University of Alberta

Fast, Scalable Algorithms for Reinforcement Learning in High Dimensional Domains

by

Marc Gendron-Bellemare

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Fall 2013
Edmonton, Alberta

To my piano.

Abstract

This thesis presents new algorithms for dealing with large scale reinforcement learning problems. Central to this work is the Atari 2600 platform, which acts as both a rich evaluation framework and a source of challenges for existing reinforcement learning methods. Three contributions are presented; common to all three is the idea of leveraging the highly structured nature of Atari 2600 games in order to achieve meaningful results.

The first part of this work formally introduces the notion of contingency awareness: the recognition that parts of an agent's observation are under its control, while others are solely determined by its environment. Together with this formalization, I provide empirical results showing that contingency awareness can be used to generate useful features for value-based reinforcement learning in Atari 2600 games.

The second part investigates the use of hashing in linear value function approximation. My work provides a new, theoretically sound hashing method for linear value function approximation based on prior work on sketches. Empirically, the new hashing method offers a significant performance advantage compared to traditional hashing, at a minuscule computational cost.

My third contribution is the quad-tree factorization (QTF) algorithm, an information-theoretic approach to the problem of predicting future Atari 2600 screens. The algorithm relies on the natural idea that future screens can be efficiently factored into image patches. QTF goes a step further by providing a hierarchical-decomposition screen model, so that image patches are only as large as they need to be.

Together, the contributions in this thesis are motivated by the need to efficiently handle the Atari 2600's large observation space – the set of all possible game screens – in arbitrary Atari 2600 games. This work provides evidence that general, principled approximations can be devised to allow us to tackle the reinforcement learning problem within complex, natural domains.

Acknowledgements

I would like to thank, first and foremost, Rich Sutton for getting me started on this journey and insisting on aiming high, and for many a game of chess; Michael Bowling for carefully, patiently guiding me to the end of this journey; finally Joel Veness for taking the time to show me the finer details of research, and butting heads with me at every occasion. A particular thank you to Bernardo Ávila Pires for helping with some of the proofs of Chapter 5. I would like to thank, too, those with whom I have collaborated over the years, for helping me grow academically: Marc Lanctot, Michael Sokolsky, Adam White, Brian Tanner, Anna Koop, Elliot Ludvig, and Keir Pearson. Gábor Bartók and Agi, for being such fine neighbours and friends and feeding me on every occasion. Doina Precup, for starting me on the path of reinforcement learning research. Last but not least, my other teachers, who bring colour to what would otherwise be a monochromatic life: Patrick Pilarski, Candas Jane Dorsay, Dan Davis, and most especially Judy Loewen.

Contents

List of Tables

List of Figures

> Here then is the only expedient, from which we can hope for success in our philosophical researches, to leave the tedious lingering method, which we have hitherto followed, and instead of taking now and then a castle or village on the frontier, to march up directly to the capital or center of these sciences, to human nature itself; which being once masters of, we may every where else hope for an easy victory.
>
> *A Treatise of Human Nature*
> David Hume

Chapter 1

Introduction

Early in Jack London's novel *White Fang*, one of the characters, Henry, finds himself hunted by a pack of wolves eager to make a meal out of him. As he sets up camp and prepares for a sleepless night in the Yukon wilds, he begins to marvel at the complexity of the human body:

> He watched his moving muscles and was interested in the cunning mechanism of his fingers. [...] He studied the nail-formation, and prodded the finger-tips, now sharply, and again softly, gauging the while the nerve-sensations produced. It fascinated him, and he grew suddenly fond of this subtle flesh of his that worked so beautifully and smoothly and delicately[1].

In the twenty-first century, a less poetic Jack London might have instead found himself concerned with reinforcement learning research. Perhaps then he would have couched his feelings in more succinct terms; he might have wondered at the astoundingly high-dimensional nature of the human brain's observation space. More importantly, he would have reminded his readers that we are rarely aware of this fact, and go about our daily lives as if we were not taking big and small decisions every fraction of a second, decisions that depend on successfully integrating a large number of sensory cues. By contrast, most modern reinforcement learning techniques remain plagued by various forms of the curse of dimensionality (Bellman, 1957): as the number of dimensions in the state, observation or action spaces increases, the computation, memory or sample complexity of such techniques increases exponentially.

The overarching research question behind this thesis is **whether efficient algorithms can be devised for a given set of structured reinforcement learning**

[1] Jack London, *White Fang*, 1906, retrieved from `http://www.gutenberg.org/ebooks/910`.

domains whose structure is unknown but assumed to come from a large class of possible structures. This question is addressed through three independent contributions:

1. Chapter 4 investigates *contingency awareness*: the notion that only a portion of the observation space is under an agent's control.

2. Chapter 5 studies *tug-of-war hashing* as a mean of compactly learning from a large number of possible state features, of which only a few are relevant to any particular domain.

3. Finally, Chapter 6 directly tackles learning generative models of domains with recursively factored observation spaces, in particular domains whose observation space is organized in a image-like manner.

At the center of this work is the Atari 2600 video game platform, which offers us a highly challenging set of reinforcement learning environments. Although challenging, these environments remain simple enough that we can hope to achieve measurable progress as we attempt to solve them. One of the many challenges offered by the Atari 2600 lies in having agents perceive their environment as a 160×210-pixel screen image. While still far from Jack London's "subtle flesh", the Atari 2600 is a sufficiently complex platform to force us to face the challenges of high-dimensionality.

The algorithms presented here share three common traits. They are *fast*: all implemented agents operate at or faster than Atari 2600 real-time, without requiring undue low-level optimizations. These algorithms are also *scalable* by design: in all cases computation is linear in the dimensionality of the observation space. Finally, these algorithms aim to be *domain-independent*, i.e. to be applied to arbitrary Atari 2600 games, where here I consider each game to be a particular domain. The ideal aim of domain-independence, proposed by Naddaf (2010), provides a certain degree of confidence in the generality of these algorithms.

I begin by reviewing basic reinforcement learning ideas (Chapter 2), before describing the Atari 2600 platform and formalizing relevant Atari 2600 notions (Chapter 3). Chapter 4 investigates how knowledge of an agent's contingencies – what it controls – can significantly improve learning. In Chapter 5, I consider the problem of high-dimensionality from the perspective that features generated for linear value

function approximation can often be hashed into a much smaller, and thus more easily learned, space. Finally, in Chapter 6 I propose a new algorithm, quad-tree factorization, for learning forward, probabilistic models of Atari 2600.

Chapter 2

Reinforcement Learning

In reinforcement learning we consider an agent needing to perform sequential decision-making in a known or unknown environment. The agent's goal is to maximize its future rewards; such rewards may be directly provided by the environment (Sutton and Barto, 1998), intrinsically generated (Oudeyer et al., 2007) or even more broadly derived from gathering knowledge about the world (Sutton et al., 2011).

This chapter describes two complementary views of reinforcement learning: the Markov Decision Process (MDP) setting (Puterman, 1994) and the approximate AIXI setting (Hutter, 2005; Veness et al., 2011). In an MDP, the environment dynamics are assumed to be summarized within a *state* variable; knowledge of this state (and of the environment dynamics) is sufficient to predict future rewards. The MDP setting allows us to derive strong algorithmic guarantees, but often obfuscates the reality of complex domains. By contrast, the *approximate AIXI setting* makes no Markov assumption and allows the agent to make decisions based on its whole action-observation history. This flexibility makes it easy to specify many different kinds of learning agents. While real-world approximate AIXI implementations necessarily impose limitations on the class of models they consider, the framework's generality helps the designer to focus on interesting modelling techniques, such as the quad-tree factorization described in Chapter 6.

2.1 Markov Decision Processes

I first review Markov Decision Processes, using material largely drawn from the introductory textbook of Sutton and Barto (1998). Under the MDP setting, an agent's environment is described as an MDP $\mathcal{M}$ whose components are

- a state space $\mathcal{S}$,
- an action space $\mathcal{A}$,
- a transition function $P : \mathcal{S} \times \mathcal{A} \to \textsc{Dist}(\mathcal{S})$,
- a reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and
- a discount factor γ,

where $\textsc{Dist}(\mathcal{X})$ is the space of all probability distributions over the set $\mathcal{X}$. In this thesis, I assume that $\mathcal{S}$ and $\mathcal{A}$ are finite. At each time step t, the agent observes the state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$. The agent then receives a reward $r_t := R(s_t, a_t)$ and a new state s_{t+1} is drawn from the distribution $P(\cdot \mid s_t, a_t)$. In an MDP, state transitions satisfy the Markov property: for all s_{t+1} and any history of states and actions $s_0, a_0, \ldots, s_t, a_t$, we have

$$\Pr(s_{t+1} \mid s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t, a_t) \quad = \quad \Pr(s_{t+1} \mid s_t, a_t) := P(s_{t+1} \mid s_t, a_t).$$

An agent's behaviour is typically described by a *policy* $\pi : \mathcal{S} \to \textsc{Dist}(\mathcal{A})$ mapping states to probability distributions over actions. From any state s_t, the agent's goal is to find a policy that maximizes its *expected return* defined as the expected sum of discounted future rewards:

$$\mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \,\middle|\, s_t\right],$$

where the expectation is over the sequence of future states $\{s_{t+1}, s_{t+2}, \ldots\}$ induced by P and π.

2.1.1 Optimal Policies and Value Functions

For a given policy π and MDP $\mathcal{M}$ there exists a corresponding a *value function* $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ mapping state-action pairs to the expected return achieved by starting in s, taking action a and subsequently following π. Formally,

$$Q^\pi(s, a) := R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s'\right]$$

The Markov property allows us to express $Q^\pi(s, a)$ recursively in a form known as the *Bellman equation*:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s) Q^{\pi}(s',a') \tag{2.1}$$

An *optimal policy* π^* maximizes the expected return at every state. The *Bellman optimality equation* captures this fact by replacing the expectation above with the maximum operator in order to define the (unique) optimal value function:

$$Q^{*}(s,a) = R(s,a) + \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q^{*}(s',a') \tag{2.2}$$

Knowing $Q^*(s,a)$ allows an agent to behave optimally by constructing π^* such that

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q^*(s,a') \\ 0 & \text{otherwise} \end{cases},$$

where ties in $\arg\max_{a' \in \mathcal{A}} Q^*(s,a')$ are assumed to be broken arbitrarily. As learning $Q^*(s,a)$ is sufficient to obtain π^*, many reinforcement learning approaches focus on estimating the former; such approaches are called *value-based*. In particular, the SARSA(λ) algorithm (Rummery, 1995; Sutton and Barto, 1998), described next, learns a value function from state transitions.

2.1.2 SARSA(λ): Learning from Experience

Often the environment dynamics are unknown and the agent must learn its policy from experience. In this case, value-based reinforcement learning typically involves the repetition of two steps: (1) learning $Q^{\pi}(s,a)$ from interactions with the environment, for a fixed π, and (2) improving the current policy π. The SARSA(λ) algorithm performs the first of these tasks[1].

Given a fixed policy π, SARSA(λ) produces a sequence of value function estimates $\{Q_t^{\pi}(s,a)\}$; under appropriate conditions, $Q_t^{\pi}(s,a)$ converges to $Q^{\pi}(s,a)$ (Rummery, 1995). For each observed transition $s_t, a_t, r_t, s_{t+1}, a_{t+1}$, SARSA($\lambda$) performs the following update:

[1] In practice, the policy improvement step is often reduced to *ϵ-greedy action selection*: the agent selects actions uniformly at random with probability ϵ and otherwise acts greedily with respect to $Q^{\pi}(s,a)$.

$$
\begin{aligned}
\delta_t &\leftarrow r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \\
e_t(s, a) &\leftarrow \gamma\lambda e_{t-1}(s, a) + \mathbb{I}_{[s=s_t, a=a_t]} \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \\
Q_{t+1}(s, a) &\leftarrow Q_t(s, a) + \alpha_t \delta_t e_t(s, a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A},
\end{aligned}
$$

where $\alpha_t \in [0, 1)$ is a step-size parameter, $\lambda \in [0, 1]$ is an eligibility trace parameter, $e_t(s, a)$ is an eligibility trace with $e_0(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$, and $\mathbb{I}_{[.]}$ is the indicator function. Note that, at each time step, the eligibility trace for *every* state-action pair (s, a) is updated by SARSA(λ). The term δ_t is known as the *temporal-difference error*: the difference between the predicted return $Q_t(s_t, a_t)$ and the one-step estimate of the return, $r_t + \gamma Q_t(s_{t+1}, a_{t+1})$. The purpose of the λ parameter is to propagate the temporal-difference error backwards in time. On one end of the spectrum, $\lambda = 1$ corresponds to estimating $Q^\pi(s, a)$ using Monte-Carlo samples of the return; $\lambda = 0$ instead bootstraps each estimate from its successor. Varying λ trades off bias and variance, with intermediate values often performing best (Sutton, 1996). Because $Q_t^\pi(s, a)$ is stored explicitly in a table, this method is referred to as *tabular SARSA*(λ).

2.1.3 Linear Value Function Approximation

In most reinforcement learning problems, explicitly storing a value function for all state-action pairs is undesirable: there are often more states than what can be stored in memory. In discrete domains, this issue usually arises because the state is described by a set of features, so that the size of the state space is exponential in the number of such features[2]. In this case, we must approximate the value function estimates. Linear approximation is one such value function approximation scheme, of particular interest because it has been well-studied (Tsitsiklis and Van Roy, 1997; Boyan, 2002; Sutton et al., 2008), is stable, and can be efficiently implemented. Given $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$ mapping state-action pairs to feature vectors, the value function estimate $Q_t^\pi(s, a)$ is linearly approximated as $\theta_t \cdot \phi(s, a)$, where $\theta_t \in \mathbb{R}^n$ is the weight vector at time t. The gradient descent SARSA(λ) update is defined as:

[2] Assuming that each feature takes on finitely many values.

$$
\begin{aligned}
\delta_t &\leftarrow r_t + \gamma\theta_t \cdot \phi(s_{t+1}, a_{t+1}) - \theta_t \cdot \phi(s_t, a_t) \\
e_t &\leftarrow \gamma\lambda e_{t-1} + \phi(s_t, a_t) \\
\theta_{t+1} &\leftarrow \theta_t + \alpha_t \delta_t e_t,
\end{aligned}
$$

where $\alpha_t \in (0, 1)$ is a step-size parameter, $\lambda \in [0, 1]$ is an eligibility trace parameter and $e_t \in \mathbb{R}^n$ is an eligibility trace vector.

2.1.4 Convergence of SARSA(λ)

As discussed in Section 2.1.2, the value function estimate $Q_t^\pi(s, a)$ learned by tabular SARSA(λ) converges to $Q^\pi(s, a)$ under certain conditions on the step-size α_t, the policy π and the MDP $\mathcal{M}$ (Rummery, 1995; Singh and Sutton, 1996). Under a similar set of assumptions, SARSA(λ) with linear value function approximation also converges, albeit to a solution whose approximation error depends on the choice of ϕ and λ (Tsitsiklis and Van Roy, 1997). I now review a version of this convergence result pertaining to state-action values.

Together, a fixed policy π and a MDP $\mathcal{M}$ form a Markov chain whose states are (s, a) pairs and the transition probability between (s, a) and (s', a') is given by $P(s'|s, a)\pi(a'|s')$. Let $\Phi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times n}$ be the matrix of feature vectors $\phi(s, a)$. I begin with a set of assumptions.

Assumptions

1. The Markov chain induced by π and $\mathcal{M}$ is ergodic and has a unique stationary distribution $\mu \in \textsc{Dist}(\mathcal{S} \times \mathcal{A})$,

2. Φ has full column rank (there are no redundant features), and

3. The step-sizes α_t are positive, nonincreasing and predetermined; furthermore, $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$.

Denote by $\langle \cdot, \cdot, \rangle_\mu$ the inner product induced by μ, i.e. $\langle x, y \rangle_\mu := x^T D y$, where $x, y \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ and $D \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}||\mathcal{A}|}$ is a diagonal matrix with entries $\mu(s, a)$. The norm $\|\cdot\|_\mu$ is defined as $\sqrt{\langle \cdot, \cdot \rangle_\mu}$. The following theorem, originally presented by Tsitsiklis and Van Roy (1997), bounds the error of SARSA(λ) with linear value function approximation:

Theorem 2.1.1. *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ be an MDP and $\pi : \mathcal{S} \to \text{DIST}(\mathcal{A})$ be a policy. Denote by $\Phi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times n}$ the matrix of full feature vectors and by μ the stationary distribution on $(\mathcal{S}, \mathcal{A})$ induced by π and P. Under assumptions 1-3), SARSA(λ) converges to a unique $\theta^\pi \in \mathbb{R}^n$ with probability one and this θ^π satisfies*

$$\|\Phi\theta^\pi - Q^\pi\|_\mu \leq \frac{1 - \lambda\gamma}{1 - \gamma} \|\mathbf{\Pi} Q^\pi - Q^\pi\|_\mu ,$$

where $Q^\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is a vector representing the exact solution to Equation 2.1 and $\mathbf{\Pi} := \Phi(\Phi^T D \Phi)^{-1} \Phi^T D$ is the projection operator.

Theorem 2.1.1 states that SARSA(λ) with linear value function approximation converges to the nearest approximation to $Q^\pi(s, a)$ within the space of value functions $\{\Phi\theta : \theta \in \mathbb{R}^n\}$, up to a multiplicative factor depending on λ and γ; the error in value for each state-action pair (s, a) is weighted by the pair's relative stationary distribution frequency $\mu(s, a)$.

2.2 The AIXI Setting

The Markov assumption simplifies the design and theoretical validation of reinforcement learning algorithms; however, in real-world applications it is often what its name implies: a theoretical assumption rather than a property of the domain. An alternative, the AIXI setting (Hutter, 2005), obviates the question of Markov state and instead deals directly with the sequence of past observations, actions and rewards. This perspective on the reinforcement learning problem allows us to emphasize the need for more general (non-Markov) agents. In this section I focus on the AIXI description of environments and environment models necessary to the work of Chapters 4 and 6 and omit the decision-making component for clarity. For a gentle introduction to the full AIXI setting, see Chapter 2 in (Legg, 2008); most of the ideas described in this section stem from the presentation by Veness et al. (2011).

2.2.1 Environments

In the AIXI setting, an environment is defined using

- An observation space $\mathcal{O}$,
- An action space $\mathcal{A}$, and
- A reward space $\mathcal{R}$.

Again, I assume here that these spaces are finite. Observations and rewards are grouped together as *percepts* drawn from the percept space $\mathcal{X} := \mathcal{O} \times \mathcal{R}$. The history space is denoted by $\mathcal{H} := (\mathcal{A} \times \mathcal{X})^* \cup (\mathcal{A} \times \mathcal{X})^* \times \mathcal{A}$. A sequence of percepts $x_1, x_2, \ldots, x_n$ is denoted by $x_{1:n}$, with $x_{<n}$ denoting the prefix $x_{1:n-1}$ and ϵ denoting the empty sequence. Sequences of actions are similarly denoted by $a_{1:n}$ and histories by $(ax)_{1:n}$. An *environment* ρ is a sequence of conditional probability functions $\{\rho_0, \rho_1, \ldots\}$, with $\rho_i : \mathcal{A}^n \to \textsc{Dist}(\mathcal{X}^n)$, constrained by the *chronological condition*:

$$\forall a_{1:n} \forall x_{1:n-1} : \rho_{n-1}(x_{<n} \,|\, a_{<n}) = \sum_{x_n \in \mathcal{X}} \rho_n(x_{1:n} \,|\, a_{1:n}), \tag{2.3}$$

with $\rho_0(\epsilon|\epsilon) = 1$; $\rho_n(x_{1:n}|a_{1:n})$ can naturally be interpreted as the probability of observing $x_{1:n}$ if the agent were to execute a fixed action sequence $a_{1:n}$[3]. The subscript to ρ_n is always implied by its arguments, and is dropped from here onwards. Throughout I assume that, for any sequence $x_{1:n} \in \mathcal{X}^n$ and any $a_{1:n} \in \mathcal{A}^n$, $\rho(x_{1:n} \,|\, a_{1:n}) > 0$.

The probability of observing x_n given $(ax)_{<n}$ and a_n is defined as

$$\rho\left(x_n \,|\, (ax)_{<n} a_n\right) := \frac{\rho(x_{1:n} \,|\, a_{1:n})}{\rho(x_{<n} \,|\, a_{<n})} \tag{2.4}$$

This notion of environment is used in two distinct ways. It is first used to describe the true, underlying dynamical system that generates the string of percepts $x_{1:n}$. It also describes an agent's *environment model*. In the AIXI setting, an environment model provides a probability distribution over future symbols and this distribution depends on the observed history. In general, we do not have access to the environment; one question of interest is then how to learn an approximate description of its dynamics, i.e. an environment model.

2.2.2 Redundancy

Given an environment model ρ and an environment μ, we quantify the accuracy of ρ's predictions about the string $x_{1:n}$ by measuring its *redundancy* with respect to μ, defined as

$$-\log_2 \rho(x_{1:n} \,|\, a_{1:n}) - (-\log_2 \mu(x_{1:n} \,|\, a_{1:n}))$$

[3]Note that $a_{1:n}$ is not a random variable, but rather a parameter to the mapping $\rho_n(\cdot \,|\, \cdot)$. In this setting, therefore, the query $\Pr(a_{1:n}|x_{1:n})$ cannot be answered through ρ_n.

The redundancy of ρ is the additional number of bits required to encode $x_{1:n}$ using the probability distribution of ρ rather than the true model μ. If we further consider the expected redundancy when $x_{1:n}$ is drawn from μ, we obtain the Kullback-Leibler divergence of $\rho(\cdot|a_{1:n})$ from $\mu(\cdot|a_{1:n})$:

$$D_{KL}(\mu \,\|\, \rho) := \mathbb{E}_{x_{1:n}\sim\mu}\left[\frac{\log_2 \mu(x_{1:n} \,|\, a_{1:n})}{\log_2 \rho(x_{1:n} \,|\, a_{1:n})}\right].$$

Note that the Kullback-Leibler divergence is not symmetric, i.e. in general $D_{KL}(\mu \,\|\, \rho) \neq D_{KL}(\rho \,\|\, \mu)$.

2.2.3 Mixture Environment Models

Often we are given a class $\mathcal{M}$ of environment models whose predictions we would like to combine so as to guarantee that the resulting environment model is as good as the best-performing model in $\mathcal{M}$. *Bayesian model averaging* is one way to achieve this goal. Informally, model averaging weighs each model $\rho \in \mathcal{M}$ in proportion to how accurate its predictions are. Initially, each $\rho \in \mathcal{M}$ is assigned a prior weight w_0^ρ, leading to the following definition:

Definition 2.2.1. *Given a finite model class $\mathcal{M}$ and for each model $\rho \in \mathcal{M}$ a prior $w_0^\rho > 0$ such that $\sum_{\rho\in\mathcal{M}} w_0^\rho = 1$, the* mixture environment model *ξ is defined as*

$$\xi(x_{1:n} \,|\, a_{1:n}) := \sum_{\rho\in\mathcal{M}} w_0^\rho \rho(x_{1:n} \,|\, a_{1:n})$$

A mixture environment model satisfies Equation 2.3, and so is also an environment model. This equivalence can be derived as follows:

$$\begin{aligned}
\forall a_{1:n} \forall x_{1:n} : \sum_{x_n\in\mathcal{X}} \xi(x_{1:n} \,|\, a_{1:n}) &= \sum_{x_n\in\mathcal{X}} \sum_{\rho\in\mathcal{M}} w_0^\rho \rho(x_{1:n} \,|\, a_{1:n}) \\
&= \sum_{\rho\in\mathcal{M}} w_0^\rho \sum_{x_n\in\mathcal{X}} \rho(x_{1:n}|a_{1:n}) \\
&= \sum_{\rho\in\mathcal{M}} w_0^\rho \rho(x_{<n}|a_{<n}) \\
&= \xi(x_{<n} \,|\, a_{<n}).
\end{aligned}$$

The probability of observing x_n given $(ax)_{<n}$ and a_n under ξ is therefore

$$\xi(x_n \,|\, (ax)_{<n} a_n) = \frac{\xi(x_{1:n} \,|\, a_{1:n})}{\xi(x_{<n} \,|\, a_{<n})}$$

Given a model class $\mathcal{M}$, a prior weight $w_0^\rho > 0$ for each $\rho \in \mathcal{M}$ and an arbitrary $\rho' \in \mathcal{M}$, we can bound the redundancy of the environment mixture model as follows:

$$-\log_2 \xi(x_{1:n} \,|\, a_{1:n}) = -\log_2 \sum_{\rho \in \mathcal{M}} w_0^\rho \rho(x_{1:n} \,|\, a_{1:n}) \leq -\log_2 w_0^{\rho'} - \log_2 \rho'(x_{1:n} \,|\, a_{1:n}). \tag{2.5}$$

In particular, Equation 2.5 guarantees that on a per-symbol prediction basis, a mixture environment model asymptotically performs as well as the best model in $\mathcal{M}$. This follows by considering the average log-likelihood of each symbol under ξ and ρ':

$$\lim_{n\to\infty} \left[-\frac{1}{n} \log_2 \xi(x_{1:n} \,|\, a_{1:n}) \right] \leq \lim_{n\to\infty} \left[-\frac{1}{n} \log_2 w_0^{\rho'} - \frac{1}{n} \log \rho'(x_{1:n} \,|\, a_{1:n}) \right] \tag{2.6}$$

where the term $-\frac{1}{n} \log_2 w_0^{\rho'}$ vanishes as $n \to \infty$.

Chapter 3

The Arcade Learning Environment

The

contributions in this thesis are all evaluated using the *Arcade Learning Environment*, a set of reinforcement learning domains based on Atari 2600 games (Bellemare et al., 2013a). The aim of this chapter is to

1. Describe the Atari 2600 platform and the Arcade Learning Environment,
2. Provide Atari-specific notation used in later chapters, and
3. Describe my contributions to the problem of multi-domain evaluation, in particular to the question of how to best compare agents evaluated on a set of Atari 2600 games.

This chapter focuses on details and concepts necessary to understand the other contributions of this thesis; further precision, discussion, and exhaustive benchmark results can be found in the journal article on the Arcade Learning Environment (Bellemare et al., 2013a).

3.1 The Atari 2600

The Atari 2600 is a home video game console developed in 1977 and sold in retail stores for over a decade (Montfort and Bogost, 2009). The Atari 2600 popularized the use of general-purpose CPUs in game console hardware, with game code distributed through cartridges. Over 500 original games were released for the console;

A version of this chapter has been published (Bellemare et al., 2013a).

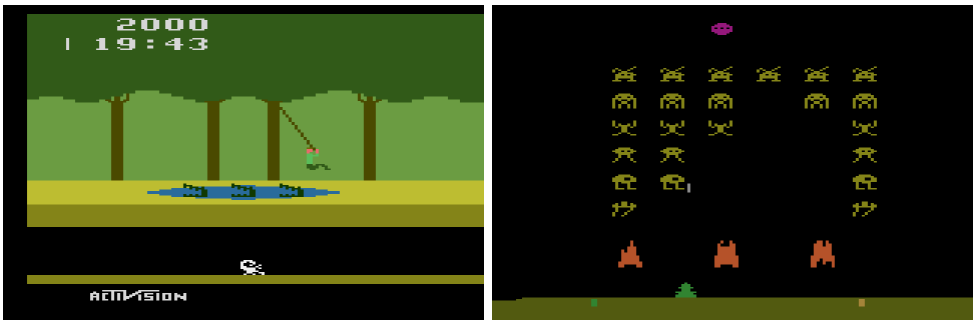

Figure 3.1: Screenshots of PITFALL! and SPACE INVADERS.

"homebrew" games continue to be developed now, over thirty years later. The games vary widely and include puzzles, sports, action-adventure games, board games, as well as a plethora of shooter variants. The console's joystick, as well as some of the original games such as ADVENTURE and PITFALL!, are iconic symbols of early video games. Nearly all arcade games of the time – PAC-MAN and SPACE INVADERS are two well-known examples – were ported to the console.

Despite the number and variety of games developed for the Atari 2600, its hardware is relatively simple. It has a 1.19MHz CPU and can be emulated much faster than real-time on modern hardware. The cartridge ROM (typically 2–4kB) holds the game code, while the console RAM itself only holds 128 bytes (1024 bits). A single game screen is 160 pixels wide and 210 pixels high with pixel colours drawn from a 128-colour palette. A one-button, two-axis digital joystick provides input to most games; other controllers, including an analog "paddle" for PONG-like games, are also available. While human-level performance across a wide range of Atari 2600 games has yet to be demonstrated using current artificial intelligence methods, the platform's hardware limitations ensures that these games are not too complex. As such, studying the limitations of applying existing methods to Atari 2600 games should conceivably lead to near-term advancements in learning, modelling, and planning.

3.1.1 The Arcade Learning Environment

The Arcade Learning Environment (ALE), originally developed by Yavar Naddaf as part of his Master's work (Naddaf, 2010) and subsequently improved throughout my thesis work, provides a reinforcement learning interface around Atari 2600 games. ALE is built on top of Stella[1], an open-source Atari 2600 emulator. It allows the user to interface with the Atari 2600 by receiving joystick motions, sending screen

[1] `http://stella.sourceforge.net/`

and RAM information, and emulating the platform. ALE also provides a game-handling layer which transforms each game into a standard reinforcement learning problem by identifying the accumulated score and whether the game has ended. By default, each observation consists of a single game screen (frame): a two-dimensional array of 7-bit pixels, 160 pixels wide by 210 pixels high. The action space consists of the 18 discrete actions defined by the joystick controller. Analog controllers such as the well-known paddle are approximated from the discrete action set using simple differential equations. The Atari 2600 generates 60 frames per second of real-time play, and at full speed emulates up to 6000 frames per second on a modern laptop computer. The reward at each time-step is defined on a game by game basis, typically by taking the difference in score or points between frames. An episode begins on the first frame after a reset command is issued, and terminates when the game ends. The game-handling layer also offers the ability to end the episode after a predefined number of frames; this functionality is needed for a small number of games to ensure that they always terminate. This prevents situations such as in TENNIS, where a degenerate agent could choose to play indefinitely by refusing to serve. The Arcade Learning Environment is open-source and publicly available[2].

3.1.2 Domain-independent Agents

In his Master's thesis, Naddaf (2010) proposed the concept of domain-independent Atari 2600 agents: agents that can play arbitrary Atari 2600 games without game-specific tuning. The appeal of such agents is twofold: first, they are by their very nature a step closer to "big" general intelligence (e.g., Russell, 1997; Hutter, 2005; Legg, 2008); second, the domain-independent requirement helps guarantee the broad usefulness of developed algorithms. The domain-independent methodology (Bellemare et al., 2013a) is based on three principles:

1. **Use many games.** Agents should be tested on a large number of domains.
2. **Diversify games.** Domains should exhibit different environment dynamics.
3. **Divide into training/testing.** Games should be designed using a small fraction of domains (the training set), and later evaluated using many domains (the testing set).

[2] `http://www.arcadelearningenvironment.org`

The Arcade Learning Environment provides us with a large collection of Atari 2600 games; this collection constitutes our set of reinforcement learning domains. In particular, algorithms in Chapters 4, 5, and 6 were respectively evaluated using 51, 55 and 20 games. Games were drawn at random from a larger set (Naddaf, 2010) and range from map-based adventure games to three-dimensional shooters. The same five training games[3] were used throughout.

3.2 Evaluating Metrics for General Atari 2600 Agents

Evaluating agents on many games, albeit desirable, poses an interpretation problem. While in all games, the agent's goal is to maximize its score, scores for different games cannot be easily compared. Each game uses its own scale for scores, and different mechanics make some games harder to learn than others. This section describes the approach taken in this thesis: scores are first normalized, then aggregated. Normalizing scores is necessary to perform inter-game comparisons, while aggregation allows us to concisely give results across the full set of games used.

3.2.1 Score Normalization

Let $s_{g,1}$ and $s_{g,2}$ denote the scores achieved by two algorithms in a particular game g. The aim of score normalization is to translate two sets of scores $S_1 = \{s_{g_1,1}, \ldots, s_{g_n,1}\}$ and $S_2 = \{s_{g_1,2}, \ldots, s_{g_n,2}\}$ so that they can be compared across games. Each score $s_{g,i}$ is transformed into a normalized score $z_{g,i}$; in the ideal case, $z_{g,i} = z_{g',i}$ implies that algorithm i performs as well on game g as on game g'. The two normalization methods described below are defined using a *score range* $[r_{g,\min}, r_{g,\max}]$ computed for each game. Given a score range, the corresponding normalized score is defined as

$$z_{g,i} := \frac{s_{g,i} - r_{g,\min}}{r_{g,\max} - r_{g,\min}}$$

The Baseline Score

We can often easily design and evaluate agents that act according to trivial policies, for example the random policy. Although we expect them to perform poorly across the set of Atari 2600 games, these agents constitute an interpretable performance

[3] Asterix, Beam Rider, Freeway, Seaquest, and Space Invaders.

baseline. Given a set of baseline agents (acting according to trivial policies or not), the range of scores they achieve on a particular game naturally defines a score range for that game. Let $b_{g,1}, \ldots, b_{g,k}$ be a set of reference scores. A method's *baseline score* is computed using the score range $[\min_{i \in \{1,\ldots,k\}} b_{g,i}, \max_{i \in \{1,\ldots,k\}} b_{g,i}]$. In this thesis, baseline scores are obtained using 37 policies:

1. **Random.** Actions are selected uniformly at random,
2. **Constant.** The same action is repeated at every time step (one policy per action, for a total of 18 policies), and
3. **Perturb.** At every time step, the agent takes a fixed action with probability 0.95 and otherwise acts uniformly at random (one policy per action, for a total of 18 policies).

Incorporating scores achieved by human players into the baseline score range is desirable for interpretability purposes, but raises a wealth of issues. Humans often play games without seeking to maximize score; humans also benefit from prior knowledge that is difficult to incorporate into domain-independent agents. More importantly, selecting an appropriate range of scores (or an appropriate skill level) poses a significant challenge, as scores achieved by humans vary considerably from player to player.

The Inter-Algorithm Score

An alternative to baseline normalization is to define a score range based on the scores achieved by the algorithms being evaluated. Given k algorithms, each achieving score $s_{g,i}$ on game g, $1 \leq i \leq k$, the *inter-algorithm score* is defined using the score range $[\min_{i \in \{1,\ldots,k\}} s_{g,i}, \max_{i \in \{1,\ldots,k\}} s_{g,i}]$. By definition, $z_{g,i} \in [0, 1]$; a special case of this kind of normalization is when $k = 2$, where $z_{g,i} \in \{0, 1\}$ indicates which of the two compared algorithms performs better.

Because inter-algorithm scores are bounded, they are often more easily interpreted than baseline scores (see Chapter 4 for a comparison of the two methods). However, inter-algorithm score normalization gives no indication of the objective performance of the best algorithm. A good example of this is the game VENTURE, for which none of the agents described in this thesis achieves a meaningful score. Inter-algorithm scores are thus best used to complement other scoring metrics.

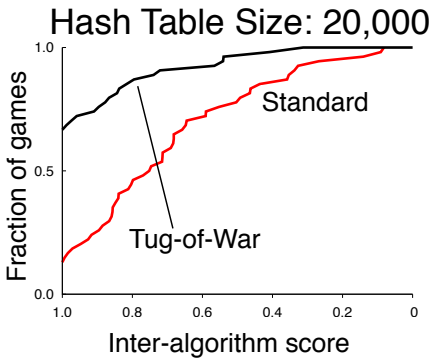

Figure 3.2: A score distribution (taken from Chapter 5) over fifty-five games.

3.2.2 The Score Distribution

Once normalized scores are obtained for each game, the next step is to produce measures that reflect how well each agent performs across the set of games. Two simple measures are the average and median normalized scores. Although easily comparable, both average and median hide the details of how algorithms perform, are unduly affected by outliers, and in general paint a picture that is difficult to interpret. On the other hand, simply reading a large table of scores is cumbersome. An alternative is to present the set of normalized scores as a *score distribution*. The score distribution is a natural generalization of the median: it shows the fraction of games on which an algorithm achieves a certain normalized score or better. It is essentially a quantile plot or inverse empirical CDF. Figure 3.2 presents some of the empirical results of Chapter 5, comparing two methods (tug-of-war hashing and standard hashing), as a score distribution. Here the tug-of-war method performs better, and so achieves good scores on a larger fraction of games compared to standard hashing.

3.3 Notation

This section provides the notation necessary to mathematically describe Atari 2600 games as reinforcement learning environments. In later chapters, this notation is used to instantiate various theoretical results to Atari 2600 domains. Although for practical reasons some of the terms below explicitly refer to components of Atari 2600 games, the aim of this notation is to describe the general class of domains whose

observation space exhibits two-dimensional structure, i.e. image-like observations.

Let $\mathcal{O}$, the observation space, be the set of all possible Atari 2600 images. Denote by $\mathcal{D}_x \subset \mathbb{N}$ and $\mathcal{D}_y \subset \mathbb{N}$ the set of row and column indices, respectively, and let $\mathcal{D} := \mathcal{D}_x \times \mathcal{D}_y$ be the joint index space. The colour space $\mathcal{C}$ is a finite set of possible pixel colours. A pixel is a tuple $(x, y, c) \in \mathcal{D} \times \mathcal{C} =: \mathcal{P}$ the pixel space. Each observation is therefore a set of $|\mathcal{D}|$ pixels. Whenever convenient, I also use the notation $o_{x,y}$ to denote the colour at location (x, y) in observation $o \in \mathcal{O}$: $o_{x,y} = c$ such that $(x, y, c) \in o$. When a time index is given, as in o_t, the notation is $o_{t,x,y}$.

The action space $\mathcal{A}$ of the Atari 2600 consists of 18 discrete actions, corresponding to joystick motions along the x and y axes and button presses. Throughout this work I ignore the factorization of $\mathcal{A}$ into its components and treat $\mathcal{A}$ as a set of atomic actions.

The reward space $\mathcal{R} \subset \mathbb{R}$ is bounded above and below by the largest and smallest achievable rewards. Note that, by virtue of the finite state machine nature of the Atari 2600, the reward space is also finite. In effect, all games considered in this thesis have integer rewards, though this is not a requirement of the proposed algorithms.

Much of my experimental work relies on the decomposition of the image into *image patches*, which often provide more information than individual pixels. In this work an image patch is a rectangular portion of the whole image; image patches are defined as sets of pixels. the Let $(a, b) \in \mathbb{R} \times \mathbb{R}$ be a pair representing the width and height of an image patch. A patch $p^{a,b}_{x,y} : \mathcal{O} \to 2^{\mathcal{P}}$ selects a subset of the image:

$$p^{a,b}_{x,y}(o) := \left\{(x', y', c) : (x', y', c) \in o, x \leq x' < x + a, y \leq y' < y + b\right\}$$

Similarly we define the subset of locations $l^{a,b}_{x,y} : \mathcal{O} \to 2^{\mathcal{D}}$ corresponding to the patch $p^{a,b}_{x,y}(o)$:

$$l^{a,b}_{x,y}(o) := \left\{(x', y') : \exists c \in \mathcal{C} s.t. (x', y', c) \in o, \text{ and } x \leq x' < x + a, y \leq y' < y + b\right\}$$

Each image patch of size $a \times b$ corresponds to a unique lexicographic index $\imath^{a,b}_{x,y}(o) \in \mathbb{N}$ ranging from 0 to $|\mathcal{C}|^{a \times b} - 1$. This index is constructed by iterating through the patch's pixels in row-column order and encoding each pixel as a 7-bit value. Formally:

$$\imath_{x,y}^{a,b}(o) := \sum_{y'=y}^{y+b-1} \sum_{x'=x}^{x+a-1} |\mathcal{C}|^{a(y'-y)+(x'-x)} \text{bit}(o_{x',y'}) \ ,$$

where $\text{bit} : \mathcal{C} \to \{0, \ldots, |\mathcal{C}| - 1\}$ denotes a mapping from colours to indices. Out-of-screen pixels are encoded using a special colour in $\mathcal{C}$.

3.4 Related Work

Diuk et al. (2008) were the first to experiment with the Atari 2600 domain. Their research on object-oriented reinforcement learning focused on a restricted version of the game of Pitfall!, in which the agent's goal is to exit the screen to the right. Wintermute (2010) similarly studied how to extract objects from Atari 2600 game screens, focusing on a restricted version of Frogger II; once extracted, the objects were then embedded into a logic-based architecture (SOAR) for planning purposes. Cobo et al. (2011) investigated automatic feature discovery using their own implementations of Pong and Frogger. Stober and Kuipers (2008) also used their own version of Pong to study an algorithm for the lifelong learning setting, in which the agent seeks to organize raw sensory information into a high-level representation of its environment. Hausknecht et al. (2012) proposed an algorithm, the Visual Processing Architecture, whose purpose is to extract high-level features – object labellings – from Atari 2600 screens. Their procedure improves on the DISCO algorithm (Naddaf, 2010) by adding the ability to cluster objects based on pixel similarity and including a form of self-detection. They evaluated their architecture, including a generic algorithm-based policy search algorithm, on Asterix and Freeway. Finally, Talvitie and Singh (2008) studied predictions in partially observable domains using a simplified version of Breakout.

The research on Atari 2600 games (and clones) described above has collectively shown the critical importance of algorithms that can handle large, factored observation spaces. However, common to this research is a focus on a very small number of games – here, one or two. Often this focus stems from a need to engineer aspects of the problem, for example labelling Pitfall! objects in the work of Diuk et al. (2008). Emphasizing the development of domain-independent agents by considering a large number of games is appealing because it diminishes the benefits of domain engineering, if for no other reason than the sheer amount of work required to engineer fifty or sixty different domains.

> At once Stockstill thought, He's taking credit for it, in his own mind. Paranoid delusions of omnipotence; everything that takes place is due to him.
>
> *Dr. Bloodmoney*
> Philip K. Dick

Chapter 4

Contingency Awareness

Large domains typically exhibit regularities in their observation and action spaces which facilitate decision-making. In Atari 2600 games, one interesting source of regularity is pixel motion: most pixels within a game screen "move" from frame to frame. Furthermore, most of this motion is independent of the agent's choice of action. Thus in Freeway (Figure 4.1) the agent controls the left-hand chicken but not the motion of the cars. This idea is captured by the notion of *contingency awareness*, which this chapter introduces and formalizes.

Contingency awareness describes the agent's knowledge of which parts of the observation are under its control, and which are solely determined by the environment. Contingency awareness is a natural notion when used to describe human behaviour: most humans are keenly aware that the better part of their world acts independently of their will. The formalism that I present here pertains to a notion of contingency awareness that is directly related to learning agents acting in factored domains.

Following the notation of Section 3.3, I use the term *pixel* to denote one of the atomic components of a larger factored observation space. Section 4.1 first defines the notion of contingent regions: the subset of pixels whose colour depends on the agent's action. I then describe supervised learning models that map observations to contingent regions (Section 4.2), and subsequently how to make use of such predictions to design better features for linear value function approximation (Sections 4.3 and 4.4). Finally, these ideas are empirically validated in Section 4.5.

A version of this chapter has been published and presented at the AAAI Conference on Artificial Intelligence (Bellemare et al., 2012a).

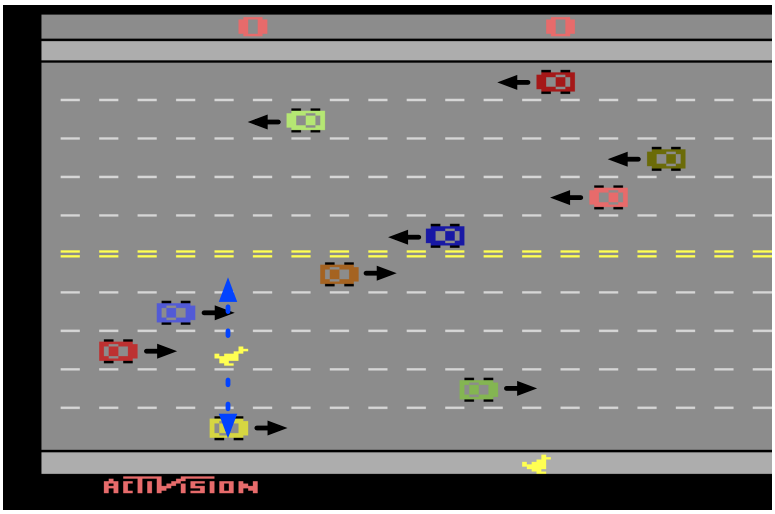

Figure 4.1: Regularity in the game of Freeway. The agent controls the chicken (blue dotted arrows) but not the motion of the cars (solid black arrows).

4.1 Contingent Regions

Recall the AIXI setting described in Section 2.2: at every step the agent observes a percept $x_t \in \mathcal{X}$, where $\mathcal{X} = \mathcal{O} \times \mathcal{R}$, and takes an action $a_t \in \mathcal{A}$. In this setting, the history $h_t := a_1 x_1 a_2 x_2 \dots x_t$ contains all of the information available to the agent. Recall also that an environment ρ maps histories $h \in \mathcal{H}$ to probability distributions. To simplify the description of contingent regions, I define a *deterministic environment* as a mapping $\rho : (\mathcal{A} \times \mathcal{X})^* \times \mathcal{A} \to \mathcal{X}$ and take $\mathcal{X} = \mathcal{O}$.

Informally, the contingent regions at time t is the set of pixels whose colour at time $t+1$ depends on a_t. When the environment is deterministic and can be simulated, computing the contingent regions can be done efficiently. The first definition of contingent regions reflects this fact:

Definition 4.1.1. *Given a history $h \in (\mathcal{A} \times \mathcal{X})^*$ and a deterministic environment $\rho : (\mathcal{A} \times \mathcal{X})^* \times \mathcal{A} \to \mathcal{X}$, the contingent regions $C(h)$ of history h are defined as*

$$C(h) := \left\{(x, y) \in \mathcal{D} : \exists a, a' \in \mathcal{A}, o_{x,y}(ha) \neq o_{x,y}(ha')\right\},$$

where $\mathcal{D}$ is the joint index space and $o_{x,y}(ha)$ denotes the colour of the pixel at location (x, y) within the observation $\rho(ha)$.

This definition is called the *oracle* contingent regions to reflect the fact that computing it requires more information than is usually available to the agent: it

requires testing all actions $a \in \mathcal{A}$ from h. Figure 4.2 depicts the process through which $C(h)$ can be computed in Atari 2600 games.

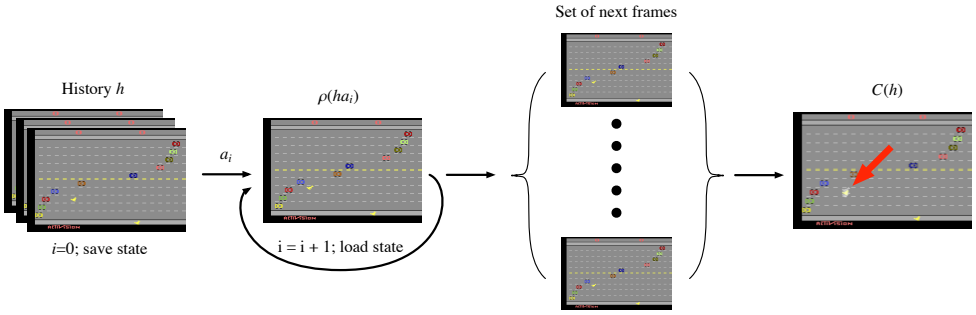

Figure 4.2: Process for generating the oracle contingent regions $C(h)$. At the core of the process is the ability to reset the simulator to a given state (corresponding to a particular history). On the right, $C(h)$ is represented as a white overlay on top of the normal screen (the red arrow indicates the location of $C(h)$).

Definition 4.1.1 can be revised to account for stochastic environments by replacing the pixel comparison $o_{x,y}(ha) \neq o_{x,y}(ha')$ by a comparison between distributions over pixel colours: $T_{x,y}(\cdot|ha) \neq T_{x,y}(\cdot|ha')$, where $T_{x,y}(\cdot|ha)$ refers to the distribution over the colour of the pixel at (x, y). Given a stochastic environment $\rho : (\mathcal{A} \times \mathcal{X})^* \times \mathcal{A} \to \textsc{Dist}(\mathcal{X})$, this pixel colour distribution is defined as

$$T_{x,y}(c|ha) := \sum_{z \in \mathcal{X}} \rho(z|ha)\mathbb{I}_{[(x,y,c)\in z]},$$

leading to the second definition of contingent regions:

Definition 4.1.2. *Given a history $h \in (\mathcal{A} \times \mathcal{X})^*$ and a stochastic environment $\rho : (\mathcal{A} \times \mathcal{X})^* \times \mathcal{A} \to \textsc{Dist}(\mathcal{X})$, the* online *contingent regions $C(h)$ of history h are defined as*

$$C(h) := \left\{(x, y) \in \mathcal{D} : \exists a, a' \in \mathcal{A}, T_{x,y}(\cdot \,|\, ha) \neq T_{x,y}(\cdot \,|\, ha')\right\},$$

where $T_{x,y}(\cdot|ha)$ denotes the distribution over colours at (x, y).

The second definition is called the online contingent regions to reflect that, unlike Definition 4.1.1, it can be computed from a learned observation model[1]. When

[1]The term *online*, also used in Bellemare et al. (2012a), reflects the desire to learn the contingent regions without a preparatory model learning period. In the experiments presented here, however, we separated model learning and contingent regions learning in order to simplify the experimental setup.

the true pixel colour distribution $T_{x,y}(\cdot|ha)$ is unavailable, Definition 4.1.2 can be approximated by replacing each $T_{x,y}(\cdot|ha)$ with a learned approximation $\tilde{T}(\cdot|ha)$ and the non-equality test with a KL divergence test:

$$\tilde{C}(h) := \left\{(x,y) \in \mathcal{D} : \exists a, a' \in \mathcal{A}, D(\tilde{T}_{x,y}(\cdot|ha) \,||\, \tilde{T}_{x,y}(\cdot|ha')) \geq \delta\right\}, \tag{4.1}$$

where $\delta \in (0, \infty)$ is a threshold parameter and $D(p||q)$ denotes a symmetric form of the KL divergence,

$$\begin{aligned} D(p \,||\, q) &:= D_{KL}(p \,||\, q) + D_{KL}(q \,||\, p) \\ D_{KL}(p||q) &:= \sum_{s \in S} p(s) \log \frac{p(s)}{q(s)} \end{aligned}$$

Despite the approximations involved, Definition 4.1.2 is appealing because it can be applied to no-reset, stochastic domains, and is thus more broadly applicable. In the experiments below, computing the online contingent regions first involves learning the pixel models $\tilde{T}_{x,y}(\cdot|ha)$ from data, and subsequently computing $C(h)$ from the pixel models.

4.2 Learning a Contingent Regions Model

The two definitions of contingent regions above formalize the notion of contingency awareness. In this section, I describe how to learn a model that can predict $C(h)$. The purpose of such a model is twofold: first, to enable agents to access $C(h)$ when a simulator is unavailable and second, to cache the otherwise slow process of extracting $C(h)$ at run-time. Learning a contingent region model can thus be viewed as a preprocessing step whose sample cost is amortized by repeated use: the contingent region models used later this chapter and in Chapter 5 were learned once and subsequently reused.

We now cast the problem of learning a contingent region model as a supervised learning task. Let r be a set of tuples $(x, y, t) \in \mathcal{D}_x \times \mathcal{D}_y \times \{0, 1\}$ representing a labelling of the pixels within an observation. Let R denote the set of all such labellings, and let $r_{x,y} := t \in \{0, 1\}$ such that $(x, y, t) \in r$. For a given set of reference histories $H := \{h_1, h_2, \dots\}$, we construct the following set of training examples:

$$\big\{\big(h, r\big) \in H \times R : r_{x,y} = \mathbb{I}\big[(x, y) \in C(h)\big]\big\}. \tag{4.2}$$

Predicting $C(h)$ atomically, as per Equation 4.2, is generally neither feasible nor desirable: in Atari 2600 games, a single prediction consists of 210×160 pixels. The remainder of this section is thus concerned with developing approximations that help learn efficient and accurate contingent region models. At the core of these approximations is the factored nature of the Atari 2600 observation space, which lets us make individual pixel predictions using local screen regularities.

4.2.1 Block Decomposition

In many cases, predicting the contingent regions at the pixel level is unnecessary. To reduce the method's computational cost, we divide the observation space into $k \times k$ blocks, $k \in \mathbb{N}$, and assume that pixels within a block share the same contingency behavior. This leads to the following definition of the contingent regions using a block size of k:

$$B_k(h) := \Big\{(x, y) \in \mathcal{D} : S_k(x, y) \cap C(h) \neq \emptyset\Big\},$$

where $S_k(x, y) := \big\{(x', y') \in \mathcal{D} : g_k(x, y) = g_k(x', y')\big\}$, $g_k(x, y) := (f_k(x), f_k(y))$ and $f_k(x) := k\lceil x/k \rceil - \lfloor k/2 \rfloor$. The approximate block contingent regions $\tilde{B}_k(h)$ is defined similarly, with $\tilde{C}(h)$ replacing $C(h)$ in the above definition.

The block decomposition leads to a speedup proportional to k^2 since only one prediction is made for every $k \times k$ block of pixels. Figure 4.3 shows an example of the exact contingent regions of a SEAQUEST frame and its corresponding 5×5 block decomposition.

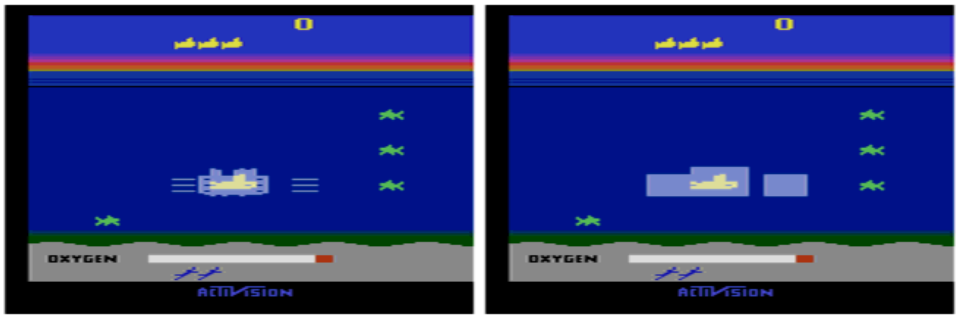

Figure 4.3: Exact (left) and 5x5 block-decomposed (right) contingent regions in SEAQUEST.

4.2.2 Local Features

Whether a particular pixel is part of a screen's contingent regions can usually be inferred from its colour and the colour of neighbouring pixels. A natural idea is to use features of the history h, rather than h itself, to predict whether a location (x, y) is part of $C(h)$. These features can then be used to learn a contingent region model, as will be discussed in Section 4.2.3 below.

Let $\phi : H \times \mathcal{D} \to \mathbb{R}^m$ be a mapping from histories to feature vectors. Given an image $o \in \mathcal{O}$ and a location $l = (x, y)$ within this image, recall the definition of the lexicographic index of an image patch of size $a \times b$ (Section 3.3):

$$\imath_{x,y}^{a,b}(o) := \sum_{y'=y}^{y+b-1} \sum_{x'=x}^{x+a-1} |\mathcal{C}|^{a(y'-y)+(x'-x)} \text{bit}(o_{x,y})$$

There are $|C|^{ab}$ such possible indices; here each pixel is encoded using $|C|$ binary features[2], so that each index $\imath_{x,y}^{a,b}(o)$ corresponds to a binary vector of length $|C|ab$. Let $x' = x - \lfloor \frac{a}{2} \rfloor$ and $y' = y - \lfloor \frac{b}{2} \rfloor$; the concatenation of the binary vectors corresponding to $\imath_{x',y'}^{a,b}(o_t)$ and $\imath_{x',y'}^{a,b}(o_{t-1})$ then forms our mapping ϕ. This mapping represents a binary encoding of the pixels within a $a \times b$ window centered on a given pixel, for both the current and last images. Here the window size $a = b = 21$ is large enough to capture local motion, and thus contingency that depends on nearby pixels. Figure 4.4 depicts the process of converting the neighbourhood of location l, derived from history h, into a binary feature vector $\phi(h, l)$.

4.2.3 Logistic Regression

When predicting each pixel individually and using the block decomposition, the set of training examples becomes

$$\mathcal{T} := \big\{ \big(h, l, t\big) \in H \times \mathcal{D}/k \times \{0, 1\} : t = \mathbb{I}_{[l \in C(h)]} \big\}, \tag{4.3}$$

where $\mathcal{D}/k$ stands for the set of representatives in the k block contingent regions, i.e. locations $(x, y) \in \mathcal{D}$ for which $f_k(x) = x$ and $f_k(y) = y$.

The training set above can easily be constructed from a given set of histories H. The next step is to learn a model of the contingent regions; this is done by fitting a parametrized logistic regression model to the training set and finding an

[2]While each pixel can be uniquely encoded using 7 binary features, the form of encoding used here is better simplifies model learning using logistic regression.

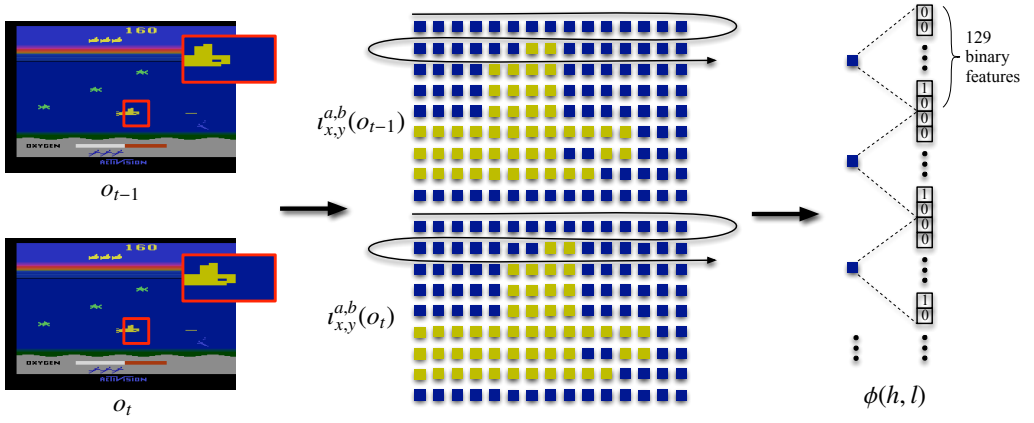

Figure 4.4: Process for generating the binary feature vectors representing pixel neighbourhoods. Each pixel is ultimately encoded using one of 129 binary features, including one feature indicating the out-of-bounds colour.

approximate maximum likelihood solution. Under this logistic model, the likelihood that a pixel at location $l \in \mathcal{D}$ is part of $B_k(h)$ is $p(l \,|\, h; \mathbf{w}) := \sigma\left(\mathbf{w}^T \phi(h,l)\right)$, where $\mathbf{w} \in \mathbb{R}^m$ is a vector of weights, $\sigma(x) := (1 + \exp(-x))^{-1}$ is the logistic function and $\phi(h,l)$ is the feature vector described above. The logistic function was chosen here because it outputs a probability-like quantity and for its ease of use in the presence of many, redundant features. Given a set of training points $\mathcal{T}$ of the form specified by Equation 4.3, the likelihood of the observed data under the logistic model is

$$p(\mathcal{T}; \mathbf{w}) := \prod_{(h,l,t) \in \mathcal{T}} p(l \,|\, h; \mathbf{w})^t (1 - p(l \,|\, h; \mathbf{w}))^{1-t}. \tag{4.4}$$

Equation 4.4 can be maximized by minimizing the *cross entropy error function*[3] $E(\mathbf{w}) := -\log(p(\mathcal{T}; \mathbf{w}))$, which upon rearranging and simplifying becomes

$$E(\mathbf{w}) := -\sum_{(h,l,t) \in \mathcal{T}} \Big[t \log p(l \,|\, h; \mathbf{w}) + (1 - t) \log\left(1 - p(l \,|\, h; \mathbf{w})\right) \Big].$$

Stochastic gradient is used to minimize $E(\mathbf{w})$ in order to avoid keeping the entire training set in memory. For the k^{th} training example $(h_k, l_k, t_k) \in \mathcal{T}$, this yields the update equation

$$\mathbf{w}_k \leftarrow \mathbf{w}_{k-1} + \eta \left[t_k - \sigma\left(\mathbf{w}_{k-1}^T \mathbf{\Phi}_{l_k}(h_k)\right) \right] \phi(h_k, l_k) \ ,$$

where $\eta \in \mathbb{R}$ is a step-size parameter.

[3]The cross entropy error function is the matching loss for the logistic function(Auer et al., 1995).

4.2.4 The Pixel Colour Change Model

Although Equation 4.1 gives a natural criterion for detecting contingent regions with a probabilistic pixel colour model, learning such a model over the full 7-bit Atari colour space requires a prohibitive number of samples. Instead, an acceptable level of accuracy can be achieved by predicting whether a pixel changes colour rather than predicting the colour to which it changes. In effect, this reduces learning a full pixel colour model (the $\tilde{T}^a_{x,y}$ term) to learning a binary predictor. This also reduces the number of summands needed to compute the symmetrized KL divergence by a factor of 2^6. The pixel colour model is otherwise learned in the same fashion as the contingent region model, using logistic regression and stochastic gradient descent.

4.2.5 Model Learning Results

I studied how well contingent region models could learn to predict the contingent regions across all games, in both oracle (Definition 4.1.1) and online (Definition 4.1.2) settings. In the oracle setting I constructed the training set using the emulator's load/save state functions. In the online setting I first trained a pixel colour change model, then generated the training set using a KL divergence test.

In both settings, a time step corresponds to five frames. Training data was generated by sampling game situations that occurred near a human-provided trajectory. I played each game for at least two minutes and recorded the resulting trajectory. These trajectories do not constitute expert trajectories, but rather help provide representative coverage of the histories. Hybrid trajectories were then constructed by following the human-provided trajectory for a uniformly random number of steps and then taking 300 additional, uniformly random actions. Pixel colour change models were learned from 2,000,000 samples. Predictors for both settings were then trained on 200,000 samples. The divergence threshold parameter in Equation 4.1 was set to $\delta = 4.0$.

Table 4.1 reports relevant statistics for the learned models, while Figure 4.5 compares the ground truth block decomposition to sample predictions made by both models. A model's accuracy is the average proportion of pixels that it correctly labels. Precision is the average fraction of pixels labelled by the model as part of $C(h)$ which actually are under the agent's control. Recall is the average fraction of pixels in the contingent regions which are labelled as such by the model. Of note, the oracle learning procedure yielded a highly accurate model; the low

	Accuracy		Precision		Recall	
	Offline	Online	Offline	Online	Offline	Online
Mean	0.969	0.919	0.396	0.217	0.573	0.356
Min.	0.800	0.365	0.016	0.000	0.000	0.000
Max.	1.000	0.998	0.769	0.836	0.985	0.985

Table 4.1: Prediction statistics for the learned contingent region models. Each model was learned once; minimum and maximum reflect prediction accuracy across the 51 games.

precision and recall values can be explained by the use of the block decomposition (Section 4.2.1). In practice, the contingent regions predictor correctly identified the contingent regions surrounding important, controllable objects. Block decomposition yields results that are sufficiently accurate for many purposes, including avatar tracking (described below in Section 4.3), even though it cannot precisely capture the boundaries of the contingent regions. The accuracy of the online learning procedure varied widely across games. While the pixel change models were generally accurate, no single value of δ provided a good approximation of the contingent regions (Equation 4.1) for all games: Space Invaders worked best with $\delta = 0.1$, while Beam Rider required $\delta = 10.0$. The online results of Table 4.1 reflect the choice of $\delta = 4.0$ as a compromise, which resulted in false positives in Beam Rider and Seaquest and predicted contingent regions in Space Invaders.

4.3 Tracking the Player Avatar

The contingent region model described in the previous section does not directly lead to improved agents. It does, however, provide us with key information about the domain. In most Atari 2600 games, the player controls an avatar: a chicken, a plane, or a superhero. This section presents an algorithm that uses a learned contingent region model to track the location of this avatar. Here I assume that such an avatar

- exists,
- moves smoothly, and
- generates large, contiguous contingent regions in its neighbourhood.

The algorithm is a standard Bayes filter (Russell and Norvig, 2003) with the following components:

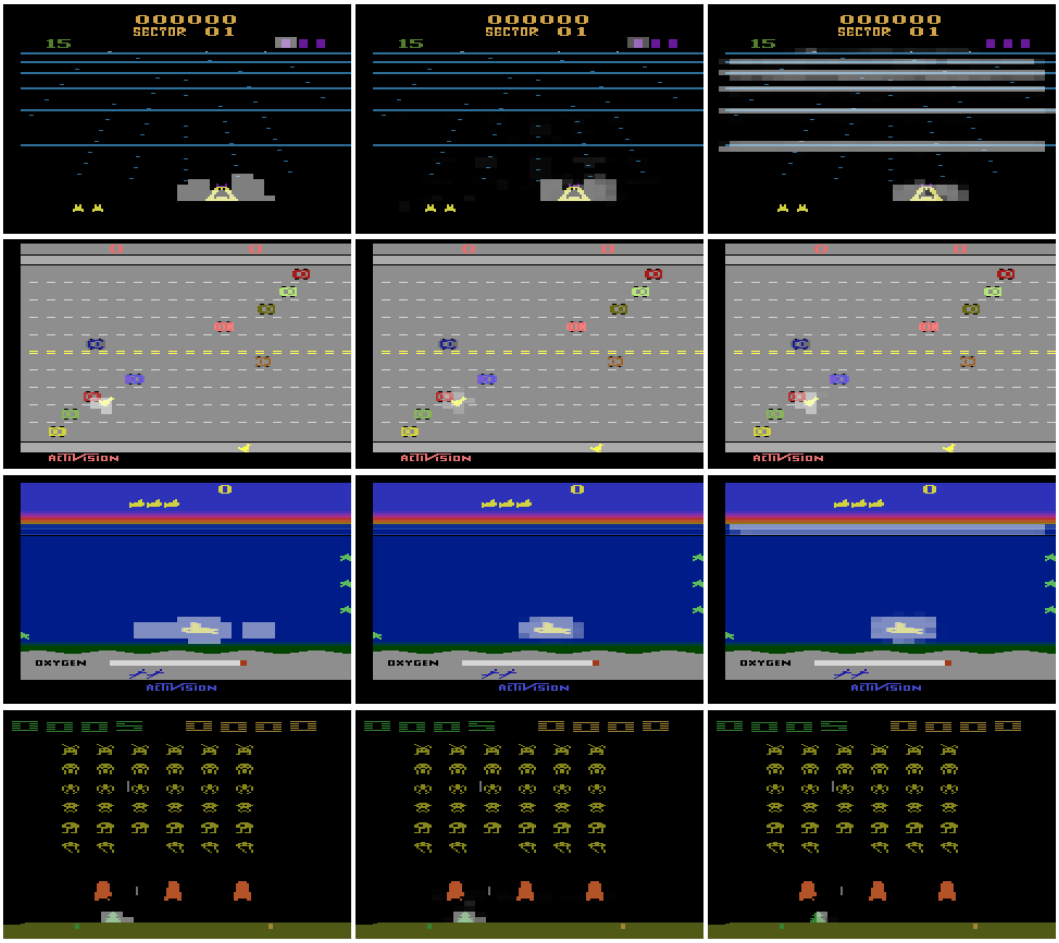

Figure 4.5: From left to right: contingent regions ground truth $B_5(h)$; predictions from the model learned in the oracle setting; predictions from the model learned in the online setting.

- a truncated Gaussian motion model,
- a Gaussian observation model, and
- a prior over the player location.

In what follows, (x_t, y_t) denotes the (unobserved) player avatar location, with $\mu_t(x, y) := \Pr(x_t = x, y_t = y)$ and a uniform prior $\mu_0(x, y) = (d_w d_h)^{-1}$, where $d_w = 160$ and $d_h = 210$ denote the screen width and height, respectively. I first describe tracking for when the true contingent regions $C(h_t)$ are observed, and then describe how to adapt it to use predictions from the contingent region model of the previous section.

4.3.1 Motion Model

Let $r_m \in \mathbb{N}$ be a radius parameter. The truncated motion model $M : \mathbb{Z} \times \mathbb{Z} \to [0, 1]$ is defined as

$$M(\Delta_x, \Delta_y) := \begin{cases} \frac{1}{Z} \exp^{-(\Delta_x^2 + \Delta_y^2)/2} & \text{if } -r_m \leq \Delta_x, \Delta_y \leq r_m \\ 0 & \text{otherwise} \end{cases}$$

where Z is an appropriate normalizing constant so that $\sum_{\Delta_x} \sum_{\Delta_y} M(\Delta_x, \Delta_y) = 1$. The motion posterior μ'_t is then defined as

$$\mu'_t(x, y) := \begin{cases} \sum_{x'=0}^{d_w-1} \sum_{y'=0}^{d_h-1} M(x - x', y - y') \mu_{t-1}(x', y') & \text{if } 0 \leq x < d_w, 0 \leq y < d_h \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Note that in the formulation above, the motion model is independent of the action taken. Although a more natural model might involve shifting the probability mass left when the agent presses left, and similarly with the other directions, this approach fails in games where the range of motion is restricted, such as Space Invaders. The need for a more agnostic motion model is an example of the difficulty in designing domain-independent Atari 2600 agents.

4.3.2 Observation Model

Within our Bayes filter an observation corresponds to a labelling of each pixel as belonging or not to the contingent regions, similar to the labelling r used to describe the training set of Equation 4.2. Let $c_t(x, y) \in \{0, 1\}$ denote the noisy observation

that $(x, y) \in C(h_t)$, and let $c_t \in \{0, 1\}^{d_w \times d_h}$ be the noisy observation of the whole contingent regions. Separating the true underlying contingent regions $C(h_t)$ from its observation c_t is necessary to avoid computing a degenerate posterior, and will later allow us to interpret the output of the contingent region model as a probability. We make the assumption that each location is independently observed, and that $c_t(x, y) = 1$ is more likely in pixels close to the avatar location:

$$\begin{aligned}
\Pr(c_t|x_t, y_t, C(h_t)) &:= \prod_{0 \le x < d_w} \prod_{0 \le y < d_h} \Pr(c_t(x, y)|x_t, y_t, C(h_t)) \\
\Pr(c_t(x, y) = 1|x_t, y_t, C(h_t)) &:= O(x - x_t, y - y_t) \\
\Pr(c_t(x, y) = 0|x_t, y_t, C(h_t)) &:= 1 - O(x - x_t, y - y_t)
\end{aligned}$$

with

$$O(\Delta_x, \Delta_y) := \begin{cases} \frac{1}{Z} \exp^{-(\Delta_x^2 + \Delta_y^2)/c} & \text{if } -r_o \le \Delta_x, \Delta_y \le r_o \\ 0 & \text{otherwise} \end{cases}$$

where $r_o \in \mathbb{N}$ is a radius parameter, Z is a normalizing constant and $c \in \mathbb{R}$ is a width parameter (empirically $c = 5$ was found to give good tracking performance). While c effectively determines r_o, the latter is made explicit to emphasize the need for a fast tracking algorithm; in the experiments below $r_o = 7$.

4.3.3 Belief Update

We combine the observation model and the motion posterior to obtain $\mu_t(x, y)$, the posterior distribution over the avatar location at time t given $C(h_t), C(h_{t-1}), \ldots$:

$$\mu_t(x, y) := \frac{1}{Z} \mu'_t(x, y) \prod_{x', y'} \left[O(x - x', y - y')^{\mathbb{I}_{[(x', y') \in C(h_t)]}} (1 - O(x - x', y - y'))^{\mathbb{I}_{[(x', y') \notin C(h_t)]}} \right], \tag{4.6}$$

where Z is again a normalizing constant so that $\sum_{0 \le x < d_w} \sum_{0 \le y < d_h} \mu_t(x, y) = 1$. The full belief update proceeds as follows:

1. Observe $C(h_t)$,
2. Compute $\mu'_t(x, y)$ as per Equation 4.5, then
3. Compute $\mu_t(x, y)$ as per Equation 4.6.

4.3.4 Belief Update with the Contingent Regions Model

In practice, the contingent region model produces the probability $\Pr((x,y) \in C(h_t)|h_t)$, rather than an actual observation $C(h_t)$. Although sampling $C(h_t)$ from this probability distribution would allow us to update $\mu_t(x,y)$ using Equation 4.6, the following shows that it is also possible to directly use $\sigma_t(x,y)$. Let $\sigma_t(x,y) \in (0,1)$ denote the output[4] of the contingent region model σ for location (x,y), interpreted as a probability:

$$\Pr((x,y) \in C(h_t)|x_t, y_t, \sigma) := \sigma_t(x,y)$$

We now recover the observation $c_t \in \{0,1\}^{d_w \times d_h}$ from the assumed independence of the $c_t(x,y)$ terms:

$$\begin{aligned}
\Pr(c_t|x_t,y_t,\sigma) &= \sum_{C(h_t)'} \Pr\left(c_t|x_t,y_t,C(h_t)'\right) \Pr\left(C(h_t) = C(h_t)'|x_t,y_t,\sigma\right) \\
&= \prod_{0\le x<d_w} \prod_{0\le y<d_h} \Big[\Pr\left(c_t(x,y)|x_t,y_t,(x,y) \in C(h_t)\right) \Pr\left((x,y) \in C(h_t)|x_t,y_t,\sigma\right) + \\
&\qquad \Pr\left(c_t(x,y)|x_t,y_t,(x,y) \notin C(h_t)\right) \Pr(\left((x,y) \notin C(h_t)|x_t,y_t,\sigma\right)\Big] \\
&= \prod_{0\le x<d_w} \prod_{0\le y<d_h} \Big[\Pr\left(c_t(x,y)|x_t,y_t,(x,y) \in C(h_t)\right)\sigma_t(x,y) + \\
&\qquad \Pr\left(c_t(x,y)|x_t,y_t,(x,y) \notin C(h_t)\right)\left(1-\sigma_t(x,y)\right)\Big],
\end{aligned}$$

where $C(h_t)$ denotes the true, unobserved contingent regions and the second line follows from the binomial expansion (there are $2^{d_w \times d_h}$ possible contingent regions $C(h_t)'$). We obtain the belief update equation

$$\mu_t(x,y) := \frac{1}{Z}\mu'_t(x,y) \prod_{x',y'} \left[O(x-x',y-y')\sigma_t(x',y') + (1-O(x-x',y-y'))(1-\sigma_t(x',y'))\right].$$

where Z is such that $\sum_{x,y} \mu_t(x,y) = 1$.

4.3.5 Avatar Tracking Results

Quantitatively evaluating avatar tracking across a large set of games poses a challenge: obtaining a ground truth signal involves either a difficult, error-prone study of

[4]Recall that the sigmoid function $\sigma(x) := (1+\exp(-x))^1$ takes values in $(0,1)$.

the RAM contents for each game, or manually labelling tens of thousands of frames. Here, I instead provide some qualitative tracking results.

Figure 4.6 depicts nine frames for both SEAQUEST and BEAM RIDER. Each frame shows the predicted contingent regions, as well as the estimated avatar location. In both cases, the Bayes filter closely tracks the avatar. In BEAM RIDER, there are in fact two contingent regions: the space ship and the missile (top-right corner). Here, the Bayes filter naturally selects the largest contiguous contingent region.

Even when the contingent region model predicts well, avatar tracking can be nontrivial. The following three examples illustrate the difficulties in tracking based on contingent regions. In ASTERIX (Figure 4.7, top) the player's lives are *correctly* labelled as part of the contingent regions. Here the issue is that controllable portions of the screen generate large, contiguous contingent regions unrelated to the avatar. In KUNG-FU MASTER (Figure 4.7, middle), the situation is different. KUNG-FU MASTER belongs to a genre called *scrollers*, where the whole background shifts with the avatar's motions. As depicted, the tracking mechanism fails when the left and right actions are applied. Another, subtler issue arises in ALIEN (Figure 4.7, bottom): the player's location on the screen determines the colour of other objects (pebbles and monsters). These and similar issues indicate that avatar tracking via the notion of contingency awareness is necessarily imperfect. However, anecdotal evidence suggests the avatar tracking mechanism degrades gracefully rather than catastrophically. In ALIEN, for example, the avatar is successfully tracked a fraction of the time. As the next section will show, the learning agent can sometimes compensate, for example by learning to stay still in KUNG-FU MASTER.

4.4 Feature Generation Methods

I now describe a way to provide the tracked avatar location to a learning agent in order to improve its performance. Here I consider a reinforcement learning agent using linear value function approximation and SARSA(λ) (Section 2.1.2). The avatar location is provided to the agent by extending the set of features it uses to learn value functions in Atari 2600 games.

4.4.1 Basic

In his Master's thesis, Naddaf (2010) proposed a simple way to generate features for Atari 2600 games, called *BASS*. BASS encodes the presence of colour at various

Figure 4.6: Avatar tracking in SEAQUEST and BEAM RIDER. The predicted contingent regions is overlaid in white and the estimated location is indicated by the red crosshairs.

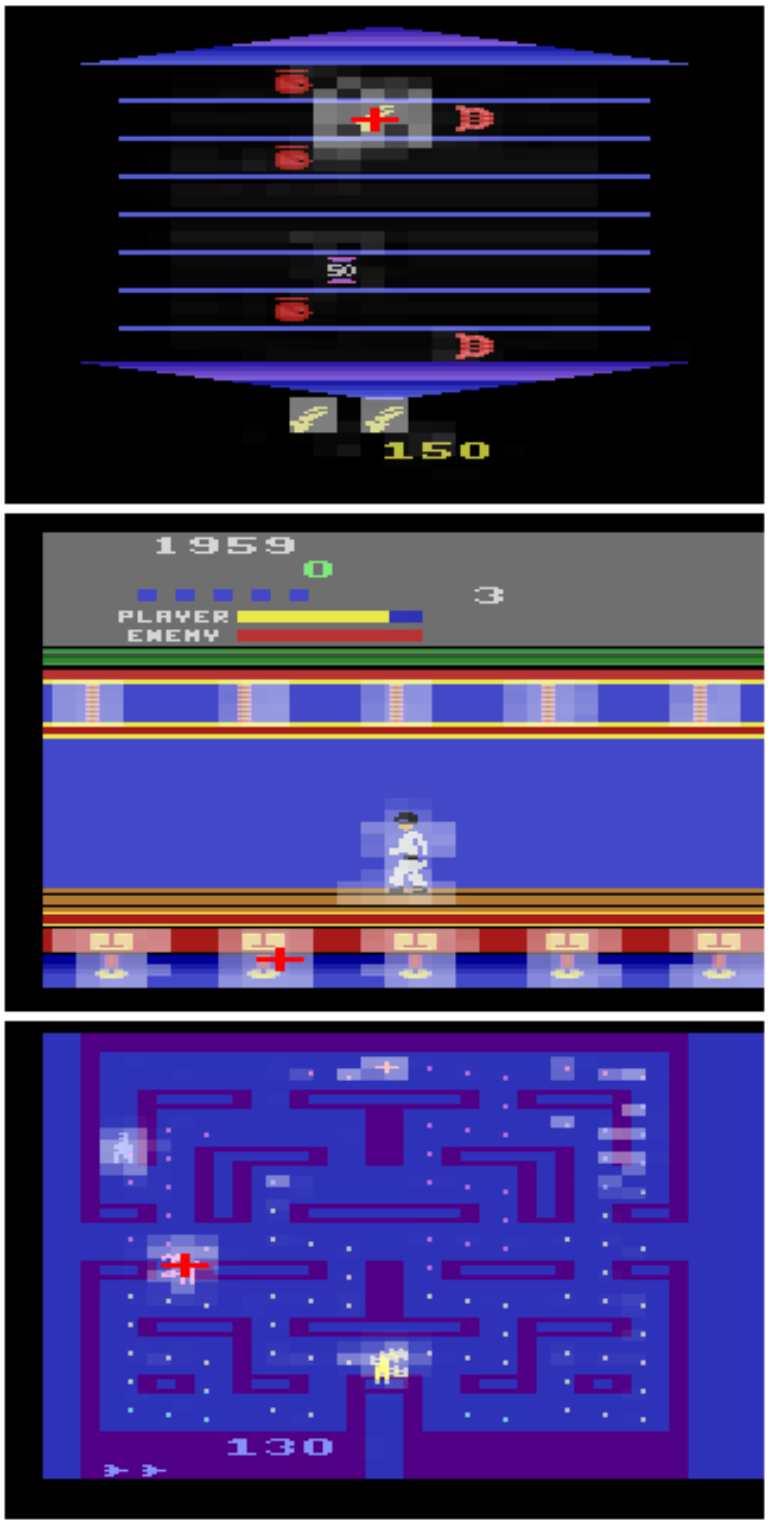

Figure 4.7: Three problematic games for avatar tracking. From top to bottom: Asterix, Kung-Fu Master and Alien.

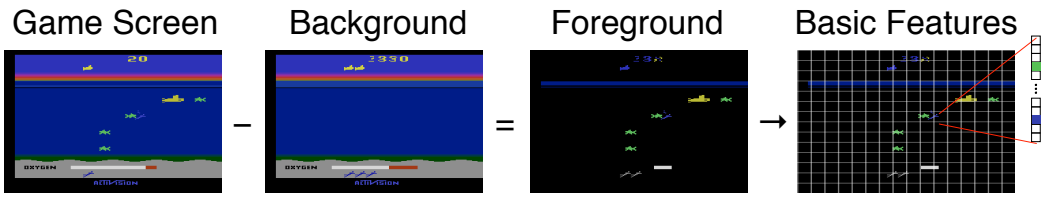

Figure 4.8: The Basic feature generation method. The background is first subtracted; the resulting foreground is then divided into tiles and encoded as a binary vector.

locations on the screen as a set of binary features, and also encodes all pairwise combinations of these features. The *Basic* method, first introduced in the companion paper to this chapter (Bellemare et al., 2012a), is a simplified version of BASS where pairwise combinations are omitted.

Figure 4.8 summarizes the actual feature generation process. As a preprocessing phase to the Basic method, the game's background is extracted by taking, from a small set of screen samples, the most frequently occurring pixel at each location. The Basic method first subtracts the background from the current game screen. The resulting foreground image is then divided into $a \times b$ tiles, $a, b \in \mathbb{N}$. Finally, the presence of each of the 128 colours in each tile is encoded as one large binary vector. For a given observation o, $\bar{o}$ denotes the foreground image

$$\bar{o} := \{(x, y, c) : (x, y, c) \in o, c \neq \text{bg}_{x,y}\},$$

where $\text{bg} \in \mathcal{O}$ denotes the extracted background; the foreground $\bar{o}$ is thus a subset of the full observation. The term $\text{tile}[(i, j, c)]$ denotes that tile (i, j) *contains* colour c, formally

$$\text{tile}[(i, j, c)] := \begin{cases} 1 & \text{if } \bar{o} \cap \{(x, y, c) : (x, y) \in l^{a,b}_{ai,bj}(o)\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

where $l^{a,b}_{ai,bj}(o)$ is the set of locations corresponding to image patch $p^{a,b}_{ai,bj}(o)$ (Section 3.3). Recall that $\mathcal{C}$ is the set of possible colours, d_w the screen width and d_h the screen height; assuming that a is a multiple of d_w and b, a multiple of d_h, there are $\frac{d_w}{a}\frac{d_h}{b}|\mathcal{C}|$ possible (i, j, c) tuples. Let $\text{f}(i, j, c) := \frac{d_w}{a}|\mathcal{C}|j + |\mathcal{C}|i + c$ be the unique index describing the tuple (i, j, c) and $\text{f}^{-1}(k)$ its reverse mapping. The k^{th} component of the feature vector generated by the Basic method, ϕ^{Basic}, is defined as

$$\phi_k^{Basic} := \text{tile}[\text{f}^{-1}(k)]$$

In other words, the Basic feature generation method provides a binary encoding representing the presence of colours in the tiled foreground.

4.4.2 Extended

In many Atari 2600 games, the most relevant object relationship concerns the avatar and other objects, for example when obstacles must be avoided. The *Extended* feature generation method focuses on this relationship by jointly encoding the location of the player avatar (as determined by the tracking mechanism of the previous section) and ϕ^{Basic}. Figure 4.9 depicts the two discretizations involved: the screen is divided into $a \times b$ tiles by the Basic feature generation, and also divided into a different set of $c \times d$ tiles when encoding the avatar location. Let $L := |\phi^{Basic}|$ be the length of the Basic vector, and $T := \frac{d_w}{c} \times \frac{d_w}{d}$ be the number of possible locations output by the tracking mechanism. The vector generated by the Extended method, $\phi^{Extended}$, has length $L \times (T+1)$, and consists in the concatenation of the vector ϕ^{Basic} with T other vectors of the same length. Of these T vectors, exactly one contains nonzero entries at any given time, corresponding to the current avatar location. Formally, this vector is defined component-wise (with $0 \leq k < L(T+1)$) as

$$\phi_k^{Extended} := \begin{cases} \phi_k^{Basic} & \text{if } 0 \leq k < L \\ \phi_{k \bmod L}^{Basic} & \text{if } g(x,y) = \left\lfloor \frac{k-L}{L} \right\rfloor \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

where (x, y) is the estimated avatar location and $g(x, y) := cx' + y'$ denotes the indexing from avatar locations to a discretized tile index, with $x' := \left\lfloor \frac{x}{c} \right\rfloor$, $y' := \left\lfloor \frac{y}{d} \right\rfloor$. From Equation 4.7 it is clear that, when there are l non-zero features in ϕ^{Basic}, only $2l$ non-zero features are present in $\phi^{Extended}$, a significant economy in comparison to the $O(l^2)$ features generated by BASS.

4.4.3 MaxCol

The *MaxCol* feature generation method is a simplification of the Basic method which does not depend on a background extraction step. Rather than performing background subtraction to reduce the number of colours present in a tile, MaxCol

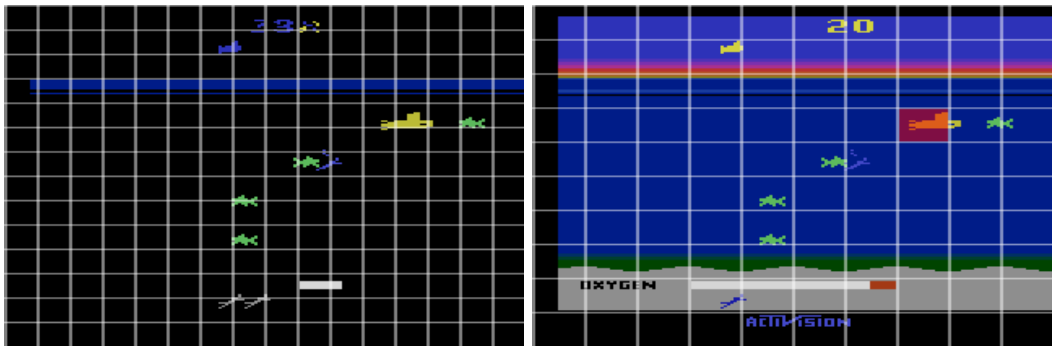

Figure 4.9: The Extended feature generation method jointly encodes the discretized foreground (left) and discretized avatar location (right).

only encodes the tile's two most frequent colours. For a given $a \times b$ image patch $p^{a,b}_{x,y}(o)$, its two most frequent colours c_1 and c_2 are defined as

$$\begin{aligned} c_1 &:= \arg\max_{c \in \mathcal{C}} |\{(x,y,c) : (x,y,c) \in p^{a,b}_{x,y}(o)\}| \\ c_2 &:= \arg\max_{c \in \mathcal{C} \setminus \{c_1\}} |\{(x,y,c) : (x,y,c) \in p^{a,b}_{x,y}(o)\}| \end{aligned}$$

Denote by $\mathbf{c}^{a,b}_{x,y}(o)$ the set of the two most frequent colours of patch $p^{a,b}_{x,y}(o)$. MaxCol generates a feature vector ϕ^{MaxCol}, formally defined as

$$\begin{aligned} \text{freq}[(i,j,c)] &:= \begin{cases} 1 & \text{if } c \in \mathbf{c}^{a,b}_{ai,bj}(o) \\ 0 & \text{otherwise} \end{cases} \\ \phi^{MaxCol}_k &:= \text{freq}[f^{-1}(k)] \end{aligned}$$

4.4.4 Extended MaxCol

The Extended MaxCol method jointly encodes MaxCol features and avatar location using the same technique as the Extended method of Section 4.4.2:

$$\phi^{Ext.MaxCol}_k := \begin{cases} \phi^{MaxCol}_k & \text{if } 0 \leq k < L \\ \phi^{MaxCol}_{k \bmod L} & \text{if } g(x,y) = \lfloor \frac{k-L}{L} \rfloor \\ 0 & \text{otherwise} \end{cases}$$

4.5 Empirical Study

The training games (Section 3.1.2) were used to design, test and parameterize the feature generation methods above. All methods were subsequently evaluated on an

	Background Subtraction	Avatar Location	Grid Size	Number of Features
Basic	✓		16×14	28,672
Extended	✓	✓	16×14	2,867,200
MaxCol			32×30	122,880
Extended MaxCol		✓	32×30	12,288,000

Table 4.2: Overview of the feature generation methods described in this chapter.

additional 46 games, listed in Appendix B. The aim of the empirical evaluation was twofold:

1. To investigate the benefits of providing the avatar location to learning agents, and

2. To compare the performance of agents using either the oracle or online models.

Table 4.2 summarizes the differences between the feature sets. Each game's background was precomputed offline, using a sample trajectory containing 18,000 observations from a uniformly random policy. Across feature generation methods, the avatar location was divided into a 10×10 grid.

4.5.1 Reinforcement Learning Setup

Agents were trained using the SARSA(λ) reinforcement learning algorithm. The value function was approximated using linear approximation with one of the four feature sets of Section 4.4. The agent followed the ϵ-greedy policy (Sutton and Barto, 1998), which takes uniformly random exploratory actions with probability ϵ and otherwise chooses the action estimated to have the highest value.

The discount factor was set to $\gamma = 0.999$, the eligibility trace parameter $\lambda = 0.9$ and the exploration rate $\epsilon = 0.05$. These values were chosen through brief empirical experimentation. The learning rate α_t was set to $\alpha_t := \frac{\alpha}{\max_t \|\phi(s_t,a_t)\|_0}$, where α is a user-specified constant and $\|\phi(s_t, a_t)\|_0$ is the number of non-zero features at time t; this method is a heuristic for selecting α when the size of the feature vector is not known ahead of time. The constant α was tuned for each specific feature generation method by using parameter sweeps over the training games. The best values were 0.2, 0.1, 0.5, 0.5 for Basic, MaxCol, Extended and Extended MaxCol respectively. Note that these values are not directly comparable since $\|\phi(s_t, a_t)\|_0$ depends on the feature generation method and the particular game being played.

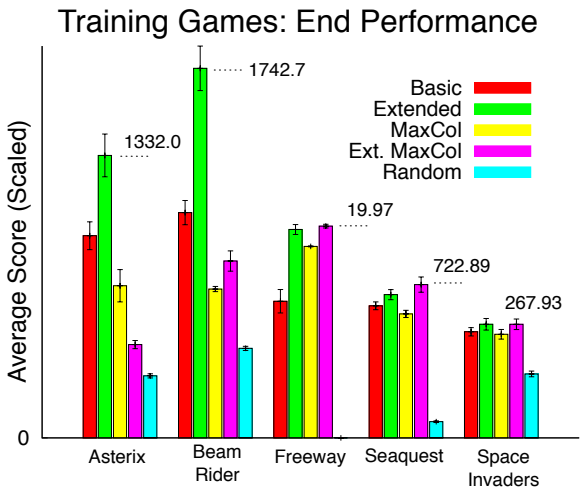

Figure 4.10: Average score over the last 500 episodes for each of the training games. Each result is an average over 30 trials; error bars show 99% confidence intervals.

4.5.2 Training Evaluation

I first evaluated the different methods on the training games. Each feature set was tested using 30 independent trials per game. Each trial consisted in training the agent on a particular game for 10,000 episodes, each lasting up to 18,000 frames. A time step from the agent's perspective corresponds to five frames, during which the selected action is repeated. The performance within a trial was obtained by taking the average score obtained during the last 500 episodes of training. The reported overall performance estimate is the average performance across all 30 trials.

The results obtained on the training games are shown in Figure 4.10. Extended performs statistically better than Basic on 4/5 games, while Extended MaxCol performs better than MaxCol on 3/5 games and worse on Asterix. These results illustrate the advantage of using contingency awareness. It was also observed (not shown here) that the performance of Basic agents reached a plateau within 10,000 episodes. In contrast, for Asterix, Beam Rider and Space Invaders, agents using Extended were still learning at the end of the trial. In Asterix the lower performance of Extended MaxCol is likely a consequence of the agent controlling the appearance of the remaining lives icons, as discussed in Section 4.3.5; despite this imperfect tracking, the benefits of encoding the avatar location is sufficient to allow Extended to outperform Basic in Asterix. All learning agents perform better than the random policy.

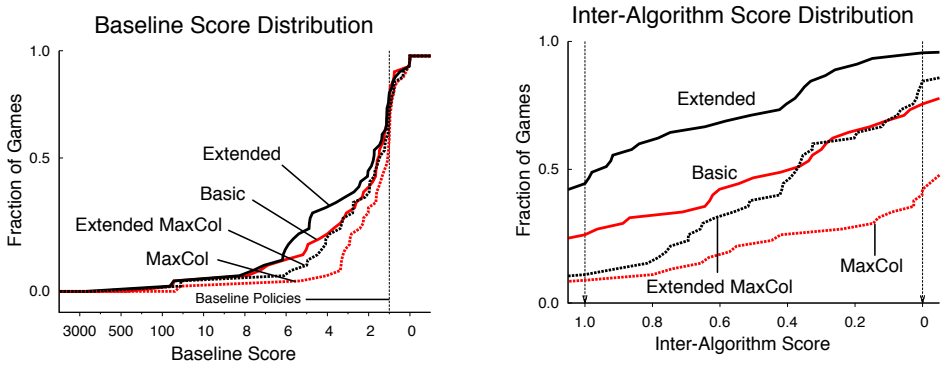

Figure 4.11: **Left.** Baseline score distribution. **Right.** Inter-algorithm score distribution. Each line indicates the fraction of testing games that achieve a score at least as high as the value on the x axis.

4.5.3 Testing Evaluation

The feature generation methods were next evaluated across forty-six games from the test set, using the same experimental setup as for the training games. I report the results using score distributions (described in Section 3.2); these distributions *exclude* the training games.

Figure 4.11 (left) depicts the baseline score distribution for all four feature sets. The graph demonstrates that learning took place: all methods achieve a score of at least 1.0, representing the score of the best baseline policy, in a majority of games. Furthermore, for games where learning occurred, incorporating contingency information led to improvements in performance.

The differences observed in the baseline score distribution become clearer when considering the inter-algorithm score distribution (Figure 4.11, right). Here, Extended performs better than Basic, and Extended MaxCol performs better than MaxCol. In particular, Extended achieves the highest score on nearly half of the games.

I also computed the proportion of testing games for which the extended methods provide a statistically significant (non-overlapping 99% confidence intervals) advantage. Extended did better than Basic in 17 games and worse in 7, while Extended MaxCol did better than MaxCol in 22 games and worse in 6. These results highlight the benefits of using contingency information for feature construction.

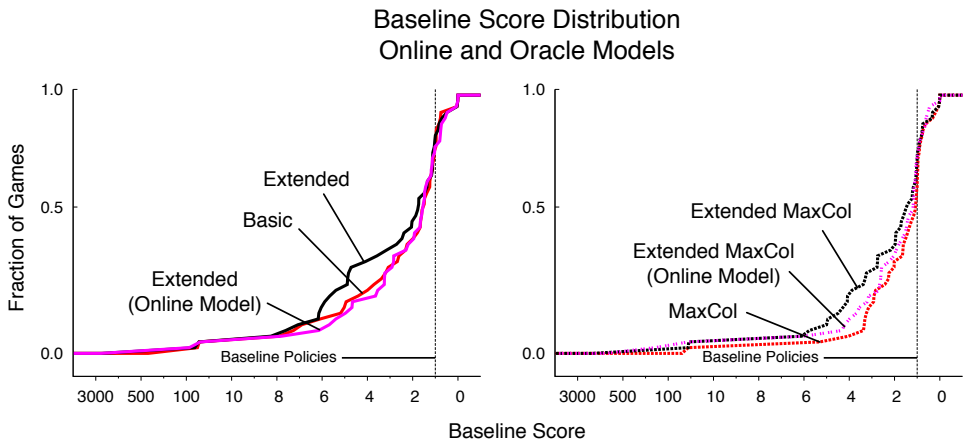

Figure 4.12: Baseline score distributions comparing the online and oracle methods on the testing games. **Left.** Basic and Extended feature sets. **Right.** MaxCol and Extended MaxCol feature sets.

4.5.4 Online Contingency Learning

I next compared agents using the online contingent region model to agents using the oracle contingent region model. Figure 4.12 shows the baseline score distributions for all algorithms. Across the 46 testing games, the online Extended MaxCol method showed significant improvements over MaxCol, while the online Extended method did no better than Basic. Unsurprisingly, the methods using the oracle contingent region model performed better than their online counterparts. As discussed in Section 4.2.5, the single choice of divergence threshold led in many cases to a loss of accuracy in the contingent regions predictor, which made it difficult to track the player avatar. One possible explanation for the lack of improvement in Figure 4.12 is that the Basic feature set is sufficiently rich to handle many situations without the need for the avatar location, in contrast to the impoverished MaxCol feature set.

All six methods were also compared using inter-algorithm scores, as shown in Figure 4.13. A single score range computed from all six algorithms was used for normalization. Extended MaxCol with the online contingent region model outperformed MaxCol, but did not achieve the performance level of the offline method. The online Extended method rarely achieved the best score for a game, as indicated by the low fraction of games for which its inter-algorithm score is 1.0; across all games it performed roughly the same as the Basic method.

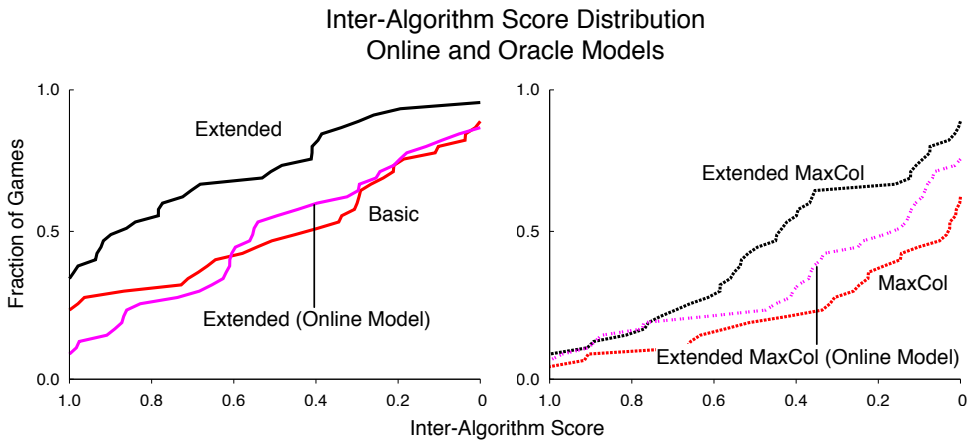

Figure 4.13: Inter-algorithm score distributions comparing the online and oracle methods on the testing games.

4.5.5 Discussion

The empirical results above provide evidence towards the validity of using contingent regions to improve the performance of reinforcement learning agents. On the other hand, the less-than-ideal performance of the online contingent region model suggests the need for a different approach to the no-reset, stochastic setting; such an approach would need to improve both the learning of its colour distribution models and the subsequent comparison of the output colour distributions.

4.6 Conclusion

In this chapter I proposed a formalization of the notion of contingency awareness. In turn, I described algorithms for

- Learning a contingent region model,
- Tracking the avatar location using the output of such a model, and
- Extending a feature generation method for Atari 2600 games using the avatar location.

These algorithms were then put together into a reinforcement learning agent that learned to play arbitrary Atari 2600 games. As with any architecture, this approach is only as strong as its weakest link; thus, the decrease in accuracy when using the online contingent region model leads to a decrease in agent performance.

Furthermore, avatar tracking from the output of a contingent region model is only meaningful if the model output reflects the presence of an avatar; this assumption is violated in side-scroller games and games with a three-dimensional perspective. Despite these limitations, the empirical results of Section 4.5 suggest the benefits of contingency-based feature generation for many Atari 2600 games.

Beyond the Atari 2600, the notion of contingency awareness naturally extends to domains in which the observation space is factored (Degris et al., 2006). It also seems plausible that contingency awareness could be useful for option discovery, for example by restricting option policies to only use contingent regions as state information.

The main failure of this chapter is perhaps the sub-par performance of the online contingent region model. In Chapter 6 I describe the quad-tree factorization (QTF) algorithm, which can learn forward models of Atari 2600 games and could be used to improve the online contingent region model. As QTF implicitly learns both action-conditional and action-independent models (through variable length prediction contexts) it could be extended to answer the question: is this pixel affected by the agent's choice of action? This idea may be investigated in future work.

4.7 Related Work

Although the notion of contingency awareness presented in this chapter is novel, the idea of exploiting a sense of location has been studied from various angles. Closest is the work of Hausknecht et al. (2012), where the *self* is identified in preprocessed Atari 2600 frames using an information gain approach. This approach differs from the contingency awareness-based detection of avatars: it relies on an object detection step whose purpose is to label important pixels according to the object class to which they belong. While object detection simplifies feature generation and planning, it is in general a complex problem to which we still lack an error-free solution. Leffler et al. (2007) studied the relocatable action model formalism, in which the effect of actions are grouped into types. Effectively, this allows faster learning of environment models by assuming invariance, such as translation invariance, in the transition function. Relocatable action models can be used to encode a form of agent egocentrism similar to contingency awareness, but learning such a model requires knowing additional domain knowledge. An interesting, but computationally expensive approach by Bowling et al. (2006) to the problem of map learning also

explicitly considers action invariance across states.

While the definition of contingency awareness (Definition 4.1.1) is not tied to feature generation, it is chiefly for this purpose that it is used in this chapter. The term *lifelong learning* (Thrun and Mitchell, 1993) describes the idea of an agent accreting knowledge of its world, often in the form of a set of features or representation, through a long period of undirected interactions with its environment. Greedy feature construction approaches were investigated within the context of connectionist function approximation and partially observable grid worlds (Ring, 1997) and continuous-sensor robotic domains (Pierce and Kuipers, 1997). More recently, Oudeyer et al. (2007) proposed an approach wherein the agent explores its environment through an objective function based on learning improvement; the agent then partitions its observation space based on differences in sensor predictions. Stober and Kuipers (2008) similarly studied learning an agent's high-level environment representation from raw sensors in the context of a Pong clone. An alternative approach to the question of lifelong learning, laid out in the work of Sutton et al. (2011), studies how an agent can gain knowledge about its environment by simultaneously learning a myriad of prediction problems. This approach seems particularly promising as it explicitly seeks to obtain a general representation of the agent's environment, similar to the objectives of deep learning algorithms (Bengio, 2009). The contingency awareness approach is more restricted in scope than lifelong learning, and so perhaps more easily implemented for new domains.

Another interesting line of research is the design of algorithms that explicitly seek features that improve an agent's return. The U-Tree algorithm (McCallum, 1995) progressively expands a tree-based value function based on statistical differences in expected return; the parti-game algorithm (Moore and Atkeson, 1994) similarly decomposes the continuous state space of deterministic environments into variable resolution partitions using a cost-to-goal objective function. The Bellman error (Williams and Baird, 1993) has been used in a variety of way to generate features: Menache et al. (2005) used it in conjunction with gradient descent-based algorithms to optimize radial basis functions, Keller et al. (2006) constructed basis functions using neighborhood component analysis, and Parr et al. (2007) also constructed basis functions for linear approximation and provided theoretical guarantees for their approach. While contingency awareness does not rely on the agent's return, it is often the case that the components of the observation directly under the agent's

control, in particular the avatar location, strongly determine reward.

"Stability," said the Controller. "stability. No civilization without social stability. No social stability without individual stability."

Brave New World
Aldous Huxley

Chapter 5

Tug-of-War Linear Value Function Approximation

The quality of policies obtained through linear value function approximation (Section 2.1.3) depends on the quality of the features used in the approximation. Designing good features for a particular application is usually time-consuming, and the resulting features can rarely be transferred to new domains. An alternative approach is to exhaustively generate many features through a set of simple rules, and then perform feature selection or dimensionality reduction. Such a set of rules may, for example, enumerate all k-wise combinations of boolean predicates. Many successful applications of linear value function approximation have implicitly relied on this approach. Tile coding, a canonical example of exhaustive feature generation for continuous-state reinforcement learning, jointly encodes quantized state variables. Tile coding has been applied to standard benchmark domains (Sutton, 1996), to learn to play keepaway soccer (Stone et al., 2005), in multiagent robot learning (Bowling and Veloso, 2003), to train bipedal robots to walk (Schuitema et al., 2005; Tedrake et al., 2004) and to learn mixed strategies in the game of Goofspiel (Bowling and Veloso, 2002). In the game of Go, Silver et al. (2007) obtained good features by enumerating all stone patterns up to a certain size; Sturtevant and White (2006) similarly produced promising results for Hearts by using a feature generation method that enumerated all 2-, 3- and 4-wise combinations of a set of atomic features.

Because exhaustive feature generation often generates more features than can reasonably stored in memory, features generated this way are usually hashed to a

A version of this chapter has been published and presented at the Neural Information Processing Systems Conference (Bellemare et al., 2012b).

smaller vector. The traditional approach in reinforcement learning is to tightly couple hashing with feature generation and consider hashing as part of the generation process, rather than as part of the approximation process. Such a tight coupling ignores the potentially harmful effects of hashing; in typical hashed tile coding, for example, the inner product of two feature vectors is a biased estimate of the inner product between the original (unhashed) vectors. In other words, hashing as it is usually used in reinforcement learning is an unaccounted-for source of error in the value function approximation process.

Recent advances in sketch data structures (e.g. Achlioptas, 2003; Cormode and Muthukrishnan, 2005) and in the study of sparse Johnson-Linderstrauss transforms (Dasgupta et al., 2010; Kane and Nelson, 2010) have produced new algorithms and theoretical support for hashing techniques. One such sketch, the tug-of-war sketch[1] (Cormode and Garofalakis, 2005), leads to unbiased inner products; its computational simplicity is particularly well-suited to linear value function approximation. The aim of this chapter is to describe tug-of-war hashing, based on its namesake sketch, as an alternative to the standard hashing used in feature generation for reinforcement learning. This chapter provides a bound on the approximation error of SARSA(1) with tug-of-war hashing, and a number of empirical results showing the superior performance of tug-of-war hashing over standard hashing.

This chapter is divided as follows. Section 5.1 describes previous theoretical results on hashing and sketches necessary to the rest of the chapter. Section 5.2 then provides notation relevant to hashing in the context of linear value function approximation. Following this, Section 5.3 gives the main theoretical result of the chapter: the convergence of SARSA(1) with tug-of-war hashing. Finally, in Section 5.4 I describe a series of experiments comparing standard hashing and tug-of-war hashing, first considering small benchmark problems before moving on to value function approximation in Atari 2600 games.

5.1 Background

In this section I review the prior work on which this chapter's contribution is built.

[1] Another popular name for the tug-of-war sketch is Fast-AGMS, derived from the initials of the inventors of its predecessor, the AGMS sketch.

5.1.1 Universal Families of Hash Functions

Let $[n] := \{1, 2, \ldots, n\} \subset \mathbb{N}$. A hash function $h : [n] \to [m]$ maps elements from $[n]$ to $[m]$ in constant time; typically, $n \gg m$. I denote by $\mathbb{H}(n, m)$ a family of such functions, and omit the arguments to $\mathbb{H}(n, m)$ whenever unambiguous. $\mathbb{H}$ is said to be a *universal family* of hash functions if, drawing h uniformly at random from $\mathbb{H}$ we have

$$P(h(i_1) = h(i_2)) = \frac{1}{m}$$

for all $i_1, i_2 \in [n], i_1 \neq i_2$ (Carter and Wegman, 1979). A *pairwise independent* universal family offers a stronger guarantee: for all $i_1, i_2 \in [n], i_1 \neq i_2$ and $k_1, k_2 \in [m]$, we have

$$P(h(i_1) = k_1, h(i_2) = k_2) \leq \frac{1}{m^2}. \tag{5.1}$$

The notion of pairwise independence similarly generalizes to k-wise independence, for any integer $k \leq n$; for all integers $l < k$, a k-wise independent family is also l-wise independent. When $\mathbb{H}(n, m)$ is a n-wise independent universal family, we call it a ∞-wise independent family, or more simply an independent universal family.

The notion of k-wise independence arises in the computation of $h(a)$. In this thesis I use the $\mathbb{H}_2$ class[2] introduced by Carter and Wegman (1979). Let $p > n$ be a prime number and two integers $a, b \in \{0, 1, 2, \ldots, p-1\}$. Define the hash function $h_{a,b}(i) : [n] \to [m]$ as

$$h_{a,b}(i) := ai + b \mod p \mod m.$$

In their work, Carter and Wegman showed that the family $\mathbb{H}_2 := \{h_{a,b} : a, b \in \{0, 1, 2, \ldots, p-1\}\}$ is pairwise independent. That is, drawing a hash function uniformly at random from the set $\mathbb{H}_2$ satisfies Equation 5.1. Similarly, the hash function

$$h_{a,b,c,d}(i) := ai^3 + bi^2 + ci + d \mod p \mod m$$

leads to a 4-wise independent family of hash functions; this family is denoted $\mathbb{H}_4 := \{h_{a,b,c,d} : a, b, c, d \in \{0, 1, 2, \ldots, p-1\}\}$.

[2]Carter and Wegman denoted this family as H_1, but here I use the subscript to denote the number of terms in the hash function polynomial.

5.1.2 The Tug-of-War Sketch

The tug-of-war sketch was developed to approximate inner products of large vectors (Cormode and Garofalakis, 2005). The name "sketch" refers to the data structure's function as a summary of a stream of data. As with other sketches (Cormode and Muthukrishnan, 2005; Nelson and Woodruff, 2010), the tug-of-war sketch has two components: a hashing procedure, which maps elements to a hash table, and a duplication procedure, which ensures a high-probability bound on the resulting approximation.

In the canonical sketch setting, a large count vector $\theta \in \mathbb{R}^n$ is summarized using a sketch vector $\tilde{\theta} \in \mathbb{R}^m$. At each time step a vector $\phi_t \in \mathbb{R}^n$ is received; the purpose of the sketch vector is to approximate the count vector $\theta_t := \sum_{i=1}^{t} \phi_i$. Given $h : [n] \to [m]$ drawn from a pairwise independent family and $\xi : [n] \to \{-1, 1\}$ drawn from a 4-wise independent family, the tug-of-war sketch maps ϕ_t to a vector $\tilde{\phi}_t$ whose i^{th} component $\tilde{\phi}_{t,i}$ is defined as

$$\tilde{\phi}_{t,i} := \sum_{j=1}^{n} \mathbb{I}_{[h(j)=i]} \phi_{t,j} \xi(j), \tag{5.2}$$

where $\phi_{t,j}$ is the j^{th} component of the original vector. Throughout this chapter, this operation is referred to as *tug-of-war hashing*. Intuitively, the hash function h encodes which of the m buckets the j^{th} feature is hashed to, while ξ indicates whether to add or subtract feature $\phi_{t,j}$ from the bucket. Using matrix notation, Equation 5.2 becomes

$$\tilde{\phi}_t := H\phi_t$$

where $H \in \{0, \pm 1\}^{m \times n}$ is the tug-of-war hashing matrix whose elements are defined as $H_{ij} := \mathbb{I}_{[h(j)=i]} \xi(j)$.

Let $\tilde{\theta}_t := \sum_{i=1}^{t} \tilde{\phi}_i$. For an arbitrary $\phi \in \mathbb{R}^n$ and its corresponding vector $\tilde{\phi} \in \mathbb{R}^m$, we have

$$\mathbb{E}_{h,\xi}[\tilde{\theta}_t \cdot \tilde{\phi}] = \theta_t \cdot \phi.$$

In other words, the tug-of-war sketch produces unbiased estimates of inner products. The proof relies on the following property of the ξ function:

$$\mathbb{E}_\xi[\xi(j_1)\xi(j_2)] = \begin{cases} 1 & \text{if } j_1 = j_2 \\ 0 & \text{otherwise} \end{cases} \quad \text{(by pairwise independence)} \tag{5.3}$$

Now, consider that $\tilde{\theta} = \sum_{i=1}^{t} \tilde{\phi}_i$, and consider an arbitrary $\phi_{t'}$, where the index t' is used solely for ease of notation. Then $\tilde{\theta}_t \cdot \tilde{\phi}_{t'} = \sum_{i=1}^{t} \tilde{\phi}_i \cdot \tilde{\phi}_{t'}$ and

$$\begin{aligned}
\mathbb{E}_{h,\xi}[\tilde{\phi}_i \cdot \tilde{\phi}_{t'}] &= \phi_i^T H^T H \phi_{t'} \\
&= \mathbb{E}_{h,\xi}\left[\sum_{j_1=1}^{n}\sum_{j_2=1}^{n} \mathbb{I}_{[h(j_1)=h(j_2)]}\phi_{i,j_1}\phi_{t',j_2}\xi(j_1)\xi(j_2)\right] \\
&= \sum_{j_1=1}^{n}\sum_{j_2=1}^{n} \mathbb{E}_h\left[\mathbb{I}_{[h(j_1)=h(j_2)]}\right]\mathbb{E}_\xi\left[\xi(j_1)\xi(j_2)\right]\phi_{i,j_1}\phi_{t',j_2} \\
&= \sum_{j=1}^{n} \phi_{i,j}\phi_{t',j} = \phi_i \cdot \phi_{t'}.
\end{aligned}$$

The proof stems from the independence of h and ξ and Equation 5.3. The tug-of-war sketch stores a number of sketch vectors $\tilde{\theta}^1, \tilde{\theta}^2, \ldots,$ constructed from k independently drawn matrices $H_1, \ldots, H_k$. The dot product $\theta_t \cdot \phi_{t'}$ is then estimated as the median of the set of dot products $\{\tilde{\theta}_t^i \cdot \tilde{\phi}_{t'}^i\}$, where $\tilde{\phi}_{t'}^i := H_i\phi_{t'}$ and $i \in [k]$. This enables us to derive a high probability bound on the sketch's approximation error.

5.1.3 Johnson-Lindenstrauss Transforms

A Johnson-Linderstrauss transform maps a set of points in $\mathbb{R}^n$ to a smaller space $\mathbb{R}^m$ while preserving distances between points (Achlioptas, 2003). The Johnson-Linderstrauss lemma (Frankl and Maehara, 1988) states the existence of such a transform for any set of vectors:

Lemma 5.1.1 (Johnson-Linderstrauss)**.** *Given $\epsilon > 0$, $m, n, k \in \mathbb{N}$ such that $m \geq O(\epsilon^{-2}\log k)$, for every set P of k vectors in $\mathbb{R}^n$ there exists a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ such that for all $x, y \in P$,*

$$(1-\epsilon)\left\|x-y\right\|_2^2 \leq \left\|f(x)-f(y)\right\|_2^2 \leq (1+\epsilon)\left\|x-y\right\|_2^2$$

A variety of Johnson-Linderstrauss transforms have been proposed (Achlioptas, 2003; Maillard and Munos, 2009; Li et al., 2006; Kane and Nelson, 2010). A common Johnson-Linderstrauss transform is the *random projection* (Indyk and Motwani,

1998). Given a set of vectors $P := \{x_1, x_2, \ldots, x_k\}$, let $R \in \mathbb{R}^{m \times n}$ be a matrix whose elements are drawn from $\mathcal{N}(0,1)$, the standard normal distribution. The random projection of x_i is Rx_i, and R is a Johnson-Linderstrauss transform in the sense that if $m \geq O(\epsilon^{-2} \log \frac{n}{\delta})$ then for any $x_i, x_j \in P$, with probability $1 - \delta$ we have

$$(1-\epsilon) \left\| x_i - x_j \right\|_2^2 \leq \left\| Rx_i - Rx_j \right\|_2^2 \leq (1+\epsilon) \left\| x_i - x_j \right\|_2^2$$

5.1.4 Tug-of-War Hashing as a Johnson-Linderstrauss Transform

The theoretical contribution of this chapter relies on interpreting tug-of-war hashing as a particular form of Johnson-Linderstrauss transform. As discussed in Section 5.1.2, tug-of-war hashing can be described using a matrix H whose elements are the composition of the two hash functions h and ξ. Unlike the random projection matrix R, whose elements are drawn i.i.d., the elements of the columns of H are not i.i.d.: $H_{i,j} = 1 \implies H_{i,k} = 0 \ \forall k \in [n], k \neq j$. Despite this difference, tug-of-war hashing is, under certain conditions, a Johnson-Linderstrauss transform. This result is formally stated in the following theorem:

Theorem 5.1.1 (Dasgupta et al. (2010), Theorem 2)**.** *Let $h : [n] \to [m]$ and $\xi : [n] \to \{-1, 1\}$ be two independent hash functions chosen uniformly at random from ∞-universal families and let $H \in \{0, \pm 1\}^{m \times n}$ be a matrix with entries $H_{ij} = \mathbb{I}_{[h(j)=i]} \xi(j)$. Let $\epsilon < 1$, $\delta < \frac{1}{10}$, $m = \frac{12}{\epsilon^2} \log\left(\frac{1}{\delta}\right)$ and $c = \frac{16}{\epsilon} \log\left(\frac{1}{\delta}\right) \log^2\left(\frac{m}{\delta}\right)$. For any given vector $x \in \mathbb{R}^n$ such that $\left\|x\right\|_\infty \leq \frac{1}{\sqrt{c}}$, with probability $1 - 3\delta$, H satisfies the following property:*

$$(1-\epsilon) \left\| x \right\|_2^2 \leq \left\| Hx \right\|_2^2 \leq (1+\epsilon) \left\| x \right\|_2^2 .$$

Theorem 5.1.1 states that, under certain conditions on the input vector x, tug-of-war hashing approximately preserves the norm of x. When δ and ϵ are constant, the requirement on $\left\|x\right\|_\infty$ can be waived by applying Theorem 5.1.1 to the normalized vector $u = \frac{x}{\left\|x\right\|_2 \sqrt{c}}$.

5.2 Notation

Before providing the main theoretical result of this chapter – the convergence of SARSA(λ) with tug-of-war hashing – I first provide the notation needed to describe the role of hashing in linear value function approximation.

5.2.1 Standard Hashing

Recall that $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$ maps state-action pairs to feature vectors, and that the value function $Q_t^\pi(s,a)$ is approximated as $\theta_t \cdot \phi(s,a)$. With standard hashing, the feature vector $\phi(s,a) \in \mathbb{R}^n$ is reduced to a smaller vector $\hat{\phi}(s,a) \in \mathbb{R}^m$ using a hash function $h : [n] \to [m]$. Let $\hat{H} \in \mathbb{R}^{m \times n}$ be a matrix whose elements are $\hat{H}_{i,j} := \mathbb{I}_{[h(j)=i]}$. The standard hashing feature vector $\hat{\phi}$ is defined as:

$$\hat{\phi}(s,a) := \hat{H}\phi(s,a)$$

By contrast to tug-of-war hashing, the entries of the standard hashing matrix $\hat{H}$ are not multiplied by $\xi(j)$. This form of hashing has been successfully used in many reinforcement learning applications (Sutton, 1996; Stone et al., 2005; Schuitema et al., 2005).

Let $\hat{\theta}_t$ be the weight vector representing the standard hashing approximation of $Q^\pi(s,a)$, $\hat{\theta}_t \cdot \hat{\phi}(s,a)$. If $\phi(s,a)$ has k non-zero entries, computing $\hat{\theta}_t \cdot \hat{\phi}(s,a)$ takes $O(k)$ operations: for each feature, one hash from $[n]$ to $[m]$ and one addition.

5.2.2 Tug-of-War Value Function Approximation

Tug-of-war hashing behaves in the same manner as standard hashing, with the addition of the ξ hash function. Given the tug-of-war hashing matrix H described in Section 5.1.2, the tug-of-war feature vector is defined as:

$$\tilde{\phi}(s,a) := H\phi(s,a)$$

Similarly, the tug-of-war value function approximation is defined as $\tilde{Q}_t(s,a) := \tilde{\theta}_t \cdot \tilde{\phi}(s,a)$, where $\tilde{\theta}_t$ is the tug-of-war weight vector. If $\phi(s,a)$ has k non-zero entries, the additional cost of computing $\tilde{\theta}_t \cdot \tilde{\phi}(s,a)$ rather than $\hat{\theta}_t \cdot \hat{\phi}(s,a)$ is $O(k)$. More precisely, this is the cost of evaluating

$$\xi'(i) := ai^3 + bi^2 + ci + d \mod p \mod 2 \tag{5.4}$$

for every $i \in [n]$ such that $\phi_i(s,a) \neq 0$, with the multiplication $\phi_i(s,a)\xi(i)$ implemented as an if/else statement over $\xi'(i)$. In practice, Equation 5.4 can be efficiently implemented using the decomposition $\xi'(i) = d + i(c + i(b + ia)) \mod p \mod 2$ and, on a 64-bit computer[3], $p = 2^{31} - 1$; the algorithm's theoretical guarantees do not

[3] On 32-bit computers one may instead choose $p = 2^{13} - 1$ or $p = 2^{17} - 1$.

depend on the particular choice of p. Overflow is avoided by applying the mod p operation after every multiplication and, if necessary, providing i mod p as the input to ξ.

5.3 Convergence of Tug-of-War SARSA(1)

Recall the SARSA(λ) update equations (Section 2.1.2). The equivalent *tug-of-war SARSA(λ) equations* are

$$\begin{aligned} \tilde{\delta}_t &\leftarrow r_t + \gamma \tilde{\theta}_t \cdot \tilde{\phi}(s_{t+1}, a_{t+1}) - \tilde{\theta}_t \cdot \tilde{\phi}(s_t, a_t) \\ \tilde{e}_t &\leftarrow \gamma\lambda \tilde{e}_{t-1} + \tilde{\phi}(s_t, a_t) \\ \tilde{\theta}_{t+1} &\leftarrow \tilde{\theta}_t + \alpha \tilde{\delta}_t \tilde{e}_t. \end{aligned} \tag{5.5}$$

The aim of this section is to show that the tug-of-war SARSA(1)[4] equations converge to a solution, and to provide an upper bound on the approximation error introduced by tug-of-war hashing. The point of reference in the upper bound is the error of linear function approximation without hashing, i.e. when the full vector $\phi \in \mathbb{R}^n$ is used to approximate $Q(s, a)$. The first part of the proof, Lemma 5.3.1, applies Theorem 5.1.1 to bound the error in the inner product between a set of vectors. I then apply Theorem 2.1.1 to prove the convergence of tug-of-war SARSA(1); a simple application of the triangle inequality and Lemma 5.3.1 yields the desired bound. Of note, the bound provided here is somewhat tighter, as well as more general, than the bound that was originally presented (Bellemare et al., 2012b).

Lemma 5.3.1. *Let $x_1 \dots x_K$ and y be vectors in $\mathbb{R}^n$. Let $H \in \{0, \pm 1\}^{m \times n}$ with ϵ, δ and m defined as in Theorem 5.1.1. With probability at least $1 - 3(K+1)\delta$, for all $k \in \{1, \dots, K\}$,*

$$x_k \cdot y - \epsilon |x_k \cdot y| \leq Hx_k \cdot Hy \leq x_k \cdot y + \epsilon |x_k \cdot y|.$$

Proof. The proof given here is an adaptation of the proof of Maillard and Munos (2009), and involves studying the eigenvalues of H within the subspace spanned by $x_1, \dots, x_K$ and y. For notational simplicity, define $x_{K+1} := y$. Let $X \in \mathbb{R}^{n \times (K+1)}$ be the matrix whose columns are $x_1, \dots, x_{K+1}$:

[4] The proof technique used here does not seem to carry to the case of arbitrary $\lambda \in [0, 1]$.

$$X := \begin{bmatrix} | & \dots & | \\ x_1 & \dots & x_{K+1} \\ | & \dots & | \end{bmatrix}$$

Denote by C the subspace spanned by the columns of X and by $C_\perp$ the corresponding nullspace. Let $p := \text{rank}(C) \leq K+1$ such that also $\text{rank}(C_\perp) = n-p$, and denote by $c_1, c_2, \dots, c_p$ an orthonormal basis for C; thus we can express any vector x_k, $k = 1 \dots K+1$, as a linear combination $x_k = \sum_{i=1}^{p} \alpha_{k,i} c_i$ with $\alpha_{k,i} \in \mathbb{R}$. Now decompose H into two matrices $H := H_1 + H_2$ such that for all $x \in C$, $H_2 x = 0$ and for all $x \in C_\perp$, $H_1 x = 0$. Effectively, the subspace spanned by the rows of H_2 is orthogonal to C and the subspace spanned by the rows of H_1, orthogonal to $C_\perp$. It follows that $H^T H = H_1^T H_1 + H_2^T H_2$. Now we apply Theorem 5.1.1 to $c_1, c_2, \dots, c_p$: with probability $1 - 3p\delta \geq 1 - 3(K+1)\delta$, for every c_i, $i = 1 \dots p$, we have

$$(1-\epsilon) \|c_i\|_2^2 \leq \|Hc_i\|_2^2 \leq (1+\epsilon) \|c_i\|_2^2$$

Because $c_i \in C$, $H_2 c_i = 0$ and $\|Hc_i\|_2^2 = \|H_1 c_i\|_2^2$. The above equation can then be expressed to highlight the eigenstructure of H_1:

$$(1-\epsilon) c_i^T c_i \leq c_i^T H_1^T H_1 c_i \leq (1+\epsilon) c_i^T c_i,$$

thus guaranteeing that the largest eigenvalue of $H_1^T H_1$ is at most $1+\epsilon$ and its smallest eigenvalue, at least $1-\epsilon$. Because $H_1^T H_1$ is symmetric, the eigenvalues of $H_1^T H_1 - I$ are similarly bounded between $-\epsilon$ and ϵ. We can then prove the statement of the lemma. We rewrite the dot product $Hx_i \cdot Hx_j$ as

$$\begin{aligned} Hx_i \cdot Hx_j &= H_1 x_i \cdot H_1 x_j \text{ (by choice of } H_1) \\ &= x_i^T H_1^T H_1 x_j \\ &= x_i^T x_j + x_i^T (H_1^T H_1 - I) x_j \\ &\leq x_i^T x_j + \epsilon |x_i^T x_j| \\ &= x_i \cdot j_j + \epsilon |x_i \cdot x_j| \end{aligned}$$

The lower bound is derived from a similar argument, thus completing the proof. □

Recall from Section 2.1.4 the matrix with rows $\phi(s,a)$, $\Phi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times n}$. Here I similarly denote by $\tilde{\Phi} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times m}$ the matrix of tug-of-war feature vectors $\tilde{\phi}(s,a)$,

such that $\tilde{\Phi} := H\Phi$. I consider the case of SARSA(1) learning $Q^\pi(s,a)$ for a given policy π and MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$. We make the following assumptions, of which 1–3 are restated from Section 2.1.4:

Assumptions

1. The Markov chain induced by π and $\mathcal{M}$ is ergodic with unique stationary distribution $\mu \in \textsc{Dist}(\mathcal{S} \times \mathcal{A})$,
2. Φ has full column rank, i.e. there are no redundant features,
3. The step-sizes α_t are positive, nonincreasing and predetermined; furthermore, $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$,
4. $\tilde{\Phi}$ has full column rank.

When the last assumption is not satisfied, SARSA(1) converges to a set of solutions $\tilde{\Theta}^\pi$ satisfying the bound of Theorem 5.3.1.

Theorem 5.3.1. *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ be an MDP and $\pi : \mathcal{S} \to \textsc{Dist}(\mathcal{A})$ be a policy. Let $\Phi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times n}$ be the matrix of full feature vectors and $\tilde{\Phi} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}| \times m}$ be the matrix of tug-of-war vectors. Denote by μ the stationary distribution on $(\mathcal{S}, \mathcal{A})$ induced by π and P. Let $\epsilon < 1$, $\delta < 1$, $\delta' = \min\left\{\frac{1}{10}, \frac{\delta}{3(|\mathcal{S}||\mathcal{A}|+1)}\right\}$ and $m \geq \frac{12}{\epsilon^2} \log \frac{1}{\delta'}$. Under assumptions 1-4), SARSA(1) with tug-of-war hashing (Equations 5.5) converges to a unique $\tilde{\theta}^\pi \in \mathbb{R}^m$ and with probability at least $1 - \delta$*

$$\left\|\tilde{\Phi}\tilde{\theta}^\pi - Q^\pi\right\|_\mu \leq \left\|\Phi\theta^\pi - Q^\pi\right\|_\mu + \epsilon \left\|\Phi\theta^\pi\right\|_\mu ,$$

where $Q^\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is the exact value function and $\theta^\pi = \arg\min_\theta \left\|\Phi\theta - Q^\pi\right\|_\mu$.

Proof. First note that Theorem 2.1.1 implies the convergence of SARSA(1) with tug-of-war hashing to a unique solution $\tilde{\theta}^\pi$. We apply Lemma 5.3.1 to the set $\{\phi(s,a) : (s,a) \in \mathcal{S} \times \mathcal{A}\}$ and θ^π. By our choice of m, for all $(s,a) \in \mathcal{S} \times \mathcal{A}$ and with probability at least $1 - 3(|\mathcal{S}||\mathcal{A}| + 1)\delta' = 1 - \delta$,

$$|H\phi(s,a) \cdot H\theta^\pi - \phi(s,a) \cdot \theta^\pi| \leq \epsilon |\theta^\pi \cdot \phi(s,a)| \ .$$

SARSA(1) converges to $\tilde{\theta}^\pi = \arg\min_\theta \left\|\tilde{\Phi}\theta - Q^\pi\right\|_\mu$; compared to $\tilde{\Phi}\tilde{\theta}^\pi$, the solu-

tion $\tilde{\Phi} H \theta^\pi$ is thus an equal or worse approximation to Q^π. It follows that

$$\begin{aligned}
\left\| \tilde{\Phi}\tilde{\theta}^\pi - Q^\pi \right\|_\mu &\leq \left\| \tilde{\Phi} H\theta^\pi - Q^\pi \right\|_\mu \leq \left\| \tilde{\Phi} H\theta^\pi - \Phi\theta^\pi \right\|_\mu + \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu \\
&= \sqrt{\sum_{s\in\mathcal{S}, a\in\mathcal{A}} \mu(s,a) \left[H\phi(s,a)\cdot H\theta^\pi - \phi(s,a)\cdot\theta^\pi \right]^2} + \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu \\
&\leq \sqrt{\sum_{s\in\mathcal{S}, a\in\mathcal{A}} \mu(s,a) \left[\epsilon |\theta^\pi \cdot \phi(s,a)| \right]^2} + \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu \\
&\leq \epsilon \left\| \Phi\theta^\pi \right\|_\mu + \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu ,
\end{aligned}$$

where the second inequality follows from Lemma 5.3.1. □

Of note, the error term in Theorem 5.3.1 is proportional to the μ-weighted state-value approximation $Q(s,a) = \theta^\pi \cdot \phi(s,a)$. The significance of this error term can be made clearer through Hölder's inequality. Given two vectors $x, y \in \mathbb{R}^n$, Hölder's inequality states that

$$|x \cdot y| \leq \|x\|_p \|y\|_q,$$

where $p, q \in (0, \infty]$ such that $\frac{1}{p} + \frac{1}{q} = 1$. Using this inequality, we can derive a bound for the common occurrence in reinforcement learning domains where the agent is provided with a sparse, binary feature vector ϕ. This case arises, for example, when using tile coding. Assuming that ϕ is composed of k binary features, such that for all $s \in \mathcal{S}, a \in \mathcal{A}$, $|\phi(s,a)| = k$, applying Hölder's inequality with $p = 1$, $q = \infty$ yields

$$\begin{aligned}
\left\| \tilde{\Phi}\tilde{\theta}^\pi - Q^\pi \right\|_\mu &\leq \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu + \epsilon \left\| \Phi\theta^\pi \right\|_\mu \\
&\leq \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu + \epsilon \sup_{s\in\mathcal{S}, a\in\mathcal{A}} |\theta^\pi \cdot \phi(s,a)| \\
&\leq \left\| \Phi\theta^\pi - Q^\pi \right\|_\mu + \epsilon k \left\| \theta^\pi \right\|_\infty ,
\end{aligned}$$

showing that in this case the tug-of-war error directly depends on the largest component of the weight vector θ^π. Hölder's inequality can similarly be used to recover the bound given in the original paper (Bellemare et al., 2012b).

Although the tug-of-war sketch produces unbiased estimates of inner products, Theorem 5.3.1 implies that the tug-of-war value function approximation is a biased estimate of the full feature vector approximation. This bias arises from complex interactions between the TD-error $\tilde{\delta}_t$ and $\tilde{\phi}_t$ in Equation 5.5. These interactions are due to the self-referential nature of the TD error, and are difficult to analyze

in closed form; instead, in the next section I provide empirical results that directly study this value function bias.

5.4 Empirical Study

This section begins with an empirical comparison of the bias and mean squared error of standard hashing and tug-of-war hashing, then moves on to experimental results on two benchmark domains and Atari 2600 games. The list of testing games used in this section is provided in Appendix B.

Benchmark Domains

Mountain Car is a two-dimensional, continuous-state domain widely used for benchmarking reinforcement learning algorithms (Sutton, 1996; Sutton and Barto, 1998). In Mountain Car (Figure 5.1, left), the agent's goal is to navigate an underpowered car from the bottom to the top of a hill. The agent has access to three actions: forward, backward and stay. At each time step the agent receives a reward of -1, except at the goal where it receives no reward.

Acrobot (Figure 5.1, right) is a four-dimensional, continuous-state benchmark domain (Sutton, 1996). The agent's goal is to bring the tip of a two-jointed pendulum above a fixed height by varying the torque at the second joint; three torque values are available. As in Mountain Car, the agent receives a reward of -1 at every step except at the goal.

Implementation Notes

All agents in this section learn using SARSA(λ) and behave according to the ϵ-greedy action selection mechanism. The hash function h is drawn uniformly at random from the $\mathbb{H}_2$ family and ξ, from the $\mathbb{H}_4$ family (Section 5.1.1). Throughout, the feature vector weights are initialized to 0; this ensures that every state is assigned the same initial value. Both Mountain Car and Acrobot feature vectors are constructed using tile coding, a feature generation method that divides the state space into overlapping tilings (Miller and Glanz, 1996; Sutton, 1996). Figure 5.2 depicts a typical tiling of the space, in which the range of every state variable is divided into a number of equal intervals. Each tile corresponds to a binary feature, and each tiling is offset from its predecessor. If k denotes the number of tilings, then the resulting feature vector $\phi(s_t, a_t)$ contains k non-zero entries at each time step t. The Mountain Car

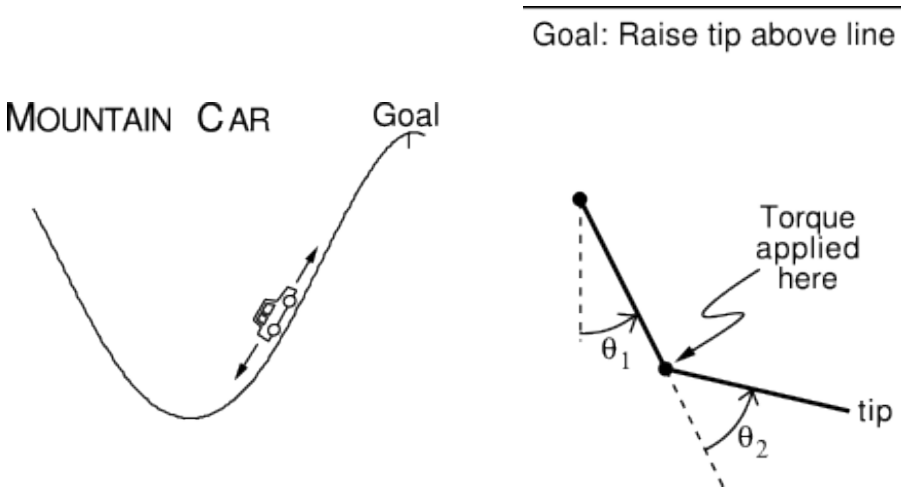

Figure 5.1: **Left.** The Mountain Car domain. **Right.** The Acrobot domain. Both figures are reproduced from the work of Sutton (1996).

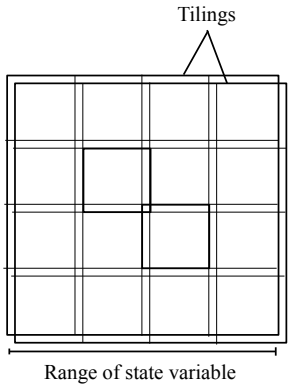

Figure 5.2: Example tile coding over a two-dimensional state-space. For clarity, only two tilings are depicted.

state is represented using 10 9×9 tilings and the Acrobot state, 48 $6 \times 6 \times 6 \times 6$ tilings. The procedure for generating features in Atari 2600 games is described in Appendix A.

5.4.1 Bias of Tug-of-War Hashing

The aim of this section is to provide empirical evidence that the bias in the tug-of-war value function estimate is significantly smaller than the bias in the value function obtained through standard hashing. In this experiment, the agent uses SARSA(0) to learn the *full vector approximation* (no hashing) to a value function over a short trajectory in either the Mountain Car or Acrobot domain. Simultaneously the agent

also applies SARSA(0) with standard hashing and tug-of-war hashing (Section 5.5). Bias is computed as the difference between exact and hashed value functions.

One trial in the experiment consists in recording updates along a fixed, 1,000-step trajectory obtained by following an ϵ-greedy policy. At every step of this trajectory the agent updates a full feature weight vector θ_t using SARSA(0) with $\gamma = 1.0$ and $\alpha = 0.01$. Parallel to this update the agent also updates a tug-of-war weight vector $\tilde{\theta}_t$ and standard hashing weight vector $\hat{\theta}_t$ using the same values of γ and α. Both methods use a hash table size of $m = 100$ and the same randomly selected hash function $h \in \mathbb{H}_2$; tug-of-war hashing also use a randomly selected hash function $\xi \in \mathbb{H}_4$. At every step the experiment records the difference in value function approximations:

$$\begin{aligned}
\hat{Q}_t(s_t, a_t) - Q_t(s_t, a_t) &:= \hat{\theta}_t \cdot \hat{\phi}(s_t, a_t) - \theta_t \cdot \phi(s_t, a_t) \text{ (standard hashing)} \\
\tilde{Q}_t(s_t, a_t) - Q_t(s_t, a_t) &:= \tilde{\theta}_t \cdot \tilde{\phi}(s_t, a_t) - \theta_t \cdot \phi(s_t, a_t) \text{ (tug-of-war hashing)}
\end{aligned}$$

The experiment consisted in one million trials, each using hash functions selected uniformly at random, and served to estimate two quantities:

1. the relative biases $\frac{\left|\mathbb{E}\left[\hat{Q}_t(s_t,a_t)\right] - Q_t(s_t,a_t)\right|}{|Q_t(s_t,a_t)|}$ and $\frac{\left|\mathbb{E}\left[\tilde{Q}_t(s_t,a_t)\right] - Q_t(s_t,a_t)\right|}{|Q_t(s_t,a_t)|}$
2. the mean squared errors $\frac{\mathbb{E}\left(\hat{Q}_t(s_t,a_t) - Q_t(s_t,a_t)\right)^2}{Q_t(s_t,a_t)^2}$ and $\frac{\mathbb{E}\left(\tilde{Q}_t(s_t,a_t) - Q_t(s_t,a_t)\right)^2}{Q_t(s_t,a_t)^2}$

Figure 5.3 shows the results of this experiment. The difference is unequivocal: the bias from tug-of-war hashing is minimal compared to the bias from standard hashing.

5.4.2 Tug-of-War Hashing for Control

So far in this chapter the focus has been on the *evaluation* aspect of reinforcement learning: estimating the value function for a fixed policy π, and determining the approximation error of tug-of-war hashing with respect to this value function. A related question is how tug-of-war hashing affects *control*. In the control case, the aim is to produce a policy that achieves high return. Theorem 5.3.1 provides no guidance with respect to control, nor do the bias results of the previous section guarantee than tug-of-war SARSA(λ) achieves better policies than SARSA(λ) with standard hashing. Here I empirically compare the control performance of SARSA(λ) with standard and tug-of-war hashing on Mountain Car and Acrobot.

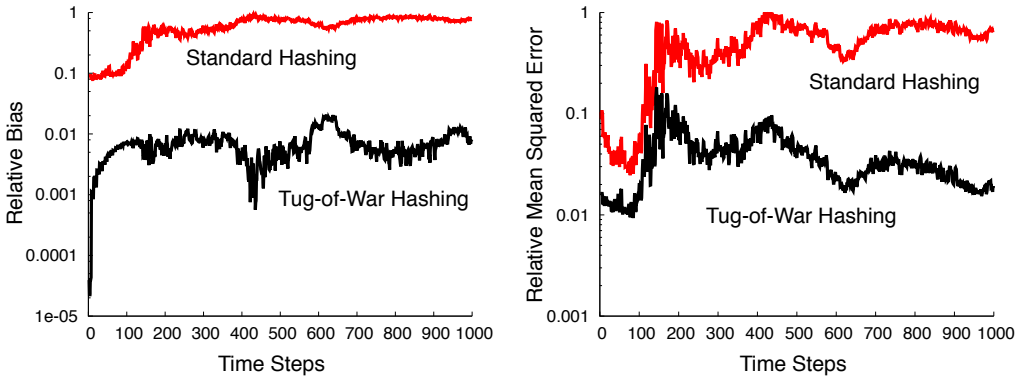

Figure 5.3: Bias and mean squared error in Mountain Car. Note the log scale; both values are relative to the exact value function estimate at time t (see text for details). Note that the relative bias and mean squared errors at $t = 0$ are defined to be 0 (not shown on the graphs).

For each domain and each hashing method I performed a parameter sweep over the learning rate α and selected the best value which did not cause the value estimates to diverge[5]. For Mountain Car these values were $\alpha = 0.5$ (standard hashing) and $\alpha = 0.2$ (tug-of-war hashing); for Acrobot these values were respectively $\alpha = 0.02$ and $\alpha = 0.1$. Other parameters were set to $\gamma = 1.0$, $\lambda = 0.9$, $\epsilon = 0.0$.

I experimented with hash table sizes $m \in [20, 1000]$ for Mountain Car and $m \in [100, 2000]$ for Acrobot. Each experiment consisted of 100 trials; the sources of inter-trial variation were the choice of hash function(s) and random action selection by the ϵ-greedy policy. One trial consisted of 10,000 episodes whose length was restricted to a maximum of 5,000 steps. At the end of each trial, I disabled learning by setting $\alpha = 0$ and evaluated the agents on an additional 500 episodes.

Figure 5.4 shows the performance of standard hashing and tug-of-war hashing as a function of the hash table size. When the hashed vector is small relative to the full vector – but not so small that approximation becomes simply impossible –, tug-of-war hashing performs better than standard hashing. This is especially true in Acrobot, where the number of features (over 62,000) necessarily results in harmful collisions. Note that the right half of the Mountain Car graph in Figure 5.4 describes the regimen where m, the size of the hashed feature vector, is close to n the size of the original feature vector ($10 \times 10 \times 9 = 810$ for Mountain car). In this case, both algorithms naturally perform equivalently.

[5] As in Section 4.5.1, the actual learning rate α_t is α divided by $\max_t \|\phi(s_t, a_t)\|_0$.

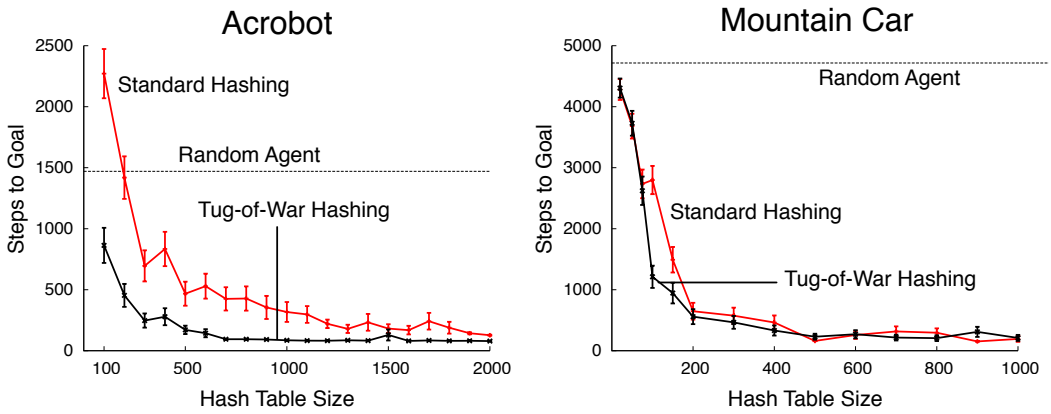

Figure 5.4: Performance of standard hashing and tug-of-war hashing in two benchmark domains. The performance of the random agent is provided as reference.

5.4.3 Evaluation on Atari 2600 Games

To conclude this chapter, I now provide empirical results comparing the two hashing methods on a suite of Atari 2600 games. By virtue of its size, the Atari 2600 platform offers a challenging set of domains for reinforcement learning agents. The feature generation scheme used here (see Appendix A for details) produces over half a billion potential features; explicitly representing such a feature vector is not an option, and so hashing becomes necessary.

I trained ϵ-greedy SARSA(0) agents using both standard hashing and tug-of-war hashing with hash tables of size $m = 1{,}000$, 5,000 and 20,000. I chose the step-size α using a parameter sweep over the training games, selecting the best-performing α that never resulted in divergence in the value function[6]. For standard hashing, $\alpha = 0.01, 0.05, 0.2$ for $m = 1{,}000$, 5,000 and 20,000, respectively. For tug-of-war hashing, $\alpha = 0.5$ across table sizes. I set $\gamma = 0.999$ and $\epsilon = 0.05$. Each experiment was repeated over ten trials lasting 10,000 episodes each; episodes were limited to 18,000 frames to avoid issues with non-terminating policies. One time step lasted five frames, during which the selected action was repeated.

Figure 5.5 compares the inter-algorithm score distributions (Section 3.2) of agents using either standard hashing or tug-of-war hashing for $m = 1{,}000$, 5,000 and 20,000. Tug-of-war hashing consistently outperforms standard hashing across hash table sizes. For each m and each game, I also performed a two-tailed Welch's t-test with 99% confidence intervals to determine the statistical significance of the

[6]The actual learning rate α_t is α divided by $\max_t \|\phi(s_t, a_t)\|_0$.

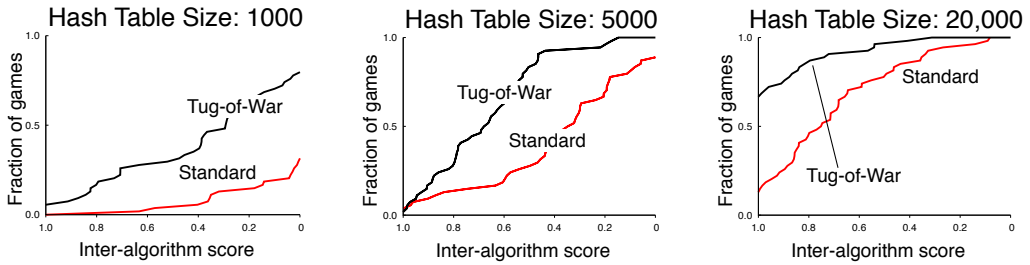

Figure 5.5: Inter-algorithm score distributions over fifty-five Atari 2600 games. Higher curves reflect higher normalized scores.

average score difference between the two methods. For $m = 1{,}000$, tug-of-war hashing performed statistically better in 38 games and worse in 5; for $m = 5{,}000$, it performed better in 41 games and worse in 7; and for $m = 20{,}000$ it performed better in 35 games and worse in 5. The results on Atari 2600 games confirm what was observed on Mountain Car and Acrobot: in practice, tug-of-war hashing performs much better than standard hashing. Furthermore, computing the ξ function took less than 0.3% of the total experiment time, a negligible cost in comparison to the benefits of using tug-of-war hashing.

5.5 Conclusion

In this chapter I adapted tug-of-war hashing to value function approximation. The main contributions of this work are

1. A convergence guarantee for SARSA(1) with tug-of-war hashing,
2. A bound on the additional approximation error due to tug-of-war hashing, and
3. Empirical results showing the superiority of tug-of-war value function approximation, compared to value function approximation using standard hashing.

In most empirical results in this chapter, the hash table size was chosen much lower than what a modern computer's memory can support. It is therefore reasonable to wonder whether tug-of-war hashing is necessary in practical applications, or if its improved performance is simply an artifact of experimental design. I now provide a few reasons why tug-of-war hashing should still be preferred to standard hashing in the context of value function approximation.

1. **Theoretical guarantees**. Interpreting tug-of-war hashing as a Johnson-Linderstrauss transform allows us to guarantee its good behaviour.

2. **Minimal overhead.** In most applications, the additional cost of computing the ξ function is negligible.

3. **Learning about more tasks**. In settings where the agent must learn about many policies or tasks, such as with the Horde architecture (Sutton et al., 2011), the memory required to store value functions restricts scalability.

In all but a few domains, tug-of-war hashing offers something more than standard hashing without a significant computational cost. Wherever linear value function approximation with hashing is used to solve large reinforcement learning domains, it is my expectation that tug-of-war hashing will improve performance.

An oft-heard criticism of hashing is that, when memory is limited it is bound to negatively affect performance, while if memory is not an issue we should avoid it altogether. This criticism is perhaps best answered with a quote from Sutton (1996): *"Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task."*

5.6 Related Work

While hashing has long been established for linear value function approximation (Sutton, 1996), the study of random projections (Frankl and Maehara, 1988) in this context is fairly recent. Using well-established theoretical tools, Ghavamzadeh et al. (2010) provided finite-sample bounds on the approximation error arising from the use of least-squared temporal difference (LSTD) learning (Bradtke and Barto, 1996) in conjunction with random projections; however, the practical benefits of their method remains to be empirically validated.

Sketch algorithms, while not based on the Johnson-Linderstrauss lemma, also use hashing to approximate large vectors. The Count-Min sketch (Cormode and Muthukrishnan, 2005), which was the starting point of the work presented in this chapter, uses hashing and feature duplication to guarantee a bound on the L_1 norm of the approximated vector. The tug-of-war sketch (Cormode and Garofalakis, 2005) uses the same tools to provide similar guarantees on the L_2 norm of a vector; it differs

from the Count-Min sketch through its use of $\{-1, 1\}$ variables, as was done here. Other sketches have been studied and improved for various purposes, such as the Cauchy sketches used by Nelson and Woodruff (2010) to obtain a fast sketch for L_1 norms. A good comparison of various sketches from a practical perspective can be found in the work of Rusu and Dobra (2007).

Although random projections are powerful and mathematically elegant, their naive application requires $O(mn)$ time, where n is the dimension of the input space and m the dimension of the output space. To overcome this computational hurdle, Achlioptas (2003) proposed a sparse random projection, represented as a matrix whose m rows each contains $k \sim \frac{n}{3}$ nonzero entries. He further proposed to use $\{-1, +1\}$, rather than normally distributed variables, to speed up the projection. This work was later followed by even sparser random projections (Li et al., 2006), using $k \sim \frac{1}{\sqrt{n}}$ and a slightly different theoretical analysis.

Using hashing in machine learning is computationally appealing: hashing is much faster than random projections, even of the sparse variety. The idea of combining hashing to kernel methods was championed by Shi et al. (2009). In this work the authors studied hash kernels, which approximate the typical inner product of a kernel with a hash function, and suggested that the large number of duplicate features in a typical machine learning application ensures smooth behaviour in hash kernels. Additional theoretical results on this topic were provided by Weinberger et al. (2009), leading to an approximation bound that depends strongly on the inf-norm of the input vectors. Simultaneously, Dasgupta et al. (2010) provided an interpretation of hashing as a Johnson-Linderstrauss transform; their work is the basis of the theoretical results in this chapter. Kane and Nelson (2010) subsequently improved their results to provide some definitive answers as to the benefits and limitations of using hashing as a Johnson-Linderstrauss transform. Of note, their work applies to k-independent hash functions; such functions can be implemented in practice, in contrast to the fully independent functions required by Dasgupta et al. (2010).

The universe (which others call the Library) is composed of an indefinite, perhaps an infinite, number of hexagonal galleries [...] From any hexagon the upper or lower stories are visible, interminably.

The Library of Babel
Jorge Luis Borges
(Translation Anthony Kerrigan)

Chapter 6

Model Learning in Large, Factored Domains

In the previous two chapters I described algorithms directly or indirectly aimed at improving *model-free* reinforcement learning methods such as SARSA(λ). Model-free methods, as their name implies, seek to learn a policy without using a forward model of the environment. While such methods are often simple to deploy, using a forward model can greatly simplify the agent's decision-making process. For example, UCT (Kocsis and Szepesvári, 2006), POCMP (Silver and Veness, 2010) and FSSS (Walsh et al., 2010) can all be used to produce refined value function estimates tailored to the current situation facing the agent. Within the context of Bayesian reinforcement learning, forward search can provide a principled means to explore the environment (Asmuth and Littman, 2011; Guez et al., 2012). Having access to a model also allows for hybrid techniques such as Dyna (Sutton, 1991; Silver et al., 2008) and TD-Search (Silver et al., 2012), which use a model of the environment to enhance the performance of more traditional model-free reinforcement learning techniques.

When the environment dynamics are unknown a priori, model-based agents must learn their model from experience. A variety of promising approaches have been recently put forward; of note, Doshi-Velez (2009), Walsh et al. (2010), Veness et al. (2010), Veness et al. (2011), Nguyen et al. (2012), and Guez et al. (2012) have collectively demonstrated that it is feasible to learn good probabilistic models of small but challenging domains containing various degrees of partial observability and stochasticity. However, most of these methods have yet to be applied to domains

A version of this chapter has been published and presented at the International Conference on Machine Learning (Bellemare et al., 2013b).

whose observation space is as large as the space of Atari 2600 game screens.

Although learning arbitrary environment dynamics is challenging, many domains of interest possess a structured observation space. In turn, this structure enables us to derive efficient model learning algorithms. This chapter provides a theoretical framework for describing a particular kind of observation structure: recursively decomposable observation spaces. From this theoretical framework I derive an algorithm for learning forward models in domains with recursively decomposable observation spaces. Key to this algorithm is its efficient Bayesian model averaging over a large class of factorizations of the observation space. While prior work on factored models (Ross and Pineau, 2008; Poupart, 2008; Diuk et al., 2009) has assumed a known factorization, this new algorithm is guaranteed to asymptotically perform as well as any factorization within its class.

The second part of this chapter provides an instantiation of the general algorithm for recursively decomposable spaces. This instantiation, the quad-tree factorization (QTF) algorithm, is easily applied to the Atari 2600 observation space by recursively decomposing the image space into image patches. As a result, we can efficiently learn a variable-resolution forward model of Atari 2600 games. In turn, this variable decomposition allows us to sample quickly whenever large image patches are sufficient. In the implementation described here, QTF requires only a fraction more computation than an equivalent algorithm learning a forward model using only the finest-resolution factorization available; essentially, memory is the limiting factor of our implementation. At the end of this chapter I provide a variety of model learning results exemplifying the benefits of using QTF.

This chapter is divided as follows. In Section 6.1 I review the Context Tree Weighting algorithm of Willems et al. (1995); its extensions Context Tree Switching and Action-Conditional CTW; and describe the Sparse Sequential Dirichlet estimator. Following this, Sections 6.2 and 6.3 lay out the general framework of recursive factorizations. Section 6.4 then describes the quad-tree factorization algorithm. Empirical results on using QTF for learning forward models of Atari 2600 games are then given in Section 6.5.

6.1 Background

This chapter builds on the work of Willems et al. (1995) on the Context Tree Weighting algorithm and its recent extension to the approximate AIXI setting by Veness

et al. (2011). I now review both ideas and additional improvements that enable us to learn forward screen models of Atari 2600 games.

This section (and the rest of the chapter) uses the AIXI notation described in Section 2.2. The Context Tree Weighting algorithm, however, was designed for the compression setting. As such, in what follows I focus on the sequence prediction problem: predicting x_n given $x_{<n}$. By contrast, the case that interests us is the action-conditional sequence prediction problem: predicting x_n given $(ax)_{<n}$ and a_n; thus at the end of this section I discuss the extension of Context Tree Weighting to action-conditional environment models.

6.1.1 Context Tree Weighting

The Context Tree Weighting (CTW) algorithm (Willems et al., 1995) is a method for modelling binary sequences in the context of universal data compression. The algorithm's appeal lies in that it asymptotically achieves the lower bound on encoding k-Markov sequences: sequences for which the probability distribution over the next symbol x_n only depends on $\{x_{n-k}, \ldots, x_{n-1}\}$. To achieve this lower bound, CTW combines three key components: prediction suffix trees, the Krichevsky–Trofimov parameter estimator, and Bayesian model averaging.

Prediction Suffix Trees

In the binary compression setting, a (binary) prediction suffix tree (PST) (Rissanen, 1983; Ron et al., 1996) is a proper binary tree whose leaves correspond to probability distributions over the binary alphabet $\mathcal{X}$. Formally, a binary PST is described as a pair $\mathcal{T} := (S, \Psi)$ where S is a proper and complete set of suffices and Ψ is a set of probability distributions over $\mathcal{X}$. The PST maps each binary string $x_{1:n} \in \mathcal{X}^n$ to a probability distribution over the next symbol $x_{n+1} \in \mathcal{X}$ by first mapping $x_{1:n}$ to the matching suffix $s \in S$, then mapping s to its corresponding probability distribution $\psi \in \Psi$. Figure 6.1 depicts a depth 3 binary prediction suffix tree; each leaf l corresponds to a unique suffix $s(l) \in S$ and contains a Bernoulli distribution parametrized by $\theta_{s(l)}$. Whenever unambiguous, l directly refers to the suffix $s(l)$, so that $\theta_l := \theta_{s(l)}$.

Let d be the depth of $\mathcal{T}$. Given a string $c \in S$ of length $k \leq d$, let $x^c_{1:n}$ denote the substring $x_{i_1} x_{i_2} \ldots x_{i_m}$ of $x_{1:n}$ such that[1] for all i_j, $j \leq m$, $x_{i_j-k:i_j-1} = c$. A

[1]For convenience, $x_{1:n}$ can be extended to non-positive indices by defining $x_i = 0$ for $i < 1$.

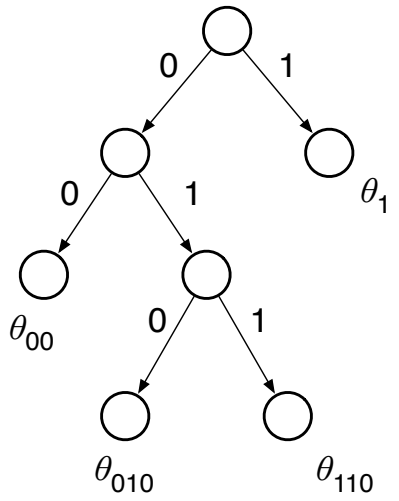

Figure 6.1: A binary prediction suffix tree. Each leaf corresponds to a Bernoulli distribution parametrized by θ_l, where $l \in \mathcal{X}^*$ is the path (of length at most 3) from the root to the leaf.

prediction suffix tree $\mathcal{T} = (S, \Psi)$ thus partitions $x_{1:n}$ into $|S|$ substrings, where each substring corresponds to a different context.

The KT Estimator

When the parameters θ_l at the leaves are unknown they must be learned from data. In Context Tree Weighting this is done by treating the data falling under each leaf as being generated by a stationary, memoryless source; θ_l is then estimated using the *Krichevsky–Trofimov (KT) estimator*.

The Krichevsky-Trofimov estimator, first proposed by Krichevsky and Trofimov (1981), represents the uncertainty over the parameter θ_l using a prior $\text{Beta}_{\frac{1}{2},\frac{1}{2}}(\theta_l)$. This prior is a special case of the *Beta distribution*, defined as

$$\text{Beta}_{a,b}(\theta) := \frac{\theta^{a-1}(1-\theta)^{b-1}}{\text{B}(a,b)},$$

where a and b are parameters of the Beta distribution and $\text{B}(a,b)$ is the Beta function defined as

$$\text{B}(a,b) := \int_{x=0}^{1} x^{a-1}(1-x)^{b-1}\text{d}x \ .$$

Consider a random Bernoulli variable X whose parameter θ is unknown, and denote by $\Pr(\theta)$ the Beta prior $\text{Beta}_{a,b}(\theta)$ over this parameter. Suppose that we observe k samples $x_1, \ldots, x_k \sim X$. By Bayes' rule, the posterior distribution $\Pr(\theta|x_1, \ldots, x_k)$ is

$$\Pr(\theta|x_1, \ldots, x_k) := \frac{\Pr(x_1, \ldots, x_k|\theta)\Pr(\theta)}{\int_0^1 \Pr(x_1, \ldots, x_k|\theta')\Pr(\theta')d\theta'}$$

which can be shown to simplify to

$$\Pr(\theta|x_1, \ldots, x_k) := \text{Beta}(a + N_1, b + N_0),$$

where $N_x := \sum_{i=1}^k \mathbb{I}_{[x_i=x]}$. Thus the parameters a and b (and posterior parameters $a + N_1$ and $b + N_0$) can be interpreted as counting how many times each symbol has been observed. Since $\Pr(x_1, \ldots, x_k|\theta)$ is distributed according to a Binomial distribution, the Beta prior is said to be conjugate to the Binomial distribution. Furthermore,

$$\mathbb{E}_\theta\left[\text{Beta}_{a,b}(\theta)\right] = \frac{a}{a+b}$$

The KT estimator assigns prior counts $a = b = \frac{1}{2}$; these counts are updated with every new symbol. This particular choice of prior counts guarantees uniform convergence of the KT estimator to the Bernoulli distribution it estimates (Krichevsky and Trofimov, 1981), and is used to derive the bound of Lemma 6.1.1 below. Given a string $x_{1:n}$, denote by $N_x(x_{1:n}) := \sum_{i=1}^n \mathbb{I}_{[x_i=x]}$ the number of occurrences of $x \in \mathcal{X}$ in $x_{1:n}$ and define the probability assigned to x by the KT estimator as

$$\text{KT}(x|x_{1:n}) := \frac{\mathbb{I}_{[x=0]}N_0(x_{1:n}) + \mathbb{I}_{[x=1]}N_1(x_{1:n}) + \frac{1}{2}}{N_0(x_{1:n}) + N_1(x_{1:n}) + 1} \tag{6.1}$$

The following lemma, due to Willems et al. (1995), bounds the redundancy of the KT estimator:

Lemma 6.1.1. *Let $x_{1:n} \in \mathcal{X}^n$ be a binary string generated by a Bernoulli source μ with unknown parameter θ_μ. Let $\mu(x_{1:n}) := \theta_\mu^{N_1(x_{1:n})}(1-\theta_\mu)^{N_0(x_{1:n})}$ be the probability of $x_{1:n}$ under μ, and let $\text{KT}(x_{1:n}) := \prod_{i=1}^n \text{KT}(x_i|x_{<i})$ be the probability of $x_{1:n}$ under the KT estimator. Then*

$$-\log_2 KT(x_{1:n}) - (-\log_2 \mu(x_{1:n})) \leq \frac{1}{2}\log_2(n) + 1$$

Context Tree Weighting

The idea behind the Context Tree Weighting algorithm is to perform Bayesian model averaging over the set of all possible, bounded depth prediction suffix trees with KT estimators at the leaves. Let $\mathbb{T}$ be the set of all prediction suffix trees of depth at most d; in what follows d is assumed fixed. For a given tree $\mathcal{T} \in \mathbb{T}$, $\mathcal{T} := (S, \Psi)$, denote by $\rho_{\mathcal{T}} : \mathcal{X}^n \to [0, 1]$ the model described by this tree, with Ψ the set of KT estimators corresponding to the suffix set S. The Context Tree Weighting mixture $\xi : \mathcal{X}^n \to [0, 1]$ is defined as

$$\xi(x_{1:n}) := \sum_{\mathcal{T} \in \mathbb{T}} w_{\mathcal{T}} \rho_{\mathcal{T}}(x_{1:n}), \tag{6.2}$$

where $w_{\mathcal{T}} := 2^{-\Gamma_d(\mathcal{T})}$ is a prior weight on $\rho_{\mathcal{T}}$; $\Gamma_d(\mathcal{T})$ is the description length of $\mathcal{T}$ (Willems et al., 1995). For $\mathcal{T} := (S, \Psi)$ the particular form of this description length is

$$\Gamma_d(\mathcal{T}) := |S| - 1 + |\{s : s \in S, |s| \neq d\}|,$$

such that $\sum_{\mathcal{T} \in \mathbb{T}} w_{\mathcal{T}} = 1$. As d is assumed fixed, below I simply use $\Gamma(\mathcal{T})$ to denote the description length of $\mathcal{T}$ with respect to the set of all possible prediction suffix trees of depth at most d.

The right-hand side of Equation 6.2 is a sum over a doubly-exponential number of terms (there are $O(2^{2^d})$ prediction suffix trees of depth at most d). The key idea behind CTW is to express this equation recursively. Let $\mathrm{KT}(x^c_{1:n})$ denote the probability of the substring $x^c_{1:n}$ under the KT estimator. More explicitly,

$$\mathrm{KT}(x^c_{1:n}) := \prod_{i=1}^{n} \mathrm{KT}(x^c_i | x^c_{<i}),$$

where x^c_i denotes the i^{th} symbol in the subsequence $x^c_{1:n}$. The recursive Context Tree Weighting equation is

$$\mathrm{CTW}(x^c_{1:n}) := \begin{cases} \mathrm{KT}(x^c_{1:n}) & \text{if } |c| = d \\ \frac{1}{2}\mathrm{KT}(x^c_{1:n}) + \frac{1}{2}\mathrm{CTW}(x^{0c}_{1:n})\mathrm{CTW}(x^{1c}_{1:n}) & \text{otherwise} \end{cases} \tag{6.3}$$

with $\xi(x_{1:n}) = \mathrm{CTW}(x^\epsilon_{1:n})$.

Equation 6.3 enables us to process a string $x_{1:n}$ in time $O(dn)$; the algorithm's memory is similarly bounded by $O(dn)$. In practice, it is possible to optimize CTW in a variety of ways to improve its efficiency (Willems and Tjalkens, 1997).

Given a prediction suffix tree $\mathcal{T} := (S, \Psi)$, define the model $\rho_{\mathcal{T}}(x_{1:n})$ as

$$\rho_{\mathcal{T}}(x_{1:n}) := \prod_{s \in S} KT(x^s_{1:n}) \ .$$

Intuitively, $\rho_{\mathcal{T}}(x_{1:n})$ partitions $x_{1:n}$ according to its suffix set S, and uses a separate KT estimator for each resulting substring. The following theorem bounds the redundancy of Context Tree Weighting with respect to $\mathbb{T}$, the set of models using prediction suffix trees of depth at most d.

Theorem 6.1.1 (Willems et al. (1995))**.** *Let $\mathbb{T}$ denote the set of all prediction suffix trees of depth at most d with KT estimators at the leaves. Given a string $x_{1:n}$, the redundancy of Context Tree Weighting with respect to any model $\rho_{\mathcal{T}}(x_{1:n})$, $\mathcal{T} = (S, \Psi)$ with suffices $s \in S$ of length at most d and fixed parameters is given by*

$$-\log_2 \xi(x_{1:n}) - (-\log_2 \rho_{\mathcal{T}}(x_{1:n})) \leq \Gamma(\mathcal{T}) + |S| \mathfrak{z}\left(\frac{n}{|S|}\right),$$

where

$$\mathfrak{z}(x) := \begin{cases} x & \text{for } 0 \leq x < 1 \\ \frac{1}{2}\log_2 x + 1 & \text{for } x \geq 1 \end{cases}$$

Theorem 6.1.1 follows from the redundancy bound on individual KT estimators (Lemma 6.1.1) and the bound on mixture models (Equation 2.5). The first term in the bound, $\Gamma(\mathcal{T})$, is the cost associated with the unknown tree structure; the second term is the cost associated with learning the parameters at the leaves of the tree. Of note, Theorem 6.1.1 shows that the cost of learning the best tree structure is bounded by the complexity of the model, embodied in the description length $\Gamma(\mathcal{T})$.

Algorithm

The Context Tree Weighting stores its variables within a *context tree*: a perfect binary tree of depth d. At *every* node c in this context tree, we store the following:

1. A weighted probability $\mathrm{CTW}(x^c_{1:n}) \in (0, 1]$, and
2. A KT estimator KT_c.

In what follows, whenever unambiguous c refers to either a node of the context tree or its corresponding context string. Let $x_{1:n}$ be the string of symbols observed so far, and $x^c_{1:n}$ be the substring observed by the estimator KT_c. This estimator

contains counts $N_x(x^c_{1:n})$ for all $x \in \mathcal{X}$, as well as the probability of the substring $x^c_{1:n}$ observed by KT_c, $\mathrm{KT}(x^c_{1:n})$. When a node c is created, the counts of its estimator KT_c are initialized to $\frac{1}{2}$ and its weighted probability to 1.

When a new symbol x_t is observed, CTW recursively updates all $d+1$ nodes along the path of the current context $x_{t-d:t-1}$ according to Equation 6.3. The root of the tree corresponds to the empty context string ϵ.

To predict a new symbol, CTW predicts the symbol probability $\Pr(x_t = x \,|\, x_{<t})$ as

$$\Pr(x_t = x \,|\, x_{<t}) = \frac{\mathrm{CTW}(x_{<t}x)}{\mathrm{CTW}(x_{<t})}, \tag{6.4}$$

whose numerator is easily obtained by updating then reverting the context tree with x given an already observed string $x_{<t}$. Note that Equation 6.4 matches the symbol probability $\rho(x_n \,|\, (ax)_{<n}a_n)$ under an environment model in the AIXI setting (Equation 2.4). Alternatively, $\Pr(x_t = x \,|\, x_{<t})$ can be recursively computed without modifying the tree. Given a context string c, define the recursive symbol probability $P_c(x_t, x_{<t})$ as

$$P_c(x_t, x_{<t}) := \begin{cases} \mathrm{KT}(x \,|\, x_{<t}) & \text{if } |c| = d \\ \alpha_c \mathrm{KT}(x \,|\, x_{<t}) + (1-\alpha_c) P_{z(c,x_{<t})}(x_t, x_{<t}) & \text{otherwise} \end{cases} \tag{6.5}$$

where $z(c, x_{<t}) := x_{t-1-|c|}c$ is the extension of c according to $x_{<t}$ and

$$\alpha_c := \begin{cases} \frac{\frac{1}{2}\mathrm{KT}(x^c_{<t})}{\mathrm{CTW}(x^c_{<t})} & \text{if } |c| < d \\ 1 & \text{otherwise} \end{cases}$$

It then follows that $\Pr(x_t = x \,|\, x_{<t}) = P_\epsilon(x_t, x_{<t})$. Equation 6.5 can be interpreted as a mixture over $d+1$ KT estimators, each defined over nested substrings of $x_{<t}$; the posterior weight assigned to each estimator $KT(x^c_{<t})$ is the product $\alpha_c \prod_{s \in W(c)}(1-\alpha_s)$, where $W(c)$ is the set of strict suffices of c, including the empty suffix ϵ.

The Context Tree Weighting algorithm is summarized as Algorithm 1. As discussed above, its run time per step is a constant $O(d)$ and up to d nodes are added at every time step.

6.1.2 Context Tree Switching

Context Tree Switching (Veness et al., 2012) is a recent extension of Context Tree Weighting. As its name implies, CTS replaces the model weighting step in Equation

Algorithm 1 Context Tree Weighting

Require: A context tree T initialized with a root node ϵ
Initialize the root node ϵ
  for $t = 1 \dots n$ **do**
    Output probability distribution $\Pr(x_t \mid x_{<t})$ from CTW_ϵ
    Observe x_t
    RECURSIVE-UPDATE$(\epsilon, d, x_t, x_{1:t-1})$
  end for

recursive-update(node v, depth d, symbol x, context $x_{1:m}$)
  Update KT_v with x
  if $d = 0$ **then**
    return ; v is a leaf
  else
    if $\textsc{child}(v, x_m) = \textsc{null}$ **then**
      Create $\textsc{child}(v, x_m)$
      Set $\mathrm{CTW}_{\textsc{child}(v,x_m)} = 1$, initialize its KT estimator to $(\frac{1}{2}, \frac{1}{2})$
    end if
    RECURSIVE-UPDATE$(\textsc{child}(v, x_m), d-1, x, x_{1:m-1})$
    $\mathrm{CTW}_v \leftarrow \frac{1}{2}\mathrm{KT}_v + \frac{1}{2} \prod_{x' \in \{0,1\}} \mathrm{CTW}_{\textsc{child}(v,x')}$
  end if
  return

6.2 by a *switching* operation. The fundamental idea behind this operation is to incorporate a switch probability α to the mixture $\xi(x_{1:n})$. Using a switch probability allows the model to mix over all model sequences.

At every node c of its context tree, CTS stores the following:

1. A weighted probability $\mathrm{CTS}(x^c_{1:n}) \in (0, 1]$,
2. A KT estimator KT_c, and
3. The auxiliary quantities k_c and s_c.

As previously, denote by $\mathrm{KT}(x \mid x^c_{1:n})$ the probability of observing x according to KT_c and by $\mathrm{CTS}(x \mid x^c_{1:n})$ the probability of observing x according to the distribution over x implied by $\mathrm{CTS}(x^c_{1:n})$; as with CTW (i.e. Equation 6.4), both of these quantities can respectively be computed directly or from the weighted probabilities $\mathrm{KT}(x^c_{1:n})$ and $\mathrm{CTS}(x^c_{1:n})$. Context Tree Switching replaces the weighting step (updating $\mathrm{CTW}(x^c_{1:n})$) in Algorithm 1 with the following updates:

$$
\begin{aligned}
\mathrm{CTS}(x^c_{1:n}) &\leftarrow \mathrm{KT}(x^c_{1:n}) \qquad \text{if } |c| = d, \text{ otherwise:} \\
\mathrm{CTS}(x^c_{1:n}) &\leftarrow k_{c,n-1}\mathrm{KT}(x_n \,|\, x^c_{<n}) + s_{c,n-1}\mathrm{CTS}(x_n \,|\, x^{z(c,x_{<n})}_{<n}) \\
k_{c,n} &\leftarrow (1-\alpha_{n+1})k_{c,n-1}\mathrm{KT}(x_n \,|\, x^c_{<n}) + \alpha_{n+1}s_{c,n-1}\mathrm{CTS}(x_n \,|\, x^{z(c,x_{<n})}_{<n}) \\
s_{c,n} &\leftarrow (1-\alpha_{n+1})s_{c,n-1}\mathrm{CTS}(x_n \,|\, x^{z(c,x_{<n})}_{<n}) + \alpha_{n+1}k_{c,n-1}\mathrm{KT}(x^c_{1:n}),
\end{aligned}
$$

where $z(c, x_{<n}) := x_{n-|c|-1}c$ is the extension of c according to $x_{<n}$, and $\alpha_t := \frac{1}{t}$. The purpose of the time indices to $k_{c,n}$ and $s_{c,n}$ is algorithmic clarity: only the most current values ($k_{c,n}$ and $s_{c,n}$) ever need to be stored, and initially $k_{c,0} = s_{c,0} = 0.5$. While each CTW node constructs a mixture model from its KT estimator and the mixture models of its children by weighing the two according to their block probabilities over the whole data, a CTS node instead puts more weight on recent predictions through the switching term α_{n+1}. Let $\xi_{\mathrm{CTS}}(x_{1:n}) := \mathrm{CTS}(x^\epsilon_{1:n})$. The following Theorem gives a bound on the redundancy of Context Tree Switching:

Theorem 6.1.2 (Veness et al. (2012))**.** *Given a string $x_{1:n}$, the redundancy of Context Tree Switching run with parameter depth d with respect to any model $\rho_{\mathcal{T}}(x_{1:n})$, $\mathcal{T} = (S, \Psi)$ with suffices $s \in S$ of length at most d and fixed parameters is given by*

$$
-\log_2 \xi_{\mathrm{CTS}}(x_{1:n}) - (-\log_2 \rho_{\mathcal{T}}(x_{1:n})) \leq \Gamma(\mathcal{T}) + |S|\mathfrak{z}\left(\frac{n}{|S|}\right) + (d+1)\log_2 n,
$$

where

$$
\mathfrak{z}(x) := \begin{cases} x & \text{for } 0 \leq x < 1 \\ \frac{1}{2}\log_2 x + 1 & \text{for } x \geq 1 \end{cases}
$$

Theorem 6.1.2 guarantees that the cost of switching is at most $(d+1)\log_2 n$ compared to weighting. However, while CTW performs well with respect to any fixed model in $\mathbb{T}$, CTS performs well with respect to the set of all possible *sequences* of models in $\mathbb{T}$; in practice, CTS has been shown to outperform CTW (Veness et al., 2012).

6.1.3 The Sparse Sequential Dirichlet Estimator

Although Section 6.1.1 described prediction suffix trees and Context Tree Weighting using the KT estimator, any estimator suitable for memoryless and stationary

sources may be used. In particular, one limitation of the KT estimator as defined above is its restriction to a binary alphabet. Although the KT estimator can be extended to a finite set of symbols using a Dirichlet prior, the redundancy bound of Lemma 6.1.1 then becomes linear in $|\mathcal{X}|$ (Tjalkens et al., 1993b).

It is not uncommon to be faced with a large alphabet $\mathcal{X}$ from which only a small subset $\mathcal{Y} \subset \mathcal{X}$ is used. For example, although we may want to predict an English text using the set of all valid combinations of letters from the Latin alphabet, only a comparatively small subset of words constitutes valid English language. The *Sparse Sequential Dirichlet estimator* (SSD, Veness and Hutter (2012)) is an estimator for stationary, memoryless sources whose redundancy bound depends only logarithmically on $|\mathcal{X}|$.

Let $\mathrm{SSD}(x_n \,|\, x_{<n})$ denote the probability assigned to x_n by an SSD estimator that has observed $x_{<n}$. This probability is defined as

$$\mathrm{SSD}(x_n \,|\, x_{<n}) := \begin{cases} \alpha_n \frac{1}{|\mathcal{X}| - |U(x_{<n})|} & \text{if } c(x_n, x_{<n}) = 0 \\ (1 - \alpha_n) \frac{N_{x_n}(x_{<n}) + \frac{1}{2}}{n + \frac{1}{2}|U(x_{<n})| - 1} & \text{otherwise} \end{cases},$$

where $N_x(x_{<n}) := \sum_{i=1}^{n-1} \mathbb{I}_{[x_i = x]}$ is the number of times x occurs in $x_{<n}$, $U(x_{<n}) := \{x : x \in \mathcal{X}, N_x(x_{<n}) > 0\}$ and $\alpha_n := \frac{1}{n}$. Effectively, the SSD estimator assigns a uniform probability to all unseen symbols, and estimates the probability of seen symbols using a KT estimator over $U(x_{<n})$; the two probabilities (unseen and seen symbols) are combined by means of the decaying α_n term. The following Theorem bounds the redundancy of the SSD estimator:

Theorem 6.1.3 (Veness and Hutter (2012))**.** *Given alphabets $\mathcal{X}$ and $\mathcal{Y}$ such that $\mathcal{Y} \subseteq \mathcal{X}$, for all $n \in \mathbb{N}$, for all $x_{1:n} \in \mathcal{Y}$ we have*

$$-\log_2 \mathrm{SSD}(x_{1:n}) - (-\log_2 \mathrm{KT}^{\mathcal{Y}}(x_{1:n})) \leq \log_2 n + |\mathcal{Y}| \log_2 |\mathcal{X}|,$$

where $\mathrm{KT}^{\mathcal{Y}}(x_{1:n})$ denotes the probability assigned to $x_{1:n}$ by the KT estimator using the alphabet $\mathcal{Y}$.

While other approaches have been put forward for dealing with sparse alphabets (Friedman and Singer, 1999; Tjalkens et al., 1993a), the SSD estimator has the advantage of being both algorithmically elegant and computationally efficient.

6.1.4 Context Trees as AIXI Models

Context Tree Weighting has been cast into the AIXI setting by Veness et al. (2011), resulting in an algorithm called Action-Conditional CTW. In Action-Conditional CTW, actions are treated as *side information* available to Context Tree Weighting (or Switching). This is done by appending an encoding of the action history $a_{1:n}$ (or part thereof) to the context string $x_{n-d:n-1}$. It is then possible to derive a bound on how well the true environment is modelled:

Lemma 6.1.2 (Simplified from Veness et al. (2011))**.** *Let μ be an environment expressible as a prediction suffix tree with binary KT estimators at the leaves, $\mathcal{T} := (S, \Psi)$, of depth at most d. Then for the Action-Conditional Context Tree Weighting ξ_{CTW} run with depth parameter $d' \geq d$, we have*

$$\mathbb{E}_\mu \left[\frac{\mu(x_{1:n} \,|\, a_{1:n})}{\xi_{CTW}(x_{1:n} \,|\, a_{1:n})} \right] \leq \Gamma(\mathcal{T}) + |S| \mathfrak{z} \left(\frac{n}{|S|} \right),$$

where $\mathfrak{z}(x)$ is defined as in Theorems 6.1.1 and 6.1.2.

In short, Lemma 6.1.2 states that k-Markov environments can be modelled by the Context Tree Weighting algorithm adapted to the AIXI setting. The given bound arises as a sum of different costs: the cost of learning the tree structure, and the individual parameters at each node. It thus easily extends to environments expressible with multi-alphabet KT estimators, and from there to environment models using SSD or other memoryless estimators.

6.2 Factored Environment Models

While Action-Conditional CTW is a powerful tool for learning environment models, its treatment of percepts as atomic entities precludes its direct application to the Atari 2600. To see this, recall that a single game screen is composed of 160×210 7-bit pixels: thus the Atari 2600 observation space $\mathcal{O}$ contains $2^{7 \times 160 \times 210}$ different screens. The first step in developing a CTW-like algorithm for the Atari 2600 is to define the notion of a *factored environment model*: a model whose observation space can be decomposed into a Cartesian product of factors.

Consider a percept space $\mathcal{X} := \mathcal{X}_1 \times \cdots \times \mathcal{X}_k$, the Cartesian product of $k \in \mathbb{N}$ subspaces. First, let $\mathcal{X}_{<i} := \mathcal{X}_1 \times \cdots \times \mathcal{X}_{i-1}$ for $1 \leq i \leq k$. Given a string $x_{1:n} \in (\mathcal{X}_1 \times \cdots \times \mathcal{X}_k)^n$, denote by $x_t^i \in \mathcal{X}_i$ or $x_t^{\mathcal{X}_i} \in \mathcal{X}_i$ the i^{th} component of $x_t \in \mathcal{X}$, with

$1 \leq i \leq k$ and $1 \leq t \leq n$. Further denote by $\mathcal{X}_i^n := (\mathcal{X}_i)^n$ and $\mathcal{X}_{<i}^n := (\mathcal{X}_{<i})^n$ the n-dimensional string subspaces, with $\mathcal{X}_i^n \ni x_{1:n}^i := x_1^i x_2^i \dots x_n^i$.

Now, given an action space $\mathcal{A}$ and a factored percept space $\mathcal{X} := \mathcal{X}_1 \times \cdots \times \mathcal{X}_k$, a k-factored environment is defined by a tuple $(\rho^1, \dots, \rho^k)$, where each component of the tuple is an environment model factor

$$\rho^i := \left\{ \rho_n^i : (\mathcal{A} \times \mathcal{X})^{n-1} \times \mathcal{A} \times \mathcal{X}_{<i} \to \textsc{Dist}(\mathcal{X}_i) \right\}_{n \in \mathbb{N}}$$

for $1 \leq i \leq k$, with each ρ_n^i defining a parametrized probability mass function. Using the chain rule, this naturally induces a factored environment given by

$$\begin{aligned} \rho(x_{1:n} \,|\, a_{1:n}) &= \prod_{t=1}^{n} \rho(x_t \,|\, a x_{<t}) \\ &= \prod_{t=1}^{n} \prod_{i=1}^{k} \rho_t^i(x_t^i \,|\, a x_{<t} a_t x_t^{<i}) \\ &= \prod_{i=1}^{k} \rho^i(x_{1:n}^i \,|\, a_{1:n}), \end{aligned}$$

where the final line uses the notation

$$\rho^i(x_{1:n}^i \,|\, a_{1:n}) := \prod_{t=1}^{n} \rho_t^i(x_t^i \,|\, a x_{<t} a_t x_t^{<i}).$$

One can easily verify that a k-factored environment satisfies the chronological condition (Section 2.2), and is therefore a valid environment.

6.3 Recursive Factorizations

The goal of this chapter is to develop an algorithm that performs Bayesian model averaging over a large class of factored environment models. To do so, I now introduce an efficiently computable class of structured factorizations. As noted in the introduction, this section focuses on a domain-independent presentation and defers to Section 6.4 a more concrete instantiation tailored to the Atari 2600 platform. Although the presentation focuses on factoring the observation space, the reward can be modelled by using an additional environment model factor.

Definition 6.3.1. *A recursively decomposable space of nesting depth $d = 0$ is a set. When $d \in \mathbb{N}$, a recursively decomposable space is the set formed from the Cartesian product of two or more recursively decomposable spaces of nesting depth $d - 1$. The*

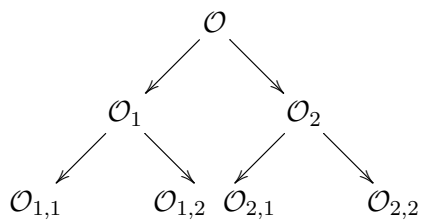

Figure 6.2: A recursively decomposable space.

number of factors in the Cartesian product defining a recursively decomposable space $\mathcal{F}$ *at a particular nesting depth is denoted by* $\dim(\mathcal{F})$*, and is defined to be 1 if the nesting depth is* 0*.*

Figure 6.2 shows an instance of a recursively decomposable space $\mathcal{O}$ of nesting depth 2, naturally represented as a tree. The leaves of the tree correspond to sets $\mathcal{O}_{1,1}$, $\mathcal{O}_{1,2}$, $\mathcal{O}_{2,1}$, and $\mathcal{O}_{2,2}$. Such sets may, for example, describe individual sensors on a robot. In what follows, the notation $\mathcal{O}_k$ denotes the k^{th} factor of the Cartesian product defining the recursively decomposable space $\mathcal{O}$.

A recursively decomposable space can clearly be factored in many possible ways. The following definition describes the set of all possible factorizations of a recursively decomposable space:

Definition 6.3.2. *Given a recursively decomposable space* $\mathcal{F}$ *with nesting depth at least* $d \in \mathbb{N}$*, the set* $\mathcal{C}_d(\mathcal{F})$ *of all recursive factorizations of* $\mathcal{F}$ *is defined by*

$$\mathcal{C}_d(\mathcal{F}) := \left\{\mathcal{F}\right\} \cup \left\{ \bigtimes_{i=1}^{\dim(\mathcal{F})} \mathcal{S}_i : \mathcal{S}_i \in \mathcal{C}_{d-1}\left(\mathcal{F}_i\right) \right\}, \tag{6.6}$$

with $\mathcal{C}_0(\mathcal{F}) := \left\{\mathcal{F}\right\}$*.*

Returning to the example of Figure 6.2, this gives the following set of factorizations:

$$\begin{aligned} \mathcal{C}_2(\mathcal{O}) \;:=\; & \Big\{\mathcal{O}, \\ & \mathcal{O}_1 \times \mathcal{O}_2, \\ & \mathcal{O}_{1,1} \times \mathcal{O}_{1,2} \times \mathcal{O}_2, \\ & \mathcal{O}_1 \times \mathcal{O}_{2,1} \times \mathcal{O}_{2,2}, \\ & \mathcal{O}_{1,1} \times \mathcal{O}_{1,2} \times \mathcal{O}_{2,1} \times \mathcal{O}_{2,2}\Big\} \end{aligned}$$

Notice that although the number of factorizations for the above example is small, in general the number of possible factorizations grows super-exponentially in the nesting depth. In particular, if $\mathcal{F}$ can be described as a binary tree of depth d, then the number of possible factorizations is $O(2^{2^d})$; this corresponds to the number of prediction suffix trees of depth up to d (Veness et al., 2012).

6.3.1 A Prior over Recursive Factorizations

The next step is to describe a prior on $\mathcal{C}_d(\mathcal{F})$. The aim is to design a prior similar to the Context Tree Weighting prior: it should both support efficient computation and be biased towards simpler (more compact) factorizations.

From the recursive construction of $\mathcal{C}_d(\mathcal{F})$ in Equation 6.6 stem two cases that need to be considered: either we stop decomposing the space $\mathcal{F}$ (corresponding to the $\{\mathcal{F}\}$ term in Equation 6.6) or we continue to split it further. This observation naturally suggests the use of a hierarchical prior, which recursively subdivides the remaining prior weight amongst each of the two possible choices. If we use a uniform weighting for each possibility, this gives a prior weighting of $2^{-\Gamma_d(f)}$, where $\Gamma_d(f)$ returns the total number of stop/split decisions needed to describe the factorization $f \in \mathcal{C}_d(\mathcal{F})$, with the base case of $\Gamma_0(f) := 0$ (since when $d = 0$, no stop or split decision needs to be made). As desired, this prior weighting is identical to the prior weighting of Context Tree Weighting over tree structures. Furthermore, it constitutes a valid prior, as one can show $\sum_{f \in \mathcal{C}_d} 2^{-\Gamma_d(f)} = 1$ for all $d \in \mathbb{N}$.

One side-effect of this recursive construction is that it assigns more prior weight towards factorizations containing smaller amounts of nested substructure. For instance, in the example above $2^{-\Gamma_2(\mathcal{O})} = \frac{1}{2}$, while $2^{-\Gamma_2(\mathcal{O}_1 \times \mathcal{O}_{2,1} \times \mathcal{O}_{2,2})} = \frac{1}{8}$. Such a prior is appealing when the computational complexity of a factorization is proportional to its size, as is the case for the quad-tree factorization of Section 6.4. As with CTW, the prior also implicitly contains structure that makes model averaging easier, as the next section details.

6.3.2 Recursively Factored Mixture Environment Models

After defining a prior over factorizations, I now combine it with the model averaging technique described in Section 2.2 to define the class of recursively factored mixture environment models.

The first step is to describe the set of base environment model factors from which

each factored environment is formed. This requires specifying an environment model factor for each element of the set of possible stopping points

$$\mathcal{S}_d(\mathcal{F}) := \{\mathcal{F}\} \cup \bigcup_{i=1}^{\dim(\mathcal{F})} \mathcal{S}_{d-1}(\mathcal{F}_i) \qquad \text{for } d > 0,$$

with $\mathcal{S}_0(\mathcal{F}) := \{\mathcal{F}\}$, within a recursively decomposable space $\mathcal{F}$ of nesting depth d. In the previous example, $\mathcal{S}_2(\mathcal{O}) = \{\mathcal{O}, \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_{1,1}, \mathcal{O}_{1,2}, \mathcal{O}_{2,1}, \mathcal{O}_{2,2}\}$; to apply the factorization $\{\mathcal{O}_1 \times \mathcal{O}_{1,2} \times \mathcal{O}_{2,1}\}$ we need a model with base environment model factors $\mathfrak{f}_{\mathcal{O}_1}$, $\mathfrak{f}_{\mathcal{O}_{1,2}}$ and $\mathfrak{f}_{\mathcal{O}_{2,1}}$, where $\mathfrak{f}_s$ denotes the base environment model factor corresponding to the stopping point $s \in \mathcal{S}_d(\mathcal{F})$. Effectively, each of these factors operates on a separate alphabet ($\mathcal{O}_1$, $\mathcal{O}_{1,2}$, $\mathcal{O}_{2,1}$). More precisely, each environment model factor needs an appropriate type signature that depends on both the history and the parts of the percept space preceding it. For instance, an environment model factor processing $\mathcal{O}_2$ at time n depends on a history string that is an element of the set $(\mathcal{A} \times \mathcal{O})^{n-1} \times \mathcal{A} \times \mathcal{O}_1$. Similarly, an environment model factor for $\mathcal{O}_{2,2}$ depends on a history string from the set $(\mathcal{A} \times \mathcal{O})^{n-1} \times \mathcal{A} \times \mathcal{O}_1 \times \mathcal{O}_{2,1}$. Now, given a history $ax_{1:n}$ and a $d \geq 0$ times recursively decomposable space $\mathcal{F}$, a recursively factored mixture environment is defined as

$$\xi^d_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n}) := \sum_{f \in \mathcal{C}_d(\mathcal{F})} 2^{-\Gamma_d(f)} \rho_f(x_{1:n} \,|\, a_{1:n}), \tag{6.7}$$

where each factored model is defined by a product of environment model factors

$$\rho_f(x_{1:n} \,|\, a_{1:n}) := \prod_{\tau \in U(f)} \mathfrak{f}_\tau(x^\tau_{1:n} \,|\, a_{1:n}),$$

with $U(f) \subset S_d(\mathcal{F})$ the set of stopping points in the factorization $f \in \mathcal{C}_d(\mathcal{F})$. Note that $\xi^0_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n}) = \mathfrak{f}_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n})$.

Notice that there exists a significant amount of shared structure in Equation 6.7, since each environment model factor can appear multiple times in the definition of each factored environment model. This property, along with the recursive definition of the prior in Section 6.3.1, allows us to derive the identity

$$\xi^d_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n}) = \frac{1}{2}\mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n}) + \frac{1}{2} \prod_{i=1}^{\dim(\mathcal{F})} \xi^{d-1}_{\mathcal{F}_i}(x_{1:n} \,|\, a_{1:n}), \tag{6.8}$$

that allows us to compute Equation 6.7 efficiently.

Proof. Along the lines of the argument used to prove Lemma 2 in Willems et al. (1995), we write

$$\begin{aligned}
\xi^d_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n}) &= \sum_{f \in \mathcal{C}_d(\mathcal{F})} 2^{-\Gamma_d(f)} \rho_f(x_{1:n} \,|\, a_{1:n}) \\
&= \sum_{f \in \mathcal{C}_d(\mathcal{F})} 2^{-\Gamma_d(f)} \prod_{\tau \in U(f)} \mathfrak{f}_\tau(x^\tau_{1:n} \,|\, a_{1:n}) \\
&= \frac{1}{2} \mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n}) + \sum_{f \in \mathcal{C}_d \setminus \{\mathcal{F}\}} 2^{-\Gamma_d(f)} \prod_{\tau \in U(f)} \mathfrak{f}_\tau(x^\tau_{1:n} \,|\, a_{1:n}) \\
&= \frac{1}{2} \mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n}) + \\
&\quad \frac{1}{2} \sum_{\substack{f_1 \in \\ \mathcal{C}_{d-1}(\mathcal{F}_1)}} \cdots \sum_{\substack{f_{\dim(\mathcal{F})} \in \\ \mathcal{C}_{d-1}(\mathcal{F}_{\dim(\mathcal{F})})}} \prod_{i=1}^{\dim(\mathcal{F})} 2^{-\Gamma_{d-1}(f_i)} \prod_{\tau \in U(f_i)} \mathfrak{f}_\tau(x^\tau_{1:n} \,|\, a_{1:n}) \\
&= \frac{1}{2} \mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n}) + \\
&\quad \frac{1}{2} \prod_{i=1}^{\dim(\mathcal{F})} \left(\sum_{f \in \mathcal{C}_{d-1}(\mathcal{F}_i)} 2^{-\Gamma_{d-1}(f)} \rho_f(x_{1:n} \,|\, a_{1:n}) \right) \\
&= \frac{1}{2} \mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n}) + \frac{1}{2} \prod_{i=1}^{\dim(\mathcal{F})} \xi^{d-1}_{\mathcal{F}_i}(x_{1:n} \,|\, a_{1:n}).
\end{aligned}$$

□

Equation 6.8 is essentially an application of the Generalized Distributive Law (Aji and McEliece, 2000), a key computational concept underlying many efficient algorithms. By using dynamic programming to compute each $\xi^d_{\mathcal{F}}$ term only once, the time overhead of performing exact model averaging over $\mathcal{C}_d(\mathcal{F})$ is reduced to just $O(n|\mathcal{S}_d(\mathcal{F})|)$. Furthermore, provided each base environment model factor can be updated online and the $\{\xi^d_{\mathcal{F}'}\}_{\mathcal{F}' \in \mathcal{S}_d(\mathcal{F})}$ terms are kept in memory, each percept can be processed online in time $O(|\mathcal{S}_d(\mathcal{F})|)$. The size of $\mathcal{S}_d(\mathcal{F})$ is in general dominated by the number m of spaces of depth 0, yielding a method that is effectively linear in m. For example, if $\mathcal{F}$ can be represented by a binary tree of depth d, there are $m = 2^d$ spaces of depth 0, $|\mathcal{S}_d(\mathcal{F})| = 2^{d+1} - 1$ and the algorithm described here performs model averaging over $O(2^{2^d})$ factorizations.

As the technique performs exact model averaging, the bound of Equation 2.5 applies:

Theorem 6.3.1. *Given a recursively decomposable space $\mathcal{F}$ with nesting depth $d \in$*

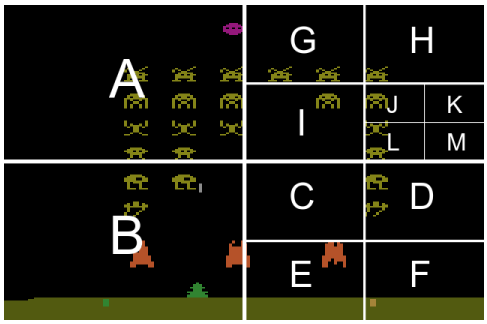

Figure 6.3: Example quad-tree factorization of a Space Invaders screen. Each labelled square corresponds to a factor.

$\mathbb{N}$, *for all* $a_{1:n} \in \mathcal{A}^n$, $x_{1:n} \in \mathcal{X}^n$, *and* $f \in \mathcal{C}_d(\mathcal{F})$, *we have*

$$-\log_2 \xi^d_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n}) - (-\log_2 \rho_f(x_{1:n} \,|\, a_{1:n})) \leq \Gamma_d(f) \ ,$$

and for any environment μ,

$$D_{1:n}(\mu \,\|\, \xi^d_{\mathcal{F}}) \leq \Gamma_d(f) + D_{1:n}(\mu \,\|\, \rho_f).$$

Hence the technique described above is asymptotically competitive with the best factorization in $\mathcal{C}_d(\mathcal{F})$.

6.4 Quad-Tree Factorization

Section 6.3 provides a general framework for specifying recursively factored mixture environment models. The aim of this section is to give an example of how this framework can extend existing model learning algorithms to domains with large observation spaces, such as Atari 2600 games. The quad-tree factorization (QTF) technique I now describe is particularly suited to image-based observation spaces. Although QTF is presented in Atari 2600 terms, it is easily extended to other domains whose observation space exhibits two-dimensional structure.

Recall the notation from Section 3.3: $\mathcal{D}_x$ and $\mathcal{D}_y$ denote totally ordered sets of row and column indices, with the joint index space given by $\mathcal{D} := \mathcal{D}_x \times \mathcal{D}_y$; $\mathcal{C}$ denotes a finite set of possible pixel colours; a pixel is a tuple $(x, y, c) \in \mathcal{D}_x \times \mathcal{D}_y \times \mathcal{C}$; finally, an observation o is defined as a set of $|\mathcal{D}|$ pixels, with each location $(x, y) \in \mathcal{D}$ uniquely corresponding to a single pixel (x, y, c). The set of all possible observations is denoted by $\mathcal{O}$.

I now describe a natural way to recursively decompose $\mathcal{O}$ using a quad-tree split operator that divides each region into four equal parts; an example of such a decomposition is shown in Figure 6.3. For a given $\mathcal{D}'_x \subseteq \mathcal{D}_x$ whose ordered elements are $x_1, x_2, x_3, \ldots, x_n$, denote by $\mathrm{l}(\mathcal{D}'_x) := \{x_1, x_2, \ldots, x_{\lfloor n/2 \rfloor}\}$ and by $\mathrm{h}(\mathcal{D}'_x) := \{x_{\lfloor n/2 \rfloor+1}, x_{\lfloor n/2 \rfloor+2}, \ldots, x_n\}$ the lower and upper halves of $\mathcal{D}'_x$; similarly let $\mathrm{l}(\mathcal{D}'_y)$ and $\mathrm{h}(\mathcal{D}'_y)$ denote the two halves of $\mathcal{D}'_y \subseteq \mathcal{D}_y$. Let

$$\mathcal{O}_{\mathcal{D}'_x,\mathcal{D}'_y} := \big\{\big\{(x, y, c) : x \in \mathcal{D}'_x, y \in \mathcal{D}'_y, (x, y, c) \in o\big\} : o \in \mathcal{O}\big\}$$

be the set of image patches that can occur in the region defined by $\mathcal{D}'_x$ and $\mathcal{D}'_y$, noting that $\mathcal{O} = \mathcal{O}_{\mathcal{D}_x,\mathcal{D}_y}$. Assuming for now that $|\mathcal{D}_x|$ and $|\mathcal{D}_y|$ are both divisible by 2^d, the recursively decomposable space $\mathcal{F}^d_{\mathcal{D}_x,\mathcal{D}_y}$ on $\mathcal{O}$ is then recursively defined as

$$\mathcal{F}^d_{\mathcal{D}'_x,\mathcal{D}'_y} := \begin{cases} O_{\mathcal{D}'_x,\mathcal{D}'_y} & \text{if } d = 0 \\ \mathcal{F}^{d-1}_{\mathrm{l}(\mathcal{D}'_x),\mathrm{l}(\mathcal{D}'_y)} \times \mathcal{F}^{d-1}_{\mathrm{l}(\mathcal{D}'_x),\mathrm{h}(\mathcal{D}'_y)} \times & \\ \quad \mathcal{F}^{d-1}_{\mathrm{h}(\mathcal{D}'_x),\mathrm{l}(\mathcal{D}'_y)} \times \mathcal{F}^{d-1}_{\mathrm{h}(\mathcal{D}'_x),\mathrm{h}(\mathcal{D}'_y)} & \text{otherwise} \end{cases} \tag{6.9}$$

The base case corresponds to indivisible image patches of size $|\mathcal{D}_x|/2^d \times |\mathcal{D}_y|/2^d$. These image patches are used to form larger image patches. Also note that there is a one-to-one correspondence between elements of $\mathcal{F}^d_{\mathcal{D}_x,\mathcal{D}_y}$ and $\mathcal{O}$. Whenever $|\mathcal{D}_x|$ or $|\mathcal{D}_y|$ is not divisible by 2^d, as is the case with Atari 2600 games, a simple solution is to enlarge $\mathcal{D}_x$ and $\mathcal{D}_y$ appropriately and insert pixels whose colour is a special *out-of-screen* colour, as was also done in Section 4.2.2[2].

6.4.1 Algorithm

Algorithm 2 provides pseudocode for an online implementation of QTF. The algorithm is invoked once per time step; its arguments are the top-level space $\mathcal{F}^d_{\mathcal{D}_x,\mathcal{D}_y}$, its nesting depth d, and the current percept. The algorithm recursively updates the base environment model factor $\mathfrak{f}_{\mathcal{F}}$ corresponding to $\mathcal{F}$ as well as its factors $\{\mathcal{F}_i\}_{i=1}^4$. The $\mathfrak{f}_{\mathcal{F}}$ and $\xi^d_{\mathcal{F}}$ variables respectively store the values $\mathfrak{f}_{\mathcal{F}}(x^{\mathcal{F}}_{1:n} \,|\, a_{1:n})$ and $\xi^d_{\mathcal{F}}(x_{1:n} \,|\, a_{1:n})$; both sets of variables are initialized to 1. As QTF is a meta-algorithm, any suitable set of base environment model factors may be used. In my implementation, I avoid numerical issues such as underflow by storing and manipulating probabilities in the logarithmic domain.

[2]Having each dimension divisible by 2^d is not strictly required, but it considerably simplifies the implementation details.

If C represents the (uniform) cost of updating a single base environment model factor and performing the addition and subtraction necessary to update the term $\xi_{\mathcal{F}}^{d}$, then the computational cost of running QTF is at most $\frac{4}{3}|\mathcal{D}|C$, which is easily derived by summing the geometric series.

Algorithm 2 Online Quad-Tree Factorization

Require: A quad-tree decomposable space $\mathcal{F}$
Require: A nesting depth $d \in \mathbb{N}$
Require: A percept x_t at time $t \in \mathbb{N}$

$\textsc{qtf}(\mathcal{F}, d, x_t)$
  Update $\mathfrak{f}_{\mathcal{F}}$ with $x_t^{\mathcal{F}}$
  if $d > 0$ **then**
    for $i = 1 \ldots 4$ **do**
      $\textsc{qtf}(\mathcal{F}_i, d-1, x_t)$
    end for
    $\xi_{\mathcal{F}}^{d} \leftarrow \frac{1}{2}\mathfrak{f}_{\mathcal{F}} + \frac{1}{2}\prod_{i=1}^{4} \xi_{\mathcal{F}_i}^{d-1}$
  else
    $\xi_{\mathcal{F}}^{d} \leftarrow \mathfrak{f}_{\mathcal{F}}$
  end if

6.4.2 Switching Quad-Tree Factorization

In the same manner that the weighting operation of Context Tree Weighting can be replaced by a switching operation (Section 6.1.2), the weighting operation in QTF can also be replaced by a switching operation. In practice, this leads to faster learning: the algorithm quickly switches from using small, more easily-learned image patches to larger patches which take more samples to model but are more accurate.

The switching quad-tree factorization algorithm is described in Algorithm 3. The only difference between this new algorithm and QTF is the need to store, at each QTF node, two quantities $k_{\mathcal{F}}$ and $s_{\mathcal{F}}$; both are initialized to 0.5.

6.5 Empirical Study

As with the other contributions of this thesis, I evaluated the quad-tree factorization using the Arcade Learning Environment (Section 3.1.1); the algorithm was first designed and optimized using the training games and subsequently evaluated on fifteen testing games; these games were selected uniformly at random from the larger set of testing games. The games used in this chapter are listed in Appendix

Algorithm 3 Online Switching Quad-Tree Factorization

Require: A quad-tree decomposable space $\mathcal{F}$
Require: A nesting depth $d \in \mathbb{N}$
Require: A percept x_t at time $t \in \mathbb{N}$

$\textsc{sqtf}(\mathcal{F}, d, x_t)$
  $z \leftarrow \xi_{\mathcal{F}}^d$
  $p_{\mathfrak{f}} \leftarrow$ the probability of $x_t^{\mathcal{F}}$ according to $\mathfrak{f}_{\mathcal{F}}$
  Update $\mathfrak{f}_{\mathcal{F}}$ with $x_t^{\mathcal{F}}$
  if $d > 0$ **then**
    for $i = 1 \ldots 4$ **do**
      $q_i \leftarrow \textsc{sqtf}(\mathcal{F}_i, d-1, x_t)$
    end for
    $\alpha_{t+1} \leftarrow \frac{1}{t+1}$
    $\xi_{\mathcal{F}}^d \leftarrow k_{\mathcal{F}}\, p_{\mathfrak{f}} + s_{\mathcal{F}} \prod_{i=1}^{4} q_i$
    $\left.\begin{aligned} k_{\mathcal{F}} &\leftarrow (1-\alpha_{t+1}) k_{\mathcal{F}}\, p_{\mathfrak{f}} + \alpha_{t+1} s_{\mathcal{F}} \prod_{i=1}^{4} q_i \\ s_{\mathcal{F}} &\leftarrow (1-\alpha_{t+1}) s_{\mathcal{F}} \prod_{i=1}^{4} q_i + \alpha_{t+1} k_{\mathcal{F}}\, p_{\mathfrak{f}} \end{aligned}\right\}$ simultaneously
  else
    $\xi_{\mathcal{F}}^d \leftarrow \mathfrak{f}_{\mathcal{F}}$
  end if
  return $\xi_{\mathcal{F}}^d / z$; the probability of $x_t^{\mathcal{F}}$ under the switching model $\xi_{\mathcal{F}}^d$

B.

6.5.1 Experimental Setup

For practical reasons, I used a quad-tree factorization that considered image patches ranging in size from 32×32 down to 4×4; this corresponds to a nesting depth of 3. Empirically, I found that larger patch sizes generalized poorly, while smaller patches performed worse due to limited contextual information. Each patch was predicted using a context tree switching (CTS) model (Section 6.1.2); to handle the large image patch alphabets, Sparse Sequential Dirichlet estimators (Section 6.1.3) were used at the CTS nodes. Each prediction was made using a *patch context* composed of eleven neighbouring patches from the current and previous time steps, similar in spirit to the $\mathcal{P}$-context trees of Veness et al. (2011). This patch context also encoded the most recent action. The exact features used are provided in Appendix A.

To curb memory usage and improve sample efficiency, I incorporated a set of well-established techniques into my implementation. For each patch size, a single

CTS model was shared across patch locations, giving a limited form of translation invariance. Finally, each CTS model was implemented using hashing (Willems and Tjalkens, 1997) so as to better control memory usage. In practice, I found that larger hash tables always resulted in better results; in the results below the hash table used 1 million entries per tree depth, for a total of 12 million entries.

I compared QTF with factored models using a fixed patch size (4×4, 8×8, 16×16, 32×32). Each model was trained on each game for 10 million frames, using a policy that selected actions uniformly at random and then executed them for k frames, where k was also chosen uniformly at random from the set $K := \{4, 8, 12, 16\}$. This policy was designed to visit more interesting parts of the state space and thus generate more complex trajectories. The models received a new game screen every fourth frame, yielding 15 time steps per second of play. Predicting at higher frame rates was found to be easier, but led to qualitatively similar results while requiring more wall-clock time per experiment.

	Time Steps Correct				
	4×4	8×8	16×16	32×32	QTF
Asterix	0.748	1.17	1.11	**1.44**	**1.44**
Beam Rider	≈ 0	0.007	0.100	**0.227**	0.185
Freeway	0.303	**4.64**	2.21	1.63	3.20
Seaquest	0.019	0.427	0.157	**2.39**	1.85
Space Invaders	0.606	0.502	0.736	0.274	**0.830**
Amidar	13.0	12.9	13.1	13.3	**13.4**
Crazy Climber	0.347	0.758	0.650	2.50	**2.66**
Demon Attack	0.004	0.004	0.006	0.013	**0.026**
Gopher	0.048	0.375	0.747	**2.27**	**2.28**
Krull	0.014	0.083	0.482	**1.14**	0.233
Kung-Fu Master	2.87	3.17	3.35	**3.57**	3.53
Ms. Pacman	0.047	0.096	0.264	0.434	**0.463**
Pong	0.319	1.79	2.50	**3.99**	3.97
Private Eye	0.000	≈ 0	0.001	**0.010**	0.003
River Raid	0.513	0.643	0.672	**1.76**	1.68
Star Gunner	0.041	0.417	0.659	**1.48**	0.802
Tennis	3.04	4.35	4.01	**4.94**	4.74
Up and Down	0.354	0.704	1.77	1.64	**2.20**
Wizard of Wor	0.957	0.946	**0.961**	0.520	0.519
Yars' Revenge	≈ 0	0.003	0.005	0.005	**0.008**

Table 6.1: Average number of forward time steps correctly predicted for the fixed-size and QTF models. The first five games constitute the training set. Highest per-game accuracy is indicated in bold blue.

6.5.2 Results

After training, I evaluated the models' ability to predict the future conditional on action sequences. This evaluation phase took place over an additional 8000 frames, with action sequences again drawn by selecting actions uniformly at random and executing them for $k \in K$ frames. At each time step t, each model was asked to predict 90 steps ahead and recorded how many frames were perfectly generated. Predictions at $t+k$ were thus made using the true history up to time t and the $k-1$ already sampled frames.

Table 6.1 summarizes the result of this evaluation. Which patch size best predicts game screens depends on a number of factors, such as the size of in-game objects, their frame to frame velocity and the presence of animated backgrounds. QTF achieves a level of performance reasonably close to the best patch size, above 95% in 13 out of 20 games. In some games, QTF even improves on the performance of the best fixed-size model. Ultimately, Theorem 6.3.1 ensures that QTF will asymptotically achieve a prediction accuracy comparable to the best decomposition available to it.

Sequential, probabilistic prediction algorithms such as QTF are often evaluated based on their n-step average logarithmic loss, defined as $\frac{1}{n}\sum_{i=1}^{n} -\log_2 \xi(x_i \mid ax_{<i}a_i)$ for an environment model ξ. This measure has a natural information theoretic interpretation: on average, losslessly encoding each percept requires this many bits. While counting the number of correct future frames is a conservative measure and is heavily influenced by the sampling process, the logarithmic loss offers a more fine-grained notion of predictive accuracy. As shown in Table 6.2, the quad-tree factorization achieves significantly lower per-frame loss than any fixed-size model. In view that each frame contains $160 \times 210 \times 7 = 235,200$ bits of data, the results for Pong and Freeway – achieving a logarithmic loss of 3 bits per frame – are particularly significant.

6.5.3 Discussion

The benefits of predicting observations using environment model factors corresponding to larger image patches are twofold. Large regular patterns, such as the invaders in Space Invaders, can easily be represented as a single symbol. When using a base model such as CTS, which treats symbols atomically, *sampling* from QTF is often faster as fewer symbols need to be generated. Thus the quad-tree factorization

	Logarithmic Loss (Base 2)				
	4×4	8×8	16×16	32×32	QTF
Asterix	81.83	29.44	188.2	2166	**17.77**
Beam Rider	850.6	335.4	710.8	4059	**68.63**
Freeway	12.68	8.19	50.88	251.1	**3.05**
Seaquest	101.8	173.9	1328	7887	**51.26**
Space Invaders	161.3	129.5	678.2	8698	**27.99**
Amidar	35.74	36.58	168.1	815.9	**9.95**
Crazy Climber	205.5	161.0	446.9	2172	**40.84**
Demon Attack	716.9	1587	5502	19680	**531.1**
Gopher	72.85	28.24	66.02	608.9	**9.97**
Krull	615.4	1103	4025	15220	**245.2**
Kung-Fu Master	74.62	59.33	179.9	1012	**20.87**
Ms. Pacman	109.2	183.5	1053	6362	**48.9**
Pong	33.75	13.71	23.07	121.0	**3.24**
Private Eye	453.5	623.4	1922	7956	**162.6**
River Raid	298.5	256.5	1034	6055	**77.35**
Star Gunner	438.0	481.2	1980	10790	**139.2**
Tennis	178.5	290.7	945.6	4134	**93.6**
Up and Down	1461	2220	5104	14490	**854.7**
Wizard of Wor	134.3	81.98	277.9	1778	**26.42**
Yars' Revenge	667.4	1251	3264	20810	**493.3**

Table 6.2: Per-frame logarithmic loss for the fixed-size and QTF models, averaged over the whole training sequence. The first five games constitute our training set. The lowest loss is indicated in bold blue.

produces a forward model of Atari 2600 games which is both efficient and accurate.

In my experiments, I used Context Tree Switching (CTS) models as environment model factors. One limitation of this approach is that CTS has no provision for partial symbol matches: altering a single pixel within an image patch yields a completely new symbol. This presents a significant difficulty, as the size of the alphabet corresponding to $\mathcal{F}$ grows exponentially with its nesting depth d. The results of Table 6.2 are symptomatic of this issue: larger patch size models tend to suffer higher loss. As QTF is a meta-algorithm independent of the choice of environment model factors, other base models perhaps better suited to noisy inputs may improve predictive accuracy, for example locally linear models (Farahmand et al., 2009), dynamic bayes networks (Walsh et al., 2010), and neural network architectures (Sutskever et al., 2008).

6.6 Conclusion

In this chapter, I described an algorithm for learning forward models of large factored domains such as Atari 2600 games. The main contributions are

1. A framework for describing recursive factorizations,
2. An instantiation of this framework, quad-tree factorization (QTF), applicable to observation spaces with two-dimensional structure such as Atari 2600 game screens, and
3. Empirical and theoretical results showing the validity of the approach: QTF performs as well as the best factorization within its recursively defined class of factorizations.

The chief benefit of using such a framework, and more specifically the quad-tree factorization, is that it obviates the need to choose a factorization a priori. The model learning results presented here illustrate the importance of the domain-independent approach: different Atari 2600 games do, in fact, require different factorizations. The quad-tree factorization algorithm can be said to be domain-independent as it provably works well across the set of Atari 2600 games.

There is another contribution that emerges, indirectly, from the work described in this chapter. To the best of my knowledge, the QTF model implemented here is the first to tackle observation spaces of the magnitude of Atari 2600 game screens. While the forward models learned here are by no means equal in power to the true Atari 2600 simulator, this work constitutes an encouraging step forward.

Simultaneously, the question arises of how to best act given an imperfect model of the environment. Recently, Joseph et al. (2013) argued that, when no member of the model class can represent the true environment, a mismatch arises between model accuracy and model usefulness: the best policy may not rely on the most accurate model. How to best address this issue is likely to be key in a real-world setting; in such a setting, the sheer complexity of the world in which our agents are immersed precludes learning error-free models.

6.7 Related Work

Learning environment models has been an ongoing area of study since the first days of reinforcement learning research (Sutton, 1991; Thrun and Mitchell, 1993). In

fully observable environments, Sutton et al. (2008) recently investigated the effects of combining Dyna planning with linear function approximation, while Hester et al. (2010) learned the dynamics of a Nao robot in order to improve its sample efficiency when learning to kick in preparation for the RoboCup competition. Holmes and Isbell (2006) developed a tree-based method for concisely modelling deterministic, partially observable environments; by virtue of the environment's determinism, it is possible to exactly model its dynamics using a prediction suffix tree augmented with loops. Talvitie and Singh (2008) proposed the notion of partial models: models that predict only a portion of the future observation. The authors argued that partial models are simpler to learn in partially observable domains, and are often sufficient to learn to behave well or predict reward. Recently, predictive state representations (PSRs, Littman et al., 2002) were shown to constitute a practical tool for learning the dynamics of partially observable environments (Boots et al., 2010). In this work, the authors proposed a relaxed version of a PSR, the transformed PSR (TPSR), that can easily be learned from data; they then showed that TPSRs can learn a model of a simple three-dimensional environment observed through a 768-pixel camera, and finally showed how this model was powerful enough to allow planning in this domain. Similar to the work presented in this chapter and in Veness et al. (2011), where compression techniques are used to model the world, Farias et al. (2010) proposed a model learning approach based on the Lempel-Ziv compression algorithm. Although related to their work, the quad-tree factorization algorithm tackles an orthogonal problem, namely the selection of a good observation factorization from a large recursive class of factorizations.

A variety of model learning solutions have been proposed for reinforcement learning domains with factored state or observation spaces. The typical framework used by such solutions is the Factored Markov Decision Process (Boutilier et al., 2000), which assumes that the environment can be represented by a dynamic bayes network (DBN) in which each state variable at time $t + 1$ is stochastically determined by a subset of variables at time t. Degris et al. (2006) proposed an algorithm that learns the model of a factored MDP environment using a decision tree; after learning, the algorithm then performs value iteration using this model. Strehl et al. (2007) proposed a theoretically-grounded algorithm for simultaneously learning the structure and parameters of a DBN environment model; their work provides a probably approximately correct (PAC) bound on the number of samples needed to learn a DBN

with bounded maximum in-degree. Following this work, Diuk et al. (2009) adapted an exploration scheme known as the Adaptive k-Meteorologists to the problem of exploration in reinforcement learning, also assuming a DBN model of the environment. Common to these algorithms is their focus on modelling state variables (with the assumption that the environment is Markov); by contrast, the recursive factorization described here explicitly considers factored observations rather than factored states.

To handle partially observable environments, McCallum (1995) proposed a series of algorithms, culminating with U-Tree, which constructs a tree-based representation of the agent's history. Shani et al. (2005) expanded on this idea to explicitly model a POMDP using Utile Suffix Memory (U-Tree's precursor). More recently, Poupart (2008) and Ross and Pineau (2008) simultaneously studied the idea of Bayesian learning of a POMDP model. In the Bayesian setting, the agent begins with a prior over possible POMDPs and updates its prior with each new observation-action pair, enabling it to act in a Bayes-optimal way (given sufficient computation). Both approaches considered POMDP models described by DBNs and Dirichlet priors over the DBN parameters and propose to track the posterior over DBNs using particle filtering.

> The young man saw himself in the mirror behind the bar. "I said I was a different man, James," he said. Looking into the mirror he saw that this was quite true. "You look very well, sir," James said. "You must have had a very good summer."
>
> *The Sea Change*
> Ernest Hemingway

Chapter 7

Conclusion

In this thesis I proposed three new algorithms suited to factored reinforcement learning domains. The first of these algorithms stems from a formal definition of contingency awareness, and consists in learning a model of such contingency and subsequently using it to detect the location of the player avatar in Atari 2600 games. This location is then used to improve the feature set used to perform value function approximation. The second algorithm, tug-of-war hashing for linear value function approximation, combines existing ideas from the sketch literature and recent work on random projections to improve over the standard hashing scheme previously used in linear value function approximation. My final contribution is the quad-tree factorization (QTF) algorithm and the more general framework of recursive factorizations in which it falls. QTF performs efficient Bayesian model averaging over a large class of factored models suitable for observation spaces with an underlying two-dimensional structure, such as the image space of Atari 2600 games. A fourth, albeit minor contribution can be found in Chapter 3, where I proposed the score distribution as a method for properly comparing empirical results on a large set of domains. As part of my thesis, I also worked on making a new release of the Arcade Learning Environment available to interested researchers.

The overarching aim of this work was to investigate fast, scalable reinforcement learning algorithms that can be easily applied to arbitrary Atari 2600 games. This domain-independence requirement was upheld by empirically validating each algorithm on a large set of games. Although the empirical results presented here will surely be bettered within the coming year, part of their relevance is to show that scientific progress *is* possible, on large, semirealistic domains such as Atari 2600 games.

One critical aspect of reinforcement learning this thesis neglects is the question of exploration. As it has amply been demonstrated elsewhere (White and White, 2010; Li et al., 2009; Kolter and Ng, 2009; Dearden et al., 1999), exploration is a vital component of any reinforcement learning agent. Two of the three contributions made here, contingency awareness and forward model learning, can be directly exploited for exploration purposes: in one case by explicitly performing exploration in the simpler, two-dimensional space of the player avatar, and in the other by looking for action sequences that result in high model uncertainty. How to best formalize and utilize these ideas is a question left for future work.

Most of the work in this thesis could also be validated on other large, factored domains. In the game of Starcraft, for example, Churchill et al. (2012) has studied k-unit coordination. Here both the action and state spaces involved are simply too large for most techniques; tug-of-war SARSA(λ) could then be used to learn an approximate value function based on exhaustive generation of simple Starcraft features.

Finally, none of the learning agents produced over the course of this thesis achieve even reasonable human-player scores on the full gamut of Atari 2600 games. Many of the standard reinforcement learning tools – such as ϵ-greedy exploration, exhaustive feature generation, and single time-scale policies – are simply unsuited to the complexity of Atari 2600 games. Consider, for example, the well-known PITFALL!. In PITFALL!, the player's goal is to navigate from screen to screen, avoiding two-frame crocodiles and pixelated quicksand, looking for jewels and gold that increase their score. Despite the game's conceptual simplicity, no domain-independent algorithm has been shown to satisfactorily tackle it. Effectively, there remains a large gap between, on one hand, our abstract understanding of most Atari 2600 games, and on the other the low-level control required to play them; filling this gap is perhaps one of the most difficult challenges ahead of us.

Bibliography

Achlioptas, D. (2003). Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687.

Aji, S. M. and McEliece, R. J. (2000). The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343.

Asmuth, J. and Littman, M. L. (2011). Learning is planning: Near bayes-optimal reinforcement learning via Monte-Carlo tree search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Auer, P., Herbster, M., and Warmuth, M. K. (1995). Exponentially many local minima for single neurons. In *Advances in Neural Information Processing 7*.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013a). The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47.

Bellemare, M. G., Veness, J., and Bowling, M. (2012a). Investigating contingency awareness using atari 2600 games. In *Proceedings of Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Bellemare, M. G., Veness, J., and Bowling, M. (2012b). Sketch-based linear value function approximation. In *Advances in Neural Information Processing Systems 25*.

Bellemare, M. G., Veness, J., and Bowling, M. (2013b). Bayesian learning of recursively factored environments. In *Proceedings of the Thirtieth International Conference on Machine Learning*.

Bellman, R. E. (1957). *Dynamic programming*. Princeton University Press, Princeton, NJ.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.

Boots, B., Siddiqi, S. M., and Gordon, G. J. (2010). Closing the learning-planning loop with predictive state representations. *Proceedings of Robotics: Science and Systems 6*.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107.

Bowling, M. and Veloso, M. (2002). Scalable learning in stochastic games. In *AAAI Workshop on Game Theoretic and Decision Theoretic Agents*.

Bowling, M. and Veloso, M. (2003). Simultaneous adversarial multi-robot learning. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 699–704.

Bowling, M., Wilkinson, D., and Ghodsi, A. (2006). Subjective mapping. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1569–1572.

Boyan, J. A. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2):233–246.

Bradtke, S. J. and Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1):33–57.

Carter, J. L. and Wegman, M. N. (1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154.

Churchill, D., Saffidine, A., and Buro, M. (2012). Fast heuristic search for RTS game combat scenarios. In *Proceedings of the Eight Artificial Intelligence and Interactive Digital Entertainment Conference*.

Cobo, L. C., Zang, P., Isbell, C. L., and Thomaz, A. L. (2011). Automatic state abstraction from demonstration. In *Proceedings of the Twenty-Second International Joint Conference on Articial Intelligence*.

Cormode, G. and Garofalakis, M. (2005). Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the Thirty-First International Conference on Very Large Data Bases*, pages 13–24.

Cormode, G. and Muthukrishnan, S. (2005). An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

Dasgupta, A., Kumar, R., and Sarlós, T. (2010). A sparse Johnson-Lindenstrauss transform. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, pages 341–350.

Dearden, R., Friedman, N., and Andre, D. (1999). Model based bayesian exploration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*.

Degris, T., Sigaud, O., and Wuillemin, P. (2006). Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the Twenty-third International Conference on Machine learning*. ACM.

Diuk, C., Andre Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning*, pages 240–247.

Diuk, C., Li, L., , and Leffler, R. B. (2009). The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the Twenty-Sixth International Conference on Machine learning*.

Doshi-Velez, F. (2009). The infinite Partially Observable Markov Decision Process. In *Advances in Neural Information Processing Systems 22*.

Farahmand, A., Shademan, A., Jägersand, M., and Csaba Szepesvári (2009). Model-based and model-free reinforcement learning for visual servoing. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Farias, V. F., Moallemi, C. C., Roy, B. V., and Weissman, T. (2010). Universal reinforcement learning. *IEEE Transactions on Information Theory*, 56(5):2441–2454.

Frankl, P. and Maehara, H. (1988). The johnson-lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Series B*, 44(3):355–362.

Friedman, N. and Singer, Y. (1999). Efficient bayesian parameter estimation in large discrete domains. *Advances in Neural Information Processing Systems 11*.

Ghavamzadeh, M., Lazaric, A., Maillard, O.-A., and Munos, R. (2010). LSTD with random projections. In *Advances in Neural Information Processing Systems 23*, pages 721–729.

Guez, A., Silver, D., and Dayan, P. (2012). Efficient bayes-adaptive reinforcement learning using sample-based search. In *Advances in Neural Information Processing Systems 25*.

Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. (2012). HyperNEAT-GGP: A HyperNEAT-based Atari general game player. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.

Hester, T., Quinlan, M., and Stone, P. (2010). Generalized model learning for reinforcement learning on a humanoid robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2369–2374.

Holmes, M. P. and Isbell, Jr., C. L. (2006). Looping suffix tree-based inference of partially observable hidden state. In *Proceedings of the Twenty-third International Conference on Machine Learning*, pages 409–416.

Hutter, M. (2005). *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer.

Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613.

Joseph, J., Geramifard, A., Roberst, J. W., How, J. P., and Roy, N. (2013). Reinforcement learning with misspecified model classes. In *IEEE International Conference on Robotics and Automation*.

Kane, D. M. and Nelson, J. (2010). A derandomized sparse Johnson-Lindenstrauss transform. *arXiv preprint arXiv:1006.3585*.

Keller, P. W., Mannor, S., and Precup, D. (2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the Twenty-Third international conference on Machine learning*, pages 449–456.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. pages 282–293.

Kolter, Z. J. and Ng, A. Y. (2009). Near-bayesian exploration in polynomial time. In *Proceedings of the Twenty-Sixth Annual International Conference on Machine Learning*, pages 513–520.

Krichevsky, R. and Trofimov, V. (1981). The performance of universal coding. *IEEE Transactions on Information Theory*, 27:199–207.

Leffler, B., Littman, M., and Edmunds, T. (2007). Efficient reinforcement learning with relocatable action models. In *Proceedings of the National Conference on Artificial Intelligence*.

Legg, S. (2008). *Machine super intelligence*. PhD thesis, University of Lugano.

Li, L., Littman, M. L., and Mansley, C. R. (2009). Online exploration in least-squares policy iteration. In *Proceedings of The Eighth International Conference on Autonomous Agents and Multiagent Systems*.

Li, P., Hastie, T. J., and Church, K. W. (2006). Very sparse random projections. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 287–296.

Littman, M. L., Sutton, R. S., and Singh, S. (2002). Predictive representations of state. In *Advances in Neural Information Processing Systems 14*, pages 1555–1561.

Maillard, O.-A. and Munos, R. (2009). Compressed least squares regression. In *Advances in Neural Information Processing Systems 22*, pages 1213–1221.

McCallum, A. K. (1995). *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester.

Menache, I., Mannor, S., and Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238.

Miller, W. and Glanz, F. (1996). The University of New Hampshire implementation of the cerebellar model arithmetic computer – CMAC. Technical report, University of New Hampshire.

Montfort, N. and Bogost, I. (2009). *Racing the beam: The Atari Video Computer System*. MIT Press.

Moore, A. and Atkeson, C. (1994). The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In *Advances in Neural Information Processing Systems 6*.

Naddaf, Y. (2010). Game-independent ai agents for playing atari 2600 console games. Master's thesis, University of Alberta.

Nelson, J. and Woodruff, D. P. (2010). Fast manhattan sketches in data streams. In *Proceedings of the Twenty-Ninth Symposium on Principles of Database Systems*, pages 99–110.

Nguyen, P. M., Sunehag, P., and Hutter, M. (2012). Context tree maximizing reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Oudeyer, P., Kaplan, F., and Hafner, V. (2007). Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286.

Parr, R., Painter-Wakefield, C., Li, L., and Littman, M. (2007). Analyzing feature generation for value-function approximation. In *Proceedings of the Twenty-Fourth International Conference on Machine learning*, pages 737–744.

Pierce, D. and Kuipers, B. (1997). Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92(1-2):169–227.

Poupart, P. (2008). Model-based bayesian reinforcement learning in partially observable domains. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc.

Ring, M. (1997). CHILD: A first step towards continual learning. *Machine Learning*, 28(1):77–104.

Rissanen, J. (1983). A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–663.

Ron, D., Singer, Y., and Tishby, N. (1996). The power of amnesia: Learning probabilistic automata with variable memory length. *Machine learning*, 25(2):117–149.

Ross, S. and Pineau, J. (2008). Model-based bayesian reinforcement learning in large structured domains. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Rummery, G. A. (1995). *Problem solving with reinforcement learning*. PhD thesis.

Russell, S. and Norvig, P. (2003). *Artificial intelligence: A modern approach*. Prentice Hall Englewood Cliffs.

Russell, S. J. (1997). Rationality and intelligence. *Artificial intelligence*, 94(1):57–77.

Rusu, F. and Dobra, A. (2007). Statistical analysis of sketch estimators. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 187–198. ACM.

Schuitema, E., Hobbelen, D., Jonker, P., Wisse, M., and Karssen, J. (2005). Using a controller based on reinforcement learning for a passive dynamic walking robot. In *Proceedings of the Fifth IEEE-RAS International Conference on Humanoid Robots*, pages 232–237.

Shani, G., Brafman, R. I., and Shimony, S. E. (2005). Model-based online learning of POMDPs. In *Proceedings of the European Conference on Machine Learning*, pages 353–364.

Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., Strehl, A., and Vishwanathan, V. (2009). Hash kernels. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*.

Silver, D., Sutton, R. S., and Müller, M. (2007). Reinforcement learning of local shape in the game of Go. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1053–1058.

Silver, D., Sutton, R. S., and Müller, M. (2008). Sample-based learning and search with permanent and transient memories. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning*.

Silver, D., Sutton, R. S., and Müller, M. (2012). Temporal-difference search in computer Go. *Machine Learning*, 87(2):183–219.

Silver, D. and Veness, J. (2010). Monte-Carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems 23*.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Stober, J. and Kuipers, B. (2008). From pixels to policies: A bootstrapping agent. In *Proceedings of the Seventh IEEE International Conference on Development and Learning*, pages 103–108. IEEE.

Stone, P., Sutton, R. S., and Kuhlmann, G. (2005). Reinforcement learning for RoboCup soccer keepaway. *Adaptive Behavior*, 13(3):165.

Strehl, A. L., Diuk, C., and Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. In *Proceedings of the National Conference on Artificial Intelligence*.

Sturtevant, N. and White, A. (2006). Feature construction for reinforcement learning in Hearts. *Computers and Games*, pages 122–134.

Sutskever, I., Hinton, G., and Taylor, G. (2008). The recurrent temporal restricted Boltzmann machine. In *Advances in Neural Information Processing Systems 21*.

Sutton, R., Modayil, J., Delp, M., Degris, T., Pilarski, P., White, A., and Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagents Systems*.

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT Press.

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. (2008). Dyna-style planning with linear function approximation. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*.

Talvitie, E. and Singh, S. (2008). Simple local models for complex dynamical systems. In *Advances in Neural Information Proceesing Systems 22*.

Tedrake, R., Zhang, T. W., and Seung, H. S. (2004). Stochastic policy gradient reinforcement learning on a simple 3D biped. In *Proceedings of the Conference on Intelligent Robots and Systems*, volume 3, pages 2849–2854.

Thrun, S. and Mitchell, T. M. (1993). Lifelong robot learning. In *Robotics and Autonomous Systems*.

Tjalkens, T. J., Shtarkov, Y., and Willems, F. (1993a). Context tree weighting: Multi-alphabet sources. In *Proceedings of the Fourteenth Symposium on Information Theory in the Benelux*, pages 128–135.

Tjalkens, T. J., Shtarkov, Y. M., and Willems, F. M. (1993b). Sequential weighting algorithms for multi-alphabet sources. In *Sixth Joint Swedish-Russian Workshop on Information Theory*.

Tsitsiklis, J. N. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690.

Veness, J. and Hutter, M. (2012). Sparse sequential Dirichlet coding. *ArXiv e-prints*.

Veness, J., Ng, K. S., Hutter, M., and Bowling, M. H. (2012). Context tree switching. In *Proceedings of the Data Compression Conference*, pages 327–336.

Veness, J., Ng, K. S., Hutter, M., and Silver, D. (2010). Reinforcement learning via AIXI approximation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Veness, J., Ng, K. S., Hutter, M., Uther, W. T. B., and Silver, D. (2011). A Monte-Carlo AIXI Approximation. *Journal of Artificial Intelligence Resesearch*, 40:95–142.

Walsh, T. J., Goschin, S., and Littman, M. L. (2010). Integrating sample-based planning and model-based reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the Twenty-Sixth Annual International Conference on Machine Learning*, pages 1113–1120. ACM.

White, M. and White, A. (2010). Interval estimation for reinforcement-learning algorithms in continuous-state domains. In *Advances in Neural Information Processing Systems 22*.

Willems, F., Shtarkov, Y., and Tjalkens, T. (1995). The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41:653–664.

Willems, F. and Tjalkens, T. (1997). Complexity reduction of the context-tree weighting algorithm: A study for KPN research. *EIDMA Report RS.97.01*.

Williams, R. and Baird, L. C. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Northeastern University.

Wintermute, S. (2010). Using imagery to simplify perceptual abstraction in reinforcement learning agents. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Appendix A

Algorithmic Details

A.1 Atari 2600 Feature Generation

In Chapter 5, I omitted for clarity the exact method used to generate the features necessary to approximate the value function of Atari 2600 games; this appendix now fills this omission. A *c-coloured pattern* ζ is a set of pixels sharing a common colour c. For a given image $o \in \mathcal{O}$, the $a \times b$ c-coloured pattern at (x, y) is defined as

$$\zeta^{a,b}_{x,y}(o, c) := \{(x', y', c) : (x', y', c) \in p^{a,b}_{x,y}(o)\}$$

Intuitively, $\zeta^{a,b}_{x,y}(o, c)$ describes the set of pixels taking on colour c within a $a \times b$ tile whose top-left corner is (x, y). These colour patterns form the basis of our feature generation process. Similar to the image patch indices described in Section 3.3, define the c-colour pattern index as

$$\imath^{a,b}_{x,y}(o, c) := \sum_{y'=y}^{y+b-1} \sum_{x'=x}^{x+a-1} 2^{a(y'-y)+(x'-x)} \mathbb{I}_{\left[o_{x',y'}=c\right]} , \tag{A.1}$$

where the previous encoding of $o_{x,y}$ is now replaced by a binary encoding indicating whether pixel (x', y', c') matches colour c. Note, in particular, that the index 0 corresponds to the absence of colour c anywhere within the tile of size $a \times b$ located at (x, y). Below, this *binary index encoding* is generalized to encode arbitrary tuples.

We define the set of all $a \times b$ c-coloured patterns for observation o as

$$Z^{a,b}(o, c) := \{(x, y, \imath^{a,b}_{x,y}(o, c)) : (x, y) \in \mathcal{D}\}$$

The next step is to encode the *presence* of patterns within a $a' \times b'$ tile located at (x', y'). For a given colour c, the set of $a \times b$ c-coloured pattern indices present in this tile is

$$\text{PATTERNS}_{a,b,c}(o, a', b', x', y') := \{\imath^{a,b}_{x,y}(o, c) : (x, y) \in l^{a',b'}_{x',y'}(o), \imath^{a,b}_{x,y}(o, c) \neq 0\}$$

This somewhat unwieldy definition now allows us to encode which patterns are present in o jointly with their location, in the same fashion as Equation A.1.

To reduce the number of patterns provided to the agent at each time step, I consider the *foreground image*, as was originally suggested by Naddaf (2010), rather than all pixels. Recall from Section 4.4.1 the definition of a foreground image $\bar{o}$ given a background image $\text{bg} \in \mathcal{O}$:

$$\bar{o} := \{(x, y, c) : (x, y, c) \in o, c \neq \text{bg}_{x,y}\}.$$

Note that $\bar{o} \subset o$: the foreground image does not constitute a full, 160×210 set of pixels. However, none of the equations above explicitly require such a full set: we can just as easily extract patterns from $\bar{o}$ as from o.

Let $R := \{\{a_1, b_1\}, \ldots \{a_r, b_r\}\}$ denote a set of tile sizes, and p a maximum pattern size (in the experiments of Chapter 5, $p = 3$). Feature generation proceeds by encoding the presence of patterns of size $1 \times 1, 2 \times 2, \ldots p \times p$ within each screen tile. This process is summarized as Algorithm 4 (recall that d_w and d_h denote the screen width and height, respectively). Figures A.1 and A.2 depict extracted patterns for game screens respectively taken from Seaquest and Space Invaders.

We add a further level of refinement to the feature set by jointly encoding the presence of colour patterns together with the avatar location; this location is obtained using the tools developed in Chapter 4. Here we take a slightly different approach from Chapter 4: instead of quantizing the estimated avatar location (x_a, y_a), the whole screen is *translated* by an amount corresponding to said location. The translated screen $\text{TRANSLATE}(o, x_a, y_a) \in \mathcal{O}$ is defined as:

$$\text{TRANSLATE}(o, x_a, y_a) := \{(x, y, c) : (x_a + x, y_a + y, c) \in o, x \in \mathcal{D}^*_x, y \in \mathcal{D}^*_y\},$$

where $\mathcal{D}^*_x := \{x : x = 0 \text{ or } x \in \mathcal{D}_x \text{ or } -x \in \mathcal{D}_x\}$ is the extension of the set of columns indices $\mathcal{D}_x$ to negative values, with $\mathcal{D}_y$ defined similarly. Thus $\text{translate}(o, x_a, y_a)$ defines a screen whose $(0, 0)$ location corresponds to the estimated avatar position. The extension to $\mathcal{D}^*_x$ and $\mathcal{D}^*_y$ is necessary to encode both positive and negative offsets from the avatar position. With minor modifications, Algorithm 4 can be used to generate a location-dependent feature vector ϕ'. The feature vector used in Chapter

Algorithm 4 Colour Pattern-Based Feature Generation

Require: An input image $o \in \mathcal{O}$
Require: A background image bg
Require: Parameters R, p
  Initialize feature vector $\phi \leftarrow \mathbf{0}$
  Extract $\bar{o}$ the foreground image
  for $(a_l, b_l) \in R$ **do**
    for $i = 0 \ldots \frac{d_w}{a_l} - 1$ **do**
      for $j = 0 \ldots \frac{d_h}{b_l} - 1$ **do**
        for $k = 1 \ldots p$ **do**
          for $c \in \mathcal{C}$ **do**
            for index $\in$ PATTERNS$_{k,k,c}(\bar{o}, \frac{d_w}{a_l}, \frac{d_h}{b_l}, i \times a_l + 1, j \times b_l + 1)$ **do**
              Encode the tuple $(l, i, j, k, c, \text{index})$ as an integer q
              $\phi_q \leftarrow 1$
            end for
          end for
        end for
      end for
    end for
  end for

5 is the concatenation of both location-dependent and location-independent feature vectors.

Implementation Notes

The tuple encoding method used in Algorithm 4 produces integers that must be stored in 64-bit precision numbers and subsequently hashed. More specifically, a tuple $(x_1, x_2, \ldots, x_l)$ whose integer variables range (without loss of generality) from 0 to $X_1, X_2, \ldots, X_l$ produces integers q ranging from 0 to $X_1 X_2 \ldots X_l - 1$. By only considering c-coloured patterns whose top row contains a pixel of colour c, the total number of $i \times i$ c-coloured patterns in a single screen is $id_w d_h$; with a few optimizations, the feature vector output by Algorithm 4 can be generated in time $O(rp^3 d_w d_h)$, excluding the time needed to perform hashing. To provide a sense of practicality, consider that a reasonable Java implementation with $r = 3$ and $p = 6$ which also performs avatar tracking runs at 60 frames per second, or real-time, using a single 2.4Ghz Intel Core i5 processor. A parallel implementation of both avatar tracking and feature generation would result in much higher speeds.

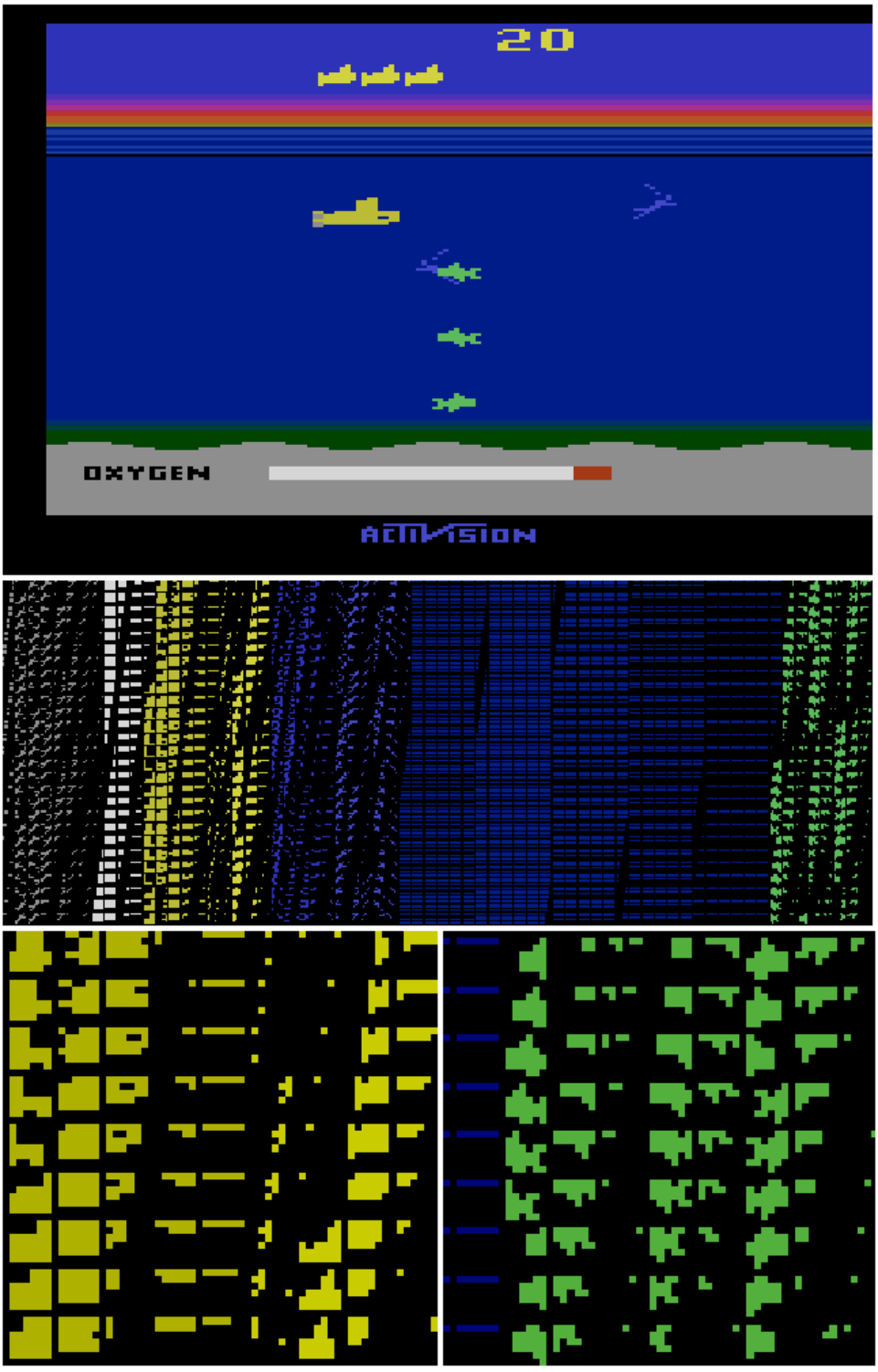

Figure A.1: Colour patterns of size up to 6×6 extracted from a given SEAQUEST screen. **Top.** Original screen. **Middle.** Mosaic of 1856 extracted patterns. **Bottom.** Mosaic of extracted patterns (detail).

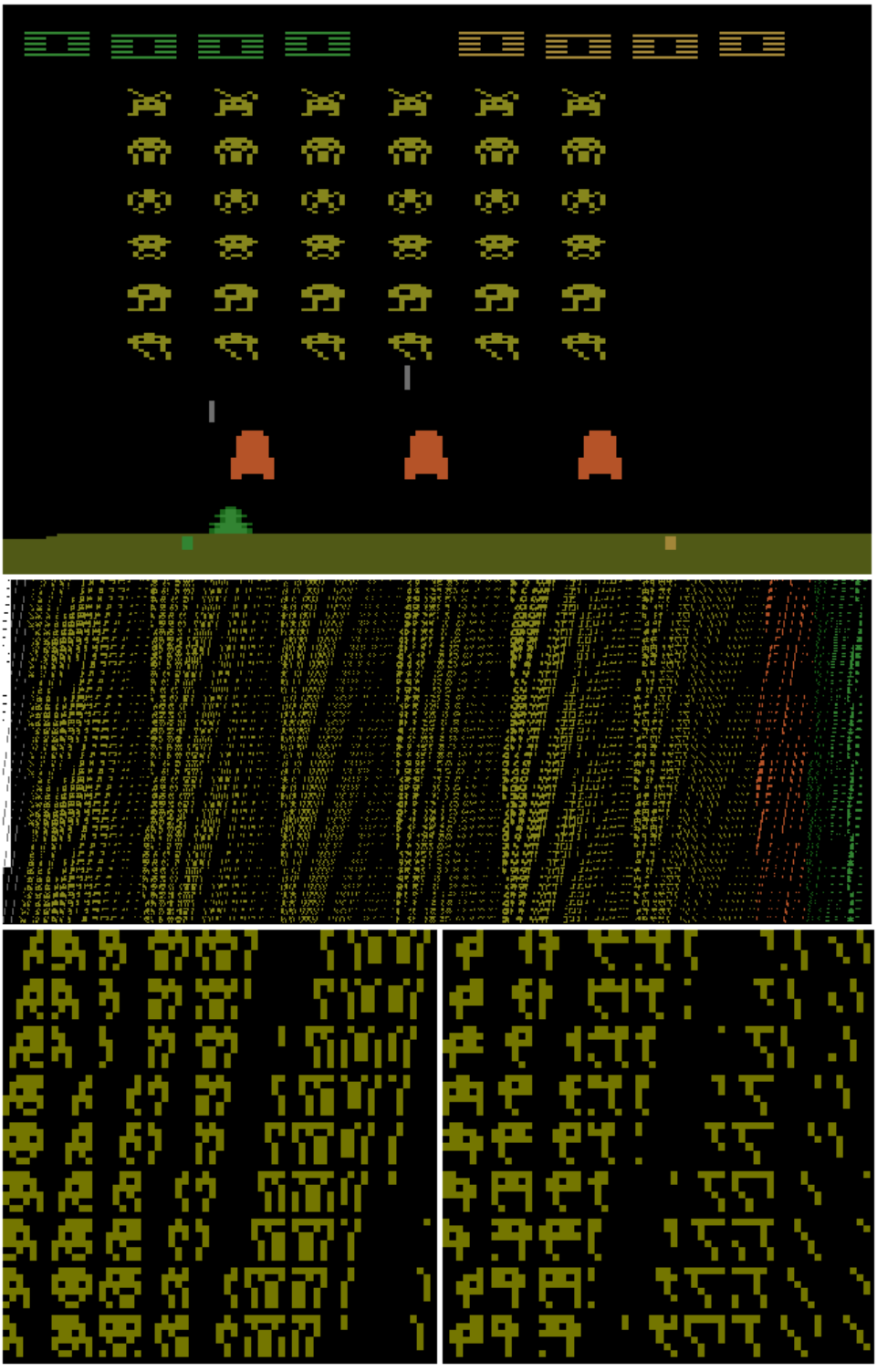

Figure A.2: Colour patterns of size up to 6×6 extracted from a given SPACE INVADERS screen. **Top.** Original screen. **Middle.** Mosaic of 4452 extracted patterns. **Bottom.** Mosaic of extracted patterns (detail).

A.2 QTF: Base Environment Model Factors

As described in Section 6.5, the QTF forward model predicted using base environment model factors corresponding to 32×32, 16×16, 8×8 and 4×4 image patches. Each patch was predicted using a single CTS model shared across all patches of the same size. I now detail the process of making predictions using this CTS model; in what follows a fixed patch size model is assumed, with the implementation easily extending to the quad-tree factorization.

Recall the notation for the screen width, d_w, and screen height d_h. For a patch size $a \times b$, define the set of tile locations $L_{a,b} := \{(x, y) : x, y \in \mathbb{N}, ax < d_w, by < d_h\}$; a factored model operating on patches of size $a \times b$ makes $|L|$ predictions per screen. Recall also the definition of an image patch index (Section 3.3):

$$\imath^{a,b}_{x,y}(o) := \sum_{y'=y}^{y+b-1} \sum_{x'=x}^{x+a-1} |\mathcal{C}|^{a(y'-y)+(x'-x)} \text{bit}(o_{x',y'}).$$

In AIXI notation, upon receiving a new screen $o_t \in \mathcal{O}$ the CTS model must predict $|L|$ symbols, corresponding to $\{\imath^{a,b}_{ax,by}(o_t) : (x, y) \in L_{a,b}\}$. Note that $\imath^{a,b}_{ax,ay}(o_t)$ ranges from 0 to $|C|^{a\times b}$; the index is therefore first hashed down to a 32-bit integer $\hat{\imath}^{a,b}_{ax,by}(o_t)$. As a, b are fixed and the coordinates ax, by, implied, I denote by $\hat{\imath}_{x,y,t} := \hat{\imath}^{a,b}_{ax,ay}(o_t)$. The following ordered set of features is provided to the CTS model as context for the prediction of $\hat{\imath}^{a,b}_{x,y,t}$:

1. $\hat{\imath}^{a,b}_{x,y,t-1}$	2. $\hat{\imath}^{a,b}_{x,y,t-2}$	3. $\hat{\imath}^{a,b}_{x-1,y,t-1}$
4. $\hat{\imath}^{a,b}_{x,y-1,t-1}$	5. $\hat{\imath}^{a,b}_{x,y+1,t-1}$	6. $\hat{\imath}^{a,b}_{x+1,y,t-1}$
7. The last action taken	8. $\hat{\imath}^{a,b}_{x-1,y,t-2}$	9. $\hat{\imath}^{a,b}_{x,y-1,t-2}$
10. $\hat{\imath}^{a,b}_{x,y+1,t-2}$	11. $\hat{\imath}^{a,b}_{x+1,y,t-2}$	

Out-of-bounds image patches, as well as patches from $t \leq 0$, are mapped to index 0. This particular context was designed using the training games; other contexts led to slightly different but qualitatively comparable results. In effect, the shared CTS model treats image patches as integer symbols, which are understood by its SSD estimators as coming from an alphabet of (finite) size $|C|^{a\times b}$.

The final step in the screen prediction process is to transform the set of CTS predictions, $P := \{\hat{\imath}_{x,y,t} : (x, y) \in L_{a,b}\}$, into a 160 × 210 Atari 2600 game screen. This is done by performing a reverse lookup on the elements $\hat{\imath}_{x,y,t} \in P$ to obtain the corresponding image patch p; this image patch is then copied onto location (ax, by) of the output screen prediction.

Appendix B

List of Games

B.1 Training Games

The following games were used, across all chapters, in the design and parameter optimization of this thesis's algorithms:

- Asterix
- Beam Rider
- Freeway
- Seaquest
- Space Invaders

B.2 Testing Games: Chapter 4

The following games were used to evaluate the algorithms of Chapter 4.

Alien	Amidar	Assault
Asteroids	Atlantis	Bank Heist
Battle Zone	Berzerk	Bowling
Boxing	Breakout	Carnival
Centipede	Chopper Command	Crazy Climber
Double Dunk	Enduro	Fishing Derby
Frostbite	Gopher	Gravitar
H.E.R.O.	Ice Hockey	James Bond
Journey Escape	Kangaroo	Krull
Kung-Fu Master	Montezuma's Revenge	Ms. Pacman
Name This Game	Pooyan	Pong
Private Eye	Q*Bert	River Raid
Road Runner	Robotank	Star Gunner
Time Pilot	Tutankham	Up and Down
Venture	Wizard of Wor	Zaxxon
Video Pinball		

B.3 Testing Games: Chapter 5

The following games were used to evaluate the algorithms of Chapter 5.

Alien	Amidar	Assault
Asteroids	Atlantis	Bank Heist
Battle Zone	Berzerk	Bowling
Boxing	Breakout	Carnival
Centipede	Chopper Command	Crazy Climber
Demon Attack	Double Dunk	Elevator Action
Enduro	Fishing Derby	Frostbite
Gopher	Gravitar	H.E.R.O.
Ice Hockey	James Bond	Journey Escape
Kangaroo	Krull	Kung-Fu Master
Montezuma's Revenge	Ms. Pacman	Name This Game
Pooyan	Pong	Private Eye
Q*Bert	River Raid	Road Runner
Robotank	Star Gunner	Tennis
Time Pilot	Tutankham	Up and Down
Venture	Video Pinball	Wizard of Wor
Yar's Revenge	Zaxxon	

B.4 Testing Games: Chapter 6

The following games were used to evaluate the algorithms of Chapter 6.

Pong	Wizard of Wor	Kung-Fu Master
Gopher	Yar's Revenge	Amidar
Tennis	Star Gunner	Ms. Pacman
Demon Attack	Private Eye	Crazy Climber
River Raid	Up and Down	Krull