Autonomously Landing a Rocket on the Moon with Reinforcement Learning Techniques

Abdoulaye Diallo
Smead Dep. of Aerospace Engineering
University of Colorado Boulder
Boulder, CO
Abdoulaye.Diallo@colorado.edu

Benjamin Chupik
Smead Dep. of Aerospace Engineering
University of Colorado Boulder
Boulder, CO
Benjamin.Chupik@colorado.edu

Guido Insinger
Smead Dep. of Aerospace Engineering
University of Colorado Boulder
Boulder, CO
Guido.Insinger@colorado.edu

***Abstract*—Reinforcement Learning (RL) is a subset area of Machine Learning that consists of training artificial intelligence models in a specific way where the agent must learn to achieve a goal within an uncertain and potentially complex environment. In this paper, we will teach a rocket to land autonomously on OpenAI Gym's *LunarLander-v2* environment through different RL algorithms and techniques such as Q-learning, Monte Carlo Tree Search (MCTS), Deep Q-Network (DQN), and Double Deep Q-Network (DDQN). We also propose a heuristic policy to solve the environment. We first experiment the algorithms in the default environment, then we introduce the wind parameter in it. After performing a grid search of a selected group of hyperparameters, the DQN and our implemented DDQN are able to achieve an average reward of approximately 265 in the default environment and 220 in the environment with wind. The Q-learning algorithm is unable to work for this problem due to the continuous state space of the environments. The heuristic policy is only a solution in the environment without wind and MCTS failed to solve both environments. Additionally, we discuss the performance of the DQN and DDQN algorithms and evaluate the differences in their results. At the end of our paper, we propose solutions to improve and/or fix the issues of these algorithms for the Lunar lander RL problem.**

***Index Terms*—Reinforcement Learning, Q-learning, Deep Q-learning, Double Deep Q-learning, Monte-Carlo Tree Search, Robotics, Artificial Intelligence**

I. Introduction

Lunar lander is a game where the task is to control the fire orientation engine to help the lander land on the landing pad. OpenAI Gym provides a Lunar Lander environment that is designed to interface with reinforcement learning agents. The environment handles the backend tasks of simulation, physics, rewards, and game control which allows one to solely focus on building an agent. By leveraging the power of reinforcement learning, artificial intelligence, and cutting-edge simulation technologies, this game enables participants to experiment with various strategies, learn from their successes and failures, and ultimately design more robust and efficient lunar landers that can withstand the harsh conditions of the moon.

II. Background and Related Work

The lunar lander problem at its root is landing a system vertically by controlling its attitude and vertical thrust. This is a highly relevant problem that multiple space companies like Space-X and Blue Origin are trying to solve. They are attempting to land their rocket boosters on a small pad, having to deal with cross winds. These boosters only have their vertical thrusters and attitude control thrusters (the same setup as the lunar lander). Both companies have had mixed successes in landing their rockets, so better solutions need to keep being created.

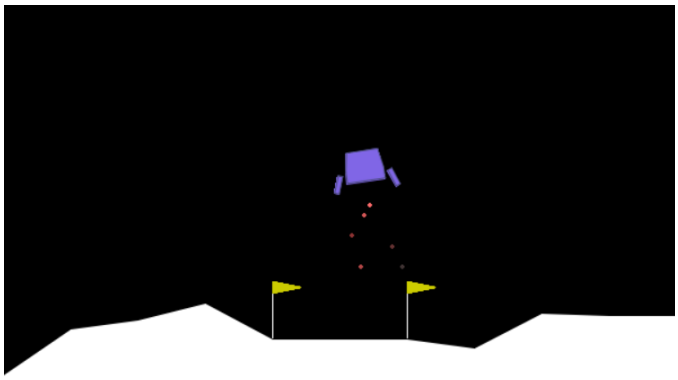

Fig. 1. Lunar lander environment

There is a lot of work done solving this problem via Open AI's Gymnasium Environment due to it being relevant and readily accessible. Most of them focus on Reinforcement learning and Neural Network approaches to the solution. One of such papers tests Deep Q-Learning on this exact problem [1]. This paper was able to get the lander to land most of the time after 2000 training iterations. Another paper uses SARSA and Deep Q-Learning to get rewards of 170 and 200 respectively [2]. More advanced learning methods are also being looked at such as Unentangled quantum reinforcement learning. This is studied in [3] where they show that their algorithm uses fewer samples to learn the lunar lander solution than traditional reinforcement learning algorithms.

There is also research into speeding up reinforcement learning using human input as teachers. This is applied to the Gymnasium environment in [4] where the authors build a framework to have humans kick-start the agent's reinforcement learning process. Another way to speed up the learning is

discussed in [5]. The authors use guided cost learning to develop a reward based on a human playing the lunar lander game. They then use this new reward to learn the best policy to solve the environment. This paper studies a variety of methods to solve the lunar lander problem, from directly making a heuristic, to double deep Q-network learning.

III. Problem Formulation

For this paper, we will investigate different ways to solve the Lunar Lander environment from Gymnasium[1]. This environment shown in [Figure 1] has four discrete actions and an 8-dimensional vector of variables as an observation space.

These variables are its positions in x and y, its linear velocities v_x, v_y, its angle θ, its angular velocity ω, and two booleans c_l and c_r that are true when the left and right leg are in contact with the ground respectively. All of this together gives the observation vector $\left[x, y, v_x, v_y, \theta, \omega, c_l, c_r\right]^T$.

The four available discrete actions are defined as shown in [Equation 1].

$$a = \begin{cases} 0 : \text{do nothing} \\ 1 : \text{fire left orientation engine} \\ 2 : \text{fire main orientation engine} \\ 3 : \text{fire right orientation engine} \end{cases} \tag{1}$$

The reward in this environment is

- increased/decreased the closer/further the lander is to the landing pad.
- increased/decreased the slower/faster the lander is moving.
- decreased the more the lander is tilted (angle not horizontal).
- increased by 10 points for each leg that is in contact with the ground.
- decreased by 0.03 points each frame a side engine is firing.
- decreased by 0.3 points each frame the main engine is firing.

The episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively. An episode is considered a solution if it scores at least 200 points.

The lander starts in the top center with a random disturbing force applied to it. An episode terminates if

- the lander crashes (the lander body gets in contact with the moon)
- the lander gets outside of the viewport (x coordinate is greater than 1)
- the lander is not awake. From the Box2D docs, a body is considered not awake if it doesn't move and/or collide with any other body.

There is also the option to make the action space continuous. For the purposes of this paper, however, the discrete version is used. In addition, the environment can be increased in difficulty by adding wind. This second option is used in this paper to compare the performance of different solution methods.

[1]https://gymnasium.farama.org/environments/box2d/lunar_lander/

IV. Solution Approach

A. Heuristic Policy

The general approach for setting up a heuristic policy for the lunar lander environment was to set up a target angle θ_t for the lander to follow in order to both stabilize the lander as well as to guide the lander back to the center when it is pushed to the side by disturbances. θ_t is calculated according to [Equation 2] where the target angle is increased based on the horizontal position and velocity, clamped to $[-0.4, 0.4]$ radian to prevent the lander from flipping over.

$$\theta_t = clamp(0.15x + v_x, -0.4, 0.4) \tag{2}$$

For each action a value function is set up to discern which action is most useful at a given time which is shown in Equation 3. Which action to take is then decided by which action has the highest value at the time.

$$\begin{cases} V(a=0) = 0.15 \\ V(a=1) = 4(\theta_t - \theta) - 3\omega \\ V(a=2) = -v_y * (\frac{1}{12y} + 0.5) \\ V(a=3) = -4(\theta_t - \theta) + 3\omega \end{cases} \tag{3}$$

The scaling values for all of the parameters were chosen by first manually updating them to generate a reasonable policy. After this, for each parameter a grid of possible values was set up around the previously determined reasonable values and simulated to find the optimal combination of parameters for these specific value functions.

The value for doing nothing, $a = 0$, is set to be a constant value such that the other values can work relatively to this. For firing the left orientation engine, $a = 1$, the value is taken to be a scaled difference between the target angle and current angle added to a scaled version of the angular speed. This combination gives a high value to $a = 1$ when θ_t is much larger than θ and also when ω gets big. $a = 2$, firing the main orientation engine, is valued based on a linear scaling of the vertical velocity in addition to the ratio of vertical velocity to vertical position, incentivizing lower speeds at lower altitudes. For $a = 3$,firing the right orientation engine, the value is calculated in the same manner as for $a = 1$, however with the opposite sign. Finally, when either or both legs are touching the ground $a = 0$ is automatically selected.

B. Q Learning

Q-Learning was introduced by Watkins (1989) and is a well-known model-free RL algorithm whose goal is to make an agent learn how to act optimally in controlled Markovian domains. This algorithm estimates the Q-value which is the value of taking an action a when the agent is in the state s with the policy π. During training, for each couple (S_t, A_t) the agent will update the Q-values as such:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_t + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4}$$

The target Y_t^Q is defined as:

$$Y_t^Q = R_{t+1} + \gamma max_a Q(S_{t+1}, a; \theta_t) \tag{5}$$

All the values are stored in a Q-table. Representing the Q-values with tabular methods might be challenging with state spaces that are continuous. Indeed, with a continuous state space, the number of possible actions are infinite and therefore estimating their Q-values will not be possible. Hence, to solve the Lunar lander environment, the Deep Q-learning algorithm was used.

C. Online Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a reasonable way to solve this problem because it is a continuous state space. MCTS is an online solver where the best action is chosen by creating an action tree to look at how valuable each action is verses each other. It can be stopped at any time and the best action chosen.

The main issue with this implementation for this problem is that the only reward is for landing successfully. This means that the rollout simulation would always return the same answer unless it goes far enough and is lucky enough to reach a landing state. To solve this, instead of using a rollout simulation to determine the value of each state, the value is determined by the previously discussed heuristic. While this may not be the exact value of the state, neither is a rollout and this way there are a larger variety of values to guide the algorithm.

The MCTS is implemented similarly as done in class, but with the rollout replaced with the heuristic policy. A node is selected to be expanded via 6 (Step 1 in figure 2). This node is expanded by every action it can take (Step 2), and then that node is assigned Q values from the previously discussed Heuristic Policy (Step 3). This Value is then backed up the tree to the root node's Q values using 7 (Step 4). After running the Monte Carlo Tree Search multiple times, the best action is selected for the root node using the created Q values.

$$a = argmax(Q(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}}, \forall A) \tag{6}$$

$$Q(s,a) = r + \gamma E[V(s')] \tag{7}$$

D. Deep Q-Network

In the Q-Learning algorithm, the Q-values are represented as a table. However, for big state and action spaces, representing the Q-values with tabular methods can be a challenging task. Therefore, it is insufficient to store state and action pair values in a 2-D array or Q-table, especially in games that have a continuous state space. Hence, to overcome that challenge we will use the Deep Q-Network (DQN) algorithm. DQN is a model- free, online, off-policy reinforcement learning (RL) method. It uses the Q-learning algorithm to learn the best action to take in the given state and a deep neural network to estimate the Q value function.

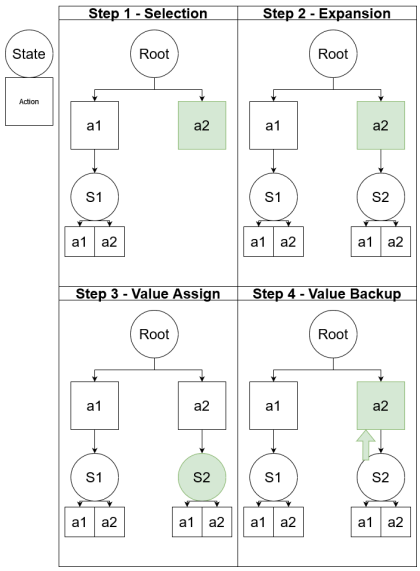

Fig. 2. Monte Carlo Search Tree

Fig. 3. Simplified DQN architecture

A DQN agent is a value-based RL agent that trains a critic to estimate the return or future rewards. In the Epsilon-Greedy Exploration strategy, the agent selects and executes actions according to an ϵ - greedy policy based on Q. The probability that the agent exploits the best known action is $1 - \epsilon$. The main model and the target model map input states to output actions. The actions with the largest Q-values predicted from the model are the best known actions for the current states. In the DQN algorithm, the Q-value function is represented with weights w:

$$Q(s,a,w) = Q^{\pi}(s,a)$$

Their updates in the Q-Network are done through the weights update with the Bellman equation. Its form that the DQN is trained to approximate is:

$$Q(s,a) = E_{(s,a \to s',r) \sim \mathcal{H}}(r_{s,a} + \gamma max_{a'} Q(s',a'))$$

The second main component of the DQN is the loss function. The latter compares our Q-values prediction and the Q-target and uses stochastic gradient descent in a supervised-learning fashion with the mean-squared error loss:

$$L(w) = E[(r + \gamma max_{a' \epsilon A} Q(s',a',w) - Q(s,a,w))^2]$$

The goal of this loss function is to minimize the error to update the weights in the Q network. Finally, to avoid large oscillations or divergence in our neural network, the Experience Replay technique was used. It allows the algorithm

at each time step of data collection to store transitions in a replay buffer and sample a small batch of experience from the replay buffer to update the Q-network. The main advantages of that technique are the increase in stability and data efficiency. The figure 4 represents the pseudo-algorithm of the DQN:

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Fig. 4. Deep Q-Network algorithm

Adapting every epoch's weight and minimizing the loss during training is a crucial task that helps to improve the accuracy and reduces the total loss in the DQN algorithm. Optimizers are algorithms that adapt the neural network's attributes, like the learning rate and weights. The optimizer used in this DQN implementation was ADAM which is probably the most commonly-used optimizer in Deep Reinforcement learning research the past few years. The ADAM optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order (expected value of the gradients) and second-order (expected value of the square gradients) moments. The main advantage of this algorithm is in its capacity to provide an efficient optimization method that can handle noisy and sparse gradients.

E. Double Deep Q-Network

In the DQN algorithm, by systematically taking the maximum value of the target Q-value as such $max_{a'}Q^*(s', a')$, that value will be overestimated which leads to a maximization bias in learning. The algorithm that addresses this overestimation is the Double Deep Q-Network (van Hasselt 2015). The method consists of using two separate Q-value estimators that update each other. Thus, the maximization bias can be avoided by disentangling the updates from biased estimates. In our study, the Q-value estimators are the Q-network Q and the target (fixed) network Q'. The network Q is used for action selection. The network calculates the best action for the next state. Then, the network Q' gets this action from network Q as an input and estimates its corresponding Q-value as such:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1}, argmax_{a'} Q'(s_t, a_t))$$

Finally, after minimizing the mean squared error between Q and Q', the parameters of Q' are slowly copied by Q' through Polyak averaging:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

where θ' is the target network parameter, θ is the Q-network parameter, and τ (soft update parameter) is set to 0.001. The figure 5 represents the pseudo-algorithm of the DDQN:

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)
Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Fig. 5. Double Deep Q-Network algorithm

V. Results

A. Heuristic Policy

When wind is turned off, simulating 1000 episodes results in a mean reward of $\bar{r} = 231.56$ with a standard error of the mean $SEM = 1.69$. Figure 6 shows the rewards collected for each episode.

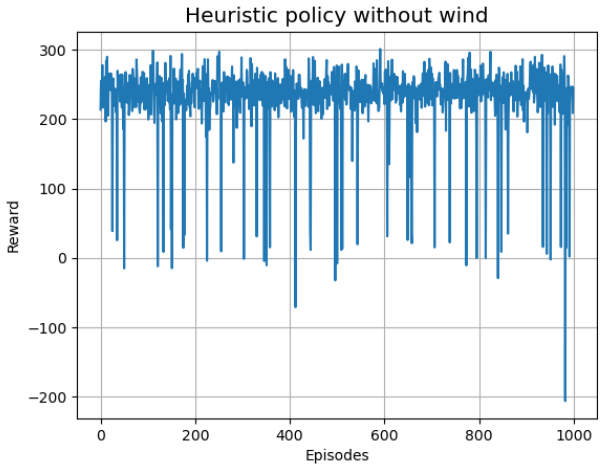

Fig. 6. Heuristic policy performance without wind

Adding wind to the environment, again for 1000 episodes, results in Figure 7, where the mean reward is $\bar{r} = 153.89$ and the standard error of the mean $SEM = 3.67$. Adding wind causes a decrease in performance of the heuristic policy and an increase in variability of the rewards per episode.

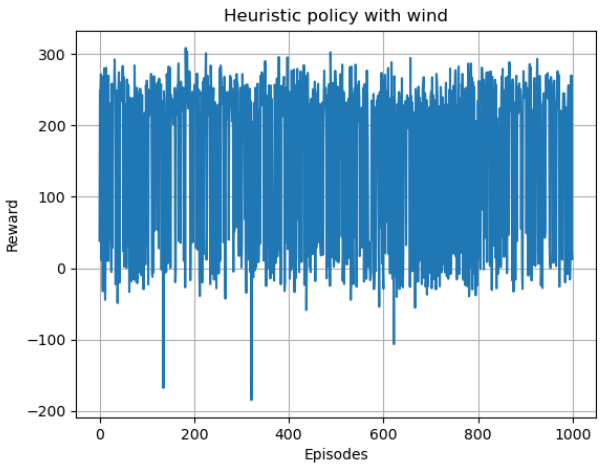

Fig. 7. Heuristic policy performance with wind

B. Q-Learning

The Q-learning algorithm does not work for this problem as is. When trained, it starts to use up a lot of memory creating its search dictionaries. after about 100,000 runs, it already uses upwards of 40 GB of RAM. Also, even if the computer has a lot of ram, it will still fail because the state space is continuous so as soon as it discovers a state that has not been discovered during training, it will not have the best action to choose in its dictionaries.

C. Online Monte Carlo Tree Search

MCTS is run with the parameters $c = 600$, max depth $d = 100$ and a maximum time to decide on an action of 40 ms. After running through 1000 cases, it receives an average score of 64.2. This score is significantly below what the heuristic could produce.

D. Deep Q-Network vs Double Deep Q-Network

In this section of the study, the performance the DQN and the DDQN algorithms will be assessed in the Open AI Lunar lander environment. At first, we used an environment without wind then we introduced the wind parameter in the environment as a second experiment. Also, choosing random hyperparameters to maximize the average reward in the game can be time consuming and not efficient. Therefore, a simple grid search was performed at each run to get an estimate of the best hyperparameter combinations. The hyperparameters that are selected for this optimization are the buffer size, the batch size, the learning rate, and the discount factor. The table displayed below shows the range of the value options.

This method is considered perfect for our case study because the search space is small ($2^4 = 16 < 100$). Each hyparameter has two value options and there are four of them, therefore search space is equal to 16 combinations.

Hyperparameters	Option 1	Option 2
Buffer size	1e4	1e5
Batch size	64	256
Discount factor	0.99	0.999
Learning rate	1e-4	1e-7

TABLE I
HYPERPARAMETER OPTION VALUES FOR SIMPLE GRID SEARCH

All the runs of the DQN algorithm had a average score higher than 200. At first, a random selection of the hyperparameters was done and the results are displayed in figure [8] and [9]. Note that these initial simulations where performed in the environment without wind.

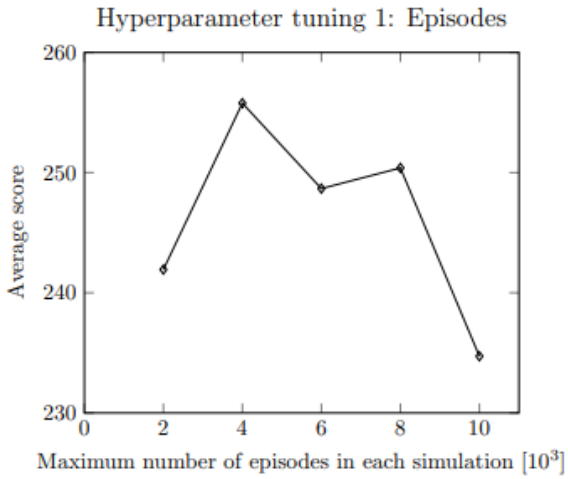

Fig. 8. Score evolution over number of episodes

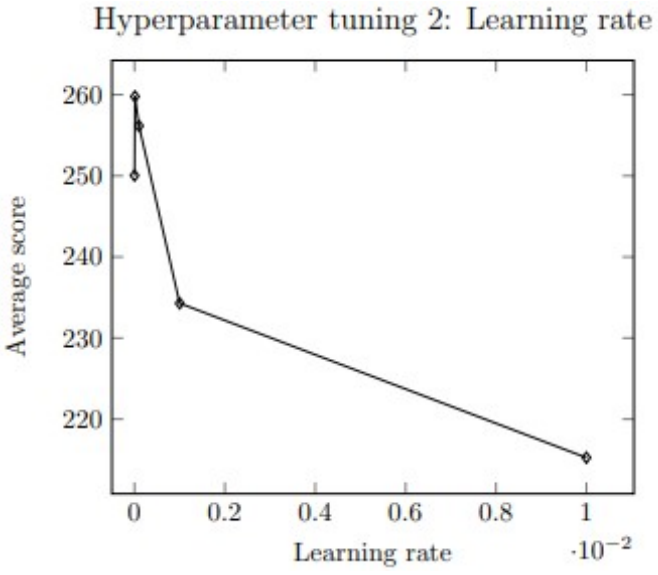

Fig. 9. Score evolution over learning rates

From figure [8] and [9], we can observe that increasing all the hyperparameters does not necessarily mean that the average score will also increase. Therefore, the simple grid search method was used to find the best hyperparameters combination. The figure [10] represents the plot of the score for each simulation over 2000 episodes with the DQN algorithm.

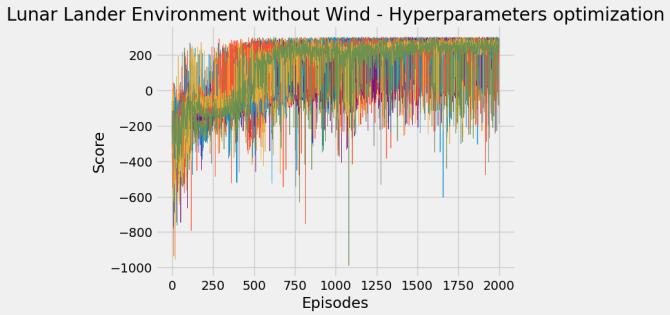

Fig. 10. DQN algorithm results in the lunar lander environment with no wind

The figure [11] represents the plot of the score for each simulation over 2000 episodes with the DQN algorithm with wind in the environment.

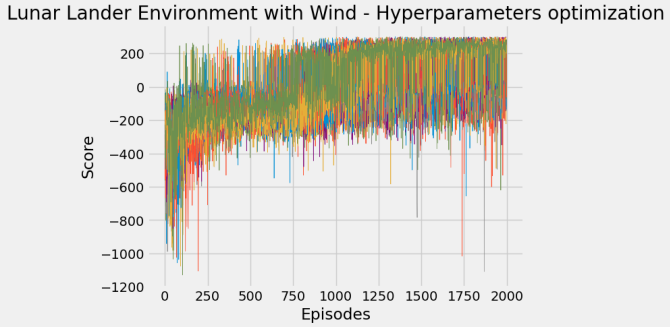

Fig. 11. DQN algorithm results in the lunar lander environment with wind

With an average score of 266 with no wind and 219 with wind over 2000 episodes and 1000 steps, the best hyperparameter combination out of those 16 simulations for DQN was:

Hyperparameters	Values (No wind)	Values (Wind)
Buffer size	1e5	1e5
Batch size	256	256
Discount factor	0.999	0.99
Learning rate	1e-4	1e-7

TABLE II
BEST HYPERPARAMETERS COMBINATION FOR DQN ALGORITHM

The DDQN was also successful in the two environments. With an average score of 261 for the environment with no wind and 217 in the wind environment and with the same hyperparameters configuration as the DQN, the best hyperparameter combination out of that grid search simulation for the DDQN was:

Hyperparameters	Values (No wind)	Values (Wind)
Buffer size	1e5	1e5
Batch size	64	64
Discount factor	0.999	0.99
Learning rate	1e-4	1e-7

TABLE III
BEST HYPERPARAMETERS COMBINATION FOR DDQN ALGORITHM

The figure [12] shows a plot of all the simulations performed by the simple grid search.

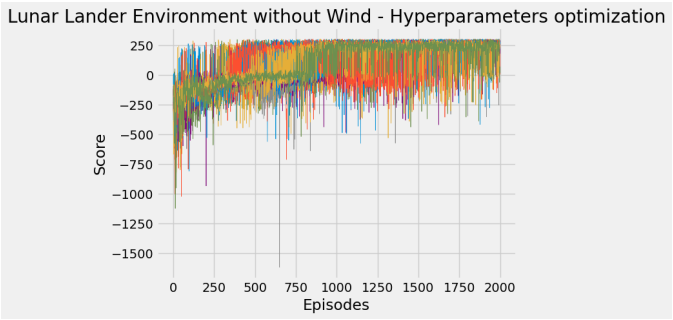

Fig. 12. DDQN algorithm results in the lunar lander environment with no wind

The figure [13] shows a plot of all the simulations performed by the simple grid search.

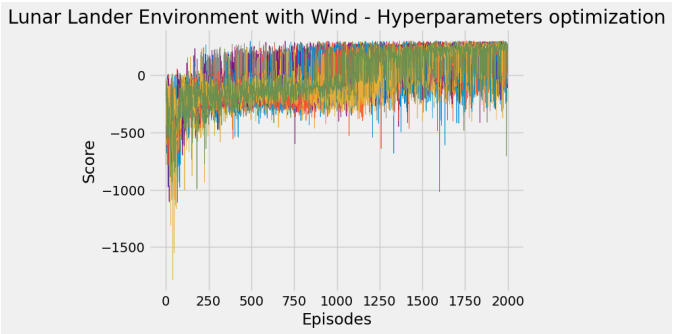

Fig. 13. DDQN algorithm results in the lunar lander environment with wind

In summary, the DQN and DDQN algorithms were solutions for both environments as their average score was around 265 which is above the solution threshold score of 200 defined by OpenAI. It is noticeable that the performance of the algorithm drops slightly to an average score of roughly 220 when wind was introduced in the lunar lander environment. Furthermore, after implementing the DDQN algorithm, the average score oscillations decreased and the learning process was more stable in comparison to the DQN algorithm. However, by introducing additional computations in the learning process of the DDQN, the computational time per learning steps increased and made it slower than the

DQN. Finally, another interesting result was the difference of the batch size in the DQN and the DDQN to achieve the best average score. After the simple grid search simulation, the batch size was 256 for DQN and 64 for DDQN. One of the reasons could be that by increasing the batch size in the DQN algorithm, the overestimation bias will be reduced because the latter algorithm tends to overestimate the Q-values at the early stages of the learning process. The stability of the DQN could have also been improved by the large batch size due to the more accurate estimate of the gradients during backpropagation.

VI. Conclusion

In conclusion, the heuristic policy was able to consistently solve the environment both with and without wind, although losing some performance in the latter situation. In either case it would be a good rollout policy to use for more powerful algorithms like MCTS. The Q-learning algorithm was not working for the lunar lander environment which has a continuous state space. Indeed, the standard Q-learning is formulated in a way that it outputs the expected Q-value for each possible action. Therefore, in a continuous state space, the number of possible of actions is infinite and the Q-values cannot be estimated. The lunar lander environment could only be solved with the Q-learning algorithm if the state space was discretized.

The DQN algorithm was able to handle that continuous state space perfectly and solve the environment. However, the oscillations and instability in its learning process were quite elevated and that was main reason to implement the DDQN. By utilizing two separate neural networks to select the best action and estimate the Q-value of that action, the DDQN was able to reduce the overestimation bias of the DQN and increase the stability of the learning process.

For future work, implementing a prioritized experience replay buffer to our DQN would be an interesting extension of this study. Since DQN samples from the replay buffer the experiences of the agent uniformly, the algorithm tends to exploit non-informative experiences. With the prioritized experience replay technique, priorities will be assigned to experiences based on their TD errors (difference between the agent's current estimate and target value). Therefore, experiences with higher priorities will be sampled more frequently which will also lead to faster learning and better convergence of the model.

Contributions and Release

- Abdoulaye Diallo: Used off-the-shelf implemented DQN, implemented DDQN algorithm, implemented grid search algorithm for DQN and DDQN, wrote the solution approach, results, and conclusion sections for DQN and DDQN, fully wrote the abstract and the introduction.
- Benjamin Chupik: Adapted an off the shelf implementation of Q-learning and created the MCST algorithm. Wrote the MCTS solution approach and results. Wrote the results for Q-learning and the background and related work section.
- Made heuristic policy and wrote about heuristic policy and the problem formulation.

The authors grant permission for this report to be posted publicly.

References

[1] U. Kose, M. Guzel, and G. E. Bostanci, "Using a deep q learning algorithm a rocket descends vertically to a specified target," 03 2022.

[2] S. Gadgil, Y. Xin, and C. Xu, "Solving the lunar lander problem under uncertainty using reinforcement learning," in *2020 SoutheastCon*, vol. 2, pp. 1–8, IEEE, 2020.

[3] J.-Y. Hsiao, Y. Du, W.-Y. Chiang, M.-H. Hsieh, and H.-S. Goan, "Unentangled quantum reinforcement learning agents in the openai gym," *arXiv preprint arXiv:2203.14348*, 2022.

[4] M. E. Taylor, N. Nissen, Y. Wang, and N. Navidi, "Improving reinforcement learning with human assistance: an argument for human subject studies with hippo gym," *Neural Computing and Applications*, pp. 1–11, 2021.

[5] D. Dharrao, S. Gite, and R. Walambe, "Guided cost learning for lunar lander environment using human demonstrated expert trajectories," in *2023 International Conference on Advances in Intelligent Computing and Applications (AICAPS)*, pp. 1–6, IEEE, 2023.