

Generic Reinforcement Learning Beyond Small MDPs

Mayank Daswani

A thesis submitted for the degree of
Doctor of Philosophy
of The Australian National University

August 2015

Certificate of originality

Except where otherwise indicated, this thesis is my own original work. Chapters 4,5 & 6 are based on publications co-authored with my supervisors, but in each case I was the primary author.

Mayank Daswani

18th August 2015

Acknowledgements

- To my supervisors Peter and Marcus: thank you for your guidance, support and close monitoring throughout my candidature.
- To my beautiful partner Jessica: thank you for being there when I needed you. You feel like breathing out.
- To my parents: thank you for your constant support and encouragement.
- To my fellow students Tor, Hadi, Jan, Tom, Daniel and many others: thank you for the discussions that kept me sane!
- To my sister Mansha: maybe one day we'll grow up.

Abstract

Feature reinforcement learning (FRL) is a framework within which an agent can automatically reduce a complex environment to a Markov Decision Process (MDP) by finding a map which aggregates similar histories into the states of an MDP. The primary motivation behind this thesis is to build FRL agents that work in practice, both for larger environments and larger classes of environments. We focus on empirical work targeted at practitioners in the field of general reinforcement learning, with theoretical results wherever necessary.

The current state-of-the-art in FRL uses suffix trees which have issues with large observation spaces and long-term dependencies. We start by addressing the issue of long-term dependency using a class of maps known as looping suffix trees, which have previously been used to represent deterministic POMDPs. We show the best existing results on the TMaze domain and good results on larger domains that require long-term memory.

We introduce a new value-based cost function that can be evaluated model-free. The value-based cost allows for smaller representations, and its model-free nature allows for its extension to the function approximation setting, which has computational and representational advantages for large state spaces. We evaluate the performance of this new cost in both the tabular and function approximation settings on a variety of domains, and show performance better than the state-of-the-art algorithm MC-AIXI-CTW on the domain POCMAN.

When the environment is very large, an FRL agent needs to explore systematically in order to find a good representation. However, it needs a good representation in order to perform this systematic exploration. We decouple both by considering a different setting, one where the agent has access to the value of any state-action pair from an oracle in a training phase. The agent must learn an approximate representation of the optimal value function. We formulate a regression-based solution based on online learning methods to build an such an agent. We test this agent on the Arcade Learning Environment using a simple class of linear function approximators.

While we made progress on the issue of scalability, two major issues with the FRL framework remain: the need for a stochastic search method to minimise the objective function and the need to store an uncompressed history, both of which can be very computationally demanding.

Contents

1	Introduction	1
1.1	A very brief history of AI ¹	1
1.2	Intelligent Agents	3
1.3	Challenges in generic RL	4
1.4	Contributions	6
1.5	Guide to the thesis	7
2	Background	9
2.1	Agent-Environment Framework	9
2.2	Reinforcement Learning	9
2.2.1	Multi-armed bandit problem	10
2.2.2	Markov Decision Process (MDP)	10
2.2.2.1	Planning	12
	Policy iteration (PI)	13
	Comparison of VI and PI	13
2.2.2.2	Learning	14
	Model-free methods	16
	TD-learning	16
	Monte-Carlo methods	17
	TD-algorithms with eligibility traces	18
	Policy search	18

¹For a more detailed history, see Russell and Norvig (2009) and Sutton and Barto (1998) which we draw on.

	Exploration vs Exploitation	19
	PAC-MDP algorithms	21
2.2.3	Function approximation	21
2.2.4	State aggregation	23
2.3	Partially observable MDPs	23
2.3.1	Perceptual Aliasing	24
2.3.2	Solving POMDPs	24
2.3.2.1	Known POMDPs	25
	Point-based based value iteration	26
	Monte-Carlo Tree Search planning for POMDPs	26
2.3.2.2	POMDPs with unknown parameters	26
	Baum-Welch based methods	26
	Finite state controllers	27
2.3.2.3	Unknown POMDPs	27
	The exacerbated exploration problem	27
	Concealment of crucial features	27
	Reactive policies for POMDPs	28
	History-based methods	28
2.3.3	History-POMDPs	29
2.4	Feature reinforcement learning	31
2.4.1	Feature dynamic Bayesian networks	34
2.4.1.1	Structure Learning	36
2.4.1.2	Feature Generation	36
2.4.1.3	Planning in Factored MDPs	37
2.4.1.4	Learning in Factored MDPs	37
2.5	Predictive State Representations (PSRs)	37
2.6	Connections to FRL	38
2.6.1	POMDPs	39
2.6.1.1	Belief states over histories	39
2.6.2	Function approximation in high-dimensional state spaces	40
2.6.3	State aggregation	40
2.6.4	PSRs	41
2.6.5	Representation learning	41

3	Domains for RL	43
3.1	Introduction	43
3.2	Preliminaries	44
3.3	Domains	45
3.3.1	Starcraft: Brood War	45
3.3.2	Poker	46
3.3.3	Chess	48
3.3.4	Go	50
3.3.5	ATARI 2600 Games	51
3.3.5.1	Arcade Learning Environment	51
3.3.5.2	Partially Observable PACMAN (POCMAN)	52
3.3.6	Hearts	53
3.3.7	Reinforcement Learning in Finance	54
3.3.8	Keepaway Problem	54
3.3.9	General Game Playing Competition	55
3.3.10	Small problems	57
3.3.10.1	Tiger	57
3.3.10.2	Cheese Maze	57
3.3.10.3	TMaze	58
3.3.10.4	Locked Door	58
3.4	Conclusion	59
4	Looping Suffix Trees	61
4.1	Introduction	61
4.1.1	Related Work	62
4.2	Preliminaries	62
4.3	Relation to hPOMDPs	63
4.4	Looping Suffix Trees in Φ MDP	68
4.5	Experiments	71
4.6	Analysis	73
4.7	Conclusion	78

5	A Model-free Off-policy Cost for ΦMDP Agents	79
5.1	Introduction	79
5.2	Formulation of the $Cost_{QL}$ function	80
5.3	Consistency of $Cost_{QL}$	82
5.4	Algorithm	84
5.4.1	Implementation of $Cost_{QL}$	84
5.4.2	Finding a good policy	85
5.4.3	Initialisation	86
5.4.4	Execution	86
5.4.5	Features	87
5.4.5.1	Suffix trees and Looping suffix trees	87
5.4.5.2	Event selectors	87
5.5	Experiments	88
5.5.1	Small domains with short-term memory	89
5.5.2	Long term memory domains	90
5.5.2.1	Deterministic TMaze	90
5.5.2.2	Stochastic TMaze	92
5.5.2.3	Locked door	92
5.5.2.4	Event selectors	94
5.5.3	A large domain : Pocman	94
5.6	Discussion	96
5.6.1	The utile distinction conjecture	96
5.6.2	Why use $Cost_{QL}$ instead of $OCost$?	97
5.6.3	Computational issues	98
5.6.4	Related Work	99
5.7	Conclusion	99

6	Reinforcement Learning with Value Advice	101
6.1	Introduction	101
6.2	Background	103
6.2.1	Supervised learning	103
6.2.1.1	Classification	104
6.2.1.2	Regression	104
6.2.1.3	Cost-sensitive multi-class classification	105
6.2.2	Convex optimisation	105
6.2.3	Online learning	106
6.2.3.1	Follow-the-leader (FTL)	106
6.2.4	Imitation learning	107
6.3	The RLAdvice algorithm	108
6.3.1	Algorithm and performance bound	109
6.3.2	Practical algorithm	112
6.4	Experiments	113
6.5	Discussion	116
6.5.1	General behaviour	116
6.5.2	RLAdvice vs SARSA	117
6.5.3	RLAdvice vs UCTRLA	117
6.5.4	RLAdvice vs DAgger	118
6.5.5	Summary of comparisons	118
6.5.6	Oracle accuracy	120
6.5.7	Stochastic Environments	120
6.5.8	Computational issues	120
6.6	Related Work	121
6.7	Conclusion	123
7	Conclusion and outlook	125
A	Omitted Proofs	131
B	Convex Optimisation	133
C	List of Notation	137
	References	141

List of Figures

1.1	Motivating examples	4
2.1	The agent-environment framework	10
2.2	A hard MDP	20
2.3	A stochastic hPOMDP	30
2.4	A suffix tree that maps strings that end in 00, 10 and 1 to s_0 , s_1 and s_2 respectively.	34
2.5	Connections to FRL	39
3.1	Popularity of some domains via Google Trends.	57
3.2	TMaze environment showing goal at left	58
3.3	Locked door domain	59
4.1	Deterministic POMDP\mathbb{R} without resolving sequences	65
4.2	Stochastic hPOMDP\mathbb{R} and corresponding LST	65
4.3	Flip automata for different conventions of writing POMDP\mathbb{R}s	66
4.4	Example (observation-based) suffix and looping suffix trees	69
4.5	A reward optimal LST for the TMaze problem	72
4.6	$LST\Phi MDP$ on TMaze length 50	72
4.7	Stochastic TMaze length 50	72
4.8	$LST\Phi$ MDP, RPG and RL-LSTM on Det. TMaze	73
4.9	Comparison on Tiger with $\text{prob}(\text{listen})=0.85$ for $LST\Phi MDP$ and $CT\Phi MDP$	73
4.10	$LST\Phi MDP$ on Locked door domain	76

5.1	Short-term memory domains	89
5.2	The best features selected on the Tiger domain	90
5.3	Long-term memory domains	91
5.4	Neighbours of the optimal looping suffix tree for TMaze	92
5.5	Examination of TMaze with stochastic corridor	93
5.6	Q-learning using event selectors on T-Maze	95
5.7	The Pocman experiment	96
6.1	Screenshots from the Atari2600 Learning Environment	103
6.2	Comparisons on Pong using 100 episodes	118
6.3	Comparisons on Atlantis using 100 episodes	119
6.4	Comparisons on Space Invaders using 100 episodes	119

List of Tables

3.1	Properties of domains	56
4.1	Constants used in LST experiments	72
5.1	Constants used for each experiment for hQL and FAhQL.	97
5.2	Computational comparison on Pocman	99
6.1	Experimental Parameters for all algorithms	113
6.2	Timing data on Atlantis	121

Introduction

“ We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. ”

1.1 A very brief history of AI¹

The field of Artificial Intelligence (AI) in its modern sense was founded at the *Dartmouth summer research project on artificial intelligence* spearheaded by John McCarthy, Marvin Minsky, Allen Newell and Herbert Simon in 1956. Their program was ambitious and optimistic, and although it never achieved their ultimate goals it did recognise many of the problems facing the field today; modern AI solutions can often be traced back to this ambitious start. Perhaps the biggest contribution of this project was the view of AI problems as search problems in a very large solution space with heuristics to prune down this search space. This could be viewed as the start of the field of planning. Other contributions during the initial period (1956-1974) included work on natural language, theorem proving, the language LISP, simple neural networks and rule-based systems.

The unfortunate side-effect of the optimism of the Dartmouth project was that when the field failed to deliver on its goal of “intelligent machines capable of performing all human tasks within 20 years”, it lost credibility and funding. The first “AI winter” began in 1974 and ended six years later with the advent of “expert systems”, rule-based programs designed to perform particular industrial tasks well (such as XCON, which translated customer requirements to orders automatically for the Digital Equipment Corporation). The immediate applicability of these systems to industry and their success at automating tasks brought funding back to the field albeit with a different focus. Expert systems were in no way representative of the grand dreams of the Dartmouth project.

¹For a more detailed history, see Russell and Norvig (2009) and Sutton and Barto (1998) which we draw on.

The expert systems failed to live up to their promise and the second AI winter began in 1987 starting with funding cuts from DARPA (Defense Advanced Research Projects Agency, U.S. Department of Defense) and DARPA-related agencies. After this point, there was arguably never a true revival of funding to the field. The introduction of probabilistic methods and decision theory into AI in the early 90s, by Judea Pearl and others changed the field and started various subfields of AI which became very popular in their own right, for example *machine learning*. Part of this could be attributed to the reluctance to label things as AI, due to the stigma now associated with the name. Some of the new successes of AI could also be attributed to Moore's law. IBM's Deep Blue beat reigning chess champion Gary Kasparov in 1997, through the use of simple and now standard methods (Campbell et al., 2002) such as minimax search, pruning techniques and a complex tailored evaluation function along with a very fast supercomputer.

More impressive than Deep Blue in terms of learning capability, was the success of Tesauro's backgammon agent called TD-gammon (Tesauro, 1995) in the early 90s. TD-gammon used a technique called temporal difference (TD) learning from a relatively new field known as *reinforcement learning* to train a neural network to learn how to play backgammon at a world class level, entirely via self-play.

As Sutton and Barto (1998) describe it, reinforcement learning (RL) was the combination of various "threads" that came together in the 1980s. The term itself comes from the psychology community where it originated with Thondike's "Law of effect"; actions that produced a rewarding effect in a particular situation were more likely to be taken in similar situations, and actions in situations producing negative consequences were more likely to be avoided. The second major thread was the literature from optimal control and dynamic programming. These algorithms did not learn, but solved stochastic control problems where the model was known. The learning part of dynamic programming was called adaptive control and spawned the field of cybernetics. The last ingredient was temporal difference learning which came from the animal learning literature, and had already been used to develop an impressive checkers-playing AI (Samuel, 1959) that was arguably the world's first self-learning agent.

This thesis is situated in the field of generic reinforcement learning. Reinforcement learning has the distinction amongst AI subfields of still aiming at the original goal of the field; namely to build an intelligent agent that can learn to perform well simply by interacting with the environment. RL has been studied in formal frameworks making it easy to make rigorous statements. Arguably, humans can also be modelled as reinforcement learning agents making it a framework that already has one generally intelligent agent implemented. In traditional reinforcement learning, some structure is imposed on the environment, typically the environment is assumed to be a (partially observably) Markov Decision Process. *Generic* reinforcement learning agents make no strong assumptions about the structure of their environment.

1.2 Intelligent Agents

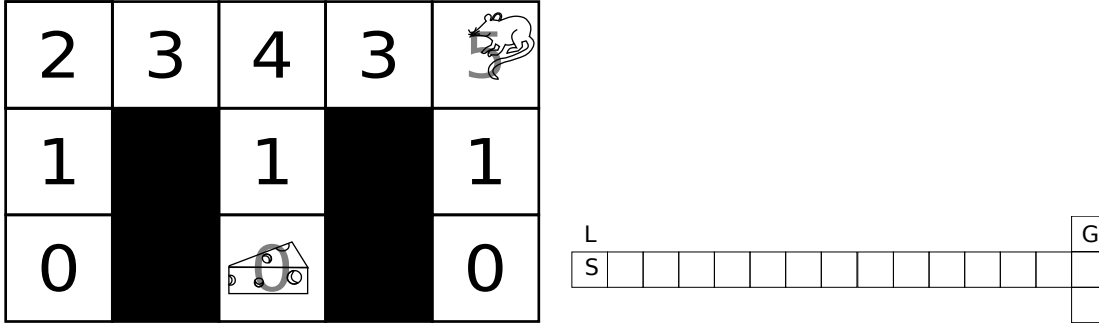
“ Should we ask what intelligence "really is"? ”
Marvin Minsky, Steps toward artificial intelligence, October, 1960.

In 1950, six years before the Dartmouth project, Alan Turing published the famous paper “Computing Machinery and Intelligence” which introduced “the imitation game” as a way to identify an *thinking machine*. In the modern interpretation of this test, a machine is said to be able to intelligent if a human judge could not distinguish it from a human being. This notion of intelligence, while crude, captures the idea that an intelligent system does not necessarily need to “think” like a human in order to “act” like a human or indeed intelligently. While the Turing test was unfortunately anthropocentric, humans are intelligent but certainly not the most intelligent agents conceivable, the notion that an agent only needs to act intelligently in order to be considered intelligent bypasses philosophical problems of consciousness and sentience.

The overarching goal of the field of Artificial Intelligence has always been the creation of an intelligent agent. The definition of “intelligence” has been hotly debated, with most definitions conflating intelligence with one (or a subset) of its properties. Legg and Hutter (2007) provide a collection of such definitions. Hutter (2005) provides a succinct definition that seems to encapsulate the properties of an intelligent agent within a single statement, “An intelligent agent is one that can achieve goals in a wide range of environments”. Of course one is now left to define “achieve goals”, “wide range”, “environments”, but Hutter (2005) shows that this can be done in a sensible way. Indeed, Hutter (2005) also provides an incomputable agent that is the most intelligent agent by this definition, called AIXI. Although there are several open problems associated with the work, this mathematical theory of artificial general intelligence can serve as a gold standard in the field.

AIXI operates within the agent-environment framework, as a generic reinforcement learning (GRL) agent. GRL agents perform actions and receive observations and rewards from the environment. The aim of an RL agent is to learn by trial-and-error to maximise some function (total, average, discounted, finite horizon, etc) of the reward. In the traditional reinforcement learning setting, the environment is structured in some way, usually as a Markov Decision Process (MDP). GRL makes no strong assumptions.

Approximations to AIXI are currently among the best performing agents on a set of general reinforcement learning toy problems. The most famous of these is the Monte-Carlo AIXI Context Tree Weighting (MC-AIXI-CTW) algorithm by Veness et al. (2011) which has shown consistently good performance. MC-AIXI-CTW is a top-down approximation of AIXI, in the sense that it approximates each component of the incomputable agent by an efficiently computable algorithm. The environment class of all semi-computable semi-measures is replaced by all context trees up to a certain depth, and the full expectimax search is replaced by the heuristic algorithm UCT. Also loosely in this category of AIXI approximations is the framework proposed by Hutter (2009b) called feature reinforcement learning (FRL). FRL aims



(a) The cheese maze demonstrates the problem of perceptual aliasing. The agent is a mouse that receives the observation denoted by the number in each grid. Several environment states have the same observation, so the agent must use past observations to determine its current position and make its way to the cheese. (b) The TMaze is a simple domain that demonstrates the need for the agent to have some form of long-term memory. The agent must use the observation received at the start of the maze in order to make a decision at the end of the long corridor. Here the agent has received the observation L or left, and must turn left at the end of the corridor.

Figure 1.1: Motivating examples

to reduce general reinforcement learning environments to Markov decision processes which allow for their optimisation via traditional RL methods. This agents defined within this thesis are defined within the FRL framework which is made formal in Section 2.4.

1.3 Challenges in generic RL

While the goal of generic RL is to act well in the most general classes of environments, the lack of structure makes this very difficult. Lattimore (2013) points out that it remains an open problem to show that even the incomputable, most intelligent agent AIXI can automatically exploit the structure of the environment to learn on par with more specialised algorithms. For example, in the sequence prediction case the currently best known upper bound on the number of prediction errors made by AIXI is exponentially worse than Solomonoff's original algorithm (Hutter (2005) Problem 6.2).

Even with the restriction to partially observable MDPs, generic RL agents face several issues. Computational issues occur even for the planning problem; solving known POMDPs is PSPACE-complete (Papadimitriou and Tsitsiklis, 1987) and learning unknown POMDPs is thus at least equally hard. The exploration-exploitation problem in unknown POMDPs is made worse by the fact that the agent does not know whether it has correctly estimated the number of states.

History-based methods are a way of avoiding the need to solve POMDP planning problem by directly learning a discrete MDP consistent with the interaction history of the agent and then solving this MDP with traditional techniques. Feature reinforcement learning is one such history-based method. There has been some empirical work on feature reinforcement learning

agents by Nguyen et al. (2011, 2012) and a thorough analysis of the original cost function in his PhD dissertation (Nguyen, 2013). These methods face a slightly different set of problems. At any given point, the history-based agent has a current choice for a discrete MDP that best represents its interactions with the environment so far. Any directed exploration that the agent performs is based on this current best guess, which can be very flawed. This is known as the exacerbated exploration-exploitation problem and is similar to the problems faced by an agent solving a POMDP with an unknown number of hidden states.

Another problem fundamental to POMDPs is that of perceptual aliasing which describes the situation where multiple environment states are mapped to a single observation, and distinguishing between these states is important to achieving good rewards. A good example of this is the Cheese Maze domain shown in Figure 1.1a. The agent is a mouse that sees observations which are the numbers on the grid. The agent must distinguish between the perceptually aliased world states that give observation 1 using its history. For instance, if the agent saw a 2 or a 5 three steps ago then it must be in the center state. This distinction is important to represent the optimal policy; it must go up in the left and right states, but down towards the cheese in the center state.

Existing practical generic RL techniques are based primarily on a class of finite state machines known as suffix trees or context trees (Rissanen, 1983). An efficient algorithm known as context tree weighting (CTW) for learning mixtures over all *predictive* suffix trees up to a certain depth came from the data compression community (Willems et al., 1995). Context trees were used in the top-down AIXI approximation MC-AIXI-CTW as a predictor, and in other history-based methods (McCallum, 1995, 1996; Farias et al., 2007). While suffix trees are useful due to the attractive computational properties of CTW, they cannot represent some simple classes of environments. A motivating example for Chapter 4 is the TMaze Figure 1.1b which demonstrates the inability of suffix trees to effectively represent environments with long-term dependencies.

Another large problem that feature RL and other history-based methods face is in dealing with large domains. The largest environment examined by the dissertation Nguyen (2013) was that of Partially Observable PACMAN (POCMAN) which required several ad hoc modifications to the defined agent (CTMRL) to work well. The above problems listed for partially observable environments are magnified in larger environments. In particular, the exploration-exploitation problem makes it hard to determine the reason a generic RL agent fails. It seems unreasonable to expect these agents to have good exploration strategies when they initially have very imperfect representations, and also unreasonable to expect them to learn good representations without appropriate samples which need a good exploration strategy. Thus, in environments with large state spaces it is useful to consider a teaching framework where an expert provides value advice to an agent, which reduces the exploration problem. This is the motivation behind Chapter 6.

The primary motivation behind this thesis is to build feature RL agents that work in practice for both larger environments and larger classes of environments.

1.4 Contributions

The primary contributions of this thesis are as follows.

1. Chapter 4 examines the problem of long-term dependencies within general environments. The class of looping suffix trees was introduced by Holmes et al. (2006) to represent the class of all deterministic POMDPs. We show that they can also fully represent a class of stochastic POMDPs known as hPOMDPs. The algorithm provided by Holmes et al. (2006) does not extend to noisy environments. By using looping suffix trees within the FRL framework, we show experimentally that environments with long-term dependencies and some stochasticity can be represented by looping suffix trees. The results on the domain T-Maze are optimal even for dependencies up to length 150.
2. Chapter 5 defines a value-based cost criterion to replace the existing model-based one. This new criterion captures the intuition that we wish to find the state representation that allows us to predict the optimal value function well. The cost is off-policy and can be evaluated model-free. It can also generalise to function approximation which makes the approach more scalable. To this end, a new class of features called *event selectors* is defined, that represent particular events in the agent's history. By using linear function approximation over these event selectors, we perform experiments on the domain Pocman and show improved performance over MC-AIXI-CTW and CTMRL.
3. Chapter 6 discusses a different setting, that of reinforcement learning with advice. Here, the agent takes advice from an oracle that gives it approximations to the value of any state-action pair. The oracle is assumed to be available only in some training phase, and the environment is so large that the value function cannot be exactly represented requiring the use of function approximation techniques. The proposed algorithm is based on the imitation learning algorithm DAgger which comes with theoretical guarantees. The algorithm extracts an explicit and complete policy from stochastic planners such as UCT, and also tests the ability of a function approximation class to represent the optimal value function in a given environment.

As a secondary contribution, Chapter 3 is a survey of various domains (primarily games) used to evaluate reinforcement learning techniques over the last few years. It culminates in a table of properties that was used to make decisions about which environments to further pursue.

A final note on the content of this thesis. This work is primarily one of empirical study, with some essential theory in each chapter. For instance, it is desirable that our proposed cost function is consistent and we show that it is with respect to a certain class of finite state machines in Chapter 5. However, in practice our algorithm may not satisfy the conditions for this consistency result to hold; indeed the consistency of feature reinforcement learning using a non-stationary explorative policy is not known. Nevertheless, the algorithms described in this thesis work well in practice unless otherwise indicated. Thus, this work may not be very interesting to those looking for a mathematical theory of general intelligence; we leave that to Hutter (2005); Lattimore (2013) and others. However, to the AI-practitioner looking to apply general reinforcement learning methods in practice, we provide extensive experiments and practical pointers for the working of each of the algorithms.

1.5 Guide to the thesis

Chapter 2 introduces background material relevant to the rest of the thesis and can be read independently of the other chapters. Chapter 3 can be read independently of any other chapter. Chapter 4 requires familiarity with the background material on the FRL framework in Section 2.4. While looping suffix trees are presented, reading Holmes et al. (2006) is highly recommended to the reader. Chapter 5 requires familiarity with the background material, particularly the sections on temporal difference learning (Section 2.2.2.2) and the FRL framework (Section 2.4). Chapter 6 relies on the background material on function approximation (Section 2.2.3). Some background information on supervised learning techniques is provided in this chapter, but Bishop (2006) is recommended for further reading. For much more information on the imitation learning algorithm DAgger, the excellent PhD thesis by Ross (2013) is recommended reading.

Background

In this chapter, we will examine some background material necessary for understanding the rest of the thesis. We do not cover *all* the necessary background for *every* chapter here, but we do cover the common elements needed for their understanding. The following chapters will contain additional background material where necessary.

2.1 Agent-Environment Framework

In this thesis we deal exclusively with agents that reside in the rational agent framework as described by Russell and Norvig (2009).

An agent acts in an Environment Env by choosing from actions $a \in \mathcal{A}$. It receives observations $o \in \mathcal{O}$ and real-valued rewards $r \in \mathcal{R}$ where we assume $\mathcal{A}$, $\mathcal{O}$ and $\mathcal{R}$ are all finite. This observation-reward-action sequence happens in cycles indexed by $t = 1, 2, 3, \dots$. We use $x_{1:n}$ throughout to represent the sequence $x_1 \dots x_n$. The history of an agent at time t is h_t which contains the sequence of observation-reward-action tuples up to time t i.e. $h_t = o_1 r_1 a_1 \dots o_{t-1} r_{t-1} a_{t-1} o_t r_t$. The space of histories is thus $\mathcal{H} := (\mathcal{O} \times \mathcal{R} \times \mathcal{A})^* \times \mathcal{O} \times \mathcal{R}$. The agent is then formally a (stochastic) function $Agent : \mathcal{H} \rightsquigarrow \mathcal{A}$ where $Agent(h_t) := a_t$ and $\rightsquigarrow$ is used to indicate stochasticity. Similarly, the environment can be viewed as a (stochastic) function of the history, $Env : \mathcal{H} \times \mathcal{A} \rightsquigarrow \mathcal{O} \times \mathcal{R}$, where $Env(h_{t-1}, a_{t-1}) := o_t r_t$.

2.2 Reinforcement Learning

“Reinforcement learning (RL) is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment”(Kaelbling et al., 1996). This broad problem we refer to as the **general reinforcement learning problem**. In this definition, the agent knows nothing about the environment apart from the observations and rewards that it receives. We would also like the agent’s learned behaviour to be *good* in some measurable way. We can define a utility function on the rewards that a *good* agent should maximise. Normally

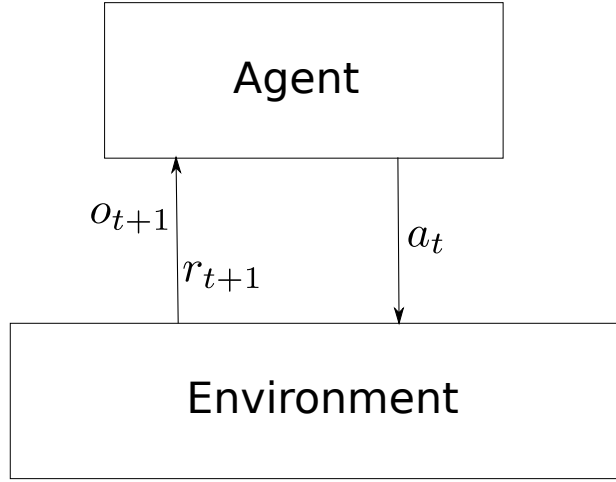


Figure 2.1: The agent-environment framework

this is the total or average reward over some finite horizon, or an infinite discounted reward sum.

In practice (and even in theory), it is hard to work with the class of all possible environments. In order to obtain guarantees, and perform well in practice, a large portion of the reinforcement learning literature studies particular classes of environments. The simplest of these is bandits (or the multi-armed bandit problem).

2.2.1 Multi-armed bandit problem

An N -armed bandit requires the agent to choose between a row of N slot-machines which each provide rewards from a specific, but unknown, distribution. The agent has to maximise the sum of money it gets from the machines and this involves balancing between pulling the arms of many potentially sub-optimal slot machines and the arm of the best one it knows so far. This is known as the *exploration-exploitation* problem and is fundamental to reinforcement learning. We will revisit this problem in Section 2.2.2.2.

In terms of the general reinforcement learning framework, an N -armed bandit problem has a single observation and N actions. If the reward set is binary $\{0, 1\}$ then it is called a *Bernoulli bandit*. Bernoulli bandits have $Pr(r = 1 \mid a = a_k) = \theta_k \in [0, 1]$ and $Pr(r = 0 \mid a = a_k) = 1 - \theta_k$. When $\theta_k \in \{0, 1\}$ for all k , it is a deterministic Bernoulli bandit. While bandits form a relatively simple class of environments, they succinctly express the exploration-exploitation problem, and solutions to this problem in the class of bandits can often be extended to other environment classes.

2.2.2 Markov Decision Process (MDP)

A larger class of environments that form the basis of most reinforcement learning approaches is MDPs. If $Pr(o_t r_t \mid h_t, a_t) = Pr(o_t r_t \mid o_{t-1} a_t)$, the observations and rewards satisfy the

Markov property and the environment is said to be a discrete MDP (Bellman, 1954; Puterman, 1994). In this case, the observations form the state space of the MDP. Formally an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, R \rangle$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$ is the (possibly stochastic) reward function which gives the (real-valued) reward gained by the agent after taking action a in state s . $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the *state-transition function*; we write $T(s' | s, a)$ for the probability of seeing s' being in state s and taking action a . The agent needs to find a *policy* that allows it to perform well by some optimality criterion (see below). A deterministic policy is a mapping from states to actions, $\pi : \mathcal{S} \rightarrow \mathcal{A}$. If the policy is time-dependent (i.e. it is defined $\pi : \mathcal{S} \times \mathbb{N} \rightarrow \mathcal{A}$) then it is said to be *non-stationary*, otherwise it is *stationary*. A policy may also be *stochastic*, in which case $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ and $\pi(a | s)$ is the probability that the agent takes action a given it is currently in state s .

Optimality criterion

When measuring the performance of an agent in an MDP, we need an optimality criterion. The total or average reward that an agent receives are two often used criteria. Below we define the *return* for each criterion and note that the agent aims to maximise the *expected return* at every time step.

If the agent uses a finite horizon T the total reward over the horizon remains bounded. The return for a finite horizon MDP at time t is then given as $R_t = \sum_{k=0}^T r_{t+k+1}$ or $\frac{1}{T} \sum_{k=0}^T r_{t+k+1}$.

If the horizon is infinite ($T = \infty$) then a geometric discount rate γ is used so that the sum does not diverge. The return is then defined as $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. The discounting has the effect that rewards far into the future are valued less. In the infinite horizon, average reward case, the return is defined as $R_t = \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{k=0}^h r_{t+k+1}$.

Unless otherwise specified, for the rest of this thesis we will focus on infinite horizon, geometrically discounted MDPs.

The *value* of a state according to a stationary policy is given by $V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]$. The optimal policy π^* is the one which provides the maximum expected discounted future reward. For an infinite horizon, discounted reward MDP, there always exists a stationary deterministic optimal policy. We call the value function of the optimal policy the optimal value function $V^{\pi^*} = V^*$ where $V^*(s) = \max_\pi V^\pi(s)$. The value of a state-action pair according to a stationary policy is given by $Q^\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a]$. For the optimal policy, $V^*(s) = \max_a Q^*(s, a)$, thus the greedy policy with respect to Q^* is optimal.

Episodic vs non-episodic environments

In an episodic environment, an agent returns to some start state once it reaches some goal state. The agent only takes into account rewards within each episode, so any expectation of future rewards should be truncated by the end of the episode i.e. the return is now written as $R_t = \sum_{k=0}^{t_{end}} \gamma^k r_{t+k+1}$ where t_{end} is a random variable that marks the end of the current episode.

2.2.2.1 Planning

Finding (exactly or approximately) the optimal policy for a known MDP is known as *planning*. If an MDP is known (i.e. T and R are known) then in theory we can solve the Bellman Equations Bellman (1957) to calculate the state/action value functions for some policy π .

$$V^\pi(s) = \sum_a \pi(a | s) Q^\pi(s, a)$$

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s'} T(s' | s, a) V^\pi(s')$$

In particular when we wish to find V^* and Q^* , we replace $\sum_a \pi(a | s)$, V^π and Q^π with $\max_a$, V^* and Q^* respectively. Then $\pi^*(s) = \arg \max_a Q^*(s, a)$ is the optimal policy.

Occasionally, it will be useful to talk about the Bellman operator T^π and the Bellman optimality operator T^* which are convenient ways of expressing the Bellman equations. Let $N = |\mathcal{S}|$, then for a deterministic policy π the linear Bellman operator $T^\pi : \mathbb{R}^N \rightarrow \mathbb{R}^N$ is defined on vectors $v \in \mathbb{R}^N$ as

$$(T^\pi v)(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' | s, \pi(s)) v(s')$$

Similarly, the non-linear Bellman optimality operator is defined $T^* : \mathbb{R}^N \rightarrow \mathbb{R}^N$ on vectors $v \in \mathbb{R}^N$ as

$$(T^* v)(s) = \max_a \left\{ R(s, a) + \gamma \sum_{s'} T(s' | s, a) v(s') \right\}$$

V^π is the fixed point of the Bellman operator T^π i.e. $T^\pi V^\pi = V^\pi$. Indeed, one can show that it is a unique fixed point. The optimal value function V^* is the unique fixed point of T^* . Both T^π and T^* are monotonic operators and contraction mappings under the max-norm with contraction factor γ .

Value iteration (VI) Value iteration uses the Bellman equations as stated above in an iterative way, in order to find the optimal policy. For each time step $k + 1$, value iteration calculates the action-values for each state-action pair from the values estimated at the previous stage k . Then it calculates $V^k(s)$ via a simple max (or *sum*) over actions. This approach is also called *successive approximations*.

$$Q^{k+1}(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s' | s, a) V^k(s')$$

$$V^k(s) = \max_a Q^k(s, a)$$

We start with $V^0(s) = \max_a R(s, a)$. Value iteration is guaranteed to converge asymptotically ($\lim_{k \rightarrow \infty} \|V_k - V^*\|_\infty = 0$), with each iteration taking $O(|\mathcal{S}|^2|\mathcal{A}|)$ steps. In practice, we set some threshold θ such that we stop when $\|V^k - V^{k-1}\|_\infty < \theta$. Puterman (1994) showed that if $\theta = \frac{\varepsilon(1-\gamma)}{2\gamma}$ then $\|V^k - V^*\|_\infty < \varepsilon$ for some ε .

An elegant way of looking at value iteration is simply as calculating for some vector $v \in \mathbb{R}^{|\mathcal{S}|}$ the limit of the Bellman (optimality) operator, $\lim_{k \rightarrow \infty} (T^*)^k v = V^*$.

Policy iteration (PI) Policy iteration also uses a dynamic programming approach to finding an optimal policy, but instead of iterating the value function, we now iterate the policy. At stage k we have some policy π_k , for which we can evaluate V^{π_k} . We then calculate π_{k+1} as a greedy policy with respect to V^{π_k} .

$$\pi_{k+1}(s) = \arg \max_a \left\{ R(s, a) + \gamma \sum_{s'} T(s' | s, a) V^{\pi_k}(s') \right\}$$

The value of the policy π_k may be determined via value iteration (for the fixed policy) or by solving the linear program defined by the Bellman equations. Policy iteration is guaranteed to converge in finite time, i.e. $\pi_k = \pi_{k-1}$ for some k , and this policy is optimal.

Value iteration, policy iteration and their variants can be performed either *synchronously* or *asynchronously*. In the descriptions of the algorithms above, we saw the synchronous case, where the updates at each iteration were over all state(-action) pairs. However, one can also perform updates for fewer states, or even one state at each iteration instead. Since the Bellman operator is a contraction mapping, the value function is guaranteed to improve with each iteration, and as long as every state is seen infinitely often we are guaranteed convergence (Sutton and Barto, 1998).

Comparison of VI and PI The conventional wisdom is that the policy can converge (to the optimal) long before the values converge, thus value iteration may run for unnecessary steps refining the value function, even though it is “good enough” for the purposes of extracting the optimal policy. Thus, using policy iteration with some fixed number of value determination steps for each policy iteration, can result in a faster performance. Unfortunately, it is hard to know in advance how many steps of value determination are necessary, just as it is hard to know what accuracy threshold to fix for value iteration to give the optimal policy. In practice, the (relative) performance of both methods depends on the size of the domain and the structure of the value function.

Monte Carlo tree search (MCTS) methods Monte-Carlo tree search methods constitute a family of algorithms that form expectimax search trees over the search space. The MCTS agent samples trajectories from the environment that terminate after some horizon or to the

end of the episode. Each node in the tree is the average of the playouts after taking the action sequence that led to the node. MCTS methods use heuristics to select the nodes most likely to lead to higher expected reward. At any point in time, the action with the highest value at the root node is the agent's current best guess of the optimal action. In this sense MCTS methods are *anytime algorithms*. MCTS algorithms operate in the following stages.

1. Selection : Traverse the tree starting from the root downward, selecting child nodes until an unexpanded node is reached.
2. Expansion : If the unexpanded node does not end the episode, expand the node to all children. Select one of them.
3. Playout : Playout a (often random) policy from the selected node.
4. Value update : Backpropagate the value of the sampled trajectory that was played out, up the tree following the path that was traversed from the root.

Upper Confidence Bounds for Trees (UCT) by Kocsis and Szepesvári (2006) is a Monte-Carlo Tree Search (MCTS) algorithm that uses UCB from the bandit setting for exploration in the generative model setting. We will use UCT in Chapter Chapter 6 where we have access to an emulator that given a state and action can execute the action in that state and provide the received reward. The pseudocode for UCT is provided in Algorithm 1.

2.2.2.2 Learning

In the case of an unknown MDP, we cannot use methods based only on the Bellman equations since we do not know the transition and reward probabilities. Learning algorithms therefore come in two flavours, model-based and model-free. Learning can also have two settings, *prediction* and *control*. Prediction refers to the case where the agent does not have to make any decisions, and is passively learning the value function of the trajectory being played out. Control is when the agent has to actively choose what to do next. We are interested in control methods for the purposes of this thesis.

Model-based methods Model-based methods estimate the model of an MDP by statistical methods based on repeated experiences. For example, one simple method is to calculate estimates of the transition probabilities and expected reward matrices based on frequency estimates from the history of the agent so far as follows.

$$\begin{aligned}
 n(s, a, s') &= \#(sa \rightarrow s') \\
 n(s, a) &= \#(sa) \\
 R(s, a, s') &= \frac{\sum_{t=1}^n r_{t+1} \mathbb{I}_{(s,a,s')}(s_t, a_t, s_{t+1})}{n(s, a, s')}
 \end{aligned}$$

Algorithm 1: UCT (Kocsis and Szepesvári, 2006), pseudocode adapted from Bellemare et al. (2013)

Input: search horizon m , simulations per step k , Environment Env with reset to state ability.

Input: search tree Ψ , current state s .

```

search( $s$ )
if  $\Psi$  is  $\emptyset$  or  $root(\Psi) \neq s$  then
     $\Psi \leftarrow$  empty search tree.
     $\Psi.root \leftarrow s$ .
end
repeat
    sample( $\Psi, m$ )
until  $\Psi.root.visits = k$ ;
 $a \leftarrow bestAction(\Psi)$ .
prune( $\Psi, a$ ).
return( $a$ )

```

Function sample(Ψ, m)

```

 $n \leftarrow \Psi.root$ .
while  $n$  is not a leaf and  $m > depth(n)$  do
    if any action  $a \in \mathcal{A}$  has not yet been taken in node  $n$  then
        reward  $\leftarrow emulate(n, a)$ .
        Create child node  $c_a$  of  $n$ .
        immediate-return( $c_a$ )  $\leftarrow$  reward.
        Change the current node to  $c_a$ , i.e.  $n \leftarrow c_a$ .
    end
    else
         $a \leftarrow selectAction(n)$ .
         $n \leftarrow child(n, a)$ .
    end
end

```

Function emulate(n, a)

```

Input: Node  $n$  containing environment state, Action  $a$ 
 $Env.resetToState(n.state)$ .
Execute action  $a$  in  $Env$  and store reward in  $r$ .
if End of episode then
    Set node to leaf node.
end
return  $r$ 

```

$$T(s' | s, a) = \frac{n(s, a, s')}{n(s, a)}$$

With these estimates of the model, we can use a planning algorithm (like value iteration) to determine an approximate optimal policy. The policy can be followed for some time during which more statistics are gathered and the process is repeated.

Model-free methods Model-free methods do not estimate the model, but instead directly estimate the state-value (or action-value) function. Commonly used model-free methods include SARSA and Q-learning which are both *temporal difference (TD)* methods and are based around estimating the value function directly.

TD-learning SARSA (State Action Reward State Action) is an *on-policy* control algorithm that uses the update rule

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

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \quad (2.1)$$

is called the temporal difference and α is known as the learning rate. *On-policy* refers to the fact that SARSA learns the Q -value for the policy that is controlling the agent's behaviour i.e. if the agent is following a policy π then Q converges to Q^π . Hence the temporal difference is calculated according to the action that the agent actually took.

In an *off-policy* algorithm, the agent wishes to learn the value of some *target policy* while following a different *behaviour policy*. Q-learning is an off-policy algorithm which learns the value of the optimal policy, while following some other behaviour policy. The update rule for TD is the same as SARSA, except the temporal difference is now calculated as, $\delta_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)$.

On-policy algorithms may seem unnecessary at first. Why do we care about estimating anything other than the optimal policy? However, the motivating “cliff-walking” example from Sutton and Barto (1998) shows us that SARSA can be very useful when we want the optimal policy, but also care about performance during the learning procedure. In this example, there is a gridworld with an agent starting just before a cliff. The agent can follow the cliff to the goal, which is the shortest path, or the agent can go around the cliff. If the agent is using ϵ -exploration strategies, then going along the cliff can be very dangerous, since it has a chance of falling off and getting the negative reward of -100. A Q-learning agent with the behaviour policy set to be ϵ -greedy, is unfortunately going to eventually do this, since it is estimating the Q -values of the optimal policy which does not fall off the cliff. Thus Q-learning will have a bad online performance compared to a SARSA agent which will instead go around the cliff, since it learns the values of the ϵ -greedy policy and knows to avoid the high-risk cliff.

One might argue that the above comparison is a little unfair, since Q-learning after the learning phase will follow the optimal policy whereas SARSA will have learned only the optimal ϵ -greedy policy (which is worse!). We can make both algorithms perform similarly by decreasing

ϵ over time and then SARSA also converges to the optimal policy while also not being as unsafe during the learning phase. Another way to guarantee that SARSA converges to the optimal policy is using optimistic initialisation (see Section 2.2.2.2) instead of ϵ -exploration. Another motivating use of on-policy algorithms will be clear when we talk about *eligibility traces* below.

Perhaps the biggest benefit of on-policy methods is that they are guaranteed to converge in the function approximation setting (see Section 2.2.3 below), whereas off-policy methods like Q-learning have no such guarantees (and do diverge in practice on some problems). This is changing as new off-policy algorithms such as Greedy GQ by Maei et al. (2010) offer some guarantees even in the off-policy function approximation case.

Monte-Carlo methods For an episodic task, we can treat the state and action values as random variables and simply average our estimates of the return over a number of episodes, given some starting state and action. This is known as the Monte-Carlo (MC) method. Note that the return in this setting is the episodic return which is truncated at t_{end} as described in Section 2.2.2.

The first-visit MC method for estimating V^π is defined as follows.

1. Collect the trajectory from an episode following policy π .
2. For each state s appearing in the episode
 - (a) Let $R(s)$ = return following the first visit of state s until the end of episode
 - (b) Append $R(s)$ to a vector of returns collected from previous episodes
 - (c) current estimate of $V(s)$ = average over $R(s)$.
3. Go back to 1.

The above is a prediction algorithm, it learns V^π . We learn Q-values similarly and use policy iteration to establish an MC control method.

The first-visit MC method for estimating Q^π is defined similarly to the above method for V^π except now we look for the return following state-action pairs. Once we have Q^π we can use the greedy policy $\pi'(s) = \max_a Q^\pi(s, a)$ to improve the policy, and then collect an estimate for the new policy. Eventually, we will converge to the optimal policy.

TD-algorithms with eligibility traces Reinforcement learning agents have to deal with the problem of delayed rewards. For example, if an agent only receives a reward in the timestep that it reaches some goal, then a TD algorithm as described above will only weakly credit temporally distant states for the reward, even though they may have been crucial for the agent in achieving the reward. Eligibility traces are a proposed solution to this temporal credit assignment problem. They mark the states and actions that are eligible for being updated via the TD-learning rule so that when a TD-error occurs, the blame/credit is assigned to the correct states and actions (Sutton and Barto, 1998).

Eligibility traces can also be viewed as interpolating between TD-learning and Monte Carlo methods. Let us look at this in the prediction setting. The return given by Monte Carlo is simply $R_t = \sum_{i=t}^{t_{end}} \gamma^{i-t} r_{i+1}$ whereas the bootstrapped 1-step return given by TD is $R_t^1 = r_{t+1} + \gamma V(s_{t+1})$. Similarly, the n -step return is defined as

$$R_t^n = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

n -step returns are not used directly in reinforcement learning, since they suffer from the issue of needing to wait n -steps in order to update V . However, we can define an algorithm that performs a backup towards a weighting over all n -step returns. Let $\lambda \in [0, 1]$. Then we can define the λ -weighted average over all n -step returns as follows.

$$R_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^n$$

TD(λ) is the algorithm that updates toward this return i.e.

$$V_{t+1}(s) = V_t(s) + \alpha \left(R_t^\lambda - V_t(s) \right)$$

When $\lambda = 1$ this reduces to the Monte Carlo setting and when $\lambda = 0$ this is the 1-step TD setting.

We described the prediction setting here, and the control setting can be explained similarly. The algorithm for on-policy TD control, SARSA(λ) is provided in Algorithm 2. When eligibility traces are used in the off-policy setting, it is not clear how to (not) assign credit to exploratory actions. There have been several attempts (Peng and Williams (1996), Watkin's $Q(\lambda)$ Sutton and Barto (1998)) at extending Q-learning to $Q(\lambda)$ -learning.

Policy search Instead of attempting to approximate the value function via a model-based or model-free method, policy search methods operate in the space of all or a subset of all policies. Often this is done via parameterising the policy space and then using a gradient-descent approach to find an optimal policy. Such methods are known as *policy gradient* algorithms. An excellent introduction to policy gradient methods is found in Peters (2010).

Algorithm 2: SARSA(λ)

```

Initialise  $Q(s, a)$  arbitrarily;
for  $m \in [0, \text{episodes})$  do
  Set  $e(s, a) = 0$  for all  $s, a$ ;
  Assume that the environment starts in state  $s_0$ ;
  Action  $a_0$  is picked uniformly at random;
   $t \leftarrow 0$ ;
  while not end of episode do
     $s_{t+1}, r_t = Env(s_t, a_t)$ ;
     $a_{t+1} = \begin{cases} \arg \max_a Q(s_{t+1}, a) & \text{with probability } 1 - \epsilon \\ \text{uniform random action} & \text{otherwise} \end{cases}$ ;
     $\delta \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ ;
     $e(s_t, a_t) \leftarrow e(s_t, a_t) + \delta$ ;
     $t \leftarrow t + 1$ ;
    foreach  $s, a$  do
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ ;
       $e(s, a) \leftarrow \gamma \lambda e(s, a)$ ;
    end
  end
end

```

Exploration vs Exploitation Both model-based and model-free methods run into the problem of exploration vs. exploitation. The simple model-based algorithm described above cannot work well unless the agent gathers the samples according to some policy that explores the state space in a way that improves the agent's current model. The agent has the choice of exploiting a behaviour that it knows to be good, or exploring further to find something better. If the agent spends too much time exploiting known states then it may be stuck on a suboptimal policy, but it should also not spend too much time visiting states with low reward, and known parameters. One simple heuristic approach known as the ϵ -**greedy strategy** is where the greedy action is chosen most of the time and a random action otherwise. However this strategy fails on simple examples which need some directed exploration in order to avoid a random walk policy (for example, Figure 2.2).

A more principled approach is that of *optimism*. The agent makes an optimistic assumption about the value of each state. Therefore it will choose to greedily explore these first, but will learn the true value of a state after a few visits to it. Unknown states still retain their optimistic initialisation and will be explored next. In theory, an agent with high enough optimism should always converge to the optimal policy, given enough time (Even-Dar and Mansour, 2001). However in practice, it is hard to decide the value or level of optimism that the agent should start with, a value that is too small may not be adequate for exploring a large state space, but a value that is too large can result in a much longer convergence time.

In model-free algorithms, optimism is easily expressed by *optimistic initial values*. Optimism can be expressed in model-based algorithms by the addition of a so-called “*Garden of Eden*”

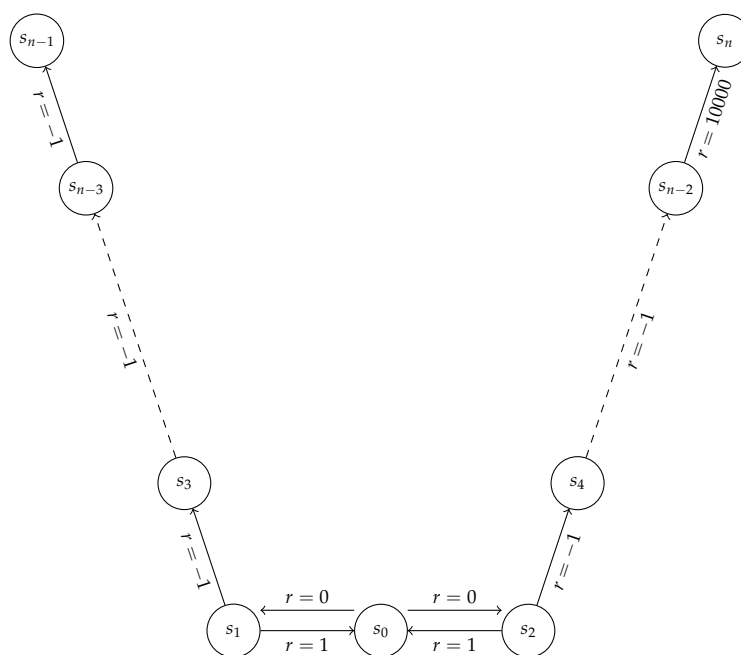


Figure 2.2: A hard MDP. The agent has no incentive to explore the long path to get to the reward of 10000, since it is paved by a road of -1s. Any strategy relying on occasionally taking a random action is likely to fail with high probability. It is for MDPs like this that we need principled exploration. Optimistic agents have the potential to do well here.

state to the state space. This approach is used in model-based learning algorithms to encourage directed exploration. An unreachable absorbing state which has maximum reward is added to the current model of the environment dynamics and the agent is told that it has been there once. Thus the agent will initially explore unknown states in an attempt to get to this garden of Eden, but as the model of the environment becomes more accurate, the estimated chance of transitioning to that state reduces greatly, and the agent converges to the true policy.

PAC-MDP algorithms There has been much work on the exploration-exploitation problem in the **PAC** (Probably Approximately Correct) setting that results in polynomial bounds. The sample complexity of an algorithm is defined as the number of time steps for which the policy is not ε -optimal at the current state i.e. time steps t for which $V^\pi(s_t) < V^*(s_t) - \varepsilon$ for some ε . An algorithm is PAC-MDP if for all ε and δ the sample complexity of the algorithm is less than some polynomial in the number of states $|\mathcal{S}|$, actions $|\mathcal{A}|$, $\frac{1}{\varepsilon}$, $\frac{1}{\delta}$ and $1 - \gamma$ with probability at least $1 - \delta$.

PAC algorithms tend to be based on the above mentioned methods of assigning optimism, with more rigorous ways of determining when to be optimistic. For example, E^3 (Kearns, 1998) and RMax (Brafman et al., 2001) rely on the concept of a *knownness state*, a state that the algorithm has visited “so many” times. While E^3 uses this explicitly to decide when to explore or exploit (using a method they call “balanced wandering”), RMax uses it implicitly to decide how to update its model which then automatically takes care of exploration and exploitation. Perhaps the most elegant method, OIM (Szita and Lőrincz, 2008) does not need such a state at all, relying on the garden of eden approach described above to completely guide its optimism. The MBIE algorithm (Model-based interval estimation) (Strehl, 2004) constructs confidence intervals on *possible* models that are consistent with experience. It then picks the most optimistic model from this set and acts according to that. In the model-free setting, the delayed Q-learning (Strehl et al., 2009) method provides PAC results using optimistic initial values.

Of note is the line of work following the Upper Confidence Reinforcement Learning algorithm (UCRL) by Auer and Ortner (2007) and its successors (UCRL2 (Jaksch et al., 2010), UCRL γ (Lattimore and Hutter, 2012)). These are model-based algorithms that provide state-of-the-art sample complexity (or regret) bounds. The general approach is to maintain the (smallest possible) set of plausible models (i.e. those statistically consistent with the history so far) and act according to the most optimistic model within this class.

2.2.3 Function approximation

When the state space of an MDP is very large (or continuous) we need to represent the value function more compactly via a suitable class of parameterized function approximators. Function approximation solves two problems associated with large state spaces, the obvious one of compact representation of the value function but also the problem of generalisation.

Generalisation is the agent's ability to predict the value of states it hasn't seen before based on value estimates of states it has seen. The presentation below follows Sutton and Barto (1998).

Assume that an approximation of the value function at time t is parameterised by some vector $\theta_t \in \mathbb{R}^k$, $V_t^\pi(s; \theta_t)$. The Mean Squared Error (MSE) for an approximation of the value function is given by

$$MSE(\theta_t) = \sum_s d(s) [V^\pi(s) - V_t^\pi(s; \theta_t)]^2$$

where $d(s)$ is a distribution over states. Normally the limiting distribution under the agent's current policy π is used $d(s) = \lim_{t \rightarrow \infty} Pr(s_t = s)$.

Assume that we are given $V^\pi(s)$ for the states that we sample, and that we want to minimise the $MSE(\theta_t)$ to find a good value-function parameterisation. Then an appropriate algorithm is (stochastic) gradient descent. Here, steps are taken in the direction of the error reduction for the sample s_t just seen.

$$\begin{aligned} \theta_{t+1} &\leftarrow \theta_t - \frac{1}{2} \alpha \nabla_{\theta_t} [V^\pi(s_t) - V_t^\pi(s_t; \theta_t)]^2 \\ &= \theta_t + \alpha [V^\pi(s_t) - V_t^\pi(s_t; \theta_t)] \nabla_{\theta_t} V_t^\pi(s_t, \theta_t) \end{aligned}$$

where $\nabla_{\theta_t} V_t(s_t, \theta_t)$ is the vector of partial derivatives of $V_t(s_t, \theta_t)$.

The above can be easily extended to when we do not know the true value function for a state s , by replacing the true value $V^\pi(s_t)$ with the current estimate $\hat{V}_t^\pi(s_t)$.

$$\theta_{t+1} \leftarrow \theta_t + \alpha [\hat{V}_t^\pi(s_t) - V_t^\pi(s_t; \theta_t)] \nabla_{\theta_t} V_t^\pi(s_t, \theta_t)$$

If $\hat{V}_t^\pi(s_t)$ is an unbiased estimator of $V^\pi(s_t)$ then the above algorithm converges to the true value. For instance, this is true when using the Monte Carlo return $\hat{V}_t^\pi(s_t) = R_t$. However, in practice we normally use bootstrapped estimates of the return such as R_t^λ or $E_\pi[R_t^1]$ which are biased.

A linear function approximation over d -dimensional features $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$ approximates the value $Q(s, a)$ by $w^\top \phi(s, a)$ where $w \in \mathbb{R}^d$ is a weight vector we learn. The SARSA update rule is then

$$w_i \leftarrow w_i + \alpha \delta \phi(s_t, a_t)$$

for all $w_i \in w$, where δ is as defined in Equation 2.1.

Other classes of function approximators are also popular. For a description of Radial Basis Functions (RBFs) and tile coding see Sutton and Barto (1998). Neural networks are very popular, and can be trained via backpropagation, amongst other methods. One of the biggest successes in the field of RL was Tesauro's TD-gammon (Tesauro, 1995) which made use of temporal difference learning, neural networks and self-play to become the best backgammon player in the world. TD-gammon made good use of a concept borrowed from biology known as *co-evolution*, where the policy evolves in response to changes in the neural representation and vice-versa. Feature reinforcement learning (Section 2.4) uses a similar idea, where there is

a dependence between the current best map ϕ and the best policy according to that map. Each of these changes in response to the other, with the policy influencing the map by altering the agent's interaction with the environment.

2.2.4 State aggregation

State aggregation reduces the size of an MDP state space by grouping together states that are considered equivalent by some similarity metric (e.g. stochastic bisimilarity by Givan et al. (2003) or ϕ -uniformity by (Hutter, 2014)). We briefly explain stochastic bisimilarity below, and ϕ -uniformity in Section 2.6.3.

Stochastic bisimilarity Let R be an equivalence relation between states. Let U_R be the set of all possible equivalent classes, $U_R = \{\{b \in \mathcal{S} \mid aRb\} \mid a \in \mathcal{S}\}$. Then, two states s and s' are said to be bisimilar ($s \sim s'$) if and only if there exists a relation R such that $\forall a \in \mathcal{A} \ r(s, a) = r(s', a)$ and for all $C \in U_R \ \sum_{c \in C} T(s, a, c) = \sum_{c \in C} T(s', a, c)$. Any equivalence relationship on the state space $\mathcal{S}$ naturally gives rise to an aggregation by aggregating states within the same equivalence class. Givan et al. (2003) show that any optimal policy in an aggregated MDP defined by bisimilarity induces an optimal policy in the original MDP.

State aggregation is a special case of function approximation as described in Section 2.2.3. Given a similarity metric, we can define features as indicator functions based on representative states. A feature for a particular state is on if and only if it is (approximately) similar to the representative state. Additionally, only one feature can be on at a time. Thus the representation is effectively tabular; a simple grouping together of states. Even though state aggregation appears to be a simple idea, proving bounds on the performance of particular metrics for aggregation is an ongoing area of research (Ortner, 2007; Hutter, 2014).

2.3 Partially observable MDPs

An interesting class of environments that is more general than MDPs but still imposes structure on the observations is the class of partially observable MDPs (POMDPs) (Smallwood and Sondik, 1973). POMDPs are analogous to MDPs in the way that Hidden Markov models are to Markov chains. In a POMDP there are observations that the agent sees directly, which are functions of Markovian states. Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, T, \Omega, \gamma, \mathcal{R} \rangle$ where $\mathcal{S}, \mathcal{A}, T, \mathcal{R}$ and γ are the same as in the MDP setting. $\mathcal{O}$ is a set of observations that are generated according to $\Omega : \mathcal{S} \rightsquigarrow \mathcal{O}$ which also define a set of *emission probabilities*. $\Omega(o \mid s')$ is the probability that the agent sees observation o if it ends up in state s' .

The way in which emission probabilities are defined above relies only on the state arrived in. It is common to use a more complex model, for instance $\Omega : \mathcal{S} \times \mathcal{A} \rightsquigarrow \mathcal{O}$ where the probability depends not only on the state the agent arrives to, but the action it took to get

there i.e. (a, s') . A less commonly used model is $\Omega(o \mid s, a)$ i.e. the emissions do not depend on the next state at all but only the current state and action taken.

In some sense all these emission models are equivalent; we can always construct POMDPs that generate the same history sequences with any emission model. For example, to convert a POMDP M whose emission probabilities depend on (a, s') to a POMDP K with emissions dependent only on s' we construct a new state space $\mathcal{S}'$ with $|\mathcal{S}| \times |\mathcal{A}|$ states. Each state-action pair (s'_M, a) in M corresponds to a state y'_a in $\mathcal{S}'$. Let $s \in M$ be associated with $y_b \in K$ and s' with y'_a . The transition probabilities of the new POMDP K for all b are $T_K(y'_a \mid y_b, a') = T_M(s' \mid s, a)$ if $a = a'$ and 0 otherwise. Then $\Omega_K(o \mid y'_a) = \Omega_M(o \mid a, s')$ are the emission probabilities.

The case of mapping an emission model Ω_M depending on (a, s') to Ω_K dependent on (s, a) is also simple. (s, a) determines s' via the transition probabilities $T(s' \mid s, a)$. For each o and (s, a) set $\Omega_K(o \mid s, a) = \sum_{s'} T(s' \mid s, a) \Omega_M(o \mid a, s')$. Note that this transformation also applies to the setting where Ω_M depends on s' alone.

A simple example of a POMDP is the famous tiger domain as described in Section 3.3.10.1.

2.3.1 Perceptual Aliasing

The POMDP representation leads to some interesting challenges. A primary issue within this framework is the so-called *perceptual aliasing* problem. *Perceptual aliasing* (Whitehead and Ballard, 1991) describes the situation when multiple unobservable environment states are represented by a single observation; furthermore distinguishing between these states is important for the performance of the agent. A simple example is the case of a navigation task, where two corridors may look exactly the same but only one leads to the goal. Hence it is important to be able to distinguish between these observations by remembering where one came from.

2.3.2 Solving POMDPs

POMDPs have been the subject of over four decades of intense research from their first description by Smallwood and Sondik (1973). Finding the optimal policies of known finite-horizon POMDPs was proven PSPACE-complete by Papadimitriou and Tsitsiklis (1987). Brazilunas (2003) provides an excellent survey on solution methods until 2003. In 2003, Pineau et al. (2003) introduced point-based value iteration (PBVI) which started a new line of approximate POMDP solvers. A survey of PBVI solvers is given by Shani et al. (2013). Here we discuss some exact techniques for solving POMDPs and a few classes of approximate techniques. The following classification of solution techniques borrows from Murphy (2000), and the reader is advised to look at the above surveys for more details.

2.3.2.1 Known POMDPs

In a known POMDP the agent knows all the quantities defined in the tuple. We will use the emission model $\Omega : \mathcal{S} \rightsquigarrow \mathcal{O}$. In order to solve this known POMDP the agent needs to estimate which state it currently is in. This is normally done using a *belief-vector* $\mathbf{b}$, where $b(s)$ is the agent's belief probability that it is currently in state s . Upon taking some action a and observing o , the agent updates its belief-vector over the next states s' using Bayes' rule.

$$\begin{aligned}
 b'(s') &= Pr(s' \mid o, a, b) \\
 &= \frac{Pr(o \mid s', a, b) Pr(s' \mid a, \mathbf{b})}{Pr(o \mid a, \mathbf{b})} \\
 &= \frac{\Omega(o \mid s') \sum_{s^* \in \mathcal{S}} T(s' \mid s^*, a) b(s^*)}{\sum_{s^* \in \mathcal{S}} \Omega(o \mid s^*) Pr(s^* \mid a, \mathbf{b})} \\
 &= \frac{\Omega(o \mid s') \sum_{s^* \in \mathcal{S}} T(s' \mid s^*, a) b(s^*)}{\sum_{s'' \in \mathcal{S}} \Omega(o \mid s'') \sum_{s^* \in \mathcal{S}} T(s'' \mid s^*, a) b(s^*)}
 \end{aligned}$$

The update rule for the belief-vector b' only relies on b , thus the belief-vectors can be viewed as forming a continuous MDP with the states being the belief-vectors, now called *belief states*. The transition probabilities for this belief-state MDP are defined by

$$Pr(\mathbf{b}' \mid a, \mathbf{b}) = \sum_{o \in \mathcal{O}} Pr(\mathbf{b}' \mid a, \mathbf{b}, o) Pr(o \mid a, \mathbf{b})$$

where

$$\begin{aligned}
 Pr(\mathbf{b}' \mid a, \mathbf{b}, o) &= \begin{cases} 1 & \text{if } \forall s' Pr(s' \mid o, a, \mathbf{b}) = b'(s') \\ 0 & \text{otherwise} \end{cases} \\
 Pr(o \mid a, \mathbf{b}) &= \sum_{s'' \in \mathcal{S}} \Omega(o \mid s'') \sum_{s^* \in \mathcal{S}} T(s'' \mid s^*, a) b(s^*)
 \end{aligned}$$

The reward function for the belief-state MDP is given by

$$R(\mathbf{b}, a) = \sum_{s \in \mathcal{S}} b(s) R(s, a)$$

Given this belief-state MDP, we would like to learn the value of each belief-state $V(\mathbf{b})$, either exactly or approximately. A key property that is utilised in all such algorithms is that the value function of the belief-state MDP is piecewise linear and convex (PWLC, Sondik (1978)). This means that the value function can be represented as the maximum over some linear segments called α -vectors i.e. $V(b) = \max\{\alpha_i \cdot b\}$ where $(\cdot)$ represents the standard dot product. Algorithms such as the Witness algorithm (Littman (1994b)) make use of this property to provide exact solutions for small POMDPs. More scalable solutions approximate V via

discretization, or b via particle filtering, or both V and b (see Murphy (2000) for a detailed presentation).

One can also solve the underlying MDP, and then use the action given by the optimal policy for the most likely state. Alternatively, one can approximate $Q(b, a)$ by $\sum_s b(s) Q^{MDP}(s, a)$ and then choose $\pi(b) = \max_a Q(b, a)$. This algorithm is known as Q -MDP (Littman et al., 1995).

Point-based based value iteration The introduction of point-based value iteration (PBVI) by Pineau et al. (2003) greatly influenced research for the next decade with several algorithms (e.g. HSVI by Smith and Simmons (2004), PERSEUS by Spaan and Vlassis (2005)) improving upon this method. PBVI solves large POMDPs quickly making it practical for more realistic problems. The key behind PBVI is to maintain a subset of the belief states and only approximate the value function at these states. This subset is constructed out of *reachable* belief states, i.e. states that will be visited during the computation of the value function.

Monte-Carlo Tree Search planning for POMDPs When a black box simulator for a large MDP exists, often the best technique to use is an MCTS planner (such as UCT). In this vein, Silver and Veness (2010) introduced a MCTS planner for POMDPs called POMCP. POMCP performs well on large POMDPs such as rocksample (15x15) which has over 7 million states, battleship (10^{18} states), and Partially Observable PACMAN (which has 10^{56} states). POMCP extends UCT to partially observable domains by constructing a search tree based on histories rather than states. Rather than sampling the next history given the current history, the algorithm maintains belief states and uses the state-based simulator by sampling states from the current beliefs. It does not perform full belief state updates, rather it approximates the belief state using an unweighted particle filter which it can update via sample observations, rewards and state transitions. This results in an efficient, albeit approximate, algorithm which performs very well in practice.

2.3.2.2 POMDPs with unknown parameters

Given that known POMDPs are so hard to solve, we can expect that also learning POMDP parameters is even harder.

Baum-Welch based methods POMDPs are closely related to Hidden Markov Models (HMMs), the analogue being that Markov processes are to MDPs what HMMs are to POMDPs. The well-known Baum-Welch (Baum et al., 1970) algorithm is used to calculate a maximum likelihood estimate of the parameters of an unknown HMM given some observation data. Baum-Welch is not guaranteed to converge to a global optimum rather to a local optimum.

The procedure extends to estimating the parameters of an unknown POMDP which can be viewed as an HMM given a fixed policy.

Some of the earliest work on unknown POMDPs (Chrisman, 1992a) used Baum-Welch to update a predictive model of the world. Koenig and Simmons (1996) used the Baum-Welch procedure in the passive setting to learn POMDP parameters given a fixed policy being followed by a robot in the real world.

Finite state controllers The agent can learn an approximate optimal policy by limiting the memory the agent has to represent the policy via a finite state machine (FSM) with N states. The transitions between these internal states depend on the observations. The policy is then simply a map from the internal states to actions. The FSM along with the policy defines a finite state controller (FSC). Gradient-based algorithms are often used to find the best FSC in some parameterised class (e.g. Internal state policy gradient (ISPG) by Aberdeen and Jonathan (2002)). FSC-based methods are most often used when the POMDP is known, but can also be used when the parameters are unknown but the number of hidden states is known.

2.3.2.3 Unknown POMDPs

In the most realistic cases, we do not know how many internal states the POMDP has. This exposes more problems that the agent must solve. History-based methods attempt to deal with this more general setting.

The exacerbated exploration problem The exacerbated exploration problem was first pointed out by Chrisman (1992b). The exploration-exploitation trade-off that occurs in unknown MDPs is further exacerbated by the perceptual aliasing problem. This means that the agent may consider an explored area of the true state space and an unexplored area to be the same, and until it has explored further cannot tell the difference between the two. Since the agent considers the area to be subjectively well-explored according to its current model of the state space, forced exploration is necessary in order for the agent to learn more about the true environment (and change its current idea of the state space).

Concealment of crucial features Chrisman (1992a) also pointed out that if crucial features of the environment are only revealed rarely, determining that they “cause” some important behaviour is hard. For example, consider a maze with a mouse trying to find some cheese. The cheese is in a mousetrap and the only way the mouse can release the trap is to go to some square and inadvertently press a hidden button. No observation is given to the mouse to indicate this. If the mouse goes to the cheese immediately after pressing the button, then it can eat it, else it is caught in the trap and suffers a high negative reward. Since the agent is unlikely to go directly to the cheese immediately after pressing the button, it does not realise the causal link between the two. However, it can still almost completely accurately predict the world as experienced according to its suboptimal policy by simply modelling the trap as most often giving a negative reward and occasionally giving a positive one.

Reactive policies for POMDPs Reactive agents only use the most recent observation to form a (memoryless) policy. A simple example of this is to use a temporal difference learning algorithm treating the most recent observation as a state. While the Markov assumption is violated, the method may still perform well if the observations are informative, and Singh et al. (1994) give bounds on such an approach. There has also been work on finding the optimal deterministic memoryless policy for a known POMDP by Littman (1994a) which was shown to be an NP-complete problem in the infinite horizon setting.

History-based methods State aliasing is not necessarily a problem. As long as there is no need for a distinction between two states in terms of expected reward, state aliasing can help give a more compact representation of the environment. Therefore, there is need for methods that can extract suitable *state representations* from the history of an agent that maintain perceptual distinctions only where necessary. There are several methods that fall into this category.

Utile-Suffix Memory (USM) by McCallum (1995) is a method of adding short-term memory to the reinforcement learning agent. It combines previous work on Nearest Sequence Memory and Utile Distinction Memory to produce an algorithm that stores the history sequence so far, and uses a statistical test (Kolmogorov-Smirnov) to determine how much of that history is important to remember in order to attempt to solve the problems associated with perceptual aliasing.

The algorithm uses suffix trees (Definition 6) in order to represent this memory. The history sequence is clustered by the nodes in the suffix tree i.e. the suffix tree maps history sequences to state sequences. The suffix tree model for USM is altered slightly to store both observations and actions at alternating levels of the tree. The algorithm keeps a fringe set of an additional layer of leaves below the existing leaf nodes. These fringe leaves are the hypothesis set of the memory, and the statistical test determines whether or not a given hypothesis will improve its reward prediction. Each additional leaf node results in a further partition of the state space (or a “longer” memory).

The U-Tree algorithm McCallum (1996) uses USM to maintain this tree, but also adds “selective perception” which can divide the percept into components and selectively ignore some of these components. Value iteration is then used to select an action at each time step given the current state representation.

A *history-list* is simply a sequence of action-observation pairs. The idea is that agents that match the same history list at two different times are at the same state. An action-observation suffix tree can be viewed as a compact representation of all possible history lists to a certain depth. Timmer and Riedmiller (2007) provide a way of performing Q-learning with history-lists with some guarantees of convergence.

CTMRL The Context Tree Maximising (CTM) for Reinforcement Learning (RL) algorithm by Nguyen et al. (2012) uses the CTM approach (Volf and Willems, 1994) to sequence prediction that analytically finds the context-tree model with the minimum description length. The sequence prediction setting is adapted for RL by predicting the state-reward sequence conditioned on the actions. For large domains, the CTMRL approach binarises the percept (observation, reward) space, and additionally adds an “unseen” context, whose action value is initialised based on the value of the first subsequent seen state.

MC-AIXI-CTW The Monte Carlo (MC) AIXI Context Tree Weighting (CTW) algorithm by Veness et al. (2011) is an approximation of the theoretically optimal universal agent AIXI (Hutter, 2005). Instead of using the universal mixture, it uses a mixture over all suffix trees with the weights being functions of their code lengths, so that simpler trees are considered more likely. It dynamically creates the contexts so that only relevant ones are used. It uses the Krichevsky-Trofimov estimator to estimate the probabilities of symbols occurring in each context of the tree and by using properties of CTW (Willems et al., 1995) it can calculate the probability of a history sequence in a computationally efficient way. The (action-conditional) CTW tree maintains a model of the world which is updated after each interaction with the environment, and a Monte-Carlo Tree Search algorithm such as UCT is used to determine which action to take next using the current mixture over suffix trees as a generative model.

2.3.3 History-POMDPs

History-POMDPs (hPOMDPs) are a class of finite POMDPs that were first described by Pendrith and McGarity (1998) to study *direct RL* in POMDPs. Direct RL methods find reactive policies as described in Section 2.3.2.3. An hPOMDP is a class of POMDPs for which the history is sufficient to disambiguate the current internal POMDP state, i.e. there exists a function $\phi : \mathcal{H} \rightarrow \mathcal{S}$ such that $\phi(h) = s \in \mathcal{S}$. Note that hPOMDPs may not require the entire history to resolve to a state. For example, an MDP can be viewed as an hPOMDP with the states also being the observations.

For clarity, we ignore rewards in the presentation below. We can consider a *percept* e to be a tuple (o, r) . Replacing observations with percepts in the lemmas below does not significantly affect any of the results.

The condition that there exists a map ϕ that can disambiguate any history to a state also applies to the empty history ϵ i.e. $\phi(\epsilon)$ should unambiguously map to the starting state of the hPOMDP. However we can relax this condition to be that there exists a length l and a map $\phi : \mathcal{H} \rightarrow \mathcal{S}$ such that for all histories h longer than l i.e. with $|h| > l$, $\phi(h) = s$. An example of an hPOMDP that satisfies this modified rule for histories greater than length n is an n -Markov model.

In the following lemmas, we show that the existence of the deterministic map ϕ imposes a structure on the transition and emission probabilities of the hPOMDP.

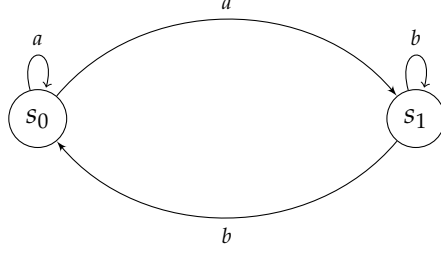


Figure 2.3: Let $T(s_0 | s_0, a) = \frac{1}{2}$ and $T(s_1 | s_1, b) = \frac{1}{4}$. Assume there are three observations $0, 1, 2$ such that $\Omega(0 | s_0) = 1$, $\Omega(1 | s_1) = \frac{1}{2}$ and $\Omega(2 | s_1) = \frac{1}{2}$. Let $\phi(\epsilon) = s_0$. This is an hPOMDP since the observations determine the next state even if the state transitions are stochastic, and we know that the initial state given by ϕ is s_0 . For example, $\phi(a0a0b1b2b0) = s_0$. It also has stochasticity in transitions as well as in emissions.

Lemma 1. Assume a history h such that $\phi(h) = s \in \mathcal{S}$. Let an agent take action a and receive observation o and assume that $\phi(hao) = s'$. Then s, a and o are sufficient to uniquely determine s' .

Proof. Let $\text{trans}(sao)$ be the set of possible states from s taking action a and observing o . Assume that $\phi(hao) = s'$ but the transition $s \xrightarrow{a} o$ remains ambiguous i.e. $|\text{trans}(sao)| > 1$. WLOG assume $\text{trans}(sao) = \{s', s''\}$. Then hao is a possible history that leads to the POMDP in state s'' . So $\phi(hao) = s''$ or else ϕ is an incorrect map. This contradicts $\phi(hao) = s'$ unless $s' = s''$ and $|\text{trans}(sao)| = 1$. $\square$

The above lemma has the implication that $Pr(s^* | s, a, o) = 1$ for $s^* = s'$ and 0 for all other states. This imposes the following constraint on the transition and emission probabilities of the hPOMDP.

Lemma 2. Let $s \in \mathcal{S}$ and suppose the agent chooses action a and then receives observation o . $\exists s' Pr(s' | s, a, o) = 1$ if and only if $\exists s' \forall s'' \neq s' T(s'' | s, a) = 0$ or $\Omega(o | s'') = 0$.

Proof. Suppose $\exists s' \forall s'' \neq s' T(s'' | s, a) = 0$ or $\Omega(o | s'') = 0$. Then $Pr(s'' | s, a, o) = \frac{\Omega(o | s'')T(s'' | s, a)}{\sum_{s'' \in \mathcal{S}} \Omega(o | s'')T(s'' | s, a)} = 0$ for all $s'' \neq s'$ and so $\text{trans}(sao) = \{s'\}$. Note that the denominator is non-zero since we know o is possible from s' and s' is possible from (s, a) . Conversely, assume $\text{trans}(sao) = \{s'\}$. Suppose $\exists s'' \neq s'$ such that $Pr(s'' | s, a, o) \propto \Omega(o | s'')T(s'' | s, a) > 0$. Then $s'' \in \text{trans}(sao)$ which is a contradiction. $\square$

Effectively, if the hPOMDP is in state s then knowledge of both a and the resulting o , determines the state s' . We call this the history condition. Deterministic POMDPs satisfy this condition by default. Note that the hPOMDP may still have stochasticity in both the transition and emission matrices, so long as the stochasticity does not violate the history condition. An example of a stochastic hPOMDP is given by Figure 2.3.

An hPOMDP should have completely determined belief-states, i.e. the belief-state MDP is discrete given the initial state is known. This is easily shown using the above lemma. Let h_i be some initial history such that $\phi(h) = s$. Then for this history we know that the belief vector b has $b(s) = 1$ and $b(s') = 0$ for all $s' \neq s$. Let us take some action a and observe o . Then the new belief vector b' can be found by computing $b'(s')$ for all $s' \in \mathcal{S}$.

$$\begin{aligned}
 b'(s') &= \frac{\Omega(o | s') \sum_{s^* \in \mathcal{S}} T(s' | s^*, a) b(s^*)}{\sum_{s'' \in \mathcal{S}} \Omega(o | s'') \sum_{s^* \in \mathcal{S}} T(s'' | s^*, a) b(s^*)} \\
 &= \frac{\Omega(o | s') T(s' | s, a)}{\sum_{s'' \in \mathcal{S}} \Omega(o | s'') T(s'' | s, a)} \\
 &= Pr(s' | s, a, o)
 \end{aligned} \tag{2.2}$$

Thus, from the previous lemma we know that $b'(s') = 1$ for exactly one $s' \in \mathcal{S}$ and zero for all others. Hence, if we know the initial (belief) state of the hPOMDP, then the associated belief-state MDP is discrete. Additionally, we can run the steps of the above proof in reverse and obtain that any POMDP which has discrete belief states given some initial state, satisfies the history condition. Abusing notation, we can write the discrete state b instead of the continuous vector $\mathbf{b}$.

2.4 Feature reinforcement learning

This thesis sits within the feature reinforcement learning (FRL) framework. FRL was proposed by Hutter (2009b) as a way of solving general reinforcement learning problems by extracting features from the history that are useful in predicting future consequences of actions. It follows in the tradition of the history-based methods described above, where the agent tries to model the agent's history in order to predict the rewards well, which then allows it to find a policy that maximises expected utility. Previous history-based methods (Section 2.3.2.3) started with an initial guess at a map from the agent's history to a Markov state space and then refined or expanded it according to some local criterion. FRL aims at being more general, by having a criterion that evaluates the map as a whole. This has the advantage of being more principled and flexible, but also comes with some disadvantages, primarily that it is hard to quickly update the cost of a map when making a small local change.

Formally, the FRL framework gives us a method to find a map $\phi : \mathcal{H} \rightarrow \mathcal{S}$ such that the state $s_t = \phi(h_t)$ at time step t is a useful summary of the history. It defines a global cost function that is inspired by the minimum description length principle (Rissanen, 1978). The cost is the sum of the code lengths of state and reward sequences given actions. This cost is combined with a global stochastic search technique (such as simulated annealing (Liu, 2008)) to find the optimal map. In order to define the original cost function used by Hutter (2009b) we first have to define the code length of an i.i.d. sequence $x_{1:n}$ for $x_i \in \mathcal{X}$ where $\mathcal{X} = \{1, \dots, m\}$. Let n_i be the number of occurrences of symbol x_i in the sequence $x_{1:n}$.

Definition 3. The entropy of frequency estimate $\hat{\theta} = (\hat{\theta}_1, \dots, \hat{\theta}_m)$ where $\hat{\theta}_i = \frac{n_i}{n}$ of symbols x_i in $x_{1:n}$ is defined to be

$$H(\hat{\theta}) = H\left(\frac{(n_1, \dots, n_m)}{n}\right) := - \sum_{i=1}^m \hat{\theta}_i \log \hat{\theta}_i$$

Definition 4. The code length of an i.i.d. sequence $x_{1:n}$ for $x_i \in \mathcal{X}$ where $\mathcal{X} = \{1, \dots, m\}$ is

$$CL(x_{1:n}) : = \begin{cases} CL((n_1, \dots, n_m)) := nH\left(\frac{(n_1, \dots, n_m)}{n}\right) + \frac{m-1}{2} \log n & \text{if } n > 0 \\ 0 & \text{else} \end{cases}$$

The code length is the length of the arithmetic code with respect to the model plus a penalty for coding the parameters. Each parameter $\frac{n_i}{n}$ can be coded to sufficient accuracy ($O(1/\sqrt{(n)})$) within $\frac{1}{2} \log_2(n) + O(1)$ bits (Hutter, 2009b).

Definition 5. The original cost ($OCost$) is defined as follows,

$$OCost(\phi \mid h_n) := CL(s_{1:n} \mid a_{1:n}) + CL(r_{1:n} \mid s_{1:n}, a_{1:n}) + CL(\phi)$$

where the state sequence $s_{1:n}$ comes from mapping each history h_t to a state s_t for $t \in [1, n]$. Then, $CL(s_{1:n} \mid a_{1:n})$ is the code length of the state sequence given the action sequence. The subsequence of states reached from a given state s via action a is i.i.d. as it is sampled from an MDP. We form a frequency estimate of the model of this MDP. The coding is optimal by construction. $CL(r_{1:n} \mid s_{1:n}, a_{1:n})$ follows similarly. $CL(\phi)$ is a complexity penalty on ϕ and can often be ignored in practice.

We primarily care about predicting the reward sequence. If the agent can predict the reward signal well it can act to maximise it. The code for the rewards depends on the states, and thus coding the state sequence is also necessary. The state sequence captures long-term information necessary for predicting the rewards. The *Cost* is well-motivated since it balances between coding states and coding rewards. A state space that is too large (worst case being $\phi(h) = h$) results in poor learning and a long state coding, while a state space that is too small can obscure structure in the reward sequence resulting in a long code for the rewards.

The consistency of this cost criterion was proven by Sunehag and Hutter (2010) for a restricted set of ϕ namely bounded finite state machines, primarily in the passive case where the agent does not have to choose actions.

The modified cost by Nguyen et al. (2011) adds a parameter α to control the balance between reward coding and state coding,

$$OCost_\alpha(\phi \mid h_n) := \alpha CL(s_{1:n} \mid a_{1:n}) + (1 - \alpha) CL(r_{1:n} \mid s_{1:n}, a_{1:n}) + CL(\phi)$$

Nguyen et al. (2011) search the map (ϕ) space of suffix trees (Definition 6). The generic Φ MDP algorithm is given in Algorithm 3. The agent is first initialised with some history based on random actions. Then it alternates between finding a “best” ϕ using the *simulated annealing* algorithm (Algorithm 4) and performing actions based on the optimal policy for that ϕ found via the *FindPolicy()* function. The *FindPolicy()* function can be any standard reinforcement learning algorithm that finds the optimal policy in an unknown MDP, and should perform some amount of exploration, generally via an optimistic initialisation.

Algorithm 3: A high-level view of the generic Φ MDP algorithm.

```

Input : Environment  $Env()$  , Cost function  $Cost()$ ;
Initialise  $\phi$  ;
Initialise history with observations and rewards from  $t = init\_history$  random actions;
Initialise  $M$  to be the number of timesteps per epoch;
while true do
     $\phi = SimulAnneal(\phi, h_{1:t}, Cost)$ ;
    for  $k = 1$  to  $t$  do
         $s_k = \phi(h_k)$ 
    end
     $\pi = FindPolicy(s_{1:t}, r_{1:t}, a_{1:t-1})$ ;
    for  $i = 1, 2, 3, \dots, M$  do
         $a_t \leftarrow \pi(s_t)$ ;
         $o_{t+1}, r_{t+1} \leftarrow Env(h_t, a_t)$ ;
         $h_{t+1} \leftarrow h_t a_t o_{t+1} r_{t+1}$ ;
         $t \leftarrow t + 1$ ;
    end
end

```

Map classes

A very commonly used map class by history-based RL methods is that of suffix trees. These are variable-order Markov models. A complete tree of depth n represents an n -order Markov model, but a partial tree can more efficiently represent certain long dependencies without forcing superfluous observations onto other parts of the tree. We formally define a suffix tree as follows.

Definition 6. Let $\mathcal{O} = \{o^1, o^2, o^3, \dots, o^d\}$ be a d -ary alphabet. A *suffix tree* is a d -ary tree in which the outgoing edges from each internal node are labelled by the elements of $\mathcal{O}$. Every suffix tree has a corresponding suffix set which is the set of strings $\mathcal{S} = \{s^1, s^2, \dots, s^n\}$ generated by listing the labels on the path from each leaf node to the root of the tree.

The suffix set has the property that no string is a suffix of any other string and any sufficiently long string must have a suffix in the set. Each string in the suffix set is called a state, and hence this is also called a suffix state set. The l -th level of the tree corresponds to the l -th last

observation in the history. By the above properties, any history of sufficient length must be mapped to one and only one state based on its suffix. An example of a suffix tree is shown in Figure 2.4.

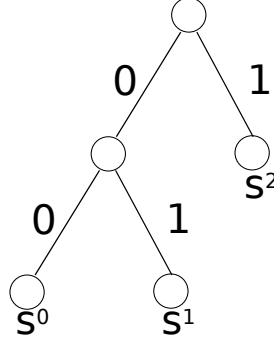


Figure 2.4: A suffix tree that maps strings that end in 00, 10 and 1 to s_0 , s_1 and s_2 respectively.

CT Φ MDP (Nguyen et al., 2011) uses the above Φ MDP algorithm along with simulated annealing over the space of context trees to find the best map, and then using approximate value iteration (AVI) to find the best policy.

2.4.1 Feature dynamic Bayesian networks

In a factored MDP (Boutilier et al., 1995) the state space is factored into n features X_i , i.e. a state in a standard MDP is represented by a feature vector $\mathbf{X} = (X_1, \dots, X_n)$, with each feature taking on values from some finite set $Val(X_i)$. The transition function of such an MDP is given by a Dynamic Bayesian Network (DBN) (Dean and Kanazawa, 1992) which provides a way of representing conditional independences over a time-slice. Formally, a DBN is a tuple (G, Θ) . G is a directed acyclic graph over $\{X_i\}$ that encodes the conditional independence assumptions over a time-slice. Each variable X_i at time t is only dependent on parent variables given by the set $Pa(X_i)$ at time $t - 1$. Intra-temporal (within time-slice) dependencies can also be allowed. Θ is a parameter set such that $\theta_{i,j_i,k_i} = Pr(X_i = k_i \mid Pa(X_i = j_i))$ for all $k_i \in Val(X_i)$ and sets j_i representing possible combinations of values of $Pa(X_i)$. The probability of a feature vector $\mathbf{X}$ can be composed from the independent conditional probabilities as $Pr(\mathbf{X}) = \prod_{i=1}^n Pr(X_i \mid Pa(X_i))$.

Φ MDP faces the problem that very large unstructured MDPs are necessary to represent a complex environment. Operations on large MDPs scale polynomial in $\mathcal{S}$. Φ DBN (Hutter, 2009a) is an extension of Φ MDP to learn a factored MDP representation. Learning a factored MDP results in operations that scale polynomial in the number of factors. However there is added complexity in learning a structure along with the appropriate feature map. Additionally, while the factors are used to form representations of the state, it is not as straightforward to form representations of the reward i.e. the factors that we use to model the states may have no relation to modelling the (scalar) reward. One way of solving this is to treat the prediction of the reward as a regression problem on the features. The rewards can then be coded according to the model found by the regression.

Algorithm 4: Simulated Annealing Search

SimulAnneal();

Input :

getNeighbour() : A neighbour function providing the next map;

schedule() : The cooling scheme;

initialMap : The starting map in the search;

Cost() : A cost function;

h : The history sequence so far;

currentMap = bestMap := initialMap;

currentCost = bestCost := Cost(h, initialMap);

for $t \leftarrow 1$ **to** N **do** candidateMap $\leftarrow$ currentMap.getNeighbour(); candidateCost $\leftarrow$ Cost(h, candidateMap); $\delta \leftarrow$ currentCost - candidateCost; $T \leftarrow$ schedule(t); $p \leftarrow \text{uniform}(0, 1)$; **if** $p < e^{\delta} / T$ **then** currentMap $\leftarrow$ candidateMap; currentCost $\leftarrow$ candidateCost; **if** currentCost < bestCost **then** bestMap $\leftarrow$ currentMap; bestCost $\leftarrow$ currentCost; **end** **end****end**

2.4.1.1 Structure Learning

Structure learning of a DBN is the problem of finding a graph structure that best represents the conditional independences given an observation history. There are three broad categories of structure learning algorithms: constraint-based, score-based and Bayesian model averaging (Koller and Friedman, 2009). *Constraint-based algorithms* perform independence tests to find a set of conditional independence properties, and then optimise structure with respect to those constraints. There are some issues with the robustness of these methods as they strongly depend on the quality of the independence tests. *Score-based algorithms* use a scoring function to measure how well a model fits the data. A heuristic search method is (normally) then used to find models with the highest scores. In general finding a model that has the highest score is NP-hard (Chickering, 1996). The scores are generally motivated by the minimum description length principle (MDL-score) or Bayesian approaches (Bayesian Information Score (BIC), BDe, etc). *Bayesian Model Averaging* does not learn a single structure, rather it maintains a weighted Bayesian mixture over possible structures. The predictions are then the average predictions of this ensemble of structures.

While these are broad categories, there are methods that don't quite fall into them perfectly, for example Guo and Schuurmans (2006) convexify the structure learning problem by expressing the Bayesian network in a general exponential form and then using a convex relaxation to learn an optimal feature ordering in polynomial time. While the algorithm performs better, it is also much slower in practice than heuristic score-based methods.

If the observations are not Markov but based on some hidden stationary Markov process (for example when we have a partially observable problem) the data is then said to be incomplete, and the hidden variables must be learned. When the data is Markovian, the data is said to be complete.

In dealing with DBNs for RL, we can also distinguish between passive and active structure learning. Passive structure learning algorithms learn structure based on the data provided but do not actively influence the policy to choose actions that allow it to learn a better structure, whereas active learning algorithms (Jonsson and Barto, 2007) do.

In the case of the DBNs that we need to learn for Φ DBN we fall into the complete data, passive learning characterisation. The problem has complete data because given a particular map the state sequence is Markovian. Since the structure is learned based on the current map which is itself being optimised, it is not possible to have an active learning algorithm. The $OCost$ function when a particular feature map is fixed reduces to the BIC score. Additionally, if we make the assumption that there are no intra-temporal dependencies then the BIC perfectly decomposes according to the conditional probability distributions which gives computational advantages.

2.4.1.2 Feature Generation

To extract a good state representation we need a good class of state representations. We also need a class that is computationally feasible to search over. Previous works have generally

used the suffix tree class, but this has limitations, particularly in representing long-term dependencies. In Φ DBN we have the additional problem that the generated features should be suitable for the structure search problem.

2.4.1.3 Planning in Factored MDPs

The advantage of having factors with conditional independences is that we can efficiently represent the probability distribution over the features. However, the value function does not usually factor in the same way that the distribution does, or at all. If we want to learn a value function exactly for a factored MDP, then we may need to use the states directly, which are exponential in the number of features. This defeats the purpose of using factored MDPs. Hence, methods such as factored value iteration (Szita and Lörincz, 2008a) find approximations to the value function that can be efficiently factored, and converge with some guarantees.

2.4.1.4 Learning in Factored MDPs

There are also factored analogues to existing PAC-MDP algorithms such as RMAX, E^3 and OIM. Factored RMAX and Factored E^3 methods rely on access to a near-optimal planner in order to guarantee an approximately optimal policy. Factored OIM relaxes this condition and simply performs as well as it can (ϵ -close) given the planner that it has. Most recently, UCRL-Factored and Posterior Sampling Reinforcement Learning (PSRL), both by Osband and Van Roy (2014), are the first factored MDP algorithms to satisfy near-optimal polynomial regret bounds for factored MDPs with known structure. UCRL-Factored is based on UCRL2 Jaksch et al. (2010) and PSRL is based on the old Thompson-sampling heuristic by Thompson (1933).

Strehl et al. (2007) combines structure learning with parameter learning to form an algorithm called SLF-RMax. SLF-RMax relies on an admissible learning algorithm that treats the structure learning problem as an online learning problem, along with a near-optimal factored MDP planner (such as RMax). Following this line of work, Diuk et al. (2009) improves the structure learning component by a reduction to the *adaptive k-metereologist* problem. The resulting new algorithm *Met-RMax* offers better sample complexity guarantees than SLF-RMax and performs much better in practice on the sample domains. Both SLF-RMax and Met-RMax rely on having a bound on the maximum in-degree of the (unknown) DBN, Chakraborty and Stone (2011) relax this assumption and show better sample complexity and empirical performance, albeit with an additional assumption about the planning horizon.

2.5 Predictive State Representations (PSRs)

Predictive representations of state (Littman et al., 2001; Singh and James, 2004) is an alternative way of framing and solving the general reinforcement learning problem that has gained

popularity over the last decade. A PSR is a vector of probabilities over predictions of some test events occurring in the future given the history so far. A length k test t is simply a history of length k , $t_{1:k} = ora_{1:k}$. The prediction of a test given the history so far is the probability that the first k interactions in the future will be exactly the same as the test. For example, given the test $a^1 o^2 r^1 a^2 o^3 r^2$ the prediction of this test is $Pr(o_{t+1} = o^2, r_{t+1} = r^1, o_{t+2} = o^3, r_{t+2} = r^2 \mid a_{t+1} = a^1, a_{t+2} = a^2, h_t)$.

Given some set of tests $\{t_i\}$ of size q , the *prediction vector* $p(h)$ has q -entries of the form $Pr(t_i \mid h) = \frac{Pr(ht_i)}{Pr(h)}$. $p(h)$ is said to be a predictive state representation if and only if it forms a sufficient statistic for the environment, i.e. for any test t and history h , $Pr(t \mid h) = f_t(p(h))$ for some projection function f_t . Littman et al. (2001) show that for every POMDP there exists an equivalent PSR with a linear projection function (linear PSR), having only as many tests as there are states in the POMDP. A set of tests Q is called a *core set* if any other test can be computed as a linear combination of tests in Q .

Given a new interaction the i -th component of a PSR can be updated as follows.

$$p_i(haor) = Pr(t_i \mid haor) = \frac{Pr(aort_i \mid h)}{Pr(aor \mid h)} = \frac{f_{aort_i}(p(h))}{f_{aor}(p(h))}$$

where we make use of the fact that $aort_i$ and aor are also tests. If the PSRs are linear then f_t can be represented by some $(1 \times k)$ -projection vector m_t and $f_t(p(h)) = p(h)m_t^T$ and the updates to the tests are simple dot products.

A more general view of PSRs can be arrived at by looking at the *systems dynamics* matrix. This is a infinite dimensional representation of the dynamical system being modelled. The rows of the matrix correspond to all possible histories, and the columns to all possible tests. The rank of this matrix is a measure of the complexity of the system and corresponds to the number of tests in the minimum linear PSR needed to model it. Singh and James (2004) point out that there are systems that have dimension k that cannot be represented by POMDPs with k hidden states. They also show that non-linear PSRs can provide an exponential reduction in the number of tests required.

In practice, learning PSRs is difficult. One must first find a suitable set of core tests (*discovery*) and then learn the values of the vectors m_t for each of these tests (*learning*). The most promising recent advances to PSRs include transformed PSRs (TPSRs) Rosencrantz et al. (2004), which use spectral methods to find a PSR in a transformed subspace defined by the core tests and the work on compressed PSRs (cPSRs) by Hamilton et al. (2013) which learns TPSRs via methods from compressed sensing. The latter has nearly state-of-the-art results on domains such as Pocman.

2.6 Connections to FRL

In this section I will describe connections to FRL from the background material that has been presented so far. I will also explain all the links in Figure 2.5 that have not already been explained above.

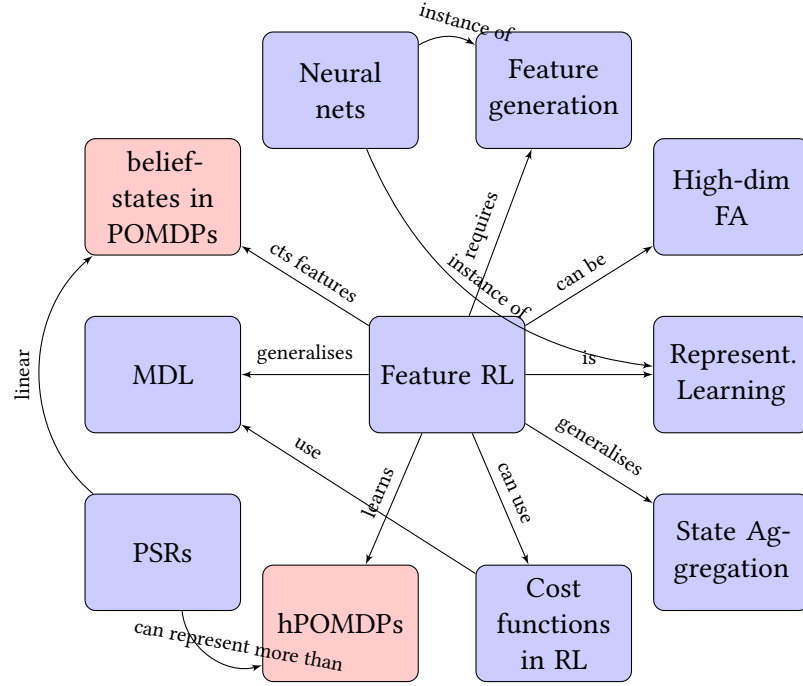


Figure 2.5: Connections to FRL

2.6.1 POMDPs

The class of hPOMDPs relates directly to Φ MDP; we know that in an hPOMDP there exists a map to a discrete MDP (namely the underlying MDP of the hPOMDP). Thus, Φ MDP is capable of representing hPOMDPs exactly.

The belief-state MDP associated with a POMDP is continuous. If FRL searched for maps from histories to MDPs with continuous state spaces, then one solution would be a map to the belief-state MDP, and Φ MDP would be able to represent all finite POMDPs exactly. Although we only search for maps to discrete MDPs in practice, we show empirically that we can still find good approximate solutions within this space for POMDPs with continuous belief states that allow us to act well within the POMDP.

2.6.1.1 Belief states over histories

Let T^a be an $|\mathcal{S}| \times |\mathcal{S}|$ matrix whose entries are $T(s' | s, a)$. Let O^o be a diagonal $|\mathcal{S}| \times |\mathcal{S}|$ matrix where the entries on the diagonal are $\Omega(o | s)$. Let $e_{|\mathcal{S}|}$ be a $(1 \times |\mathcal{S}|)$ -vector of all ones. Then belief states can be written as a function of the history as follows (Littman et al., 2001).

$$b(hao) = \frac{b(h)T^aO^o}{b(h)T^aO^oe_{|\mathcal{S}|}^T}$$

2.6.2 Function approximation in high-dimensional state spaces

Function approximation is often used to learn continuous and large MDPs. Histories can be viewed as a countably infinite dimensional state space, with the property that no state occurs more than once. Finite histories are simply a very high-dimensional state space. Hence generalisation over the state space is needed in order to perform learning and standard function approximation techniques apply. Function approximation techniques are wide and varied, and we avoid a full discussion of them here. We focus instead on making the link to FRL clearer by briefly discussing one approach to feature selection for function approximation in high-dimensional spaces.

Feature selection in high-dimensional spaces can be solved via sparsity imposing techniques. The loss is normally a function of the Bellman error plus a regularisation parameter that imposes sparsity, such as an ℓ_1 norm of the approximating parameters. For example,

$$L(w) = \frac{1}{2} \|w^T \phi - (R + \gamma w^T \phi')\|_2^2 + \beta \|w\|_1$$

where $\phi = \phi(s)$ and $\phi' = \phi(s')$ are k -dimensional features of the state and $w \in \mathbb{R}^k$. In the case that we have value functions over the history, we can define features over the history instead. This is the approach we take in Chapter 5 which will make this above link more concrete.

2.6.3 State aggregation

We are interested in state aggregation over environments that are not MDPs. Recently, Hutter (2014) gave an alternate similarity metric that applies to general environments. He defines conditions for exact and approximate aggregation.

We can define a reduction from the process P generating the environment to a new reduced process P_ϕ via a map on the histories ϕ . Then P_ϕ is (exactly) an MDP if and only if it is identical for all histories mapped to the same state, i.e.

$$\exists p : \forall \tilde{h}_t, \forall s_t (\phi(\tilde{h}_t) = s_t) \implies P_\phi(s_{t+1}, r_{t+1} \mid \tilde{h}_t a_t) = p(s_{t+1}, r_{t+1} \mid s_t, a_t)$$

In this case Hutter (2014) gives the result that the value functions and policies over the histories correspond exactly to the value functions and policies defined over the states in the reduced MDP, given the map ϕ . This is not surprising; the rest of the paper focuses on the much more interesting case where the aggregations are approximate and the MDP condition fails.

Hutter (2014) shows that even when the reduced process P_ϕ is not an MDP as long as the (action) value function can be represented by the aggregated states with enough accuracy, it is

possible to define an MDP p such that a policy constructed in p performs well in the original process. A condition known as ϕ -uniformity of the (state or action) value function results in bounds on how well a policy constructed from states in p would do in the true process P . ϕ -uniformity intuitively refers to the ability of the map ϕ to distinguish between histories whose values differ significantly. More formally, Q is said to be ϕ -uniform for some policy $\Pi : \mathcal{H} \rightarrow \mathcal{A}$ if for all h and $\tilde{h}$ such that $\phi(h) = \phi(\tilde{h})$ and for all a , $|Q^\Pi(h, a) - Q^\Pi(\tilde{h}, a)| \leq \varepsilon$. For a full discussion of the results we refer the reader to the original paper.

2.6.4 PSRs

This brief subsection explains the link between linear PSRs and POMDPs. Littman et al. (2001) showed that there exists a linear PSR for every POMDP that generates history sequences with the same probability as the POMDP.

$$Pr(a_1 o_1 \dots a_l o_l \mid h) = b(h) T^{a_1} O^{o_1} \dots T^{a_l} O^{o_l} e_n^T$$

This means that linear PSRs can represent POMDPs exactly. While in the worst case a linear PSR uses as many core tests as there are hidden states in the POMDP, it can use many fewer, for example when the underlying MDP is factored (Littman et al., 2001).

2.6.5 Representation learning

Representation learning is not a well-defined term. In its most general sense, it refers to the unsupervised discovery of useful features from some data set. A representation is said to be good when it captures explanatory details of the feature set that are helpful in some other algorithmic task, like supervised learning or probabilistic prediction (Bengio et al., 2013). Representation learning is often designed to minimise a data reconstruction error, i.e. how well can the inputs be reconstructed from the representation?

Bengio and LeCun (2007) introduce the idea of the *AI-set*, “the set of all the tasks that an intelligent agent should be able to learn”. They hypothesize that there exist representations which are useful for many tasks within the AI-set, and the goal of representation learning should be to find representations of the data that can compactly describe regularities in it. This view of representation learning shares the same goal as approaches like *minimum description length* by Rissanen (1978). Other commonly used tools for representation learning include principle component analysis (PCA, F.R.S. (1901)), subspace discovery and other spectral methods.

Representation learning in recent years (since 2006) has been dominated by a new paradigm called *deep learning*. Bengio et al. (2013) is an excellent survey of the recent advances in this area. Deep learning is fundamentally built around the idea that good representations are often hierarchical, and that each layer is a composition of simpler functions in the preceding layer.

In particular, deep learning uses representations known as *deep belief networks* where each layer is a restricted Boltzmann machine. The primary difference between deep belief nets and neural networks is that there is an initial unsupervised training phase for the deep belief net, where each layer is trained with a reconstruction error based on the input from the previous layer. After the unsupervised phase, the deep belief net can be trained via backpropagation as a neural network for any particular task (e.g. classification). The recent comprehensive survey by Schmidhuber (2014) gives a historical view of deep learning for neural networks from the early days (1940s) of neural nets as regressors to the recent boom of deep learning triggered by the unsupervised pre-training of *deep belief networks* (Hinton et al., 2006) for supervised learning tasks, particularly in computer vision.

Representation learning has seen some use in reinforcement learning recently. The recently published work by White (2014) integrates representation learning and least squares TD-learning (LSTD). Since this work is very relevant to us, we provide some of the details here.

The standard view of LSTD is that we are trying to find a weight vector $w \in \mathbb{R}^k$ to minimise the mean squared projected Bellman error (MSPBE). Let T^π be the $|\mathcal{S}| \times |\mathcal{S}|$ transition matrix given the actions fixed by a deterministic π and R be the reward vector.

$$MSPBE(w) = \|\phi w - \Pi(R + \gamma T^\pi \phi w)\|_D^2$$

Performing the minimisation $\min_w MSPBE(w)$ can be done via the least squares solution $A^{-1}B$ where $A = \phi^T D(\phi - \gamma P^\pi \phi)w$ and $b = \phi^T DR$. The finding of the above work was that LSTD could be seen as the minimisation of a squared loss plus a Concave Regulariser as follows.

$$CRTD(w) = \|\phi w - R\|_D^2 - 2\gamma w^T \phi^T D T^\pi \phi w$$

White (2014) then goes on to add a regularised loss over the representation ϕ to this loss to create a Factored Representation of CRTD,

$$FRCRTD(w, \phi, B) = \|\phi w - R\|_D^2 - \gamma f(w, \phi) + L(\phi B, X) + \alpha \|\phi\|$$

where X is a feature set (cross-product of all observations), B is a set of some basis vectors and α a weight on the regulariser. $L(\phi B, X)$ is a convex loss and the norm on ϕ could be a sparsity-inducing norm.

FR-CRTD contains both a supervised learning term and an unsupervised learning loss. In the case that ϕ is known it reduces to the CRTD error, however in general it must also find a suitable representation ϕ of the training set on the observations, which allows it to minimise the LSTD error.

In Chapter 5 we define a loss function for Φ MDP that is similar in spirit to this one, albeit without an explicit unsupervised learning term. $OCost$ can also be interpreted in this representation learning sense. There is one term, namely the reward coding $CL(r_{1:n} \mid s_{1:n}, a_{1:n})$, that is the true objective which is parameterised by a representation $\mathcal{S}$, and then there is the representation coding $CL(s_{1:n} \mid a_{1:n})$. In the case of an injective class of maps where $\phi(s_{1:n}) = o_{1:n}$, the state coding term could be viewed as a representation error.

Domains for RL

“ The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent. ”

Russell and Norvig, 2009.

In this chapter, we examine a selection of domains that could be used for testing reinforcement learning algorithms. We do a brief (and selective) summary of existing work for each, and also comment on popularity and ease of implementation. We use the work in this chapter to decide on which domains to test the algorithms we define in the rest of this thesis. We hope that this chapter may assist other practitioners in carefully choosing the correct domains to test their algorithms. The most useful part of this chapter is Table 3.1, which shows the properties of each domain. A large number of the surveyed environments are games, although some real world applications are also considered.

3.1 Introduction

Games have long captured the imagination of humankind. Physical and intellectual games have been used for many centuries as a form of entertainment and to test determined participants in feats of strength, willpower and intellectual prowess. In the modern era, computer games in particular have become very popular amongst children and adults alike, and some can even approach movies and books in their scope, for story telling and visual effects, all the while retaining both strategic and tactical aspects of more traditional games.

AI practitioners have traditionally used games to show their progress to others within the field as well as to the wider public. The great public successes of AI have been on games such as checkers, chess, backgammon and most recently Jeopardy. Some computer games also use AI bots to interact with the user, although the sophistication of these bots vary from being completely rule-based to intelligent reinforcement learners¹. The vast arena of games serve as

¹The game “Black and White” LIONHEAD STUDIOS (2001) designed by Peter Molyneux included a creature that the player could teach by providing reinforcement.

useful benchmarks in the progress of AI. Games have simple rules, specific goals and obvious metrics. Given that games have been created by humans for humans to play, we could also view the ability to play a wide-range of games as evidence for possessing general intelligence. Indeed this was the intent behind the creation of the Arcade Learning Environment (ALE) by Bellemare et al. (2013) which provides an interface to games from the once popular ATARI 2600 console.

3.2 Preliminaries

We considered the following attributes when filling in Table 3.1.

- Deterministic vs. stochastic.
Does the domain include stochastic transitions between environment states?
- Size of state space.
Small, medium, large. We will also specify whether the state space was continuous by attaching *cts* to the size, i.e. *large cts*, is a large, continuous state space.
- Size of action space.
Similar to the above.
- Partially observable vs. fully observable.
Is the domain partially observable? Also known as the imperfect information setting when dealing with partial observability due to multiple agents.
- Static vs. Dynamic.
An environment is static if the environment cannot change while the agent is considering which action to take.

The following properties refer to our judgements about the suitability of the domain for feature reinforcement learning algorithms at the current time.

- Feasibility.
We will use this term to indicate how feasible we believe the domain to be for the success of a general RL algorithm, in particular a feature RL algorithm. This is not a well-defined criterion, but might offer some insight into choosing a domain. For example, if we believe a particular game to be beyond the reach of current techniques we can choose to ignore it, or choose to focus on changing the domain to make it more tractable.
- Relevance.
Low, medium, high. The criterion for relevance is whether the domain is interesting/important to us, and to other people? We will also consider the popularity of the game when making this judgement, where popularity is measured by various factors, including the number of people signed up to play/watch the game. Statistics that are easily available will be used.

- Implementability.

How easy is the domain to implement and/or interface with? For example, the game of Starcraft might be very interesting as a challenging domain to RL practitioners but the initial implementation costs in setting up a running bot that injects into the memory of the game might be too high.

The following properties do not make it into the final table.

- Single vs multi-agent.

All problems we consider here are multi-agent. However given a fixed opponent, these can be thought of as single player games. For example, in the Arcade Learning Environment the agent could play against the default ATARI AI for many games. In this case, the environment is single-agent.

- Episodic vs sequential.

Episodic tasks repeat the same task over time (most/all games are episodic). The episodic/sequential nature of tasks is important, however all but one (the finance domain) of the tasks we consider here are episodic.

3.3 Domains

For each domain we will give a description, our estimate of the popularity/relevance of the domain and a brief summary of some existing reinforcement learning work in the domain. If implementation appears to be a significant task, then we will also provide a summary of the existing available code base.

3.3.1 Starcraft: Brood War

Starcraft: Brood War is the expansion to the original Starcraft 1 and one of the most complicated Real Time Strategy (RTS) games ever made. Agents play against each other in 1v1 games on maps which have varying terrain that can affect the strategy used in the game. Agents can play as one of three races, Protoss, Terran and Zerg which each have different units and abilities. The game includes fog-of-war (i.e. you only see what your units can see, so there is imperfect information).

Existing work. Existing agents work in the planning framework, with micromanagement of units handwritten. RL has been used for micromanagement by Wender and Watson (2012) who compare Q-learning, SARSA and others in a very simplistic setting (one unit against multiple enemy units, with two actions “fight” and “retreat” available to it). In Huang (2011), the Berkley bot Overmind uses planning for high-level build order manipulation, and reinforcement

learning to train “potential repulsion fields” for mutalisk² control (micro). EISBot uses ABL (A Behaviour Language), a reactive planning language, to specify the goals and game states of the agent. The agent is split into managers that take care of different areas of competence: strategy, income, construction, tactics and recon. Each accomplishes certain subgoals, which are part of an overall goal. The managers also have interfaces to interact with each other (in ABL).

In the 2012 AIIDE Starcraft AI competition, bots were allowed to retain information about previous matches. This allowed for the use of meta-gaming, which involves using knowledge about common strategies, or strategies generally used by a particular player/bot. This provides an interesting additional complexity to the setting. An additional constraint could be introduced in the form of limiting the number of actions per minute (APM) which is a commonly used measure of mechanical skill of human Starcraft players, but is not capped in the AI competitions. Uncapped APM allows bots like Overmind to perform micromanagement of units on a super-human level, which is unfair given that they are unconstrained by physical limitations.

Popularity. Starcraft Brood War is fading in popularity with a few thousand people still playing it. Most players have shifted to the new edition Starcraft 2, which has better graphics and different units, but is conceptually similar. However no similar API exists for this new version³. About 290,000 players currently play Starcraft 2 (Starcraft Worldwide Rankings, 2015), with 100,000 watching live streams of the game (both online and on television primarily in Korea and Sweden).

Implementation. There is a pseudo-API available, and widely in use for the competition at AIIDE (Artificial Intelligence and Interactive Digital Entertainment). Unfortunately this is not a true API, but rather an injected bot inserted into the memory of the (closed-source) BW executable. A functional starting bot is available as an open-source Google code project with full documentation. All code from previous competitions has also been released open-source making getting started relatively easy.

3.3.2 Poker

Poker is a betting card game with several variants. The variant of choice at major tournaments, and perhaps the most strategic, is known as Texas Hold'em. In Texas Hold'em each player is dealt 2 cards, after which they can bet to see the next three (the flop). After the flop another round of betting occurs to see the next card (the turn) and a final round of betting before the last card (the river). During a betting round, the following actions are available to a player:

²Mutalisks are flying creatures in the Zerg race. They tend to cluster closely when in groups which make them vulnerable to certain attacks. Reinforcement learning was used to learn the optimal clustering behaviour when attacked.

³Matt Fisher uses the D3D9 Interceptor to intercept the Direct3D API, giving a detailed view of the graphical state of the game. He has written an AI based on this graphical API (<http://graphics.stanford.edu/~mdfisher/GameAIs.html>).

fold (leave the game), check/call (commit the amount necessary to stay in the round, this is zero for a check) and bet/raise (bet an amount, or raise the amount on the existing bet). Also significant to us is the distinction between *limit* and *no-limit* poker. As the name suggests, in limit poker there is a limit to the maximum bet one can make. Having a limit on the maximum bet is beneficial to computer poker agents because it reduces the size of the actions space. *Heads-up* poker refers to a poker game between only two players.

Poker is a qualitatively different game from the traditional domains of chess, Go and checkers, in that it is a *stochastic* game of *imperfect information*. Billings et al. (2002) identify the qualities that need to be employed to be a world-class poker player: the ability to evaluate hand strength, evaluate hand potential, deal with unpredictability, and to model opponents.

The literature on poker is vast, and could easily fill up this entire chapter. We aim to only give a brief overview here. For a comprehensive survey (up to 2011), see Rubin and Watson (2011). We follow their lead in categorising poker bots into knowledge-based systems, simulation-based systems and game theoretic approaches. We also consider *opponent modelling* to be a separate category.

Existing work. Knowledge-based systems rely on heuristic formulas and other domain knowledge to approximate each of the characteristics identified by Billings et al. (2002). An example of a heuristic that humans use is the *pot odds*, a simple formula used to judge whether to stay in a round based of the size of the current pot and the cost of staying in the round. Similarly, evaluating hand strength and hand potential can be done in heuristic and rule-based ways.

Opponent modelling (Billings et al., 1998) follows two approaches. One is to build a model to predict what an opponent will do next, the other is to classify an opponent (e.g. this player is aggressive) which can then be used to exploit the opponent. In the predictive line of work, Southey et al. (2005) provide a Bayesian approach evaluated on both a simplified domain and the full Texas Hold'em domain, where Thompson sampling (Thompson, 1933) is used since calculating the posterior is intractable. Bard and Bowling (2007) treat opponent modelling as a state estimation problem, which they solve using a particle filter based approach. This approach has an advantage over previous work, in that it does not treat the opponents as static processes. However, the work was only evaluated on the toy domain of Kuhn poker. Reinforcement learning in conjunction with neural networks has also been used for predictive opponent modelling by Dahl (2001), but only in a very simplified setting. In the classification setting, Lockett and Miikkulainen (2008) use a neuroevolution approach called NEAT (Stanley and Miikkulainen, 2002) to learn the weights of a mixture model over a set of opponent models based on game-theoretic approximations.

Simulation-based methods use Monte-Carlo based algorithms to evaluate the chance of winning a round. A modified UCT proved somewhat successful at beating a human novice when playing 400 games with a significant profit. Although UCT has not (yet) seen the successes in Poker that it has in other games like Go, there has been much interest in pursuing this line of work. Most recently, Heinrich and Silver (2014) examined the performance of UCT agents learning

via self-play, with the result being that “self-play MCTS can be competitive with the state of the art of 2008”.

Game-theoretic approaches form the most significant line of work in poker research. The year 2008 mentioned in the previous paragraph is significant because it is the first year that a computer poker agent (Polaris by Johanson (2007)) beat humans at heads-up limit hold-em. Polaris used an approach called bucketing to separate hands and sequences of hands for which the strategies are similar. Bucketing reduces the state space and makes the calculation of the Nash equilibria tractable. The biggest recent breakthrough in computer poker was the declaration that heads-up limit Texas hold'em is now “weakly solved” by Bowling et al. (2015). They use a new variant (CFR^+ by Tammelin (2014)) of the counterfactual regret minimisation algorithm (CFR) which iteratively provides better approximations the Nash equilibrium for extensive form games. The implementation required “a cluster of 200 computation nodes each with 24 2.1-GHz AMD cores, 32 GB of RAM, and a 1-TB local disk”, a significant computational cost.

Popularity. PokerScout⁴ estimates that around 60,000 people actively (24hr peak) play online poker, of which around 35,000 play for money. A drawback of the domain is that there appears to be no consistent rating scheme for poker. Online poker does not have an ELO⁵ system to match players, but instead offers differently staked tables. Tables go from (small/big blind) 1c/2c to \$300/\$600, with the idea that good players play with more money (or at least if bad players do, they lose their money quickly).

Implementation. Statistics⁶ for top players on various sites show that even the best players are only ranked high enough to win some money in a tournament between 15% and 20% of the time. The variance of the game means that to demonstrate good performance one needs a large amount of data. In order to reduce this variance, the Annual Computer Poker Competition (ACPC) uses a duplicate match structure. In this structure, every pair of agents plays two games, with the second game having the same hands as the first but in reverse order.

3.3.3 Chess

Chess is a symmetric two-player game that is played on an 8x8 checkered board. Each player has 16 pieces consisting of six types available to them. A game of chess can be won, lost or drawn. Winning occurs when one player puts another in a checkmate position or a player resigns. A player is said to be checkmated when her king is under attack and no legal moves can be made to stop or avoid the attack. A game is drawn when a player has no legal moves available but is not under check. Such a position is known as a stalemate.

⁴Poker Scout (<http://www.pokerscout.com/http://www.pokerscout.com/>) accessed on 2 Aug 2015.

⁵The ELO rating system (named after Arpad Elo) is used to measure the relative skill of players in competitive games. The difference between two players' ELO provides an estimate of the probability that one player will win over the other. It was initially used for chess, but many other games have adopted similar systems.

⁶Official Poker Rankings (<http://www.officialpokerrankings.com/pokerleaderboards.html>) accessed 2 Aug 2015.

Existing Work. The history of getting machines to play chess against humans starts in 1770 with the mechanical Turk. Unfortunately, the original mechanical Turk was a hoax, with several different chess masters sitting inside it throughout its 84 year campaign. The big successes of Deep Blue (Campbell et al., 2002) against Kasparov in 1996-1997 represent the true mechanical Turks. However, mechanical is exactly the way to describe Deep Blue, since it did not learn from experience; rather it used its superior computation power to play out entire game trees (with pruning).

Given the vast history of chess AI, we focus here on reinforcement learning within chess. NeuroChess by Thrun (1995) uses TD(0) and neural networks with an additional neural network of domain knowledge having been separately trained on grandmaster games. Interestingly, Thrun reports that training on the grandmaster games can sometimes be harmful, for example when a grandmaster moves a queen to the center of the board it is relatively safe compared to when a new player does something similar; in the grandmaster database a centered queen had a positive correlation with winning. Overall, the neural network trained seems very small compared to today's standards (175 input, 0-80 hidden units and one-output unit), and the program did not use opening and closing databases so its play may have been highly affected by poor openings which resulted it losing 86% of its games against the GNU-chess bot. However, as a first attempt at reinforcement learning in chess it could be seen as relatively successful, since the program managed to learn some useful behaviours.

The idea of using TD in conjunction with the standard minimax algorithm is called TD-LEAF Baxter et al. (2000). TD-LEAF was used to train the parameters in the evaluation function of a chess program "Knight Cap" and improved its ELO from 1650 (evaluation using only material lost) to 2150. The training was performed online, by placing the bot on the Free Internet Chess Server. The hand-coded weights however proved to be very close in performance to this (within 50-100). Additionally the authors report initial tests which show some improvement even after initialising the parameters to the hand-coded parameters but this improvement is within the range of variance. The authors also report issues with learning via self-play which could restrict the use of this approach.

TD with neural nets was once again used in Levinson and Weber (2001), where a graph based representation of the board using 64 neighbourhoods (centered at each square is used), along with a 2-layer neural network with TD to learn the value function for this representation. However, this agent performs worse than the method described above, however comparable to Morph IV (Levinson and Weber, 2000) which at the time was "one of the best from-scratch methods".

Stockfish is the strongest open-source chess engine currently available, ranking in the top three chess programs available today. However, StockFish itself uses many variables (ChessProgrammingWiki, 2014) for the evaluation function which are tuned by self-play; agents with different parameters play each other (30,000-100,000 times) and the parameter is updated in the direction of the winner of each of the games. This is extremely ad-hoc and as they say, not likely to converge. However, the tuning method results in 40-70 ELO point improvement. It is conceivable that a more sophisticated method using TD-leaf could further improve the performance of Stockfish.

Popularity. Chess is one of the most popular games in the world. The statistics for the number of chess players vary widely between sources but online sources can provide a rough estimate of the popularity of the game. On Chess.com, which is arguably the most popular online chess playing website, there are approximately 11 million players registered. The number of players online at any point appears to range from 25,000 to 50,000⁷. Unfortunately, the site does not make their daily numbers public. There are several other smaller websites (Gameknot being the closest competitor at 1.4 million registered competitors) bringing our estimate of online chess play to close to 12 million players online. An indication of competitive membership is given by the United States Chess Federation (USCF) having just over 75,000 Players registered in 2012. There are around 3000 International Masters active in the world, and 1380 Chess grandmasters.

3.3.4 Go

Go is a two-player board game played on a 19x19 board (9x9 and 13x13 variants exist) that is known for having very simple rules, but being hard to master. Unlike chess there is only one type of piece called a stone. Players take turns to place (black/white) stones on the board with the aim of surrounding a larger (total) area of the board than the opponent. Stones can be captured if their liberties are removed (i.e. the stone is surrounded on all four sides by opponent stones) and captured stones count toward the score (territory + stones).

Existing work. There are several very good Go programs that have taken over a decade to get to their current performance level. The best programs (Crazy Stone, MoGo, Fuego, etc) at the moment are Monte Carlo Tree Search based (with most programs using UCT variants). Some programs use a hybrid of pre-programmed behaviours and models. Of particular note is MoGo by Gelly and Silver (2007) which was the first to combine UCT with Rapid Action Value Estimation (RAVE), a heuristic for arriving at quicker, low-variance (but biased) estimates of $Q(s, a)$ by averaging not only over all outcomes for a particular state-action pair but for any values that occurred within a playout from a particular state that took the same action. RAVE is used during the early stages of UCT where the standard estimates of the Q -values are inaccurate. MoGo also makes use of a Q -value function learned offline to improve its performance. The current highest ranking bot (2729 ELO) on the CGS server is troisgro “a version of MoGo running on a cluster with 32 octocores at 3 GhZ with infiniband interconnect” (Sensei XMP, 2015). Alternative methods include use neural networks to learn a local evaluation function. TD methods have been used to train such neural nets. Baier (2010) improves UCT methods by implementing a better (adaptive) playout policy but this work has not yet been picked up by the major Go bots.

Within reinforcement learning, Silver et al. (2007) uses TD-learning to learn weights for local shape features which are normally hand-coded by human experts. The resulting agent was tested on the Computer Go Server (CGS) against other bots and shown to have a reasonable performance (+1210 ELO), although much poorer than tree-based methods. Silver et al. (2012)

⁷This is an estimate from accessing the website at different points in the day.

use TD-search, a combination of TD-learning and tree search methods to beat all “traditional (pre-Monte Carlo) search and machine learning” bots on the 9x9 CGS server.

Very recently, deep learning has been used for move evaluation in Go by Maddison et al. (2014) with stunning results; it beats GNUGo, a traditional search program, 97% of the time and matches the performance of MCTS methods that require two million positions per move.

Popularity. The game of Go is particularly popular in East Asia where it is known under different names; Igo in Japan, Baduk in Korea, and Weiqi in China. According to the International Go Federation (IGF) the total number of players of the game is “well over 40 million, the overwhelming majority in East Asia”. The two main Go servers online are the International Go Server (IGS) and the KGS Go Server (KGS). IGS has about 3000-5000 members online (my observation), and KGS around 1000 players. As mentioned above there is also the Computer Go Server (CGS) which has 559 bots listed on its “all time” (since 2008) list. There are 34 bots listed on the active 9x9 playing list on Sensei XMP (2015) with 14 on the current 19x19 list.

Implementation. Testing Go agents is easy as there are several Go servers (CGS, KGS, etc) that allow the use of bots. However, the good Go playing agents require cluster-sized computing resources to play well on boards beyond 9x9 (although this is changing as personal computers include more cores). Given that stochastic planners require such resources, it is unlikely that the current state-of-the-art general RL techniques can perform well on this domain. However, due to its size there is also scope for RL since intelligent and automated function approximation can alleviate some of the issues associated with having such a large state space.

3.3.5 ATARI 2600 Games

Here we will look at the use of ATARI 2600 games for evaluating general RL agents. Since the literature on ATARI games is vast, we will look at two domains in particular.

3.3.5.1 Arcade Learning Environment

The reinforcement learning community has lacked a set of general environments that can be used for testing new algorithms in a robust manner. In Veness et al. (2011) a set of small challenging problems were provided, but several algorithms (Daswani et al., 2013; Nguyen et al., 2012) can no longer be differentiated based on them. The recently introduced ALE by Bellemare et al. (2013) attempts to address this big gap in the field by utilising games made for the ATARI⁸ 2600 as a test bed for reinforcement learning algorithms. The environments in this setting are games made for humans which can be relatively complex, but due to the space/processing limits of the ATARI console, still computationally feasible for current RL

⁸Interestingly, the word “atari” is also a concept from Go, used to describe the situation where a group of stones is soon going to be captured by one’s opponent.

techniques. The ALE consists of an interface to Stella which is an open-source ATARI 2600 games emulator.

This gives access to hundreds of games of this format, which range from side-scrollers, to arcade games, shooters and puzzles. The interface provides access to the screen pixel matrix and the internal state representation of the ATARI games themselves. This allows for both reinforcement learning and planning algorithms to be tested, since the ability to reset to a particular state is crucial for some planning algorithms like UCT.

“ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. ALE presents significant research challenges for reinforcement learning, model learning, model-based planning, imitation learning, transfer learning, and intrinsic motivation” Bellemare et al. (2013).

Existing work. Although the domain has not been around for very long (initial Master’s thesis by Naddaf (2010), full framework with benchmarks by Bellemare et al. (2013)), there is already a large amount of work being done with it. The initial works used SARSA(λ) agents based of simple features of the screen. Even these feature spaces were quite large, with the BASS feature set coming to over 2 million features. Since then there has been work on sketch-based linear value function approximation by Bellemare et al. (2012), factored models for observation spaces in Bellemare et al. (2013), deep reinforcement learning using convolutional neural nets to approximate the Q-value (DQN) by Mnih et al. (2013, 2015) and most recently an algorithm by Veness et al. (2014) for policy evaluation via constructing a consistent estimator for the Q-value function using existing (probability) density models. The last two algorithms out of the Google-owned AI research lab Deep Mind have results that convincingly surpass humans on several games.

Implementation. The ALE consists of an interface to Stella an open-source Atari 2600 emulator. Game states can be easily saved, resulting in an easy way to provide generative models of the games. ALE therefore allows the testing of planning as well as reinforcement learning. Several measures of performance are also indicated in the paper as a way to compare algorithms across domains in the ALE.

The Java/C++ source code provided makes it easy to integrate an agent into the framework. There code used to generate the features (BASS, DISCO, etc) written in Java has also been made available online. The environment also provides the RAM state of the game, which is a Markovian state. However, using the screen frames alone can be non-Markovian depending on the game. As pointed out in Naddaf (2010) the agent cannot tell if a laser beam is moving towards it or away from it (fired by it) without looking at the previous frame.

3.3.5.2 Partially Observable PACMAN (POCMAN)

Pocman is a modified ATARI game domain first proposed by Veness et al. (2011) in order to evaluate MC-AIXI-CTW. It consists of an abstraction of the PACMAN domain to an ASCII

format. The agent starts in the center of a standard Pacman map (17x17), see Figure 5.7a. At every time step it receives a bit sequence containing the following bits. 4 bits to code whether there is a wall in an adjacent square, 4 bits to code whether there is food in an adjacent square, 4 bits to check if there is a ghost in any **direction**, 3 bits to “smell” food within 2, 3 and 4 squares and 1 bit that is active when the agent has swallowed a power up pill. It receives a -1 reward every time it makes a valid move. If it attempts to move into a wall it receives -10. Eating a food pellet gains 10 and eating all the food on the map gains 100. Eating a ghost resets the ghost to the center of the map. The domain can either be treated episodically or non-episodically.

Existing work. Pocman has been used in the general RL community to evaluate various algorithms, including CTMRL (Nguyen et al., 2012) in the FRL line of work and CPSR (Hamilton et al., 2014) in the PSR line of work (they also consider a modified version, S-Pocman to add long-term dependencies to the environment). Silver and Veness (2010) develop a POMDP version of the simulation-based planning algorithm UCT called PO-UCT that performs extremely well on the domain, but requires access to a POCMAN emulator.

Popularity. Outside of the general RL community POCMAN is relatively unknown. However, the game it is based on (MS-PACMAN) is very famous and has been implemented on many platforms including, most recently, a playable Google doodle.

3.3.6 Hearts

Hearts is a card-game for 4 players played on a standard deck of cards. 13 cards are initially dealt to each player. The game progresses in hands. Before each hand, each player passes 3 chosen cards to another player of his choice. The objective in the passing of cards is to reduce the number of cards in a particular suit so if that suit is dealt, the player can now rid himself of cards in another suit. A player wins a trick if he plays the highest-value card of the suit that was led. The game continues until all cards are exhausted, and winning a trick confers “penalty points”. The winner is the one at the end of the game who has the least penalty points. The game requires some non-trivial strategy, and inference of the opponents hands.

Existing work. A nice summary of existing research into playing Hearts is presented by Sturtevant and White (2007). It appears that Hearts has been studied in RL as both a partially observable (Fujita et al., 2003) and fully observable (all players cards are known) environment. It has been studied from both multi-agent and single-agent perspectives Fujita et al. (2003). In a single-agent setting the other agents are fixed, and the environment is then a deterministic POMDP which Fujita et al. (2003) solves using an approximation of the belief state using function approximation. In a multi-agent setting, one can make the rough assumption that the environment is a deterministic POMDP if the agents all learn quickly enough (i.e. quicker than the change in the stationary distribution).

Popularity. It is difficult to estimate the popularity of Hearts. MSN online games offer Hearts, but only a few hundred people appear online on that server at any given point. Yahoo Games

recently (December, 2014) shutdown their “Classic games” section including Hearts. Perhaps a better estimate of popularity might be the number of downloads for free/paid Hearts apps on Android. For example, Hearts by KARMAN Games which is a free app launched for Android on 1st December 2014 currently has 500,000 - 1,000,000 installs, along with 18,000+ reviews and ratings. It offers both multiplayer and computer AI options.

3.3.7 Reinforcement Learning in Finance

While finance itself is not a game in the traditional sense, the study of economics has resulted in some of the largest contributions to game theory. The domain of finance is too broad to characterise by a simple description. However, some simple reinforcement learning methods have been successfully applied to a few subproblems within this domain as seen below. We also conjecture that successful applications of RL to the finance domain are perhaps not publicly visible for obvious reasons.

Existing Work. In Moody and Saffell (2001), the authors posit that value functions are not useful for various investment decision making problems and rather these problems should be framed as immediate reward problems ($\gamma = 0$). They solve this problem as a stochastic control problem with an algorithm called recurrent reinforcement learning, which executes a policy search via gradient descent rather than computing a value function.

In Nevmyvaka et al. (2006), reinforcement learning is used in the domain of quantitative finance, specifically in the problem of optimized trade execution (OTE). The goal in an OTE problem is to buy/sell shares of a stock in order to maximise the revenue received or minimise capital spent within a fixed time period. Available to the agent is a large database of historical trading data, as well as outstanding buy and sell limit order prices and volumes (from the NASDAQ). The paper describes an “extensive empirical application of RL” to OTE using large-scale NASDAQ market microstructure data sets. They use a combination of Q-learning with some specific domain knowledge to come up with a *fast* customized algorithm for this purpose. They essentially treat the (partially observable) domain as fully observable. The application of general RL to high-frequency trading is severely limited by speed of execution.

Implementation. The NASDAQ trading history data is readily available from the NASDAQ website. The data sets used in Nevmyvaka et al. (2006) were several gigabytes large, and running sophisticated algorithms on such large datasets could prove unfeasible for general RL algorithms. However, traditional model-free approaches with specialised features might have traction here.

3.3.8 Keepaway Problem

The Keepaway problem first described by Stone and Sutton (2001) is a sub-problem in the RoboCup Soccer competition. There are two teams, the keepers and the takers. The keepers are trying to keep the ball within a specified region, away from the takers who are trying to steal

possession. If the ball is stolen by the takers or leaves the specified region, the episode ends. The domain can be made larger or smaller by changing the number of players on each team and the size of the region. The game is simulated within the RoboCup simulator. The agent receives visual perceptions (relative distance, angle) of objects visible to it (such as the ball and other agents) once every 150ms. The agents can choose from (continuous, parameterised) actions kick, turn or dash once every 100ms. In Keepaway, agents learn independently and simultaneously.

Existing work. Stone and Sutton (2001) and Stone et al. (2005) present work using SARSA(λ) with a linear function approximator using a tile coding of the state variables as features. The resulting agent performs much better than hand-tuned policies for the same task.

Keepaway can be framed as a partially observable continuous MDP problem, by limiting the parameters `visible_distance` and `view_angle` (e.g. to be <360) as in Devlin et al. (2009). The authors replace the manually tuned confidence estimate of the belief state of the position of the other players by learned tile-coded features.

Keepaway also has been popular in the transfer learning community. Taylor and Stone (2005) transfers value-functions between closely related tasks. Taylor and Stone (2007); Taylor et al. (2007) uses policies learned in one domain (e.g. Gridworld) to help the agent perform better in another (Keepaway).

Implementation. A concerted effort to make Keepaway accessible as a testbed to RL researchers was made by Stone et al. (2006). Source code for some basic player agents has been made available⁹, and users simply have to fill in predefined functions defining the learning agent in order to complete the code. The domain is also compatible with RL-Glue, a popular reinforcement learning framework.

Popularity. The RoboCup tournament has grown from 38 teams from 11 countries in 1997 to 410 teams from 45 countries in 2013¹⁰. Over 40,000 people visited the tournament and the event was live streamed with 60,000 individual viewers. Arguably, progress on the Keepaway domain might result in progress in the full RoboCup domain, and thus the popularity of the RoboCup tournament is relevant to Keepaway.

3.3.9 General Game Playing Competition

This is not a single game, but rather a set of games defined by a Game Definition Language (GDL, Love et al. (2008)). GDL allows the specification of a game in terms of entities, actions, propositions relating them and other players. The agent is provided the description of the game at run time and then has to solve it. The syntax of GDL is quite close to PDDL. Given this description, the GGP is an exercise in logic programming, rather than reinforcement learning. Games that can be defined include Tic-tac-toe, Checkers, connect four, 8-puzzle, etc. Given this description it seems that the scope for general RL agents is limited.

⁹Source code for Keepaway agents at <http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/>.

¹⁰<http://www.robocup2013.org/final-report-robocup2013-available/>

Table 3.1: Properties of domains

Environment	Stochastic	State	Action	Observable	Feasibility	Relevance	Implementability	Static
Starcraft:BW	Y	Large	(Large cts) ^a	N	Low	Medium	Medium	N
Poker	Y	Medium	Medium (cts) ^b	N	Medium	High	Easy	Y
Go	N	Large	Large	Y	Low	High	Easy	Y
Chess	N	Medium	Medium	Y	Low	Medium	High	Y
ALE (Chapter 6)	(Y/N) ^c	Large	Large	(Y/N) ^d	Medium	High	Easy	Y
Finance	Y	Large cts	Large	N	Low	High	Medium ^e	N ^f
Keepway	Y	Medium cts	Medium cts	(Y/N) ^g	Medium	Medium	Easy	N
Hearts	Y ^h	Small	Medium	N	Medium	Low	Medium ⁱ	Y
Pocman (Chapter 5)	Y	Medium	Small	N	High	Medium	Easy	Y
Tiger (Chapter 4 Chapter 5)	Y	Small	Small	N	High	Low	Easy	Y
Cheese Maze (Chapter 4 Chapter 5)	Y	Small	Small	N	High	Low	Easy	Y
TMaze (Chapter 4 Chapter 5)	Y	Small	Small	N	High	Low	Easy	Y
Locked Door (Chapter 4 Chapter 5)	Y	Small	Small	N	High	Low	Easy	Y

^aThe action space has many parameterised actions with parameters such as position, unit, etc.

^bThe betting amounts can be arbitrary real numbers, but this is often restricted. No-limit poker further restricts this.

^cThere is an option to turn on stochasticity in the starting state of the domain.

^dSome games are fully observable, others are not.

^eThis is dependent on which particular financial problem is being attacked.

^fAlthough in a historical data setting it is static.

^gCan be made partially observable, see text.

^hIn the sense that other players policies may be stochastic and unknown to you.

ⁱWould be easy if there were hearts playing bot-servers available.

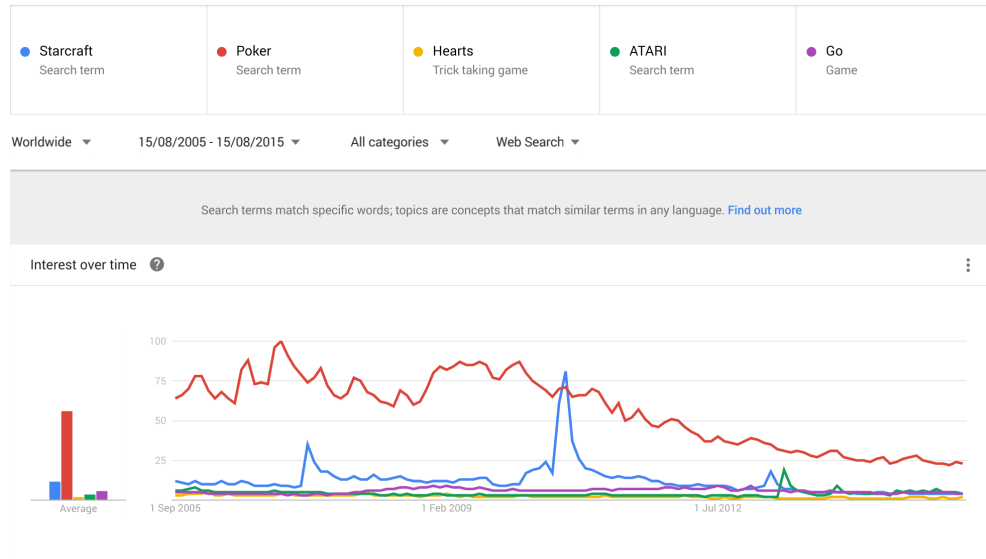


Figure 3.1: The popularity of some of the domains as Google search terms. Data source: Google Trends (www.google.com/trends). Note that for Go and Hearts we used the “Topics” feature to avoid confusion with the common nouns “go” and “hearts”.

3.3.10 Small problems

While testing on large domains is useful for demonstrating the general ability of agents, there are also certain common small benchmark domains used throughout the RL literature which focus on an agent’s ability to deal with some particular aspect of non-Markovian environments, for example long-term dependencies, or state aliasing.

3.3.10.1 Tiger

The Tiger domain (Kaelbling et al., 1998) is familiar in the partially observable reinforcement learning literature. There are two doors A and B . Behind one door is a tiger, and behind the other is a pot of gold. The state space of the POMDP is $\{tigerleft, tigerright\}$ and the agent starts in one of these. There are no transitions between the states. The set of actions $\mathcal{A} = \{openleft, openright, listen\}$, i.e. the agent can choose one of the doors or choose to listen for the Tiger’s growl. The corresponding set of observations is $\mathcal{O} = \{left, right, open\}$, where the agent receives either *left* or *right* if it chooses to listen. The correct observation is given with probability 0.85. The observation *open* is given when the agent opens a door. The episode ends after an agent opens the door. The agent receives -1 for listening, -100 for choosing the door with the tiger and +10 for the gold.

3.3.10.2 Cheese Maze

The cheese maze domain (McCallum, 1996) examines the issue of state aliasing. The agent starts in any of the available positions in the maze. The observation it receives is given by

the number on the square (see Figure 1.1a). The agent's task is to find the cheese. Each valid move it makes costs -1, hitting a wall is penalised -10, and eating the cheese is rewarded 10. Once the agent eats the cheese the episode restarts.

The cheese maze domain is a useful toy domain for testing an agent's ability to deal with perceptual aliasing.

3.3.10.3 TMaze

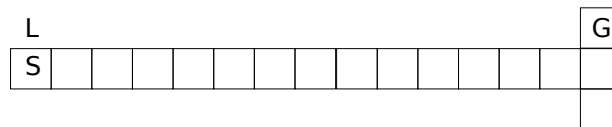


Figure 3.2: TMaze environment showing goal at left

The TMaze problem is a classic non-Markovian problem in RL. It nicely demonstrates the need for long-term memory as well as the exploration vs exploitation problem. We use the formulation as described by Bakker (2002). The environment is a T-shaped maze (see Figure 3.2) with the length of the neck of the T (the corridor) being adjustable. The observation space is $\mathcal{O} = \{0, 1, 2, 3\}$, the rewards are $\mathcal{R} = \{-0.1, 4\}$ and there are four actions denoted by up, right, left, down. The agent needs to remember the observation it receives at the start of the maze, which tells it whether to turn left or right at the end.

The agent receives an observation (either 1 or 2) at the start of the maze that it must remember until it reaches the decision node (observation 3), at which point it must turn left (1) or right (2) according to the initial observation in order to receive a reward of 4. If it chooses any other action it gets reset into the decision state and gets another observation of 0 and a reward of -0.1.

3.3.10.4 Locked Door

The locked door domain is a larger example of a non-Markovian domain that requires an agent to have long term memory to perform well. The agent is in a room (represented by a grid). The room has a locked door and in order to leave, the agent must collect a key from a particular location. We use a 7x7 grid with the door in the top-left corner, the key in the top-right corner and the agent starting in the location one square below the door. The agent has actions up, down, left and right and receives observations that are a binary coding of the adjacent walls. This means that states with the same wall configuration have the same observation. Bumping into a wall, collecting the key, and visiting the door have their own unique observations. The agent gets a reward -5 for bumping into a wall, +10 for visiting the door after obtaining the key and -1 for every other timestep.

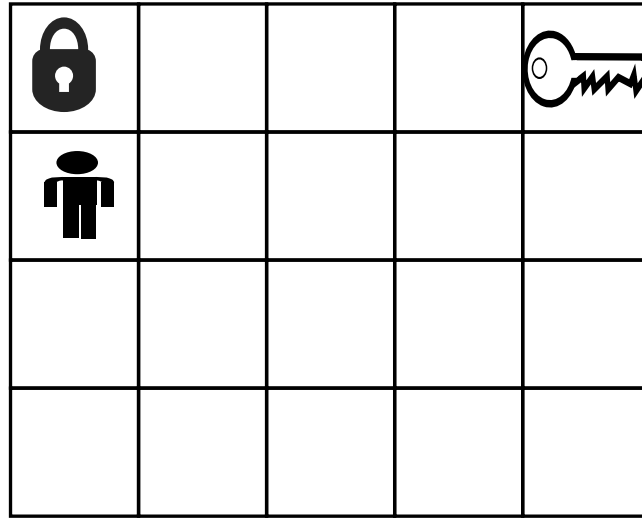


Figure 3.3: Locked door domain. The agent must collect the key for the door before it can open the door.

3.4 Conclusion

In Chapter 4 and Chapter 5 we conduct experiments on the small domains to demonstrate some basic ability in dealing with non-Markovian environments.

Ideally we want to test on large, dynamic, partially observable domains with high relevance. For our model-free algorithm (Chapter 5), as is common in RL, we limit our scope to discrete and static domains. For feasibility reasons, we also restrict to domains with a low to medium size state space, a small action space and easy implementability. This rules out the fully observable games (Chess and Go), dynamic domains (Starcraft), domains with large state spaces (ALE) and continuous domains (Finance, Keepaway). The remaining domains are Poker, Hearts and Pocman. Of these, Poker and Pocman are in close contention for the remaining attributes, and we decide to go for feasibility over relevance. Other related general reinforcement learning algorithms (Veness et al., 2011; Nguyen et al., 2011) have also been tested on Pocman, making it more appealing to us than Poker.

For Chapter 6, one of our motivations is to extract explicit policies from algorithms such as UCT on large domains, where it is infeasible to use UCT in real time. Thus, our list of potential domains now includes the ALE which becomes a clear winner. We considered testing FRL techniques on the ALE as well, but preliminary experiments indicated that such testing would be computationally infeasible with current methods and available hardware.

Looping Suffix Trees

“ It is possible to live with almost no memories, even to live happily as the animal shows; but without forgetting it is quite impossible to live at all. ”
Friedrich Nietzsche, On the Advantage and Disadvantage of History for Life, 1874.

In this chapter we introduce Looping Suffix trees, first seen in Holmes et al. (2006). We use looping suffix trees to address a fundamental problem that suffix trees have, in efficiently representing environments that have long-term dependencies. We extend the CT Φ MDP algorithm to the space of looping suffix trees which have previously only been used in solving deterministic POMDPs. The resulting algorithm performs as well as CT Φ MDP in environments with short term dependencies, and outperforms Long Short Term Memory (LSTM) based methods on TMaze, a deep memory environment. This chapter is based on Daswani et al. (2012), with Section 4.3 being a new contribution.

4.1 Introduction

Memory refers to the length of the observation history that an agent has to remember in order to perform optimally in the current environment. For example, a memoryless method finds a policy $\pi : \mathcal{O} \rightarrow \mathcal{A}$ i.e. pick an action according to the latest observation. Even finding the optimal policy in this simple case is NP-hard in deterministic POMDPs (Littman, 1994a). Importantly, an agent may not have to remember the entire history up to the provided length, but only some particular observations in that part of the observation history.

We have to make the distinction between short and long-term memory. We can make the distinction both in environments and algorithms. For example, Tiger is a short-term memory problem whereas TMaze is potentially a long-term memory problem given the length of the corridor. Of course there is no fixed point at which a short-term memory problem becomes a long-term one. It is, however, somewhat easier to distinguish between short and long-term capable algorithms. For example, suffix-tree based history methods are short-term memory

algorithms relying on local splits in suffix trees to form efficient representations. Most of the methods described in the history-based learning section are in this category of learning short-term dependencies. Suffix tree based methods remember every point in the history up to a certain length (hence are really representations of history lists (Timmer and Riedmiller, 2007)). However, when certain important events far back in a long observation history need to be remembered, these methods are inefficient. This is where representations such as looping suffix trees by Holmes et al. (2006) are useful.

Looping STs were first proposed as a class of finite state machine capable of exactly representing deterministic POMDPs. We show that they can actually represent some stochastic hPOMDPs as well. We also introduce their use in solving the long-term memory problem for some classes of environments like TMaze, where the key to performing well lies in selectively remembering an observation or some pattern of observations. By integrating this class of maps with Φ MDP we show that looping suffix trees can be used as stochastic models which can be crucial even in deterministic environments. We show that looping suffix trees in conjunction with the Φ MDP framework can be used to successfully find compact representations of environments that require long-term memory in order to perform optimally.

4.1.1 Related Work

Our looping suffix tree method learns a finite state automaton that is well suited to long-term memory tasks. While tree-based methods such as USM (McCallum, 1995), MC-AIXI-CTW (Veness et al., 2011), Active LZ (Farias et al., 2010), CT Φ MDP (Nguyen et al., 2011) and many others can in principle handle long-term memory tasks, they require excessively large trees to represent such environments. These large trees can result in large state spaces, which then promote the exacerbated exploration-exploitation problem. More related to our work, Mahmud (2010) aims at searching the very large space of probabilistic deterministic finite automata (with some restrictions). In a similar vein, but restricted to deterministic observations (Haghighi et al., 2007) also construct finite automata that aim at being the minimal predicting machine.

A popular alternative to finite state automaton learning is a class of algorithms based on recurrent neural networks (RNNs) particularly those based on the Long Short-Term Memory (LSTM) framework by Hochreiter and Schmidhuber (1997). The LSTM framework was first proposed to predict time-series data with long-term dependencies. This was introduced to the RL context by Bakker (2002) and more recently a new model-free variant based on policy gradients by Wierstra et al. (2007). These methods are more often used in the continuous case, but were also tested in the discrete setting. Recently, Echo State Networks (Szita et al., 2006) which are also RNN-based have also been tested on long-term memory tasks.

4.2 Preliminaries

Definition 7. A looping tree is a tree which may have loops from any leaf node to an ancestor.

Definition 8. A looping suffix tree based on a d -ary alphabet $\mathcal{O} = \{o^1, o^2, o^3, \dots, o^d\}$ is a d -ary looping tree in which edges coming from each internal node are labelled by the elements of $\mathcal{O}$. The loops in the tree are unlabelled. The non-looping leaf nodes in the looping ST form the state set along with an additional state λ known as the empty state.

In order to map a history sequence to a state in a looping ST we simply follow the edges in the tree until we get to a state as per Algorithm 5. If we reach the beginning of the history sequence without reaching a state, we map the sequence to the empty state.

Looping suffix trees have the effect of giving Kleene-star like representational ability to the standard suffix set. For example, Figure 4.4 shows a looping suffix tree which has the suffix set $\{0, 00(10)^*1, 1(10)^*1\}$. Let $h = [0, 0, 1, 1, 0, 1]$. We can map this history sequence to the state sequence $stateSeq = [s^0, s^0, s^1, s^2, s^0, s^2]$. The last state is mapped by following 1,0,1 down the tree, then following the loop back up the tree to finally take another 1 to end in s^2 .

4.3 Relation to hPOMDPs

A *resolving history* is one that completely determines the internal POMDP state following that history. Holmes et al. (2006) show that there exists a looping suffix tree can perfectly predict the observations of any (strongly connected) deterministic POMDP without rewards (abbreviated as POMDP\R) given a resolving history sequence to start with. Their proof uses the following steps.

- The determinism of the POMDP\R implies that given a resolving sequence, every history following that sequence is also resolving (see Lemma 1).
- Given a set of *minimal resolving sequences* for each state, one can construct an infinite-depth suffix tree that represents this set which maps any history to an internal POMDP state.
- The infinite history suffix tree can be made finite by looping over certain sequences (*excisable sequences* which we will subsequently define).

In this section we show that it is not the determinism of the POMDP\R that enables the first step, but the fact that deterministic POMDP\R also satisfy the history condition Lemma 1. Thus hPOMDP\R can also be predicted using looping suffix trees, where hPOMDP\R is the natural definition of an hPOMDP without rewards. We show that there exists a looping suffix tree such that emission probabilities at the leaf nodes correspond to the appropriate emission probabilities of the hPOMDP\R. We will need the following notation, lemmas and definitions from Holmes et al. (2006).

- We assume here that the history starts with an action and ends with an observation i.e. $h_t = a_1 o_2 \dots a_{t-1} o_t$.

- $S^\lambda = \mathcal{S} \cup \{\lambda\}$ where $\mathcal{S}$ is the state set of the hPOMDP\(\mathbb{R}\) and λ is the empty state.
- We abuse notation and define $h : S^\lambda \rightarrow S^\lambda$ to also be a function mapping each state s_i to the state reached starting from s_i and following history h . If the history sequence h cannot occur from a particular starting state s_i then we set $h(s_i) = \lambda$. In the text, we will say “the function h ” wherever it is not clear whether we are referring to the history sequence or the related function.
- $trans(h) = \{ao : a \in \mathcal{A}, o \in \mathcal{O}, \text{ and } ao \text{ is a possible transition following } h\}$

For this section, we will assume that the emission probabilities of the hPOMDP\(\mathbb{R}\) are dependent on the tuple (s, a) rather than on the state arrived at s' , see Section 2.3 for a discussion of this. The two definitions are equivalent although a POMDP\(\mathbb{R}\) with emissions depending on s' alone may have more states as seen in Figure 4.3. This change in definition is to be consistent with Holmes et al. (2006), and is easier for the purposes of constructing an associated looping suffix tree. The edges of the hPOMDP\(\mathbb{R}\) are now labelled by ao and by Lemma 1 on page 30 we know that each edge ao uniquely determines the next state s' . Thus the resulting hPOMDP\(\mathbb{R}\) can be seen as a finite state machine with transitions given by the ao pairs. $trans(h)$ also defines exactly the states s' that can follow the history h if it is in some state s . The determinism of the transitions ao also means that $h(S^\lambda)$ is a well-defined function.

Definition 9. A history sequence h *resolves* to a state s_i iff the function h maps every state in S^λ to either s_i or λ with at least one state mapping to s_i .

Figure 4.1 shows a deterministic POMDP\(\mathbb{R}\) without a resolving sequence. hPOMDPs always have resolving sequences via the map ϕ that resolves any h to an internal state, including the initial empty history ϵ (although we can slightly weaken this condition as discussed in Section 2.3.3). However, the history condition (Lemma 1) alone is not enough for the existence of a resolving sequence; the same example (Figure 4.1) satisfies this condition but has no resolving sequences.

Definition 10. A state s_i is reachable from state s_j if there exists a finite sequence of actions $a_1 \dots a_n$ such the probability of seeing state s_i after taking the sequence $a_1 \dots a_n$ is non-zero.

Definition 11. An hPOMDP\(\mathbb{R}\) is strongly-connected if every state is reachable from every other state.

The following lemma for hPOMDP\(\mathbb{R}\)s illustrates how both the strongly-connected nature and the existence of a map ϕ for the hPOMDP\(\mathbb{R}\) are both sufficient for the existence of resolving sequences for every state, given a single resolving sequence. Figure 4.2a provides an example of a strongly-connected stochastic hPOMDP, demonstrating that the class is not empty.

Lemma 12. We can construct infinitely many resolving sequences for every state of a strongly-connected hPOMDP.

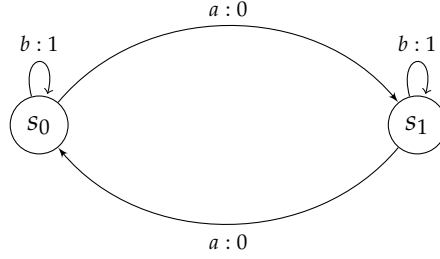


Figure 4.1: This deterministic POMDP $\mathbb{R}$ has no resolving sequences. It still satisfies Lemma 1; given s_0 taking action b determines the next state being s_0 while taking action a determines the next state to be s_1 . However, given a history (e.g. $b1b1a0$) it is impossible to say whether we are in s_0 or s_1 . If we equipped this POMDP $\mathbb{R}$ with a map ϕ such that $\phi(\epsilon) = s_0$ then the history $b1b1a0$ resolves to s_1 .

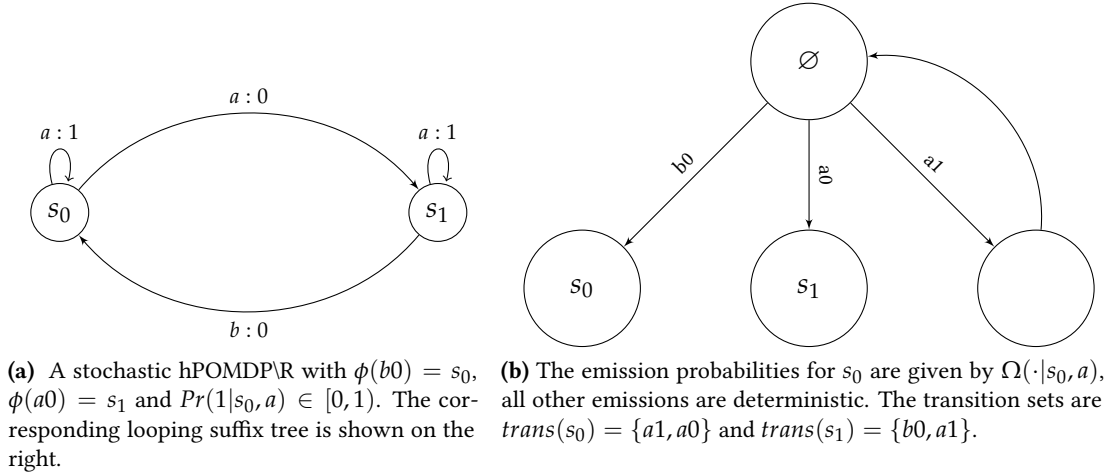
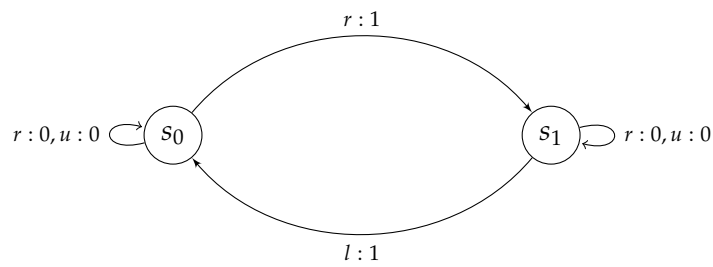
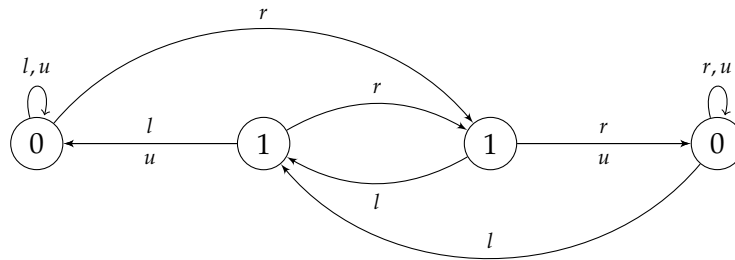


Figure 4.2: Stochastic hPOMDP $\mathbb{R}$ and corresponding LST

Proof. Let us assume we have a history sequence that resolves to some initial state s_i in an hPOMDP $\mathbb{R}$ via the map ϕ . For instance, this could be the empty history ϵ such that $\phi(\epsilon) = s_i$. Let a_i be some action feasible from state s_i that results in observation o_i and leads (deterministically) to s_j . Then, by definition $h(s) = s_i$ or λ for all states s . Therefore, $hao(s) = ao(h(s)) = s_j$ or λ i.e. hao resolves to s_j . Since the hPOMDP $\mathbb{R}$ is strongly-connected, we can construct a resolving sequence for every state by repeating this construction using an appropriate sequence of actions (and resulting observations) that make each state reachable from s_i . Note that this process does not rely on the (potentially low) probability of the sequences, simply their possibility, since we can read the possible transitions directly from the hPOMDP $\mathbb{R}$ specification. By the strongly connected nature of the hPOMDP $\mathbb{R}$, for every $s \in S$ there exists a suffix q such that hq resolves to s . This includes the state s_i itself, which allows us to construct an infinite number of resolving sequences for every state. $\square$



(a) The original flip automaton as described in Holmes et al. (2006). Here we use the convention that the emission probabilities depend on state and action pairs.



(b) An equivalent flip automaton with emission probabilities restricted to depend only on the (next) state. The state labels are now the observations that will be emitted if the agent transitions to that state.

Figure 4.3: Flip automata for different conventions of writing POMDPs

In Lemma 12 we saw that we can have arbitrarily many resolving sequences of arbitrary length. However, we are particularly interested in the smallest possible resolving sequences. A suffix tree built from these resolving sequences (called a history suffix tree) would then allow us to map any history to the corresponding internal state.

Definition 13. The set of *minimal resolving sequences* for a state s_i is the set of resolving sequences h such that no shorter sequence h' formed by removing prefixes of h is also resolving.

Definition 14. For any two histories h and q such that $h = eq$, the sequence e is *excisable* from h iff $\forall p \text{ trans}(ph) = \text{trans}(pq)$. Otherwise e is non-excisable.

Definition 15. Two histories h and q are *functionally equivalent* iff the functions $h : \mathcal{S}^\lambda \rightarrow \mathcal{S}^\lambda$ and $q : \mathcal{S}^\lambda \rightarrow \mathcal{S}^\lambda$ are equal.

A minimal resolving sequence may be unbounded in length due to the presence of excisable sequences within the resolving sequence. Thus, the history suffix tree might be infinite.

Lemma 16 (from (Holmes et al., 2006)). *For two histories h and q such that $h = eq$, if h and q are functionally equivalent then e is excisable from h .*

The proof for the above lemma can be found in Holmes et al. (2006). Intuitively, it follows from the definition of the function h and excisability. Note that excisability is precisely the property that loops in looping suffix trees cover. If there are resolving histories h and q such that $h = eq$, then we can simply treat h as if it were q since they resolve to the same internal hPOMDP state and they behave the same way for every possibly prefix. Effectively we create a loop from the node corresponding to h in the history suffix tree to the node corresponding to q .

Lemma 17 (from (Holmes et al., 2006)). *Every branch of a history suffix tree either becomes resolving or reaches a level that begins with an excisable sequence after finite depth.*

Proof. Consider a branch which does not become resolving at a finite depth represented by the (infinite) history h . Let h_i be some prefix of h . There can only be a finite number of such prefixes that are functionally distinct since each function h_i has finite domain and range. Thus for some j with $i < j < \infty$, h_j is functionally equivalent to h_i . Additionally, $h_j = eh_i$ for some e . By the previous lemma, e is excisable from h_j . $\square$

Effectively, this lemma allows us to make a finite looping suffix tree out of a potentially infinite history suffix tree by looping over the excisable sequences. We use this lemma for the following theorem.

Theorem 18. *Let M be a strongly-connected hPOMDP\mathcal{R}. Given a resolving history to begin with, there exists a prediction looping suffix tree L^M such that each (non-looping) leaf node of L^M corresponds to a state s_i in M and the emission probabilities at the node mapping to s_i correspond to $\Omega(o|s_i, a)$ for each o and a .*

Proof. The following proof is similar to the proof of Theorem 1 in Holmes et al. (2006) with the addition of the appropriate emission probabilities at the leaf nodes.

By Lemma 17, the history suffix tree either resolves at a finite depth or reaches the start of an excisable sequence. If the branch h resolves at a finite depth, we are done. If the branch h does not resolve at some finite depth, then let k be the depth at which it first begins with an excisable sequence e such that $eh_j = h_k$ for $j < k$. We can then place a loop from h_k to h_j and any history that follows h_k will be looped back to h_j , since h_j and h_k are functionally equivalent.

The emission probabilities for the observations at each leaf node are simply the emission probabilities for the corresponding hPOMDP\R state-action pairs, $\Omega(o|s, a)$ where s corresponds to the hPOMDP\R state mapped to by the corresponding branch of the looping suffix tree i.e. if the leaf node corresponds to the minimum resolving sequence mapping to s then the emission probabilities assigned to that node must be $\Omega(o|s, a)$ for each o and a . Thus we define a looping suffix tree that has the same emission probabilities as the hPOMDP\R M for each state and action. $\square$

An hPOMDP\R has the property that there exists a function $\phi : \mathcal{H} \rightarrow \mathcal{S}$ such that $\phi(h) = s \in \mathcal{S}$. The looping suffix tree constructed in the above proof is effectively a representation of this function ϕ . It should be noted that the above is a proof of existence, and not a completely constructive one, unlike the proof of Holmes et al. (2006), since the emission probabilities are simply copied from the original generating process rather than learned. It is possible that by observing the frequency of observations-action tuples at each leaf node of the constructed LST which is consistent with the history so far one can make an empirical estimate of the emission probabilities that converges asymptotically to the true emission probabilities. However, we are more interested in the use of LSTs in predicting reward distributions, which we pursue empirically in the following section.

4.4 Looping Suffix Trees in Φ MDP

The benefit of using looping STs comes from the ability to keep relevant past events in memory by 'forgetting' or looping over irrelevant details. Holmes et al. (2006) restrict their discussion to the deterministic case without rewards. The above discussion of the relation to hPOMDP/Rs extends their results to some subclass of stochastic POMDP/Rs for which there exists a resolving sequence, but is not a constructive proof. Unfortunately the loopability criterion of Holmes et al. (2006) cannot work in the general stochastic case since a loop can change not only the possible transitions but also the transition probabilities. More importantly, they use looping suffix trees to completely represent POMDPs, but this is not necessary. As long as we have an LST that can predict rewards well, there is no need to completely recover the original POMDP, in fact it is beneficial to have a more compact map. The utility of the reward prediction ability of LSTs is particularly demonstrated in the TMaze (Figure 3.2) example. Here, a very simple

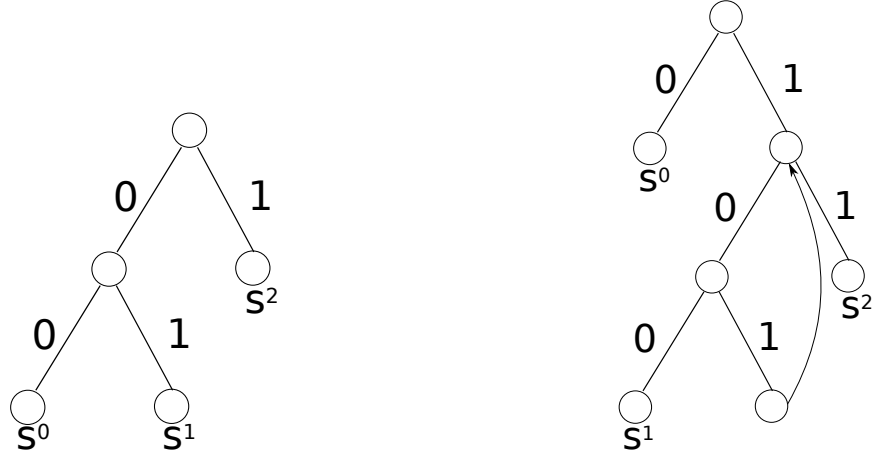


Figure 4.4: Example (observation-based) suffix and looping suffix trees

LST can predict perfectly the rewards of a TMaze of any length, whereas the LST exactly predicting every observation of the TMaze is as long as the corridor in TMaze.

In this section we present an extension of the generic Φ MDP method to looping suffix trees that can learn LSTs useful for performing well in reinforcement learning domains, where there is a need to “forget” or excise certain sequences of observations. The cost function of the Φ MDP framework ($OCost_\alpha$) immediately gives us a well-motivated criterion for evaluating looping suffix trees. Using looping suffix trees as the map class in this framework allows us to extend them to stochastic environments. Experimental results show that Φ MDP works well in the space of looping suffix trees. The extension to stochastic tree sources is also useful in deterministic environments, where in some cases a smaller stochastic tree source can sufficiently capture a deterministic environment.

For the rest of this chapter we do not use action-observation looping suffix trees, rather we restrict our trees to observations alone. This is primarily for practical reasons, it decreases the branching factor of the search space, the amount of space required to store the trees, and also the computational time required to map a history sequence to a state sequence. We do sacrifice some representation power when we do this, however the environments that we test on are representable by observation looping suffix trees.

We also note that in this chapter, we do not give the agent any information about when an episode ends in order to be as general as possible.

Algorithm

The algorithm consists of a specification of $CL(\phi)$ and the neighbourhood method which is needed for the simulated annealing algorithm in the generic Φ MDP algorithm (Algorithm 3). We call our algorithm LST Φ MDP. A tree with k nodes can be coded in k bits (Veness et al., 2011, Sec.5) and the starting and ending nodes of all s loops can be coded in $2s \log(k)$, so

we define the model cost of the map $CL(\phi)$ as $CL(\phi) = k + 2s \log k$. The `getNeighbour()` method (Algorithm 6) first selects a state randomly and then with equal probability subject to certain conditions, it selects between one of 4 operations. Note that the simulated annealing procedure that we use is a very simple generic method. However this can be extended to more sophisticated annealing schemes such as parallel tempering as done by Nguyen et al. (2011). We observe that the simple scheme we use works extremely well in practice.

- **merge** : In order to merge a state, all sibling nodes must also be states. From the definition of a suffix set, we know that every state corresponds to a unique suffix. The merge is simply the shortening of a context for those states. If s^i is the state being merged and $s^i = o^j n'$ where $o^j \in \mathcal{O}$ and n' is the remainder of the suffix corresponding to that state, then the siblings of s^i are $o^k n'$ where $k \neq i$. If these siblings are also states then the merge operator removes $o^i n'$ for all i from the suffix set and adds a new state n' .
- **split** : Analogously, we can split any state s^i by adding a depth one context to the state i.e. by constructing $|\mathcal{O}|$ new states of the form $o^j s^i$ for all $o^j \in \mathcal{O}$ and removing the state s^i .
- **addLoop** : The `addLoop` function has two cases. Either we add a loop from an existing state to its parent (thereby removing it from the state set and adding it to the loop set) or we extend an existing loop to the parent of the existing node looped to.
- **removeLoop** : The `removeLoop` function is simply a reverse of the `addLoop` function allowing us to decrease the length of a loop, or if it is a length one loop create a new state from the node.

Definition 19. A history h is said to be *consistent* with respect to a particular looping suffix tree L if L maps every prefix of h to a non-empty state.

Loops introduce a few problems to the standard Φ MDP procedure. A looped tree can be inconsistent with the current history. This can be problematic if, for instance, the optimal tree is inconsistent with the current history. One solution is to always provide a reasonable initial history that the optimal tree should be consistent with. For example in the TMaze case (see Section 4.5), we ensure that the first observation is in fact the start of an episode, which is a reasonable assumption. Then any trees that are inconsistent can be discarded. In fact to make the search quicker, we can mark nodes where loops make the tree inconsistent and no longer add those loops. The initial map is always set to be the depth one tree (i.e. one split). The single state tree can have the lowest cost for very large amount of data, and we explicitly ban it as a neighbour.

For the reinforcement learning part of the algorithm, we use the model-based method as specified by Hutter (2009b) which is based on Szita and Lörincz (2008b). This method adds an additional “garden of eden” state (s_e) to the estimated MDP, which is an absorbing state with a high reward. The agent is told that it has been to s_e once from every other state, however the

agent cannot actually transition to this state. Then we simply perform value iteration on this augmented MDP. Initially the agent will explore in a systematic manner to try and visit s_e , but as it accumulates more transitions from a particular state, the estimated transition probability to s_e decreases, and the agent eventually settles on the optimal policy.

The space of looping STs includes the space of ordinary STs. Therefore, results from the non-looping case (Nguyen et al., 2011) should be reproducible, as long as the simulated annealing procedure is not (very) adversely affected by the enlargement of the search space. Experimental results show that some care must be taken in choosing α for this to be the case. This is further discussed in Section 4.5.

4.5 Experiments

In this section we describe our experimental setup and the domains that we used to evaluate our algorithm. Each domain was used to test a different ability of the algorithm. Every experiment was run 50 times. The agent is given an initial history produced by taking random actions. Each run of an experiment was conducted over some number of epochs with each epoch containing 100 iterations of the agent performing actions according to its current policy, based on the current map with a constant ε -exploration of 0.1 until a point where it stops exploration. After every epoch, the agent was given a chance to change its optimal map via a simulated annealing procedure. The annealing procedure used an exponential cooling function with constants chosen so that the first few maps had an initial acceptance probability in the range $[0.6, 0.7]$. Plots show every 10th point with 2 standard error on either side. The exact constants used for all the experiments can be found in Table 4.1.

TMaze The TMaze domain was described in Section 3.3.10.3. It performs a tests the ability of an agent to deal with long-term dependencies in a very direct manner. Additionally, adjusting the length of the corridor in the TMaze can be used to judge the limits of the algorithm.

Note that for the TMaze domain, having a tree with one split (i.e. one state for each observation) allows the agent to represent the policy, “go straight until at the decision node, then pick action (left or right)”. This tree is what allows the agent to eventually learn the true map, which additionally remembers the first observation received.

We conducted experiments on three variants of TMaze. In the first variant, the observation it receives at the start determines where the goal lies every time. In the second variant, the agent receives two different observations in the corridor with equal probability. This means that the looping ST needs to loop over both observations in any possible order. The third variant adds uncertainty to the accuracy of the starting observation along with the stochasticity in the corridor, it predicts the position of the reward with 0.8 probability. In each variant we can adjust the length of the corridor. Note that the first variant is deterministic within a given episode, however the history itself is not deterministic since the observation received

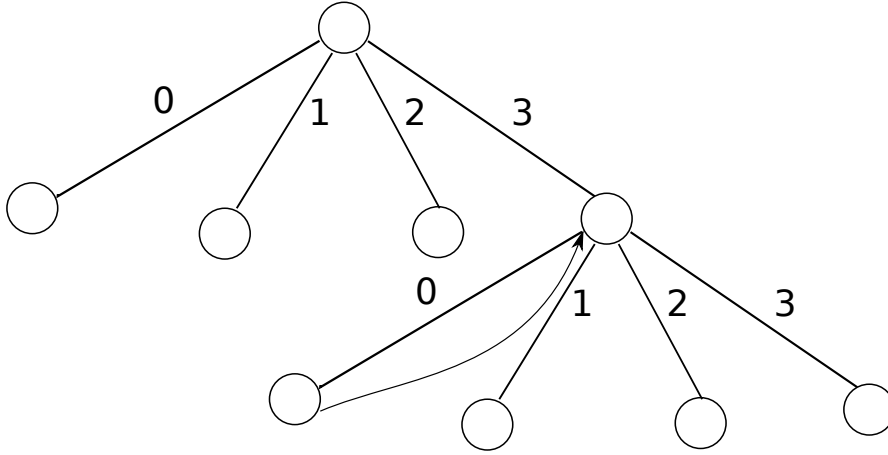


Figure 4.5: A reward optimal LST for the TMaze problem

Experiment	α	epochs	anneal-temp	init-history	stop-explore	max-reward
Det Tmaze	0.1	500	20	100	400	1
Stoch Tmaze	0.1	500	100	100	400	1
Tiger	$1 \cdot 10^{-2}$	500	100	100	400	5
Locked Door	$1 \cdot 10^{-2}$	1,000	100	1,000	900	10

Table 4.1: The table lists the various constants used for each experiment. Common to all experiments were the maximum number of steps for a single annealing run capped at 50, the value of k in the exponential cooling scheme at 0.005, $\varepsilon = 0.1$ and $\gamma = 0.99$. α is a parameter of $OCost_\alpha$ that controls the balance between state and reward code-lengths, *anneal-temp* refers to the temperature T in the cooling schedule, *init-history* is the number of initial random actions performed by the agent, *stop-explore* is the epoch beyond which the agent no longer uses ε -exploration and *max-reward* is the value of the reward given to the garden-of-eden state in the extended MDP for all actions.

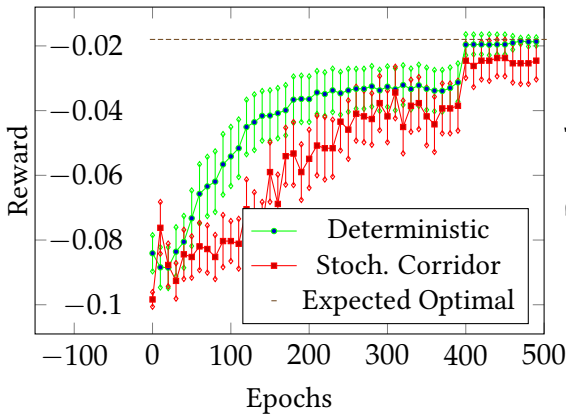


Figure 4.6: $LST\Phi MDP$ on TMaze length 50

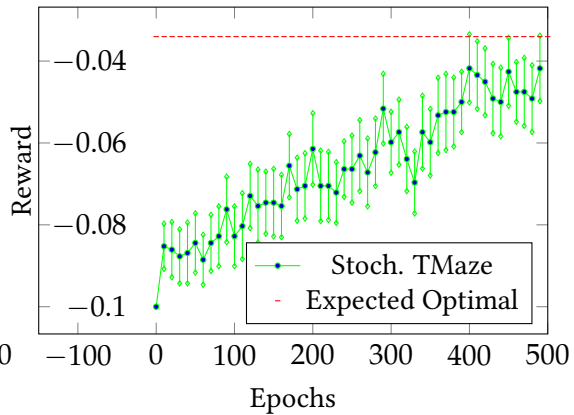


Figure 4.7: Stochastic TMaze length 50

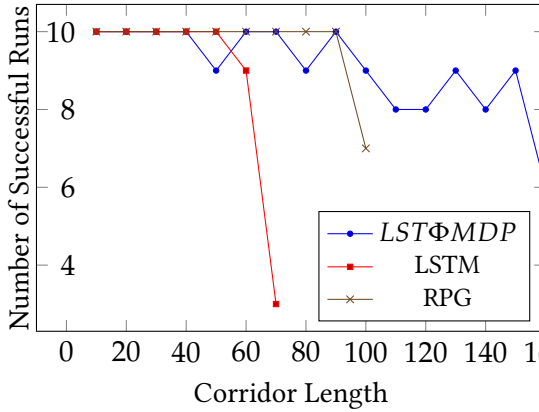


Figure 4.8: #optimal runs, varying corridor lengths for LSTΦ, RPG and RL-LSTM on Det. TMaze

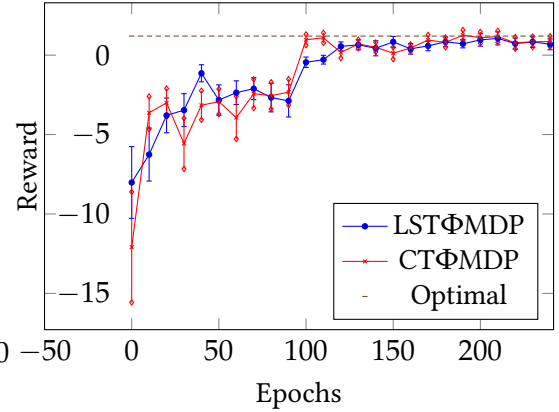


Figure 4.9: Comparison on Tiger with $\text{prob}(\text{listen})=0.85$ for LSTΦMDP and CTΦMDP

at the start of the episode is selected randomly, which is enough to prohibit deterministic approaches.

We compare our LSTΦMDP to RL-LSTM (Bakker, 2002) and Recurrent Policy Gradients (RPG) (Wierstra et al., 2007) on the deterministic TMaze. Note that we use the results from the corresponding papers; we did not implement the methods ourselves. Following the experiments in those papers, we increase the length of the corridor systematically from 10 to 100, in increments of 10. In this case each experiment was run 10 times, and we measured the number of successful runs per length. A run is said to be successful if the agent achieves the optimal policy and hence the optimal reward in at least the last 10 epochs. We used this metric to compare with other methods. All the successful runs had optimal policies from 400 epochs onward i.e. once there was no longer any ε -exploration. We continued to increase the corridor length until the performance of our algorithm was worse than the performance of the RPG method at length 100, which happened at corridor length 160 (6 successful runs).

Locked Door In order to show that our algorithm was useful in solving larger long-term dependency problems we tested on the Locked Door domain which was introduced in Section 3.3.10.4. Due to the larger size of the domain, the agent was given a history of 1000 random actions at the start and every run of the experiment was 1000 epochs long with each epoch being a 100 iterations as usual.

4.6 Analysis

In this section we analyse the results from our experiments, and explain characteristic behaviours and parameter settings. The neighbourhood function was chosen to traverse the state

Algorithm 5: Get current state given an observation sequence and a looping ST

```

getCurrentState(Observation sequence  $o_{1:t}$ );
 $currentNode$  = root;
 $i$  =  $t$ ;
while  $currentNode$  is not a state do
  if  $currentNode$  has a loop then
     $currentNode$  = node at the end of the loop;
  else
     $currentNode$  = the  $o_i$ -th child of  $currentNode$ ;
     $i$  =  $i - 1$ ;
  end
  if  $i \leq 0$  then
    return  $s^{empty}$ ;
  end
end
return  $currentNode$ 

```

Algorithm 6: getNeighbour() method for looping ST

```

Input: num_obs : number of observations, statelist : list of states in current tree,
looplist : list of loops in current tree
 $state$  = random state from current statelist;
Let  $c$  be a random number in  $\{1,2,3,4\}$ ;
if  $c == 1$  and ( $num\_states > num\_obs$ ) and
every sibling of the current state is also a state then
  merge( $state$ );
else if  $c == 2$  and ( $num\_states > 2 \times num\_obs$ ) then
  if  $uniform(0,1) > 0.5$  and looplist  $\neq \{\}$  then
     $state$  = random state from looplist;
  end
  addLoop( $state$ );
else if  $c == 3$  and looplist  $\neq \{\}$  then
   $state$  = random state from looplist;
  removeLoop( $state$ )
else
  split( $state$ );
end

```

space slowly through the looping trees linked to a particular suffix tree, after a few experiments with larger jumps failed. Loops make smaller representations of large environments possible. The difference in cost between two adjacent trees can be quite large, since a loop can suddenly explain a very large amount of (reward) data by ignoring irrelevant sequences.

Deterministic TMaze In the case of corridor length 50, the optimal policy has a value of -0.018. The agent reaches the optimal policy in every run once the ϵ -exploration has been turned off at 400 epochs. See Figure 4.6 for details. The results of the separate experiment comparing the algorithms performance on varying corridor lengths are displayed in Figure 4.8. Up to length 100 the agent reaches the optimal policy, with a few corridor lengths having one run stuck on traversing the corridor without every having seen the goal. Note that the algorithm does not necessarily reach the optimal tree, but finds a reward-optimal tree that contains it. In comparison, RL-LSTM (Bakker, 2002) has increasingly many suboptimal runs as the length of the corridor increases past 50. RPG (Wierstra et al., 2007) has optimal results up to length 90 but has 3 unsuccessful runs at length 100. We continue increasing corridor length until we have more than 3 unsuccessful runs at length 160. Additionally, our algorithm uses 50000 iterations (500 epochs) in all cases, while RPG uses around 2 million iterations for corridor length 100. We also tested CT Φ MDP but it was not successful for corridor lengths > 5 . We would need a depth n suffix tree to represent a TMaze with length n . However, a looping suffix tree with optimal reward prediction is much easier to find, as shown in Figure 4.5 and also much smaller, leading to greater data efficiency. We did not test Echo State Networks, however from (Szita et al., 2006) we note that the method was not successful on corridor lengths greater than 25. In this environment, the optimal looping suffix tree (Figure 4.5) is the same regardless of the length of the corridor, since the tree simply loops over the corridor observations. Of course the exploration-exploitation problem gets harder as the corridor length increases. Despite this the systematic exploration of the agent appears to work well. We also note that in comparison to Recurrent Neural Networks (i.e. the LSTM based methods) it is relatively much simpler to interpret a looping suffix tree.

It should be noted that the optimal LST for the TMaze problem doesn't exactly represent the POMDP that defines the TMaze problem. The POMDP itself must contain all the states for the corridor, however the predictive "reward optimal" LST does not need to. A big difference between the constructive proof technique of Holmes et al. (2006) and the use of Φ MDP is revealed here. The construction of Holmes et al. (2006) would not find this compact predictive LST, since there is in fact no loop within the TMaze POMDP, and the minimal resolving sequences are all finite. However, for the purposes of predicting the reward sequence there is no need to keep track of exactly how many corridor states are seen, it is simply enough to remember the observation at the start of the corridor and continue along till the corridor is over, whenever that may be.

Stochastic TMaze (corridor length 50) The optimal policy in the stochastic corridor TMaze case has a value of -0.018, the same as the deterministic case. However, the agent has to loop over a new observation, and hence needs a larger tree. The task is made hard by the

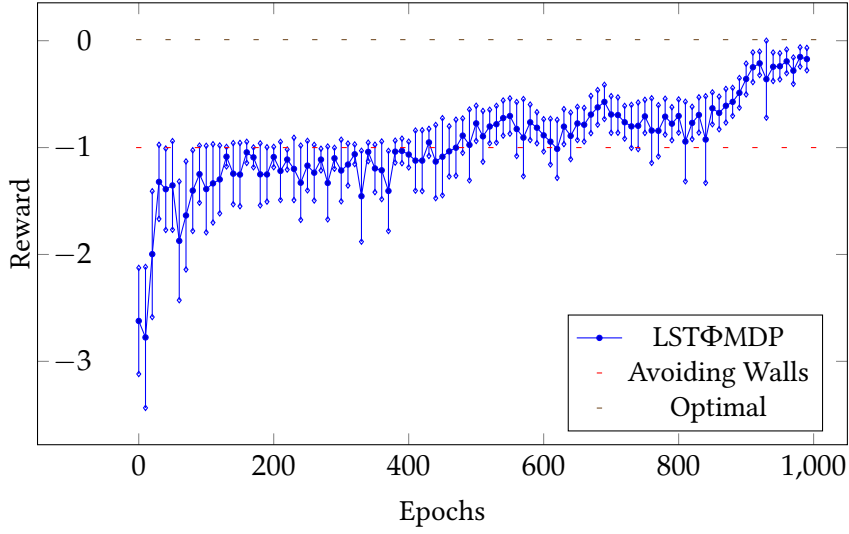


Figure 4.10: Near-optimal runs of LST Φ MDP in the locked door domain.

stochastic nature of the corridor observations. Failures occur mainly due to exploration issues (agent not finding a reward often enough) rather than problems with simulated annealing. This means that the average reward is a little lower than the optimal, however in most cases the agent did reach the optimal. In the fully stochastic case where the accuracy of the initial observation is 0.8, the expected reward is -0.03404. The results have more variability at each point as seen in the higher error bars, but overall the agent performs well in every run with the average of the final point being -0.04178.

Tiger The Tiger example is interesting since it shows that the agent can still reproduce results from the regular non-looping suffix tree case. The agent achieves the optimal reward when the parameter α is set to a lower value of 0.01. Figure 4.9 shows that LST Φ MDP and CT Φ MDP perform nearly identically on this problem.

Locked door When the agent visits the door location there are two contexts, it either has the key or it doesn't. Remembering that it has a key is much easier with loops, since it can simply loop over observations once it has collected the key. The LST Φ MDP agent with $\alpha = 0.01$ succeeds in finding a near-optimal policy in about half the runs. CT Φ MDP succeeds in learning how to avoid walls but never improves further in 1000 epochs. See Figure 4.10 for the graph of only the near-optimal runs of LST Φ MDP.

General Problems The cost function needed mild tuning of the parameter α for the experiments, generally relying on low values (especially in Tiger). This emphasises reward prediction over state prediction. Looping STs can reduce the cost of coding state sequences dramatically by looping over several observations and substantially reducing the number

of states. Obviously this can lead to a bad reward coding, which should eventually cause the tree to be rejected. However, if the agent has not seen enough of the various available rewards then the reward cost may not be particularly high. This can be self-reinforcing. Bad models of the environment can result in policies that only rarely experience critical events, for example opening the door in the Tiger or Locked door problem. This means that the reward cost changes very slowly and may not ever dominate the total cost. Note that this often means that if the agent does not find the reward early on in the run, then it has not much chance of finding it later. Inspecting the failed runs for the deterministic TMaze with very long corridors, we see that the agent never experiences the reward or only sees it once or twice. Particularly, as the length of the maze increases both the optimism and the ϵ -exploration are insufficient to fully explore the maze.

A note on the constants used by the agent The agent relies on a variety of constants as seen in Table 4.1. Of these, the constant α , which controls the contribution of state and reward code-lengths is the most important to the success of the algorithm. Nguyen (2013) shows that for any domain there exists an α such that for any $\beta < \alpha$ minimising $OCost_\beta$ correctly finds the optimal map. Thus it is easy to find an α that works, simply by lowering the value. In our experiments, we always tried to use the highest value that produced a good result, and $\alpha \in \{0.1, 0.01\}$ allowed us to solve all problems.

The annealing temperature was set to 100 for environments with any significant stochasticity, and was lowered for the deterministic T-Maze. The initial history and number of epochs for Locked Door are higher, since it is a larger environment and needed more time to converge.

Computational Complexity The most time consuming part of Φ MDP is the calculation of the cost of a new map. The calculation of $OCost_\alpha$ from the statistics is $O(|S|^2|A|) + O(|S||A||R|)$. However, since the state space changes the statistics must be recomputed. In CT Φ MDP this can be done using a pass over the history (of length n) with backtracking limited to the depth d of the tree, making the worst-case complexity $O(dn)$. However, loops can require backtracking to the start of the history making the worst-case complexity $O(n^2)$. Note that if $n < |S|^2|A|$ there are some transitions that have not been seen and can thus be ignored when calculating $OCost_\alpha$, so the complexity is dominated by the $O(dn)$ or $O(n^2)$ term. In practice, the execution times are competitive to (non-looping) suffix tree based methods on environments that do not require loops. For example on Tiger, the average execution time for LST Φ MDP is 11.49s and for CT Φ MDP it is 11.27s. In environments where loops matter, LST Φ MDP is much slower, for example on TMaze (length 50) an average run for LST Φ MDP is 216.93s while for CT Φ MDP it is 38s. The large speed difference is because CT Φ MDP remains on the (sub-optimal) minimal tree of 4 states, which results in less time spent in the annealing procedure.

Stochastic policies Looping suffix trees give the agent the ability to deal well with stochasticity in the environment, as well as stochasticity in the policy. For example, if the agent

is using an ε -greedy policy in the TMaze setting, the traversal of the corridor might take longer than the actual length of the corridor. A standard suffix tree approach might code these additional random steps into the memory of the agent, however the looping suffix tree representation allows for the elimination of these irrelevant steps using a loop.

4.7 Conclusion

We introduced looping suffix trees to the feature reinforcement learning framework (Hutter, 2009b) to create an algorithm called $\text{LST}\Phi\text{MDP}$. The experimental results show that looping suffix trees are particularly useful in representing long-term dependencies by looping over unnecessary observations. Loops allow for smaller representations leading to greater data efficiency. We outperform LSTM-based algorithms (Bakker, 2002; Wierstra et al., 2007) on TMaze. $\text{LST}\Phi\text{MDP}$ was also able to perform well on stochastic environments, which is a handicap of previous methods using looping suffix trees (Holmes et al., 2006; Haghighi et al., 2007). We also matched the results of $\text{CT}\Phi\text{MDP}$ (Nguyen et al., 2011) on short-term environments.

In the next chapter we examine a new model-free cost function that measures ability of a map ϕ to represent the value function of the optimal policy. This allows us to extend the ΦMDP method using function approximation. We will also reproduce the experiments here using the new cost function and compare the resulting performance.

A Model-free Off-policy Cost for Φ MDP Agents

“ Although this may seem a paradox, all exact science is dominated by the idea of approximation.

”

Bertrand Russell, The Scientific Outlook, 1931.

In this chapter, we formulate a new cost function that is based on the ability of a map ϕ to approximate the optimal action-value function. The cost is the pathwise squared sum of off-policy temporal difference errors with an additional ℓ_0 -regularisation term. The resulting criterion lends itself immediately to a function approximation setting where features are chosen based on the history. This algorithm can also be viewed as an extension of Q-learning to the history-based RL setting, although the use of simulated annealing to find appropriate features does negate the linear-time behaviour of traditional Q-learning. This algorithm is also related to the recent line of work on lasso temporal difference learning which aims at finding a small feature set with which one can perform policy evaluation. The distinction is that we aim directly for off-policy learning of the optimal value function and we use ℓ_0 instead of ℓ_1 regularisation.

We perform an experimental evaluation on classical benchmark domains and find improvement in convergence speed as well as in economy of the state representation against CT Φ MDP. We also compare against MC-AIXI-CTW on the large Pocman domain and achieve competitive performance in average reward. We use less than half the CPU time and 36 times less memory. Overall, our algorithm hQL provides a better combination of computational, memory and data efficiency than existing algorithms in this setting. This chapter is based on Daswani et al. (2013) with the addition of Section 5.3 and additional experiments in Section 5.5.

5.1 Introduction

In the feature RL framework, a cost function must trade-off between the predictive ability of a map and the size of the resulting state space. It is easy to imagine a cost function that does

precisely this; namely a regularised (log)-likelihood of the observation-reward sequence so far. However, this formulation can lead to unnecessarily large state spaces when the environment has a large number of observations. The original formulation of the cost function avoided this problem by predicting, instead, the likelihood of the *state-reward* sequence resulting from the application of a particular map ϕ to the history.

The hypothesis examined in this chapter is that it might be better to look for ϕ that can best represent the value function of the optimal policy accurately. This serves two purposes; firstly, it can lead to more economical state representations, since some states found by likelihood approaches may not be necessary in representing the optimal value function and secondly, a model-free representation of the cost function can lead to scalability and generalisation via function approximation.

We propose that the cost of a particular map ϕ is the sum of the squared temporal difference errors of the optimal Q -value along a trajectory traversed by the agent, along with a regulariser on the size of the state space given by ϕ . In the case that the map is fixed, this procedure simply reduces to Q -learning. This objective is related to that of the Least Squared Fitted Q -iteration algorithm by Ernst et al. (2005); Farahmand et al. (2008).

The problem of reinforcement learning with function approximation in MDPs where the feature space is large compared to the number of samples, has been intensively studied in the last 5 years. This problem has been addressed by the introduction of methods like regularised Least Squares Temporal Difference learning by Kolter and Ng (2009); Johns et al. (2010); Ghavamzadeh et al. (2011) where an ℓ_1 regulariser promotes sparsity. If we do not have many more samples than features, then regularisation is a necessity. The Dantzig selector temporal difference method by Geist et al. (2012) uses an ℓ_∞ norm instead of ℓ_2 for the error and an ℓ_1 norm for parameter complexity.

Our primary difference from the classical function approximation setting is that we start with a small number of features and grow them over time via a simulated annealing procedure, rather than starting with a large set of all possible features and finding a sparse representation. In our history-based setting, if we want to be able to assume that the representation using all features is Markov, like Ghavamzadeh et al. (2011) does, then we need a huge feature class that can capture all information in the history, and this class grows over time.

Our experiments on small domains show equal or better performance than CT Φ MDP (Nguyen et al., 2011) in terms of speed of convergence and, in the function approximation case, in economy of representation. When compared against MC-AIXI-CTW on the large domain Pocman, the performance is comparable but with a significant memory and speed advantage.

5.2 Formulation of the $Cost_{QL}$ function

From Chapter 2 we know that Q -learning aims to find the fixed point of the Bellman equation $Q = T^*Q$, where T^* is the Bellman optimality operator. It does this in an online fashion using the update rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t \Delta_t$$

where Δ_t is the temporal difference

$$\Delta_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

A key difference between Q-learning and SARSA is that Q-learning is off-policy and estimates the value function of the optimal policy, while following some other behaviour policy. It asymptotically converges to the optimal action-value function Q^* given sufficient exploration and a learning rate (α) that satisfies the Robbins-Monro (Robbins and S.Monro, 1951) conditions. It is simple to implement and works well in practice. However, Q-learning in the above form cannot be used in a history-based RL setting, since the notion of state is undefined without a feature map.

For each ϕ , we define a Q-table based on the state space given by ϕ . We denote this Q-table by $Q^\phi : \mathcal{H} \times \mathcal{A} \rightarrow \mathbb{R}$ and it is of the form $Q(\phi(h), a)$. Let $\hat{Q}_\phi^*$ be the (approximate) solution given by Q-learning to the Bellman equation. Let $\Delta_t(\phi) = r_{t+1} + \gamma \max_a \hat{Q}_\phi^*(s_{t+1}, a) - \hat{Q}_\phi^*(s_t, a_t)$. Then we define the $Cost_{QL}(\phi)$ as the sum of the squared TD-errors for $\hat{Q}_\phi^*$ on the samples we have seen so far plus a regularisation on the map ϕ .

$$Cost_{QL}(\phi) = \frac{1}{2} \sum_{t=1}^n \Delta_t(\phi)^2 + Reg(\phi)$$

We use the regulariser $Reg(\phi) = \frac{\beta}{2} |S| \log_2(n)$ which is similar to the model penalty used by MDL (Rissanen, 1978).

This cost can be better studied if we look at its behaviour as the length of the history $n \rightarrow \infty$. In order to do that, we will need the following terms. Let $d_{s,a} := \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{t=1}^n Pr(s_t = s, a_t = a) = \pi(s, a) \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{t=1}^n Pr(s_t = s)$ be the limiting state-action distribution. $d_{s,a}$ always exists for any fixed stationary policy.

The Mean Squared Bellman Error (MSBE, Sutton et al. (2009)) for Q-values is

$$\begin{aligned} MSBE(Q) &= \sum_{s,a} d_{s,a} (\mathbb{E}[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) | s, a]^2) \\ &= \sum_{s,a} d_{s,a} \left(\sum_{s'} T(s, a, s') (R(s, a, s') + \gamma \max_{a'} Q(s', a')) - Q(s, a) \right)^2 \end{aligned}$$

We also define the variance of Q as the d -weighted average of the variance of the Bellman error for each state-action pair.

$$\begin{aligned} Var(Q) &= \sum_{s,a} d_{s,a} Var(R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) | s, a) \\ &= \sum_{s,a} d_{s,a} Var(R(s, a, s') + \gamma \max_{a'} Q(s', a') | s, a) \end{aligned}$$

Given these two definitions, we observe that as n grows large and for a fixed stationary behaviour policy π , the average expected sum-of-squared errors can be expressed as the sum of the above two quantities.

$$\begin{aligned}
& \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{t=1}^n \mathbb{E}_{\pi}[\Delta_t(\phi)^2] \\
&= \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{t=1}^n \sum_{s,a} \Pr(s_t = s, a_t = a) \mathbb{E}_{\pi}[\Delta_t(\phi)^2 | s_t = s, a_t = a] \\
&= \sum_{s,a} \mathbb{E}_{\pi}[(R(s, a, s') + \gamma \max_{a^*} \hat{Q}_{\phi}^*(s', a^*) - \hat{Q}_{\phi}^*(s, a))^2 | s, a] \left(\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{t=1}^n \Pr(s_t = s, a_t = a) \right) \\
&= \sum_{s,a} d_{s,a} \left(\mathbb{E}_{\pi}[R(s, a, s') + \gamma \max_{a^*} \hat{Q}_{\phi}^*(s', a^*) - \hat{Q}_{\phi}^*(s, a) | s, a]^2 \right. \\
&\quad \left. + \text{Var}_{\pi}(R(s, a, s') + \gamma \max_{a^*} \hat{Q}_{\phi}^*(s', a^*) | s, a) \right) \\
&= \text{MSBE}_{\pi}(\hat{Q}_{\phi}^*) + \text{Var}_{\pi}(\hat{Q}_{\phi}^*)
\end{aligned}$$

If Q-learning converges to $\hat{Q}_{\phi}^* = Q_{\phi}^*$ for a particular ϕ then the $\text{MSBE}(Q_{\phi}^*)$ is zero and

$$\text{Cost}_{QL}(\phi) = n \text{Var}_{\pi}(Q_{\phi}^*) + \text{Reg}(\phi)$$

This cost extends easily to the function approximation setting, where instead of tabular Q, we have some parameterised function class to represent Q. In this chapter, we will use linear function approximation where $Q(\phi(h_t), a_t)$ is approximated by $\xi(h_t, a_t)^{\top} w$ where $\xi : \mathcal{H} \times \mathcal{A} \rightarrow \mathbb{R}^k$ for some $k \in \mathbb{R}$. In the function approximation case however, the approximation class can be such that the MSBE does not vanish even when Q-learning converges.

$$\text{Cost}_{QL}(\xi) = \min_w \frac{1}{2} \sum_{t=1}^n \left(r_{t+1} + \gamma \max_a \xi(h_{t+1}, a)^{\top} w - \xi(h_t, a_t)^{\top} w \right)^2 + \text{Reg}(\xi)$$

The regulariser on ξ is $\text{Reg}(\xi) = \frac{\beta}{2} k \log_2(n)$.

5.3 Consistency of Cost_{QL}

In this section, we prove the consistency of Cost_{QL} for the class of bounded-memory finite state machines (FSM) as a corollary of the results by Sunehag and Hutter (2010). They proved that the original Cost was consistent in the sense that asymptotically it is minimised by a map that predicts the reward sequence well. We show that Cost_{QL} is almost surely minimised by ϕ with the lowest average squared Q-learning errors. We require the following definitions and propositions from that paper.

Definition 20. (bounded memory FSM, Definition 24 in Sunehag and Hutter (2010)) Suppose that there is a constant κ such that if we know the last $\kappa + 1$ perceptions $y_{t-\kappa}, \dots, y_t$ then the present state s_t is uniquely determined. Then we say that the FSM has memory of at most length κ (not counting the current) and that it has bounded memory.

Definition 21. A sequence is *ergodic* if every finite substring has asymptotically converging frequencies.

Definition 22. A map ϕ based on an FSM is said to generate the environment if $\phi(h_t) = s_t$ and $Pr(o_{t+1}r_{t+1}|h_t, a) = Pr(o_{t+1}r_{t+1}|s_t, a)$ i.e. o_{t+1} and r_{t+1} are generated from distributions that only depend on s_t and a_t .

Definition 23. An FSM is said to generate the environment ergodically if for any sequence of actions such that the action frequencies for each state converge asymptotically, the state sequence and reward sequence are ergodic.

Proposition 24. Suppose that we have an FSM of bounded-memory generating the environment ergodically and the action frequencies for any state converge asymptotically, then we will almost surely generate an ergodic history sequence.

Proposition 25. Let the environment be ergodically generated by an FSM of bounded memory. If we consider a finite class of maps $\Phi = \{\phi_1, \dots, \phi_K\}$ and a stationary policy such that the action frequencies for any internal state of the generating FSM converge asymptotically, the sum of squared Q-learning errors $\frac{1}{n} \sum_{t=1}^n \Delta_t(\phi)^2$ almost surely converges to an expectation $\Gamma_\phi = \mathbb{E}_{s,a,s'}[\Delta_t(\phi)^2]$. Furthermore, given a sublinearly growing regulariser, $Cost_{QL}(\phi)$ is almost surely asymptotically minimised by ϕ^* with the lowest average squared Q-learning errors, i.e. $\exists N_0 > 0$ such that for all $n > N_0$ $\phi^* = \arg \min_\phi Cost_{QL}(\phi) = \arg \min_\phi \Gamma_\phi$.

Proof. By Proposition 24 the history sequence is almost surely ergodic. Suppose we pick some map $\phi \in \Phi$. Since the history uniquely determines the state via ϕ , the corresponding sequence $s_0, a_0, r_0, \dots, s_t, a_t, r_t$ is also ergodic (see Proposition 25, Sunehag and Hutter (2010)). Thus the frequency of the tuples (s, a, s') converge asymptotically. Let Q_ϕ^* be the optimal Q-value for ϕ . Then by the above convergence and since R is a function of the (s, a) tuples, the average of squared Q-learning errors also converges i.e. $\exists N_0 > 0$ such that $\forall n > N_0$

$$\frac{1}{n} \sum_{t=1}^n \Delta_t(\phi)^2 = \mathbb{E}_{s,a,s'}[\Delta_t(\phi)^2] := \Gamma_\phi$$

Given a sublinearly growing regulariser such as $Reg(\phi) = \frac{\beta}{2}|S|\log_2(n)$, there exists an $N_1 > N_0$ such that for all $n \geq N_1$, $\min_\phi Cost_{QL}(\phi) = \min_\phi n\Gamma_\phi + Reg(\phi) = \min_\phi n\Gamma_\phi = \min_\phi \Gamma_\phi$. The first equality holds since $N_1 > N_0$. For the second equality, assume two maps ϕ_1 and ϕ_2 and some initial n, N_2 with $N_0 < n < N_2$ such that $\Gamma_{\phi_1} < \Gamma_{\phi_2}$ but $Cost_{QL}(\phi_1) > Cost_{QL}(\phi_2)$. Then

$$Cost_{QL}(\phi_1) - Cost_{QL}(\phi_2) > 0$$

$$\implies \underbrace{n(\Gamma_{\phi_1} - \Gamma_{\phi_2})}_{<0} - \underbrace{(Reg(\phi_1) - Reg(\phi_2))}_{>0} > 0$$

The linearly growing (negative) difference term will eventually dominate the above difference i.e. there exists $N_1 > N_2$ such that for all $n > N_1$ $Cost_{QL}(\phi_1) - Cost_{QL}(\phi_2) < 0$. Therefore, for all $n > N_1$ $\phi^* = \arg \min_{\phi} Cost_{QL}(\phi) = \arg \min_{\phi} \Gamma_{\phi}$ as required. Suppose $\phi_i \in \arg \min_{\phi} Cost_{QL}(\phi)$ and $\phi_i \notin \arg \min_{\phi} \Gamma_{\phi_i}$ for some M , then by the above argument there exists $N_i > M$ such that $\phi_i \notin \arg \min_{\phi} Cost_{QL}(\phi)$. Since there are only a finite number of maps, there exists $N_* < \infty$ such that for all $n > N_*$, $\phi^* \in \arg \min_{\phi} Cost_{QL}(\phi)$ as required. $\square$

5.4 Algorithm

The algorithm is a replacement of the standard cost function by $Cost_{QL}$ in the generic Φ MDP algorithm given in Algorithm 3. Although the introduction of $Cost_{QL}$ into Algorithm 3 is mostly straightforward there are some implementation details that need to be taken into account. This section covers these changes in detail.

5.4.1 Implementation of $Cost_{QL}$

The implementation of $Cost_{QL}$ is provided in Algorithm 7. We fix the $cost_precision = 0.001$ and $cost_iterations = 20$. The other parameter to control is the learning rate (α) of Q-learning itself. If it is set too low, then at the initial stages of the algorithm the learned weights w might greatly underestimate the true weights which has the effect of making Γ_{ϕ} small. This has an impact on how we set the regularisation constant β such that the regulariser is initially on the same scale as Γ_{ϕ} . Thus a rule-of-thumb is that decreasing α should be matched by a corresponding decrease in β .

An important point here is that the learning rate we set is global to all maps i.e. we are using the same α to learn Q_{ϕ}^* for all ϕ . In the function approximation case, we use a technique that disperses the Q-learning update equally across the number of active features (explained in Section 5.4.5.2). In the tabular case, we observe no problems with using a constant learning rate for all maps, although we use small learning rates (e.g. $\alpha = 0.01$) for some environments.

For this chapter, to make Q-learning perform more effectively, we give the algorithm additional information in the form of end of episode markers. The Q-learning portion of the algorithm uses this information to evaluate the value of the current state-action pair only until the end of the episode.

In the tabular setting we could use any off-policy algorithm to find Q_{ϕ}^* . However, in the function approximation setting, the fixed point and Bellman error solutions are different. The recently proposed algorithm Greedy-GQ by Maei et al. (2010) minimises the projected Bellman

Algorithm 7: The $Cost_{QL}$ function

```

Input: ;
   $\phi$  : The current map;
   $h_n$  : The history sequence so far of size  $n$ ;
   $\alpha$  : The learning rate for Q-learning;
   $R_{max}$  : The maximum possible reward (or an upper bound);
for  $t = 1$  to  $n$  do
  |  $s_t = \phi(h_t)$ ;
end
 $w = 0$ ;
iterations = 0;
 $n\hat{\Gamma}_\phi = Qlearn(w, s_{1:n}, r_{1:n}, a_{1:n-1})$ ;
 $n\hat{\Gamma}'_\phi = 0$ ;
while  $n\hat{\Gamma}_\phi - n\hat{\Gamma}'_\phi > cost\_precision$  or  $iterations < cost\_iterations$  do
  |  $n\hat{\Gamma}_\phi = Qlearn(w, \alpha, s_{1:n}, r_{1:n}, a_{1:n-1})$ ;
  |  $n\hat{\Gamma}'_\phi = n\hat{\Gamma}_\phi$ ;
  | iterations += 1;
end
 $Reg(\phi) = \frac{\ell}{2} R_{max} |\mathcal{S}| \log_2(n)$ ;
Return  $n\hat{\Gamma}_\phi + Reg(\phi)$ ;

```

error via approximate gradient descent. The convergence guarantees only apply to the case where the behaviour policy is fixed, which is unfortunately not true in our setting, but the algorithm can still be used without the guarantees. We found that Greedy-GQ converged slower than Q-learning for all our domains (with reasonable amount of parameter selection), and Q-learning did not diverge on any of them. Thus, for the rest of this chapter we use standard Q-learning unless otherwise stated. Note that Q-learning is a special case of Greedy-GQ that can be obtained by setting the update parameter for the damping factor to be zero.

5.4.2 Finding a good policy

After we have found a ϕ_{best} that minimises $Cost_{QL}$ on the history so far, we need to find a good policy for it. Since we have already estimated $Q_{\phi_{best}}^*$ in calculating $Cost_{QL}$, we can simply use $\pi(s) = \arg \max_a Q_{\phi_{best}}^*$ as the policy. However, we may choose to (re)learn $Q_{\phi_{best}}^*$ using a different algorithm. The function *FindPolicy* in Algorithm 3 is currently substituted with Q-learning although any off-policy model-free algorithm could be used. Model-based algorithms may also be used to find the policy, but this would involve computing the model for ϕ_{best} and additionally make the extension to function approximation unnatural. In the implementation used for our experiments, we use a different instance of Q-learning to find a policy. The primary difference is optimistic initialisation; while calculating $Cost_{QL}$ we wish to be accurate about finding Γ_ϕ , however the agent can use a more optimistic policy to trade-off acting and exploring. In fact, this influences results on domains such as T-Maze where it is important to

have systematic exploration at the start. Additionally, we execute an ε -exploration policy in order to compensate for an potentially inaccurate state representation. Both optimism and ε -exploration can help alleviate (but not completely remove) the exacerbated exploration-exploitation (e^3) problem discussed in Section 2.3.2.3.

On-policy methods may also be used to find the policy, however in our algorithm the history has (most likely) been created by several different stationary policies (or one non-stationary one) so their use is not well-motivated. However, it is possible to only use the data from the most recent epoch to learn the policy, although this throws away a lot of useful data. In such a setting, it would make sense to increase the epoch size.

Note that Proposition 25 only holds for stationary policies. Unfortunately in practice, it is very hard to use a single policy across the entire learning process, primarily due to the exacerbated exploration-exploitation problem (Section 2.3.2.3); it is hard to explore well within the environment without having a good map to start with.

5.4.3 Initialisation

As in $CT\Phi$ MDP and $LST\Phi$ MDP the agent executes a number of initial random actions which provide with a starting history. The agent is also initialised with a map ϕ_{init} which it uses as a starting point for the first simulated annealing procedure. The number of initial random actions is given as a parameter, and is generally fixed to 200 for small domains but a larger number of random actions is needed for domains like POCMAN. This is, again, to provide a boost of exploration to avoid the e^3 problem.

A large number of initial random actions can be a curse rather than a blessing in domains such as the TMaze, where particular history sequences make it easier for the agent to learn a minimal looping suffix tree. For example, with any non-trivial map the agent in the TMaze can learn the policy, go to the end of the corridor and choose to go left. This will be correct half the time, and also allows the agent to fill its history with useful observation sequences such as $L0^*DL$ or $L0^*DR$ where L =left, R =right and 0 is the observation received in the corridor. This history allows the agent to eventually learn the true map. Random exploration is very unlikely to produce such sequences, since the such a strategy must random walk a corridor of length 50 to get to the decision point in the maze.

The starting map provided to the agent depends on the feature space and we will discuss that for each feature map independently. However, the general principle for an initial map is that it must be small, but not the trivial map that maps all observations to a single state. Such maps can lead to problems in both $OCost$ and $Cost_{QL}$.

5.4.4 Execution

In the next section we will talk about the features we use in both the tabular and function approximation settings. The feature sets will determine the neighbourhood function and

initial map. However, the general procedure for execution remains the same and is provided in Algorithm 3.

After initialisation, the agent acts according to the policy found by Q-learning for the starting map for M iterations, which we call an *epoch*. Before the next epoch, the agent uses simulated annealing Algorithm 4 to search for a better map. This process repeats for some pre-specified number of epochs, or until the learned policy is good enough (user-determined).

5.4.5 Features

We evaluate the new cost in the tabular setting on some standard small domains using suffix trees and on long term dependency domains using looping suffix trees. In the function approximation setting we define a new feature set called *event selectors*, and a modification of this set called *bit selectors*.

5.4.5.1 Suffix trees and Looping suffix trees

The implementation of suffix trees and looping suffix trees remains identical to Chapter 4, and the corresponding neighbourhood functions are as specified there. We emphasise that the neighbourhood function is not allowed to add loops until the currently chosen map has more than $2|\mathcal{O}|$ states, this heuristic proves important in not adding loops that disregard important observations early on in the annealing process, when the agent does not yet have enough data to correctly evaluate the cost of a map.

5.4.5.2 Event selectors

An event selector is a set of features ξ_j . Each feature ξ_j consists of a position m and an observation o . Feature ξ_j is on (i.e. equal to 1) if the $(n - m)$ -th position in the history has observation o . More formally we use Definition 26.

Definition 26 (Event selector). Let o_i be the i -th observation in $h_{1:n}$. An event selector is a set $\xi = \{\xi_1, \xi_2, \dots, \xi_k\}$ where $\xi_j : \mathbb{N} \times \mathcal{O} \times \mathcal{O}^n \rightarrow \mathbb{B}$ is a function such that $\xi_j(m, o, o_{1:n}) = 1$ if $o_{n-m} = o$ and 0 otherwise.

The neighbour of an event selector is either the addition or removal of a feature to the set. Note that we can similarly define an event selector for observation-action pairs, but we use the definition above for this chapter.

The bit selector is a modification of the event selector. Instead of picking out whether the $(n - m)$ -th position in the history has observation o we check whether the c -th bit of o_{n-m} is 0 or 1. This class of features is particularly useful in dealing with environments where the individual bit structure has relevance (such as Pocman). This binarisation is similar to those performed by MC-AIXI (Veness et al., 2011) and CTΦMDP (Hutter, 2009b).

Definition 27 (Bit selector). Let $\text{bin}(o_i)$ be the binarisation of the i -th observation in $h_{1:n}$. Let $\text{bin}(o_i)^c$ represent the c -th bit of this binary number. A bit selector is a set $\xi = \{\xi_1, \xi_2, \dots, \xi_k\}$ where $\xi_j : \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathcal{O}^n \rightarrow \mathbb{B}$ is a function such that $\xi_j(m, c, b, o_{1:n}) = 1$ if $\text{bin}(o_{n-m})^c = b$ and 0 otherwise.

An important detail when implementing Q-learning with function approximation in this setting is to normalise the change across the features, i.e. the update is now,

$$w_{t+1} \leftarrow w_t + \frac{\alpha}{\sum_i \phi_i(s, a)} \Delta_t(\phi) \phi(s, a)$$

We need the above heuristic in order to alleviate the fact that we have a single learning rate across all maps in the class. The normalisation helps with maps where one feature may have a very high value compared to the rest, providing a uniform update magnitude across features. We also observed empirically that this appears to resolve potential divergence issues. In the tabular case, $\sum_i \phi_i(s, a) = 1$ and so the re-weighting has no effect.

5.5 Experiments

In this section we describe the experimental setup and the domains used to evaluate the new cost. On the smaller domains we test three algorithms : history Q-learning (hQL) using suffix trees, the function approximation of hQL using event selector features (FAhQL) and the CT Φ MDP using suffix trees. Φ MDP has proven to be competitive (Nguyen et al., 2011, 2012) against MC-AIXI (Veness et al., 2011), and better than both Active LZ (Farias et al., 2007) and U-tree (McCallum, 1996). We also test on the long-term memory domains from Chapter 4 comparing against LST Φ MDP on TMaze and Locked door.

Every experiment was run 30 times. The agent was given an initial history produced by taking random actions. Each run of an experiment was conducted over 100 epochs. Each epoch contains 100 iterations of the agent performing actions according to its current policy, based on the current map. Additionally the agent uses ε -exploration until a certain number of epochs is completed. After the completion of an epoch, the agent is given a chance to change current map via the simulated annealing procedure. The annealing procedure uses an exponential cooling function. Plots show every 5th point with 2 standard error on either side. The exact constants used for all the experiments can be found in Table 5.1. When using Q-learning (both within the cost and without) we use several runs through the data such that it converges. We found that for the small domains we test on ε -exploration was enough to ensure convergence to optimality.

We mildly tune the parameters by running the algorithm for a small number of iterations on each environment. The FAhQL and hQL constants are very similar. The choice of regularisation constant appears to have some dependence on the stochasticity of the environment.

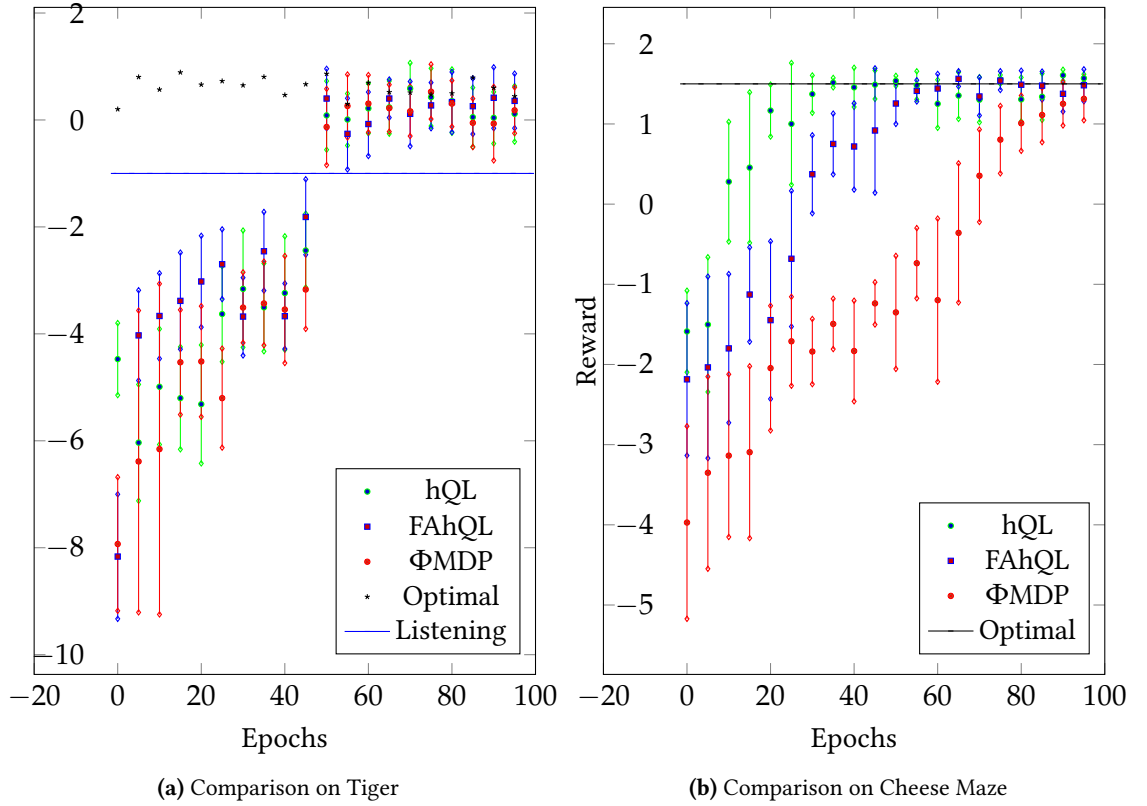


Figure 5.1: Short-term memory domains

5.5.1 Small domains with short-term memory

Tiger

A description of the Tiger domain can be found in Section 3.3.10.1. The optimal policy for tiger with a listen probability of 0.85 is to wait until the last two listens agree and choose accordingly. We show this in Figure 5.1a. The graph showing the optimal policy is not constant due to the fact that each epoch does not necessarily end at the end of an episode, hence some rewards can be carried to the next epoch.

From Figure 5.1a we see a very similar performance from the three algorithms. It should be noted that the suffix tree based algorithms hQL and Φ MDP found features of size 21 (7 states, 3 actions) to be optimal, the function approximator generally uses between 15 and 18 features (although sometimes goes up to 24). The features used by the linear function approximator for optimal performance in the Tiger problem are shown in Figure 5.2b, with the suffix tree (having 21 features) is shown in Figure 5.2a.

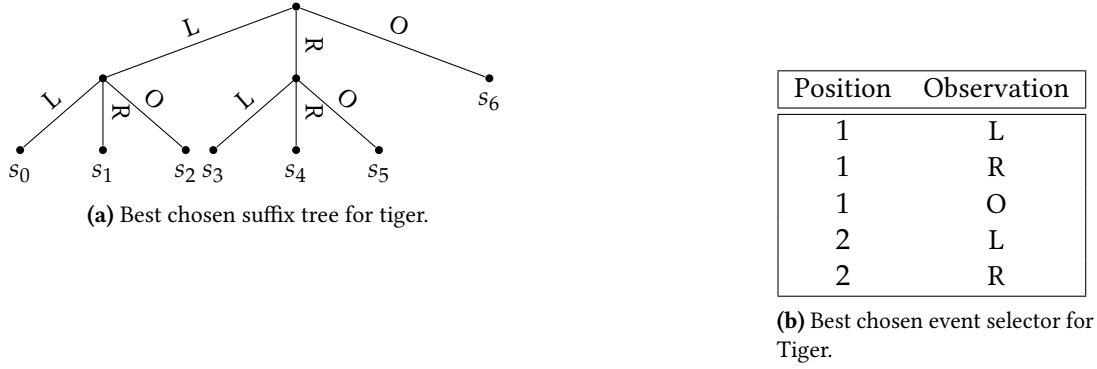


Figure 5.2: The observations L, R and O refer to Left, Right and Open (Door) respectively. The suffix tree on the left shows that the agent remembers two observations in the past of listening in order to make its decision about which door to open. With a linear function approximator however, this can be represented more compactly as seen in the table on the right.

Cheese Maze

The cheese maze domain is described in Section 3.3.10.2. On this domain there are clearer differences between the algorithms. hQL converges the fastest, while Φ MDP converges only in the last few epochs. We observe that hQL at best uses 64 features (16 states, 4 actions) while FAhQL generally converges to using 36 features. In the above experiments we also see some sudden drops in the reward (for e.g. hQL at epoch 20 in Figure 5.1a). These dips are generally formed by the algorithm changing its map. A new map implies new states and the algorithm does not necessarily know how to act optimally in these new states given the data it currently has.

5.5.2 Long term memory domains

In order to compare these results to the previous chapter, we follow the same setup described in Section 4.5. The following experiments all use looping suffix trees as the map class.

5.5.2.1 Deterministic TMaze

We provide results for hQL in Figure 5.3a using a 50 length TMaze as in the original experiment in Chapter 4. hQL provides results more data efficiently than LST Φ MDP, we found that we could stop searching for a map at 100 epochs and get perfect results. LST Φ MDP on the other hand takes 250 epochs to achieve map convergence.

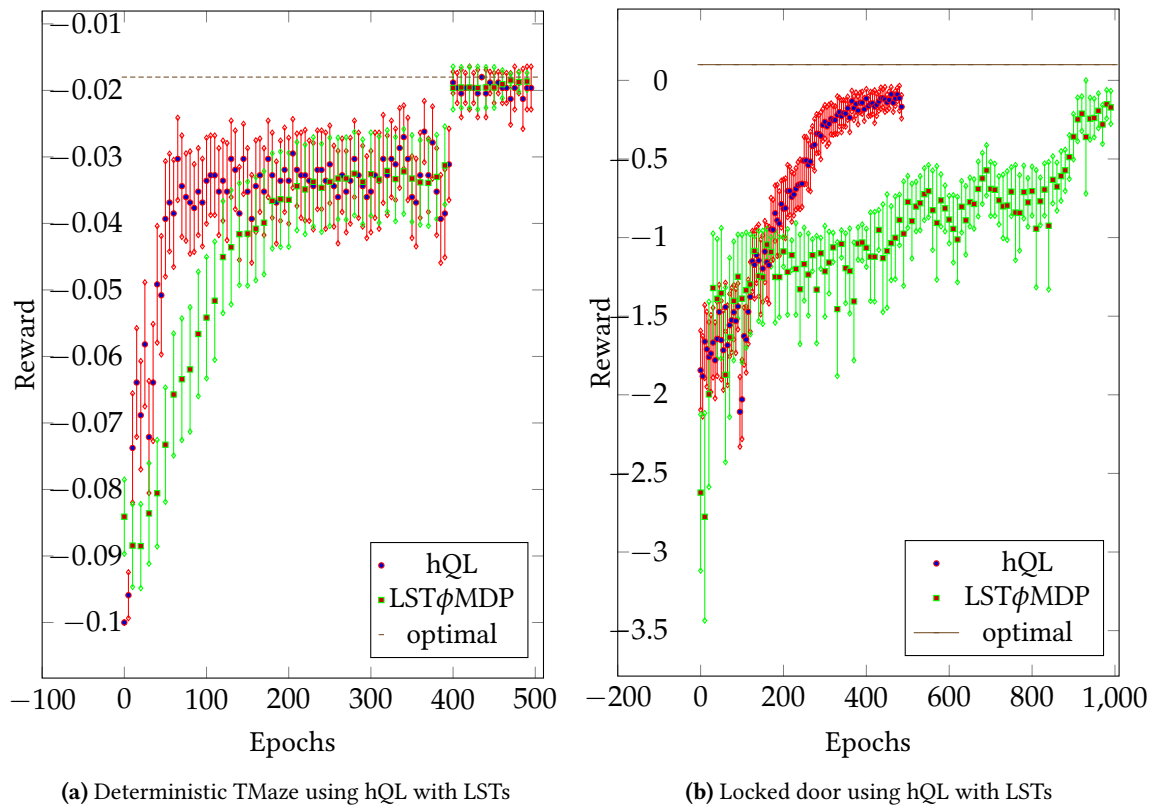


Figure 5.3: Long-term memory domains

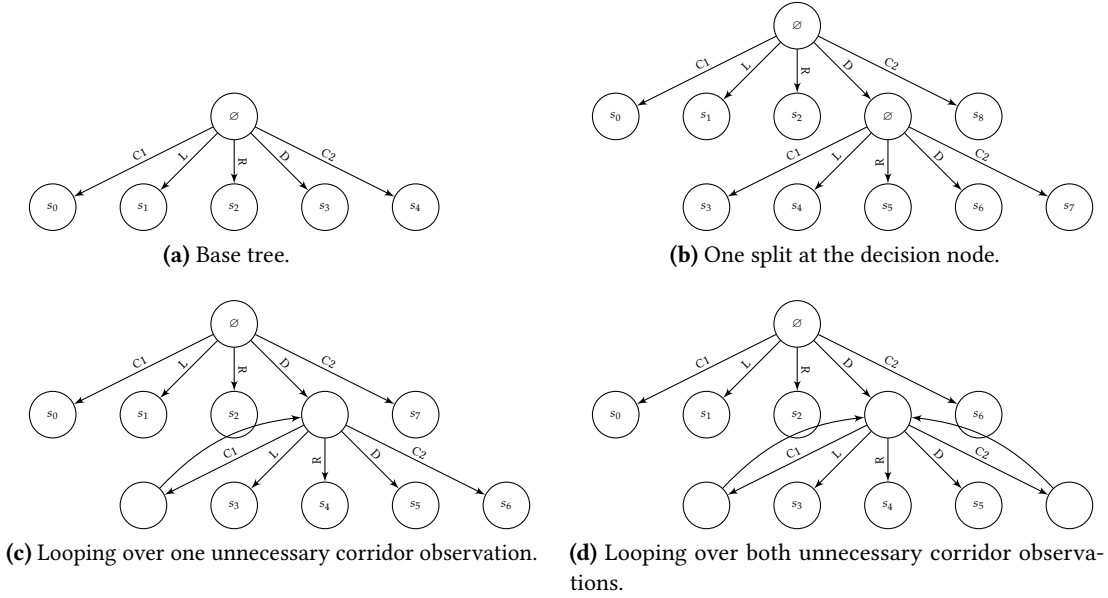


Figure 5.4: The compared trees used to explain why simulated annealing cannot find the optimal tree easily in TMaze with a stochastic corridor. L is the left observation, R is the right, C1 and C2 are the noise in the corridor and D is the observation received at the decision node.

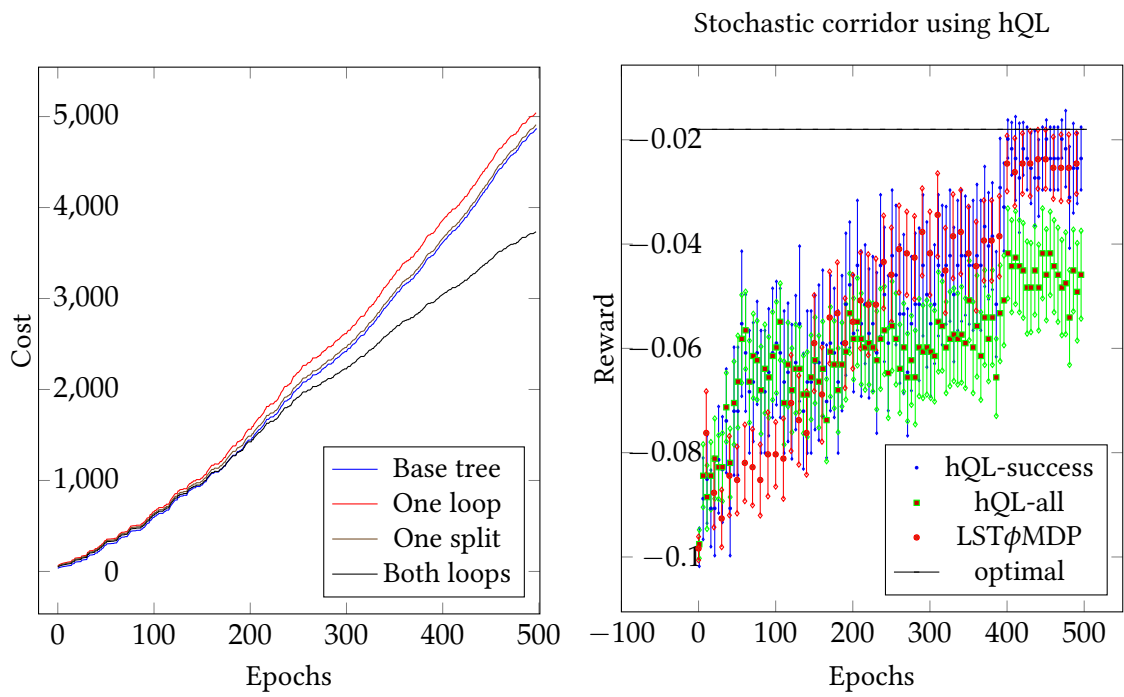
5.5.2.2 Stochastic TMaze

The cost of the optimal map for the version of stochastic TMaze with a noisy corridor was always lower than the other considered maps, but there were problems with simulated annealing which are not present when using *OCost*. This meant that the agent did not find the optimal map in every run, it was successful in 22 out of 50 runs. On the successful runs it always achieved the optimal reward by the end. We provide graphs of only the successful runs and all the runs in Figure 5.5b. The successful runs converge in a similar way to $LST\Phi$ MDP.

Figure 5.5a shows us why the simulated annealing algorithm did not always succeed. The base tree starts out as the cheapest, but around 200 epochs the optimal tree with both loops starts to win out against it, with the difference growing much larger over time. However, the neighbours of the optimal tree have very high costs when compared to the base tree, in particular the tree with one split only. Thus the agent must make all the right neighbourhood operations to arrive at the optimal tree which is very unlikely.

5.5.2.3 Locked door

We also tested on the larger domain, locked door. Exploration was turned off at 4000 epochs for hQL as opposed to 9000 epochs for $LST\Phi$ MDP. The results perform much better over $LST\Phi$ MDP in several respects. Over 50 runs, the algorithm only failed to converge to a reasonable result once. $LST\Phi$ MDP had only a success rate of 50%. The graphs shows that the



(a) A comparison of $Cost_{QL}$ for different maps on TMaze with a stochastic corridor. Lower is better. For a description of the maps see Figure 5.4.

(b) TMaze with stochastic corridor

Figure 5.5: Examination of TMaze with stochastic corridor

agent converges much quicker to close to the optimal, and remains there consistently. An important observation is that though the optimal policy is relatively simple, namely, go to the key and return to the lock via the shortest path, the tree that the agent learns may be only able to represent some slightly more convoluted policy (for example, go to the key, go down 3 steps, then go back to the goal). Such artefacts are a result of the agent’s exploration history so far and can be avoided by dominating the agent’s history by examples of the optimal policy to begin with, which would defeat the purpose of these experiments.

5.5.2.4 Event selectors

The event selector for a TMaze of length n should look like the one shown in Figure 5.6a. A graph of the convergence of a single run of Q-learning for the below map for a TMaze of length 50 shows that the map is optimal or contains the optimal map. However, the convergence is very sensitive to the learning rate and also the exploration constant. In the tabular case, an $\varepsilon = 0.1$ exploration is not enough to complete reduce the average of the optimal policy so drastically as here. This is because event selectors are not as flexible as a looping suffix tree. An event selector looks for the decision point in the corridor *exactly* n steps ago. If exploration results in the agent wandering the corridor for longer, the agent cannot find the initial observation, and thus cannot make the correct action. Thus if the agent does not follow the optimal policy, it needs a different map. Thus the event selector on TMaze is not robust to exploration. Unfortunately, this means that running the full Φ MDP search is also sensitive to these conditions, and also sensitive to the different policies induced by the maps. We can alleviate this by minimising the exploration period to the first few (10) epochs, and only using the mildly optimistic initialisation of the agent. This results in the agent mostly having histories that look like $\{R|L\}C^nD\{R|L\}$ which is what is necessary for it to find the correct map. However, this reveals a flaw in event selectors; they are not robust to histories generated by changing policies.

5.5.3 A large domain : Pocman

The Partially Observable Pacman domain was introduced by Veness et al. (2011). We described the domain in Section 3.3.5.2. In this chapter, we treat it as a non-episodic discounted task in an effort to be as general as possible and to allow for a fair comparison to the implementation of MC-AIXI which treats the domain similarly.

For this domain, we change the experimental setup. We now test FahQL with a bit selector feature space, similar to the bitwise representation used by MC-AIXI-CTW. It is only feasible to do one run so the graphs show a rolling average over the previous 1000 epochs (or 100,000 iterations). We no longer run through the Q-learning multiple times, and while initial estimates might be inaccurate we use enough data that this is not a problem, and we change the map every 200 iterations rather than 100. After 100,000 iterations we delete 10,000 timesteps of data from the start of the history, so that our stored history is now always 100,000 timesteps

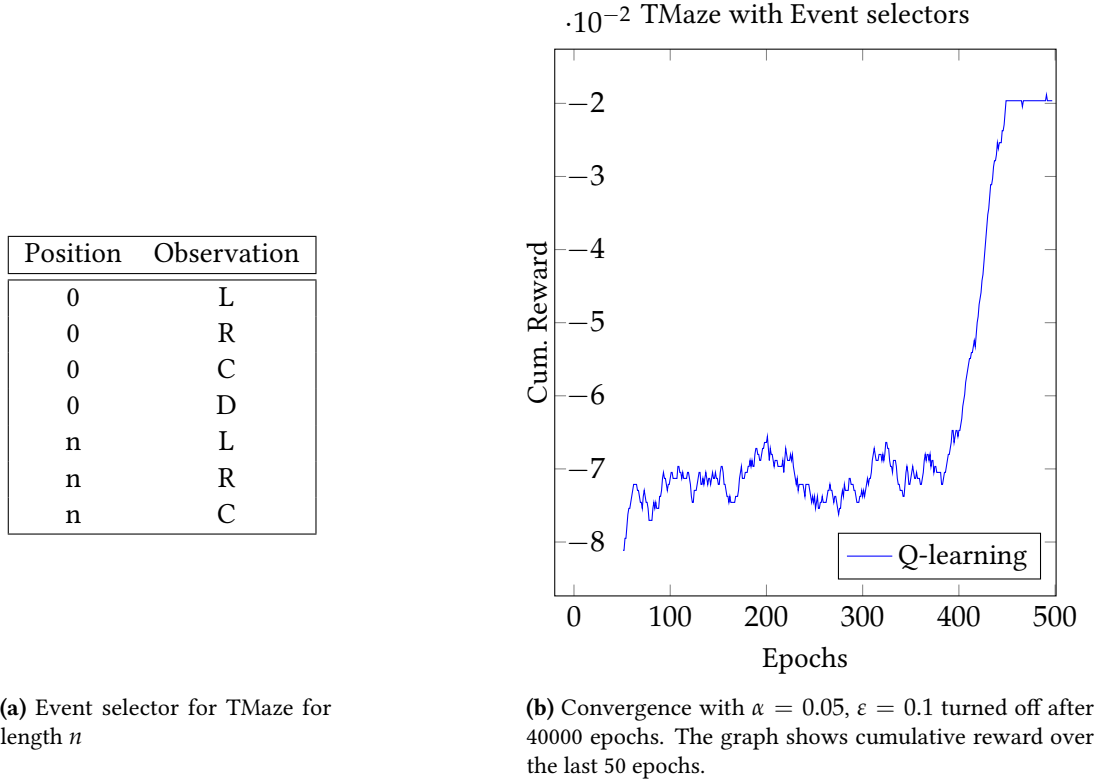


Figure 5.6: Q-learning using event selectors on T-Maze

long (i.e. we consume a constant amount of memory). This helps both speed and memory efficiency, and the number of timesteps is enough to ensure convergence. We compared against MC-AIXI-CTW which is the best performing algorithm on the domain. In order to make this comparison fair we use the same setup as in Veness et al. (2011) with an exploration rate starting at 0.9999 that geometrically decreases at a rate of 0.99999 per timestep. CT Φ MDP cannot deal with such large observation spaces due to the large explosion in number of trees it must consider. The context-tree maximising algorithm upon which CTMRL is based, is a variant of CTW, and consumes a large amount of memory. CTMRL also needs many ad-hoc manipulations to run on Pocman. These included sacrificing data efficiency to avoid memory problems by discarding the CTMs in each learning loop, making the algorithm space and relatively time efficient but needing 100 million iterations.

Unfortunately we could not run MC-AIXI with the exact parameter settings as in Veness et al. (2011) due to memory constraints (see Table 5.2). The furthest that the 96 bit, 4 look ahead algorithm went was to a 100,000 iterations before running out of memory. We instead ran MC-AIXI using 48-bit (which is 2 percepts in the past) and 2 look-ahead. FAhQL shows better performance than the 48-bit MC-AIXI on this domain using a maximum of 4 lookahead with final rolling average of 1.143 as compared to 0.863. The baseline Q-learning algorithm using only 16-bits or one percept in the past outperformed MC-AIXI and came close to performing as well as FAhQL with a score of 1.117. This indicates that the performance gain from dealing

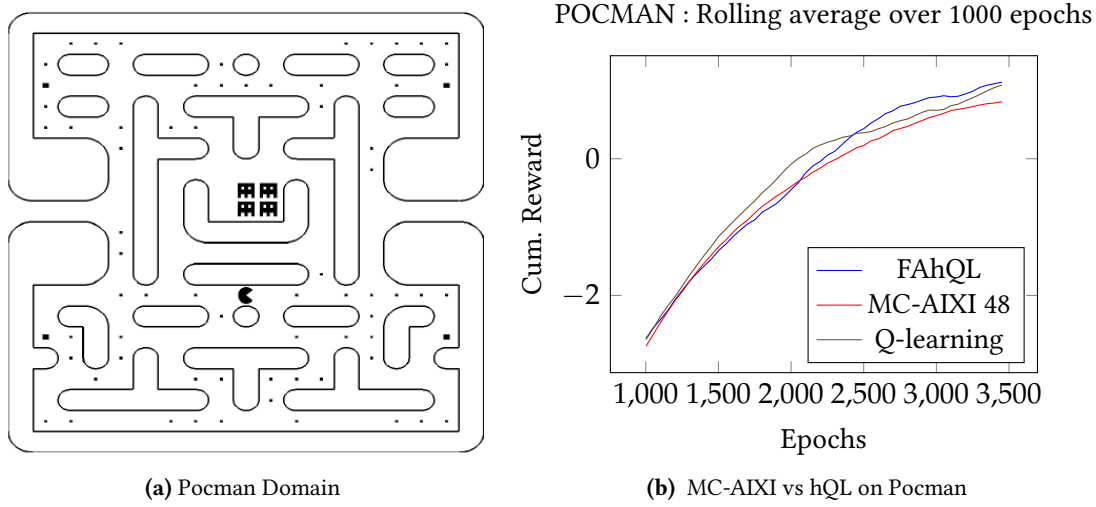


Figure 5.7: The Pocman experiment

with partial observability is not great, unless the agent models more sophisticated behaviour such as eating powerpills to chase ghosts. This was also noted by Hamilton et al. (2014) who further tested on a modified version of Pocman known as S-Pocman with food sparsely distributed in fixed positions on the map, which the agent could learn over time. They also removed the observations that allowed the agent to see the food, so the agent had to rely on the “smell” observations to perform well.

Episodic vs non-episodic Pocman While Pocman is inherently an episodic domain, interesting behaviour arises if we treat it non-episodically as above. In this case, it is possible that once the agent has exhausted the rewards in the immediate horizon ($\frac{1}{1-\gamma}$) it may choose to commit suicide by running toward a ghost or at least not running away from one. This is because it takes into account the future rewards from starting the next episode where it is once again surrounded by a large amount of food pellets. Thus the distinction between episodes is important here, in the episodic setting the agent cannot use potential future rewards from the next episode to impact its decisions in the current one.

5.6 Discussion

5.6.1 The utile distinction conjecture

Chrisman (1992a) conjectured that it was impossible to introduce distinctions (construct states) only for those features of the history that were useful in predicting the utility, since the agent must make those distinctions first in order to learn about their predictive ability. McCallum (1995) refuted this conjecture with the USM approach. Our off-policy $Cost_{QL}$ also provides

Table 5.1: Constants used for each experiment for hQL and FAhQL.

Experiment	α	β	ε	$amaxs$	$astartT$	$acoolRate$
hQL Tiger	0.01	1.5	0.1	10	4,000	$5 \cdot 10^{-2}$
hQL Cheese	0.01	0.02	0.1	10	5,000	$5 \cdot 10^{-2}$
hQL TMaze	0.01	0.03	0.1	30	100	$5 \cdot 10^{-4}$
FAhQL Tiger	0.01	1	0.1	20	7,000	$5 \cdot 10^{-2}$
FAhQL Cheese	0.01	0.02	0	10	5,000	$5 \cdot 10^{-2}$
FAhQL Pocman	0.001	2	decay	20	4,000	$5 \cdot 10^{-7}$

Common to all experiments was $\gamma = 0.99$, *init-history* the number of initial random actions performed by the agent at 200 and *stop-explore*=50 is the epoch beyond which the agent no longer uses ε -exploration (N/A for Pocman). α is the learning rate, *astartT* refers to the starting temperature T in the cooling schedule, *acoolRate* to the decay of the exponential cooling schedule and *amaxs* refers to the number of timesteps allowed in the annealing procedure.

a way of learning utile distinctions by measuring how useful a feature would have been in representing the value function on the history so far, thus refuting the conjecture. However, in some sense Chrisman was correct, since this approach (and USM) only works when the history is rich enough to tell that certain distinctions could have been useful in the past under the assumption that the future will be similarly distributed to the past.

5.6.2 Why use $Cost_{QL}$ instead of $OCost$?

While $Cost_{QL}$ appears to perform better than $OCost$ on the above toy domains in most situations, we have not yet made a clear case for using it over $OCost$. Intuitively, it seems that if we have a state space which can represent the value function of the optimal policy accurately, then it should be possible to learn the optimal policy using this map. In Hutter (2014) we learn that if the map is able to accurately represent the optimal value function on the history, then the Markovian-ness of a map is not necessary to ensure a good policy. However it is not clear that minimising $Cost_{QL}$ gives us a map that represents the optimal value function on the histories.

A predictive model of the observation/reward sequence might be unnecessarily large in some environments. For instance, in the TMaze environment, $OCost_{\alpha=0.5}$, which weights the predictions of the observations equally to that of the rewards, was not able to find the reward-optimal looping suffix tree. In fact, in the previous chapter we used $\alpha = 0.1$ to get $OCost_{\alpha}$ to perform well on the TMaze domains. This was worse in the Locked Door domain, where $\alpha = 0.01$ was necessary to find an optimal map. This place weights 0.9 and 0.99 respectively on the reward coding of $OCost_{\alpha}$. Nguyen (2013) shows that for any domain there exists an α such that for any $\beta < \alpha$ minimising $OCost_{\beta}$ correctly finds the optimal map (note this also includes $\alpha = 0$), this solution is an unsatisfying one. It also seems to suggest that only coding the rewards might yield a solution to this problem, i.e. maybe $OCost_0$ is

the correct cost to use, however this may result in very large state spaces, and additionally be suboptimal for high discount factors, where representing the state space is necessary to predict long-term reward.

A second advantage of $Cost_{QL}$ over $OCost$ is the ability to deal with continuous reward signals without any modification. In $OCost$ the reward signal is coded symbolically, so the rewards themselves must be discrete. Any continuous reward signal must thus be discretised, which may lead to very large reward spaces for which calculating the code length requires more data. In estimating $Cost_{QL}$ we don't code the individual rewards, we simply learn value functions for each map, which negates the need for the rewards to be discrete.

$OCost$ on small domains with small amounts of data runs a little faster than $Cost_{QL}$; in order to calculate $OCost(\phi)$ the agent only needs to go through the history once, whereas in the case of $Cost_{QL}$ it may need to run Q-learning multiple times. Asymptotically however, these have the same complexity of $O(n)$ where n is the length of the history. The computational advantage of $Cost_{QL}$ comes from the fact that there is no dependence on the size of the state space induced by ϕ in order to calculate $Cost_{QL}$ unlike when calculating $OCost$ from the model, which takes $O(|S_\phi|^2|\mathcal{A}|) + O(|S_\phi||\mathcal{A}||R|)$.

Of course, $Cost_{QL}$ also has the big advantage of being extensible to the function approximation setting, which can greatly help when dealing with environments with very large observation spaces. It is here that we start to run into the next bottleneck which is simulated annealing, the failure of $FahQL$ on the stochastic TMaze domain shows us that this can greatly impact performance on certain domains. Of course, the other issue is the feature sets that we consider. Event selectors, while useful on some domains that we tested on, are not great at long-term dependency problems. Additionally, we would like neighbours in the space of ϕ to correspond to neighbours in the space of $Cost_{QL}$, which is unfortunately not true with our current neighbourhood function. An examination of better neighbourhood functions for both looping suffix trees and event selectors would be useful future work.

While our experiments don't conclusively show that $Cost_{QL}$ is better than $OCost_\alpha$ in every domain, we believe that there are great advantages in using a value-based cost, in terms of the ability to find smaller representations and to use function approximation.

5.6.3 Computational issues

On Pocman, $FahQL$ used 400MB of memory and finished 350,000 iterations in about 17 hours. It was allowed to look up to 4 observations in the past. It shows comparable performance to MC-AIXI (48-bits) with over 36 times the memory efficiency and about 3 times the speed despite being single-threaded. It also has a much more significant speed-up over the 96-bit MC-AIXI but this is harder to quantify given our data. We should note that we can regulate the speed by controlling the epoch length.

Table 5.2: Computational comparison on Pocman

Agent	Cores	Memory(GB)	Time(hours)	Iterations
MC-AIXI 96 bits	8	32	60	$1 \cdot 10^5$
MC-AIXI 48 bits	8	14.5	49.5	$3.5 \cdot 10^5$
FAhQL	1	0.4	17.5	$3.5 \cdot 10^5$

5.6.4 Related Work

In comparison with the existing work in the feature reinforcement learning line (Hutter, 2009b; Nguyen et al., 2011, 2012; Nguyen, 2013; Daswani et al., 2012) our model-free criteria is much more discriminative since it is only concerned with predicting expected return under an optimal policy. Furthermore, we extend the agent to use function approximation. The resulting agent is similar in form to the lasso-TD algorithms of (Ghavamzadeh et al., 2011; Kolter and Ng, 2009; Johns et al., 2010) but we use the optimal policy TD error instead of a policy evaluation setting and we use an ℓ_0 regulariser instead of ℓ_1 .

CT Φ MDP can also use Q-learning to update the Q-values during the agent’s interaction with the environment. Q-learning is also used in CTMRL by Nguyen et al. (2012) as a substitute for value iteration in large environments. However, both these methods are inherently model-based. Our contribution therefore is not being the first algorithm that can use Q-learning in this setting, but rather being completely model-free, with a cost function that is evaluated using Q-learning itself.

The Internal Policy State Gradient method by Aberdeen and Jonathan (2002) also uses a map from observations to histories (in their case finite state controllers (FSC)). In ISPG, the FSC is used to directly parameterise the policy space and then gradient ascent algorithms are used to find the best policy. In our case, we use the learned state space to find the optimal value function which then gives us a policy. Compared to ISPG, we have the advantage of not needing to fix the number of internal belief states but instead learn a suitable size.

5.7 Conclusion

We introduced a new value-based model-free cost into the feature reinforcement learning framework. The cost is more discriminative and captures our intuitions about choosing features that can approximate the optimal value function well.

The original CT Φ MDP algorithm by Nguyen et al. (2011) is incapable of dealing with large observation spaces effectively whilst the CTMRL agent needs many ad-hoc modifications to work on the large Pocman domain and due to that suffered bad data efficiency. This has been a motivation for this work on finding an algorithm that works naturally with function approximation and which is as discriminative as possible. Our empirical evaluation shows some improvement in convergence speed on classical POMDP benchmark domains with

function approximation resulting in more economical feature vector sizes. We demonstrated our performance on a large domain Pocman where we performed competitively against MC-AIXI while using 20 times less memory. Our algorithm hQL provides computational efficiency at least on par with CTMRL (given the same number of map re-estimation points) while retaining the data efficiency of MC-AIXI and superior memory efficiency to both.

Reinforcement Learning with Value Advice

“ Make your own nature, not the advice of others, your guide in life. ”
The Oracle of Delphi to Cicero, 83 BC.

This chapter considers the problem of reinforcement learning with value advice. We consider this setting for two reasons. Firstly, because there are some practical cases where extracting a reactive policy from anytime algorithms such as UCT is useful. Secondly, because it is useful to separate the feature learning problem from the exploration-exploitation problem, particularly in large domains. This helps us understand whether performance errors are being made in representation or in learning. This chapter is based on Daswani et al. (2014) with Theorem 30 being a new addition.

6.1 Introduction

In this chapter, the agent is given limited access to an oracle that can tell the expected return (value) of any state-action pair with respect to the optimal policy. The agent must use this value to learn an explicit policy that performs well in the environment. We provide an algorithm called RLAdvice, based on the imitation learning algorithm DAgger. We illustrate the effectiveness of this method in the Arcade Learning Environment on three different games, using value estimates from UCT as advice.

Reinforcement learning (RL) agents (Sutton and Barto, 1998) learn how to act well via trial-and-error interactions with an environment that provides a real-valued reward signal. An RL agent has less information than in supervised learning, since the reward signal provides only partial feedback. Additionally, the agent needs to make choices about which parts of the environment to explore, leading to the famous exploration-exploitation problem. There has been research into reducing the difficulty of the reinforcement learning problem, particularly for large environments, by providing additional information to the agent in various forms. The

related fields of imitation learning, learning from demonstration, reinforcement learning with policy advice, inverse reinforcement learning and transfer learning all fall into this category.

This chapter attempts to answer the following problem. An agent is given some class of function approximators $\mathcal{Q}$ of the value function, and access to an oracle that provides the expected return of any state-action pair under the optimal policy. We assume that the agent cannot always use the oracle because of some constraints. For example, the oracle costs a lot of computation time/memory or the agent loses access to the oracle at some point e.g. separate training/testing stages. Thus, the agent should use the value information provided by the oracle to find a policy derived from $\hat{Q} \in \mathcal{Q}$ that performs well in the environment.

Motivation The primary motivation of this approach comes from the need to extract explicit policies from anytime algorithms such as UCT (Kocsis and Szepesvári, 2006). Consider a problem that has two stages, training and testing. In the training stage, the algorithm may use UCT on the simulator for control. However in the testing stage there may not be the computing resources or time to run UCT. In such cases it is useful to extract a reactive policy from UCT that will perform well without need for further simulation. While UCT does not use features of the environment, an explicit policy will rely on such features. This leads to a second use of an explicit reactive policy; to judge the usefulness of a particular class of function approximators in representing a good (or optimal) policy for a problem. If the learned policy extracted from UCT by our approach performs well, it indicates that the approximators being used are capable of representing such a policy. Thus, this method also provides a tool for evaluating classes of function approximators, although there is no guarantee that it finds the best performing policy in the class.

A related problem is to imitate the oracle policy as closely as possible. Focusing on this leads to a solution to our value advice problem. For example, we may simply treat it as a regression problem with training samples being the features and the oracle return following the oracle's policy. Unfortunately, the regression model from this dataset can result in a policy that does not necessarily perform well. The intuition behind this failure is that the class of function approximators cannot represent the value function of the oracle's policy, and the agent is learning according to the oracle's state distribution rather than its own. In imitation learning, an algorithm known as Dataset Aggregation (Dagger) by Ross and Bagnell (2010) deals with this problem, while still retaining the goal of imitating the oracle. Our proposed solution to the original problem is a modification of Dagger to suit our setting.

Main contribution The main contribution of this chapter is an algorithm for learning how to act well in a reinforcement learning problem given access to an oracle that can provide the value of any state-action pair in a training stage. Our algorithmic contribution is in the form of a modification of an existing imitation learning algorithm (Dagger) for this task, along with a comparison of various methods. Our testing suite comprises of three games from the Arcade Learning Environment, which shows the scalability of our method. Using just 100 episodes of

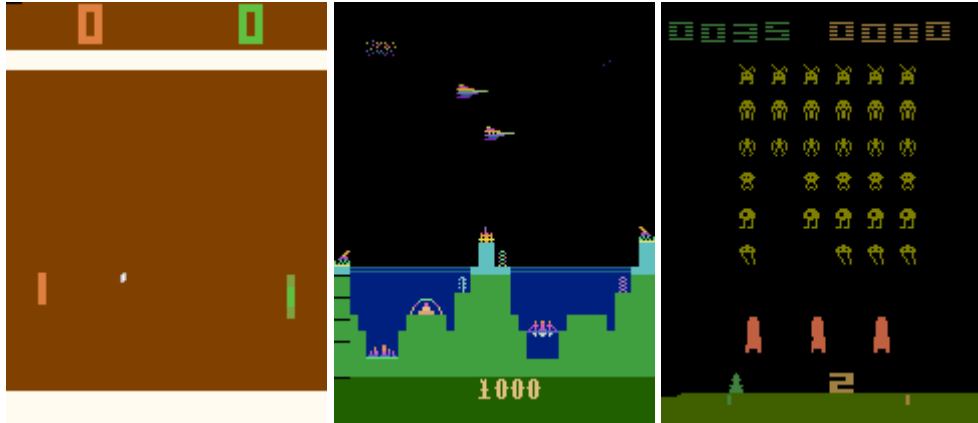


Figure 6.1: Screenshots from the Atari2600 Learning Environment

data, we obtain much better results than SARSA after 5000 episodes, although the settings are not directly comparable.

The chapter is organised as follows. Section 6.2 explains the background necessary for the rest of the chapter. In Section 6.3 we formally introduce our objective and our algorithm RLAdvice. We also bound the performance of the algorithm, finding similar guarantees to DAgger. Section 6.4 describes the experimental setup, methodology, and displays our results. In Section 6.5 we discuss the results. Section 6.6 provides information on related work. We conclude in Section 6.7 and talk about some possible future work.

6.2 Background

The setting of this chapter differs from the others, and the problem considered here requires supervised learning techniques. To this end, we provide an introduction to supervised learning, convex optimisation and online learning here.

6.2.1 Supervised learning

In the traditional supervised learning setting, the agent is provided with a *training set* D consisting of input-output tuples $(x, y) \in \mathcal{X} \times \mathcal{Y}$. The agent’s aim is to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ based on the training data that achieves good predictive performance of the outputs given the inputs on some unseen test data set. If the outputs are the labels of (a small, finite set of) categories then this is called a classification problem. If they are the values of a continuous function this is a regression problem. The function f normally transforms the raw inputs to multi-dimensional features of the problem being solved, for example in the case of the ATARI games used in this work, the features used are the Basic Abstraction of Screenshots (BASS), originally defined by Bellemare et al. (2013).

6.2.1.1 Classification

In a classification problem the agent must learn a function of the inputs to the class labels that will perform well on some hidden test set. The performance of the classifier is measured by the number of errors it makes i.e. $L(f) = \sum_{(x,y) \in D} \mathbb{I}_y(f(x))$.

The most studied form of classification is *binary*, i.e. the case where there are only two categories: positive and negative. Multiclass classification is often implemented as multiple binary classification problems. For example, given K classes a naive approach is implemented as $K(K-1)/2$ binary classifiers, with each classifier receiving samples for two of the labels. When the agent must make a decision on some data, it consults each of the binary classifiers, and assigns the class label to the class with the most positive guesses.

6.2.1.2 Regression

A regression model is traditionally parameterised by some vector $\theta \in \mathbb{R}^k$ i.e. the predictive model is $f : \mathcal{X} \times \mathbb{R}^k \rightarrow \mathcal{Y}$. In a regression problem, the magnitude of errors made by the predictor is given, as opposed to just the occurrence. The L2-regularised least squares error of a predictor f on a training set D of n instances is given by

$$L(f) = \sum_{(x,y) \in D} (f(x, \theta) - y)^2 + \frac{\lambda}{2} \|\theta\|_2^2$$

where $\lambda > 0$ is the regularisation constant. The use of an ℓ_2 regulariser with the squared loss is often called *ridge regression*. In the case that we have a *linear regression model* $f(x; \theta) = \theta^\top \phi(x)$ where $\phi : \mathcal{X} \rightarrow \mathbb{R}^m$ is the feature mapping that transforms the inputs to m -dimensional vectors, we can minimise the above loss function by choosing $\theta^* = (\Phi^\top \Phi - \lambda I)^{-1} \Phi^\top \mathbf{y}$ where $\phi_j(x_n)$ is the element in the j -th row and n -th column of Φ (alternatively, $\phi(x_n)$ is the n -th row of Φ).

The functions ϕ in the above presentation are called *basis functions* and may be non-linear. For example, a polynomial basis has $\phi_j(x) = x^j$. In continuous reinforcement learning, one often uses *radial basis functions* which satisfy the condition $\phi(x) = \phi(\|x\|)$. An example of a radial basis function is a *Gaussian basis* which has the form

$$\phi_j(x) = \exp \left\{ -\frac{(x - \mu_j)^2}{2\sigma^2} \right\}$$

where μ_j are the *centers* of the basis function. The usual Gaussian normalisation term is unnecessary since the regression model has adaptive parameters θ_j which can account for it. For a full (and probabilistic) treatment of regression in machine learning please refer to Bishop (2006).

In the following sections, we use regression to learn a model for the optimal value function, with the training outputs given to us by an oracle. We use the form of regression described above, ℓ_2 -regularised least squares linear regression, although we don't use the analytic closed form solution, since it is too inefficient to compute and store in practice.

6.2.1.3 Cost-sensitive multi-class classification

In this chapter, we will make use of a particular supervised learning problem known as the cost-sensitive multi-class classification (CSMC) problem. In this setting, there are K classes, a sample space X and a cost-function which provides a cost vector c_x whose entries are the cost for each class. The cost can be dependent on the sample $x \sim X$; i.e. each sample can have an associated cost-vector. Let D be a joint distribution over the sample space and the costs which can be factored as $D = D_X \times D_{c|x}$ where D_X is a distribution over the sample space and $D_{c|x}$ is a distribution over cost-vectors for a particular sample. The aim of a CSMC classifier $h : X \rightarrow K$ is to minimise regret, defined as

$$r_{\text{csmc}}(h) = \mathbb{E}_{x \sim D_X} \left[\mathbb{E}_{c \sim D_{c|x}} [c(h(x))] - \min_{k \in K} \mathbb{E}_{c \sim D_{c|x}} [c_x(k)] \right]$$

We will relate this formally to the value-advice problem later on; loosely, the samples x correspond to states s and the cost vector c_x corresponds to the (negative) action-value vector $-Q^*(s, \cdot)$, which are given to us by the oracle.

We will also need an error reduction of the CSMC problem to regression. The aim is to reduce the problem of finding a CSMC h to that of finding a regressor g that closely models the cost function. The regressor then provides a solution to the original problem via an arg min strategy, $h_g(x) = \arg \min_{k \in K} g(x, k)$.

We can define the least-squares error of the regressor as follows

$$e(g) = \mathbb{E}_{(x,c) \sim D} \left[\frac{1}{2|K|} \sum_{k \in K} (g(x, k) - c_x(k))^2 \right]$$

Now we can state the following theorem.

Theorem 28. *Mineiro (2010) The regret of a CSMC h_g based on a regressor g can be written in terms of the regression error $e(g)$ as follows*

$$r_{\text{csmc}}(h_g) \leq \sqrt{|K|e(g)}$$

The proof of this theorem is in Appendix A and is attributed to Mineiro (2010), although a similar result can be found in Tu and Lin (2010) for a one-sided regression error.

6.2.2 Convex optimisation

We will solve our regression problem as a convex optimisation problem using a method known as Stochastic Dual Coordinate Ascent (SDCA) by Shalev-Shwartz and Zhang (2013). SDCA has great performance for our particular objective, and we largely treat it as a black box convex optimisation routine with some modifications. We provide a brief description of convex optimisation and the SDCA algorithm in Appendix B.

6.2.3 Online learning

Online learning is a general framework used for dealing with sequential data. An agent in an online learning setting operates in rounds. In each round, the agent is given an instance (for example, a sample to be classified) which it must then offer a prediction (a class label). The solution that the agent offers is based on some hypothesis. After the agent has offered its solution, the true solution (actual class label) is revealed and the agent incurs a loss based on its hypothesis, the prediction and the true solution.

In particular, we are interested in *online convex optimisation*. Here the hypothesis space is parameterised by vectors from a convex set S . In round t , the agent picks a vector w_t from this set. The environment picks some sample (x_t, y_t) on which the hypothesis incurs a loss $l_t(w_t, x_t, y_t)$, where l_t is convex and chosen by the environment. We measure the performance of this online learning algorithm with respect to a competing set of vectors U by

$$R_T(U) = \sum_{t=1}^T l_t(w_t, x_t, y_t) - \min_{u \in U} \sum_{t=1}^T l_t(u, x_t, y_t)$$

The set U can be equivalent to S but this is not necessary.

We also use the weaker notion of *average regret* in this chapter. The average regret is defined as

$$\bar{R}_T(U) = \frac{1}{T} \sum_{t=1}^T l_t(w_t, x_t, y_t) - \min_{u \in U} \frac{1}{T} \sum_{t=1}^T l_t(u, x_t, y_t)$$

An algorithm is said to have *no-regret* if it has zero average regret asymptotically, i.e. $\bar{R}_T(U) \rightarrow 0$ as $T \rightarrow \infty$.

We have followed the presentation of Shalev-Shwartz (2011), and recommend it to the reader for further background on online learning.

6.2.3.1 Follow-the-leader (FTL)

In order to solve the online convex optimisation problem described above, the simplest algorithm is to simply consider the weights that minimise the loss on the data collected so far. For all t pick¹

$$w_t = \arg \min_u \sum_{i=1}^{t-1} l_i(u)$$

This simple algorithm is no-regret at the rate of $O(\frac{\log T}{\sqrt{T}})$ for strongly convex loss functions. In the case where the loss function is convex but not strongly convex, a modified version known as Follow-the-regularised-leader offers no-regret at the rate of $O(1/\sqrt{T})$ (Hazan et al., 2006) by adding a strongly convex regularisation penalty (on u) with a regularisation constant $\lambda_t \in \Theta(\frac{1}{\sqrt{t}})$, to the criterion for FTL.

¹Note that for the rest of this chapter we will use the convention that $\arg \min_{x \in X} f(x)$ returns a single element rather than a set, with tie-breaks performed randomly.

6.2.4 Imitation learning

Imitation learning (see Bakker and Kuniyoshi (1996) for one of the earliest survey papers) concerns the problem of imitating an expert policy, often as a proxy for performing well in an environment, since the expert policy is hopefully a good one. In the standard imitation learning setting, the expert provides the learner with a correct action for any state the learner requires. Alternate settings include having a labelled set of expert trajectories which is called learning from demonstrations (LfD, see Schaal (1997) for early work, and Argall et al. (2009) for a more recent survey). The learner has no access to the return, and simply sees the action the oracle prescribes. In practice, the distinction between imitation learning and LfD is often blurred.

In imitation learning problems, the agent may not have access to the true cost (as in reinforcement learning), and thus the task is now normally framed as minimising a *surrogate loss* based on the oracle’s policy, where the surrogate loss is defined as a distance between the each action and the optimal one, with the simplest example being a 0-1 loss. As long as the surrogate loss is an upper bound on the true cost, theoretical guarantees can be shown.

Throughout this chapter, we will refer to the Dataset Aggregation (DAgger) algorithm by Ross and Bagnell (2010). This algorithm solves the imitation learning problem by reducing it to an online learning problem. The algorithm is given in Algorithm 8. DAgger operates in iterations. It keeps a data set D which is initially empty. In each iteration i , it collects a data set D_i of tuples $(s, \pi^*(s))$ as it acts out the policy π_i , where $\pi^*(s)$ is the expert provided action for state s . After it has collected D_i it adds it to D . Then it learns a new policy π_{i+1} using the aggregated data set D .

Algorithm 8: DAgger by Ross and Bagnell (2010)

```

Initialise  $D \leftarrow \emptyset$ .
Initialise  $\hat{\pi}_1$  to any policy in  $\Pi$ .
for  $i = 1$  to  $N$  do
    Let  $\pi_i = \beta_i \hat{\pi}^* + (1 - \beta_i) \hat{\pi}_i$ 
    Sample T-step trajectories using  $\pi_i$ .
    Collect a data set  $D_i = \{(s, \pi^*(s))\}$  of visited states
    by  $\pi_i$  and actions given by the expert.
    Aggregate the data sets  $D = D \cup D_i$ 
    Train classifier  $\hat{\pi}_{i+1}$  on  $D$ .
end
Return best  $\hat{\pi}_i$  on validation.
```

DAgger differs from previous approaches to imitation learning, in that it trains successive policies on data collected according to the learner’s policy rather than the expert’s policy. At each iteration it learns how to correct mistakes (or deviations from the expert) that it made in the previous iterations, and over many iterations it builds up a “set of inputs that the learned policy is likely to encounter during its execution” (Ross and Bagnell, 2010). This is particularly

useful if the expert's policy is not present within the class of policies Π that the learner can choose from.

DAGger can be viewed as an online learning algorithm, where at each iteration i it is collecting a data set D_i that gives it an estimate of a (strongly convex) loss l_i provided by an adversary. Then, the no-regret property of follow-the-leader provides bounds on the performance of DAGger itself. We refer the reader to Ross and Bagnell (2010) for more information on DAGger. The thesis by Ross (2013) contains many modifications and applications of DAGger.

6.3 The RLAdvice algorithm

The ultimate goal of an agent in a reinforcement learning setting is to maximise expected reward. In the *value-advice problem* considered in this chapter, the oracle is an agent that performs well in the environment, and the agent must make use of the advice provided by the oracle to learn *faster* and more *data-efficiently* than it otherwise would. In the case that the oracle is perfect, and the agent can learn its behaviour then there is no more to be done; however, if this is not true the agent can use the learned behaviour as a starting point for further learning. The problem is complicated by the fact that the capabilities of the oracle and the agent may be very different, in particular in terms of the agent's ability to represent the world around it.

The *value-advice problem* is described more formally as follows. The agent acts in a standard reinforcement learning environment with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$ and rewards $r \in \mathcal{R}$. The agent has access to an oracle in a training phase. The oracle can provide the agent with the action-value vector $Q^*(s, \cdot)$ for any state, with each entry being $Q^*(s, a)$. The agent must use this information to learn a good policy that it can use in a later testing phase, where it does not have access to the oracle, and it should react quickly. Thus, the learned policy must produce an action for all possible states (complete) and must produce this action quickly (reactive). We focus on the setting where the agent is given a class of function approximators that it can use to approximate Q^* , and we use a regression-based approach to solve the value-advice problem.

The algorithm (RLAdvice) we define is a modification of the Dataset Aggregation algorithm (DAGger) (described briefly in Section 6.2.4) which is used in imitation learning in various forms. The variant of DAGger closest to RLAdvice is called DAGger with Cost-to-go, and is discussed by Ross (2013). RLAdvice varies from DAGger with Cost-to-go in the following way; rather than being given the expert's policy and needing to sample from it for different t , the algorithm is allowed to ask the expert for value-advice directly about *every action* at any timestep. Additionally, RLAdvice operates in the infinite horizon discounted reward setting, where stationary policies exist, which simplifies the algorithm. Later in this section, we show that we can derive a similar performance guarantee to DAGger with Cost-to-go. Theorem 30 below can be seen as an analogue of Theorem 4.2.1 in Ross, 2013.

6.3.1 Algorithm and performance bound

We can now solve the value-advice problem as an online convex optimisation problem, just as in DAgger. We start by defining a distribution from which states are sampled, when an agent follows a policy π .

Definition 29. The γ -discounted future state distribution $d_{\pi, s_0, \gamma}$ on $\mathcal{S}$ while following a policy π starting in state s_0 is defined as

$$d_{\pi, s_0, \gamma}(s) := (1 - \gamma) \sum_{t=0}^{\infty} \gamma^t \Pr(s_t = s \mid \pi, s_0)$$

We leave out the γ dependence and write d_{π, s_0} for brevity.

Now consider the following online convex optimisation formulation for solving the value-advice problem, that we call RLAdvice.

1. The agent is initialised with some initial regressor Q_0 which induces an $\arg \max$ policy π_0 , and a data set D that is initialised to be empty.
2. In iteration i , it follows policy π_i starting from state s_0 collecting samples $D_i = \{(s, Q^*(s, \cdot))\}$. These samples are distributed according to d_{π_i, s_0} .
3. Aggregate the data sets, $D = D \cup D_i$.
4. An adversary provides a convex loss $l_i(Q) = \mathbb{E}_{s \sim d_{\pi_i, s_0}} \left[\frac{1}{2|A|} \sum_{a \in A} (Q(s, a) - Q^*(s, a))^2 \right]$.
5. The agent picks a regressor Q_{i+1} trained on the aggregated dataset D , which defines a policy π_{i+1} with $\pi_{i+1}(s) = \arg \max_{a \in A} Q_{i+1}(s, a)$.
6. After N iterations, the algorithm returns the best policy π_i seen so far.

Let $R_{class} = \min_{Q'} \frac{1}{N} \sum_{i=1}^N l_i(Q')$ be the loss of the best average in-class regressor and $R_{regret} = \frac{1}{N} \sum_{i=1}^N l_i(Q_i) - R_{class}$ be the corresponding average online learning regret. Let $\bar{\pi}$ be the mixture policy which in iteration i selects, uniformly at random, a policy π_j for $j \in \{1, \dots, N\}$ that was previously executed by RLAdvice (one can think of $\bar{\pi}$ as the “average” policy selected by RLAdvice). We can now state the following theorem bounding the regret of the above algorithm.

Theorem 30. After N iterations of RLAdvice

$$V^*(s_0) - \max_i V^{\pi_i}(s_0) \leq V^*(s_0) - V^{\bar{\pi}}(s_0) \leq \frac{\sqrt{|A|}}{(1 - \gamma)} \sqrt{R_{regret} + R_{class}}$$

Furthermore, if a no-regret algorithm is used to pick the regressors $Q_{1:N}$ then as $N \rightarrow \infty$

$$\lim_{N \rightarrow \infty} V^*(s_0) - V^{\bar{\pi}}(s_0) \leq \frac{\sqrt{|A| R_{class}}}{1 - \gamma}$$

Before we can prove the theorem, we need the following lemma from Kakade and Langford, 2002.

Lemma 31 (Performance difference). *For all stationary policies π and π' and for all s_0 and γ ,*

$$V^{\pi'}(s_0) - V^\pi(s_0) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim d_{\pi', s_0}} [Q^\pi(s, \pi'(s)) - V^\pi(s)]$$

The above lemma measures how much better or worse π' is over π when sampling from the distribution of states induced by π' . This is exactly the quantity we wish to measure in DAgger-like approaches. We care about the performance difference between the RLAdvice policy and the optimal policy on the distribution sampled by RLAdvice.

We also need to relate the CSMC problem described in Section 6.2.1.3 to the value-advice problem. We do this by observing that a policy π can be viewed as a classifier predicting the best action for a particular state s . We formalise this below.

Assume that we have some distribution d from which we sample states $s \in \mathcal{S}$. For some state s , let $d_{Q|s}$ be any distribution over the set of all possible vectors $Q(s, \cdot)$ with mean $\mathbb{E}_{\mathbf{Q}_s \sim d_{Q|s}}[\mathbf{Q}_s] = Q^*(s, \cdot)$ (we can do this since the variance is unconstrained). For any policy π , consider the expectation (over states) of the difference between the optimal value, and the value of taking an action according to π for the sampled state, and following the optimal policy afterward.

$$\begin{aligned} \mathbb{E}_{s \sim d} [V^*(s) - Q^*(s, \pi(s))] &= \mathbb{E}_{s \sim d} [-Q^*(s, \pi(s)) - \min_{a \in A} (-Q^*(s, a))] \\ &= \mathbb{E}_{s \sim d} \left[\mathbb{E}_{\mathbf{Q}_s \sim d_{Q|s}} [-\mathbf{Q}_s[\pi(s)]] - \min_{a \in A} \mathbb{E}_{\mathbf{Q}_s \sim d_{Q|s}} [-\mathbf{Q}_s[a]] \right] \end{aligned}$$

Consider a CSMC problem with $|\mathcal{A}|$ classes, a sample space $\mathcal{S}$ and a cost function $\mathbf{Q}$ which provides the cost vector $-\mathbf{Q}_s$ defined above. Then we can view the policy π as a CSMC classifier with regret given by

$$r_{\text{csmc}}(\pi) = \mathbb{E}_{s \sim d} \left[\mathbb{E}_{\mathbf{Q}_s \sim d_{Q|s}} [-\mathbf{Q}_s[\pi(s)]] - \min_{a \in A} \mathbb{E}_{\mathbf{Q}_s \sim d_{Q|s}} [-\mathbf{Q}_s[a]] \right]$$

By the reduction to regression in Theorem 28, we know that if we have a regressor $\hat{Q}$ which models the cost function Q , then we can state the regret of the arg min classifier $\hat{\pi}$ in terms of the error of the regressor. More formally, the following lemma holds.

Lemma 32. *If $\hat{\pi}$ is defined as follows,*

$$\begin{aligned} \hat{\pi}(s) &= \arg \min_{a \in \mathcal{A}} (-\hat{Q}(s, a)) \\ &= \arg \max_{a \in \mathcal{A}} \hat{Q}(s, a) \end{aligned}$$

then $r_{csmc}(\hat{\pi}) \leq \sqrt{|\mathcal{A}|e(\hat{Q})}$ where

$$\begin{aligned} e(\hat{Q}) &= \mathbb{E}_{s \sim d} \mathbb{E}_{\mathbf{Q}_s \sim d_{\mathbf{Q}|s}} \left[\frac{1}{2|\mathcal{A}|} \sum_{a \in \mathcal{A}} (\hat{Q}(s, a) - [\mathbf{Q}_s[a]])^2 \right] \\ &= \mathbb{E}_{s \sim d} \left[\frac{1}{2|\mathcal{A}|} \sum_{a \in \mathcal{A}} (\hat{Q}(s, a) - Q^*(s, a))^2 \right] \end{aligned}$$

Proof of Theorem 30. Since $V^{\bar{\pi}}(s_0) \leq \max_i V^{\pi_i}(s_0)$ we have that $V^*(s_0) - \max_i V^{\pi_i}(s_0) \leq V^*(s_0) - V^{\bar{\pi}}(s_0)$. The second inequality is proved as follows,

$$\begin{aligned} V^*(s_0) - V^{\bar{\pi}}(s_0) &= \frac{1}{N} \sum_{i=1}^N (V^*(s_0) - V^{\pi_i}(s_0)) \\ &= \frac{1}{N} \sum_{i=1}^N \frac{1}{1-\gamma} \mathbb{E}_{s \sim d_{\pi_i, s_0}} [V^*(s) - Q^*(s, \pi_i(s))] \\ &\leq \frac{1}{N(1-\gamma)} \sum_{i=1}^N \sqrt{|\mathcal{A}|} \sqrt{l_i(Q_i)} \\ &\leq \frac{\sqrt{|\mathcal{A}|}}{(1-\gamma)} \sqrt{\frac{1}{N} \sum_{i=1}^N l_i(Q_i)} \\ &= \frac{\sqrt{|\mathcal{A}|}}{(1-\gamma)} \sqrt{R_{regret} + R_{class}} \end{aligned}$$

The first equality is by the definition of $\bar{\pi}$. The second equality is by the performance difference lemma (Lemma 31). The first inequality is by the reduction of a CSMC classifier π_i with regret $r_{csmc}(\pi_i) = \mathbb{E}_{s \sim d_{\pi_i, s_0}} [V^*(s) - Q^*(s, \pi_i(s))]$ to a regressor Q_i with error $l_i(Q_i)$ by Lemma 32. The last equality is by the definitions of R_{regret} and R_{class} .

Assuming we have an infinite number of samples at each iteration i , we are collecting data in D_i that gives us an accurate estimate of the loss $l_i(\hat{Q})$. The loss $l_i(\hat{Q})$ is convex but not strongly convex. Thus, if follow-the-regularised-leader (with an appropriate strongly convex regulariser) is used to pick the sequence of regressors $Q_{1:N}$, we have that $R_{regret} \rightarrow 0$ as $N \rightarrow \infty$ as required. $\square$

In Theorem 30 we bound the difference between the optimal policy and the best policy given by RLAdvice, in terms of the value of a starting state s_0 . We proved this result for a single starting state s_0 , but the result can be extended to a distribution over all (starting) states. This bound does not mean that we converge, the algorithm could conceivably oscillate between approximations. If the class of regressors can approximate the optimal value function well, then the oscillations will be between good approximators and therefore not a problem. A problem occurs only if the class of approximators is weak.

6.3.2 Practical algorithm

The RLAdvice algorithm we use in practice is given by Algorithm 9. We use a class of linear functions as our regressors, i.e. $\hat{Q}(s, a) = w^T \phi(s, a)$ where $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$ are d -dimensional features over states and actions and w_d is a d -dimensional weight vector. We further approximate the setting and use $\phi(s_t, a) := \phi(s_t)$, that is we learn a separate weight vector for each action a . We add tuples of the form $(\phi(s_t), Q^*(s_t, a))$ to the dataset D_a for all a and for each timestep t in the episode i , where $Q^*(s_t, a)$ is provided by the oracle. In the first episode, the agent follows the policy provided by the oracle, i.e. $\arg \max_a Q^*(s, a)$. For every following episode, the agent acts based on its own prediction which it obtains from the current (action) value function approximation $w_i^a \top \phi(s)$, and adds the corresponding tuples to D_a . The intuition behind using the agent's own policy over the oracle's is that the oracle policy might not be representable in the agent's approximation space and the agent might make mistakes that the oracle never makes. Due to this representation issue, the agent might not learn to avoid end up in certain states, which the oracle would not visit but would know how to act in. In order to learn about these situations, we use the agent's policy. Once an episode is over, the agent learns a new set of regression weights (w_{i+1}^a) for each dataset D_a , using the following objective optimised by Stochastic Dual Coordinate Ascent (SDCA).

$$w_{i+1}^a = \arg \min_v \left(\sum_{(\phi(s), Q^*(s, a)) \in D_a} (v^\top \phi(s) - Q^*(s, a))^2 + \frac{\lambda}{2} v^\top v \right) \quad (6.1)$$

λ is the regularisation constant. For the next episode $i + 1$, the agent follows the policy $\pi_{i+1}(s) = \arg \max_{a \in \mathcal{A}} w_{i+1}^a \top \phi(s)$.

Algorithm 9: Reinforcement learning with value advice

```

Initialise  $D \leftarrow \emptyset$ .
Initialise  $\pi_1 (= \pi^*)$ .
 $t = 0$ 
for  $i = 1$  to  $N$  do
    while not end of episode do
        foreach action  $a$  do
            Obtain feature  $\phi(s_t)$  and oracle expected return  $Q^*(s_t, a)$ .
            Add training sample  $\{\phi(s_t), Q^*(s_t, a)\}$  to  $D_a$ .
        end
        Act according to  $\pi_i$ .
    end
    foreach action  $a$  do
        Learn new model  $\hat{Q}_{i+1}^a := w_{i+1}^a \top \phi$  from  $D_a$  using regression.
    end
     $\pi_{i+1}(\cdot) = \arg \max_a \hat{Q}_{i+1}^a(\cdot)$ .
end

```

Table 6.1: Experimental Parameters for all algorithms

Setting	Environment	Parameter	Value
All	All	γ	0.999
ALE	All	Environment Distribution	False
	All	Frame skip	5
UCT	All	Exploration constant	0
	All	Number of simulations	36
	All	Horizon	50
BASS features	All	Grid width	16
	All	Grid height	14
	All	Colours	8
RLAdvice	Pong	Regulariser	2000
	Pong	stopping gap	0.0001
	Space Invaders	Regulariser	1000
	Space Invaders	stopping gap	10
	Atlantis	Regulariser	1.0
	Atlantis	Stopping gap	5000
Dagger	Pong	Regulariser	0.0001
	Space Invaders	Regulariser	0.001
	Atlantis	Regulariser	0.001
SARSA	All	λ	0.9
	All	ϵ	0.05
	Pong	α	0.5
	Space Invaders	α	0.5
	Atlantis	α	0.3

6.4 Experiments

Firstly, we show that using the oracle’s own trajectory does not work well. RLAdvice using the oracle that provides the return works very well, providing the best known results *on this class of linear function approximators* for Pong and Atlantis. This result demonstrates that the class is capable of representing a good Pong playing policy. On the other hand, we see no improvement over SARSA in the Space Invader results, which might indicate that the problem here is the feature representation. We can thus see the diagnostic use of this approach.

The Oracle As pointed out previously, our primary motivation is in extracting efficient reactive policies from slow MCTS algorithms such as UCT, which can be computationally expensive to use outside of a training phase. The oracle that we use is therefore UCT with a specified horizon and number of simulations. As an aside, note that UCT itself can be viewed as a reinforcement learning algorithm in a deterministic environment in the following sense. It is possible to mimic the ability to reset to a particular state, by saving in each node,

the action sequence that led to that node. Then in order to evaluate a child of a particular node, the agent simply has to reset to the start of the game either by completing the current episode or by using an *end game* action, and then play out the saved actions, followed by the action of the child it wishes to evaluate. We provide this replaced emulate function in Algorithm 10. Given that the ALE is a deterministic environment, the impressive UCT results of Bellemare et al. (2013) can be said to be reinforcement learning results (albeit with a much higher number of trajectories used) rather than planning results and this brings into critical light the weak SARSA results provided in that paper. It is important to remember that even in this “reinforcement learning” mode, UCT still does not learn either an explicit or complete policy, but just an action sequence. By testing our algorithm on the ALE with UCT as our oracle in similar configurations to that used in Bellemare et al. (2013) we are also trying to discover whether the fault lies with SARSA or with the function approximation class.

Algorithm 10: The modified **emulate** function for UCT as an RL agent

Input: MCTS Node containing action sequence $a_{1:l-1}$, Environment Env , Action a

Output: Reward for the execution of action a , after sequence $a_{1:l-1}$

// The following loop can be replaced by a call to
reset_game() if available.

while not end of game **do**

 | Execute random action in Env .

end

Start new episode.

Execute all actions in Node action sequence $(a_{1:l-1})$.

Execute latest action a in Env and store reward in r .

if End of game **then**

 | Set node to leaf node.

end

return r

Features We use the feature class described by Bellemare et al. (2013) as Basic Abstraction of Screen Shots (BASS). It consists of a tiling of the screen into blocks, with each block containing indicator functions for each SECAM colour i.e. the feature is 1 if a particular colour was present in the block. BASS consists of these features along with the pairwise AND of all those features. In the default setting of a 16x14 grid this results in a feature space that contains 1,606,528 features.

Games The legal action set for a game in the ALE contains all 18 actions that could physically be pressed on an ATARI2600 controller. The minimal action set contains only the actions that are needed for a particular game. For computational reasons, we selected games that have a minimal action set with size less than the number of legal actions (18). We also wanted games that showed better than random performance on SARSA (on the BASS feature set), as some indication of a linear function approximator being successful. Given those constraints, we chose Pong and Atlantis. We also chose one game, Space Invaders, where picking a constant

action with some ε -random actions performed better than SARSA. There is some indication here that the function approximator cannot represent the value function, and we attempt to confirm that with our experiments. In order to save on computation time and memory usage we limited the episodes to 5000 frames.

Pong is a game with two paddles and a ball, a 2D version of table tennis. The aim is to hit the ball back such that the opponent cannot reply. Our agent plays against the hard coded ATARI 2600 agent, which is hard to play against even for a human. If the agent scores it receives 1 point, and -1 point when the opponent scores. Thus the total score is the difference between the agent's score and the opponent's score. The minimal action set contains 6 actions.

Space Invaders involves shooting down columns of alien spaceships, while avoiding their return fire. The enemy spaceships are arranged in columns, with lower rows worth lesser points. The columns move from left to right and then back, with each movement to the end advancing the spaceships further down the screen. The agent also has the option of occasionally shooting down a special purple fighter for an extra 200 points. The game ends when the agent loses 3 lives, or when the moving columns of spaceships get to the bottom row. The minimal action set contains 6 actions.

Atlantis is also a shooter, but here the enemy spacecraft fly across the sky very quickly. The agent is in charge of three fixed guns, a primary central one and two secondary guns on the sides and is tasked with protecting the city of Atlantis. The agent must shoot down as many enemy spacecraft as it can. The enemy spacecraft also occasionally attempt to use lasers to take out the agent's 3 guns and 4 other structures of Atlantis. Lost structures, including the guns, can be regained by destroying enough enemy spaceships. The agent loses when all structures are destroyed. The minimal action set contains 4 actions.

Methodology A trial consists of running the chosen algorithm for $N = 100$ episodes. For each environment we do 5 trials, and our graphs show these results with error bars suppressed for clarity. We use SDCA to learn linear regression models after each iteration based on the data accumulated so far. We chose regularisation constant λ based on trials on a small set of episodes in each environment, selecting from $\{0.001, 0.01, 0.1, 1.0, 10.0, 100, 1000\}$. We also decrease the impact of the regulariser linearly in N . We warm start SDCA with the agent's previous solution to speed-up the learning procedure.

Comparisons We compare the following algorithms on Pong, Space Invaders and Atlantis within the ALE framework. Note that we use the minimal action set for each environment. This gets rid of superfluous actions for each environment. It also means that our experiments are computationally less demanding both in memory and time, since we learn a model for each action independently. The SARSA results are taken from Bellemare et al. (2013). Note that RLAdvice-best, UCTRLA-best and DAgger-best show the best total reward in any episode so far for RLAdvice and UCTRL.

- SARSA (traditional reinforcement learning) [SARSA]².
- RLAdvice using regression. We have the following variations,
 - Training starting with a UCT policy and then iterating the model [RLAdvice].
 - Training only with the UCT policies [UCTRLA].
- RLAdvice using classification [DAGger]. This is a version of DAGger, with parameters $\beta_1 = 1$ for the first iteration and $\beta_i = 0$ for all other iterations i . The classifier used is an L2-regularised linear SVM from the library LIBLINEAR by Fan et al. (2008).

We also provide the average of the best reward received in the iterations so far as RLA-best, UCTRLA-best and DA-best.

6.5 Discussion

It seems intuitively clear that the class of function approximators we are using should be able to represent the value functions of good Pong policies. We have pairwise features of grids of SECAM colours. There are only three objects in the Pong domain, the two paddles and the ball. The value function of states where the ball is approaching the agent can be represented by the pairwise function of the agent's paddle and the ball position.

On Space Invaders we expect the various agents to perform about equally. Our hypothesis is that the feature space is not adequate, as SARSA performs on par with a random agent. On Atlantis, we hope to see much improvement over SARSA. In general we expect that RLAdvice performs better than UCTRLA and DAGger. UCTRLA does not provide adequate information about mistakes, and DAGger has less information than RLAdvice, since it only makes use of what the correct action was, not the action-values themselves.

6.5.1 General behaviour

We will focus on the behaviour of the agent in the Pong domain, since it is clearest to see when the agent is performing very well. Additionally, the Pong playing agent exhibits interesting characteristics. An examination of the Pong behaviour requires observation of the video of an RLAdvice agent playing Pong. <http://mdaswani.me/rlavideos/> contains a playlist of videos of the agent playing various games, including Pong at different stages during training. The first thing we notice is the jitteriness of the agent (green, on the right). This can be explained by the behaviour of the oracle UCT. Before the ball is very close to the agent, all actions are

²In the original paper (Daswani et al., 2014), we also tested improving the policy of RLAdvice using SARSA on the learned Q-values. Unfortunately, this provided no improvement (or made the policy worse) so we did not run the experiment again for the extended iterations provided in this thesis.

nearly equal in reward, since the agent can always reach the position it needs quite quickly. So the oracle acts randomly. The agent inherits this jittery behaviour.

The agent makes mistakes when the data collected from previous episodes is insufficient to learn a good value function approximation. As it makes mistakes, it collects data about the values for the actions in those situations which improves the approximation in those areas. This analysis also illustrates why learning from the oracle policy alone is not satisfactory. If one looks at the video for a UCT agent playing Pong, the agent wins nearly all the time, so the opponent does not even get to serve, or gets to serve only a few times. Thus learning from this policy does not provide the agent with any data on how to return a serve, and so UCTRLA fails on this task.

On Space Invaders, we have some intuition for why the value function may not be representable by a linear combination of the pairwise BASS features. Space Invaders is fairly chaotic, and has the same colour for shots fired by the enemy spaceships and for the agent's own. This makes it hard to distinguish between the two, even though the ALE provides a colour averaging between every two frames. The much higher density of objects in the domain compared to Pong, also makes it harder for the objects to be clearly defined, resulting in feature vectors that look very similar for fairly different situations (such as being hit and being missed by a laser beam).

6.5.2 RLAdvice vs SARSA

The hypothesis is that RLAdvice should do better with much fewer episodes than SARSA, since it has the advice information for each episode. On Pong, SARSA using 5000 training episodes achieves an average reward of -13. It is nearly the worst performing method; UCTRLA is marginally worse. On Atlantis, it is the worst performing method closely followed by UCTRLA. On Space Invaders, SARSA actually performs identically to all other algorithms; lending credence to the hypothesis that the BASS feature set cannot represent this domain.

6.5.3 RLAdvice vs UCTRLA

As pointed out above UCTRLA performs very badly on all three domains. UCTRLA-best does perform better than SARSA, indicating that the agent can have occasional good performance due to the exploration provided by the optimal oracle. However, it is clear that following the oracle policy is not a useful learning tool when using function approximation classes that can't exactly represent the true value function. On Space Invaders, all algorithms perform equally badly, so while UCTRLA and SARSA are of similar performance, the result is essentially meaningless.

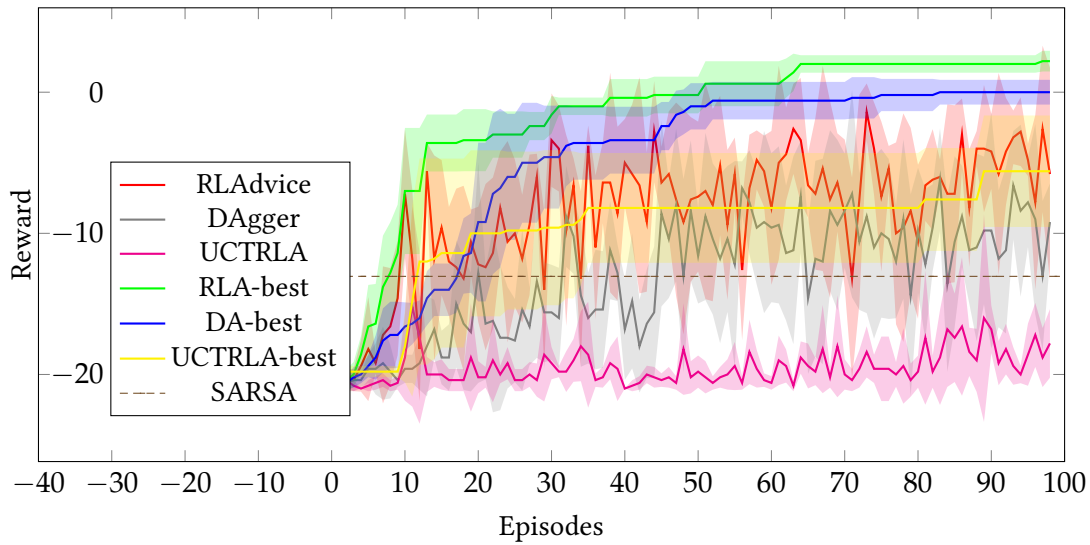


Figure 6.2: Comparisons on Pong using 100 episodes

6.5.4 RLAdvice vs DAgger

On Pong, RLAdvice outperforms DAgger with RLAdvice-best achieving an average maximum of 2.2 versus the 0.0 achieved by DAgger-best. A similar performance difference is also seen in the average case, UCTRLA-best outperforms DAgger but not RLAdvice.

On Atlantis, DAgger performs the same as RLAdvice, while DAgger-best is marginally better than RLAdvice-best although this is well within the margin of error. This is somewhat surprising, it seems that the additional value information is not beneficial on Atlantis. One possible reason could be that the agent approaches peak performance relatively easily on this domain, UCT itself scores approximately 45000, compared to the 35000 achieved by DAgger and 33000 by RLAdvice. It is possible that this is an effect caused by the limiting of the domain to 5000 frames. Hard events, which include a spacecraft flying across the sky worth several thousand points occur much more often after 5000 frames into the game.

DAgger takes much longer computationally, most likely due to the different optimisation technique used in LIBLINEAR. On Atlantis we see that it takes 95.9 hours instead of the 72 hours of RLAdvice. which is about an extra day of computation time.

6.5.5 Summary of comparisons

From the above discussion we see that the expected performance of each algorithm, as outlined in Section 6.5, holds true except on Atlantis, where DAgger performs equally as well as RLAdvice. We note that the deciding factor on whether any of these algorithms performs well is the ability of the function approximation class to represent the action-values of the optimal policy within the environment. The average reward of the RLAdvice-best policy on

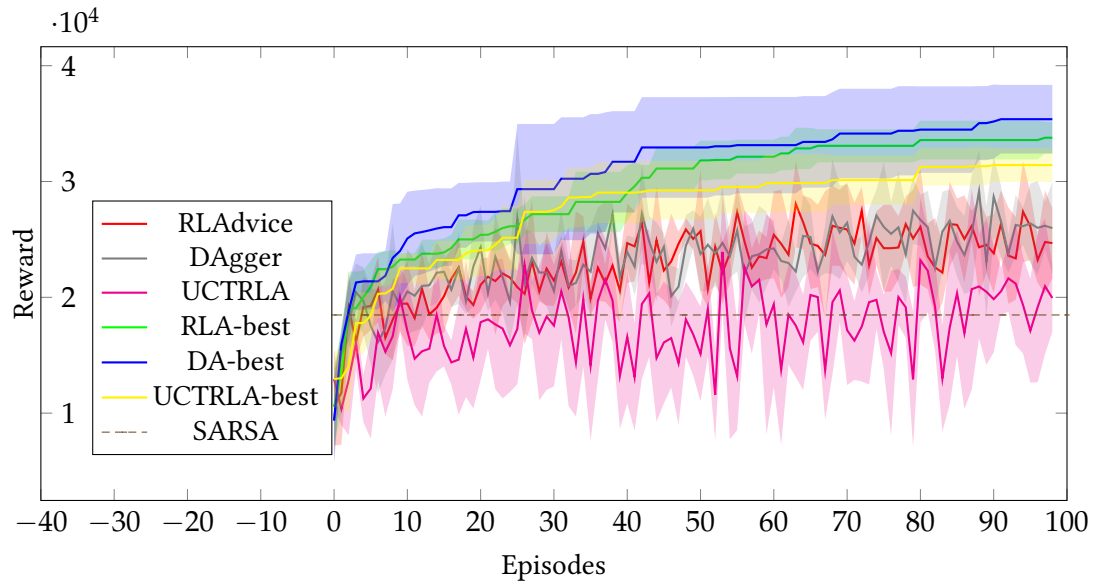


Figure 6.3: Comparisons on Atlantis using 100 episodes

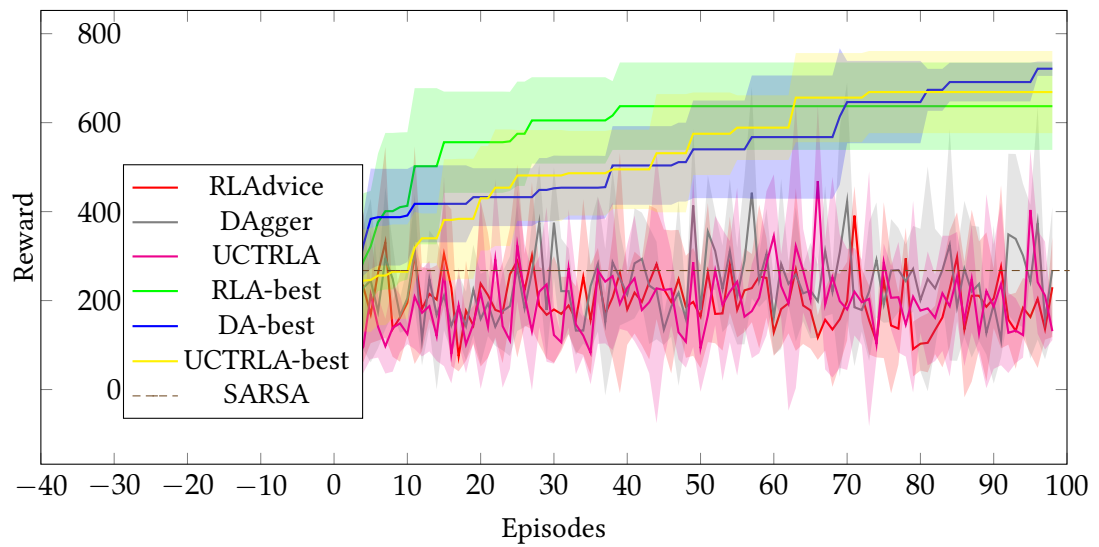


Figure 6.4: Comparisons on Space Invaders using 100 episodes

Pong is the best reported to date on this class of function approximators (BASS) found in the literature³, indicating that the failure of SARSA to learn on this domain, using these features, was not one of representation but of learning.

6.5.6 Oracle accuracy

An issue that we need to take into account, is the accuracy of the estimates of the action-values provided by the oracle (UCT). There are two sources of error. One stems from the number of simulations that we use to sample. The other is the UCB formula itself, which does not select actions which seem to not have a high value within a few iterations. While this exploration-exploitation strategy is useful when acting well in an environment (and indeed solves it in the bandit setting), in our setting we need estimates for all actions, not just the good ones. Pruning the amount of simulations spent on bad actions for the oracle policy means that we have good value estimates for the good actions. The UCB formula means that we will underestimate the value of the bad actions, which is better than overestimation since it gives us a better margin for error.

6.5.7 Stochastic Environments

Though the experiments were performed on a class of deterministic environments, RLAdvice has no dependence on the determinism of the environment. In the case of randomised initial state in ALE, we can also use the policy trained on the deterministic environment since it generalises according to the linear function approximator. Thus, it is much more versatile than simply learning a fixed trajectory.

6.5.8 Computational issues

We can look at the computation time taken in various stages of the algorithm. The experiments were primarily performed on a Intel Xeon X5650 (2.67 Ghz) with 12 cores and 141GB of RAM. Consider Space Invaders, in which the worst time to perform regression for a given action is 30 minutes, and the average is not much better around 20 minutes. Thus in 30 runs, using 6 actions the regression part cumulatively contributes 5400 minutes to the running time. UCT on the other hand takes 40 minutes cumulatively per episode (approximately 3 seconds for one call to UCT), meaning over 30 episodes this is around 1200 minutes. Other games have similar computational profiles. The learning scales linearly with the number of possible actions, which adds a severe constraint on the number of actions we can learn the value of using this method. On the other hand, if we can speed up the regression process, we can significantly improve the running time of the procedure.

³This statement is true as of 18th August 2015.

Table 6.2: Timing data on Atlantis, the most computationally expensive game, using LIBLINEAR for RLAdvice and UCTRLA. The time for RLAdvice includes generating the advice using UCT. The time for SARSA-RLA here is post-RLAdvice training and would be comparable to time taken by standard SARSA.

Algorithm	Training episodes	Training time	Testing time (per episode)
SARSA	5000	22.01 hours	<1 second
RLAdvice	100	72 hours	<1 second
Dagger	100	95.9 hours	<1 second
UCTRLA	100	48.1 hours	<1 second
UCT	N/A	N/A	3600 seconds

In terms of memory, the usage is substantial. A single run of 100 iterations for Atlantis could come to a little over 30 GB of RAM on average. This memory is used in storing the feature vectors for every screen visited, along with the current model for each action. The models are approximately 200MB each, which is a small fraction of the total. The models essentially contain a weight for each feature stored in a sparse format. Even though the sparsity of feature vectors is high (around 1%), given 1.6 million features and trajectories in the order of a few thousand frames is enough to add up to a large amount of RAM for each iteration. In particular the features for Atlantis are less sparse than the other games, for example Pong uses under 10GB of RAM.

6.6 Related Work

We have already discussed the relation between RLAdvice and Dagger above. The (other) closest related work is by Guo et al. (2014) which was published after we submitted (the final version of) the paper (Daswani et al., 2014) from which this chapter is derived. They tested both regression and (multinomial) classification settings, with neural networks instead of linear function approximators to approximate the Q-values (or learn a softmax for classification). They focused primarily on learning from (static) trajectories already sampled by UCT, which has obvious advantages in terms of speed, but could have other issues since the training distribution may not match the distribution induced by the resulting evaluation policy. Their algorithm UCTtoClassification-Interleaved comes closest to the method Dagger, although they only perform the interleaving once every 200 runs and Dagger performs this interleaving for every episode. Thus Dagger may be more data efficient but at a significant computational cost. Their results are striking; they outperform even the superhuman performance of Mnih et al. (2015) on the ATARI games they selected even using the non-interleaved methods. It seems that the use of neural networks as function approximators allows the optimal policy to be completely represented, which alleviates any problems with matching the input and evaluation distributions. Another difference in results is that they observe that their non-interleaved

classification methods outperformed their non-interleaved regression method. We observe the reverse in our (interleaved) RLAdvice and DAgger. However, it should be noted that they did not test an interleaved regression method which would be closest to our algorithm RLAdvice.

Approximate Policy Iteration We examined policy iteration in Chapter 2. Approximate versions of policy iteration are often used when the MDP being considered is very large and/or continuous. As the name implies, in API the learned policies at each stage of the algorithm are approximations of the Bellman operator improved policies (and thus API loses the guarantees of exact policy iteration).

This chapter could nearly be viewed as an approximate policy iteration method with the evaluation of the value function being performed via UCT. The key difference is that in standard approximate policy iteration only data collected using the last policy is used to iterate the new one. However, here we follow a Dataset Aggregation approach and use all the data collected so far.

Lagoudakis and Parr (2003) have an approximate policy iteration algorithm which makes use of classification to learn the new policy at each iteration. It even makes use of Monte-Carlo rollouts to evaluate the action-value function, so in this sense it is very close to our work where we use UCT as the oracle. However the rollouts it uses are of the value function for the current policy, whereas our oracle provides the value of the optimal policy.

Previous work on reinforcement learning with advice There has been interest in various formulations of the reinforcement learning with advice problem. Maclin and Shavlik (1996) define a learner that can accept advice in the form of instructions in a simple imperative programming language. Wiewiora et al. (2003) define potential-based advice which uses shaping functions defined over states and actions to give the agent hints about whether a state-action pair is good or bad. Maclin et al. (2005) construct agents that can accept advice in the form of bounds on the Q-value in certain parts of the state-action space. Azar et al. (2013) look at regret bounds for the case where the agent is given advice in the form of some set of (hopefully good) input policies. Taylor et al. (2014) define a teacher-student framework where both teacher and student are reinforcement learning agents, and the teacher must choose when to give advice in the form of recommended actions to the student. The advice is assumed to be budgeted. This work is of interest to us, since the experiments are also performed on video games (Starcraft and Pacman). However, none of these various advice settings address our particular problem.

Previous work on the ALE Also of interest to us is previous work on the ALE. The initial paper by Bellemare et al. (2013) extensively described the performance of a vanilla SARSA implementation using features based on the pixel matrix and linear function approximation. These agents do not perform so well. However, later papers use other function approximators that perform much better. For example, Hausknecht et al. (2013) use neuro-evolutionary

techniques and Mnih et al. (2013) use convolution neural nets in a deep learning style, to learn features of the matrix. There has also been work in the model-based setting by Bellemare et al. (2013) but these agents have so far only been used for prediction rather than control, since it is still quite computationally difficult to find a good policy using a model in such a large space. In our work, we focus on trying to improve the performance on a particular feature set (BASS) using only linear function approximation.

6.7 Conclusion

We introduced a modification of the DAgger agent for the reinforcement learning with advice problem. RLAdvice can be used to find explicit policies for anytime algorithms such as UCT, and for checking the usefulness of a function approximation class. It shows improved performance on the Pong and Atlantis domains in the Arcade Learning Environment indicating that value functions of good policies are representable in the class, and similar bad performance to SARSA on Space Invaders which suggests a problem with the function approximation class in representing this domain.

Future Work As pointed out in the paragraph on computational issues, the computation time is dominated by the model learning. A suitable next step would be to consider a budgeted advice setting where the agent must limit the number of calls it makes to the oracle. This would save time via less UCT computations as well as a smaller number of samples which results in quicker model learning.

Another natural next step would be to combine automated feature discovery techniques (such as in Chapter 5 with the RLAdvice agent to find classes of function approximators that can best represent the policy of an oracle.

The current work treats the oracle as a black box. However, instead of the oracle-learner framework that we have here, it might be better to examine a teacher-learner setup as in Taylor et al. (2014), where the teacher is attempting to optimise the policy it shows to the agent. RLAdvice corrects flaws in its own policy, however this can still fail as seen in the Pong example, where not enough data about a serve from one side of the screen results in suboptimal behaviour. This could be prevented if the teacher was able to predict this failure and show the learner the right data for the learner's function approximation class.

Conclusion and outlook

“ Any AI smart enough to pass a Turing test is smart enough to know to fail it. ”
Ian McDonald, River of Gods, 2004.

New insights

This thesis presented a set of improvements to the current state of generic reinforcement learning.

- In Chapter 4 we offered a solution to the problem of representing domains with *long-term dependencies*. We used the class of *looping suffix trees*, extending the work of Holmes et al. (2006) from deterministic POMDPs to the FRL setting. We also showed that LSTs could exactly represent a subset of stochastic POMDPs known as hPOMDPs.
- In Chapter 5 we presented an alternative cost function ($Cost_{QL}$) to the initial proposals by Hutter (2009b) and Nguyen (2013) which is *value-based*, can be evaluated *model-free* and can thus be extended to the *function approximation* setting. $Cost_{QL}$ is also more discriminative; it aims at finding exactly those features necessary in approximating the value function of the optimal policy. Furthermore, we proved its consistency with respect to a certain class of bounded finite state machines. We showed empirically that the resulting algorithm (FA)hQL performed equally well to the original model-based cost, and did not require any ad-hoc modifications to run on large domains such as POCMAN.
- In Chapter 6 we presented an algorithm RLAdvice derived from Dagger, an imitation learning algorithm, that can be used to extract efficient reactive and complete policies from anytime algorithms such as UCT, in the full-information setting. It can also be used to test the ability of a function approximation class to represent the optimal value function in a given environment.

Discussion and Future work

Throughout this thesis we have highlighted various problems that generic reinforcement learning agents face: perceptual aliasing, exacerbated exploration vs exploitation, the difficulty of learning a good representation for a policy that the agent does not yet have, identifying and representing long-term dependencies, computational issues, and others. We do not claim to have fully solved any of these issues, but do hope to have offered insights and partial solutions to some of them. We believe that the long-term dependency domain TMaze is solved completely by using looping suffix trees; new algorithms should use other test domains in addition to TMaze to demonstrate progress. We believe, particularly in the light of Hutter (2014), that a value-based cost has great benefits in terms of capturing the exact quantity we wish to represent. We are unsure whether $Cost_{QL}$ is *the* cost function we need to optimise. We also are unsure about whether a final solution to the generic reinforcement learning problem will be model-free, given the data-efficiency of model-based approaches, along with their greater ability to generalise.

The biggest problem still facing feature reinforcement learning is the reliance on a stochastic search method. In this thesis, we treated this as a black box, and did not place too much time or effort in optimising the insides of this box. Nguyen (2013) examined a more sophisticated version of simulated annealing called *parallel tempering* but the results in this thesis weakly indicate that it did not provide much (or any) improvement over simple simulated annealing with appropriately tuned parameters, though it helps make the algorithm more parameter free. The *context tree maximising* algorithm provided in Nguyen et al. (2012) is an FRL method with an analytic solution to the optimal context tree, but cannot be easily generalised to other map classes and cost functions.

A big problem that current FRL techniques have, is the need to store an *uncompressed* version of the history. This can be very problematic for complex environments. For example, in the ALE environment, an FRL agent needs to store a history of all frames (pixel matrices) seen so far, if implemented naively. Even if better representations are used, this can be very memory intensive.

Feature reinforcement learning is solving a complex optimisation problem. In general, an optimisation problem with no constraints on the function class is impossible to solve. For example, finding a minimiser of such a function on the real numbers is intractable; it is a needle in an uncountably large haystack. Lipschitz continuous functions are one well-behaved class for which there always exists a computable way to find an ε -close minimiser using an ε -grid search. Ideally, any function that we want to minimise would have nicer properties like convexity to which we could then apply existing machinery. The cost functions defined for Φ MDP so far do not have such properties on the map classes we define. In order to bring notions such as convexity to the domain, we want our map classes to be more well-behaved with respect to the cost. For example, it is clear that the neighbourhood function we define for looping suffix trees is not ideal, since the cost can change very quickly between two neighbours.

We have focussed on partially observable MDPs in this thesis, and indeed the literature for environments more general than POMDPs is scarce. There is a jump from POMDPs to solving all computable environments, see Hutter (2005) for example. There is room in between POMDPs and all computable environments. PSRs (Littman et al., 2001; Singh and James, 2004), U-tree (McCallum, 1996) and FRL live in this space. New research could look further into pushdown automata which offer the ability to count, extending the range of applicable domains very far; indeed planning in these domains will start to approach modern general planning techniques.

Feature reinforcement learning in the age of deep learning

The newest evolution of neural networks has taken the machine learning community by storm. Deep learning seems to accomplish many tasks that previously took a lot of expert knowledge. Many areas of natural language processing and computer vision have seen significant improvements in the last 5 years due to deep learning.

Deep learning offers a way to automatically learn features that are useful for a particular task, usually supervised, but also occasionally unsupervised. The recent Nature paper (Mnih et al., 2015) from Google Deep Mind showed super human performance on several ATARI games using deep reinforcement learning (with the algorithm DQN). In the light of these (and other) results, do we need feature reinforcement learning as proposed by Hutter (2009b)?

It would be easy to say that deep reinforcement learning is a form of feature reinforcement learning. This may be true in some general sense, in that they are both used for learning useful features of the environment, but the two differ in implementation and in philosophy. FRL was intended as a principled framework for extracting useful features from very general environments. DQN is in a high-level sense, a simple (and impressive) neural net approximation of the Q-values combined with a simple learning algorithm (Q-learning), with better engineering and effective use of new and old tricks (for example, experience replay). DQN does not have necessarily have the ability to perform well in partially observable environments; the ATARI environments become fully observable when the last four frames are provided to the agent. However, this is being addressed by extensions such as Hausknecht and Stone (2015).

There are probabilistic interpretations of neural networks (Bengio, 2000), that could enable their use as part of the FRL framework. With $Cost_{QL}$, we may not even care about the interpretation, only the ability of the neural network to model the Q-values. The NEAT framework by Stanley and Miikkulainen (2002) and follow-up work (most recently by Hausknecht et al. (2013) on the ALE) use genetic algorithms to evolve neural network topologies *along with* the weights. FRL could be viewed as an alternative to NEAT, using neural networks as the function approximation class and finding better topologies using a global stochastic search method. In the end though, it may be true that there is no need to evolve topologies when a sufficiently large neural network might be able to perform equally well.

In summary, while the practical applications are still unclear, theoretical advances in FRL, such as Hutter (2014), can greatly advance other areas of reinforcement learning.

Personal philosophical concerns

We are seeing another boom in AI at the moment. Over the last decade, the adoption of machine learning by major technology companies like Google, Facebook, Yahoo and Microsoft along with ambitious AI startups like Deep Mind (now owned by Google), Vicarious, NNAISense and others, indicates that funding, at least in the private sector, is approaching an all-time high.

There has also been recent awareness of the possibility that general AI poses an existential risk. A recent article¹ signed by Stephen Hawking, Max Tegmark, Stuart Russell and Frank Wilczek, arguably all giants in their respective fields, said the following about the possibility of a machine with superhuman intelligence arising.

“ One can imagine such technology outsmarting financial markets, out-inventing human researchers, out-manipulating human leaders, and developing weapons we cannot even understand. Whereas the short-term impact of AI depends on who controls it, the long-term impact depends on whether it can be controlled at all.

”

Good (1965) proposed that there could be an “intelligence explosion” once AI has reached a certain threshold of intelligence, since such an AI would be able to build another, more intelligent AI in a shorter period of time than was needed for its own creation. Nick Bostrom’s recent book *Superintelligence* (Bostrom, 2014) offers great insight into the scenarios that may occur should this happen. The orthogonality thesis (Bostrom, 2012) argues that intelligence and (the agent’s) utility are orthogonal systems; a very intelligent agent might optimise for a completely pointless (to humans) utility function such as the infamous (and self-descriptive) “paperclip maximiser”. Even with the best intentions of human programmers, the agent might still behave in unexpected and unwanted ways. Bostrom also argues that any sufficiently intelligent AI will pursue certain “convergent instrumental goals” that are necessary to fulfil any utility function (for example, self-protection and acquiring resources); furthermore, these instrumental goals are likely to be harmful to humans.

The article quoted above was also a call-to-action for mainstream AI researchers and organisations to be more aware of the risks that general AI may pose. A survey of AI researchers by Müller and Bostrom (2014) reports that the median AI researcher believes with 50% probability that human-level machine intelligence will happen by the year 2040 (mean 2073, std. dev 144). Early in 2015, Elon Musk donated 10 million US dollars to the *Future of Life* institute (FLI) as grant money to be distributed to AI-Safety related projects. The first round of grant recipients was announced² recently, and both academic institutions as well as private research organisations were represented. This seems like a good first step toward AI-Safety research gaining mainstream academic acceptance.

¹<http://www.independent.co.uk/news/science/stephen-hawking-transcendence-looks-at-the-implications-of-artificial-intelligence--but-are-we-taking-ai-seriously-enough-9313474.html>

²View the grant recipients at <http://futureoflife.org/AI/2015awardees>.

It is true that not knowing what a generally intelligent agent is going to look like, makes the AI-Safety problem very hard. However, there are certain issues in AI-Safety that will need to be solved regardless of the shape of future AGI. FLI hosts an open letter³ signed by many top AI researchers and technocrats from all around the world, who believe that research on how to make AI systems “robust and beneficial is both important and timely, and that there are concrete research directions that can be pursued today.” Soares and Fallenstein (2014) argue that the issue of superintelligent value alignment should be studied formally “well in advance of making design decisions about smarter-than-human systems.”

A related important philosophical question is asking what such an AI would experience internally. Of course, we have only our own human experience with which we can relate to such questions, but it does raise certain ethical issues. If future AI algorithms become “sentient enough”⁴ then it seems that we would have to take into account their suffering. Tomasik (2014) makes the argument that current reinforcement learning agents might be of non-negligible sentience and we should be careful of the treatment of future, more intelligent, reinforcement learners. Along with some other students, we recently co-founded “People for the Ethical Treatment of Reinforcement Learners”⁵ to make people aware of the possibility that future algorithms *may* be worthy of moral consideration.

AI has the potential to greatly impact our current way of life in positive ways, but like any other powerful technology comes with risks. It is important that we do not fail to address them, given how high the stakes are. The next few decades could well be the most exciting and important time in the field of artificial intelligence.

³http://futureoflife.org/AI/open_letter

⁴Sentience, consciousness and related terms are of course currently ill-defined both within this thesis and outside of it.

⁵Visit us at <http://petrl.org>.

Omitted Proofs

Proof of Theorem 28. This proof is due to Mineiro (2010)¹. The idea behind the proof is to consider an adversary that wants to create the least possible regression error which induces some fixed amount of regret r in the CSMC. The best way to do this would be to swap the minimum cost class k^* and an adjacent class k' such that the regression error remains small, but the classifier predicts incorrectly causing the regret r . We use this intuition, and formalise it by framing the adversary's problem as a constrained optimisation problem, solvable via the KKT conditions.

Consider a particular sample x . Assume that the chosen regressor g has an error vector δ_x with respect to the true cost i.e. $\delta_x(k) = g(x, k) - c_x(k)$ for each k . Following the intuition described above, we consider an adversary that wants to find δ_x that minimises $e(g)$ on this instance x while causing a certain amount (r) of regret (for this instance) to the CSMC. The adversary thus faces the following optimisation problem

$$\begin{aligned} \min_{\delta_x} \mathbb{E}_{c \sim D_{c|x}} \frac{1}{2|K|} \left[\sum_{k \in K} (c_x(k) + \delta_x(k) - c_x(k))^2 \right] \\ \text{s.t. } \mathbb{E}_{c \sim D_{c|x}} [c(h(x))] - \min_{k \in K} \mathbb{E}_{(x,c) \sim D_{c|x}} [c_x(k)] = r \end{aligned}$$

We assume that $c_x(k)$ is different for every k . This can be easily imposed by adding a small random number to each $c_x(k)$. Then, the regret r can take on exactly $|K|$ values, one for each decision proposed by h_g . Let $k^* = \arg \min_k c_x(k)$ be the minimum cost class. Thus the adversary's problem is a family of optimisation problems indexed by k' as follows.

$$\begin{aligned} \min_{\delta_x} \frac{1}{2|K|} \sum_{k \in K} \delta_x(k)^2 \\ \text{s.t. } \forall k \neq k' \quad c_x(k') + \delta_x(k') \leq c_x(k) + \delta_x(k) \end{aligned}$$

We look at the KKT stationarity conditions in order to solve this optimisation problem. Let us write $f(\delta_x(k)) = \frac{1}{2} \sum_{k \in K} \delta_x(k)^2$ for the function we wish to minimise (since x is fixed). Thus $\nabla f(\delta_x(k))$ is a K -dimensional vector, and the $|K| - 1$ constraints are $m_k(\delta_x(k)) = c_x(k') + \delta_x(k') - c_x(k) - \delta_x(k) \leq 0$ for all $k \neq k'$.

¹Unusually, this is a blog article rather than a paper. Every step has been thoroughly checked by us.

Stationarity gives us that for each k

$$-\nabla f(\delta_x(k)) = \sum_{k \in K \setminus \{k'\}} \mu_k \nabla m_k(\delta_x(k))$$

where μ_k are the multipliers. We therefore have for $k \neq k'$

$$\delta_x(k) = \begin{cases} \mu_k & \text{if } k \neq k' \\ -\sum_{k \neq k'} \mu_k & \text{if } k = k' \end{cases}$$

Since $c_x(k') < c_x(k^*)$ we know that $\delta_x(k') < 0$. Complementary slackness gives us that for $k \neq k'$, $\mu_k m_k(\delta_x(k)) = 0$. Thus, $\delta_x(k) = 0$ for all $k \neq k'$, and $\delta(x, k^*) = -\delta(x, k')$, therefore $\delta_x(k^*) > 0$. Substituting this into the constraint and keeping in mind that $\sum_{k \in K} \delta_x(k) = 2\delta_x(k^*)$, we get

$$\begin{aligned} c_x(k') - c_x(k^*) &\leq 2\delta_x(k^*) \\ &= \sqrt{|K| \frac{1}{2|K|} \sum_{k \in K} \delta_x(k)^2} \\ \implies \mathbb{E}_{(x,c) \sim D} [c_x(k') - c_x(k^*)] &\leq \sqrt{|K|} \mathbb{E}_{(x,c) \sim D} \sqrt{\frac{1}{2|K|} \sum_{k \in K} \delta_x(k)^2} \\ &\leq \sqrt{|K| e(g)} \end{aligned}$$

The last inequality follows from Jensen's inequality. Therefore, $r_{csmc}(h_g) \leq \sqrt{|K| e(g)}$ as required.

□

Convex Optimisation

This brief appendix contains a short introduction to some convex optimisation theory necessary to understand the Stochastic Gradient Dual Ascent algorithm used in Chapter 6.

A function f is convex if for any two points x_1 and x_2 the line segment between them lies above the graph of the function. Common examples of convex functions are e^x , $\log x$, x^2 , etc.

Definition 33 (Convex function). A function f is convex iff for any two points x_1 and x_2 and for all $c \in [0, 1]$, $f(cx_1 + (1 - c)x_2) \leq cf(x_1) + (1 - c)f(x_2)$.

Additionally, a function is *strictly convex* if the inequality above is tight. We can also talk about *strongly convex* functions which generalise the idea of *strict convexity*. A strongly convex function is one that has a quadratic lower bound in the following sense.

Definition 34 (Strongly convex function). A function f is α -strongly convex if for some parameter $\alpha \in \mathbb{R}$ and for any two points x_1 and x_2 , $f(x_1) - f(x_2) \leq \nabla f(x)^\top (x_1 - x_2) + \frac{\alpha}{2} \|x_1 - x_2\|^2$.

In particular, if $f : X \rightarrow \mathbb{R}$ is twice continuously differentiable then f is strongly convex if and only if $f''(x) \geq \alpha$. This last fact can be made evident via a Taylor approximation expanded to second-order terms.

We will define a *smooth* function to be one that has a Lipschitz-continuous gradient function as follows.

Definition 35 (Smooth function). A function f is said to be β -smooth if and only if for any x_1, x_2 , $\|\nabla f(x_1) - \nabla f(x_2)\| \leq \beta \|x_1 - x_2\|$.

Our goal is to solve a ridge regression problem using convex optimisation. So we wish to find the weights w that minimise the loss

$$L(w) := \frac{1}{n} \sum_{i=1}^n (w^\top x_i - y_i)^2 + \frac{\lambda}{2} \|w\|^2$$

$L(w)$ is a strongly convex and 1-smooth function. In order to use SDCA, we need to define the *Lagrangian dual* of this problem. In order to do that we first need the definition of a convex conjugate.

A convex function f is called *proper* if it takes values on $(-\infty, +\infty]$ and has $f(x) < \infty$ for at least one x .

Definition 36 (Convex conjugate). The convex conjugate of a proper convex function f is given by

$$f^*(u) = \sup_{x \in X} (u^\top x - f(x))$$

The dual problem of minimising the above loss $L(w)$ is given by

$$\max_{\alpha \in \mathbb{R}^n} D(\alpha) := \frac{1}{n} \sum_{i=1}^n f^*(-\alpha_i) - \frac{\lambda}{2} \left\| \frac{1}{\lambda n} \sum_{i=1}^n \alpha_i x_i \right\|^2$$

where f^* is the convex conjugate of $f(u) = (u - y_i)^2$, given by $f^*(-a) = -ay_i + \frac{a^2}{4}$. The variables α_i are known as the dual variables, and there is an α_i associated with each training sample. SDCA does not attempt to maximise the objective with respect every dual variable at each timestep, rather it picks a random α_i at each timestep and optimises the dual objective with respect to only this α_i i.e. it assumes the others are kept constant. This results in Algorithm 11.

In order to recover the primal parameters, define $w(\alpha) = \frac{1}{\lambda n} \sum_{i=1}^n \alpha_i x_i$. Then Shalev-Shwartz and Zhang (2013) show that $w(\alpha^*) = w^*$. Let ε be the required duality gap. Since the squared loss is 1-smooth, Shalev-Shwartz and Zhang (2013) state that to have $\mathbb{E}[L(w) - D(\alpha)] \leq \varepsilon$ requires a total number of iterations $O(n + \frac{1}{\lambda}) \log \frac{1}{\varepsilon}$.

Algorithm 11: Stochastic Dual Coordinate Ascent by Shalev-Shwartz and Zhang (2013).

Input: starting dual parameters $\alpha^{(0)}$, required duality gap ε ;

$w^{(0)} = w(\alpha^{(0)})$;

Let $t = 0$;

while *duality gap greater than ε* **do**

for $j = 1, \dots, m$ **do**

$t \leftarrow t + 1$;

 Pick i randomly from $1, \dots, n$;

 Find $\delta\alpha_i$ to increase dual ;

$\alpha^{(t)} \leftarrow \alpha^{(t-1)} + \delta\alpha_i e_i$;

$w^{(t)} \leftarrow w^{(t-1)} + \frac{1}{\lambda n} \delta\alpha_i x_i$;

end

end

Choose a random t ;

return $\bar{w} = w^{(t)}$;

List of Notation

Acronyms and abbreviations

AI	Artificial Intelligence
RL	Reinforcement Learning
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
POMDP \ R	POMDP without rewards
hPOMDP	history-POMDP
DBN	Dynamic Bayesian Network
FRL	Feature Reinforcement Learning
MCTS	Monte-Carlo Tree Search
UCT	Upper-confidence bound for trees
TD	Temporal Difference
FA	Function Approximation
MC-AIXI-CTW	Monte-Carlo AIXI with Context Tree Weighting
PSR	Predictive State Representation
MSBE	Mean-squared Bellman error
MSPBE	Mean-squared Projected Bellman error
LSTM	Long short-term memory is a type of neural network architecture.
PAC	Probably Approximately Correct algorithm
$OCost$	the original cost function by Hutter (2009b).
$Cost_{QL}$	the model-free cost function defined in Chapter 5.

Sets

$\mathbb{N}$	set of natural numbers, including zero
$\mathbb{R}$	set of real numbers
$\mathcal{S}$	set of states
$\mathcal{A}$	set of actions
$\mathcal{R}$	set of rewards
$\mathcal{O}$	set of observations
$ \mathcal{S} $	size of the set $\mathcal{S}$

Other notation

iff	if and only if
i.e.	"that is" from the Latin "id est"
w.r.t	with respect to
etc.	"and so on" from the Latin "et cetera"
e.g.	"for example" from the Latin "exempli gratia"
$(a, b]$	an interval in $\mathbb{R}$ that is open at a closed at b
$\min \setminus \max$	minimum or maximal element of a set
$\arg \min_{x \in X} f(x)$	(usually) the $x \in X$ that minimises $f(x)$ with ties broken arbitrarily (sometimes) the set of all $x \in X$ that minimise $f(x)$
i, k, n	(usually) natural numbers
∞	infinity
$:=$	define
$\implies$	implies
$\iff$	if and only if
$f : A \rightsquigarrow B$	defines a stochastic mapping f between A and B such that there exists a function $p : A \times B \rightarrow [0, 1]$ where $p(a, b)$ is the probability that $f(a) = b$.
$\propto$	proportional to
$\forall, \exists$	for all, there exists
$\square$	q.e.d, "which had to be proven"
$\sqrt{x}$	square root of x
$\lim_{n \rightarrow \infty}$	limiting value of argument as n tends to infinity
$\sum_{i=1}^N$	summation from $i = 1$ to n (inclusive)
$\prod_{i=1}^N$	product from $i = 1$ to n (inclusive)
$\mathbb{I}_y(x)$	the indicator function that is 1 when $x=y$ and 0 otherwise
$\log$	logarithm to any basis
$\log_b$	logarithm to basis b
ε	a small positive number
ϵ	(usually) the empty history
$\ x\ _p$	p-norm of a vector x
γ	discount rate for a geometrically discounted environment
$T(s' \mid s, a)$	transition probabilities for the transition $s \xrightarrow{a} s'$.
$\Omega(o \mid s)$	emission probability of observation o in state s
π, π^*	policy, (an) optimal policy
$V^\pi(s)$	value of state s following policy π
$Q^\pi(s, a)$	value of state-action pair (s, a) following policy π
$V^*(s)$	value of state s following the optimal policy
$Q^*(s, a)$	value of state-action pair (s, a) following the optimal policy
$x_{1:n}$	a sequence of length n , $x_1, x_2, \dots, x_n$.
$CL(x_{1:n})$	the code-length of a sequence $x_{1:n}$
t	(usually) time step
s_t, a_t, o_t, r_t	the state, action, observation or reward at time step t

h_t	a history sequence of the agent's interaction with the environment up to (and including) time step t
$\phi(h_t)$	a map from a history h_t to a state s .
$uniform(0,1)$	a uniform distribution over $[0,1]$.

References

- ABERDEEN, D. AND JONATHAN, B., 2002. Scaling internal-state policy-gradient methods for POMDPs. In *Proc. ICML-02*, pp.3–10, 3–10. (cited on pages 27 and 99)
- ARGALL, B. D.; CHERNOVA, S.; VELOSO, M.; AND BROWNING, B., 2009. A Survey of Robot Learning from Demonstration. *Robot. Auton. Syst.*, 57, 5 (May 2009), 469–483. doi:10.1016/j.robot.2008.10.024. <http://dx.doi.org/10.1016/j.robot.2008.10.024>. (cited on page 107)
- AUER, P. AND ORTNER, R., 2007. Logarithmic online regret bounds for undiscounted reinforcement learning. In *Advances in Neural Information Processing Systems 19*, 49–56. MIT Press. (cited on page 21)
- AZAR, M. G.; LAZARIC, A.; AND BRUNSKILL, E., 2013. Regret Bounds for Reinforcement Learning with Policy Advice. In *ECML/PKDD (1)*, vol. 8188 of *Lecture Notes in Computer Science*, 97–112. Springer. (cited on page 122)
- BAIER, H., 2010. *Adaptive Playout Policies for Monte-Carlo Go*. Ph.D. thesis, MS thesis, Institute of Cognitive Science, Osnabrück University, Osnabrück, Germany. (cited on page 50)
- BAKKER, B., 2002. Reinforcement Learning with long short-term Memory. *Advances in neural information processing systems*, 2, 14 (2002), 1475–1482. (cited on pages 58, 62, 73, 75, and 78)
- BAKKER, P. AND KUNIYOSHI, Y., 1996. Robot See, Robot Do : An Overview of Robot Imitation. In *In AISB96 Workshop on Learning in Robots and Animals*, 3–11. (cited on page 107)
- BARD, N. AND BOWLING, M., 2007. Particle Filtering for Dynamic Agent Modelling in Simplified Poker. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, 515–521. (cited on page 47)
- BAUM, L. E.; PETRIE, T.; SOULES, G.; AND WEISS, N., 1970. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics*, 41 (1970), 164–171. (cited on page 26)
- BAXTER, J.; TRIDGELL, A.; AND WEAVER, L., 2000. Learning to play chess using temporal differences. *Machine Learning*, 40, 3 (2000), 243–263. (cited on page 49)
- BELLEMARE, M.; VENESS, J.; AND BOWLING, M., 2012. Sketch-Based Linear Value Function Approximation. In *Advances in Neural Information Processing Systems 25*, 2213–2221. Curran Associates, Inc. <http://papers.nips.cc/paper/4540-sketch-based-linear-value-function-approximation.pdf>. (cited on page 52)

-
- BELLEMARE, M.; VENESS, J.; AND BOWLING, M., 2013. Bayesian Learning of Recursively Factored Environments. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, vol. 28, 1211–1219. JMLR Workshop and Conference Proceedings. (cited on pages 52 and 123)
- BELLEMARE, M. G.; NADDAF, Y.; VENESS, J.; AND BOWLING, M., 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47 (06 2013), 253–279. (cited on pages 15, 44, 51, 52, 103, 114, 115, and 122)
- BELLMAN, R., 1954. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60, 6 (11 1954), 503–515. (cited on page 11)
- BELLMAN, R., 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edn. (cited on page 12)
- BENGIO, Y., 2000. Probabilistic neural network models for sequential data. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, vol. 5, 79–84 vol.5. doi:10.1109/IJCNN.2000.861438. (cited on page 127)
- BENGIO, Y.; COURVILLE, A.; AND VINCENT, P., 2013. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35, 8 (2013), 1798–1828. doi:10.1109/TPAMI.2013.50. (cited on page 41)
- BENGIO, Y. AND LECUN, Y., 2007. Scaling learning algorithms towards AI. In *Large-Scale Kernel Machines*. MIT Press. (cited on page 41)
- BILLINGS, D.; DAVIDSON, A.; SCHAEFFER, J.; AND SZAFRON, D., 2002. The challenge of poker. *Artificial Intelligence*, 134, 1 (2002), 201–240. (cited on page 47)
- BILLINGS, D.; PAPP, D.; SCHAEFFER, J.; AND SZAFRON, D., 1998. Opponent modeling in poker. In *AAAI/IAAI*, 493–499. (cited on page 47)
- BISHOP, C. M., 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 0387310738. (cited on pages 7 and 104)
- BOSTROM, N., 2012. The Superintelligent Will: Motivation and Instrumental Rationality in Advanced Artificial Agents. *Minds and Machines*, 22, 2 (2012), 71–85. doi:10.1007/s11023-012-9281-3. <http://dx.doi.org/10.1007/s11023-012-9281-3>. (cited on page 128)
- BOSTROM, N., 2014. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press. (cited on page 128)
- BOUTILIER, C.; DEARDEN, R.; GOLDSZMIDT, M.; AND GOLDSZMIDT, M. E., 1995. Exploiting Structure in Policy Construction. In , 1104–1111. (cited on page 34)
- BOWLING, M.; BURCH, N.; JOHANSON, M.; AND TAMMELIN, O., 2015. Heads-up Limit Hold'em Poker is Solved. *Science*, 347, 6218 (January 2015), 145–149. (cited on page 48)

-
- BRAFMAN, R. I.; TENNENHOLTZ, M.; AND KAEHLING, P., 2001. R-MAX - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. (cited on page 21)
- BRAZIUNAS, D., 2003. POMDP Solution Methods. Technical report. (cited on page 24)
- CAMPBELL, M.; HOANE, J. A. J.; AND HSU, F.-H., 2002. Deep Blue. *Artif. Intell.*, 134, 1-2 (Jan. 2002), 57–83. doi:10.1016/S0004-3702(01)00129-1. [http://dx.doi.org/10.1016/S0004-3702\(01\)00129-1](http://dx.doi.org/10.1016/S0004-3702(01)00129-1). (cited on pages 2 and 49)
- CHAKRABORTY, D. AND STONE, P., 2011. Structure Learning in Ergodic Factored MDPs without Knowledge of the Transition Function’s In-Degree. In *Proceedings of the Twenty Eighth International Conference on Machine Learning (ICML’11)*. <http://www.cs.utexas.edu/users/ai-lab/?ICML11-chakraborty>. (cited on page 37)
- CHESSPROGRAMMINGWIKI, 2014. Chess Programming Wiki for Stockfish, accessed June 2014. <http://chessprogramming.wikispaces.com/Stockfish>. (cited on page 49)
- CHICKERING, D. M., 1996. Learning Bayesian Networks is NP-Complete. In *Learning from Data: Artificial Intelligence and Statistics V* (Eds. D. FISHER AND H. LENZ), 121–130. Springer-Verlag. (cited on page 36)
- CHRISMAN, L., 1992a. Reinforcement learning with perceptual aliasing: the perceptual distinctions approach. In *Proceedings of the tenth national conference on Artificial intelligence, AAAI’92* (San Jose, California, 1992), 183–188. AAAI Press. (cited on pages 27 and 96)
- CHRISMAN, L., 1992b. Reinforcement Learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, 183–188. AAAI Press, San Jose, California. (cited on page 27)
- DAHL, F. A., 2001. A Reinforcement Learning Algorithm Applied to Simplified Two-Player Texas Hold’em Poker. In *Machine Learning: ECML 2001* (Eds. L. DE RAEDT AND P. FLACH), vol. 2167 of *Lecture Notes in Computer Science*, 85–96. Springer Berlin Heidelberg. ISBN 978-3-540-42536-6. doi:10.1007/3-540-44795-4_8. (cited on page 47)
- DASWANI, M.; SUNEHAG, P.; AND HUTTER, M., 2012. Feature Reinforcement Learning using Looping Suffix Trees. *JMLR Workshop and Conference Proceedings : EWRL 2012*, 24 (2012), 11–24. (cited on pages 61 and 99)
- DASWANI, M.; SUNEHAG, P.; AND HUTTER, M., 2013. Q-learning for History-based Reinforcement Learning. In *JMLR Workshop and Conference Proceedings: Volume 29: Asian Conference on Machine Learning*, 213–228. (cited on pages 51 and 79)
- DASWANI, M.; SUNEHAG, P.; AND HUTTER, M., 2014. Reinforcement Learning with Value Advice. In *Proceedings of the Sixth Asian Conference on Machine Learning*, 299–314. (cited on pages 101, 116, and 121)
- DEAN, T. AND KANAZAWA, K., 1992. Probabilistic Temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, 524–528. American Association for Artificial Intelligence, St. Paul, Minnesota. (cited on page 34)

-
- DEVLIN, S.; GRZES, M.; AND KUDENKO, D., 2009. Reinforcement learning in robocup keep-away with partial observability. In *Web Intelligence and Intelligent Agent Technologies. IEEE International Joint Conferences on*. (cited on page 55)
- DIUK, C.; LI, L.; AND LEFFLER, B. R., 2009. 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 (ICML 2009)*, 249–256. ACM. <http://www.princeton.edu/~cdiuk/papers/meteorologist.pdf>. (cited on page 37)
- ERNST, D.; GEURTS, P.; WEHENKEL, L.; AND LITTMAN, L., 2005. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6 (2005), 503–556. (cited on page 80)
- EVEN-DAR, E. AND MANSOUR, Y., 2001. Convergence of Optimistic and Incremental Q-Learning. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, 1499–1506. MIT Press. <http://www.math.tau.ac.il/~mansour/papers/01nips.ps>. (cited on page 19)
- FAN, R.; CHANG, K.; HSIEH, C.; WANG, X.; AND LIN, C., 2008. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9 (2008), 1871–1874. (cited on page 116)
- FARAHMAND, A.; GHAVAMZADEH, M.; SZEPESVÁRI, C.; AND MANNOR, S., 2008. Regularized Fitted Q-Iteration: Application to Planning. In *Recent Advances in Reinforcement Learning*, vol. 5323 of *Lecture Notes in Computer Science*, 55–68. Springer Berlin Heidelberg. ISBN 978-3-540-89721-7. (cited on page 80)
- FARIAS, V. F.; MOALLEMI, C. C.; WEISSMAN, T.; AND VAN ROY, B., 2007. Universal Reinforcement Learning. *IEEE Transactions on Information Theory*, 56, 5 (2007), 2441–2454. (cited on pages 5 and 88)
- FARIAS, V. F.; MOALLEMI, C. C.; WEISSMAN, T.; AND VAN ROY, B., 2010. Universal Reinforcement Learning. *IEEE Transactions on Information Theory*, 56, 5 (2010), 2441–2454. (cited on page 62)
- F.R.S., K. P., 1901. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series* 6, 2, 11 (1901), 559–572. doi:10.1080/14786440109462720. <http://dx.doi.org/10.1080/14786440109462720>. (cited on page 41)
- FUJITA, H.; MATSUNO, Y.; AND ISHII, S., 2003. A reinforcement learning scheme for a multi-agent card game. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, vol. 5, 4071–4078. IEEE. (cited on page 53)
- GEIST, M.; SCHERRER, B.; LAZARIC, A.; AND GHAVAMZADEH, M., 2012. A Dantzig Selector Approach to Temporal Difference Learning. In *International Conference on Machine Learning*. (cited on page 80)

-
- GELLY, S. AND SILVER, D., 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning, ICML '07* (Corvalis, Oregon, 2007), 273–280. ACM, New York, NY, USA. doi:10.1145/1273496.1273531. <http://doi.acm.org/10.1145/1273496.1273531>. (cited on page 50)
- GHAVAMZADEH, M.; LAZARIC, A.; MUNOS, R.; AND HOFFMAN, M. W., 2011. Finite-Sample Analysis of Lasso-TD. In *ICML*, 1177–1184. Omnipress. (cited on pages 80 and 99)
- GIVAN, R.; DEAN, T.; AND GREIG, M., 2003. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147, 1–2 (2003), 163–223. doi:10.1016/S0004-3702(02)00376-4. <http://www.sciencedirect.com/science/article/pii/S0004370202003764>. Planning with Uncertainty and Incomplete Information. (cited on page 23)
- GOOD, I. J., 1965. Speculations Concerning the First Ultrainelligent Machine. In *Advances in Computers*, volume 6. (Eds. F. ALT AND M. RUMINOFF). Academic Press. (cited on page 128)
- GUO, X.; SINGH, S.; LEE, H.; LEWIS, R. L.; AND WANG, X., 2014. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. In *Advances in Neural Information Processing Systems 27*, 3338–3346. Curran Associates, Inc. <http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf>. (cited on page 121)
- GUO, Y. AND SCHUURMANS, D., 2006. Convex Structure Learning for Bayesian Networks: Polynomial Feature Selection and Approximate Ordering. (cited on page 36)
- HAGHIGHI, D. K.; PRECUP, S. D.; PINEAU, J.; AND PANANGADEN, P., 2007. Learning Algorithms for Automata with Observations. (cited on pages 62 and 78)
- HAMILTON, W.; FARD, M. M.; AND PINEAU, J., 2014. Efficient Learning and Planning with Compressed Predictive States. *J. Mach. Learn. Res.*, 15, 1 (Jan. 2014), 3395–3439. <http://dl.acm.org/citation.cfm?id=2627435.2750354>. (cited on pages 53 and 96)
- HAMILTON, W. L.; FARD, M. M.; AND PINEAU, J., 2013. Modelling Sparse Dynamical Systems with Compressed Predictive State Representations. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, vol. 28, 178–186. JMLR Workshop and Conference Proceedings. (cited on page 38)
- HAUSKNECHT, M.; LEHMAN, J.; MIKKULAINEN, R.; AND STONE, P., 2013. A Neuroevolution Approach to General Atari Game Playing. In *IEEE Transactions on Computational Intelligence and AI in Games*. (cited on pages 122 and 127)
- HAUSKNECHT, M. J. AND STONE, P., 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. *CoRR*, abs/1507.06527 (2015). <http://arxiv.org/abs/1507.06527>; <http://dblp.uni-trier.de/rec/bib/journals/corr/HausknechtS15>. (cited on page 127)
- HAZAN, E.; KALAI, A.; KALE, S.; AND AGARWAL, A., 2006. Logarithmic regret algorithms for online convex optimization. In *Learning Theory*, 499–513. Springer. (cited on page 106)

-
- HEINRICH, J. AND SILVER, D., 2014. Self-play Monte-Carlo tree search in computer poker. In *Proceedings of the AAAI Workshop on Computer Poker and Perfect Information*. (cited on page 47)
- HINTON, G.; OSINDERO, S.; AND TEH, Y.-W., 2006. A fast learning algorithm for deep belief nets. *Neural computation*, 18, 7 (2006), 1527–1554. (cited on page 42)
- HOCHREITER, S. AND SCHMIDHUBER, J., 1997. Long Short-Term Memory. *Neural Computation*, 9, 8 (Nov. 1997), 1735–1780. doi:10.1162/neco.1997.9.8.1735. (cited on page 62)
- HOLMES, M. P.; ISBELL, C. L.; AND JR., 2006. Looping Suffix Tree-Based Inference of Partially Observable Hidden State. In *In Proceedings of ICML. 2006*, 409–416. Apple Developer Press. (cited on pages 6, 7, 61, 62, 63, 64, 66, 67, 68, 75, 78, and 125)
- HUANG, H., 2011. Skynet meets the Swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition. <http://arstechnica.com/gaming/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition/>. (cited on page 45)
- HUTTER, M., 2005. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin. (cited on pages 3, 4, 6, 29, and 127)
- HUTTER, M., 2009a. Feature Dynamic Bayesian Networks. In *Proc. 2nd Conf. on Artificial General Intelligence (AGI'09)*, vol. 8, 67–73. Atlantis Press. doi:10.2991/agi.2009.6. (cited on page 34)
- HUTTER, M., 2009b. Feature Reinforcement Learning: Part I: Unstructured MDPs. *Journal of Artificial General Intelligence*, 1 (2009), 3–24. (cited on pages 3, 31, 32, 70, 78, 87, 99, 125, 127, and 137)
- HUTTER, M., 2014. Extreme State Aggregation beyond MDPs. In *Proc. 25th International Conf. on Algorithmic Learning Theory (ALT'14)*, vol. 8776 of *LNAI*, 185–199. Springer, Bled, Slovenia. doi:10.1007/978-3-319-11662-4_14. <http://arxiv.org/abs/1407.3341>. (cited on pages 23, 40, 97, 126, and 127)
- JAKSCH, T.; ORTNER, R.; AND AUER, P., 2010. Near-optimal Regret Bounds for Reinforcement Learning. *Journal of Machine Learning Research*, 11 (2010), 1563–1600. (cited on pages 21 and 37)
- JOHANSON, M. B., 2007. Robust strategies and counter-strategies: Building a champion level computer poker player. In *Masters Abstracts International*, vol. 46. (cited on page 48)
- JOHNS, J.; PAINTER-WAKEFIELD, C.; AND PARR, R., 2010. Linear complementarity for regularized policy evaluation and improvement. In *In Advances in Neural Information Processing Systems 23*. (cited on pages 80 and 99)
- JONSSON, A. AND BARTO, A., 2007. Active learning of dynamic Bayesian networks in Markov decision processes. In *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation, SARA'07* (Whistler, Canada, 2007), 273–284. Springer-Verlag, Berlin, Heidelberg. <http://dl.acm.org/citation.cfm?id=1770681.1770705>. (cited on page 36)

-
- Kaelbling, L. P.; Littman, M. L.; AND Cassandra, A. R., 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1–2 (1998), 99–134. doi:10.1016/S0004-3702(98)00023-X. (cited on page 57)
- Kaelbling, L. P.; Littman, M. L.; AND Moore, A. W., 1996. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4 (1996), 237–285. (cited on page 9)
- Kakade, S. AND Langford, J., 2002. Approximately Optimal Approximate Reinforcement Learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 267–274. Morgan Kaufmann Publishers Inc. (cited on page 110)
- Kearns, M., 1998. Near-optimal reinforcement learning in polynomial time. In *Machine Learning*, 260–268. Morgan Kaufmann. (cited on page 21)
- Kocsis, L. AND Szepesvári, C., 2006. Bandit based Monte-Carlo planning. In *The 17th European Conference on Machine Learning*, 99–134. (cited on pages 14, 15, and 102)
- Koenig, S. AND Simmons, R. G., 1996. Unsupervised Learning of Probabilistic Models for Robot Navigation. In *in Proceedings of the IEEE International Conference on Robotics and Automation*, 2301–2308. (cited on page 27)
- Koller, D. AND Friedman, N., 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press. (cited on page 36)
- Kolter, J. Z. AND Ng, A. Y., 2009. Regularization and feature selection in least-squares temporal difference learning. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning* (Montreal, Quebec, Canada, 2009), 521–528. ACM, New York, NY, USA. doi:10.1145/1553374.1553442. (cited on pages 80 and 99)
- LAGOUDAKIS, M. G. AND PARR, R., 2003. Reinforcement Learning as Classification: Leveraging Modern Classifiers. In *in Proceedings of the Twentieth International Conference on Machine Learning*, 424–431. (cited on page 122)
- Lattimore, T., 2013. *Theory of general reinforcement learning - PhD thesis*. Ph.D. thesis, Australian National University. (cited on pages 4 and 6)
- Lattimore, T. AND Hutter, M., 2012. PAC Bounds for Discounted MDPs. In *Proceedings of the 23th International Conference on Algorithmic Learning Theory*, vol. 7568 of *Lecture Notes in Computer Science*, 320–334. Springer Berlin / Heidelberg. (cited on page 21)
- Legg, S. AND Hutter, M., 2007. A Collection of Definitions of Intelligence. In *Proceedings of the 2007 Conference on Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006*, 17–24. IOS Press, Amsterdam, The Netherlands, The Netherlands. <http://dl.acm.org/citation.cfm?id=1565455.1565458>. (cited on page 3)
- Levinson, R. AND Weber, R., 2001. Chess neighborhoods, function combination, and reinforcement learning. In *Computers and Games*, 133–150. Springer. (cited on page 49)

-
- LEVINSON, R. AND WEBER, R. J., 2000. Pattern-level temporal difference learning, data fusion, and chess. In *Sensor Fusion: Architectures, Algorithms, and Applications IV*, 43 (April 3, 2000), vol. 4051, 43–54. doi:10.1117/12.381657. <http://dx.doi.org/10.1117/12.381657>. (cited on page 49)
- LIONHEAD STUDIOS, 2001. Black and White. (cited on page 43)
- LITTMAN, M. L., 1994a. Memoryless Policies: Theoretical Limitations and Practical Results. In *In*, 238–245. The MIT Press. (cited on pages 28 and 61)
- LITTMAN, M. L., 1994b. The Witness Algorithm: Solving Partially Observable Markov Decision Processes. Technical report, Brown University, Providence, RI, USA. (cited on page 25)
- LITTMAN, M. L.; CASSANDRA, A. R.; AND KAEHLING, L. P., 1995. Learning policies for partially observable environments: Scaling up. (cited on page 26)
- LITTMAN, M. L.; SUTTON, R. S.; AND SINGH, S., 2001. Predictive Representations of State. In *In Advances In Neural Information Processing Systems 14*, 1555–1561. MIT Press. (cited on pages 37, 38, 39, 41, and 127)
- LIU, J. S., 2008. *Monte Carlo Strategies in Scientific Computing*. Springer, 2nd edn. (cited on page 31)
- LOCKETT, A. J. AND MIKKULAINEN, R., 2008. Evolving Opponent Models for Texas Hold 'Em. In *IEEE Conference on Computational Intelligence in Games*. Perth, Australia. <http://www.cs.utexas.edu/users/ai-lab/?lockett:cig08>. (cited on page 47)
- LOVE, N.; HINRICHS, T.; HALEY, D.; SCHKUFZA, E.; AND GENESERETH, M., 2008. General Game Playing: Game Description Language Specification. (cited on page 55)
- MACLIN, R. AND SHAVLIK, J. W., 1996. Creating Advice-Taking Reinforcement Learners. *Machine Learning*, 22, 1-3 (1996), 251–281. (cited on page 122)
- MACLIN, R.; SHAVLIK, J. W.; TORREY, L.; WALKER, T.; AND WILD, E. W., 2005. Giving Advice about Preferred Actions to Reinforcement Learners Via Knowledge-Based Kernel Regression. In *AAAI*, 819–824. AAAI Press / The MIT Press. (cited on page 122)
- MADDISON, C. J.; HUANG, A.; SUTSKEVER, I.; AND SILVER, D., 2014. Move Evaluation in Go Using Deep Convolutional Neural Networks. *arXiv preprint arXiv:1412.6564*, (2014). (cited on page 51)
- MAEI, H. R.; SZEPESVÁRI, C.; BHATNAGAR, S.; AND SUTTON, R. S., 2010. Toward Off-Policy Learning Control with Function Approximation. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 719–726. Omnipress. <http://www.icml2010.org/papers/627.pdf>. (cited on pages 17 and 84)
- MAHMUD, M. M. H., 2010. Constructing States for Reinforcement Learning. In *ICML*, 727–734. Omnipress. (cited on page 62)

-
- McCALLUM, A. K., 1996. Learning to Use Selective Attention and Short-Term Memory in Sequential Tasks. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 315–324. MIT Press. (cited on pages 5, 28, 88, and 127)
- McCALLUM, A. K., 1996. *Reinforcement Learning with Selective Perception and Hidden State*. . Ph.D. thesis, University of Rochester. (cited on page 57)
- McCALLUM, R. A., 1995. Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. In *In Proceedings of the Twelfth International Conference on Machine Learning*, 387–395. Morgan Kaufmann. (cited on pages 5, 28, 62, and 96)
- MINEIRO, P., 2010. Error and Regret Bounds for Cost-Sensitive Multiclass Classification Reduction to Regression. <http://www.machinedlearnings.com/2010/08/error-and-regret-bounds-for-cost.html>. (cited on pages 105 and 131)
- MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLU, I.; D.; AND RIEDMILLER, M., 2013. Playing Atari with Deep Reinforcement Learning. (2013). (cited on pages 52 and 123)
- MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIQ, A.; ANTONOGLU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; AND HASSABIS, D., 2015. Human-level control through deep reinforcement learning. *Nature*, 518, 7540 (02 2015), 529–533. <http://dx.doi.org/10.1038/nature14236>. (cited on pages 52, 121, and 127)
- MOODY, J. AND SAFFELL, M., 2001. Learning to trade via direct reinforcement. *Neural Networks, IEEE Transactions on*, 12, 4 (2001), 875–889. (cited on page 54)
- MÜLLER, V. C. AND BOSTROM, N., 2014. Future Progress in Artificial Intelligence: A Poll Among Experts. *AI Matters*, 1, 1 (Sep. 2014), 9–11. doi:10.1145/2639475.2639478. <http://doi.acm.org/10.1145/2639475.2639478>. (cited on page 128)
- MURPHY, K. P., 2000. A Survey of POMDP Solution Techniques. Technical report. (cited on pages 24 and 26)
- NADDAF, Y., 2010. *Game-Independent AI Agents for Playing Atari 2600 Console Games*. Masters, University of Alberta. (cited on page 52)
- NEVMYVAKA, Y.; FENG, Y.; AND KEARNS, M., 2006. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd international conference on Machine learning*, 673–680. ACM. (cited on page 54)
- NGUYEN, P., 2013. *Feature Reinforcement Learning Agents - PhD thesis*. Ph.D. thesis, Australian National University. (cited on pages 5, 77, 97, 99, 125, and 126)
- NGUYEN, P.; SUNEHAG, P.; AND HUTTER, M., 2011. Feature Reinforcement Learning in Practice. In *Proc. 9th European Workshop on Reinforcement Learning (EWRL-9)*, vol. 7188 of *LNAI*, 66–77. Springer. doi:10.1007/978-3-642-29946-9_10. (cited on pages 5, 32, 34, 59, 62, 70, 71, 78, 80, 88, and 99)

-
- NGUYEN, P. M.; SUNEHAG, P.; AND HUTTER, M., 2012. Context Tree Maximizing. In *AAAI*. AAAI Press. (cited on pages 5, 29, 51, 53, 88, 99, and 126)
- ORTNER, R., 2007. Pseudometrics for State Aggregation in Average Reward Markov Decision Processes. In *Algorithmic Learning Theory* (Eds. M. HUTTER; R. SERVEDIO; AND E. TAKIMOTO), vol. 4754 of *Lecture Notes in Computer Science*, 373–387. Springer Berlin Heidelberg. ISBN 978-3-540-75224-0. doi:10.1007/978-3-540-75225-7_30. http://dx.doi.org/10.1007/978-3-540-75225-7_30. (cited on page 23)
- OSBAND, I. AND VAN ROY, B., 2014. Near-optimal Reinforcement Learning in Factored MDPs. In *Advances in Neural Information Processing Systems 27*, 604–612. Curran Associates, Inc. <http://papers.nips.cc/paper/5445-near-optimal-reinforcement-learning-in-factored-mdps.pdf>. (cited on page 37)
- PAPADIMITRIOU, C. AND TSITSIKLIS, J. N., 1987. The complexity of Markov decision processes. *Math. Oper. Res.*, 12, 3 (1987), 441–450. doi:10.1287/moor.12.3.441. (cited on pages 4 and 24)
- PENDRITH, M. D. AND MCGARITY, M., 1998. An Analysis of Direct Reinforcement Learning in Non-Markovian Domains. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*, 421–429. Morgan Kaufmann. <http://dblp.uni-trier.de/rec/bib/conf/icml/PendrithM98>. (cited on page 29)
- PENG, J. AND WILLIAMS, R. J., 1996. Incremental Multi-Step Q-Learning. In *Machine Learning*, 226–232. Morgan Kaufmann. (cited on page 18)
- PETERS, J., 2010. Policy gradient methods. *Scholarpedia*, 5, 11 (2010), 3698. revision #137199. (cited on page 18)
- PINEAU, J.; GORDON, G.; AND THRUN, S., 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1025–1032. (cited on pages 24 and 26)
- PUTERMAN, M. L., 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0471619779. (cited on pages 11 and 13)
- RISSANEN, J., 1983. A Universal Data Compression System. *IEEE Trans. Inf. Theor.*, 29, 5 (Sep. 1983), 656–664. (cited on page 5)
- RISSANEN, J. J., 1978. Modeling by Shortest Data Description. *Automatica*, 14, 5 (1978), 465–471. (cited on pages 31, 41, and 81)
- ROBBINS, H. AND S.MONRO, 1951. A stochastic approximation method. *Annals of Mathematical Statistics* 22, (1951), 400–407. (cited on page 81)
- ROSENCRANTZ, M.; GORDON, G.; AND THRUN, S., 2004. Learning low dimensional predictive representations. In *Proceedings of the twenty-first international conference on Machine learning*, 88. ACM. (cited on page 38)

-
- ROSS, S., 2013. Interactive Learning for Sequential Decisions and Predictions. Ph.D. Thesis, Carnegie Mellon University. (2013). (cited on pages 7 and 108)
- ROSS, S. AND BAGNELL, D., 2010. Efficient Reductions for Imitation Learning. In *AISTATS*, vol. 9 of *JMLR Proceedings*, 661–668. JMLR.org. (cited on pages 102, 107, and 108)
- RUBIN, J. AND WATSON, I., 2011. Computer poker: A review. *Artificial Intelligence*, 175, 5–6 (2011), 958–987. doi:10.1016/j.artint.2010.12.005. <http://www.sciencedirect.com/science/article/pii/S0004370211000191>. Special Review Issue. (cited on page 47)
- RUSSELL, S. J. AND NORVIG, P., 2009. *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall. (cited on pages ix, 1, 9, and 43)
- SAMUEL, A. L., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, 3, 3 (Jul. 1959), 210–229. doi:10.1147/rd.33.0210. <http://dx.doi.org/10.1147/rd.33.0210>. (cited on page 2)
- SCHAAL, S., 1997. Learning from demonstration. In *Advances in Neural Information Processing Systems 9*, 1040–1046. MIT Press. (cited on page 107)
- SCHMIDHUBER, J., 2014. Deep Learning in Neural Networks: An Overview. *CoRR*, abs/1404.7828 (2014). <http://arxiv.org/abs/1404.7828>. (cited on page 42)
- SENSEI XMP, 2015. Wiki about the Computer Go Server on Sensei XMP. <http://senseis.xmp.net/?ComputerGoServer>. (cited on pages 50 and 51)
- SHALEV-SHWARTZ, S., 2011. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4, 2 (2011), 107–194. (cited on page 106)
- SHALEV-SHWARTZ, S. AND ZHANG, T., 2013. Stochastic Dual Coordinate Ascent Methods for Regularized Loss. *J. Mach. Learn. Res.*, 14, 1 (Feb. 2013), 567–599. (cited on pages 105, 134, and 135)
- SHANI, G.; PINEAU, J.; AND KAPLOW, R., 2013. A Survey of Point-based POMDP Solvers. *Autonomous Agents and Multi-Agent Systems*, 27, 1 (Jul. 2013), 1–51. doi:10.1007/s10458-012-9200-2. <http://dx.doi.org/10.1007/s10458-012-9200-2>. (cited on page 24)
- SILVER, D.; SUTTON, R.; AND MÜLLER, M., 2007. Reinforcement Learning of Local Shape in the Game of Go. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI'07* (Hyderabad, India, 2007), 1053–1058. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. <http://dl.acm.org/citation.cfm?id=1625275.1625446>. (cited on page 50)
- SILVER, D.; SUTTON, R. S.; AND MÜLLER, M., 2012. Temporal-difference search in computer Go. *Machine Learning*, 87, 2 (2012), 183–219. doi:10.1007/s10994-012-5280-0. <http://dx.doi.org/10.1007/s10994-012-5280-0>. (cited on page 50)

-
- SILVER, D. AND VENESS, J., 2010. Monte-Carlo Planning in Large POMDPs. In *Advances in Neural Information Processing Systems 23* (Eds. J. LAFFERTY; C. WILLIAMS; J. SHAWE-TAYLOR; R. ZEMEL; AND A. CULOTTA), 2164–2172. Curran Associates, Inc. <http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps.pdf>. (cited on pages 26 and 53)
- SINGH, S. AND JAMES, M. R., 2004. Predictive state representations: A new theory for modeling dynamical systems. In *In Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference (UAI)*, 512–519. AUAI Press. (cited on pages 37, 38, and 127)
- SINGH, S. P.; JAAKKOLA, T.; AND JORDAN, M. I., 1994. Learning Without State-Estimation in Partially Observable Markovian Decision Processes. In *In Proceedings of the Eleventh International Conference on Machine Learning*, 284–292. Morgan Kaufmann. (cited on page 28)
- SMALLWOOD, R. D. AND SONDIK, E. J., 1973. The Optimal Control of Partially Observable Markov Processes over a Finite Horizon. *Operations Research*, 21, 5 (1973), 1071–1088. doi:10.1287/opre.21.5.1071. <http://dx.doi.org/10.1287/opre.21.5.1071>. (cited on pages 23 and 24)
- SMITH, T. AND SIMMONS, R., 2004. Heuristic Search Value Iteration for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04 (Banff, Canada, 2004), 520–527. AUAI Press, Arlington, Virginia, United States. <http://dl.acm.org/citation.cfm?id=1036843.1036906>. (cited on page 26)
- SOARES, N. AND FALLENSTEIN, B., 2014. Aligning superintelligence with Human Interests: A Technical Research Agenda. Technical report. <https://intelligence.org/files/TechnicalAgenda.pdf>. (cited on page 129)
- SONDIK, E. J., 1978. The Optimal Control of Partially Observable Markov Processes Over the Infinite Horizon: Discounted Costs. *Operations Research*, 26, 2 (March–April 1978), 282–304. <http://www.jstor.org/stable/169635>. (cited on page 25)
- SOUTHEY, F.; BOWLING, M.; LARSON, B.; PICCIONE, C.; BURCH, N.; BILLINGS, D.; AND RAYNER, C., 2005. Bayes' bluff: Opponent modelling in poker. *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, (2005), 550–558. (cited on page 47)
- SPAAN, M. T. J. AND VLASSIS, N., 2005. Perseus: Randomized Point-based Value Iteration for POMDPs. *J. Artif. Int. Res.*, 24, 1 (Aug. 2005), 195–220. <http://dl.acm.org/citation.cfm?id=1622519.1622525>. (cited on page 26)
- STANLEY, K. O. AND MIKKULAINEN, R., 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10, 2 (2002), 99–127. <http://nn.cs.utexas.edu/?stanley:ec02>. (cited on pages 47 and 127)
- STARCRRAFT WORLDWIDE RANKINGS, 2015. Starcraft Worldwide Rankings, accessed January 2015. <http://nios.kr>. (cited on page 46)

-
- STONE, P.; KUHLMANN, G.; TAYLOR, M. E.; AND LIU, Y., 2006. Keepaway Soccer: From Machine Learning Testbed to Benchmark. In *RoboCup-2005: Robot Soccer World Cup IX* (Eds. I. NODA; A. JACOFF; A. BREDENFELD; AND Y. TAKAHASHI), vol. 4020, 93–105. Springer Verlag, Berlin. (cited on page 55)
- STONE, P. AND SUTTON, R. S., 2001. Scaling Reinforcement Learning toward RoboCup Soccer. In *ICML*, 537–544. Morgan Kaufmann. (cited on pages 54 and 55)
- STONE, P.; SUTTON, R. S.; AND KUHLMANN, G., 2005. Reinforcement Learning for RoboCup Soccer Keepaway. *Adaptive Behaviour*, 13 (2005), 165–188. (cited on page 55)
- STREHL, A. L., 2004. An empirical evaluation of interval estimation for markov decision processes. In *In Proc. of the 16th Int. Conf. on Tools with Artificial Intelligence (ICTAI'04)*, 128–135. (cited on page 21)
- STREHL, A. L.; DIUK, C.; AND LITTMAN, M. L., 2007. Efficient structure learning in factored-state MDPs. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-07)*. (cited on page 37)
- STREHL, A. L.; LI, L.; AND LITTMAN, M. L., 2009. Reinforcement Learning in Finite MDPs: PAC Analysis. *Journal of Machine Learning Research.*, 10 (dec 2009), 2413–2444. (cited on page 21)
- STURTEVANT, N. R. AND WHITE, A. M., 2007. Feature construction for reinforcement learning in hearts. In *Computers and Games*, 122–134. Springer. (cited on page 53)
- SUNEHAG, P. AND HUTTER, M., 2010. Consistency of Feature Markov Processes. In *Proc. 21st International Conf. on Algorithmic Learning Theory (ALT'10)*, vol. 6331 of *LNAI*, 360–374. Springer, Berlin, Canberra. doi:10.1007/978-3-642-16108-7_29. (cited on pages 32, 82, and 83)
- SUTTON, R. S. AND BARTO, A. G., 1998. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edn. ISBN 0262193981. (cited on pages ix, 1, 2, 13, 16, 18, 22, and 101)
- SUTTON, R. S.; MAEL, H. R.; PRECUP, D.; BHATNAGAR, S.; SILVER, D.; SZEPESVÁRI, C.; WIEWIORA, E.; AND INTRODUCED, M. R., 2009. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *In Proceedings of the 26th International Conference on Machine Learning*. (cited on page 81)
- SZITA, I.; GYENES, V.; AND LŐRINCZ, A., 2006. Reinforcement learning with echo state networks. In *Proceedings of the 16th international conference on Artificial Neural Networks - Volume Part I, ICANN'06* (Athens, Greece, 2006), 830–839. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/11840817_86. (cited on pages 62 and 75)
- SZITA, I. AND LŐRINCZ, A., 2008. The many faces of optimism: a unifying approach. In *Proceedings of the 25th international conference on Machine learning, ICML '08* (Helsinki, Finland, 2008), 1048–1055. ACM, New York, NY, USA. doi:10.1145/1390156.1390288. <http://doi.acm.org/10.1145/1390156.1390288>. (cited on page 21)

-
- SZITA, I. AND LÖRINCZ, A., 2008a. Factored value iteration converges. *Acta Cybern.*, 18, 4 (Jan. 2008), 615–635. <http://dl.acm.org/citation.cfm?id=1515867.1515873>. (cited on page 37)
- SZITA, I. AND LÖRINCZ, A., 2008b. The Many Faces of Optimism: a Unifying Approach. In *Proc. 12th International Conference (ICML 2008)*, vol. 307. Helsinki, Finland. (cited on page 70)
- TAMMELIN, O., 2014. Solving Large Imperfect Information Games Using CFR+. *CoRR*, abs/1407.5042 (2014). <http://arxiv.org/abs/1407.5042>. (cited on page 48)
- TAYLOR, M. E.; CARBONI, N.; FACHANTIDIS, A.; VLAHAVAS, I. P.; AND TORREY, L., 2014. Reinforcement learning agents providing advice in complex video games. *Connect. Sci.*, 26, 1 (2014), 45–63. (cited on pages 122 and 123)
- TAYLOR, M. E. AND STONE, P., 2005. Behavior transfer for value-function-based reinforcement learning. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 53–59. ACM. (cited on page 55)
- TAYLOR, M. E. AND STONE, P., 2007. Cross-domain Transfer for Reinforcement Learning. In *Proceedings of the 24th International Conference on Machine Learning, ICML '07* (Corvalis, Oregon, 2007), 879–886. ACM, New York, NY, USA. doi:10.1145/1273496.1273607. <http://doi.acm.org/10.1145/1273496.1273607>. (cited on page 55)
- TAYLOR, M. E.; STONE, P.; AND LIU, Y., 2007. Transfer Learning via Inter-Task Mappings for Temporal Difference Learning. *Journal of Machine Learning Research*, 8, 1 (2007), 2125–2167. (cited on page 55)
- TESAURO, G., 1995. Temporal Difference Learning and TD-Gammon. *Commun. ACM*, 38, 3 (Mar. 1995), 58–68. doi:10.1145/203330.203343. <http://doi.acm.org/10.1145/203330.203343>. (cited on pages 2 and 22)
- THOMPSON, W. R., 1933. On the Likelihood that one Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25 (1933), 285–294. (cited on pages 37 and 47)
- THRUN, S., 1995. Learning to play the game of chess. *Advances in neural information processing systems*, 7 (1995). (cited on page 49)
- TIMMER, S. AND RIEDMILLER, M., 2007. Safe Q-Learning on Complete History Spaces. In *Proceedings of the 18th European Conference on Machine Learning, ECML '07* (Warsaw, Poland, 2007), 394–405. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-74958-5_37. (cited on pages 28 and 62)
- TOMASIK, B., 2014. Do Artificial Reinforcement-Learning Agents Matter Morally? *CoRR*, abs/1410.8233 (2014). <http://arxiv.org/abs/1410.8233>. (cited on page 129)
- TU, H.-H. AND LIN, H.-T., 2010. One-sided support vector regression for multiclass cost-sensitive classification. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 1095–1102. (cited on page 105)

-
- VENESS, J.; BELLEMARE, M. G.; HUTTER, M.; CHUA, A.; AND DESJARDINS, G., 2014. Compress and Control. *CoRR*, abs/1411.5326 (2014). <http://arxiv.org/abs/1411.5326>. (cited on page 52)
- VENESS, J.; NG, K. S.; HUTTER, M.; UTHER, W.; AND SILVER, D., 2011. A Monte Carlo AIXI Approximation. *Journal of Artificial Intelligence Research*, 40 (2011), 95–142. doi:10.1613/jair.3125. (cited on pages 3, 29, 51, 52, 59, 62, 69, 87, 88, 94, and 95)
- VOLF, P. A. J. AND WILLEMS, F. M. J., 1994. Context maximizing: Finding mdl decision trees. In *In Symposium on Information Theory in the Benelux, Vol.15*, 192–200. (cited on page 29)
- WENDER, S. AND WATSON, I., 2012. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 402–408. doi:10.1109/CIG.2012.6374183. (cited on page 45)
- WHITE, M., 2014. Integrating Representation Learning and Temporal Difference Learning: A Matrix Factorization Approach. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*. (cited on page 42)
- WHITEHEAD, S. D. AND BALLARD, D. H., 1991. Learning to Perceive and Act by Trial and Error. *Machine Learning*, 7 (1991), 45–83. (cited on page 24)
- WIERSTRA, D.; FOERSTER, A.; PETERS, J.; AND SCHMIDHUBER, J., 2007. Solving deep memory POMDPs with recurrent policy gradients. In *ICANN'07: Proceedings of the 17th international conference on Artificial neural networks* (Porto, Portugal, 2007), 697–706. Springer-Verlag, Berlin, Heidelberg. (cited on pages 62, 73, 75, and 78)
- WIEWIORA, E.; COTTRELL, G. W.; AND ELKAN, C., 2003. Principled Methods for Advising Reinforcement Learning Agents. In *ICML*, 792–799. AAAI Press. (cited on page 122)
- WILEMS, F.; SHTARKOV, Y.; AND TJALKENS, T., 1995. The context tree weighting method: Basic properties. In *IEEE Transactions on Information Theory*. (cited on pages 5 and 29)