Gradient Ascent Critic Optimization

William Dabney WDABNEY@CS.UMASS.EDU
Andrew G. Barto BARTO@CS.UMASS.EDU
Department of Computer Science
University of Massachusetts
140 Governor's Dr.
Amherst, MA 01003

Technical Report: UM-CS-2010-052

1. Abstract

In this paper, we address the *critic optimization* problem within the context of reinforcement learning. The focus of this problem is on improving an agent's critic, so as to increase performance over a distribution of tasks. We use *ordered derivatives*, in a process similar to *back propagation through time*, to compute the gradient of an agent's fitness with respect to its reward function. With each iteration, the gradient is computed based upon a trajectory of experience, and the reward function is updated. We evaluate this method on three domains: a simple three room gridworld, the hunger-thirst domain, and the boxes domain. Starting from a random reward function, our gradient ascent critic optimization is able to find high performing reward functions which are competitive with ones that are hand crafted and those found through exhaustive search. We conclude that our sample based gradient ascent method for critic optimization is a fast and highly effective means of learning reward functions for use with reinforcement learning agents.

2. Introduction

The *actor-critic* framework for reinforcement learning separates the agent into two parts, the *actor* and the *critic*. The *actor* selects actions in each state so as to maximize, over some time horizon, a received reward signal. The *critic* provides this reward signal by mapping state information to a scalar value. Both are part of the *agent*, which directly interacts with the environment. The actor learns over the course of the agent's lifetime, but the critic remains fixed during a single lifetime. Both the environment and the agent provide to the actor-critic state information. The state, s^E, coming from the environment is said to come from the *external environment* (cite here), also known as the *problem space*. The state, s^I, coming from the agent is said to come from the *internal environment*, also known as the *agent space* (cite george). Figure 1 shows these terms in their overall context. In general we will simply refer to the *state* as being the composition of these two variables, $s = < s^E, s^I >$.

Traditionally, the critic has been designed by hand to allow the agent to solve a specific problem intended by the designer. This approach is challenging in that it requires significant fine tuning, trial and error, and an understanding of the dynamics of the environment. Furthermore, designing a critic that can transfer across multiple tasks is an even greater challenge. To avoid this tedious task, one might choose instead to use a naive critic which rewards only the achievement of the task at hand. However, this approach leads to significantly longer exploration times and is not practical beyond toy problems. This has led many researchers to look for methods to improve the critic without human intervention.

Intrinsically Motivated Reinforcement Learning (IMRL) adds to the traditional critic the use of intrinsic rewards from the saliency of events based on conditions such as novelty, progress, and surprise (Barto and Şimşek, 2005; Oudeyer and Kaplan, 2008; Kaplan and Oudeyer, 2007; Schmidhuber, 2005). Generally IMRL approaches have required the notion of saliency to be pre-defined, and because of this, early IMRL methods could be viewed as being somewhere in between hand designed critics and automatically designed critics. More recently, much research has focused on automatically discovering instrinsic rewards without any pre-defined notions of what they will look like (Uchibe and Doya, 2008; Niekum, 2010; Schembri et al., 2007). These, and other related approaches to automated critic optimization, have used genetic algorithms or other evolutionarily inspired approaches (Elfwing et al., 2008; Ackley and Littman, 1991; Snel and Hayes, 2008). Some have even shown that automated critic optimization mitigates the bounds on an agent's range of learnable tasks (Sorg et al., 2010). A common theme in all this work is the use of uninformed search methods for critic optimization. In this paper we present a gradient ascent based method for automated critic optimization, which is able to incrementally improve a critic using the sampled trajectory of a single agent's lifetime for each iteration.

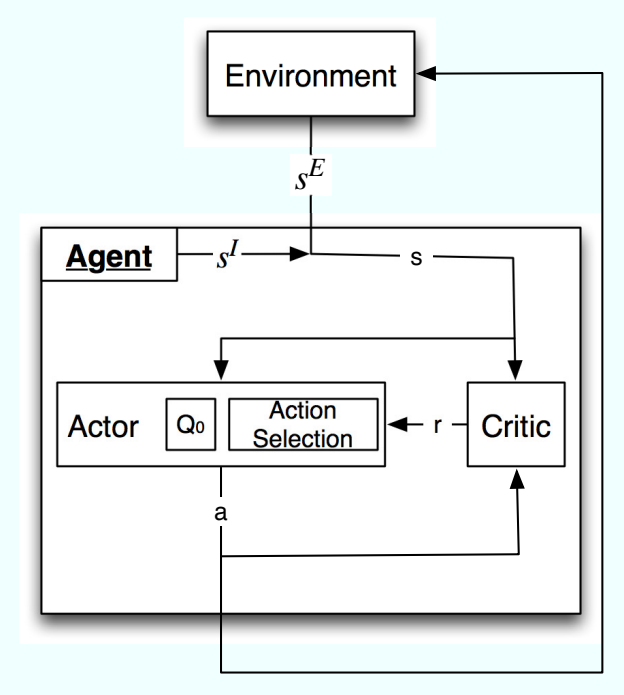

Figure 1: Actor-critic architecture in context

3. Problem Specification

This paper focuses on *critic optimization* using gradient ascent and back-propagation through time. We define the *critic optimization problem* as finding solutions that maximize the expected *fitness* over an unknown distribution of environments. *Fitness* is defined as a scalar evaluation of an agent's lifetime, which is provided by the environment. In practice we are only interested in distributions of environments containing some common structure and similarity between tasks, as otherwise the problem would have no solution. The space of potential solutions contains all possible critics representable given the agent's function approximation. In this paper we restrict ourselves to critics represented as a *reward function*, which is itself represented through linear function approximation.

Our approach to solving this problem works iteratively. For each iteration, back-propagation through time is applied to the trajectory of sampled state-actions making up the agent's lifetime. This computes a gradient for fitness, which is used to update the agent's reward function. This approach requires that we assume the fitness function to be differentiable with respect to the reward function and that its equation is known. For all experiments in this paper we used a fitness function of the form: $\rho_l(H_l) = \sum_{s,a \in H_l} F(s,a)$, where $F(s,a)$ is an unknown function over states and actions taken during the agent's lifetime. We rewrote the fitness function, for purposes of derivation, as $\rho_l(H_l) = \sum_{s,a \in S \times A} f_T(s,a)$, where f_T is a value-function over the values of $F(s,a)$. Although the method presented in this paper can be used with a model of the environment, we instead demonstrate that it is highly effective even when applied to sampled trajectories of a single agent's lifetime. Furthermore, the methods can be trivially be applied to sub-trajectories of an agent's lifetime, which would allow critic optimization within a single lifetime.

For this paper we use a policy-dependent $SARSA(0)$ actor (reinforcement learning agent). The dependency network for this agent is shown in Figure 2. We can see that values from the initial state-action value function and the reward function affect the performance of the learning agent in a fairly straight forward way. The reward function influences the Q-values, which in turn influence the policy and thus the value function for fitness, f. The environmental dynamics directly affect which states are visited during the agent's lifetime, and thus the dependencies at each level of Figure 2.

We will now examine the details of how we compute the gradient of the fitness function with respect to the reward function. In the following section it may be noticed that a intermediate variable used in our computations are the derivatives of the fitness function with respect to the initial state-action values, Q_0. This implies that we could be co-optimizing the reward and initial value functions. However, for clarity we have chosen only to optimize the reward function.

4. Ordered Derivative of Fitness (Chain Rule Applied)

We compute the derivative of the fitness function with respect to the reward function using repeated applications of the chain rule to Equations 1 - 4. A common term for this type of computation is the *ordered derivative*, and the same technique is used for performing Back-Propagation Through Time (BPTT)(Werbos, 1994). Whereas BPTT computes

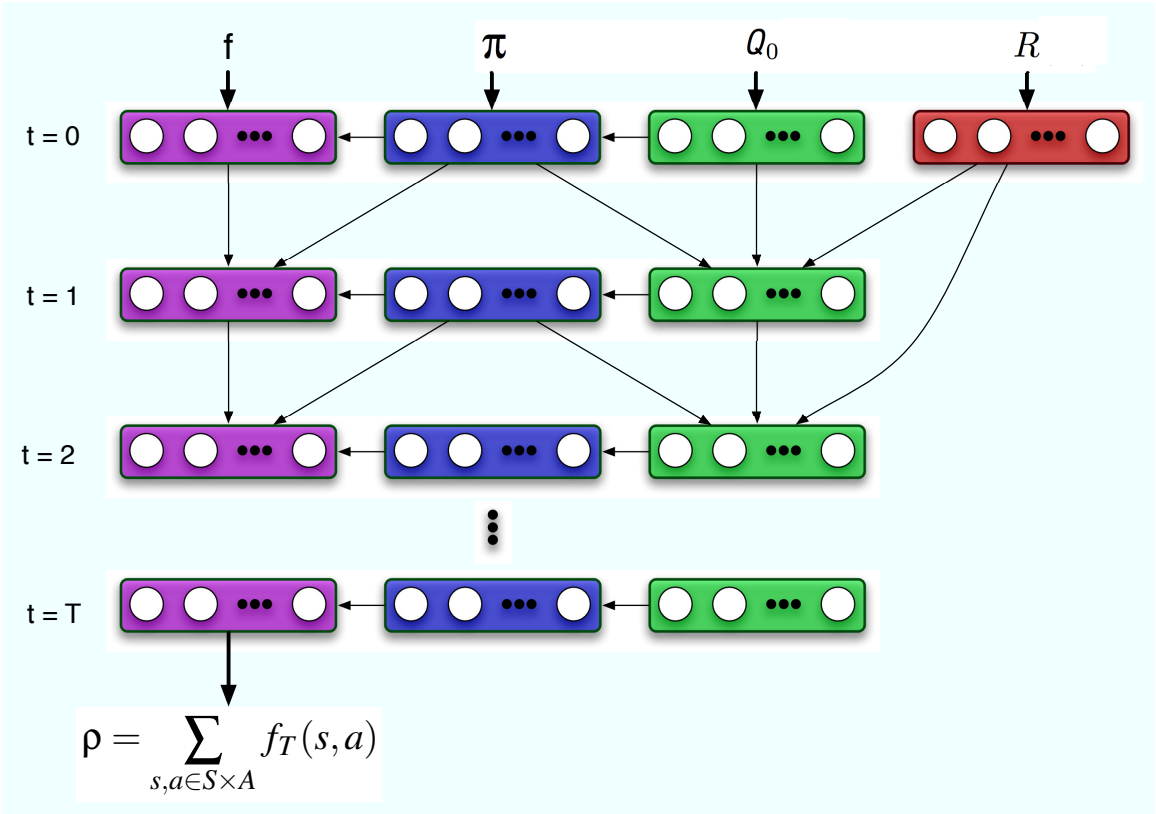

Figure 2: Dependency network for SARSA(0). Each box shows which nodes represent a certain function's values. Each circle is a node representing the value for a function at a given state-action pair.

the gradient of the weights between nodes in the network, we compute the gradient for the values at each layer of the network. This approach is similar to what is typical in the optimal control literature. We cannot directly compute the gradient without such a method due to the dependence, at every time step, on functions at the previous time step. Furthermore, unlike the policy gradient technique, in our case the reward function affects the policy continually throughout the agent's lifetime and thus the final policy is not independent of the value function or the reward function.

Due to the incremental nature of this method of computing the gradient, a reasonable approximation to our method is to iterate over a small fraction of the entire lifetime, such as, the last K time steps of the lifetime. In its most extreme form, this approximation has $K = 1$ and would compute only the *explicit derivatives*. Similarly, this approximation technique could be used to apply our methods to optimizing the reward function online, by applying it as soon as a non-trivial fitness signal is encountered. These approximations are an interesting area for future research, but are not explored further in this paper.

$$f_{t+1}(s_t,a_t) = (1-\alpha) f_t(s_t,a_t) + \alpha[F(s_t,a_t,s_{t+1}) + \gamma \sum_{a'} \pi_t(s_{t+1},a') f_t(s_{t+1},a')] \tag{1}$$

$$Q_{t+1}(s_t,a_t) = (1-\alpha) Q_t(s_t,a_t) + \alpha[R(s_t,a_t,s_{t+1}) + \gamma \sum_{a'} \pi_t(s_{t+1},a') Q_t(s_{t+1},a_{t+1})] \tag{2}$$

Our first set of equations are for the policy dependent $SARSA(0)$ algorithm. Equations 1 - 2 show how the actor computes its state-action value function, Q, and how we compute the fitness value function, f.

$$\pi_t(s,a) = \frac{e^{Q_t(s,a)/\tau}}{\sum_{b \in \mathcal{A}} e^{Q_t(s,b)/\tau}} \tag{3}$$

$$\rho = \sum_{s,a \in S \times A} f_T(s,a) \tag{4}$$

The policy, used to compute the above two functions, uses *soft-max* (Sutton and Barto, 1998), and is shown in Equation 3. The fitness function, common in form across all our experiments, is shown in Equation 4. With these four equations we are ready to begin the derivation, shown in Appendix 7.

$$\frac{\partial^+ \rho}{\partial R(s,a,s')} = \sum_{t>0} \frac{\partial^+ \rho}{\partial Q_t(s,a)} \frac{\partial Q_t(s,a)}{\partial R(s,a,s')} \tag{5}$$

$$\frac{\partial^+ \rho}{\partial Q_t(s,a)} = \sum_{s',a'} \frac{\partial^+ \rho}{\partial Q_{t+1}(s',a')} \frac{\partial Q_{t+1}(s',a')}{\partial Q_t(s,a)} + \sum_{a'} \frac{\partial^+ \rho}{\partial \pi_t(s,a')} \frac{\partial \pi_t(s,a')}{\partial Q_t(s,a)} \tag{6}$$

$$\frac{\partial^+ \rho}{\partial \pi_t(s,a)} = \sum_{a'} [\frac{\partial^+ \rho}{\partial f_{t+1}(s,a')} \frac{\partial f_{t+1}(s,a')}{\partial \pi_t(s,a)} + \frac{\partial^+ \rho}{\partial Q_{t+1}(s,a')} \frac{\partial Q_{t+1}(s,a')}{\partial \pi_t(s,a)}] \tag{7}$$

$$\frac{\partial^+ \rho}{\partial f_t(s,a)} = \frac{\partial \rho}{\partial f_t(s,a)} + \sum_{s',a'} \frac{\partial^+ \rho}{\partial f_{t+1}(s',a')} \frac{\partial f_{t+1}(s',a')}{\partial f_t(s,a)} \tag{8}$$

The above equations show the updates performed at every time step, in the reverse direction of the lifetime of the agent. That is, the computation starts with $t = T$ and continues until $t = 0$. The derivation of these equations and the equations for the explicit derivatives (such as $\frac{\partial Q_t(s,a)}{\partial R(s,a,s')}$) are shown in Appendix A. This method is readily applicable to table linear and non-linear function approximation. In this paper we only look at table based and other simple linear function approximations.

5. Experimental Results

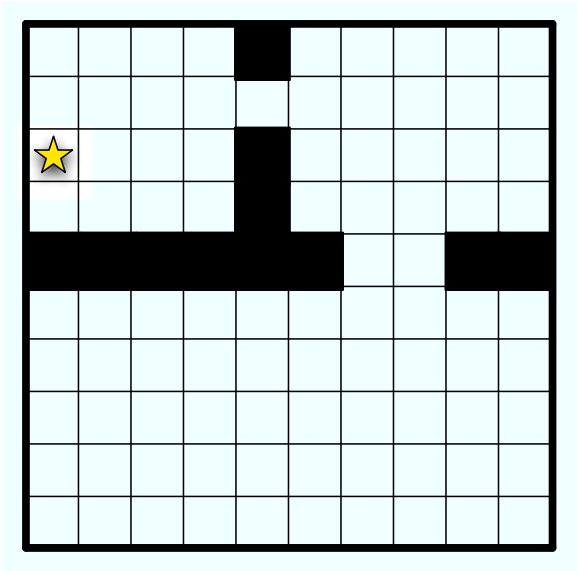

Figure 3: Simple gridworld used for navigation tasks. The star indicates goal location.

5.1 Three Room Gridworld: Navigation

Our first domain is a simple three room gridworld with a fixed goal location, shown in Figure 3 with a star marking the goal. The agent has four actions that allow it to navigate the environment: up, down, left, and right. Each agent's lifetime begins in a randomly selected grid location. We begin with a random negative reward function and apply the gradient based optimization to the reward function for 25 iterations.

We compare the reward function found using our gradient ascent method with three other possible reward functions: A naive reward function which mirrors the fitness function (reward of 1.0 at the goal and 0.0 everywhere else), a simple handcrafted reward function (reward of 1.0 at the goal and -0.01 everywhere else), and ΔV based reward function which is defined as $R_{\Delta V}(s,a,s') = \gamma max_{a' \in \mathcal{A}} Q^*(s,a') - max_{a' \in \mathcal{A}} Q^*(s',a') + F(s,a,s')$. This last reward function is essentially a shaping reward combined with the fitness based rewards, with the potential function computed from the optimal value function Q^*. The optimal value function was computed separately using a lifetime of 200000 steps, with Q-Learning, and using just the fitness function as a reward ($\alpha = 0.1, \gamma = 0.9$). The results are averaged over 30 runs in which the agent's starting location is randomly selected, and are shown in Figure 4. Another view of these same experiments is given by Figure 5, which shows the performance as the cumulative fitness over a lifetime.

We can see that when the optimal value function can be completely solved that using $R_{\Delta V}$ as the critic is a sound choice. However, the gradient learned critic is able to match its performance and was found without needing to

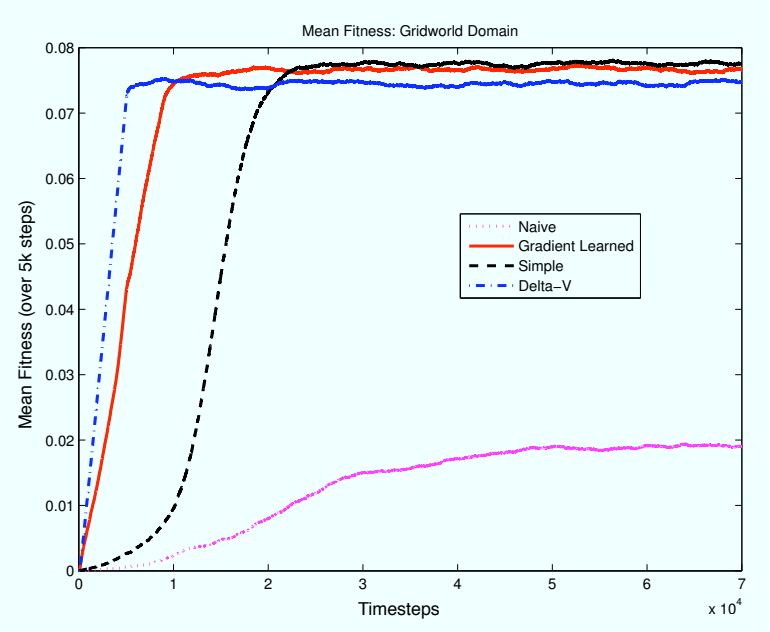

Figure 4: Three room gridworld, sliding window average of the mean fitness over a lifetime of 70000 steps. Averaged over 30 evaluation runs. Shows the four best performing critics for comparison.

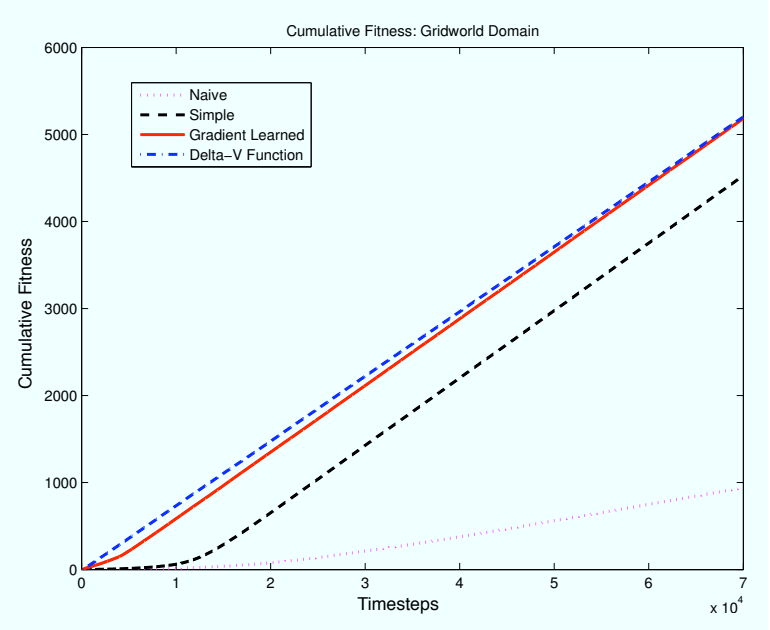

Figure 5: Three room gridworld, cumulative fitness over a lifetime of 70000 steps. Averaged over 30 evaluation runs. Shows the four best performing critics.

completely solve the domain. While these details are of less significance in such a simple domain, they become very important as our domains become more challenging and as they begin to change between agent lifetimes.

We also used this domain to answer an intuitive question that often comes up when considering reward functions. Why not use the optimal value function as the reward function? When a reasonable approximation to the optimal value function is available this seems very appealing, especially considering the performance of $R_{\Delta V}$. But, as it turns out, performs rather poorly. Figures 6 and 7 show a comparison of three potential reward functions, from the same experiment that produced Figures 4 and 5. We compare in these figures only the optimal value function as a reward, the random negative reward function used as the starting point for the gradient ascent algorithm, and the naive reward function discussed above. What we have found is that using the optimal value function as a reward function does not work well at all, not even out performing a completely random negative reward function.

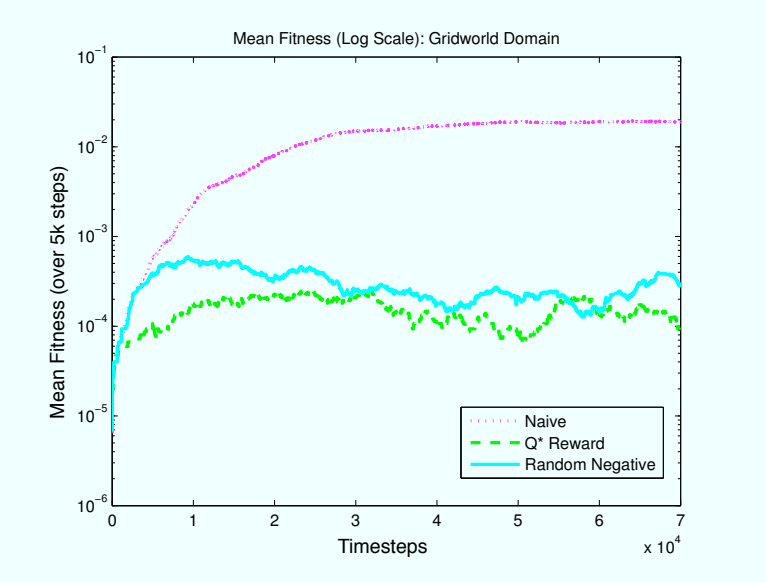

Figure 6: Three room gridworld, sliding window average of the mean fitness over a lifetime of 70000 steps. Averaged over 30 evaluation runs. Shows the two worst performing critics, and the Naive critic to provide context. The performance is shown in log-scale.

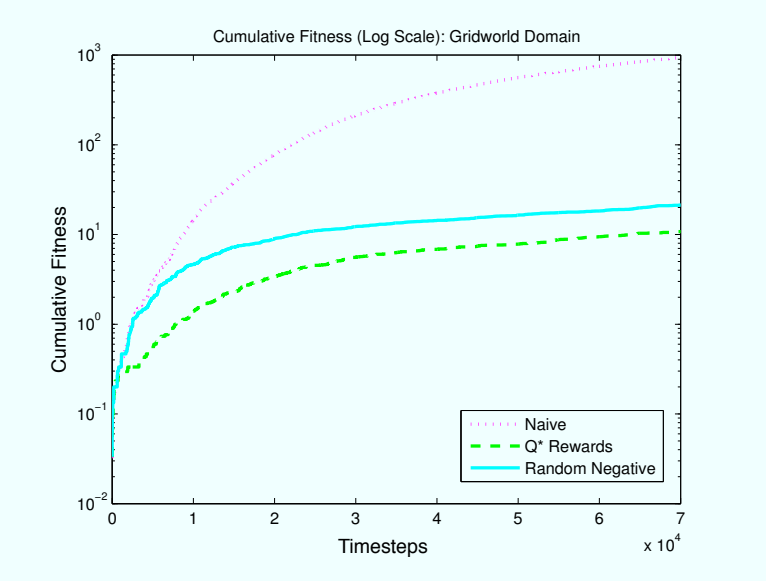

Figure 7: Three room gridworld, cumulative fitness over a lifetime of 70000 steps. Averaged over 30 evaluation runs. Shows the two worst performing critics with the Naive critic for comparison. Performance is shown on a log-scale.

By examining the state visitation probabilities induced by the optimal value reward function, and comparing it to those induced by the gradient learned reward function, we are able to see what causes the poor performance, as shown in Figures 8 & 9. The state visitations for the gradient learned reward function, indicative of a good policy, increase in a shortest path towards the goal. However, the visitation probabilities for the optimal value function rewards are very high at a single state which is not the goal state. This suggests the problem is that the agent has found a loop of states that give a positive reward that it prefers over finding and achieving the goal.

Although this domain is very simple, it demonstrates an important test case. That is, when the fitness signal does not change over lifetimes, our gradient ascent method for optimizing a reward function works quickly and highly effectively. Additionally, the optimized reward function would provide a better starting point for additional optimization, as opposed to a random or simple reward function, should the fitness signal begin to slowly change.

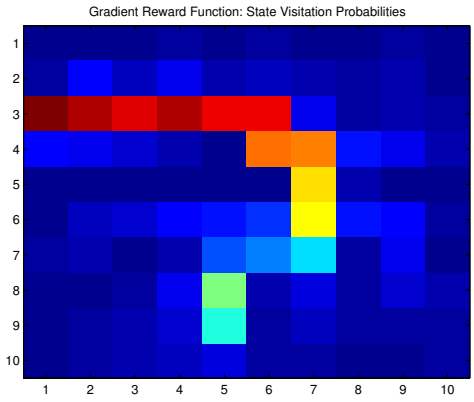

Figure 8: State visitation probabilities induced by gradient learned rewards

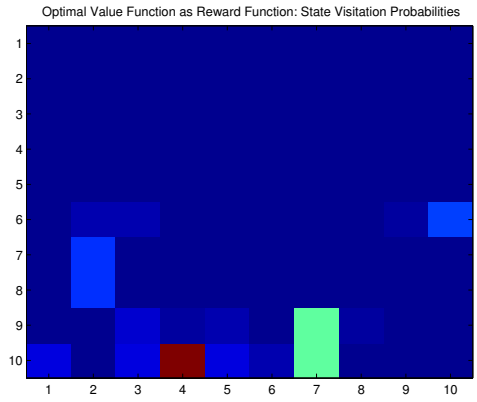

Figure 9: State visitation probabilities induced by optimal value function rewards

5.2 Hunger-Thirst Domain

The hunger-thirst domain is a distribution of similar tasks in a gridworld and has already been explained and studied in detail by others (Singh et al., 2009). Essentially, it consists of a 6×6 gridworld with walls that create four 3×3 rooms, shown in Figure 10. Within a given environment, sampled from the distribution, there is a single grid location which is a water supply, and a single separate grid location which is a food supply. The location of the food and water vary randomly among the four corners of the gridworld. Within a single lifetime these locations are fixed, but will change from lifetime to lifetime.

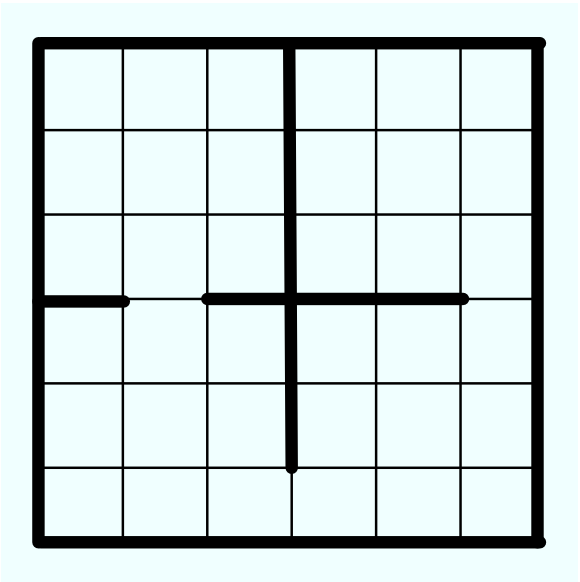

Figure 10: Hunger-thirst domain with walls shown.

Due to the placement of the walls, some environments are far easier than others. This makes hunger-thirst both more interesting and difficult to use as a comparison domain. The challenge in using the hunger-thirst domain is due to the large variance in performance resulting from the differences in environments. To reduce the effects of this variance as much as possible, we compare the performance of different reward functions over an identical sampled set of environments. That is, the locations for food and water faced by each agent are the same for each comparison. If we did not do this, it is conceivable that one agent could be fortunate enough to experience many easier environments while another might receive many difficult ones, making a comparison meaningless.

Due to the difficulty of some environments in this domain, and the prevalence of local extrema in the reward function space, we used a simple method for getting out of local extrema for experiments within this domain. The method applies only when an agent's lifetime does not result in any fitness. Instead of throwing away the trajectory, which is unusable for computing the gradient, we use simulated annealing, weighted by state-aciton visitation, to take a small random step in reward function space. We compute the semi-random update as follows. Let $v(s,a)$

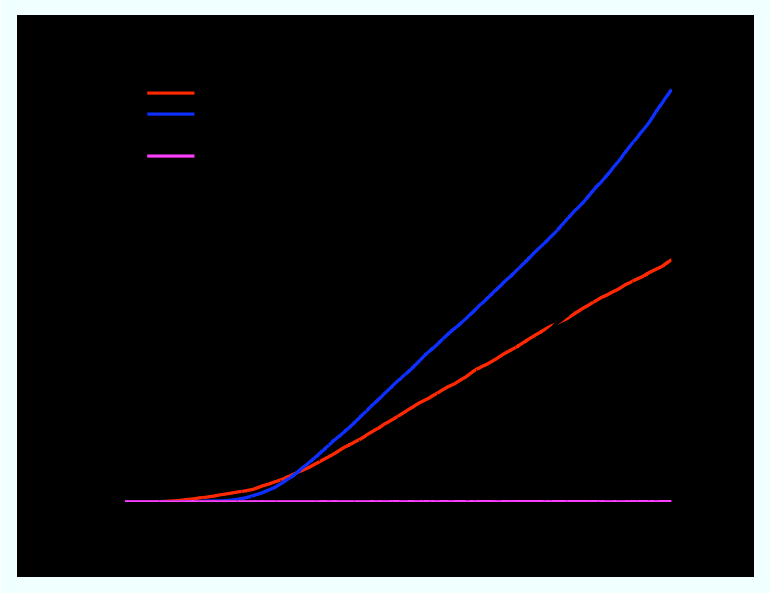

Figure 11: Mean Cumulative Fitness of different linear reward functions

be the ratio between the visitation frequency for state-action (s,a) and the frequency of the mode. Then we define: $B(s,a) = \tau e^{v(s,a)\times c}$ and the update to the reward function is $\forall s,a : random(-1.1,-1.0) \times \frac{B(s,a)}{max_{s',a'}B(s',a')}$. The variables $\tau = 1.0$, decreasing multiplicatively by 0.1 each use, and $c = 5.0$ were informed guesses at reasonable values but were not specifically optimized.

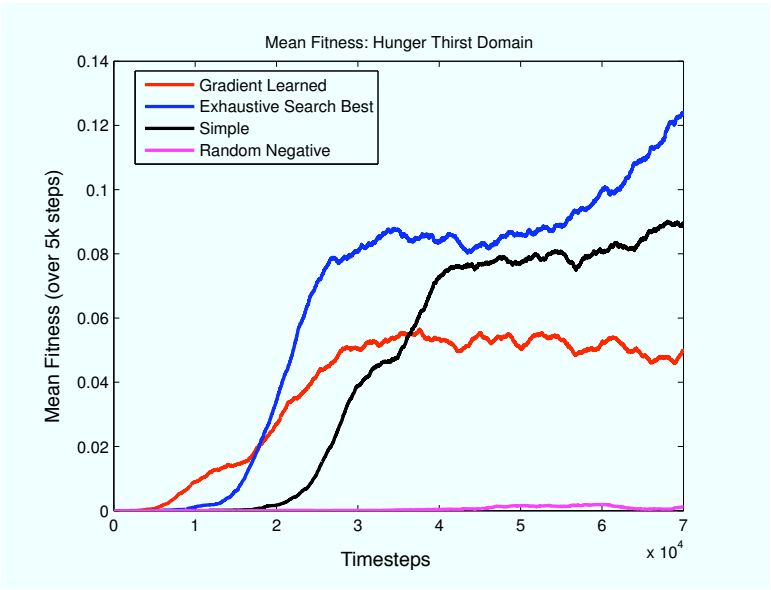

Figure 12: Running Average Mean Fitness of different linear reward functions.

In Figure 11 we compare the performance, as cumulative fitness, of a gradient learned reward function (critic) to other known reward functions for this domain. The *Exhaustive Best* reward function is the one found through an exhaustive search of the discretized reward function space by Singh et al. (2009). The *Random Negative* has reward randomly assigned between $[-0.1,0)$. Finally, the *Simple* reward function is based on the fitness function, and has reward of 1.0 for becoming not-hungry and -0.1 everywhere else. We also show these same results in Figure 12 with performance displayed as average fitness over a sliding window throughout the agent's lifetime.

These results show that, starting from a randomly initialized reward function, gradient ascent critic optimization is able to find a reward function that does competitively well against reward functions either hand crafted or found through exhaustive search. Although this took over 100 iterations its run time is still measured in minutes and not hours, making it significantly faster than either exhaustive search or genetic algorithms approaches. The comparison of cumulative sums of the fitness received over an agent's lifetime, Figure 11, show most accurately how well the

optimization has worked. The performance function evaluated total fitness over a lifetime, and by this measure, the gradient ascent method's reward function is almost as successful as the two non-random reward functions. This is an especially notable performance because the derivatives are computed over trajectories, which are only samples of the state space. A model based approach can be viewed as a trivial extension of this method in which every state-action is taken at every time-step. Doing so would also have the advantage of speeding up the process because nearly half of the runtime is spent generating the lifetime trajectory on each iteration.

5.3 Boxes Domain

The Boxes domain has the largest state space of the three domains we consider, but as we will discuss later is actually an easier task. The setting is an identical gridworld as in the Hunger-Thirst domain, shown in Figure 10. However, instead of food and water being located in the corners of the environment there are two boxes, each behaving identically. A box starts out closed, and an action can be choosen to *open* the box, at which point it will become *half-open* and the food hidden within the box will be exposed. The box will spend one time step in the *half-open* state before becoming *fully-open*, and when this happens the food also disappears. Thus, instead of the *drink* action there is an action called *open* which will transition a box from the *closed* state to *half-open*, during this time the agent must immediately choose the *eat* action to consume the food. Because the state space must include the states of the boxes, the state space and action space for this domain is larger than for the Hunger-Thirst domain.

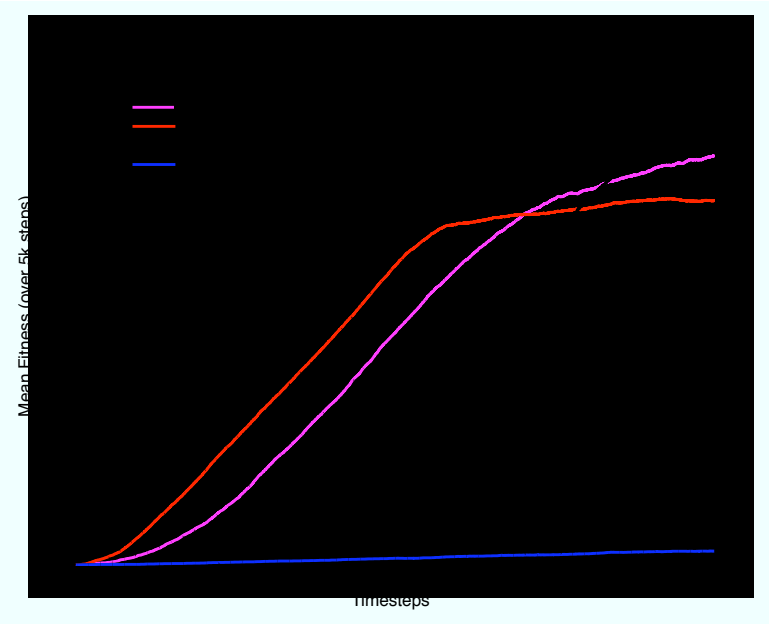

Figure 13: Running Average Mean Fitness of different reward functions on the Boxes domain.

We ran our gradient ascent algorithm for 30 lifetimes to find an optimized reward function, and then compared this reward function, in the same manner as above, with the reward function found through exhaustive search by Singh et al. (2009); a simple hand crafted reward function based on fitness, and the random reward function that was the starting point of our optimization algorithm. This comparison is shown, averaged over 30 randomly sample environments from the Boxes domain, in Figure 13.

We see that the optimized reward function establishes an early lead, but converges to a lower average fitness than the two non-random reward functions. Figures 15 & 16 illustrate why this happens. The gradient algorithm has optimized performance for some of the environments well over that of others, allowing it to reach higher performance much faster on those it optimized for, but unable to learn in the ones it did not optimize for. The result is a similar average performance as the reward function found through exhaustive search, but with higher variance and a preference for quickly accumulating fitness early in the lifetime.

A reasonable question at this point would be, 'Why is the optimization algorithm optimizing one environment at the expense of others?' The answer is that we specified the performance function as the total fitness received over the lifetime of the agent, which means that, for our reinforcement learning agents, accumulating fitness as fast as possible in some of the domains is worth the sacrifice of a lower average fitness late in the lifetime. The total fitness received by the three best reward functions are: Simple Reward (11847), Best Exhaustive Search Reward (13374), Gradient Learned Reward (13833). Thus, based on the performance metric of total fitness received, which is what we were optimizing for, the gradient ascent algorithm found the best reward function, though it was quite close in performance

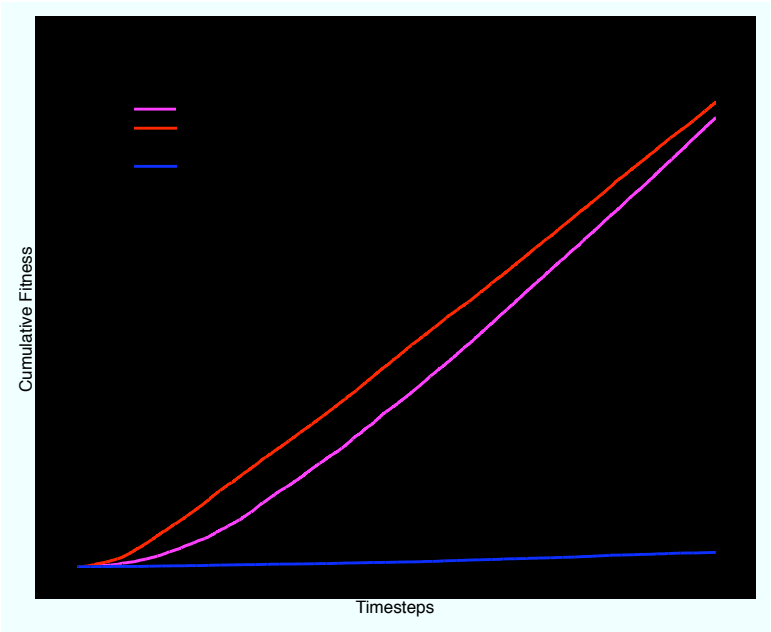

Figure 14: Cumulative Fitness of different reward functions on the Boxes domain.

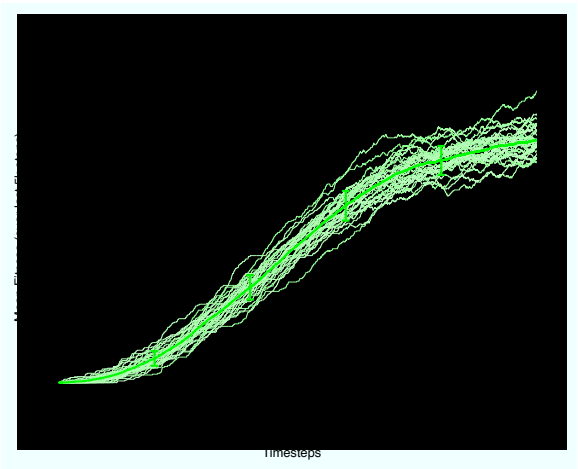

Figure 15: Performance of the best exhaustive search reward function on Boxes domain, with variance shown.

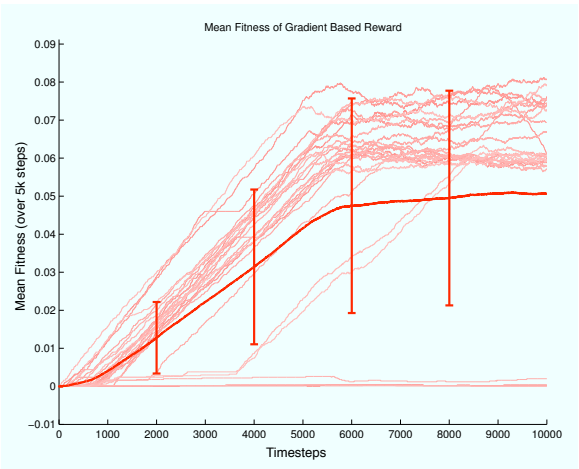

Figure 16: Performance of the gradient learned reward function on Boxes domain, with variance shown.

to that of the one found through brute force search. This is shown, by comparing the cumulative fitness achieved in a lifetime averaged over 30 runs, in Figure 14.

As previously mentioned, the Boxes domain turns out to actually be an easier task than the Hunger-Thirst domain. We will now demonstrate why this is true. Assume that the probability of a box closing is p, and that for a given environment, drawn from the boxes domain distribution, the two boxes are a distance d steps away from each other. Now, let us define a *camping* policy as one which stays at a given box and waits for it to close, compared to a *foraging* policy which goes from box to box opening each one and then moving to the next. Then, the probability, $P(\pi_c)$, of a *camping* policy being better (resulting in higher fitness) than a *foraging* policy can be computed as the probability of the current box closing before the agent would be able to reach the other box: $P(\pi_c|d) = \sum_t^d = 1(1-p)^{t-1}p$.

In the boxes domain there are four locations where boxes can exist (the four corners of the environment), and thus four different distances d between boxes. These are, with the probability of occurring: $d = 5, P(d = 5) = 1/3$; $d = 7, P(d = 7) = 1/6$; $d = 10, P(d = 10) = 1/3$; $d = 15, P(d = 15) = 1/6$. Thus, the probability of a *camping* policy out performing a *foraging* policy in the boxes domain is given by: $\sum_{d'=0}^{20} P(d = d')P(\pi_c|d')$.

The boxes domain presented by Singh et al. (2009) uses $p = 0.1$. Therefore, the probability that a *camping* policy is optimal is 0.684, which is greater than $1/2$ and therefore allows us to conclude that a *camping* policy is the optimal policy for an agent in the boxes domain. Thus, the optimal policy for an agent in the boxes domain involves finding

one box, sitting by it, and eating whenever possible. This is a far easier policy to learn than the optimal policy for the hunger-thirst domain, which requires the agent to move between boxes repeatedly.

6. Discussion

The performance of our gradient ascent method for critic optimization showed significant improvements over the random negative reward function which was used as the starting point. Moreover, the performance of our critic optimization method performed competitively with reward functions found through exhaustive search, despite requiring exponentially less computation time. We have demonstrated that these results are consistent across three domains of varying complexities. However, we have not yet applied gradient ascent critic optimization to a domain where a hand craft reward function would be unable to perform well.

Looking at the table based gradient with respect to reward provides some initial insights into ways to improve on our current method. The updates to the reward, which are based upon the gradient with respect to the Q-values have a tendency to either blow up or vanish. This phenomena is common in other applications of the method of Backpropagation Through Time (BPTT), to which this method is directly related. Fortunately, this suggests a direction for improvement. Long Short Term Memory (LSTM) has been used successfully by others in place of BPTT and was designed with these specific problems in mind (Hochreiter and Schmidhuber, 1997a,b).

One inescapable drawback to any gradient sample based method is that non-zero fitness must be observed in the agent's lifetime in order to have a gradient to compute. Thus, applying this method to very challenging domains, where even a reasonable guess at an initial reward function is not possible, may present problems. An example of this problem can be seen with the hunger-thirst domain. If we use a completely random reward function the agent is very unlikely to see non-zero fitness in a reasonable amount of time. This significantly impacts how many iterations are required for optimization because if no fitness signal is present then no gradient can be found. Additionally, through our experiments it became clear that reward function space has local extrema, and that these increase as the domain complexity increases. What this suggests is that we should pursue methods of overcoming local extrema in the future. Methods such as simulated annealing may work well, as could incorporating a global search method to move optimization from local extrema.

Through this research we have found a number of directions that can be taken to greatly improve performance. First, using SARSA(λ) instead of SARSA(0) will provide much more information to use when computing the gradient and at greater distances from the original fitness inducing states. This will likely improve the quality of learned reward functions significantly. Second, the current time and space complexity of our algorithm is $\mathbb{O}(T \times (C + A^2))$ and $\mathbb{O}(S^2 \times A)$ respectively, but this space requirement can be highly minimized by doing the computations with respect to the reward function parameters from the beginning. For these, T is the length of the agent's lifetime, C a constant, and A the number of actions. Finally, as mentioned before, LSTM has been used in related gradient computation methods and would likely be highly effective if applied to critic optimization.

7. Appendix A: Derivation of update rules

First, we need to get the performance function, ρ, into a format that provides more information at each time step. The reason becomes obvious if we follow the values that each derivative passes along. If we use the naive form of ρ, shown as the second line of Equation 9, then the value of $\pi_t(s,a)$ affects performance as much as the fitness at state s varies between different actions. In most cases the fitness functions we use are uniform across the actions for a single state. We will not go further into the details now, however, a full proof of the requirement that the performance function be non-uniform over actions can be provided upon request. We see below that maximizing the sum of fitness over the agent's lifetime is equivalent to maximizing the expected future non-discounted fitness of the start state, $f_T(s_0,a_0)$. Furthermore, we notice that maximizing $f_T(s,a); \forall s,a$ also maximizes sum of fitness, and thus the performance. Therefore, we use this formulation of the performance function because it provides a stronger gradient signal than alternatives and is easily computable.

$$\begin{aligned}
\rho &= \rho(R) && (9)\\
&= E_R[\sum_{t\geq 0}^{T} F(s_t,a_t)] && (10)\\
&= E_R[F(s_0,a_0) + \sum_{t\geq 0}^{T} F(s_{t+1},a_{t+1})] && (11)\\
&= E_R[f_T(s_0,a_0)|s_0,a_0] && (12)\\
&\rightarrow \sum_{s,a} f_T(s,a) && (13)
\end{aligned}$$

Next, we notice that because we are computing the gradient based upon samples received during the lifetime of the agent some correction for oversampling (due to policy preferences) need to be performed. However, because the policy changes throughout the lifetime of the agent, and thus the oversampling is not based upon any one time step's policy we compute the oversampling correction, $C^a_{s,s'}$, based upon observed probabilities during the lifetime and Bayes' Rule. This is shown in Equation 14.

$$C^a_{s,s'} = \frac{Pr(s'|s,a)}{Pr(s,a|s')} \tag{14}$$

$$= \frac{Pr(s')}{Pr(s,a)} \tag{15}$$

The foundation of our application of the chain rule is the approach called *ordered derivatives*, which in essence is just the repeated application of the chain rule; but, it does provide an easy to understand structure for the equations that we use. The ordered derivative equation is given in Equation 16, and describes how to apply the chain rule to compute the derivatives given by the dependency network shown in Figure 17. The *Target* in this case would probably be z_T. In our case, the target is ρ. Thus, we apply this structure to the dependency network for our policy-dependent SARSA agent, shown in Figure 2.

$$\frac{\partial^+ Target}{\partial z_i} = \frac{\partial Target}{\partial z_i} + \sum_{j>i} \frac{\partial^+ Target}{\partial z_j} \frac{\partial z_j}{\partial z_i} \tag{16}$$

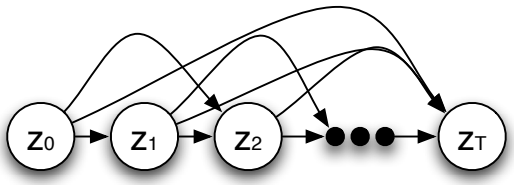

Figure 17: Simple ordered derivative network

The only non-zero explicit derivative of ρ is with respect to $f_T(s,a); \forall s,a$. Based on this assumption and our dependency network for our agent, we derive the ordered derivative equations, shown in Equations 18 - 23.

$$\frac{\partial^+ \rho}{\partial R(s,a,s')} = \frac{\partial \rho}{\partial R(s,a,s')} + \sum_{t>0} \frac{\partial^+ \rho}{\partial Q_t(s,a)} \frac{\partial Q_t(s,a)}{\partial R(s,a,s')} \tag{17}$$

$$= \sum_{t>0} \frac{\partial^+ \rho}{\partial Q_t(s,a)} \frac{\partial Q_t(s,a)}{\partial R(s,a,s')} \tag{18}$$

$$\frac{\partial^+ \rho}{\partial Q_t(s,a)} = \frac{\partial \rho}{\partial Q_t(s,a)} + \sum_{s',a'} \frac{\partial^+ \rho}{\partial Q_{t+1}(s',a')} \frac{\partial Q_{t+1}(s',a')}{\partial Q_t(s,a)} + \sum_{a'} \frac{\partial^+ \rho}{\partial \pi_t(s,a')} \frac{\partial \pi_t(s,a')}{\partial Q_t(s,a)} \tag{19}$$

$$= \sum_{s',a'} \frac{\partial^+ \rho}{\partial Q_{t+1}(s',a')} \frac{\partial Q_{t+1}(s',a')}{\partial Q_t(s,a)} + \sum_{a'} \frac{\partial^+ \rho}{\partial \pi_t(s,a')} \frac{\partial \pi_t(s,a')}{\partial Q_t(s,a)} \tag{20}$$

$$\frac{\partial^+ \rho}{\partial \pi_t(s,a)} = \frac{\partial \rho}{\partial \pi_t(s,a)} + \sum_{a'} [\frac{\partial^+ \rho}{\partial f_{t+1}(s,a')} \frac{\partial f_{t+1}(s,a')}{\partial \pi_t(s,a)} + \frac{\partial^+ \rho}{\partial Q_{t+1}(s,a')} \frac{\partial Q_{t+1}(s,a')}{\partial \pi_t(s,a)}] \tag{21}$$

$$= \sum_{a'} [\frac{\partial^+ \rho}{\partial f_{t+1}(s,a')} \frac{\partial f_{t+1}(s,a')}{\partial \pi_t(s,a)} + \frac{\partial^+ \rho}{\partial Q_{t+1}(s,a')} \frac{\partial Q_{t+1}(s,a')}{\partial \pi_t(s,a)}] \tag{22}$$

$$\frac{\partial^+ \rho}{\partial f_t(s,a)} = \frac{\partial \rho}{\partial f_t(s,a)} + \sum_{s',a'} \frac{\partial^+ \rho}{\partial f_{t+1}(s',a')} \frac{\partial f_{t+1}(s',a')}{\partial f_t(s,a)} \tag{23}$$

Now, we must compute the remaining explicit derivatives referenced by the above equations ($\frac{\partial Q_t(s,a)}{\partial R(s,a,s')}$, $\frac{\partial Q_{t+1}(s',a')}{\partial Q_t(s,a)}$, $\frac{\partial \pi_t(s,a')}{\partial Q_t(s,a)}$, $\frac{\partial f_{t+1}(s_t,a_t)}{\partial \pi_t(s_{t+1},a)}$, and $\frac{\partial \rho}{\partial f_t(s,a)}$). These come from taking the partial derivatives of Equations 1- 4.

$$\frac{\partial Q_t(s,a)}{\partial R(s,a,s')} = \begin{cases} \alpha C^a_{s,s'} & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

$$\frac{\partial Q_{t+1}(s',a')}{\partial Q_t(s,a)} = \begin{cases} 1 & \text{if } s=s', a=a', \text{ and } s \neq s_t \\ \alpha\gamma\pi_t(s,a)C^{a'}_{s',s} & \text{if } s=s_{t+1}, s'=s_t, a'=a_t, \text{ and } s_t \neq s_{t+1} \\ 1-\alpha & \text{if } s=s'=s_t, a=a'=a_t, \text{ and } s_t \neq s_{t+1} \\ 1-\alpha+\alpha\gamma\pi_t(s,a)C^{a'}_{s',s} & \text{if } s=s'=s_t=s_{t+1}, a=a'=a_t=a_{t+1} \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

In Equation 26, where π_t is the soft-max policy equation, we use the quotient rule of derivatives to find the explicit derivative of $\pi_t(s,a')$ with respect to $Q_t(s,a)$. There are two equations here based upon whether $a'=a$ or $a' \neq a$. With a little rearranging we are able to get them into a compact form that depends only on π_t. Note that τ is the temperature parameter of the soft-max equation.

$$\begin{aligned}
\frac{\partial \pi_t(s,a)}{\partial Q_t(s,a)} &= \frac{[\frac{\partial}{\partial Q_t(s,a)} e^{Q_t(s,a)/\tau}]\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau} - e^{Q_t(s,a)/\tau}\sum_{b\in\mathcal{A}} \frac{\partial}{\partial Q_t(s,a)} e^{Q_t(s,b)/\tau}}{[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} && (26)\\
&= \frac{e^{Q_t(s,a)/\tau}\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau} - e^{2Q_t(s,a)/\tau}}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} && (27)\\
&= \frac{e^{Q_t(s,a)/\tau}\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} - \frac{e^{2Q_t(s,a)/\tau}}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} && (28)\\
&= \frac{e^{Q_t(s,a)/\tau}}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]} - \frac{[e^{Q_t(s,a)/\tau}]^2}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} && (29)\\
&= \frac{1}{\tau}[\pi_t(s,a) - \pi_t(s,a)^2] && (30)\\
\frac{\partial \pi_t(s,a')}{\partial Q_t(s,a)} &= \frac{-e^{Q_t(s,a')/\tau}e^{Q_t(s,a)/\tau}}{\tau[\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}]^2} && (31)\\
&= \frac{1}{\tau}[\frac{e^{Q_t(s,a')/\tau}}{\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}} \times \frac{e^{Q_t(s,a)/\tau}}{\sum_{b\in\mathcal{A}} e^{Q_t(s,b)/\tau}}] && (32)\\
&= \frac{1}{\tau}[\pi_t(s,a') \times \pi_t(s,a)] && (33)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial f_{t+1}(s_t,a_t)}{\partial \pi_t(s_{t+1},a)} &= \frac{\partial}{\partial \pi_t(s_{t+1},a)}[(1-\alpha)f_t(s_t,a_t) + \alpha[F(s_t,a_t,s_{t+1}) + \gamma\sum_{a'}\pi_t(s_{t+1},a')f_t(s_{t+1},a')]] && (34)\\
&= \frac{\partial}{\partial \pi_t(s_{t+1},a)}\alpha\gamma\sum_{a'}\pi_t(s_{t+1},a')f_t(s_{t+1},a')] && (35)\\
&= \alpha\gamma f_t(s_{t+1},a)C^{a_t}_{s_t,s_{t+1}} && (36)\\
\frac{\partial Q_{t+1}(s_t,a_t)}{\partial \pi_t(s_{t+1},a)} &= \alpha\gamma Q_t(s_{t+1},a)C^{a_t}_{s_t,s_{t+1}} && (37)
\end{aligned}$$

$$\frac{\partial \rho}{\partial f_t(s,a)} = \begin{cases} 1 & \text{if } t=T \\ 0 & \text{otherwise} \end{cases} \tag{38}$$

With these equations we can now look at the pseudo-code which implements the computation of the derivative of ρ with respect to R.

8. Appendix B: Pseudo-Code

Below we include the pseudo-code for the sample based computation of the gradient. However, our experiments were run using python, and the python code itself is almost the same size and perhaps more readable. It is available upon request.

$\partial Q_t = 0$
$\partial Q_{t+1} = 0$
$\partial f_t = 0$
$\partial f_{t+1} = 0$
$\partial \pi_t = 0$
$\partial R_t = 0$

for $i \leq T; i = i-1$ **do**
  // Computing $\frac{\partial \rho}{\partial f_t}$
  $\partial f_t = \partial f_{t+1}$
  $\partial f_t[s_t, a_t] = (1-\alpha) * \partial f_{t+1}[s_t, a_t]$
  $\partial f_t[s_{t+1}, :] = \alpha * \gamma * \pi_t[s_{t+1}, :] * \partial f_{t+1}[s_t, a_t] * C^{a_t}_{s_t, s_{t+1}}$
  // Computing $\frac{\partial \rho}{\partial \pi_t}$
  $\partial \pi_t[s_{t+1}, :] = \alpha * \gamma * f_t[s_{t+1}, :] * \partial f_{t+1}[s_t, a_t] * C^{a_t}_{s_t, s_{t+1}}$
  $\partial \pi_t[s_{t+1}, :] += \alpha * \gamma * Q_t[s_{t+1}, :] * \partial Q_{t+1}[s_t, a_t] * C^{a_t}_{s_t, s_{t+1}}$
  // Computing $\frac{\partial \rho}{\partial Q_t}$
  $\partial Q_t = \partial Q_{t+1}$
  $\partial Q_t[s_t, a_t] = (1-\alpha) * \partial Q_{t+1}[s_t, a_t]$
  $\partial Q_t[s_{t+1}, :] = \alpha * \gamma * \pi_t[s_{t+1}, :] * \partial Q_{t+1}[s_t, a_t] * C^{a_t}_{s_t, s_{t+1}}$
  for $a \in ACTIONS$ **do**
    for $a' \in ACTIONS$ **do**
      if $a == a'$ **then**
        $\partial Q_t[s_{t+1}, a] += (1/\tau) * \partial \pi_t[s_{t+1}, a'] * (\pi_t[s_{t+1}, a] - \pi_t[s_{t+1}, a]^2)$
      else
        $\partial Q_t[s_{t+1}, a] += -1 * (1/\tau) * \partial \pi_t[s_{t+1}, a'] * \pi_t[s_{t+1}, a] * \pi_t[s_{t+1}, a']$
      end if
    end for
  end for
  // Computing $\frac{\partial \rho}{\partial R}$ update
  $\partial R[s_{t+1}, a_{t+1}, s_{t+2}] += \alpha * \partial Q_t[s_{t+1}, a_{t+1}] * C^{a_{t+1}}_{s_{t+1}, s_{t+2}}$
  $\partial Q_{t+1} = \partial Q_t$
  $\partial f_{t+1} = \partial f_t$
end for

References

Ackley, D. and Littman, M. (1991). Interactions between learning and evolution. *Artificial Life II (Proceedings Volume X in the Santa Fe Institute Studies in the Sciences of Complexity)*, X(487–509).

Barto, A. G. and Şimşek, Ö. (2005). Intrinsic motivation for reinforcement learning systems. *Yale Workshop on Adaptive and Learning Systems*.

Elfwing, S., Uchibe, E., Doya, K., and Christensen, H. I. (2008). Co-evolution of shaping rewards and meta-parameters in reinforcement learning. *Adaptive Behavior*, 16:400–412.

Hochreiter, S. and Schmidhuber, J. (1997a). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hochreiter, S. and Schmidhuber, J. (1997b). Lstm can solve hard long time lag problems. *Advances in neural information processing systems*, pages 473–479.

Kaplan, F. and Oudeyer, P.-Y. (2007). The progress drive hypothesis: an interpretation of early imitation. *Models and mechanisms of imitation and social learning: Behavioural, social and communication dimensions*, pages 361–377.

Niekum, S. (2010). Evolved intrinsic reward functions for reinforcement learning. *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*.

Oudeyer, P.-Y. and Kaplan, F. (2008). How can we define intrinsic motivation? *International Conference on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*.

Schembri, M., Mirolli, M., and Baldassarre, G. (2007). Evolving internal reinforcers for an intrinsically motivated reinforcement-learning robot. *Proceedings of the 6th International Conference on Development and Learning (ICDL)*.

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

Singh, S., Lewis, R. L., and Barto, A. G. (2009). Where do rewards come from? *Proceedings of the 31st Annual Conference of the Cognitive Science Society*.

Snel, M. and Hayes, G. M. (2008). Evolution of valence systems in an unstable environment. *In Proceedings of the 10th International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 12–21.

Sorg, J., Singh, S., and Lewis, R. L. (2010). Internal rewards mitigate agent boundedness. *In Proceedings of the 27th International Conference on Machine Learning*.

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

Uchibe, E. and Doya, K. (2008). Finding intrinsic rewards by embodied evolution and constrained reinforcement learning. *Neural Networks*, 21(10):1447–1455.

Werbos, P. J. (1994). *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley-Interscience.