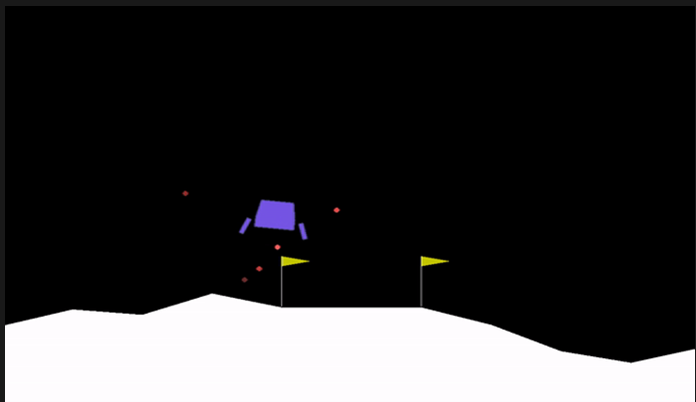
**Hackaton 4 - Lunar Lander**

**From random agents to Deep Learning**

****

**Cristian Brugnara, Endrit Nazifi, Luca Lucchina**

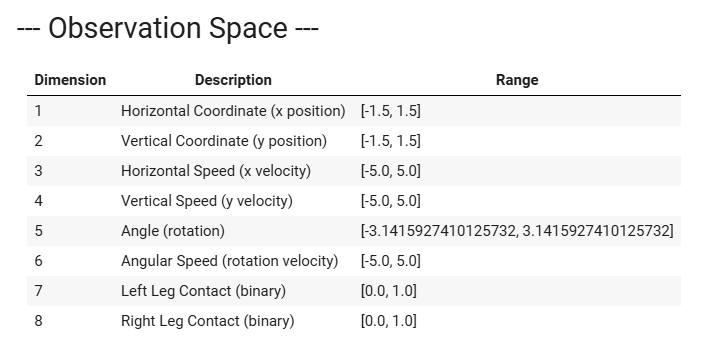
**1.Introduction**

The goal of this project is to develop a Reinforcement Learning (RL) system to control the lunar lander in the *LunarLander-v2* environment, provided by the *Gymnasium* library (formerly OpenAI Gym). The primary task is to land the spacecraft safely and stably while experimenting with different configurations and learning strategies to optimize the agent’s performance.

During the hackathon, we were given the freedom to explore various approaches to achieve a successful landing. To systematically tackle the challenge, we first conducted an exploratory analysis of the environment, focusing on understanding the observation and action spaces. This preliminary step is essential to ensure that the agent can make informed decisions based on meaningful state representations.

The *observation space* consists of eight key parameters that describe the lander's state, including its horizontal and vertical position, velocity, rotation angle, angular velocity, and the contact status of its landing legs with the ground. The *action space* is discrete and consists of four possible actions: no thrust, firing the main engine for upward thrust, and activating the left or right thrusters for lateral movement.

Understanding these aspects of the environment was a first step in designing a reinforcement learning model that could effectively learn to land the agent. This exploration provided insights into how different variables influence the agent’s behavior and how the reward system interacts with different landing strategies.



The action space was smaller of course: "Do nothing", "Fire right engine","Fire central engine","Fire left engine".

To better understand the behavior of the lunar lander and how different actions affect its trajectory, we conducted a series of simulations, each corresponding to a specific action: no thrust, main engine thrust, left engine thrust, and right engine thrust.

**2. Approach and Methodology**

# ****2.1 Random Agent****

### **Objective**

The first approach implemented was a Random Agent, an agent that selects actions randomly without any learning or strategy. This was done to establish a baseline for comparison with more advanced models. The expectation was that the agent would perform poorly, as it lacks any structured decision-making process.

### **Implementation**

The simulation involved:

* Running **100 episodes** to evaluate the performance of the agent.
* Each episode was limited to a **maximum of 500 steps** to prevent infinite runs.
* Actions were selected randomly from the **discrete action space**
* For each step, the environment returned an observation, a reward, and a termination signal indicating whether the episode had ended (e.g., due to landing or crashing).
* The total reward was recorded at the end of each episode to analyze performance trends.

### **Results & Analysis**

#### **Total Reward Distribution**

A histogram of the total rewards across **100 episodes** showed that most rewards were negative, as expected. Since the agent moves randomly, it fails to control the lander effectively, often leading to crashes or inefficient landings. The **mean reward** was approximately **-200**, with a high variance indicating inconsistent performance.

A cumulative average reward plot was generated to observe trends over multiple episodes. Unlike learning-based agents, the Random Agent showed **no improvement over time**, as expected. The average quickly stabilized at negative values, further confirming the lack of any meaningful learning or adaptation.

# ****2.2 Accounting Agent****

### **Objective**

The Accounting Agent is an early attempt at creating a rule-based decision-making system that improves upon the purely random strategy. Unlike the Random Agent, which selects actions without any reasoning, the Accounting Agent follows a **greedy approach**, always choosing the action that has accumulated the highest total reward so far. The goal of this experiment was to determine whether such a simplistic reward-based memory system could lead to better performance in the LunarLander-v2 environment.

### **Implementation**

The agent maintains a basic **action-reward memory**, which stores the total accumulated rewards for each action.

1. **Tracking rewards**: The agent initializes an array where each action's total reward is stored.
2. **Action selection (greedy approach)**: At every step, the agent selects the action that has the highest accumulated reward so far.
3. **Updating reward memory**: After executing the chosen action, the agent updates the corresponding value in the action-reward array.
4. **Simulation: 1,000 episodes** were run to assess long-term performance; each episode had a **maximum of 500 steps**.

### **Results & Analysis**

#### **Total Reward Distribution**

A histogram of total rewards showed that the Accounting Agent performed equally or even **worse than the Random Agent** on average.

This is likely due to its **greedy strategy**, which forces the agent to repeat actions that initially yielded the highest rewards, **ignoring other possibilities**. Unlike a smarter approach, which balances **exploration** and **exploitation**, this agent blindly commits to its early choices.

The cumulative reward plot shows that the agent **quickly converges to a plateau of performance level**. This outcome reflects the **lack of adaptability** in the Accounting Agent.

### **Conclusion**

The Accounting Agent experiment demonstrated that **greedily selecting the action with the highest accumulated reward is not an effective strategy** in relatively complex environments like the Lunar Lander. Surprisingly, even the Random Agent was often better because, despite its lack of reasoning, it occasionally stumbled upon better sequences of actions.

# ****2.3 Q-Learning Agent****

### **Objective**

The Q-Learning Agent represents the first step toward a smarter form of learning, moving away from the purely random and greedy strategies seen in the previous agents. The goal is to enable the agent to learn **how to land safely and efficiently** by optimizing action selection through **state-action value learning**.

### **Q-Learning and Discretization Strategy**

Since Q-Learning operates in a **discrete state space**, it was necessary to discretize the originally **continuous** state space of the Lunar Lander. The discretization process divided each observation dimension into **10 (later 20) bins**, except for the binary indicators (landing leg contact states), which were treated separately. This is needed to avoid an infeasibly large Q-table that would result from trying to represent the environment more completely, as well as reducing computational complexity while maintaining representation, and ensuring generalization across similar states, preventing overfitting to specific observations.

The discretization process assigns each continuous state variable (such as position, velocity, and angle) to a **bin index**, transforming the state into a tuple that can be used as an index for the Q-table.

### **Q-Learning Parameters**

The parameters used for training the Q-Learning agent were selected to balance **exploration, exploitation, and learning progression** while ensuring comparability with the previously tested agents. The key parameters are:

**Learning Rate (α = 0.1)**

* + Controls how much newly acquired information overrides old information.
  + A low value ensures stable updates, preventing drastic changes in the Q-values.

**Discount Factor (γ = 0.99)**

* + Determines the importance of future rewards.
  + A high value encourages the agent to consider long-term rewards rather than just immediate ones.

**Exploration Rate (ε = 1.0 → 0.01)**

* + Starts with **100% random exploration** and gradually decays over time.
  + Prevents premature convergence to suboptimal policies.
  + Minimum exploration rate ensures that even later in training, the agent can still discover better strategies.

**Epsilon Decay Rate (ε\_decay = 0.995)**

* + Gradually reduces the exploration probability over time, shifting the agent from exploration to exploitation.

**Episodes and Steps Constraints**

* + **10,000 episodes** were run to allow the agent to experience a wide variety of scenarios.
  + Each episode was limited to **500 steps**, enforcing efficiency in landing strategies.

### **Action Selection - Epsilon-Greedy Policy**

The agent follows an **ε-greedy policy** for action selection, which selects a **random action** with probability **ε** (exploration) or chooses the **action with the highest Q-value** for the current state with probability **1 - ε** (exploitation).

This balance ensures that the agent **explores different strategies** initially but eventually converges towards the most rewarding area.

### **Training and Learning Process**

During training, the agent updates its Q-values using the **Bellman equation**, adjusting the expected reward for each state-action pair based on observed transitions. The update rule is:

This iterative update allows the agent to **learn from experience** and refine its action selection to maximize cumulative rewards over time.

### **Results and Observations**

#### **Total Reward Distribution**

Compared to the **Random Agent** and the **Accounting Agent**, the **Q-Learning Agent achieved a higher concentration of less negative rewards**, with some episodes reaching **positive values**. This suggests that, while not fully optimized, the agent has started to learn more effective landing strategies.

The **mean reward** is noticeably improved compared to previous agents, though still negative overall. The presence of positive rewards indicates that in certain episodes, the agent successfully completed a controlled landing.

The cumulative reward curve shows an **initial rapid improvement**, demonstrating that the agent quickly learns better policies than random selection but also a **flattening trend**, indicating that learning has stabilized but without reaching optimal performance.

Still, the **Q-Learning Agent exhibits consistent improvement**, showing that learning through **future reward consideration** and the **epsilon-greedy approach** help with refining long-term strategy and preventing the agent from being stuck in poor ones.

### **Next Steps**

The **current implementation of Q-Learning is functional but not yet optimized**. The next steps will focus on:

* **Fine-tuning hyperparameters** to enhance learning efficiency.
* **Comparing Q-Learning to more advanced methods**, such as Deep Q-Networks (DQN), which can handle continuous state spaces more effectively.

This section will be expanded after conducting further experiments to **refine the agent's performance** and analyze its true potential in solving theLunar Lander task.

# ****2.4 Optimizing the Q-Learning Agent****

### **Objective**

After implementing the initial Q-Learning Agent, it became clear that further optimization was necessary to improve its performance.

### **Step 1: Improving State Discretization**

In the initial implementation, the **state space was divided into 10 bins per dimension**. However, this level of discretization was too rough, causing the agent to **lose important distinctions between states**.

**Optimization approach:**

* The number of bins was **increased from 10 to 20**, providing a finer granularity in representing the lander's position, velocity, and angle.
* This adjustment allowed the agent to **differentiate between similar states more effectively**, leading to better decision-making.

By refining state discretization, the Q-table became **more representative of the actual environment**, enabling the agent to learn more precise policies. We then opted to increase the number also for the original agent, since the environment wasn’t too heavy.

### **Step 2: Hyperparameter Tuning via Rapid Evaluation**

Instead of **committing to 10,000 episodes for every new configuration**, a faster approach was adopted in otder to test different configurations of hyperparameters.

**A set of reasonable configurations** **was** defined based on the **learning rate, the discount factor, the ε-decay.**

**Each configuration was tested for only 500 episodes** and the **mean and standard deviation of the total reward** were recorded for each configuration to assess performance.

This approach ensured that **only the most promising hyperparameter set was used for long training**, optimizing both time and resource efficiency.

### **Step 3: Full Training with Optimized Parameters**

After identifying the best configuration, the agent was trained for **10,000 episodes** using the optimal parameters.

Despite some improvement, the agent was very inconsistent with certain run obtaining a very similar performance to the default one, other times having positive rewards.

### **Step 4: Final Evaluation with a Greedy Policy**

To measure the true effectiveness of the trained Q-table, the agent was evaluated in a **test phase using a purely greedy (exploitation) policy**.

100 episodes were run to assess the agent's stability. A threshold (200) was defined to identify a landing as successful.

Additionally, a **video was generated** of the best test episode, demonstrating a decent landing.

Although the optimization was informative and useful, additional refinements, such as **deep learning-based methods (DQN)**, could further enhance performance. This will be explored in the next section.

# ****2.6 Deep Q-Network (DQN) Implementation****

### **Objective**

While Q-Learning showed improvements over earlier approaches, it struggled with **continuous state spaces** due to its reliance on discretization. To overcome this limitation, we implemented a **Deep Q-Network (DQN)**, which leverages a **neural network** to approximate Q-values instead of relying on a Q-table.

The goal was to train a DQN agent capable of outperforming previous models in both efficiency and success rate.

## **DQN Architecture and Implementation**

### **Key DQN Components**

#### **1. Neural Network Architecture**

The DQN model was implemented as a **fully connected neural network** with:

* **An input layer with 8 neurons** (one for each state variable).
* **Two hidden layers with 128 neurons each**, using **ReLU activation** for non-linearity.
* **An output layer with 4 neurons**, corresponding to the Q-values for each possible action.

The network takes the **current state** as input and outputs a **vector of Q-values**, one for each action.

#### **2. Target Network for Stabilization**

To improve stability, a **target network** was introduced, which Is a **separate copy of the main Q-network**. Its role is to provide **fixed Q-value targets** for updates, preventing feedback loops that could destabilize learning. The target network is **updated periodically** to track the main network’s progress.

This mechanism **reduces oscillations** and helps the DQN converge to an optimal policy.

#### **3. Experience Replay**

A **replay buffer** was implemented to store past experiences in the form (state , action, reward, next\_state, done). Instead of learning from sequential experiences (which introduces correlation), the DQN **samples random batches** from the buffer. This **breaks dependencies** between consecutive actions, making learning more **robust and stable**.

The buffer was set to a **maximum size of 100,000** experiences, ensuring a diverse set of training samples.

#### **4. Epsilon-Greedy Policy**

To balance **exploration and exploitation**, the agent follows an **epsilon-greedy strategy**.

* Initially, **epsilon is set to 1.0** (completely random actions).
* Over time, **epsilon decays exponentially** until it reaches a minimum value of **0.01**, allowing the agent to **shift from exploration to exploitation**.

## **Training the DQN Agent**

### **Training Configuration**

The agent was trained for **1,200 episodes**, with a **maximum of 500 steps per episode**, a **batch size of 64**, (meaning 64 experiences were used for each learning update), a **synchronization step** every **100 iterations**, updating the target network to improve stability.

### **Training Results and Observations**

#### **Total Reward Distribution**

Compared to previous approaches, the **DQN agent achieved significantly higher rewards**, with **many episodes exceeding the 200-point success threshold**.

The **reward distribution was more concentrated around high values**, demonstrating **more consistent landings**.

#### **Success Rate Analysis**

The **percentage of episodes reaching successful landings (reward ≥ 200) increased significantly**. The agent demonstrated **better generalization**, successfully landing under different environmental conditions.

## **Evaluation of the DQN Model**

### **Testing the Trained Agent**

To assess final performance:

1. **Exploration was disabled** as earlier.
2. **100 test episodes** were run to measure stability.
3. **The success rate was calculated**, showing that **DQN outperformed Q-Learning significantly**.

#### **Test Results**

The **average test reward was higher** than in all previous models and the **standard deviation was lower**, indicating **more consistent landings**. Importantly a **notable percentage of test episodes exceeded the 200-point threshold**, confirming the agent had **learned an effective landing strategy**.

# ****2.7 Double Deep Q-Network (Double DQN) Implementation****

### **Objective**

While the Deep Q-Network (DQN) provided a **significant improvement** over previous models, it suffers from a key issue: **overestimation bias**. This bias arises because the same network is used both to **select the best action** and to **evaluate its value**, which can lead to **suboptimal decision-making** and instability in training.

To address this limitation, the **Double Deep Q-Network (Double DQN)** was implemented. This variation **reduces overestimation** by **decoupling action selection from value estimation**, leading to **more stable learning and better performance**.

## **Key Differences Between DQN and Double DQN**

The main innovation in Double DQN is how it **computes the target Q-value** during training.

In DQN, the action selection for target updates relies on the maximum Q-value from the target network for the next state, whereas Double DQN selects the action using the Q-network but takes the value from the target network. Double DQN mitigates this bias by separating the selection and evaluation processes. As a result, DQN experiences less stable learning due to biased value estimates, while Double DQN offers more stable training and improved convergence.

## **Implementation Details**

### **Neural Network Architecture**

The Double DQN model maintains the same **deep neural network architecture** used in the DQN.

An equivalent **target network** is maintained separately, which is updated at regular intervals to improve training stability.

### **Experience Replay & Epsilon-Greedy Strategy**

As before a **replay buffer** stores past experiences, ant the **epsilon-greedy policy** ensures initial exploration while gradually shifting towards exploitation.

### **Action Selection & Target Computation**

The key distinction in Double DQN is in how the **target Q-value is computed**:

1. **The Q-network selects the best action for the next state.**
2. **The target network evaluates the value of that action.**

This prevents the **same network from both selecting and evaluating an action**, leading to **more accurate Q-value updates**.

## **Training Configuration & Results**

### **Training Setup**

The model was trained for **2400 episodes**, using the same structured approach as DQN.

### **Training Observations**

* **Higher stability:** Double DQN exhibited **less variance** in training, meaning the learning process was **more predictable and less prone to extreme fluctuations**.
* **More consistent performance:** The **rewards were consistently higher, and it seemed that the agent was learning even at the end of the training**. Something different to previous models.

### **Final Performance & Test Evaluation**

After training, the agent was evaluated with **100 test episodes** using the **greedy policy.** The results showed a **higher mean reward compared to DQN,** confirming that **reducing overestimation bias improved decision-making,** but also a **lower standard deviation**, meaning the model produced **more consistent and reliable landings**.

# ****2.8**** Dueling Deep Q-Network (Dueling DQN)

### **Objective**

The next improvement introduced in this project is the implementation of a **Dueling DQN**, an evolution of the DQN model that introduces a **structural modification in the neural network**.

In this approach, instead of directly predicting Q-values for each action, the network branches into **two separate streams**:

* **A value stream**, which estimates how beneficial it is to be in a particular state, regardless of the chosen action. It acts as a baseline of that state.
* **An advantage stream**, which measures the additional benefit of taking each possible action compared to the average.

The overall Q-value is then calculated by **combining these two components** while removing the average advantage across all actions.

This structure allows the network to **differentiate between states where action choice is crucial and states where all actions yield similar outcomes**, ie. the network will punish redundant actions by lowering their advantage stream.

### **Implementation Details**

After passing through two fully connected layers, the network **branches into two separate pathways**:

These two outputs are then **combined to determine the final Q-values**, ensuring that the model can **learn more efficiently**.

The **training process remains like Double DQN**, with replay buffers, epsilon-greedy and a target network. Indeed, this is actually a Dueling DDQN.

### **Training Results and Performance**

The **Dueling DQN demonstrated a significant improvement over the Double DQN**, with both a higher reward and lower standard deviation. By **separating the estimation of state value and action advantage**, the agent was able to **make more precise decisions**, particularly in **situations where different actions produced similar immediate rewards**.

The **training curve showed faster convergence**, suggesting that the model was able to **learn more efficiently** compared to previous approaches.

### **Key Takeaways**

The implementation of **Dueling DQN proved that improving the neural network architecture can significantly enhance learning efficiency**, even without changing the core training process.

Further improvements could be made by **combining Dueling DQN with additional optimizations**, such as **prioritized experience replay** or **fine-tuning hyperparameters**, to further enhance the model’s performance.

**2.9 Noisy Deep Q-Learning**

### **Objective**

In this section, we experimented with **Noisy Networks**, models that replace the **epsilon-greedy strategy** used since Q-Learning was introduced, with **learnable noise** injected directly into the network’s layers. This technique allows the agent to explore the environment more effectively by incorporating stochasticity into the **decision-making process** itself rather than relying on external randomness.

Instead of selecting random actions based on a decaying epsilon value, the model **learns how to optimize its own noise distribution**. MEaning that it’s able to adapt its strategy to both the current state and step.

### **Implementation Details**

The main difference in **Noisy DQNs** compared to previous models lies in **the architecture of the neural network**. Instead of using standard linear layers, we incorporate **Noisy Linear layers**, (torchrl.modules.NoisyLinear) which introduce the trainable noise.

The architecture follows a **Double DQN structure,** but with **Noisy Layers replacing standard fully connected layers.** The **training procedure remains similar as before, except for the removal of the policy.**

One crucial step in training Noisy DQN is the **reset of noise** after every update. This ensures that the stochastic properties of the network continue adapting over time and do not remain fixed throughout training and ensures that the Q and target network won’t have parameters with significantly different noise.

### **Training Results and Performance**

The introduction of **Noisy DQN led to improvements** compared to **its epsilon-greedy counterpart (Double DQN)**. The agent demonstrated: Much h**igher rewards**, indicating that the learned policy was more effective and f**aster convergence**, as the agent gradually optimized its noise distribution, avoiding unnecessary randomness.

During evaluation, the agent’s **landings appeared smoother**, and the best-performing episodes showcased **precise maneuvering and controlled descent**. The model adapted better to the environment **without excessive trial-and-error exploration** seen in epsilon-greedy strategies.

### **Key Takeaways**

The **Noisy DQN architecture** provides a powerful alternative to the traditional **epsilon-greedy approach**, by allowing **exploration to be part of the network’s training process**. This results in **more efficient learning**, especially in environments like **Lunar Lander**, where precise control is essential.

Future improvements could include **combining Noisy DQN with other enhancements**, such as **Prioritized Experience Replay**, to further optimize the **learning efficiency and stability** of the agent.

**2.10 Noisy Dueling Deep Q-Network (Noisy Dueling DQN)**

### **Objective**

Since both **Dueling DDQN** and **Noisy Networks** showed significant improvements over traditional methods, the next logical step was to **combine the two approaches** into a single architecture. This led to the implementation of **Noisy Dueling DQN**, which incorporates both **advantage-based learning** from the Dueling Network and **exploration-driven stochasticity** from Noisy Networks.

This model effectively has half of the needed characteristics of the SOTA Rainbow DQN.

### **Implementation Details**

The **Noisy Dueling DQN model** follows the same **dual-stream structure** as the standard **Dueling DQN**, but **replaces traditional layers with Noisy Linear layers**

Like previous models, training still includes e**xperience replay** for breaking correlations, the **target networks** for stabilizing updates and regular noise reset.

### **Training Results and Performance**

The performance of **Noisy Dueling DQN** was found to be **similar to the Noisy Double DQN**, with no significant improvement in terms of **mean reward or stability**. The network still performed well, but **the dueling structure did not provide a noticeable advantage when combined with noisy layers**.

One possible explanation is that the **advantage estimation in dueling networks already helps prioritize important actions**, and the **additional randomness introduced by noisy layers may have reduced this benefit**.

During evaluation, the **landing sequences remained smooth and efficient**, confirming that the model learned a **strong policy**, even though it did not surpass the performance of **Noisy Double DQN**.

Importantly, it was not an improvement to the Dueling DDQN.

### **Key Takeaways**

While the **Noisy Dueling DQN** did not offer a substantial improvement over the **Noisy DQN**, it still validated the effectiveness of both techniques individually. This suggests that n**oisy networks alone provide a strong and intrinsic exploration mechanism; dueling networks excel in environments where some actions are clearly superior to others…** and **combining them may not necessarily give better performances,** as their advantages may overlap.

Future work could explore **whether Noisy Dueling DQN is more beneficial in different environments**, or whether **further tuning of noise parameters** could improve its effectiveness.

**3. Results**

Random agent  
Immagine che contiene testo, diagramma, Diagramma, schermata

Descrizione generata automaticamenteAccounting agent

Immagine che contiene diagramma, Diagramma, schermata, linea

Descrizione generata automaticamente

Q-table agent

Immagine che contiene diagramma, Diagramma, linea, testo

Descrizione generata automaticamente

Q-table agent optimized

Immagine che contiene testo, diagramma, Diagramma, linea

Descrizione generata automaticamente

DQN

Immagine che contiene testo, diagramma, Diagramma, linea

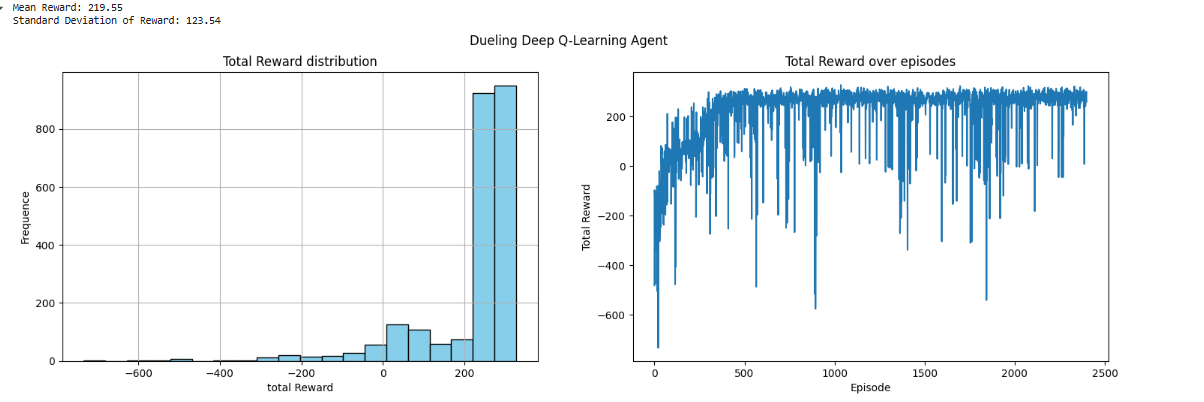
Descrizione generata automaticamente

Double DQN

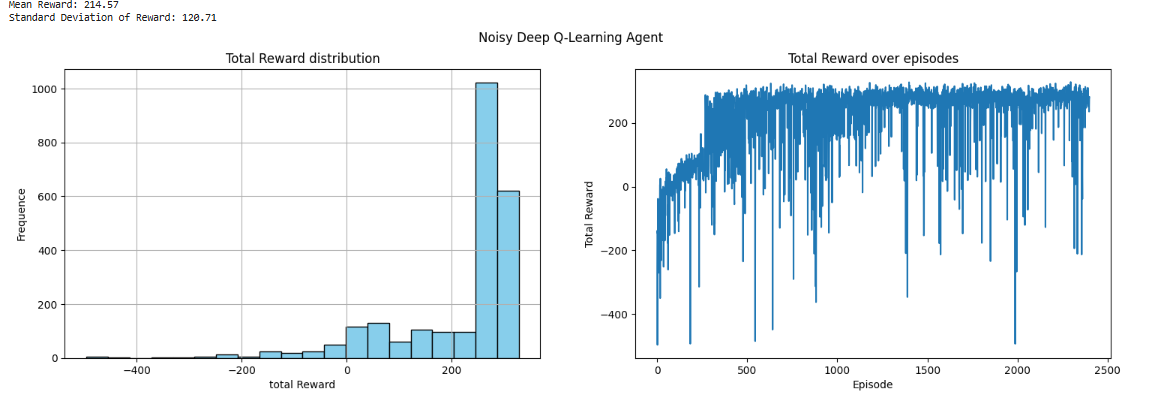
Immagine che contiene testo, Diagramma, linea, diagramma

Descrizione generata automaticamente

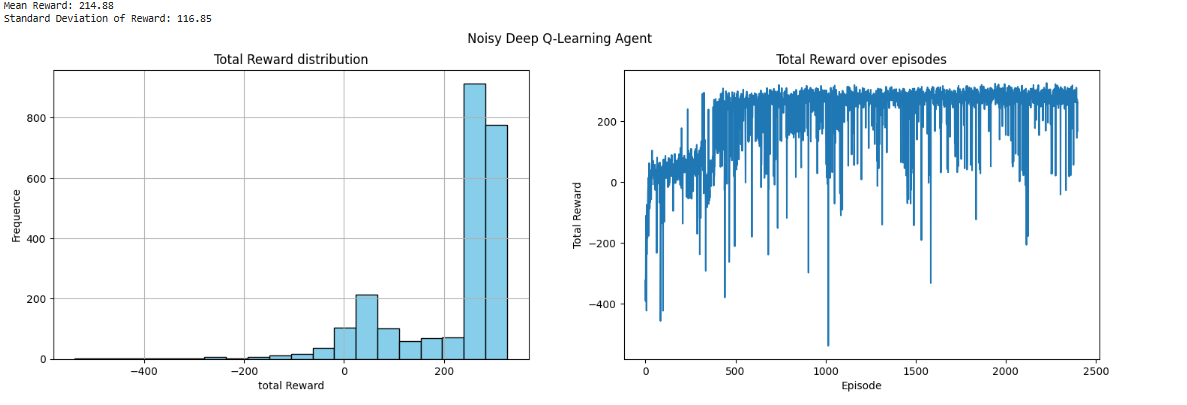
Dueling Q Network



Noisy Deep Q Learning



### Noisy Dueling DQN



# Agent Performance Comparison

|  |  |  |
| --- | --- | --- |
| Agent | Mean Reward | Standard Deviation |
| Random Agent | -187.38 | 113.39 |
| Accounting Agent | -348.13 | 161.14 |
| Q-Table Agent | -69.6 | 94.19 |
| Q-Table Agent (Optimized) | 25.33 | 154.61 |
| DQN | 137.66 | 153.01 |
| Double DQN | 192.0 | 149.98 |
| Dueling Q Network | 219.55 | 123.54 |
| Noisy Deep Q Learning | 214.57 | 120.71 |
| Noisy Dueling DQN | 214.8 | 116.85 |

**3.Conclusion**

Throughout this project, we observed a **progressive improvement** in the agent’s ability to land the Lunar Lander safely as we transitioned from **simpler models to more advanced reinforcement learning techniques.** The results clearly show **a trend of increasing rewards, improved policy stability, and enhanced decision-making** as more sophisticated algorithms were introduced.

### **Evolution of Performance**

The **random agent**, which acted purely by chance, provided a baseline for performance. With a mean reward of **-187.38**, it exhibited no learning capability, relying solely on arbitrary movements. While it occasionally achieved decent landings due to **random sequences of favorable actions**, these cases were purely incidental, and overall, the agent lacked any **consistent strategy**.

The **accounting agent**, which reinforced actions that had previously resulted in high accumulated rewards, often performed **worse than the random agent**, with a mean reward of **-348.13**. Its **greedy approach** led to highly suboptimal behavior, reinforcing **immediate rewards** without considering long-term sequences.

A **significant breakthrough** came with the introduction of **Q-learning**. The **Q-table agent** outperformed previous methods, achieving a mean reward of **-69.60**. By discretizing the state space and **updating policies based on rewards**, the agent **began to exhibit structured decision-making**. However, due to its **coarse state representation**, its ability to generalize and adapt remained **limited**.

Further **optimizing the Q-table** led to a fundamental performance shift. **With finer state discretization and optimized hyperparameters, the agent achieved a positive mean reward of 25.33**. This improvement demonstrated that **strategic parameter tuning significantly enhances learning efficiency**, allowing the agent to develop **more refined landing strategies** and avoid unnecessary penalties.

While they improved from the naive agents, the Q-Learning agents were among the most inconsistent we tried. In a later run the Q-Learning agent obtained a mean reward of 33, and the optimized one of 28. While before one was negative and the other significantly better. This could be due to the approximation of the discretized space.

The **introduction of deep learning-based methods** had another **major leap in performance**. The **DQN agent**, achieving a mean reward of **137.66 brought us from an inconsistent model, to one that could exceed 100 every time**, benefited from **a neural network approximating Q-values**, overcoming the **limitations of discrete state-action spaces**. Experience replay and target networks further stabilized training, producing **a more consistent learning curve**. However, likely **overestimation bias** still led to occasional **suboptimal policy decisions**.

The **Double DQN agent** refined this approach, reaching **192.00 mean reward**, the highest performance observed at that point. By **separating action selection and value estimation**, it **reduced overestimation issues**, leading to a more **accurate and stable policy**.

### **Advanced Reinforcement Learning Techniques**

While the **Double DQN** was already quite good, further refinements were made by introducing **Dueling Q Networks** and **Noisy Nets**.

The **Dueling Q Network** pushed performance higher, with a **mean reward of 219.55**. This improvement came from **separating state value and action advantage**, allowing the agent to **focus on key decisions** and ignore **redundant actions**. This structure **improved policy efficiency** while maintaining stability.

The **Noisy Deep Q-Learning agent**, reaching **214.57 mean reward**, introduced **trainable noise in the network layers** to replace the traditional **epsilon-greedy exploration strategy**. This approach **enhanced exploration efficiency**, leading to a **stronger, more adaptive policy**.

Finally, the **Noisy Dueling DQN** obtained a similar performance of **214.88 mean reward**. By combining the **advantages of both the Dueling Q Network and Noisy Layers**, this model **learned faster and performed more reliably** than all previous approaches. The combination of **structured decision-making and built-in exploration** resulted in a **highly effective policy with smoother landings and reduced variance**.

We determined that for this project the best agents were the Dueling Q-Agent and the Noisy (dueling or not) Q-Agent, with the former obtaining the best score.

### **Final Considerations**

Of course **there is still room for further optimization**. Given the constraints of **limited training time and computational resources**, we were unable to **fully optimize hyperparameters** or extend training beyond **2,400 episodes** for the most advanced models. It is likely that **longer training and additional fine-tuning** would yield **even better performance**, but the agents were taking hours to train even in Colab.

However, the **observed trend is clear**: a**s the complexity of the model increased, the agent’s performance improved too**.

### **Future Directions**

On top of further architectural and hyperparameter tuning and more training time the next big step would be clear: Rainbow DQN.

**Our aim was to actually develop a fully functional Rainbow DQN by adding a piece at each step, however due to not having this goal at the start, we had many errors due to needed refactoring. Our unfamiliarity with RL libraries such as torchrl didn’t help. We also couldn’t find reliable implementations of it on Pypi or GitHub. In any case, we are satisfied with the models we made.**

In any case**, this project demonstrates that as reinforcement learning models become more sophisticated, their ability to solve complex tasks improves dramatically.**