Comparative Analysis of DQN, Double DQN, and Dueling DQN Algorithms in Solving the Lunar Lander Environment

1st Lucas Ribeiro do Rêgo Barros
Instituto Tecnológico de Aeronáutica
São José dos Campos, Brazil
lucas.barros.8924@ga.ita.br

2nd Luiz Felipe Vezzali de Paula
Instituto Tecnológico de Aeronáutica
São José dos Campos, Brazil
luiz.paula.8735@ga.ita.br

3rd Mateus Pereira Alves
Instituto Tecnológico de Aeronáutica
São José dos Campos, Brazil
mateus.alves.8808@ga.ita.br

***Abstract*—This paper presents the implementation and results of three different algorithms tested for the Lunar Lander environment from OpenAI's Gym library. The algorithms tested are Deep Q-Learning (DQN), Double Deep Q-Learning and Dueling Deep Q-Learning, the last two being variations of the vanilla DQN. The model, state and action spaces are presented and so are the evolution of agents performance across training and evaluation. At the end of training, the agents were capable of having a 94% of success rate using vanilla DQN, while the success rate was 95% using Double DQN and 97% when using Dueling DQN. In general, the agents were capable of performing a game score of 270+.**

***Index Terms*—Reinforcement Learning, Lunar Lander, DQN, Double DQN, Duel DQN,**

I. Introduction

In recent years, reinforcement learning (RL) has emerged as a powerful paradigm for training agents to make decisions in complex environments. Unlike supervised learning, where the model learns from labeled data, RL involves an agent interacting with its environment, learning from the rewards or penalties received for its actions. This trial-and-error approach allows the agent to develop policies that maximize long-term rewards.

The Deep Q-Network (DQN), developed by DeepMind, represents a significant improvement in the field of reinforcement learning. Introduced in 2013, DQN combines Q-learning with deep neural networks to enable agents to learn and make decisions in high-dimensional state spaces [1]. This breakthrough was demonstrated on the Atari 2600 game environment, where DQN agents achieved human-level performance across a variety of games by learning directly from raw pixel inputs. Key innovations such as experience replay, which stabilizes learning by breaking the correlation between sequential observations, and the use of target networks to reduce the divergence of Q-value updates, were critical in achieving this success. DQN has since served as a foundation for numerous advancements in deep reinforcement learning.

A significant challenge in reinforcement learning is dealing with high-dimensional state and action spaces. Traditional Q-Learning, a foundational RL algorithm, uses a tabular approach to store state-action values, which becomes impractical as the dimensionality increases. To address this, Deep Q-Learning (DQN) was introduced, leveraging deep neural networks to approximate the Q-values, thus enabling the handling of more complex environments.

Building upon the DQN framework, researchers have developed several enhancements to improve learning stability and performance. Double Deep Q-Learning mitigates the overestimation bias often encountered in Q-Learning by decoupling the selection and evaluation of actions. Dueling Deep Q-Learning introduces a network architecture that separately estimates the state value and the advantage for each action, providing more robust learning updates.

This paper explores the implementation and comparative performance of these three algorithms within the Lunar Lander environment from OpenAI's Gym library.

II. Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning where agents learn to make decisions by interacting with an environment. At each time-step the agent gets a observation from the environment ($\mathcal{S}_t$), performs a action ($\mathcal{A}_t$), transits to a new state ($\mathcal{S}_{t+1}$) and receives feedback in the form of rewards or penalties ($\mathcal{R}_{t+1}$) [2]. The agent's goal is to maximize the cumulative reward over time. This representative dynamics is summarized in figure 1.

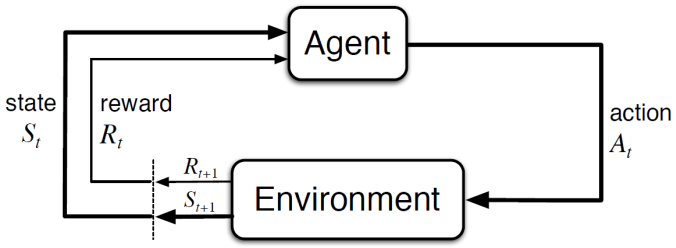

Fig. 1. Dynamics of RL problems.

Based in this dynamics, RL algorithms are based in Markov Decision Processes (MDPs), defined by a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$. With $\mathcal{S}$ standing for the state space, which includes all possible states in the environment. The $\mathcal{A}$ is the action space, holding all possible actions that the agent can take. $p = p(s' \mid s, a)$ represents the probability of transitioning to state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ by performing action $a \in \mathcal{A}$. $r = r(s, a)$ is a numerical reward received by

the agent for taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. Finally, $\gamma \in [0,1]$ is the discount factor, for future rewards influence.

The goal of an RL agent is to learn a policy $\pi(a \mid s)$ that maximizes the expected cumulative reward, also known as the return, which is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where r_{t+k+1} is the reward received at time step $t+k+1$.

III. Problem Description

This project addresses the challenge of solving the LunarLander-v2 environment from the OpenAI's Gym toolkit using reinforcement learning (RL) methods. OpenAI's Gym is a toolkit that provides challenging benchmarks for developing and comparing reinforcement learning algorithms. The LunarLander-v2 environment simulates the scenario where a lunar module must successfully land on a designated location under low-gravity conditions. The environment is built upon a well-defined physics engine, providing a realistic and complex simulation for RL training. The scenario randomly generates surfaces at every episode. Our environment is defined by it's state space, action space and rewards. The state space is continuous as is real physics, but the action space is discrete.

State Space - The state space form our environment is an eight-dimensional vector

$$\textbf{state} = \begin{bmatrix} x & y & v_x & v_y & \theta & \omega & leg_L & leg_R \end{bmatrix}^{\mathrm{T}}$$

which represents the following aspects of the lander's state:

$$\textbf{state} \rightarrow \begin{cases} x: \text{Horizontal position} \\ y: \text{Vertical position} \\ v_x: \text{Horizontal velocity} \\ v_y: \text{Vertical velocity} \\ \theta: \text{Angular orientation in space} \\ \omega: \text{Angular velocity} \\ leg_L: \text{Left leg touching the ground (Boolean)} \\ leg_R: \text{Right leg touching the ground (Boolean)} \end{cases}$$

The landing goal's position represents the origin of the coordinate system, therefore, the lander's position is expressed relative to this origin.

Action Space - The action space consists of four discrete actions that can control the lander's descent and orientation.

$$\textbf{action} \rightarrow \begin{cases} \text{Do nothing} \\ \text{Fire right engine} \\ \text{Fire main engine} \\ \text{Fire left engine} \end{cases}$$

Reward Structure - The reward structure is designed to encourage the agent to land softly and accurately at the designated landing pad in minimal time steps. The reward table is found in table I.

TABLE I
Reward Structure for LunarLander-v2 Environment

Situation	Reward
Side engine firing	-0.03 points per frame
Main engine firing	-0.3 points per frame
Crashing	-100 points
Leg in contact with ground	+10 points per leg
Safe landing	+100 points

In addition, reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. If the lander moves away from the landing pad, it loses reward, and if the lander moves faster, it loses reward. An episode is considered a solution if it scores at least 200 points. No additional reward engineering was required to solve the environment. The representation of the problem is found in figure 2.

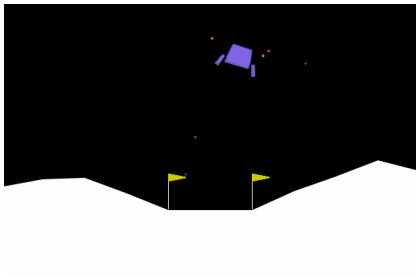

Fig. 2. Representation of the problem.

IV. Model

One of the most famous algorithm in RL is the Q-Learning which is a model-free algorithm used to find the optimal policy in a MDP. It operates by learning the value of action-state pairs, known as Q-values, which represent the expected utility of taking a given action in a given state and following the optimal policy thereafter, these values are stored in a lookup table, being a great approach for discrete state space problems.

Under a given policy π, the action-value function is defined as the expected return starting from state s and action a, as follows.

$$\mathcal{Q}_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]$$

The Q-learning algorithm iteratively updates the Q-values using the Bellman equation. The update rule for Q-learning is given by:

$$\mathcal{Q}(s,a) \leftarrow \mathcal{Q}(s,a) + \alpha\left[r + \gamma \max_{a'} \mathcal{Q}(s',a') - \mathcal{Q}(s,a)\right]$$

where $\mathcal{Q}(s,a)$ is the Q-value for state s and action a. α is the learning rate, r is the immediate reward received after transitioning from state s to state s' by taking action a. γ is the discount factor, which represents the importance

of future rewards. At least, $\max_{a'} \mathcal{Q}(s', a')$ is the maximum Q-value for the next state s' over all possible actions a'.

The goal of reinforcement learning is to find the optimal policy, that maximizes the expected return. In this case, the optimal action-value function $\mathcal{Q}^*(s, a)$ is defined as follows.

$$\mathcal{Q}^*(s, a) = \max_{\pi} \mathcal{Q}_\pi(s, a)$$

Given that our state vector consists of six continuous variables and two boolean variables, using a tabular approach like Q-Learning or SARSA would require discretizing these features. This means the complexity of the state space is $\mathcal{O}(n^6 \times 2 \times 2)$, where n represents the number of discrete values for each state variable. If we choose to discretize each variable into ten values, the total number of possible states would be 400,000, that is far too large to explore effectively within a reasonable number of training episodes, turning the tabular approach impracticable.

Therefore, we use three modified versions of Q-Learning: Deep Q-Learning, Double Deep Q-Learning and Duel Deep Q-Learning. In general, the DQN and variations models approximates the Q-table by utilizing neural networks to handle the continuous state space. This Deep Q-Learning approach employs a multi-layer neural network, known as a Deep Q-Network, to predict Q values.

A. ε-greedy Policy

Since DQN and its variations are off-policy algorithms, the agent learns the optimal policy (π) independently of the policy followed during training ($\mu \sim \varepsilon$-greedy). Therefore, the agent is not explicitly instructed on which actions to take but must discover which actions yield the highest rewards through trial and error. The ε-greedy policy, which rules the agent's behavior during training, is a simple strategy designed to balance exploration and exploitation. It assigns a probability $\varepsilon \in [0, 1]$ of executing a random action (exploration) and a probability $1 - \varepsilon$ of executing a greedy action, thereby maximizing rewards (exploitation).

The objective is for the agent to explore the environment extensively in the early episodes to encounter as many states as possible, which aids in the generalization process of the neural network. In the subsequent episodes, the focus shifts towards exploiting the actions that maximize the agent's rewards. This approach represents a trade-off between exploration and exploitation, which the proposed policy seeks to balance. At the end of each episode, the probability associated with exploration decays according to the following rule:

$$\varepsilon_t = \max[\varepsilon_{\min}, \varepsilon_{t-1} \cdot \epsilon]$$

where ϵ represents the decay rate of ε, and ε_{min} denotes the minimum probability of exploration in the environment.

B. Experience Replay

Experience replay improves the stability and performance of the learning process by breaking the temporal correlations between consecutive experiences. By randomly sampling state transitions from a set of past experiences, the agent can learn more robust policies that generalize better across different states. This learn method uses a replay memory buffer, which is a data structure that enables the agent to store observed transitions for a finite period. This buffer employs a uniform sampling to draw from past experiences, facilitating the network's update and learning process from it [3]. Given its fixed capacity, the buffer operates on a first-in-first-out basis, discarding the oldest experiences when it's full to accommodate new data.

C. Deep Q-Learning

In order to approximate the value function, we use a deep Q-network: $\mathcal{Q}(s, a, \theta)$. For an n-dimensional state space and a m-dimensional action space, the network is a function $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, $\mathcal{N} = \mathcal{N}(\theta)$ that takes the current state (which is 8-dimensional vector) as input and outputs the Q values for all possible state-action pairs associated with that state. At each timestep t, the agent perceives the current state s_t, selects an action a_t following an ε-greedy policy, receives a corresponding reward r_t, and transitions to the next state s_{t+1}. This transition is represented as a tuple $(s_t, a_t, r_t, s_{t+1}, \text{done})$ and is subsequently appended to a replay memory buffer and latter is used for learning. The update rule in the parameters of the neural network are trained by gradient descent to minimize the following loss function:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(Y^{DQN} - \mathcal{Q}(s, a, \theta)\right)^2\right] \tag{1}$$

The target value is predicted by the network as well:

$$Y^{DQN} = \begin{cases} r & \text{if terminal,} \\ r + \gamma \max_{a'} \mathcal{Q}(s', a', \theta) & \text{otherwise} \end{cases} \tag{2}$$

In the original paper that introduced the DQN algorithm, DeepMind's implementation used a CNN since the inputs were the game screen [1]. In our problem, the input is the state vector, which means we use a fully connected neural network (FCNN) instead of a convolutional neural network (CNN). In this implementation no target network was needed to stabilize learning, a single network is used to predict all Q-values, which is referred as the online network in literature, in this case, both $\mathcal{Q}(s, a)$ and Y^{DQN} are predicted by the network. The network architecture used to solve the problem was based in [4], and can be summarized as follows in table II. Adam was used as optmizer.

TABLE II
NEURAL NETWORK ARCHITECTURE FOR DQN

Layer	# Neurons	Activation Function
Input	8 (observation space)	Linear
Hidden layer (Dense)	256	ReLu
Hidden layer (Dense)	128	ReLu
Output	4 (action space)	Linear

The pseudocode for the DQN algorithm is found in algorithm 1. The value **done** stands for a boolean: true if episode has finished, false otherwise.

Algorithm 1 Deep Q-Learning

Input: Environment, hyperparams, number of episodes, minibatch size
Output: Trained network (Q-function)

Initialize parameters of the Q-network and replay memory buffer
for episode < max_episodes **do**
  Reset env
  while not done (crashed or landed) **do**
    Choose action **a** $\sim \varepsilon$-greedy policy
    Given **a**, observe the state transition (s_t, a_t, r_t, s_{t+1}, done*)
    if Replay memory is full **then**
      remove oldest transition
    end if
    Store the state transition (s_t, a_t, r_t, s_{t+1}, done) in the buffer
    if replay buffer size ≥ minibatch size **then**
      Sample a batch of past experiences
      Calculate target values Y^{DQN} in eq (2)
      Update parameters of Q by gradient descendent with learning rate α through the loss function in eq (1)
    end if
    $\varepsilon \leftarrow \max\left[\varepsilon_{\min}, \varepsilon \times \epsilon_{\text{decay}}\right]$
  end while
end for=0

D. Double Deep Q-Learning

Despite the effectiveness of Deep Q-Learning in solving many reinforcement learning problems, it has been observed to overestimate action values due to the use of the same value function to select and to evaluate an action [5]. This overestimation can lead to suboptimal policies. To address this issue, Double Deep Q-Learning (Double DQN) was introduced, which decouples the selection and evaluation of the action, thereby reducing overestimation bias.

In Double DQN, two separate networks are maintained: the policy/online network ($\mathcal{Q}_\pi$) and the target network ($\mathcal{Q}_\text{T}$). The policy network is used to select the action, while the target network is used to evaluate the action. The action-value function for Double DQN is defined as:

$$\mathcal{Q}_\pi(s,a) = \mathbb{E}\left[R_t + \gamma \mathcal{Q}_\text{T}(s', \text{argmax}_{a'} \mathcal{Q}_\pi(s',a')) | S_t = s, A_t = a\right]$$

The target value for Double DQN is given by:

$$Y^{DDQN} = \begin{cases} r & \text{if terminal,} \\ r + \gamma \mathcal{Q}_T(s', \text{argmax}_{a'} \mathcal{Q}_\pi(s',a'), \theta_T) & \text{otherwise} \end{cases} \tag{3}$$

Like in DQN, at each timestep t, the agent perceives the current state s_t, selects an action a_t using an ε-greedy policy based on $\mathcal{Q}_\text{policy}$, executes the action a_t, receives a reward r_t, and observes the next state s_{t+1}. The transition $(s_t, a_t, r_t, s_{t+1}, \text{done})$ is stored in the replay buffer. Thus, the agent samples a random minibatch of transitions from the replay buffer. Then, for each sampled transition $(s_i, a_i, r_i, s_{i+1}, \text{done})$, it computes the target value and loss. Finally, it performs a gradient descent step to update the parameters of the policy network following the loss function:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(Y^{DDQN} - \mathcal{Q}_\pi(s,\, a,\, \theta_\pi)\right)^2\right] \tag{4}$$

The target network weights are periodically synchronized with the policy network weights to improve training stability. The architectures of the networks employed to address this problem are identical and based on the work described in [4]. Table III provides a summary of the architecture. The Adam optimizer was utilized for training.

TABLE III
NEURAL NETWORK ARCHITECTURE FOR DDQN

Layer	# Neurons	Activation Function
Input	8 (observation space)	Linear
Hidden layer (Dense)	128	ReLU
Hidden layer (Dense)	128	ReLU
Output	4 (action space)	Linear

The pseudocode for the Double DQN algorithm is as follows in algorithm 2. The value **done** represents the same.

E. Dueling Deep Q-Learning

Dueling Deep Q-Learning (Dueling DQN) is an enhancement to the standard DQN architecture that helps the agent learn which states are (or are not) valuable, regardless of the action taken in those states. This approach separates the representation of the state value and the advantage for each action, combining them to produce the final Q-values.

In the Dueling DQN architecture, the neural network is structured into two separate streams: one for estimating the state value function $V(s)$ and the other for estimating the advantage function $A(s,a)$ [6]. These two streams are then combined to produce the Q-values using the following equation:

$$\mathcal{Q}(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')\right)$$

where $\mathcal{A}$ is the set of possible actions.

This architecture can help the learning process by allowing the network to learn the value of a state independently of the advantages of each action in that state, leading to a more stable learning process.

The architecture of the Dueling DQN network used in this study is summarized in Table IV.

In this implementation, we use a single neural network to represent both the state value and the advantage streams.

Algorithm 2 Double Deep Q-Learning

Input: Environment, hyperparameters, number of episodes, minibatch size
Output: Trained policy network

Initialize parameters of the policy network ($\mathcal{Q}_\pi$) and target network ($\mathcal{Q}_T$) with same parameters and weights
Initialize replay memory buffer
for episode < max_episodes **do**
  Reset environment
  while not done (crashed or landed) **do**
    Choose action **a** $\sim \varepsilon$-greedy policy based on $\mathcal{Q}_\pi$
    Execute action **a**, observe state transition (s_t, a_t, r_t, s_{t+1}, done)
    Store transition (s_t, a_t, r_t, s_{t+1},done*) in replay buffer
    if replay buffer size ≥ minibatch size **then**
      Sample random minibatch of transitions from replay buffer
      Calculate target values Y^{DDQN} in eq (3)
      Update parameters of $\mathcal{Q}_\pi$ by gradient descendent with learning rate α through the loss function in eq (4)
    end if
  end while
  $\varepsilon \leftarrow \max(\varepsilon_{\min}, \varepsilon \times \epsilon_{\text{decay}})$
  if Every τ steps **then**
    Update target network ($\mathcal{Q}_T$) weights with the policy network ($\mathcal{Q}_\pi$) weights
  end if
end for=0

TABLE IV
NEURAL NETWORK ARCHITECTURE FOR DUEL DQN

Layer	# Neurons	Activation Function
Input	8 (observation space)	Linear
Hidden layer (Dense)	256	ReLU
Hidden layer (Dense)	128	ReLU
$V(s)$ Stream (Dense)	1	Linear
$\mathcal{A}(s,a)$ (Dense)	4 (action space)	Linear
Output	4 (action space)	Linear

The model is a non-sequential neural network, that after the second hidden layer it branches into two different streams [6]. The fist computes the state value and the other computes the advantage function. After that, the last layer combines the outputs of these streams to produce the Q-values for each action. The training process for the Duel DQN follows the same general procedure as the DQN. The Dueling DQN uses the same target and loss from DQN (eq (2) and (1)). Besides this key difference in the model architecture used, the algorithm for Dueling DQN is exactly the same as the DQN (algorithm 1).

V. RESULTS

For all three algorithms, the tranning parameters used follows in table V. The learning rate and the initial value of ε were varied as shown in the table (assumed the two values), and all combinations were tested to identify the optimal configuration for each algorithm.

TABLE V
HYPERPARAMETERS USED FOR TRAINING ALL AGENTS

Hyperparameter	Value
Learning Rate (α)	$5 \cdot 10^{-5}$ / 10^{-3}
γ	0.99
$\varepsilon_{\text{Start}}$	0.5 / 1.0
$\varepsilon_{\min}$	0.1
ϵ (ε decay)	0.994
Mini Batch Size	2^6
Buffer Size	2^{16}

- **Learning Rate** (α): determines the step size during gradient descent optimization.
- **Discount Factor** (γ): represents the importance of future rewards.
- **Initial Exploration Rate** ($\varepsilon_{\text{Start}}$): initial probability of selecting a random action instead of the greedy action derived from the policy.
- **Minimum Exploration Rate** (ε_{min}): lower bound for the exploration rate, ensuring that there is always some level of exploration throughout the training process to avoid local optima.
- **Exploration Rate Decay** (ϵ (ε**decay**)): determines the rate at which the exploration rate ε decreases over time.
- **Mini Batch Size**: number of experiences sampled from the replay buffer for each training step.
- **Buffer Size**: defines the capacity of the replay buffer, which stores past experiences.

The training process was conducted such that, for each algorithm, and for each combination of learning rate (α) and initial exploration rate ($\varepsilon_{\text{Start}}$), three distinct agents were trained. This resulted in a total of 36 trained agents. All the results for the agents can be found on the group's GitHub. We present the best two results for each algorithm in the following three subsections.

For the next sections, consider that a agent solves the environment when it achieves a moving average game score of 200 over the last 100 episodes. The graphics in figure 3 to 8 shows the game score obtained during 2000 training episodes (light blue line), the moving average over the last 100 episodes (dark blue line), and the game score obtained during the evaluation of 100 episodes (red line).

A. DQN Results

Overall, for the agents trained with the DQN algorithm, the learning rate did not significantly impact performance during training or evaluation, with both rates resulting in very high and similar performance, exceeding expectations. However, using a lower learning rate resulted in an average increase of 30% in training time for agents with $\varepsilon = 1.0$. No significant change in training time was observed for agents with $\varepsilon = 0.5$. The variation of ε affected the algorithm's

convergence, with agents trained with $\varepsilon = 0.5$ solving the environment in approximately 46% fewer episodes compared to agents trained with $\varepsilon = 1.0$. The overall success rate of agents trained with DQN was approximately 94% during testing. Figures 3 and 4 show two graphs of DQN agents displaying the graphic results. The average training time for a DQN agent was approximately 2 hours.

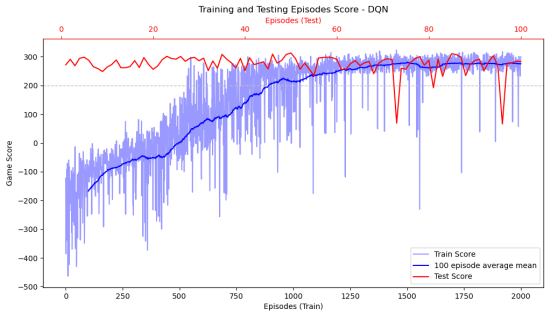

Fig. 3. DQN Algorithm. Hyperparams: $\alpha = 5 \cdot 10^{-5}$; $\varepsilon = 1.0$; Testing mean game score ≈ 274; Testing success rate = 97%; Minimum episodes to solve = 894.

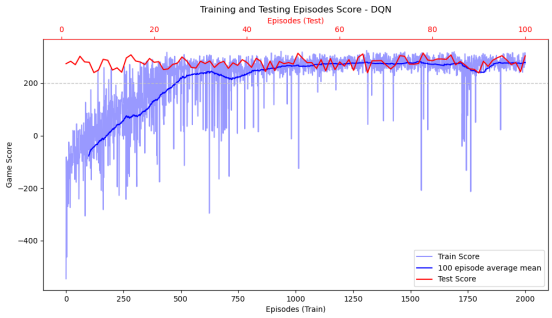

Fig. 4. DQN Algorithm. Hyperparams: $\alpha = 10^{-4}$; $\varepsilon = 0.5$; Testing mean game score ≈ 276; Testing success rate = 100%; Minimum episodes to solve = 480.

B. Dueling DQN Results

For the Dueling DQN, a lower learning rate resulted in slightly higher performance, although not significantly. Both learning rates showed high and similar performance. However, similar to the previous algorithm, using a lower learning rate resulted in an average increase of 38% in training time for agents with $\varepsilon = 1.0$. No significant change in training time was observed for agents with $\varepsilon = 0.5$. As with DQN, using $\varepsilon = 0.5$ resulted in resolving the environment in 40% fewer episodes, maintaining similar and high levels of performance. The overall success rate of agents trained with Dueling DQN was approximately 97% during testing. Figures 5 and 6 show two graphs of Dueling DQN agents displaying the graphic results. The average training time for a Dueling DQN agent was approximately 2 hours.

C. Double DQN Results

Finally, for the Double DQN algorithm, similar to the Dueling DQN, using a lower learning rate resulted in slightly

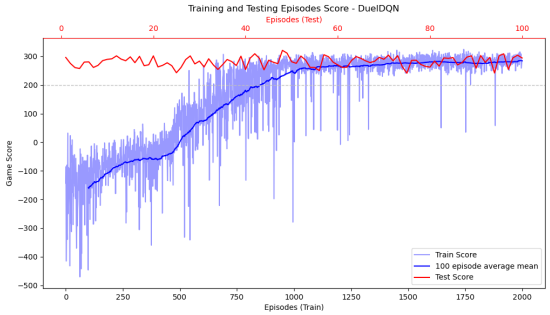

Fig. 5. Dueling DQN Algorithm. Hyperparams: $\alpha = 5 \cdot 10^{-5}$; $\varepsilon = 1.0$; Testing mean game score ≈ 281; Testing success rate = 100%; Minimum episodes to solve = 853.

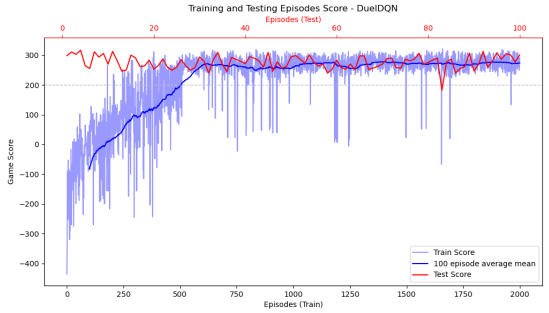

Fig. 6. Dueling DQN Algorithm. Hyperparams: $\alpha = 5 \cdot 10^{-5}$; $\varepsilon = 0.5$; Testing mean game score ≈ 276; Testing success rate = 99%; Minimum episodes to solve = 505.

lower performance, though not significantly. However, similar to the previous algorithm, using a lower learning rate resulted in an average increase of 42% in training time for agents with $\varepsilon = 1.0$. No significant change in training time was observed for agents with $\varepsilon = 0.5$. Using $\varepsilon = 0.5$ also resulted in a 34% reduction in the number of episodes required to solve the environment. The performance of the algorithms was similarly high during both training and testing. The overall success rate of agents trained with Double DQN was approximately 95% during testing. Figures 7 and 8 show two graphs of Double DQN agents displaying the graphic results. The average training time for a Double DQN agent was 2 hours and 10 minutes.

Thereat, there´s a comparison of the game score moving average from figures 3, 5, and 7 in figure 9 and the same comparison from figures 4, 6, and 8 in figure 10. These trained agents plotted showed great performances and very similar.

Finally, the 100-episode moving average for all 36 agents trained is plotted in figure 11. The interesting aspect here is the visualization of the gap caused by the two different $\varepsilon_{\text{start}}$. The learning rate typically affects the slope of the curve. Another notable observation is that all trained agents resolved the environment in under 1000 episodes.

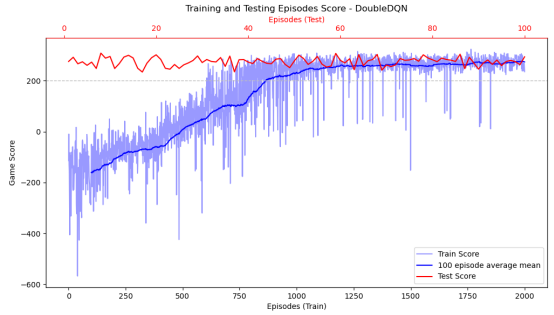

Fig. 7. Double DQN Algorithm. Hyperparams: $\alpha = 5 \cdot 10^{-5}$; $\varepsilon = 1.0$; Testing mean game score ≈ 275; Testing success rate = 100%; Minimum episodes to solve = 865.

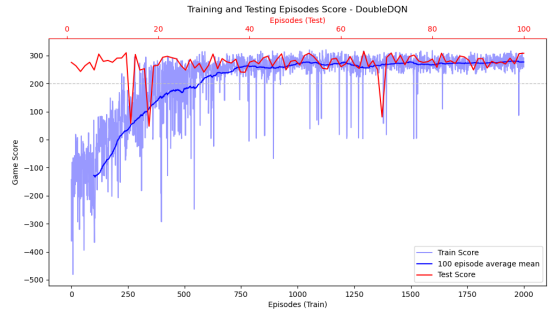

Fig. 8. Double DQN Algorithm. Hyperparams: $\alpha = 5 \cdot 10^{-5}$; $\varepsilon = 0.5$; Testing mean game score ≈ 271; Testing success rate = 97%; Minimum episodes to solve = 567.

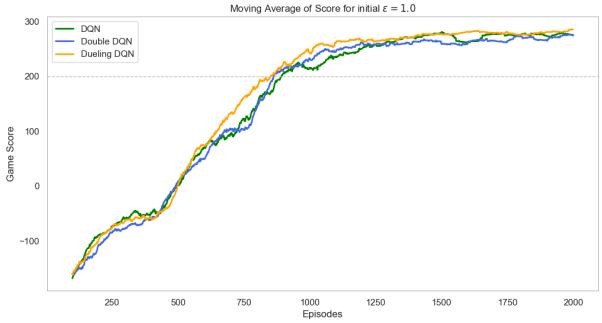

Fig. 9. Comparison of the game score moving average for the best result of each algorithm for agents with $\varepsilon_{start} = 1.0$

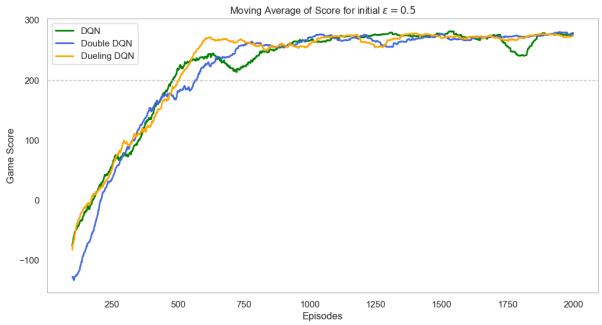

Fig. 10. Comparison of the game score moving average for the best result of each algorithm for agents with $\varepsilon_{start} = 0.5$

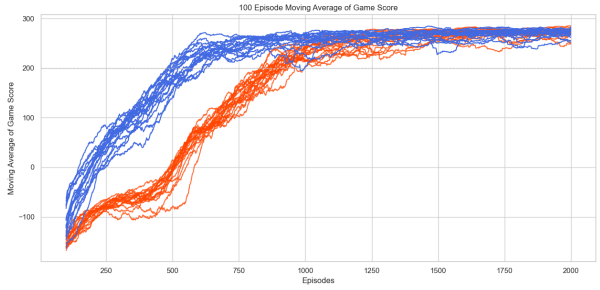

Fig. 11. 100 episode moving average comparison for all 36 trained agents. The curves in blue represent the agents trained with $\varepsilon_{start} = 0.5$. Agents trained with $\varepsilon_{start} = 1.0$ are ploted in orange

VI. Conclusions

Therefore, in this work, an extensive evaluation of three deep reinforcement learning algorithms—Deep Q-Network (DQN), Dueling DQN, and Double DQN—was conducted across various hyperparameter configurations. This paper explored the implementation and comparative performance of these three algorithms within the Lunar Lander environment from OpenAI's Gym library, focusing on understanding how different combinations of learning rates (α) and initial exploration rates ($\varepsilon_{\text{Start}}$) influence the training dynamics and performance of these algorithms on the environment.

Across the board, reducing the initial exploration rate ($\varepsilon_{\text{Start}}$) consistently accelerated the learning process for all algorithms tested. Agents initialized with $\varepsilon_{\text{Start}} = 0.5$ solved the environment in notably fewer episodes compared to those with $\varepsilon_{\text{Start}} = 1.0$. This underscores the importance of balancing exploration and exploitation effectively in reinforcement learning tasks.

All algorithms demonstrated high success rates during testing, ranging from 94% to 97%. Dueling DQN consistently achieved the highest average game scores, suggesting its enhanced learning efficiency under the studied configurations. As an episode would be considered a solution if it scored at least 200 points, considering that the agents were capable of scoring more than 270 points, it is possible to say that the trainings presented were successful.

From a practical standpoint, these findings highlight the importance of hyperparameter tuning in optimizing reinforcement learning algorithms. The choice of initial exploration rate ($\varepsilon_{\text{Start}}$) significantly influences both convergence speed and final performance, making it a critical parameter to consider in algorithm design. Additionally, while learning rate (α) adjustments can affect training dynamics, its impact on final performance may vary across different algorithm architectures.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through

deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[3] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," 2020. [Online]. Available: https://arxiv.org/abs/2007.06700

[4] X. Yu, "Deep q-learning on lunar lander game," 05 2019.

[5] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.06461

[6] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016. [Online]. Available: https://arxiv.org/abs/1511.06581