Reinforcement Learning Project 2

Peter Lucia
plucia3@gatech.edu

March 14, 2020

Git hash: 9c6250b060d4d0827509ace87b8776b83f663dad

1 Abstract

In this work, I implement a Deep Q-Learning algorithm that learns to pilot a Lunar Lander inside the OpenAI gym environment with the goal of landing between two goal posts on the Lunar surface. Using Deep Q-Learning with experience replay, the algorithm performs nonlinear function approximation to achieve optimal action value approximation in the continuous domain of its environment. I discuss the experiments and analyze the effect of three hyper-parameters: the proportion of random actions taken, ϵ; the learning rate, α; and the discount factor, γ.

2 Background

At the beginning of the game, the Lunar Lander agent is positioned to make its descent toward the surface. There are eight continuous parameters that form the state space and four discrete possible actions that allow the lander to either fire one of its three directional engines or do nothing. The state space is described by the lander's x and y coordinates, velocity in the x and y directions, angle θ, angular velocity $v\theta$, and leg_L and leg_R, which indicate whether the legs are currently touching the ground.

The lander receives between +100 and +140 points as it approaches the pad, and between -100 and +100 depending on whether it comes to rest successfully between the flags or crashes. Firing engines incurs a small -0.3 reward penalty, but fuel is infinite. The game is won when the total reward over 100 consecutive runs is at least 200 points.

Since the environment's state space in this problem is continuous and therefore prohibitively large, model-based approaches like policy iteration and value iteration could be considered so long as careful sampling of the transition tuples from the continuous domain is done such that those transition tuples can be representative enough of the environment and generalize well to new inputs. Linear function approximation methods such as fourier basis and polynomial basis can be more efficient in terms of data and computation, however taking into account interactions between features can be a hindrance [2]. Other approaches such as tile coding, which is a flexible and computationally efficient form of coarse-coding for multi-dimensional continuous spaces can be bolstered with techniques such as "hashing" to avoid the curse of dimensionality [2]. However, this is not much of an advantage in this problem since the feature space is fixed and small. Radial basis networks, which require substantial additional computation complexity, would not be a good choice for this problem because of the reduction in performance when there are more than two state dimensions [2].

Upon considering a model-free, Q-Learning approach, especially with regards to the established performance of the approach toward similar problems in the literature, we may find there are more suitable options than the previously mentioned approaches, provided we perform some modification to the algorithm. An off policy, unmodified Q-Learning approach is impractical for learning successful control policies from raw video data because the action-value function is estimated separately for each sequence without any generalization [5]. While the Lunar Lander environment does contain raw video data, the agent must learn similar control policies from an 8-dimensional continuous state space of highly relevant features. Therefore, the success of a deep Q-Network in [4] for the task of playing classic Atari games motivated exploration into adapting the deep Q-Network approach to the Lunar Lander problem at hand.

Deep learning is responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems [2]. The deep Q-Network agent is an artificial agent that can learn directly from high dimensional sensory inputs by using hierarchical layers of tiled convolutional filters that mimic processing in the human visual cortex [5]. With sufficient data, deep neural networks are often able to learn better representations than handcrafted features [4]. While a Deep Q-Learning algorithm is implemented and analyzed in this experiment, it is suggested that future iterations of this experiment explore the possible tradeoffs between linear methods such as tile coding, fourier basis, polynomial basis functions, or other model-free, off-policy methods such as least-squares policy iteration discussed in [8], and compare their performance to non-linear methods like deep Q-Learning for the Lunar Lander problem.

3 Methods

In constructing the full connected, feedforward neural network, the rectified linear activation function (ReLU),

$$f(x) = max(o, x)$$

is used at the first and second dense layers. A linear activation function

$$f(x) = cx$$

is used at the output layer to avoid forcing positive action values on negative reward transitions. Figure 1 from [2] resembles the network constructed for this experiment and shows a generic feedforward neural network with four input units, two output units, and two hidden layers. The network of this experiment is also a feedforward network but instead has only one hidden layer with 64 units, an input layer with 64 units, and an output layer with 4 units. Each unit of the

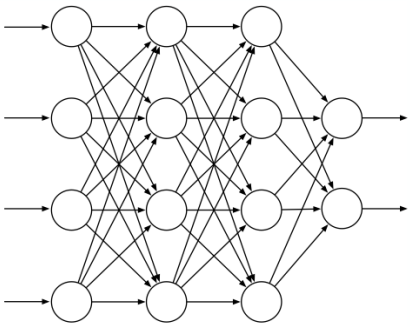

Figure 1: Generic Feed Forward ANN from [2]

output layer is intended to correspond to the action-value associated with doing nothing, or firing the left, right, or main engines.

If all units in a multi-layer feedforward neural network were to have linear activation functions, the entire network would be equivalent to one with no hidden layers. This would prevent the network from being able to to approximate practically any continuous function [2]. Therefore, the hidden layer and use of non-linear activation functions are instrumental in enabling the network to learn while exploring the stochastic lunar lander environment.

The input state space of the neural network is set to be equal to the square of the state space while the output of the network is set to equal the size of the action space. The output of the first two layers is set to the square of the size of the state space to form a simple kind of "feature map" based on the number of features or possible sub-features that the network should discern. A feature map detects the same feature no matter where it is located in the input and recognizes a specific pattern of activity [2]. An Adam optimizer is used to reduce the loss by refining the network weights and is set to use the mean squared error to quantify the loss of each prediction:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2$$

The Adam optimizer is selected because it has low memory requirements, is computationally efficient, and can take advantage of sparse features and obtain faster convergence rates than normal stochastic gradient descent with moments. Additionally, the Adam optimizer has also demonstrated good performance in practice and compares favorably to other stochastic optimization methods [6]. After performing hyper parameter tuning, the optimal learning rate selected for the Adam optimizer was 0.001. More discussion on this portion of the experiment can be found in section 4. The Deep Q-Learning with Experience Replay algorithm is adapted from [4]:

In this algorithm the agent experience tuple,

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

forms the entire replay memory,

$$D = e_1, ..., e_N$$

and

$$a_t = \text{argmax}_a Q^*(\phi(s_t), a; \theta)$$

Algorithm 1 Deep Q Learning with experience replay

for episode in episodes **do**
  $s_t \leftarrow$ reset OpenAI gym environment
  terminated $\leftarrow$ false
  steps $\leftarrow 0$
  while ($\neg$terminated) and steps $<$ maxEpisodeSteps **do**
    with probability ϵ, randomly choose a_t from s_t
    otherwise select $a_t = \text{argmax}_a Q^*(\phi(s_t), a; \theta)$
    Execute a_t and observe s_{t+1}, r_t
    add the transition tuple, (s_t, a_t, r_t, s_{t+1}) to D
    $s_t \leftarrow s_{t+1}$
    experienceReplay()
    steps $\leftarrow$ steps + 1
  end while
  $\epsilon = \epsilon \cdot$ epsilonDecayFactor
  $\epsilon = \max(\epsilon,$ minimumEpsilon)
end for

returns the action maximizing the action-value produced by a non-linear function approximator (in this case a neural network) having function $\phi(s_t)$ and weights θ [4]. As is shown in Algorithm 1, the agent randomly chooses a_t from s_t with probability ϵ, otherwise selecting a_t based on the action maximizing future reward, which is determined by $\text{argmax}_a Q^*(\phi(s_t), a; \theta)$. Note that the value of ϵ is decayed by an epsilonDecayFactor after each episode. This strategy is also known as an ϵ-greedy approach, because the agent makes tradeoffs between exploring the environment and exploiting what it already knows about the environment with probability ϵ. Also, because the agent takes this ϵ-greedy approach and updates the neural network weights regardless of whether it actually exploits that knowledge at each step, it is an off-policy approach. The experience replay component also necessitates the categorization of the algorithm as off-policy because the randomly selected samples that form each replayed experience are used to update $Q_j^*(\phi_{j+1}, a'; \theta)$, which is used to train the neural network instead of directly dictating policy via action selection.

With random sampling from memory and replaying of experiences, the experience replay component of the overall agent training procedure in Algorithm 1 forms what Mnih et. al in [4] coined, the deep Q-Learning algorithm. Adapted from [4] for training the Lunar Lander, the experience replay algorithm is as follows:

Algorithm 2 Experience Replay

if batchSize $>$ numExperiences **then**
  return
end if
samples $\leftarrow \{e_j, e_{j+1}, ..., e_{batchSize}\}$ from D.

$$Q_j^*(\phi_{j+1}, a'; \theta) = \begin{cases} r_j, & s_t = \text{terminal} \\ r_j + \gamma \text{max}_{a'} Q^*(\phi_{j+1}, a'; \theta), & s_t \neq \text{terminal} \end{cases}$$

$y_j = \text{max}_a Q_j^*(\phi_{j+1}, a; \theta)$
Train the neural network on y_j

The advantages of using experience replay are many. First, it improves data efficiency by using a single experience tuple in multiple neural network weight updates. Second by randomizing the samples, it reduces correlation between consec-

utive samples and overall variance of updates. And third, by preventing the training distribution from following the maximizing action through random sampling, it minimizes feedback loops and smoothes out the learning, which ameliorates the chances of getting stuck in poor local minima [4].

4 Results

Figure 2 shows the rewards during the training of the agent with $\epsilon = 1.0$, epsilonDecay $= 0.97$, minimumEpsilon $= 0.01$, discount factor, $\gamma = 0.99$, the learning rate, $\alpha = 0.001$, the batchSize $= 50$, and a replay memory, D, set to have a maximum size of 500000 experience tuples. It is clear from Figure 2 that the most dramatic increase in average reward occurred within the first 200 episodes, while afterward a slow increase helped push the agent's average reward over 200.

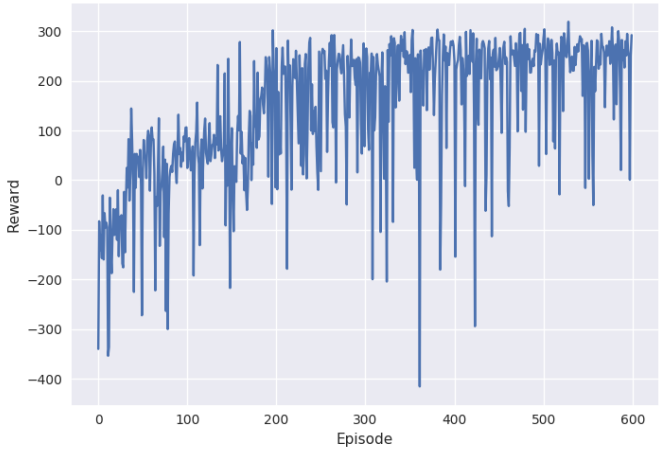

Figure 2: Rewards during training

Figure 3 shows the reward achieved by the trained agent. Of note is the stability of the per-episode reward, since out of 200 episodes, the agent only scored below 200 five times, and only once of those times received a negative reward. This demonstrates that model did not overfit either, and that a trained deep Q-Learner can consistently win the Lunar Lander game much like an experienced human player.

Figure 4 shows the average of the last one hundred episodes for two hundred episodes of testing the trained agent. Since fewer than 100 episodes have been run in the first half of the test, the average is taken over all previous episodes. Afterward, the mean reward is taken over only the previous one hundred episodes. The problem is considered solved when a score of 200 points or higher is achieved over 100 consecutive runs. Therefore, the results of Figure 4 demonstrate the algorithm's success in solving the lunar lander problem.

Choosing a starting value as well as the decay rate for ϵ is an important step because it defines how the agent makes tradeoffs between exploitation and exploration. If we set ϵ too low at the start, the agent will try to exploit its knowledge of the environment only having explored a small percentage of it, while if we set it too high, the agent might take random actions for too long without spending enough time refining its weights before the last training period has completed. Neural

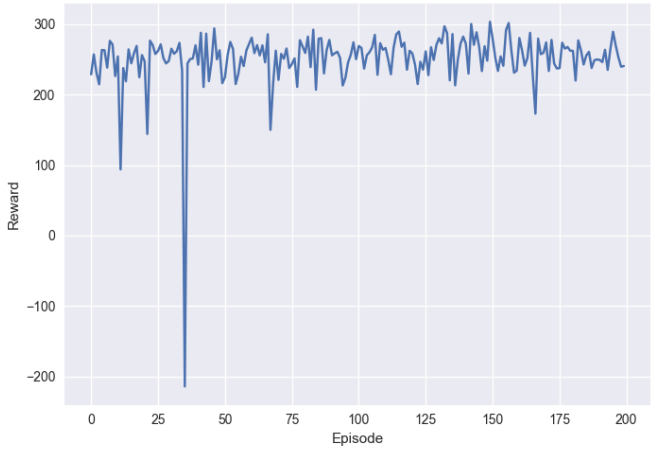

Figure 3: Rewards during testing of the trained agent

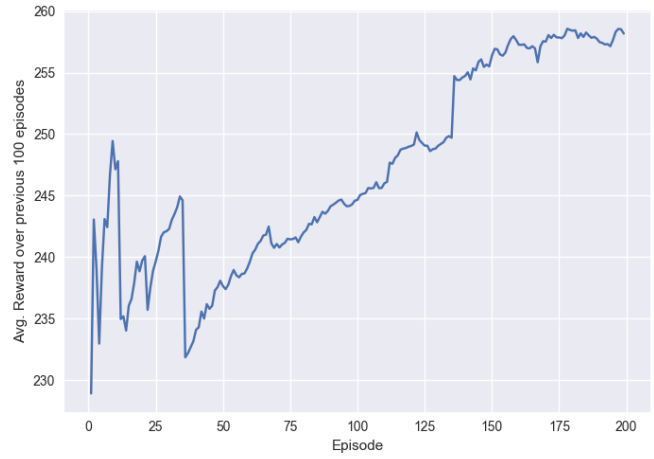

Figure 4: Avg Reward of last 100 episodes during testing of the trained agent

networks typically require longer training times [3]. Therefore many weight adjustments are needed during the exploitation phase before the weights will converge to optimal values that maximize reward.

Figures 5 and 6 show the effect of modifying ϵ between 0.50 and 1.0. Note that the epsilon decay rate was fixed at 0.97 for all of the hyper-parameter experiments. Examining these two figures without regard for the variability in the results, the optimal starting value for ϵ was 0.75. There is a clear increase in Reward as ϵ approaches 0.75 in Figure 6, followed by a slow decline, where each point falls within the standard deviation of the point with highest reward. Given the large standard deviation of per-episode reward, ϵ values below 0.75 were discarded, but those above it were still considered during the final optimization phase of the model.

Figures 7 and 8 show the effect of modifying the learning rate, α, between 0.001 and 0.1. The learning rate's role is to moderate the degree to which weights are changed at each update step during training [3]. It can be made to decay as the number of iterations increases, however no such decay was applied in this experiment. The learning rate is only

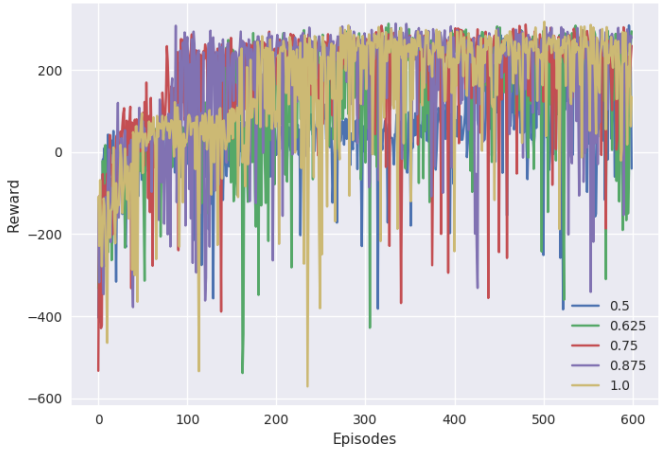

Figure 5: Reward for various starting values of ϵ

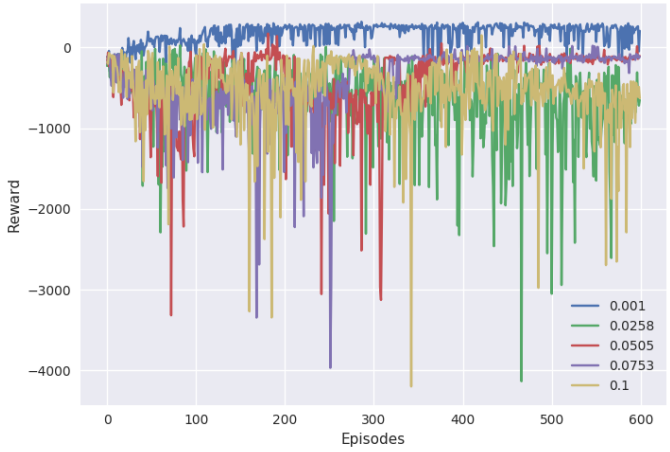

Figure 7: Reward for various values of α

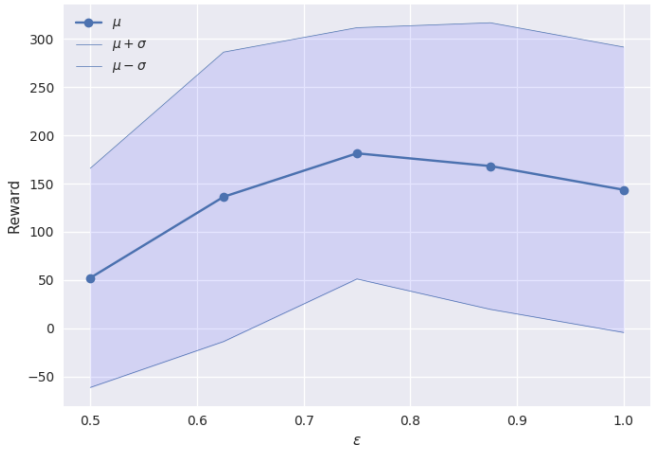

Figure 6: Mean Reward +/- σ over the last 100 episodes for various starting values of ϵ

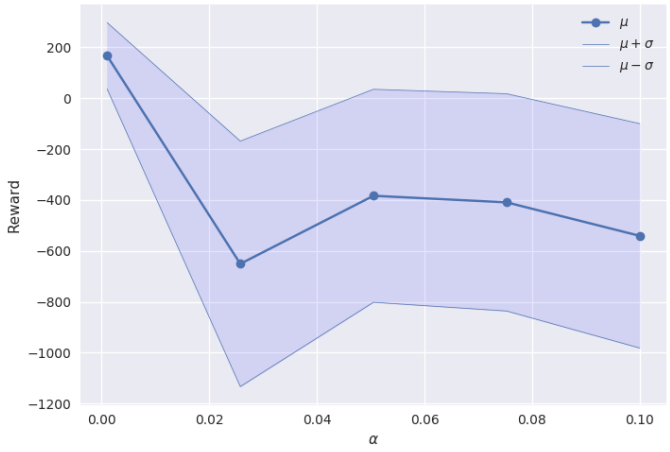

Figure 8: Mean Reward +/- σ over the last 100 episodes for various values of α

used when passed as a parameter to the constructor of the Adam optimizer during the initial configuration of the neural network model for training.

Figures 7 and 8 show that the best learning rate is 0.001. The average reward for larger learning rates was significantly lower, suggesting that a low learning rate is crucial in this experiment for the agent's success. Given the long training period and the large, continuous state space, it stands to reason that a low learning rate would be essential for the network to avoid overfitting.

Figures 9 and 10 show the effect of modifying the discount rate, γ, between 0.10 and 1.0. The discount factor tells the agent at each step how much importance to put on future rewards. A high discount factor will cause the agent to favor future rewards more, while a low discount factor will value them less. Examining Figures 9 and 10, it becomes clear that optimal discount factor was just under 1. There is a slight trend of increase in the mean reward in Figure 10. However, at $\gamma = 1$, the mean reward drops from between 0 and -100 down to around -900. The difference is also apparent in Figure 9. While we may expect that the agent should value the longest attainable reward at each state because a final large reward of +100 is given when the game is won, there are many smaller incremental positive rewards the agent must prioritize to get there. Because there is a small negative reward cost associated with firing its engines, the agent is also incentivized to land in the right spot, quickly. When the discount factor was set to 1, the agent tended to hover for longer, and did not show much urgency in finding the landing spot since it could receive the same reward in reaching it no matter how long it took to get there. The discount factor is used is in Algorithm 2 in the following manner:

$$Q_j^*(\phi_{j+1}, a'; \theta) = \begin{cases} r_j, & s_t = \text{terminal} \\ r_j + \gamma \text{max}_{a'} Q^*(\phi_{j+1}, a'; \theta), & s_t \neq \text{terminal} \end{cases}$$

Here, γ determines the weight of the maximum optimal action-values possible at a'. With $\gamma = 1$, we expect the agent to value all future rewards equally. By slightly discounting the value of future rewards with $\gamma = 0.99$, Figures 9 and 10 suggest that our Lunar Lander was better able to learn the urgency in landing by valuing the short term rewards more that guide it toward the landing area quickly and safely.

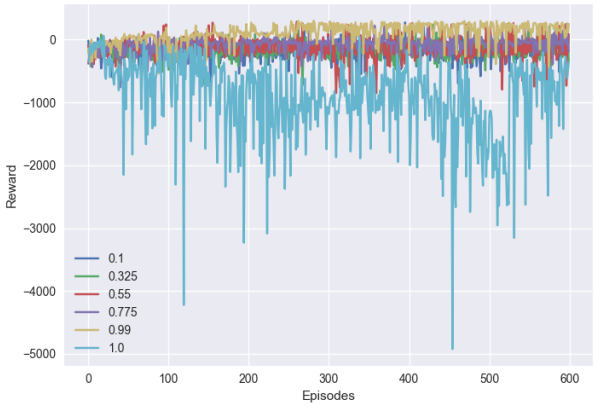

Figure 9: Reward for various values of γ

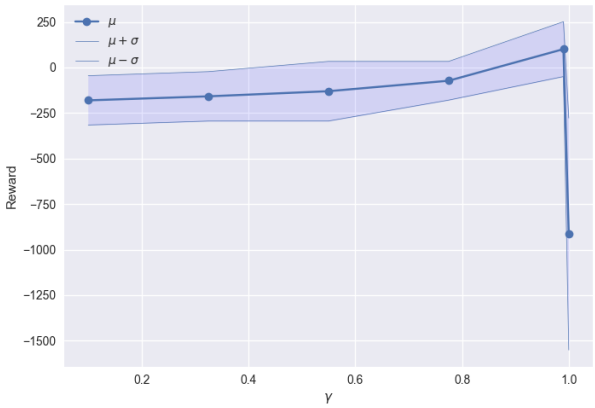

Figure 10: Mean Reward +/- σ over the last 100 episodes for various values of γ

5 Discussion

Early on in this experiment, hyperparameter tuning was computationally expensive, particularly because the deep Q-Learning algorithm implementation had not yet been completed and fully optimized with vectorization. Since the deep Q-Learning class was built to accept an Open AI gym environment as a parameter, it was helpful to solve the Cartpole-v1 problem first, before tackling the LunarLander-v2 problem. It is worth mentioning that even if the entire deep Q-Learning algorithm had been fully vectorized, the algorithm still requires about 600 episodes to train long enough to beat the game. Therefore, it is recommended that future work investigate ways of decreasing the number of episodes required to train a winning model. This may lead to other techniques entirely, since neural networks typically require longer training times.

With regards to optimization, another difficulty encountered during this experiment was determining how to vectorize each iteration over all actions in the last part of the experience replay algorithm where the values of each action for the current state are replaced by the values of the best action at the next best state. In Algorithm 2, this replacement is represented for an individual sample as an assignment to a new variable, y_j, on which the model is trained:

$$y_j = \max_a Q_j^*(\phi_{j+1}, a; \theta)$$

To test the speed of training upon vectorization, an incorrect vectorization implementation was applied to the last part of the experience replay algorithm. After observing how much faster the model would train on incorrect vectorization attempts of that last section, it became clear that a correct vectorization implementation would be a major time complexity improvement to the deep Q-Learning algorithm.

6 Conclusion

In this work, a Deep Q-Learning algorithm was adapted from [4] to perform non-linear function approximation and learn how to pilot a Lunar Lander to a specified location on a simulated 2D moon environment. After training, the deep Q-Learning agent successfully accrued a 100-episode average score of over 230 points per episode, winning the game by a safe margin. Pitfalls discovered using this approach include the long training times, and difficulties surrounding vectorization to improve those durations. When examining the effect of hyperparameters, the standard deviation of the reward was too high for there to be any clear advantages to selecting any particular value for a starting ϵ, provided that ϵ was at least greater than 0.75. This is certainly related to the chosen epsilon decay rate, and suggests that as long as the agent spends enough time exploring during training with a high ϵ, it can fully master the environment.

The best value for the learning rate, which moderates the degree to which weights are changed at each update step, was found to be 0.001. There was a clear increase in negative reward received and variance in reward received for learning rates greater than 0.001. The best discount factor, which determines how much emphasis the agent puts on future vs. immediate rewards, was found to be 0.99. This can be attributed to a higher need for the agent to learn from incremental rewards that guide it to the landing spot as opposed to equally valuing all future rewards.

It is suggested that future iterations of this experiment explore possible advantages to using linear function approximation methods such as tile coding, fourier basis, polynomial basis, or other model-free, off-policy methods such as least-squares policy iteration. Also, with improvements to the vectorization implementation of the experience replay algorithm, it is expected that training times could be optimized even further.

References

[1] Sutton, Richard S. "Learning to predict by the methods of temporal differences." Machine learning 3.1 (1988): 9-44.

[2] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[3] Mitchell, Tom M. "Machine learning." (1997).

[4] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[5] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[6] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[7] Chollet, François and others. "Keras" https://keras.io (2015)

[8] Lagoudakis, Michail G., and Ronald Parr. "Least-squares policy iteration." Journal of machine learning research 4.Dec (2003): 1107-1149.