EL2805 Reinforcement Learning
Computer Lab 2

Abdessamad Badaoui & Nasr Allah Aghelias
20011228-T118 & 20010616-T318

1 Problem 1 : Deep Q-Networks (DQN)

b.

We use an experience replay buffer to ensure that successive updates are less correlated. Otherwise, if we simply follow a trajectory, these updates will be strongly correlated, impacting the convergence of the algorithm.

d.

- **Optimizer** : We chose the Adam optimizer because it excels due to its adaptive learning rates, combination of momentum and RMSProp, and efficiency in computation and memory. With default hyperparameters and bias correction, it often leads to faster convergence in training deep neural networks.
- **Network Layout** : We decided to go with a network of two hidden layers. The dimensionality of the input is similar to that of the state space (8). The dimensionality of the output is equal to the number of possible actions (4). Furthermore, the number of neurons per hidden layer is equal to 32. The activation functions in the first and second hidden layers is the ReLu function. We decided to go with two hidden layer instead of one to increase the complexity of the model in order to match the complexity of the problem. However, we did not increase the number of neurons too much so that the computation would be faster
- **Discount Factor** γ : We chose the value $\gamma = 0.98$. We did not choose a small value so that we take the past experiences into account during updates. We do not want them to be forgotten rapidly. Since, we assumed that the training process would take some time, we made the discount factor as close to one as possible so that the previous Q-values shrink slowly.
- **Buffer size** L : Our buffer size was $L = 20000$. We chose a value in the middle of 10000 and 30000. Too small of a buffer capacity means that we do not store enough past experiences that will be used for batch updated. In contrast, too large of a buffer capacity means we are storing too many experiences that we need and the algorithm might diverge.
- **Number of episodes** T_E : We experimented a bit with the number until we found the proper one. It was the last hyperparameter we tuned. We noticed that the algorithm was still converging around $T_E = 400$ and has not stabilized yet. So we chose the value 600 as a final estimate of the number of episodes we need for our algorithm to converge.
- **Batch size** N : The batch size we used is $N = 32$. We noticed that our network do not need a large batch size to perform well so we went for a moderate value.
- **Update frequency** C : We calculated C as follows $C = L/N$. C should be large enough so that the algorithm does not jump from target to another too rapidly. However, if it was too large, the updates would be done very slowly.

- **Decay ε** : We used the following formula
  $$\varepsilon_k = max\left(\varepsilon_{min}, \varepsilon_{max} - \frac{(\varepsilon_{max} - \varepsilon_{min}(k-1)}{T_E - 1}\right) \quad \text{for } k = 1, 2, \ldots$$
  We start with a sufficiently large ε to explore the action space properly, then ε shrinks over time so that we go for the greedy policy more often.

e.

e..1

Figure 1 shows the total episodic reward and the total number of steps taken per episode during training :

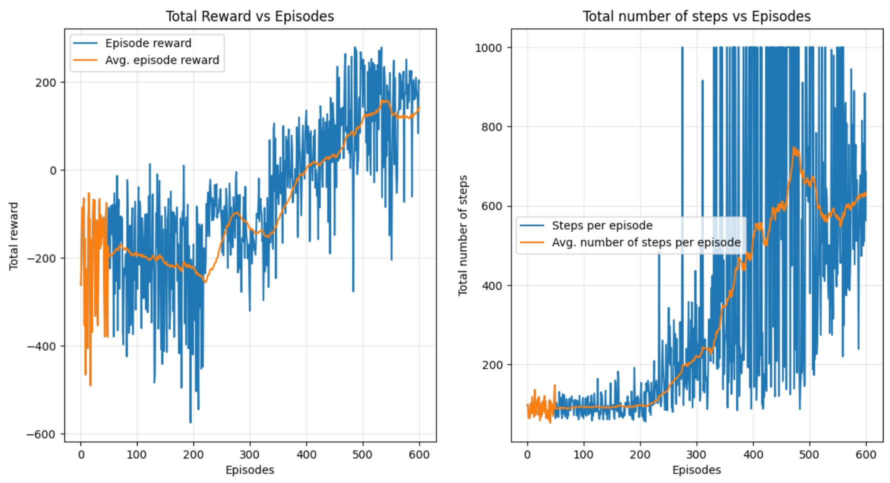

Figure 1: Average episodic reward with number of steps

We observed that during the initial episodes, the lunar lander rapidly crashes, leading to a quick arrival at the terminal state. As the training progresses, the agent begins to learn and takes optimal actions, resulting in an increase in the average reward over episodes. Additionally, it is noteworthy that as the network starts to learn, the training duration extends, as depicted in the figure on the right-hand side. This prolonged training duration is attributed to the lunar lander having to execute numerous actions to maximize its reward. Another key point to note is that as the agent learns the optimal policy, it will take fewer steps to achieve its goal.

e..2

Figures 2 and 3 show respectively the total episodic reward and the total number of steps taken per episode during training for $\gamma_1 = 1$ and $\gamma_2 = 0.6$, without changing the other parameters :

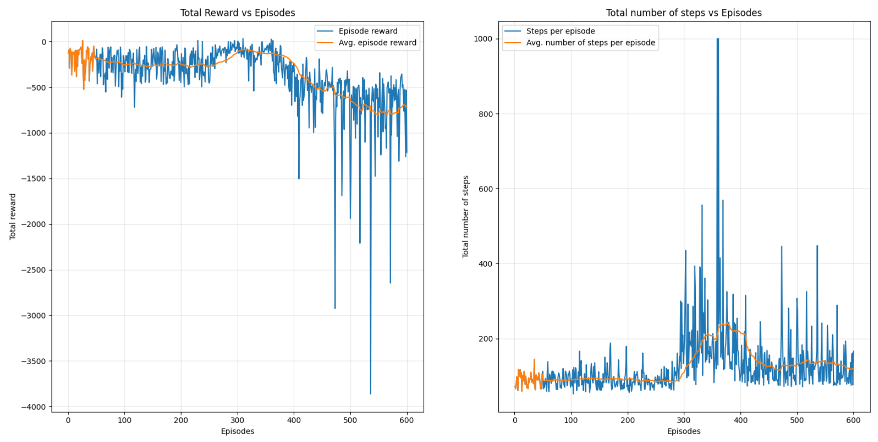

Figure 2: Average episodic reward with number of steps for $\gamma = 1$

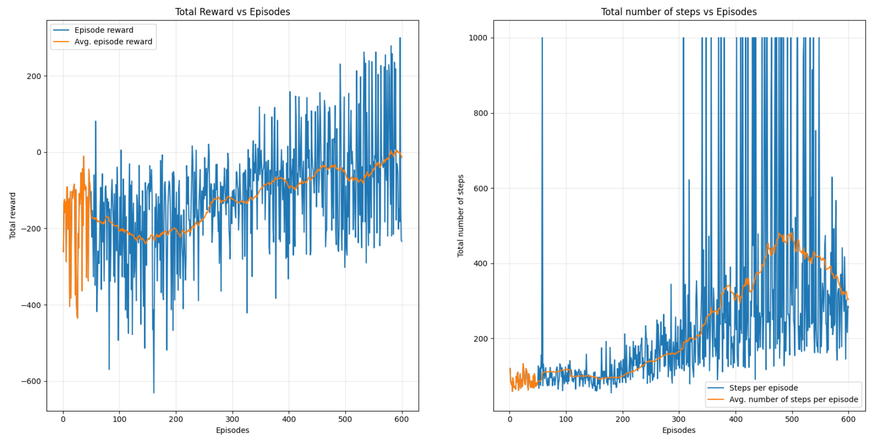

Figure 3: Average episodic reward with number of steps for $\gamma = 0.6$

When the discount factor is set to 1, it means that the agent is giving equal importance to both immediate and future rewards. This can make it challenging for the agent to "forget" or properly discount old rewards and focus on new ones. It becomes then challenging for the agent to properly attribute credit to specific actions that contributed to the final outcome. This can make it difficult for the agent to learn which actions are beneficial for achieving its goals. Whereas choosing a relatively small discount factor, such as 0.6, can have certain effects on the learning behavior of the agent. The discount factor influences the agent's consideration of future rewards, and a smaller gamma typically means that the agent will prioritize more immediate rewards over distant ones, and because the optimal policy in our case involves considering long-term consequences and planning ahead, a smaller discount factor slows down the convergence.

e..3

We trained the network for an extended duration (2000 episodes) to observe the progression of the average reward over episodes. The obtained results are presented in Figure **??**.

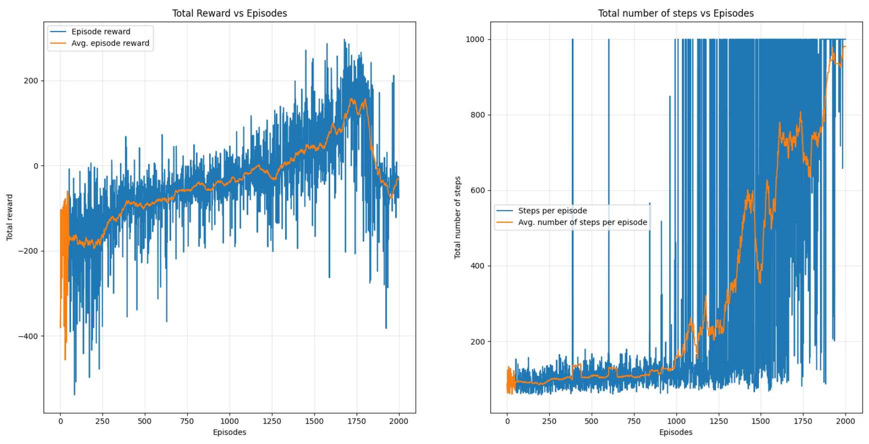

Figure 4: Average episodic reward with number of steps for 2000 episodes

We can explain this by the fact that the algorithm suffers from overfitting. This occurs when the agent becomes too specialized in the training environment, memorizing specific states and performing poorly when presented with others, potentially new trajectories encountered only in the final stages of training. This implies that the algorithm has already overfitted to some states before encountering these new trajectories. The occurrence of overfitting can be attributed to the agent having too many free parameters and being trained for too long.

Figure 5 shows the effect of the memory size on the performance of the algorithm :

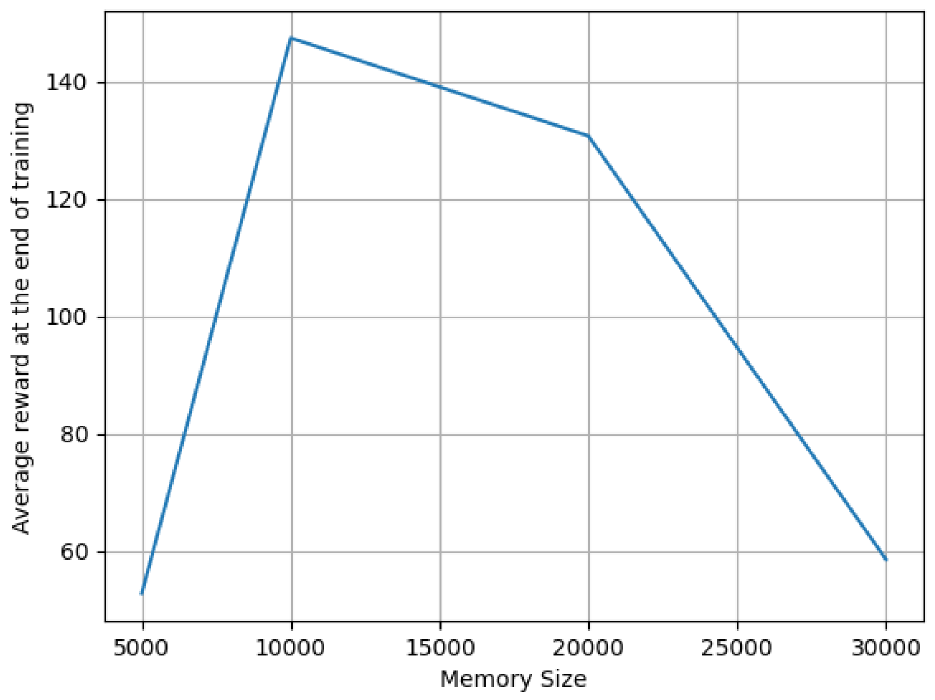

Figure 5: Average episodic reward at the end of the training with respect to the memory size

In the initial phase of increasing the replay buffer size, the positive trend in total average reward can be attributed to the buffer's ability to effectively decorrelate experiences. A larger replay buffer facilitates a more diverse set of samples for learning, reducing the likelihood of the algorithm becoming overly sensitive to the order of experiences. This decorrelation contributes to stability in learning, allowing the algorithm to better generalize from past experiences and improve overall performance. However, in the subsequent decreasing phase, the diminishing returns may be linked to the fact that an excessively large replay buffer introduces the risk of relying on outdated experiences for updates, leading to a decline in its adaptability to the current environment and a potential decrease in overall effectiveness. Striking the right balance in replay buffer size is crucial for ensuring a trade-off between stability and adaptability.

f.

f..1

Let $s(y, w) = (0, y, 0, 0, w, 0, 0, 0)$ where y is the height of the lander and w is the angle of the lander.

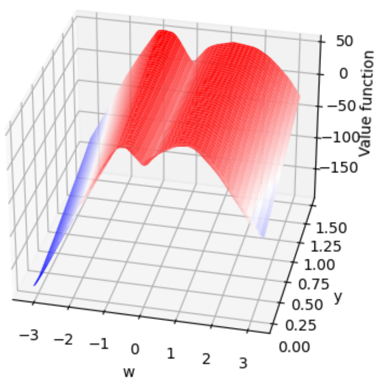

Figure 6: 3D plot of $\max_a Q_\theta(s(y,w),a)$

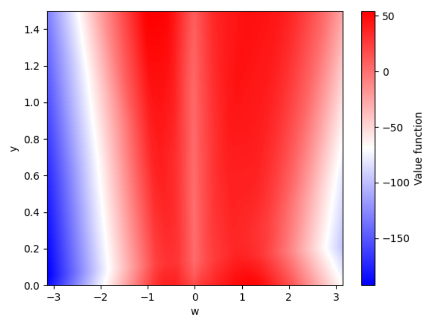

Figure 7: Temperature plot of $\max_a Q_\theta(s(y,w),a)$

We observe that $\max_a Q_\theta(s(y,w),a)$ is very high when $w = 0$ and low when w approaches π or $-\pi$. The height of the lander y, does not influence the shape of the plots (6) and (7) significantly. When $w \approx 0$, the orientation of the lander is good for landing. Consequently, $\max_a Q_\theta(s(y,w),a)$ is high since its angular velocity and velocities along x and y are all equal to 0. In contrast, $w \approx \pi$ or $w \approx -\pi$, the lander is upside down and has a high chance of crashing. Hence, the value of $\max_a Q_\theta(s(y,w),a)$ is low.

Now, let's observe $\text{argmax}_a Q_\theta(s(y,w),a)$ with respect to y and w

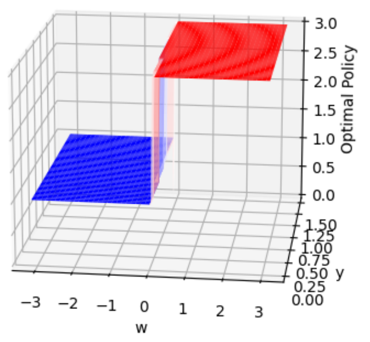

Figure 8: 3D plot of $\text{argmax}_a Q_\theta(s(y,w),a)$

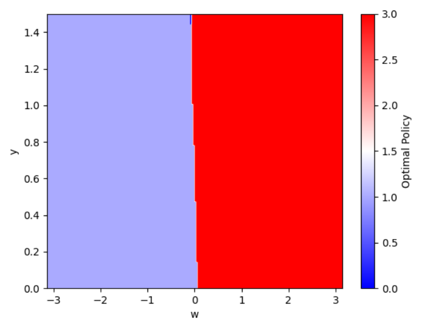

Figure 9: Temperature plot of $\text{argmax}_a Q_\theta(s(y,w),a)$

We observe that the figures 8 and 9 are divided into two spaces. If $w < 0$, $\text{argmax}_a Q_\theta(s(y,w),a) = 1$. If $w > 0$, $\text{argmax}_a Q_\theta(s(y,w),a) = 3$. This holds true almost independently of the value of y. The action (1) corresponds to firing the left orientation engine whereas the action (3) corresponds to firing the right orientation engine.
Since the lander in this scenario is above the proper lading zone (i.e. $x = 0$) and his velocity is equal to 0, the only thing our network worries about is the orientation of the lander. When $w < 0$, our network decides to fire the left orientation engine whereas when $w > 0$, our network fires the right orientation engine.

g.

Let us compare the behaviour of our Q-network to the random agent.

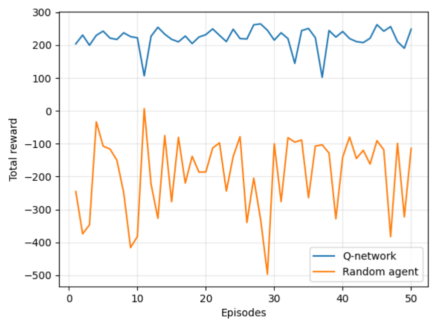

Figure 10: Total episodic reward with respect to the episodes

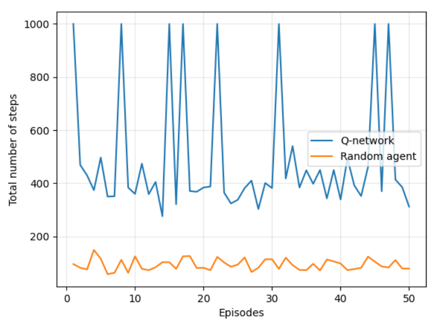

Figure 11: Total number of steps with respect to the episodes

In the figure 11, we notice that our Q-network performs significantly better than the random agent. All of the network total episodic rewards are over 100 whereas the values of the random agent are all negative.
In the figure 11, we observe that our Q-network takes a lot of steps to finish the episode compared to the random agent. We can explain that by the fact that the random agent choice of actions leads the lunar lander to crash almost certainly. In contrast, our Q-network should be careful with the spaceship so that it lands with small velocity in the proper location.