out to be far from trivial, due to convergence issues. After the initial work the research was put on hold for a long period of time, during which best-match learning was developed. When the work on representation selection continued, the research focus was mainly on the conditions under which convergence could be achieved. This resulted in a book chapter (van Seijen et al., 2010), in which representation selection was presented in a formal way. The research in Chapter 5 improves and extends the research from this book chapter in a major way, by proving that under certain conditions the derived task resulting from representation selection obeys the Markov property, and hence can be solved with regular reinforcement learning methods. In addition, representation selection for MDPs with context-specific structure (Section 5.4) was added. The research as presented by Chapter 5 is currently under review for journal publication.

Bibliography

C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning. *Artificial intelligence review*, 11(1):11–73, 1997. 32

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002. 9, 76

H.B. Barlow. Unsupervised learning. *Neural Computation*, 1(3):295–311, 1989. 1

R. Bellman. A problem in the sequential design of experiments. *Sankhy*, 16:221–229, 1956. 2

R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ., 1957. 3, 15

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA, 1996. 1

C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. 1

C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 1104–1113, 1995. 6, 71, 72, 75

C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations* 1. *Artificial Intelligence*, 121(1-2):49–107, 2000. 103

J. Boyan and A.W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, 1995. 65

R.I. Brafman and M. Tennenholtz. R-max: a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002. 3, 15, 53

D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731. Citeseer, 1991. 103

R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, pages 72–83, 2006. 130

R.H. Crites and A.G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2):235–262, 1998. 1, 10

T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(2):142–150, 1989. 6, 71

T. Dean, R. Givan, and S. Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 124–131. Citeseer, 1997. 103

Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000. 103

C. Diuk, L. Li, and B.R. Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009. 3, 6, 15, 71, 102

D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(1):503–556, 2005. 68

N. Ferns, P. Panangaden, and D. Precup. Metrics for finite Markov decision processes. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 162–169. AUAI Press, 2004. 103

R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003. 103

M. Grzes and D. Kudenko. Learning shaping rewards in model-based reinforcement learning. In *Proc. AAMAS 2009 Workshop on Adaptive Learning Agents*, 2009. 115

Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research*, 19:399–468, 2003. 75

J.A. Hartigan. *Clustering algorithms*. John Wiley & Sons, Inc., 1975. 1

T. Jaakkola, M.I. Jordan, and Satinder Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994. 18, 92, 133

Nicholas K. Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 752–757, 2005. 103

Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. 1, 2, 9

Shivaram Kalyanakrishnan and Peter Stone. Characterizing reinforcement learning methods through parameterized learning problems. *Machine Learning*, 84(1–2):205–247, July 2011. 7

M. Kearns and D. Koller. Efficient reinforcement learning in factored mdps. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 740–747, 1999. 6, 71

M. Kearns and S. Singh. Finite-sample convergence rates for Q-learning and indirect algorithms. *Advances in Neural Information Processing Systems*, 11:996–1002, 1999. ISSN 1049-5258. 6, 41

Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2):209–232, 2002. 3, 15, 53

L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *15th European Conference on Machine Learning*, pages 282–293, 2006. 130

G. Konidaris and A. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 489–496, 2006. 115

Mark Kroon and Shimon Whiteson. Automatic feature selection for model-based reinforcement learning in factored MDPs. In *ICMLA 2009: Proceedings of the Eighth International Conference on Machine Learning and Applications*, pages 324–330, December 2009. 6, 71, 102

M.G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1149, 2003. 68

T.L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985. 9, 76

J. Langford and T. Zhang. The epoch-greedy algorithm for contextual multi-armed bandits. *Advances in Neural Information Processing Systems*, 2007. 10, 76

J. Langford, A. Strehl, and J. Wortman. Exploration scavenging. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning*, pages 528–535. ACM, 2008. 10, 76

L. Li, M.L. Littman, and T.J. Walsh. Knows what it knows: a framework for self-aware learning. In *Proceedings of the 25th international conference on Machine learning*, pages 568–575. ACM New York, NY, USA, 2008. 6, 71, 102

L.J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992. 41, 48

L.J. Lin. *Reinforcement learning for robots using neural networks*. PhD thesis, 1993. 1

H.R. Maei and R.S. Sutton. Gq (λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *AGI*, pages 91–96. Citeseer, 2010. 18

Andrew Kachites McCallum. *Reinforcemennt Learning with Selective Perception and Hidden States*. PhD thesis, University of Rochester, 1995. 103

Andrew Moore and Christopher Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993. 3, 15, 42, 52

D.E. Moriarty, A.C. Schultz, and J.J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11(241-276), 1999. 7

A.Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning*, pages 278–287, 1999. 115

S. Pandey, D. Agarwal, D. Chakrabarti, and V. Josifovski. Bandits for taxonomies: A model-based approach. In *SIAM Data Mining Conference*. Citeseer, 2007. 76

J. Peng and R.J. Williams. Incremental multi-step q-learning. *Machine Learning*, 22(1): 283–290, 1996. 5, 18

D. Precup. *Temporal abstraction in reinforcement learning*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, USA, 2000. 105, 115

M.L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. 1994. 2, 10

B. Ravindran and A.G. Barto. SMDP homomorphisms: An algebraic approach to abstraction in semi-Markov decision processes. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1011–1018. Citeseer, 2003. 103

G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, Tech. rep. CUED/F-INENG/TR166, Cambridge University, 1994. 3, 15

Gavin Adrian Rummery. *Problem Solving With Reinforcement Learning*. PhD thesis, University of Cambridge, 1995. 5, 19

J. Rust. Using randomization to break the curse of dimensionality. *Econometrica*, 65(3): 487–516, 1997. 130

L. Schomaker. Using stroke-or character-based self-organizing maps in the recognition of on-line, connected cursive script. *Pattern Recognition*, 26(3):443–450, 1993. 1

D. Silver. *Reinforcement learning and simulation-based search in computer Go*. PhD thesis, Department of Computing Science, University of Alberta, Canada, 2009. 130

S. Singh, T. Jaakkola, M.L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforecement-learning algorithms. *Machine Learning*, 38:287–308, 2000. 21, 92, 142

Matthijs Snel and Shimon Whiteson. Multi-task evolutionary shaping without pre-specified representations. In *GECCO 2010: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1031–1038, July 2010. 115

Matthijs Snel and Shimon Whiteson. Multi-task reinforcement learning: Shaping and feature selection. In *EWRL 2011: Proceedings of the Ninth European Workshop on Reinforcement Learning*, September 2011. 115

K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. 7

Martin Stolle and Doina Precup. Learning options in reinforcement learning. *Lecture Notes in Computer Science*, 2371:212–223, 2002. 105, 115

Alexander Strehl and Michael Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 856–863, 2005. 3, 15, 53

Alexander L. Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L. Littman. PAC model-free reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 881–888, 2006. 3, 15, 41, 53, 68

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988. 3, 15, 17, 41

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990. 3, 12, 15, 34

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1045, 1996. 3, 15, 65

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachussets, 1998. 1, 2, 5, 9, 19, 25, 52, 64, 65, 82, 133, 134

Richard S. Sutton and Satinder P. Singh. On step-size and bias in temporal-difference learning. In *Proceedings of the Eight Yale Workshop on Adaptive and Learning Systems*, 1994. 48, 133

R.S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999. 103, 105, 115

M.E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *The Journal of Machine Learning Research*, 10:1633–1685, 2009. 103

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994. 1

Harm van Seijen and Shimon Whiteson. Postponed updates for temporal-difference reinforcement learning. In *ISDA 2009: Proceedings of the Ninth International Conference on Intelligent Systems Design and Applications*, pages 665–672, November 2009. 153

Harm van Seijen, Bram Bakker, and Leon Kester. Switching between different state representations in reinforcement learning. In *Proceedings of the Artificial Intelligence and Applications Conference*, 2008. 153

Harm van Seijen, Hado van Hasselt, Shimon Whiteson, and Marco Wiering. A theoretical and empirical analysis of expected sarsa. In *ADPRL 2009: Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 177–184, March 2009. 153

Harm van Seijen, Shimon Whiteson, and Leon Kester. Switching between representations in reinforcement learning. In Robert Babuska and Frans Groen, editors, *Interactive Collaborative Information Systems*, Studies in Computational Intelligence, pages 65–84. Springer, Berlin, Germany, 2010. 153

Harm van Seijen, Shimon Whiteson, Hado van Hasselt, and Marco Wiering. Exploiting best-match equations for efficient reinforcement learning. *Journal of Machine Learning Research*, 12:2045–2094, 2011. 153

V Vapnik. *The nature of statistical learning theory*. Springer, 1995. 1

C.C. Wang, S.R. Kulkarni, and H.V. Poor. Bandit problems with side observations. *IEEE Transactions on Automatic Control*, 50:338–355, 2005. 76

C. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, England, 1989. 3, 5, 15, 17, 18, 41, 53, 133

Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):9–44, 1992. 32

Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 21(1):1–27, 2010. 7

Marco Wiering and Jürgen Schmidhuber. Fast online Q(λ). *Machine Learning*, 33:105–115, 1998. 32

E. Wiewiora. Potential-based shaping and q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19(1):205–208, 2003. 115

N. Zhang and D. Poole. On the role of context-specific independence in probabilistic inference. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1288–1293, 1999. 75

Summary

Many important real-life tasks can be modeled as an *agent* (which represents some decision making process) that interacts with its *environment* by taking a sequence of actions on different moments in time. Often, the effect of an action is (initially) unknown. The task of finding a good policy for an agent that interacts with an initially unknown environment is the research domain that *reinforcement learning* (RL) is concerned with.

The challenges presented by such a domain should not be underestimated. In general, the number of possible policies grows exponentially with the number of problem parameters. Besides that, the number of environment states and/or the number of actions can be infinite and the current environment state can only be partially observable. Despite these challenges, there are multiple examples of successful RL applications in practise, and the potential rewards that lie in broadening this application domain makes RL an active area of research.

The research focus of this thesis lies on methods that are efficient with respect to the computation time and memory space they require. Efficiency with respect to these resources is important, since any practical system has only a limited amount of resources. Hence, the efficiency of methods with respect to these resources determines the problem size a system can deal with.

This thesis discusses the problem of efficiency in the context of *Markov decision processes* (MDPs), a core formalism for describing RL tasks. An effective strategy for solving MDPs is to use *value functions*, that predict the payoff for each state-action combination. Value-function methods improve their policy by iteratively improving a value function, using samples collected from interaction with the environment. This thesis has three main contributions, in the form of three ideas that build on the value-function theory.

The first one is *best-match learning*, which builds on *temporal-difference* (TD) learning, an important strategy for efficient RL. TD methods have at their core an update that computes a new value for a state-action pair by taking a weighted average between its old value and an update target, constructed from sample information and values of other state-action pairs. To improve the sample efficiency, i.e., to decrease the number of environment samples required to obtain a good policy, a popular strategy is to use the same sample multiple times for an update, as if it was observed more than once. To avoid an unfair bias towards such a sample, which hurts performance, every sample receives a similar number of additional updates. The idea behind a best-match update is that instead of performing additional updates with the same sample, previous updates with that sample are *corrected*. Like the strategy of additional updates, the strategy of correcting updates improves the sample efficiency. However, in contrast to additional updates, correcting updates removes the constraint that every sample has to receive a similar number of updates.

This idea is remarkably powerful. It allows for the construction of new RL methods with unique properties. For example, it allows for methods that can tune their time and

space requirements to the available amount of resources, while maintaining a guarantee of convergence to the optimal policy. Earlier methods with the property of tuning their space requirements could only achieve this at the cost of losing the convergence guarantee. Empirically, this means that methods using best-match updates can substantially outperform these earlier methods, while using the same amount of space and time resources.

The second idea this thesis introduces involves switching on-line between state representations. The state representation of a problem defines which environment features the agent can observe. The task of selecting the right features is a typical one for the problem designer. Selecting good features is an important and difficult task, since it not only determines the maximum performance that can be obtained by an agent, but also the size of the problem. Our proposed strategy removes part of this burden from the designer, by allowing the designer to specify multiple *candidate representations*. The agent then determines on-line the most effective representations as well as an optimal policy corresponding to that representation.

The final idea deals with *policy restrictions*. Like the idea involving switching between representations, at its core this idea is about exploiting prior knowledge. Exploiting prior knowledge effectively reduces the problem size, allowing for an improved sample efficiency for the same space and time requirements. For the idea of switching between representations, prior knowledge about potentially effective feature sets is exploited. The prior knowledge that is exploited in this final idea is knowledge about restrictions on the policy set that should be considered when searching for an optimal policy. These restrictions form a natural way to specify prior knowledge about a task. In case of policy-search RL, that is, RL methods that search for the optimal policy directly in the policy space, the advantage of a smaller policy set is obvious. However, policy-search methods excel in different task domains than value-function methods. Therefore, for task domains where value-function methods are superior, one would like to use value-function methods but still be able to exploit prior knowledge about policy restrictions. This thesis presents and evaluates a strategy that can achieve this. We demonstrate that the range of restrictions that can be effectively described using the proposed strategy is larger than that of existing alternatives.

This thesis provides a thorough theoretical analysis of each of the above ideas, gives intuitive examples for understanding the benefits of each approach and provides extensive empirical comparisons with relevant competitive techniques. The thesis concludes by answering in detail a number of specific questions related to reinforcement learning under space and time constraints and by discussing the three most promising avenues of future work that came out of this research.

Samenvatting

Veel situaties uit het alledaagse leven kunnen worden gezien als een beslissingsproces dat op verschillende tijdstippen keuzes maakt en acties uitvoert. Deze acties veroorzaken bepaalde veranderingen in de omgeving, terwijl niet altijd van tevoren bekend is wat voor veranderingen dit zijn. De taak van het vinden van een goede strategie voor het beslissingsproces in zo'n situatie is het onderzoeksdomein waar *reinforcement learning* (RL) zich mee bezig houdt. Het beslissingsproces wordt de *agent* genoemd en de strategie van de agent heet de *policy*.

Het onderzoeksdomein van reinforcement learning bevat veel uitdagingen. Zo kan het totaal aantal omgevingstoestanden oneindig zijn, en ook het aantal acties waaruit de agent kan kiezen. Bovendien groeit het aantal mogelijke policies, in het algemeen, exponentieel met het aantal probleem parameters, waardoor al snel een zeer groot probleem kan ontstaan dat erg lastig op te lossen is. Toch zijn er verschillende succesvolle RL toepassingen in de praktijk, en de belofte van meer succesvolle toepassingen maakt dat RL een actief onderzoeksgebied is.

Het onderzoek waar dit proefschrift over gaat houdt zich bezig met RL methodes die efficiënt zijn in het gebruik van de beschikbare rekenkracht en geheugenruimte. Efficiëntie ten opzichte van deze *resources* is belangrijk, omdat elk systeem in de praktijk maar beperkte resources heeft. Daarom zullen methodes die deze resources optimaal kunnen benutten in staat zijn grotere problemen op te lossen.

Een formele manier om RL problemen te beschrijven is door middel van *Markov decision processes* (MDPs). Een veelgebruikte techniek om deze MDPs op te lossen, is het gebruik van *value functions*, die een voorspelling geven van hoe voordelig het is om een bepaalde actie in een bepaalde situatie te nemen. Methodes gebaseerd op value functions verbeteren hun policy door stap voor stap de voorspellingen die de value function geeft te verbeteren, gebruik makend van samples. Een sample is opgebouwd uit informatie die verkregen is door een enkele interactie met de omgeving. De bijdrage van dit proefschrift bestaat uit drie nieuwe ideeën, gebaseerd op de value function theorie.

Het eerste idee noemen we *best-match learning*, dat voortborduurt op een belangrijke techniek voor efficiënte RL, genaamd *temporal-difference* (TD) learning. TD methodes gebruiken een update, om een nieuwe waarde van de value function te berekenen, die gebaseerd is op een gewogen gemiddelde tussen een oude waarde en een update target, dat een nieuwe voorspelling geeft van hoe voordelig een actie is gebaseerd op een sample. Om de *sample efficiëntie* te verbeteren, dat wil zeggen, om het aantal samples dat nodig is om een goede policy te berekenen te verkleinen, wordt vaak een techniek gebruikt waarbij hetzelfde sample meerdere keren wordt gebruikt voor een update (alsof de agent dit sample meer dan een keer heeft geobserveerd). Om een bias ten opzichte van dit sample te voorkomen (wat een slechtere sample efficiëntie tot gevolg heeft) ontvangt elk sample dat door de agent wordt geobserveerd een vergelijkbaar aantal extra updates. Het idee achter

best-match learning is dat in plaats van dat een sample meerdere keren wordt gebruikt voor extra updates, dit sample wordt gebruikt om eerdere updates te corrigeren. Het effect hiervan is dat de sample efficiëntie wordt verbeterd, zoals ook gebeurt als extra updates worden uitgevoerd, alleen is het niet langer nodig dat elk sample een vergelijkbaar aantal updates gebruikt. Hierdoor is het mogelijk nieuwe methodes te maken met unieke eigenschappen.

Een voorbeeld van zo'n nieuwe methode is een methode waarbij de benodigde rekenkracht en geheugenruimte kunnen worden aangepast aan de beschikbare hoeveelheid resources, terwijl convergentie naar een optimale policy wordt gegarandeerd. Eerdere methodes waarbij de benodigde rekenkracht en geheugenruimte kan worden aangepast hebben deze garantie niet. In de praktijk komt dit erop neer, dat een methode gebaseerd op best-match learning een betere sample efficiëntie behaalt dan deze eerdere methodes, terwijl de benodigde rekenkracht en geheugenruimte vergelijkbaar is.

Het tweede idee dat dit proefschrift beschrijft gaat over het switchen tussen verschillende omgevings-representaties. De omgevings-representatie bepaalt welke features van de omgeving een agent kan observeren. Normaal gesproken is de *problem designer*, dat wil zeggen, de persoon die een taak definieert, verantwoordelijk voor het uitkiezen van deze features. Dit is een erg belangrijk onderdeel, omdat de features de maximale sample efficiëntie bepalen die kan worden behaald en bovendien de grootte van een probleem bepaalt. Een goede representatie bestaat uit features die nuttige informatie bevatten maar die er ook voor zorgen dat het totaal aantal omgevings-toestanden niet te groot wordt, wat het moeilijker maakt een probleem op te lossen. De techniek die in dit proefschrift wordt besproken neemt een deel van de verantwoordelijkheid van de problem designer uit handen, door toe te staan dat verschillende kandidaat representaties worden gedefinieerd. De RL agent bepaalt vervolgens zelf wat de beste representatie is en wat de optimale policy is voor die representatie.

Het laatste idee dat dit proefschrift beschrijft gaat over *policy restricties*. Net als het tweede idee, gaat ook dit idee over het benutten van bepaalde voorkennis. In het algemeen is het gebruiken van voorkennis nuttig, omdat de grootte van het probleem hierdoor kleiner wordt. Daardoor kan, met dezelfde hoeveelheid rekenkracht en geheugenruimte, een beter sample efficiëntie worden behaald. In het geval van switchen tussen representaties was dit voorkennis over mogelijk effectieve feature sets. In het geval van dit laatste idee gaat het over voorkennis van restricties ten aanzien van de totale policy set van een MDP. Als bijvoorbeeld van tevoren bekend is dat bepaalde policies slecht zijn, kunnen deze uit de policy set worden gehaald zodat de zoekruimte waarbinnen naar de optimale policy wordt gezocht, kleiner wordt. Het probleem bij RL gebaseerd op value-functions, waar dit proefschrift over gaat, is dat niet rechtstreeks naar de optimale policy wordt gezocht, maar indirect, door te zoeken naar de optimale value function. Dit laatste idee gaat erover hoe, terwijl er wordt gezocht naar een optimale value function, toch voorkennis kan worden benut dat is uitgedrukt in termen van policy restricties. Hoewel er ook andere methodes zijn die dit kunnen bereiken, kan met de methode die wij voorstellen, een veel grotere hoeveelheid voorkennis worden benut, wat een betere sample efficiëntie tot gevolg heeft.

In dit proefschrift geven we, naast een uitgebreide theoretische analyse van elk idee, ook intuïtieve voorbeelden, alsmede uitgebreide empirische vergelijkingen met relevante andere technieken. Het proefschrift eindigt met het geven van een gedetailleerd antwoord

op enkele specifieke vragen ten aanzien van reinforcement learning methodes die efficiënt zijn met betrekking tot de rekenkracht en geheugenruimte, en het geven van een aantal suggesties voor vervolg-onderzoek.

Acknowledgements

The motivation for starting a Ph.D. and route towards it is different for every Ph.D. student. In my case, I took the rather unusual route of finishing a master in applied physics, then working at TNO as a research scientist for four years, and after that starting a Ph.D. in reinforcement learning. I want to use this section to recount the story behind this route and to credit the people along the way that have helped me go the distance.

The route towards this thesis began in the summer of 2002 in California, where I was working as a graduate intern at a company that makes scanning probe microscopes. While really enjoying my stay in California, I was very aware of the fact that I was close to finishing my master in applied physics. Therefore, I spend some time contemplating my future career and figuring out what I was looking for in a job. This made me realize that, although in general I liked physics and doing research, it was not a perfect fit. I felt there was something missing, although I couldn't really pinpoint what it was.

So, in order to get a better understanding of what I was looking for, I spend some of my free time reading about different types of research and different types of jobs. This continued until, on one afternoon, I came across an article in the New Scientist called 'Go for it'. The article narrated the story of Michael Reiss, a guy that had spend more than a decade working on a computer program that plays the game of Go. The article continued to describe the challenges involved in trying to program a top-level Go-playing computer and the possible consequences of achieving this for the field of artificial intelligence.

The article intrigued me on multiple levels. How can a game constructed from such simple rules pose such a challenge? Also, if humans are so good in it, why can't they formulate the strategy that brings them this success. And finally, apparently it is possible to make a living, albeit a modest one, by working on such problems. By the time I was done reading, the mystery surrounding the type of intelligence a game of Go required and the potential implications of being able to capture this in a computer program had won me over completely.

While the article marked the end of a long search period and gave me some inner peace, there was the inevitable follow-up question "OK, so now what?". I was close to having a degree in applied physics and was looking forward to getting off the tight student budget and finally start making some money. Besides that, my knowledge of AI techniques was only rudimentary. So, landing a job in AI or starting all over again were no realistic scenarios. After evaluating my different options, I eventually applied for a job at TNO, a large Dutch research institute, where a wide variety of research topics is covered. I figured that working in such a versatile environment would allow me to move from physics in the direction of AI over time.

I started working in the Electro-Optics group, headed by Jan Olijslager. While there were mainly physics projects in this group, I also got to work on several AI-related projects over the years. However, a returning theme at the yearly performance evaluations I had with

Jan was that I wanted to work more on AI. At the fourth yearly evaluation, Jan confronted me with this returning theme and gave the advice that if I was really serious about my plans to work in AI, I should stop talking about it and start taking some action. This was a much needed push in the right direction, for which I'm thankful to this day. I made up my mind and finally felt ready to leave my comfortable surroundings, and make some bold and uncertain moves to pursue my wish to do research in AI fulltime. About a month after this conversation, when I was about to quit my job at TNO, I heard that there was a position coming available within TNO to work fulltime as a Ph.D. student on an artificial intelligence related topic. Obviously, I was interested.

The interview for the position was with Leon Kester, the co-promotor and technology expert at the Distributed Sensor Systems group at TNO, and Frans Groen, the promotor and head of the Intelligent Autonomous Systems group at the University of Amsterdam (UvA). While I didn't have a strong background in AI at that point, they were willing to give me a chance for which I want to thank them both sincerely. I also want to thank them for providing a comfortable working environment throughout my Ph.D. period and giving me some freedom in terms of the research topic, as well as for the many stimulating conversations we had about the research.

So, in may 2007, almost five years after reading the Go article, I was in a position where I could spend 100% of my time on AI research. This ended the challenge to find projects related to AI, but started a new one, since the road towards a Ph.D. degree is one full of pitfalls.[6] In this context, there are two persons in particular I want to thank for guiding me around many of them.

The first one is Bram Bakker, at the time a PostDoc at the UvA working on reinforcement learning (RL), whom I met about three months into my Ph.D. period. Until I met Bram, I had only a vague idea of what I wanted to do and my topic of research changed frequently (AI is a *very* broad area). Bram introduced me to reinforcement learning, a research area I immediately loved, and became, besides Leon and Frans, a third supervisor, with whom I met at a weekly basis to discuss about RL. My meetings with Bram were always inspiring and put my research into high gear at an early moment in my Ph.D. period.

The second one is Shimon Whiteson, who took over the role as weekly supervisor from Bram, about one year into my Ph.D. period, after Bram left UvA. Shimon proved time and again to be an excellent supervisor with whom I had many long and vivid discussions about RL and doing research in general, which helped me around several Ph.D. pitfalls. In addition, through his methodical feedback and high standards, I learned a lot from Shimon in terms of writing high quality papers.

I also want to thank Hado van Hasselt and Marco Wiering, coauthers on two papers. I worked closely together with Hado on the convergence proofs for the best-match LVM and NTM classes, two very hard nuts, which eventually we managed to crack. Marco was a valuable discussion partner and has been inspiring in terms of out-of-the-box ideas.

Finally, since life's all about balance, I want to thank my family, friends and colleagues at TNO as well as at the UvA for keeping my life in balance by providing the necessary

[6] An entertaining account of the many pitfalls a Ph.D. student faces can be found here: `http://homepages.inf.ed.ac.uk/bundy/how-tos/resbible.html`

support and relaxation in the form of (including, but not limited to) Dalmuti games, frisbee games, Take 5, beers, hiking trips (abroad, obviously), Take 4, dart games, more beers, Take 3? (no, thank you), random discussions, discussions about randomness, Lowlands, Nespresso moments combined with xkcd moments ('no mister Bond, I expect you to die!'), BBQs, crossloop challenges, nieuwjaarsduiken, city trips (♫ Barcelona! ♪), movies, dinners and random parties. :)