Developing a predictive approach to knowledge

by

Adam White

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Understanding how an artificial agent may represent, acquire, update, and use large amounts of knowledge has long been an important research challenge in artificial intelligence. The quantity of knowledge, or knowing a lot, may be nicely thought of as making and updating many predictions about many different courses of action. This predictive approach to knowledge ensures the knowledge is grounded in and learned from low-level data generated by an autonomous agent interacting with the world. Because predictive knowledge can be maintained without human intervention, its acquisition can potentially scale with available data and computing resources. The idea that knowledge might be expressed as prediction has been explored by Cunningham (1972), Becker (1973), Drescher (1990), Sutton and Tanner (2005), Rafols (2006), and Sutton (2009, 2012). Other uses of predictions include representing state with predictions (Littman, Sutton &, Singh 2002; Boots et al. 2010) and modeling partially observable domains (Talvitie & Singh 2011). Unfortunately, technical challenges related to numerical instability, divergence under off-policy sampling, and computational complexity have limited the applicability and scalability of predictive knowledge acquisition in practice.

This thesis explores a new approach to representing and acquiring predictive knowledge on a robot. The key idea is that value functions, from reinforcement learning, can be used to represent policy-contingent declarative and goal-oriented predictive knowledge. We use recently developed gradient-TD methods that are compatible with off-policy learning and function approximation to explore the practicality of making and updating many predictions in parallel, while the agent interacts with the world from continuous inputs on a robot.

The work described here includes both empirical demonstrations of the effectiveness of our new approach and new algorithmic contributions useful for scaling prediction learning. We demonstrate that our value functions are practically learnable and can encode a variety of knowledge with several experiments—including a demonstration of the psychological

phenomenon of nexting, learning predictions with refined termination conditions, learning policy-contingent predictions from off-policy samples, and learning procedural goal-directed knowledge—all on two different robot platforms. Our results demonstrate the potential scalability of our approach; making and updating thousands of predictions from hundreds of thousands of multi-dimensional data samples, in realtime and on a robot—beyond the scalability of related predictive approaches. We also introduce a new online estimate of off-policy learning progress, and demonstrate its usefulness in tracking the performance of thousands of predictions about hundreds of distinct policies. Finally, we conduct a novel empirical investigation of one of our main learning algorithms, GTD(λ), revealing several new insights of particular relevance to predictive knowledge acquisition. All told, the work described here significantly develops the predictive approach to knowledge.

Acknowledgements

There are many people that helped me along the long road to finishing this document. First and foremost my supervisor Rich Sutton. He is of course a visionary, a careful scientist, a leader of my field of study, and someone who encourages his students to work on big and bold ideas—to not be afraid of being ahead of the field. He would always generate new and unexpected idea's and look at a problem in a unique way to help me get unstuck. Rich taught me two important lessons as a student which I think had a large role in my success. The first lesson was to pay attention and love getting the tiny details right. The second was to never be afraid of a new problem or obstacle; many things can be solved and we should be excited about the prospect of such a pursuit.

I also owe many thanks to Joseph Modayil. He joined our research group when I first starting working with off-policy learning and robots, and his vision, ideas, wisdom, and guidance were essential in helping me become a useful scientist. Joseph was my co-supervisor in every way but official title.

I thank the postdocs, Thomas Degris, Patrick Pilarski, Hado van Hasselt, and Harm van Seijen. The lively research meetings, long afternoon coffees, dinners, and board game nights were so productive both professionally and personally.

Finally, I thank the members of my examining committee, Pierre-Yves Oudeyer, Marek Reformat, Pierre Boulanger, and Michael Bowling for their intriguing questions, and helpful suggestions for improving both the content of my work and presentation of my ideas.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Understanding how an artificial agent may represent, acquire, update, and use large amounts of knowledge has long been an important research challenge in artificial intelligence. Intelligent agents know a lot about their world. Humans know that the sun rises every day, how long it will take to get to the bathroom, and that significant energy is required to lift a heavy box. Artificial agents, such as Google's self-driving car, know the regions that are drivable and not, the likely future positions of moving objects, and what to do when a light turns red. Future intelligent agents—such as a nanny bot to care for the elderly—may know how to vacuum the floor, and what time you usually wake.

Much of what people believe to be everyday or common-sense knowledge may be thought of as predictive. Knowing your keys are in the desk is to predict that you will see the keys if you opened the desk drawer. Knowing about a chair is to predict the success of attempting to sit on it. Knowing how a ball will feel in your hand is to predict the sensations you should observe if you picked up the ball. We call these policy-contingent predictions because they are statements about the outcome of a course of action; opening the drawer, sitting, and picking up. The quantity of knowledge, or knowing a lot, can be nicely thought of as making and updating many predictions about many different courses of action.

Predictive knowledge acquisition is potentially more scalable than other forms of knowledge that rely on correctness defined by people. Updating a prediction can be as simple as making a prediction, following some course of action, observing the outcome, and refining the prediction—learning from an agent's interaction with the world. A prediction's correctness is defined by how well it matches what actually occurs when the agent takes the course of action corresponding to the prediction. In comparison, conventional objective knowledge—such as a database of logical facts—may be maintained correct by logical

inference or human consistency checking. Unfortunately, as the size of the database of knowledge increases, both human-driven and logical consistency checking become error prone and computationally infeasible, respectively. Predictive knowledge acquisition, on the other hand, may be scaled with computation and data. Computational resources constrain how many predictions the agent can store and how quickly those predictions may be updated; experience governs the accuracy of the agent's predictions. The potential scalability of predictive knowledge acquisition has by and large remained an untested hypothesis.

This thesis explores the practicality of predictive representations of knowledge by attempting to improve the applicability and scalability of prediction specification and acquisition. The idea that knowledge might be expressed as prediction has been explored by Cunningham (1972), Becker (1973), Drescher (1990), and Sutton (2009, 2012), but in only small scale simulations. Other well-developed uses of prediction, including predictive state representations (PSRs) (Littman et al., 2002; Singh et al., 2004; Bowling et al., 2006; Wingate & Singh, 2008), temporal difference networks (TD-nets) (Sutton & Tanner, 2005; Sutton et al., 2006), and prediction profiles (Talvitie & Singh, 2011) have also been limited to small, discrete simulations. More recently spectral learning techniques, from system identification, have been used to learn the parameters of a PSR from continuous low-level sensor data produced by a robot (Boots et al., 2010; Boots & Gordon, 2011). All told, these prior demonstrations of prediction learning have been limited to making and updating a few dozen predictions, and do not easily extend to updating predictions from continuous inputs at the same time as the agent is interacting with the world. This thesis describes a new approach to predictive knowledge based on ideas and algorithms from reinforcement learning, and demonstrates for the first time making and updating thousands of predictions from hours of high-dimensional sensor data produced by a mobile robot. Our new approach improves the applicability and scalability of the predictive approach to knowledge.

1.1 Objective

This thesis seeks to answer the following question:

> Can we develop a predictive approach to knowledge, improving the applicability and scalability of incremental prediction learning?

Learning predictive knowledge is taken to mean approximating a mapping from observed data to summary statements about the future. To acquire predictive knowledge is to refine and update predictions based on the agent's experience: the data generated by an

agent exploring its world. We call this learning online, because the agent modifies its prediction immediately after each new observation, using an incremental learning algorithm. Our agent will learn many predictive mappings, each about the outcome of following a target policy.

Improving the applicability of prediction learning refers to both prediction specification and how the predictions are learned. This thesis investigates policy-contingent predictions: what-if predictions about the outcomes of following a decision making policy. Such predictions can compactly summarize an observed outcome contingent on a closed-loop temporally extended way of behaving (the policy), while abstracting away a potentially long sequence of observations and decisions that are not relevant to the prediction's outcome. This thesis also investigates how to learn predictions from continuous data, such as a robot's sensor readings, and how predictions can be updated off-policy. In the off-policy setting, we update a prediction about the observed outcome of following a target policy from data generated while the agent follows another, potentially different, policy. This combination of the policy-contingent prediction, learned off-policy from continuous inputs, has not been previously demonstrated.

One of the main objectives of this thesis is to build agents that could know a lot, and thus our work focuses on scaling predictive knowledge learning. Here we consider scaling in two dimensions: scaling the length of data stream produced by the agent interacting with the world, and scaling the number of unique predictions to be learned. Our objective is to demonstrate learning more predictions, from more data than any other previous predictive knowledge implementation, and to develop representations and learning algorithms that facilitate scaling with available computational resources.

Overall, our aim is to broaden the class of predictions that may be posed, increase the domains in which those predictions may be learned from data, and scale up the number of things that can be predicted online during learning. The next section describes our approach to achieving these objectives.

1.2 Approach

The approach explored in this thesis has three key components: (1) learning predictions from and about low-level data produced by mobile robots, (2) framing the task of learning a prediction as one of estimating a value function, and (3) learning predictions with incremental temporal difference (TD) off-policy learning algorithms from reinforcement

learning.

Similar to prior work on PSRs and TD-nets, the agent's task is to predict the low-level data produced by its own interaction with the world. In this setting, the agent has no access to data structures with externally defined semantics, such as a true state or a motion model; rather, the agent only observes a stream of low-level data. As opposed to a task-driven setting, our aim for the agent is for it to continually make and update many different predictions about the future values of its own data stream.

Unlike almost all prior work in predictive knowledge learning, our agent will seek to predict the sensor stream produced by a mobile robot. One of our robots, called the Critterbot, produces a 53 dimensional vector of real-valued sensor readings roughly every 10 milliseconds (ms): $1.8\text{x}10^7$ sensor readings an hour. Our agent's task is to predict this continuous-valued, low-level sensor stream on a moment-by-moment basis, making and updating each prediction in realtime as the data are produced. This setup facilitates specification of large numbers of predictions and generating ample data for learning and evaluating each prediction.

We represent the problem of predicting future sensory data as a value estimation problem. Reinforcement learning is a problem formalism in which an agent learns a policy to maximize a scalar reward signal, usually corresponding to some explicit goal or desired behavior. Many reinforcement learning algorithms estimate a value function as an intermediary step inside iterative policy learning algorithms. A value function is an estimate of the expected total future reward the agent would receive if it followed some policy from the current state—it is a prediction of future reward. In this thesis, we develop the idea that rewards need not be tied to some objective the agent is trying to achieve; rather a reward may be taken to be any signal of interest that the agent wishes to predict, such as one of the robot's sensors. The agent's task is to estimate a general value function (GVF), similar to a conventional value function but defined with this broader view of reward, and a time-varying definition of discounting and termination. One consequence of this approach is that value function learning algorithms from reinforcement learning may be applied to the task of learning predictive knowledge. Specifically, efficient, linear-complexity TD-based learning methods can be used to learn GVFs online and incrementally from continuous inputs via function approximation. Our GVF-based problem formulation is similar to prior work on option models and TD-nets. The main point of distinction is our use of modern off-policy reinforcement learning methods and more general function approximation, which improves applicability and substantially increases scalability compared to previous work.

Our approach to GVF learning makes use of recently developed off-policy gradient-TD methods. Off-policy updating facilitates asynchronous updating of multiple GVFs at a time, enabling massive scaling of online prediction learning. Somewhat surprisingly, conventional off-policy value function estimation methods, such as Q-learning, may diverge when used with function approximation (see Baird, 1995; Sutton & Barto, 1998). In addition, prior off-policy learning methods based on importance sampling (Precup et al. 2000, Sutton, Rafols & Koop, 2006) exhibited high variance, and thus the agent often learned more slowly than if it had sequentially updated each prediction with on-policy data (see Rafols, 2006). Recently, new TD methods were introduced which have convergence guarantees under off-policy updating with function approximation, and do not exhibit the same large variance under off-policy sampling. We use these new gradient-TD methods to learn thousands of predictions represented as GVFs on two different mobile robot platforms, demonstrating a measurable improvement in both the applicability and scalability of predictive knowledge learning.

1.3 Contributions

This thesis contains five proposed contributions.

General value functions (Chapter 4)

We propose GVFs as a language for representing predictive knowledge. Although formally presented in prior work (Maei & Sutton, 2010), we are the first to develop GVFs as a representation for policy-contingent predictive knowledge. We describe how the discount rate may be a function of the state and how rewards and discounting can be mixed, providing substantial additional flexibility and expressive power, compared to conventional value functions. GVFs may be learned in parallel with efficient, off-policy gradient-TD methods from continuous inputs using general function approximation architectures; an improvement in both applicability and potential scalability over previous approaches to representing and learning predictive knowledge.

Nexting (Chapter 5)

Psychologists refer to *nexting* as the propensity of people and many other animals to continually predict what will happen next in an immediate, local, and personal sense. We develop a computational model of nexting based on GVFs, and present results with a robot that

learns to next in realtime, making thousands of predictions about sensory input signals at time scales from 0.1 to 8 seconds. We show that 6000 GVFs, each computed as a function of 6000 features of the state, can be learned and updated online 10 times per second on a laptop computer, using the standard TD(λ) algorithm with linear function approximation. This approach is sufficiently computationally efficient to be used for realtime learning on the robot and sufficiently data efficient to achieve substantial accuracy within 30 minutes. Moreover, a single shared tile-coded feature representation suffices to accurately predict many different signals over a significant range of time scales. Our nexting experiment provides evidence for our claims regarding the expressiveness of GVFs as a language for predictive knowledge and the practicality of our specific approach to learning.

GVF learning on robots (Chapter 6)

We move beyond the simple time scales used in on-policy predictions nexting, and demonstrate learning GVFs of a significantly more general and expressive form, on two different robot platforms. Specifically, our experiments provide concrete examples of learning GVFs defined by mixtures of hard and soft terminations, outcome signals, and policy-contingent predictions learned from off-policy samples. These experiments help further demonstrate the representational capabilities of GVFs, and the practicality of learning GVFs with reinforcement learning methods on robots. This chapter contains both the first demonstrations of GVF learning on robots, and some of the most extensive empirical investigations of the practicality of off-policy gradient-TD learning on robots.

An empirical study of the GTD(λ) algorithm (Chapter 7)

Gradient-TD methods make exploring parallel, policy-contingent prediction learning practical for the first time, but our empirical experience with these new algorithms is limited. In Chapter 7, we perform an extensive empirical study of one gradient-TD method, the GTD(λ) algorithm. We investigate the secondary weight vector of GTD(λ) in several variants of a Markov chain task and one domain that causes divergence of conventional TD-based learning methods. Our experiments provide new insight into how well these weights are learned in practice, and how these weights affect the performance of the GTD(λ) algorithm overall. Our experiments are the first to specifically investigate the secondary weights and they reveal several new insights of particular relevance to predictive knowledge learning.

RUPEE, a measure of off-policy learning progress (Chapter 8)
Estimating the accuracy of predictions learned off-policy may be done in several ways, but many of these involve restricting the learning process in some way. We propose two new estimates of off-policy learning progress, called RUPEE, that estimate the objective function which gradient-TD methods optimize—the mean squared projected Bellman error (MSPBE). Our new measures can be estimated incrementally during learning with linear computational complexity and storage, and do not interfere with the learning process. We empirically evaluate how well our new measures estimate the MSPBE on several experiments in the Markov chain domain, and then show how both RUPEE measures provide a surrogate measure of the prediction accuracy of hundreds of policy-contingent predictions learned off-policy by GTD(λ) on a robot. Finally, we demonstrate one of the benefits of an online estimate of progress using one of the new RUPEE measures to estimate the learning progress of thousands of GVFs with hundreds of distinct policies on a robot.

This thesis also contains two smaller contributions. The first, found in Chapter 9, is a demonstration of adapting the behavior policy on a robot, based on the learning errors of multiple GVFs, similar to artificial curiosity. The second, found in Appendix A, is the derivation of a new gradient-TD method, called Hybrid-TD(λ), that automatically performs conventional TD updates when data are generated on policy, and performs corrected, off-policy updates otherwise, ensuring convergence.

1.4 Thesis layout

This document contains 10 chapters, one that covers background material, one that describes the robot hardware and sensorimotor data, followed by six chapters of technical content. This section provides selected descriptions of the contents of each chapter.

The second and third chapters can be viewed as background material that can be skipped by some readers. Chapter 2 describes important concepts in reinforcement learning, specifically value functions, function approximation, on-policy temporal difference learning, the MSPBE and off-policy temporal difference learning. This second chapter is not essential, as our problem formulation and notations will be defined in Chapter 4. Chapter 3 describes the two robots used throughout our experiments and contains several visualizations of the low-level sensory data produced by one of our robots. In the cases where some concept is also explained in Chapters 2 and 3, we will explicitly point out the connection.

Chapter 4 is divided into three main parts. The first part formally defines our prediction learning problem, specifying the assumptions, notations, and naming conventions used throughout the rest of the thesis. The second part defines GVFs and describes how predictions may be represented using GVFs. The third and final part discusses prior work on representing and learning predictive knowledge, pointing out the similarities and differences to GVFs.

Demonstrating the kinds of predictions that can be represented with GVFs and the practicality of learning GVFs on robots is the topic of Chapters 5 and 6. These chapters, along with the remaining technical chapters of this thesis are, summarized in the previous section. Chapter 10 concludes the thesis with a discussion of how successful we were in addressing the research question posed above. Chapter 10 closes the thesis with a discussion of related work, one idea corresponding to each technical chapter of the thesis.

1.5 Summary

In this chapter we introduced the topic of investigation tackled by this thesis, developing predictive knowledge learning, and described our approach based on learning policy-contingent predictions, represented as value functions, from low-level sensory data produced by a robot. We outlined the five proposed contributions described in the pages of this document, and closed with an outline of the thesis.

Chapter 2

Background

This chapter introduces and describes basic concepts of reinforcement learning which will be useful for understanding the rest of this document. We begin with the reinforcement learning problem formulation and Markov decision processes, then move on to value functions, linear function approximation, and algorithms for learning value functions from an agent's interaction with the world. We describe the off-policy learning problem, divergence of temporal difference methods, an objective function for off-policy learning, and a new family of gradient TD methods for learning value functions from off-policy samples. We finish the chapter with brief overviews of algorithms for learning policies to maximize reward, recent algorithmic advances in reinforcement learning, and option models.

The contents of this chapter are meant to serve as background only. Our precise problem formulation and notations will be described in Chapter 4. Readers familiar with estimating value functions, the mean squared projected Bellman error, and gradient temporal difference learning algorithms should feel free to skip this chapter.

2.1 Reinforcement learning

The decision maker and learner is called the *agent* and the entity the agent interacts with (and in fact everything that is not directly under the control of the agent) is called the *environment*. An agent's interaction with the world can be represented using the agent-environment interaction model of reinforcement learning (see Sutton and Barto, 1998). The agent's interaction with the environment occurs in discrete time steps. At the beginning of each time-step $t = 1, 2, 3, \ldots$, the agent receives a representation of the environment's state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is a finite set of possible states. Based on S_t the agent selects an action denoted by A_t, stochastically according to the *target policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\mathcal{A}$ is a finite set of possible actions available to the agent. We denote the probability of selecting

action a in state s under π with $\pi(s, a)$. On the next time-step, the agent receives a scalar reward $R_{t+1} \in \mathbb{R}$, and environment transitions to a new state S_{t+1} based, in part, on the agent's action A_t. This interaction, depicted in Figure 2.1, continues producing a temporal interaction stream of random states, actions, and rewards:

$$S_1, A_1, S_2, R_2, A_2, S_3, R_3, \ldots. \tag{2.1}$$

Often, the agent-environment interaction continues forever without interruption, which is referred to as *continuing*. Sometimes, there may be natural divisions in time where some goal is achieved, causing a transition into a *terminal state* $S_T \in \mathcal{S}$, at time T. In this *episodic* setting, time is divided into discrete, not necessarily equal length chunks called *episodes*. The specification of the environment's state-transition dynamics and reward function specifies an instance of a *reinforcement learning task*.

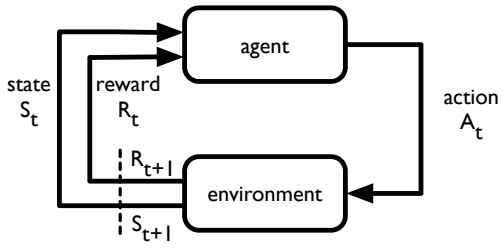

Figure 2.1: An reinforcement learning agent's interaction with an unknown environment.

An important concept underlying the design of many reinforcement learning algorithms is called the Markov property. The state, environment, and learning task are said to be *Markov* if the environment's state transition and reward at time $t + 1$ depend only on the previous state and action:

$$\mathbf{Pr}\{R_{t+1}, S_{t+1} | S_1, A_1, R_2, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} = \mathbf{Pr}\{R_{t+1}, S_{t+1} | S_t, A_t\},$$

for all possible histories of states, actions, and rewards. In other words, the state is said to be Markov if knowledge of the current state and action are sufficient for predicting the next state and expected next reward.

One special class of reinforcement learning tasks are Markov Decision Processes, or MDPs. If the state and action spaces are finite and the state is Markov, then the learning problem can be represented as a finite MDP. Here we define an MDP with a set of states and actions, and a specification of the one-step transition dynamics $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ where $P(s, a, s')$ is the probability of transitioning into state s', when taking action a in

state s. Additionally, the expected next reward is given by

$$r(s') \stackrel{\text{def}}{=} \mathbb{E}[R_{t+1}|S_{t+1} = s'].$$

Oftentimes, the expected next reward is defined as $r(s, a, s') \stackrel{\text{def}}{=} \mathbb{E}[R_{t+1}|S_{t+1} = s', A_t = a, S_t = s]$, instead of (2.1). We use this slightly less general definition of expected next reward, and consequently slightly less general formulation of MDPs, because it will be sufficient for our purposes and is more simple. At this point, it is also convenient to define the state transition dynamics while following policy π as: $P^\pi : \mathcal{S} \times \mathcal{S} \to [0, 1]$ where $P^\pi(s, s') \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}} \pi(s, a) P(s, a, s')$.

MDPs are a subclass of reinforcement learning problems, and provide the theoretical framework on which most modern reinforcement learning algorithms are based. All the algorithms used in this thesis assume that the world (which a robot operates in) corresponds to some unknown, possibly massive MDP. As you will see later, this is not as limiting as it appears. In fact, we can use the MDP framework to design learning algorithms with asymptotic guarantees, even in cases where the agent has access to an incomplete representation of the environment's state.

2.2 Value functions

Many reinforcement learning algorithms estimate the discounted sum of future rewards called the *return*. The return $G_t \in \mathbb{R}$, at time t is defined by:

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.2}$$

where the discount factor $\gamma \in (0, 1]$ discounts the contribution of future rewards as a function of time, ensuring the infinite sum is bounded. In episodic domains, the sum in Equation 2.2 terminates at a random time T, and γ is typically set to one indicating that all rewards are equally valuable. We can define the return to cover both continuing and episodic problems:

$$G_t \stackrel{\text{def}}{=} \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1},$$

where T may be ∞.

The agent might compute a value function in order to estimate how good a particular state is, in terms of future reward. A *state-value function* is a function $v^\pi : \mathcal{S} \to \mathbb{R}$, that maps states to expected returns. The value of state s is equal to the discounted sum of

future rewards the agent would observe, starting in state s, and selecting actions according to policy π and transitioning according to the dynamics of the MDP is defined by:

$$v^\pi(s) \stackrel{\text{def}}{=} \sum_{s' \in \mathcal{S}} P^\pi(s, s') \left[r(s') + \gamma v^\pi(s') \right] \tag{2.3}$$

$$= \mathbb{E}_{A_{t:\infty} \sim \pi, S_{t+1:\infty} \sim P}[G_t | S_t = s]$$

$$= \mathbb{E}_\pi \left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s \right]. \tag{2.4}$$

The notation in the subscript of the expectation is somewhat awkward, but is useful to explain how things are sampled. The expectation is computed over future states, which are generated by selecting actions according to π and state transitions according to P. From now on, we will use the shorthand $\mathbb{E}_\pi$. Equation 2.3 is known as the Bellman equation for v^π, and specifies the recursive relationship between the value of state s and the value of the next state s'.

The *state-action value function*, $q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, maps states and actions to expected returns. The state-action value of state s and action a is equal to the discounted sum of future rewards the agent would observe, starting in state s, selecting the action a, and selecting actions according to policy π thereafter transitioning according to the dynamics of the MDP is defined by:

$$q(s, a) \stackrel{\text{def}}{=} \sum_{s' \in \mathcal{S}} P(s, a, s') \left[r(s') + \gamma \sum_{a' \in \mathcal{A}} \pi(s', a') q^\pi(s', a') \right] \tag{2.5}$$

$$= \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

$$= \mathbb{E}_\pi \left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \tag{2.6}$$

Equation 2.5 is known as the Bellman equation for q^π.

2.3 Function approximation

In domains where there are a large number of states, it can be challenging to compute a value function. In order to store and update an estimate of the value function, the agent must fill in a table with one entry for each state or for each state or state-action pair. What if we wish to estimate value functions based on high dimensional and continuous robot sensor data? The state space may be very large and the state is likely unobservable by the agent. The agent may never observe the same state twice; filling in the table may take considerable time and memory. In such domains, we seek to generalize the values of previously observed states to

other states. In order to generalize, we can use samples of the desired function (usually the value function) to construct an approximation of the entire function. We approximate the value function. Function approximation is an instance of supervised learning and therefore, in principle, many supervised methods such as artificial neural networks, decision trees, and multi-variate regression can be used to approximate the value function.

In this thesis we use linear function approximation. We do not assume the agent can directly observe the state of the environment; instead the agent's learning is based on a vector representation of the state called a *feature vector*. Corresponding to every state, $s \in \mathcal{S}$, there is a column vector of features

$$\mathbf{x} \stackrel{\text{def}}{=} \mathbf{x}(s) \stackrel{\text{def}}{=} (f^{(1)}(s), f^{(2)}(s), \ldots, f^{(n)}(s))^\top, \tag{2.7}$$

where the $f^{(i)} : \mathcal{S} \to \mathbb{R}$ are called feature functions. The size of the feature vector, $\mathbf{x}$, is denoted by $n \ll |\mathcal{S}|$. This mapping is usually not one-to-one: many underlying MDP states may correspond to the same feature vector. The feature may be constructed from incomplete and noisy observations of the state, for example robot sensor readings or a history of sensor readings. We denote the current feature vector by $\mathbf{x}_t \stackrel{\text{def}}{=} \mathbf{x}(S_t)$.

In the linear function approximation setting, the value function estimate $\hat{v} : \mathcal{S} \times \mathbb{R}^n \to \mathbb{R}$, is a linear function of a modifiable weight vector $\mathbf{w} \in \mathbb{R}^n$. The *approximate state value function* $\hat{v}$ is defined by:

$$v(s;\pi) \approx \hat{v}(s, \mathbf{w}) \stackrel{\text{def}}{=} \mathbf{w}^\top \mathbf{x}(s),$$

and the agent's current estimate of the value for state S at time t is defined by:

$$\mathbf{w}_t^\top \mathbf{x}_t = \sum_{i=1}^{n} f^{(i)}(S_t) w_t(i).$$

Usually, the agent's approximation of the value function is limited by the approximation architecture, perhaps because the number of feature functions is much less than the number of states in $\mathcal{S}$ or because v is not a linear function. As a consequence, we must balance the agent's limited resources—the finite set of modifiable weights. We may be forced to decide in which situations a higher approximation error is acceptable, and in which situations a more accurate value estimation is needed.

In the case of approximate state-action value functions, the feature vector is defined in a slightly different way. Corresponding to every state, $s \in \mathcal{S}$, and action, $a \in \mathcal{A}$ there is a column vector of features:

$$\mathbf{x}_a \stackrel{\text{def}}{=} f(s,a) \stackrel{\text{def}}{=} (f^{(1)}(s,a), f^{(2)}(s,a), \ldots, f^{(m)}(s,a))^\top, \tag{2.8}$$

and the $f^{(i)} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ are feature functions. The size of the state-action feature vector, $\mathbf{x}_a$, is denoted by $m \ll |\mathcal{S}|$. The state-action feature vector for the current state and action is denoted by $\mathbf{x}_{a,t} \stackrel{\text{def}}{=} \mathbf{x}(S_t, A_t)$. The approximate state-action value function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^m \rightarrow \mathbb{R}$, is defined by:

$$q(s, a; \pi) \approx \hat{q}(s, a, \mathbf{w}) \stackrel{\text{def}}{=} \mathbf{w}^\top \mathbf{x}(s, a),$$

with a modifiable weight vector $\mathbf{w} \in \mathbb{R}^m$. The agent's current estimate of the value for state s and action a at time t is defined by:

$$\mathbf{w}_t^\top \mathbf{x}_{a,t} = \sum_{i=1}^{m} f^{(i)}(s, a) w(i).$$

Linear approximation architectures have several attributes that make them attractive for learning value functions in large and continuous domains. Linear architectures are simple to implement and debug. One can determine which feature components have the highest and lowest contributions to the approximate value function by comparing the learned parameter values, which can help with feature function engineering. Many of the theoretical guarantees for reinforcement learning algorithms apply to the case of linear function approximation (see Sutton and Barto, 1998). Linear architectures, however, are unable to represent many functions that can be represented by non-linear approximation architectures like neural networks. This limitation can sometimes be overcome by using *non-linear* basis functions $f^{(i)}(S)$, as we do in tile coding, which is described next. In this case, the approximate value function is still linear in $\mathbf{w}$, but the representation capacity may be improved.

Tile coding is a technique for converting continuous variables into sparse feature vectors, and this approach is well suited for learning while interacting with the environment. The inputs are taken in small groups and partitioned into non-overlapping regions called tiles. One example tiling over two continuous sensor readings is shown on the left side of Figure 2.3. In tile coding, the feature vector is binary and sparse: $\mathbf{x} \in [0, 1]^n$. The number of active components or ones is constant and thus the norm of the feature vector remains constant. The approximate value function can be efficiently queried by only accessing the small set of active indices. We can query $\hat{v}(s, \mathbf{w})$ by:

$$\hat{v}(s, \mathbf{w}) = \sum_{i \in \mathcal{F}} w(i).$$

We can query $\hat{q}(s, a, \mathbf{w})$ by:

$$\hat{q}(s, a, \mathbf{w}) = \sum_{i \in \mathcal{M}} w(i),$$

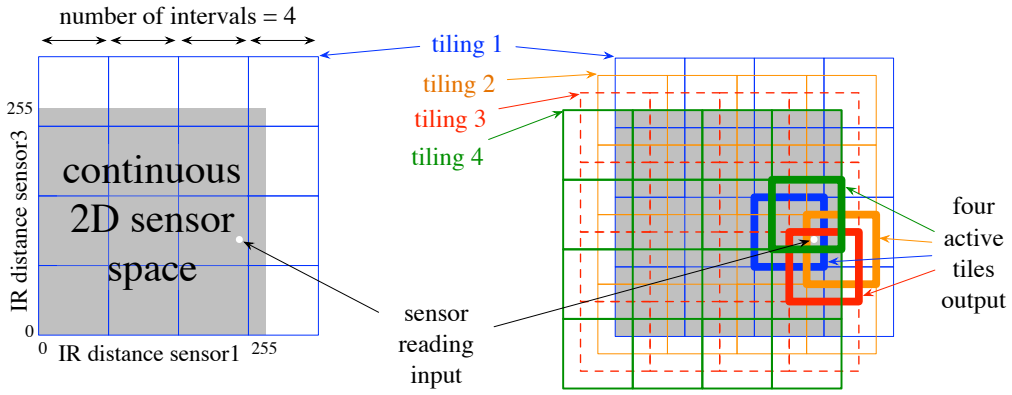

Figure 2.2: This is an example of how tile coding can be used to map continuous sensor data into a binary feature vector. On the left we see a single tiling of the continuous 2D space corresponding to the output of two distance sensors. The space was tiled into four intervals in each dimension, for 16 tiles overall. On the right we see all four tilings, each offset by a different negative amount such that they were equally spaced, with the first tiling starting at the lower left of the sensor space (as shown on the left) and the last tiling ending at the upper right of the space. A sensor reading input to the tile coder is a point in the space, like that shown by the white dot. The output of tile coding is the indices of the four tiles that contain the point, as shown on the right. These tiles are said to be active, and their corresponding components take on the value one, while all the non-active tiles correspond to feature components with the value zero. Note how the four tilings provide a dense grid of lines, each a distinction that can be made between input points, yet the four active tiles together span a substantial portion of the sensor space. In this way, multiple tilings provides a binary representation that enables both fine resolution and broad generalization.

where $\mathcal{F}$ and $\mathcal{M}$ denote the indices of the active tiles for $\mathbf{x}$ or $\mathbf{x}_a$. In tile coding, the size of the feature vector corresponds to the total number of tiles, not the dimensionality of that state space. The agent is thus free to use available computational resources in whatever way best suits the task at hand: using fewer tiles when a small number of components is sufficient for learning, and the maximum allowable computation as desired.

Tile coding is considerably more powerful than a simple discretization of the states. Multiple overlapping tilings can be offset from each other, as shown in the right side of Figure 2.3. An input (e.g., the white dot in the figure) activates a single tile in each tiling. The resolution of the tile coding is finer than that of the individual tilings, as can be seen by the greater density of lines in Figure 2.3 (right). With four tilings, the effective resolution is roughly four times that of the original tiling. The advantage of the multiple tilings over a single tiling with four times the resolution is that generalization will be broader with multiple tilings, which typically leads to much faster learning. With tile coding, one can quickly learn a coarse approximation to the desired mapping and then refine it with further

data, simultaneously obtaining the benefits of both coarse and fine discretizations.

2.4 Estimating the value function

An agent can learn an estimate of the state value function from samples generated by interaction with the environment. On each time-step, the agent observes a sample transition $\{\mathbf{x}(S_t), R_{t+1}, \mathbf{x}(S_{t+1})\}$, from which the agent can update its estimate of $\hat{v}(S_t, \mathbf{w})$ by adapting the modifiable weight vector $\mathbf{w}$:

$$\delta_t = R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(S_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(S_t) \tag{2.9}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{x}(S_t) \tag{2.10}$$

where $\alpha > 0$ is the step-size parameter and δ_t is a random variable. This algorithm is called linear temporal difference learning of linear TD(0).

The TD error δ_t, is the target of the learning rule; it is the quantity we wish to make zero. In this case, the target is formed by a difference between the value estimate in state S_t and value estimate in state S_{t+1}, plus the reward for the transition into S_{t+1}. Loosely speaking, the algorithm is trying to make successive value estimates close. More precisely, the TD error will be zero if $\hat{v}(S_t, \mathbf{w}) = \hat{v}(S_{t+1}, \mathbf{w}) + R_{t+1}$, and the algorithm would make no update to $\mathbf{w}$. The update of linear TD(0) is said to *bootstrap* its estimate of the value of one state with the value of the next state.

At convergence, as $t \to \infty$, the weight vector learned by linear TD(0) satisfies:

$$\begin{aligned}\mathbb{E}_\pi[\delta_t \mathbf{x}(S_t)] &\stackrel{\text{def}}{=} \sum_{s_t \in \mathcal{S}} d_\pi(s_t) \sum_{s_{t+1} \in \mathcal{S}} P^\pi(s_t, s_{t+1}) \big(r(s_{t+1}) \\ &\quad + \gamma \mathbf{x}(s_{t+1})^\top \mathbf{w}_t - \mathbf{x}(s_t)^\top \mathbf{w}_t \big) \mathbf{x}(s_t) \\ &= 0,\end{aligned}$$

called the TD(0) fixed point. The expectation is weighted by the *stationary distribution* under the policy, $d_\pi : \mathcal{S} \to [0, 1]$. This is the state occupation probability for each state $s \in \mathcal{S}$, equal to the limiting distribution produced by π and P: $d_\pi(s) \stackrel{\text{def}}{=} \lim_{t \to} \mathbf{Pr}(S_t = s)$.

Instead of bootstrapping the estimate between two states, we can use the TD error of several state transitions in the weight vector $\mathbf{w}$. The *linear TD(λ)* algorithm (Sutton, 1988) is specified as follows:

$$\begin{aligned}\delta_t &= r_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(S_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(S_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \delta_t \mathbf{e}_t \\ \mathbf{e}_t &= \mathbf{e}_{t-1} \gamma \lambda + \mathbf{x}(S_t),\end{aligned}$$

where $\mathbf{e} \in \mathbb{R}^n$ is called the eligibility trace, $\mathbf{e}_0 = \vec{0}$, and $\lambda \in [0, 1]$ is the bootstrapping parameter. The eligibility trace can be updated in a number of ways. The equation above, which we use, is called an *accumulating trace*. The linear TD(λ) algorithm spans a spectrum from Monte Carlo methods, when λ is one, to one-step TD methods at the other, when λ is zero. For values of λ in between, TD(λ) becomes of intermediate method that is often better than either extreme (see Sutton & Barto, 1998). The algorithm uses $O(n)$ storage and computation per update. In the limit of infinite sampling, the linear TD(λ) can find the weight vector $\mathbf{w}_t$ that satisfies:

$$\mathbb{E}_\pi[\delta_t \mathbf{e}_t] = 0, \tag{2.11}$$

called the TD(λ) fixed point. The proof of the convergence of linear TD(λ) algorithm to the TD(λ) fixed point depends on several technical conditions on α, the mixing characteristics of the MDP and policy π, and that linear function approximation is used (see Szepesvari, 2010 for precise details). In practice, the linear TD(λ) algorithm can outperform the well-known least mean squares rule (also known as Widrow-Hoff algorithm) on temporal prediction tasks (Sutton, 1988), and has been successfully applied to more complex domains like Backgammon (Tesauro, 1995), Computer GO (Silver et al., 2007), and elevator scheduling (Crites & Barto, 1996).

Instead of incrementally updating the modifiable weight vector from a sequence of agent- environment interactions, we can directly compute the solution to the TD fixed point using least squares. Given a finite batch of T samples, we wish to make the error zero: $\frac{1}{T}\sum_{t=1}^{T} \delta_t \mathbf{e}_t = 0$. Here $\delta_t \in \mathbb{R}$ refers to the non-random sample of the TD error from time-step t:

$$\delta_t = r_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(s_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(s_t).$$

We form a linear system of equations,

$$A_T \mathbf{w}_T = \mathbf{b}_T, \tag{2.12}$$

where $\mathbf{b}_T \overset{\text{def}}{=} \frac{1}{T}\sum_{t=1}^{T} r_{t+1}\mathbf{e}_t$, $\mathbf{b} \in \mathbb{R}^n$, $A_T \overset{\text{def}}{=} \frac{1}{T}\sum_{t=1}^{T}(-\mathbf{e}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top)$, and $A_T \in \mathbb{R}^{n \times n}$. Assuming A is non-singular, then:

$$\mathbf{w}_T = A_T^{-1}\mathbf{b}_T$$

gives the solution equal to the weight vector to which the linear TD(λ) converges. This algorithm is called *least-squares temporal difference learning* or LSTD(λ) due to Boyan

(2002),[1] The LSTD(λ) algorithm has an incremental form:

$$\begin{aligned}
\mathbf{e}_t &= \gamma\lambda\mathbf{e}_{t-1} + \mathbf{x}(S_t) \\
\mathbf{b}_{t+1} &= \mathbf{b}_t + \mathbf{e}_t R_{t+1} \\
\mathbf{y} &= \mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1}) \\
\hat{A}_{t+1} &= \hat{A}_t - \frac{(\hat{A}_t\mathbf{e}_t)(\mathbf{y}^\top\hat{A}_t)}{1 + (\mathbf{y}^\top\hat{A}_t)\mathbf{e}_t}.
\end{aligned}$$

The matrix $\hat{A}_0$ can be initialized equal to the identity matrix times a scalar, $\mathbf{I}\eta$, $\eta \in \mathbb{R}$, to improve the stability of the algorithm in early learning when few samples have been observed. Aside from this optional initialization, there are no learning rate parameters for this incremental form of the LSTD(λ). This update procedure uses the Sherman-Morrison formula to incrementally compute the inverse of the A matrix, which we denoted by $\hat{A}$. This algorithm uses $O(n^2)$ computation and storage per update.

The LSTD(λ) algorithm can be substantially more sample efficient than the TD(λ) algorithm (Boyan, 1999; Boyan, 2002; Xu et al., 2002; Geramifard et al., 2006). This may not be too surprising given LSTD(λ) computes the solution to which the TD(λ) algorithm converges. In some domains, particularly non-stationary ones, the TD(λ) algorithm can outperform LSTD(λ).

2.5 Off-policy learning

The agent can learn about some policy other than the one used to generate actions. In reinforcement learning, the actions are chosen according to some *behavior policy* $\mu : \mathcal{S} \times \mathcal{A} \to [0, 1]$, while the objective of learning is to estimate the value function v^π. The expectation in the definition of the value function is conditioned on actions selected according π. This is the target of the agent's learning, and thus π is the target policy. In the usual, *on-policy* setting, the target and behavior policies are the same. In *off-policy* learning, the target and behavior policies differ.

One challenge in off-policy reinforcement learning is to estimate the value function for the target policy with samples generated from the behavior policy. The behavior policy may choose actions differently than the target policy in some states. To correct for potential mis-matches between π and μ, we can compute the *likelihood ratio* of the probability of choosing action A_t in state S_t under both policies:

$$\rho_t \stackrel{\text{def}}{=} \frac{\pi(S_t, A_t)}{\mu(S_t, A_t)}. \tag{2.13}$$

[1] The LSTD(0) algorithm was introduced by Bradtke & Barto (1996).

The expectation of ρ_t is one under the behavior policy:

$$\mathbb{E}_\mu[\rho_t | S_t = s] = \sum_{a \in \mathcal{A}} \mu(s,a) \frac{\pi(s,a)}{\mu(s,a)} = \sum_{a \in \mathcal{A}} \pi(s,a) = 1.$$

The likelihood ratio will be exactly one for on-policy transitions and greater than zero on off-policy transitions. Note that $\mu(S_t, A_t)$ is always greater than zero. Using the likelihood ratios, we can construct a linear off-policy TD(0) update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \rho_t \alpha \delta_t \mathbf{x}(S_t), \tag{2.14}$$

where δ_t is the same as before.

Unfortunately, temporal difference methods, like the linear off-policy TD(0) algorithm described above, can be shown to diverge under off-policy sampling in the case of linear function approximation. Baird (1995) provided a counterexample that causes the linear TD(0) update to become unstable with off-policy updating, and the modifiable weight vector to diverge to infinity. Another example of divergence is given by Bertsekas and Tsitsiklis (1996), example 6.7. Counterexamples for convergence exist for other temporal difference methods including Q-learning (Baird, 1995) and TD(0) in the case of non-linear function approximation (Bertsekas & Tsitsiklis, 1996).

Recently a new family of algorithms was introduced that can be shown to *not* diverge under off-policy sampling in the case of linear function approximation. These methods minimize the mean squared projected Bellman error or MSPBE:

$$\text{MSPBE}(\mathbf{w}) \stackrel{\text{def}}{=} ||\mathbf{X}\mathbf{w} - \Pi T^{\gamma\lambda} \mathbf{X}\mathbf{w}||_B^2, \tag{2.15}$$

first introduced by Antos et al. (2008).

At this point, it is convenient to use matrix notation to explain the terms inside the MSPBE. Let $\mathbf{X} \in \mathbb{R}^{|\mathcal{S}| \times n}$ be a matrix with $|\mathcal{S}|$ rows, one for each $s \in \mathcal{S}$, and each row of X is equal to $\mathbf{x}(s)$. The approximation of the value of each state is given by the column vector $V^\pi \stackrel{\text{def}}{=} \mathbf{X}\mathbf{w} \in \mathbb{R}^{|\mathcal{S}|}$, one entry for each state in $\mathcal{S}$. The matrix $B \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ is a diagonal matrix whose diagonal enteries correspond to the stationary distribution under the behavior policy: $B(s,s) \stackrel{\text{def}}{=} d_\mu(s) \; \forall \; s \in \mathcal{S}$. The projection matrix $\Pi \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ is equal to $\mathbf{X}(\mathbf{X}^\top B \mathbf{X})^{-1} \mathbf{X}^\top B$. The *Bellman operator* $T^{\gamma\lambda} : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ helps to define the Bellman equation of the value function, in matrix form:

$$V^\pi \stackrel{\text{def}}{=} \mathbf{R} + \gamma P^\pi V^\pi \tag{2.16}$$

$$\stackrel{\text{def}}{=} T^{\gamma\lambda} V^\pi, \tag{2.17}$$

where $\mathbf{R} \in \mathbb{R}^{|\mathcal{S}|}$ is a column vector with each component equal to $\mathbb{E}[R_{t+1}|S_t = s]$. Finally, the norm is weighted by the matrix B: $||\mathbf{y}||_B^2 \stackrel{\text{def}}{=} \mathbf{y}^\top B\mathbf{y}$.

The projection matrix and weighting by the stationary distribution probabilities have important practical consequences. First, the projection matrix projects the points in the value space back into the space of representable functions defined by $\mathbf{X}$. This means the optimal value function, with respect to the MSPBE, is guaranteed to be representable by the function approximator, and that we can achieve MSPBE($\mathbf{w}$) equal to zero even if the features cannot precisely represent the true, un-approximated value function v^π. Second, the MSPBE in each state are scaled by the probability of each state under stationary distribution, induced by the behavior policy. States that are never visited contribute zero error; errors in states that are infrequently visited contribute less to the MSPBE than errors in states visited frequently.

The MSPBE can also be written in terms of expectations (as originally shown by Maei 2011), which is a more useful form to derive incremental learning algorithms:

$$\text{MSPBE}(\mathbf{w}) = \mathbb{E}_\mu[\delta_t \mathbf{e}_t]^\top \mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]^{-1}\mathbb{E}_\mu[\delta_t \mathbf{e}_t]. \tag{2.18}$$

Taking the gradient of the objective function, we get:

$$-\frac{1}{2}\nabla \texttt{MSPBE}(\mathbf{w}_t) = \mathbb{E}_\mu[\delta_t \mathbf{e}_t] - \mathbb{E}_\mu[\gamma(1-\lambda)\mathbf{x}(S_{t+1})\mathbf{e}_t^\top]\mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]^{-1}\mathbb{E}_\mu[\delta_t \mathbf{e}_t],$$

where $\mathbf{e}_t \stackrel{\text{def}}{=} \rho_t(\gamma\lambda\mathbf{e}_{t-1} + \mathbf{x}(S_t))$ (see Maei, 2011, page 72). In order to design an online algorithm for minimizing the MSPBE, we need to sample the gradient, which is challenging due to the product of expectations. We can let the last two terms equal a secondary weight vector $\mathbf{h} \in \mathbb{R}^n$, and sample the remaining terms:

$$\Delta\mathbf{w}_{t+1} \propto \left(\delta_t \mathbf{e}_t - \gamma(1-\lambda)\mathbf{x}(S_{t+1})(\mathbf{e}_t^\top \mathbf{h}_t)\right). \tag{2.19}$$

Maei (2011, page 72) shows that

$$\mathbf{h}_t \propto E_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]^{-1}\mathbb{E}_\mu[\delta_t \mathbf{e}_t] = \mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]^{-1}\mathbb{E}_\mu[\delta_t^{\lambda\rho}\mathbf{x}(S_t)],$$

where $\delta_t^{\lambda\rho} \in \mathbb{R}$ is called the off-policy truncated forward-viewed TD error, defined using future states and rewards. Now, notice that $E_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]^{-1}\ \mathbb{E}_\mu[\delta_t^{\lambda\rho}\mathbf{x}(S_t)]$ is of the same form as a least squares problem: $\mathbf{h} = (X^\top X)^{-1}\ X^\top \mathbf{y}$, where $X = \mathbf{x}(S_t)$ and $\mathbf{y} = \delta_t^{\lambda\rho}$. This means we can form an incremental update rule for $\mathbf{h}_t$ using a least mean squared (LMS) rule (due to Widrow & Stearns, 1985):

$$\Delta\mathbf{h}_{t+1} = \alpha_{\mathbf{h}}\left[(\delta_t^{\lambda\rho} - (\mathbf{h}_t^\top \mathbf{x}(S_t)))\mathbf{x}(S_t)\right],$$

where $\alpha_{\mathbf{h}} > 0$ is a step size parameter. The above LMS rule, however, uses future states and rewards to compute $\delta_t^{\lambda\rho}$, so we exploit that $\mathbb{E}_\mu[\delta_t^{\lambda\rho}\mathbf{x}(S_t)] = \mathbb{E}_\mu[\delta_t \mathbf{e}_t]$ and get an update for the secondary weights:

$$\Delta \mathbf{h}_{t+1} = \alpha_{\mathbf{h}} \left(\delta_t \mathbf{e}_t - (\mathbf{h}_t^\top \mathbf{x}(S_t))\mathbf{x}(S_t)\right).$$

Putting it all together, we get a MSPBE-based, off-policy, value estimation algorithm called gradient TD(λ) or GTD(λ). The following equations specify the algorithm:

$$\begin{aligned}
\mathbf{e}_t &= \rho_t(\mathbf{e}_{t-1}\gamma\lambda + \mathbf{x}(S_t)) \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \left(\delta_t \mathbf{e}_t - \gamma(1-\lambda)\mathbf{x}(S_{t+1})(\mathbf{e}_t^\top \mathbf{h}_t)\right) \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_h \left(\delta_t \mathbf{e}_t - (\mathbf{h}_t^\top \mathbf{x}(S_t))\mathbf{x}(S_t)\right),
\end{aligned}$$

where ρ_t and δ_t are the same as before. This algorithm is due to Maei (2011). The algorithm converges to the minimum of the MSPBE under off-policy sampling, given several technical conditions including that the trace vector, $\mathbf{e}_t$, is bounded (Maei, 2011). The GTD(λ) algorithm, like TD(λ), uses $O(n)$ memory computation per update step.

There is also a gradient temporal difference algorithm for approximating state-action value functions, called GQ(λ), specified by the following equations:

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \mathbf{w}_t \overline{\mathbf{x}}_{t+1} - \mathbf{w}_t^\top \mathbf{x}(S_t, A_t) \\
\mathbf{e}_t &= \rho_t \mathbf{e}_{t-1}\gamma\lambda + \mathbf{x}(S_t) \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \left(\delta_t \mathbf{e}_t - \gamma(1-\lambda)\overline{\mathbf{x}}_{t+1}(\mathbf{e}_t^\top \mathbf{h}_t)\right) \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_{\mathbf{h}} \left(\delta_t \mathbf{e}_t - (\mathbf{h}_t^\top \mathbf{x}(S_t, A_t))\mathbf{x}(S_t, A_t)\right).
\end{aligned}$$

The $\overline{\mathbf{x}}_t$ term denotes the expected next feature vector:

$$\overline{\mathbf{x}}_t \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}} \pi(S_t, a)\mathbf{x}(S_t, a).$$

The GQ(λ) algorithm has been proven to converge under off-policy sampling in the case of linear function approximation (see Maei & Sutton, 2010). The algorithm uses $O(m)$ computation and storage per step, linear in the size of the state-action feature vector. Note that we have omitted the interest function $I : \mathcal{S} \to [0, 1]$, included in Maei and Sutton's (2010) original specification of the algorithm.

There are other reinforcement learning algorithms with convergence guarantees under off-policy sampling and linear function approximation. The first is equivalent to GTD(λ) with λ equal to zero, which we call TDC for historical reasons, and is due to Sutton et

al. (2009). Another algorithm called GTD2 minimizes the MSPBE, but is produced by a different derivation strategy compared to the GTD(λ) algorithm, and does not make use of eligibility traces. The following equations specify the GTD2 algorithm:

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(S_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(S_t) \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_t))(\mathbf{x}(S_t)^\top \mathbf{h}_t) \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_{\mathbf{h}}(\delta_t - \mathbf{h}_t^\top \mathbf{x}(S_t))\mathbf{x}(S_t).
\end{aligned}$$

The original GTD algorithm minimizes a different objective:

$$\text{NEU}(\mathbf{w}) \stackrel{\text{def}}{=} \mathbb{E}_\mu[\delta_t \mathbf{x}(S_t)]^\top \mathbb{E}_\mu[\delta_t \mathbf{x}(S_t)].$$

The algorithm is due to Sutton et al. (2009) and is specified by the following equations:

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(S_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(S_t) \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))\mathbf{x}(S_t)^\top \mathbf{h}_t \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_{\mathbf{h}}(\delta_t \mathbf{x}(S_t) - \mathbf{h}_t).
\end{aligned}$$

The TDC, GTD, and GTD2 algorithms have convergence results, and use computation and storage that is linear in the size of the feature vector. In practice, the empirical results in the literature (Sutton et al., 2009) suggest that TDC is superior to GTD and GTD2 in both Markov chains and computer GO.

Finally, the non-linear versions of GTD2 and TDC converge to a local optimum of a non-linear variant of the MSPBE, enabling, for the first time, convergence of a temporal difference method with smooth value function approximation, such as neural networks.

Two different models have been used over the years to develop off-policy TD algorithms. The GTD(λ) algorithm and other MSPBE minimization methods make use of the *excursion* model for off-policy learning. In this model, the state-value function is defined as the expected discounted return of executing the target policy from the current state. For example, if the behavior policy were random and the target policy were `go-to-nearest-wall`, then the value function would estimate the expected, discounted sum of rewards if the agent ran the `go-to-nearest-wall` policy, in the current state. Prior work on off-policy TD algorithms used the *alternative-life* model, where the value function is defined as the expected, discounted sum of rewards, assuming the agent had been following the target policy since the beginning of time. In practice, the algorithms derived under the alternative-life model exhibit large variance (demonstrated by Precup et al., 2001; Precup et al., 2006; Rafols, 2006).

2.6 Computing the MSPBE

Given the parameters of the MDP, we can compute the MSPBE for the weight vector learned by a gradient TD algorithm. This can be useful for experiments. The following describes how we arrive at our final equation for the MSPBE.

We begin with an expansion of $\mathbb{E}_\mu[\delta_t \mathbf{e}_t]$:

$$\begin{aligned}
\mathbb{E}_\mu[\delta_t \mathbf{e}_t] &= \sum_s d_\mu(s) \mathbb{E}_\mu[\mathbf{e}_t(r_{t+1} + (\gamma \overline{\mathbf{x}}_{t+1} - \mathbf{x}(S_t))^\top \mathbf{w}_t) | S_t = s] \\
&= \sum_s d_\mu(s) \mathbb{E}_\mu[\mathbf{e}_t | S_t = s] \mathbb{E}_\mu[r_{t+1} + (\gamma \overline{\mathbf{x}}_{t+1} - \mathbf{x}(S_t))^\top \mathbf{w}_t) | S_t = s] \\
&= \sum_s d_\mu(s) \mathbb{E}_\mu \left[\sum_{m=-\infty}^{t} (\gamma\lambda)^{t-m} \mathbf{x}(S_m) | S_t = s \right] \cdot \\
&\qquad \mathbb{E}_\mu \left[r_{t+1} + \left(\gamma (\sum_{s' \in \mathcal{S}} P^\pi(s_t, s') \mathbf{x}(s')) - \mathbf{x}(S_t) \right)^\top \mathbf{w}_t | S_t = s \right] \\
&= \sum_s d_\mu(s) \mathbb{E}_\mu \left[\sum_{m=-\infty}^{t} (\gamma\lambda)^{t-m} \mathbf{x}(S_m) | S_t = s \right] \mathbf{1}(S_t = s)^\top (\mathbf{R} + \gamma P^\pi \mathbf{X}\mathbf{w}_t - \mathbf{X}\mathbf{w}_t).
\end{aligned}$$

Let $\overline{\mathbf{x}}_t \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}} \pi(S_t, a) \mathbf{x}(S_t)$. Now, we can simplify the eligibility trace using matrices. Considering the sum over states before S_t:

$$\begin{aligned}
&\mathbb{E}_\mu \left[\sum_{m=-\infty}^{t} (\gamma\lambda)^{t-m} \mathbf{x}(S_m) | S_t = s \right] \\
&= \mathbf{x}(s) + \gamma\lambda \sum_{s_{t-1} \in \mathcal{S}} P^\pi(s_{t-1}, s) \mathbf{x}(s_{t-1}) \\
&\qquad + (\gamma\lambda)^2 \sum_{s_{t-1} \in \mathcal{S}} P^\pi(s_{t-1}, s) \sum_{s_{t-2} \in \mathcal{S}} P^\pi(s_{t-2}, s_{t-1}) \mathbf{x}(s_{t-2}) + \dots \\
&= \mathbf{X}^\top \mathbf{1}(S_t = s) + \gamma\lambda (P^\pi \mathbf{X})^\top \mathbf{1}(S_t = s) + (\gamma\lambda)^2 ((P^\pi)^2 \mathbf{X})^\top \mathbf{1}(S_t = s) + \dots \\
&= \mathbf{X}^\top (\sum_{m=0}^{\infty} (\gamma\lambda)^m (P^\pi)^m)^\top \mathbf{1}(S_t = s) = \mathbf{X}^\top (I - \gamma\lambda P^\pi)^{-\top} \mathbf{1}(S_t = s).
\end{aligned}$$

Since $\mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top$ is simply a matrix with precisely a single one on the diagonal corresponding to state s, $(I - \gamma\lambda P^\pi)^{-\top} \mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top$ selects the row in $(I - \gamma\lambda P^\pi)^{-\top}$ corresponding to s. Similarly, $\mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top (I - \gamma\lambda P^\pi)^{-1}$ selects the column in $(I - \gamma\lambda P^\pi)^{-1}$ corresponding to s. Therefore, $(I - \gamma\lambda P^\pi)^{-\top} \mathbf{1}(S_t = s)\mathbf{1}(S_t =$

$s)^\top = \mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top (I - \gamma\lambda P^\pi)^{-1}$, and we obtain:

$$\begin{aligned}
\mathbb{E}_\mu[\delta_t \mathbf{e}_t] &= \sum_s d_\mu(s)\mathbf{X}^\top (I - \gamma\lambda P^\pi)^{-\top}\mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top(\mathbf{R} + \gamma P^\pi \mathbf{X}\mathbf{w}_t - \mathbf{X}\mathbf{w}_t) \\
&= \sum_s \mathbf{X}^\top d_\mu(s)\mathbf{1}(S_t = s)\mathbf{1}(S_t = s)^\top (I - \gamma\lambda P^\pi)^{-1}(\mathbf{R} + \gamma P^\pi \mathbf{X}\mathbf{w}_t - \mathbf{X}\mathbf{w}_t) \\
&= \mathbf{X}^\top B(I - \gamma\lambda P^\pi)^{-1}(\mathbf{R} + \gamma P^\pi \mathbf{X}\mathbf{w}_t - \mathbf{X}\mathbf{w}_t) \\
&= \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}\mathbf{R} - \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}(I - \gamma P^\pi)\mathbf{X}\mathbf{w}_t. \qquad (2.20)
\end{aligned}$$

Now using the expectation-based defintion of the MSPBE($\mathbf{w}$), we get:

$$\text{MSPBE}(\mathbf{w}_t) = (-A^\pi \mathbf{w}_t + \mathbf{b}^\pi)^\top C^{-1}(-A^\pi \mathbf{w}_t + \mathbf{b}^\pi)$$

where $C = \mathbf{X}^\top B\mathbf{X}$, $A = \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}(I - \gamma P^\pi)\mathbf{X}$, and $b = \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}\mathbf{R}$. Now, given access to P^π, γ, and R, we can compute the MSPBE for our algorithms.

2.7 Recent algorithmic advances

During the development of this document, several reinforcement learning algorithms have been improved in non-trivial ways. These new algorithms could likely be integrated into my learning systems without much trouble.

One of these new algorithmic developments involves methods for computing eligibility traces in TD learning. The derivation of the linear TD(λ) algorithm uses the theoretical constructs of *forward* and *backward views* (see Sutton and Barto, 1998 for additional details). The forward view specifies how a learning algorithm should update the value function on each time-step using information about future rewards, from the current time-step. To generate incremental algorithms that do not rely on future data, the forward view is converted into a backward view, which uses data from the past to make updates on each time-step using the eligibility traces, $\mathbf{e}_t$. It was previously believed that this conversion from forward to backward views could only be approximate (Sutton & Barto, 1998), and thus the linear TD(λ) algorithm only approximately implements the corresponding forward view. Recently, however, it has been shown that this conversion can be made equivalent in the interim (see van Seijen & Sutton, 2014). The new conversion yields a new linear TD(λ) algorithm called *True Online TD(λ)* and a new form of eligibility. True Online TD(λ) appears to always equal or outperform linear TD(λ), and the control version, true online Sarsa(λ), shows great promise as well.

New True Online off-policy learning algorithms have also been recently introduced (van Hasselt, Mahmood & Sutton, 2014). Empirical studies show that these new off-policy

algorithms can out-perform the GTD(λ) algorithm in some simulation tasks. These new off-policy algorithms use computation and storage that is still linear in the size of the feature vector. It will be intriguing to investigate if these new algorithms result in noticeable improvements in sample efficiency or prediction accuracy on robot data.

2.8 Learning policies

The work in this thesis almost exclusively focuses on the case of value function estimation or prediction learning. However, reinforcement learning methods are perhaps best known for learning policies that maximize reward, like finding the shortest path in a maze. Temporal difference value-function learning methods can be used iteratively, in concert with a policy improvement step, to improve the agent's policy over time. This process is called generalized policy iteration, and like value function estimation methods, these algorithms come in two flavors: on- and off-policy. On-policy, TD-based policy iteration methods include the well-known Sarsa(λ) algorithm (Rummery, 1995), and the expected Sarsa(λ) algorithm (van Seijen et al., 2009). Off-policy, policy iteration methods find a policy to maximize reward, while selecting actions according to some other, possibly exploratory, behavior policy. Q-learning (Watkins, 1989) is an example of an off-policy, TD-based, policy iteration algorithm, and is perhaps the best known algorithm for reinforcement learning. Unfortunately, the update of the Q-learning algorithm can diverge (see Baird, 1995 and Boyan & Moore, 1995). A policy learning variant of the GQ(λ) algorithm, called greedy-GQ(λ) (Maei et al., 2010b), provides some convergence guarantees and solves Baird's Q-learning counterexample. This thesis includes one demonstration of off-policy policy learning on a robot in Chapter 6, while the rest of our results and demonstrations focus on prediction learning for fixed behavior policies.

2.9 Options

An option is a generalization of actions to include extended courses of action (Sutton, Precup & Singh, 1999). An option, denoted by o, is defined by a tuple $\langle \pi, \beta, \mathcal{I} \rangle$, where π is called the option policy, $\beta : \mathcal{S} \to [0, 1]$ is the termination function where $\beta(S_t)$ denotes the probability of π terminating in state S_t, and $\mathcal{I} \subseteq \mathcal{S}$ is the set of states in which the option policy is applicable. A MDP with options included in the action set becomes a discrete time, semi-MDP, which is like an MDP with variable-length time steps. An option is like a closed-loop control rule that enables *temporal abstraction*: abstraction of action rather than

state.

For example, one might define a `pass-through-door` option. In this case, π selects the forward action in every state, β equals one once the agent passes through a doorway, and I includes all states where the agent is facing toward the doorway. Every primitive MDP action is also an option. For example, the hypothetical *up* action is an option, where $\pi(s, \text{up}) = 1$ and $\beta(s) = 1 \; \forall \; s \in \mathcal{S}$ (indicating certain termination after one step), and $\mathcal{I} = \mathcal{S}$.

An agent can use options in several different ways. Given a set of options, the agent can learn an approximate state-action value function and policy over a mixture of MDP actions and options policies. The agent can also learn the transition and reward models of an option. The transition model gives the probability than the option policy will terminate at any other state (given an input state) and the reward model gives of the expected total reward over the option policies' execution. The agent can update an option value function and option model, without ever executing the option's policy to termination, using off-policy learning methods. Finally, the agent can learn the policies inside options, learning how to execute an option to a achieve a subgoal, defined by a special subgoal terminal reward.

2.10 Summary

This chapter covered basic concepts in reinforcement learning, which will be useful for understanding the rest of this document. Our survey covered the reinforcement learning problem formulation and MDPs. We introduced value functions, linear function approximation, and described algorithms for learning value functions from an agent's interaction with the world. We discussed the off-policy learning problem, divergence of TD methods, the MSPBE objective function, and the family of gradient TD methods for learning value functions from off-policy samples. We closed the chapter with a brief overview of policy learning algorithms, recent algorithmic advances in reinforcement learning, and option models.

Chapter 3

Sensorimotor data streams and robots

In this chapter, we describe the target of our predictive knowledge learning approach: learning predictions from and about low-level data produced by robots. We describe two robots capable of generating data suitable for our efforts, which are used in the prediction learning experiments throughout this document. We finish the chapter with an investigation of the data stream produced by one of our robots.

In this thesis we use mobile robots as a platform for demonstrating how our approach improves the applicability and scalability of predictive knowledge learning. Our robots are capable of generating multi-dimensional sensor vectors of continuous numbers, multiple times a second, producing data streams that are longer and of higher dimension than any used in prior investigations of predictive knowledge learning. The main aim of this chapter is to introduce these robots and give the reader an idea of what these data streams look like.

3.1 Learning about sensorimotor data

We follow in the footsteps of previous work on predictive representations of state and temporal difference networks, using low-level interaction data as the target of learning in our approach to predictive knowledge. Specifically, the data will be a stream of instantaneous, uninterpreted sensor readings (including low-level motor commands) generated by a mobile robot interacting with a real physical environment. This sensorimotor data is low-level, fine-grained, and rapidly changing. Our agent will seek to predict the sensorimotor data itself—future values of its sensor readings—and not things external to the data stream such as the robot's $x - y - \theta$ position in 3D space. In Chapter 4 we will more precisely formalize how we form predictions about future sensorimotor data. In addition, the agent's learning

will be exclusively based on the sensorimotor data stream, and its predictions will not rely of constructs with special external semantics. For example, our agent cannot use human defined hidden states or motion models, but we do allow feature vectors generated by an algorithm because the features can be used without external interpretation. These restrictions are enforced to guarantee that the contents of the agent's knowledge can always be checked for correctness by comparing the predictions with the data stream.

We seek to give our agent access to the robot's hardware at the lowest level reasonably possible. The nature of how an agent's action choices are manifest in observed consequences in the world relates to the difficulty of the agent's prediction task. If there is significant delay between action selection and observed consequences, then learning can be more challenging, as longer delays are more difficult to predict. If the agent's actions are converted into motor commands by some complex subsystem, then accurately predicting the effects of actions may involve modeling the complex action processing procedure. Of course delayed and unexpected action consequences are a fact of life. We want to learn about the consequences of the robots' interaction with the world, and reduce unnecessary preprocessing and internal abstractions.

We are not the first to attempt to learn predictive knowledge from data produced by a robot (see Boots et al., 2010), but we are the first to attempt to learn in realtime while the robot is operating. The sensorimotor data is collected and presented to our agent at a semi-regular cycle time. The variability in the cycle time is primarily due to the variability in wireless communication, when it is used. From the agent's perspective, the cycle length is strictly enforced. The agent must process the sensory data, perform learning updates, and generate a new motor command (its action) within the cycle time. If the agent fails to select a new action in time, the robot executes the most recent command issued by the agent and delivers a new batch of sensor readings to the agent. Our learning agent learns its predictions online.

The next two sections describe our two robot platforms. One is a commercial product based on the Roomba, and the other is a custom built robot that enables low-level hardware access and fast cycle times.

3.2 The iRobot Create

The iRobot Create is a commercially produced robot based on the Roomba vacuum robot. The Create can run multiple times a day, for weeks on end, with inconsequential wear and

tear. This robustness is partially due to the low speed wheelchair drive system. Bumping into objects at maximal velocity results in no apparent damage, although the bumpers eventually wear out. Damaging motor overheating is rare. Only once did we observe some melted body plastic due to the robot driving into a corner for over an hour (same action while staying in place). The drive system of the Create can repeatedly perform the same forward and backward movements reliably with little translational variance, even over a couple of meters. The Create's built-in `find-charger` routine can autonomously find and connect with its charger using a simple infrared sensor and charging station infrared beacon. The update cycle of the Create can be as fast as 30 ms. Communication with the robot can be wireless via bluetooth to a nearby computer, or hard wired using a small computer, such as a Raspberry Pi connected to the robot's serial port and battery. Like many mobile robots, the Create experiences wheel slippage, communication interruptions, and occasional erratic sensor readings. This robot is robust, reliable, and simple to program.

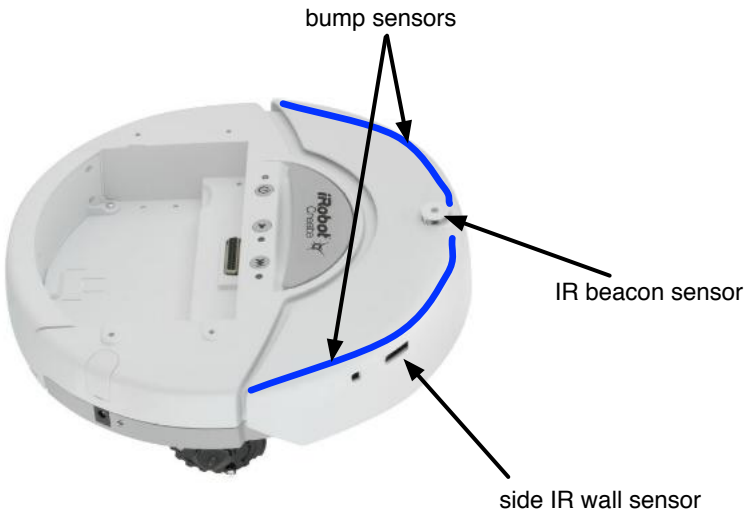

Figure 3.1: The top-view of an iRobot Create. This small wheelchair drive robot can rotate and move forward and backward at a maximum speed of 0.5 meters per second. The front left and right bump sensors are marked with blue on the figure, and report binary values. An additional virtual sensor provides a binary indication if both bump sensors are activated. The side-facing infrared (IR) sensor reports the nearby obstacles (approximately two inch range) as an integer from zero to 255. The IR beacon sensor reports integer values from zero to 255, and is used to detect IR emittance from the robot's charger and button presses on the Create's remote control.

The Create has limited sensing abilities. The robot's sensors include nine external readings and basic wheel odometry, described in Figures 3.1 and 3.2. The Create will report unique sensor readings when it is beside a wall, bumping an object, near its charger, and

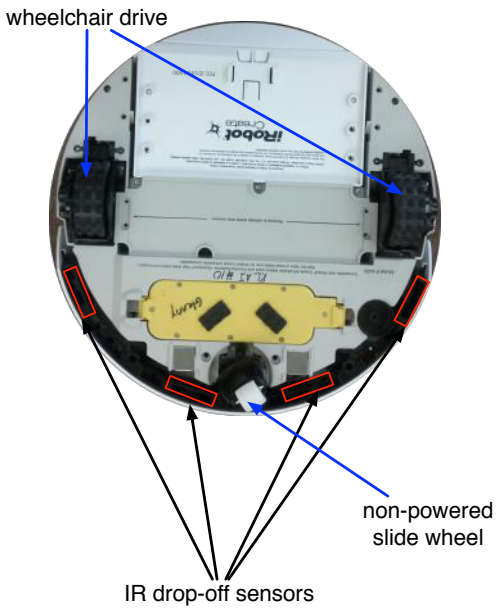

Figure 3.2: A view of the underside of the iRobot Create. Two powered wheels and a rotating non-powered slide wheel comprise the wheelchair drive system. The drive velocity, measured in wheel rotations, and rotation direction of each of the two powered wheels, are reported by the robot. The four downward facing IR cliff sensors are located along the front bumper of the robot and report integer values between zero and 255.

near a drop-off in the floor (e.g., top of the stairs). Using only its sensor readings, the Create cannot tell where it is, once it loses sensor contact with the walls and is outside its charge beacon's range. Although this robot's sensor set seems ill-suited for conventional robot tasks, such as mapping and navigating an unknown environment, there are many predictive questions we can pose and learn from and about the Create's sensorimotor data stream.

3.3 The Critterbot

We built a custom robot designed specifically for learning about the sensorimotor stream of a robot.[1] This robot is called the RLAI Critterbot, and can be seen in Figure 3.3. The majority of our experiments with the Critterbot and the Create were conducted inside a small pen. The pen is a two meter by two meter box with short walls (about the hight of the Critterbot), and either smooth or carpeted floor (it can be easily changed). The Critterbot's charger is located in one corner of the pen. The pen (1) reduces annoyance to the humans occupying the lab, (2) prevents potentially dangerous unplanned human-robot interaction, and (3) limits the complexity of the robot's environment, enabling interesting

[1]The Critterbot was designed by Richard Sutton and Mike Sokolsky, and constructed by Mike Sokolsky.

but constrained experiments. All our experiments with the Critterbot were run on a laptop computer communicating with the robot in realtime over a wireless link.

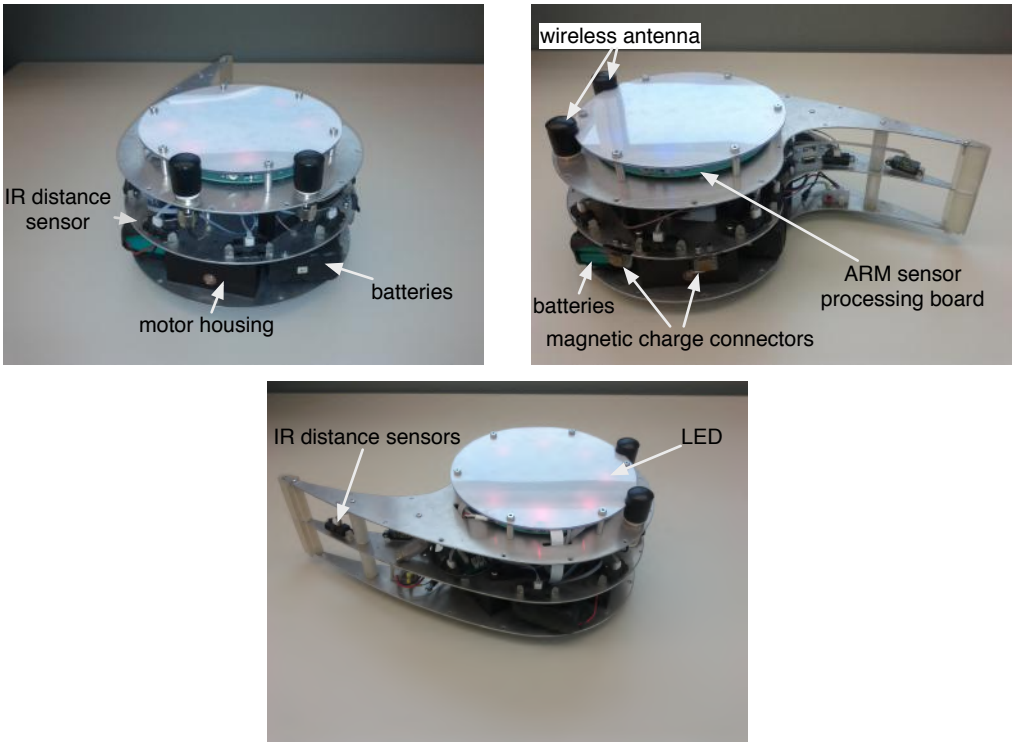

Figure 3.3: Side and front views of the Critterbot. The labels indicate the locations of several IR distance sensors, the charger connection points, two of the robot's three batteries, a motor drive housing, the sensor board responsible for polling the sensors, the wireless antenna, and the ring of LED lights on top of the robot used for basic communication. The Critterbot is composed of three plates, and each support a computing board. The top-plate contains an arm processor and the light, thermal, acceleration, magnetic, and gyroscope sensors. The middle plate contains the IR distance sensors and a Linux computer. The bottom plate contains the three motor housing, three motor controllers, and three batteries. An external skin (not shown) can be attached to prevent piercing the internals of the robot.

The Critterbot is outfitted with a reasonably large, heterogeneous sensor set. The robot has five groups of external sensors, and six groups of internal sensors. The external sensors measure distance via IR reflectance distance (ten), ambient light (four), thermal energy (eight), magnetic field strength (three-axis) and IR light (eight). The external sensors, excluding the magnetic sensor, are arranged in circular pattern around the robot to provide sensing in multiple directions. Internally, the robot reports its power source, charge state, bus voltage, acceleration (three-axis), rotational velocity, and wheel and motor information (motor current, motor command, motor speed, and motor temperature) for each of the

three motors. In all, 52 sensor readings are pooled and made available every 10 milliseconds. Figure 3.3 shows the locations of several sensors in the robot, and Figure 3.4 gives an overhead view of the sensor layout.

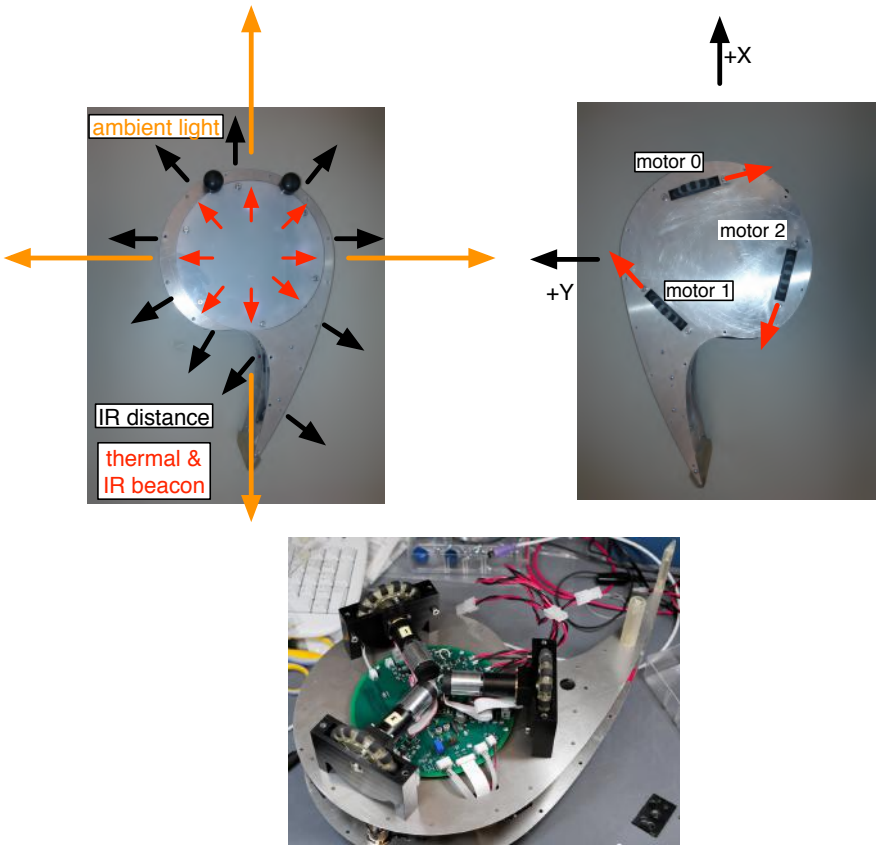

Figure 3.4: A view top and bottom of the Critterbot. In the left figure we can see how the IR distance, light, thermal, and IR light sensors are arranged. The right figure shows the robot's wheel layout, and the frame of reference for robot control. Three wheels are offset by 120 degrees and each wheel contains many rollers. One wheel can act as a slider while the other two wheels rotate for translation motion. Commands can be sent using one of three modes. In wheel velocity control mode, the desired wheel velocity of each wheel is sent to the motor controllers and achieved (if possible by PID control). In robot velocity or X-Y-θ mode, the command is specified as a forward velocity (+/-) in the X direction, sideways velocity in the Y direction, and rotational velocity or θ. In voltage mode, each dimension of the command specifies a voltage sent to each wheel. Voltage mode, aside from overheating control, is the only mode that does not use an intermediate controller, and thus is the lowest level of control possible on the Critterbot. The bottom-most figure shows the motor controller board, the motors, and wheels in detail.

The Critterbot sensors provide a kind of awareness and sense of location in the pen. The Critterbot reports unique sensory information when it is impacted, lifted off the ground, flipped over, pushed across the ground, pushing a heavy object, or driving on different

surfaces. External to itself, the robot reports when it is near a large magnetic or heat source, an object like a wall, when it is connected to its charger, and when lights are turned on. The Critterbot might even perceive large vibrations on its accelerometer from a heavy object dropped on the floor or a heavy footed person nearby. Equally, Figure 3.5 illustrates how the robot's sensors can help distinguish many interesting situations.

The Critterbot's was designed to be robust to physical interaction with other objects in the world. The robot is assembled from three aluminum plates locked together with a steel frame to protect sensory equipment and on-board computers from collision. The holonomic drive system allows the robot to rotate in place and translate in a number of directions regardless of the robot's initial pose. Commands can be sent to the robot as low-level motor currents, desired wheel velocity (converted into motor currents by a proportional-integral-derivative [PID] controller), or in a X-Y-θ mode (also by a PID). The robot is shaped like a comma with a a rigid tail to facilitate basic object interaction, such as moving or shooting a ball.

The Critterbot's translational movements, in X-Y-θ control, have greater variance in comparison to the Create's movements. This is partially due to the differences between wheelchair drives and omni-drives, but also due the Critterbot's wheel design. The wheels of the Critterbot are 3D printed, and due to the strain of motion, warp fairly quickly. The consequence of this is slightly curved translational movements (that change over time), and many hours of the authors time spent printing and installing new wheels.

The Critterbot was designed to run for many hours a day and for multiple days on end. The robot's charger emits a IR pulse at 50 times per second, which is visible to the robot through its IR light sensors. This pulse can be used to determine the position of the robot relative to the charger, facilitating autonomous connection and recharging. The robot does not sleep during charging, with all sensors continuing to report while the robot is connected to the charger. On average, the Critterbot can run for six to eight hours continuously, depending on its activity level and motor speeds. Figure 3.4 provides a view of the Critterbot's omni-wheels and drive system.

3.4 Critterbot sensorimotor data

The aim of this section is twofold. The first is to give the reader a sense of what the sensorimotor data stream of the Critterbot looks like. Our secondary aim is to discuss why the Critterbot's sensorimotor stream might be useful for developing our predictive approach

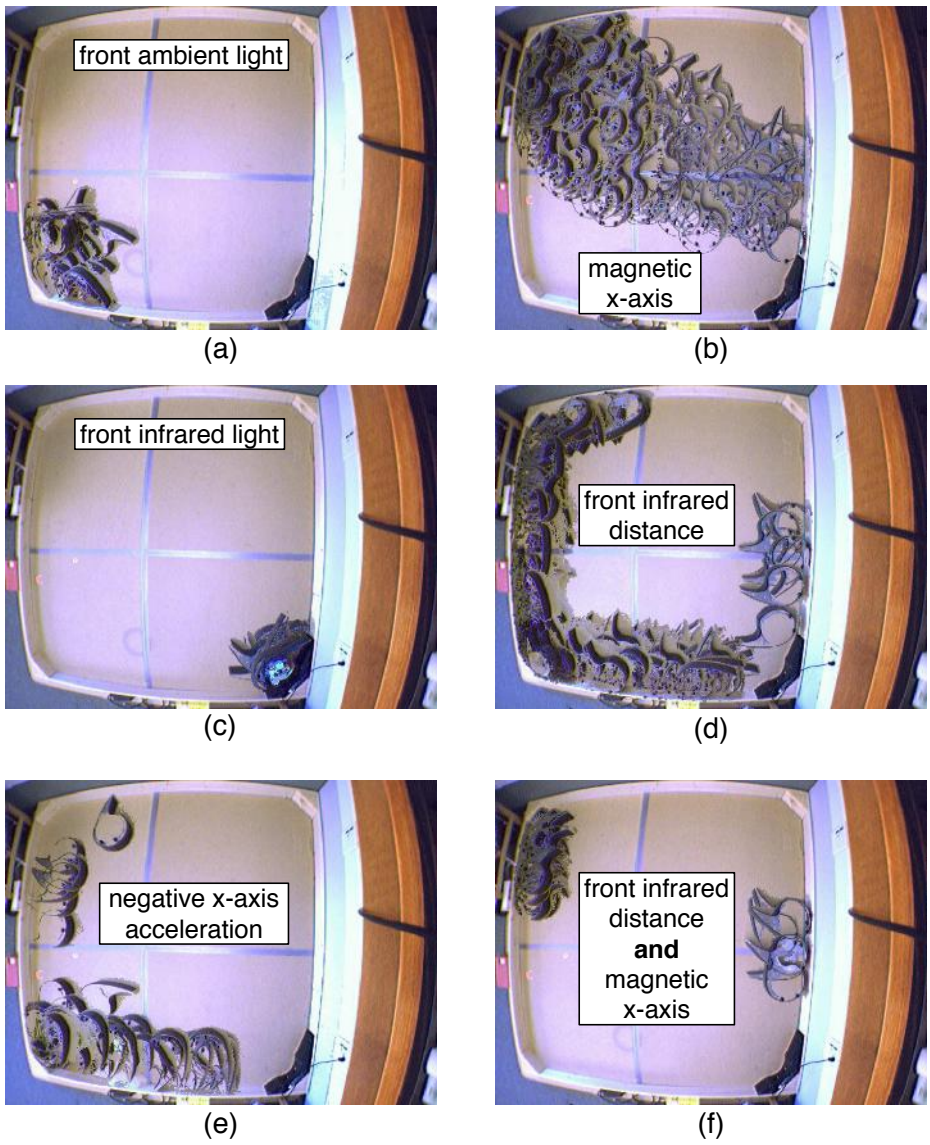

Figure 3.5: These figures provide a sense of the distinctiveness of the Critterbot's sensor readings in a variety of orientations and locations in the pen. The Critterbot was photographed once per second, on one day, for over an hour while moving about the pen. Using the batch of photos and the corresponding sensor-log, we overlay the pictures of the robot corresponding to different sensor ranges. The images above show six such compositions. (a) The front ambient light sensor reading greater than 200 corresponds to the corner of the pen, facing toward the wall: the brightest location in the pen. (b) A low reading on the magnetic x-axis sensor was recorded in the middle of the pen. (c) Highest infrared light was recorded when the robot faced the dock. (d) The front IR distance sensor was near its max when the robot was pressed against the wall. (e) Large negative accelerations occurred when the robot impacted the wall with its tail during a backward motion. (f) A compositional condition where front infrared-distance was high and magnetic x-axis reading was low corresponds to facing the wall in two distinct locations in the pen: an intersection of image (b) and image (d).

to knowledge.

The Critterbot can facilitate the generation of an enormous amount of data. Assuming a 100 ms update cycle, the Critterbot can produce over 19 million sensor readings an hour, over 150 million readings a day (assuming 8 hour runs), and 5.57×10^{10} readings a year—that is over 222 Gigabytes of raw sensor data a year. We can get a sense of the scale and complexity of the data generated by the Critterbot by looking at a snapshot over several seconds in Figure 3.6, and over an entire year (see Figure 3.7). The Critterbot can produce sufficient data for updating and evaluating a large number of predictions.

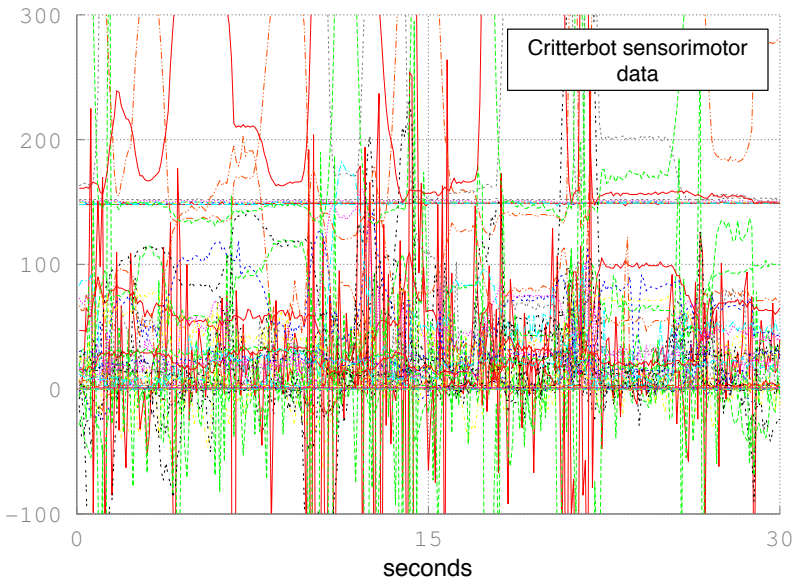

Figure 3.6: A plot of the sensorimotor data stream produced by the Critterbot. Fifty sensor readings are plotted over several seconds, sampled every 100 ms. Several sensor readings, such as the thermal readings, are well off the scale.

The sensorimotor data produced by the Critterbot exhibits different regularities depending on how the robot's actions are selected. For instance, if the robot executes persistent movements as in Figure 3.8, then the sensor readings change in expected, certainly non-random ways. The data in Figure 3.8 suggest that our agent might hope to predict the consequences of these basic movements. In Figure 3.9, we can see that different policies produce very different sensorimotor data streams.[2] Sometimes the data exhibits repeated patterns, such as the light and magnetic readings from the wall-following policy. Sometimes there appears to be almost no regularity regardless of the policy, as seen in the rotational velocity plots. The data exhibits predictable regularity over seconds (e.g., Figures 3.9 and

[2]The Brownian motion dataset was collected by Joseph Modayil.

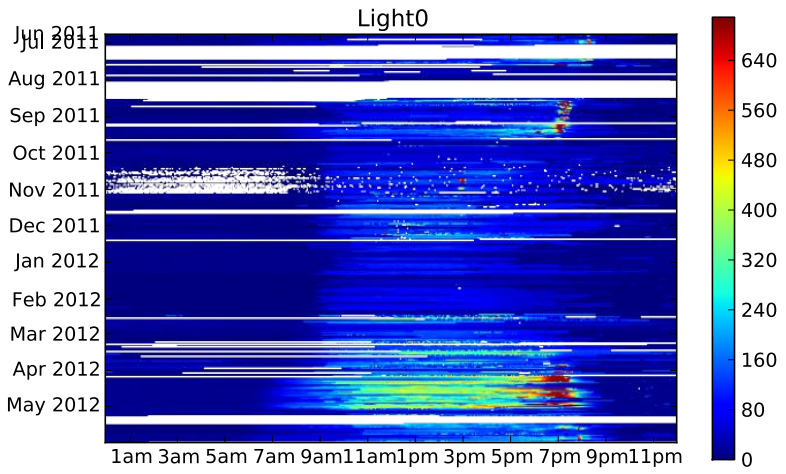

Figure 3.7: A summary of one year's worth of ambient light readings from the Critterbot. The heat map shows the light intensity with time of day on the x-axis and month of year on the y-axis, displaying one reading every ten minutes. Bright red-orange indicates high ambient light, and dark blue indicates low light. White indicates no data was available, typically when the robot was broken and powered down. Notice the seasonal change in the light intensity over the months. There is a faint light reading during most nights, corresponding to the LED lights flashing during charging.

3.8), days, and years (e.g., Figure 3.7). There are a great diversity of things to predict in the sensorimotor stream of the Critterbot, and those things are potentially predictable and exhibit non-random and non-trivial variations.

3.5 Summary

In this chapter we introduced the target of our predictive knowledge learning approach: learning about low-level sensorimotor data produced by a robot interacting with its world. We described the two robot platforms that will be used in the experiments throughout this thesis: the iRobot Create and the RLAI Critterbot. Finally, we provided several plots of the sensorimotor stream generated by the Critterbot, under several different polices, along the way highlighting the potential benefits of predicting a sensorimotor data.

The hardware platforms introduced in this chapter are useful for demonstrating our contributions to predictive knowledge learning. The majority of prior work on predictive knowledge (e.g., Drescher, 1991; Sutton, 2009; Sutton, 2011) has focused on small, discrete simulation worlds, with the notable exception of the work of Boots et al. (2010) on predictive representations of state. The robots described in this chapter, are capable of generating dozens of sensor readings many times per second for hours, and potentially days, on end. The ability to accurately predict data of this kind would represent a development in both the

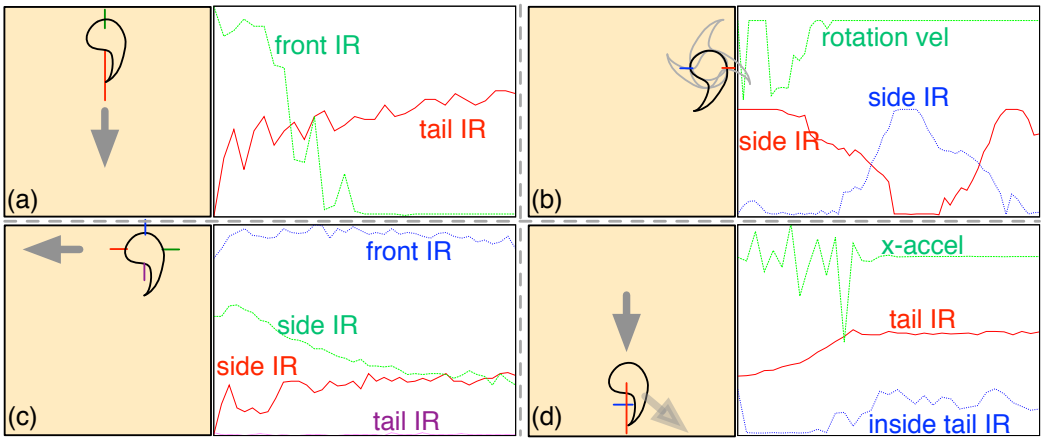

Figure 3.8: This figure shows variations in sensor readings from the Critterbot during persistent movements in its pen. The lefthand side of each subfigure provides a cartoon illustration of what the robot was doing, and the righthand side plots several sensor readings that occur over the course of the movement. (a) The robot executed a southward translation, with the front of the robot facing the north wall of the pen. Correspondingly, the front-facing, IR distance reading decreased while the back-facing, tail IR reading increased. (b) A counter-clockwise rotation near the east wall. The robot initially contacted the wall, reducing the rotational velocity (which pushes the robot westward), and then the robot freely rotated at a constant velocity completing a full rotation. (c) A westward translation while facing the north wall. The front and rear IR distance readings remained within a bounded range. The right-side IR reading dropped slowly, while the left side IR reading slowly increased. (d) A northward translation, followed by a wall impact which caused a negative spike in acceleration and then near zero acceleration. After impact, the robot continued pushing against the wall causing a slow counter-clockwise rotation and a small increase in the inside-tail, IR distance reading.

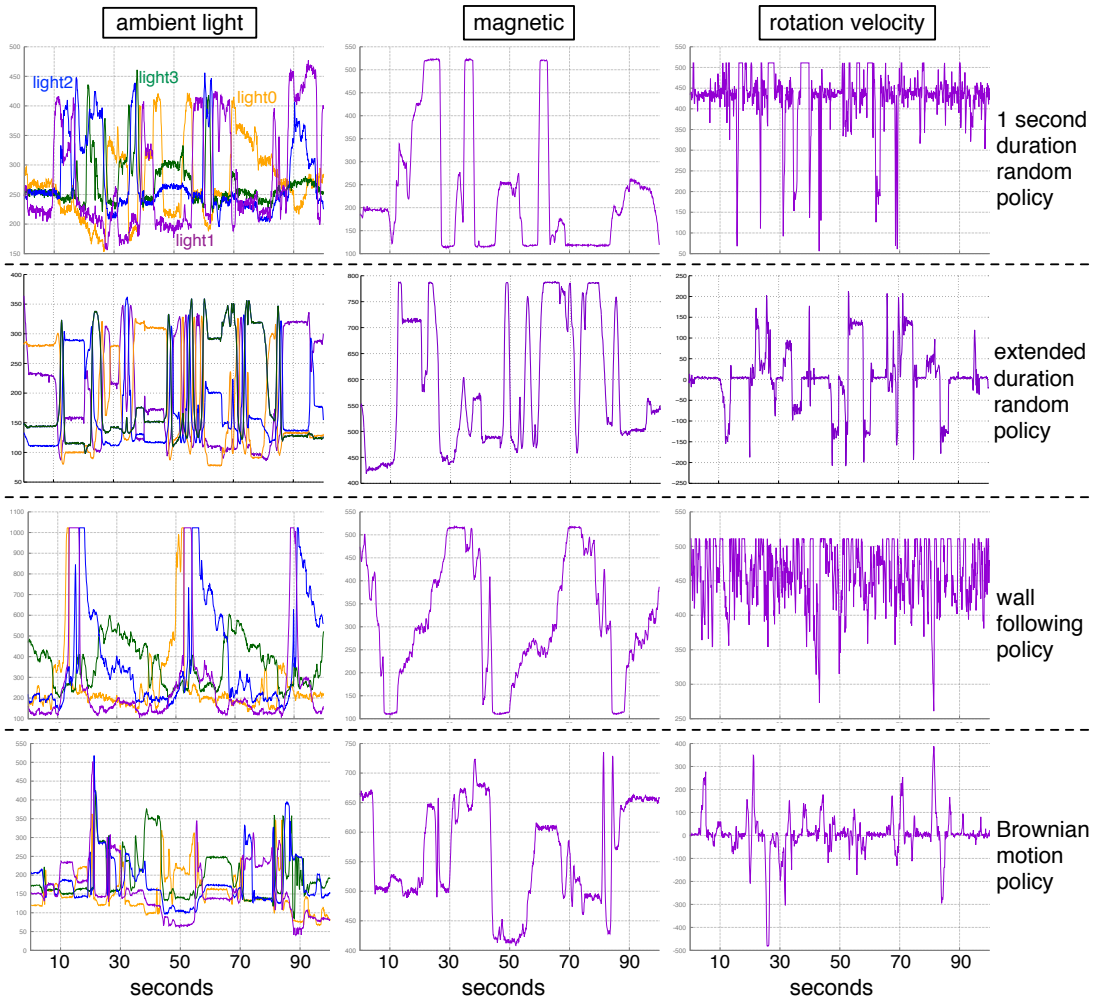

Figure 3.9: This figure describes how the action selection mechanism can affect the sensor patterns observed on the Critterbot. Each row above corresponds to a different policy, including the *one-second duration random* actions policy, *extended duration random* actions policy (selecting the same action for 50 consecutive time steps), hand-coded *wall following* policy, and random walk in motor-voltage space via *Brownian motion*. Each column corresponds to one of three different sensors of interest, including ambient light (all four), magnetic x-axis, and rotational velocity. All four policies selected actions every 100 ms. The wall following policy yielded roughly cyclic patterns, while the data from the one-second duration random action policy did not yield obvious patterns. The wall following policy rotation data exhibited many small adjustments to keep the robot parallel to the wall, and periodic large changes corresponding to each corner of the pen. In summary, different methods for selecting actions can produce noticeably different sensorimotor data streams on the Critterbot.

applicability and scalability of predictive knowledge learning. The next chapter describes our approach to learning predictive knowledge from and about a robot's sensorimotor data stream.

Chapter 4

General Value Functions[1]

This chapter describes general value functions (GVFs), our proposed approach to representing predictive knowledge. The main idea is to adapt conventional value functions from reinforcement learning in order to (a) represent potentially useful predictive knowledge about the world, and (b) leverage the strengths of conventional value function learning methods.

Compared to other predictive knowledge systems and other well-developed uses of prediction, the relative strength of GVFs is the practicality and scalability of learning. Considerable progress has been made in demonstrating the potential compactness of predictive representations of knowledge (Littman, Sutton, & Singh, 2000; Talvitie & Singh, 2011), learning compositional predictions (Sutton & Tanner, 2005), learning temporally abstract predictions (Sutton et al., 2006), and learning predictive knowledge from continuous inputs (Boots et al., 2010). Our GVFs build on prior work, and make an important step towards improving both the applicability of prediction specification and the scalability of prediction learning via value function estimation algorithms from reinforcement learning. Consequently, GVFs are learnable from continuous-valued inputs, such as robot sensors, and can be incrementally updated online and in parallel. In addition, GVFs can represent a broad class of predictive knowledge similar to PSRs (Littman, Sutton, & Singh, 2000), TD-nets (Sutton & Tanner, 2005), TPSRs (Boots et al., 2010), and prediction profile models (Talvitie & Singh, 2011).

The focus of this chapter is to introduce and explain GVFs. We demonstrate the potential benefits of representing predictions as GVFs in later chapters. The contributions of this chapter are a description of how GVFs may represent predictive knowledge and a discussion of how GVFs relate to the literature. We begin by formalizing the setting under which

[1] General value functions were first introduced by Maei & Sutton (2010), and proposed as an approach for predictive knowledge in two papers coauthored by this author (Sutton et al., 2011; Modayil, White & Sutton, 2014). The formalisms and text contained in this chapter are either adapted from co-authored sections of these two papers or originally composed for this document.

we propose to learn GVFs. The second part of this chapter describes GVFs and how multiple GVFs may be organized and learned in parallel. The third and final part of this chapter summarizes related work, highlighting the similarities and points of distinction compared to GVFs.

4.1 The setting

The algorithmic and experimental work described in this thesis apply to a simple interface depicted in Figure 4.1, similar to the agent-environment interaction from reinforcement learning. On each discrete time-step $t = 1, 2, ...$, the agent observes a new feature vector and action. The *feature vector* is a real vector of n elements, where $\mathbf{x}_t \in \mathbb{R}^n$ denotes the feature vector at time instance t. The *action* is an integer in a finite set: $A_t \in \{1, 2, ..., k\}$.

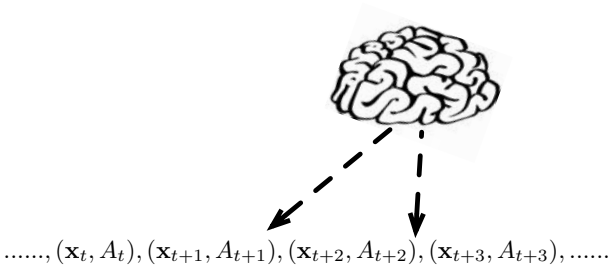

Figure 4.1: The learning setting: the agent passively observes an unending stream of feature vectors and discrete actions.

The objective of our learning system is to estimate a scalar signal $G_t \in \mathbb{R}$, called the *target*. We consider the case in which a new estimate is made repeatedly, at regular intervals, on each time-step. The estimate of the target is called the agent's *prediction*, and is denoted by $V_t \in \mathbb{R}$. The extent to which V_t matches G_t defines the accuracy of the agent's prediction. The target may be thought of as specifying a *question* about the agent's interaction with the world, and the agent's prediction is the agent's *answer* to the target's question.

The target is a summary of the future. The target is computed from two other signals. The first is called the cumulant $Z_t \in \mathbb{R}$, and plays a similar role to reward as used in reinforcement learning. As the name suggests, the target accumulates future values of the cumulant. The cumulant signal is weighted by a termination signal $\gamma_t \in [0, 1]$, that we previously defined as the discount factor in Chapter 2. The termination signal has a different interpretation compared to the discount factor in reinforcement learning, and will be explained in the following section. G_t is computed from the future values of the cumulant

and termination signal, beyond the current time-step:

$$G_t = Z_{t+1} + \gamma_{t+1} Z_{t+2} + \gamma_{t+1}\gamma_{t+2} Z_{t+3} + \gamma_{t+1}\gamma_{t+2}\gamma_{t+3} Z_{t+4} + \dots. \tag{4.1}$$

The target therefore summarizes future values of the cumulant. The system's interaction evolves in a straightforward manner: on each time-step, the agent observes the current $\mathbf{x}_t$ and A_t, and produces a prediction V_t to estimate G_t.

It is useful to think of the cumulant, termination, and feature vector as the outputs of functions of an unobservable MDP state. The *termination function* $\gamma : \mathcal{S} \to [0, 1]$, outputs the termination signal based on the agent's observation of the MDP state (defined in Section 2.1) and $\gamma_t \stackrel{\text{def}}{=} \gamma(S_t)$. The *cumulant function*, $z : \mathcal{S} \to \mathbb{R}$, and the *feature function*, $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$, are defined similarly. We may also view the actions as produced by some mapping from states to actions, $\mu : \mathcal{S} \times \mathcal{A} \to [0, 1]$, called the behavior policy. We will use these state-based definitions to define our general value functions.

The sensorimotor data stream produced by our robots, and discussed in Chapter 3, is manifested in the feature vectors, cumulants, and termination signals. In practice, Z_t and γ_t may be computed from any information available to the agent at the current time, including the sensorimotor data and the components of $\mathbf{x}_t$. Similarly, $\mathbf{x}_t$ may be computed from available state information.

To make things more concrete, consider a simple example of predicting a robot's light sensor as it drives around a room. In this case, the feature vector could be constructed from the robot's external sensor readings, and the actions might correspond to a finite set of motor commands. The cumulant could be the current light sensor reading and the time-step could be one second long. The target could be the total future light, with future light readings weighted exponentially less each second in the future. In this example, our aim is to predict the total discounted future light based on the robot's sensors.

4.2 Cumulants

Usually in reinforcement learning, the objective of learning is to maximize a scalar reward signal. The agent approximates a value function, because an estimate of the value function is helpful for adapting the agent's policy to accumulate more reward. Predictions of future reward are used as an intermediary step inside policy learning algorithms.

Our cumulant signal plays a similar role to the reward in construction of the target, but the agent does not necessarily maximize the cumulant. The cumulant is similar to reward in that it is used to define the target, and that target is computed by summing cumulants into

the future, just as a reward. Our chief concern is predicting future cumulants rather than maximizing the cumulant signal. The cumulant may be any signal observable by the agent, even ones that may not make much sense to maximize. In fact, we may specify multiple cumulant signals for an agent to learn about, even if the agent's behavior maximizes a conventional reward signal. A cumulant could equal any scalar signal that the agent wishes to predict via the target.

To make these ideas more concrete, consider an agent driving a car from one location to another. The policy that generates actions may be learned by reinforcement learning, with a constant negative reward on each time-step. We may specify several cumulants, such as a cumulant equal to the continuous fuel consumption of the car, or a cumulant equal to one when the car is too close to the car in front of it. A third cumulant could equal to the temperature outside the car. This third cumulant is not obviously related to the goal of driving from one location to another quickly, but predicting future temperature may be useful for other tasks, or might be best thought of as general knowledge of the agent's environment.

4.3 Termination

In conventional reinforcement learning, γ is a discount factor that weights the value of future rewards. In continuing tasks, the target would become infinite without discounting future rewards; thus, each reward in the sum must be weighted by powers of γ, and γ must be less than one. The concept has reasonable intuition: rewards in the near future are more valuable to the agent than rewards in the distant future. In episodic tasks, the discount factor is one because the episodes are always a finite length and the target is bounded, meaning all rewards are considered equally valuable.

It is useful to think of discounting as a constant probability of terminating or continuing on each time-step. In episodic tasks, γ is one on each time-step, indicating the certainty of continuing the episode on the current step t. When the agent reaches a terminating state S_T, γ becomes equal to zero, indicating zero probability of continuing the episode beyond step T. We call this hard termination. In a continuing task, the discount is a constant and is less than one, indicating a γ probability of continuing on the current step and $1 - \gamma$ probability of the trajectory terminating on the current step. This may be thought of as a constant probability of terminating on each step of the trajectory, which we call soft termination. In reinforcement learning, γ is usually treated as a binary signal that only

changes on termination, not during an episode. Relating termination and continuation of a trajectory to γ makes it easier to think about γ changing with time, which we discuss next.

Hypothetical terminations, reflected by γ_t, may occur at any time. Termination conventionally refers to an interruption in the normal flow of state transitions and a reset to a starting state or starting-state distribution. On the other hand, a GVF's termination signal, γ_t, does not directly interrupt the state transitions. The behavior policy may still have real terminations, different from the hypothetical terminations defined by γ_t. Just as with cumulants, we may specify many different termination signals.

Let us continue the car driving example to solidify the concepts of hypothetical termination. The policy controlling the car may experience a real termination when the car reaches its destination. We can define a valid termination signal that becomes zero at the half-way mark of the car's route, or when it begins to rain, or when the car's remaining fuel falls below some threshold. Instead of these hard, binary terminations, we could define soft terminations. For example, a constant γ_t corresponding to the target's time horizon—larger values corresponding to longer horizons—or a value of γ related to the remaining fuel in the gas tank. The termination signal may take on any values between zero and one, based the data stream. The termination signal also facilitates mixing soft and hard terminations; for example, a γ_t equal to zero when the are arrives at its destination, and equal to 0.9 otherwise. In Chapter 6, we provide several different examples of cumulants and termination signals. We formalize this concept in the next section.

In the special case where γ_t is less than one and constant the termination signal maps to prediction time scale in the following way. Recall that $1 - \gamma(S_t)$ corresponds to the termination probability upon entry into state S_t. The expected termination time for a constant γ prediction is $\frac{\Delta t}{1-\gamma_t}$. This is like the expected number of roles of a dice to get a particular outcome: if we had a fair six sided dice we would expect to role it six times before observing a three. The Δt simply corresponds to the update cycle of the agent-environment interaction. Using this language we can talk about a prediction with $\gamma_t = 0.95$ and $\Delta t = 1$ as corresponding to a 20 step prediction.

4.4 General Value functions

The goal of this section is to describe a particular kind of value function that is well suited for representing predictive knowledge of the world. The first step is to define the GVF's target which the agent must estimate. With a constant discount factor γ, the targets are

restricted to simple time scales in which the cumulants are weighted geometrically less the more each one is delayed, as was given by the earlier definition of the return in the background chapter. With time-variable termination signals, the weighting is not by simple powers of γ, but by products of γ_t:

$$G_t \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \left(\Pi_{j=1}^{k} \gamma_{t+j}\right) Z_{t+k+1}. \tag{4.2}$$

This small change results in a significant increase in the kinds of targets that may be expressed as you will see in our experiments later (and in the literature, including Sutton, 1995; Sutton et al., 2011; Modayil, White & Sutton, 2014). If γ_t is taken to be a constant, then $\prod_{j=1}^{k} \gamma_{t+j} = \gamma^k$, and 4.2 becomes the familiar discounted sum used in reinforcement learning. Note that for the first term in the sum of 4.2, we define $\prod_{j=1}^{0} \gamma_{t+j} = \gamma^0 = 1$.

Formally, we define a general value function, or GVF, as a function $v : \mathcal{S} \to \mathbb{R}$, with three auxiliary functional inputs: a *target policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, $\gamma : \mathcal{S} \to [0, 1]$, and $z : \mathcal{S} \to \mathbb{R}$. A GVF specifies the expected value of the target, given actions are generated according to the target policy:

$$v(s; \pi, \gamma, z) \stackrel{\text{def}}{=} \mathbb{E}_\pi \left[G_t | S_t = s\right], \tag{4.3}$$

where G_t is defined by 4.2, but now with respect to the given functions; the $\mathbb{E}_\pi$ was introduced in Equation 2.3 of Chapter 2. Using the state-based definitions of the cumulant and termination functions, we can ensure the expectation in 4.3 is well defined. The expectation is conditioned on the future states produced by the state-transition dynamics of the underlying MDP (P) and the actions selected by π. We can write the GVF's definition in terms of these state-based functions:

$$v(s; \pi, \gamma, z) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \left(\Pi_{j=1}^{k} \gamma(S_{t+j})\right) z(S_{t+k+1}) | S_t = s\right] \tag{4.4}$$

$$= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} P(s, a, s') \left[z(s') + \gamma(s') v(s'; \pi, \gamma, z)\right]. \tag{4.5}$$

Equation (4.5) is called the GVF's Bellman equation. We can also define a state-action GVF as a function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, again with three auxiliary functional inputs: inputs π, γ, and z:

$$q(s, a; \pi, \gamma, z) \stackrel{\text{def}}{=} \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right]. \tag{4.6}$$

A GVF specifies a precise question about the agent's interaction with the world, and the prediction is the agent's learned approximate answer to that question. The three question functions, π, γ, and z, are referred to collectively as the GVF's *question functions*. Note

that conventional value functions remain a special case of GVFs. In section 4.1, we referred to the target G_t as specifying a question. The precise question is defined by the GVF: the expected value of the target given actions are selected according to the target policy. The next section describes how we can learn approximate answers to a GVF's question.

4.5 Learning GVFs

We begin by introducing approximate GVFs. Our approximate GVFs, denoted $\hat{v} : \mathcal{S} \times \mathbb{R}^n \to \mathbb{R}$, approximates the expected value of the target G_t:

$$\hat{v}(s, \mathbf{w}) \approx v(s; \pi, \gamma, z), \tag{4.7}$$

where $\mathbf{w} \in \mathbb{R}^n$ is the weight vector to be learned. Similarly, the approximate state-action GVF, $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^m \to \mathbb{R}$, approximates the state-action GVF:

$$\hat{q}(s, a, \mathbf{w}) \approx q(s, a; \pi, z, \gamma). \tag{4.8}$$

Note the weight vector for the state-action function can be a different size than the weight vector for the state function; for convenience we overload $\mathbf{w}$ in this way.

Our approximate GVFs are linear in the feature vector. That is, we assume that the state of the world at time t is characterized by the feature vector, $\mathbf{x}_t = \mathbf{x}(S_t)$ with $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$, and $\hat{v}$ is approximated with a finite number of weights, $\mathbf{w} \in \mathbb{R}^n$, giving prediction V_t:

$$V_t \stackrel{\text{def}}{=} \hat{v}(S_t, \mathbf{w}_t) = \mathbf{x}_t^\top \mathbf{w}_t. \tag{4.9}$$

Similarly, the approximate state-action GVF is also a linear function of a feature vector $\mathbf{x}_{a,t} = \mathbf{x}(S_t, A_t)$ with $\mathbf{x} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^m$, giving the prediction:

$$Q_t \stackrel{\text{def}}{=} \hat{q}(s, a, \mathbf{w}_t) = \mathbf{x}_{a,t}{}^\top \mathbf{w}_t. \tag{4.10}$$

In conventional reinforcement learning, V_t and Q_t are referred to as the agent's state value estimate and the state-action value estimate, respectively. From here on, we refer to Q_t as simply the agent's prediction. We adopt a linear approach to function approximation, common in reinforcement learning; this is a convenient special case, but is not essential to our approach.

An approximate GVF can be learned using temporal difference learning algorithms from reinforcement learning. A GVF is similar to a conventional value function, except the termination signal may change over time. The learning algorithms remain unchanged

in form and computational complexity. All the temporal difference learning methods described in the background chapter remain the same, except γ is replaced by γ_t or γ_{t+1}, and Z_{t+1} replaces R_{t+1}. The TD(λ) algorithm for learning approximate state GVFs from a sequence of feature vectors and actions is given by (introduced in Modayil, White & Sutton 2014):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(Z_{t+1} + \gamma_{t+1}\mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t)\mathbf{e}_t \\
\mathbf{e}_t &= \gamma_t \lambda \mathbf{e}_{t-1} + \mathbf{x}_t.
\end{aligned}$$

The GTD(λ) algorithm for learning state GVFs from a sequence of feature vectors and actions is given by (introduced in Maei 2011):

$$\begin{aligned}
\delta_t &= Z_{t+1} + \gamma_{t+1}\mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\left(\delta_t \mathbf{e}_t - \gamma_{t+1}(1-\lambda)(\mathbf{e}_t^\top \mathbf{h}_t)\mathbf{x}_{t+1}\right) \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_h\left(\delta_t \mathbf{e}_t - (\mathbf{h}_t^\top \mathbf{x}_t)\mathbf{x}_t\right) \\
\mathbf{e}_t &= \rho_t(\gamma_t \lambda \mathbf{e}_{t-1} + \mathbf{x}_t).
\end{aligned}$$

Finally, the GQ(λ) algorithm for learning state-action GVFs from a sequence of feature vectors and actions is given by (introduced in Maei & Sutton 2010):

$$\begin{aligned}
\delta_t &= Z_{t+1} + \gamma_{t+1}\overline{\mathbf{x}}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_{a,t}^\top \mathbf{w}_t \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\left(\delta_t \mathbf{e}_t - \gamma_{t+1}(1-\lambda)(\mathbf{e}_t^\top \mathbf{h}_t)\overline{\mathbf{x}}_{t+1}\right) \\
\mathbf{h}_{t+1} &= \mathbf{h}_t + \alpha_{\mathbf{h}}\left(\delta_t \mathbf{e}_t - (\mathbf{h}_t^\top \mathbf{x}_{a,t})\mathbf{x}_{a,t}\right) \\
\mathbf{e}_t &= \rho_t \gamma_t \lambda \mathbf{e}_{t-1} + \mathbf{x}_{a,t},
\end{aligned}$$

where $\overline{\mathbf{x}}_t \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}} \pi(S_t, a)\mathbf{x}(S_t)$.

The nature of the agent's approximation of a GVF is governed by the feature vector generation scheme, the eligibility trace-decay rate, λ, and the behavior policy which can be thought of as functions of state. The current state feature vector[2] can be defined as the output of a feature generating function $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$ where $\mathbf{x}_t = \mathbf{x}(S_t)$, as described in Chapter 2. Similarly, λ can be defined as a function $\lambda : \mathcal{S} \to \mathbb{R}$ where $\lambda_t = \lambda(S_t)$ In all our experiments we will consider the special case where lambda is a constant function. For simplicity we have not included a time index on λ in any of our equations to highlight our focus on this special case.

[2]We define the state-action feature function in an analogous way.

We call these three functions the *answer functions*. The majority of the experiments contained in this thesis focus on the case where these three functions are given in the experimental design—the setting we introduced in the beginning of this chapter.

Learning approximate GVFs does not necessitate the invention of new learning algorithms; we can learn the predictive knowledge specified by a GVF with existing value function learning algorithms from reinforcement learning. Specifically, we use temporal difference methods that minimize the mean squared projected Bellman error (MSPBE) and require linear computation per time-step (in the size of the feature vector): TD(λ) and GTD(λ) to approximate state GVFs ($v(s;\pi,\gamma,z)$), and GQ(λ) to approximate state-action GVFs ($q(s,a;\pi,\gamma,z)$). These methods have several desirable properties, including incremental updating during interaction, robustness to non-stationary settings, compatibility with off-policy sampling, convergence guarantees, and compatibility with linear and non-linear function approximation. Chapter 5 discusses the possibility of using quadratic temporal difference methods that also minimize the MSPBE (e.g., LSTD(λ)) for GVF learning, and Appendix B contains a discussion on the possibility of using algorithms that minimize a different objective, the *mean-square Bellman error* or MSBE.

4.6 Independence of predictive span

An important computational property of incremental multi-step prediction learning algorithms is independence of the span of a prediction. The span of a prediction is the number of time-steps elapsing between when the prediction is made and when its target value is known. In this thesis, the time-step is taken as discrete, and therefore the span is always an integer. An example span would be seven days or less, if the agent's prediction was about rain at the end of the week. An algorithm achieves span-independent computation if the computation and storage used to update and make each prediction on each time step are independent of the span of the prediction. Span independence is a computational statement; it is not a statement regarding how much data is needed to learn an accurate prediction. The significance of independence of predictive span was first introduced by Sutton & van Hasselt (in prep.).

Let us consider a particular learning algorithm that does not achieve independence of span in it's update mechanism. Consider predicting the precise value of the cumulant Z at some termination time T. The learning procedure begins at time t equals zero and proceeds in the usual way with the agent observing a feature vector, $\mathbf{x}_t$, and making predictions, V_t,

on each time-step until Z is observed at time T. Once Z is observed, we could update all the previous predictions, V_0 through to V_{T-1}, toward Z using an LMS rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(Z - \mathbf{w}_t^\top \mathbf{x}_t)\mathbf{x}_t \quad t = 0, 1, ..., T-1.$$

The algorithm cannot update any of the predictions until Z is observed. If the algorithm waits for Z to be known, then the agent must remember T previous feature vectors and make all the updates at once, and that is not span independent [3].

An algorithm may also violate span independence in the way it computes or makes new predictions. One-step prediction models, popular in time-series forecasting (Ljung, 1998), are not span independent if they are iterated to form multi-step predictions. Consider predicting rainfall. As before, the learning procedure begins at time zero and proceeds with the agent observing the amount of rainfall Z_t, and making predictions V_{t+1} about the amount of rainfall to be observed on the next time-step Z_{t+1}. Instead of using a feature vector for prediction, the algorithm keeps a finite-length history of previously observed rainfall, $[Z_t, Z_{t-1}, ..., Z_{t-k}]$. We can construct a one-step prediction model $\mathcal{M}$ that outputs an estimate of what the rainfall will be on the next time-step based on the history:

$$\mathcal{M}([Z_t, Z_{t-1}, ..., Z_{t-k}]) = V_{t+1} \approx Z_{t+1}.$$

In order to estimate rainfall two steps into the future, we iterate the model, using the model's estimate of rainfall on time-step $t+1$ as a stand-in for Z_{t+1}:

$$\mathcal{M}([V_{t+1}, Z_t, Z_{t-1}, ..., Z_{t-k+1}]) = V_{t+2} \approx Z_{t+2}.$$

Producing a k-step prediction typically involves iterating the model k-steps, and again that is dependent on the span of the prediction.

Span independence is a useful attribute for an algorithm to have. Although available computation continues to grow yearly, our agent's resources will be limited. A learning and prediction making algorithm that is not span independent may have to balance the span and number of predictions it learns. In addition, a non-span independent algorithm may take considerable time just to output a prediction, if the span of the prediction is large.

Estimating GVFs with temporal difference methods and making predictions with approximate GVFs is span independent. The GVF's termination signal, γ_t, determines the span of a GVF's prediction. The computation and storage needed to estimate a value function (learning $\mathbf{w}$) is independent of γ_t. Because of the recursive definition of a GVF given

[3] This case cannot be handled independent of span with GVFs either. Nevertheless, we can form a GVF whose prediction terminates upon observing some event (as you will see in Chapter 6).

by the Bellman equation (4.5), the value function can be estimated incrementally with constant computation per step with temporal difference learning methods (e.g., the TD(λ) algorithm). In addition, producing a new prediction (e.g., computing V_t) involves the computation of a simple dot product, and is clearly span independent. A temporal difference based algorithm pays the same computational price for updating and making both short and long term predictions. Span independence is a desirable property, and it is achievable with temporal difference methods.

4.7 A Horde of Demons

In the conventional on-policy setting, the GVF's question is about the course of actions specified by the behavior policy, and thus $\pi = \mu$. In the off-policy setting, $\pi \neq \mu$, and the GVF's question is about a hypothetical course of action defined by the target policy π. Just as we may define multiple termination signals for the same stream of experience generated by the agent's behavior, we may also define multiple target policies. Each GVF specifies a policy-contingent question about the expected future cumulant *if* the agent were to execute some target policy π. The agent's prediction answers the what-if question specified by a GVF.

Let us continue with the car driving example to solidify the concepts of hypothetical policies. As we drive home we might think about how long it would take the to reach the store if we deviated from the current course, or how long it will take to get home if we took a different route. Finally, we may consider other target policies completely unrelated to driving. For example, am I likely to feel pain in the next couple minutes if I attempt to shave while the car is moving? Figure 4.2 provides several other concrete examples of GVF, specified by different cumulants, terminations, and target policies.

Off-policy learning methods, such as GTD(λ) and GQ(λ) update the agent's approximation to the GVF using snippets of experience where the actions selected by the behavior policy are similar to the actions that the target policy would take. For example, we can update the prediction about how long it takes to get to the store as we drive home, because the store is along the way; we are getting relevant experience about driving to the store as we drive home. At any given time, multiple target policies corresponding to multiple GVFs may match the behavior's action choices, and in this case, multiple approximate GVFs can be updated off-policy and in parallel.

The Horde architecture consists of a main agent composed of many sub-agents, called

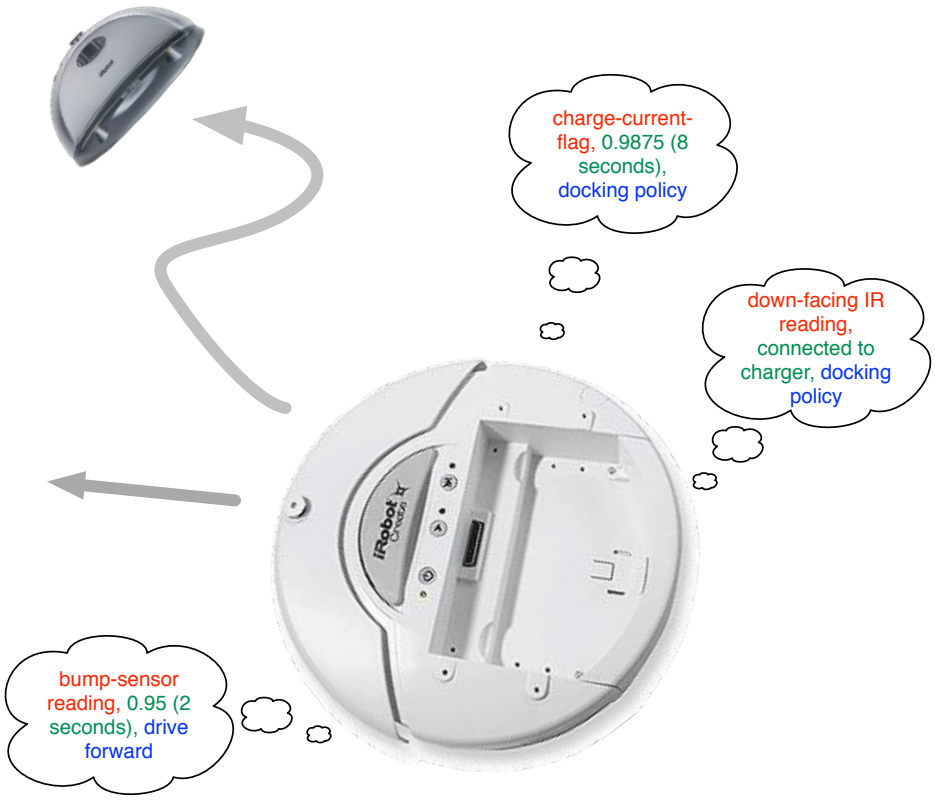

Figure 4.2: Several GVFs can be specified for a robot exploring it's world. The red text specifies the cumulant signal, the green text specifies the termination signal, and the blue text specifies the target policy for each GVF. In this example, the behavior policy is random, selecting from a discrete set of actions with equal probability. We can specify several GVFs and approximate each in parallel from a single interaction stream produced by one robot.

demons. Each demon is an independent reinforcement-learning agent responsible for learning one piece of predictive knowledge about the main agent's interaction with its environment. Each demon learns an approximation to the GVF that corresponds to the setting of the three question functions, π, γ, and z. Each demon learns it's approximate GVF using a temporal difference method: TD(λ) for on-policy demons, and GTD(λ) or GQ(λ) for off-policy demons.

We consider two kinds of demons. A prediction demon learns an approximate GVF given a target policy π; the target policy is not learned. A control demon learns its target policy. For example, a greedy policy with respect to its own approximate state-action GVF (i.e., $\pi = greedy(\hat{q})$, or $\pi(S_t, \text{argmax}_{a \in \mathcal{A}} \hat{q}(S_t, a, \mathbf{w})) = 1$). Control demons can learn and represent how to achieve goals, whereas the knowledge in prediction demons is better thought of as declarative facts.

The Horde architecture supports parallel demon learning. Each demon updates it's

corresponding approximate GVF. Each demon may use an independent feature vector generation process, but in all the experiments in this document, the demons used a common shared feature vector. Finally, all demons update their approximate GVFs from a shared common sequence of actions produced by the behavior policy. Given parallel computing hardware, many demons can be updated in parallel. Figure 6 provides a diagram of how many demons might be arranged in a single learning system.

The demons in the Horde architecture are nearly independent, but we do allow them to interact in two ways. One way in which the demons are not completely independent is that one demon can reference the approximate GVF or target policy of another demon. For example, in this way we could ask questions such as "If I follow this wall as long as I can, will my light sensor then have a high reading?". In this case, a prediction demon is referencing the target policy of a control demon. Demons may also use each others answers in their questions. For example, one demon might learn a prediction of *near obstacle*: the probability of a high proximity-sensor reading after executing a random wander. Then a second demon could learn a different prediction dependent on the first demon's prediction. For example, "If I follow this wall to its end, will I then be near an obstacle?". This can be encoded using the first demon's approximate GVF in the second demon's cumulant function (e.g., $z(S_t) = (1 - \gamma_t) \max_{a \in \mathcal{A}} \hat{q}(S_t, a, \mathbf{w}_{first\_demon})$ and $\gamma_t = 1$ during wall following and $\gamma_t = 0$ at competition). This is an example of a compositional prediction: a prediction about the outcome of another prediction.

The second way in which the demons are not completely independent is that the predictions of one or more demons may be used as input to the feature vector generation process. The prediction, Z_t, of a prediction demon from time-step t may participate in the generation of the next feature vector $\mathbf{x}_{t+1}$. If the feature vector was produced by tile coding a robot's instantaneous sensor readings, for example, then Z_t could participate in the generation of $\mathbf{x}_{t+1}$ by tile coding Z_t with the sensor readings observed on time-step $t + 1$. This method of generating feature vectors may be thought of as a form of predictive state representation. Figure 4.3 illustrates how predictions can participate in feature vector generation. Chapter 6 presents a robot learning experiment where predictions are used as input to feature generation process to improve off-policy prediction learning.

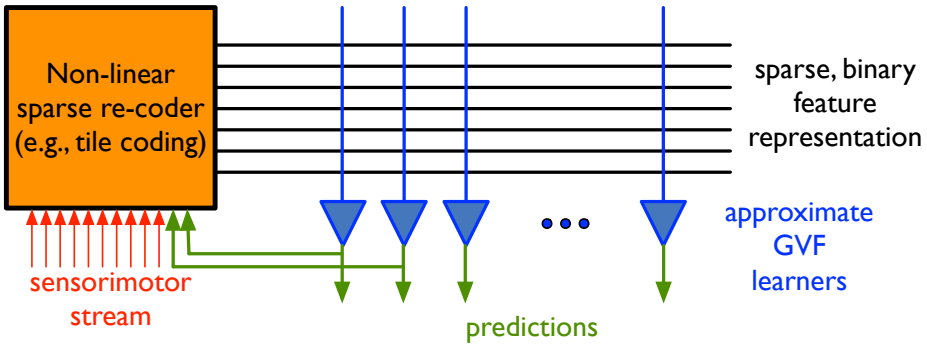

Figure 4.3: A Horde of demons. This diagram shows a possible arrangement of multiple prediction and control demons, each updating their approximate GVFs from a shared feature vector. The feature vector in this example is produced by a sparse recoding of input sensory data (such as tile coding). Some of the predictions are fed into the sparse re-coder, enabling predictions to participate in the generation of the next feature vector.

4.8 Related approaches

In this section we review prior work on predictive knowledge learning, highlighting the similarities to our approach based on GVFs. We conclude the review with a summary of how our approach is distinct from prior work, and how we extend the practicality and scalability of predictive knowledge learning.

The idea that an agent's knowledge may be represented as predictions dates back to at least the work of Cunningham (1972) and Becker (1973), who investigated constructivist approaches to learning and cognition. One of the main ideas behind constructivism is that the agent constructs its own understanding of the world, rather than a human building it in. One of the early computer simulations of prediction learning and constructivism was performed by Drescher (1991). Drescher built a computer program to simulate an infant learning, inspired by the developmental learning stages described by Piaget. The simulation contained a movable hand, an eye, and two moveable objects in a grid. Drescher reported on a single run of his program, lasting a couple hours before it ran out of memory. Although Drescher's thought experiments were more advanced than his computer simulations, his program exhibited acquisition of several high-level concepts, such as the expected outcome when closing the hand around an object. Drescher's schemas are similar to GVFs conditioned on abstract actions about abstract outcomes (both given a priori) rather than weighted sums of cumulants.

Another approach involves modeling the world solely in terms of predictions about the low-level data stream produced by an agent's interaction with the world. This approach was

explored as predictive state representations or PSRs (Littman, Sutton & Singh, 2002) and the closely related observable operator models (Jaeger, 1997). A PSR represents the state of the world by a real-valued vector of predictions about the outcomes of experiments or tests. A test is a sequence of actions and observations. The agent can compute the probability of any test succeeding, starting at the current time-step, via a linear or non-linear projection from the probability of success of each of a minimal set of core tests. The vector probabilities of each core test succeeding provides a complete summary of the entire history of the agent's interaction with the world, and are thus called state. PSRs are strictly more general than other models for partially observable domains including partially observable MDPs (POMDPs) and k-Markov models (Singh, James, & Rudary, 2004), and potentially more compact (Littman, Sutton & Singh, 2002; Singh, James & Rudary, 2004). Because a PSR is defined using only observable quantities, unlike a POMDP with hidden states, PSR parameter learning can occur without human specified models, and should be less susceptible to local minima compared to POMDPs (Singh, James, & Rudary, 2004). Learning the parameters of a PSR involves learning the weights used in the projection (see Wolf, James & Singh, 2005). Finding the minimal set of core tests involves a combinatorial search (McCracken & Bowling, 2006). GVFs are different from PSRs in that a GVF specifies an expectation about the future values of a single cumulant signal, whereas a PSR is a models the success of all possible length tests, given the history. A demon's feature vector may include components computed from other demon's predictions, enabling predictive representations, but unlike PSRs, a demon's feature vector can contain non-predictive components as well.

Another approach to low-level prediction learning, called transformed PSRs or TPSRs, attempts to improve PSR parameter learning using a singular-value decomposition (SVD) and features instead of action-observation primitives. A TPSR transforms the learning problem by multiplying the matrix test probabilities, learned from data, by a matrix produced by an SVD. Learning with continuous inputs is achieved by estimating the probabilities of sequences of feature vectors, rather than a sequence of observations as in PSRs. The TPSR approach reduces the dimensionality of PSR learning with a SVD and then computes the parameters of the TPSRs based on a batch of data. This approach was pioneered in subspace identification (Van Overschee & Moor, 1996), where SVD-based approaches achieve state-of-the-art modeling (e.g., the N4SID method). TPSRs have been applied to high-dimensional continuous observation systems such as robots (Boots et al., 2010). Recent extensions enable online parameter estimation during learning (Boots & Gordon, 2011), though the system has never been used in this way due to the computational cost of TPSR

learning algorithms.

To improve both the efficiency and scalability of prediction learning, local modeling techniques have also been explored. Both PSRs and TPSRs specify a complete representation of an agent's interaction with an unknown environment. The core tests can be used to predict any length action-observation sequences: a PSR learns and makes all possible predictions. In many domains it may be either infeasible to learn this full model, or it may not be practically useful for the agent. For example, a robot with infrared distance sensors may predict how long it takes to cross the room without representing the dynamics of the overhead lights in the room. Recently, Talvatie and Singh (2011) explored learning a subset of a PSR's tests called prediction profile models rather than learning the full PSR. A prediction profile model is similar to a demon, in that it specifies a single independent predictive question of interest, and a collection of demons forms a kind of distributed model of the agent's interaction with an unknown world, like a collection of prediction profile models.

Temporal difference networks, or TD-nets, are another approach to learning predictions about the agents interaction with the world. A TD-net represents and estimates a collection of scalar predictions arranged in an interconnected network (Sutton & Tanner, 2005). Network nodes represent primitive observations, or predictions. Each non-primitive node predicts the output of another node, conditioned on a discrete action. This arrangement enables compositional predictions where one node may predict the output of another node, which in turn predicts the output of another node, and so on. In addition, a node's prediction may be conditioned on an option policy and updated off-policy (as in Sutton, Rafols & Koop, 2006). The arrangement of nodes in the network defines each prediction's question, and is called the question network. The approximation architecture and learning rules used to compute the estimates of the predictions define the answers to the prediction's question, and is called the answer network. A TD-net's predictions can be compositional and policy contingent, just like a GVF. The main algorithmic differences between TD-nets and Horde is that Horde has a more straightforward handling of state and function approximation, and Horde uses more efficient algorithms for off-policy learning (Maei & Sutton 2010; Maei 2011).

Option models, from reinforcement learning, are similar to GVFs. An option model is a partial model of the world that consists of predictions about which states the option terminates in, and the cumulative reward observed during the option. An agent may learn many option models, and update them in parallel using off-policy learning algorithms (as done by Singh, Barto & Chentanez, 2005). Option models use a single global reward signal,

whereas each GVF has its own cumulant. To construct an option to achieve a goal, we define a special goal reward function. A GVF can achieve the same effect using a specific instantiation of the cumulant signal (discussed in Chapter 6)[4], without the need for a special outcome variable in the update equations of the learning algorithm.

Horde builds on prior work, achieving online and off-policy learning from continuous sensor streams generated by a robot. Previous work was done in small simulations, with the work on TPSRs being the notable exception. In comparison with Horde, TPSRs do not support policy-contingent prediction, and empirical demonstrations have been limited to learning from a few thousands samples off-line (not learning while collecting data). Our results with Horde demonstrate incrementally learning thousands of predictions, on- and off-policy, from hours of data, while maintaining a 100 ms update cycle. Later chapters of this thesis demonstrate the practicality and scalability of our architecture with experiments on two robot platforms as well as simulation worlds. Although we make no formal arguments of the representational ability of GVFs, our empirical demonstrations illustrate that GVFs can capture much predictive knowledge. Taken together, our approach and empirical demonstrations further the development of predictive knowledge learning.

4.9 Summary

In this chapter we formalized the setting under which we propose to learning predictive knowledge. We introduced GVFs, a new way to represent predictive knowledge. We described how many GVFs can be organized and learned by a Horde of demons. The chapter closed with a description of previous efforts to represent and learn knowledge that is predictive.

A variety of predictions may be represented by a general kind of value function called a GVF. Because GVF are value functions, they can be learned online with computationally efficient value function estimation algorithms that are compatible with function approximation. Compared with other predictive representations of knowledge, GVFs appear to be more suitable for online, incremental updating from continuous outputs, and are potentially more scalable because of the linear, span-independent computation and parallel updating of value function learning. At this point, these supposed benefits remain, for the most part, theoretical. The next few chapters provide more concrete illustrations of the usefulness of GVFs for representing predictive knowledge, with experiments on robots.

[4]This simplification was first observed by Joseph Modayil.

Chapter 5

Nexting[1]

The term nexting has been used by psychologists to refer to the natural inclination of people and many animals to continually predict what will happen next. In this chapter, we describe how a robot can learn to next in realtime, making 6000 predictions, each computed as a function of 6000 features of the state. These predictions are formulated as GVFs. The results presented in this chapter demonstrate how GVFs can be used to represent an interesting and important class of predictions, and that these predictions can be learned, at scale, using conventional value-function learning algorithms with function approximation.

5.1 Predicting what will happen next

Psychologists conjecture that people and animals continually make large numbers of short-term predictions about their sensory input (Gilbert, 2006; Brogden, 1939; Pezzulo, 2008; Carlsson et al., 2000). When a person hears a melody they might predict what the next note will be or when the next downbeat will occur. When you see an object in flight, hear your own footsteps, or handle an object, you seem to continually make and confirm multiple predictions about our sensory input. Our surprise and disappointment when these predictions are disconfirmed—an unexpected note or a sudden change in direction—suggests we are engaged in continual prediction. When you ride a bike you have finely tuned moment-by-moment predictions of whether you will fall and of how your trajectory will change as a result. In all these examples, we continually predict what will happen to us *next*. Making predictions of this simple, personal, short-term kind has been called *nexting* by Gilbert (2006).

[1]The approach to nexting, experimental results, and text contained in this chapter are either adapted from co-authored sections of several papers (Modayil, White, Pilarski & Sutton, 2012; Modayil, White, Pilarski & Sutton, 2012b; Modayil, White & Sutton, 2012; Modayil, White & Sutton, 2014) or originally generated for this document.

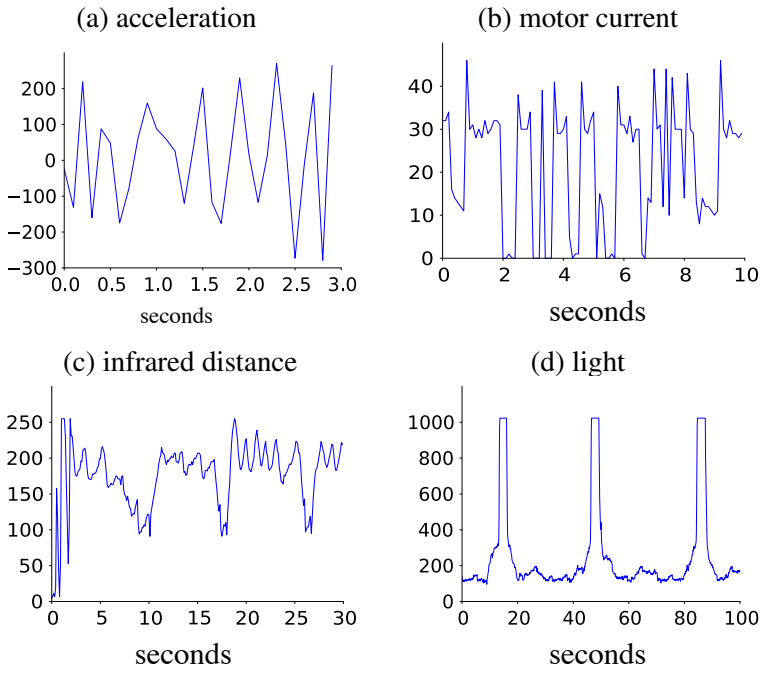

Figure 5.1: Examples of the variance of sensorimotor data at different time scales on the robot: (a) acceleration varying over tenths of a second, (b) motor current varying over fractions of a second, (c) infrared distance varying over seconds, and (d) ambient light varying over tens of seconds. The ranges of the sensorimotor data vary across the different sensor types.

Nexting predictions are specific to one individual's environment, capabilities, and experiences. Predictions of the stock market, of political events, or of technological trends seem to involve cultural and communal knowledge, are fewer in number, and are about much longer time scales. Nexting, on the other hand, is the continual process of making massive numbers of short-term predictions in parallel. Nexting prediction appear to happen automatically and unconsciously. Moreover, nexting predictions seem to be made simultaneously at multiple time scales. When we read, for example, it seems likely that we next at the letter, word, and sentence levels, each involving substantially different time scales (Gilbert 2006). In a similar fashion to these regularities in a person or animal's experience, our robot observes predictable regularities at time scales ranging from tenths of seconds to tens of seconds (Figure 5.1).

The ability to predict and anticipate has often been proposed as a key part of human and animal intelligence (e.g., Tolman, 1951; Hawkins & Blakeslee, 2004; Butz, Sigaud & Gérard, 2003; Wolpert, Ghahramani & Jordan, 1995; Clark, 2012). The modern view of the basic learning and behavioral modification known as *classical conditioning* says that people

and a wide variety of animals learn and make simple predictions at a range of short time scales (Rescorla, 1980; Pavlov, 1927). In a standard classical conditioning experiment, an animal is repeatedly given a neutral sensory cue followed by a special stimulus that invokes a built-in reflex response. For example, the sound of a bell might be followed by a shock to the paw, which causes the animal retract its limb. After a while the limb starts to be retracted early, in response to the bell. This is interpreted as the bell causing a prediction of the shock, which then triggers limb retraction. In other experiments, known as *sensory preconditioning* (Brogden, 1939; Rescorla, 1980), it has been shown that animals learn predictive relationships between neutral stimuli (e.g., light or tone). These stimuli that are neither inherently good or bad (unlike food or shock) and not connected to an built-in response. In this case, the animal still makes predictive associations between the stimuli even though there is no behavioral evidence of the association; later experiments must be performed to show prediction is occurring. Animals seem to be wired to learn many predictive relationships in their world.

To be able to next is to have a basic kind of knowledge about how the world works in interaction with one's body. It is to have a limited form of forward model (Jordan & Rumelhart, 1992) of the world's dynamics, specifically, "What will happen next if I continue my current course of action?". In the next section, we show how nexting predictions can be represented as state GVFs and learned using the TD(λ) algorithm.

5.2 Nexting as multiple value functions

We will represent each nexting prediction with a GVF, learned by a prediction demon. We are interested in learning to predict the multi-dimensional sensorimotor data produced by a robot interacting with the world (see Figure 5.2). The data is multi-dimensional, and thus we must specify multiple cumulant signals. The value at time t of the cumulant signal pertaining to the ith GVF is denoted $Z_t^{(i)} \in \mathbb{R}$. The ith prediction itself, denoted $V_t^{(i)} \in \mathbb{R}$, is meant to estimate a GVF with a constant termination signal denoted $\gamma^{(i)} \in [0, 1)$, and actions selected by π:

$$V_t^{(i)} \approx \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} (\gamma^{(i)})^k Z_{t+k+1}^{(i)} \big| S_t = s \right] = \mathbb{E}_\pi [G_t^{(i)} | S_t = s], \tag{5.1}$$

where $G_t^{(i)}$ denotes the current value of the target of the ith GVF. In this on-policy setting, the behavior policy is equal to the target policy π. In experimental results, the cumulant corresponded to one of the robot's sensors or else a component of a feature vector, and the

termination signal was one of four fixed values, $\gamma^{(i)} = \{0, 0.8, 0.95, 0.9875\}$, corresponding to time scales (T values) of 0.1, 0.5, 2, or 8 seconds. In the following experiments the update cycle of our robot was set to 10 times a second, and thus our $\{1, 5, 20, 80\}$ step predictions correspond to 0.1, 0.5, 2, or 8 seconds.

Each demon's prediction $V_t^{(i)}$ is formed as an inner product of $\mathbf{x}_t$ with a corresponding weight vector $\mathbf{w}_t^{(i)}$:

$$V_t^{(i)} \;=\; \mathbf{x}_t^\top \mathbf{w}_t^{(i)} \;=\; \sum_{j=1}^{n} \mathbf{x}_t(j)\mathbf{w}_t^{(i)}(j).$$

In our experiments, the feature vectors had $n = 6065$ components, but only a fraction of them were nonzero, so the sums could be very cheaply computed.

We used linear TD(λ) inside each prediction demon for learning the weight vectors:

$$\mathbf{w}_{t+1}^{(i)} = \mathbf{w}_t^{(i)} + \alpha \left(Z_{t+1}^{(i)} + \gamma^{(i)} \mathbf{x}_{t+1}^\top \mathbf{w}_t^{(i)} - \mathbf{x}_t^\top \mathbf{w}_t^{(i)} \right) \mathbf{e}_t^{(i)}, \tag{5.2}$$

where a common step-size parameter and feature vector is shared amongst all demons, and $\mathbf{e}_t^{(i)}$ is the eligibility trace vector for each demon. As usual, $\mathbf{e}_t^{(i)}$ is initially set to zero and then updated on each step by:

$$\mathbf{e}_t^{(i)} = \gamma^{(i)} \lambda \mathbf{e}_{t-1}^{(i)} + \mathbf{x}_t, \tag{5.3}$$

where trace-decay parameter $\lambda \in [0, 1]$ is shared amongst all demons.

The TD(λ) algorithm has been used as a model of classical conditioning (Sutton & Barto, 1990) within which various different stimuli are viewed as playing the role of reward in the learning algorithm. The approach to nexting taken here can be seen as taking this approach to the extreme, using GVFs to predict a large number of a great variety of reward-like time series at many time scales (cf. Sutton 1995, Sutton & Tanner 2005).

Under common assumptions and a decreasing step-size parameter, TD(λ) with $\lambda = 1$ converges asymptotically to the weight vector that minimizes the mean squared error between the prediction and the target (5.1). In practice, smaller values of λ are almost always used because they can result in significantly faster learning (e.g., see Sutton & Barto 1998, Figure 8.15), but the $\lambda = 1$ case still provides an important theoretical touchstone. In this case we can define the *best static weight vector* $\mathbf{w}_*^{(i)}$ as that which minimizes the squared error over the first N predictions:

$$\mathbf{w}_*^{(i)} = \arg\min_{\mathbf{w}} \sum_{t=1}^{N} \left(\mathbf{x}_t^\top \mathbf{w} - G_t^{(i)} \right)^2. \tag{5.4}$$

The best static weight vector can be computed offline by standard algorithms for solving large least-squares regression problems. In the experiments that follow, we computed the

least-squares solution for the best static weight vector, using an eigen decomposition to invert the sample covariance matrix of feature vectors. We reduced the dimension of the system by thresholding based on the eigenvalues. This algorithm uses $O(n^2)$ memory and $O(Nn^2)$ computation (because N is much larger than n), per prediction, and is just barely tractable for offline use at the scale we consider here (in which $n = 6065$). Although this algorithm is not practical for online use, its solution $\mathbf{w}_*^{(i)}$ provides a useful performance standard. Note that even the best static weight vector will incur some error. It is even theoretically possible that an online learning algorithm could perform better than $\mathbf{w}_*^{(i)}$, by adapting to gradual changes in the world or robot.

5.3 A scaling experiment

To explore the practicality of our approach to nexting as described above, we performed a large scale experiment where thousands of prediction demons are learned from thousands of features in realtime on the Critterbot. The robot's interaction with its environment was structured in a tight loop with a 100 millisecond (ms) time-step. At each step, the sensory information was used to select one of seven actions corresponding to basic movements of the robot (forward, backward, slide right, slide left, turn right, turn left, and stop). Each action caused a different set of voltage commands to be sent to the three motors driving the wheels. The state of the robot was characterized by 53 real or virtual sensors of 13 types, as summarized in the first two columns of Table 5.1.

The experiment was conducted in the robot's pen with a lamp on one edge (Figure 5.2). The robot selected actions according to a fixed stochastic policy that caused it to generally follow a wall on its right side. The policy selected the forward action by default, the slide-left or slide-right action when the right-side-facing IR distance sensor exceeded or fell below given thresholds, and selected the backward action when the front IR distance sensor exceeded another threshold (indicating an obstacle ahead). The thresholds were chosen such that the robot rarely collided with the wall and rarely strayed more than half a meter from the wall. By design, the backward action also caused the robot to turn slightly to the left, facilitating the many left turns needed for wall following on the right. To inject some variability into the behavior, on 5% of the time steps, the policy instead chose an action at random from the seven possibilities with equal probability. Following this policy, the robot usually completed a circuit of the pen in about 40 seconds. A circuit took significantly longer if the motors overheated and temporarily shut themselves down. In this case the

sensor type	num of sensors	tiling type	num of intervals	num of tilings
IRdistance	10	1D	8	8
		1D	2	4
		2D	4	4
		2D+1	4	4
Light	4	1D	4	8
		2D	4	1
IRlight	8	1D	8	6
		1D	4	1
		2D	8	1
		2D+1	8	1
Thermal	4(8)	1D	8	4
RotationalVelocity	1	1D	8	8
Magnetic	3	1D	8	8
Acceleration	3	1D	8	8
MotorSpeed	3	1D	8	4
		2D	8	8
MotorVoltage	3	1D	8	2
MotorCurrent	3	1D	8	2
MotorTemperature	3	1D	4	4
LastMotorRequest	3	1D	6	4
OverheatingFlag	1	1D	2	4

Table 5.1: Summary of the tile-coding strategy used to produce feature vectors from sensorimotor data. For each sensor of a given type, its tilings were either 1-dimensional or 2-dimensional, with the given number of intervals. Only the first four of the robot's eight thermal sensors were included in the tile coding due to a coding error.

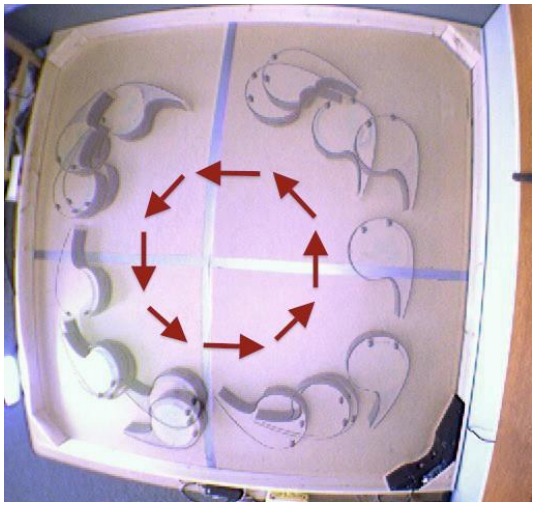

Figure 5.2: An illustration of the wall-following behavior that generated the data. The circuits around the pen involved substantial random variation, but almost always included passing the bright light on the lower-left side.

robot did not move, irrespective of the action chosen by the policy. Shut downs occurred approximately every eight minutes and lasted for about seven minutes. This simple policy was sufficient for the robot to reliably follow the wall for hours.

To produce the feature vectors needed for each demon, the sensorimotor data were coarsely coded according to a tile-coding strategy as summarized in Table 5.1. Most of the tilings were 1-dimensional (1D), that is, over a single sensor's output, in which case a tile was simply an interval of the sensor's output value. For some sensors, 2-dimensional (2D) tilings were used by taking neighboring sensors in pairs. This enabled the robot's features to distinguish between, for example, a wall and a corner. To provide further discriminatory power, for some sensor types, 2-dimensional tilings from pairs consisting of a sensor and its second-neighboring sensor were used. These are indicated as 2D+1 tilings in Table 1. Finally, a tiling with a single tile that covered the entire sensor space and thus whose corresponding feature, called the *bias feature*, was always active. Altogether, the tile-coding strategy used 457 tilings, producing feature vectors with $n = 6065$ components, most of which were zeros, but exactly 457 of which were ones. See Chapter 2 (background) for a description of tile coding.

Each nexting prediction was formalized as a GVF with the question functions defined as follows: a target policy equal to the behavior, a cumulant equal to a sensor reading or feature vector component, and a constant termination signal. The cumulant $Z_t^{(i)}$ of each GVF

corresponded to each of the 53 sensors and a random selection of 487 from the 6064 non-bias feature vector components. For each, four GVFs were learned with the four values of the termination signal $\gamma^{(i)} = 0, 0.8, 0.95$, and 0.9875, corresponding to time scales of 0.1, 0.5, 2, and 8 seconds respectively. Thus, we want to update a total of $(53+487)\times 4 = 2160$ prediction demons.

Temporal difference learning was used in all 2160 prediction demons. The answer functions were the same for every demon: a hand-designed behavior policy for wall following, a feature vector produced by multiple independent tilings of the robot's sensors, and a constant $\lambda = 0.9$. The parameters of TD(λ) were $\alpha = \frac{0.1}{457}$ (as there are 457 active features), and the initial weight vector was zero. Data was logged to disk for later analysis. The total run time for this experiment was approximately three hours and twenty minutes (120,000 time steps).

We can now address the main question: is realtime nexting practical at this scale? This comes down to whether or not all the computations for making and learning so many complex predictions can be reliably completed within the robot's 100 ms time-step. The wall-following policy, tile-coding, and TD(λ) were all implemented in Java and run on a laptop computer connected to the robot by a dedicated wireless link. The laptop used an Intel Core 2 Duo processor with a 2.4GHz clock cycle, 3MB of shared L3 cache, and 4GB DDR3 RAM. Four threads were used for the learning code. The total memory consumption was 400MB. With this setup, the time required to make and update all 2160 predictions was 55ms, well within the 100ms duty cycle of the robot. This demonstrates that it is indeed practical to do large-scale nexting on a robot with conventional computational resources.

Later, a newer laptop computer (Intel Core i7, 2.7 Ghz quad core, 8GB 1600 Mhz DDR3 RAM, 8 threads), with the same style of predictions and the same features, was able to make 8000 predictions in 85ms. This shows that with more computational resources, the number of predictions (or the size of the feature vectors) can be increased proportionally. This strategy for nexting easily scales to millions of predictions with foreseeable increases in computing power over the next decade.

The experiment described above and the robot experiments of Chapter 6 and 8 were implemented in a publicly available framework for reinforcement learning called RL-Park.[2]

[2]RL-Park was developed by Thomas Degris, with contributions from this author and several others. The framework supports plugins for interfacing with the Critterbot and the iRobot Create. The code can be downloaded from http://rlpark.github.io/

5.4 Accuracy of learned predictions

All learning algorithms used in this thesis are available within RL-Park.

The predictions were learned with substantial accuracy. For example, consider the eight-second prediction whose cumulant is the third light-sensor data (Light3). Notice that there is a bright lamp in the lower-left corner of the pen (Figure 5.2). On each trip around the pen, the output from this light sensor increased to its maximal level and then fell back to a low level, as shown in the upper portion of Figure 5.3. If the features are sufficiently informative, then the robot should be able to anticipate the rising and falling of this sensor's output. Also shown in the figure is the target for the data, $G_t^{(i)}$, computed retrospectively from the subsequent output produced by the light sensor. Of course no incremental prediction algorithm could achieve this without access to future data—our learning algorithms seek to approximate this 'clairvoyant' prediction using only the sensorimotor data available in the current feature vector.

The prediction made by TD(λ) is shown in the lower portion of Figure 5.3, along with the prediction made by the best static weight vector $\mathbf{w}_*^{(i)}$ computed retrospectively as described in Section 2. The key result is that the TD(λ) prediction anticipates both the rise and fall of the light. Both the learned prediction and the best static prediction track the target, though with visible fluctuations.

To remove these fluctuations and highlight the general trends in the eight-second predictions of Light3, we can average the predictions over 100 circuits around the pen, aligning each circuit's data to the time of initial saturation of the light sensor. The average of the target, TD(λ) prediction, and best-static-weight-vector prediction for 15 seconds near the time of saturation are shown in Figure 5.4. All three averages rise in anticipation of the onset of Light3 saturation and fall rapidly afterwards. The target peaks before saturation, because the Light3 output regularly became elevated prior to saturation. The two learned predictions are roughly similar to the target and to each other, but there are substantial differences. These differences do not necessarily indicate error or essential characteristics of the algorithms. For example, such differences can arise because the average is over a biased sample of data—those time steps that preceded a large rise in the cumulant. We have established that some of the differences are due to the motor shutdowns. Notably, if the data from the shutdowns are excluded, then the prominent bump in the best-static-$\mathbf{w}$ prediction (in Figure 5.4) at the time of saturation onset disappears.

Accessing the quality of the eight-second Light3 prediction as it evolves over time and

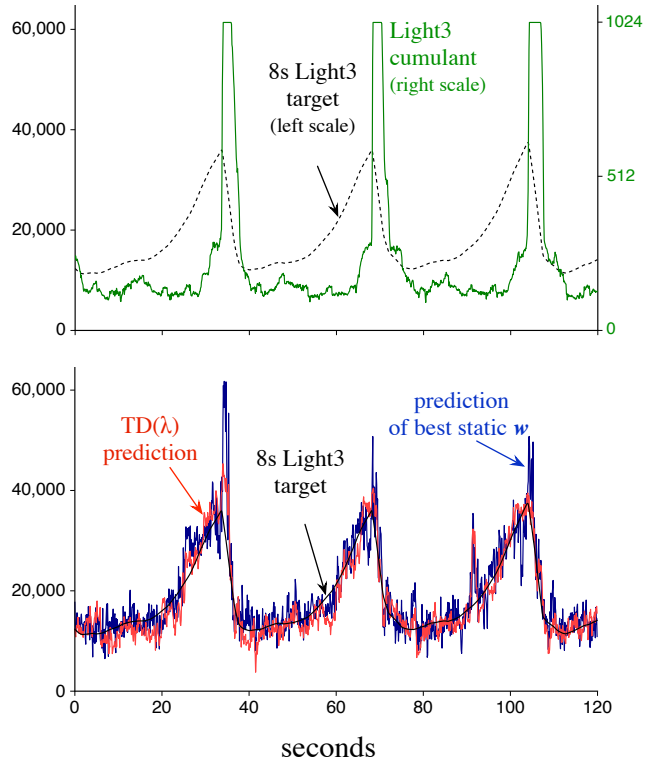

Figure 5.3: Predictions of the Light3 cumulant at the eight-second time scale. The upper graph shows the Light3 sensor data spiking and saturating on three circuits around the pen and the corresponding target (computed afterwards from the future cumulants). Note that the target shows the signature of nexting—a substantial increase prior to the spikes in cumulant. The lower graph shows the same target compared to the prediction of the TD(λ) algorithm and of the prediction of the best static weight vector. These feature-based predictions are more variable, but substantially track the target.

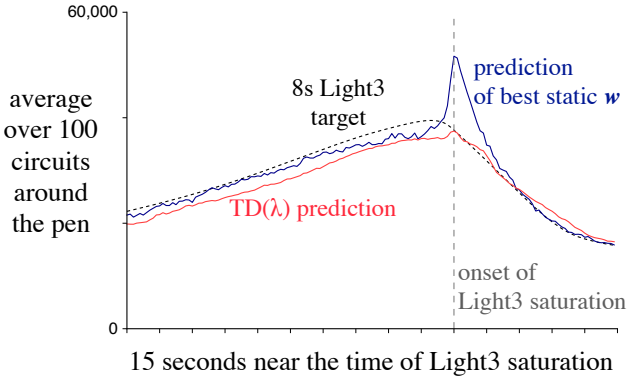

Figure 5.4: Average of Light3 predictions (like those in the lower portion of Figure 5.3) over 100 circuits around the pen and aligned at the onset of Light3 saturation.

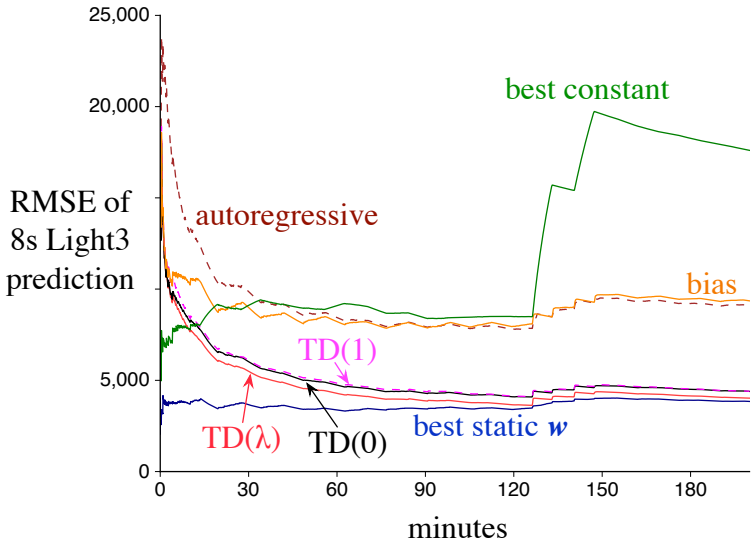

Figure 5.5: Learning curves for eight-second Light3 predictions made by various algorithms over the full data set. Each point is the RMSE of the prediction of the algorithm up to that time. Most algorithms use only the data available up to that time, but the best-static-$\mathbf{w}$ and best-constant algorithms use knowledge of the whole data set. The errors of all algorithms increased at about 130 and 150 minutes because the motors overheated and shutdown at those times while the robot was passing near the light, causing an unusual pattern in the sensorimotor data. In spite of the unusual events, the RMSE of TD(λ) still approached that of the best static weight vector. See text for the other algorithms.

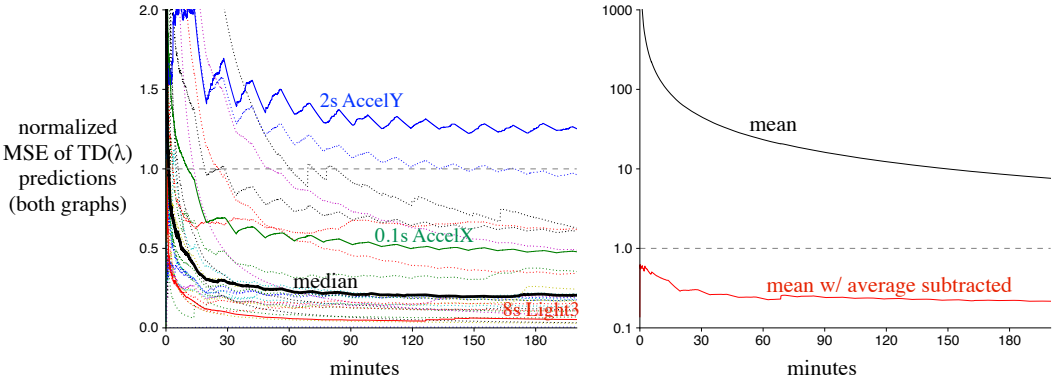

Figure 5.6: Learning curves for the 212 predictions whose cumulant corresponds to each sensor. The median and several representative learning curves are shown on a linear scale on the left, and the mean learning curve is shown on a logarithmic scale on the right. The mean curve is high because of a minority of the sensors whose absolute values are high and whose variance is low. If the experiment is rerun using cumulants with their average value subtracted out, then the mean performance is greatly improved, as shown on the right, explaining 78% of the variance in the target by the end of the data set.

data is relatively straightforward. As a measure of the quality of a prediction sequence $\{V_t^{(i)}\}$ up through time T, we can use the root mean squared error, defined as:

$$\text{RMSE}(i,T) = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(V_t^{(i)} - G_t^{(i)})^2}.$$

Figure 5.5 shows the RMSE of the eight-second Light3 predictions for various algorithms. For TD(λ), the parameters were set as described above. For all the other algorithms, their parameters were tuned manually to optimize the final RMSE. The other algorithms included TD(λ) for $\lambda = 0$ and $\lambda = 1$, both of which performed slightly worse than $\lambda = 0.9$. Also shown is the RMSE of the prediction of the best static weight vector and of the best constant prediction. In these cases, the prediction function does not actually change over time, but the RMSE measure varies as harder or easier situations from which to make predictions are encountered. Note that the RMSE of the TD(λ) prediction comes to closely approach that of the best static weight vector after about 90 minutes. This demonstrates that online learning on robots can be effective in realtime with a few hours of experience, even with a large feature vector.

The benefits of a large representation are shown in Figure 5.5 by the substantially improved performance over the 'Bias' algorithm, which was TD(0) with a trivial representation consisting only of the bias feature (the single feature that is always 1). As an additional performance standard, also shown is the RMSE of a multi-step (non-iterated) variation of the autoregressive algorithm (e.g., see Box, Jenkins & Reinsel, 2011) that uses previous output of the Light3 sensor as features of a linear predictor, with the weights trained according to the least-mean-square rule. To incrementally train the autoregressive model, the learning was delayed by 600 time steps to compute the target. The best performance of this algorithm was obtained using a model of order 300, meaning the last 300 Light3 sensor readings were used. The autoregressive model performed much worse than all the algorithms that used a rich feature representation.

Moving beyond the single prediction of one light sensor at one time scale, we would like to evaluate the accuracy of all 212 GVF about sensors at various time scales. To measure the accuracy of predictions with different magnitudes, we can use a normalized mean squared error:

$$\text{NMSE}(i,t) = \frac{\text{RMSE}^2(i,t)}{\text{var}(i)},$$

in which the mean squared error is scaled by $\text{var}(i)$, the sample variance of the targets $G_t^{(i)}$ over all the time steps. This error measure can be interpreted as the percent of variance *not*

explained by the prediction. It is equal to one when the prediction is constant at the average target value, but can be much larger than one.

Learning curves using the NMSE measure for the 212 prediction demons whose cumulant corresponds to a sensor are shown in Figure 5.6. The left panel shows the median learning curve and curves for a selection of individual prediction demons. In most cases, the error decreased rapidly over time, falling substantially below the unit variance line. The median prediction explained 80% of the variance at the end of training and 71% of the variance after just 30 minutes. In many cases the decrease in error was not monotonic, sometimes rising sharply (presumably as a new part of the state space was encountered) before falling further. In some cases, such as the 2-second Y-acceleration GVF shown, the sensor was never effectively predicted, evidenced by its NMSE never falling below one. This cumulant was simply unpredictable given the feature representation provided.

The mean learning curve, shown in the right panel of Figure 5.6 on a log scale, fell rapidly but was always substantially above one. This was due to a minority of the sensors (mainly the thermal sensors) whose values were far from zero but whose variance was small. The learning curves for the corresponding predictions were all very high (and do not appear in the left panel because they were well off the scale). Why did this happen? Note that the prediction algorithm was biased in that all the initial predictions were zero (because the weight vector was initialized to zero). When the cumulants are large relative to their variance, this bias can result in a very large NMSE that takes a long time to subside. One way to eliminate the bias is to modify the cumulant by subtracting from each sensor value the average of its values up to that time (e.g., the first cumulant is always zero). This average was computed as a long-run sample average, not a moving or windowed average. This is easily computed and uses only information readily available at the time. Most importantly, choosing the initial predictions to be zero is no longer a bias but simply the right choice. When we modified our cumulants in this way and reran TD(λ) on the logged data, we obtained the much lower mean learning curve shown in Figure 5.6 (right). In the mean, the prediction learned with the average subtracted explained 78% of the variance of the target by the end of the data set.

Finally, consider the majority of the prediction demons whose cumulant was one of the binary features making up the feature vector. There were 170 constant features among the 487 binary features that were selected to be cumulants, and thus with the average subtracted, both the targets and the learned predictions were constant at zero. For these constant predictions, the RMSE was zero and the variance was zero, and we excluded these predictions

from further analysis. For the remainder of the prediction demons, both the median and mean explained 30% of the variance of the target by the end of the data set.

5.5 Unmodeled situations

When learning and interacting on a real physical hardware, there are often unexpected events and unknowns that are not considered ahead of time. For example, the magnetic sensors of the robot were designed to give the robot a rough sense of direction. However, when we inspect the magnetic data from the robot (see Figure 5.7) we can observe an unusual pattern in the data that does not correspond with our a priori expectations. There appears to be something in the floor of one corner of the pen, perhaps a power cable, that dramatically affected the magnetic sensor reading. This little unexpected regularity in the sensorimotor stream was discovered and eventually predicted by our robot.

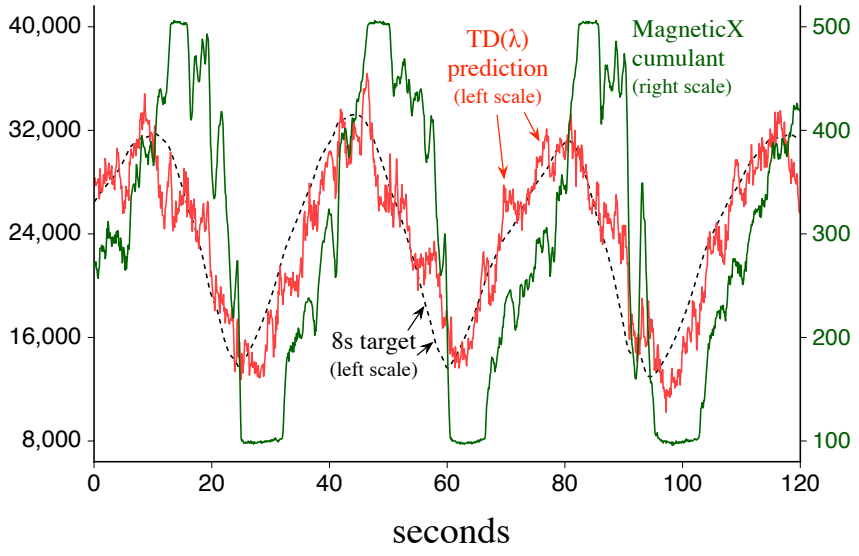

Figure 5.7: Predictions of the MagneticX sensor at the eight-second time scale. The TD(λ) prediction was close to the target, explaining 91 percent of its variance.

5.6 Linear and quadratic computation

We have demonstrated that the nexting predictions learned by TD(λ) obtained good accuracy compared to several important baselines, but what about the computational requirements of nexting? We used the TD(λ) algorithm to learn nexting predictions because the

algorithm's linear memory and computational footprint were deemed essential for real-time updating of thousands of predictions. In problems where quadratic computation can be accommodated, the LSTD(λ) algorithm is sometimes preferred to TD(λ) due to its potentially superior data efficiency. Could we learn thousands of nexting predictions in realtime updating each demon with LSTD(λ)? The aim of this section is to answer to this question.

First let us make some rough calculations on the computational demands of nexting. Our setup with TD(λ), was capable of learning over 6000 predictions, with a 6065 dimensional feature vector at 10 times a second, requiring approximately 364 million operations a second, using dense vector computations. LSTD(λ), in contrast, would perform approximately 2.2×10^{12} dense vector operations per second. In our nexting experiment, we used sparse-vector optimizations, reducing TD(λ)'s computation to approximately 327 million operations per second. A sparse implement of LSTD(λ), with incremental matrix inversion, would perform 1.1×10^{12} operations per second to achieve nexting-scale prediction learning.

Let us move beyond rough calculations and perform a timing experiment to directly compare prediction learning with TD(0) and LSTD(0). We exclude traces for simplicity. The problem is one of learning a single nexting prediction using a binary feature vector, which is generated randomly with 10% of $\mathbf{x}_t$ equal to one on each time-step. This problem has a single cumulant equal to the Light3 sensor from the log of the nexting experiment and a constant termination equal to 0.9875. In this experiment we tested matlab and java implementations of the TD(0) and LSTD(0) algorithms (with sparse operations optimized). Each algorithm was timed with different feature vector sizes ranging from 1 to 316228. Then the experiment was repeated for 100 runs. The random seed was initialized to the same value for all algorithm instances at the beginning of each group of 100 runs. For each run, the runtime for each feature vector length was recorded and averaged over runs to produce the curves shown in Figure 5.8. These results do not simulate other costs, such as queuing the function approximator, communicating with the robot, or imperfect parallel execution. These runtimes should be considered an upper limit on what is achievable with a full prediction learning system.

Next, we used the timing results in the timing experiment to approximate the number of nexting demons the java implementations of TD(0) and LSTD(0) could update within the time-step of our robot. Using the runtime measurements of each implementation, and assuming perfect parallelism on a four core CPU, we can compute the number of predictions that could be updated in a 100 millisecond time-step for a given feature vector length.

Figure 5.9 plots theses results. The results indicate that the LSTD(0) algorithm can only update a single prediction within the time-step of our robot, using the same size feature vector as we used in the nexting experiment.

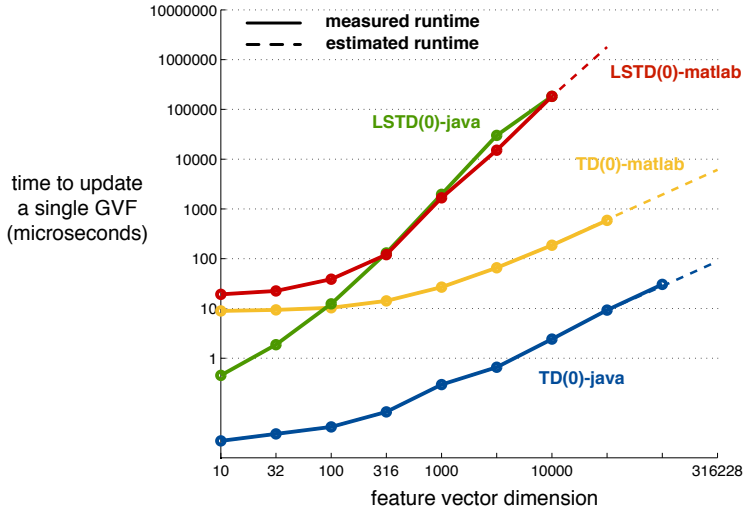

Figure 5.8: A plot of the average runtime used by the TD(0) and LSTD(0) algorithms to update a single nexting demon with different feature vector lengths on a simulated problem. Note the log scale. This graph also includes extrapolations of the runtimes exhibited by each algorithm. Both TD(0) implementation fit a linear trend, which we then extrapolated to estimate the runtime for larger feature vector lengths. The two LSTD(0) implementations exhibit a quadratic trend, which is used to extrapolate the runtime of the LSTD implementations. See text for a description of the experimental setup.

We conclude from these results that nexting with a quadratic learning method, like LSTD(0), is not computationally feasible given the feature vectors we used. Our approach to nexting involves learning many predictions in real-time with large feature vectors. For our approach of nexting, we conclude that a linear-complexity TD method is more suitable, from a computational perspective.

We might hope to improve upon the efficiency of LSTD(λ) in several ways. For instance, nexting predictions could share data structures. In particular, the A matrix and eligibility trace vector of LSTD(λ) could be shared amongst any demons with the same termination signal. This approach is of limited value because, as highlighted by our timing results, we cannot update even one nexting prediction with LSTD(λ). Another option is to use lower-dimensional random projections to estimate LSTD's A matrix, therefore reducing computation. This optimization, like sharing data structures, would involve more complex implementations of LSTD(λ), whereas our conventional implementation of TD(λ) enabled large-scale, realtime parallel updating and accurate prediction.

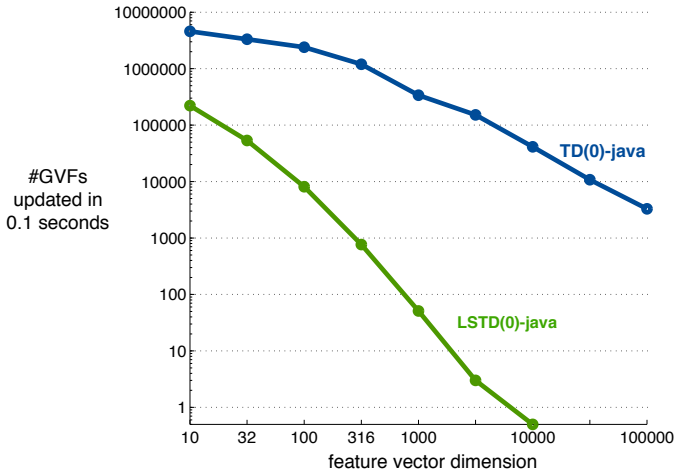

Figure 5.9: This graph shows an estimate of number of nexting predictions that could be updated in a 100 millisecond time-step, assuming perfect parallelism on a four core CPU. Note the log scale. These runtimes should be considered an approximation of what is achievable with a full prediction learning system on a robot.

5.7 Other ways to encode nexting predictions

Using a constant γ geometrically down-weights the contribution of future cumulant values in the computation of the target, just as in conventional discounting down-weights future reward in reinforcement learning. This usage of γ can be called dispersive, because it smoothly blurs the cumulants over time, whereas a non-dispersive γ might place all the weight at precisely one time-step in the future. The weighting of cumulants due to γ, follows an exponential profile. Although, not easily represented as a GVF, we might be interested in other non-exponential weightings of future cumulants, including a rectangle weighting, or a Gaussian weighting. Figure 5.10 shows several different weightings, both dispersive and specific.

In order to better access these alternative weightings, and examine their usefulness of nexting let us use each to form targets using data from the Critterbot. In Figure 5.11, we see several different targets, each with a different weighting, plotted over time based on IR beacon data from the Critterbot (no learned predictions are included, just the targets). The by-k weighting illustrates that perfect precision using non-dispersive weightings may not always be desirable, especially with low-level robot data. The target fluctuates widely from time-step to time-step, which may make accurate prediction challenging. The other weightings summarize future data in a more smooth fashion, while providing an intuitive demonstration of anticipation—predicting the rise and fall of the cumulant in advance of

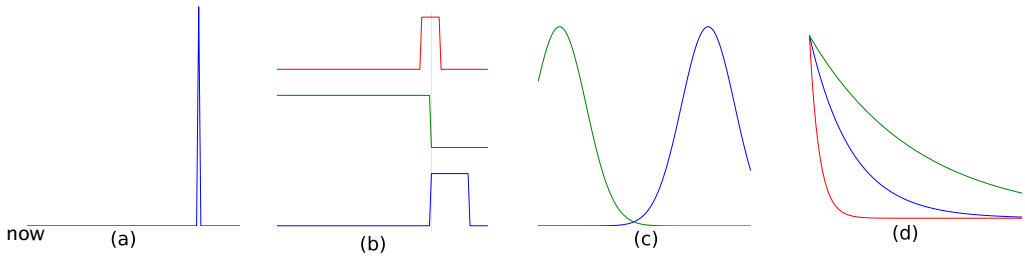

Figure 5.10: Possible weightings for future cumulants. Consider the zero on the X-axis to be the current time labelled "now". Weighting (a) corresponds to the classical prediction by $k = 80$ steps from now with no averaging or smoothing. A rectangle weighting (b) can be used to generate different windowed averages the cumulant. The red weighting results in a uniform weighting of the data cumulants 20 steps centered at 80 steps into the future. The green weighting equally averages all observed cumulants up until 80 steps into the future and the blue weighting averages a window of cumulants starting at 80 steps. Continuous weightings, like the gaussian function (c), can place more weight on the near term cumulants (green) or in the future (blue). Finally (d) the exponential weighting, produced by a constant γ, always weights near term cumulants highest and cumulants in the far future exponentially less.

changes. Interestingly, given different parameterizations, the dispersive weightings provide fairly similar summaries of future data.

Predictions formed using exponential weightings, or GVFs with constant γ, can be incrementally updated with computation independent of span. These exponential weightings have a recursive form that allows computation of the target with with constant computation and storage per step. The Gaussian weighting, on the other hand, does not have a recursive form, and must be re-computed on each time-step (Mozer, 1993). Of the weightings discussed so far, only the exponential can be updated with constant work. Nexting predictions, represented as GVFs with constant γ, can be efficiently updated and appear to provide similar representations of the target as other weightings when applied to Critterbot data.

5.8 Distinctiveness of nexting

From the perspective of conventional robotics research, there are three aspects of our nexting robot that are distinctive. The first is that the robot updates a very large number of predictions, in real time, about diverse aspects of its experience, giving it a distinctively rich awareness of its surroundings. This contrasts with the conventional approach to robot engineering, in which designers identify the minimal set of state variables needed to solve a specific task, and the robot is oblivious to all others. This approach is perhaps the source of

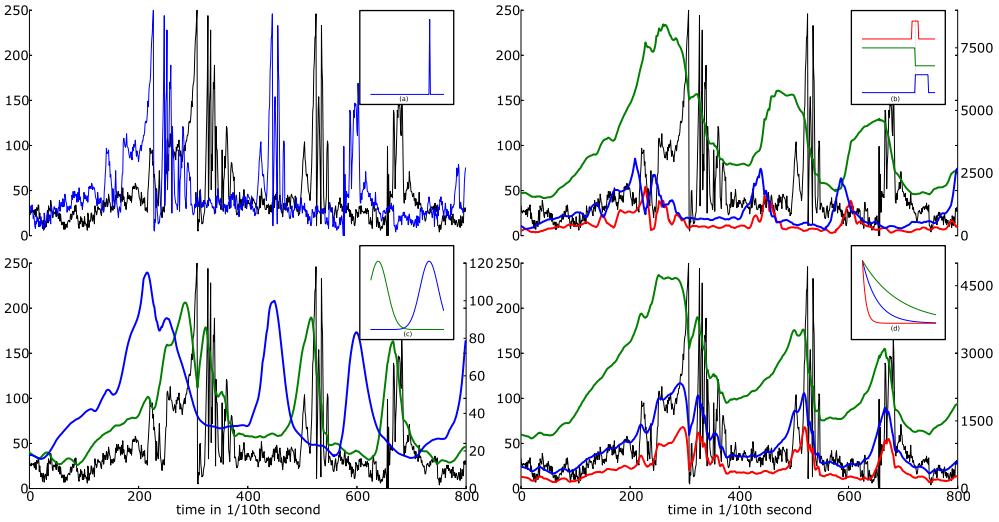

Figure 5.11: The target computed using several different cumulant weightings with Critterbot data, over time. The IR beacon sensor values are plotted in black. The IR sensor's output has three large 'humps' as the robot drives past its charger (which pulses IR light 50 times a second). The pulsing, however, also causes significant variability in the cumulant signal. The inset figures correspond to the weighting used in each target computation. In this example, we see the challenge of learning and using a prediction about a single time-step—the prediction by-k weighting. The exponential weighting—corresponding to constant γ values—smooths out the cumulant, but also provide a clear sense of anticipation. The target rises and fails in advance of change in the data series. Different time scales of prediction can be achieved with different parameterizations of the exponential weighting (i.e. different γ values). Faster decays represent near term predictions and slower decays represent longer term predictions. Finally note that the rectangle, gaussian, and exponential weightings can produce similar target plots depending on their parameter values.

the popular notion that to perform a task "like a robot" is to do it with minimal awareness and understanding.

Nowadays, it is not unusual for advanced robots to have a substantial awareness of their environment. The premier example of this is probably self-driving cars (e.g., Wang, Thorpe & Thrun, 2003; Markoff, 2010). These systems can simultaneously track many objects including cars, people, bicycles, and traffic signs. Self-driving cars are highly engineered, and the things predicted by the underlying algorithms are strongly interrelated and cover a few structured types. This contrasts with our approach to nexting, in which no prior model is used and each prediction is formed independently of the others. Our realization of nexting facilitates scaling to large numbers of sensors of arbitrary types.

The second way in which our nexting robot is distinctive is that it learns to make its predictions online, during its normal operation. Most learning robots complete their learning before being put into use, in special training sessions requiring information that will not be available during use, such as human-provided labels, demonstrations, or calibrations. Most of the learning in self-driving cars and in SLAM robots is of this sort, with important final tuning and local mapping done online. Classical state-estimation methods, such as Kalman filters, adapt only low-dimensional gain parameters online. Finally, there have been a handful of works with reinforcement learning robots that learn value functions or policies online (e.g., Peters & Schaal, 2008; Tedrake, Zhang & Seung, 2005; Degris et al., 2012). In all cases, the online learning is limited in its scale and diversity; learning a single value function.

The third way in which our nexting robot is distinctive is that its predictions are relatively long-term, extending significantly beyond a single time-step. Although prediction is widely used in modern control theory, it is almost always limited to one-step (or differential) predictions (e.g., conventional Kalman filtering (Welch & Bishop, 1995) and system identification (Ljung, 1998)). In time-series prediction, similar to nexting, one attempts to make predictions about the future values of temporally correlated data series (e.g., classical autoregressive methods). These prediction methods are typically applied off-line (after data collection), to shorter (hundreds to thousands of samples), and stationary (after preprocessing) time series. Often, one-step time series prediction models are iterated to make multi-step predictions. That can work well, but it does not scale to long time scales (not span independent) or to large numbers of predictions such as we have used here. Recent work has explored direct multistep time series prediction (Bontempi, 1999; Bontempi & Taieb, 2011; Coulibaly, 2000; Zhang & Qi, 2005; Wong, 2010, Parlos, 2000; Cheng et al.,

2006; Chevillon & Hendry, 2005). Nevertheless, time series prediction attempts to predict data at precise instances in time, as apposed to the dispersive predictions defined by a GVF's target. It is not entirely clear how to predict GVF targets with conventional time series prediction methods without computing the targets ahead of time.

Compared to previous AI research on predictive knowledge validated from experience, our work is distinctive in showing practicality and scalability with a physical robot. The supposition that knowledge might be expressed in terms of predictions has been explored by Cunningham (1972), Becker (1973), Drescher (1990), and Sutton (2009, 2012), but only in small scale simulations, and in most cases with substantial abstractions given a priori.

5.9 Summary

We have successfully implemented a robot version of the psychological phenomenon of nexting. The robot learned to predict thousands of aspects of its near future experience ten times each second. It predicted at a range of time scales, from one-tenth of a second to eight seconds. Perhaps our most important result was to show that robot nexting is not only possible, but practical. Using computationally inexpensive methods such as TD(λ), linear function approximation, and tile coding, we showed that the nexting computations easily scale to thousands of predictions based on thousands of features on a small computer. Although these algorithms are computationally cheap, they worked well. An extensive analysis of a subset of the learned predictions found them to be substantially accurate within 30 minutes of real-time training—fast enough for frequent retraining or adaptation to new sensors or environments. It is also notable that we used a single set of parameters and a single set of features for all predictions, despite variations in signal scales, signal variability, and time scales. Being able to treat all predictions uniformly in these ways facilitates the general application of nexting.

This chapter provides our first piece of evidence that our approach to predictive knowledge enables practical, large-scale learning. Our study of nexting shows that GVFs can represent an interesting and important class of predictive knowledge, and that this knowledge can be learned at scale and on a robot with a value function learning algorithm from reinforcement learning. Our results show that a predictive approach to knowledge is practical on a physical robot from the level of sensors and motors, using features that are constructed from the same.

The next chapter continues our experimentation with learning GVFs on robots. In

particular, the next chapter provides demonstrations of time-varying termination signals, cumulants that mix terminations with observed signals, off-policy updating, and learning control—all on robots.

Chapter 6

Experiments with GVFs on robots[1]

The nexting experiment of the previous chapter illustrated how GVFs capture an interesting class of predictions, and that those GVFs can be learned accurately and at scale from data generated by a mobile robot. The aim of this chapter is to provide additional empirical evidence that GVFs and Horde increase the applicability and practicality of predictive knowledge learning. Specifically, this chapter contains experiments illustrating learning both state and state-action GVFs with more complex, time-varying termination signals. These GVFs were learned with a variety of feature generation schemes, with three different learning algorithms, on two different robot platforms, and from data generated by wall following, simple hand-coded behaviors, and human tele-operated behaviors. The results indicate our reinforcement learning-based approach to predictive knowledge learning works across a range of feature representations, parameters, questions, and goals. This chapter contains a series of prediction demon experiments, and one off-policy control-demon learning experiment. The chapter concludes with a discussion of other demonstrations of GVF learning from the literature.

The experiments in this chapter illustrate some of the possibilities for specifying GVFs. In many cases the examples are designed to have a straight forward objective meaning to make it easier for the reader to follow. Whereas our robots could certainly learn predictive questions with little objective to meaning to humans. The topic of automatically generating GVFs is beyond the scope of this work.

[1]The experimental results and text of this chapter were both adapted from co-authored sections of several papers (Sutton et al., 2011; Modayil, White, Pilarski & Sutton, 2012b; Modayil, White & Sutton, 2012; and Modayil, White & Sutton, 2014) or originally generated for this document.

6.1 Experiments with more complex terminations

A GVF's ability to represent interesting predictions about the world is related to the definition of the cumulant and termination signals. In our nexting experiments, we investigated the case where γ is taken to be a constant, and less than one. With a constant termination signal, predictions are restricted to simple time scales in which cumulants are weighted geometrically less the longer they are delayed. In this section, we move beyond these simple time scales, and describe variable termination signals.

The aim of this section is to apply and demonstrate these generalized terminations with TD(λ) in three examples of nexting on a robot. All three experiments involved applying TD(λ) to the log of data collected during the nexting experiment of the previous chapter. Therefore, all three examples will make use of prediction demons and on-policy updating.

Up to now, the termination signal, $\gamma^{(i)}$, has been varied only from prediction to prediction; for the ith prediction, $\gamma^{(i)}$ was constant and determined its time scale. Now we will consider the case where the termination signal for an individual prediction varies over time depending on the state the robot finds itself in. We can rewrite the definition of the target given in Equation 4.2 of Chapter 4, to include both terminations that vary with time and multiple GVF indexed by $i = 1, 2, \ldots$:

$$G_t^{(i)} \;=\; \sum_{k=0}^{\infty} \left(\Pi_{j=1}^{k} \gamma_{t+j}^{(i)}\right) Z_{t+k+1}^{(i)}, \tag{6.1}$$

and our ith prediction, as before, is $V_t^{(i)} \;\approx\; \mathbb{E}_\pi[G_t^{(i)} | S_t = s]$. Technically, γ changes with time because it is a function of state.

Event triggered termination

In our first experiment, consider a termination signal that is usually constant and near one, but falls to zero when some designated event occurs:

$$\gamma_t^{(i)} = \begin{cases} 0 & \text{if Light3 is saturated at time } t; \\ 0.9875 & \text{otherwise.} \end{cases}$$

As long as Light3 is not saturated, this termination signal works like a simple eight-second time scale—cumulants are weighted by 0.9875 carried to the power of how many steps they are delayed. But if Light3 becomes saturated, then all cumulants after that time are given zero weight. This kind of termination enables us to predict how much of something will occur prior to a designated event (in this case, prior to Light3 saturation).

The cumulant, in this first experiment, is a measure of the total power consumption of the Critterbot's three motors,

$$Z_t^{(i)} = \sum_{j=1}^{3} |\text{MotorVoltage}_t(j) \times \text{MotorCurrent}_t(j)|.$$

As shown in Figure 6.1, power consumption tended to vary between 1000 and 3000 depending on how many motors were active. The target G_t (defined in Equation6.1), also shown in Figure 6.1, was similar to that of a simple eight-second time scale for much of the time, but notice how it falls all the way to zero during Light3 saturation. Even though there was substantial power consumption within the subsequent eight seconds, this has no effect on the target because of the saturation triggered termination.

The target policy was the same as in nexting, the hand-designed wall following policy. The answer functions are the same as the ones used in the nexting experiment: the wall following behavior policy (equal to the target policy, and thus on-policy), tile coding features, and $\lambda = 0.9$.

Figure 6.1 shows that the TD(λ) demon performed well (after training on the previous 150 minutes of experience): over the entire data set the predictions captured approximately 88% of the variance in the target.

Predicting outcomes

The target G_t in the above experiment, like those of simple time scales, always weights delayed cumulants less than immediate ones. It cannot put higher weight on the cumulants received later than it puts on those received immediately. This limitation is inherent in the definition of the target (Equation 6.1) together with the restriction of the termination signal to $[0, 1]$. However, it is only a limitation with respect to the cumulant; if signals are mixed into the cumulant in the right way, then predictions about the signals can be made with different temporal profiles. Our next experiment demonstrates one possible mixture.

This experiment demonstrates how a demon can predict the value of a signal at the time some event occurs. Suppose we have some signal O_t whose value we wish to predict, not in the short term, but rather at the time of some event. To do this, we define a termination signal $\gamma_t^{(i)}$ that is one up until the event has occurred, then is zero. The cumulant is then constructed as follows:

$$Z_t^{(i)} = (1 - \gamma_t^{(i)})O_t \tag{6.2}$$

This cumulant is forced to be zero prior to the event (because $1 - \gamma_t^{(i)}$ is zero) and thus nothing that happens during this time can affect the target. The target will be exactly the

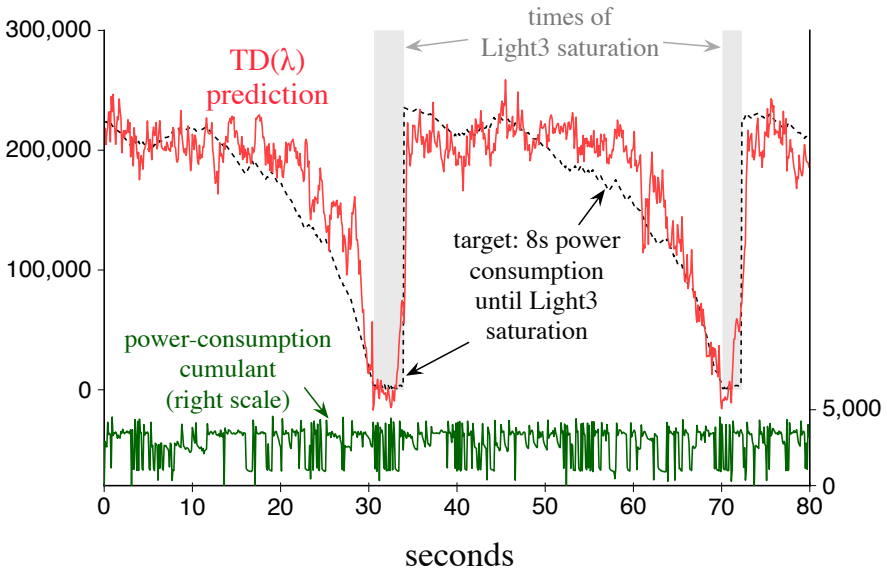

Figure 6.1: A demon's prediction of total power consumption over an eight-second time scale or up until the Light3 sensor value is saturated. This corresponds to how much power will be used to reach light saturation or the effective end of the time horizon. To express this kind of prediction, the termination signal must vary with time (in this case dropping to zero upon Light3 saturation).

value of O_t at the time $\gamma_t^{(i)}$ first becomes zero. The definition of the target remains the same as Equation 6.1.

For our second experiment, we made O_t equal to the indicator function for an event (equal to one during it, and zero otherwise) and $\gamma_t^{(i)}$ is a constant less than one prior to the event (and zero during the event). In this configuration, the prediction will be of how imminent the onset of the event is. More precisely, we specified a single demon with the question functions defined as: $\gamma_t^{(i)}$ prior to an event was 0.8 (corresponding to a half-second time scale) and 0 otherwise, the cumulant defined in (6.2) with O_t equal to one during the event and zero otherwise, and the target policy the same as before. The event was defined as the right-facing IR sensor exceeding a threshold (corresponding to being within 12cm of the wall). The answer functions were the same as the previous experiment. Figure 6.2 shows results of the demon's learning using the data from the Critterbot.

Soft termination

Constructing the cumulant by (6.2) has several possible interpretations depending on the exact form of O_t and $\gamma_t^{(i)}$. In our final experiment of this section, shown in Figure 6.3, we

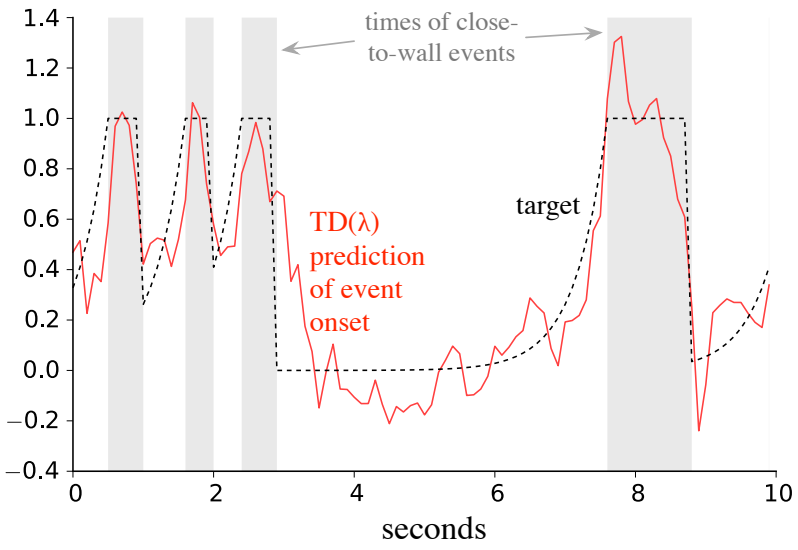

Figure 6.2: Predictions of the imminence of the onset of an event regardless of its duration. The event here is being too close ($<\approx$12cm) to a side wall of the pen, and imminence is with respect to a half-second time scale. The demon's learned predictions rise before the event and follow the shape of the target. Overall, the normalized mean squared error of this prediction was 23%.

illustrate the use of signals O_t that are not binary and termination signals $\gamma_t^{(i)}$ that do not fall all the way to zero—a soft termination. The idea here is to predict what the four light-sensor readings will be as the robot rounds the next corner of the pen. Four demons are specified—one for each light sensor. The question functions of each demon were defined as follows. The outcome signal is equal to the sensor reading: $O_t^{(i)} = \text{Light}^{(i)}$. The termination signal, $\gamma_t^{(i)}$, of each demon is set equal to 0.9875 (an eight-second time scale) unless the robot rounded a corner, during which time the termination signal changed to 0.3. Because the termination signal is greater than zero during the event, the light readings from several time steps contribute to the target, $G_t^{(i)}$, as the corner is entered. Rounding a corner is an event indicated by a value from the side IR distance sensor corresponding to a large distance ($>\approx$25cm). This typically occurs for several seconds during the corner turn's. The target policy is equal to the behavior, which completes our specification of the question functions. Figure 6.3 shows each demon's prediction, learned from the nexting log file. In this experiment, the future cumulants that are observed in the next corner the robot will encounter, have the largest contribution to the target. The target gives non-zero weight to only a few cumulants beyond the next corner, but not so much that the target would include

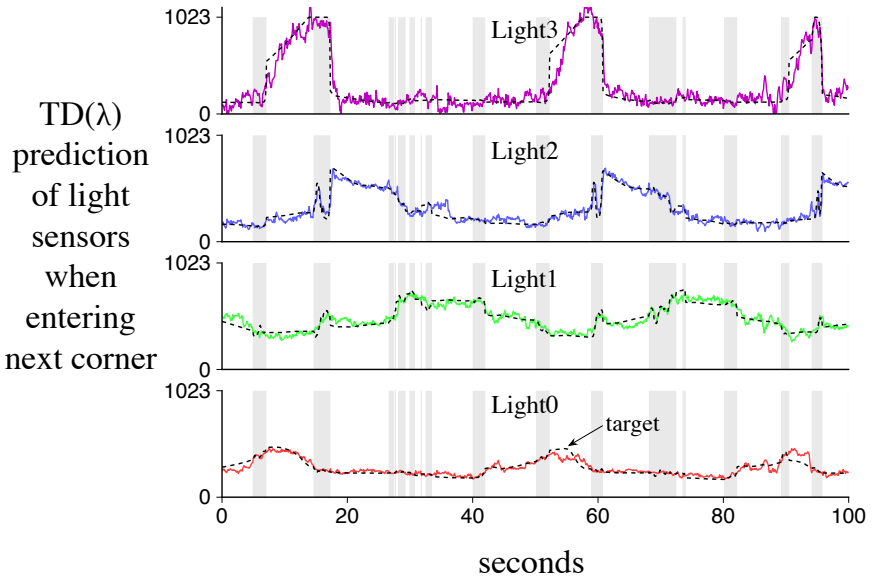

Figure 6.3: Predictions of what each of the four light-sensor outputs will be when the robot rounds its next corner. The greyed time steps indicate those in which the robot was considered to be rounding a corner. Because the termination signal is greater than zero during the event, the light data from several time steps contribute to the target as the corner is entered. The normalized mean squared errors for the four predictions were 6%, 10%, 10%, and 11% over the course of the data set.

light two corners ahead in time.

6.2 Off-policy prediction learning

The experiments of the previous section were all learned via on-policy TD(λ). In this section, we provide four examples of off-policy prediction learning on two different robots: the Critterbot and the iRobot Create. These examples demonstrate realtime learning of both state-action GVFs with GQ(λ), and state GVFs with GTD(λ).

The remaining experiments in this chapter were conducted before the nexting experiment of the previous chapter. At the time these experiments were conducted we were just beginning to understand how to effectively use the new gradient-TD methods on robots. In the following experiments you will notice how we used different time-step, feature vectors, and learning parameters in each experiment as we sought to find the best ways to achieve stable learning. As a result of the nexting experiment and the experiments described in Chapters 7 and 8, we have a clearer understanding of how to set the parameters for off-policy learning on robots.

6.2.1 Experiments on the Critterbot

We performed two experiments to examine demon learning with GQ(λ) on the Critterbot.[2] In both experiments, the robot's sensors and actions were tiled to form a state-action feature vector $\mathbf{x}_a$. A discrete set of actions were selected. Both experiments involve a single prediction demon. The time-step used in these experiments was approximately 30 ms.

Our first off-policy experiment involved a single demon that might be useful in ensuring safety: "How much time will pass before the robot hits an obstacle, if the robot drives straight forward?". The question functions for this demon were: $\pi(s, \text{FORWARD}) = 1 \; \forall \; s \in \mathcal{S}$, $Z_t = 1$, and $\gamma_t = 0$ if the value of the Critterbot's front-pointing IR proximity sensor was over a fixed threshold, else $\gamma_t = 1$. The answer functions were $\lambda = 0.4$ and the features were generated from a single tiling into twenty-six regions of the robot's front IR sensor (therefore $\mathbf{x}_a$ has a single active component). The behavior policy cycled between three actions: driving forward, driving in reverse, and resting. The GQ(λ) step sizes were $\alpha = 0.3$ and $\alpha_{\mathbf{w}} = 0.00001$.

Figure 6.4 shows a comparison between predicted and observed time steps needed to reach obstacles (the wall) when driving forward. Shown is the demon's prediction Q_t on each visit (bold line) and the target from that step, G_t (thin line), in one of three regions. Each of the three regions corresponds to non-overlapping discretization of the the front IR sensor. The regions were only used for evaluation and visualization purposes—only a single prediction demon was used. A visit to a region corresponds to a time-step in which the IR sensor reading was within the corresponding range. Note as the robot drives forward and passes through a region multiple visits are recorded (both the current prediction and a sample of the target) and plotted in Figure 6.4.

Our second off-policy experiment also involved a single demon that might be useful in ensuring robot safety: "How much time does the robot need to stop?". We defined a single demon to predict the number of time steps until one of the robot's wheels approaches zero velocity (i.e., comes to a complete stop) under current environmental conditions.

The question functions for this demon were: $\pi(s, \text{STOP}) = 1 \; \forall \; s \in \mathcal{S}$, $Z_t = 1$, and $\gamma_t = 0$ 0 if the wheel's velocity sensor was below a fixed threshold, else $\gamma_t = 1$. The answer functions were $\lambda = 0.1$ and the features were generated from a single tiling into eight regions of the wheel's velocity sensor. The behavior policy alternates at fixed intervals between spinning at full speed and resting. The floor surface, and thus the nature of the

[2] Both experiments were run by Patrick Pilarski.

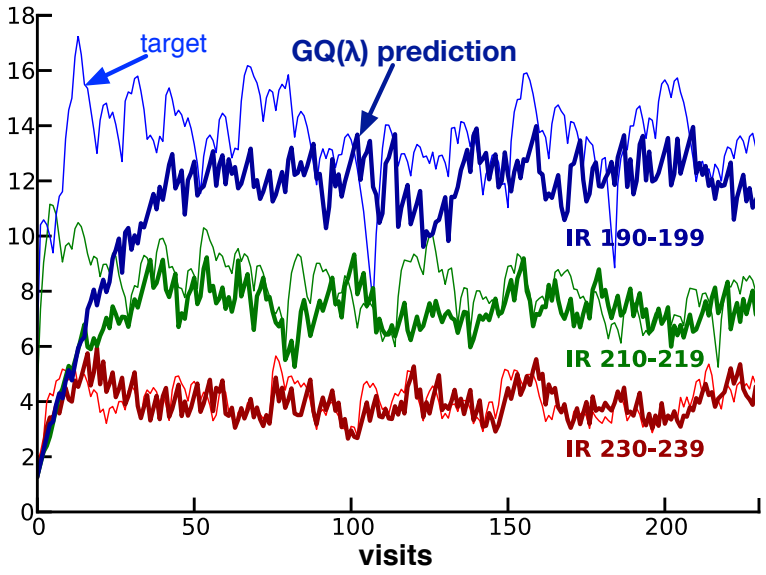

Figure 6.4: Predicting time-to-obstacle on the Critterbot. The robot was repeatedly driven toward a wall at a constant wheel speed. For each of three regions of the sensor space, for each time-step spent in that region, we plot the prediction, $Q_t = \hat{q}(S_t, A_t, \mathbf{w})$, on that step (bold line) and the target from that step (thin line).

stopping problem, was changed after visits 338 and 534. The GQ(λ) step sizes were $\alpha = 0.1$ and $\alpha_{\mathbf{w}} = 0.001$.

Figure 6.5 demonstrates a demon's ability to accurately predict stopping times on different surfaces. Shown is the prediction $Q_t = \hat{q}(S_t, A_t; \pi, \gamma, z)$ made on visits to a region of high velocity while stopping (bold line), together with the target computed from that visit (thin line). As illustrated in Figure 6.5, this demon learned to correctly predict the target (time steps to stopping) on carpet, then adapted its prediction when the surface was changed to air and then changed to wood flooring.

6.2.2 Experiments on the Create

We performed two experiments to examine GVF learning with GTD(λ) on the iRobot Create. In both experiments, the robot's sensors and actions were tiled to form a state feature vector $\mathbf{x}$. A discrete set of actions were selected, matching the formulation of the GTD(λ) algorithm. The first experiment involves a single prediction demon, and the second experiment involves several prediction demons learned in parallel. The time-step used in these experiment was approximately 0.5 seconds in length.

The previous experiments of this chapter provide a quantitative assessment of demon prediction learning. In the following two experiments we take a qualitative approach to

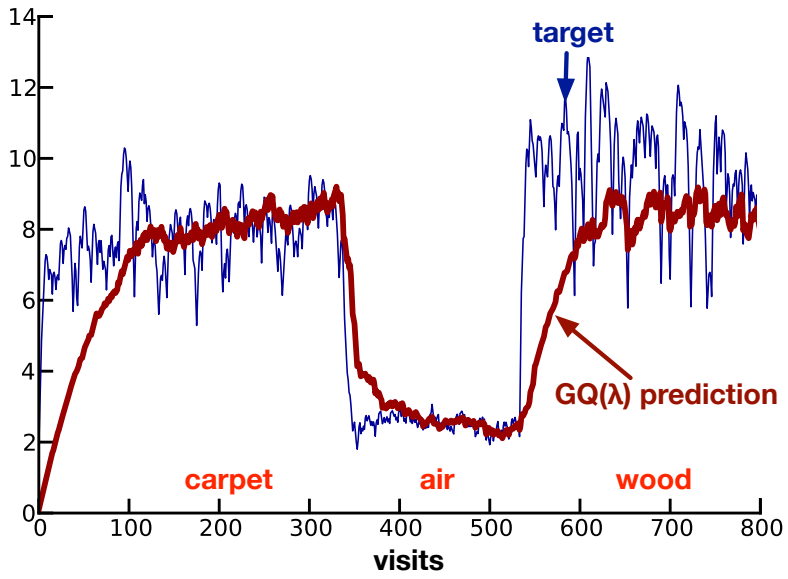

Figure 6.5: Predicting time-to-stop on the Critterbot. The robot was repeatedly rotated up to a standard wheel speed, then switched to a policy that always took the STOP action, on each of three different floor surfaces. Shown is the prediction Q_t made on visits to a region of high velocity while stopping (bold line) together with the target computed from that visit (thin line). The floor surface was changed after visits 338 and 534.

evaluating prediction learning, specifically we will visually inspect and assess each prediction to evaluate each prediction's correctness during a teleoperation phase after learning is complete. This style of prediction evaluation is of interest because a quantitative measures, during learning, may not always represent the whole story. For example, the predictions may behave erratically or not as expected in new situations, even if a quantitative measure indicates low prediction error during learning. By visualizing the learning in a qualitative way, we can further validate the prediction accuracy claims from the quantitative measures.

Predictive feature components

Our first Create prediction experiment was designed to illustrate how a demon's prediction can participate in the generation of the feature vector. This experiment's demon was also related to robot safety, and can be loosely interpreted as: "How imminent is bumping, if the robot were to drive toward the wall?". The question functions for this demon were: a target policy that always selects the forward action , Z_t equal to one if either bumper is activated and zero otherwise, and γ_t equal to zero if the value of the Create's front bumper sensor was activated, and otherwise $\gamma_t = 0.8$ corresponding to a 2.5 second time scale. The robot always remained essentially perpendicular to the wall (due to the specification of the

behavior policy, described below), and thus driving forward will eventually lead to a bump while driving backward will not induce a bump.

Unlike our previous experiments, the feature vector in this experiment includes predictive components. The feature vector $\mathbf{x}_t$ is a binary vector with two components which encode if bumping was detected or not, three components to encode the previous time-step's action (forward, backward, or pause), 128 components produced by tile coding the demon's prediction of bumping from the previous time-step, and a bias unit. Specifically, 128 components of $\mathbf{x}_t$ were produced by tile-coding $V_{t-1} = \hat{v}(S_{t-1}, \mathbf{w})$ with eight one-dimensional tilings into 16 regions. Overall $\mathbf{x}_t$ contains 134 binary components, and 11 components are non-zero on each time-step. Due to the Create's limited sensory apparatus directly sensing the distance to the wall is not often possible. This feature vector was used to disambiguate the robot's current situation.

The remaining answer functions were $\lambda = 0.9$, and the behavior policy was fixed, probabilistically switching between driving forward (60%) and driving backward (40%), and thus keeping the robot near the wall. The behavior never included rotation actions. The robot's movement with respect to the wall can be seen in the screen captures in Figure 6.6. The GTD(λ) algorithm's learning-rate parameters were set to $\alpha = 0.1/11$ and $\alpha_{\mathbf{w}} = 0.0001$.

In order to test the accuracy of the robot's prediction, we controlled the robot with a remote and recorded the predictions. The prediction demon learned an estimate of the imminence of the onset of a bump from approximately three minutes of interaction generated by the behavior policy. After learning, the demon's learning was disabled and the robot was tele-operated for approximately two minutes. The robot was driven toward and away from the wall, and the demon's predictions were recorded. Figure 6.6 shows several representative frames from a video of the robot under human control. The demon's prediction of imminence of the onset of bumping is included with each frame along with an indication if bumping was detected. The intensity of the color of the prediction represents the magnitude of the demon's prediction.

We conclude from the results in Figure 6.6 that the demon's predictions of the imminence of the bumping are substantially correct. Here we take correctness to mean matching our human expectations of how the prediction should change with the robot's actions. The prediction is high when the robot is near the wall, low when the robot is far from the wall, and rises and falls as the robot moves forward and backward. Thus, the predictions match our expectations.

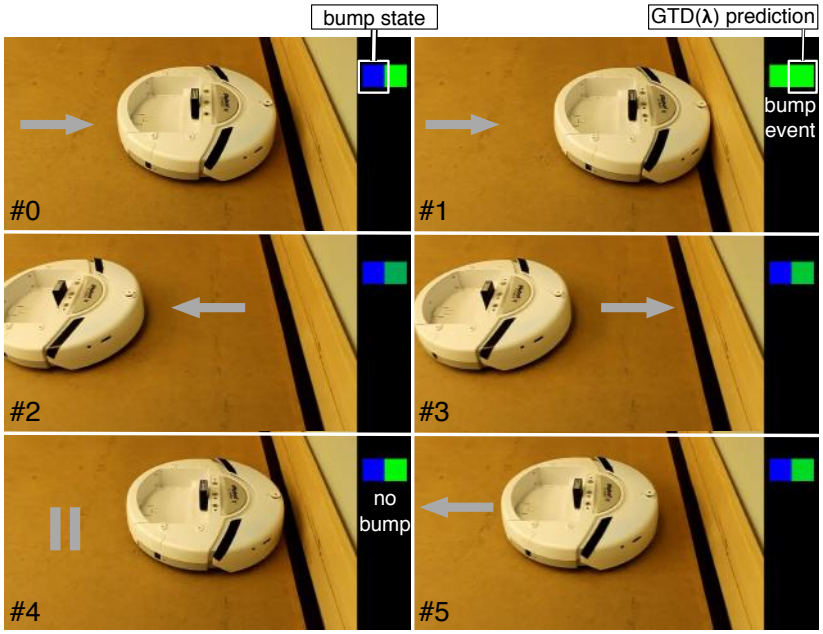

Figure 6.6: Predicting the imminence of the onset of a front bump on the iRobot Create. This figure includes a set of representative frames gathered while testing a single demon's predictions. Each of the six frames above includes a video still of the robot under human remote control, a visualization of whether one of the bumpers was active (lefthand box labelled 'bump state'), a visualization of the demon's prediction (righthand box labeled 'GTD(λ) prediction'), and a visualization of the current action (forward and backward with a gray arrow, no action with gray pause symbol). For the bump state, blue indicates no bump detected and green indicates bump detected. For the predictions, blue indicates low prediction magnitude, green indicates high prediction magnitude, and blue-green indicates an intermediate prediction magnitude. Above, we see the robot driving toward the wall (frames #0, #1, #3) and the demon's prediction rising in anticipation of bump. As the robot backs away from the wall (frames #2 and #5) the prediction falls. A bump event occurs in frame #1. Even on an interrupted attempt (frame #4), the prediction recedes as the robot then begins to back away from the wall (frame #5).

Multiple predictions

Our second Create prediction experiment was designed to illustrate multiple prediction learning in parallel. In this experiment, we used three prediction demons corresponding to the robot's orientation with respect to the wall, and can be loosely interpreted as: "How imminent is bumping in the current configuration, with respect to each bump sensor, if the robot were to drive forward?". In this experiment, the robot was allowed to rotate, changing its orientation with respect to the wall. Depending on the robot's orientation, driving forward will cause a left bump, right bump, left and right bump, or no bump.

Each GVF was about a different bump sensor: left bumper, right bumper, and both

bumpers. The question functions corresponding to the first demons were: $\pi^{(1)}$ continuously selects the forward action, $Z_t^{(1)}$ equal to the binary left bumper reading, and $\gamma_t^{(1)}$ equal to zero if the left bumper was activated, else $\gamma_t^{(1)}$ equal to 0.5 corresponding to a 0.5 second time scale. The second demon's question functions were defined in a similar way: $\pi^{(2)}$ equal to the forward policy, $Z_t^{(2)}$ equal to the right bumper reading, and $\gamma_t^{(2)}$ equal to zero if the right bumper was active, and 0.5 otherwise. Finally, the third demon's question functions were: $\pi^{(3)}$ equal to the forward policy, $Z_t^{(3)}$ equal to ones and $\gamma_t^{(3)}$ equal to zero if either left or right bumper was active, and $Z_t^{(3)}$ equal zero and $\gamma_t^{(3)}$ equal to 0.5 otherwise.

All three prediction demons used the same answer functions. The robot's behavior policy in this experiment was generated by human tele-operated control. The robot was controlled with a remote control switching between rotating back and forth and driving forward, producing a three minute log of data. Because the behavior was generated by human inputs, we set $\mu(S_t, A_t)$ equal to one. [3] During learning, the Create remained close to the wall, never facing away from the wall. The downward-facing IR sensor's reading of a black strip of tape along the ground near the wall provided the robot with a noisy indication of the robot's current orientation (electrical tape gives a distinctive IR output compared to the rest of the floor). The feature vector included six two-dimensional tile codings of all pairwise combinations of the four IR sensors, each using four tilings into eight regions of the downward IR sensors. The feature vector also included four components encoding the state of the left and right bump sensors, four components encoding the previous time-step's action (forward, backward, rotate right, rotate left), and a bias unit. In total the feature vector contained 1545 components, 27 of which were equal to 1 on at each time-step. The λ function was set to a constant value of 0.9. The parameters to GTD(λ) were $\alpha = 0.1/27$ and $\alpha_h = 0.0001$. We trained the three prediction demons off-line and off-policy in five passes over the training set.

Figure 6.7 shows several representative frames from a video of the robot under human tele-operation in various configurations near the wall, the current bump readings, and the three demon predictions. As before, we tested the predictions by controlling the robot by remote, while recording the predictions, after learning. The predictions were not perfect; for example the prediction in frame #3 was not 1.0, and the predictions when the robot was facing away from the wall were not precisely zero. Overall, we conclude that the three predictions are substantially correct, illustrating that the demons can predict imminence of the

[3] This choice technically introduces bias, causing the algorithm to converge to a different solution. In this case the final predictions were reasonable, but this is more generally a challenge due to using behavior generated by human control.

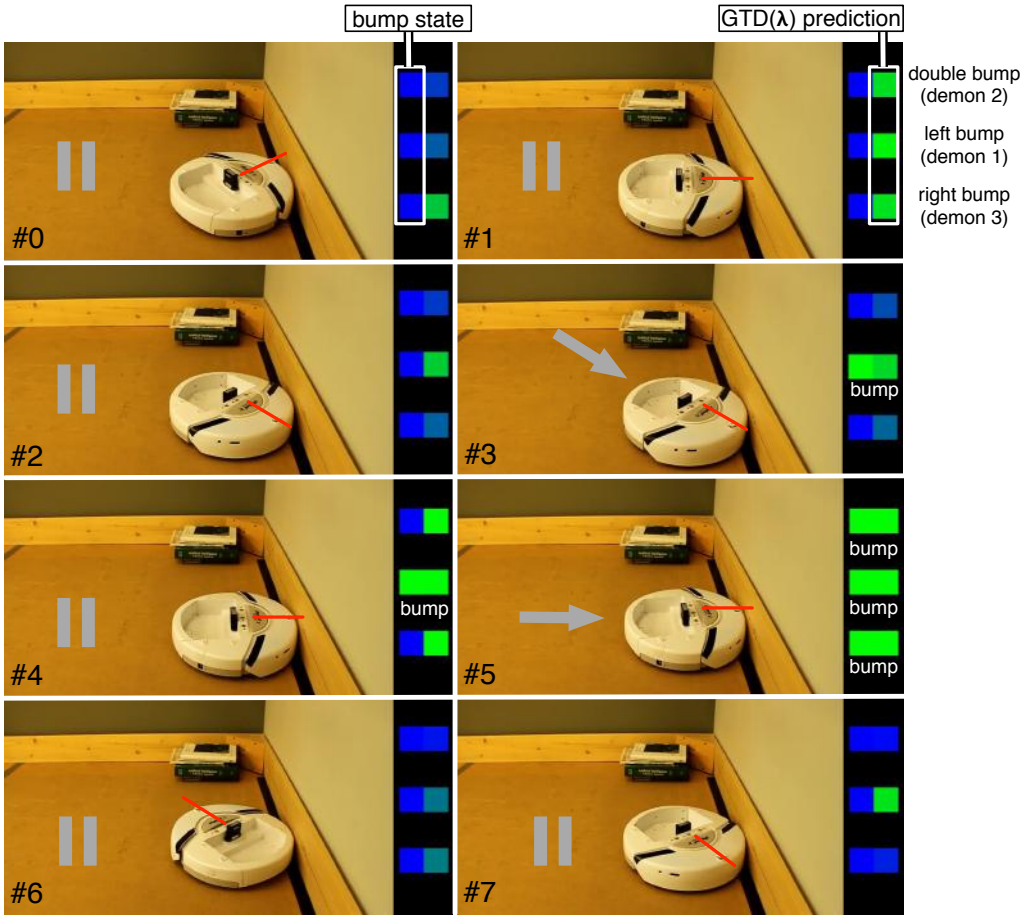

Figure 6.7: Three demon's predictions of the imminence of the onset of a front bump learned on the iRobot Create. This figure includes a set of representative frames gathered while testing the robot's prediction. Each of the seven frames above includes a video still of the robot under human remote control, a visualization of current bump state (left three boxes), a visualization of each demon's prediction (right-hand three boxes labeled 'GTD(λ) prediction'), and a visualization of the current action with gray arrows and pause symbols. The orientation of the robot is marked with a red line. The blue and green colors have the same interpretation as they did in Figure 6.6. Above we see the robot bumping into the wall (frames #3, #4 and #5), facing the wall and high prediction of imminence of bump onset if the forward action were taken (frames #0, #1, #2 and #7), and facing away from the wall with a low prediction of imminence of bump onset (frame #6).

onset of bumping according to each bump sensor in a variety of positions. More generally these results indicate that multiple GVFs can be learned off-policy from robot data, and that those predictions are reasonably accurate when tested in a variety of configurations, different from the training data.

6.3 Off-policy control learning

Our final experiment in this chapter examined whether a control demon, using GQ(λ), could learn a goal-directed policy when given a much greater breadth of experience. The objective was to learn how-to knowledge about maximizing the light sensor reading. We used a large discrete action set with 27 possible actions corresponding to velocities for the robot's three wheels (e.g., $\{\{$10,10,10$\}$, $\{$10,10, 0$\}$, ..., $\{$-10,-10,-10$\}\}$). The time-step corresponded to approximately 500 ms.

The question functions corresponding to the goal of maximizing the near-term value of the front light sensor of the Critterbot (Light0): $\pi = greedy(\hat{q}), \gamma_t = 0.9, Z_t$ equal to a scaled reading from Light0. The answer functions were $\lambda = 0.9$, a behavior policy that picks randomly from the set of 27 actions, and the state-action feature vector was produced by tile coding the light sensor readings. In particular, 32 individual tilings over each of the four light sensors, where each tile covers about 1/8th of the range. With the addition of a bias unit, this made for a total of 27, 675 binary features components, of which 129 were active on each time step. The parameters to GQ(λ) were $\alpha = 0.1/129$ and $\alpha_{\mathbf{h}} = 0.0$.[4]

Using the random behavior policy, we collected a training set of 61,200 time steps (approximately 8.5 hours) with a bright light at nearly floor level on one side of the Critterbot's pen. During this time, the robot wandered all over the pen in many orientations. We trained the control demon off-line and off- policy in two passes over the training set. To assess what had been learned, we then placed the robot in the middle of the pen facing away from the light and gave control to the demon's learned policy. The robot would typically turn immediately and drive toward the light, as shown in the first panel of Figure 6.8. We evaluated the learned policy with eight independent runs, four with the light on the south side of the pen and four with the light on the west side of the pen. In all eight trials the robot successfully navigated to and stayed at the light.

We conclude that the results of this experiment demonstrate that a demon can learn an effective goal-directed behavior from substantially different training behavior.

[4]Setting $\alpha_{\mathbf{h}} = 0.0$ means the secondary weights are not used in the GQ(λ) update. In practice we have found that very small $\alpha_{\mathbf{h}} = 0.0$ results in faster learning compared to $\alpha_{\mathbf{h}}$ values closer in magnitude to α. The experiments of Chapter 7 show that $\alpha_{\mathbf{h}} = 0.0$ can lead to divergence, and thus setting $\alpha_{\mathbf{h}}$ very small is a dangerous choice for a predictive knowledge aquisition. Nevertheless, at the time this experiment was conducted this setting produced the best result for the particular GVF we specified.

Figure 6.8: Learning light-seeking behavior from random behavior. Shown are superimposed images of robot positions: Left) In testing, the robot under control of the target policy turns and drives straight to the light source at the bottom of the image; Middle) Under control of the random behavior policy for the same amount of time, the robot instead wanders all over the pen; Right) Light sensor outputs averaged over seven such pairs of runs, showing much higher values for the learned target policy.

6.4 Other demonstrations of GVF learning

The experiments described in this Chapter and our paper (Sutton et al., 2011) were the first empirical demonstrations of GVF learning, but they also inspired several other empirical studies on GVF learning. This section summarizes these other works, adding further supporting evidence of the usefulness of GVFs as a language for predictive knowledge and the evidence of the practicality of learning GVFs with reinforcement learning methods.

On-policy GVF learning has been explored in other robot domains, demonstrating concrete benefits of predicting what will happen next. In the domain of adaptive prosthetics, GVFs have been used to represent predictions about a multi degree-of-freedom robotic arm (Pilarski et al., 2012). Each prediction was represented as a constant termination GVF and learned on-policy and in parallel with the TD(λ) algorithm and tile coding function approximation. After learning, the predictions were then used to adapt the switching order of the robot arm to significantly improve upon task completion efficiency. Later work showed that these nexting predictions can be learned online while a human amputee generated the behavior data, significantly improving the amputee's user experience (Edwards et al., 2014). This adaptive prosthetics work is important because it provides another demonstration of our basic learning scheme—linear temporal difference learning and large binary feature representations—working on a very different robot platform. Their work also goes beyond the experiments of this chapter, demonstrating a concrete application of GVF learning: improving the performance of switching time. Our experiments (in this chapter and Sutton et

al., 2011), remain the only ones combining off-policy GVF learning on robots.

General value function learning has also been studied in simulated robot domains. For example, nexting predictions of laser welding-process sensors were learned in simulation from a multi-level representation computed from simulated camera images (Gunther et al., 2014). General value function learning has also been explored in RoboCup 3D soccer simulation league environment: where a set of control demons were used to learn the role assignment of each robot in order to maximize winnings of the team (Abeyruwan et al., 2014).

Beyond the results of this chapter, the practicality of learning GVFs and representational abilities of GVFs have been demonstrated in small simulation worlds. The work of Degris et al. (2012) contains the most extensive investigation of control demon learning, with four different off-policy reinforcement learning methods compared in three continuous-state domains. The results of Degris's study indicate that control policies learned off-policy from randomly generated behavior can effectively solve several simulated cost-to-go problems. Recently, multi-step predictions represented as GVFs were compared to PSRs for representing partially observable domains (Schaul & Ring, 2013). Specifically, the study compared estimating a state-action value function for navigating a discrete maze with features computed from a set of GVF predictions and features computed from a set of PSR predictions. The results showed that predictive features based on GVF predictions were more useful than predictive features based on PSR features, although the predictions were given not learned.

6.5 Summary

This chapter presented several demonstrations of GVF learning on robots: three on-policy prediction learning experiments on the Critterbot, two off-policy prediction learning experiments on the Critterbot, two off-policy prediction learning experiments on the Create, and one off-policy control learning experiment on the Critterbot. The chapter closed with an discussion of experiments with GVF learning from the literature.

The aim of this chapter was to demonstrate how many different predictions can be formulated as GVFs, and that these GVFs can be practically learned from samples generated by a robot. To achieve this goal we conducted several robot experiments demonstrating demon learning with Horde. Our experiments covered a wide variety of configurations including on-policy updating, off-policy updating, learning state GVFs, learning state-action GVFs, learning policies, different feature representations, predicting many different signals

with time varying termination, and three different temporal difference learning methods, on two different robots. In each configuration, the approximate GVFs were learned to acceptable levels of accuracy using simple linear function approximation. The results of this chapter provide evidence of the contribution of GVFs and Horde towards developing predictive knowledge.

Chapter 7

Experiments with gradient-TD learning

Gradient-TD methods make exploring parallel demon learning practical for the first time, but our empirical experience with these new algorithms is limited. Gradient-TD methods, in theory, have all the right attributes for large scale prediction learning, but like any new algorithm, experimental validation is needed.[1] This chapter contributes new experimental results and insights to the growing body of experimental work on linear-complexity, off-policy reinforcement learning (see Sutton et al., 2009; Delp, 2010; Hackman, 2012; Geist & Scherrer, 2014; Dann et al., 2014).

The experiments of this chapter focus on the GTD(λ) algorithm. The GTD(λ) algorithm, as our experiments in Chapter 6 show, can be effective for learning state GVFs on robots. To improve our understanding, we test the GTD(λ) algorithm in more controlled empirical settings, which is difficult to do on a robot. In particular, our experiments investigate the role of the secondary weight vector of the GTD(λ) algorithm. Prior studies have cast some small doubt on the benefit of the secondary weights in practice (specifically Delp, 2010; Hackman, 2012; Degris et al., 2012), suggesting that in some domains, a large learning-rate for the secondary weights, $\alpha_{\mathbf{h}}$, can slow learning. Our experiments investigate how well the GTD(λ) algorithm learns its secondary weights in a Markov chain domain and Baird's counterexample. Our experiments also investigate how well the GTD(λ) algorithm can minimize the MSPBE when the secondary weights are ignored, when the secondary weights are replaced by their correct (unlearned) values, and when the secondary weights are learned in the usual way. Taken together, the experiments of this chapter provide new

[1] Appendix B contains a discussion of other GVF learning methods besides ones that minimize the MSPBE and a summary of the evidence for and against the MSPBE as an objective function for GVF learning.

insights into how the performance of the GTD(λ) algorithm relates to variations in its parameter values.

All the experiments of this chapter focus on state GVF estimation and linear function approximation. We begin with experiments in a Markov chain problem and then move to Baird's counterexample.

7.1 Experiments on Markov chains

We begin our experimental exploration of GTD(λ) and its secondary weight vector on the Markov chain problem, previously used in other empirical explorations of gradient-TD learning methods (e.g., Sutton et al., 2009; Maei, 2011; Hackman, 2012).

7.1.1 Problem

We use a Markov chain task, depicted in Figure 7.1, to perform experiments to better understand secondary weights of the GTD(λ) algorithm. The chain has two actions, *left* and *right*, and two terminal states at the ends of the chain, with the left termination producing a cumulant of -1 and the right termination producing a cumulant of +1. Cumulant values on all other transitions are zero. Upon entry into one of the termination states, the agent is teleported to the middle state in the chain.

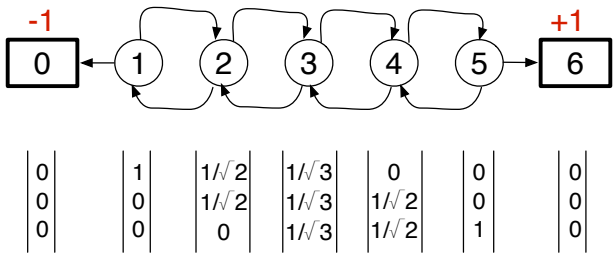

Figure 7.1: The Markov chain domain with discrete states and actions. The nonzero cumulants are labelled in red, and the feature vector corresponding to each state is given below each state.

The learning problem is specified by the three question functions. The target policy selects the *right* action with probability p, and the *left* action with probability $1 - p$. The cumulants are given by the definition of the problem, and γ_t becomes zero upon entry into either terminal state and is 1.0 otherwise.

The answer functions are: λ is a constant function equal to 0.9, and the behavior policy,

like the target policy, takes the *right* action with probability p and the *left* action with probability $1-p$. Depending on the value of p for the target and behavior policies, an instance of the chain problem can be on- or off-policy. The feature vectors are given in Figure 7.1. Note that the feature vector is of lower dimension than the number of states in the chain, and the representation is insufficient to exactly represent the value function.

The chain domain enables direct computation of the optimal value of $\mathbf{h}_t$, given the current estimate of $\mathbf{w}_t$ which we denote $\mathbf{h}^\star$. We can compute $\mathbf{h}^\star$ directly from the parameters of the MDP and current estimate $\mathbf{w}_t$:

$$\mathbf{h}_t^\star = C^{-1}(-A^\pi \mathbf{w}_t + b^\pi), \tag{7.1}$$

where $C = \mathbf{X}^\top B \mathbf{X}$, $A^\pi = \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}(I - \gamma P^\pi)\mathbf{X}\mathbf{w}_t$, and $b^\pi = \mathbf{X}^\top B(I - \lambda\gamma P^\pi)^{-1}\mathbf{R}$. See section 2.6 for the derivation of this equation.

We can also compute the least squares solution for the secondary weights, denoted by $\mathbf{h}^{\texttt{GTD}}$. This approximation is computed from the data generated by the demon's interaction with the Markov chain task and $\mathbf{w}_t$:

$$\mathbf{h}_t^{\texttt{GTD}-\texttt{LS}} = \left(\overline{\mathbf{x}\mathbf{x}^\top}\right)^{-1} \overline{\delta_t \mathbf{e}}, \tag{7.2}$$

where $\overline{\cdot}$ denotes the sample average. Note that $\mathbf{h}_t^{\texttt{GTD}-\texttt{LS}}$ estimates $\mathbf{h}_t^\star$ by sampling the expectations: $\mathbb{E}_\mu[\mathbf{x}(S_t) f s(S_t)^\top]^{-1}\mathbb{E}_\mu[\delta_t \mathbf{e}_t]$. Computing $\mathbf{h}_t^{\texttt{GTD}-\texttt{LS}}$ via Equation 7.2 makes use of the complete history of sample transitions, $(\mathbf{x}_t, \rho_t, r_{t+1}, \mathbf{x}_{t+1})$, to recompute $\mathbb{E}_\mu[\delta_t \mathbf{e}_t]$ on each time-step. This computation is needed because δ_t is a function of the current $\mathbf{w}_t$.

7.1.2 Learning $\mathbf{h}$

Our first set of experiments seeks to answer a basic question: "Are the secondary weights learned by GTD(λ) similar to their theoretical values ($h^\star$), if the parameters of GTD(λ) are tuned to minimize projected bellman error?". We are interested in the case where the parameters of GTD(λ) are tuned to minimize MSPBE because this represents the usual use case for GTD(λ). That is, informally, we are interested in how well the GTD(λ) algorithm learns the $\mathbf{h}$ vector under normal use. Similarity will be defined below.

The prior experimental work on gradient-TD learning did not address the question stated above. Several studies reported good results when the secondary weights are learned slowly, using a small value for $\alpha_{\mathbf{h}}$ compared to α (Sutton et al., 2009; Degris et al., 2012; Hackman, 2013). This is perplexing because we might expect it to be difficult for the update of $\mathbf{h}$ to track the moving target $\mathbb{E}_\mu[\delta_t \mathbf{e}_t]$, when $\alpha_{\mathbf{h}}$ is smaller than α. Recall that GTD(λ)

uses an LMS rule to learn $\mathbf{h}$ based in part on the TD errors produced by the update to the primary weight vector $\mathbf{w}_t$. On the other hand, Maei's (2011) experiments with GTD(0) on Baird's counterexample used a value of $\alpha_{\mathbf{h}}$ much larger than α, but those experiments did not investigate or report if $\mathbf{h}$ approximated $\mathbf{h}^\star$ well.

Based on prior evidence, we hypothesize that the GTD(λ) algorithm's best performance will be achieved with small values of $\alpha_{\mathbf{h}}$ relative to α, and thus the secondary weights will not match what the theory tells us $\mathbf{h}$ should be.

Experiment

The task of our first experiment was to find the values of α and $\alpha_{\mathbf{h}}$ that produce the the lowest MSPBE over the last 50 episodes of a 200 episode experiment on five instances of the chain problem. We tested 72 instances of GTD, each with a different combination of α and $\alpha_{\mathbf{h}}$: all combinations of $\alpha \in \{0.0025, 0.005, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32\}$ and $\alpha_{\mathbf{h}} \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 0.25, 0.5, 0.75, 1.0\}$. The five problem instances included two on-policy instances and three off-policy instances. Each algorithm instance was initialized with $\mathbf{w} = \vec{0}$ and $\mathbf{h} = \vec{0}$, and then run for 200 episodes. Then the whole thing was repeated for 100 runs. The random seed was initialized to the same value for all algorithm instances at the beginning of each group of 100 runs. We used the MSPBE as a measure of performance like many previous studies of gradient-TD methods (e.g., Sutton et al., 2009; Maei, 2011; Hackman, 2012). For each run, the root MSPBE or RMSPBE was recorded at the end of each episode and averaged over runs. At the beginning of each run, the weight vectors of the GTD(λ) were both set to zero. We selected the α and $\alpha_{\mathbf{h}}$ that achieved the best total RMSPBE over the last 50 episodes.

The values of α and $\alpha_{\mathbf{h}}$ that achieved the best total RMSPBE over the last 50 episodes are given in Table 7.1. Small values of $\alpha_{\mathbf{h}}$ yielded the best RMSPBE, but the smallest available value, $\alpha_{\mathbf{h}} = 0.00001$, did not produce the lowest average RMSPBE in an chain instance.

chain instance	$p_\mu = 0.5$ $p_\pi = 0.5$	$p_\mu = 0.5$ $p_\pi = 0.25$	$p_\mu = 0.5$ $p_\pi = 0.75$	$p_\mu = 0.6$ $p_\pi = 0.4$	$p_\mu = 0.95$ $p_\pi = 0.95$
α	0.01	0.01	0.01	0.01	0.08
$\alpha_{\mathbf{h}}$	0.01	0.1	0.0001	0.01	0.01

Table 7.1: The values for α and $\alpha_{\mathbf{h}}$ found to minimize RMSPBE on average, in each instance of the chain problem. Each instance is defined by the combination of target policy and behavior policy. The probability of taking the *right* action for the target policy is denoted by p_π, and likewise for the behavior: p_μ. See text for a description of the experiment.

cosine-similarity$(\mathbf{a}, \mathbf{b})$	$\frac{\mathbf{x}^\top B \mathbf{y}}{\sqrt{x^\top B x}\sqrt{y^\top B y}}$, where $x = \mathbf{X}\mathbf{a}, y = \mathbf{X}\mathbf{b}$	
weighted-difference$(\mathbf{a}, \mathbf{b})$	$\sqrt{\sum_{s \in \mathcal{S}} d(s)[(\mathbf{a}-\mathbf{b})^\top \mathbf{x}(s)]^2}$ $\sqrt{(\mathbf{X}(\mathbf{a}-\mathbf{b}))^\top B (\mathbf{X}(\mathbf{a}-\mathbf{b}))}$	$=$
correction-difference$(\mathbf{a}, \mathbf{b})$	$\left\lvert \mathbf{e}_t^\top \mathbf{a} - \mathbf{e}_t^\top \mathbf{b} \right\rvert$	

Table 7.2: Three different ways to measure the similarity of the secondary weights of GTD(λ) and their optimal values given by the parameters of the MDP.

We now turn to describing how we measured the similarity of $\mathbf{h}$ to $\mathbf{h}^\star$, in order to answer our question specified in Section 7.1.2. We used three measures summarized in Table 7.2. The first was the *cosine-similarity* which measures the angle between two vectors, weighted by the state visitation probabilities induced by the behavior policy and the transition dynamics of the MDP. The cosine-similarity takes on values between -1 and +1, with -1 indicating the two vectors point in opposite directions, +1 indicating the two vectors point in precisely the same direction, and 0 indicating the two vectors are orthogonal.

The second measure, the *weighted-difference*, measures the 2-norm difference between two vectors, again weighted by the distribution of state visitation. The minimum value of the weighted-difference is zero, indicating high similarity, while higher values indicate low similarity. In all our experiments the weighted-difference was always less than one.

The third measure, the *correction-difference*, computes the absolute difference in the correction term used in the GTD(λ) algorithm's update, $\mathbf{h}^\top \mathbf{e}_t$. Again, the minimum value of the correction-difference is zero, indicating high similarity, while higher values indicate low similarity.

To answer our question, we performed an experiment using the parameter settings from Table 7.1. One instance of GTD(λ) was run on each of the five chain problem instances. Each algorithm instance was initialized with $\mathbf{w} = \vec{0}$ and $\mathbf{h} = \vec{0}$, and the values of α and $\alpha_{\mathbf{h}}$ were set according to Table 7.1. Each combination was run for 200 episodes. Then the whole thing was repeated for 200 runs. For each run, the similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ and the similarity of $\mathbf{h}^{\texttt{GTD}-\texttt{LS}}$ and $\mathbf{h}^\star$ (according to our three measures) were recorded at the end of each of the 200 episodes and averaged over runs. We also recorded the ℓ_2-norm of the $\mathbf{h}$ and $\mathbf{h}^\star$ for each problem instance, at the end of each episode, again averaged over 200 runs. Figure 7.2 presents the results.

Results and conclusions

The results in Figure 7.2 show that over five problem instances, the similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ was low compared to the similarity of $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$. The cosine-similarity of $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ approached its max value of 1.0 after only a few episodes, and the weighted-difference of $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ quickly decreased to its min-value after a few episodes. In comparison, the cosine-similarity of $\mathbf{h}$ and $\mathbf{h}^\star$ seemed to get worse with each episode. The weighted-difference and correction-difference of $\mathbf{h}$ and $\mathbf{h}^\star$ appeared to improve over time.

The smallest weighted-difference and correction-difference between $\mathbf{h}$ and $\mathbf{h}^\star$ were achieved in the last problem instance, which featured on-policy updates and a behavior that nearly always selected the *right* action. The cosine-similarity was low for this problem instance. The plot of the norm indicates that both the learned $\mathbf{h}$ and $\mathbf{h}^\star$ did not change much from zero, perhaps due to the speed of learning the primary weights (due to the high $\alpha = 0.08$ value). Good reduction in weighted-difference and correction-difference was also observed in the $(p_\mu = 0.5, p_\pi = 0.75)$ problem instance. In this instance, $\alpha_{\mathbf{h}}$ was small, equal to 0.0001, and $\mathbf{h}$ was initialized to the zero vector. Therefore, $\mathbf{h}$ was never changed far from the zero vector, which can be seen in the plot of the norm of $\mathbf{h}$. In this instance, the reduction in the norm of $\mathbf{h}^\star$ over time seems to account for the shape of the weighted-difference and weight-correction plots.

Overall, we conclude that the secondary weights learned by the GTD(λ) algorithm do not match their theoretical values ($\mathbf{h}^\star$) well, when compared to how well $\mathbf{h}^{GTD-LS}$ matched $\mathbf{h}^\star$. However, our results do suggest that the similarity between the secondary weights and $\mathbf{h}^\star$ can improve over time, but not according to the cosine similarity. The overall similarity is typically highest when $\alpha_{\mathbf{h}}\alpha > 1.0$, meaning that $\mathbf{h}$ can track changes in δ_t, caused by more slowly changing $\mathbf{w}_t$, as observed in our results.

7.1.3 Tuning GTD(λ) for similarity

Our next question is: "Does the similarity between $\mathbf{h}$ and $\mathbf{h}^\star$ improve if we tune α and $\alpha_{\mathbf{h}}$ to optimize some measure of similarity?". There are no related experiments in the literature to give us insight into the answer of this question. One might expect that directly optimizing for similarity should improve similarity, but we will be limited by our ability to define a useful measure to optimize the learning rate parameters of the GTD(λ) algorithm. In addition, we cannot guarantee that the GTD(λ) algorithm will be able to learn $\mathbf{h}$ well in the chain instances we have chosen. Therefore, we hypothesize a small improvement in similarity over the results of the previous section.

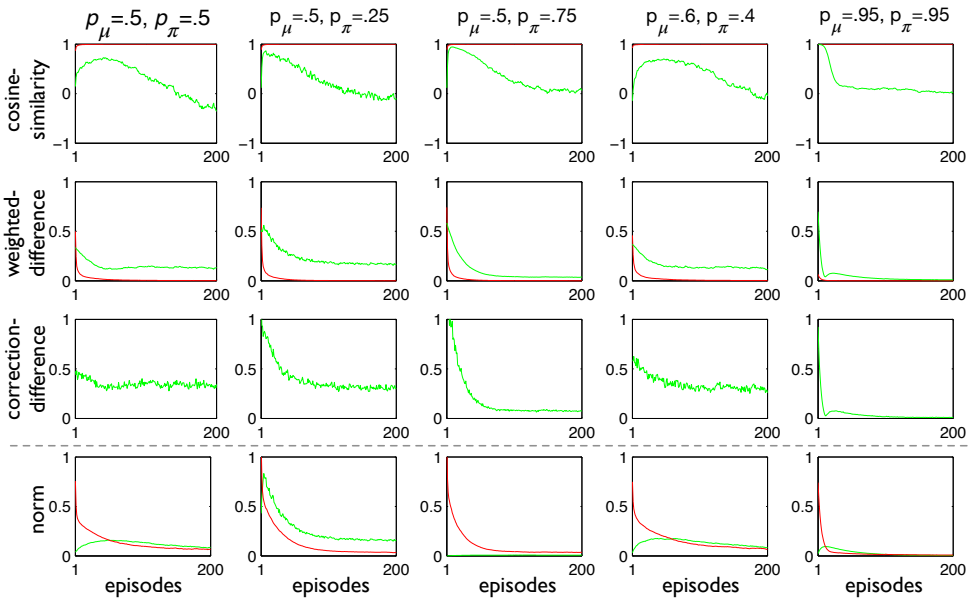

Figure 7.2: The similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ and $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ when the parameters of the GTD(λ) algorithm were tuned to minimize the MSPBE on several instances of the Markov chain domain. Each column corresponds to one of five chain problem instances. The first three rows corresponds to one of the three similarity measures described in text. Each subfigure in the first three rows plot the similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ (plotted with green lines) and $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ (plotted with red lines) over 200 episodes on one instance of the chain task. The last row of figures plots the ℓ_2-norm of $\mathbf{h}$ (in green) and $\mathbf{h}^\star$ (in red) for each problem instance. The results for the correction-difference between $\mathbf{h}^{GTD-LS}$ and $\mathbf{h}^\star$ were always near zero and were excluded from the results presentation. Note that the cosine-similarity of $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ quickly approaches one—high similarity—but it is difficult to see in the first row of results.

Experiment

We used the same five instances of the Markov chain tasks, and conducted an experiment to find values of α and $\alpha_{\mathbf{h}}$ that optimize the overall similarity. We created a *total-difference* measure that attempts to balance the cosine-similarity and weighted-difference measures used in the previous experiment:

$$\texttt{total} - \texttt{difference}(\mathbf{a}, \mathbf{b}) = \frac{\texttt{weighted} - \texttt{difference}(\mathbf{a}, \mathbf{b})}{e^{\texttt{cosine}-\texttt{similarity}(\mathbf{a},\mathbf{b})}}. \tag{7.3}$$

The total-difference measures how well two vectors match in both magnitude and direction, while incorporating the weighting by the stationary distribution included the definitions of the weighted-difference and the cosine-similarity. The exponential accounts for the cosine-similarity taking on values less than zero. The correction-difference was not included because it did not seem to contribute much more information than the weighted-difference,

in the previous experiment. If the total-difference is zero, its minimum value, then the two vectors are the same.

Our first task was to find values of α and $\alpha_{\mathbf{h}}$ that produce the the lowest total-difference over the first 100 episodes on five instances in the chain problem. We choose to optimize the parameters over the first 100 episodes because manual exploration revealed that near the end of the 200 episodes, $\mathbf{h}_t^\star$ begins moving toward the zero vector. We were more interested in early learning when $\mathbf{h}_t$ is not zero, and thus our optimization criteria focused on the first 100 episodes.

We tested 72 instances of GTD, each with a different combination of α and $\alpha_{\mathbf{h}}$ sampled from the same sets as the previous experiment. The experiment was run exactly as before, running 200 episodes per run and 100 runs for each algorithm instance on each problem instance. For each run, the mean total-difference over the first 100 episodes was recorded and averaged over runs.

The values of α and $\alpha_{\mathbf{h}}$ that achieved the lowest total-difference over the first 100 episodes are given in Table 7.3. In all problem instances tested, $\alpha_{\mathbf{h}} > \alpha$ produced the lowest total-difference.

chain instance	$p_\mu = 0.5$ $p_\pi = 0.5$	$p_\mu = 0.5$ $p_\pi = 0.25$	$p_\mu = 0.5$ $p_\pi = 0.75$	$p_\mu = 0.6$ $p_\pi = 0.4$	$p_\mu = 0.95$ $p_\pi = 0.95$
α	0.0025	0.005	0.005	0.0025	0.04
$\alpha_{\mathbf{h}}$	0.01	0.01	0.01	0.01	0.1

Table 7.3: The values for α and $\alpha_{\mathbf{h}}$ found to minimize total-difference on average, in each instance of the chain problem.

Given these values of α and $\alpha_{\mathbf{h}}$, we ran an experiment to discover the similarity of $\mathbf{h}$ compared to $\mathbf{h}^\star$ on the chain problem. The experiment was run exactly as before, except the values of α and $\alpha_{\mathbf{h}}$ for each problem instance were set according to Table 7.3. Figure 7.3 summarizes the results.

Results and conclusions

The results of this experiment (see Figure 7.3) illustrate some improvement in similarity between $\mathbf{h}$ and $\mathbf{h}^\star$, by all three measures, in comparison with the results found in Figure 7.2. Specifically, the weighted-difference and correction-difference improved in the first two chain instances. The weighted-difference and correction-difference improved steadily over time for the $(p_\mu = 0.5, p_\pi = 0.75)$ instance, but the norm of $\mathbf{h}$ was not zero as in Figure 7.2. Overall, the cosine-similarity showed the most significant improvement compared

to the results in Figure 7.2, and the norm of $\mathbf{h}$ aligned well with the norm of $\mathbf{h}^\star$ in every task, after an initial transient period. Overall, we conclude that, in the chain problem instances

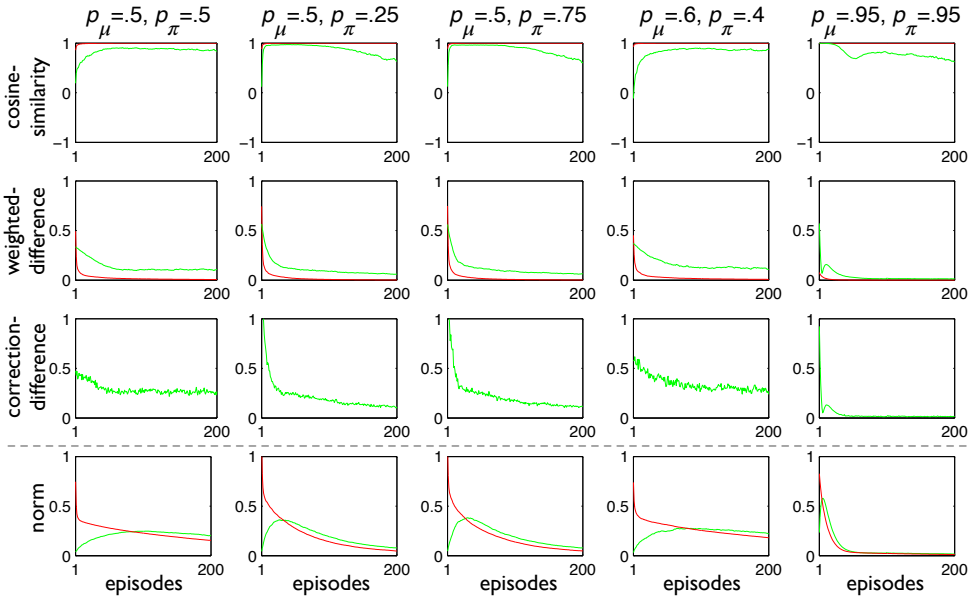

Figure 7.3: The similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ and $\mathbf{h}^{GTD-LS}$ to $\mathbf{h}^\star$ when the parameters of the GTD(λ) algorithm were tuned to optimize similarity on several instances of the Markov chain domain. The format, organization, and notational conventions of these graph are the same as those established in Figure 7.2.

tested here, and when the parameters of the GTD(λ) algorithm are tuned to optimize our total-difference measure, the secondary weights learned by the GTD(λ) algorithm match their theoretical values ($\mathbf{h}^\star$) well in comparison to the results of the previous section's experiments (Section 7.1.2). We can force the secondary weights to better match their optimal values through optimization of similarity, at least in the configurations tested. The ratio of $\frac{\alpha_\mathbf{h}}{\alpha}$, which was found to optimize similarity, for each of the five problem instances, was always larger than 1.0. We speculate that the $\mathbf{h}$ vector can more easily track the changes in its non stationary update target, the error due to current value of the primary weights ($\delta_t \mathbf{e}_t$), when $\alpha_\mathbf{h} \geq \alpha$.

7.1.4 MSPBE minimization and $\mathbf{h}$

Prior experimental studies of gradient-TD methods (e.g., Sutton et al., 2009; Degris et al., 2012; Hackman, 2013; Dann et al., 2014) demonstrated that $\alpha_\mathbf{h}$ values near zero can achieve good performance in terms of MSPBE optimization and reward maximization (see Degris et al.'s (2012) policy-learning experiments). Perhaps the GTD(λ) algorithm works

best when not using the secondary weights at all ($\alpha_{\mathbf{h}} = 0$). Our experiments so far suggest that the values of α and $\alpha_{\mathbf{h}}$ that optimize the similarity of $\mathbf{h}$ to $\mathbf{h}^\star$ can be very different than the ones found to optimize the MSPBE.

Our next question is long but simple: "How does the performance of GTD(λ) with α and $\alpha_{\mathbf{h}}$ optimized for similarity compare to the performance of GTD(λ) with (1) α and $\alpha_{\mathbf{h}}$ optimized to minimize the MSPBE, (2) $\alpha_{\mathbf{h}} = 0$ and α optimized to minimize the MSPBE, and (3) $\mathbf{h}_t = \mathbf{h}_t^\star$ and α optimized to minimize the MSPBE?". Based on the experimental evidence from the literature, we hypothesize that the variant of GTD(λ) with $\alpha_{\mathbf{h}}$ equals zero and the variant that uses $\mathbf{h}^\star$ will yield the best MSPBE minimization.

Experiment

We used the same five problem instances as the proceeding experiments, but this time we compared the performance of four variants of the GTD(λ) algorithm. The first variant used the settings for α and $\alpha_{\mathbf{h}}$ given in Table 7.1. The second variant of GTD(λ) used the settings for α and $\alpha_{\mathbf{h}}$ given in Table 7.1. The third variant used $\alpha_{\mathbf{h}} = 0$ and α optimized for minimum RMSPBE (same range and setup as previous experiments). The final variant used $\mathbf{h}_t^\star$ instead $\mathbf{h}_t$ in the update of of GTD(λ) and α optimized for minimum RMSPBE.

The first and second variants of GTD(λ) correspond to optimizing MSPBE and similarity. The third variant corresponds to not using the secondary weights in the update, and the final variant uses the theoretically optimal $\mathbf{h}$ vector given the current estimate of $\mathbf{w}_t$ and the parameters of the MDP.

In our final experiment with the chain problem, we tested each variant of GTD(λ) on each of the five chain problem instances. Our experimental setup was similar to before. Each algorithm instance was initialized with $\mathbf{w} = \vec{0}$ and $\mathbf{h} = \vec{0}$, and then run for 200 episodes. Then the whole thing was repeated for 1000 runs. For each run, the RMSPBE was recorded at the end of each of the 200 episodes and averaged over runs to produce the learning curves in Figure 7.4.

Results and conclusions

The results show that the GTD(λ) algorithm with $\alpha_{\mathbf{h}} = 0$ produced the best learning curve in each of the five problem instances. In addition, the variant of GTD(λ) with $\alpha_{\mathbf{h}}$ tuned to minimize RMSPBE was at least as good across the five problems as the variant of GTD(λ) that used $\mathbf{h}^\star$. Finally, the variant of the GTD(λ) algorithm with $\alpha_{\mathbf{h}}$ tuned to optimize

similarity was the slowest to learn in all five problem instances, but did approach the performance of the other algorithm variants by the end of the experiment, in three problem instances.

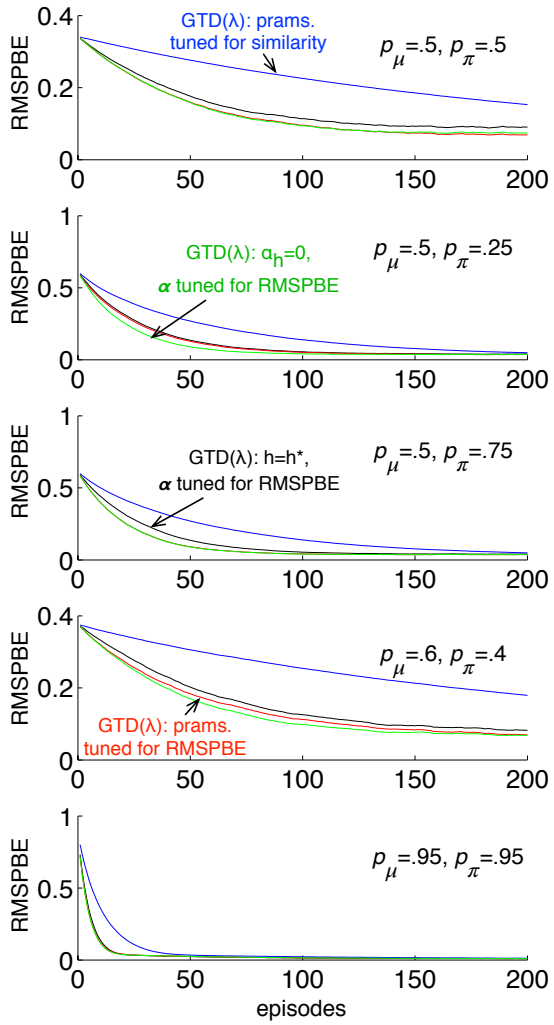

Figure 7.4: The RMSPBE curves for four instances of the GTD(λ) algorithm on five instances of the Markov chain domain. We have labelled each variant of GTD(λ) in the figure itself. See text for a description of the experiment and discussion of the results.

Our main conclusion from these results is that learning the secondary weights with GTD(λ) and α and $\alpha_{\mathbf{h}}$ set to minimize RMSPBE performs quite well compared to not using the secondary weights at all (i.e., $\alpha_{\mathbf{h}} = 0$), and compared to using the optimal secondary weights (i.e., $\mathbf{h}^\star$ instead of $\mathbf{h}$). We also conclude that tuning the learning rate parameters of GTD(λ) to optimize similarity results in a noticeable performance penalty in terms of MSPBE reduction over time. However, it appears that there is less of a performance penalty

in problem instances with more deterministic behavior policies. Finally, we also conclude that the variant of GTD(λ) with $\alpha_{\mathbf{h}} = 0$ can perform better in initial learning than the variant that used $\mathbf{h}^\star$. We conjecture that, in problem instances where this is true, the correction term is not needed for convergence and the correction term actually slows learning, even when $\mathbf{h}^\star$ is used. Verifying this conjecture requires future experiments which we leave to future work.

7.1.5 Overall conclusions

Overall we make several conclusion, from our experiments on the Markov chain problem. The first is that optimizing the parameters of GTD(λ) for low MSPBE will not necessarily facilitate learning a secondary weight vector that is near equal to $\mathbf{h}^\star$. It appears that learning the secondary weights, albeit with $\frac{\alpha_{\mathbf{h}}}{\alpha} < 1.0$, yields similar MSPBE learning curves as using the secondary weight vector specified by $\mathbf{h}^\star$. If, on the other hand, $\frac{\alpha_{\mathbf{h}}}{\alpha} > 1.0$ is true, then the learned secondary weights can approximate $\mathbf{h}^\star$ better but not perfectly, and the algorithm will typically yield slower MSPBE minimization.

7.2 Experiments on Baird's counterexample

There is some uncertainty surrounding the importance, in practice, of the secondary weights in problems featuring off-policy sampling. Prior experiment studies have demonstrated that both off-policy linear TD(0) and expected Sarsa(0) can outperform their gradient-TD counterparts (Sutton et al., 2009; Maei, 2011; Hackman, 2012) in on-policy simulation problems, and expected Sarsa(0) can outperform GQ(0) in some off-policy domains (see Hackman, 2012) . Our experiments in this chapter show that fast learning can be achieved using GTD(λ) with the secondary weights equal to $\vec{0}$ under off-policy sampling. Our experiments with learning off-policy on robots in the previous chapter showed accurate predictions and that policies can be learned with $\alpha_{\mathbf{h}}$ small and even zero. On the other hand, there are known counterexamples for convergence of linear off-policy TD(0) with function approximation (Baird, 1995), with non-linear function approximation (Maei, 2011), and Q-learning (Baird, 1993). Each of these counterexamples have been solved by gradient-TD learning methods.

In order to gain some new understanding about the importance of the secondary weights in off-policy domains, we performed several experiments with the GTD(λ) algorithm on Baird's counterexample: an MDP that causes linear off-policy TD(0)'s value estimates to diverge.

7.2.1 Problem

We use a variant of Baird's counterexample (Baird, 1995) described by Maei (2011) and shown in Figure 7.5. We will refer to this problem slightly inaccurately as Baird's counterexample. This MDP contains seven discrete states and no terminal states; it is a continuing problem. There are two actions available in each state: action one (solid arrow) and action two (dashed arrow). Action one moves the demon to state seven from every state, including from state seven. The second action, in states one to six, moves the demon to a new state from one to six randomly with equal probability, but never to state seven. The second action, in state seven, moves the demon to any state from one to six, randomly with equal probability, but never to state seven.

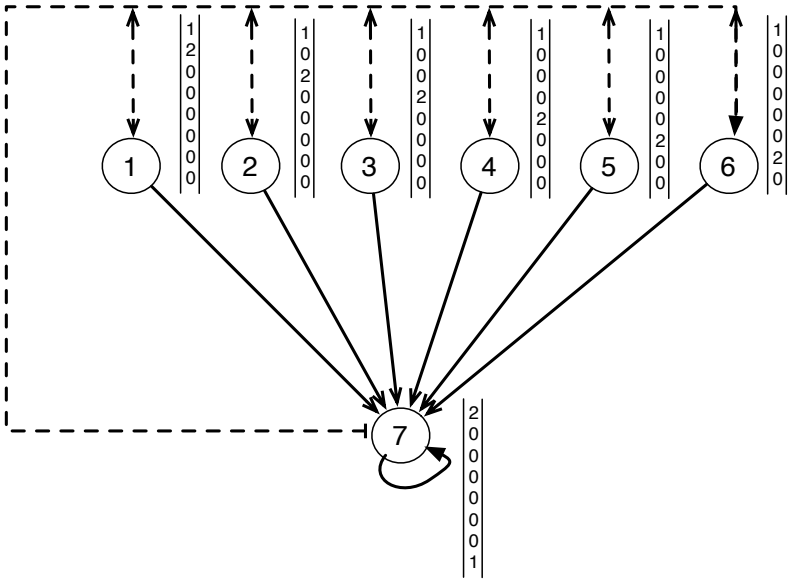

Figure 7.5: A variation of Baird's counterexample due to Maei (2011).

The learning problem is specified by the question functions. The target policy selects action one deterministically in every state. The cumulants are zero on every transition, and γ_t is constant and equal to 0.99. This setting of the question functions specifies a single GVF to learn.

The answer functions for this task are straightforward. The behavior policy selects action one with probability 1/7, and action two with probability 6/7, in every state. The feature vectors for each state are given in Figure 7.5. The λ value is a constant function equal to 0.0. The learning task specified by these question functions has a solution of $v(s; \pi, \gamma, z) = 0$ for all states, with $\mathbf{w} = \vec{0}$ and $\mathbf{w} = [-2, 1, 1, 1, 1, 1, 1, 4]c$ where $c \in \mathbb{R}$: an infinite number of solutions.

The combination of question and answer functions causes the parameter value estimates of TD($0 \leq \lambda < 1$) to diverge to infinity. The difficulty is due to the interaction of the initialization of the weight vector, $\mathbf{w}_0 = [1, 1, 1, 1, 1, 1, 1, 10]^\top$, the large value of γ, the feature component shared amongst all states, and the mis-match between the target and behavior policies.

Let us trace some transitions through the MDP and see what happens to the off-policy linear TD(0) algorithm, described in Chapter 2. Consider the first transition into state seven. The weight vector $\mathbf{w}_t$ will still equal $[1, 1, 1, 1, 1, 1, 1, 10]$ because any transitions from states one through six are off-policy, with $\rho_t = 0$, and cause no update. The TD error for the transition from state six to seven will be:

$$\delta_t = z_t + \gamma_{t+1}\mathbf{x}_{t+1}^\top \mathbf{w}_t + \mathbf{x}_t^\top \mathbf{w}_t = 0 + 0.99(2 + 10) - (1 + 2) = 8.88.$$

Assuming $\alpha = 0.1$, then $\mathbf{w}_{t+1}$ becomes $[7.216, 1, 1, 1, 1, 1, 13.432, 10]^\top$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\rho_t\delta_t\mathbf{x}_t = \mathbf{w}_t + 0.1(7.0)8.88\mathbf{x}_t.$$

Next, the behavior will likely transition the demon into some other state from one to six, with a TD error of zero and no update. The demon will jump around between states one through six (again with no updates to $\mathbf{w}_t$) and then eventually transition to state seven again. Every time the demon transitions from a state to state seven for the first time, $\mathbf{w}(1)_t$ is increased along with the component of $\mathbf{w}_t$ corresponding to the state the transition originated. Subsequent repeated transitions into state seven (e.g., six to seven from the example above) will cause alternating small magnitude negative and positive updates to $\mathbf{w}(0)_t$ and no changes to $\mathbf{w}(8)_t$. Only a transition from state seven to seven will depress $\mathbf{w}(8)_t$. For example, if such a transition happened next:

$$\delta_t = 0 + 0.99(24.432) - 24.432 = -0.24432,$$

then $\mathbf{w}_t = [6.873952, 1, 1, 1, 1, 1, 13.432, 9.828976]^\top$. These seven to seven transitions are rare and produce smaller magnitude updates compared to the transitions into state seven. Overtime, the magnitude of δ_t and $\mathbf{w}_t$ grow larger and larger causing divergence in the value estimates.

7.2.2 The role of $\mathbf{h}$

The ambition of these experiments, is to investigate the contribution of the secondary weights and the parameter sensitivity of the GTD(λ) in a problem in which the TD(λ)

algorithm will surely diverge. Specifically, we seek to answer two questions: "(1) How will GTD(0) perform on a known counterexample (a) if we learn the secondary weights as usual, and (b) if $\alpha_{\mathbf{h}} = 0$, and (c) if we use $\mathbf{h}^\star$ instead of $\mathbf{h}$?"; (2) "What is the parameter sensitivity of GTD(λ) with respect to λ, α, and $\alpha_{\mathbf{h}}$ on a known counterexample?".

We hypothesize that the answer to question one will be that GTD(0) with $\alpha_{\mathbf{h}} = 0$ will diverge, but it is unclear which of the other two variants will perform best. The answer to the second question is a mystery, as no prior work has attempted this specific study.

Our first task was to find the best parameters for the three variants of GTD(0) on Baird's counterexample. We ran an experiment, sweeping over many settings of the learning rate parameters. The first variant of GTD(0) had its α and $\alpha_{\mathbf{h}}$ parameters determined by a parameter sweep. The second variant used $\alpha_{\mathbf{h}} = 0$ and a value of α determined by a parameter sweep. The final variant used $\mathbf{h}^\star$ instead of $\mathbf{h}$ (as before) and a value of α determined by a parameter sweep. The parameters α and $\alpha_{\mathbf{h}}$ (when applicable) were sampled from all combinations of $\alpha \in (.1) * 2^{\{-1,-2,...,-20\}}$ and $\alpha_{\mathbf{h}} = \eta * \alpha$, where $\eta \in 1.5^{\{-2,-1,...,12\}}$. Each combination of algorithm instance, α, and $\alpha_{\mathbf{h}}$ was initialized with $\mathbf{w} = \vec{0}$ and $\mathbf{h} = \vec{0}$, and then run for 5000 steps. Then the whole thing was repeated for 200 runs. The random seed was initialized to the same value for all algorithm instances at the beginning of each group of 200 runs. For each run, the mean RMSPBE over all 5000 steps was recorded and averaged over runs. The best parameter settings for each algorithm variant was used in the next experiment.

Next we ran the experiment with the three variants of GTD(0), each with its α and η value found to be best in the parameter sweep. All values of α tested caused divergence for the variant of GTD(0) with $\alpha_{\mathbf{h}} = 0$; we used $\alpha = 0.0125$ (the learning rate parameter value that was best for the GTD(λ) variant with $\alpha_{\mathbf{h}} > 0$). Again we ran each variant for 5000 steps and repeated the experiment 200 times recording the RMSPBE on each step averaged over runs to produce the learning curves in Figure 7.6.

We also investigated the parameter sensitivity of the GTD(λ) algorithm. We tested GTD(λ) with λ equal to 0.0, 0.5, and 0.9, and the same combinations of α and η described above. Again we ran each variant combination of λ, α, and η for 5000 steps, and repeated the experiment 200 times recording the mean RMSPBE over all 5000 steps, and averaged over runs. The results of this experiment are summarized in the heat maps in Figure 7.7.

Results, analysis, and conclusions

The results of our experiment show that GTD(0) with $\alpha_{\mathbf{h}} = 0$ diverged, GTD(0) with learned $\mathbf{h}$ did not diverge, and GTD(0) with $\mathbf{h}^\star$ performed best over 5000 steps on Baird's counterexample. The best performance of the variant of GTD(0) with learned $\mathbf{h}$ was achieved with $\alpha = 0.0125$ and $\alpha_{\mathbf{h}} = 3.37500 * \alpha$. The best performance of GTD(0) with $\mathbf{h} = \mathbf{h}^\star$ was achieved with $\alpha = 0.025$.

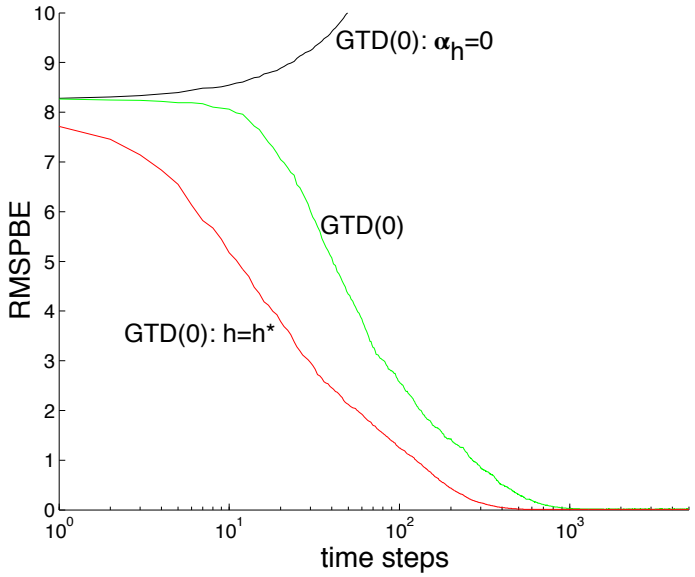

Figure 7.6: The learning curves for three variants of GTD on Baird's counterexample. Plotted is the RMSPBE verses time-step with a log-scale on the x-axis. Each variant uses the best parameter's found over a large systematic sweep.

Consider the updates of the GTD(0) algorithm in Baird's counterexample. Recall that the GTD(0) algorithm updates the primary weights, $\mathbf{w}_{t+1}$, corresponding to the next state's feature vector by a correction term:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\rho_t[\delta_t\mathbf{x}_t - \gamma_{t+1}(\mathbf{x}_t^\top\mathbf{h}_t)\mathbf{x}_{t+1}],$$

since $\lambda = 0$. In Baird's counterexample, the correction helps by depressing the value of $\mathbf{w}(8)$. The GTD(0) algorithm's estimate of $\delta_t \approx \mathbf{x}_t^\top\mathbf{h}_t$ will become large after several transitions into state seven. Once $\mathbf{x}_t^\top\mathbf{h}_t$ becomes large, $\mathbf{w}(8)$ will be decreased proportional to $(-\alpha_t(\mathbf{x}_t^\top\mathbf{h}_t)\mathbf{x}_{t+1})$ on each transition into state seven. In this problem, the correction helps prevent divergence and helps GTD(0) find a close approximation to the correct weight vector, as supported by the results in Figure 7.6.

From these results in Figure 7.6 we conclude that the secondary weights of GTD(0) are

useful for avoiding divergence in Baird's counterexample, and that learning is faster if the theoretically optimal secondary weights ($\mathbf{h}^\star$) are used, compared to learning the secondary weights in the usual way.

The results of the parameter sweep showed that the average RMSPBE achieved by GTD(λ) in this domain were worse for the two values of λ larger than zero. Also, the range of α and η values that did not cause divergence appeared to become smaller for the larger λ settings. The individual learning curves of GTD($\lambda = 0.9$), for various parameter settings, exhibited highly erratic performance (not shown), and all parameter combinations caused divergence, even for α as small as $.1 * 2^{-20}$.

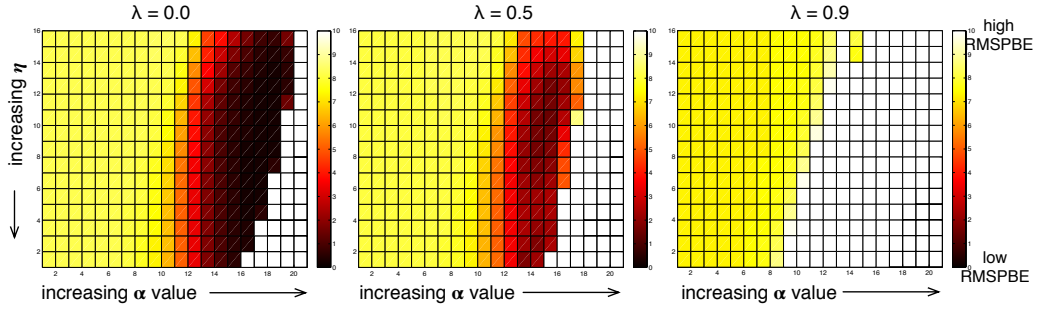

Figure 7.7: The average RMSPBE of GTD(λ) on Baird's counterexample, for three different values of λ. Each heat-map shows the mean RMSPBE (color) over 5000 steps for different combinations of α and η. White corresponds to massive errors, dark red (near black) regions correspond to low RMSPBE, and lighter colors indicate large RMSPBE.

One explanation for the performance of the GTD(λ) algorithm on Baird's counterexample might be large weight updates, caused by large likelihood ratios multiplying in the trace update. In this problem, the traces are frequently cleared by the off-policy transitions. However, in the unlikely event that several self transitions are made in state seven, the trace updates of the GTD(λ) algorithm can grow large quickly. To see this, consider successive state seven$\rightarrow$ state seven transitions and $\lambda = 0.9$, assuming $\mathbf{e}_{t-1} = \mathbf{x}_{t-1}$ because $\rho_{t-1} = 0$:

$$
\begin{aligned}
e_t &= \rho_t(\gamma\lambda\mathbf{e}_{t-1} + \mathbf{x}_t) = 7(0.99 * .9\mathbf{x}_{t-1} + \mathbf{x}_t) = 6.236\mathbf{x}_{t-1} + 7\mathbf{x}_t \\
e_{t+1} &= \rho_{t+1}(\gamma\lambda\mathbf{e}_t + \mathbf{x}_{t+1}) = 7(0.99 * .9(6.236\mathbf{x}_{t-1} + 7\mathbf{x}_t) + \mathbf{x}_{t+1}) \\
&= 41.31\mathbf{x}_{t-1} + 43.659\mathbf{x}_t + 7\mathbf{x}_{t+1} \\
e_{t+2} &= \rho_{t+2}(\gamma\lambda\mathbf{e}_{t+1} + \mathbf{x}_{t+2}) = 7(0.99 * 0.9(41.31\mathbf{x}_{t-1} + 43.659\mathbf{x}_t + 7\mathbf{x}_{t+1}) + \mathbf{x}_{t+2}) \\
&= 357.65\mathbf{x}_{t-1} + 272.30\mathbf{x}_t + 43.659\mathbf{x}_{t+1} + 7\mathbf{x}_{t+2}.
\end{aligned}
$$

The effect of such sequences, though rare, is a large update which causes instability in the

GTD(λ) algorithm. Larger λ values lead to larger possible updates. Individual runs of the GTD(λ) algorithm on Baird's counterexample can exhibit large spikes in RMSPE due to these large updates, causing the performance of the algorithm to suffer in practice. This instability did not arise in our experiments with the Markov chain, because the likelihood ratio was never greater than 1.5 and the episodic structure of the problem precludes large traces.

Another item of interest concerns the diminished role of the GTD(λ) algorithm's correction term in Baird's counterexample when λ is large. The update of the primary weights in the GTD(λ) algorithm is corrected by $-\gamma_{t+1}(1-\lambda)(\mathbf{h}_t^\top \mathbf{e}_t)\mathbf{x}_{t+1}$, and thus as λ becomes large the correction becomes smaller. Additional experiments are needed to conclude whether (1) the large updates are due to the traces, or (2) the smaller magnitude gradient corrections are the primary cause for the unsatisfactory performance we observed on Baird's counterexample with large λ.

These experiments shed light on a potential sensitivity problem of the GTD(λ) algorithm, previously unreported in the literature. Our experiments also highlight why the proof of convergence for the GQ(λ) algorithm—the action-value variant of GTD(λ)—includes a condition regarding the boundedness of the eligibility traces (see Maei & Sutton, 2010).

7.2.3 Similarity measures of $\mathbf{h}$

In our final experiment of this chapter, we return to investigating vector similarity. Our previous experiments on the Markov chain suggest that the similarity between $\mathbf{h}$ and $\mathbf{h}^\star$ may be low when the parameters of GTD(λ) algorithm are optimized to minimize MSPBE. Our experiments on Baird's counterexample suggest that the secondary weights of GTD(0) were important for avoiding divergence. The question we seek to answer with our final experiment is: "How similar is the $\mathbf{h}$ learned by GTD(0) compared to $\mathbf{h}^\star$ in Baird's counterexample?".

Based on all our previous experiments, our hypothesis is that even when α and $\alpha_{\mathbf{h}}$ are tuned to optimize RMSPBE, the similarity of $\mathbf{h}$ and $\mathbf{h}^\star$ will be high in Baird's counterexample.

Experiment

We conducted an experiment to test the similarity of $\mathbf{h}_t$ to $\mathbf{h}^\star(\mathbf{w}_t)$ on Baird's counterexample. We tested the GTD(0) algorithm with values of α and $\alpha_{\mathbf{h}}$ found to be best in the previous experiment, recording the similarity according to our three measures. The experiment

lasted for 5000 steps, and was repeated 400 times recording the mean cosine-similarity, mean weighted-difference, and mean correction-difference on each of the 5000 steps, and averaged over runs. The results of this experiment are summarized in the similarity curves in Figure 7.8.

Results and Conclusions

We conclude that, in Baird's counterexample, the $\mathbf{h}$ vector learned by GTD(0) matches $\mathbf{h}^\star$ well. Across all three measures, $\mathbf{h}$ was quite similar to $\mathbf{h}^\star$, in comparison to the previous similarity results (see Figure 7.3), except for the cosine similarity measure. This high similarity, however, was achieved without tuning α and $\alpha_{\mathbf{h}}$ to optimize similarity. This is not

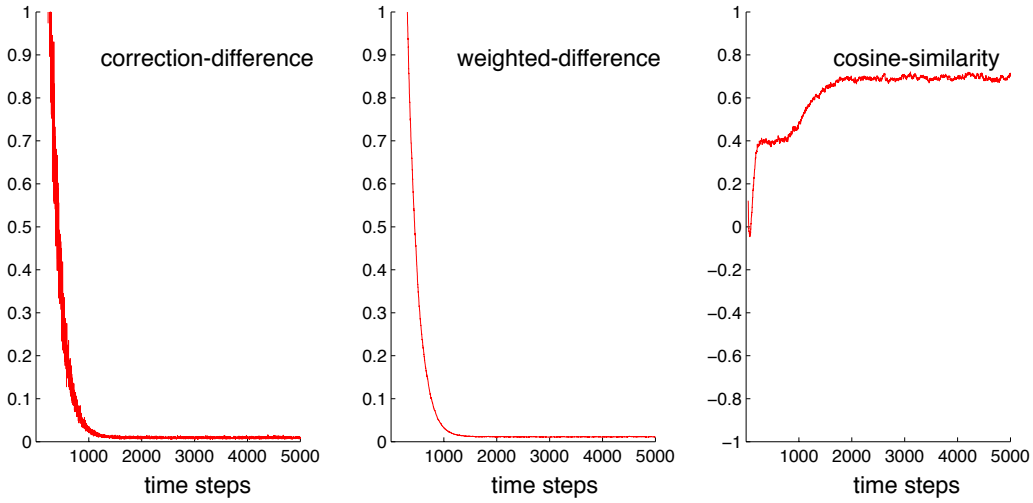

Figure 7.8: The similarity between the secondary weight vector $\mathbf{h}$, learned by GTD(0) and $\mathbf{h}^\star$ measured on Baird's counterexample. As before, we measure the correction-difference, the weighted-difference, and the cosine-similarity (defined in Table 7.2).

entirely surprising given our previous results with this problem, and the fact that $\alpha_{\mathbf{h}}$ was over seven times larger than α—therefore $\mathbf{h}_t$ can track the changes in its update target over time.

7.3 Related Work

Much of the experimental work on gradient-TD methods has rightly focused on the question of overall performance in comparison to other methods. Sutton et al. (2009) provided the first positive demonstrations of the GTD2 and TDC algorithms on on-policy Markov chains,

Boyan's chain, Baird's counterexample, and even computer go. Degris et al. (2012) compared the greedy-GQ(λ) algorithm to Q-learning and an off-policy actor critic algorithm for learning control policies in three simulation domains, demonstrating that sometimes greedy-GQ(λ) can perform poorly. Finally, Maei (2011) contributed experiments illustrating convergence of non-linear GTD(0) and greedy-GQ(λ) on counterexamples.

There have also been several studies for the parameter sensitivity of off-policy learning methods. The largest of these studies was by Dann et al. (2014), comparing nine value function estimation methods on six simulation domains (including Baird's counterexample). Dann's work provides insights into the role of λ, α, and $\alpha_{\mathbf{h}}$ in several off-policy domains, but the results are difficult to trust because the parameter sweeps were based on only three repetitions of their experiment. Geist and Scherrer (2014) compared the role of λ in eight value function estimation algorithms (including TD(λ) and GTD(λ)) on two randomly generated MDPs. The result of that particular experiment was inconclusive (as noted by the authors), but their remaining experiments showed that MSPBE minimization methods, such as GTD(λ) and LSTD(λ), outperformed bellman error minimization methods across 200 random MDPs. Hackman (2012) contributed an impressive set of experiments concerning state-action value function estimation in the Markov chain domain (slightly different than ours) with GQ(0) and expected Sarsa(0). Hackman found that the η parameter ($\alpha_{\mathbf{h}} = \alpha * \eta$) had little effect on the RMSPBE achieved by a variant of GQ(0) in on-policy problem instances. Delp (2011) studied the parameter sensitivity of state-action value function estimation with GQ(λ) in a discrete gridworld with linear function approximation. Delp found that when $\lambda = 0$ or 0.5, $\alpha_{\mathbf{h}}$ equal to zero performed best, just as in our Markov chain results. Finally, a recently proposed gradient-TD method was compared empirically with the GTD(λ) algorithm, including a sweep of λ and α (van Hasselt et al., 2014). The results showed that high values of λ and α yield high value function error in a random walk, whereas the new method produced good results over a wide range of λ and α values.

The experiments in this chapter are distinct in three ways. First, our experiments are the first to study a gradient-TD method's ability to learn the secondary weights. Second, we are the first to compare learning the secondary weights with not using them at all, and with using the optimal secondary weights given by the parameters of the MDP. Third and finally, our experiments are the first to empirically demonstrate the instability of the GTD(λ) algorithm with λ greater than zero in Baird's counterexample.

Finally, although we cannot claim our results and insights encompass other gradient-TD

methods (e.g., GQ(λ)), the questions we posed and results generated may provide clues and a starting place for the study of other related methods.

7.4 Summary

In this chapter, we presented a series of experiments designed to provide insight into the inner workings of one gradient-TD learning algorithm, GTD(λ). Our first set of experiments used variants of a Markov chain and studied how well the GTD(λ) algorithm learns the secondary weight vector. We then tested how well GTD(λ) minimized RMSPBE with different variations on how the secondary weights were learned. Our second batch of experiments focused on a variant of Baird's counterexample. Again we studied RMSPBE minimization with different variations on how the secondary weights were learned, and we tested the sensitivity of the GTD(λ) algorithm to variations in its parameter settings. Finally we investigated how well GTD(λ) learned the secondary weight vector in Baird's counterexample.

The overall conclusion from all our experiments is: in some problems the secondary weights of the GTD(λ) algorithm are important, but in other problems the correction can slow learning. In the chain, the fact that the secondary weights were not learned well did not effect overall MSPBE optimization, agreeing with many of the experimental results in the literature. On the other hand, in Baird's counterexample, a large value of $\alpha_{\mathbf{h}}$ is needed to achieve stable learning and the secondary weights are important for preventing divergence. Reducing the influence of the secondary weights and using longer eligibility traces caused the GTD(λ) algorithm to perform poorly in Baird's counterexample. In addition, our experiments demonstrate that the GTD(λ) algorithm can achieve good performance with very different parameter settings, depending on the problem. Without a prior knowledge of the problems our learning agents will face, it can be difficult to set the learning rate parameter values of this algorithm. There are several possible approaches we could take to make the GTD(λ) algorithm more robust, including heuristic rules to adjust $\alpha, \alpha_{\mathbf{h}}$, and λ during learning, but this topic is left to future research.

Off-policy updating is a key aspect of our approach to predictive knowledge learning, and understanding one of our learning methods is important for legitimizing the practicality of our approach. It is only recently that efficient, linear complexity, and non-divergent off-policy algorithms like GTD(λ) were invented. New insight and understanding of a key algorithm for GVF learning contributes a better understanding of the stability and overall

practicality of our approach to predictive knowledge.

Chapter 8

Estimating off-policy progress[1]

Tracking the learning progress of a large collection of demons is both useful and non-trivial. In the off-policy setting, gradient temporal difference methods can be used to update and refine each demon's prediction over time, but the algorithms themselves do not provide a measure of how much learning progress has been made by each demon. As an experimenter, we often want to know which demons are learning, which demons are not learning (no progress), which demons need significantly more time to learn, and which demons are nearly done learning. Developing a predictive approach to knowledge involves experimentation and testing; a measure of the learning progress is important for demonstrating the practicality of our approach. We have demonstrated several different approaches to measure off-policy learning progress on a robot in earlier chapters. As you will see later, these approaches are difficult to use with a large number of demons and a large number of distinct target policies.

This chapter introduces a new approach to measuring learning progress based on estimating the RMSPBE. We demonstrate empirically that our new measure provides a useful measure of learning progress with experiments on a Markov chain and the Critterbot.

8.1 Measuring progress on a robot

Estimating the off-policy learning progress on a robot is particularly challenging, as our previous experiments illustrate. On a robot, we do not have access to the parameters of the MDP, so we cannot exactly compute the MSPBE or the true value function, $v(s; \pi\gamma, z)$. We must estimate progress in some other way. In on-policy learning, estimating progress is more straightforward because all the data generated for learning is also useful for progress

[1] Some of the text and results contained in this chapter also appear in a conference paper (White, Modayil & Sutton, 2013). That paper was drafted in its entirety and its experiments conducted by this author.

evaluation, but this is not necessarily the case in off-policy learning. We have already tried several approaches to measuring off-policy learning progress. We estimated the target G_t and then computed the RMSE for each demon by forcing the behavior policy to execute long sequences of actions according to each target policy. This method was used in Chapter 6, experiments 6.4 and 6.5. We also tried a training-testing split, where the robot followed the behavior and updated the demons. Then, after learning, the demons were tested by executing each demon's target policy in turn. We used this approach in experiments 6.6, 6.7, and 6.8 of Chapter 6. A testing-training split may not be ideal because we cannot measure learning progress on each step during learning and thus debugging and improving the system may become time consuming and frustrating. A final approach involves interspersing the tests during learning, and periodically pausing learning to generate a trajectory to evaluate a demon's prediction. We demonstrate this approach on the Critterbot in this chapter.

All of the approaches induce a trade-off between how much time we devote to updating the demons and how much time we want to devote to progress estimation. If the measures are updated too infrequently, then the estimates of learning progress may become out-of-date and potentially highly variant. As the number of unique target policies grows large, it will typically take more time to evaluate each demon's progress, thus potentially reducing the amount of samples of learning or the timeliness of the estimate of progress.

We now specify a set of criteria for an estimate of learning progress that be useful for large-scale demon learning on a robot. First, ideally the measure should be *online*, updated after each learning step, and thus always up-to-date for each demon. The measure should be updated on each step with *linear* (or less) computation and storage (linear in the number of feature components) to ensure the scalability of Horde. The estimate should not place strong requirements on the behavior policy; the system could then be free to learn on every interaction step with no interruptions for testing. Finally, the estimate should provide an *accurate* estimate of learning progress, and in particular be robust to non-stationarity, which is common on real hardware systems, such as robots. In the next section we propose a new estimate of learning progress that we believe meets all four of these criteria.

8.2 A new proposal

Instead of using prediction accuracy or value error, we propose to estimate learning progress by estimating the mean squared projected bellman error of a demon's learned weight vector

$\mathbf{w}$:

$$\texttt{MSPBE}(\mathbf{w}) = ||\mathbf{X}\mathbf{w} - \Pi T^{\gamma\lambda}\mathbf{X}\mathbf{w}||_B^2,$$

originally defined in Chapter 2. This measure of progress does not always correspond to prediction accuracy. Our notion of learning progress does, however, indicate when a demon's predictions are approaching their best estimates, given the setting of the demon's answer functions. Additionally, in the case where the features can perfectly represent the true GVF, the MSPBE will correspond to the closeness of the demon's estimate to the true GVF: $(v(s;\pi,\gamma,z) - \hat{v}(s,\mathbf{w}))^2$ weighted by the distribution over states induced by the behavior. We choose to estimate the MSPBE, because that is the objective that the demons attempt to optimize.

Our aim is to estimate the learning progress for any value function estimation algorithm that minimizes the MSPBE, by estimating the RMPSBE.[2] Recall that the secondary weight vector of GTD(λ) is equal to part of the gradient of the MSPBE,

$$\mathbf{h} = \mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]\mathbb{E}_\mu[\delta_t\mathbf{e}_t],$$

which can be learned by an LMS rule,

$$\mathbf{h}_{t+1} = \mathbf{h}_t + \alpha_{\mathbf{h}}[\delta_t\mathbf{e}_t - (\mathbf{h}_t^\top\mathbf{x}_t)\mathbf{x}_t],$$

the same LMS rule used in the update of the GTD(λ) algorithm. The MSPBE of $\mathbf{w}_t$ is equal to $\mathbb{E}_\mu[\delta_t\mathbf{e}_t]\mathbf{h}_t$ using the forward backward equivalence described in the background chapter. Therefore, we can sample $\mathbb{E}_\mu[\delta_t\mathbf{e}_t]$, and form an estimate of the RMSPBE for each demon, on each time-step t, by:

$$\texttt{RMSPBE}(\mathbf{w}_t) \approx \texttt{RUPEE}_{vec}(\mathbf{w}_t) \stackrel{\text{def}}{=} \sqrt{\left|\hat{\mathbf{h}}_t^\top \overline{\delta\mathbf{e}}^\beta\right|}, \tag{8.1}$$

or

$$\texttt{RMSPBE}(\mathbf{w}_t) \approx \texttt{RUPEE}_{scalar}(\mathbf{w}_t) \stackrel{\text{def}}{=} \sqrt{\left|\overline{\hat{\mathbf{h}}_t^\top \delta\mathbf{e}}^\beta\right|}. \tag{8.2}$$

These measures are Recent Unsigned Projected Error Estimates, and thus we call them both RUPEE. The first measure, RUPEE$_{vec}$, stores a moving average of $\delta\mathbf{e}$, and the second measure RUPEE$_{scalar}$ makes use of an average of a single scalar. Both estimators use an independent and additional weight vector, $\hat{\mathbf{h}}_t \in \mathbb{R}^n$, updated by the same LMS rule as

[2] In intrinsically motivated reinforcement learning learning progress is often defined as the derivative of the prediction error (see Oudeyer et al., 2007). Our notion of progress is simply an estimate of the MSPBE. We do not claim our notion of progress is more or less effective for intrinsically motivated reinforcement learning, but simply highlight the terminology difference compared to the literature.

GTD(λ), but not used by the GTD(λ) algorithm. This duplication of weights enables each estimator's learning-rate parameter $\alpha_{\hat{\mathbf{h}}} \in \mathbb{R}$ to be tuned independently of $\alpha_{\mathbf{h}}$.

The moving average of $\delta\mathbf{e}$ in RMSPBE$_{vec}$ is computed using a variant of an exponential moving average with an unbiased initialization,

$$\begin{aligned}
\tau_{t+1} &= (1-\beta_0)\tau_t + \beta_0 \\
\beta &= \frac{\beta_0}{\tau_{t+1}} \\
\overline{\delta\mathbf{e}}_{t+1} &= (1-\beta)\overline{\delta\mathbf{e}}_t + \beta\delta_t\mathbf{e}_t,
\end{aligned}$$

where $\beta_0 > 0$ is the user-specified averaging constant for the trace of $\delta\mathbf{e}$ and $\tau_0 = 0$. This method acts like a sample average when t is small, and smoothly transitions to a normal exponentially weighted moving average when t grows large. The moving average inside RUPEE$_{scalar}$ is computed in a similar fashion, as are all moving averages computed in this thesis.

Estimating the RMSPBE using either estimator provides a notion of learning progress contingent on the representational and data constraints induced by the choice of the demon's answer functions. The RMSPBE can be zero even when the demon's approximate GVF is very different than the true GVF, and even when the demon's predictions are not accurate in comparison to samples of the target G_t. The RMSPBE can be zero in these cases both because it is a projected objective and because the objective is weighted by d_μ (both discussed in the background chapter). In other words, the RMSPBE does not care about states the behavior never visits or value functions outside the representational class of the features, and neither do our RUPEE measures.

The TD error could be used to measure demon progress, but it is not ideal in our setting for two reasons. The squared TD error can be used to form an unbiased estimate of the mean squared Bellman error (MSBE). However, our TD-based demons optimize the MSPBE, and thus the MSBE would be a surrogate measure of MSPBE optimization. In addition, it is not clear how to use the TD error to estimate the MSBE when λ is greater than zero. Regardless of the relative merits and weaknesses of using an estimate of the RMSPBE to measure learning progress, a major benefit of the RMSPBE is that it is computable from observable quantities (see Appendix B for a discussion of objective functions for value function estimation).

Our new progress measures appear to satisfy the first three criteria we specified in the previous section. Both RUPEE measures can be updated on every time-step with computation and storage that is linear for each demon. They can be computed online from

immediately available data and is thus always up-to-date with recent experience. We do not need to interrupt learning with tests or use train and test phases, and thus the behavior is unrestricted, producing data for both learning and updating our estimates of the RMSPBE on every time-step. The the accuracy of these new RUPEE measures and their usefulness for measuring the progress of many demons is still unclear, and this is subject of the remaining sections of this chapter.

8.3 Experiments on the Markov chain

In our first set of experiments, we are primarily concerned with how well our RUPEE measures estimate the RMSPBE of the GTD(λ) algorithm in several simple simulation problems. In addition, we also investigate how well our new measures tracks the RMSPBE of the GTD(λ) algorithm when a large change occurs which causes a spike in the MSPBE. In later sections of this chapter we will investigate how well both RUPEE measures relate empirically with prediction accuracy, and investigate estimating the RMSPBE of thousands of demons on a robot. We begin with an experiment to investigate both RUPEE measures in a simulation domain that affords exact computation of the RMSPBE.

8.3.1 Comparing RUPEE and the RMSPBE

Our first experiment seeks to answer the question: "How well do both RUPEE measures estimate the RMSPBE of the GTD(λ) algorithm compared to several other possible estimators?". By "other possible estimators" we mean different ways to compute RUPEE, as well as other ways to estimate the RMSPBE that use either parameters of the MDP or more than linear computation and storage. Our experiments on the GTD(λ) algorithm suggest that the secondary weights can be learned well at the expense of slightly higher RMSPBE, and thus we might expect that both RUPEE measures, which are based on GTD(λ), might do a good job approximating the RMSPBE. We hypothesize that the estimators of the RMSPBE that use the parameters of the MDP or greater than linear computation will do a better job approximating the RMSPBE than either variation of RUPEE.

Experiment

The problem specification, question functions, and answer functions were the same as those used in the experiments of Chapter 7. We used the Markov chain domain, with the same five combinations of behavior and target policy probabilities, and the same feature vectors given in our original problem description. In all experiments, λ was taken to be a constant

function equal to 0.9. A single state GVF was learned (for each problem instance) by a single prediction demon using the GTD(λ) algorithm with α and $\alpha_{\mathbf{h}}$ set to the values that minimize the RMSPBE found in the parameter sweep described in Chapter 7 (Table 7.1, specifically).

In order to access the performance of proposed measures, we defined four alternative estimates of the MSPBE. The first three estimators use computation and storage that is greater than linear:

- $\texttt{RMSPBE1}(\mathbf{w}_t) = \sqrt{\left|(\overline{\delta \mathbf{e}})^\top (\overline{\mathbf{x}\mathbf{x}^\top})^{-1} \overline{\delta \mathbf{e}}\right|}$
- $\texttt{RMSPBE2}(\mathbf{w}_t) = \sqrt{\left|(\overline{\delta \mathbf{e}}^{\beta})^\top \mathbf{h}_t^\star\right|}$
- $\texttt{RMSPBE3}(\mathbf{w}_t) = \sqrt{\left|\mathbb{E}_\mu[\mathbf{e}\delta]^\top \hat{\mathbf{h}}_t\right|}$.

The RMSPBE1 estimator computes the least squares estimate of the RMSPBE from a history of all previously observed features and cumulants. The $\bar{\cdot}$ denotes the sample long run average. The RMSPBE2 estimator combines a linear estimator of $\mathbb{E}_\mu[\delta_t \mathbf{e}_t]$ with the value of $h_t^\star = \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}_\mu[\delta_t \mathbf{e}_t]$ as defined by the parameters of the MDP. Finally, the RMSPBE3 estimator combines the true value of $\mathbb{E}_\mu[\mathbf{e}_t \delta_t]$ with an estimate of $\mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}_\mu[\delta_t \mathbf{e}_t]$ computed similarly to RUPEE$_{vec}$ (an LMS rule).

The remaining estimator can be considered an alternative implementation for RUPEE:

$$\texttt{RMSPBE4}(\mathbf{w}_t) = \sqrt{\left|\hat{\mathbf{h}}_t^{\ \top} \delta_t \mathbf{e}_t\right|}$$

The RMSPBE4 estimator uses only instantaneous samples to estimate the RMSPBE, which might yield less bias but higher variance compared to RUPEE$_{vec}$ and RUPEE$_{scalar}$ in some domains. RMSPBE4 has no memory term, compared to the $O(n)$ storage of RUPEE$_{vec}$.

In our first experiment, we performed a parameter sweep to find the best parameters values for each estimator, in each instance of the Markov chain problem. The parameters were chosen from $\alpha_{\hat{h}} \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.05, 10^{-1}$,0.25 ,0.5, 0.75, 1.0$\}$ and $\beta_0 \in \{$0.001, 0.012, 0.023, 0.034, 0.045, 0.056, 0.067, 0.078, 0.089, 0.1, 0.25, 0.5, 0.75, 1.0$\}$. Each parameter combination was evaluated at the end of 200 episodes using all 200 RMSPBEs and estimator values observed during the run:

$$\text{estimator-strength}(\hat{\theta}) = \frac{\text{the correlation between RMSPBE and } \hat{\theta} \text{ after 200 episodes}}{\text{the total squared difference between RMSPBE and } \hat{\theta} \text{ over 200 episodes}}, \tag{8.3}$$

The parameter estimate $\hat{\theta} \in \mathbb{R}$ was one of {RMSPBE1, RMSPBE2, RMSPBE3, RMSPBE4, RUPEE$_{vec}$, RUPEE$_{scalar}$}. An estimator performs well when the estimator strength is large and positive. We defined 8.3 in order to provide one metric to optimize the parameter's of RUPEE and the other estimators for comparison. Once the best parameters were selected we investigated several measures of performance.

All six estimators of the GTD(λ) algorithm's RMSPBE were run on each of the five problem instances, and then run for 200 episodes. Then the whole thing was repeated for 100 runs. The random seed was initialized to the same value for each estimator instances at the beginning of each group of 100 runs. The GTD(λ) algorithm's weight vectors and the vectors and averages of each estimator (including RUPEE$_{vec}$ and RUPEE$_{scalar}$) were all set to zero at the beginning of each run. For each run, the estimator's strength (according to Equation 8.3) was recorded and averaged over runs. We selected the β_0 and $\alpha_{\hat{\mathbf{h}}}$ that achieved the best estimator's strength for each estimator, on each problem instance.

To answer our question posed at the beginning of this section, we performed an experiment using the parameter settings found in the previous experiment. In this experiment, the GTD(λ) algorithm was run on each of problem instance for 200 episodes with the RMSPBE estimate of each of the six estimators recorded at the end of each episode. Then the whole thing was repeated for 200 runs. We recorded the closeness of each estimator to the true RMSPBE of the GTD(λ) algorithm, after each episode i, using three different measures:

- empirical-bias$(\hat{\theta}_i) = \hat{\theta}_i -$ (RMSPBE at the end of episode i)
- squared-error$(\hat{\theta}_i) =$ (empirical-bias$(\hat{\theta}_i))^2$
- correlation$(\hat{\theta}_i) =$ the correlation between RMSPBE and $\hat{\theta}_i$ up to episode i.

Each of the three measures for each of the six estimators (including RUPEE$_{vec}$ and RUPEE$_{scalar}$) was averaged over runs, and the weight vectors of the GTD(λ) algorithm and data structures of all the estimators were set to zero at the beginning of each run. Figure 8.1 presents the results. We also include a plot of the RMSPBE of the GTD(λ) algorithm and the estimates of the RMPBSE according to RUPEE$_{vec}$, RUPEE$_{scalar}$, RMSPBE1, and RMSPBE4 over episodes.

Results and conclusions

The results show that RUPEE$_{vec}$ achieved lower empirical bias, lower squared error, and higher correlation than the RMSPBE4 and RUPEE$_{scalar}$ estimators. The RMSPBE1 estimator (dashed line) performed best across all five problems and all three measures. The RMSPBE2 (cyan) and RMSPBE3 (purple) estimators achieved better empirical bias, squared difference, and correlation on the problem instances where the target policy was less random, but in problem instances ($p_\mu = 0.5, p_\pi = 0.5$) and ($p_\mu = 0.6, p_\pi = 0.4$) the RUPEE$_{vec}$ was nearly as good, especially according to the correlation measure. The RMPSBE4 (red) estimator performed the worst according to all measures in all problem instance. All estimators besides RMSPBE1 and RMSPBE2 exhibited non-vanishing bias in problem instances ($p_\mu = 0.6, p_\pi = 0.4$), while only RMSPBE1 and RUPEE$_{vec}$ exhibited diminishing bias in problem instances ($p_\mu = 0.5, p_\pi = 0.5$). This may be due to limited sampling (not enough episodes), and optimizing the estimator's parameters, over a window of episodes. The plots of the estimators over episodes shows that RUPEE$_{vec}$ (blue) tracks the RMSPBE (black) well, and that the RMSPBE1 estimator is nearly equal to the RMSPBE.

Overall we conclude from these results that RUPEE$_{vec}$ estimates the RMSPBE well across several problem instances compared to several other estimators. RUPEE$_{vec}$ performed better than the other two more computationally frugal estimators, RMSPBE4 and RUPEE$_{scalar}$. The RMSPBE4 performed particularly badly, suggesting that the averaging performed in both RUPEE measures is helpful. The RMSPBE1 estimator performed best overall, but it uses substantial computation and storage which would make it difficult to update quickly with many demons. The RMSPBE2 performed next best overall, indicating that knowledge of $\mathbf{h}^\star(\mathbf{w}_t) = \mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)]^{-1}\mathbb{E}_\mu[\mathbf{e}_t\delta_t]$ helps more than knowledge of $\mathbb{E}_\mu[\mathbf{e}_t\delta_t]$ in these problem instances.

8.3.2 A non-stationary domain

Our next experimental question concerns "How well does RUPEE$_{vec}$ track the RMSPBE of the GTD(λ) algorithm when a demon experiences a single change which invalidates its current predictions?". This question is motivated by a situation that might occur while learning many GVFs on a robot: perhaps, the robot's wheels malfunction, or a human turns off the lights in the lab. In either case, several of the demon's predictions may become incorrect, and it would be useful if our estimate of the RMSPBE could reflect the change in the world. This would be useful to human experimenters enabling analysis of what happened, and it would be useful to the robot so that the robot might change its behavior in

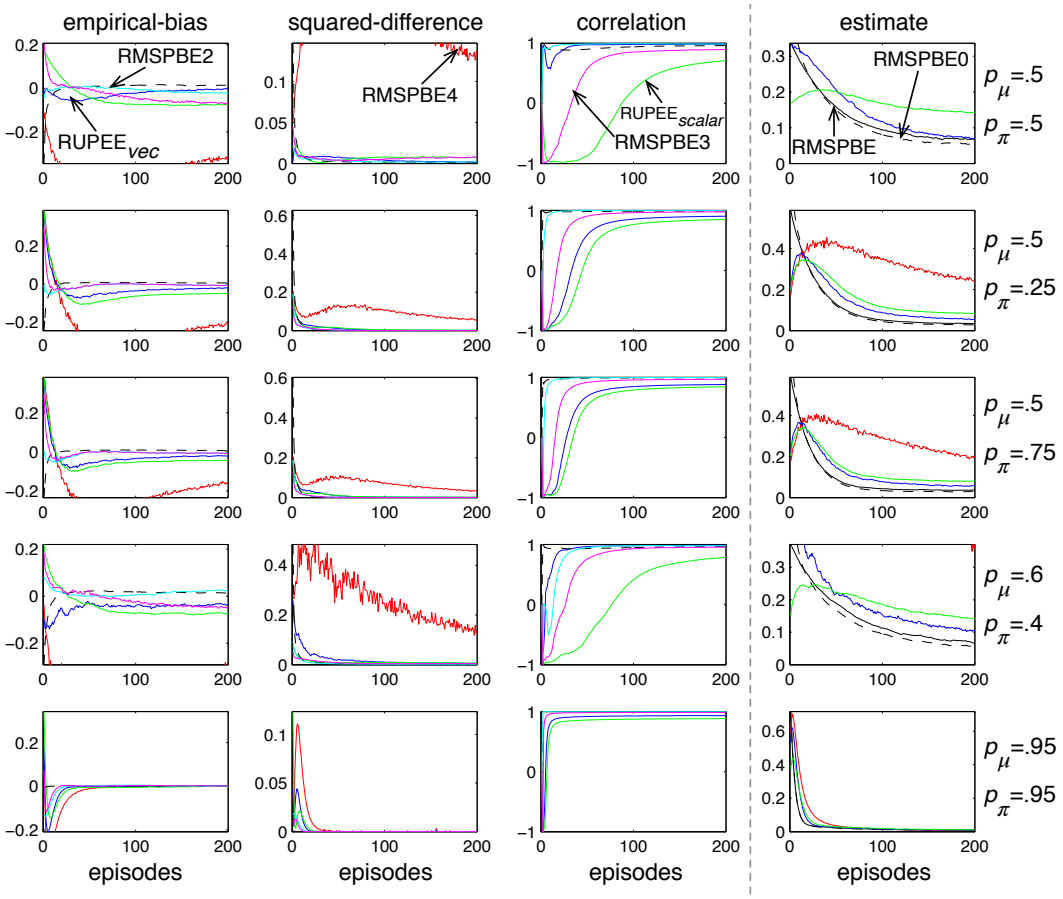

Figure 8.1: An evaluation of RUPEE$_{vec}$ and RUPEE$_{scalar}$ compared to four other estimators for estimating the RMSPBE of GTD(λ) on five instances of the Markov chain task. Each row of the figure correspond to an instance of the Markov chain problem. Each of the first three columns corresponds to a different evaluation measure, while the last column simply plots the RMSPBE of the GTD(λ) algorithm and several estimates of the RMSPBE versus episode number. See text for a description of the experiment and discussion of the results.

an effort to more efficiently relearn the inaccurate predictions.

The key challenge for RUPEE$_{vec}$ here is to detect the large increase in the RMSPBE of the GTD(λ) algorithm via observations of the GTD(λ)'s TD error, before relearning returns the RMSPBE to normal pre-change levels. It is difficult to hypothesize how quickly RUPEE$_{vec}$ will track the change in RMSPBE, however we are optimistic that the linear LMS rule and the moving average used inside RUPEE$_{vec}$ should adapt fairly quickly. We omitted RUPEE$_{scalar}$ from this experiment because RUPEE$_{vec}$ outperformed RUPEE$_{scalar}$ in all five variants of the chain domain in the previous experiment.

Experiment

In this experiment we compared RUPEE$_{vec}$'s estimate of the RMSPBE to the true RMSPBE of the GTD(λ) algorithm in each of the five chain problem instances over 400 episodes. As before, a single demon was used to learn the state GVF. This time, the $\mathbf{w}$ vector of the GTD(λ) algorithm was negated at the beginning of the 200th episode, $\mathbf{w}_{200} = -\mathbf{w}_{200}$, to simulate a large change in the environment. Aside from this modification, the experiment was run as before. The $\alpha_{\hat{\mathbf{h}}}$ and β_0 parameters were the same as those used in the previous experiment, except for the fourth chain instance where we increased the value of β_0 slightly. Each combination of problem instance and RUPEE$_{vec}$ was run for 400 episodes. Then the whole thing was repeated for 400 runs, with the weight vectors of GTD(λ), ($\mathbf{w}$, $\mathbf{h}$), and the $\hat{\mathbf{h}}$ and $\overline{\delta\mathbf{e}}^{\beta}$ of RUPEE$_{vec}$ all initialized to zero at the beginning of each run. Figure 8.2 plots the RUPEE$_{vec}$ and the true RMPSBE after each episode, averaged over runs.

Results and conclusions

The results indicate that RUPEE$_{vec}$ tracks the true RMSPBE of the GTD(λ) algorithm in each of the five Markov chain problem instances. The RUPEE$_{vec}$ measure reacts to the change in the TD errors of the GTD(λ) algorithm, increasing its estimate of the RMPSBE well before the true RMSPBE is suppressed by relearning.

We conclude from this experiment that RUPEE$_{vec}$ can react to changes in the world that cause an increase in the RMSPBE. Any changes in the accuracy of the GTD(λ) algorithm's predictions are immediately reflected in the RMSPBE. In comparison, RUPEE$_{vec}$ uses linear complexity operations and two simple memories (the exponential average and the $\hat{\mathbf{h}}$ vector) of the TD errors to estimate and track changes in the RMSPBE. Overall, in these chain tasks, it seems RUPEE$_{vec}$ provides a reasonable estimate of RMSPBE, especially given that it meets our somewhat restrictive criteria of being always up-to-date and computationally efficient.

8.4 Experiments on the Critterbot

Consider the task of learning and accessing the learning progress of many prediction demons, perhaps thousands of demons, on a robot. On a robot, we do not have access to the parameters of the MDP, so we cannot compute the RMSPBE. In addition, our computation is very limited, and thus it is not practical to compute the least squares estimate of the RMSPBE (e.g., RMSPBE1 from the previous section) for more than a handful of demons. In this

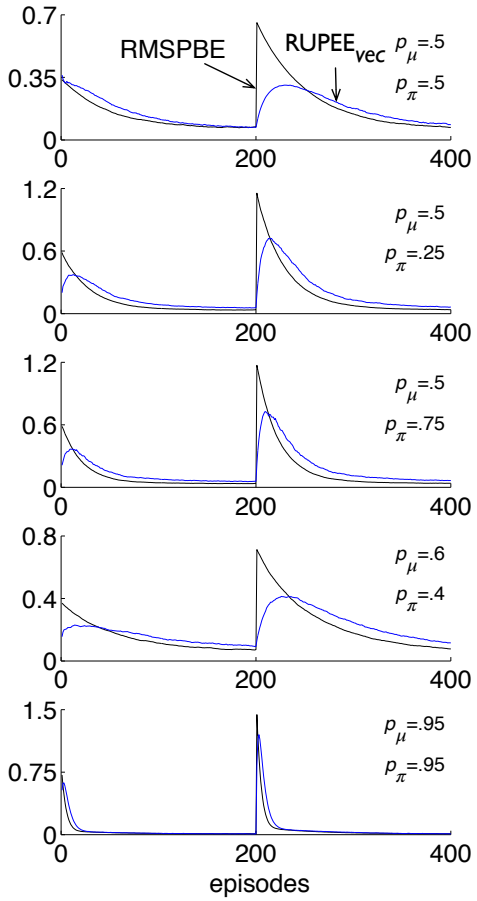

Figure 8.2: A comparison of the RMSPBE of the GTD(λ) algorithm (black) and the RUPEE$_{vec}$ measure's estimate of the RMSPBE (blue) on a non-stationary variant of the Markov chain domain. Each subplot corresponds to each of five Markov chain problem instances. The GTD(λ) algorithm learned for 200 episodes, and then the primary weight vector was modified: $\mathbf{w}_{200} = -(\mathbf{w}_{200})$. See text for a description of the experiment and discussion of the results.

setting, we can estimate learning progress by using either RUPEE measure or some other method that is not based on the RMSPBE.

Another method for estimating progress in this scenario is to compute the accuracy of each demon's predictions. If we could sample each demon's target (G_t), then we could compute the RMSE of each prediction. Estimating prediction accuracy in this way on a robot, however, is not ideal in many ways, as you will see in the following experiments. If the prediction accuracy and the RUPEE measures exhibited similar empirical convergence profiles and both indicated a plateau in performance improvement at roughly the same time, then we might avoid these problematic prediction accuracy estimates and just use one of the

RUPEE measures. The experiments of this section investigate the empirical relationship between RUPEE$_{vec}$, RUPEE$_{scalar}$, and prediction accuracy across hundred of demons.

8.4.1 Prediction accuracy of many demons

In order to compare RMSPBE estimation and prediction accuracy, we first develop a framework for estimating the prediction accuracy of policy-contingent predictions on a robot. In this experiment, we use interspersed on-policy test excursions during learning to evaluate the accuracy of each demon's prediction. We attempt to answer a simple question: "Can the robot learn hundreds of prediction demons in realtime, while measuring prediction accuracy with on-policy test excursions?". Off-policy state GVF learning of hundreds of predictions on a robot has never been demonstrated before. Our first experiment concerns learning many off-policy GVFs with performance evaluated using test excursions, not with RUPEE. Later we will use the same setup to investigate measuring progress with RUPEE compared to test excursions.

Problem

The robot's interaction with its environment was structured in a 100 millisecond (ms) time-step. At each step, the sensory information was used to select one of five actions corresponding to basic movements of the robot (*forward, backward, turn right, turn left*, and *stop*). Each action caused a different set of voltage commands to be sent to the three motors driving the wheels. The state of the robot was characterized by 49 real or virtual sensors of 13 types, summarized in the first two columns of Table 5.1 of Chapter 5. We exclude the stall flag and the previous wheel commands which were used in the nexting experiment).

The question functions of each demon are specified as follows. The ith demon is constructed using one of five possible target polices, each one corresponding to execution of one of the robot's actions repeatedly (e.g., $\pi^{(i)}(s, forward) = 1 \forall s \in \mathcal{S}$), which we call a constant-action policy. The termination signal of the ith demon was a constant function taking on one of three possible values: $\{0.0, 0.5, 0.8\}$ corresponding to time scales of 0.1 sec, 0.3 seconds, and 0.5 seconds. The cumulant of the ith demon, $Z_t^{(i)}$, corresponds to one of the robot's 49 sensors, for example $Z_t^{(i)} = \text{IRDistance0}_t$, just as in the nexting experiment. The question asked here is what will happen next if the robot were to execute a target policy differing from nexting in that the predictions are about many different ways of behaving. In total we have 735 prediction demons.

The answer functions are similar to the nexting experiment except for the behavior policy. The behavior policy selects one of the five actions randomly with equal probability, and then repeatedly executes the selected action with probability 0.5, and with 0.5 probability randomly selects a new action. The feature vector is constructed in the same way as in the nexting experiment, except the tile coding of the thermal sensors was corrected to include all eight sensors, producing a binary feature vector of length 6065, of which 473 components are equal to one on each time-step (see Table 5.1 of Chapter 5 to refresh your memory). Finally, the λ function was again constant and equal to 0.9.

Experiment

In order to evaluate the prediction accuracy, we interspersed on-policy tests into the behavior policy. During each test, the robot selects actions in agreement with one of the five target policies (on-policy), selected randomly at the beginning of the test and called the *testing policy*. While the testing policy was executing, we calculated a truncated sample of the target for precisely 147 of the demons whose target policy was equal to the testing policy. To do this, the test policy was followed for a fixed number of steps that was long enough to compute a reasonably accurate truncated sample of the target for each of the three termination signals $\{0, 0.5, 0.8\}$. The length for the test was set to 50 steps. This number was produced by calculating the number of steps T, for which the value of $(0.8)^T$ in the estimation of the target, $G_t^{(i)}$, would be less than a small tolerance level (equal to 0.00001 by the formula $\frac{\ln(0.00001)}{\ln(0.8)}$ and rounding up). At the beginning of the kth test phase, we recorded the demon's prediction, denoted $V_k^{(i)} \in \mathbb{R}$, for each of the 147 demons, and incrementally computed a truncated sample of each demon's target, denoted $\widehat{G^{(i)}}_k \in \mathbb{R}$:

$$\widehat{G^{(i)}}_k = \sum_{j=0}^{50} (\gamma_t^{(i)})^j Z_{t+j+1}^{(i)},$$

where t denotes the time-step of the beginning of the test. After 50 test steps, indexed above by j, each prediction was compared to its truncated sample of the target and the squared error, recorded $(\widehat{G^{(i)}}_k - V_k^{(i)})^2$. As a measure of the quality of the prediction sequence $\{V_k^{(i)}\}$ of demon i up through test number K, we can use the root mean squared error, defined as:

$$\text{RMSE}(i, K) = \sqrt{\overline{(V^{(i)} - \widehat{G^{(i)}})^2}^{\beta}},$$

where β is the averaging parameter of the moving average incrementally computed from samples. The β_0 parameter of moving average was equal to 0.001 in this experiment. We

use a different definition of RMSE from the one used in Chapter 5 because we have significantly less samples to compute errors from (>100,000 in nexting, verses hundreds here). This means the RMSE curves of each demon may be slightly more noisy and the average will overcome initial errors more quickly.

After every test, the robot followed a hand-coded recovery policy to move the robot back to the center of the pen. This recovery phase did not last longer than two seconds, and during this time learning was disabled. The recovery policy was found to be useful to balance the amount of training time the robot spent in free space and near the walls. Algorithm 8.4.1 provides pseudo code for how the behavior, on-policy test excursions, and recovery occurred during our experiment. No learning occurs during test excursions or recovery.

Algorithm 1 Off-policy demon training with interspersed on-policy tests

let $\mathbf{x}$ be the initial feature vector
for each time-step $t \approx 100ms$ **do**
  if (**not** testing) **then**
    ## following the behavior policy and learn
    $A \leftarrow \mu(\mathbf{x}, \cdot)$
    take action A and observe next feature vector $\mathbf{x}'$
    update each demon using sample $(\mathbf{x}, A, \mathbf{x}')$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    interrupt learning probabilistically
      ## start test
      select test policy $\pi^{(j)}$ from set of five target policies
      for each demon i whose $\pi^{(i)} = \pi^{(j)}$ **do**
        record predictions: $V^{(i)} = \mathbf{x}^\top \mathbf{w}^{(i)}$
        $\widehat{G^{(i)}} = 0$
      $k = 0$; testing = *true*
  else
    ## on-policy test following target policy j
    $a \leftarrow \pi^{(j)}$
    for each demon with $\pi^{(i)} = \pi^{(j)}$ **do**
      update truncated target: $\widehat{G^{(i)}} \leftarrow \widehat{G^{(i)}} + (\mathbf{w}^{(i)})^k Z_{k+t+1}^{(i)}$
    $k \leftarrow k + 1$
    if $k \geq 50$ **then**
      ## `test concluded`
      testing = *false*
      for each demon with $\pi^{(i)} = \pi^{(j)}$ **do**
        record MSE: $(V^{(i)} - \widehat{G^{(i)}})$
      execute recovery policy and observe current feature vector $\mathbf{x}$ after recovery
    end if
  end if
end for

In order to to evaluate the accuracy of all 749 predictions about sensors at various time scales, we aggregated the performance across all predictions. To measure the accuracy of predictions with different magnitudes, we used a normalized mean squared error,

$$\text{NMSE}(i,k) = \frac{\text{RMSE}^2(i,k)}{\text{var}(i)},$$

in which the mean squared error is scaled by $\text{var}(i)$ (the sample variance of the truncated targets from all K tests of the ith demon's prediction). This error measure can be interpreted as the percent of variance *not explained* by the prediction. It is equal to one when the prediction is constant at the average target value, but can be much larger than one. Several demon's curves were excluded from the results because their truncated targets were always constant and had zero variance (e.g., the ones corresponding to the motor current of motor two).

In addition to the NMSE, we also recorded another aggregate measure of accuracy based on the absolute error. The symmetric mean absolute percentage error, or SMAPE,

$$\text{SMAPE}(i,k) = \overline{\left(\frac{|V_k^{(i)} - \widehat{G^{(i)}}_k|}{|V_k^{(i)}| + |\widehat{G^{(i)}}_k|} \right)}^{\beta}, \tag{8.4}$$

where β is the parameter to the moving average over tests, and β was equal to 0.001 in our experiments. The SMAPE is bounded between zero and one. The SMAPE is undefined if $|V_k^{(i)}| + |\widehat{G^{(i)}}_k|$ is zero, and this situation arose for 15 demons (again the predictions of future motor current of motor two). These predictions were excluded from the calculations involving SMAPE.

We ran the Critterbot for 6.7 hours, updating 749 demons with the parameters of GTD(λ) equal to: $\alpha = (1-\lambda)(\alpha_0)/||\mathbf{x}|| = (1-.9)(.1)/473$ and $\alpha_{\mathbf{h}} = 0.01\alpha$. The run produced 238,000 samples, and approximately half of the time was spent performing test excursions. Data was logged to disk for later analysis. Each prediction was tested 400 times, yielding the aggregate learning curves for NMSE and SMAPE shown in Figure 8.3.

Results and conclusions

The results of this experiment indicate that hundreds of predictions can be updated and tested in realtime on the Critterbot. The randomized behavior policy, tile-coding, test excursions, recovery policy, and GTD(λ) were all implemented in Java and run on a laptop computer connected to the robot by a dedicated wireless link. The laptop used an Intel quad-core processor with a 2.7GHz clock cycle, and 16GB of DDR3 RAM. Eight threads

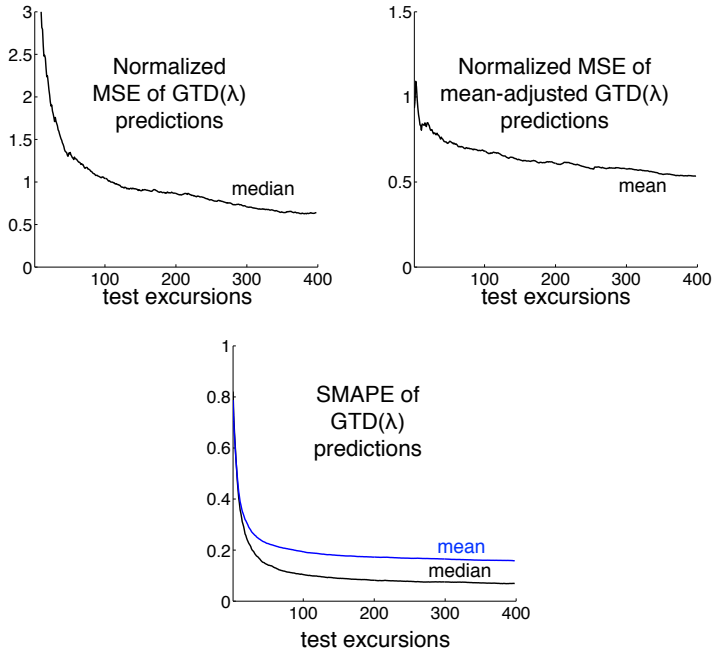

Figure 8.3: Aggregate learning curves for the 749 predictions whose cumulant corresponds to each sensor and one of five constant-action target policies. Each point in each subgraph is the aggregate error of the prediction of the demons up to that test. The top left plot shows the median NMSE curve, and the bottom plot shows the mean and median SMAPE curves. The top right curve shows what the the mean NMSE would be if the experiment were rerun using cumulants with their average value subtracted out.

were used for the learning code. With this setup, the time required to make and update all 749 predictions was 2.2 ms, well within the 100 ms cycle of the Critterbot.

The results in Figure 8.3 report the overall accuracy of the predictions learned by each demon. The top left plot reports the median NRMSE computed from the individual predictions, achieving 40% variance explained by the end of training. The learning curve in the top right plot reports the average NSME, when the cumulant reward of each prediction was adjusted by the mean of the cumulant (as done in Chapter 5). When we modified our cumulants in this way, we reran GTD(λ) on the logged data. In the mean, the prediction learned with the average subtracted explained 49% of the variance of the target by the end of the data set. The bottom plot reports the average and median prediction SMAPE curves. The SMAPE value at the end of the data set was 15.8% and 0.67% for the average and median aggregations, respectively.

We also recorded accuracy measures for predictions aggregated by sensor group. The predictions whose cumulant (not mean adjusted) was an external sensor (Thermal, IRLight,

IRDistance, Mag, Light) explained 52% of the variance by the end of the run, while the predictions whose cumulant was an internal sensor (motor information, acceleration, rotational velocity, etc.) explained only 24% of the variance at the end of the run. Using absolute error to aggregate prediction accuracy, the external sensor predictions achieved 5.8% median SMAPE, and the internal sensor predictions achieved 42% median SMAPE by the end of the run.

In conclusion, these results indicate that it is indeed practical to do large-scale prediction learning, off-policy, and on a robot with conventional computational resources, and that the prediction accuracy can be measured by on-policy test excursions. Overall, no divergence was observed, and the predictions achieved reasonable accuracy.

These policy-contingent predictions are not as accurate as the nexting predictions learned in Chapter 5. We used the same robot, a similar learning algorithm, and the same feature generation mechanism, but the important difference was the policies. In the nexting experiment, the robot generated long trajectories of experience (several minutes between overheating stalls), observing similar sensory patterns with each loop around the pen. Each sample generated by the robot was relevant to every single demon on every time-step. In the off-policy experiment, the data generated by behavior was only ever relevant to a subset of demons at any one time. In addition, the behavior produced short trajectories of on-policy experience before switching to another policy: just snippets of experience to update each demon. Of course there may have been other factors at play, but it seems the simple answer is the off-policy learning problem was more challenging than the task of on-policy nexting.

8.4.2 Comparing RUPEE and prediction accuracy

Although it may not be apparent from the results of the previous section, estimating prediction accuracy using interspersed on-policy test excursions is not entirely compatible with what we mean by large-scale prediction learning on a robot. The first source of incompatibility is caused by the need to follow each target policy many times, limiting the number of target policies the robot can learn about at one time. Second, this test excursion framework limits us to consider GVFs with short time scales (γ). We must test each target policy, and thus we must run each test long enough to form a reasonable approximation of the infinite sum of the target. A prediction defined by an eight second time scale, for example, would need at least a 90 second test. Using on-policy tests for evaluation restricts scaling by limiting the number of distinct what if questions we can ask, and the span of each prediction.

Using on-policy test excursions to evaluate policy-contingent predictions makes our experiment apparatus unnecessarily complex. We need to take care that the tests are long enough, and frequent enough to ensure the estimated accuracy is up to date and reflective of the robot's learning progress. If tests are frequently started near a wall, but most behavior data was generated in open space, then the prediction accuracy might indicate no learning has occurred. On the other hand, frequent testing consumes valuable robot time that could have been used more effectively by the behavior policy, and striking the right balance between testing and training is hard. Perhaps we can use one of RUPEE$_{vec}$ or RUPEE$_{scalar}$ to estimate learning progress and avoid many of these difficulties.

The question we seek to answer with our next experiment is: "Does the learning progress of prediction demons measured by RUPEE$_{vec}$ and RUPEE$_{scalar}$ empirically correspond to the prediction accuracy measured by the RMSE?". Our previous experiments with RUPEE$_{vec}$ show that it corresponds well with the RMSPBE on simulation domains. In our next experiment we seek to compare our new measures to RMSE of predictions learned on a robot.

Experiment

In this experiment, we simply used the log file produced by the last experiment to generate aggregate plots of RUPEE$_{vec}$, RUPEE$_{scalar}$, and the RMSE for comparison. The log file contained approximately six hours of data: two hours and 52 minutes of the log corresponded to time-steps where the behavior policy was in control and the demons were updated (or 103,200 time-steps). For this experiment, we consider a subset of only 245 demons corresponding to the longest time scale, $\gamma_t = 0.8$. In addition, all the cumulants are scaled between zero and one, using the minimum and maximum values for each sensor observed in the log. We used a subset of demons and scaled cumulants in order to facilitate aggregating many MSE and RUPEE curves for comparison.[3] As before, each demon was tasked to predict the future values of a sensor, if actions were selected according to one of the five constant-action target policies. Each demon used the GTD(λ) algorithm to update its prediction incrementally from the log file, and the parameters of GTD(λ) were the same as the previous experiment.

We compared RUPEE$_{vec}$, RUPEE$_{scalar}$, and the RMSE from test excursions, all computed from the log of data. RUPEE$_{vec}$ and RUPEE$_{scalar}$ of each demon was recorded for

[3]We cannot compute a variance unexplained variant of RUPEE, thus comparing aggregate (e.g., median) NMSE curves and RUPEE curves would be miss-leading.

every time-step in the log file, but only updated on every non-test time-step. The RMSE was also recorded for every demon on each time-step, only updated after the completion of each test. The averaging parameter of RUPEE$_{vec}$ was set to $\beta_0 = (1-\lambda)\alpha_0/30$, and the averaging parameter of RUPEE$_{scalar}$ was set to $\beta_0 = (1-\lambda)\alpha_0/20$. Both RUPEE measures used $\alpha_{\hat{\mathbf{h}}} = 5\alpha$. The averaging parameter β, for the RMSE, was set to 0.0005.

Mirroring our experiments in the Markov chain, we reset the primary weight vector of each demon, $\mathbf{w}^{(i)} = -\mathbf{w}^{(i)}$, at the halfway point of the log to induce a sudden and unexpected change. The reset caused each demon to predict the opposite of what it had learned up to the halfway point, and should significantly perturb both the prediction accuracy and RMPBSE of all the demons. Figure 8.4 plots the median RMSE, RUPEE$_{vec}$, and RUPEE$_{scalar}$ curves computed from the log file, sampled on the time-step that the behavior policy was in control in the original experiment.

Results and conclusions

The results in Figure 8.4 show that the RMSE from test excursions and both RUPEE measures have roughly the same overall shape, aggregating via the mean, and all three react similarly to the reset of the weights at the halfway point of the experiment. Using the median to aggregate errors resulted in similar curves, which was expected because the cumulants were scaled to the zero-one range.

Notice that the mean RMSE, mean RUPEE$_{vec}$, and mean RUPEE$_{scalar}$ though similar in profile are not equal in magnitude. The RMSE does not equal the two RUPEE measures because they estimate different quantities: the prediction error and the RMSPBE. Given enough data the RUPEE measures should both approach zero. The RMSE, on the other hand, will likely never achieve zero on robot data for a number of reasons including but not limited too (1) the features do not capture the state of the environment, and (2) the testing regiment does not precisely reflect the true accuracy of the predictions (e.g., the tests sometimes occur in situations which have correspondingly less training data). The two RUPEE measures achieve similar magnitude but are not exactly equal either because they used different averaging schemes and thus will likely have different biases. There may be cases when all three measure are nearly equal (e.g., tabular features), but that is not the case here. We have plotted each measure separately to focus of the shape of the curves rather than the precise magnitude, which are expected to differ in general.

We conclude from the results of this experiment, that the estimates of the RMSPBE

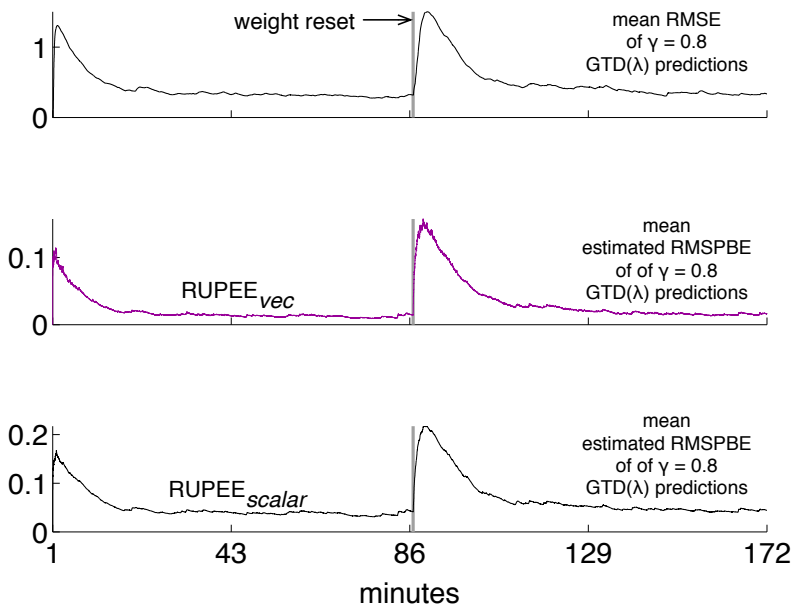

Figure 8.4: Off-policy learning progress of 245 predictions learned by the GTD(λ) algorithm, as measured by the RMSE from on policy test excursions and two estimates of the RMSPBE. The figure shows the mean RMSE, mean RUPEE$_{vec}$, and mean RUPEE$_{scalar}$ plotted against time. The mean of each curve was computed from the learning curves of 245 predictions. A sudden change (marked by the gray vertical line at the mid-way point) was simulated by reseting each prediction's primary weight vector, $\mathbf{w}^{(i)} = -\mathbf{w}^{(i)}$. The change caused the demon's predictions to become inaccurate and cause all three measures to increase dramatically.

of the GTD(λ) algorithm and the RMSE of its predictions exhibit similar profiles. Furthermore, the RMSPBE estimates seem to react just as quickly as the RMSE to a sudden change in the accuracy of the demon's predictions. Our experiment provides evidence that both RUPEE measures can track the learning progress of many demons on a robot, and the RUPEE measures can also track the improvement in prediction accuracy of those demons over time.

8.4.3 Many demons and many target policies

Our last experimental investigation concerns scalability of RMSPBE estimation. So far, our experiments have explored learning about only a few target policies at a time. Given enough computation and sufficient training data, we may be able to learn about many different target policies in realtime and in parallel. Using RUPEE$_{vec}$, we may be able to estimate the learning progress of each demon, and thus we may be able to greatly scale up the number of distinct target policies without ever having to run a single on-policy test excursion. We

seek to answer: "Is it practical to perform the GTD(λ) update and estimate RUPEE$_{vec}$ for many demons with many distinct target policies, all within the update cycle time of the Critterbot?". Additionally, can we do it with at least as many demons as the nexting experiments of Chapter 5? Our final experiment seeks the answer to these questions.

Problem

In order to generate a large number of distinct target policies, we use randomly parameterized probabilistic policies. Each policy is represented as a discrete distribution over actions. The next action is sampled probabilistically on each time-step from a discrete-action linearly parameterized Gibbs distribution,

$$\pi^{(i)}(\mathbf{x}, a) = \frac{e^{(\mathbf{x}^\top u^{(i)}(a))}}{\sum_{a' \in \mathcal{A}} e^{(\mathbf{x}^\top u^{(i)}(a'))}},$$

where $u \in \mathbb{R}^{n|\mathcal{A}|}$ is a vector of policy parameters and $u(a) \in \mathbb{R}^n$. Each policy is generated by selecting 50 components of $u(a)$ at random at the beginning the experiment and then assigning each component a value independently drawn from a normal distribution with mean zero and variance one. In our experiment we generated 300 different probabilistic policies.

The only other significant change from our previous experiments was how the robot behaved. We eliminated the on-policy test excursions, and periodically invoked the recovery policy to ensure the robot's behavior was not concentrated near the walls of the pen. The behavior policy and the λ function were exactly the same as before.

Experiment

We instantiated 2500 prediction demons by randomly combining one of the 300 target policies, a cumulant equal to one of the robot's sensors, and a constant termination signal from the set $\{0.0, 0.5, 0.8, 0.95\}$. The randomization process checked to ensure that each demon was created with a unique combination of $\pi^{(i)}$, $\gamma^{(i)}$, and $Z^{(i)}$. Each demon used the GTD(λ) with $\alpha = (1-\lambda)\alpha_0/||\mathbf{x}|| = (1-.9)(0.05)/473$, and $\alpha_{\mathbf{h}} = 0.001\alpha$.

The robot was run for six hours, updating 2500 demons and estimating learning progress with RUPEE$_{vec}$.The RUPEE$_{vec}$ estimator was used with $\beta_0 = (1-\lambda)\alpha_0/100$ and $\alpha_{\hat{\mathbf{h}}} = 5 * \alpha_0$. Figure 8.5 shows the estimated mean and median RMSPBE curves aggregated over all 300 demons, and a sampling of estimated RMSPBE curves for 30 demons.

Results and conclusions

The time required to make and update all 2500 predictions was 91 ms, well within the 100 ms cycle of the Critterbot. Roughly a quarter of the demons used $\gamma = 0.95$, which would need a 22 second excursion to estimate the RMSE under the on-policy testing framework. Appendix E contains additional timing results and a discussion of the parallel scalability of Horde. It is not entirely appropriate to compute the median and mean RMSPBE curves because the demons have very different time scales and cumulant ranges. Nevertheless, we do so to give an idea of overall RMSPBE reduction, as well as plotting the RMSPBE curves for several demons individually. Afterwards, we reprocessed the data using RUPEE$_{scalar}$ to estimate the RMSPBE, yielding qualitatively similar RMSPBE curves for both individual demons and in the aggregate. Using RUPEE$_{scalar}$, we could update 3000 demons in 77.96 ms per step, on average.

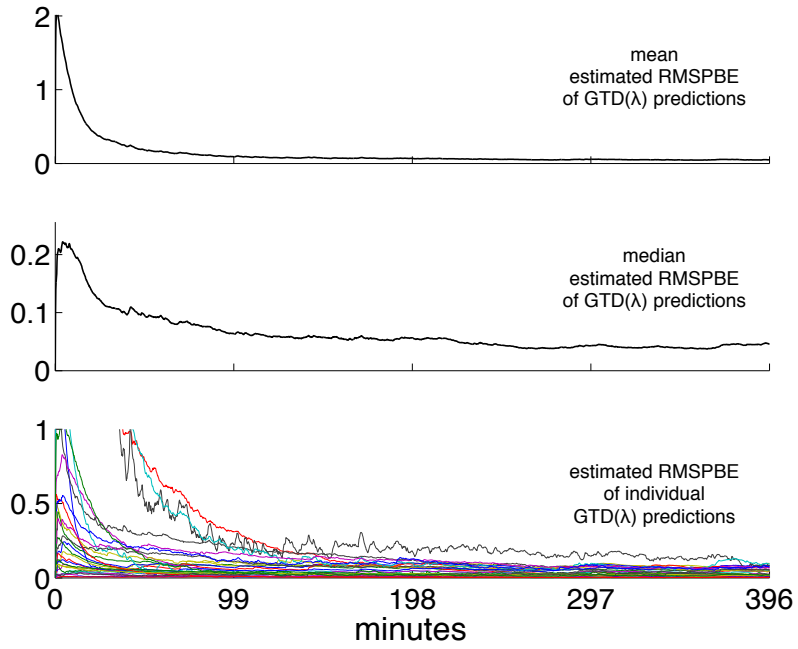

Figure 8.5: Scaling off-policy learning to 2500 demons and 300 distinct target policies on a robot, with progress measured using RUPEE$_{vec}$. The bottom plot shows the estimated RMSPBE versus time for a selection of 30 of the 2500 demons. The top two plots show the mean and median estimated RMSPBE curves computed over all 2500 demons. The median estimated RMSPBE is higher than the previous experiment because the individual cumulants were not scaled to be between zero and one.

The plots in Figure 8.5 show that the estimated RMSPBE, via RUPEE$_{vec}$, in the aggregate and for individual demons reduces significantly during the course of the experiment.

Assuming that the accuracy of each demon's prediction (or RMSE) follows the same profile as the estimate of the RMSPBE—as it did in the previous experiment—then the overall prediction accuracy also reduces significantly during the course of the experiment.

In conclusion, these results suggest that learning predictions about the consequences of many different target policies is feasible in realtime and on a robot. The ability to monitor and track the learning progress of many different demons and target policies is only possible due to the availability of the an online estimate of RMSPBE using RUPEE$_{vec}$.

8.5 Summary

In this chapter we introduced two new methods for estimating off-policy learning progress for gradient temporal difference learning methods, called RUPEE$_{vec}$ and RUPEE$_{scalar}$. We conducted a series of experiments on the Markov chain domain, which aimed to empirically test the practicality of estimating the RMSPBE of the GTD(λ) algorithm in a domain that affords exact calculation of the MSPBE, using our two RUPEE measures. We then moved to the Critterbot to investigate how well our RUPEE measures compared to the prediction accuracy of hundreds of prediction demons. Our final experiment provided a demonstration of learning thousands of prediction demons, with hundreds of distinct target policies, measuring learning progress with RUPEE$_{vec}$, all on a robot.

Overall, RUPEE$_{vec}$ appears to be a reasonable estimator of the RMSPBE, given its light memory and computational footprint compared to computing a least-squares solution (e.g., RMSPBE1). It also appears that the empirical convergence profile of both RUPEE$_{vec}$ and RUPEE$_{scalar}$ are similar to the empirical convergence profile of the prediction accuracy, computed from interspersed test excursions. Scaling the number of predictions about distinct target policies on a robot was made easier using a computationally frugal online performance estimate, such as RUPEE$_{vec}$.

Our new estimators are useful for experimentation and analysis of predictive knowledge architectures like Horde. The accuracy and correctness of the predictive knowledge is essentially private and uninterpretable by humans, and thus an estimate of the MSPBE facilitates monitoring the learning progress of a large collection of demons individually and in the aggregate, without human intervention, testing, or expensive computations.

Chapter 9

Adapting the behavior policy[1]

Up to now, our agent has operated in a passive regime: observing an unending sequence of feature vectors and actions, and predicting future cumulants. In all our experiments so far, the behavior policy has been hand-coded ahead of time by humans, who use imperfect knowledge of the situations the agent might face in the future. Even in the restricted world of the wooden pen, the robot encountered unexpected things, like the strange magnetic readings in the floor. The behavior policy is an answer function, and just like the other answer functions its definition determines the nature of each demon's approximation of its GVF. Improving the answer functions could improve a demon's approximation. An improved behavior policy might be one that adapts action selection during learning, perhaps specializing to each demon's unique requirements, in order to improve prediction accuracy and improve overall system responsiveness.

This chapter investigates how feedback from several demons can be used to adapt the behavior policy of a robot, demonstrating one possible use of our demons. Other systems have used predictions as components of state representations, as in predictive state representations (Littman, Sutton & Singh, 2002; Singh, James & Rudary, 2004; Boots, Siddiqi & Gordon, 2011; Sutton & Tanner, 2005). Cunningham (1972), Becker (1973), Drescher (1990), and Sutton (2009, 2012) explored representing knowledge adapting behavior based on predictions, but only in small scale simulations. The potential uses of Horde's demons are too many and too varied to treat properly in a single chapter. Nevertheless, we propose a simple scheme for behavior adaption and conduct a small-scale experiment on a robot, contributing both the first demonstration of behavior adaption based on demon feedback on a robot, and a concrete demonstration of the potential usefulness of prediction demons. Our

[1] Some of the experiments and text in this chapter are borrowed from a workshop paper (White, Modayil & Sutton, 2014). The text of that paper and the experiments within were almost exclusively produced by this author.

demonstrations constitutes further development of the predictive approach to knowledge.

9.1 Unexpected demon error

One way to adapt the behavior is to listen to the feedback of the demons: triggering behavior change when one or more of the demons encounters an unexpected error. This idea is motivated by situations where something unexpected happens, and situations where the robot encounters a new and previously not encountered situation. In these cases, we might want the robot to investigate the situation. The purpose of the robot's investigation might be to update predictions that have become inaccurate due to a change in the world, or to learn, for the first time, about a new part of the world. After some updating and exploration, the robot might move on and do something else. These changes in the robot's behavior could be triggered by changes in the error of each demon's predictions. In particular, we could track each demon's learning, recording whenever a demon's predictions were unexpectedly inaccurate.

In the previous chapter we developed a measure of learning progress. Here we are interested in a measure of sudden or unexpected change in the demon's regular learning, which is different. Learning progress, as measured by RUPEE, reflects how much learning has occurred and how much learning remains, that is updated on the same time scale as the learning occurs. Unexpected error, on the other hand, is a measure of some unexpected event, which we wish to recognize and react to before the base learning accounts for the change.

We can define a measure of Unexpected Demon Error, or UDE, based on the internal error of each demon's approximate GVF i,

$$\text{UDE}_t^{(i)} = \left| \frac{\overline{\delta^{(i)}}^{\beta}}{\sqrt{\text{var}[\delta^{(i)}] + \epsilon}} \right|, \tag{9.1}$$

where $\bar{\cdot}^{\beta}$ denotes the moving average, $\delta^{(i)} \in \mathbb{R}$ is the TD error of each demon, and $\epsilon \in \mathbb{R}$ in a small constant to avoid division by zero. In all our experiments, we used β_0 equal to 10 times the learning rate of the demon's learning algorithm (α). The variance is incrementally computed with a long run sample average. We take the absolute value of the measure because equal magnitude unexpected errors, either positive or negative, are considered the same. The UDE is independent of cumulant scale, and can aggregate the performance of many demons. Updating the UDE requires $O(1)$ computation and storage per demon.

Consider how the UDE responds to changes if applied to zero mean error generated by

a symmetric unimodal distribution, shown in Figure 9.1. When the error signal becomes unexpectedly large, the moving average becomes larger than the sample standard deviation, and the UDE is large. The error is unexpected because it is much larger than the sample standard deviation. If the fluctuations in the error are expected, again, in comparison to the sample standard deviation then, the UDE is small. A high value of UDE will become small if the TD error, becomes small, or it remains large long enough such that the sample standard deviation becomes large. In either case, the UDE will decrease.

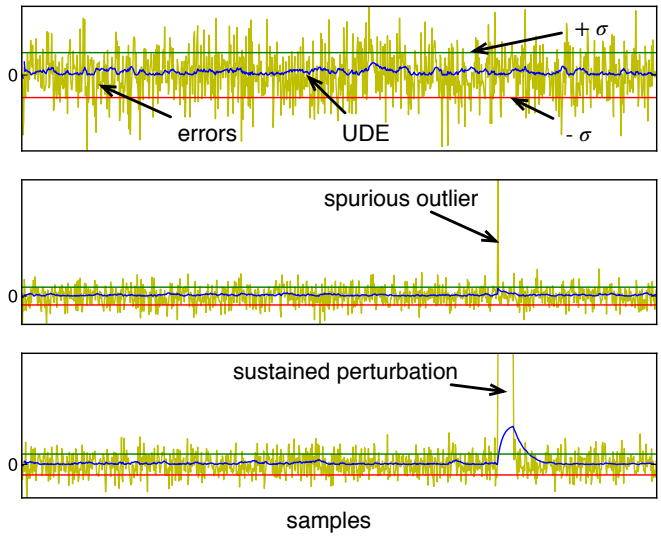

Figure 9.1: A plot of UDE and an error signal sampled from a normal distribution: $\delta_t \sim \mathcal{N}(0, 1.0)$. In the *top panel*, we see the UDE fluctuates close to zero, taking on a value close to zero, compared with the confidence bounds equal to $+\sigma$ and $-\sigma$ plotted in green and red. The fluctuations in the errors are small and so is the UDE. In the *middle panel*, we see the outcome of a single spurious error that is significantly outside the confidence bounds. The *bottom panel* shows a large sustained perturbation of the error signal, causing a large increase in UDE that does not vanish for an extended period of time.

As a second illustration, we computed the UDE of TD(λ) from the log file of the nexting experiment of Chapter 5. In Figure 9.2 we see the UDE of the Light3 prediction. After over an hour of learning (corresponding zero on the x-axis), we see the TD(λ) predictions were well aligned with the Light3 target and consequently the UDE was small, but not zero. After the 80 second mark, the robot stalled in front of the light source, saturating its Light3 sensor. Usually, the light reading would subside as the robot drove past (which can be seen by the small down tick in the robot's light prediction highlighted in the graph). In this case, the UDE shot up and remained high and only began to decay as the the Light3 prediction began

to match the robot's new reality. In the next section, we describe an experiment where the robot is allowed to change its action selection based on unexpected demon error.

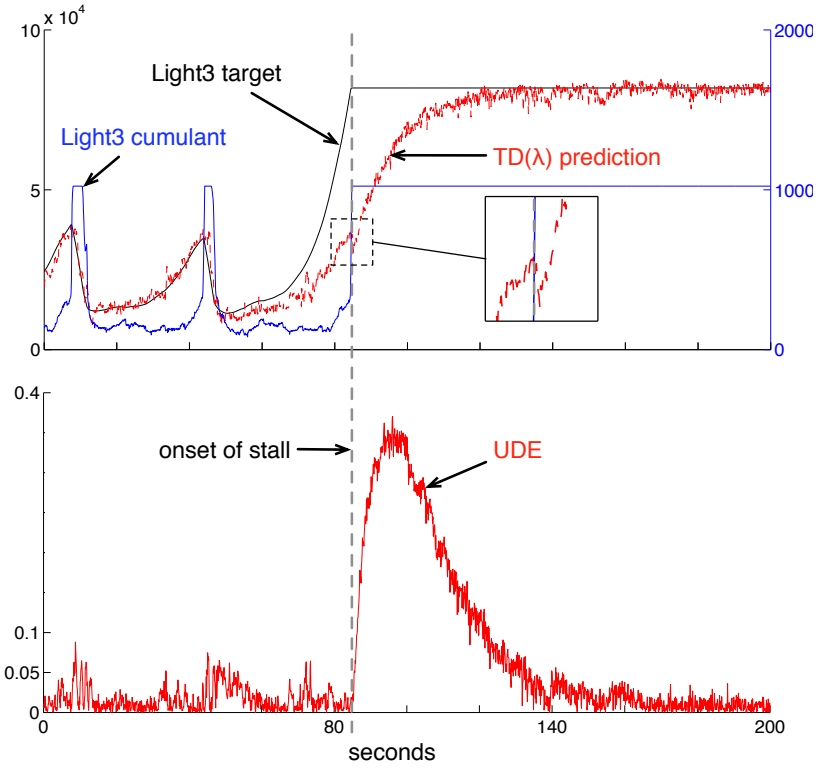

Figure 9.2: The UDE measure computed from the TD error of the TD(λ) algorithm's eight second prediction of future light from the nexting data set. Both graphs share the same x-axis. During the first 80 seconds, the predictions are accurate and the UDE is low. After the 80 second mark, the robot stalls in front of the light source, and the prediction of future light become substantially different from the target: highly inaccurate predictions and the UDE becomes large. The zoomed-in section of the graph shows the TD(λ) initially go down in anticipation of passing the light.

9.2 Adapt the behavior policy of a robot

This section describes and demonstrates one simple way to use UDE to adapt a robot's behavior. We consider the case where each demon's learning has plateaued, and then an event occurs that causes a substantial and sustained increase in the UDE of only one of the demons. We attempt to demonstrate how the behavior can be adapted in response to the increase in UDE, in order to improve the demon's prediction. Specifically, we place a five pound weight on the back of an iRobot Create causing one demon's prediction about future battery current to become inaccurate. The behavior responds by continually rotating until

the demon's prediction about battery current becomes accurate again and the UDE goes down. This following sections describe the experiment in detail.

9.2.1 Problem

The robot's interaction with its environment is structured in a loop with a 30 ms time-step. We use the iRobot Create with a top-mounted, front-facing USB camera and all computing performed onboard via a Raspberry Pi computer. The action set contains four discrete actions, {forward, backward, rotate clockwise, rotate counter-clockwise, stop}, which are mapped to constant wheel-velocity commands on the robot. The 120×160 USB camera images are sampled at 30 frames per second, and the actions are also polled every 30 ms.

The learning task is specified by two different prediction demons, each with different question functions. The first demon's question corresponds to a prediction about the number of time steps until the onset of bumping: "How many time steps until the robot bumps or the effective end of the time horizon is reached, if the robot were to drive forward?". This question is encoded by the question functions: a target policy, $\pi^{(1)}$, that always selects the forward action, $Z^{(1)}$ equal to one, and $\gamma^{(1)}(s)$ equal to zero if either bump sensor equals one, and $\gamma^{(1)}(s)$ equal to 0.9 otherwise (corresponding to three second time scale). We shall refer to this first demon as the forward demon. The second demon's question corresponds to a prediction about the total discounted future battery current draw if the robot rotates in place (similar to a nexting prediction with the target policy equal to constant rotation). This question is encoded by the question functions: a target policy, $\pi^{(2)}$, that continuously rotates counter clockwise, $Z^{(2)} =$ battery-current, and the constant function $\gamma^{(2)}(s)$ equal to 0.8 (corresponding to 1.5 second time scale). We will refer to the second demon as the rotation demon.

Each demon uses a distinct feature vector generation method. The forward demon's feature vector is constructed from the most recent camera image. At the start of the experiment, 100 pixels are selected at random, and from these pixels either the luminance or color channel is selected equally and at random. Figure 9.3 demonstrates the random pixel sampling scheme. Each pixel value (between 0 and 255) is independently binned into 16 non-overlapping bins of width 16, producing a binary feature vector with 16,000 components, of which 100 are non-zero on each time-step. The second demon's feature vector is constructed by discretizing a decaying trace of the battery current reading, scaled to be between zero and one, with bin widths of 1/16th. The rotation demons feature vector contains 16 components, of which only one is active on each time-step.

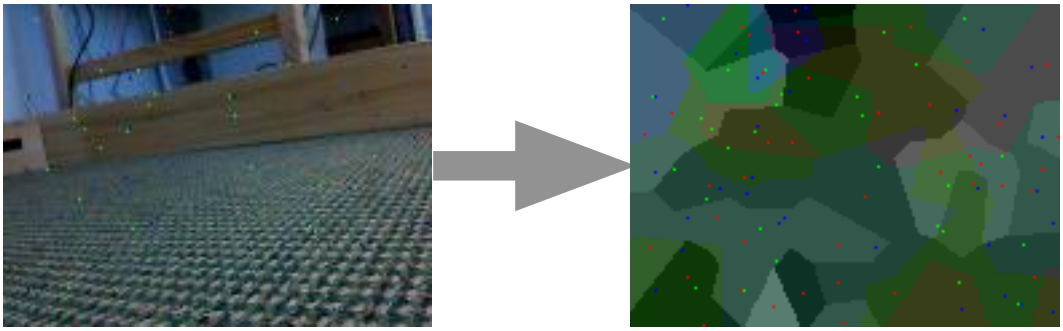

Figure 9.3: A visualization of how the pixels from the USB camera image are randomly sampled and binned to create the forward demon's feature vector. On the left side, we see the 100 randomly sampled color components (red and blue) and luminance components (green) sampled from the current 120 by 160 image. The selection is performed at the beginning of each experiment, and the mapping remains fixed throughout the experiment. The right hand side shows a visualization intended to give the reader an idea of the demon's view of the world using this feature construction method.

The remaining answer functions were shared by both demons. The λ function was a constant function and equal to 0.9. Both demons share a common behavior policy, but the behavior is comprised of two distinct policies. The first policy, called the *non-reactive behavior*, alternates between driving forward until bumping and rotating counter clockwise in free space (not against the wall). The non-reactive behavior performs the same sequence of actions every time the robot bumps into a wall: stop, reversing for five steps, rotating clockwise (probabilistic duration), and then driving forward again until the next bump or and interruption. The forward movement is probabilistically interrupted to initiate an extended and probabilistic duration counter-clockwise rotation. Figure 9.4 shows a snapshot of the non-reactive behavior.

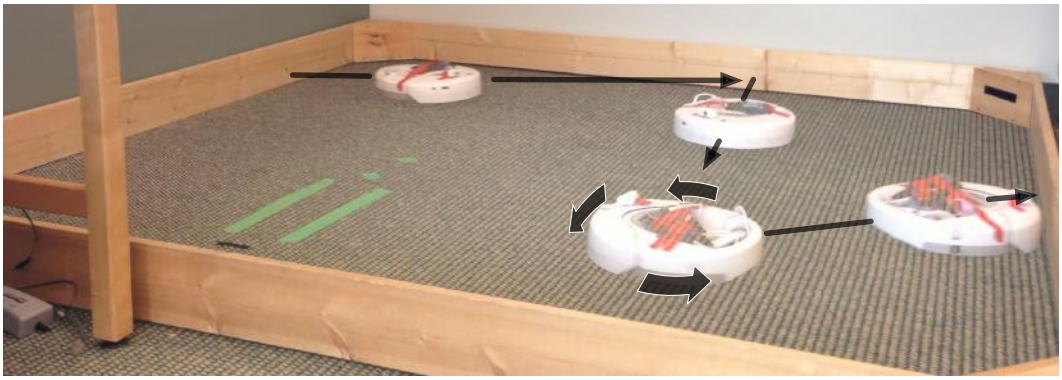

Figure 9.4: The iRobot Create selecting actions according to the non-reactive behavior policy in its pen.

The second policy, called *adaptive behavior*, is rule-based and adapts action selection

based on the UDE of each demon. The adaptive behavior is designed to be very similar to the non-reactive behavior. The only difference is that instead of probabilistically interrupting forward movement with a rotation, the adaptive behavior decides to continue the forward movement or start an extended counter-clockwise rotation based on the UDE of each demon. Algorithm 2 contains the pseudo code for the target policy selection mechanism. Once the adaptive behavior selects one of the demons' target policies, it follows that

Algorithm 2

$j = \text{argmax}_{j \neq i} \text{UDE}_t^{(j)}$
if $\text{UDE}_t^{(j)} < 0.2$ **then**
  pick j randomly
end if
$\mu = \pi^{(j)}$
select actions according to μ for 120 consecutive steps

policy for 120 consecutive steps, unless it is interrupted by a bump. If 120 steps occur and no bump is observed, then the adaptive behavior policy decides whether to continue selecting actions according to $\pi^{(i)}$ for another 120 steps, or switch to different target $\pi^{(j)}$ based on the UDE of each demon. If all demons UDE measures are below the 0.2 threshold, a target policy is selected at random. The adaptive behavior policy enables selecting actions according to the same target policy, for many steps consecutively. When the UDE of each demon is below 0.2, the adaptive behavior will act almost identically to the non-reactive behavior, except the duration of rotations under the adaptive behavior is fixed to 120 steps.[2]

9.2.2 Experiment

All learning, feature processing, and data collection was done on the Raspberry Pi directly connected to an iRobot Create, within the 30 ms update cycle of the robot. Both demons used separate instances of the GTD(λ) algorithm for learning with the same learning-rate parameter values: $\alpha = 0.05$ and $\alpha_{\mathbf{h}} = \alpha/100$. The code for this experiment was based on a publicly available c programming interface for the iRobot create (Mahmood & Sutton, 2013).

Our experiment involved three phases. During the first phase (typically four to six minutes), the robot followed the non-reactive behavior policy. After the UDE of both demons

[2]The threshold of 0.2 was determined via trial and error and worked well for our robot and the particular GVFs specified. The demonstration of this chapter is meant to simply highlight one use of learning multiple GVFs. As such our *adaptive behavior* algorithm is functionally useful for our purposes, but perhaps not as general as it could be. Much more worked in needed to determine a general purpose algorithm for adapting behavior based on parallel GVF learning.

decreased substantially, the behavior policy was switched to the adaptive behavior (Algorithm 2), beginning the second phase. After approximately two minutes, the robot (including learning) was paused and a five pound load was placed in the cargo bay of the Create. Then the robot resumed following the adaptive behavior and learning. The load had a significant effect on the battery current draw, but was not heavy enough to affect the robot's ability to achieve the requested wheel velocity for the forward policy. This experiment was repeated several times, varying the lighting conditions, wall-clock duration of each phase, and camera orientation on the robot. Figures 9.5 and 9.6 show the results of one such run.

9.2.3 Results and conclusions

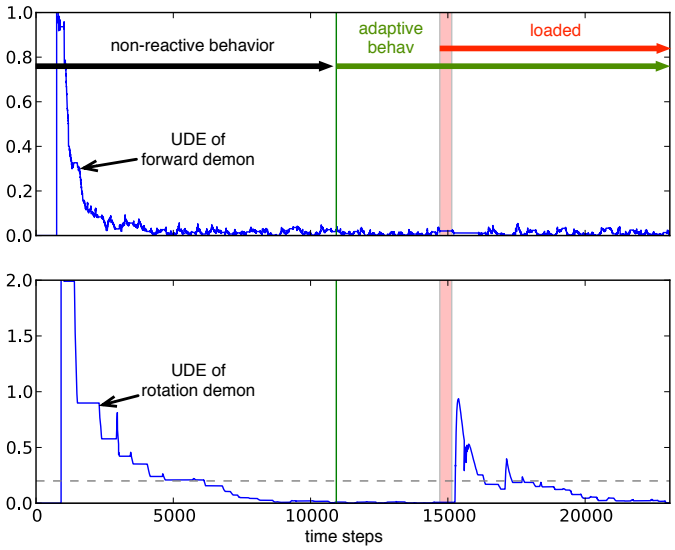

Figure 9.5: A plot of the UDE measure for the forward demon (top) and rotation demon (bottom). Both graphs show the three phases of the experiment: (1) initial learning under the non-reactive behavior, (2) learning under the adaptive behavior, and (3) learning under the adaptive behavior after the load was added to the robot. The dotted line on the bottom plot marks the 0.2 threshold used by the adaptive behavior to switch target policies. The light red vertical shaded region marks when the robot (and learning) were paused to place the load on the robot. A large spike in the UDE measure of the rotation demon, occurs after the load is added in phase three. The UDE reduces quickly thereafter, because the robot continuously rotates providing data to update the demon's prediction of battery current. In this particular run, a second spike in UDE occurs after the initial spike. The robot's initial rotation due to the increase in UDE continued until the UDE was suppressed below the 0.2 threshold. After several seconds, the rotation demon encountered a new situation—according to its feature vector—where its prediction of battery current was still incorrect, and thus the UDE spiked again. This caused another extended, but shorter, rotation.

Figure 9.5 shows the UDE of each demon. There was no noticeable change in the UDE

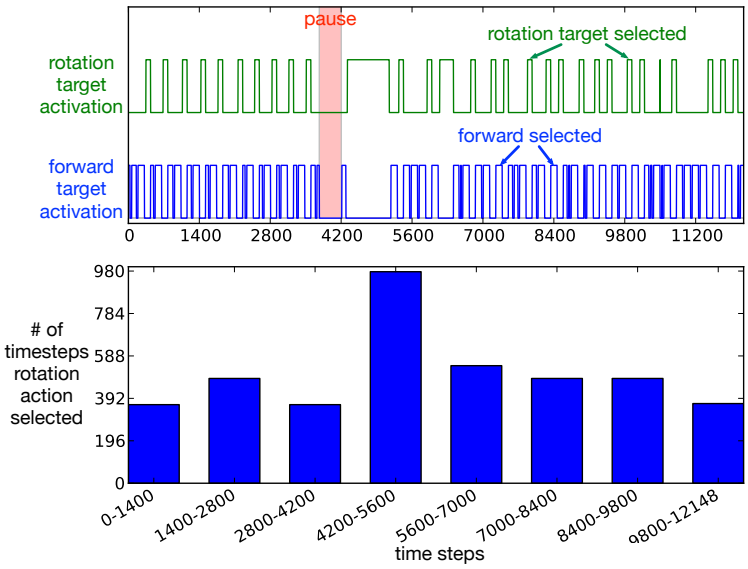

Figure 9.6: These two plots show the action selection choices of the adaptive behavior policy before and after the load was added to the robot (marked pause). The top figure shows which demon's policy was followed by the behavior as a binary time course. At times, neither policy is followed because the robot automatically reverses and counter-clockwise rotates after bumping which is not part of either target policy. The bottom histogram shows the number of time steps on which the rotation action was selected over the same time period. The upper plot shows that the robot continually selected actions in accordance with the rotation demon's target policy after the load was added, and then returned to normal operation. The second extended rotation is also visible. The lower plot illustrates the same phenomenon, and also shows the variability in the number of rotations throughout the run.

of either demon when the behavior policy was changed at the end of phase one. After the load was added to the robot in phase three, the UDE of the forward demon was largely unaffected. The UDE of the rotation demon, on the other hand, shot up quickly after the load was added to the robot, but reduced back down to a low level in comparison to UDE reduction during the beginning of the experiment. The decrease in the rotation demon's UDE under continual rotation (with the load added) was faster than the decrease of UDE observed during initial learning, when rotation occurred sporadically.

The extra load generated a clear and obvious change in the robot's behavior, as shown in Figure 9.6 and observed by the experimenters. During phase two, when neither demon's UDE measure exceeded the threshold, the adaptive behavior only occasionally executed rotations in free space. The first time the adaptive behavior selected a counter-clockwise rotation, after the load was added in phase three, a large increase in UDE was produced

by the rotation demon. The increased UDE caused the adaptive behavior to produce an extended counter-clockwise rotation: a very distinctive change in behavior. Figure 9.6 provides two quantitative measures of how the actions were selected during phase two and phase three. After several seconds of rotation, the UDE of the rotate demon subsided below 0.2, constant rotation ceased, and the behavior returned to normal operation (in comparison to phase two).

This simple experiment demonstrates one way to usefully adapt a robot's behavior based on the feedback of several demons. We have demonstrated that a change in the robot's environment, that causes a substantial change in a single demon's prediction accuracy, can be detected using our UDE measure, and that the increase in UDE can be used to adapt the robot's behavior to help relearn the demon's predictions.

9.3 Discussion

Our simple adaptive behavior corresponds to a limited form of artificial curiosity. In our experiment, the behavior would focus its action selection on one of the target policies if the robot happened to stumble across a situation that caused a increase in UDE—in this case, a human loading the robot. The robot would continue to focus on the new error-generating situation until learning caused the UDE to decrease, eventually permitting the robot to return to normal operation. This behavior is curious and reactive to situations that result in increased demon error, becoming temporarily fixated and then moving on.

Our approach is limited because the behavior policy is not permitted to select a UDE maximizing action on each time-step. The adaptive behavior only periodically queries the UDE measures, and only permits following a single target policy for several consecutive time steps. This means the robot cannot actively search out new situations that yield high UDE, instead relying on happening upon high UDE inducing situations. In addition, the robot cannot discover unique sequences of actions different than those produced by the target policies that might reduce the ODE of many demons simultaneously.

The ideas and demonstrations in this chapter are related to and inspired by the rich history of work in intrinsically motivated reinforcement learning (IMRL). Many IMRL systems make use of an additional intrinsic reward function computed from things internal to the agent, such as value function change (Simsek & Barto, 2006; Schembri et al., 2007), model error (Schmidhuber, 1991; Lopes et al., 2012), and prediction error (Singh et al., 2005; Oudeyer et al., 2007). These internal rewards can be used to encourage the agent to

select actions it would not normally select under the external reward, in order to facilitate learning more generally applicable policies or increase the speed of learning on tasks encountered in the future. This approach views internal motivations and curiosity as a tool to help improve the agent performance. Internal rewards can also be used to simulate artificial curiosity, where the agent takes actions to generate new experiences solely for the sake of exploring (e.g., Oudeyer et al., 2007). Finally, a recent and significant departure from previous IMRL work takes the view that all reward functions are both internal and external, and these functions are produced by an evolutionary search (see Singh et al., 2010). Table 1 in Appendix C summaries several studies of artificial curiosity and IMRL, focusing in particular on each system's definition of internal and external reward. In addition, Baldassarre & Mirolli (2013) provide an excellent survey of work in intrinsic motivation in reinforcement learning and psychology.

There are similarities between previous work on IMRL and our behavior adaption demonstration on the Create. Several studies demonstrated the utility of IMRL for multiple value functions and policies learning (Singh et al., 2005; Simsek & Barto, 2006; Oudeyer et al., 2007; Schembri et al., 2007; Singh et al., 2010), like our parallel demon learning, and several others also used either prediction error or TD error as a measure of learning (Schmidhuber, 1991; Singh et al., 2005; Schembri et al., 2007), including the one work that included demonstrations on a real robot (Oudeyer et al., 2007). The learning system of Singh et al. (2005) is most related to ours: they also used parallel off-policy reinforcement learning methods (Q-learning) to learn value functions and defined internal rewards equal to prediction errors. We could improve our own system by using a reinforcement learning algorithm to learn a behavior policy that maximizes the sum UDE over all the demons—an intrinsic reward. Such a behavior might seek out situations where the demons knowledge was incorrect, similar to Oudeyer's robot (2007). Our demonstration is the first demonstration combining off-policy reinforcement learning, function approximation and behavior adaption on a robot.

9.4 Summary

In this chapter we investigated how feedback from several demons could be used to adapt the behavior policy of a robot. First we introduced a measure of unexpected demon error, which captures when a demon's predictions become unaligned with recent experience. Next we described a simple behavior policy, which periodically selects actions according to the

demon with highest UDE. We then demonstrated the usefulness of our approach on the iRobot create, showing the robot could effectively adapt its behavior due to a dramatic increase in UDE, when a load was placed on the top of the robot. The chapter closed with a discussion of prior work related to UDE and adapting behavior based on demon feedback.

This experiment provides a demonstration of how feedback from a Horde of demons can be used to improve learning. In addition, our demonstration also constitutes the first demonstration of predictive knowledge learning and online behavior adaption on a robot. The work described in this chapter makes a small addition to the development of predictive knowledge learning. There remain many interesting questions regarding how to use predictions and other feedback from demons to adapt the behavior policy. These items are left to future work.

Chapter 10

Perspectives and Future Work

This thesis investigated the problem of representing an agent's knowledge of the world with predictions. The aim was to develop the predictive approach to knowledge, improving both applicability and scalability of learning with a novel application of value functions and learning algorithms from reinforcement learning. Towards this goal, we claim progress has been demonstrated in three areas. First, GVFs have been proposed as a new representation for predictive knowledge that is expressive and potentially learnable with computationally efficient value function learning algorithms. GVF learning has been demonstrated at large scale from long, high-dimensional, continuous data streams with several experiments on mobile robots. Second, a new measure called RUPEE has been presented for estimating off-policy learning progress in situations where measuring prediction accuracy is practically challenging, such as learning about many distinct target policies on a robot. Third, several issues have been exposed and investigated regarding the empirical performance of the GTD(λ) algorithm, including the role of the secondary weight vector and the algorithm's parameter sensitivity in on-policy and off-policy learning settings. We begin by summarizing the three proposed contributions of our work, and close with future work.

10.1 General Value Functions

The application of GVFs to represent predictive knowledge facilitates posing a large diversity of predictive questions. Through various combinations of target policies, cumulants, and termination signals, we may pose a variety of predictive questions. In this thesis, we investigated several such questions including, questions about what will happen next, such as the future value of the magnetic sensor in the next 100 ms (see Chapter 5), policy-contingent questions, such as the time to stop on different surfaces (see Chapter 6), and questions about complex termination conditions, such as predicting light after rounding the

next corner (see Chapter 6). Our approach is promising because GVFs make it possible to capture policy-contingent declarative and goal-oriented predictive knowledge, and a large amount of knowledge is of this form. Conventional knowledge representations that can capture this kind of knowledge (e.g., high-level, symbolic methods such as rules, operators, and production systems) are not as grounded and therefore not as learnable as GVFs.

Although value functions have always been potentially learnable, only recently have scalable learning methods become available that make it practical to explore the idea of GVFs with off-policy learning and function approximation. With small modifications for parallel learning, temporal difference learning algorithms were used to make and update predictions online, while the agent interacted with the world, from continuous inputs and off-policy samples. In comparison, prior prediction learning systems are either not well suited for continuous domains, or are difficult to scale to large numbers of predictions and long data streams. We have demonstrated with our nexting (Chapter 5) and policy-contingent prediction experiments on the Critterbot (Chapter 8) that GVF learning can be scaled to learning thousands of predictions from hours of data, well beyond the demonstrations of previous work.

10.2 Online off-policy progress estimation

One special challenge for large-scale off-policy learning on a robot, involves evaluating learning progress of each individual prediction. Many approaches to progress estimation use knowledge of the MDP parameters, limit the span of individual predictions, or limit the number of distinct target policies that can be used to specify predictions. In Chapter 8, we proposed an alternate approach based on estimating the MSPBE of each demon. Exploiting the fact that the secondary weights of the GTD(λ) algorithm estimate a portion of the MSPBE, we developed two measures, called RUPEE$_{vec}$ and RUPEE$_{scalar}$, that combine GTD(λ)'s partial estimate of the MSPBE with a trace of recent errors. The resultant measures can be updated incrementally with linear computation per demon, refining their estimate of the MSPBE with each sample, and are both naturally parallelizable. These RUPEE measures do not require special testing phases, and thus may simplify the design and implementation of the behavior policy.

Our RUPEE measures performed well in both a Markov chain domain and on the Critterbot. One of our RUPEE measures provided a reasonable estimate of the MSPBE in several variants of the Markov chain domain. In addition, both RUPEE measures seemed

to provide a surrogate measure of aggregate prediction accuracy, measured by test excursions, for hundreds of policy-contingent predictions learned on the Critterbot. It was also empirically demonstrated that our measures can react quickly to changes, in both the chain domain and robot prediction tasks. Finally, RUPEE was essential for our demonstration of policy-contingent prediction learning on the Critterbot, providing an estimate of learning progress for thousands of demons whose GVFs were contingent on hundreds of distinct target policies. Such a demonstration would not have been possible without an online measure of learning progress.

10.3 Empirical study of GTD(λ)

Gradient-TD methods make exploring parallel demon learning practical for the first time, but our empirical understanding of these new algorithms is limited. This thesis provides new insights for one such algorithm, GTD(λ). The Markov chain experiments of Chapter 7 investigate a distinctive aspect of the GTD(λ) algorithm, the secondary weight vector and gradient correction. Our experiments demonstrate that when GTD(λ) is tuned to optimize MSPBE, the algorithm may not accurately learn the secondary weights. On the other hand, when we optimized the learning-rate parameters of GTD(λ) to learn the secondary weights, GTD(λ) more effectively learned these weights, and did not exhibit divergence. We also demonstrated that GTD(λ) can learn efficiently without the secondary weights—even on off-policy problem instances—but the performance of GTD(λ) with the secondary weights, learned in the usual way, was competitive.

The gradient-TD methods are intended to prevent divergence under off-policy sampling; to continue our study we moved to a variant of Baird's (1995) counterexample that causes divergence of linear off-policy TD. The results in this domain were different from those in the chain domain: the secondary weights appear to be important for avoiding divergence, and the default behavior of the algorithm (minimizing MSPBE) cause the secondary weights to be learned well. In addition, our counterexample experiments revealed that large values of λ can lead to instability in the GTD(λ) due to successive large importance sampling corrections multiplying in the trace update. Our experiments in these two problems illustrates that good performance for GTD(λ) can be produced with very different parameter settings. Our results highlight limitations of the GTD(λ) algorithm, and suggest a starting point for future algorithmic development.

10.4 Limitations and future work

The investigation of GVFs as a representation for predictive knowledge begun here is far from complete. We have opened or continued study on several topics, and much remains to be done, including: (1) comparing the representation capabilities of GVFs with that of PSRs, TPSRs, TD-nets, and prediction profile models, (2) empirically exploring using predictive-state features with Horde, (3) a theoretical characterization of RUPEE, (4) developing algorithms for mitigating variance in off-policy learning attributed to large importance sampling corrections, and (5) combining on- and off-policy demon learning, control demon learning, and behavior learning in a much larger experiment.

10.4.1 Comparing predictive representations of knowledge

Our discussion of related approaches in Chapter 4 was limited to high-level comparisons with other representations of predictive knowledge. Our focus in this thesis was to ensure our knowledge representation based on GVFs achieved certain desirable attributes present, in part, in other predictive representations, including compatibility with predictive feature components, compositional prediction, and policy contingent prediction. A possible direction for future work involves more direct and specific comparisons among GVFs and PSRs, TPSRs, TD-nets, and prediction profile models. This could be done by characterizing what types of predictions can be encoded and learned with each representation. It may also be fruitful to analyze and relate the compactness of GVFs with other representations, similar to how PSs have been compared with POMDPS models (Littman, Sutton & Singh, 2002; Singh et al., 2004). Finally, the usefulness of predictive knowledge encoded as GVFs could be evaluated against other representations, for example examining their usefulness as features for policy learning, similar to the work of Rafols (2005) and Schaul and Ring (2013).

10.4.2 Predictive features and a Horde of demons

In Chapter 6, we provided a small demonstration of how predictions can participate in feature generation, similar to the idea of predictive state representations. Our demonstration was limited in that only a single prediction was used as part of the feature vector, and that prediction was used only to improve its own learning. Although an interesting special case, our demonstration was far from the other extreme where all the feature components are computed from predictions, exemplified by PSRs. Horde enables part of the feature vector to be predictive and part to be non-predictive. An initial subset of the predictions could

be learned using only the non-predictive components of the feature vector, and then later, once this subset of the predictions has become accurate, other more complex predictions could be learned with the greater representational capacity afforded by the predictive feature components. Future work could investigate expanding the representational power of the feature vector through predictive features, with experiments in several domains with varying degrees of partial state information.

10.4.3 A theoretical analysis of RUPEE

In Chapter 8, we proposed two ways to estimate the MSPBE. Our analysis of these new estimators was limited to empirical studies in simulation domains and a robot, where these new measures appeared to suit our purposes. In addition to empirical success, the bias, consistency, and variance of these estimators is also of interest. Both our RUPEE measures used the secondary weight vector that is updated by an LMS rule inside the GTD(λ) algorithm. Much of the LMS literature is concerned with the statistical properties of the value estimate $\mathbf{x}^\top \mathbf{h}$, compared to the estimate given the optimal weight vector $\mathbf{x}^\top \mathbf{h}^\star$, whereas in RUPEE the quantity of interest is the vector $\mathbf{h}$ itself. A theoretical analysis of RUPEE could potentially yield improvements in how we can estimate the MSPBE, and insights into how the GTD(λ) algorithm could be improved.

10.4.4 Mitigating variance in off-policy learning

The experiments of Chapter 7 revealed a previously undocumented stability problem with the GTD(λ) algorithm on Baird's counterexample. At first it appeared we had come across an example of divergence for an algorithm we believed to be convergent, given that a large sweep of the learning rate parameters did not yield a non-divergent learning curve. Independently, Sutton et al. (2015) encountered similar behavior with a different TD based algorithm. Our empirical study of this issue was limited to a single counterexample, and it remains unclear how pervasive such variance is in prediction tasks. Nevertheless, this behavior is concerning because we would like to ensure, if at all possible, that no particular predictive question causes a demon's predictions to become unstable. A possible direction for future work involves developing new learning algorithms that are robust to this setting. One idea is to set λ as a function of state, allowing it to change with time, perhaps reducing λ to zero when the magnitude of the trace becomes large. The λ parameter is part of the definition of the MSPBE, and thus making an objective function to enable optimization of λ is not at all clear at this time. Another option would be to use some form of

weighted importance sampling variant of GTD(λ), that should reduce variance. The details of weighted importance sampling have been recently been worked out for off-policy LSTD (Mahmood et al., 2014), but extensions to linear complexity learning methods remains an open problem.

10.4.5 Putting it all together

Throughout this thesis we demonstrated several components of the Horde architecture, including large-scale nexting predictions in Chapter 5, large-scale policy-contingent prediction in Chapter 8, predictive state components in Chapter 6, and behavior adaption based on demon learning in Chapter 9.1. Our experiments were limited to fixed linear function approximation architectures, and only learning from a few hours of data. A potentially fruitful trajectory for future work involves running experiments with more data and more adaption—including the learning the behavior policy and automatically generating new features—and even more demons. These experiments could provide insight into how many demons could be either exhaustively generated, or even adaptively generated (i.e., autonomous goal generation). Such experiments may require more advanced parallel implementations of Horde, such as on GPUs and new robot platforms. Appendix D contains one proposal for a scaling experiment.

10.5 Summary

This chapter closed the thesis with concluding thoughts, a discussion of the limitations of our work, and possible directions for future work. We concluded with a recap of the three areas in which progress can be claimed, owing to the work summarized in this document. We closed the thesis with a discussion of topics for future work, including comparing different representations of predictive knowledge, investigating predictive feature components, analysis of RUPEE, mitigating variance in off-policy learning, and further scaling up and integrating prediction learning.

Bibliography

Antos, A., Szepesvari, C., & Munos, R. (2008). Learning near-optimal policies with Bellman-residual minimization based fitted policy iteration and a single sample path. *Machine Learning*, 71(1), 89–129.

Abeyruwan, S., Seekircher, A., & Visser, U. (2014). Off-policy general value functions to represent dynamic role assignments in RoboCup 3D soccer simulation. arXiv preprint arXiv:1402.4525.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, 30–37.

Baldassarre, G., & Mirolli, M. (Eds.). (2013). *Intrinsically Motivated Learning in Natural and Artificial Systems*. Berlin, Springer.

Becker, J. D. (1973). A model for the encoding of experiential information. In R. C. Shank and K. M. Colby (Eds.), *Computer models of thought and language* (pp. 396–434). San Francisco, W. H. Freeman and Company.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

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

Boots, B., Siddiqi, S. M. & Gordon, G. J. (2011). Closing the learning-planning loop with predictive state representations. *International Journal of Robotics Research*, 30(7), 954–966.

Boots, B., & Gordon, G. J. (2011). An online spectral learning algorithm for partially observable nonlinear dynamical systems. In *Proceedings of the 25th National Conference on Artificial Intelligence*.

Bontempi G. (1999). *Local Learning Techniques for Modelling, Prediction and Control* (Doctoral dissertation). Universit Libre de Bruxelles, Belgium.

Bontempi, G., & Ben Taieb, S. (2011). Conditionally dependent strategies for multiple-step-ahead prediction in local learning. In *International Journal of Forecasting*, 27(3), 689–699.

Box, G. E., Jenkins, G. M., & Reinsel, G. C. (2011). *Time Series Analysis: Forecasting and Control*. Wiley.

Boyan, J. A. (1999). Least-squares temporal difference learning. *International Conference on Machine Learning*, 49–56.

Boyan, J. A. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2-3), 233–246.

Bradtke, S. J., & Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1-3), 33-57.

Brogden, W. (1939). Sensory pre-conditioning. *Journal of Experimental Psychology*, 25(4),323–332.

Butz, M., Sigaud, & O., Gérard, P. (Eds.). (2003). *Anticipatory Behaviour in Adaptive Learning Systems: Foundations, Theories, and Systems*, LNAI 2684, Springer.

Camacho, E. F., & Bordons, C. (2004). *Model Predictive Control*. Springer.

Carlsson, K., Petrovic, P., Skare, S., Petersson, K., & Ingvar, M. (2000). Tickling expectations: Neural processing in anticipation of a sensory stimulus. *Journal of Cognitive Neuroscience*, 12(4), 691–703.

Cheng, H., Tan, P., Gao, J., & Scripps, J. (2006) Multistep-ahead time series prediction. *Lecture Notes in Computer Science*, Springer, 765–774.

Chevillon, G., & Hendry, D. F. (2005). Non-parametric direct multi-step estimation for forecasting economic processes. *International Journal of Forecasting*, 21(2), 201–218.

Clark, A. (2012). Whatever next? Predictive brains, situated agents, and the future of cognitive science. *Behavioral and Brain Sciences*, 36(3), 181–204.

Crites, R. H., & Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3), 235–262.

Cunningham, M. (1972). *Intelligence: Its Organization and Development*, New York, Academic Press.

Dann, C., Neumann, G., & Peters, J. (2014). Policy evaluation with temporal differences: A survey and comparison. *Journal of Machine Learning Research*, 15, 809–883.

Degris, T., Pilarski, P. M., & Sutton, R. S. (2012). Model-free reinforcement learning with continuous action in practice. In *Proceedings of the American Control Conference*, 2177–2182.

Degris, T., White, M., & Sutton, R. S. (2012). Off-policy Actor-Critic. *International Conference on Machine Learning*.

Delp, M. (2011). *Experiments in Off-Policy Reinforcement Learning with the GQ(λ)* (Masters thesis), University of Alberta.

Drescher, G. L. (1991). *Made-up Minds: A Constructivist Approach to Artificial Intelligence*. MIT Press.

D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer & C. Welty. (2010) Building Watson: An overview of the DeepQA project. *AI Magazine*, 31(3), 59–79.

Edwards, A. L., Dawson, M. R., Hebert, J. S., Sutton, R. S., Chan, K. M., & Pilarski, P. M. (2014). Adaptive switching in practice: Improving myoelectric prosthesis performance through reinforcement learning. In *Proceedings of the Myoelectric Controls Symposium*, 69–73.

Geist, M., & Scherrer, B. (2014). Off-policy learning with eligibility traces: A survey. *Journal of Machine Learning Research*, 15(1), 289–333.

Geramifard, A., Bowling, M., & Sutton, R. S. (2006). Incremental least-squares temporal difference learning. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 356–361.

Gilbert, D. (2006). *Stumbling on Happiness*. Knopf Press.

Grush, R. (2004). The emulation theory of representation: Motor control, imagery, and perception. *Behavioural and Brain Sciences*, 27, 377–442.

Gunther, J., Pilarski, P. M., Helfrich, G., Shen, H., & Diepold, K. (2014). First steps towards an intelligent laser welding architecture using deep neural networks and reinforcement

learning. *Procedia Technology*, 15, 474–483.

Hackman, L. (2012). *Faster Gradient-TD Algorithms* (Masters thesis). University of Alberta.

Hawkins, J., & Blakeslee, S. (2004). *On Intelligence*. Times Books.

Huron, D. (2006). *Sweet Anticipation: Music and the Psychology of Expectation*. MIT Press.

Itti, L., & Baldi, P. F. (2005). Bayesian surprise attracts human attention. *Advances in Neural Information Processing Systems*, 547–554.

Jordan, M. I., & Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16(3), 307–354.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Levitin, D. (2006). *This is Your Brain on Music*. Dutton Books.

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

Ljung, L. (1998). *System Identification: Theory for the User*. Prentice Hall.

Lopes, M., Lang, T., Toussaint, M., & Oudeyer, P. Y. (2012). Exploration in model-based reinforcement learning by empirically estimating learning progress. *Advances in Neural Information Processing Systems*, 206–214.

Maei, H. R., Szepesvari, C., Bhatnagar, S., Precup, D., Silver, D., & Sutton, R. S. (2010). Convergent temporal-difference learning with arbitrary smooth function approximation. *Advances in Neural Information Processing Systems*.

Maei, H., Sutton, & R. S. (2010). GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, 91–96.

Maei, H. R., Szepesvri, C., Bhatnagar, S., & Sutton, R. S. (2010b). Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on Machine Learning*, 719–726.

Maei, H. R. (2011). *Gradient Temporal-Difference Learning Algorithms* (Doctoral dissertation). University of Alberta.

Mahmood, A. R., & Sutton, R. S. (2013). Create Serial Port Packet Processor. [Software]. Available from https://github.com/armahmood/Create-Serial-Port-Packet-Processor

Mahmood, A. R., van Hasselt, H & Sutton, R. S. (2014). Weighted importance sampling for off-policy learning with linear function approximation. *Advances in Neural Information Processing Systems 27*.

Markoff, J. (2010, October 9). Google cars drive themselves, in traffic. *The New York Times*.

McCracken, P., & Bowling, M. (2005). Online discovery and learning of predictive state representations. *Advances in Neural Information Processing Systems*, 875-882.

Modayil, J., White, A., & Sutton, R. S. (2012a) Multi-timescale nexting in a reinforcement learning robot. In T. Ziemke, C. Balkenius, J. Hallam (Eds.), *From Animals to Animats: 12th International Conference on Simulation of Adaptive Behavior, SAB 2012* (pp. 299–309), LNAI 7426, Heidelberg, Springer.

Modayil, J., White, A., Pilarski, P. M., & Sutton, R. S. (2012b). Acquiring a broad range of empirical knowledge in real time by temporal-difference learning. *2012 IEEE International Conference on Systems, Man, and Cybernetics*, 1903-1910.

Modayil, J., White, A., Pilarski, P. M., & Sutton, R. S. (2012c). Acquiring diverse predictive knowledge in real time by temporal-difference learning. *International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*.

Modayil, J., White, A., & Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, 22(2), 146–160.

Mozer, M. C. (1993). Neural network architectures for temporal pattern processing. In A. S. Weigend & N. A. Gershenfeld (Eds.), *Time series prediction: Forecasting the future and understanding the past* (pp. 243–264). Redwood City, CA: Sante Fe Institute Studies in the Sciences of Complexity, Proceedings Volume XVII, Addison-Wesley Publishing.

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

Pacheco, P. S. (1997). *Parallel Programming with MPI*. Morgan Kaufmann.

Parlos, A. G., Rais, O. T., & Atiya, A. F. (2000). Multi-step-ahead prediction using dynamic

recurrent neural networks. *Neural networks*, 13(7), 765–786.

Pavlov, I. (1927). *Conditioned Reflexes: An Investigations of the Physiological Activity of the Cerebral Cortex*. G. V. Anrep (Ed. & Trans.), Oxford University Press.

Peters, J., & Schaal, S. (2008). Natural actor-critic. *Neurocomputing*, 71(7), 1180–1190.

Pezzulo, G. (2008). Coordinating with the future: the anticipatory nature of representation. *Minds and Machines*, 18(2), 179–225.

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

Pilarski, P.M., Dawson, M.R., Degris, T.,Carey, J.P., & Sutton, R.S. (2012). Dynamic switching and real-time machine learning for improved human control of assistive biomedical robots. In *Proceedings of the 4th IEEE International Conference on Biomedical Robotics and Biomechatronics*, 296–302.

Precup, D., Sutton, R. S., & Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. *In Proceedings of the International Conference on Machine Learning*, 417–424.

Precup, D., Sutton, R. S., Paduraru, C., Koop, A., & Singh, S. (2005). Off-policy learning with options and recognizers. *Advances in Neural Information Processing Systems*, 1097–1104.

Rafols, E. J., Ring, M. B., Sutton, R. S. & Tanner, B. (2005) Using predictive representations to improve generalization in reinforcement learning. In *Proceedings of the 2005 International Joint Conference on Artificial Intelligence*, 835–840.

Rafols, E. (2006). *Temporal Abstraction in Temporal-difference Networks* (Masters thesis). University of Alberta.

Rescorla, R. (1980). Simultaneous and successive associations in sensory preconditioning. *Journal of Experimental Psychology: Animal Behavior Processes*, 6(3), 207–216.

Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning* (Doctoral dissertation). Cambridge University.

Schaul, T., & Ring, M. (2013). Better generalization with forecasts. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 1656–1662.

Schembri, M., Mirolli, M., & Baldassarre, G. (2007). Evolving internal reinforcers for an

intrinsically motivated reinforcement-learning robot. *Development and Learning*, 282–287.

Scherrer, B. (2010). Should one compute the temporal difference fix point or minimize the Bellman residual? The unified oblique projection view. In *Proceedings of The International Conference on Machine Learning*.

Scherrer, B. & Geist, M. (2011). Recursive least-squares learning with eligibility traces. *European Workshop on Reinforcement Learning*, 11.

Schmidhuber J. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proceedings of the 1st International Conference on Simulation of Adaptive Behavior*, 222–227.

Silver, D., Sutton, R. S., & Muller, M. (2007). Reinforcement Learning of Local Shape in the Game of Go. *The Joint Conferences on Artificial Intelligence*, 7, 1053–1058.

Simsek, O., & Barto, A. G. (2006). An intrinsic reward mechanism for efficient exploration. In *Proceedings of the 23rd international conference on Machine learning*, 833–840.

Singh, S., James, M. R., & Rudary, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. In: *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, 512–519.

Singh S., Barto, A. G., & Chentanez, N. (2005). Intrinsically motivated reinforcement learning. *Advances in Neural Information Processing Systems*, 17, 1281–1288.

Singh, S., Lewis, R. L., Barto, A. G., & Sorg, J. (2010). Intrinsically motivated reinforcement learning: an evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2), 70–82.

Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3), 165–188.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3, 9–44.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: *Proceedings of the Seventh International Conference on Machine Learning*, 216–224.

Sutton, R. S. (1995). TD models: modeling the world at a mixture of time scales. In:

Proceedings of the International Conference on Machine Learning, 531–539.

Sutton, R. S. (2009). The grand challenge of predictive empirical abstract knowledge. In: *Working Notes of the IJCAI-09 Workshop on Grand Challenges for Reasoning from Experiences.*

Sutton, R. S. (2012). Beyond reward: The problem of knowledge and data. In S. H. Muggleton, A. Tamaddoni-Nezhad, F. A. Lisi (Eds.), *Proceedings of the 21st International Conference on Inductive Logic Programming* (pp. 2–6), LNAI 7207, Springer, Heidelberg.

Sutton, R. S., & Barto, A. G. (1990). Time-derivative models of Pavlovian reinforcement. *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, 497–537. MIT Press.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Sutton, R. S., Koop, A. & Silver, D. (2007). On the role of tracking in stationary environments. In *Proceedings of the International Conference on Machine Learning*.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvri, C., & Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 993–1000.

Sutton, R. S., Mahmood, A. R., Precup, D., & van Hasselt, H. (2014). A new Q(λ) with interim forward view and Monte Carlo equivalence. In *The Proceedings of the 31st International Conference on Machine Learning*.

Sutton, R. S., Mahmood, A. R., & White, M. (2015). An emphatic approach to the problem of off-policy temporal-difference learning. *arXiv preprint arXiv:1503.04269*.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., & Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In: *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems*, 761–768.

Sutton, R. S., & Precup, D., Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112, 181–211.

Sutton, R. S., Rafols, E. J., & Koop, A. (2006). Temporal abstraction in temporal-difference networks. *Advances in Neural Information Processing Systems*, 18, MIT Press.

Sutton, R. S., & Tanner, B. (2005). Temporal-difference networks. *Advances in Neural Information Processing Systems* 17, 1377–1384.

Sutton, R. S., Szepesvári, Cs., & Maei, H. R. (2009). A convergent $O(n)$ algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in Neural Information Processing Systems*, 21. MIT Press.

Szepesvari, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1), 1–103.

Talvitie, E., & Singh, S. (2011). Learning to make predictions in partially observable environments without a generative model. *Journal of Artificial Intelligence Research*, vol. 42(1), pp. 353–392.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. In *Communications of the ACM*, 38(3), 58–68.

Tedrake, R., Zhang, T., & Seung, H. (2005). Stochastic policy gradient reinforcement learning on a simple 3D biped. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3, 2849–2854.

Thrun, S., Burgard, W., & Fox, D. (2005). Probabilistic robotics. *MIT press*.

Thrun, S., Montemerlo, M., et al. (2006). Stanley: The robot that won the DARPA grand challenge. *Journal of Field Robotics*, 23(9), 661–692.

Tolman, E. C. (1951). *Purposive Behavior in Animals and Men*. University of California Press.

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

van Hasselt, H., Mahmood, A. R., & Sutton, R. S. (2014). Off-policy TD(λ) with a true online equivalence. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence*.

van Overschee, P., & De Moor, B. (1996). Subspace identification for linear systems: theory, implementation, applications. *Springer*.

van Seijen, H., van Hasselt, H., Whiteson, S., & Wiering, M. (2009). A theoretical and empirical analysis of Expected Sarsa. *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 177–184.

van Seijen, H., & Sutton, R. S. (2014). True online TD(λ). In *Proceedings of the 31st International Conference on Machine Learning*.

Wang, C. C., & Thorpe, C., Thrun, S. (2003). Online simultaneous localization and mapping with detection and tracking of moving objects: theory and results from a ground vehicle in crowded urban areas. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 842–849.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation). Cambridge University.

Welch, G., & Bishop, G. (1995). *An Introduction to the Kalman filter* (Technical Report 95-041). University of North Carolina at Chapel Hill, Computer Science Department.

White, R. W. (1959). Motivation reconsidered: The concept of competence. In *Psychological review*, 66(5).

White, A., Modayil, J., & Sutton, R. S. (2012). Scaling life-long off-policy learning. In *Proceedings of the Second Joint IEEE International Conference on Development and Learning and on Epigenetic Robotics*.

White, A., Modayil, J., & Sutton, R. S. (2014). Surprise and curiosity for big data robotics. In: *The Workshop Proceedings of The Twenty-eighth AAAI Conference on Artificial Intelligence: Sequential Decision Making and Big Data*.

White, A., White, M., & Sutton, R. S. (in prep). The off-policy advantage. Manuscript in preparation.

Widrow, B., & Stearns, S. D. (1985). *Adaptive Signal Processing*. Englewood Cliffs, NJ, Prentice-Hall, Inc.

Williams, R. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), 229–256.

Wingate, D. (2008). *Exponential Family Predictive Representations of State* (Doctoral dissertation). University of Michigan.

Wingate, D., & Singh, S. (2008). Efficiently learning linear-linear exponential family predictive representations of state. In *Proceedings of the 25th International Conference on Machine learning*, 1176–1183.

Wolfe, B., James, M. R., & Singh, S. (2005). Learning predictive state representations in dynamical systems without reset. In *Proceedings of the 22nd International Conference on Machine learning*, 980–987.

Wolpert, D., Ghahramani, Z., & Jordan, M. (1995). An internal model for sensori-motor integration. *Science*, 269(5232), 1880–1882.

Wong, W. K., Xia, M., & Chu, W. C. (2010). Adaptive neural network model for time-series forecasting. *European Journal of Operational Research*, 207(2), 807–816.

Xu, X., He, H. G., & Hu, D. (2002). Efficient reinforcement learning using recursive least-squares methods. *Journal of Artificial Intelligence Research*, 16(1), 259-292.

Yu, C., & Ballard, D. (2004). A multimodal learning interface for grounding spoken language in sensory perceptions. *ACM Transactions on Applied Perception*, 1, 57–80

Zhang, G. P., & Qi, M. (2005). Neural network forecasting for seasonal and trend time series. *European Journal of Operational Research*, 160(2), 501–514.

Appendix

A A new hybrid-TD algorithm[1]

Conventional temporal difference updating can be more data efficient than gradient temporal difference updating, but the correction term used by gradient-TD methods helps prevent divergence. Sutton's (2009) results suggested that there are situations (specifically on-policy) where linear TD(0) can outperform gradient TD methods, and Hackman (2012) demonstrated that expected Sarsa(0) can outperform multiple variants of the GQ(0) algorithm, even under off-policy sampling. Our own experiments on the Markov chain demonstrated that a variant of GTD(λ), without gradient correction, can outperform GTD(λ) with gradient correction. On the other hand, linear off-policy TD(0) and GTD(λ) without gradient correction diverged on Baird's counter example.

The idea of hybrid-TD methods is to achieve sample efficiency closer to conventional TD learning, while ensuring non-divergence under off-policy sampling. To achieve this, a hybrid algorithm could do conventional, uncorrected TD updates when the data is sampled on-policy, and use gradient corrections when the data is sampled off-policy. This approach was pioneered by Maei (2011), leading to the derivation of Hybrid Temporal Difference learning, or HTD(0). Later, Hackman (2012) produced hybrid versions of the GQ(0) algorithm. In this section, we derive the first hybrid temporal difference method to make use of eligibility traces, called HTD(λ), using a derivation scheme similar to Maei (2011).

The key idea behind the derivation of HTD learning methods is to modify the gradient of the MSPBE to produce a new learning algorithm. Let $\mathbb{E}_\mu$ represent the expectation according

[1]The derivation below has been updated, to fix a few bugs we found when preparing the algorithm for publication. Thus this section, as presented below, differs from the original document which was submitted to FGSR. The recent paper is titled *Investigating practical, linear temporal difference learning*, and was published at AAMAS 2016.

to samples generated under the behavior policy, μ. The MSPBE can be written as

$$\texttt{MSPBE}(\mathbf{w}) = \underbrace{\mathbb{E}_\mu[\delta_t \mathbf{e}_t]^\top}_{-A_\pi \mathbf{w} + b_\pi} \underbrace{\mathbb{E}_\mu[\mathbf{x}(S_t)\mathbf{x}(S_t)^\top]}_{C}{}^{-1} \mathbb{E}_\mu[\delta_t \mathbf{e}_t],$$

with $\mathbf{e}_t \stackrel{\text{def}}{=} \rho_t(\lambda_t \gamma_t \mathbf{e}_{t-1} + \mathbf{x}(S_t))$, γ_t is shorthand for $\gamma(S_t)$, and

$$\begin{aligned} A_\pi &\stackrel{\text{def}}{=} \mathbb{E}_\mu[\mathbf{e}_t(\mathbf{x}(S_t) - \gamma_{t+1}\mathbf{x}(S_{t+1}))^\top] \\ &= \sum_{s_t \in \mathcal{S}} d^\mu(s_t) \sum_{a_t \in \mathcal{A}} \underbrace{\mu(s_t, a_t)\rho_t}_{\pi(s_t, a_t)} (\gamma_t \lambda \mathbb{E}_\mu[\mathbf{e}_{t-1}|s_t] + \mathbf{x}(s_t)) \\ &\qquad \sum_{s_{t+1} \in \mathcal{S}} P(s_t, a_t, s_{t+1})(\mathbf{x}(s_t) - \gamma_{t+1}\mathbf{x}(s_{t+1}))^\top \end{aligned} \tag{1}$$

$$\begin{aligned} b_\pi &\stackrel{\text{def}}{=} \mathbb{E}_\mu[R_{t+1}\mathbf{e}_t] \\ &= \sum_{s_t \in \mathcal{S}} d^\mu(s_t) \sum_{a_t \in \mathcal{A}} \pi(s_t, a_t)(\gamma_t \lambda \mathbb{E}_\mu[\mathbf{e}_{t-1}|s_t] + \mathbf{x}(s_t)) \\ &\qquad \sum_{s_{t+1} \in \mathcal{S}} \pi(s_t, a_t) P(s_t, a_t, s_{t+1}) r_{t+1}. \end{aligned}$$

Therefore, the relative importance given to states in the MSPBE is weighted by the stationary distribution of the behavior policy, $d_\mu : \mathcal{S} \to \mathbb{R}$, (since it is generating samples), but the transitions are reweighted to reflect the returns that π would produce.

The gradient of the MSPBE is:

$$-\frac{1}{2}\nabla_{\mathbf{w}} \texttt{MSPBE}(\mathbf{w}) = -A_\pi^\top C^{-1}(-A_\pi \mathbf{w} + b_\pi). \tag{2}$$

Assuming A_π^{-1} is non-singular, we get the TD-fixed point solution:

$$0 = -\frac{1}{2}\nabla_{\mathbf{w}} \texttt{MSPBE}(\mathbf{w}) \implies -A_\pi \mathbf{w} + b_\pi = 0. \tag{3}$$

The value of $\mathbf{w}$, for which (3) is zero, is the solution found by linear TD(λ) and LSTD(λ) where $\pi = \mu$. The gradient of the MSPBE yields an incremental learning rule with the following general form (see Bertsekas 1996):

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha(M\mathbf{w}_t + b), \tag{4}$$

where $M = -A_\pi^\top C^{-1} A_\pi$ and $b = A_\pi^\top C^{-1} b_\pi$. The update rule, in the case of TD(λ), will yield stable convergence if A_π is positive definite (as shown by Tsitsiklis and van Roy (1997)). In off-policy learning, we require $A_\pi^\top C^{-1} A_\pi$ to be positive definite to satisfy the conditions of the ordinary differential equation proof of convergence (Maei, 2010), which

holds because C^{-1} is positive definite and therefore $A_\pi^\top C^{-1} A_\pi$ is positive definite, because A_π is full rank (true by assumption). See Sutton et al. (2015) for a nice discussion on why the A_π matrix must be positive definite to ensure stable, non-divergent iterations. The C matrix in Equation (3), can be replaced by any positive definite matrix and the fixed point will be unaffected, but the rate of convergence will almost surely change.

Instead of following the usual recipe for deriving GTD, let us try replacing C^{-1} with

$$A_\mu^{-\top} \stackrel{\text{def}}{=} \mathbb{E}_\mu[(\mathbf{x}(S_t) - \gamma_t \mathbf{x}(S_{t+1}))\mathbf{e}_t^{\mu\top}],$$

where $\mathbf{e}^\mu$ is the regular on-policy trace for the behavior policy (i.e., no importance weights)

$$\mathbf{e}_t^\mu \stackrel{\text{def}}{=} \gamma_t \lambda \mathbf{e}_{t-1}^\mu + \mathbf{x}(S_t).$$

The matrix $A_\mu^{-\top}$ is a positive definite matrix (proved by Tsitsiklis and van Roy (1997)). Plugging $A_\mu^{-\top}$ into (2) results in the following expected update:

$$\begin{aligned}
&\frac{1}{\alpha}\mathbb{E}[\Delta \mathbf{w}_t] = A_\pi^\top A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi) \\
&= (A_\mu^\top - A_\mu^\top + A_\pi^\top) A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi) \\
&= (A_\mu^\top A_\mu^{-\top})(-A_\pi \mathbf{w}_t + b_\pi) + (A_\pi^\top - A_\mu^\top) A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi) \\
&= (-A_\pi \mathbf{w}_t + b_\pi) + (A_\pi^\top - A_\mu^\top) A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi) \\
&= (-A_\pi \mathbf{w}_t + b_\pi) + \\
&\mathbb{E}_\mu\left[(\mathbf{x}(S_t) - \gamma_{t+1}\mathbf{x}(S_{t+1}))\,(\mathbf{e}_t - \mathbf{e}_t^\mu)^\top\right] A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi)
\end{aligned} \tag{5}$$

As in the derivation of GTD(λ) (Maei, 2011), let the vector $\mathbf{h}_t$ form a quasi-stationary estimate of the final term,

$$A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi).$$

Getting back to the primary weight update, we can sample the first term using the fact that,

$$(-A_\pi \mathbf{w}_t + b_\pi) = \mathbb{E}_\mu[\delta_t \mathbf{e}_t]$$

(see Maei 2011), and use (1) to get the final stochastic update

$$\Delta \mathbf{w}_t \leftarrow \alpha\left(\delta_t \mathbf{e}_t + (\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1})\,(\mathbf{e}_t - \mathbf{e}_t^\mu)^\top \mathbf{h}_t\right). \tag{6}$$

Notice that when the data is generated on-policy ($\pi = \mu$), $\mathbf{e}_t = \mathbf{e}_t^\mu$, and thus the correction term disappears and we are left with precisely linear TD(λ). When $\pi \neq \mu$, the TD update is corrected as in GTD: unsurprisingly, the correction is slightly different but has the same basic form.

To complete the derivation, we must derive an incremental update rule for $\mathbf{h}_t$. We have a linear system, because

$$\mathbf{h}_t = A_\mu^{-\top}(-A_\pi \mathbf{w}_t + b_\pi) \implies A_\mu^\top \mathbf{h}_t = -A_\pi \mathbf{w}_t + b_\pi.$$

Following the general expected update in (4),

$$\mathbf{h}_{t+1} \leftarrow \mathbf{h}_t + \alpha_\mathbf{h} \Big((-A_\pi \mathbf{w}_t + b_\pi) - A_\mu^\top \mathbf{h}_t \Big) \tag{7}$$

which converges if $A_\mu^\top$ is positive definite for any fixed $\mathbf{w}_t$ and $\alpha_\mathbf{h}$ is chosen appropriately (see Sutton et al.'s recent paper (2015) for an extensive discussion of convergence in expectation). To sample this update, recall

$$A_\mu^\top \mathbf{h}_t = \mathbb{E}_\mu[(\mathbf{x}(S_t) - \mathbf{x}(S_{t+1}))\mathbf{e}_t^{\mu\top}]\mathbf{h}_t$$

giving stochastic update rule for $\mathbf{h}_t$:

$$\Delta \mathbf{h}_t \leftarrow \alpha_\mathbf{h} \Big[\delta_t \mathbf{e}_t - (\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1})\mathbf{e}_t^{\mu\top}\mathbf{h}_t \Big].$$

As in GTD, $\alpha \in \mathbb{R}$ and $\alpha_\mathbf{h} \in \mathbb{R}$ are step-size parameters, and $\delta_t \stackrel{\text{def}}{=} R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t$. This hybrid-TD algorithm should converge under off-policy sampling using a proof technique similar to the one used for GQ(λ) (see Maei & Sutton's proof (2010)), but we leave this to future work. The HTD(λ) algorithm is completely specified by the following equations:

$$\begin{aligned}
\delta_t &\stackrel{\text{def}}{=} R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t \\
\mathbf{e}_t &\leftarrow \rho_t(\lambda_t \gamma_t \mathbf{e}_{t-1} + \mathbf{x}_t) \\
\mathbf{e}_t^\mu &\leftarrow \lambda_t \gamma_t \mathbf{e}_{t-1}^\mu + \mathbf{x}_t \\
\Delta \mathbf{w}_t &\leftarrow \alpha \Big[\delta_t \mathbf{e}_t + (\gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t)(\mathbf{e}_t^\mu - \mathbf{e}_t)^\top \mathbf{h}_t \Big] \\
\Delta \mathbf{h}_t &\leftarrow \alpha_\mathbf{h} \Big[\delta_t \mathbf{e}_t + (\gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{e}^{\mu\top}_t \mathbf{h}_t \Big]
\end{aligned}$$

The implementation of this new hybrid TD algorithm can be made more efficient by exploiting the common terms in $\Delta \mathbf{w}_t$ and $\Delta \mathbf{h}_t$. Like the proof of convergence, empirical study of this new algorithm is left to future work.

Summary

In this appendix we derived a new TD algorithm, called HTD(λ). This state GVF estimation algorithm performs conventional temporal difference updates when data is sampled

on-policy, and gradient corrected updates otherwise. This new algorithm should be non-divergent, even in the off-policy case (not shown), and it is the first hybrid temporal difference method to make use of eligibility traces. Although not empirically validated here, this new algorithm should improve the efficiency of off-policy GVF learning in practice. We leave it to future work to more fully demonstrate HTD(λ)'s contribution to predictive knowledge learning.

B Algorithms for off-policy GVF learning

Our approach to learning predictive knowledge is based upon learning GVF with incremental, linear complexity, temporal difference learning methods. The experiments contained in these pages have shown that these TD algorithms allow substantial scaling of GVF learning, and these algorithms can learn accurate predictions from real robot data. There are other value function learning algorithms from reinforcement learning that we could use instead of temporal difference methods. This section discusses the merits and limitations of these alternative methods, in the context of large-scale demon learning.

Although we have already discussed the scalability of least square temporal methods in Chapter 5, there are important considerations concerning non-linear function approximation. Linear gradient temporal learning methods, such as GTD(λ), extend with little modification to smooth arbitrary function approximation, and convergence to a local minimum is guaranteed (Maei et al., 2010). Least squares reinforcement learning methods, on the other hand, are not applicable to non-linear function approximation by design. To see this recall that LSTD computes the analytical solution to the MSPBE:

$$w^\star = arg\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \Pi T_\pi^{\gamma\lambda}\mathbf{X}\mathbf{w}||_B^2,$$

a least squares loss function. The projection matrix Π linearly transforms the updates to the value function due to the application of the Bellman operator $T_\pi^{\gamma\lambda}$, enabling algebraic isolation and direct solution of $\mathbf{w}^\star$. A non-linear function approximator, such as a neural network, requires non-linear projection, in which case there is typically no closed form solution for $\mathbf{w}$.

Within the class of linear complexity value function learning algorithms, there are several methods potentially suitable for GVF learning. One important requirement, in the context of predictive knowledge acquisition, is **non**-divergence under off-policy sampling, which facilitates parallel demon learning. Three families of linear-time complexity methods have such convergence guarantees: 1) the new gradient TD methods that minimize the MSPBE, 2) residual gradient methods, and 3) incremental least-squares methods.

Residual gradient methods use the *mean square Bellman error*:

$$\texttt{MSBE}(\mathbf{w}) = ||\mathbf{X}\mathbf{w} - T^{\lambda\gamma}\mathbf{X}\mathbf{w}||_B^2.$$

Temporal difference algorithms, including TD(λ), GTD(λ), and LSTD(λ), do not typically converge to the minimum of the MSBE, with function approximation. Residual gradient

(RG) algorithms, due to Biard (1995), are true stochastic gradient algorithms in that they perform updates based on sample estimates of the gradient of the MSBE with respect to the weight vector, and are guaranteed to converge under off-policy sampling. The RG algorithm requires two independent samples of the next state feature vector ($\mathbf{x}_{t+1}$ and $\mathbf{x}'_{t+1}$), as can be seen it the following update equation:

$$\Delta \mathbf{w} = \alpha(r_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t)(\gamma \mathbf{x}'_{t+1} - \mathbf{x}_t).$$

Using a single sample of the next state can induce bias. To avoid this bias we could generate two independent samples of the next state with a simulator or restrict application to deterministic domains where the two samples would be equal.

There has been some confusion in the community about which objective function, the MSPBE or MSBE, is best for reinforcement learning. A reinforcement learning algorithm may find a bad solution, in the sense that the value estimates, $\hat{v}(s, \mathbf{w})$, are significantly different than the true un-approximated value function in a mean squared error sense (e.g., $\sum_{s \in \mathcal{S}} d_\mu(s)(\hat{v}(s, \mathbf{w}) - v(s; \pi, \gamma, z))^2$). This is not the same as divergence. There have been several studies comparing MSBE minimization to MSPBE minimization. Scherrer (2010) for instance, found that RG, on average on random MDPs, produced worse solutions than TD(0), in comparison to the true value function. Scherrer also reported that worst performance of TD(0) produced value function estimates that were further from the true value function than the worst case estimates of RG. Scherrer called this "numerical instability" of TD(0), but did not report divergence. Scherrer's results, however, were obtained using the two sample version of RG, and thus he did not compared the biased version of RG. Scherrer also conjectured that using the MSPBE minimization algorithms with λ greater than zero would eliminate the numerical instability issues. A later study confirmed this hypothesis (Geist & Scherrer 2013), showing that LSTD(λ) did indeed outperform RG(λ) (a variant of RG with eligibility traces) across a large set of random MDPs. Geist and Scherrer claimed existence of a case of divergence for LSTD(λ), but this was in fact an example of a bad solution not divergence, and could be avoided by changing the value of λ. A recent study of value function approximation in reinforcement learning (Dann et al., 2013), reported no evidence of "numerical instability" of TD, and found MSPBE minimization methods always outperformed MSBE methods across 16 small simulation problems. We conclude, based on the evidence from the literature, that MSPBE-based methods should be robust and efficient for GVF learning.

Another class of learning algorithms of interest, which do minimize the MSPBE, attempt

to achieve the sample efficiency of LSTD with computational requirements closer to that of linear TD methods. The incremental-least squares temporal difference learning algorithm, or iLSTD, (Geramifard et al.. 2006) method does indeed achieve $O(n)$ computation—though not as computationally frugal as TD(λ)—and requires $O(n^2)$ storage. In practice iLSTD, like LSTD does not easily extend to non-linear function approximation, and so far related extensions to the control case have not been found to be well suited to online use (Bowling, Geramifard, & Wingate 2008).

C Intrinsically motivated reinforcement learning

Exploration bonuses	State-action-value is augmented: $\widehat{q}(s,a) = q(s,a) + \sqrt{nc(s,a)}$. $nc(s,a)$ is a count of visits to a state-action
(Sutton 1990)	**Goal**: maximize external reward & explore recently unexplored states
Maximizing error	Reward to controller is $r_t = M(o_t, \theta_{t-1}) - M(o_t, \theta_t)$. M is a forward model with parameters θ
(Schmidhuber 1991)	**Goal**: take actions that induce large modeling errors
Option progress	Behavior reward: $r_t = r_t^e + r_t^i$. Internal reward $r_t^i = \tau[1 - M_o^\pi(s_{t+1}\|s_t)]$ if transition to *salient state* s_{t+1}. $M_o^\pi(s_{t+1}\|s_t)$: predicts probability of reaching s_{t+1} from s_t under π. Option-policy π maximizes external reward r_t^e
(Singh et al., 2005)	**Goal**: select actions to improve option training by reducing salience, and maximize external reward on a complex hierarchical task
Value-function Δ	Behavior reward $r_t^i = \tau + \sum_{s\in\mathcal{S}}(v_t^{max}(s) - v_{t-1}^{max}(s))$: $v_t(s)$ updated using external reward r_t^e, and $v_t^{max}(s) = \max_{j\leq t} v_j(s)$ is the max value observed over all previous visits to s
(Simsek & Barto 2006)	**Goal**: focus exploration in regions of the state space where learning improves the value function
Δ in prediction error	Behavior takes action of expert with max estimated progress in current context. Experts progress is equal to $r_t^i = -(\Delta_0 - \Delta_k)$: $\Delta_k = 1/J \sum_{i=0}^{J} \delta_{t-i-k}$ and $\delta_t = \|\|o_{t+1} - M^i(o_t, a_t)\|\|^2$ the one-step prediction error of expert i.
(Oudeyer et al., 2007)	**Goal**: select expert's action with highest expected change in prediction error

Table 1: A summary of the different internal and external reward functions used in several in intrinsically motivated reinforcement learning systems. We use $r^i \in \mathbb{R}$ to denote the internal reward, $r^e \in \mathbb{R}$ to denote the external reward, and $r \in \mathbb{R}$ to denote total reward. Input data, either sensorimotor data or observations are denoted with $o \in \mathbb{R}$. Models, either one-step forward models or multi-step option models are denoted by M. The δ_t corresponds generically to an instantaneous error, such as TD error or squared one-step prediction error. Finally $v(s)$ and $q(s,a)$ refer to conventional state and state-action value functions. Table continues in 2, on the next page.

Subtask reward	Behavior reward $r^e_{t+1} = \delta^i_t$ of selected expert i. Expert i learns policy with internal reward: $r^i_t = M^i(o_{t+1})$. M^i trained to predict external subtask reward from executing the policy of expert-i.
(Schembri et al., 2007)	**Goal**: select actions from expert with largest critic error
Evolved rewards	Find the best reward function, on average, over a distribution of agents and environments, with externally defined fitness.
(Singh et al., 2010)	**Goal**: Search for rewards then compute reward maximizing policy
Model progress	Behavior reward equal to external task reward plus model improvement estimation. Model improvement is estimated by log likelihood of data evaluated on all observed data and a subset of the data.
(Lopes et al., 2012)	**Goal**: Behavior gets a bonus for exploring state-action pairs where expected progress is large

Table 2: A summary of the different internal and external reward functions used in several in intrinsically motivated reinforcement learning systems. We use $r^i \in \mathbb{R}$ to denote the internal reward, $r^e \in \mathbb{R}$ to denote the external reward, and $r \in \mathbb{R}$ to denote total reward. Input data, either sensorimotor data or observations are denoted with $o \in \mathbb{R}$. Models, either one-step forward models or multi-step option models are denoted by M. The δ_t corresponds generically to an instantaneous error, such as TD error or squared one-step prediction error. Finally $v(s)$ and $q(s, a)$ refer to conventional state and state-action value functions.

D Thoughts on organizing predictive knowledge

This section addresses the question of how to we might organize and use the predictive knowledge stored in our Horde of demons. Let us set aside the question of how to generate new demons for the moment, and first consider how to organize different types of on-policy, off-policy, exponential-form, and state-based GVFs. We can take further inspiration from psychology and nexting, and imagine a demon organization scheme that could improve demon learning and behavior adaption.

A fundamental characteristic of nexting knowledge, as argued by Gilbert (2007), is that nexting-knowledge is a special form of knowledge of the future which is unconscious and automatic. Nexting knowledge is also often related to the efficiency of the agent (in his case, humans). For example, we next at the letter and word level while reading, which in turn allowsfaster reading. We make predictions about the next locations of a ball—as it flies through the air—which aids tracking and ultimately improves our chances of catching the ball. As we bike down the road we predict future bumps resulting in better steering and a smoother ride. These short instant-by-instant predictions seem to happen automatically without deliberation. Nexting predictions might play a role in updating other predictions and basic skill acquisition.

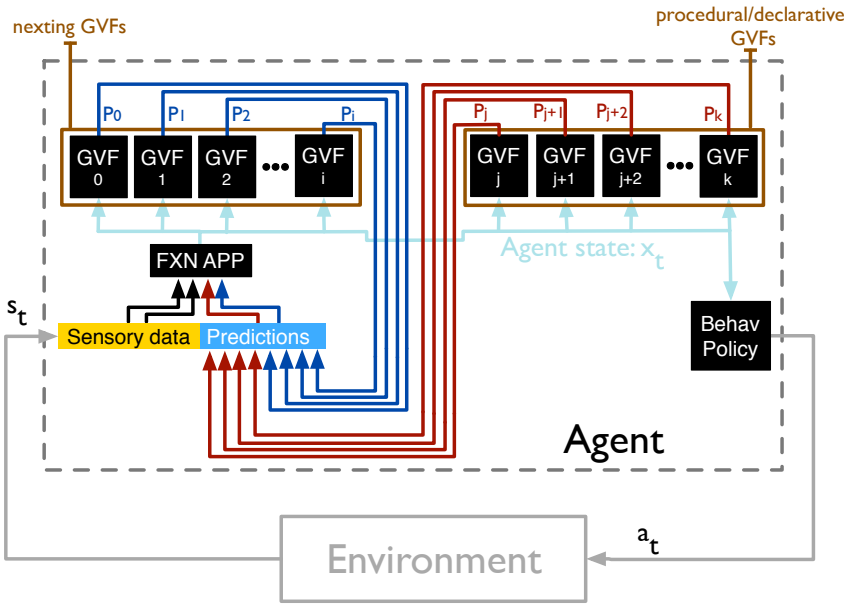

Figure 1: A diagram of how many demon's predictions could participate in the feature vector generation process, and then indirectly the behavior. At the highest level (dotted lines and gray box) we see the familiar reinforcement learning agent-environment interaction diagram.

One way to leverage nexting knowledge, is to allow nexting demon predictions to participate int the feature vector generation process. The feature vector can be constructed from many different demon's predictions, as we demonstrated in the iRobot Create experiments of Chapter 6. All the demons have access to and share the feature vector, and thus any demon can take advantage of the potential increased representational power afforded by this form of predictive state feature vector. Figure 1 visualizes how the outputs of many demons could interact.

On policy nexting predictions might be good candidates to generate predictive feature components because they are easy to generate and are faster to learn than other forms of prediction. The question of where do the demons come from is more straightforward to answer in the case of on-policy nexting. Essentially we generate as many nexting demons that can be updated given our computational budget: forming unique combinations of sensor and feature component cumulants, with as many time scales as possible. Nexting predictions obtain accuracy quickly compared to policy-contingent predictions, because all interaction data is relevant and useful for learning, unlike off-policy learning.

The usefulness of an nexting-based feature component is related, in part, to the relevance of the behavior each demon's question. If the behavior data is not relevant to some demon's target policy then, we would expect that demon's learning to be slowed. In addition, the nexting-based feature components might not be useful either. The opposite should also be true. If the behavior policy generates training data relevant to some demon, then we might expect that the nexting predictions would also be relevant to that demon. Many demon's predictions could participate in the feature generation process, over time yielding increasingly complex interactions between the behavior policy, feature vector generation, and prediction learning.

Given the demon organization scheme outlined above, let us get back to the question of how to generate new demons. Loosely speaking, an ambitious overall goal for Horde would be to understand a simple and limited physical world, in some concrete measurable way. For example, an agent that could predict the sensorimotor consequences of all its actions, and attempt to control its sensorimotor stream. Predicting and controlling the world have been referred to as *mastery over ones environment* (White 1965). White proposed that (1) knowing what is possible in a particular environment and (2) bringing the environment under one's control are two important abilities of an autonomous agent. The idea of mastery suggests an intriguing experiment and a straightforward way to generate tens of thousands

of GVFs: predicting and controlling all of a robots sensors. This could be implemented as a collection of control demons learning policies which maximize and minimize each of the robots sensors, at several time scales. In addition we would specify prediction demons using (1) each sensor, (2) several constant termination signals, and (3) each control demon's policy. All total this setup would produce approximately 90,000 demons. This experiment could provide a concrete demonstration of a robot that is aware of its surroundings, and knows about its environment in a real sense, even if the robot was restricted to a limited space like the pen.

E The parallel scalability of Horde

The Horde architecture is a parallel computing system, and thus we can analyze the speedup and parallel scalability of its implementation. The majority of the operations within each demon are easily parallelizable: we can update each target policy, reward function, GTD(λ) instance, and RUPEE$_{scalar}$ measure asynchronously. Only the update of the tile coder and communication with the robot are sequential. Amdahl's law (see Pacheco 1997) specifies the theoretical speedup possible using $Pn \in [1, 2, ...]$ processors/cores, given the fraction of operations requiring sequential processing, $F \in [0, 1]$, equals:

$$S(Pn) = \frac{1}{F + \frac{1}{Pn}(1 - F)},$$

where $S(Pn)$ is the speedup possible with Pn processors. In addition, we can calculate the total possible speedup; letting Pn go to infinity $S(Pn)$ becomes $\frac{1}{F}$. Our parallel architecture, while updating 3000 demons, achieved $F = 1\%$ measured by the YourKit Java profiler, meaning that 99% of the system's operations are parallelizable. We should therefore, under ideal conditions, expect nearly 100 times speedup given enough parallel cores.

# Parallel Cores	Runtime (ms)	Speedup (SU)	Theoretical SU
1	200.74	1.0	1.0
2	115.60	1.74	1.98
3	90.26	2.22	2.94
4	86.15	2.33	3.88
8	76.84	2.61	7.48

Table 3: Number of parallel cores verses time to perform a single update of 3000 off-policy demons, the corresponding speedup, and theoretical speedup.

These calculations provide an upper bound on possible speedup, and are therefore somewhat idealistic because they do not account for cache coherence, communication, or other implementation details. As a test we ran 3000 demons with 300 distinct target policies with performance measured via RUPEE$_{scalar}$. We used this measure because it only stores of a scalar value, rather than the vector storage used by RUPEE$_{vex}$, and thus RUPEE$_{scalar}$ is more computationally frugal. We varied both the number of parallel cores and the number of demons. Table E reports the results.

We do not achieve theoretical speedup. Our implementation achieves parallelization using multithreading in java, which is not a classic parallel computing language like to C or C++ with OpenMP or message passing interface. Test machine had only four physical cores,

# GVFs	Runtime (ms)	Linear trend
3000	77.96	77.96
1500	39.42	39.88
750	24.84	19.49
375	11.90	9.75

Table 4: Number of demons verses runtime on an eight parallel core machine.

using hyper threading to simulate eight virtual cores, and thus the results with eight cores might not be representative of eight physical cores.

Table E reports how the update time changed as a function of the number of demons. As the number of demons decreased, the computation time decreased almost linearly, as we would expect. Once the number of demons became smaller than 750, F measured equal to 18%, potentially explaining the deviation from the linear trend.

The take home message is that the majority of operations involved in demon learning are parallelizable. We predict that the number of demons updatable in realtime should continue to scale, approaching tens or hundreds of thousands of in the next couple of years, as the processing power of computing resources increases. Further scaling could be achieve using vectorized GPU implementations and hybrid shared/distributed compute clusters requiring message passing between nodes. These implementations are left to future research.