KUNGLIGA TEKNISKA HÖGSKOLAN

EL2805 - Homework 02

Emillio Antonio Zubizarreta: 20020112-T274

Problem 1

b

As whith Q-learning algorithms, DQN relies on bootstrapping to calculate the Q-values of the network. However, when dealing with big number of episodes, these samples are highly correlated, with the target values change continuously as the network updates. To solve this issue, de implementation of a relay buffer that selects experiences with random uniform probabilities breaks temporal correlations between samples and improves training stability, reducing the effect of rapidly changing targets during bootstrapping.

d

To build the final network from the given code, several modifications have been taken in order to adapt the training of our agent in order to satisfy the goal requirement. This includes selecting a valid optimizer to perform function approximation with, creating an efficient neural network to calculate the results with, and choosing the correct set of hyperparameters $(\gamma_0, L, T_E, C, N, \epsilon)$ that will obtain the desired output.

For the selection of the optimizer, we used Adam optimizer, since it is a reliable option when dealing with RL problems by adjusting the learning rate per parameter, which combined with a bias-correction mechanism avoids slow convergence and instability, generating faster convergence and more stable optimization.

For the definition of neural networks, we conserved the layered structure of input $\rightarrow$ hidden $\rightarrow$ output layers seen in exercise session 3 for both the final and target networks, establishing 64 neurons per layer to achieve the desired behavior.

Additionally, in order to improve the results of the DQN algorithm, we decided to implement a modification by using double DQN (DDQN), which ensures faster and more stable learning by reducing the overestimation of the Q-values.

This way, after several tries and modifications, we achieved our goal network with the following hyperparameter configuration:

- Discount factor (γ_0): 0.97
- Size of experience relay buffer (L): 15000
- Number of episodes (T_E): 100
- Update frequency of the target neural network (C): $int(L/N) = 234$
- Training batch (N): 64
- Epsilon (ϵ): $\begin{cases} 1 & k = 1 \\ max\left(0.05, 0.99\left(\frac{0.05}{0.99}\right)^{\left(\frac{k-1}{0.95T_E-1}\right)}\right) & otherwise \end{cases}$, with $k \in [1, T_E]$

Where some of them were directly extract from the initial recommendations from the exercise, such as C, N or the exponentially decayed ϵ, updated within every new episode (experiments where epsilon was updated more often and with lower rates have been tested, with worse results); and the rest of the parameters were tuned according to the resulting average reward

obtain from each episode in the *DQN_check_solution.py* file, with which we get an average total reward of **110.5** $\pm$ **29.3** with a confidence of **95%**, reaching the desired goal.

```
Network model: MyNetwork(
(input_layer): Linear(in_features=8, out_features=64, bias=True)
(hidden_layer): Linear(in_features=64, out_features=64, bias=True)
(output_layer): Linear(in_features=64, out_features=4, bias=True)
(activation): ReLU()
)
Checking solution...
Policy achieves an average total reward of 110.5 +/- 29.3 with confidence 95%.
Your policy passed the test!
Episode 49: 100%|[][][][][][][][][][]| 50/50 [00:21<00:00,  2.38it/s]
```

More arguments related to the tuning decision of these hyperparameters will be detailed in the following sections.

e

1

In Figure 1 we observe how the total episode reward and the number of steps taken in each episode of the algorithm evolves throughout the completion of T_E episodes, implementing our final layout.

With this plot we can justify the positive results reflected in the previous section, observing that the agent reaches the 200 points reward threshold in several episodes, resulting in an almost strict monotonic average episode reward curve after crossing the $n\_ep\_running\_average$ mark, which allows the network to delay the calculation of both averages until it has started to develop. Lastly, when observing the steps graph, we observe that the algorithm requires a big number of steps in order to reach the needed goals, ending with an average number of steps per episode of around 680 steps/episode. However, this may not reflect a real necessity of the network, so the fact that we are limiting the maximum number of steps to 1000 is a good practice to maintain a good performance.

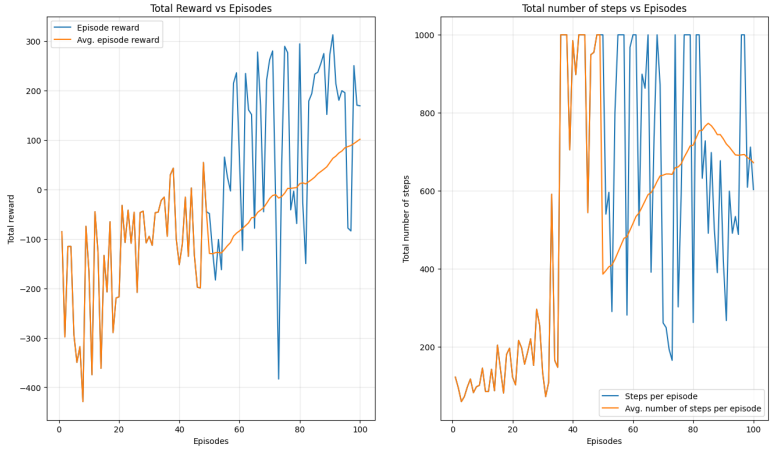

Figure 1: Reward vs Episodes and Steps vs Episodes representation for optimal layout

2

Originally we established $\gamma = 0.95$, but after several iterations we decided to raise the value to the definitive $\gamma_0 = 0.97$. For this section we run the code with two new discount factors ($\gamma_1 = 1$ and $\gamma_2 = 0.25$), in order to extract information regarding its relevance in the output of the network and in the training process.

In Fig 2 we observe the plots of the episode reward and steps when we selected $\gamma = \gamma_1$. Regarding the steps taken in each episode, there is no big difference with respect to the one from the optical layout, but, when comparing the reward graph, we observe a poor performance of the algorithm. This may be caused by the fact that, fixing the discount factor to 1, we allow the system to carry errors in each trajectory (since we are not discounting at all). This may end up with an increase in the Q-values of less optimal states that may lead to a bad action selection in the bootstrapping process, resulting in a bad performance and low total rewards in each episode.

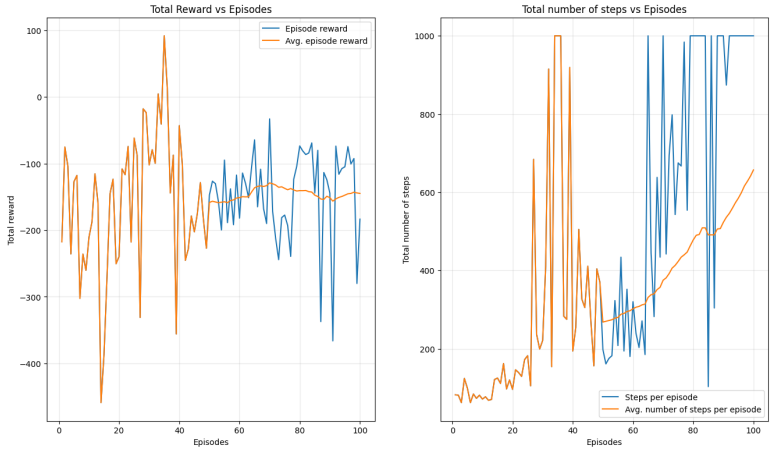

Figure 2: Reward vs Episodes and Steps vs Episodes representation for $\gamma_1 = 1$

On the other hand, in Fig 3 we observe an experiment where we select a discount factor $\gamma_2 \ll \gamma_0$. In this case we are restricting too much the calculation of the Q-values, since only phew times per episode we involve the selection of the best action from the Q-value function in the updating process, resulting in both graphs maintaining their mean during the episodes.

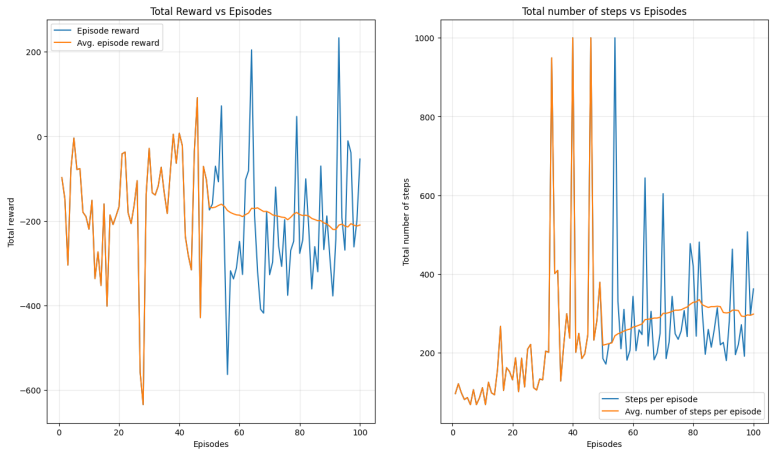

Figure 3: Reward vs Episodes and Steps vs Episodes representation for $\gamma_2 = 0.25 \ll \gamma_0$

3

For our optimal layout, we established the number of episodes of our simulation at $T_E = 100$, but we will investigate the results of the experiment when we reduce and increase this value.

In Fig 4 we doubled the number of episodes up to 200, obtaining apparently great results for both reward and steps, even if the resulting average steps is higher than expected. However, when checking the generated .pth file, we find out that we don't satisfy the criteria, even if we observe in the graph that we reach 200 points in the reward function multiple times. One hypothesis may reside in the high dependence of the epsilon exploring with the number of episodes, which reduces the exploring rate when the number of episodes increases and produces excessive exploration in later stages, which may affect the replay buffer behavior, degrading the learned policy.

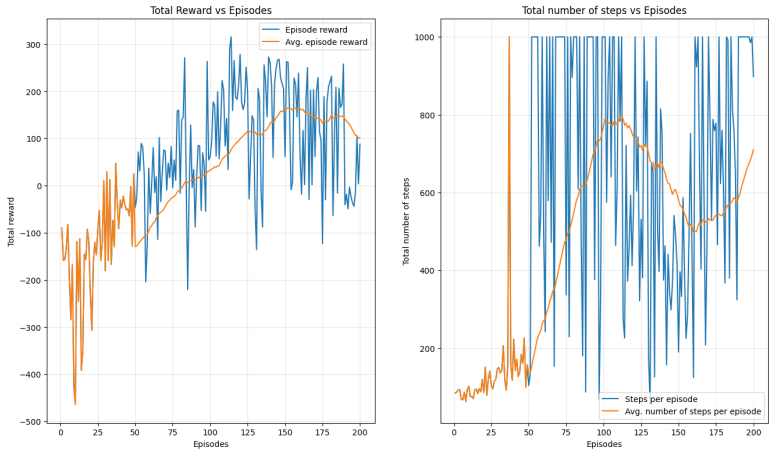

Figure 4: Reward vs Episodes and Steps vs Episodes representation for $Episodes = 200$

On the other hand, if we choose a smaller number of episodes(in this case 75), what we observe is that the simulation starts similarly than with the optimal layout for both the step and episode reward functions, but since it stops very soon we can't reach a valid solution. This result can be observed in Fig 5.

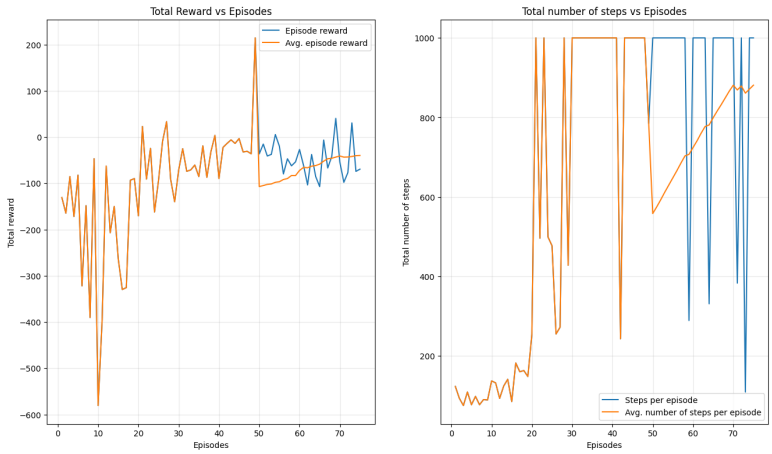

Figure 5: Reward vs Episodes and Steps vs Episodes representation for $Episodes = 75$

Now, if we focus on the memory of the neural network, in the final layout we established a size of the relay buffer of 15000, but we can also analyze the behavior of the simulation if we lower or increase its value.

In Fig 6 and Fig 7 we observe the resulting graphs when we reduce the buffer size to 5000 and when we increase it to 30000 respectively. For the first case, we reduce the update frequency C while reducing the capacity of the buffer, which causes longer waiting periods until the target network gets updated, while training the network with similar policies every time, resulting in bad episode reward results. On the other hand, if we increase heavily the memory size, we may include past suboptimal policies, which lead to slower learning (observed in the steps plot, with almost every episode reaching the maximum number of steps) and less consistant targets.

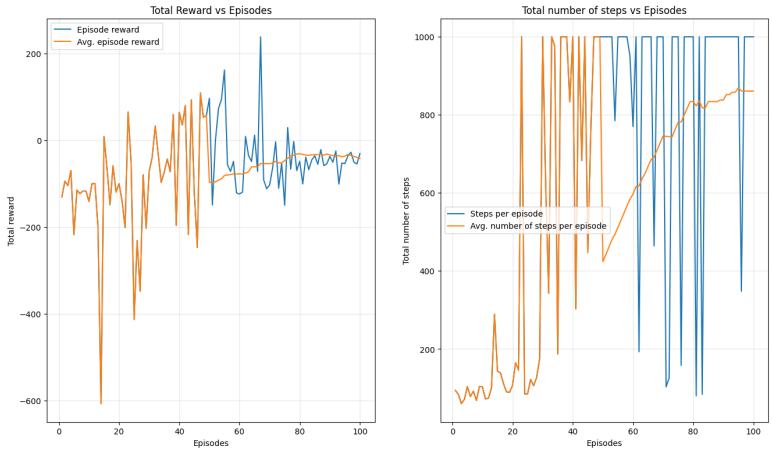

Figure 6: Reward vs Episodes and Steps vs Episodes representation for $Buffer\ size = 5000$

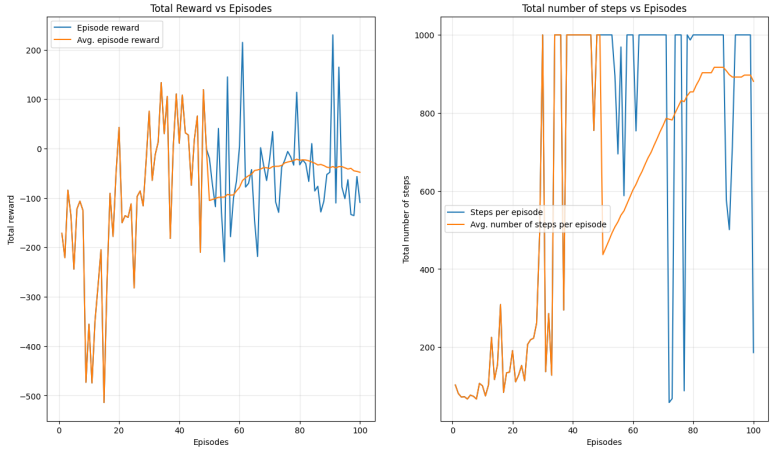

Figure 7: Reward vs Episodes and Steps vs Episodes representation for $Buffer\ size = 30000$

f

1

With the obtained neural network and the state restriction presented in the problem statement we obtain the 3D graph shown in Fig 8, observing the maximum value of the Q function for each state where all the state space elements of the lunar lander are 0 except the height and the angle of the lander, observing a smooth symmetric figure along w. However, it is concerning the fact that the max Q-values of the states with the most critical orientations ($\pm\pi$) is very high, generating the possibility of misleading the network to make sub optimal decisions. Nevertheless, we need to remember that we are not plotting the whole set of states that the system can go to, but instead a restrictive set, which may never reach in the learning process of

the algorithm. This way, the value estimates are only reliable in values with more reasonable orientations w, since this are the ones that we are going to use in reality.

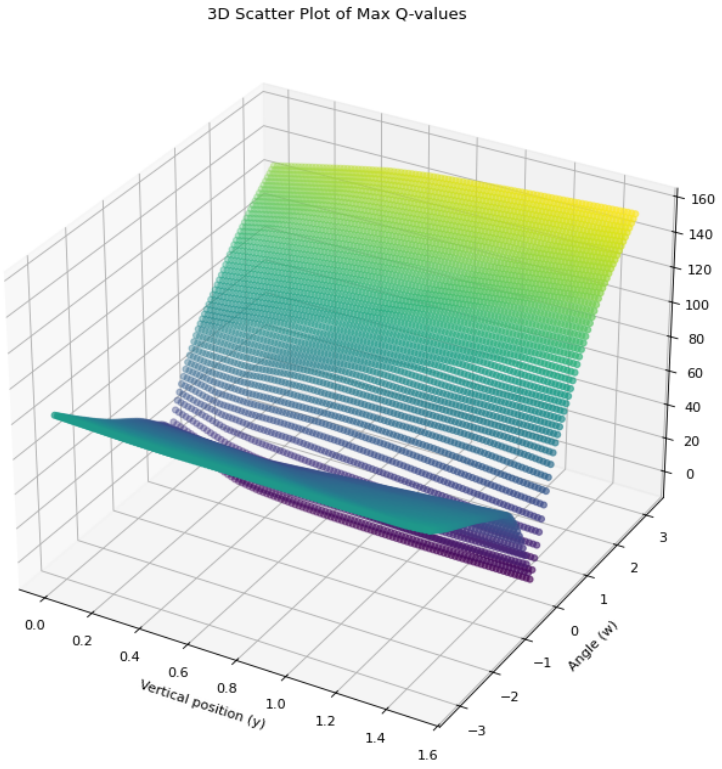

Figure 8: Max Q-values representation

2

In Fig 9 we observe another 3D plot where the optimal action to be selected after each state has been represented, where each number represents an action taken by the lunar lander:

- a $= 0 \rightarrow$ Do nothing
- a $= 1 \rightarrow$ Fire left orientation engine
- a $= 2 \rightarrow$ Fire main engine
- a $= 3 \rightarrow$ Fire right orientation engine

This representation make sense, since the objective of choosing the actions of the graph resides in moving the lunar lander closer to the goal area. This is why when we have a negative orientation, where the right side of the lander is facing closer to the surface, we want to fire the left engine and do the opposite when we deal with positive orientations. Additionally, we observe a set of states when we are dealing with angles around 0.5 rad where the best action is to not use any engine at all, which can be valid if the angle with which we are approximating the landing zone is oriented perfectly. However, as we discussed in the previous section, the results obtained with critical and rarely visited states may cause strange answers for our simulation. That is the reason why, for example, we observe that the best action when we are practically perpendicular to the ground with very low heights is to fire the main engine, which will not help at all to avoid crashing with the land.

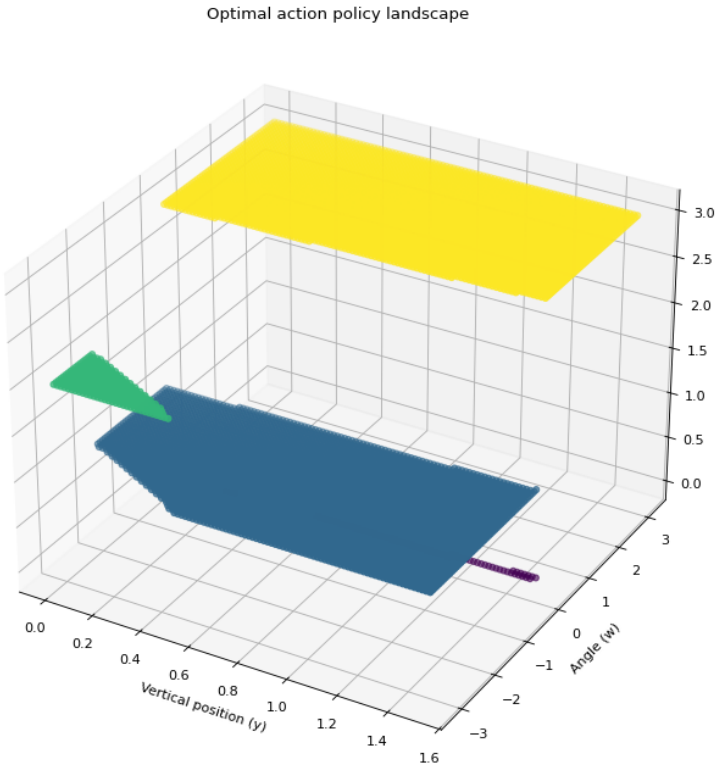

Figure 9: Optimal action policy landscape representation

g

Lastly, in Fig 10 we observe the episodic reward and step graphs of the network when we base the training process simply on a random agent. Since we are not using any algorithm to find an optimal policy, we will be just choosing a random state and computing their reward without learning throughout the episodes.

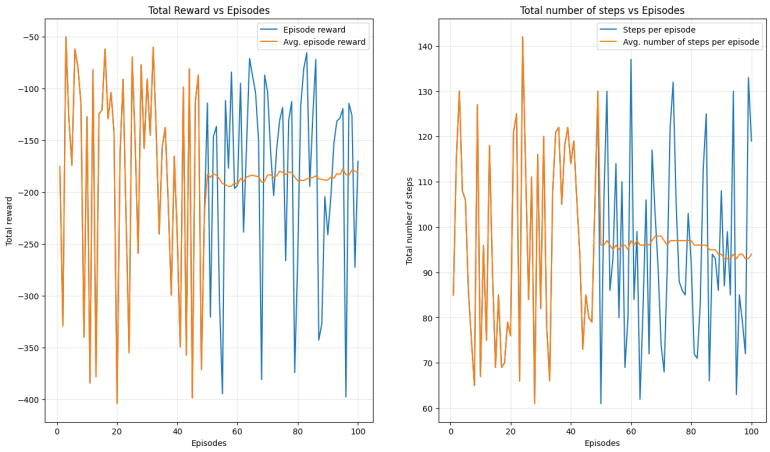

Figure 10: Reward vs Episodes and Steps vs Episodes representation for *RandomAgent* implementation