University of Alberta

Experiments in Off-Policy Reinforcement Learning with the GQ(λ) Algorithm

by

Michael Delp

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

Master of Science

Department of Computing Science

© Michael Delp
Spring 2011
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

Richard Sutton, Computing Science, Supervisor

Hong Zhang, Computing Science

Marek Reformat, Electrical and Computer Engineering, External Examiner

Abstract

Off-policy reinforcement learning is useful in many contexts. Maei, Sutton, Szepesvari, and others, have recently introduced a new class of algorithms, the most advanced of which is GQ(λ), for off-policy reinforcement learning. These algorithms are the first stable methods for general off-policy learning whose computational complexity scales linearly with the number of parameters, thereby making them potentially applicable to large applications involving function approximation. Despite these promising theoretical properties, these algorithms have received no significant empirical test of their effectiveness in off-policy settings prior to the current work. Here, GQ(λ) is applied to a variety of prediction and control domains, including on a mobile robot, where it is able to learn multiple optimal policies in parallel from random actions. Overall, we find GQ(λ) to be a promising algorithm for use with large real-world continuous learning tasks. We believe it could be the base algorithm of an autonomous sensorimotor robot.

Acknowledgements

First and foremost, I would like to thank two people, without whom little of this research would have been possible, Hamid Maei and Dr. Thomas Degris-Dard. Hamid spent countless hours explaining to me the GQ(λ) algorithm and how to derive the TD Fixed point. Thomas shared his beautifully coded reinforcement learning and Critterbot libraries, and spent many hours installing programs on my machine. I would also like to thank my supervisor, Dr. Richard Sutton, for teaching me about Reinforcement Learning and all the steps involved in evaluating RL problems empirically. I would like to thank Michael Sokolsky and Marc Bellemare, for making and supporting the Critterbot, respectively. I extend my appreciation to Doina Precup for late nights discussing off-policy learning as we tried to finish papers, Patrick Pilarski for taking the time to read my thesis and offer comments, and Alberta Ingenuity for funding most of my thesis work. Thanks go out as well to my committee members, Dr. Hong Zhang, and Dr. Marek Reformat for taking the time to listen to me. Lastly, I would like to thank my father, Peter Delp, for his help with editing and for moral support.

Contents

1 Introduction: Off-Policy Learning **1**

2 Background **4**

2.1 Reinforcement Learning 4

2.2 Function Approximation 5

2.3 Temporal Difference Learning 6

2.4 On-Policy Learning 7

2.5 Off-Policy Learning 7

2.6 Off-Policy Advantages 8

2.7 Off-Policy Instability 9

2.8 A Brief History of Off-Policy Methods 11

2.9 Summary 12

3 The GQ(λ) Algorithm **14**

3.1 Theory 14

3.2 Algorithm 16

3.3 Calculating the TD Fixed Point 17

4 Effect of the Disparity between Behavior and Target Policies **20**

4.1 The Off-Policy Distance 20

4.2 The Random Walk Problem 21

4.3 Experiment 1: Varying the Target Policy 22

4.4 Experiment 2: Varying the Behavior Policy 26

4.5 Experiment 3: Varying Both the Target and Behavior Policies 27
4.6 Change in Error as the Off-Policy Distance Grows 31
4.7 Conclusions 33

5 Learning Multiple Optimal Policies Simultaneously on a Robot 35

5.1 Objectives 36
5.2 Experimental Setup 37
5.2.1 Sensors to Maximize 37
5.2.2 Behavior Policy 38
5.2.3 Time Step Duration 39
5.2.4 Speed of Movement 40
5.2.5 Feature Representation 40
5.2.6 Feature Selection 41
5.2.7 Algorithm 41
5.2.8 Evaluation 42
5.3 Results 43
5.4 Discussion 44

6 GQ(λ) vs. TD(λ) with Importance Sampling 46

6.1 Experimental Setup 47
6.2 Results 48
6.3 Analysis of Results 51

7 Conclusion 55

7.1 Strengths and Weaknesses of GQ(λ) 57
7.2 Future Directions 57

Appendix **62**

A The Critterbot **62**

A.1 Sensors 62

A.2 Timing Delays 65

List of Figures

2.1 Q-learning Cliff Walk Example 8

2.2 Baird's Counterexample 10

3.1 GQ(λ) Algorithm with Linear Function Approximation 16

4.1 5-State 2 exit random walk example from Sutton and Barto (1998), pg 139, used with permission 21

4.2 Learning curves for various λ with $\alpha = 1$ for the B50-T50 group . . . 23

4.3 Parameter study over all α and λ values on the 50/50 target policy. 25

4.4 Best learning curves on every target policy 26

4.5 Performance over all target policies 26

4.6 Performance using various behavior policies over the equal random target policy 27

4.7 Performance using each behavior policy (lines) over different target policies. Bottom graph is a zoomed in view. 29

4.8 Performance as the target policy differs further from each behavior policy 32

4.9 Performance as the behavior policy differs further from each target policy 33

4.10 Learning performance as a function of the off-policy distance 34

5.1 The Critterbot 36

5.2 Tile coding with 2 overlapping tilings (from Sutton & Barto, 1998, used with permission 40

5.3 Learning curves for eight sensor maximization tasks. 43

5.4 Illustration of policies learned for sensor maximization. 44

6.1 Gridworld environment from *Off-Policy Temporal-Difference Learning with Function Approximation* (Precup et al., 2001, used with permission) 47

6.2 GQ(λ) parameter studies on 11x11 gridworld problem 49

6.3 Weight value estimates of the leftmost and rightmost features for the down action 50

6.4 Heat graphs of the exact values for different policies 52

6.5 The distribution of the behavior policy 53

6.6 Heat graph for value function $V^{\pi}(s)$ for tabular case 54

A.1 The Critterbot 63

A.2 The Critterbot's omnidirectional wheels 63

A.3 Various sensor locations and there orientations on the Critterbot . . 64

A.4 Critterbot Latencies (courtesy of Thomas Degris) 66

A.5 Movement at different time scales on the Critterbot 68

Chapter 1

Introduction: Off-Policy Learning

There is a need for automatic controllers that can learn from their experience in many real world applications (e.g., nautical control, cell phone tower switching, and precision metal cutting). For example, in the case of automatically controlling a boat, a computer based *agent* would need to determine which actions to take, like turning the rudder left or right and increasing or decreasing the throttle, in order to stay on course. The agent could keep track of how actions previously chosen affected the heading and distance traveled in order to better pick future actions, forming a *policy* of which actions to take in each situation or state. The agent could keep learning and reacting as the boat travels in order to adjust to new currents and other factors affecting the boats movement in the water. This is an *on-policy* problem because the agent is learning about the actions it is currently performing.

Sometimes it is not possible for the agent to learn while performing the actions. For example, a company that wants to analyze its past decisions to improve future profits may have a large amount of previously generated data. The company may wish to research the possible impact of running the production line in a different way or of putting more emphasis on overseas sales. This would be an *off-policy* problem, because the policy being learned about, the *target policy*, would be different than the policy used to generate the data, the *behavior policy.*

Learning about many separate policies from a single stream of data is another off-policy problem. Learning about multiple policies in parallel could lead to large time savings over learning about each sequentially. For example, if a robot was learning how to affect the values reported by certain sensors, it might notice that turning right towards the window would increase its light sensor values. Turning right would also increase its infrared distance sensor value, because it would be pointed at the wall where the window is placed. However, turning right might

decrease its magnetometer sensor value which measures magnetic north because the wall is south facing. If learning off-policy, it could learn how to affect all three of these values at once by taking the same action, thus reusing one set of results to learn policies to affect three different sensor values, the target policies.

Although an ability to learn off-policy would be useful, it has proven difficult to design efficient algorithms with this ability. Historically, algorithms for off-policy learning have been either slow, or unstable, or nonlinear batch algorithms that can not be performed online on big problems. Recently, Maei, Sutton, Svespari, and others, proposed a family of gradient-based learning algorithms that theoretically guarantee stability. GQ(λ) is the algorithm from this family meant for use on off-policy problems. It can be run online, in linear time and space complexity.

GQ(λ) seems like a promising algorithm for off-policy learning. With GQ(λ), we could learn optimal control policies for large real-world problems like SPAM filters. We could make a sensorimotor robot build up a set of actions and predictions from simple trial and error interaction with its environment. According to the authors of the algorithm, GQ(λ) "brings us closer...to the development of a universal prediction learning algorithm suitable for learning experientally grounded knowledge of the world" (Maei and Sutton, 2010). In effect, it could be the base algorithm of the "killer app" for artificial intelligence, a general knowledge learning algorithm.

However, to date, no empirical work has been published using this algorithm. Does it work? What are the issues with it? Is it faster than similar competitors? Is the solution that it approximates what we want? It remains a question as to whether this algorithm is really useful for off-policy learning.

This thesis aims to close this gap in the literature by testing the algorithm off-policy in a range of domains, from simple random-walk experiments to a mobile robot experiment involving the learning of multiple optimal target policies simultaneously. Along the way, some opinions are developed about the strengths and weaknesses of the algorithm which are presented in the conclusion.

This thesis presents two primary contributions. The first contribution is the empirical experiments using GQ(λ), which represent the first published empirical work with this algorithm. One experiment uses an offshoot of GQ(λ) *GreedyGQ(λ)* in a robotics domain. This is the first experiment using control with any of the new gradient based family of algorithms. The other experiment compares GQ(λ) to TD(λ) with importance sampling, a previous off-policy algorithm, in a gridworld domain(Precup et al., 2001). The two algorithms are compared in terms of speed

of learning and the solutions they approximate (see Chapter 6). This thesis is also useful source material for developing some intuitions about how to set parameters when using GQ(λ).

The other primary contribution is a demonstration of learning multiple optimal policies simultaneously on a robot (see Chapter 5). There have been few, if any, experiments showing this in the robotics domain. This experiment serves as a building block towards building an autonomous sensorimotor agent.

As a secondary contribution of this work, we will develop some insights into how off-policy learning performance suffers as the target and behavior policies get further apart from each other through experiments in a random-walk domain (see Chapter 4).

In Chapter 2, we will take a look at a brief history of reinforcement learning, and especially off-policy methods. Then we will take a detailed look at the GQ(λ) algorithm in Chapter 3.

Chapter 2

Background

This chapter provides a background on reinforcement learning and off-policy learning that sets the context for the experimental inquiry and analytical results that follow.

2.1 Reinforcement Learning

This thesis is meant to be a contribution to the field of reinforcement learning, which can generally be described as the study of learning from experience, and has extensions to trial and error learning, automatic controllers, and even neuroscience. In the standard reinforcement learning framework, a learning agent interacts with an environment consisting of a finite Markov decision process (MDP). At each (generally discrete) time step t, the agent starts at a state, $s_t \in S$, takes action $a_t \in A(s)$, moves to a new state in the environment $s_t + 1 \in S$ and receives a scalar reward $r_t \in R$, takes another action, and so on. The state and action sets are finite. State transitions are stochastic and dependent on the immediately preceding state and action. The agent selects actions according to a policy π which is a probability distribution over A(s) at each state s.

The objective is to learn an approximation to its state-value function:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s_0 = s\right] \tag{2.1}$$

where $\gamma \in [0, 1]$ is a factor for discounting future rewards. Alternatively, a state-action value function could be learned, denoted $Q^{\pi}(s, a)$. Solving a reinforcement learning problem over a fixed policy is called a *prediction* task.

A *control* task is defined as a task in which the agent is concerned with finding

the best policy for maximizing reward over time. The optimal policy π^* is the policy which maximizes $V^\pi(s)$ for all $s \in S$. The optimal action-value function, denoted $Q^*(s,a)$, is used to find the optimal policy.

$$Q^*(s,a) = max_\pi Q^\pi(s,a) \tag{2.2}$$

A sequence of interactions between the agent and the environment from time step 0 to n is referred to as a *trajectory*. In some MDPs, there are goals or end states, and learning takes place over episodes, termed *episodic* problems. When a termination point is reached the agent is placed at a starting state or states determined by the MDP. If learning continues indefinitely with no termination points it is called a *continuous* task.

2.2 Function Approximation

In a Markov Decision Process, each state is ideally meant to encapsulate enough information such that state transitions are dependent only on the immediately preceding state and action. When each state is directly represented, the representation of state is referred to as *tabular*.

In many problems of interest the state set is too large for it to be practical to approximate the value of each state individually. In this case, we use *function approximation* to generalize over groups of states instead of reason about them directly. In a type of function approximation called *linear function approximation*, states are mapped to a vector of features, $\phi \in \Re^n$. These features are then multiplied by a vector of learned weights $\theta \in \Re^n$ to get the value function V_θ

$$V_\theta(s) = \theta^\top \phi(s). \tag{2.3}$$

For control or to predict the value of each action, we use a separate set of features for each action to approximate Q_θ:

$$Q_\theta(s,a) = \theta^\top \phi(s,a). \tag{2.4}$$

We will only use linear function approximation in this thesis.

2.3 Temporal Difference Learning

There are many methods to learn estimates of value functions. When state-action transition probabilities are known beforehand, dynamic programming methods can be used, such as Policy evaluation and improvement or Value Iteration. Unlike dynamic programming, *Monte Carlo* methods can learn from experience with the environment. They must wait until the agent has reached the end of the episode before updating the value at each state visited along the way. *Temporal difference* learning, or TD, takes the opposite approach and updates each state value immediately based upon the current reward and its previous state estimate (Sutton, 1988).

TD(0) is an algorithm which updates based upon the last state transition only, termed the *one-step backup*. A method which combines one-step backups, two-step backups, three-step backups and so on is called TD(λ). It uses the parameter $\lambda \in [0, 1]$ combined with an eligibility trace to influence the weight of each of these predictions. An eligibility trace, e, is a vector of weights that keeps track of previously visited states or features. These are updated every time step along with the current state and then the eligibility is trace is decayed by $\lambda\gamma$. Lower values of λ put more importance on shorter predictions and higher values of λ place more importance on longer predictions (Sutton and Singh, 1994). When eligibility traces are allowed to grow unbounded this is called *accumulating traces*. When they are bounded to a maximum value (usually 1 for each feature) this is called *replacing traces* (Singh and Sutton, 1996).

Temporal Difference learning is an important concept because it allows the agent to learn from fragments of experience. It *bootstraps* from previous experience. The main update step in a temporal difference algorithm is the TD update. The TD update is the current reward plus the discounted value estimate at the next state, minus the estimate at the current state. Here is the TD update for tabular Sarsa (an on-policy TD algorithm for control)(Sutton, 1996):

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where $\alpha \in (0, 2)$ is a step-size parameter.

2.4 On-Policy Learning

When a reinforcement learning agent is learning about the policy that is used to choose the actions, this is referred to as the *on-policy* problem. In the on-policy control case, the agent takes a new action at each time step using a policy derived from the current value function. If this policy is to simply take the action with the highest estimated action value at state s, we call it the *greedy* policy. However, learning with a greedy policy can sometimes lead to poor results, especially in non-deterministic or non-stationary environments. To achieve the optimal policy it is usually necessary to incorporate exploratory actions in order to reach more of the state space. Thus policies that randomly incorporate sub-optimal actions may be preferable (e.g.policies that use soft-max action selection (Luce, 1959; Bridle, 1990) or ϵ-greedy action selection). ϵ-greedy action selection chooses the action which would achieve the highest value according to the value function with probability $1-\epsilon$, and a random action with probability ϵ, thus guaranteeing that the agent will keep exploring the action space.

2.5 Off-Policy Learning

Learning about a different policy than the one being used to interact with the environment is referred to as an *off-policy* problem. A reinforcement learning agent that learns an off-policy problem is doing off-policy learning. The policy the agent is running is referred to as the *behavior policy*, b, and the policy it is trying to learn about is referred to as the *target policy*, π.

The first learning methods for use on off-policy problems were Off-Policy Monte Carlo control methods (Sutton and Barto (1998), pp.126-127). The most renowned off-policy method is Q-learning (Watkins and Dayan, 1992). The *one-step update* for tabular Q-learning is:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.5}$$

Q-learning is remarkable because it allows the learning agent to use an exploratory behavior policy while still learning the optimal target policy. A classic example of this is the cliff walking problem depicted in Figure 2.1. This is a deterministic episodic problem where the reward is -1 per time step in order to induce

(htb)

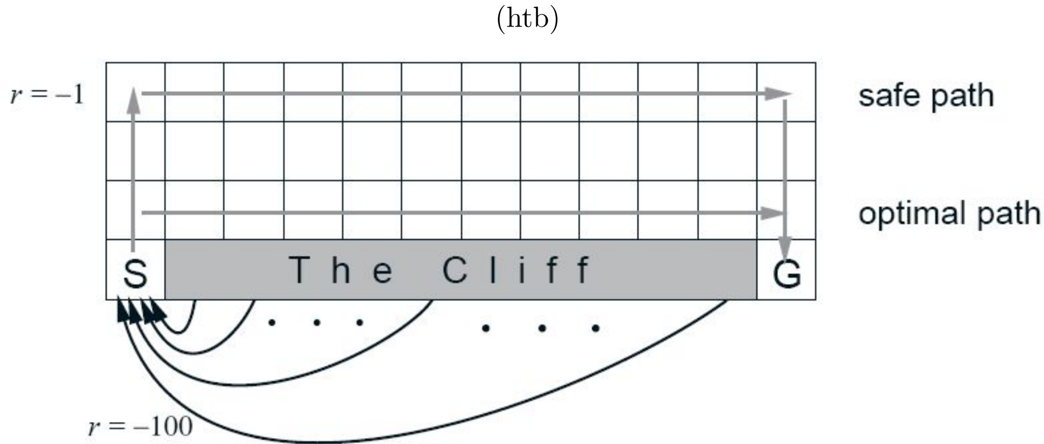


Figure 2.1: Q-learning Cliff Walk Example (from Sutton and Barto (1998), pp. 150, used with permission)

learning the quickest path to the goal, and any step off the cliff results in a huge penalty of -100. ϵ-greedy action selection is used, with $\epsilon = 0.1$.

In order to contrast on-policy and off-policy methods, both are tried on this problem: Sarsa (on-policy) and Q-learning (off-policy). Using Sarsa, a policy is learned which steers several steps clear of the cliff (the "safe path" in Figure 2.1). By using Q-learning, a policy is learned which takes a shorter path to the goal, one that skirts the edge of the cliff (the "optimal path" in Figure 2.1). Sarsa is an on-policy learning algorithm, so it must take into account exploratory actions which could lead it to step off the cliff resulting in a large negative reward. Q-learning, an off-policy algorithm, can ignore the effects of exploratory actions and instead pick the maximal action in each case, and thus the shortest path. This shows the power of off-policy methods to find the best path and ignore the exploratory moves in the behavior policy.

2.6 Off-Policy Advantages

Off-policy learning is useful for the following:

1. **Learning from Preexisting Data:** There is a wealth of preexisting data in the real-world. Learning on this data is referred to as *offline* because the learning is being done after the data is collected, not while it is being collected which is referred to as *online* learning. These are not to be confused with

off-policy and on-policy learning, which are orthogonal concepts. When faced with preexisting data an on-policy learner can only learn about the policy used to create the data. An off-policy learner could potentially learn about any other policy from this data including an optimal control policy. This is important because in many cases, collecting more data to be used by an on-policy learner may be expensive or even impossible.

2. **Parameter Studies:** Off-Policy learning can be used to pick the best parameter settings on the data offline. Picking parameters is an important burden of many reinforcement learning algorithms that must be accounted for until better auto-step size algorithms and feature selection algorithms are made.

3. **State Space Exploration:** On-policy learning can suffer from having to incorporate exploratory actions into its value function. This was illustrated earlier with the cliff-walk example. With off-policy learning, the agent is free to explore the state space however it wishes and the learner will be able to discard the exploratory actions and concentrate on the optimal ones.

4. **Learning many functions simultaneously:** With off-policy learning, one can learn about many different target policies in parallel. On policy methods, by definition, must focus on learning one value function at a time, but using off-policy techniques the agent can learn as many policies and predictions as it wants as long as they all fit into computer memory. This could lead to tremendous time savings over learning each target policy sequentially.

2.7 Off-Policy Instability

Off-policy learning is clearly useful. However, there is an issue with off-policy methods when combined with three desirable traits: 1) linear function approximation, 2) temporal-difference learning, and 3) $O(n)$ complexity per-time-step computation and memory as in Q-learning or TD(λ). In 1995, Leemon Baird discovered a counter example on which TD(λ) diverges, that is, the weights grow towards infinity over time (see Figure 2.2 for more details).

This is unfortunate because these three traits are highly desirable. The first trait, linear function approximation, is the simplest and most widely used form of function approximation. Function Approximation is necessary in most large scale applications of reinforcement learning. The second trait, temporal-difference learning, is desirable because it allows the reinforcement learning agent to learn from snippets

of experience. In an off-policy setting, the agent may only follow the target policy for short periods of time, and thus a monte-carlo learning agent that must wait until the end of an episode to update its weights will have trouble piecing together an update appropriately. Temporal-difference learning is also important in continuous problems where there are no episodes, and thus its necessary to continually update knowledge. The third trait, $O(n)$ complexity, is necessary on large scale applications where there are simply too many features to fit in a computer's memory when they are squared. For example, a computer program to learn how to play the game of Go can have one million features (Silver et al., 2008). Applications for online spam or suspicious URL filters (Ma et al., 2009) can have as many as 300,000 features. If each feature only took one byte of memory, an $O(n^2)$ algorithm would have to have on the order of 686 gigabytes of memory to process 300,000 features. It would also take a very long time to make all of the computations involved. Online algorithms need to be able to finish processing within one time step or they will lose data. Even problems with a small number of features can blow up exponentially when used with certain types of function approximators. Thus we can see that an $O(n)$ algorithm in both time and space complexity is desirable.

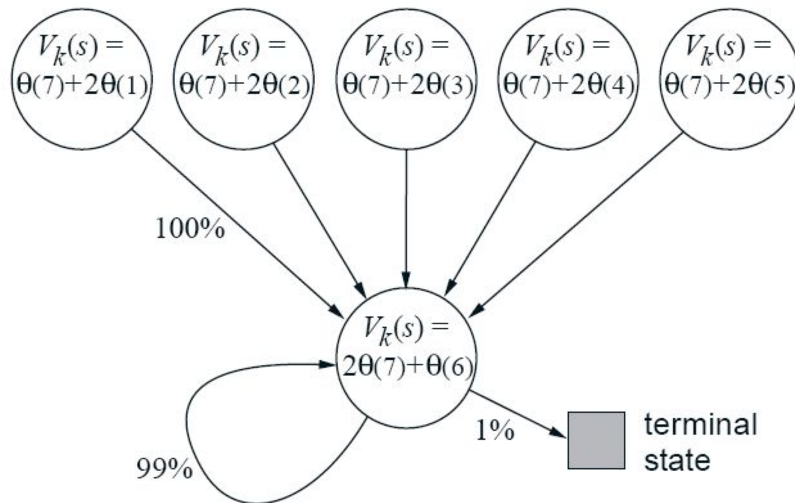


Figure 2.2: Baird's Counterexample. The approximate value function for this Markov process is of the form shown by the linear expressions inside each state. The reward is always zero. If the parameters are set to $\gamma = 0.99$, and $\overline{theta}_0 = (1, 1, 1, 1, 1, 10, 1)^T$, then the θ weights will diverge to infinity over time using Q-learning with any positive step-size. From (Sutton and Barto, 1998), used with permission.

In practice, Q-learning rarely diverges when using small step-sizes and when the behavior policy is close to the target policy (as in $\epsilon - greedy$ methods). However, its potential instability is unsettling. In order to do general off-policy learning in a

function approximation setting, where we are learning multiple different targets, or learning from precomputed data that does not follow closely to our intended target policy, we need an algorithm that is guaranteed not to diverge with few limiting conditions.

2.8 A Brief History of Off-Policy Methods

There have been several attempts over the years to find a stable algorithm for reinforcement learning which incorporates the four desirable traits mentioned earlier: off-policy updates, temporal-difference learning, linear function approximation, and $O(n)$ complexity.

Baird's residual gradient algorithm (Baird, 1995) is stable; however, it is much slower than Q-learning. In addition, it requires two independent samples at each state, which is impractical for a learning agent operating in the real-world. Lastly, it does not converge to the TD solution. It converges to a different solution, which makes the value of a state look like the value of the states preceding it and following it, and should be avoided (Sutton et al., 2009a; Sutton, 1988; Dayan, 1992).

Precup et al. (2001) developed an algorithm called TD(λ) with importance sampling. This algorithm updates the state-action pairs according to the same distribution that would be experienced under the target policy π. It uses importance sampling corrections to make the expected value of the sum of the weight changes match the expected value of on-policy TD(λ). Because Tsitsiklis and Van Roy (1997) proved that on-policy TD(λ) will always converge to near the best solution using function approximation without diverging, TD(λ) with importance sampling is also stable under certain conditions.

TD(λ) with importance sampling is not a perfect solution, however. The problem is that importance sampling weights are cumulative products of target-to-behavior-policy likelihood ratios from the entire episode, and thus updates will have high variance. This can lead to very slow learning. Using this algorithm, two gridworld function approximation examples from the literature required hundreds of thousand or even millions of episodes to learn on (Precup et al., 2001; Rafols, 2006). Eddie Rafols used modified versions of TD(λ) with importance sampling to learn multiple predictions and option-conditional TD Networks off-policy (Rafols, 2006). He found that it was slow and less data-efficient than an on-policy agent using the same data. Another problem is that in order to prove convergence, the episode lengths must be

bounded, which limits the use of the algorithm to episodic problems.

Least Squares Temporal Difference learning (Bradtke and Barto, 1996; Boyan, 2002) and Least Squares Policy Improvement(Lagoudakis and Parr, 2003) are reinforcement learning techniques that directly estimate the Bellman Residual error. These techniques are stable for off-policy learning. The main drawback of LSTD and LSPI is that they are second-order method which use $O(n^2)$ memory and computation per-time-step, where n is the number of features. Incremental methods based on these, such as iLSTD(Geramifard et al., 2006), improve the per computation time to $O(n)$ but not the memory requirements. For large problems like solving Go, filtering out harmful websites, and others, it may not be possible to use this algorithm given computer memory constraints.

GTD (Sutton et al., 2009b), GTD2, and TDC (Sutton et al., 2009a) are a family of gradient descent methods for off-policy temporal-difference learning with linear function approximation. These algorithms satisfy all four of the desired traits. They are stable for learning on off-policy linear function approximation problems, and their computations per time step and memory used are linear in the number of features. These algorithms have been tested on Baird's counterexample and found to converge. They have also been tested on a variety of small function approximation problems, and one large problem with 63,000 features (Sutton et al., 2009b). All of these experiments were on-policy prediction problems. The best learner of the group is TDC (temporal difference learning with correction), which performed similarly to TD on the problems they were compared on (Sutton et al., 2009b).

GQ(λ) (Maei and Sutton, 2010) is an algorithm that extends TDC for use with action values, eligibility traces, and learning of temporally extended actions. Theoretically it satisfies all four of the desired properties, although no experiments using it have been published before this thesis. Greedy-GQ (Maei et al., 2010) is an extension of GQ(λ) to the control case. No empirical work with this algorithm has been previously published either, so we don't know much about the practical application of these algorithms.

2.9 Summary

In this chapter, we explained reinforcement learning, function approximation, and temporal difference learning. We explained the difference between the on-policy and off-policy problems. We pointed out that the original temporal-difference learning

methods have a problem. Their value function estimates may diverge to infinity using function approximation in an off-policy setting. We discussed several newer reinforcement learning methods that are stable under function approximation, and their drawbacks.

The one method which we know little about is Maei & Sutton's GQ(λ) algorithm, published in 2010, which has not yet been empirically evaluated. GQ(λ) is exciting because it combines four desirable traits: off-policy updates, temporal-difference learning, linear function approximation, and $O(n)$ complexity; which will allow for use on large-scale real-world real time applications. This thesis aims to close the gap in the literature by using the GQ(λ) algorithm in both control and prediction experiments.

Chapter 3

The GQ(λ) Algorithm

GQ(λ) is part of a family of gradient-descent based temporal-difference learning methods. In this chapter, we will explain some theory behind GQ(λ), discuss its implementation, and describe how to compute its objective function exactly.

3.1 Theory

In gradient-descent methods, the updates to the weight vector θ are proportional to the gradient of the objective function the algorithm is trying to minimize. The objective function which most TD methods are trying to minimize is based on the Bellman equation. The action-value function Q^{π} satisfies the Bellman equation exactly (Sutton et al., 2009a):

$$\begin{aligned} Q^{\pi} &= R + \gamma P_{\pi} Q \\ &\stackrel{\text{def}}{=} T_{\pi} Q, \end{aligned}$$

where R is the vector with components $\mathbb{E}[r_{t+1}|s_t = s, a_t = a]$, and P_{π} is a matrix of the state-action to state-action transition probabilities using policy π, and T_{π} is known as the *Bellman operator* with respect to the policy π (Sutton et al., 2009a). It might be natural to choose the *mean-square Bellman error*, or MSBE, with respect to the θ weights as the objective to minimize:

$$MSBE(\theta) = \| Q_{\theta} - T_{\pi} Q_{\theta} \|_D^2 \tag{3.1}$$

where Q_{θ} is the state-action value function with respect to the linear weights

$\theta \in \mathbb{R}^n$ from equation 2.4 and the norm is calculated with respect to the distribution of the data ($\| x \|_D^2 = x^\top D x$).

In the function approximation case, the value function may not be representable by the feature space. Therefore, to find the gradient of the value function updates in the linear function approximation case GQ(λ) does gradient descent in a different objective, called the *mean-square projected Bellman error*, or MSPBE, which uses a projection operator Π to project a value function to the nearest value function representable by the function approximation:

$$MSPBE(\theta) = \| Q_\theta - \Pi T_\pi Q_\theta \|_D^2 \tag{3.2}$$

From equations 10, 13 and 19 in Maei & Sutton's GQ paper we see that the lambda weighted version of equation 3.2 can be written in terms of statistical expectations as:

$$MSPBE(\theta) = \mathbb{E}_D[\delta_\theta e]^\top \mathbb{E}_D\left[\phi\phi^\top\right]^{-1} \mathbb{E}_D[\delta_\theta e] \tag{3.3}$$

where

$$\delta_\theta = r_{t+1} + \gamma\theta^\top\overline{\phi}(s_{t+1}) - \theta^\top\phi(s_t, a_t), \tag{3.4}$$

$$\overline{\phi}(s_{t+1}) = \sum_a \pi(s_{t+1}, a)\phi(s_{t+1}, a) \tag{3.5}$$

$$e = \phi(s_t, a_t) + \gamma\lambda\rho(s_t, a_t)e(s_{t-1}, a_{t-1}) \tag{3.6}$$

where r_{t+1} is the reward given at state s_t when taking action a_t and transitioning to s_{t+1} and π is the target policy. $\mathbb{E}_D$ refers to the expected value with respect to the behavior distribution.

The fact that the expectations in equation 3.3 are with respect to the behavior distribution is an important detail. The objective function that GQ(λ) approximates is with respect to the behavior distribution, not the distribution of the target policy. This means that GQ(λ) will make estimates based on the distribution of the data it is using to learn. If this distribution is different than the distribution that would come from an on-policy distribution, then the value function estimates will be different. This differs from the TD(λ) with importance sampling algorithm, which makes corrections to learn the same value function estimates as would be learned on-policy.

GQ$(b, \pi, \gamma, \lambda, \alpha_\theta, \alpha_w, n)$
Initialize vector θ arbitrarily, and vectors w and e to 0, all of size n
s = current state of environment
Repeat at each time step of interaction with the environment:
Select action a according to behavior-policy distribution $b(s,.)$
Send a to the environment; receive next reward r and next state s'
$\phi = \phi(s, a)$
$\overline{\phi'} = \sum_a \pi(s', a)\phi(s', a)$
$\rho = \frac{\pi(s,a)}{b(s,a)}$
$e = \phi + \rho\gamma\lambda e$
$\delta = r + \gamma\theta^\top\overline{\phi}' - \theta^\top\phi$
$\theta = \theta + \alpha_\theta[\delta e - \gamma(1-\lambda)(w^\top e)\overline{\phi'}]$
$w = w + \alpha_w[\delta e - (w^\top\phi)\phi)]$
$s = s'$

Figure 3.1: GQ(λ) Algorithm with Linear Function Approximation

We will explore the consequences of this difference in Chapter 6.

3.2 Algorithm

The algorithm for GQ(λ) is shown in Figure 3.1. GQ(λ) uses a second set of weights, $w \in \mathbb{R}^n$, to approximate the projection. The second set of weights forms a quasi-stationary estimate, which avoids the need for two separate independent samples to approximate each of the two $delta_\theta$ terms in the objective function 3.3. The w weights are used to incrementally approximate two of the three expectations from equation 3.3:

$$w \approx \mathbb{E}_D\left[\phi\phi^\top\right]^{-1}\mathbb{E}_D[\delta_\theta\phi] \tag{3.7}$$

GQ(λ) learns to estimate policy π, while b is the policy that is used or was used to choose actions. ρ is the ratio between the probability that the target policy π would take action a in state s and the probability that the behavior policy b would take the same action:

$$\rho(s, a) = \frac{\pi(s, a)}{b(s, a)} \tag{3.8}$$

If GQ(λ) is used to learn about data generated by an unknown policy, then the probabilities for the behavior policy can be sampled from the data (*i.e.*, make a first

pass thru the data and count the action choices at each state, or in an online setting start with equal action probabilities and update the action choice probabilities incrementally at each state). If GQ(λ) is used to learn a control problem, then the greedy policy should be used for π. π will equal 1 if the state-action that leads to the highest Q_θ value is picked, and 0 otherwise, except in the case where there are multiple actions that have the highest Q_θ value. In this case, the probability will be the reciprocal of the number of actions tied for the highest Q_θ value.

An element of the GQ(λ) algorithm that is different from traditional methods is that in the update it uses the expectation over all actions available at the following state instead of just the action taken. That is, it uses:

$$\overline{\phi}_{t+1} = \sum_a \pi(s_{t+1}, a)\phi(s_{t+1}, a)$$

instead of $\phi(s_{t+1}, a_{t+1})$. This helps to reduce variance in the updates. The algorithm *Expected Sarsa* is a modification of Sarsa that also uses expected updates. In the Seijen et al. (2009) paper, Expected Sarsa is proven to have lower variance than Sarsa and is empirically shown to perform better.

3.3 Calculating the TD Fixed Point

Over time, GQ(λ) converges to the *TD fixed point*, as long as the following conditions are met (Maei and Sutton, 2010):

- $\alpha_\theta, \alpha_w > 0$
- $\sum_{t=0}^{\infty} \alpha_{\theta,t} = \sum_{t=o}^{\infty} \alpha_{w,t} = \infty$
- $\sum_{t=0}^{\infty} \alpha_{\theta,t}^2, \sum_{t=o}^{\infty} \alpha_{w,t}^2 < \infty$
- $\frac{\alpha_{\theta,t}}{\alpha_{w,t}} \to 0$ as $t \to \infty$
- ϕ_t is a Markov process with a unique invariant distribution
- ϕ_t, e_t, and r_t sequences have uniformly bounded second moments
- A (described below) and $C = \mathbb{E}_b\left[\phi_t \phi_t^\top\right]$ are non-singular matrices

The TD fixed point is used to calculate learning performance for the experiments in the toy domains in this thesis. Thus it is prudent to explain how the fixed point is

calculated in order to reproduce those experiments. The TD fixed point is achieved when

$$\mathbb{E}_D[\delta_\theta(s,a,s')e(s,a)] = 0$$

because then the expectation in equation 3.3 will be 0 as well. This section explains how to translate this expectation into matrix form in order to easily compute it. The rest of this section was developed through correspondence with Hamid Maei.

To translate the expectation above into Matrix form we first expand it:

$$\begin{aligned}
\mathbb{E}_D\big[\delta(s,a,s')e(s,a)\big] &= \sum_{s,a} d_b(s,a)e(s,a)\mathbb{E}\big[\delta(s,a,s')|s,a\big] \\
&= \sum_{s,a} d_b(s,a)e(s,a)\mathbb{E}\big[r(s,a,s')|s,a\big] \\
&\quad + \sum_{s,a} d_b(s,a)e(s,a)\mathbb{E}\Big[(\gamma\overline{\phi}(s') - \phi(s,a))^\top|s,a\Big]\,\theta^* \\
&= x + A\theta^*
\end{aligned}$$

where

$$x = \sum_{s,a} d_b(s,a)e(s,a)\mathbb{E}\big[r(s,a,s')|s,a\big]$$

is a vector and

$$A = \sum_{s,a} d_b(s,a)e(s,a)\mathbb{E}\Big[(\gamma\overline{\phi}(s') - \phi(s,a))^\top|s,a\Big]$$

is a matrix. Thus the TD fixed point is $\theta^* = -A^{-1}x$.

To put these into matrix form, we define a matrix Φ whose number of rows is $N = |\mathcal{S} \times \mathcal{A}|$ and number of columns, n, is the size of the feature vector and whose rows are $\phi(s,a)^\top$. Note that $\mathcal{S}$ is the set of all states and $\mathcal{A}$ is the set of all actions. We set R to be the reward transition vector of size N which holds the expected rewards given at each state-action pair, $R(s,a) = \mathbb{E}[r(s,a,s')|s,a]$. D is the behavior distribution over state action pairs, a diagonal matrix with diagonal elements of $d_b(s,a)$. Lastly, we define an eligibility matrix E of size $N \times n$ with each row made up by the vector $e(s,a)^\top$. Then determined from a recursive equation:

$$x = \Phi^\top D R$$

And to calculate A

$$\begin{aligned}
A & \\
= & \sum_{s,a} d_b(s,a) e(s,a) \mathbb{E}\Big[(\gamma \overline{\phi}(s') - \phi(s,a))^\top | s,a \Big] \\
= & \sum_{s,a} d_b(s,a) e(s,a) \mathbb{E}\Bigg[(\gamma \sum_{a'} \pi(a'|s') \phi(s',a'))^\top | s,a \Bigg] \\
& - \sum_{s,a} d_b(s,a) e(s,a) \phi(s,a)^\top \\
= & \sum_{s,a} d_b(s,a) \phi(s,a) \left(\gamma \sum_{s',a'} \mathbb{P}\big(s'|s,a\big)\, \pi(s',a') \phi(s',a') \right)^\top - E^\top D \Phi \\
= & E^\top D (\gamma P_\pi) \Phi - E^\top D \Phi \\
= & E^\top D (\gamma P_\pi - I) \Phi
\end{aligned}$$

where P_π is the state/action transition matrix for the target policy. Its elements $((s,a),(s',a'))$ are constructed according to $\mathbb{P}(s'|s,a)\,\pi(s',a')$.

Thus the TD fixed point θ^* solution is

$$\begin{aligned}
\theta^* &= -A^{-1} x && (3.9) \\
&= (\Phi^\top D (I - \gamma P_\pi) \Phi)^{-1} \Phi^\top D \overline{r} && (3.10)
\end{aligned}$$

Chapter 4

Effect of the Disparity between Behavior and Target Policies

As a first step in the exploration of off-policy learning with GQ(λ), we explore the effect of the disparity between the behavior and target policy probabilities on learning performance. This will give us some idea of how well we can expect to learn when the target and behavior policies are far apart, and potentially yield some heuristics to follow in future experiments.

We use the experiments in this chapter to empirically evaluate the following questions about off-policy learning on a prediction setting:

1. *Is it best to learn a prediction problem on-policy?*
2. *How does off-policy learning performance change as the disparity between the target and the behavior policies grows?*

These questions will not be answered definitively for all off-policy problems. Instead, I will give modest answers to these questions from the results in these experiments and develop some intuitions about how to deal with off-policy problems which may be applicable to larger domains.

4.1 The Off-Policy Distance

In order to more easily discuss and graph the difference in probabilities between policies, I define the *Off-Policy Distance*, OPD.

$$OPD(\pi, b) = \frac{1}{|S||A|} \sum_{s,a} |\pi(s,a) - b(s,a)| \tag{4.1}$$

where π is the target policy and b is the behavior policy. The OPD is best described as a percentage.

4.2 The Random Walk Problem

For all the experiments in this chapter, I use a simple random walk domain with *tabular features* (one distinct active feature per state). I chose a simple domain for this experiment, with no function approximation, and only two actions in order to concentrate on the difference between the policies. Because its a tabular problem, and not a function approximation problem there is no use for the second set of weights in GQ(λ). So the algorithm I use is GQ(λ)with α_w set to zero, which zeros out the second set of weights. This algorithm reduces to Expected Sarsa with eligibility traces, or *Expected Sarsa(λ)*.

Figure 4.1: 5-State 2 exit random walk example from Sutton and Barto (1998), pg 139, used with permission

The MDP is the 5-state random walk example from Sutton and Barto (1998), depicted in (Figure 4.1). It is an episodic problem, with two exits. At the beginning of each episode, the agent starts at the center state (C). Every transition gives zero reward except the terminal transition on the right, which gives a reward of 1. γ is 1, so the value of each state-action is simply the probability of an exit occurring on the right side given that state and action. The feature representation uses tabular features, which means for example that state B is represented by the vector $\phi(B) = (0, 1, 0, 0, 0)^\top$.

There are only two actions, LEFT and RIGHT, so it is easy to compare action probabilities. This is a deterministic MDP, which means that when an action is taken from a state it always results in the same state transition. In this MDP, the

off-policy distance reduces to taking the absolute value of the difference between the action RIGHT (or LEFT) probabilities of the behavior and target policies.

I ran the experiments using 25 different parameter settings—combinations of α_θ from 2^{-4} to 2^0 in increments of powers of 2, and λ from the set $[0, 0.25, 0.5, 0.75, 0.9]$. The θ values were set to 0.5 initially.

4.3 Experiment 1: Varying the Target Policy

It is common place in real-world problems to have a trajectory of previously gathered data from which to learn on. For example, a factory may want to analyze its design process to improve production times, or a football coach may want to analyze plays and results from previous years in order to win more games this year. It is also common that changing behaviors in order to have new trajectories to learn from is not possible. The factory may be reluctant to experimentally change their pipeline for risk of sacrificing production for a period of time. The football coach may not want to run experimental plays for fear of looking like a fool, and losing the confidence of his team.

When there exists precompiled data, and we can not learn from new data, we can consider the behavior policy to be fixed. In this case it would be useful to determine how well we can expect to learn a certain target policy on this data, if at all. This is the prediction case where we know the target policy we want to learn ahead of time. In this section we study learning performance on various target policies given a fixed behavior policy.

The behavior policy was fixed to the equal random policy and the algorithm was run on the random walk MDP to learn 10 different target policies. The target policy percentages were from 90/10 [LEFT/RIGHT] to 0/100 in 10% increments. Throughout the rest of the chapter, the target polices will be described by the probability of action RIGHT with prefix T. The behavior policies will be described similarly but with prefix B.

As a performance measure of each algorithm after each episode, I used the root mean squared difference between our current value estimates and Q_θ:

$$RMSE(\theta) = \| Q^*_\theta - Q_\theta \|_D = \sqrt{\sum_{s,a} d_b(s,a)(Q^*_\theta(s,a) - Q_\theta(s,a))^2} \qquad (4.2)$$

where $Q^*_\theta(s)$ is calculated from the TD fixed point using equation 3.9 and equation 2.4, and $d_b(s,a)$ is the distribution of the behavior policy when at state s and taking action a. Note that this performance measure takes a weighted average of the features, weighted by the behavior policy's state-action distribution.

I performed 100 runs on the MDP for 2000 time steps each run. All parameter settings and target policies were learned with the exact same action choices. For each run the error was calculated (per equation 4.2) at each time step. Values were averaged together every 10 steps, and recorded. The errors from all 100 runs were then averaged together. The learning curves for the different λ settings with α_θ set to 1 using the target policy T50 are shown in Figure 4.2. I call this the B50-T50 group, because the behavior policy is 50/50 and thus has a 50% action RIGHT probability and the target policy is also 50/50 thus has a 50% action RIGHT probability.

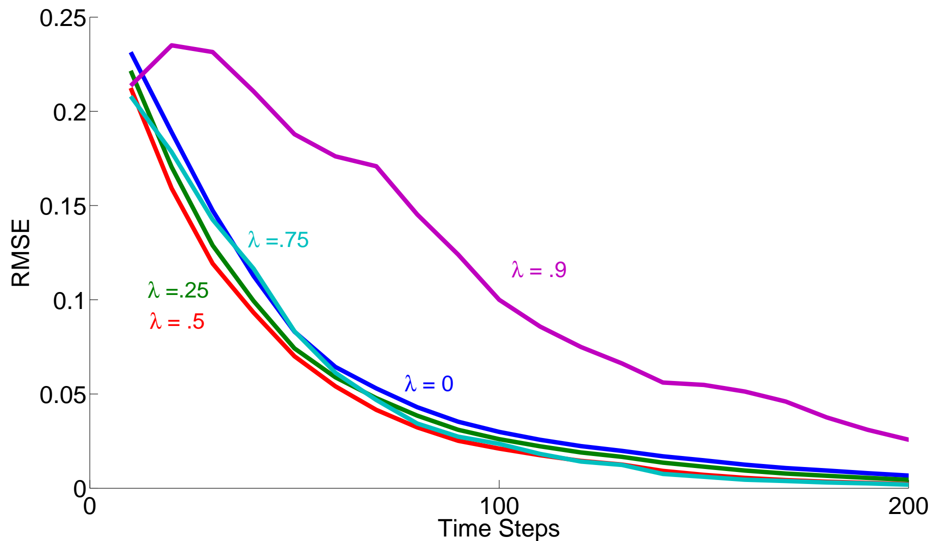


Figure 4.2: Learning curves for various λ with $\alpha = 1$ for the B50-T50 group

In order to easily compare learning performance between groups and between settings within a group it is useful to distill each learning curve down to a single number that represents the performance of the learner. One way to do this is to determine the speed with which the learner reaches a near perfect guess of the answer. I set the measure of convergence to be the time it takes for the learner to achieve an error of less than one percent of the initial guess (.005 = 1% of the starting θ value of 0.5 in this MDP). The curves in Figure 4.2 have the following convergence times (to the nearest 10 steps):

$\lambda =$	0	.25	.5	.75	.9
Time Steps	220	200	170	160	290

$\lambda = .75$ performed the best by this measure. However, convergence speed fails to account for the error accumulated along the way to the point of convergence. For example, in the case of the learning curves for $\lambda = .75$ and $\lambda = .5$ in Figure 4.2, learning with $\lambda = .5$ showed better performance than learning with $\lambda = .75$ for most of the time steps even though the convergence speed was slightly slower.

Another consideration when choosing a performance measure is that TD-based algorithms will only converge to the fixed point when the step size is continually reduced to 0. If the step size remains constant, as in this experiment, then the predictions will tend to oscillate around the fixed point values forever. One way to capture both the rate of convergence and the final asymptotic error into a single number, is to integrate the error under the curve over a fixed number of time steps. It is important to realize that the duration chosen will affect the result, because the slower learners that converge to a smaller error will tend to look better when more time steps are used. I chose to integrate over 2000 time steps, because that was sufficient time to capture the convergence points of all the various experiments in this chapter. I then divided this integral by the number of time steps to achieve an average error value per time step which is used in subsequent plots.

$$Error = 1/2000 \sum_{t=1}^{2000} RMSE(\theta_t) \tag{4.3}$$

A *parameter study* is a graph which generally compares error or performance of an algorithm using different parameter settings. A parameter study over all α and λ values for learning on T50 (the on-policy setting) is shown in Figure 4.3. The error is measured in the y axis. Lower values are better. It is clear that setting α to 1 results in the best performance. The plot of $\alpha = 1$ shows the learning curves from Figure 4.2 each distilled down to a single scalar value: the RMSE averaged over the first 2000 time steps (equation 4.3). By this measure, setting $\lambda = 0.5$ and $\alpha = 1$ gave the best performance.

Parameter studies were performed to find the best parameters for learning all 10 target policies. The learning curves using the best parameter settings are shown in Figure 4.4. Here we see our first comparison of learning performance over different target policies. From this graph, the on-policy group, B50-T50, performs the best at the beginning of learning. The extreme policies T90 and T100, achieve lower

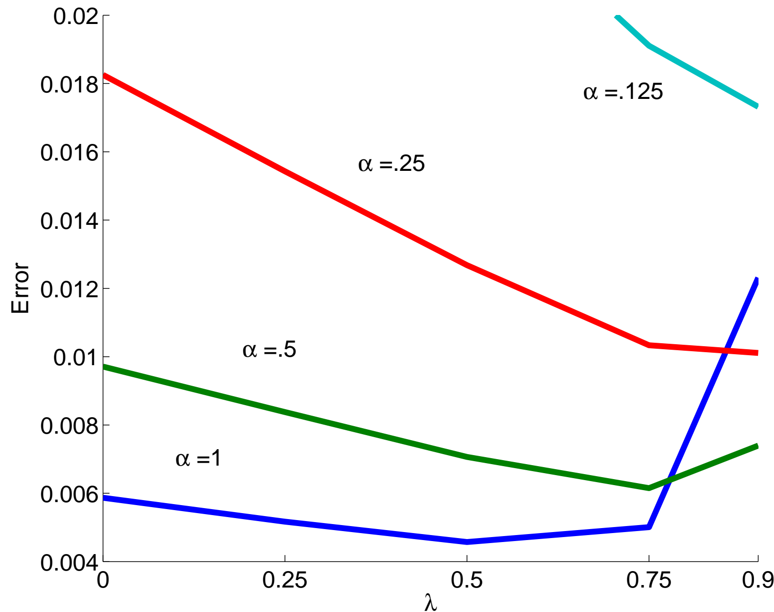


Figure 4.3: Parameter Study over all α and λ values on the 50/50 target policy. The line for $\alpha = .0625$ lies above the boundaries of this graph.

error after about 70 time steps.

This is because the targets with extreme percentages are easier to learn on this MDP. A 50/50 policy is harder to learn because it spends more time wandering around in the state space, whereas a 10/90 policy will generally only make a few moves before exiting since it is so heavily biased towards one side. Wandering around more increases the variance in predicted values because some episodes will be very long and some very short. Since calculating RMSE is similar to measuring variance, it follows that a lower variance problem would have better performance.

Now if we reduce these learning curves down to the average RMSE value (equation 4.3) and plot them on a line where the x-axis is the $\pi(\text{RIGHT})$ of each target policy, we can compare performance of the random behavior to targets near and far (Figure 4.5). From this figure it is clear that the learning agent performed best when the policy to be learned matched the behavior policy, T50, and performance seems to get worse as the target policy becomes more different, but then gets slightly better at the extremes due to the variance of the problems decreasing.

In this MDP, the learning performance is good across all target policies when using an equal random behavior policy distribution. Notice how small the scale is for the error in Figure 4.5. Notice how all of the target policies were learned

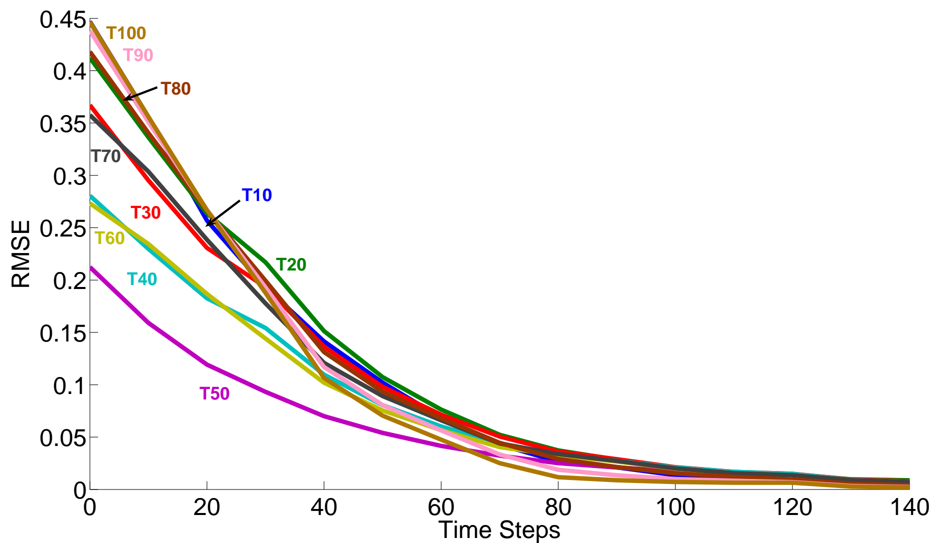


Figure 4.4: Best learning curves on every target policy

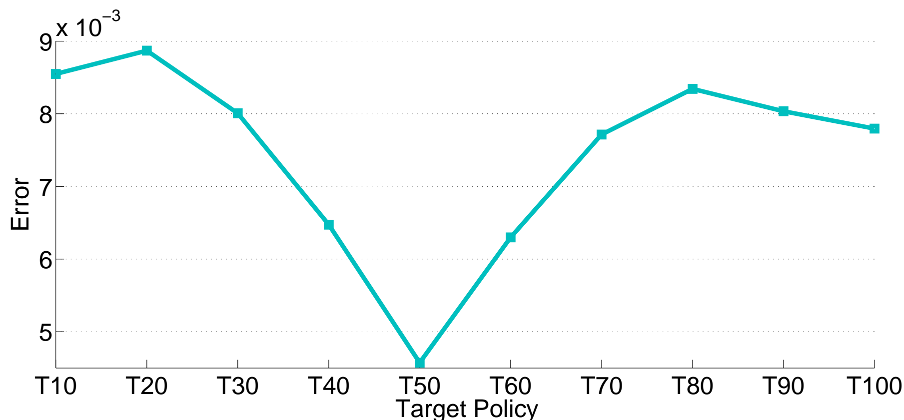


Figure 4.5: Performance over all target policies

nearly perfectly in less than 200 time steps in Figure 4.4. This suggests that with a trajectory from an equal random behavior policy, any target policy is easy to learn on this small MDP. It is important to remember that this is an empirical result on a simple random walk problem, and not a proof. These results may not hold up in more complex environments.

4.4 Experiment 2: Varying the Behavior Policy

When given a specific task to learn we may ask what policy we should use to learn this task most quickly? If the task is a prediction task we may assume that it

would be best to learn using the same policy as the policy we want to predict, the on-policy case. However, this may not be the best policy to use. In this section we will compare the learning-performance of agents using different behavior policies to try to learn the same target policy.

I use the same MDP and parameters as described in section 4.2. The target policy we will try to learn is the equal random policy, T50, and we will learn this policy using 7 different behavior policies ranging from B20 to B80 in 10% increments.

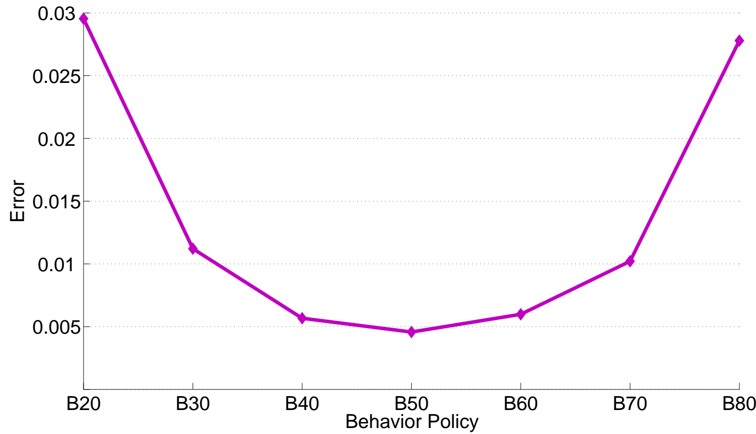


Figure 4.6: Performance using various behavior policies over the equal random target policy

Parameter studies were performed and the best parameters were chosen from each target policy group. The performance measure, the errors in learning calculated using equation 4.3, are plotted in Figure 4.6. The results show that learning the equal random target policy using the equal random behavior policy leads to the best performance. Using a behavior policy that differs by only 10% or 20% from the target policy is not much worse. Only the behavior policies at the extremes show markedly worse performance. They perform 5 to 6 times worse than the equal random behavior policy by this performance measure.

4.5 Experiment 3: Varying Both the Target and Behavior Policies

In previous experiments we focused on learning performance with respect to an equal random target policy (Experiment 1) and an equal random behavior policy (Experiment 2). Equal random policies have the most even behavior distributions

and thus learning with or about them may have different characteristics than learning *off-balance policies*, policies whose actions are not distributed equally randomly. In this section, as in past ones, I may refer to the behavior policies being used in each problem with a prefix B and the action RIGHT percentage, (*i.e.*, B60 refers to the 40/60 [LEFT/RIGHT] behavior policy). For the target policies being learned I will do the same but with a prefix T. All of the problems learned using a specific behavior policy or about a specific target policy will be referred to as a *group*. Specific behavior and target policy combinations will be called *groupings*.

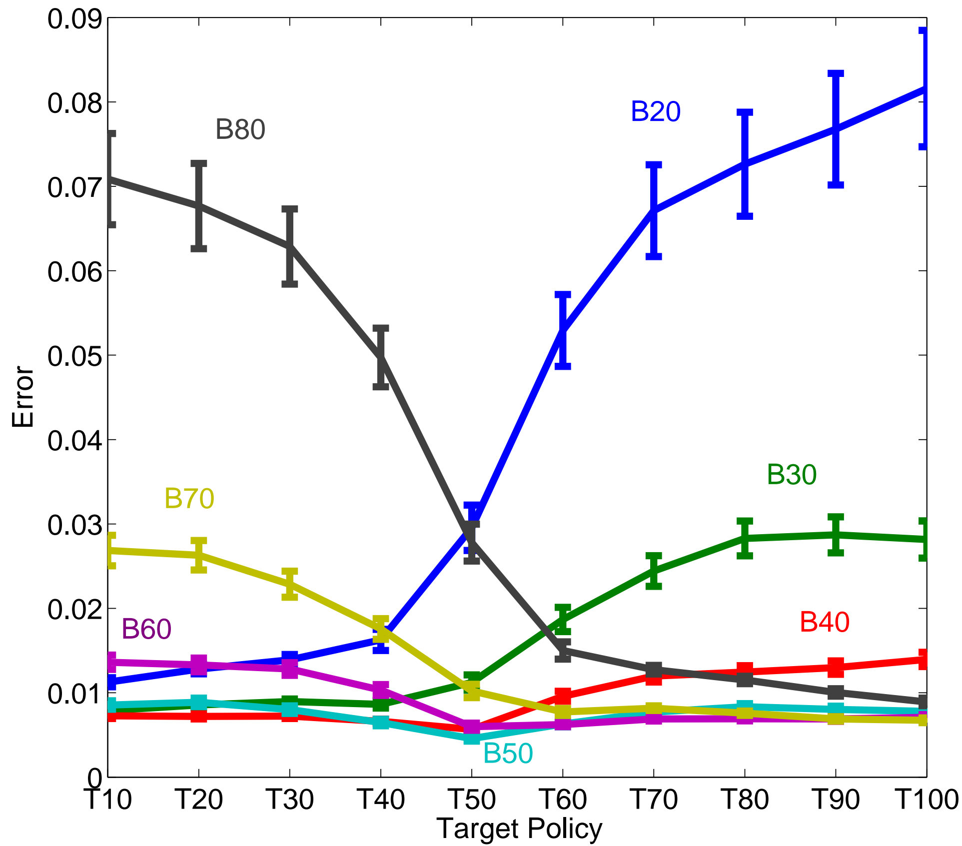
For this experiment, I used the same MDP and parameter settings as in section 4.2. I took the 10 target policies used in Experiment 1 and learned each one using the 7 behavior policies that were used in Experiment 2, for a total of 70 groupings. I evaluated each grouping using the same 25 combinations of parameter settings on the same problem as in the other experiments.

Each learning problem was run 100 times for 2000 time steps and the results were averaged together. The performance measure used is the same as in the previous experiments (equation 4.3). For all 70 groupings, the best α value was 1 (the highest α value used). The best λ parameters varied between 0.25 and 0.75.

The errors using each behavior to learn over different target policies are plotted in Figure 4.7. These graphs include the line from Figure 4.5 plotted in different scales. Error bars are displayed in the graph to show the statistical significance of each data point. An obvious result is that errors were highest when the extreme behaviors, B20 and B80, were used on the policies they are most distant from.

The first important thing to notice is that when a learning agent's behavior policy matched the target policy (the on-policy case), it resulted in the least error only once—in the B50-T50 grouping. The agents using B40 and B60 statistically tied with the agent using B50 on their respective on-policy target policy groupings. The other four on-policy groupings had more error than two or more of the off-policy learners on their respective target policies.

Another interesting thing to note is that using B40 and B60 yielded lower error than B50 on the more extreme target policies on their respective sides of the 50% demarker. Specifically, using B40 showed better performance than using B50 on learning T30, T20, and T10. Using B60 showed better performance than using B50 on learning all of the target policies groups from T70 to T100. Using these behaviors also resulted in less error than using an on-policy behavior on the respective target policy groupings.



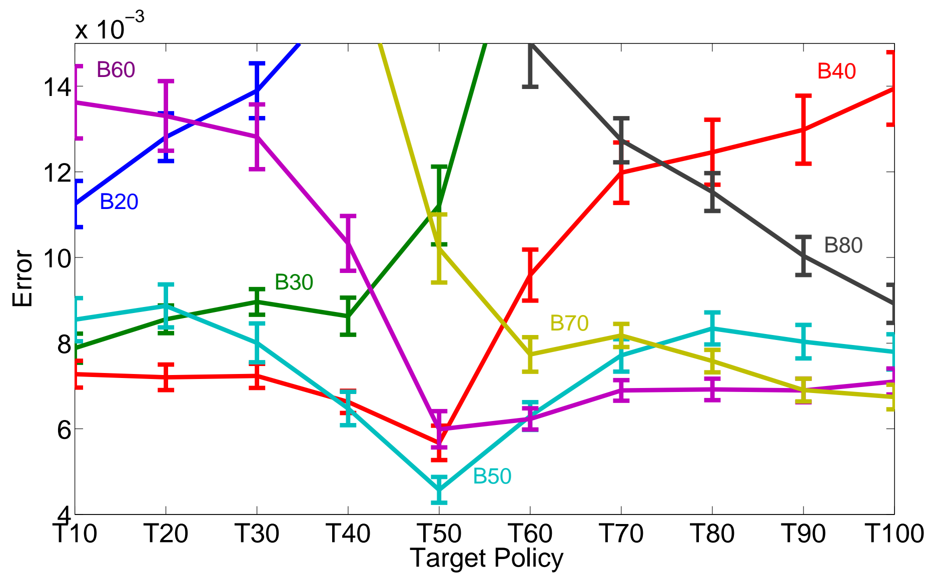


Figure 4.7: Performance using each behavior policy (lines) over different target policies. Bottom graph is a zoomed in view.

Lastly, notice the large jump in error that occurred for every single off-balance behavior policy when learning target policies on the other side of the 50/50 line. For instance using B60, there was less error in learning T100 than learning T40 even though T40 is twice as close in terms of the off-policy distance.

These are interesting and somewhat unexpected results. The first result shows that it is not necessarily best to learn on-policy on a prediction problem. This may be related to the state-action distributions of each behavior policy. The policies with percentages close to 50/50 will have the most even state-action distributions, whereas the policies farther away will rarely visit some states. The table below shows the extreme difference in state-action distributions between behavior policies. The first distribution listed is that of the 50/50 behavior (B50), which samples state-actions to the left and right of center an equal number of times. The second is of the 20/80 (LEFT/RIGHT) policy (B80):

B50 Distribution	A	B	C	D	E
Left	0.0556	0.1111	0.1667	0.1111	0.0556
Right	0.0556	0.1111	0.1667	0.1111	0.0556

B80 Distribution	A	B	C	D	E
Left	0.0032	0.0159	0.0667	0.0635	0.0508
Right	0.0127	0.0635	0.2667	0.2540	0.2032

The B80 distributions are heavily skewed towards the rightmost states. In fact, this behavior visits State A less than 1.6% of the time. If a policy rarely visits a state then it won't be able to learn much about it, and learning performance will suffer. This may be why on this problem it was better to use B50, B60, and B70, rather than B80 to learn T80, because each of those behaviors will have a more even state action distribution. Remember that the error measure being used, the RMSE, weights each error in the value function by the state distribution, so states that are visited less frequently will have less of an effect on the overall error. However this weighting is not enough to overcome the performance on problems that have more even state distributions, like B40, B50, and B60.

The second result of interest shows that an even distribution is not the only thing that matters, a combination of being close in off-policy distance as well as more evenly distributed than the target policy shows the best learning performance in general on this MDP.

The last result of interest (that performance suffered sharply across the 50/50 line) may show that it is important that the behavior policy chooses the action

favored by the target policy more or equally as often as the other actions, otherwise performance will suffer greatly. It would be interesting to check this result on other MDPs with larger action spaces.

4.6 Change in Error as the Off-Policy Distance Grows

In this section, I analyze the data to answer two new questions regarding the change in error over each policy as the off-policy distance grows. The first question is:

How does off-policy learning performance change as the target policy differs further from each behavior policy on the random-walk problem?

To help answer this, I developed an error measure which I call the increase in error with respect to the behavior policy (IE_B):

$$IE_B(B,T) = Error(B,T) - Error(B,B) \tag{4.4}$$

where T and B are the target and behavior policies for each group and Error(B,B) means the Error when the target policy matches the behavior policy (the on-policy case).

I calculated the IE_B over each of the behavior policies from the results compiled in experiment 3. Data points that had the same off-policy distance for each behavior were averaged together. For example, T60 and T40 both have an off-policy distance of 10 from B50, so these would be averaged together. Each line plotted in Figure 4.8 represents a different behavior policy and the x-axis denotes the off-policy distance from each behavior policy to the various target policies being estimated. Notice that some plotted points are less than zero which indicates that error *decreased* by learning off-policy versus on-policy in these cases.

From this figure we see that performance was similar within a 20% difference in behavior and target policy distributions. As the off-policy distance grew, performance decayed logarithmically to an asymptotic error level. The degree of this error is related to how evenly distributed each behavior policy is, with more even distributions displaying lower asymptotic error. When learning off-policy, it was best to learn target policies that were within a 20% difference in action selection probabilities. Unless the precollected data was well distributed amongst all state-action pairs in which case learning performance was adequate on any target policy.

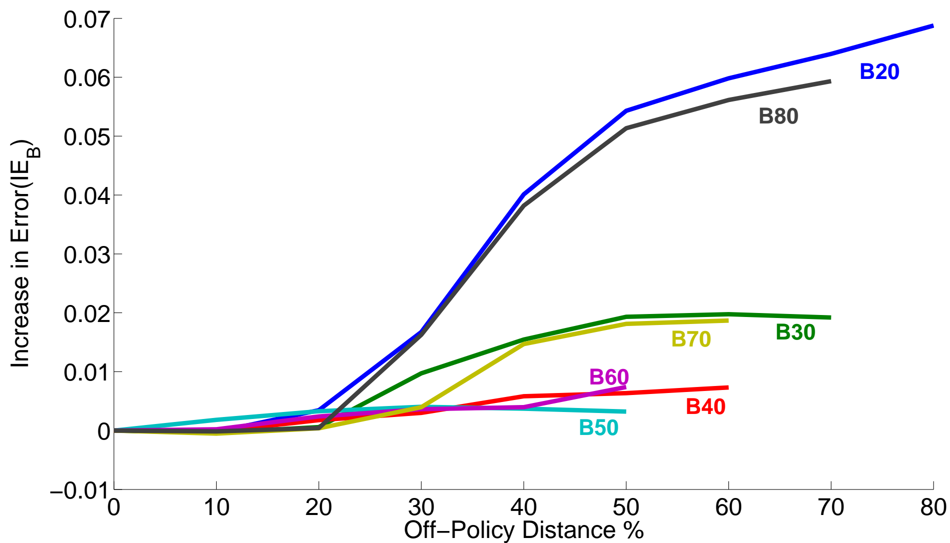


Figure 4.8: Performance as the target policy differs further from each behavior policy

This last result may only be relevant for small problems like this one. If the state space is large and reaching the goal states requires a detailed sequence of actions, then learning over an equal random distribution of state-actions will be very slow. Another complication with this analysis is that as the policies become more extreme, the problems become easier to learn. This effect may be what leads to the flattening out of each graph at the tail ends. In future work, it would be useful to try to account for this effect somehow.

The next question I answer is:

How does off-policy learning performance change as the behavior policy differs further from each target policy on the random-walk problem?

To help determine the answer to this question, I modified the error measure slightly to be the increase in error with respect to the target policy (IE_T):

$$IE_T(B,T) = Error(B,T) - Error(T,T) \tag{4.5}$$

The IE_T for each target policy with respect to the off-policy distance from each behavior policy is plotted in Figure 4.9. Behavior policies with the same off-policy distance from each target policy were averaged together. Each line represents a different target policy and the x-axis denotes the off-policy distance between each target policy and the various behaviors. The three target policies that did not have

on-policy groupings used the closest grouping as their baseline (*i.e.*, T100 used the B80-T80 result in the error calculation). In this figure we see that performance does not change dramatically as the behavior policy gets further from the target policy until it reaches the extremes of the behavior policies we tested over. This is partially due to the fact that as the target policies become more extreme, they also become easier to learn.

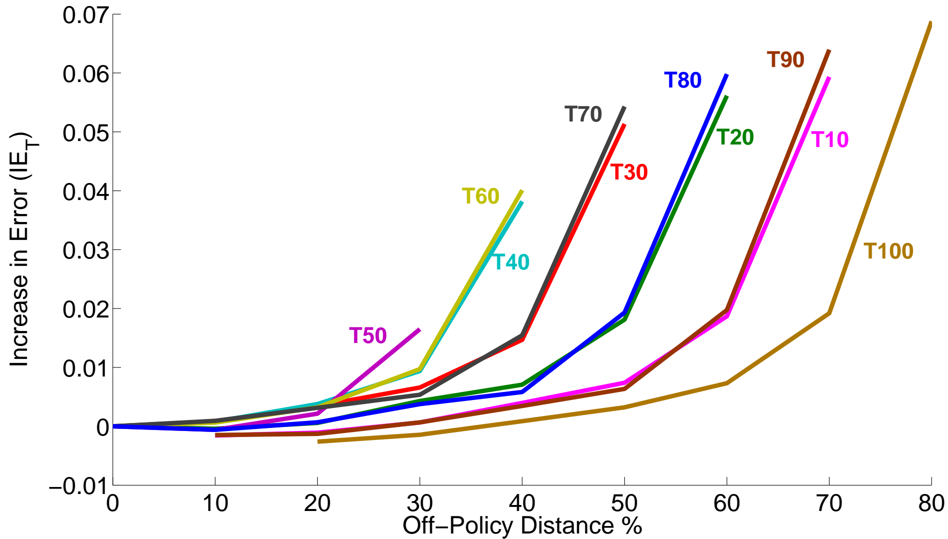


Figure 4.9: Performance as the behavior policy differs further from each target policy

4.7 Conclusions

I draw three heuristics about off-policy learning from the results in Experiment 3. First, the distribution over the state space seems to be a very important factor. Given the same off-policy distance, it is better to learn using a behavior policy that is more evenly distributed than more narrowly focused. Second, it is important to use a behavior policy that tends to choose actions favored by the target policy more or equally as often as the other actions. There was a large loss in performance when this was not the case. Third, it was found that it is not necessarily best to learn on-policy. Learning using a policy that is slightly more evenly distributed than the target policy can lead to better performance. Thus the answer to our first question:

Is it best to learn a prediction problem on-policy?

is no, not necessarily.

Comparing learning performance between off-policy problems presents complications because different policies may be more or less difficult to learn due to variance. In order to account for these differences, I averaged together the IE_B over all of the different behaviors used. This way, for example, the performance of B20-T50, which consists of a behavior policy with a highly skewed state-action distribution learning the target policy with the highest variance, is averaged together with the IE_B of B50-T80 which consists of a behavior policy that is the most evenly distributed amongst state-action pairs with a target policy that has relatively low variance, because they both have an OPD of 30%. All of the different combinations should help mute the effects of distribution and variance and isolate the effect of off-policy distance. The result is plotted in Figure 4.10.

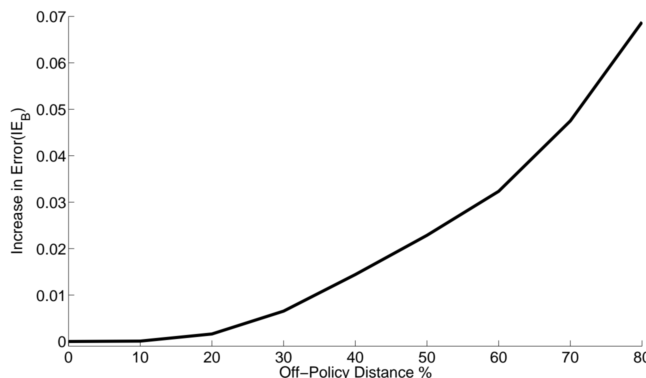


Figure 4.10: Learning performance as a function of the off-policy distance

Looking at the question posed in the introduction:

How does off-policy learning performance change as the disparity between the target and the behavior policies grows?

The simple answer that can be drawn from this plot is that *learning will suffer as the disparity between the behavior and target policies grows.* A more specific answer is that learning will suffer quadratically as the disparity grows.

It is important to remember that the results and conclusions in this chapter were drawn from a simple random walk experiment and do not necessarily hold across all MDPs. However, these results may still be useful for informing behavior policy choices and boundaries on target policies to be learned in future experiments and larger problems.

Chapter 5

Learning Multiple Optimal Policies Simultaneously on a Robot

Reinforcement Learning on artificial problems such as those described in the previous chapter is informative, but in order to progress further in artificial intelligence, it is important to apply algorithms to much larger problems. One example is robotics. A mobile robot could use reinforcement learning to learn to interact with the real-world from its sensorimotor experience. With an off-policy method like GQ(λ) it could learn many different ways to interact with the world at the same time. This chapter describes an experiment that demonstrates learning of multiple optimal policies in parallel using a robot. For this experiment, we will use the Critterbot, a custom-built mobile robot (see Figure A.1), that was expressly designed for reinforcement learning experiments. The Critterbot is described in detail in the appendix.

Whereas the previous experiments were prediction experiments, predicting the value function of a fixed policy, this is a control experiment, learning a policy to maximize reward. For this control experiment, the learning agent will use a slightly modified version of GQ(λ) which could be called *Greedy-GQ(λ)*. This algorithm extends the Greedy-GQ algorithm from Maei et al. (2010) to include eligibility traces. Greedy-GQ was proven to be stable for learning control policies in an off-policy unrestricted linear function approximation setting as long as the behavior policy is stationary, which it is in this experiment. Greedy-GQ(λ) is the same as GQ(λ) with a greedy target policy and with the behavior policy probability always set to 1.0, so we will continue to call it GQ(λ) throughout.

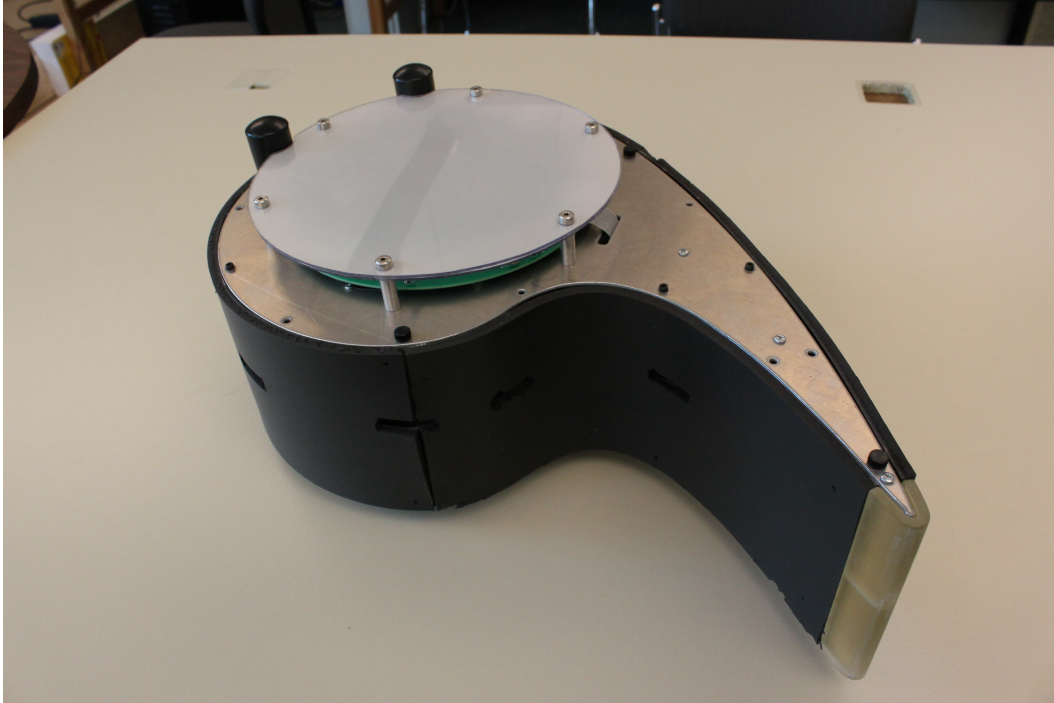


Figure 5.1: The Critterbot

5.1 Objectives

In this experiment, the Critterbot will spin left and right randomly while a computer agent learns how to maximize various sensors. For example, it will learn how to spin to the direction where its Magnetometer X value is maximized and stop at which point it would be facing magnetic north. It will also learn many other sensor maximization policies in parallel.

This experiment has 2 primary objectives:

1. **Demonstrate off-policy learning for control with GQ(λ) in a real-world domain**

   The robotic domain is a much more complex domain than those used in the other experiments in this thesis. The Critterbot is a physical device which must deal with inertia, motor ramp up times, wireless delays, and sensor delays. When a command is given it can take up to half a second for the motor to achieve the stated value depending on the strength of the command and the state of the motors when the action was sent (see Appendix A.2). When a stop action is given it can take up to 300 ms for the robot to actually come to a stop. Sensor values are reported in their raw form and are always

changing even when the robot is sitting still. One sensor, the beacon light turns on at half second frequencies so the learning agent should be able to deal with a stochastic reward signal in order to learn how to maximize this sensor.

Another complication is that the robot reports raw sensor value data. None of its sensors directly and accurately show objective information like orientation or location, and I will not convert any of these sensors to a more easily recognizable output. So the robot's representation of the world is not as straightforward as an x,y position and orientation. Function Approximation is necessary in this setting. For all these reasons learning control policies on the robot, even if its only spinning in place, is a complex problem.

2. **Demonstrate the ability of GQ(λ) to learn multiple value functions for control in parallel**

   A robot that can learn many things at once would be able to understand its sensorimotor interactions with the environment quickly. In this experiment the critterbot will learn how to maximize various sensors from random interaction with the world. The ability to learn multiple value functions simultaneously is one of the advantages of learning off-policy.

5.2 Experimental Setup

There were several important choices to be made for this experiment: sensors to maximize, behavior policy, time step duration, speed of movement, feature selection, and feature representation. Each choice and the reasoning behind it is described below.

5.2.1 Sensors to Maximize

In this experiment the agent will learn policies to maximize eight sensor readings simultaneously on the Critterbot. Eight is a large enough number to show off learning of a variety of sensors while still being small enough to allow enough time to repeatedly test each learned policy in order to achieve statistical significance. Most importantly, eight was the maximum number of different GQ(λ) updates that my computer could reliably compute before the next time step (every 100 milliseconds).

The eight sensors I chose to maximize are:

1. Wheel 0 velocity
2. Magnetometer X value
3. Infrared Distance Sensor 0
4. Infrared Distance Sensor 2
5. Infrared Distance Sensor 7
6. Infrared Distance Sensor 9
7. Infrared Beacon Sensor 0
8. Temperature Sensor 0

These sensors were chosen because they represent a wide range of sensors on the critterbot (Appendix A.1). The light sensors were ignored for this experiment because the amount of light in the lab changes over the course of the day (and night), which may have skewed the results of some of the evaluations and thus hinder comparisons.

5.2.2 Behavior Policy

In order to achieve statistically significant evaluation, it is important to have an environment in which experiments can be easily recreated. It can be difficult to automatically return a free-moving robot to the same starting position in order to consistently evaluate each policy. However, if the robot's actions are restricted to spinning only, then it will stay in relatively the same location, and only the orientation will have to be reset to achieve similar starting conditions. This will allow for easier comparison of learned policy performance at different time scales and with different parameters. The action set was simply three actions: [STOP, SPIN LEFT, SPIN RIGHT].

An equal random behavior policy was used because it is a simple method to get an even distribution over the state space when the state-space is a simple loop. The experiments in Chapter 4 suggest that having an even distribution makes learning of various policies easier. The random policy had a bias (50%) toward repeating the same action taken on the previous time step, so that the critterbot would be subjected to lengthier sequences of single actions more often than a policy without repeating bias would provide. These lengthier sequences were necessary to actually

achieve the desired velocities over the short time steps uninterrupted (Figure A.5). A single three and a half hour trajectory of data was used, which yielded 114,000 time steps.

5.2.3 Time Step Duration

An important consideration in deciding what time scale to run this experiment at is precision of movement. The Critterbot should operate at as small a time scale as possible in order to achieve the best policy. If we make the time step or command speed too large, then the robot will often shoot past its desired goal. It will lack the precision to stop on target, instead oscillating back and forth over it.

The Critterbot reports sensor data, and takes in new actions, 100 times per second, so 10 milliseconds is the fastest possible time step that could be used. However, after 30 ms the off-robot agent may not have even communicated its new action to the agent due to wireless latency times, and it takes even longer (anywhere from 300-500 ms) for the robot to reach the full command speed passed to it (see Figure A.4). Operating at the shortest time step will make it difficult for the agent to realize the effect its commands have on the robot since they will constantly be interrupted by the next action and the Critterbot may not even move at all (Figure A.5).

Another important consideration is adherence to the *Markov property*, in which each state signal must hold all of the relevant information from the past needed to predict the next state (Sutton and Barto (1998), pp 62,63). When setting up a reinforcement learning problem it is a good idea to adhere as closely as possible to the Markov property. At the smallest time step it would be necessary to keep history about the last 4 actions just to understand what action the robot is currently processing. Adding history to our feature state would quickly grow the number of features, which would expand the computer memory used and slow learning.

Given these considerations, for this experiment a time step of 100 ms, or one tenth of a second, was chosen in order to compromise between having a small enough time step in order to exhibit precise control, and a large enough time step to compensate for the various sensor delays and inertia in movement. The last action taken was added to the feature representation to help make the state signal more Markov—more descriptive of the current state.

5.2.4 Speed of Movement

The motor command sent to the critterbot was chosen to be 10 for each wheel (out of 25), at which speed it takes the Critterbot a little less than 3 seconds to make a full revolution. The Critterbot can spin much faster, but this speed was chosen to balance movement speed with precision. Ideally the robot would learn to spin as fast as possible to each direction of maximization and slow down gracefully when necessary, but this would require a much larger discrete action set, or even continuous actions, which is not the focus of this experiment.

5.2.5 Feature Representation

The linear function approximator chosen for this experiment was *tile coding*. Tile coding is a function approximator used to partition the observation space into binary features. Each partition is called a tiling. There is only 1 active feature or tile per tiling. This feature is set to 1 and all other features are set to 0. Tile coding is a popular linear function approximation technique for reinforcement learning because it is easy to calculate, and the number of features are fixed. Having a fixed number of features makes it simpler to set the step size values which are dependent on the number of features. Figure 5.2 shows a gridtiling over two continuous inputs with two randomly offset overlapping tiles. A gridtiling scheme generalizes locally, but other tiling schemes can be used to generalize in different ways. Gridtiling is able to represent many functions, although it uses more data than other tiling schemes and does not necessarily generalize well.

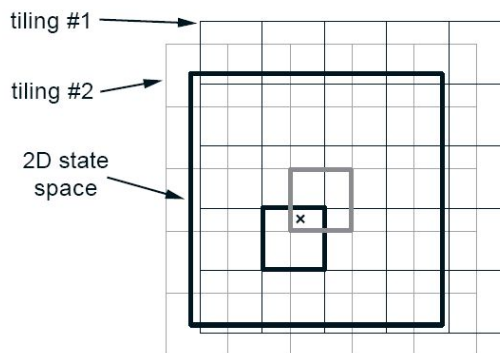


Figure 5.2: Tile coding with 2 overlapping tilings (from Sutton & Barto, 1998, used with permission). The x represents a sample joint data point and the active tiles are marked.

5.2.6 Feature Selection

To learn this problem well the agent should have an idea of the robot's orientation and current rotational velocity. The Magnetometer value on the X axis accurately describes the direction the robot is facing, much like a compass, except that its values are mirrored across the north/south axis, so when the Critterbot is pointing east it gives the same value as pointing west, northeast gives the same value as pointing northwest, and so on. Thus the learning agent will need a hint to distinguish which way the Critterbot is facing. The IR distance 2 sensor works well for this task (see Figure A.3(b)). It is on the side of the Critterbot, and thus, if the Critterbot is pointing west, the IR distance sensor would be facing the wall and would have a high value. If the Critterbot is facing east, this sensor would be facing out towards the rest of the pen and should have a low value. Lastly, one of the Motor velocities was included to enrich the state space with an idea of the current rotational velocity. Facing one way and being at rest is a much different situation than facing the same way and rotating at full speed since the Critterbot can not stop immediately the next time step, due to inertia. Its also important for the learning agent to understand when the Critterbot is not moving since the optimal policies should come to rest at the desired orientation of maximization for 7 of the 8 sensor maximization tasks. Motor speed was chosen over the rotational velocity because the gyroscope had been reporting odd values and did not seem like a reliable sensor to use.

The 3 sensors were tile coded together jointly. Each sensor was discretized into eight bins and used four overlapping joint tilings, resulting in $8^3 \times 4 = 2048$ binary features. Three binary features were added to represent the last action taken, so the agent would have some idea of what action the Critterbot was currently processing. Lastly, a bias unit always equal to one was added which is considered good practice when learning linear weights. Thus there were a grand total of 2052 features.

5.2.7 Algorithm

The algorithm I used is GQ(λ) with one modification. Instead of using the actual behavior policy probabilities in the calculation of $\rho(s, a)$, the behavior policy was set to always be 1.0. This was done to make the algorithm adhere closer to the *Greedy-GQ* algorithm (Maei et al., 2010), which does not use a ρ parameter. Since the value of π will always be 1 or 0 under the greedy policy (except in the case of ties), GQ(λ) reduces to Greedy-GQ with eligibility traces if the denominator of ρ is fixed at 1.0 and a greedy target policy is used. The algorithm used could thusly be

called Greedy-GQ(λ).

Separate learning agents were used for each sensor maximization task, each with their own weights and eligibility traces. These learning agents ran in parallel over each subsequent observation from the trajectory. Greedy-GQ(λ) was used by each learning agent (Figure 3.1). *Accumulating traces* were used, which means no limit was put on the size of eligibility traces.

The parameter α_w was set to 0.001, because in other experiments with GQ(λ), like the one in Chapter 6, a larger second step size slowed learning. The γ value was set to 0.98 to make the learning agent take a longer term approach to its actions. All weights were initially set to zero. The parameter settings for α_θ and λ are discussed in the evaluation section.

5.2.8 Evaluation

To evaluate the learned control policies, each policy was run with learning turned off for 250 time steps (25 seconds). Evaluations were run at 3 different starting orientations for each target policy one after the other to form a group of evaluations which I will call a trial. Ten trials were run for a total of 30 evaluations of each target policy. The reward–the sensor value being maximized normalized between 0 and 1—was recorded every time step. From this the average reward and the average return were calculated over all 250 time steps. Both produced similar results, so only average reward will be discussed further.

Before each set of trials, the robot was placed in the same location near a wall. A calibration routine was run before every trial. The ranges of each of the sensors being maximized were recorded by running each of the three actions for 5 seconds (the robot would spin one way more than a full revolution, stop, and spin back the other way). These ranges were used for normalization of the rewards and for the bounds of the tile coder. This way any slight changes in starting position or time of day that reflected in the sensor readings would be accounted for. Ranges were also calculated before the behavior policy was run to create the trajectory used in offline learning.

Limited evaluations (*i.e.*, less than 10 trials) were performed with α_θ chosen from the set $[0.1, 0.25, 0.5]$ and λ from the set $[0.2, 0.4]$. From these limited evaluations $\alpha_\theta = 0.1$, and $\lambda = 0.4$ performed the best after one-hundred thousand time steps so these parameters were chosen for the full study.

Ideally if the experiments were run correctly, we would see the robot spin immediately in the fastest direction to the point of maximization (if there is one) and stop. At worst, it should oscillate around the point of maximization if it can't find the best point or its action set is not discrete enough to reach the best point exactly.

5.3 Results

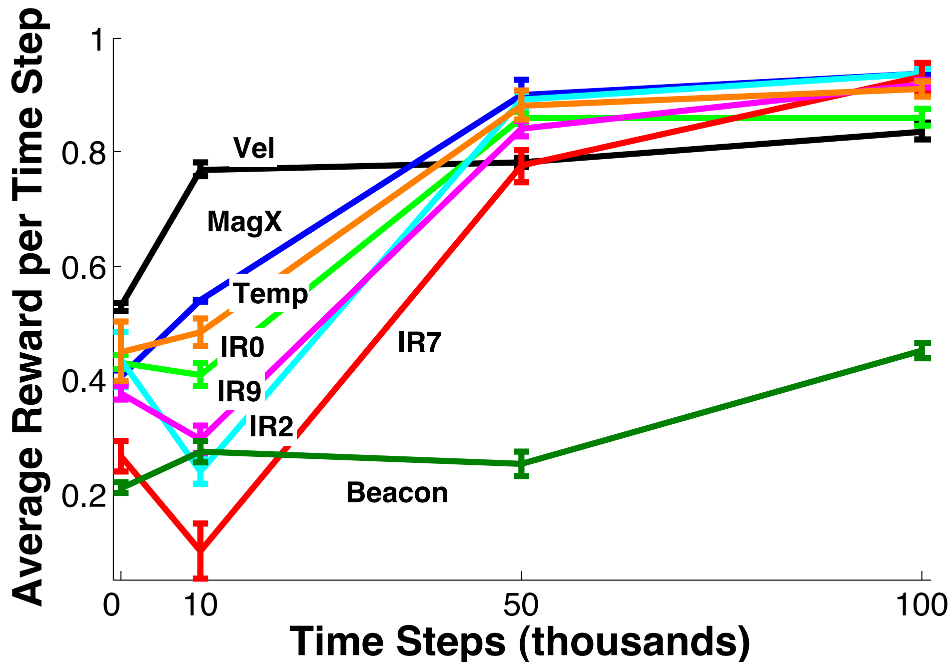


Figure 5.3: Learning curves for eight sensor maximization tasks.

Off-policy learning was halted at ten thousand, fifty thousand, and one-hundred thousand time steps and evaluations were run to measure learning progress. An evaluation was also run with no learning (time step 0). The results are shown in Figure 5.3. After only ten thousand steps of learning, the Critterbot's control policy was often just to spin in place, which is why some of the rewards are lower than the starting evaluation. After fifty thousand time steps the robot learns significantly, and continues to learn better control policies after one-hundred thousand steps.

Time delayed pictures of sample evaluation runs learned at 114,000 steps (the maximum amount of steps I had data for) for 4 of the 8 target policies are shown in Figure 5.4. The starting position for the evaluation runs shown is displayed in the first figure. The Figure named "VEL" refers to the wheel 0 velocity, and the policy

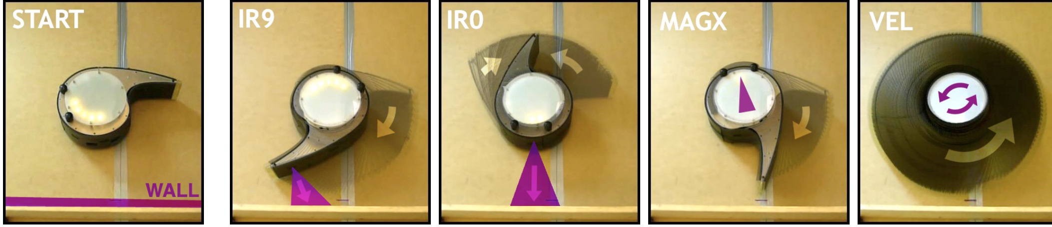


Figure 5.4: Time delayed photographs of the policies learned for sensor maximization. The starting orientation is on the left.

is to simply spin in the forward direction (counter-clockwise). The other policies turn directly to the point of maximization (sometimes overshooting and turning back) and the final resting place is superimposed over the photographs for clarity. In these examples it took less than 3 seconds to reach this resting position. Some trials took a little longer to achieve a resting position but they generally performed similarly to the trials shown.

5.4 Discussion

These results show that optimal target policies can be learned from snippets of random behavior. They also show that GQ(λ) scales to learning multiple policies in parallel. The polices were robust across the state space. The trials in this algorithm were started from three different orientations and the policies learned always made the robot turn in the direction which would achieve the goal most quickly. This experiment validates the algorithm GQ(λ) as a useful algorithm for off-policy reinforcement learning on real-world linear function approximation problems.

After I performed these experiments, the code was optimized so that the learning agent ran 10 times faster. With the current code, I could learn 80 or more control problems simultaneously in real time if memory would allow. Since GQ(λ) uses an amount of memory linear in the number of features, memory constraints are not as much of a concern as learning algorithms like LSTD (Boyan, 2002), that require enough memory to hold the number of features squared. Although, I ended up running the off-policy experiment offline so that I could learn with different parameter settings on the same trajectory of data, it is important to realize that this experiment could have been done on-line while the robot was running. When learned offline, the number of optimal policies that can be learned from the data is basically unbounded, because the data can be reused for each separate problem and there are no real time constraints.

The next step is to use GQ(λ) to learn temporally abstract models like *options* (Sutton et al., 1999) or *TD networks* (Sutton et al., 2006; Rafols, 2006). Then these temporally-abstract actions could be combined for use on larger problems, and so on. From this enormous potential of off-policy learning with linear methods, we have the seeds of a learning agent that could learn general knowledge about its sensorimotor experience autonomously.

Chapter 6

GQ(λ) vs. TD(λ) with Importance Sampling

The last step in the evaluation of the GQ(λ) algorithm in this thesis is to compare it to the previous off-policy method that is stable with function approximation that most closely resembles it. The second order methods are excluded because they are not O(n) in time and space complexity. The residual gradient algorithm is excluded because it is not practical on most real-world algorithms where the dynamics of the MDP are not known since it requires multiple samples at each state. This leaves the TD(λ) with importance sampling algorithm (Precup et al., 2001), which I will refer to as *TD(λ) IS* for the remainder of this chapter. This algorithm is stable for off-policy learning with linear function approximation on episodic problems of bounded length. The main drawback of TD(λ) IS is that it is subject to high variance and has been known to learn slowly. In the experiment in this chapter I compare GQ(λ) to TD(λ) IS to determine if it performs faster than the previous algorithm.

This chapter will also highlight the difference between the two algorithms in terms of the objective function each tries to estimate. Whereas GQ(λ) estimates the value function for the target policy with respect to the behavior policy's distribution, TD(λ) IS estimates the value function for the target policy with respect to the target policy's distribution. This is an important difference. Importance sampling is trying to learn about a target policy as if it had behaved this way from the beginning of time. GQ(λ) is trying to learn about the target policy in terms of an excursion from its normal behavior.

6.1 Experimental Setup

In order to do this comparison I evaluated GQ(λ) on a problem from the Precup et al. (2001) paper which TD(λ) IS was previously evaluated on. The problem domain, which is depicted in Figure 6.1, is an 11x11 grid world. This is an episodic problem. The MDP is deterministic and there are four actions, [DOWN, UP, LEFT, RIGHT]. The reward on every transition is zero, except for transitions into the 4 terminal states at each corner. Actions that would take the agent beyond the borders of the gridworld, such as DOWN from the bottom row, instead, leave the agent at the same spot. The initial state is in the center and the initial action is chosen randomly to be RIGHT or LEFT.

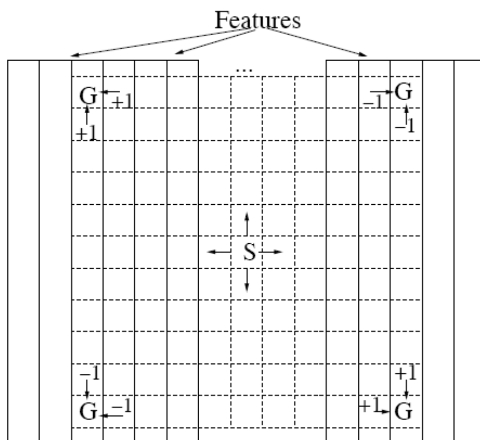


Figure 6.1: Gridworld environment from *Off-Policy Temporal-Difference Learning with Function Approximation* (Precup et al., 2001, used with permission)

Linear function approximation is used in this problem. The features are thirteen overlapping vertical stripes of horizontal width three. Three of the thirteen features are *active* (set to one) at all times, and the other features are set to zero. These features make it so that the agent's vertical location is indistinguishable. Thus I will call it the indistinguishable vertical location problem.

The behavior and target policies at every state are summarized in the following table:

	DOWN	UP	LEFT	RIGHT
Behavior	10%	40%	25%	25%
Target	40%	10%	25%	25%

They differ only in that their up and down action probabilities are flipped. The

behavior policy can well be thought of as an upward-drifting policy and the target policy as a downward drifting policy. Under the downward drifting target policy the features on the right side should be positive because the agent will tend to reach the bottom right terminal state, and the features on the left side should be negative, whereas under the upward-drifting behavior policy, the situation is reversed.

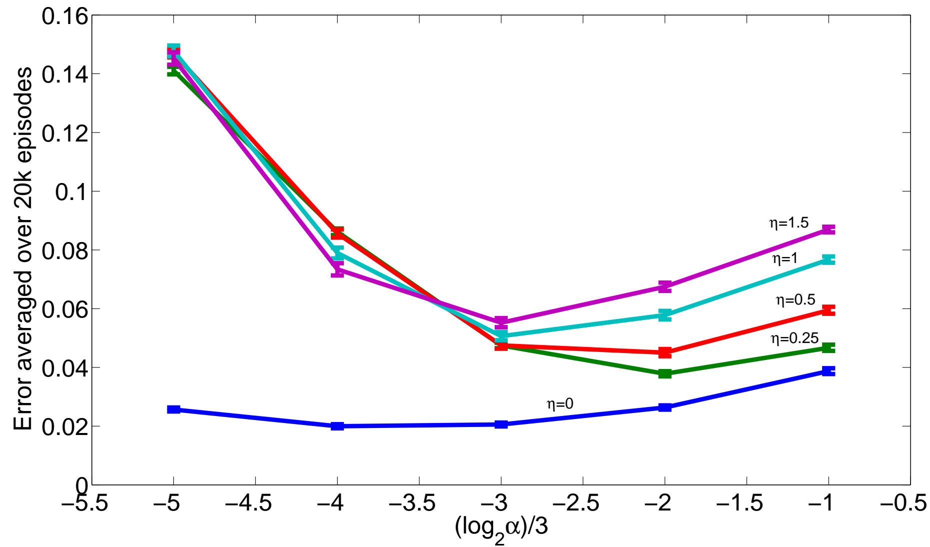
In this experiment I generated new results for GQ(λ) on the indistinguishable vertical location problem and compare them to the previous results of TD(λ) IS from the Precup et al. (2001) paper. The authors of that paper set γ to 1, so I did the same. In the Precup et al. (2001) paper, the incremental version of importance sampled TD(λ), was used with λ values of 0 and 0.9. I used two settings for λ, 0 and 0.5. These two parameter settings will be referred to as GQ(0), and GQ(0.5) respectively. I used a different range of alpha values for each λ setting. For $\lambda = 0$, I used α_θ from 2^{-5} to 2^{-1} in powers of 2 divided by the number of active features 3. For $\lambda = 0.5$, α_θ varied from 2^{-12} to 2^{-3} in powers of 2 divided by the number of active features 3. I divided each alpha setting by the number of active features, 3, as is common practice to minimize divergence. For both λ settings, I chose η from the set $[0, 0.25, 0.5, 1, 1.5]$. η is the ratio between the α_θ step-size and the α_w step-size:

$$\begin{aligned} \eta &= \frac{\alpha_w}{\alpha_\theta} \\ \alpha_w &= \eta\alpha_\theta \end{aligned}$$

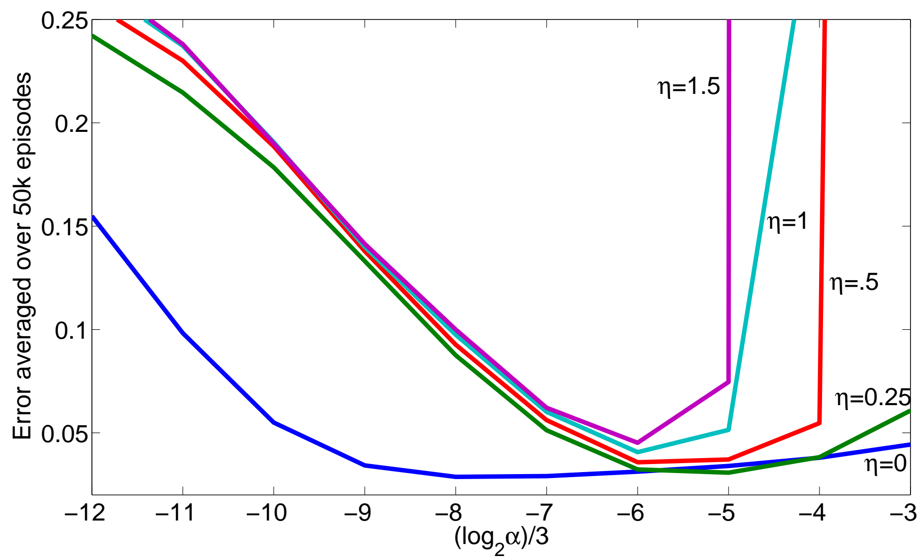
θ values were set to 0 initially in both the new and old experiment.

6.2 Results

I ran the experiment 50 times for 20,000 episodes each with $\lambda = 0$. GQ(λ) used the same trajectory of actions for learning with all 25 α and η combinations. The error measure was the root mean squared projected Bellman error averaged over all 20,000 episodes. Figure 6.2(a) shows the GQ(0) parameter study on the gridworld problem. The best parameter setting over all alpha values is with an η of zero. When η is set to zero, the second set of learning weights w is zeroed out, and GQ(λ) reduces to TD(λ) with linear function approximation and updates based on expectation over all possible actions. There were two parameter settings statistically tied for the best performance by this measure, $\alpha_\theta = \frac{2^{-4}}{3}$ and $\alpha_\theta = \frac{2^{-3}}{3}$, both with $\eta = 0$.

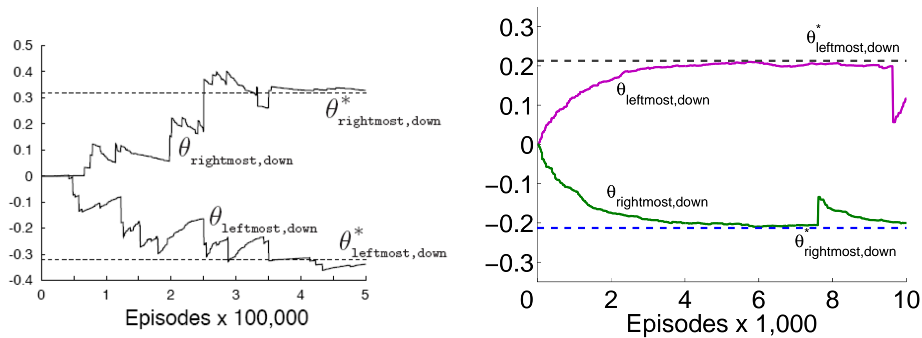


(a) $\lambda = 0$



(b) $\lambda = 0.5$

Figure 6.2: GQ(λ) parameter studies on 11x11 gridworld problem



(a) TD(λ) IS from Precup et al., 2001, used with permission

(b) GQ(λ) with step size $\alpha_\theta = \frac{2^{-3}}{3}, \eta = 0$

Figure 6.3: Weight value estimates of the leftmost and rightmost features for the down action from a single run. Note the scale for the number of episodes importance sampling used to learn is significantly higher.

I ran the GQ(0.5) algorithm for 50,000 episodes, because the problem took longer to learn with this setting. Figure 6.2(b) shows a parameter study of GQ(0.5) over all 50 α and η combinations. Again, a setting of zero for η gave the best performance in almost every case except for $\alpha_\theta = \frac{2^{-5}}{3}$ and $\alpha_\theta = \frac{2^{-6}}{3}$, where it was statistically tied with the next smallest η value, 0.25.

In order to compare the speed of learning of GQ(λ) and TD(λ) IS, the learning curve generated by TD(λ) IS from the Precup et al. (2001) paper is shown in Figure 6.3(a). This figure shows the objective values of two θ parameters, denoted by the dashed lines, and their estimates over time from a sample run of the algorithm. It takes about 350,000 episodes for the estimates from TD(λ) IS to settle near the correct values. By contrast, GQ(λ) learns its objective function much faster. Figure 6.3(b) shows a sample run using GQ(0) with one of the best parameter settings. It takes about 5000 episodes for GQ(0) to settle near its objective function. GQ(λ) learned its value function using 70 times less data then TD(λ) IS in these sample runs.

When you consider that the upward-drifting behavior policy reaches one of the bottom exit transitions less than 1 out of every 5000 times, and the downward-drifting target policy exits at the bottom 99.98% of the time, it becomes clear why the TD(λ) IS algorithm learns the target policy so slowly on this MDP. If the agent only exits out of the bottom row once every 5000 episodes, than it will rarely be exposed to the fact that the rewards for these exit transitions are reversed. Because learning relies on experience, learning about the exit transitions of the downward drifting policy will take much longer on this problem. Notice the spikes in the GQ(λ)

θ value plots in Figure 6.3(b). These signify an episode that ended in the bottom area of the grid, which changed the weights dramatically, but then they creeped back up to the fixed point values. Notice that there are only two of these in 10,000 episodes on this run.

6.3 Analysis of Results

GQ(λ) is faster at learning its objective function than TD(λ) IS is on this problem. However, notice that the parameter values that each is trying to converge to, displayed in Figure 6.3, have reversed signs. The value of the rightmost DOWN θ^* parameter which TD(λ) IS converges to is positive, while the same θ^* parameter GQ(λ) is trying to converge to is negative. Under the target policy the agent will spend most of its time in the lower section of the grid and thus we would expect the rightmost parameters to be of positive value, which TD(λ) IS eventually discovers (see figure 6.1).

Why does GQ(λ) converge to a negative value for the rightmost DOWN θ^* value? Perhaps this is a fluke. To check this out I calculated the TD fixed point that GQ(λ) approximates at $\lambda = 0.5$, as well as the on-policy TD fixed points of both the behavior and target policies for comparison. The V^* values for each column of the gridworld are shown as heat graphs in Figure 6.4. The V^* values for the downward drifting policy with the downward-drifting distribution, the target policy in this experiment, are shown on the top left in Figure 6.4. This is the value function that TD(λ) IS approximates. The V^* values for the upward drifting policy with the upward-drifting distribution, the behavior policy in this experiment, are shown on the top right. Lastly the V^* values for the downward drifting policy given the distribution of the upward drifting policy are shown on the bottom, which is the value function that GQ(λ) approximates in this experiment. Remarkably, this value function looks more like the value function calculated on-policy using the behavior policy than it does of the value function calculated on-policy using the target policy.

This has to do with the objective function that GQ(λ) approximates, the global minimum of the projected Bellman error (equation 3.3), which takes the distribution of the behavior policy into account. The distribution of the behavior policy for this problem is shown in Figure 6.5. As the figure reveals, the agent spends the majority of its time in the top row of the grid and very little time at all in the bottom half of the grid. Perhaps these squares are close enough to the corners on the top that the target policy would exit out of them more often than from the bottom even though

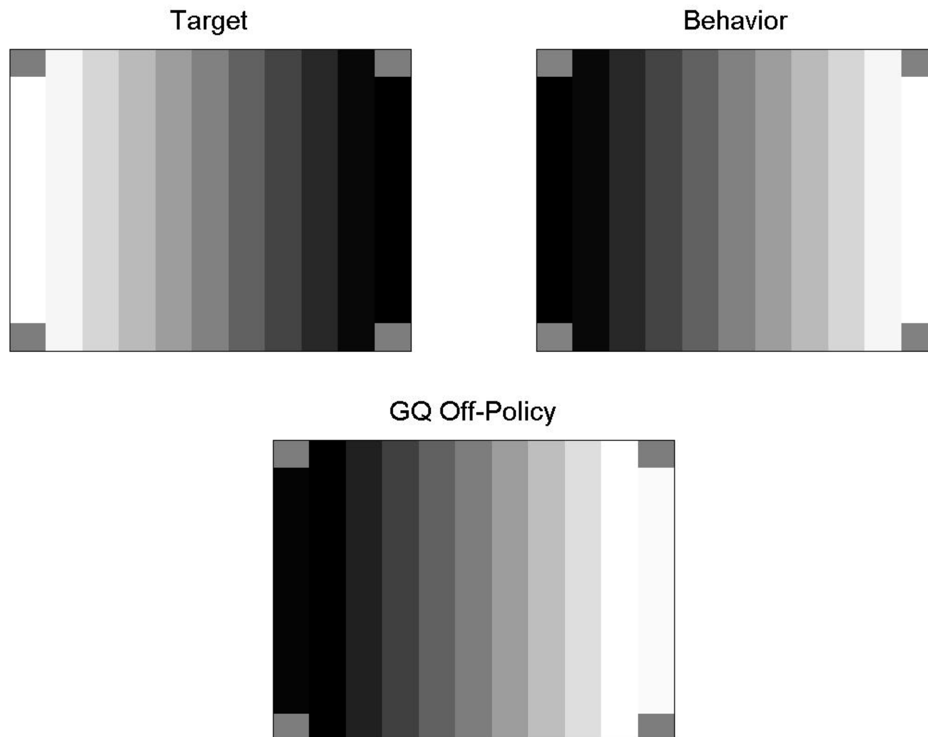


Figure 6.4: Heat graphs of the exact values, V^*, for the downward drifting policy used as the target (top left), upward drifting policy used as the behavior (top right), and the downward drifting policy given upward drifting policy distribution which GQ(λ) learns (bottom). Darker shades indicate higher reward. The learned GQ(λ) value function look more like the behavior values than the target they are trying to learn .

the target policy is downward-drifting. Then it would make sense that GQ(λ) would give negative value estimates on the right side of the grid and positive ones on the left side.

To test if this is true, I ran simulations from several different locations near the top of the gridworld. Starting from the top middle square of the grid, the location of highest distribution under the behavior policy (the location labeled 1 in Figure 6.5), I ran one million episodes with the target policy and found that the agent exited out of the bottom row of the graph 98% of the time. I performed a similar test with the square directly below the top right corner (location 2); and in this case, the agent exited out of the bottom 81% of the time and accumulated positive reward. I tried the square to the left of the top right corner (location 3) and found the agent still exited out of the bottom a majority of the time but accumulated negative reward. The square directly to the left of that (location 4), accumulated positive reward. In fact, location 3 and the corresponding square on the left side

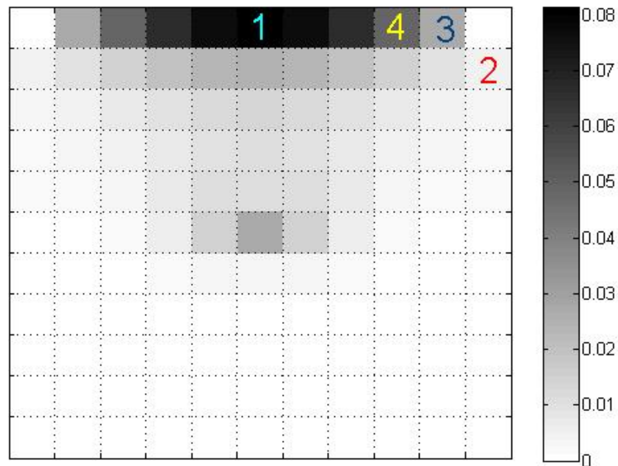


Figure 6.5: The distribution of the behavior policy

were the only squares with a large distribution of visits by the behavior policy whose estimated reward from starting at that square had the same sign as the TD fixed point V^* values. Under the target policy the agent would exit out of the bottom squares more often even if starting from the squares where the behavior policy is most heavily distributed. As a result, the agent would accumulate positive reward at the rightmost features and negative reward at the leftmost features which is the opposite of the TD fixed point values at $lambda = 0$ or $lambda = 0.5$ for this off-policy experiment.

I tried calculating the TD fixed point with different behavior policies and the same target policy to see if a more even distribution would cause the value function to look more like the on-policy value function. It turns out that to get the signs of the rightmost and leftmost features to switch at $\lambda = 0.5$, the behavior policy probabilities for up and down actions would have to be nearly even. The behavior policy would have to choose down .235% of the time or greater.

It may be that using a larger λ would reverse the signs, because a larger λ would put more importance on longer trajectories and it takes longer on average for the agent to reach the bottom states. I tried using $\lambda = 0.9$ with replacing traces, because the experiment consistently diverged with accumulating traces, but the values learned were very similar to those learned at $\lambda = 0.5$ even after 100,000 time steps.

If this were the tabular case, then GQ(λ) would converge to a value function

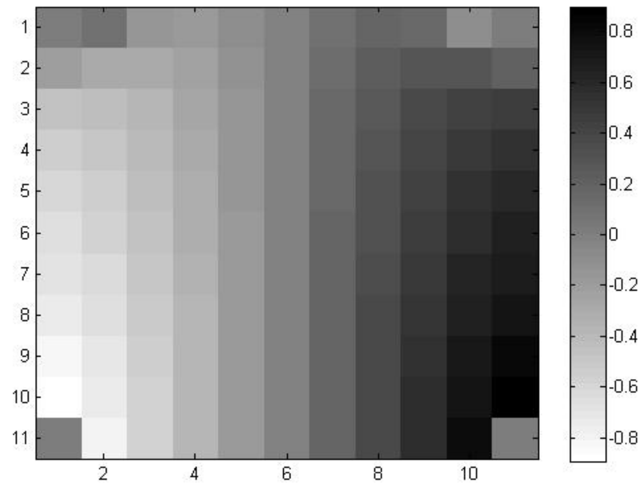


Figure 6.6: Heat graph for value function $V^{\pi}(s)$ for tabular case

that is similar to the target policy's value function, shown as a heat graph in Figure 6.6. When every state is represented discretely, the TD fixed point under the target policy is the same regardless of what distribution it is taken from, as long as the distribution visits all of the states. It is not even necessary to have a tabular case as long as the function approximation can jointly discriminate which corner the agent is closest to. In this experiment, since the function approximation obfuscates the vertical position, the behavior policy's state-action distribution becomes more important, and GQ(λ) converges to a solution that could lead to negative reward. For example, if an agent was made that chose up and down actions at the same % that the target policy chose them, 10% and 40%, but greedily chose left and right actions based on the value function given by GQ(λ) this agent would be lead to the bottom left corner of this gridworld where it would accumulate negative reward.

This shows that the GQ(λ) objective may give odd results for certain episodic off-policy tasks in which the function approximation yields poor generalization and the behavior policy rarely visits locations that the target policy regularly visits. In order to learn a value function the same way that an on-policy learner would, one should use TD(λ) IS instead.

One conclusion to be drawn from this is that GQ(λ) does not supersede TD(λ) with importance sampling. The older algorithm still has value, because it can be used off-policy to learn about a policy with respect to the policy's own distribution.

Chapter 7

Conclusion

In this thesis we set out to learn more about the GQ(λ) algorithm and off-policy learning, and both were achieved.

We employed a modified version of the GQ(λ) algorithm, which we called Greedy-GQ(λ), to learn a real-world off-policy function approximation problem: making a robot simultaneously learn many precise ways to interact with its world. Using Greedy-GQ(λ), the robot learned optimal policies to maximize eight different sensor values from simple random actions. The robot spun back and forth randomly like a newborn cycling its legs and arms, but learned how to stop at the exact moment so that it would come to rest several time steps later at an orientation that maximized the sensor value, like facing a wall, the dock, or magnetic north. It learned these near perfect policies despite all the problems inherit to the mobile robot domain like inertia, sensor delays, and wheel slippage. Though thousands of features were used to approximate the state space, it was possible to run this experiment in real time (at 10 hertz) because GQ(λ) is linear in the number of features. This was the first control experiment performed with any of the gradient based TD methods. It showed that GQ(λ) scales to real-world problems with linear function approximation. It showed that the RMSPBE objective can be used to approximate optimal extended policies from randomly generated data.

Despite great success with the mobile robot experiment, GQ(λ) had difficulty with a gridworld problem featuring indistinguishable vertical locations. GQ(λ) learned its value function much faster than TD(λ) with importance sampling, an older off-policy TD algorithm, learned its objective. However, the value function GQ(λ) approximated on this problem was the opposite of the value function that an on-policy TD learner would approximate, due to a feature representation that hid the vertical location and the fact that the agent spent most of its time in states that the target policy would not visit.

This was a disappointing result. Though it was on a specific gridworld problem, which was set up to make the TD(λ) with importance sampling algorithm look good, one could imagine a similar situation arising in other problems. If the agent spends more time in one state represented by certain features and not as much in another state represented by the same features then the reward received at states it spends more time in will dominate the value for those features.

This may have negative repercussions on the usefullness of this algorithm. For instance, trying to enhance a value function learned on-policy with off-policy updates from GQ(λ) could actually weaken the value function. Learning many value functions simultaneously might not be as useful as learning each sequentially on-policy. Learning worked in the robot experiment, but in that experiment, care was taken to use features that represented the possible states well and the distribution of state-actions was relatively even as well. There are advantages to running GQ(λ) off-policy in terms of exploration, but given these results, there may be big disadvantages as well. More research needs to be done to determine if these negative repercussions are real or imagined.

That said, perhaps it is asking too much of GQ(λ) to learn perfectly in situations with very poor function approximation and behavior distributions that are skewed towards areas of negative reward. The fact is that this algorithm was able to learn multiple policies with precise movements from a trajectory of only random actions. Unlike an online ϵ-greedy learner, which figures things out along the way and is then able to practice them over and over to perfect them, this algorithm was able to piece together extended actions from random snippets of movement. Learning policies in this way made them robust across the state space. This experiment showed promising results for both off-policy learning in general and the GQ(λ) algorithm, and will hopefully encourage further use of these in concert.

As for off-policy learning, we can make a few modest statements. We learned that in a prediction setting, it may be better to learn off-policy than on-policy, by using a behavior policy that is slightly more even distributed than the target policy. We also showed in a small domain, that if the affects of the distribution are accounted for, learning suffers as the disparity between the behavior and target policies grows.

7.1 Strengths and Weaknesses of GQ(λ)

GQ(λ) was used in all of the experiments in this thesis. What follows are some strengths and weaknesses of working with the algorithm.

GQ(λ) is a good out-of-the-box off-policy algorithm, because it has target and behavior policy probabilities built into the algorithm (the ρ variable), and it can be used on any off-policy problem without diverging (given the correct parameter settings). It also reached an answer within a reasonable time frame in all of the experiments.

GQ(λ) can be frustrating to use because there are not one, but two step sizes to consider, plus the lambda parameter. Setting one step size is a frustrating part of dealing with incremental methods. Having two step sizes to set only adds to the frustration. It would be helpful if an auto-stepsize adaption was made for both parameters. However, setting the second step size above 0 is typically unneccessary and can be harmful to learning rates. In most cases where the problem does not diverge, it is best to simply set the second step size to 0. A second step size of 0 performed the best or statistically tied with the best in the gridworld experiments. The second step size was set to a very small value of .001 in the robot experiment. It appears for some reason that the second set of weights, which is used to project the value onto the function approximation space, slows learning. Lastly, GQ(λ) learns value functions with respect to the behavior distribution. This can cause odd value functions in some off-policy experiments as shown in Chapter 6.

7.2 Future Directions

There are many more experiments that should be done to fully evaluate the GQ(λ) algorithm. For example, it would be interesting to try learning multiple policies on-policy, while also learning every other one off-policy using GQ(λ) to see if this helps or hinders learning. It would be interesting to robustly compare learning policies sequentially versus learning them in parallel. It would also be good to find a gridworld task that GQ(λ) excels at, perhaps something in a continuous setting.

The robotic experiment from Chapter 5 showed a robot learning many things about its sensorimotor interaction with the environment through random behavior. One future direction to take would be to explore *intrinsic motivation* (Singh et al., 2005) or *curiosity* (Schmidhuber, 2005) or other behavior policy algorithms that

might achieve better performance on learning many targets than a simple random policy would. Then it might be easier to learn on the robot with a larger action set and freedom to move anywhere.

It would also be possible to greatly expand the number of target policies being learned to give the robot greater knowledge about its environment. The next step would be to combine these value functions somehow to make the robot even smarter, perhaps by learning hierarchically time-extended actions such as options (Sutton et al., 1999), or TD Networks (Rafols, 2006). From the seeds planted by this experiment we could see a tree of general knowledge and interaction grow.

References

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. *Preiditis & Russell*, pages 30–37.

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

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

Bridle, J. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing: Algorithms, architectures and applications*, 227:236.

Dayan, P. (1992). The convergence of TD (λ) for general λ. *Machine Learning*, 8(3):341–362.

Geramifard, A., Bowling, M., and Sutton, R. (2006). Incremental least-square temproal difference learning. *Procedings of the National Conference on Artificial Intelligence*, pages 356–361.

Lagoudakis, M. and Parr, R. (2003). Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149.

Luce, R. (1959). *Individual choice behavior.* John Wiley.

Ma, J., Saul, L., Savage, S., and Voelker, G. (2009). Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 681–688. ACM.

Maei, H. and Sutton, R. (2010). GQ (λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *AGI*, pages 91–96. Citeseer.

Maei, H., Szepesvári, C., Bhatnagar, S., and Sutton, R. (2010). Toward Off-Policy Learning Control with Function Approximation. In *In Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel.

Precup, D., Sutton, R., and Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the 18th International Conference on Machine Learning*, pages 417–424. Citeseer.

Rafols, E. J. (2006). Temporal abstraction in temporal-difference networks. Master's thesis, University of Alberta.

Schmidhuber, J. (2005). Self-motivated development through rewards for predictor errors/improvements. In *Developmental Robotics 2005 AAAI Spring Symposium*.

Seijen, H., Hasselt, H., Whiteson, S., and Wiering, M. (2009). A theoretical and empirical analysis of Expected Sarsa.

Silver, D., Sutton, R., and Muller, M. (2008). Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning*, pages 968–975. ACM.

Singh, S., Barto, A., and Chentanez, N. (2005). Intrinsically motivated reinforcement learning. *Advances in neural information processing systems*, 17:1281–1288.

Singh, S. and Sutton, R. (1996). Reinforcement learning with replacing eligibility traces. *Recent Advances in Reinforcement Learning*, pages 123–158.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, pages 1038–1044.

Sutton, R. and Barto, A. (1998). *Reinforcement learning: An introduction.* The MIT press.

Sutton, R., Maei, H., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., and Wiewiora, E. (2009a). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000. ACM.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211.

Sutton, R., Rafols, E., and Koop, A. (2006). Temporal abstraction in temporal-difference networks. *Advances in neural information processing systems*, 18:1313.

Sutton, R. and Singh, S. (1994). On step-size and bias in temporal-difference learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 91–96. Citeseer.

Sutton, R., Szepesvári, C., and Maei, H. (2009b). A convergent O (n) algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in Neural Information Processing Systems*, 21:1609–1616.

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

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

Appendix A

The Critterbot

The Critterbot is a custom-built mobile robot (see Figure A.1), that was expressly designed for reinforcement learning experiments. It a comma-shaped frame with a hooked tail that facilitates object interaction. It is driven by three omni-directional wheels (see Figure A.2. Each wheel has tiny black rubber tires on it so that it can slide sideways but grip in the forward and backward wheel directions.

The robot has been designed to withstand the rigors of the reinforcement learning experience. It can drive into walls repeatedly without damaging itself. To keep the motors from overheating, the critterbot engages current limiting, which kicks in whenever the motors are running but the wheels aren't spinning (i.e. when it's pushing against a wall). The robots three sets of batteries allow it to run for extended periods of time; more than 8 hours continuously without recharging. When the batteries get low, there is a docking routine we run to autonomously dock itself to the charging station. The charging station has an array of IR lights that emit a beacon of light every half second, which the robot can detect to help find it's way to the dock.

A.1 Sensors

The Critterbot has a diverse set of sensors for relaying sensory information to the learning agent.

1. Wheel Speed (3)

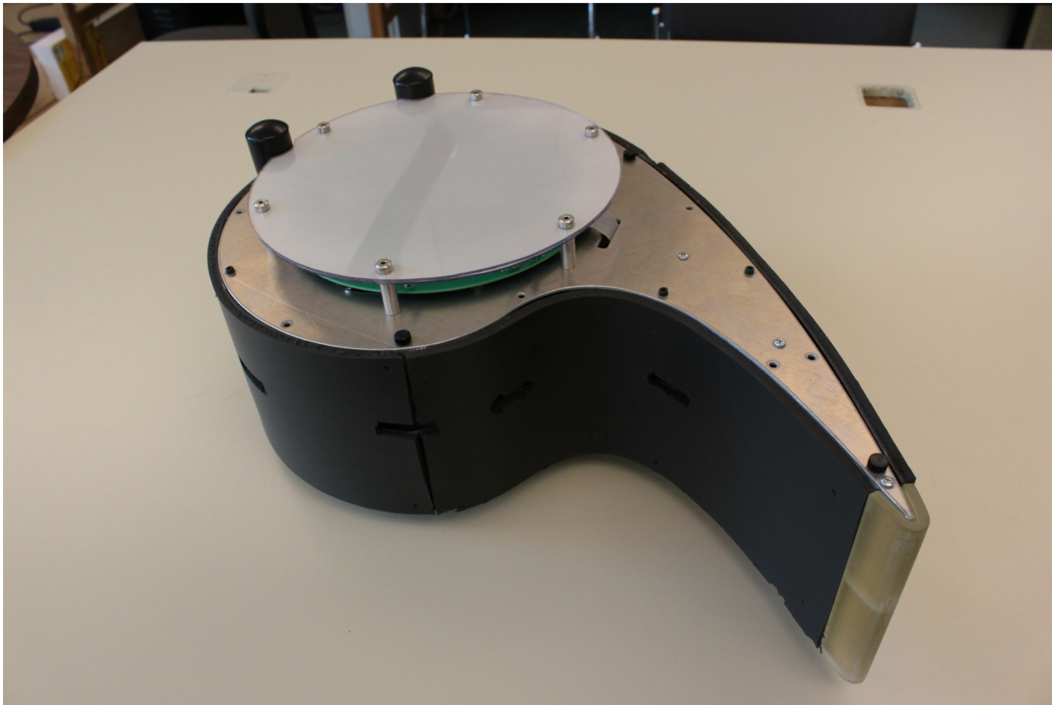


Figure A.1: The Critterbot

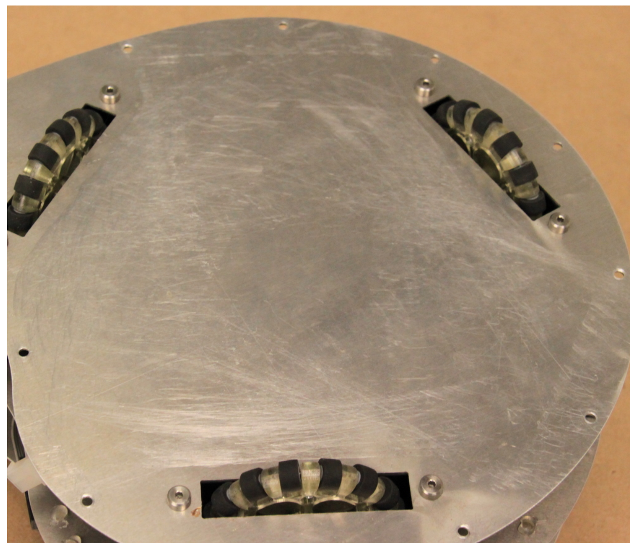
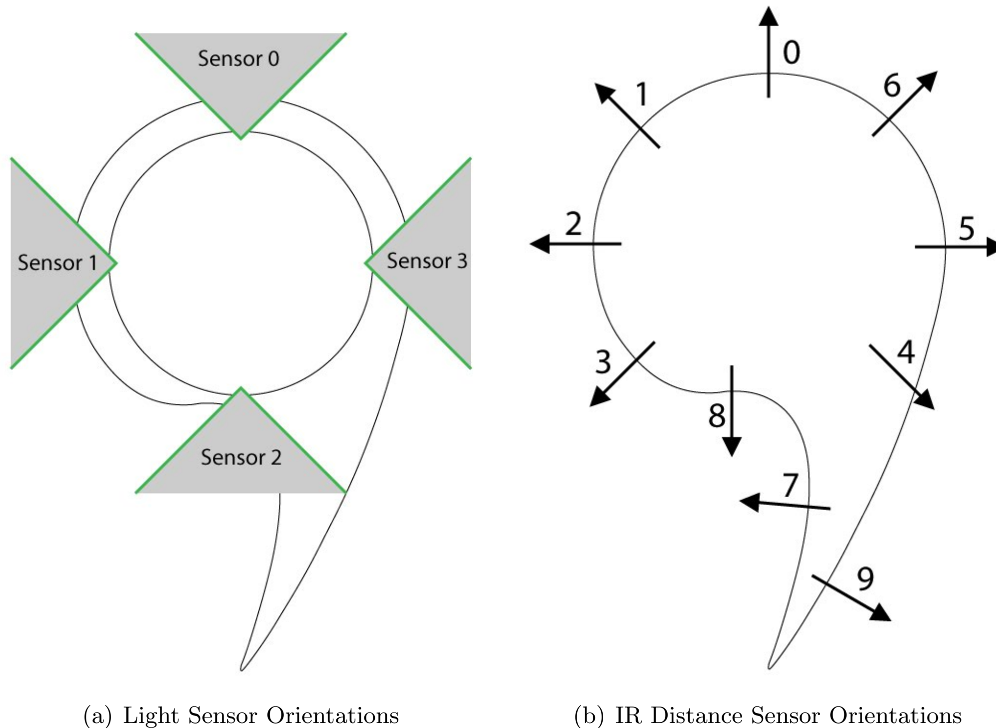


Figure A.2: The Critterbot's omnidirectional wheels



(a) Light Sensor Orientations

(b) IR Distance Sensor Orientations

Figure A.3: Various sensor locations and there orientations on the Critterbot

2. Motor temperature (3)
3. Motor current (3)
4. Rotational Velocity (gyroscope) (1)
5. Acceleration (x,y,z) (3)
6. Magnetic fields (x,y,z) (3)
7. Infrared Distance Sensors (10)
8. Light Sensors (4)
9. Infrared Beacon Sensors (8)
10. Temperature Sensors (8)

Most of the sensors are located around the top ring of the robot. The light sensors detect light in each of the cardinal directions starting from the critterbots nose (between the black pegs) counterclockwise as in Figure A.3(a). The robot detects obstacles with its infrared distance sensors. The infrared sensors are located

around the robot as in Figure A.3(b). It can use the Magnetic field sensors like a compass. The temperature sensors could be used to detect humans or computers or help detect the time of day or season.

The other sensors are proprioceptive sensors that provide information about how its moving and how its parts are functioning. The critterbot reports this information as raw sensor values without adjusting them to be easily consumable by humans. For example, a given temperature range in a day might be between 14592 and 15101 which roughly translates to around 19-22 degrees celsius, but instead we use the raw values. Another example is that the magnetometer x value is not translated to compass headings but instead mixes information about the global compass heading—magnetic north, with local information about strong magnetic fields nearby.

Sensor readings are read in at 100 hertz and sent out to nearby computers wirelessly in what is called a *StateDrop*. A program called Disco handles distributing this information to various clients (including the reinforcement learning agent). The agent sends back commands for each wheel, *ControlDrops*, to the Critterbot through Disco. There is also a 500mhz cpu on the critterbot capable of running complicated agents, so that the robot could run reinforcement learning agents as one self-contained unit.

A.2 Timing Delays

The robot has a considerable amount of inertia in its movements and delay in its sensor readings as well. Figure A.4 shows the latencies involved after sending a spin command at timestep 0 over a full second. From this Figure we can see that it takes between 3 and 4 hundredths of a second for the command to reach the critterbot due to wireless transmission times. Then the controller used to match motor commands limits the commands for a few timesteps. Finally about 8 timesteps later the full motor command is reached. It takes 45 timesteps for the motor speed to reach the command and the PID controller seems to have troubles maintaining that speed thereafter.

These delays can cause troubles for a reinforcement learning agent, especially one that is operating under a random policy. To illustrate this Figure A.5 shows a random policy executed on the Critterbot at three different timescales. The critterbot chooses amongst three actions, stopping and spinning left and right at a speed

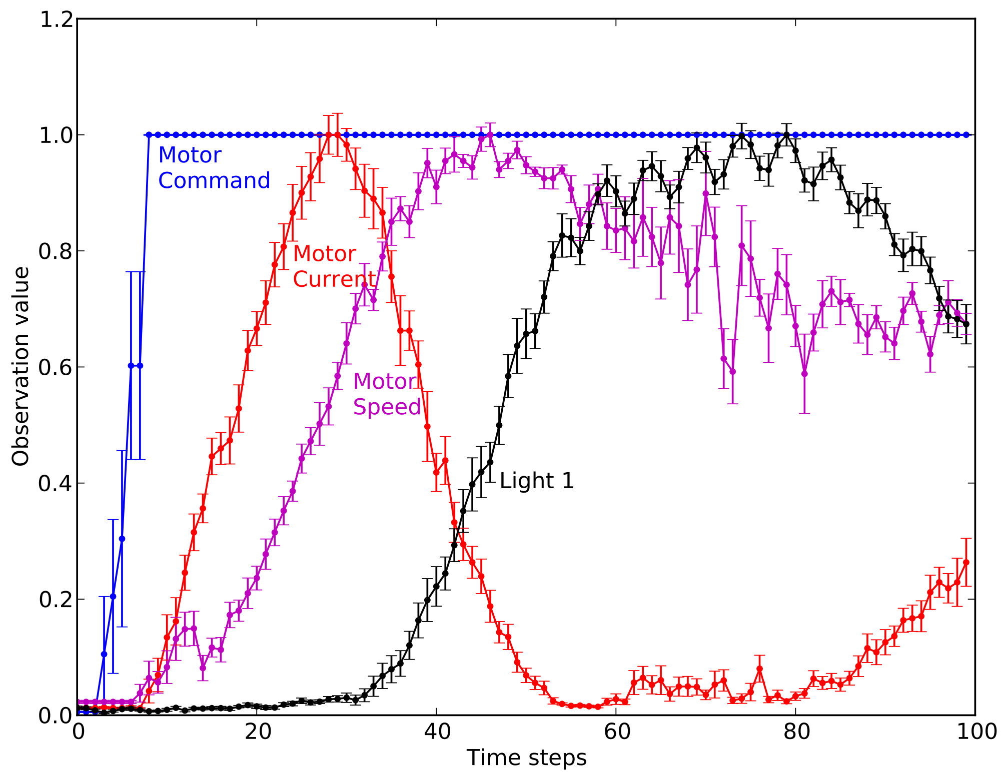
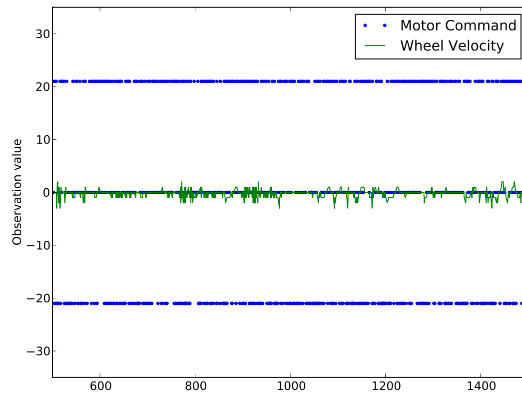


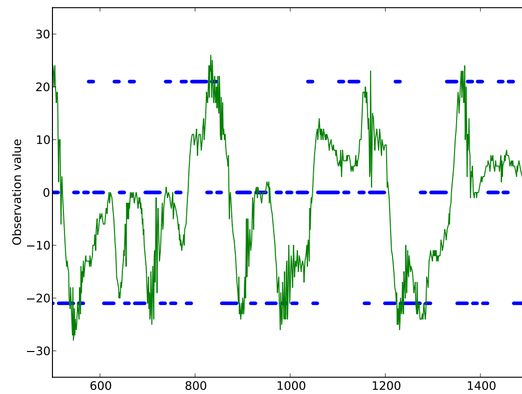
Figure A.4: Average latencies from the robot when given a command to spin from rest, and the effect on the motor command, current, speed, and a light sensor value over time since the command was sent. (courtesy of Thomas Degris)

of 21. Actions taken at 10ms barely cause a blip in movement. In fact the robot sat still, because the critterbot does not have enough time to attain any speed at this time scale before switching to something else. At 100ms, the robot starts to move but does not reach the velocity hoped for. Only at 500ms is the full command realized in terms of wheel velocity.

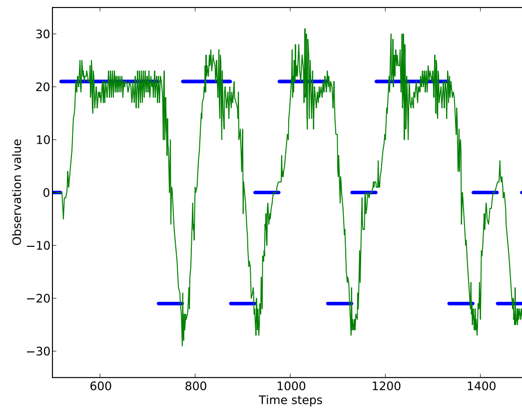
All of these various sensor inputs, delays from wireless transmission times, and movement inertia combine to make learning on the Critterbot a rich but challenging testbed for reinforcement learning algorithms.



(a) 10ms actions



(b) 100ms actions



(c) 500ms actions

Figure A.5: Random Exploration at different time scales showing how the inertia in wheel movement. At each time step an action is chosen randomly from [0, -21, 21] as shown by the blue line. Actions taken at 10ms barely cause a blip in movement, and only at 500ms is the full command realized in the wheels. (courtesy of Thomas Degris)