Lunar Lander Experiments (Project 2)

Steven Nord
snord3@gatech.edu
Hash:
ef4ff7c7e77c6306a8e47cea8fc108f1800b48fb

***Abstract*— Throughout this report, the experimentations and algorithm used to solve the OpenAI Gym's Lunar Lander are discussed.**

I. Introduction

The Lunar Lander is an environment with a continuous state space. The number of states is not finite, which presents challenges for the tabular Q-Learning algorithm to learn a policy for the lander. Instead, a function approximation algorithm is used to overcome this obstacle. Keep in mind, there are several function approximators to choose from, but this report focuses on Deep Q Learning (DQN).

There are numerous hyperparameters that need to be explored when working with the DQN algorithm, but only a few will be tested and analyzed.

II. Lunar Lander

The Lunar Lander is an agent that interacts with the environment and its goal is to safely land on the moon's surface. The agent is presented an 8-dimensional state space:

Sym	Description	Type	Min Value	Max Value
x	horizontal position	Continuous	-1.5	1.5
y	vertical position	Continuous	-1.5	1.5
$\dot{x}$	horizontal speed	Continuous	-5	5
$\dot{y}$	vertical speed	Continuous	-5	5
θ	angle of the lander	Continuous	-3.14	3.14
$\dot{\theta}$	angular speed	Continuous	-5	5
L_l	left leg touching	Boolean	0	1
L_r	right leg touching	Boolean	0	1

The agent is equipped with a main engine and an engine on both sides. Therefore, on any timestep the agent can choose from one of four actions (fire main, left, right engine, or do nothing).

The environment's terminal reward is 100 points for the agent landing safely or -100 for crashing. Rewards are also given during each timestep for moves that navigate the lander closer to or farther from the landing area. The agent has an unlimited fuel supply, but is penalized -0.3 for using the main engine and -0.03 for using either of the side engines.

Episodes are deemed a successful mission if the lander can accumulate 200 or more points.

III. Deep Q Learing Algorithm (DQN)

In a standard Q Learning, the state-action values are estimated from a tabular approach. Typically, each row represents a state and each column corresponds to a possible action that can be taken from within that given state. Given this environment's state space is continuous, Q Learning in its basic form could not be used.

One option could be to discretize the state space, but this requires manual manipulation of the six continuous state parameters. The second option, function approximation, eliminates this manual process.

Specifically, Deep Q Learning (DQN) utilizes Neural Networks (NNs) to map the continuous state spaces to the optimal action(s). The weights of the Q-network are updated with stochastic gradient descent (SGD) by minimizing a loss function (L).

A key assumption for training NNs is that the data is uncorrelated. Due to the sequential nature of reinforcement learning data, an experience replay mechanism called memory replay alleviates the correlation in the data. Memory replay works by storing previous transitions in memory. Then, minibatches are randomly sampled to use for

DQN Algorithm

Input:
  γ, ε_decay_rate, ε_min, n_hidden_nodes, α, τ
Algorithm parameters:
  memory replay D with capacity (N)
  action-value function Q with random weights
for episode$_i$ in number of episodes (M):
  for t in T:
    Choose action$_t$ for state$_t$ using ε-greedy policy
    Execute action$_t$ and observe reward$_t$ and state$_{t+1}$
    ε = max(ε*ε decay rate, ε min)
    Store transaction (s_t, a_t, r_t, s_{t+1}) in D
    Randomly sample minibatch from D
    If s_{t+1} is None: (meaning at terminal state)
      $y_t = r_t$
    else:
      $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a_{t+1})$
    Calculate loss (L) between $Q(s_t, a_t)$ and y_t
    Update Q with SGD by minimizing L

updating the weights, rather than while the data is being experienced.

The pseudocode for the DQN algorithm can be seen in more detail on the previous page.

IV. Hyperparameter Tuning

As previously mentioned, there are several hyperparameters to analyze when training a DQN. The parameters this report focuses on are:

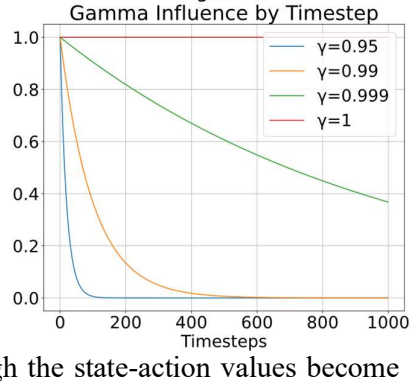
A. Discount rate (γ)
B. Epsilon (ε) decay rate
C. Epsilon (ε) minimum
D. Number of nodes (in the NN's hidden layers)
E. Loss Functions

The tuning process is an iterative process where the first several times through all the parameters are to gauge appropriate ranges of values. The figures shown in each subsection are focused on 1 parameter at a time and other hyperparameters are set to an approximate optimal value found from previous iterations. Also, figures for each hyperparameter include the rolling average (solid lines) and the spread of one standard deviation (corresponding colorful bands) over the last 25 episodes.

A. Discount Rate (γ)

The discount rate (gamma) determines the level of future rewards to consider when calculating the state-action values. Figure 2 shows how the influence of future rewards decreases over timesteps.

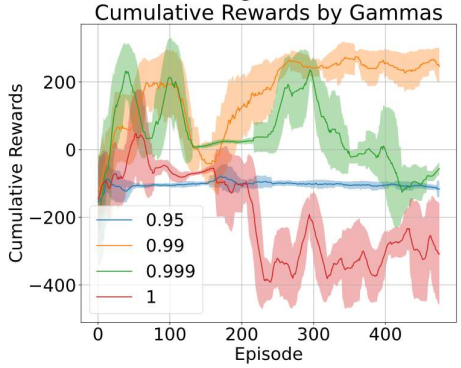
Figure 1:
Gamma Influence by Timestep



A discount rate that is too low does not equip the agent with enough far-sightedness to account for future events. On the contrary, if the discount rate is set too high the state-action values become unstable, because they allow very distant future rewards to have too much influence. For the lunar lander, gamma set to 0.99 appears to be the ideal value, as seen from Figure 2.

Table 1 (p. 5) shows the average number of timesteps for a trained lander to complete an episode, which is 417. This number coincides appropriately when γ=0.99 in Figure 1. This factor allows future rewards to carry sufficient influential power throughout the episode to accurately adjust value estimates. For smaller gammas, future rewards are discounted to a point where they become rather insignificant. As a result, optimal actions are not influential early enough in episodes to ensure a successful landing.

Figure 2:
Cumulative Rewards by Gammas



B. Epsilon (ε) Decay Rate

Epsilon is the probability of selecting an action at random versus taking the optimal action (ε-greed policy in the DQN algorithm). Tuning this value helps find the right balance between exploration and exploitation. The agent needs to explore each state space sufficiently to be able to estimate its state-action values. At the same time, over exploring causes inaccurate estimates as optimal sequences are not being taken to reach desired terminal states.

The epsilon decay rate's role is specifically used to decay epsilon over time. Decaying epsilon encourages more exploration to occur toward the beginning of agent training and eventually takes more optimal actions once the foundational estimates have been achieved.

Epsilon decay can be implemented in DQN differently throughout training. It can be applied either after each action is taken or after each episode. It is worth nothing that both are valid methods although they may result in different ranges of suitable epsilon decay rates. Figures seen throughout this report are generated from the former.

Figure 3:
Exploration Probability by Timestep

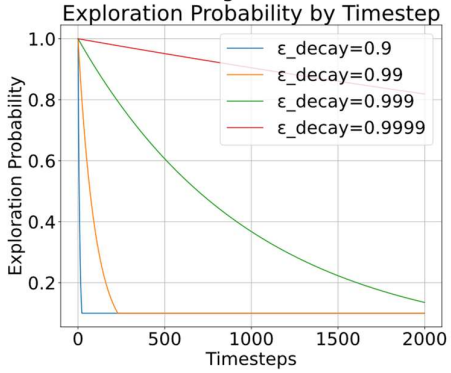
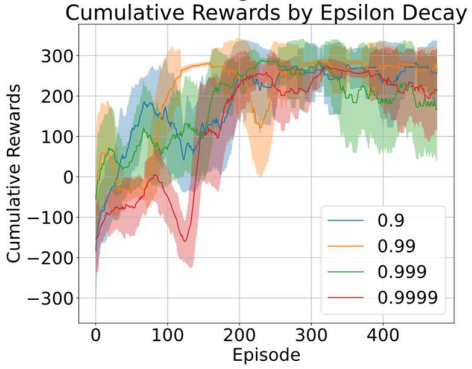


Figure 3 shows the impact on the exploration probability throughout training with various decay rates. The minimum value for epsilon (ε_min) is set to 0.1 in the plot, but this will be explored in the next subsection.

Figure 4 shows that the decay rate for epsilon does not have as strong of an influence as other parameters. This is due to other hyperparameters becoming optimized prior to this plot and specifically ε_min (discussed below).

Figure 4:
Cumulative Rewards by Epsilon Decay



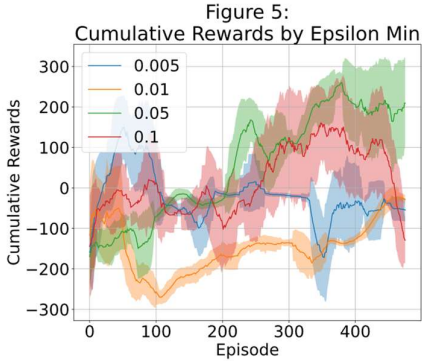
This does not mean the decay rate can be set to any arbitrary value, just that this range is in an acceptable range. Typically, lower values reduce exploration too quickly and the agent never discovers optimal action that result in successful landings. However, due to optimizing ε_min, exploration is still occurring even after epsilon fully decays. On the other hand, larger values prolong exploration out too long and state-action values do not converge to optimal action selections. This is less relevant in this experiment due to the decision to decay after each action rather than each episode.

C. Epsilon (ε) Minimum

The epsilon minimum is depicted in Figure 3 by the merging lines at the bottom of the plot. This factor ensures there is some amount of exploration that occurs even after epsilon is decayed.

Figure 5 shows the different values of ε_min across the training episodes. The smaller values of ε_min do not translate to high cumulative rewards as a result of the lack of exploration after epsilon decays. The concept of exploring translates to the spread of the standard deviations for the various ε_mins. The larger values (e.g., 0.05 and 0.1) have greater spread than the other two values. This wider spread is due to the randomness in exploration with the larger epsilons.

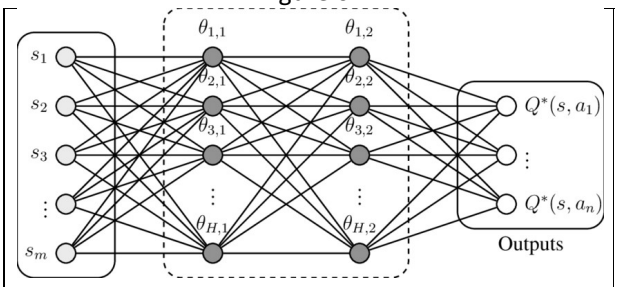
Figure 5:
Cumulative Rewards by Epsilon Min



D. Number of Nodes (in Hidden Layers)

The generic structure of the NN is shown in Figure 6 [1]. From left to right, the input layer is the 8-dimensional state space, followed by 2 hidden layers, and finally the output layer. The final layer outputs the estimated state-action value ($Q^*(s, a_i)$), where a_i corresponds to the 4 possible actions in Lunar Lander.

Figure 6:



Generic 2-Layer Neural Network (NN) for DQN each with H nodes. Adapted from "A Framework for Automated Cellular Network Tuning with Reinforcement Learning" (p. 21) [1].

The number of nodes in the hidden layers is set to be the same value (H) in both layers. Though the value of H does not have to be the same for each layer, it is set the same in this case for easier analysis. Also note, there are countless other parameters that could be tuned for NNs, however H is the only one explored in this report.

Figure 7:
Cumulative Rewards by Number Of Nodes

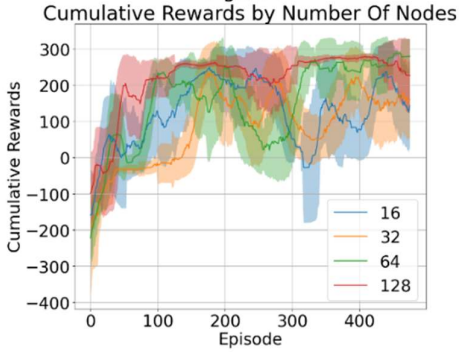


Figure 7 illustrates the complexity of the Lunar Lander. The figure shows the importance of the additional nodes within the hidden layers. When H is set to only 16, the NN does not capture the complexity necessary to accumulate enough rewards. Conversely, a value of 64 or 128 appears to be a reasonable selection based on cumulative

reward, but there are other aspects of training that can be considered.

A strong motive for considering the number of nodes is to minimize training time while still capturing the complexity of the problem. Fewer nodes require less computation during forward and backward propagation, which can lead to less training time. Figure 8 shows the training time for each value of H for the 500 episodes.

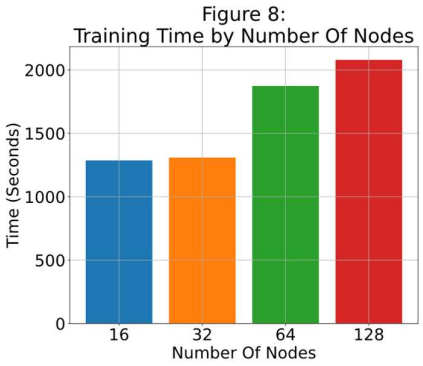


Figure 8:

The training time is definitely worth considering when looking at sizes 16 and 32 versus sizes 64 and 128. However, when combined with Figure 11, the smaller sizes struggle to capture the complexity of the problem, therefore the loss in stability of the estimated values is not worth the gain in training efficiency. Now, when comparing 64 and 128 there can be a case made for either selection given the minor trade-off between cumulative rewards and less training time. The selection for the trained agent discussed later is implemented with the 128 due to the higher cumulative rewards.

E. Loss Function

During training, the loss function is used to quantify the difference between the predicted Q-values and the target Q-values, and it guides the learning process to minimize this difference with stochastic gradient decent (SGD). The characteristics of a problem can make one loss function more effective for learning the task at hand and therefore make it an important hyperparameter to investigate. How to choose which loss functions to explore? There are countless options, but two functions are experimented with in this report: Mean Squar Loss (MSE) and Huber.

MSE is a clear choice as it is one of the most commonly used loss function. One downside to MSE is that it penalizes larger errors by squaring them [3]. This makes MSE sensitive to outliers and susceptible to exploding gradients. Huber strikes a balance between MSE and Mean Absolute Error (MAE), which helps make it more robust to outliers. There is a hyperparameter (δ) that can be tuned, but is not explored in this report (δ=1).

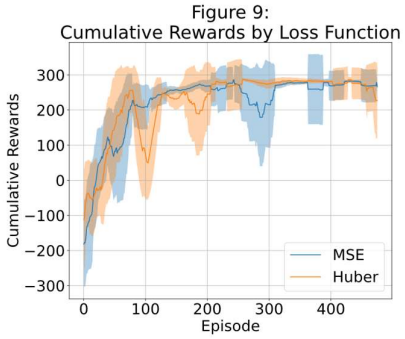


Figure 9:

Figure 9 and Figure 10 show the cumulative rewards and training time, respectively, for 500 episodes. The fact that both produce similar results might suggest that outliers are not significantly impacting the training process in a way that would favor one loss function over the other.

Figure 10:

V. Trained Lunar Lander

Ultimately, the effort put into analyzing parameters pays off for the lunar lander as Figure 11 shows the improvement in cumulative rewards as the agent gets more exposure during the training phase. The selected hyperparameters are shown in Table 1 (p. 5).

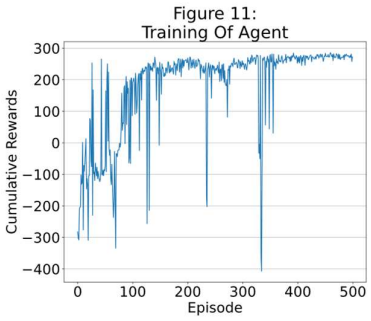


Figure 11:

Figure 12 shows the cumulative score over 100 consecutive episodes for a trained versus an untrained agent. The disparity

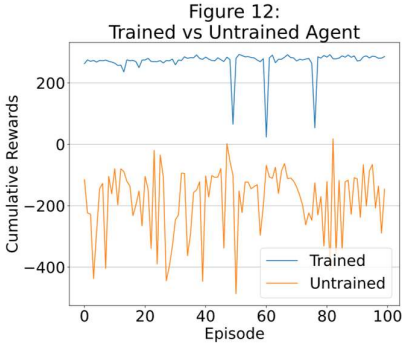


Figure 12:

between the two lines shows the knowledge the agent gains from the q-values learned from DQN.

VI. Challenges and Future Exploration

It was a challenge to select hyperparameters. The rationale for analyzing the discount rate is that it typically is the most impactful parameter. It determines the amount of influence future rewards have on the state-action values, so if not tuned correctly the agent will not plan accordingly. Next, it is important to control the proper balance between exploration and exploitation in all RL problems, hence ε_decay_rate and ε_min are essential to analyze. Finally, the number of nodes is included as it plays the crucial role of figuring out the complexity of the problem at hand.

Even though the final trained agent performs better than an untrained agent, it still has room for improvement. There are many other parameters that can be explored to help enhance the performance related to NNs, exploration/exploitation tradeoff, and the algorithm choice itself.

Here is a list of unexplored parameters related to NNs alone: the number of layers, the number of nodes in each layer, the activation function, and weight initialization. All of these and more can be analyzed to see if there is a more accurate and efficient NN architecture.

DQN is effective for environments with discrete action spaces [2], which is the case for the Lunar Lander, and it is also an easier transition from Q-Learning. These provide a strong case for using it to solve this problem. There are other algorithms for future consideration (e.g., PPO and A3C).

Another challenge is tuning the parameters. As mentioned previously, tuning is an iterative process of analyzing a single parameter with different values/techniques and holding all others consistent. Tuning took several iterations to determine optimal ranges and each iteration of training took quite a bit of time.

One improvement for this is to utilize GPU rather than CPU. The parallel processing capabilities of a GPU would help accelerate the training/tuning process.

VII. Conclusion

This exercise of teaching the Lunar Lander presents a simple environment for a real-world challenge: autopiloting a shuttle to land. This toy example simplifies that scenario by providing a flat surface to land on, no outside obstacles/debris to avoid, and unlimited fuel supply; additionally, as mentioned above still presents many challenges.

Even though 90% success rate is an improvement from 0%, there is still a lot of room for improvement because 1 out of 10 failures is not the standard for landing billion-dollar projects. Here is an image of a successful mission for the Lunar Lander.

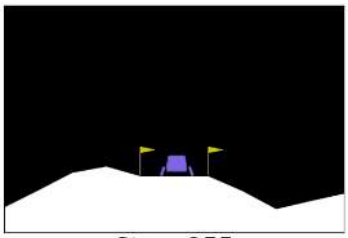


Table 1

	γ	ε Decay	ε Min	Nodes (H)	Loss
Trained	0.99	0.99	0.1	128	MSE

Results

	Win Percentage (%)	Ave. Reward	Ave. Steps
Trained	97.0	263.3	417.1
Untrained	0.0	-181.6	93.0

References

[1] Evans, B., Jinseok, C., and Mismar, F. B. "A Framework for Automated Cellular Network Tuning with Reinforcement Learning". USA, 2019

[2] Wu, F., Zhao, J., and Zhu, J. "An Overview of the Action Space for Deep Reinforcement Learning". Sanya, China, 2021

[3] Gupta, S. " The 7 Most Common Machine Learning Loss Functions". June 2023. https://builtin.com/machine-learning/common-loss-functions