

# Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout

Technical Report No. CS-TR-06-001

February 2006

Josh Beitelspacher, Jason Fager, Greg Henriques, and Amy McGovern

School of Computer Science  
University of Oklahoma  
Norman, OK 73019

Tel: 405-325-8446

Fax: 405-325-4044

E-mail: {jj\_beetle, jfager, greg.france, amcgovern}@ou.edu

---

# Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout

---

Reinforcement Learning, Sarsa( $\lambda$ ), Policy Gradient, Agent Learning, Applications and Case Studies, Artificial Neural Networks

## Abstract

This paper compares the performance of policy gradient techniques with traditional value function approximation methods for reinforcement learning in a difficult problem domain. We introduce the Spacewar task, a continuous, stochastic, partially-observable, competitive multi-agent environment. We demonstrate that a neural-network based implementation of an online policy gradient algorithm (OLGARB (Weaver & Tao, 2001)) is able to perform well in this task and is competitive with the more well-established value function approximation algorithms (Sarsa( $\lambda$ ) and Q-learning (Sutton & Barto, 1998)).

## 1. Introduction

Value function approximation has been applied to a wide variety of task domains and has demonstrated itself to be a strong solution for many difficult problems (Sutton & Barto, 1998; ?). While there have been a number of papers written about the theoretical justification for policy gradient methods in partially observable, continuous valued domains, (Williams, 1992; Sutton et al., 1999; Baxter & Bartlett, 2001), experimental data for these methods has typically been restricted to simple demonstration problems. In addition, very few papers have focused on empirically comparing policy gradient methods to traditional value function approximation methods in difficult environments. This paper compares the performance of policy gradient techniques with traditional value function approximation methods in a challenging problem domain.

Traditional reinforcement learning approaches allow an agent to determine its behavior in an environment

---

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

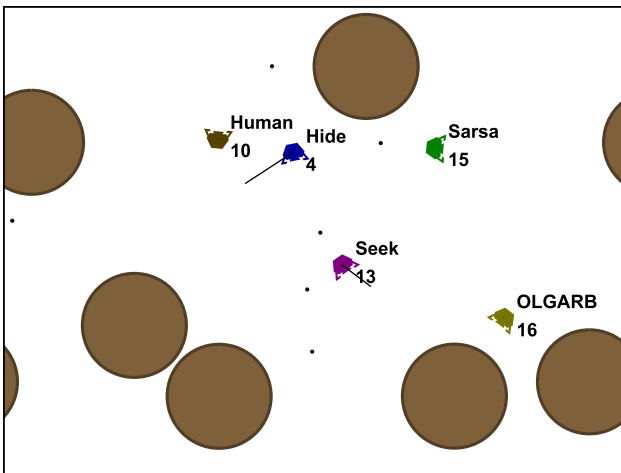


Figure 1. A screenshot from the Spacewar simulator. The black dots are bullets fired from ships. The lines indicate the current velocities of the ships. Numbers to the lower right of ships indicate the remaining health points.

by estimating the long-term expected value of each possible action given a particular state. Algorithms such as Q-Learning and Sarsa work by estimating this underlying value function, and choosing actions based on the highest estimated value.

Sutton et al. (1999) point out some of the theoretical drawbacks of value function estimation. Most implementations lead to deterministic policies even when the optimal policy is stochastic, meaning that probabilistic policies are ignored even when they would produce superior performance. Because these methods make distinctions between policy actions based on arbitrarily small value differences, tiny changes in the estimated value function can have disproportionately large effects on the policy. These problems directly relate to a demonstrable inability for value-function based algorithms to converge in some problem domains (Baxter et al., 2001).

An alternative method for reinforcement learning that bypasses these limitations is a policy-gradient approach. Instead of learning an approximation of the underlying value function and basing the policy on a direct estimate of the long term expected reward, policy gradient learning algorithms attempt to search the policy space directly in order to follow the gradient of the average reward. In addition to being able to express stochastic optimal policies and being robust to small changes in the underlying function approximation, under the appropriate conditions, policy gradient methods are guaranteed to converge (Sutton et al., 1999; Baxter & Bartlett, 2001).

While the theoretical advantages of policy gradient techniques are straightforward to demonstrate with particularly crafted tasks (Baxter et al., 2001), little is known about whether these advantages translate into superior performance for general tasks. For online implementations, the literature suggests that convergence time may be too slow to be of benefit in large tasks without the addition of significant domain knowledge (Tao, 2001; Baxter et al., 2001).

## 2. Spacewar Domain

The Spacewar problem domain is inspired by the classic game of the same name written by Stephen Russell at MIT in the early 1960s (Graetz, 1981). The original game is simple: a single star exerts a gravitational force on two ships in a toroidal space. A player can rotate her ship to the right or left, fire her ship’s thruster, jump into hyperspace, or fire a bullet. The game continues until one of the ships is hit with a bullet, or until the ships collide.

The version of Spacewar used in this paper is significantly different than the original version. Instead of a central sun exerting a gravitational force, the environment contains some number of large moving obstacles, while gravity is neglected. In addition, there can be more than two ships in a game. The amount of fuel and bullets available to each ship are limited, and regenerate at a rate that prevents their constant use. Ships are damaged and eventually destroyed by collisions with other objects. Obstacles can not be destroyed.

Each ship is controlled by a separate agent. At each timestep agents choose an action from a discrete set of actions. Possible actions include turning left, turning right, thrusting, shooting, or doing nothing. Combined actions are also possible. For example, an agent can turn left, thrust, and shoot at the same time. The 12 possible actions are listed in Table 1. The actions are designed so that a human player needs only 4 keys

Table 1. All possible actions in the Spacewar domain.

ACTION	
1.	DO NOTHING
2.	THRUST
3.	TURN RIGHT
4.	TURN LEFT
5.	FIRE
6.	THRUST AND TURN RIGHT
7.	THRUST AND TURN LEFT
8.	THRUST AND FIRE
9.	TURN RIGHT AND FIRE
10.	TURN LEFT AND FIRE
11.	THRUST, TURN RIGHT, AND FIRE
12.	THRUST, TURN LEFT, AND FIRE

to fully control a ship. Agents choose actions every 1/10 second.

The physics simulation follows traditional arcade-game physics. An object maintains its current velocity unless it is involved in a collision or fires it’s thrusters. A ship undergoes constant acceleration while thrusting, but turns at a constant angular velocity.

This problem domain is difficult for several reasons. Unlike many common reinforcement learning test problems, it has a large and continuous state space. Also, the fact that it is a competitive multi-agent environment adds complexity. Due to the way ships move, the environment often presents conflicting goals. For example, bullets fire in the direction that the ship is currently pointed, so steering the ship away from obstacles and firing at other ships cannot be viewed as independent tasks.

The simulator (Figure 1) was written in Java. The source code and compiled releases are available online.<sup>1</sup>

### 2.1. Task Definition

Success at Spacewar is defined by the ability of an agent to stay alive for as long as possible. A single game lasts up to five minutes, or until all ships have been destroyed.

We restrict the problem by setting the size of the space and the initial number of ships and obstacles. We use six obstacles with a five meter radius and five ships with a one meter radius. These 11 objects are randomly positioned in an 80 by 60 meter space such that there is a reasonable buffer distance between all

<sup>1</sup>Source code will be made available after anonymous review is completed.

the objects. The obstacles are also assigned random velocities between zero and three meters per second.

Ships are initially assigned 16 health points. A collision with a bullet takes away one health point, and a collision with an obstacle takes away one or more health points depending on the speed of the collision. Ships accelerate at six meters per second squared and turn at 150 degrees per second.

## 2.2. Heuristics

The learning agents were trained against four simple heuristics. The first heuristic is RANDOM, which chooses a random action at every time step. In a typical game, RANDOM will survive around 20 seconds. The DO NOTHING heuristic performs the DO NOTHING action at every timestep, and survives around 75 seconds.

The remaining two heuristics are slightly more complicated. The HIDE heuristic attempts to avoid other ships by hiding behind an obstacle. The SEEK heuristic takes a more aggressive strategy and chases and fires at other ships. SEEK survives an average of 100 seconds while HIDE only survives for around 70 seconds on average.

These heuristics are not challenging opponents for an experienced human player, but they provide reasonable training partners for the learning agents. Their performance also offers a good baseline for comparison.

## 3. Applying RL to Spacewar

The Spacewar task includes six obstacles and five ships moving through continuous space with continuous velocities. There is an effectively infinite number of possible states. While it is possible to break down the continuous state into a reasonable number of possible states, it is very difficult to do this in a way that retains all the important information.

We attempted to learn using Q-Learning and a discretized state space. The area surrounding the ship was partitioned by marking out ten distance rings around the agent at regular radius intervals, and ten wedges around the agent at regular angular intervals. Each sector in the resulting radial grid was checked for being occupied by up to two ships or obstacles. In addition to these external features, the agent’s speed was divided into ten bins. The reward signal corresponded to a variable “danger level” at each time step that didn’t involve a collision, and at those timesteps when collisions occurred, the reward was  $-5$  for each health point lost. The learning rate was set as  $\alpha = 0.1$ ,

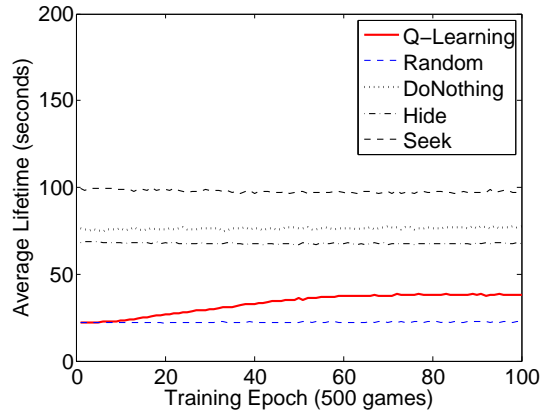


Figure 2. The average lifetime of the four heuristics and a Q-Learner based on a discrete state space.

and the discount factor was set as  $\gamma = 0.99$ .

The performance of this agent is shown in Figure 2 along with the heuristics. After 50,000 games, the agent had visited over 16,000,000 states, of which only 80,000 were unique. The agent was able to learn a better policy than RANDOM, but could not reach the level of performance of the better heuristics in a reasonable amount of time.

Given enough time and effort, it would probably be possible to design a state space that would include the necessary information to enable a learning agent to perform well. However, given the continuous nature of the Spacewar task, we hypothesized that function approximation would perform better, and would produce superior results with less time and effort than tuning the Q-learning parameters.

### 3.1. Value Function Approximation

Sarsa( $\lambda$ ) is a well known Temporal Difference (TD) learning method (Sutton & Barto, 1998). Unlike other TD learning methods, Sarsa( $\lambda$ ) has been shown to be stable when used with function approximation (Baird, 1995; Gordon, 2000). This property makes Sarsa( $\lambda$ ) an ideal algorithm to apply to Spacewar.

A backpropagation neural network is used to approximate the value function. The network has 51 input nodes for the state representation described in the next section, 80 hidden sigmoid nodes, and 5 linear output nodes that represent possible actions. Although there are 12 possible actions in the Spacewar domain, we learn on a subset of the available actions. The actions the Sarsa( $\lambda$ ) agent has access to are: DO NOTHING, TURN RIGHT, TURN LEFT, THRUST, and FIRE.

Decreasing the number of actions available increases learning speed significantly, and does not significantly damage the performance of the agent.

After an action is chosen, the TD error is attributed to the node representing the action. Backpropagation is used to compute the gradient of the squared TD error with respect to the network weights. The eligibility trace and weight update rules can then be expressed in terms of the gradient:

$$e_{ji} \leftarrow \gamma \lambda e_{ji} - \frac{1}{2 \cdot TD_{\text{error}}} \frac{\partial TD_{\text{error}}^2}{\partial w_{ji}}$$

$$w_{ji} \leftarrow w_{ji} + \alpha \cdot TD_{\text{error}} \cdot e_{ji},$$

where  $w_{ji}$  is the network weight from node  $i$  to node  $j$ , and  $e_{ji}$  is the corresponding eligibility trace. When  $\lambda = 0$ , this update rule is identical to the backpropagation update rule.

### 3.1.1. STATE REPRESENTATION

Choosing a good state representation is critical when using neural networks for reinforcement learning. In order to take advantage of generalization within the neural networks, we encode our state representation so that similar states have similar representations. This is done by converting from a global coordinate system to an egocentric coordinate system. In egocentric coordinates we can express the relative location of an object with a distance and an angle. Distances and angles have been used successfully as state representations in other domains (Stone & Sutton, 2001).

The state representation for a single obstacle consists of four values: the angle to the obstacle, the distance to the obstacle, the velocity of the obstacle toward the ship and the velocity of the obstacle around the ship. Two more values are used when encoding opponent ships: the angle from the opponent ship to the agent's ship and the health of the opponent ship. Figure 3 shows this representation.

Obstacles can not be destroyed, so there are always exactly six obstacles. Six obstacles can be encoded with four values each, for a total of 24 values. However, opponent ships can be destroyed as the game progresses. If an opponent is destroyed, it is represented in the state space as six zero values. In all, 24 values are needed to represent the opponent ships. Finally, three pieces of information about the agent's ship are added to the state space: health, fuel remaining, and magnitude of velocity.

Since obstacles play identical roles in the simulation, they can be sorted by distance without changing the meaning of the state representation. Ships do not play

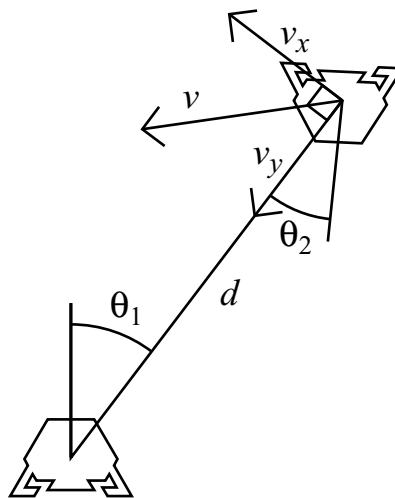


Figure 3. The state representation for a single opponent ship. A ship's location is described by  $\theta_1$ ,  $\theta_2$ ,  $d$ ,  $v_x$ , and  $v_y$ . The orientation of an obstacle is insignificant, so  $\theta_2$  is omitted when describing obstacles.

identical roles. Two opponents may have completely different playing styles. Even in this case, we choose to sort the opponent ships by distance. This step forces the neural network to play against all opponents in the same way. This allows a trained network to compete more effectively when playing against an unknown set of opponents.

### 3.1.2. REWARD STRUCTURE

The goal of the RL agent is to stay alive as long as possible so we give the agent a small but constant reward of one at every time step. We penalize the agent  $-5$  for each health point lost and  $-20$  for dying.

Although turning, thrusting, and firing are not negative actions for the agent to take, we slightly penalize them by giving a reward of  $-0.03333$  for any uses of these actions. Moving fast greatly increases the chance of a collision, so applying a small penalty to each thrust action discourages the buildup of excessive speed.

Turning the ship costs the agent nothing, so there is no strong reason to discourage it. However, if turning is not penalized the agent may turn constantly. While not necessarily a bad strategy, it is not how humans typically play the game. Turning for no reason makes the agent appear unintelligent. Giving a small penalty ensures that the agent only turns if it has a reason to do so. Firing is penalized for similar reasons.

### 3.2. Policy Gradient

Baxter and Bartlett (2001) describe a policy gradient reinforcement learning technique for partially observable Markov decision processes that involves making a biased estimate of the gradient of the average reward. Once the gradient has been estimated, adjustments to the policy parameters allow the learning agent to modify its behavior in such a way as to maximize the increase in reward from the environment.

The policy gradient method selected for implementation here is the OLGARB algorithm, given by Weaver and Tao (2001):

At each time step:

1. Observe state  $X_t$
2. Probabilistically select action  $A_t$  according to  $\mu(\cdot|X_t, \theta_t)$
3. Apply  $A_t$  and observe reward  $R_t$
4. Update variables:
 
$$B_t = B_{t-1} + \frac{R_t - B_{t-1}}{t}$$

$$Z_t = \gamma Z_{t-1} + \frac{\nabla_{\theta} \mu}{\mu}(A_t|X_t, \theta_{t-1})$$

$$\theta_t = \theta_{t-1} + \alpha(R_t - B_t)Z_t$$

In the above algorithm,  $\mu(\cdot|X_t, \theta_t)$  refers to the probabilistic policy determined by the current state  $X_t$  and policy parameters  $\theta_t$ . The average of the reward the agent has received across all timesteps is given by  $B_t$ , also called the average reward baseline.  $Z_t$  defines the gradient for each policy parameter, with prior gradient estimates contributing to current estimates at a rate determined by  $\gamma$ . Baxter and Bartlett (2001) describes  $\gamma$  as specifying a bias-variance tradeoff. Higher values of  $\gamma$  provide reduced bias in the gradient estimates, but also cause variance to increase.  $\alpha$  defines the learning rate for the agent.

OLGARB, like Sarsa( $\lambda$ ), is an online learning algorithm that attempts to improve its performance with each timestep. Note that this algorithm is a slight variation on the OLPOMDP algorithm introduced by Baxter et al. (2001). The difference is the introduction of the average reward baseline  $B_t$  in the update of the policy values. The advantage of OLGARB over standard OLPOMDP is that it reduces the variance associated with an increasing  $\gamma$  term while maintaining the bias reduction that such an increase provides (Weaver & Tao, 2001).

For this paper, the policy is encoded with a neural network that uses a *tanh* squashing function on the hidden nodes and has linear output nodes. The probabilistic policy is determined by normalizing the output

with a softmax function and interpreting the resulting values as probabilities. Updates to the policy are handled in a manner similar to backpropagation, except that the error term for each output node is given by:

$$err_j = o_j - P_j.$$

where  $o_j$  is 1 if the  $j$ th node was selected and 0 otherwise, and  $P_j$  is the probability of selecting the  $j$ th node. A detailed description of the backpropagation method used here is given by El-Fakdi et al. (2005).

#### 3.2.1. STATE REPRESENTATION

Initially, the state representation for the OLGARB learner was the same as that for the Sarsa( $\lambda$ ) learner. However, the OLGARB learner trained with this state representation could not learn effective strategies in the face of the sorted inputs and the sudden switch to six zeroes when an opponent agent died. In an effort to make the state space appear more continuous to the agent, the agent used a switching neural net, where a separate net is used for each number of opponent ships in the environment. Additionally, the obstacles and opponent ships are not sorted.

#### 3.2.2. REWARD STRUCTURE

The reward structure for the OLGARB learner was also initially the same as that used for the Sarsa( $\lambda$ ) learner. Once again, the OLGARB learner trained with the same settings as the Sarsa( $\lambda$ ) learner could not learn effective strategies. The reward structure was modified to strongly encourage the agent to minimize its speed, by penalizing the agent with the square of its velocity magnitude. Additionally, the penalties for losing health and dying were increased dramatically, from  $-5$  to  $-1000$  for each health point lost and from  $-20$  to  $-10000$  for dying.

This change makes the current learning task more similar to experimental learning tasks evaluated in the existing literature for online policy-gradient learning. Weaver and Tao (2001) demonstrate the effectiveness of OLGARB on the Acrobot task, where the reward signal is given as the height above the starting position. Baxter et al. (2001) describes a puckworld, where the reward signal is the negative distance between the puck and a goal state, as well as a combination of puckworld with a mountain climbing task, where the reward is given as a constant value on the lower plateau and as the negative velocity magnitude on the upper plateau. The trait that these experiments share is that the goal can be expressed by a single particular value of the reward function, and that the reward function is continuous with respect to the state.

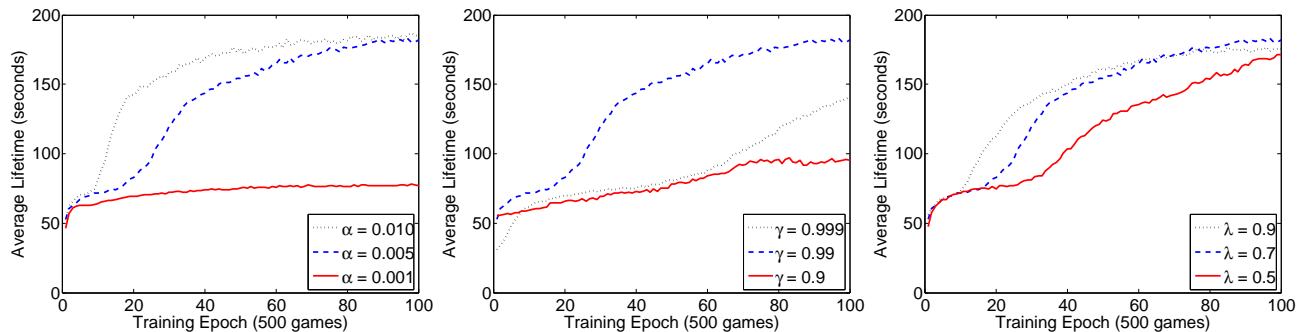


Figure 4. Sarsa( $\lambda$ ) learning results averaged over 30 runs of 50,000 episodes. (a) Varied learning rate with  $\gamma = 0.99$  and  $\lambda = 0.7$ . (b) Varied discount rate with  $\alpha = 0.005$  and  $\lambda = 0.7$ . (c) Varied eligibility trace decay rate with  $\alpha = 0.005$  and  $\gamma = 0.99$ .

In the same way, the restructured reward system provides the agent with a clear gradient for the policy to follow towards a strong heuristic player. The reward signal provides a continuous trail from policies that lead to high velocities to policies that attempt to minimize the speed of the agent’s ship. Since smaller velocity magnitudes are less likely to lead to catastrophic collisions, the performance of the agent correspondingly improves with the average reward. The addition of domain knowledge to the reward function of an OLPOMDP-based learner has been shown to provide significant improvement in convergence time (Tao, 2001); that proved to be the case here as well.

## 4. Results

In order to evaluate the performance of each algorithm, we recorded the lifetime of each agent in 30 trials of 50,000 games. We averaged the 30 trials, divided the results into 100 epochs of 500 games, and plotted the average lifetime of each agent during each epoch. Space was randomly initialized for every game, so there was a large variation in starting positions. Some starting positions were very good, while others were untenable. This led to a large standard deviation for all our results, and forced us to look at the average lifetime over a large number of games to see any patterns.

### 4.1. Value Function Approximation

Sarsa( $\lambda$ ) has three important parameters to tune: the learning rate,  $\alpha$ ; the discount rate,  $\gamma$ ; and the eligibility trace decay rate,  $\lambda$ . The final parameter,  $\varepsilon$ , or the probability of taking a random action, was set to 0.05 for all our experiments. Because actions are so short, 1/10 second, taking random actions 5% of the time should not significantly hurt performance.

After initial experimentation, we found values for  $\alpha$ ,  $\gamma$ , and  $\lambda$  that worked well on short sets of runs. These values were:  $\alpha = 0.005$ ,  $\gamma = 0.99$ , and  $\lambda = 0.70$ . We used these values as baseline parameters, and ran seven sets of 30 trials with varying parameters. Results are presented in Figure 4.

With all but two of the parameter combinations, agent performance was significantly better than any of the heuristics. Some agents were able to learn policies that kept them alive for close to three minutes, while none of the heuristics survived for even two minutes. Of the two combinations that performed poorly,  $\alpha = 0.001$  learned too slowly to achieve good performance, and  $\gamma = 0.9$  was too short sighted to learn a good overall strategy.

At the highest learning rate, 0.01, agents learned substantially faster than at the other parameter settings. Agents with these settings performed better than the heuristics after only 6,000 games.

### 4.2. Policy Gradient

The most important adjustable parameter in the OLGARB algorithm is  $\gamma$ , as this parameter directly affects the relative bias and variance of the agent’s gradient estimate. While the learning rate,  $\alpha$ , also plays an important role, the baseline value in OLGARB acts to adjust the effective learning rate by the difference between the current and mean reward. With a large variation in the reward signal, the learning rate needs to be small in order for the policy to converge.

As with Sarsa( $\lambda$ ), initial parameters were selected by evaluating the performance of various configurations in short test runs. We selected three test values,  $\gamma = 0.99$ ,  $\gamma = 0.9999$ , and  $\gamma = 0.99999999$ , and ran 6 sets of 30 trials. Three of the sets used just the defined  $\gamma$  values, while the remaining three added a weight decay

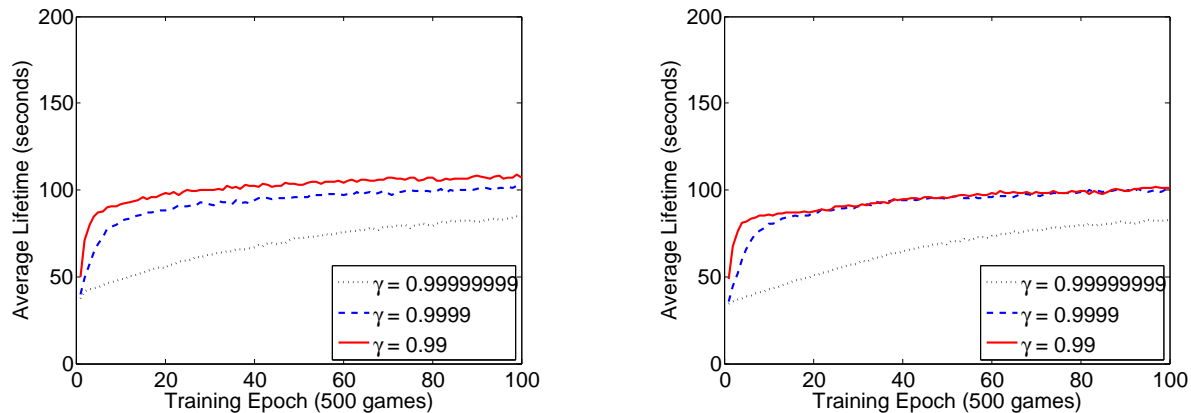


Figure 5. Performance of OLGARB with  $\gamma = 0.99$ ,  $\gamma = 0.9999$ , and  $\gamma = 0.999999999$ . (a) Without neural network weight decay. (b) With neural network weight decay.

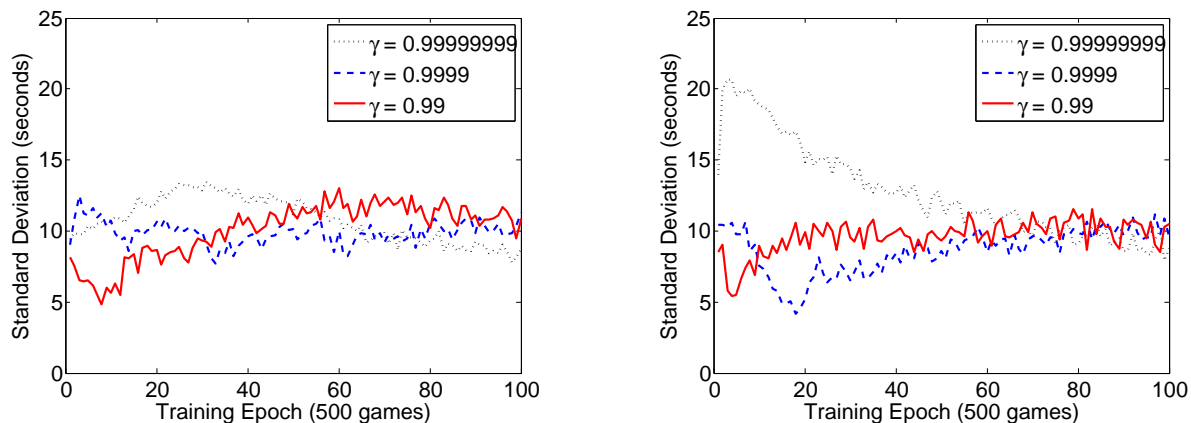


Figure 6. Standard deviation of OLGARB performance across 30 runs with  $\gamma = 0.99$ ,  $\gamma = 0.9999$ , and  $\gamma = 0.999999999$ . (a) Without neural network weight decay. (b) With neural network weight decay.

parameter in an attempt to avoid ‘wedge’ conditions at the beginning of the learning cycle (Baxter et al. (2001) describe the necessity for this in their discussion of the mountain/puck experiments). The performance for each of these runs is displayed in Figure 5, and the variance for each of these runs is displayed in Figure 6.

For the two smaller  $\gamma$  values, the agent is able to learn a good policy quickly. For the highest  $\gamma$  value, however, the rate of improvement is slow, and while it looks as though the agent is still improving at the end of all the trials, the maximum performance of the agent for the largest  $\gamma$  value is much less than the performance achieved by the other OLGARB agents after only a short time.

Figure 7 illustrates how the best OLGARB agent performs in comparison to the heuristics and the best Sarsa( $\lambda$ ) agent. The policy gradient learner surpasses the HIDE and DO NOTHING heuristics after only four to five thousand games. OLGARB is able to find a

policy that competes with SEEK more quickly than Sarsa( $\lambda$ ) is able to match DO NOTHING’s performance. At and beyond ten thousand games, however, Sarsa( $\lambda$ ) demonstrates a dramatic improvement over all of the other agents.

## 5. Conclusion

This paper demonstrates that a Sarsa( $\lambda$ ) learner was able to perform significantly better than an OLGARB learner in the Spacewar task domain. While OLGARB was able to achieve a higher level of performance more quickly than Sarsa( $\lambda$ ), Sarsa( $\lambda$ ) continued learning for a much longer time, and was ultimately able to learn a far superior policy.

The early success of OLGARB may have more to do with the modified state representation and reward structure than the change in algorithm. Like many real-world problems, Spacewar does not naturally supply an agent with a clear gradient to follow in order

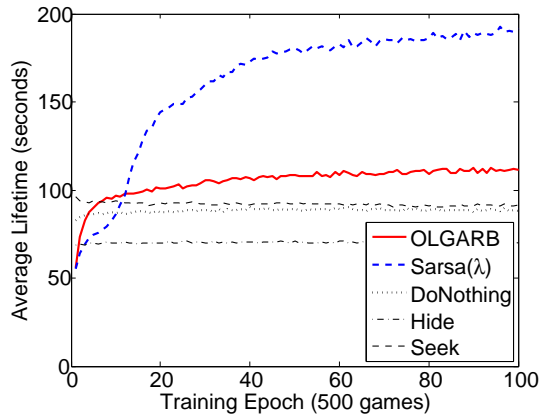


Figure 7. Learning results for competitive runs of Sarsa( $\lambda$ ) and OLGARB at their best parameter settings. Both learning agents survived longer in this test than in the individual tests.

to improve its performance. In order for policy gradient to be used efficiently in this problem domain, it was necessary to use significant prior knowledge to set up a reward structure that presented the agent with a gradient. This process could easily lead to errors. A poorly setup reward structure could lead to policies that may be good at collecting reward, but are far from optimal at the primary task.

In contrast, when using Sarsa( $\lambda$ ) we set up the reward structure to reinforce exactly the behavior we wanted, but we did not have to tell the agent how to achieve the goal. In this sense, Sarsa( $\lambda$ ) was much easier to work with, and is in general less prone to errors resulting from inappropriate reward configurations.

Currently, the theoretical benefits of policy gradient techniques appear to be outweighed by the difficulties posed in their application to real world problems. In contrast, value function approximation algorithms have a proven history of success on difficult problems, despite theoretical flaws.

## References

Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. *Proceedings of the 12th International Conference on Machine Learning* (pp. 30–37). Morgan Kaufmann, San Francisco, CA.

Baxter, J., & Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15, 319–350.

Baxter, J., Bartlett, P. L., & Weaver, L. (2001). Ex-

periments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15, 351–381.

El-Fakdi, A., Carreras, M., & Ridao, P. (2005). Direct gradient-based reinforcement learning for robot behavior learning. *ICINCO* (pp. 225–231).

Gordon, G. J. (2000). Reinforcement learning with function approximation converges to a region. *Advances in Neural Information Processing Systems* (pp. 1040–1046). MIT Press.

Graetz, J. M. (1981). The origin of spacewar. *Creative Computing*, 56–67.

Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. *Proceedings of the 18th International Conference on Machine Learning* (pp. 537–544). Morgan Kaufmann, San Francisco, CA.

Sutton, R., McAllester, D., Singh, S., & Mansour, Y. (1999). *Policy gradient methods for reinforcement learning with function approximation* (Technical Report). ATT Labs.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA, USA: MIT Press.

Tao, N. (2001). Walking the path: An application of policy gradient reinforcement learning to network routing. Bachelor’s thesis, Australian National University.

Weaver, L., & Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference* (pp. 538–545).

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 229–256.