Computer Lab 2

Benet Ramió i Comas 20031026T177
Laura Solà Garcia 20031119T225
Reinforcement Learning

December 19, 2024

1 Deep Q-Networks (DQN)

1.1 Replay buffer and target network in DQN

When implementing Q-learning algorithm using function approximation using neural networks the parameters do not converge to the optimal ones. This is because of two problems that led to the use of a replay buffer and a target network:

1. **Correlated updates due to sequential experiences:**
   During training, the parameters of the network are updated with stochastic gradient descent (SGD) based on observed experiences. However, successive updates are highly correlated because the data is generated from consecutive time steps along the same trajectory. This correlation impedes convergence.

   The solution is to use an experience replay buffer, a memory buffer of size L that stores past experiences (s, a, r, s'). At each time step, a mini-batch of size N is sampled uniformly at random from the buffer, and the network parameters θ are updated. This approach breaks the temporal correlation between consecutive experiences, stabilizing the learning process and improving convergence.

2. **Non-stationary targets in TD updates:**
   In the TD-error $(r_t + \lambda \max_b Q_\theta(s_{t+1}, b) - Q_\theta(s_t, a_t))$ used for updating θ, the target term $r_t + \lambda \max_b Q_\theta(s_{t+1}, b)$ is non-stationary because θ is updated at every time step. This rapid change in the target makes it difficult for the network to track the optimal values.

   The solution is to introduce a target network, which has parameters ϕ. The target network is fixed for C time steps and is updated periodically to match the main network: $\phi \leftarrow \theta$. The update of parameters θ then becomes:

$$\theta \leftarrow \theta + \alpha \left(r_t + \lambda \max_b Q_\phi(s_{t+1}, b) - Q_\theta(s_t, a_t) \right) \nabla_\theta Q_\theta(s_t, a_t).$$

   By decoupling the target from the rapidly-changing θ, the learning process becomes more stable.

1.2 Network layout and parameters

To implement the Deep Q-Network (DQN) algorithm, we designed a fully connected neural network to approximate the Q-function. This network consists of an input layer with 8 neurons (corresponding to the size of each state), a single hidden layer with 64 neurons, and an output layer with 4 neurons (representing the size of the action space). Additionally, a target network was employed, identical in architecture but with weights updated every C steps.

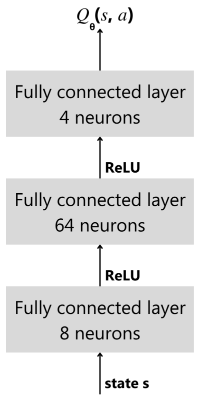

$Q_\theta(s, a)$
Fully connected layer 4 neurons
ReLU
Fully connected layer 64 neurons
ReLU
Fully connected layer 8 neurons
state s

Figure 1: Layout of the Network

The network's architecture was determined through hyperparameter optimization. Simpler networks, such as those with fewer neurons or no hidden layers, often underfitted the data, leading to low rewards. Conversely, more complex architectures, including additional hidden layers or larger neuron counts, frequently overfitted the data, as indicated by a rapid drop in episodic rewards. The choice of 64 neurons in the hidden layer struck a balance between underfitting and overfitting. Although smaller hidden sizes occasionally performed well with specific combinations of hyperparameters, the 64-neuron configuration consistently outperformed alternatives across a variety of settings (e.g., different memory sizes, batch sizes).

We also experimented with a Dueling-DQN architecture but observed a decline in performance. Our hypothesis is that the additional complexity of the dueling architecture was unnecessary for this relatively simple environment, where a single hidden-layer network proved sufficient.

The Adam optimizer was chosen due to its adaptive learning rate and momentum, which stabilize the learning process. A learning rate of $\alpha = 0.0001$ provided the best results, balancing convergence speed with policy accuracy. Smaller learning rates failed to produce effective policies, while larger ones caused unstable training. Gradient clipping was applied, with a maximum norm of 1, to prevent exploding gradients and further enhance training stability.

We used an experience replay buffer with a capacity of 10,000. To improve convergence during early episodes, 20% of the buffer was pre-filled with random experiences. This approach allowed the model to start training with some informative data. However, excessively pre-filling the buffer slowed convergence later, as samples generated by the epsilon-greedy policy (based on estimated Q-values) tended to be more valuable. A batch size of 128 was selected through hyperparameter tuning. Additionally, we implemented Combined Experience Replay (CER). When sampling a batch from the replay buffer, the latest transition was always included. This modification prioritizes recent experiences, which is particularly beneficial in environments with large replay buffers.

The model was trained for 800 episodes. The final saved model was the one achieving the highest average reward, as the stochastic nature of training meant that different runs required varying numbers of episodes to converge to optimal solutions.

For exploration, an exponentially decaying epsilon-greedy strategy was used. The maximum epsilon ($\epsilon_{\max}$) was set to 0.99, the minimum epsilon ($\epsilon_{\min}$) to 0.05, and the decay was parameterized over $Z = 0.9\times$ the total number of episodes. This strategy balanced exploration and exploitation, allowing for sufficient exploration in early episodes while focusing on exploitation in later stages.

Table 1 summarizes the final hyperparameters used. With this layout the found policy achieves an average total reward of 237.4 +/- 17.1 with confidence 95%.

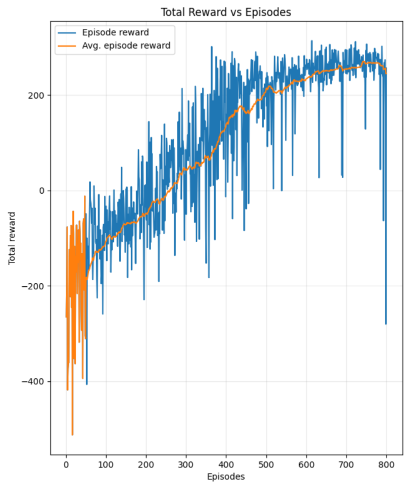
Hyperparameter	**Value**
Learning Rate (α)	0.0001
Memory Size (L)	10000
Batch Size (B)	128
Num Hidden Layers	1
Size Hidden Layer	64
Type of decay for ϵ	Exponential
Number of Episodes	800
CER	True
Dueling	False

Table 1: Selected hyperparameters for DQN

1.3 Total episodic reward and number of steps during training

In Figure 2, we observe the evolution of the total episodic reward and the number of steps per episode throughout the training process.

Regarding the total reward, we observe that while the plot exhibits significant variability, the moving average (window size = 50 episodes) steadily increases until reaching its peak. This indicates that the learning rate is appropriately chosen, and the network capacity is adequate, as there are no signs of overfitting or underfitting. The increasing reward makes sense in this context, as the agent learns to land successfully in the target zone with fewer mistakes and more efficient maneuvers.



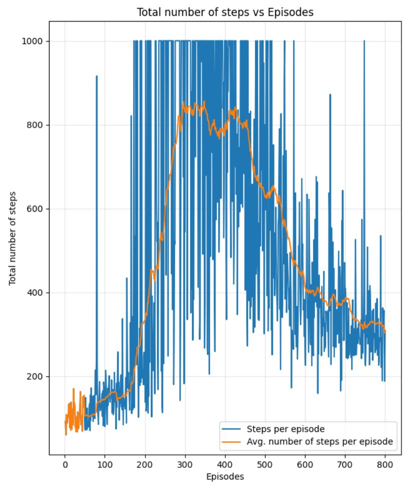


Figure 2: Total episodic reward and number of steps per episode during training.

As for the number of steps per episode, the plot shows even greater variability than the reward plot. Initially, the number of steps is low because the agent has no knowledge of the environment, causing episodes to end quickly as the agent crashes. As training progresses and the agent begins to learn, the number of steps increases significantly. This can be attributed to the agent avoiding crashes and spending more time flying around the environment, attempting to locate and land in the correct position. Finally, as the agent gains a better understanding of the environment and learns an optimal landing strategy, the number of steps steadily decreases. This behavior reflects the agent's improved ability to land efficiently with minimal corrections, achieving its goal in fewer steps.

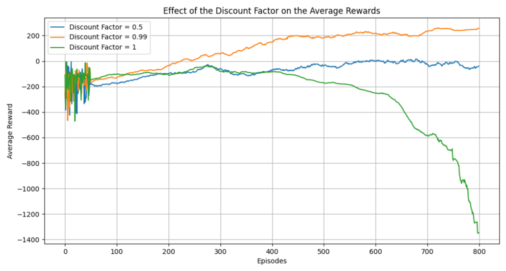
1.4 Effect of discount factor

NOTE for the following sections of Task 1: The plotted values are moving averages of rewards or steps across 50 episodes. However, confidence intervals have been omitted due to excessive clutter.

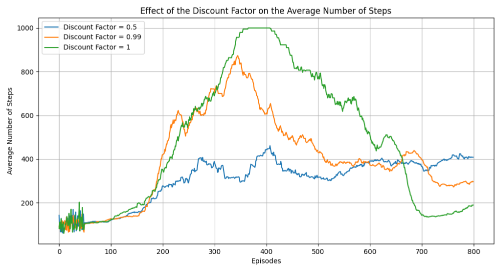
In order to analyze the impact of the discount factor γ on the learning process, we trained the network with $\gamma_1 = 1$ and $\gamma_2 = 0.5$, and compared the results with the best-performing discount factor $\gamma_0 = 0.99$.

From Figure 3a, we observe that models with $\gamma = 0.5$ and $\gamma = 1$ perform worse, as their average rewards are significantly lower. To understand this, note that the discount factor determines how much importance is given to future rewards: when $\gamma = 1$, all rewards, regardless of when they occur, are equally weighted; as γ decreases towards zero, immediate rewards are prioritized, and future rewards are heavily discounted.

For $\gamma = 0.5$, the training is stable, but the solution is suboptimal in terms of reward. This



(a) Average episodic reward



(b) Average number of steps

Figure 3: Performance during training for different discount factors

happens because the agent focuses too much on short-term gains, prioritizing actions that avoid immediate penalties, such as crashing or leaving the screen. However, the agent fails to plan further into the future to correct its position or orientation and ultimately land successfully. As shown in Figure 3b, the number of steps generally remains below the curves of the other models and after 200 episodes more or less it remains stable, suggesting the agent learns to avoid crashes but does not optimize its trajectory to reach the target landing zone.

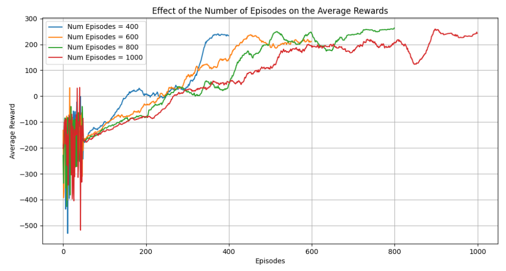
For $\gamma = 1$, the agent prioritizes long-term rewards, but this overemphasis causes instability during training. The agent equally values immediate and distant rewards, which can lead to overly optimistic behavior. For instance, it may delay corrective actions, such as stabilizing its orientation or controlling descent speed, because it expects large rewards in the future. This instability is evident in the sharp drop in average rewards after 400 episodes in Figure 3a. Similarly, Figure 3b shows that the number of steps remains higher for most of the training process, as the agent takes inefficient or delayed actions during the training process.

In contrast, the optimal discount factor $\gamma = 0.99$ achieves the best balance between immediate and future rewards. The agent learns to take strategic actions, such as firing the engines to correct its orientation and stabilize the lander, despite small immediate penalties like fuel costs. At the same time, it plans effectively for the future, recognizing the large reward associated with a successful landing. As shown in Figure 3b, while the number of steps fluctuates initially, the agent eventually learns to solve the problem efficiently in fewer steps.

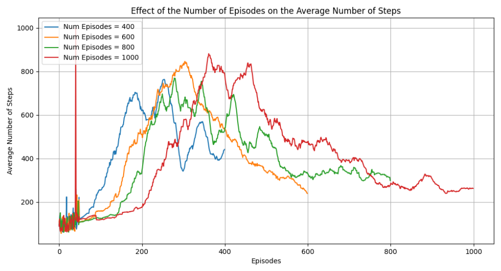
1.5 Effect of the number of episodes

Another hyperparameter we optimized is the number of episodes used to train the model.

The number of episodes affects the convergence behavior of the algorithm. Fewer episodes lead to faster convergence because epsilon decreases more rapidly. However, if the number of episodes is too small, the agent might not have enough exploration time, which can result in suboptimal solutions when the agent shifts to exploitation.



(a) Average rewards over episodes



(b) Average steps over episodes

Figure 4: Effect of the number of episodes on training performance

As shown in Figure 4a, while the algorithm converges faster with 400 or 600 episodes, the reward curves exhibit higher variability, indicating less stability. In contrast, longer training durations (e.g., 1000 episodes) yield smoother convergence but incur a higher computational cost.

As for the number of steps per episode, in Figure 4b we observe how there are no major differences between the four of them, just that the higher the number of episodes the longer it takes to decrease the number of steps (for the same reason as with the reward).

Based on these observations, we chose 800 episodes as it offers a balance between stability and computational efficiency. The agent has sufficient time to explore the environment and reliably converge to a good solution, but in less time than training for 10000 episodes.

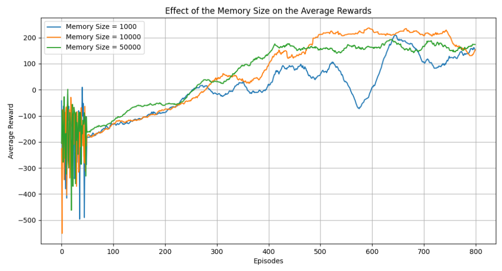
1.6 Effect of memory size

The memory size is a hyperparameter that defines how many experiences can be stored in the replay buffer. In our experiments, we initially selected a value of 10000, but in this section, we investigate how increasing or decreasing this value affects performance.

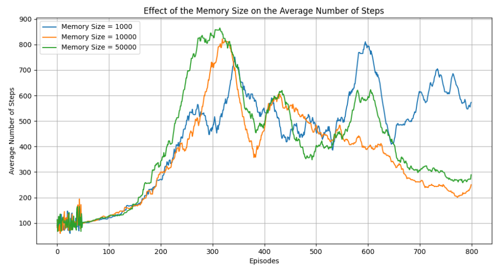
We first tested memory sizes of 5000, 10000, and 20000 experiences. However, the differences in training performance among these values were negligible. To further explore the effect of memory size, we extended our range beyond the recommended values $L \in [3000, 30000]$. Specifically, we experimented with memory sizes of 1000, 10000, and 50000.

For the smallest memory size (1000), the training process is less stable, as shown in Figure 5a. When the memory size is too small, the agent relies heavily on recent experiences, which can cause overfitting to specific situations or convergence to local optima.

In contrast, with larger memory sizes, the oscillations in episodic rewards are reduced. This can be attributed to the larger buffer allowing the agent to sample from a more diverse set of past experiences, which helps stabilize the learning process.



(a) Average Rewards



(b) Average Steps

Figure 5: Effect of memory size on rewards and steps.

However, when the memory size is excessively large (50000), the model converges to a slightly worse solution. This behavior arises because experiences stored in the replay buffer may become too outdated, providing little new information while increasing computational overhead.

Regarding the number of steps, as illustrated in Figure 5b, the agent with the smallest memory size takes more steps per episode, likely due to a less effective policy. On the other hand, the models with 10000 and 50000 experiences converge to a similar number of steps per episode.

1.7 Plot of value function

We analyze the behavior of the value function $V_\theta(s)$ for the restricted state $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, where y is the height of the lander and ω represents its angle. The plot of $\max_a Q_\theta(s(y, \omega), a)$ as a function of $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$ is shown in Figure 6.

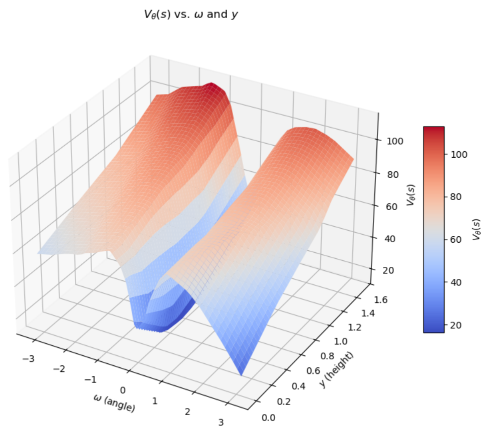


Figure 6: $V_\theta(s)$ as a function of height (y) and angle (ω).

Firstly, we notice how the value function is symmetric around $w = 0$. This makes sense as the fact that the rocket is tilted to the right or to the left should not impact the value function.

As the angle ω deviates from zero, the value function increases until it reaches a peak, then decreases. This result is unexpected, as we initially hypothesized that the maximum value function would occur at $\omega = 0$, gradually decreasing as the angle increased (since greater angles make landing more difficult). One possible explanation is that an angle very close to zero might not be ideal. When firing the engine to correct such an angle, the engine's power could overcompensate, causing the rocket to overshoot and tilt to the opposite side instead of aligning properly. Consequently, a not-so-small angle might be optimal, as it allows easier correction given the engine's power characteristics. As expected, when the angle grows further, the value function decreases due to the increasing penalties incurred from the greater effort required to realign the rocket.

When examining the effect of height y, we notice that the value function increases with height. This behavior makes sense because greater altitude allows the lander more room to maneuver and correct its trajectory, reducing the likelihood of failure. However, we did not expect to get this as we thought that the closer to the ground the rocket is, the less maneuvers and therefore less penalty he would receive for landing.

1.8 Plot of the optimal policy

In Figure 7, we observe the decisions of the policy learned by the DQN algorithm for the restricted state space $(0, y, 0, 0, \omega, 0, 0, 0)$. These states correspond to the rocket being directly above the landing pad at height y, with no horizontal, vertical or angular velocity, an angle ω, and without touching the ground. The policy demonstrates a logical approach to achieving a safe landing.

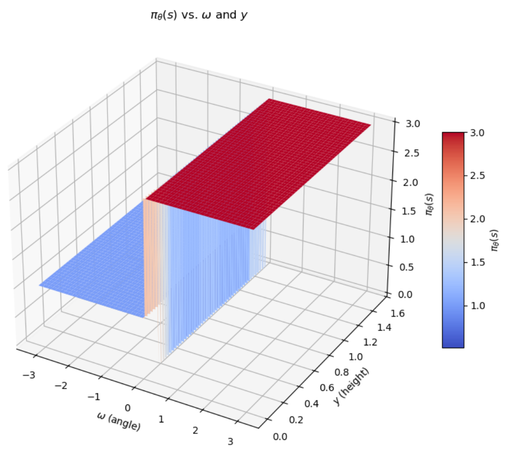


Figure 7: Optimal policy $\pi_\theta(s)$ as a function of height (y) and angle (ω).

First, the policy prioritizes correcting the angle of the rocket. When $\omega > 0$, the optimal action is to fire the left orientation engine (a = 3), while for $\omega < 0$, the optimal action is to fire the right orientation engine (a = 1). This behavior ensures that the rocket aligns itself vertically, as proper alignment is crucial for a safe landing.

Once the rocket is aligned ($\omega = 0$) and the height is close to zero, the optimal action becomes firing the main engine to counteract gravity, slow the descent, and prevent a crash. This makes sense because maintaining a controlled descent is essential to avoid collisions with the surface. However, when the height (y) is larger, then the agent opts to do nothing (a = 0) in order to descent without penalties and slow down the landing when it is closer to the ground.

1.9 Comparison of learned policy and random agent

The total episodic reward over 50 episodes was compared between the trained actor agent and a random agent. Figure 8 shows the rewards and number of steps for each episode, while Table 2 summarizes the averages.

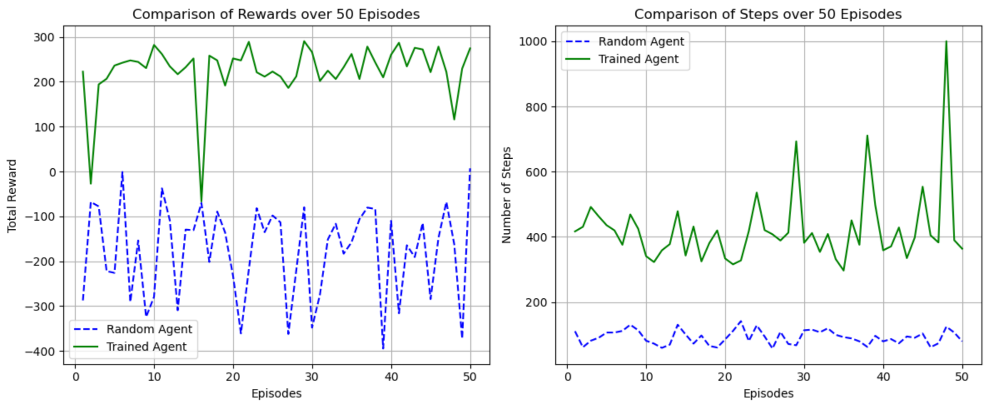


Figure 8: Comparison of total episodic reward and steps for the trained agent and random agent over 50 episodes.

Agent	**Average Reward**	**Average Number of Steps**
DQN Agent	225.1	407.42
Random Agent	-177.25	95.28

Table 2: Comparison of learned policy and random agent.

The trained DQN agent achieves significantly higher rewards compared to the random agent, as the learned policy enables safe and accurate landings, whereas choosing random actions rarely results in successful landings. Regarding the number of steps, the random agent takes fewer steps because the rocket often crashes quickly when actions are selected randomly. In contrast, the trained agent takes more steps, as it carefully adjusts the rocket's angle, position, and velocity to execute a successful landing.

2 Deterministic Policy Gradient (DDPG)

2.1 Some questions

2.1.1 Critic Target Network for Actor Update

In DDPG, the actor network is updated using the deterministic policy gradient theorem, which depends on the critic's value estimates. The critic's target network is used only to compute the target values for training the critic network, ensuring stability in its updates. Using the critic's target network when updating the actor network would not reflect the latest learned value function, potentially slowing convergence. The main critic network is preferred to guide the actor because it provides more up-to-date feedback on the policy's performance.

2.1.2 Off-Policy Nature and Sample Complexity

DDPG is an off-policy algorithm. It uses a replay buffer to store experiences and samples them randomly during training, allowing the policy to improve based on past experiences rather than requiring new ones for each update. Off-policy methods generally have better sample efficiency compared to on-policy methods, as they can reuse data multiple times. However, off-policy methods can be more complex to stabilize, as they rely on ensuring the distribution of sampled experiences aligns well with the current policy.

2.2 Network layout and parameters

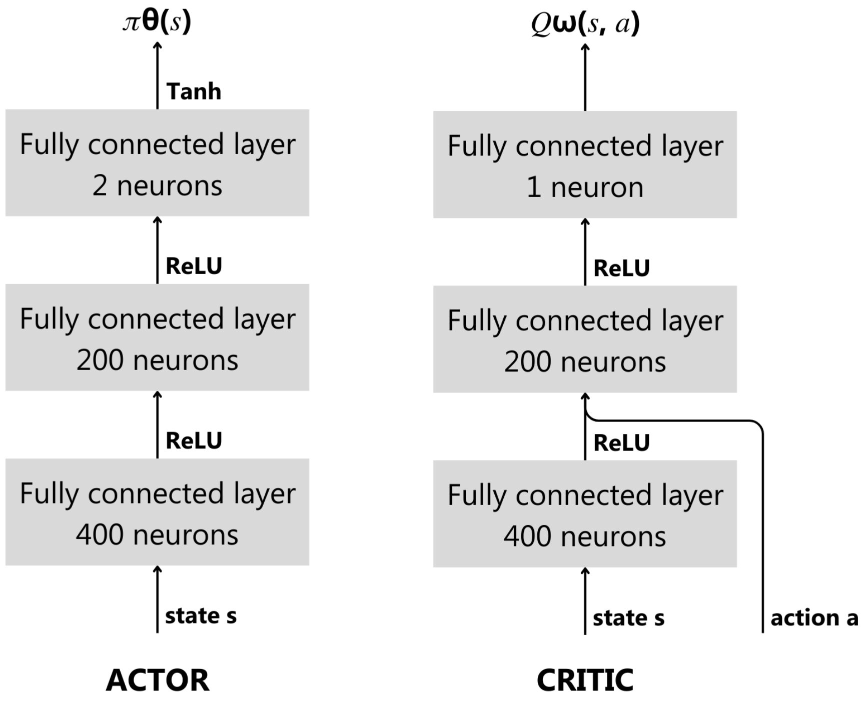

$\pi\theta(s)$
Tanh
Fully connected layer 2 neurons
ReLU
Fully connected layer 200 neurons
ReLU
Fully connected layer 400 neurons
state s
ACTOR
$Q\omega(s, a)$
Fully connected layer 1 neuron
ReLU
Fully connected layer 200 neurons
ReLU
Fully connected layer 400 neurons
state s
action a
CRITIC

Figure 9: Actor and critic network architectures

The architectures of the actor and critic networks are illustrated in Figure 9. The actor network used in this implementation is a fully connected neural network with an input layer matching the state space dimensionality, followed by a hidden and an output layer. The input layer consists of 400 neurons, and the hidden layer consists of 200 neurons, both using the ReLU activation function. The output layer has two neurons, corresponding to the action space dimensionality, with a `tanh` activation function to constrain the actions to the range $[-1, 1]$. Similarly, the critic network is a fully connected neural network, but with a slightly different structure. It takes the state as input, processes it through a layer of 400 neurons with ReLU activation, and concatenates the result with the action input. This concatenated vector is passed through a hidden layer with 200 neurons and ReLU activation, followed by an output layer that produces a scalar Q-value without any activation function.

For optimization, the Adam optimizer was chosen due to its adaptive learning rate properties, which are effective in handling noisy gradient estimates commonly encountered in reinforcement learning tasks. The learning rate for the actor network was set to $5 \cdot 10^{-5}$, while a higher learning rate of $5 \cdot 10^{-4}$ was used for the critic network, allowing faster convergence of the critic, which provides feedback to the actor.

The choice of parameters reflects a balance between performance and stability. A discount factor of $\gamma = 0.99$ was used to favor long-term rewards, while a replay buffer size of 30,000 allowed the agent to retain a wide range of experiences for training, improving sample efficiency. The training ran for 300 episodes, which was sufficient for the policy to converge, and the network with best average reward in the last 50 epochs was the one saved to avoid overfitting. The target networks were updated using a soft update mechanism with a constant $\tau = 10^{-3}$, ensuring smooth parameter updates and reducing instability. A batch size of 64 was selected to strike a balance between computational efficiency and accurate gradient estimation. Additionally, the actor network was updated less frequently, every two steps ($d = 2$), to reduce high-variance updates and stabilize training.

Exploration noise was introduced using an Ornstein-Uhlenbeck process with parameters $\mu = 0.15$ and $\sigma = 0.2$, providing smooth and temporally correlated action noise, which is well-suited for continuous control tasks.

These design choices align with recommendations in the lab instructions, ensuring a stable and efficient learning process while solving the LunarLanderContinuous problem. This design found a policy which achieves average total reward of $210.8 + / - 30.7$ with confidence 95% in the *DDPG_check_solution*.

2.3 Total episodic reward and number of steps

Figure 10 shows two plots: the total episodic reward and the total number of steps taken per episode as a function of the number of episodes during training.

The left plot, **Total Reward vs Episodes**, shows the progression of the agent's performance throughout training. In the initial phase, covering roughly the first 100 episodes, the episodic rewards are highly negative, indicating poor performance and frequent task failures. Between episodes 100 and 170, the rewards begin to increase steadily, reflecting the agent's improving ability to learn a successful policy. The moving average of the episodic rewards clearly shows this upward trend. Toward the end of training, the average reward stabilizes around 200, demonstrating that the agent has successfully learned an effective policy for solving the task. Despite this stabilization, occasional fluctuations in the rewards suggest the presence of exploration noise or training instability.

The right plot, **Total Steps vs Episodes**, shows the number of steps taken per episode over the course of training. Initially, the step count varies significantly due to the agent's trial-and-error exploration. As training progresses, the number of steps increases, peaking around episode 120. This peak coincides with the period when the rewards start to improve, indicating the end of the agent's exploration phase. After this peak, the average number of steps gradually decreases, suggesting that the agent becomes more efficient in completing the task with fewer steps.

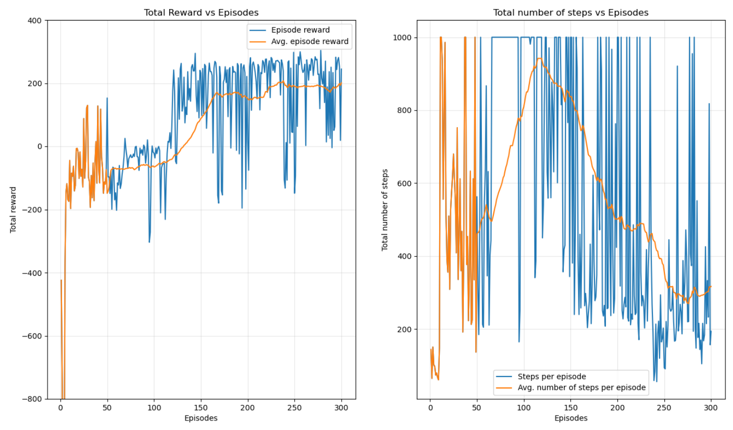


Figure 10: Total episodic reward (left) and total number of steps per episode (right) during training.

2.4 Effect of discount factor

Figure 11 shows two plots comparing the total episodic reward and the total number of steps taken per episode during training for three different discount factors: $\gamma_0 = 0.99$ (chosen optimal discount factor), $\gamma_1 = 1$ (full future reward consideration), and $\gamma_2 = 0.5$ (significantly lower than γ_0).

In the left plot, **Total Reward vs Episodes**, the performance of the agent varies significantly with the choice of the discount factor:

- For $\gamma_0 = 0.99$ (blue line), the agent achieves high rewards after an initial exploration phase, stabilizing around 200, indicating successful learning of a policy that solves the task.

- For $\gamma_1 = 1$ (orange line), the agent struggles to achieve consistent positive rewards. Although there is an initial improvement, the rewards remain negative and decline over time, suggesting that considering future rewards indefinitely makes the learning process unstable.
- For $\gamma_2 = 0.5$ (green line), the rewards remain constant negative throughout training, indicating poor performance. The agent's inability to consider long-term future rewards hinders its ability to learn an optimal policy.

In the right plot, **Total Number of Steps vs Episodes**, the step count evolution further reflects the impact of the discount factors:

- For $\gamma_0 = 0.99$ (blue line), the number of steps decreases after an initial peak, showing that the agent learns to complete the task more efficiently.
- For $\gamma_1 = 1$ (orange line), the step count initially increases but then drops sharply around episode 150. This drop corresponds to the agent's inability to explore effectively, causing it to complete episodes prematurely with poor results.
- For $\gamma_2 = 0.5$ (green line), the step count remains relatively low and stable, indicating that the agent performs suboptimal policies and terminates episodes quickly without finding efficient solutions.

These observations highlight that choosing an appropriate discount factor is crucial for successful training. A discount factor $\gamma_0 = 0.99$ balances short-term and long-term rewards effectively, leading to stable learning. In contrast, $\gamma_1 = 1$ leads to instability, and $\gamma_2 = 0.5$ results in simplistic policies that fail to optimize performance.

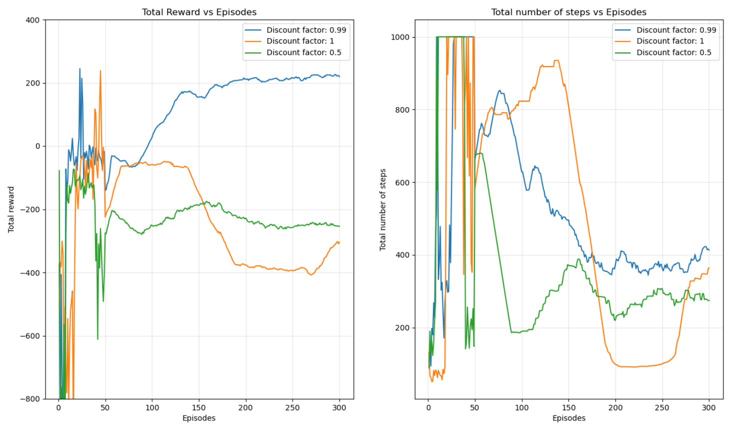


Figure 11: Total episodic reward (left) and total number of steps per episode (right) for different discount factors during training.

2.5 Effect of memory size

Figure 12 shows two plots comparing the total episodic reward and the total number of steps taken per episode during training for three different memory sizes: $L = 1000$, $L = 30000$ (initial choice), and $L = 50000$.

In the left plot, **Total Reward vs Episodes**, the agent's performance varies significantly with the choice of memory size:

- For $L = 30000$ (blue line), the agent achieves consistent high rewards after an initial exploration phase, stabilizing around 200. This indicates that the memory size of 30000 allows the agent to store enough experiences to learn a stable policy.
- For $L = 1000$ (orange line), the agent's performance is poor throughout training. The rewards remain highly negative, with significant fluctuations and no signs of improvement. This suggests that the limited memory capacity restricts the agent's ability to learn effectively by forgetting valuable past experiences.
- For $L = 50000$ (green line), the agent's performance improves more gradually but eventually reaches high rewards. The rewards peak around episode 250, showing that a larger memory size can lead to more stable learning but may slow down initial progress due to the increased amount of data to process.

In the right plot, **Total Number of Steps vs Episodes**, the effect of memory size on the agent's efficiency is evident:

- For $L = 30000$ (blue line), the number of steps initially peaks and then decreases, indicating that the agent learns to complete the task efficiently.
- For $L = 1000$ (orange line), the number of steps remains low and stable, reflecting the agent's inability to explore effectively or improve its policy due to limited memory capacity.
- For $L = 50000$ (green line), the step count initially remains high, peaking around episode 200. After this phase, the step count gradually decreases, showing that the agent eventually learns to complete the task efficiently, albeit later compared to $L = 30000$.

These observations suggest that increasing the memory size can improve training stability, but excessively large memory sizes may slow down the initial learning phase. Conversely, reducing the memory size leads to instability and poor performance due to the agent's inability to retain sufficient experience.

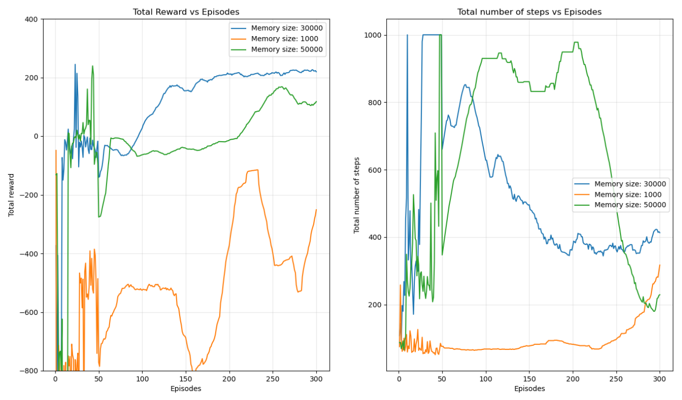


Figure 12: Total episodic reward (left) and total number of steps per episode (right) for different memory sizes during training.

2.5.1 Plot of the value function

The 3D surface plot shown in Figure 13 depicts the Q-values, $Q_\theta(s(y,\omega), \pi_\theta(s(y,\omega)))$, as a function of the lander's height $y \in [0, 1.5]$ and its angle $\omega \in [-\pi, \pi]$. Upon inspection, the resulting Q-value function is difficult to interpret and exhibits inconsistencies.

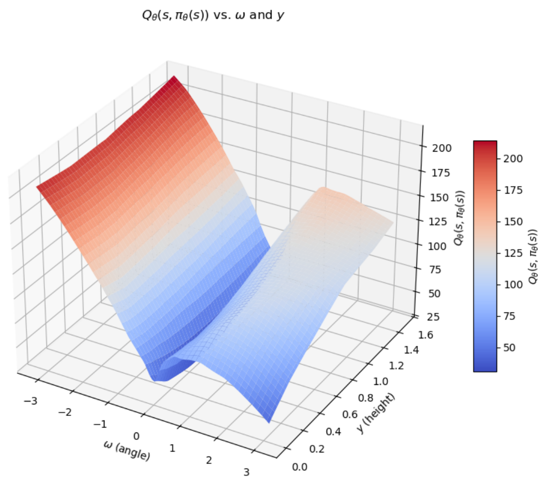


Figure 13: Q-values $Q_\theta(s(y,\omega), \pi_\theta(s(y,\omega)))$ as a function of height (y) and angle (ω).

One notable observation is that the Q-value function is not symmetric with respect to the angle ω, which is unexpected. Given that the problem dynamics and the reward structure are symmetric with respect to the lander's tilt angle, one would anticipate the Q-values to reflect this symmetry. The asymmetry might indicate that the agent has not sufficiently explored all state-action pairs during training, leading to a suboptimal estimation of the value function.

Furthermore, an unusual behavior can be observed when $\omega = 0$, where the lander starts perfectly aligned with the landing pad. Surprisingly, this state corresponds to one of the lowest Q-values, whereas tilted positions, particularly to the left, achieve higher Q-values. This outcome is counterintuitive since a state with zero tilt (horizontal alignment) should theoretically provide the best starting condition for achieving a stable descent and safe landing, as it minimizes angular penalties.

Another inconsistency is apparent when fixing the angle ω and varying the height y. When $\omega < 0$, the Q-values exhibit only small variations across different heights, which is unexpected given the reward structure. We would have expected to see what happens when $\omega > 0$, where higher y values lead to bigger rewards since the agent has more time to stabilize and correct the position of the rocket.

Overall, while the surface plot provides insights into the learned Q-values, the patterns observed are difficult to interpret intuitively. The behavior of the value function might be influenced by the complexity of the training process or the interplay between different states and rewards, making it challenging to derive clear conclusions from the plot alone.

2.5.2 Plot of the optimal engine direction policy

The behavior of the side engine control provided by the actor network is illustrated in Figure 14.

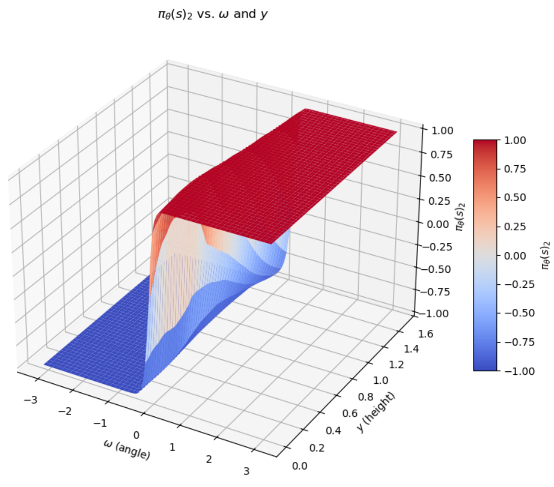


Figure 14: Optimal engine direction policy $\pi_\theta(s)_2$ as a function of height (y) and angle (ω).

The results are intuitive and align with expectations based on the problem dynamics. Since the plot considers the lander in a horizontally aligned position with the landing pad (i.e., $x = 0$), the agent only needs to fire the side engines when the lander is tilted (nonzero angle ω).

Specifically, the agent uses the side engines to adjust the tilt of the lander and bring it back to a horizontal position. When the lander is tilted to the left (i.e., $\omega < 0$), the policy outputs a negative value for the side engine control ($\pi_\theta(s)_2 < 0$), corresponding to firing the engine to the left. Conversely, when the lander is tilted to the right (i.e., $\omega > 0$), the policy produces a positive control value, indicating that the engine fires to the right. This symmetric and corrective behaviour ensures that the lander stabilizes its tilt angle during descent.

2.6 Comparison of learned policy and random agent

The total episodic reward over 50 episodes was compared between the trained actor agent and a random agent. Figure 15 shows the rewards and number of steps for each episode, while Table 3 summarizes the averages.

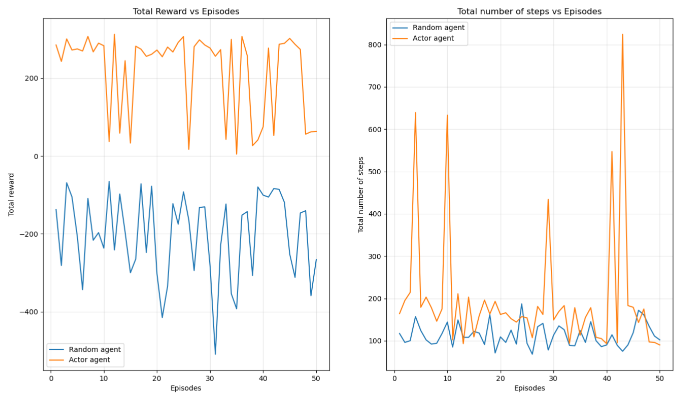


Figure 15: Comparison of total episodic reward and steps for the actor agent and random agent over 50 episodes.

Agent	Average Reward	Average Number of Steps
Actor Agent	218.52	198.56
Random Agent	-203.43	112.78

Table 3: Comparison of learned policy and random agent.

The actor agent consistently achieves high positive rewards, demonstrating that it has learned an effective policy to solve the task. In contrast, the random agent's negative rewards indicate poor performance due to ineffective decision-making. The higher number of steps taken by the actor agent could reflect bad starting states in which the agent has to take a lot of actions to stabilise and land safely, whereas the random agent's episodes tend to terminate early due to suboptimal actions.