# Final Project Report

ECE 6882 - Reinforcement Learning
Noah Rohrlich and Yonatan Beyene

## 1 Introduction

Temporal-difference (TD) methods for reinforcement learning (RL) combine features of Monte Carlo (MC) and Dynamic Programming (DP). In MC, the agent needs no knowledge of the distribution, but must complete an entire trajectory before updating policy. In DP, the agent updates its policy and value estimates after each step, but requires knowledge of the underlying distribution. TD combines the two by sampling an unknown distribution while also updating value estimates and policy after each step [4].

There remain many options for solving an optimization problem using TD: while in class we have focused on nested loop approaches that achieve limited performance with respect to convergence speed, eligibility traces provide a more computationally-efficient method for online value updates, especially when state spaces become very large. This project will attempt to solve the OpenAI Gym's Lunar Lander optimization problem using both tabular methods on a discretized state space, eligibility traces on a continuous space and deep Q-learning. The total reward, speed of convergence, and computation time will be compared for each method. This problem has been analyzed before using discretized state space and a SARSA algorithm as well as approximated in the continuous space with Deep Q-Learning [1], but with no discussion about online learning with linear approximation methods, eligibility traces and effect of hyper-parameters on the performance of deep Q-learning.

## 2 Background

The LunarLander-v2 environment in OpenAI Gym simulates a situation where a lunar lander needs to land at a specific location under low-gravity conditions. The environment created by OpenAI has a well-defined physics engine implemented, and is generated as shown in figure 1. The goal is to direct the lander to the landing pad quickly, softly, and fuel-efficiently as possible. The environment is digital and therefore has finite states, but in practice the state space is effectively continuous, as in real physics. The action space is discrete, however, with four options that will be outlined below.

### 2.1 State space

There are 8 state variables associated with the state space. These are:

- x coordinate of the lander
- y coordinate of the lander
- $v_x$ - the horizontal velocity
- $v_y$ - the vertical velocity
- $\theta$ - the orientation in space
- $\dot{\theta}$ - the angular velocity
- Left leg touching the ground (Boolean)
- Right leg touching the ground (Boolean)

When the episode is first generated, the lander will spawn at the top of the environment at a random x-axis location. All the coordinate values are given relative to the landing pad (the Cartesian origin (0,0) in this scenario). The x coordinate of the lander is 0 when the lander is on the line connecting the center of the landing pad to the top of the screen. Therefore, it is positive on the right side of the screen and negative on the left. The y coordinate is positive above the landing pad and is negative below.

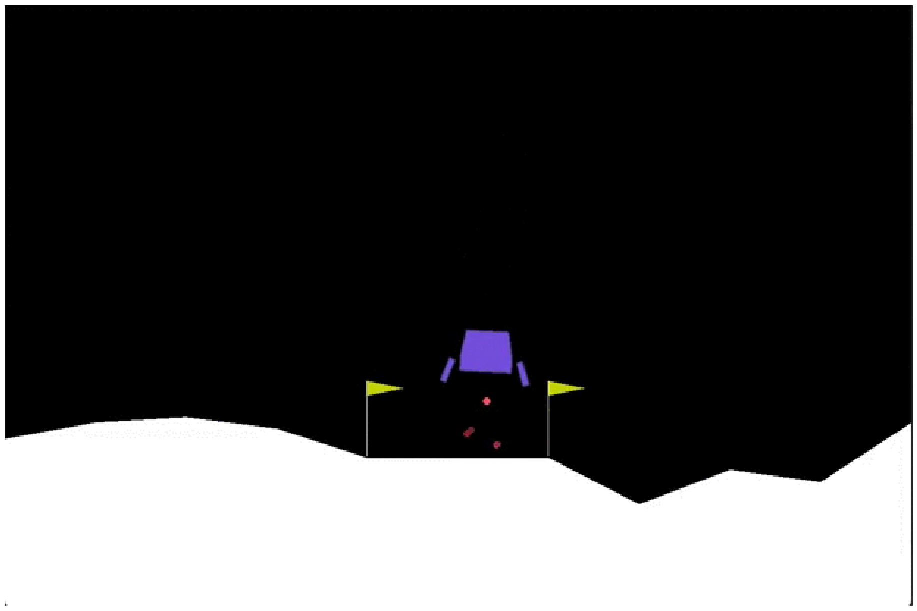


Figure 1: Visual representation of the Lunar lander problem

The state space is effectively continuous, with six floating-point variables and two booleans. We will compare tabular on and off-policy methods using a discretization of the state space against eligibility trace methods such as SARSA($\lambda$) and Advantage Actor-Critic (A2C) on the continuous state space.

## 2.2 Action space

There are four discrete actions available:

- do nothing
- fire left side engine
- fire right side engine
- fire main engine

Firing the left or right engine pushes the lander along the x-axis, but also introduce torque that rotates the lander, adding to the challenge.

## 2.3 Reward

Rewards are issued at each step for different reasons. At each step, dynamics are used to shape the reward according to the equation

$$r = -100 \times \left(\sqrt{x^2 + y^2} + \sqrt{v_x^2 + v_y^2} + |\theta|\right) \tag{1}$$

According to (1), this reward function penalizes distance from the landing pad (defined as the origin), higher velocities, and angle with respect to the vertical axis.

A successful landing anywhere is 100 points, but crashing is -100. Each lander leg making ground contact in a given step rewards 10 points. Using the main engine rewards -0.3 per step, while using the side engine rewards -0.03 per step. More points are rewarded if the lander comes to rest on both legs and at lower speed, so the agent is incentivized to use the side thrusters effectively. Fuel is infinite in the scenario, but the negative reward incentivizes lower fuel consumption.

## 3 Environment

To run the project, a new Anaconda environment that runs python had to be created. Then the necessary OpenAI Gym libraries and packages were installed, one of which–Box2D–proved troublesome on Windows systems due to dependency issues. After the packages were installed, we were able to simulate the game successfully.

## 4 Techniques

### 4.1 Temporal Difference methods

In class, we observed TD methods for solving the problem of predicting the values of non-terminal states. TD(0) used the difference in reward over one step.

$$V(s_t) = V(s_t) + \alpha \left[R_{t+1} + \gamma V(s_{t+1} - V(s_t)\right] \tag{2}$$

A single step may be insufficient to detect a trend of increasing value for a policy, so TD(n) was introduced where the value over several subsequent steps is estimated.

$$V_{t+n}(s_t) = V_{t+n-1}(s_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(s_t)\right], 0 \leq t < T \tag{3}$$

Where $G_{t:t+n}$ is the expected return over the next n steps.

$$G_{t:t+n} = \sum_{i=1}^{n} \gamma^{i-1} R_{t+i} + \gamma^n V_{t+n-1}(s_{t+n}) \tag{4}$$

SARSA($\lambda$) adds the mechanism of eligibility traces, where a set of basis functions capture the features of the value function. The traces identify how much each feature contributes to the value function overall. A trace-decay hyperparameter $\lambda \in [0, 1]$ defines the rate at which the trace goes to 0. Overall, this method provides computational advantages over the n-step approach because the full state space need not be known nor every solution measured.

Traces are initialized to zero, and updated according to

$$\boldsymbol{z_t} = \gamma\lambda \boldsymbol{z_{t-1}} + \nabla\hat{q}\left(S_t, A_t, \boldsymbol{w_t}\right) \tag{5}$$

In this case $\nabla\hat{q}\left(s_t, \boldsymbol{w_t}\right)$ signifies the approximation being made of the state-action pair value. The weight vector is updated as

$$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} + \alpha\delta_t \boldsymbol{z_t} \tag{6}$$

$$\delta_t = R_{t+1} + \gamma\hat{q}\left(S_{t+1}, A_{t+1}, \boldsymbol{w_t}\right) - \hat{q}\left(S_t, A_t, \boldsymbol{w_t}\right) \tag{7}$$

A newly-introduced approach, RLS-TD($\lambda$) is superior to conventional TD($\lambda$) algorithms in that it makes use of recursive least-square (RLS) methods to improve the learning efficiency and eliminates the step-size schedules [2]. While the results demonstrated substantial reduction in number of episodes to converge, the method was only applicable to state value estimation and could not account for actions, making it unsuitable for our purposes here.

#### 4.1.1 Discretization of Continuous States - TD(0)

Since six out of the eight state variables are continuous, discretization is needed to implement state-action TD algorithms. As a reference, the naïve method with 10 values was tested with one-step SARSA and Q-Learning.

The second discretization method involves two large generalization regions with a smaller quantization region around the origin, depicted in 2 for the x position variable.

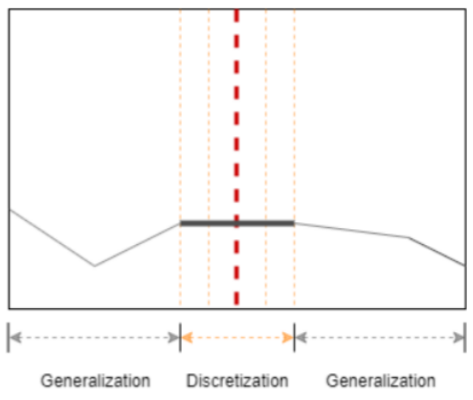


Figure 2: Optimized Discretization with Generalizations Outside the Landing Pad and Quantization Within

The example figure has four discrete zones across the landing pad coordinates. To discretize the x-coordinate with n=10 zones within the landing pad (which has width 0.5 ), we need a step size of 0.05 within the landing pad. Then, the function discretizing the x-coordinate is given as

$$d(x) = min\left(\left\lfloor\frac{n}{2}\right\rfloor, max\left(-\left\lfloor\frac{n}{2}\right\rfloor, int\left(\frac{x}{0.05}\right)\right)\right) \tag{8}$$

Since n=10 for our purposes, this means that the states will be quantized to integers between -5 and 5. States with x-coordinate value in the range [-0.25, 0.25] will be quantaized to $\frac{x}{0.05}$. States with x-coordinate value greater than x = 0.25 (far right) will be discretized as 5 and states with x-coordinate values less than x = -0.25 (far left) will bes discretized as -5. This means that states with x-coordinate values concentrated around the origin are discretized and those on far from the center are generalized to a single value. This can be done for all the other five continuous states as well.

### 4.1.2 Feature Construction - TD($\lambda$)

In order to use function approximation, a set of linearly independent basis functions must be selected. Sutton provides an algorithm for SARSA($\lambda$) that applies to those linear approximations using binary features (consisting only of 0 and 1). This led to implementing function approximation using tile coding [3].

Much like tabular methods, tile coding aggregates states using relatively low-resolution discrete spaces, known as "tiles". A sequence of adjacent tiles makes up a "tiling." Tile coding uses multiple offset copies of a low-resolution tiling to increase the resolution of the state aggregation. An example of tile coding for two-dimensional state is shown in figure 3.

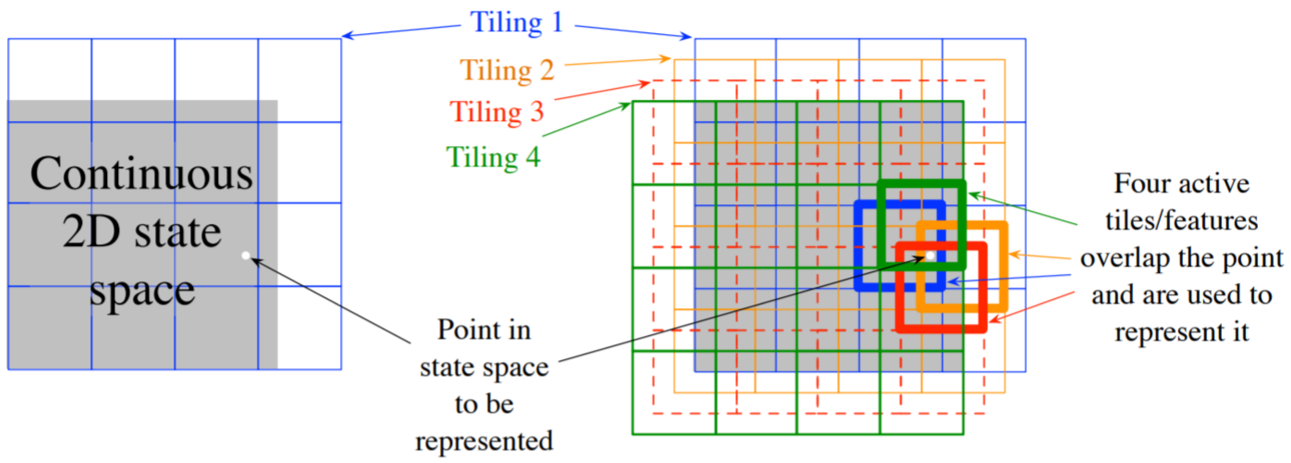

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 3: Tile coding example from Sutton and Barto [4]

Sutton provides an open-source tile coding implementation that uses index hash tables [3]. When a state is tiled, the hash table returns a set of indices that represent the active tiles. As new states are tiled, new indices are returned. The literal value of the feature indices is irrelevant, since the same index will be returned if the feature is active in a future state. As training continues across episodes, new states become less common, and the agent has usually identified preferred states.

Tile coding can be performed on a state with as many dimensions as needed, with increasing memory costs. With 6 continuous state variables in the Lunar Lander observation, each tiling could be made 6-dimensional at the cost of massively increasing the size with each added tile of resolution. If each dimension were 5 tiles long, the resulting tiling would have $5^6 = 15,625$ entries, then multiplied by the number of tilings used to increase resolution. Using multiple lower-dimensional tilings allows the approximation to use more tiles per dimension, then improve the state aggregation accuracy by increasing the number of tilings used.

Relevant variables were paired to create three two-dimensional state vectors:

- x and y position
- x and y velocity
- orientation and angular velocity

Sutton recommends with this implementation of tile coding using a power of two that is at least four times as many as the number of dimensions being tiled, therefore 8 tilings are used for each of the continuous pairs above, and the two discrete elements of the state vector (the lander leg status) need only one tiling of two states. Thus the total number of tilings is 8+8+8+1=25. This will be important for hyperparameter selection.

#### 4.1.3 Hyperparameters - TD($\lambda$)

One benefit of linear binary features is that hyperparameter estimation becomes quite simple. Tile coding will always produce the same number of active features, and the magnitude of those features will always be 1. Thus, according to Sutton, the step size $\alpha$ may be set according to

$$\alpha \doteq \left(\tau \mathbb{E}\left[\mathbf{x^T x}\right]\right)^{-1} \tag{9}$$

In this case, $\mathbf{x^T x}$ is a constant equal to the number of active features, in other words the sum of the number of tilings for each 2-dimensional state-pair mentioned above. $\tau$ indicates the number of training experiences for once complete training step to take place. This has been set to 10 for the experiment.

$$\alpha = \frac{1}{10 \times 25} = 0.004 \tag{10}$$

The other hyperparameter in question (the trace decay rate) must be evaluated through a search.

### 4.2 Advantage Actor-Critic (A2C)

A2C as a policy-gradient method has the distinguishing feature that it separates policy decisions from the state-value function (or, in this case, its tile-coding approximation). The policy derives the weights of its actions from a set of parameters labeled $\theta$, while the state-value function continues to use weights $\mathbf{w}$. This makes the policy function the "actor" and the state-value function the "critic," hence the name.

Instead of exploring using $\epsilon$-greedy policy, the parameter vector $\theta$ is used, since now certain actions are weighted more heavily than others. Given the state and known discrete set of actions, feature vectors $\mathbf{x}(s,a)$ are obtained from tile coding as mentioned above. With separate feature vectors for each possible action, a set of preferences is constructed such that

$$h(\theta, s, a) = \theta^T \mathbf{x}(s,a) \tag{11}$$

With these preferences, a softmax distribution is created to ensure that exploration remains a possibility, but less frequent as the agent learns.

$$\pi(a|s,\theta) = \frac{e^{h(\theta,s,a)}}{\sum_b e^{h(\theta,s,b)}} \tag{12}$$

For any actor-critic method, the gradient of the policy function is required, sometimes called the eligibility vector. Fortunately, for this softmax distribution with linear features, this gradient can be solved linearly, as shown by Sutton [4].

$$\nabla \ln \pi(a|s,\theta) = \mathbf{x}(s,a) - \sum_b \pi(b|s,\theta)\mathbf{x}(s,b) \tag{13}$$

Even when using binary linear features, no rule-of-thumb currently exists for selecting step sizes for parameters and weights. This will rely on extensive simulation.

## 4.3 Deep Q-learning

In Q-learning, value iteration is used to directly compute Q values and find optimal Q-function. When a system becomes complex with many states and action, this technique will not be ideal to use because memory required to save and update the table increases as the number of states increases and amount of time required to explore each state to create the required Q-table is not practical. Deep Q-learning uses function approximation to estimate optimal Q-function using deep neural networks. Using hidden layers in the neural network, we can have non-linear approximations. The states will be an input to the network and pass through a number of hidden layers. Then we will have an output layer with the Q-value per action.

The input layer of the network we implemented has a neuron for each state (8 neurons) and the output layer has a neuron for each action (4 neurons). We implemented two hidden layers with 64 neurons each. The Deep network we implemented is shown in figure 4.

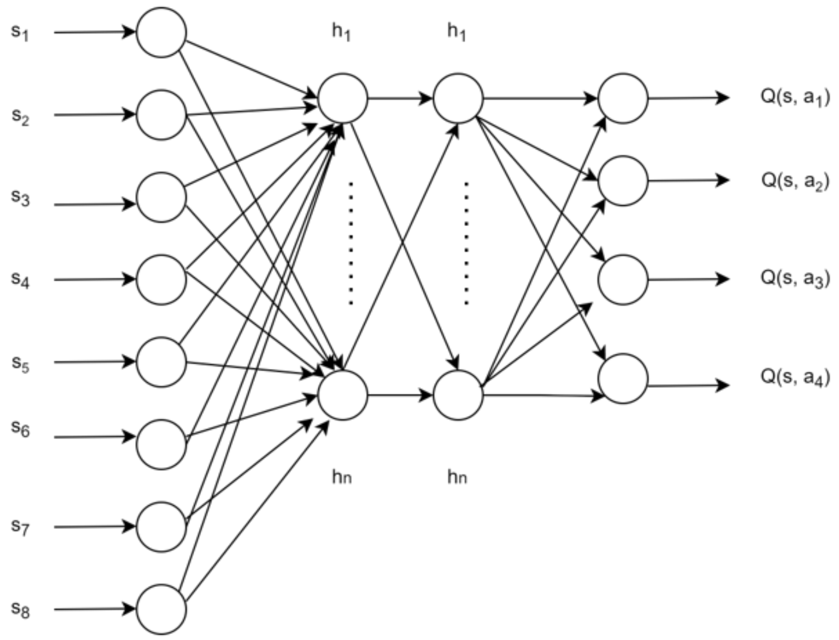


Figure 4: Deep Q-network [4]

After passing through the network and obtaining Q-values for all actions, we continue as usual. Approximating the optimal Q function becomes a regression task so gradient descent will be used to minimize the loss function. The squared error loss function at iteration i can be obtained as:

$$\ell_i = \mathbb{E}_{(s,a)\sim\mu}[(y_i - Q(s,a;\theta_i))^2] \tag{14}$$

where

$$y_i = \mathbb{E}_{(s,a)\sim\pi}[r + \arg\max_{a'} Q(s', a'; \theta_{i-1})] \tag{15}$$

$\theta_{i-1}$ are the network parameters like weights and biases from the previous iteration. Gradient descent is used to minimize the loss function. The derivative of the loss function with respect to each weight will be computed, then multiplied by the learning rate and added to the weight to update the weights. To balance exploration and exploitation, we utilized a basic $\epsilon$-greedy policy. We set $\epsilon$ to be 1 at the start of the simulation and set it to decay by 0.95 every iteration (100 iterations) until it reaches 0.01 after which it stays the same.

#### 4.3.1 Hyperparameters - DQN

It is important to analyze how hyperparameters affect the training process and the performance of the agent. For the deep neural network we implemented in this project, we tested the effect of batch size and learning rate hyperparameters of the neural network. Batch size is the number of state space that will be passed through to the network at one time. Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. For both hyperparameters, we compare the average reward obtained over 100 episodes. We found that different values of the hyperparameters have impacts on the learning process of the agent.

## 5 Results

### 5.1 Single-step Temporal Difference

Tabular methods appeared to achieve nonzero returns just after 8000 episodes, as shown in figure 5. With traditional uniform state quantization, off-policy learning performed noticeably better than SARSA while still providing a negative average return. The optimized discretization method allowed both on-policy and off-policy methods to eventually achieving net-positive returns around episode 8000, with little difference between on and off-policy methods in this case.

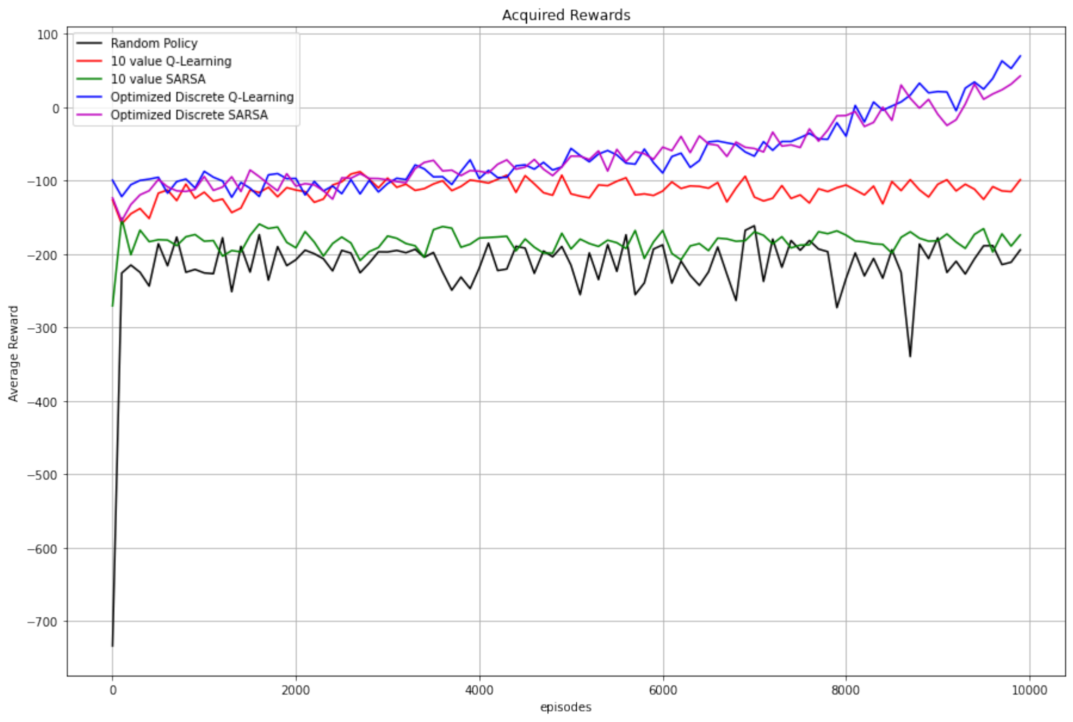


Figure 5: Average return per episode using one-step SARSA with different discretizations, compared to random policy.

## 5.2 SARSA($\lambda$)

Figure 6 shows the results from adding eligibility traces to the on-policy learning algorithm. Certain decay rates $\lambda$ resulted in positive average rewards from the very first episodes. It was observed that using a higher decay parameter, which allows the agent to retain more memory of earlier steps in the episode, incentivized training the lander to complete the episode in less time, with noticeably worse average rewards.

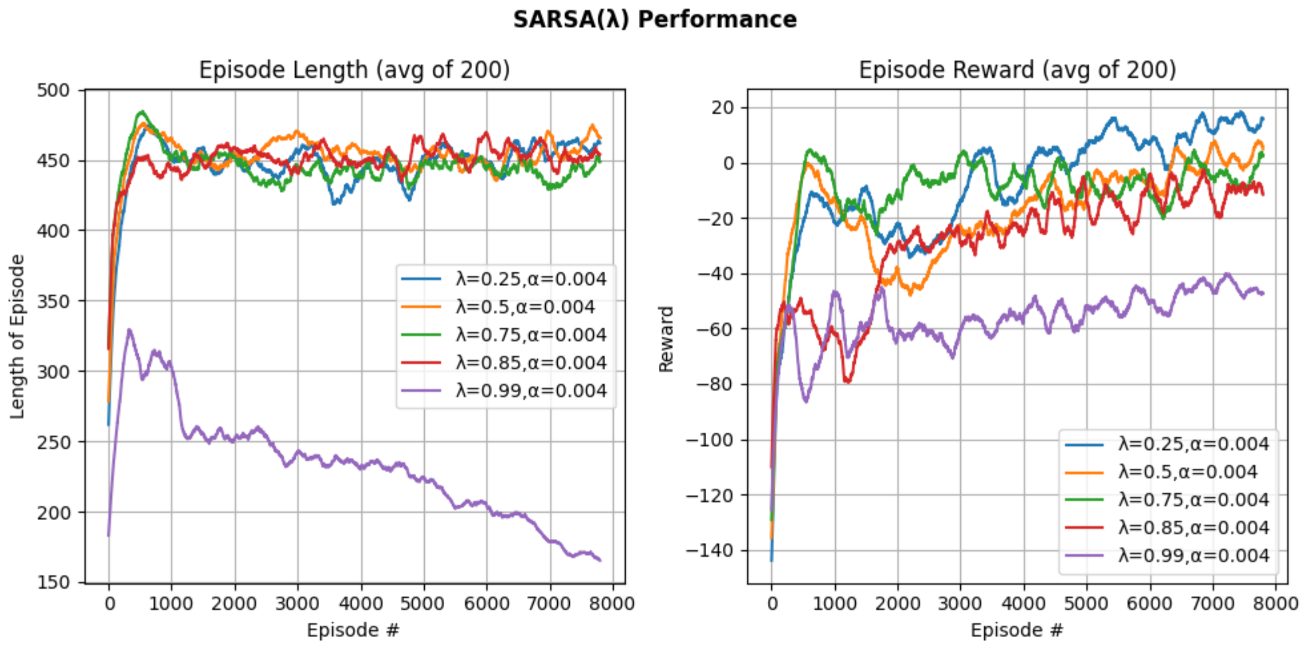


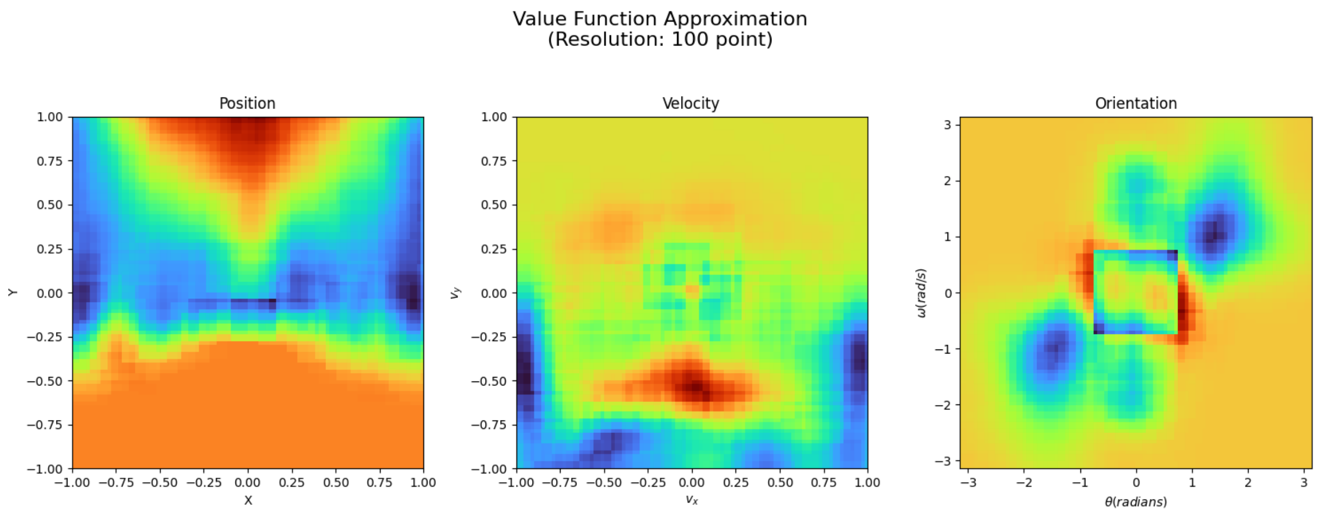
Figure 6: Average returns per episode for SARSA($\lambda$) with eligibility traces

The higher average reward associated with lower decay rates is deceptive, however, because the simulation is not actually being solved. Instead, the agent learns that its best option to have the lander hover in the air rather than attempt to land. To discern why this might be the case, the value function approximation using the established tiles and weights trained over 500-step episodes is reconstructed in figure 7, where the approximations using $\lambda = 0.75$ and $\lambda = 0.99$ are shown. Each figure has a map for position, velocity, and orientation.

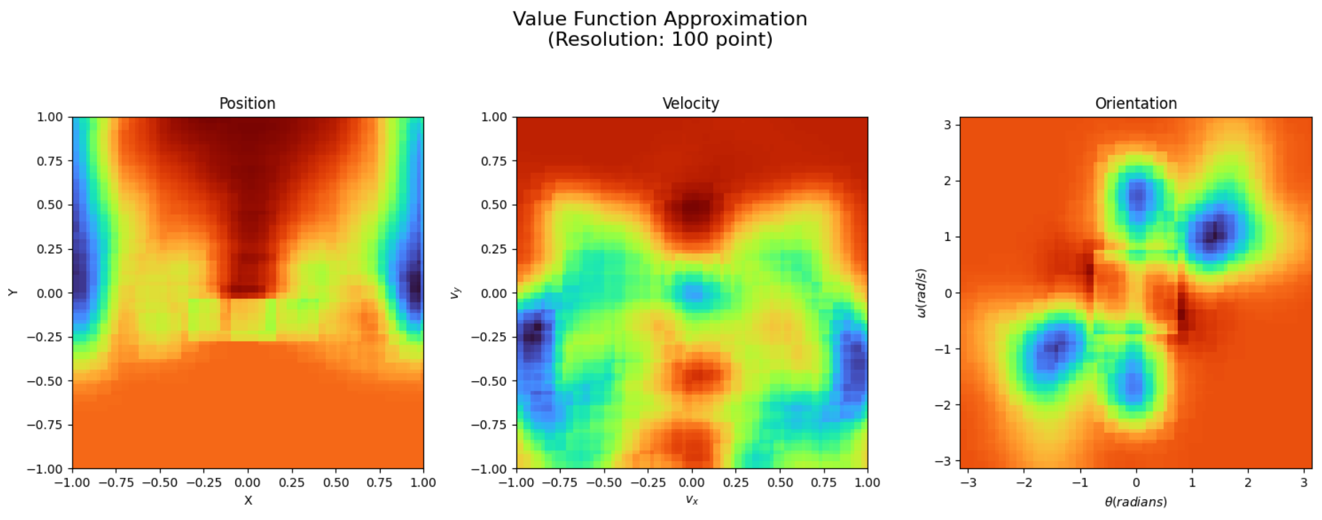
In figure 7a, the position map shows a deeper shade of red in the upper middle, indicating that this is a high-value zone. The zone has value because the lander always begins there, and the high $\lambda$ value means that the traces retain this state in memory whenever it lands successfully, increasing its weight. The observed cone of value is much wider when $\lambda = 0.75$. Effectively any position state where $y > 0$ appears to have high value for the lander.

The velocity maps also explain some of the observed behaviors. As noted previously, higher velocities result in lower return. For $\lambda = 0.99$, the agent prefers $v_y < 0$, pushing the agent to land faster; this shortens the episode duration but simultaneously reduces the overall reward when the lander hits the ground with higher speed. The velocity map for $\lambda = 0.75$ shows a high-value zone in the $v_y > 0$ area, which tells the agent to use main engine thrust to keep itself aloft and reduce its speed of descent. There must therefore be a decay rate that balances the position valuation of 7a and the velocity valuation of 7b.

The orientation map is logical for both. It indicates that when $\theta < 0$ (lander is oriented counter-clockwise), the highest value state will be when $\omega > 0$, thus incentivizing the lander to rotate clockwise. The opposite is true when $\theta > 0$. Simulating for any decay value eventually learns this behavior.



(a) $\lambda = 0.99$



(b) $\lambda = 0.75$

Figure 7: State value function approximations with different decay rates, models trained using 8000 500-step episodes

An interesting change is observed when the step size of the episodes is increased to 1000. The OpenAI version of the LunarLander environment caps the number of steps per episode at 1000, so to compare results with other literature this was the maximum number of episode steps tested. When the step sizes were no longer limited, the curves appeared to converge into each other without reaching positive return, as shown in figure 8. State-tiling heatmaps for these tests are shown in figure 9.

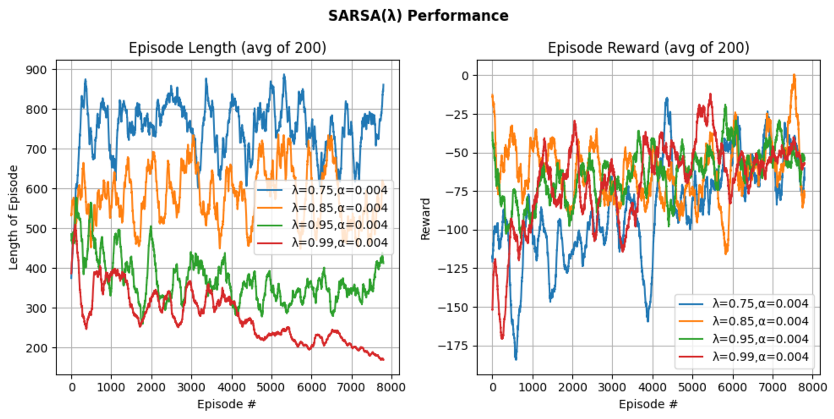
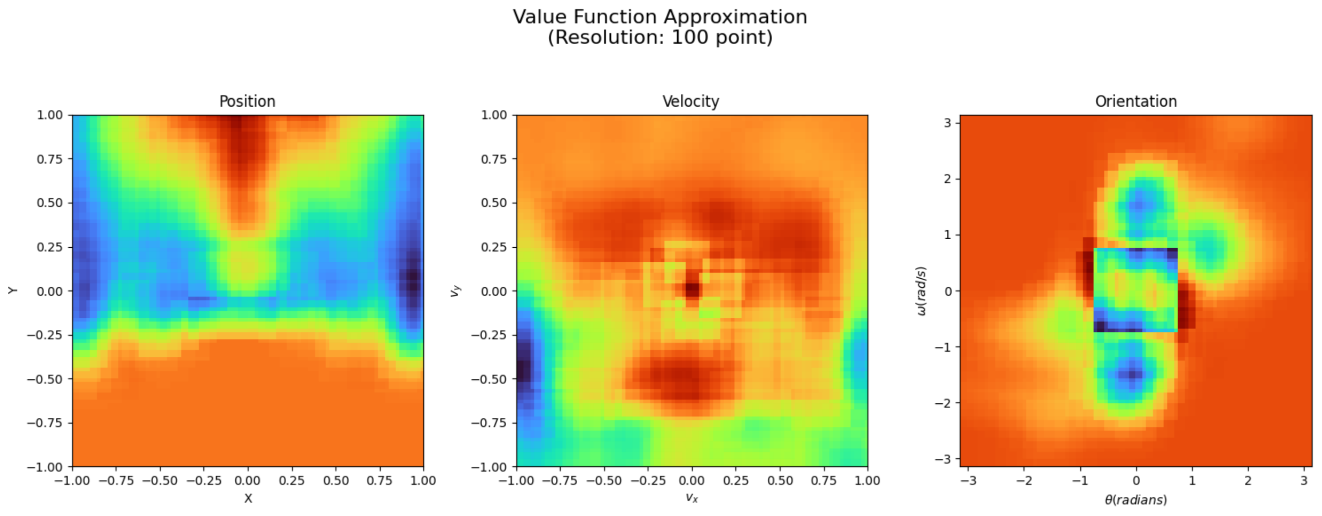
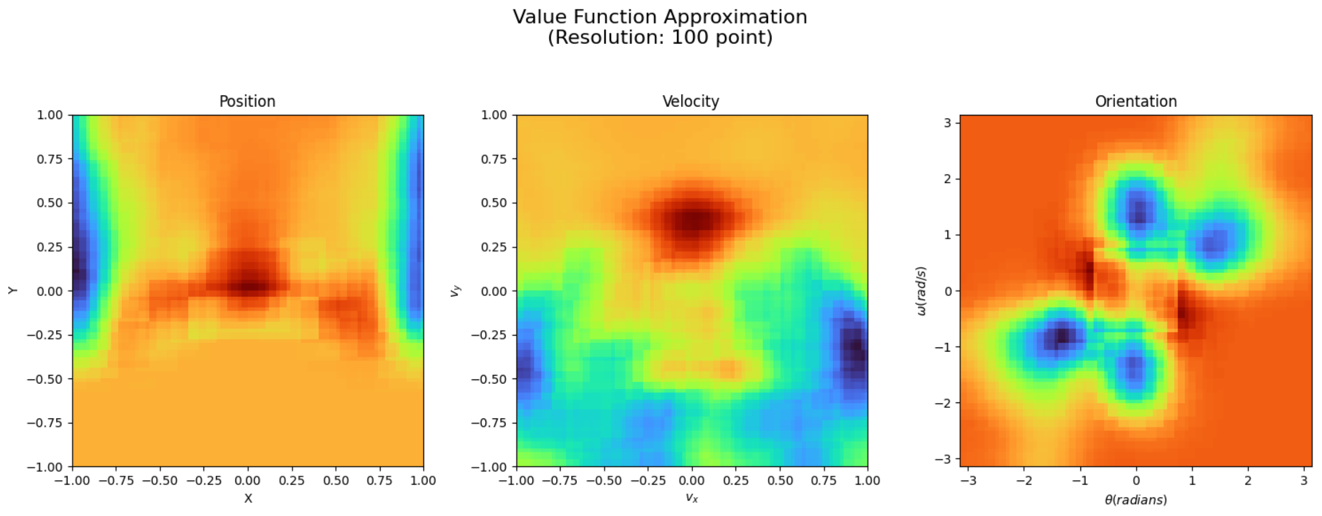


Figure 8: Average SARSA($\lambda$) returns using 1000-step episode training



(a) $\lambda = 0.99$



(b) $\lambda = 0.75$

Figure 9: State value function approximations with different decay rates, models trained using 8000 1000-step episodes

Predictably, training for more steps per episode resulted in more precise state valuation. The position map of 9a remains similar to the 500-step training, without learning to land anywhere in particular. The velocity map, however, now shows a precise high-value zone around zero velocity magnitude, evidently learning to control its speed.

The position map in figure 9b shows ground landings having more value than hovering in the air, an improvement over the 500-step episodes, showing slightly more value in the landing zone and moderate value zone almost every position where $y > 0$. The velocity map shows the same pattern as with 500 steps but with more contrast than before, still emphasizing low positive y-velocity to slow the lander's descent. This begs the question of why it doesn't learn what the $\lambda = 0.99$ lander does regarding velocity control.

With the belief that coarser tiles and thus fewer states might result in faster training, the test was repeated using 4 tiles per continuous state variable, while maintaining 8 tilings for each. The reward and episode curves for a few decay values are presented in figure 10. $\lambda = 0.99$ appeared to perform best by actually landing instead of hovering, but still fails to achieve average positive return.

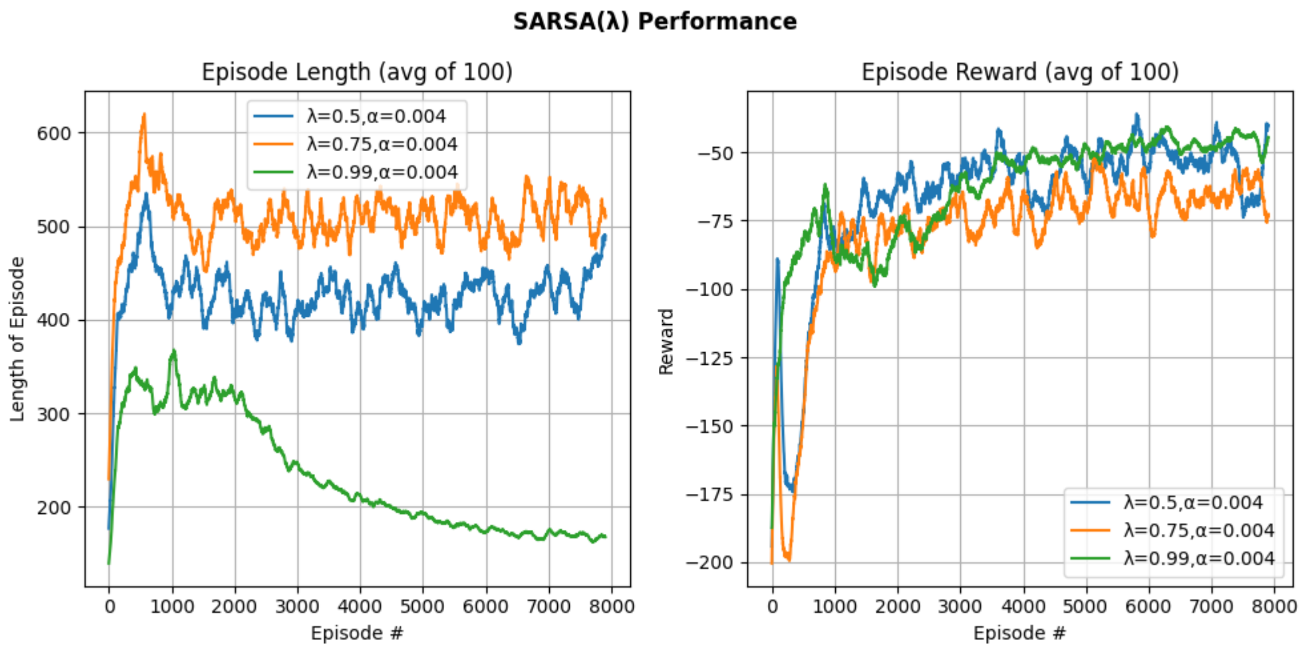


Figure 10: Average SARSA($\lambda$) returns using 4-tile state aggregation and 1000-step episode training

## 5.3 Advantage Actor-Critic (A2C)

Results for A2C simulations are shown in figure 11. After fine-tuning hyperparameters, simulations produced positive average returns after only 4000 episodes, a marked improvement over tabular SARSA. Somewhat surprisingly, the reward tends to crest around 5000 episodes, and then begins to decline.

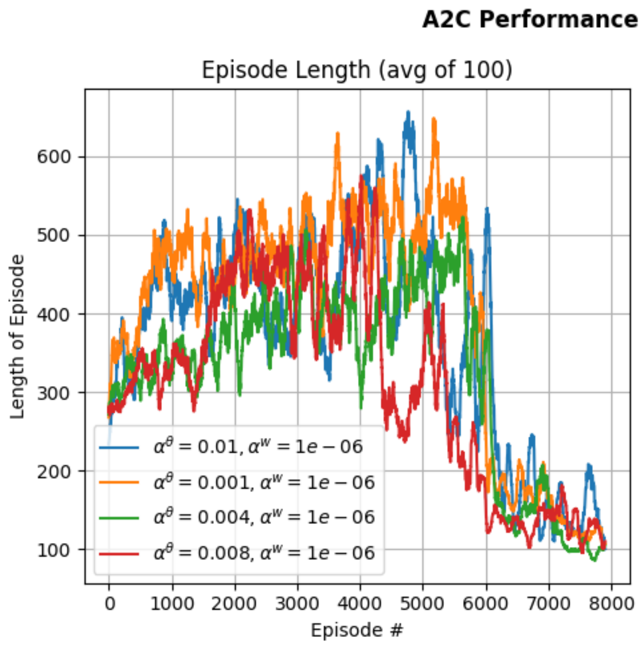


Figure 11: Average return of episodic one-step A2C using 1000-step episodes

The definite reason for this behavior is unknown, but can be hypothesized by observing the state-value function approximation shown in 12.

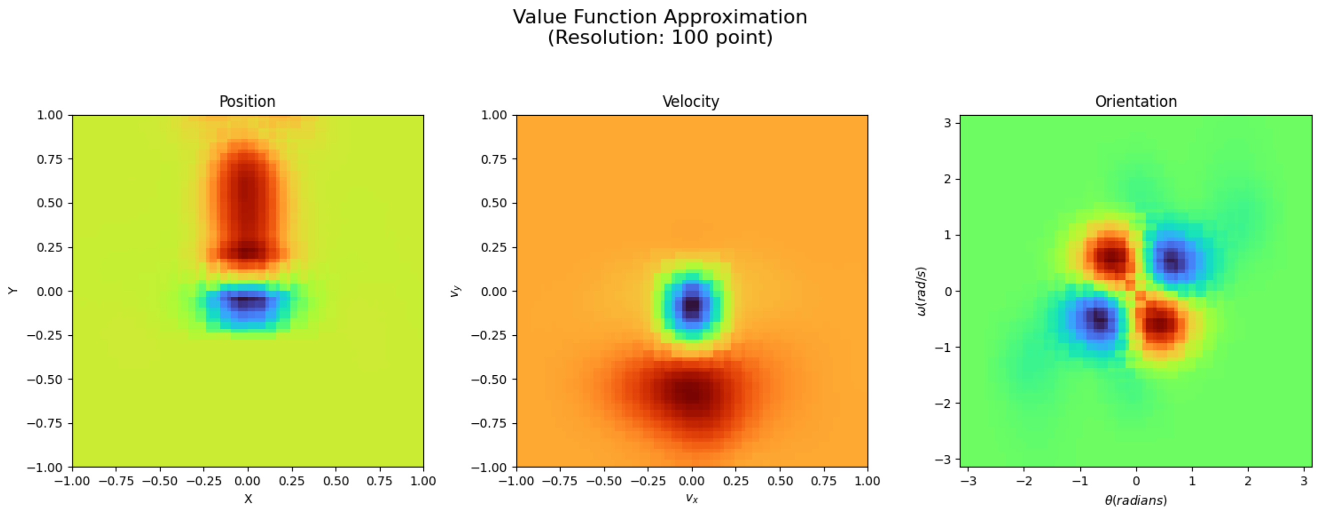


Figure 12: Average return of episodic one-step A2C using 1000-step episodes

The orientation function approximation somewhat maintains those found for SARSA($\lambda$). The position and velocity heatmaps, however, show entirely different behavior compared to before. From the position map, the agent has clearly learned that the landing zone is above the x=0 position. The centroid of the lander is slightly elevated from zero due to the height of the lander's legs, so the darkest red area is in the appropriate location. The velocity graph demonstrates that the agent has learned that moving downwards is the goal, and will move left or right occasionally in order to reach the high-value states shown in the position graph.

This begs the question: "why, then, does the reward begin falling after episode 5000?" It is possible that by episode 5000, the agent has solved the scenario, but is still continuing to train. The only change it can make to continue reaching the goal and increase reward, however, is to use less fuel. The only way for it to do this is to incorporate more frames where no action is taken, letting gravity pull the lander downward. This increases the lander's velocity and reduces the overall reward given by the reward shaping function (1). This is also observed by the dramatically shortened episode lengths observed in figure 11.

In theory, the agent would learn an optimal policy, given that the problem is stationary and deterministic. A possible solution to this problem would be to end training when the agent has solved the scenario a certain number of times out of a certain number of recent episodes.

## 5.4 Deep Q-learning

### 5.4.1 Learning rate performance comparison

We used different learning rates to check how it influences the training process. We test three learning rate: 0.001, 0.0015, 0.002. Figure 13 shows that when learning rate is 0.002 the performance is better and the model has an average reward of 100 with around 100 episodes. As learning rate increases, the neural network weights update more from gradient descent, causing more training steps to find the global minimal for the loss function.

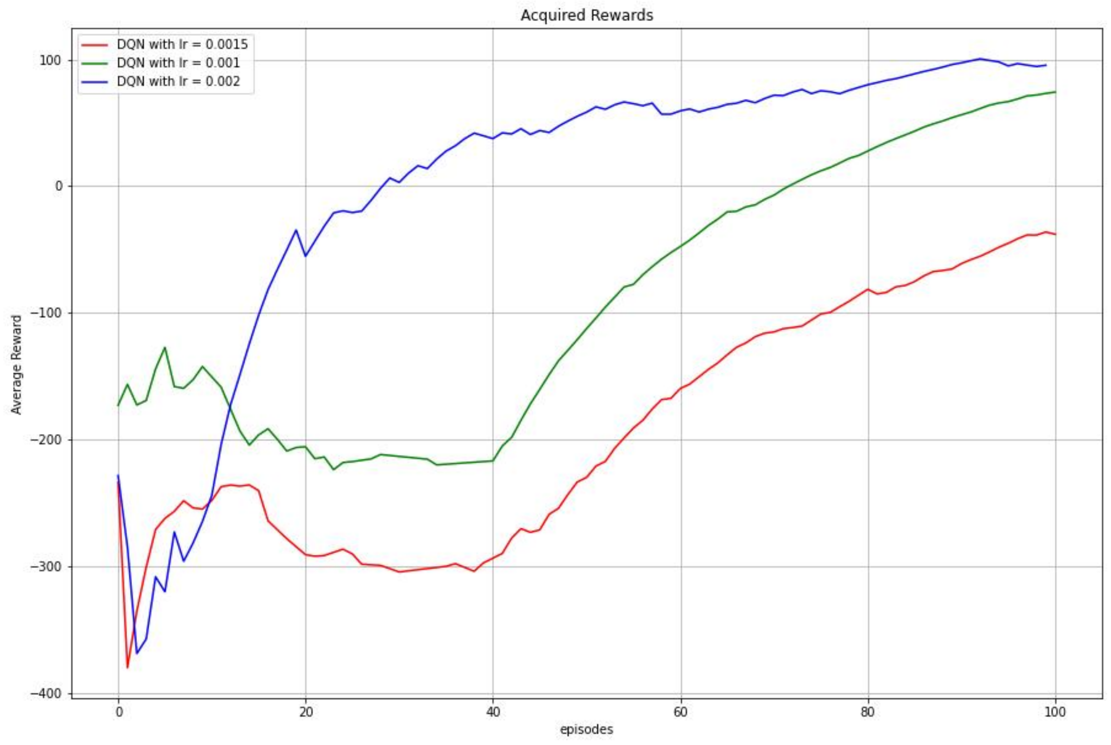


Figure 13: Average rewards for three learning rate values

### 5.4.2 Batch size performance comparison

Batch size also plays an important role in training the agent. As we can see in figure 14, the agent with batch size 64 converges faster than the agent with batch size 32. If the batch size is too small(32), it needs nearly 1,000 iterations to get enough data to converge as shown in figure 15. We also observed that it took more time to train the smaller batch size. This is because we are passing a small batch of state space through the neural network at a time.

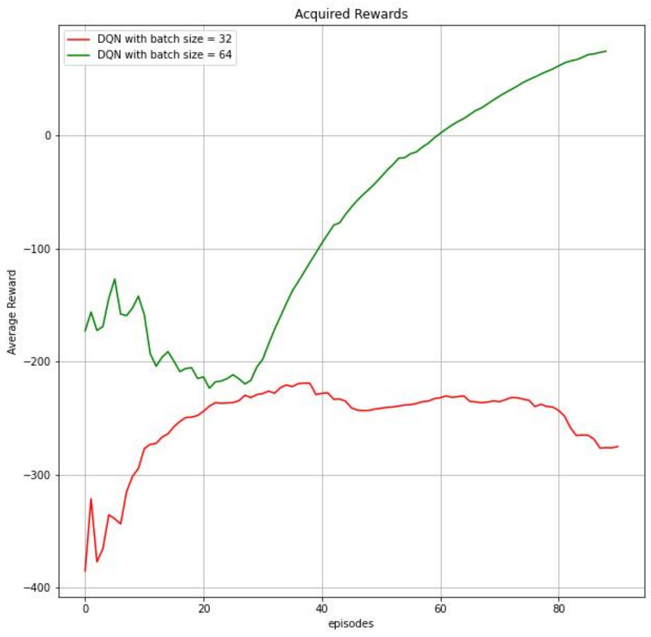


Figure 14: Average rewards for two batch size values of 32 and 64 over 100 episodes

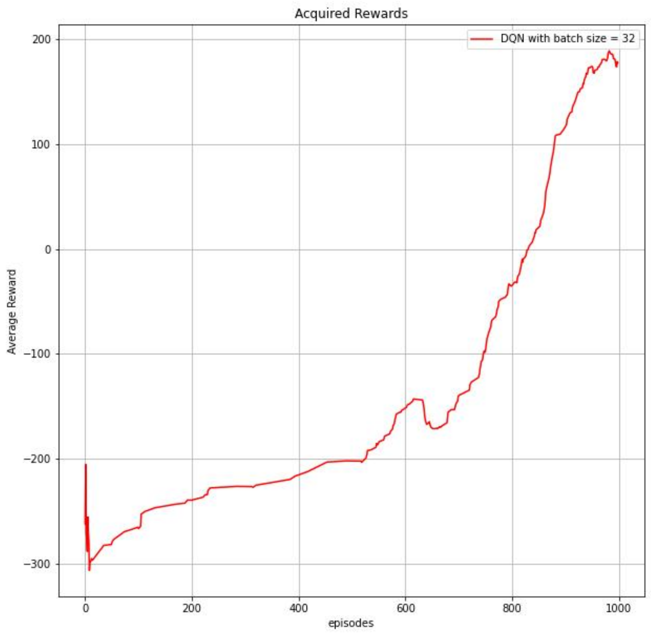


Figure 15: Average rewards for batch size 32 over 1000 episodes

## 6 Conclusions

In terms of solving a simulated environment or game, testing a multitude of RL methods proves highly informative about their efficiencies and drawbacks. The biggest challenge of the Lunar Lander environment proved to be the high-dimensionality of the state space. Considering the lander's observable state has 6 continuous dimensions and 2 discrete ones, optimizing for memory and training time becomes a considerable challenge: as the quantized resolution increases, state aggregation becomes more precise but the simulation requires more memory and many more training episodes. Because the state changes with each step of each episode, TD methods were the obvious choice to develop an on-line policy. In order to use DP methods, a massive table approximating every possible state would have been needed, which is not only inefficient but prohibitively large on most computers.

Tabular SARSA and Q-Learning were demonstrated to be ineffective within 10000 episodes without optimizing the discretization method. With this change, both methods achieved an average positive reward after about 8000 episodes. Since all other methods were being compared to this baseline, only 8000 episodes were simulated for all subsequent methods to reduce simulation time. Approximately one hour was required to simulate 10000 episodes under tabular learning.

Despite having high hopes for eligibility traces, SARSA($\lambda$) turned out to also lack effectiveness. An average positive return was only obtained when the simulation was artificially cut short at 500 steps; this achieved artificially-high returns because fuel was being expended for less time. When the simulation was lengthened to its full 1000 steps, a rising trend was observed that could not achieve positive average return under 8000 episodes. Other changes to be made would have been implementing a softmax distribution as was done in A2C, or at least using a variable, decaying $\epsilon$ value to explore as much as possible early-on. Simulation time was directly related to the size of the hash table used for tiling. This being smaller than the state table used for tabular SARSA resulted in slight reduction in simulation time per episode, although without positive results to show for it.

In terms of linear approximation methods, A2C was decidedly the victor. Simulations were faster than SARSA($\lambda$) by approximately 15%, and achieved positive results in approximately half the number of training episodes. Using coarser tile coding and a smaller hashing table could increase the time benefits even further, at the expense of more training episodes. Adding a clause to cease training when the simulation has been "solved" also seems necessary given the parabolic return curve.

In terms of raw performance, DQN was the undisuputed champion, achieving positive results in 825 episodes in the worse case, and under 100 in the best case. This is largely due to the ability of the artificial neural network to process floating-point states with nonlinear function approximation. This came at a huge cost, however. Simulating just 1000 episodes for the DQN model using batch size of 32 took approximately 40 hours of continuous processing on a home desktop (Intel Core i7-10700K). We were unable to use GPU acceleration to reduce simulation time, but no GPU improvement would reduce it to the scale observed with the linear approximation methods.

Gauging success by the number of training episodes is relative, given that simulating one-eighth as many nonlinear episodes as linear ones took over 4000% more time. All things considered, the sheer performance improvement by the neural network demonstrates that once training has been done, the deep neural network is by far the most accurate and reward-maximizing function approximator.

## References

[1] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. "Solving the lunar lander problem under uncertainty using reinforcement learning". In: *2020 SoutheastCon*. Vol. 2. IEEE. 2020, pp. 1–8.

[2] Han-gen He, Dewen Hu, and Xin Xu. "Efficient reinforcement learning using recursive least-squares methods". In: *arXiv preprint arXiv:1106.0707* (2011).

[3] Richard Sutton. *Tile Coding Software – Reference Manual, Version 3.0*. URL: `http://incompleteideas.net/tiles/tiles3.html`.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.