[EL2805] Reinforcement Learning - Laboratory 2

Valeria Grotto & Dalim Wahby
ICT Innovation M.Sc., EIT Digital
Kungliga Tekniska högskolan
Stockholm, Sweden
vgrotto@kth.se | dalim@kth.se
20010126-6021 | 19970606-T919

All of our code can be found in our GitHub repository[1].

1 Problem 1: DQN

a) Familiarize yourself with the code in `DQN_problem.py`. Check the agent taking actions uniformly at random (implemented in `DQN_agent.py`). You will use this agent as a baseline for comparison with your final agent.

In Figure 1, we can see the behaviour of the agent over 100 episodes. The agent choses actions randomly at every step, in fact we can see an high variance of both the number of steps and the obtained reward.

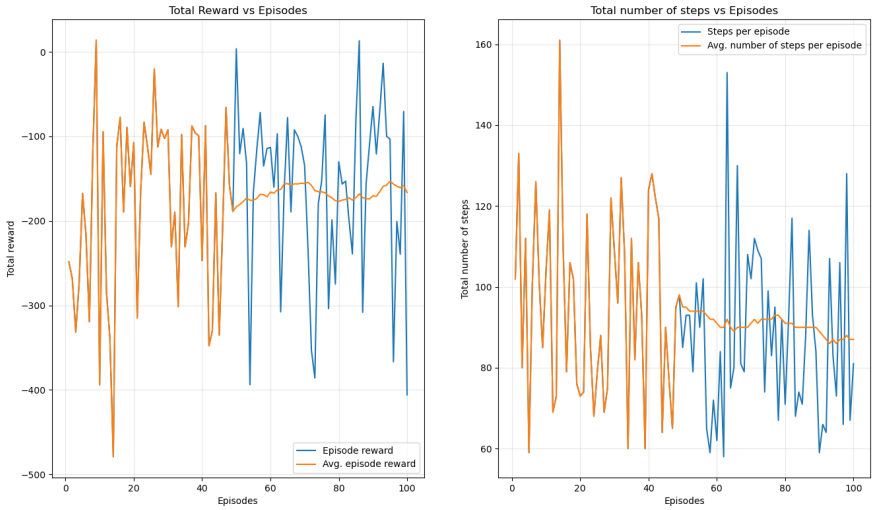

Figure 1: Random Agent behaviour over 100 episodes

b) Why do we use a replay buffer and target network in DQN?

According to (1), using a replay buffer has several advantages. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Second,

[1]GitHub: `https://github.com/valegr8/EL2805_lab2`

learning from consecutive samples is inefficient due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates, the network is therefore more stable and it is possible to apply the SGD algorithm which requires i.i.d. samples. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. By using experience replay the behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

Adding a target network makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_t + 1, a)$ for all a and hence also increases the target y_i, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_i, making divergence or oscillations much more unlikely.

c) Solve the problem

Figure 3 shows the training process of the final network and Figure 18 in section (h) validates our network.

d) Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used (γ, L, T_E, C, N, ϵ) and which modification did you implement (if any). Motivate why you made those choices.

In Figure 2 is possible to see the layout of the final network which consists in a Dueling DQN. In fact, we have an initial linear layer with a *ReLu* as activation function, then we have two different branches: the first is the one that learns the value function $V(s)$, the second learns the Advantage $A(s, a)$. This modification gives stability to the network. We used Adam as optimizer with a learning rate of 0.001.

```
Network model: DuelingDQNetwork(
  (net): Sequential(
    (0): Linear(in_features=8, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=4, bias=True)
  )
  (value_function_layer): Sequential(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=1, bias=True)
  )
  (advantage_layer): Sequential(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=4, bias=True)
  )
)
```

Figure 2: Dueling DQN model

We choose the following hyperparameters:

- $\gamma = 0.99$, we tried different values for this parameter (0.2, 0.5, 0.8, 0.95, 0.99) and the only one that made the network converge was last one.
- L, the length of the buffer is 30000. We tried different sizes, starting from 5000, however we saw that a bigger size helped the learning process since there are many possible different trajectories to consider.
- T_E, the number of episodes, has been set to 550, however we implemented *early stopping* when the average reward is higher than a threshold that we setted to 220. We saw that on average it takes around 500 episodes to train our network.

- C was set to `int(L/N)`, in fact setting C to a value either too big or too small can make the learning process unstable.
- N, the batch size was setted to 64. We saw that a batch size of 32 was not enough for the algorithm to converge, instead 128 was too big and slowed down the learning process.
- ϵ: we choose an ϵ parameter that decreases exponentially according to the following formula

$$\epsilon_k = max(\epsilon_m in, \epsilon_m ax(\frac{\epsilon_m in}{\epsilon_m ax})^{\frac{k-1}{Z-1}})$$

  Where $\epsilon_m ax = 0.99$, $\epsilon_m in = 0.05$ and $Z = 0.95 T_E$. A decreasing ϵ helps to minimize the regret of the algorithm, in fact we use an ϵ greedy strategy to select the action to take at each step.

Finally, we implemented a Combined experience replay which helps prioritize new experience by always adding the latest experience to the batch that we use for training.

e) Once you have solved the problem, do the following analysis:

1) Plot the total episodic reward and the total number of steps taken per episode during training. What can you say regarding the training process?

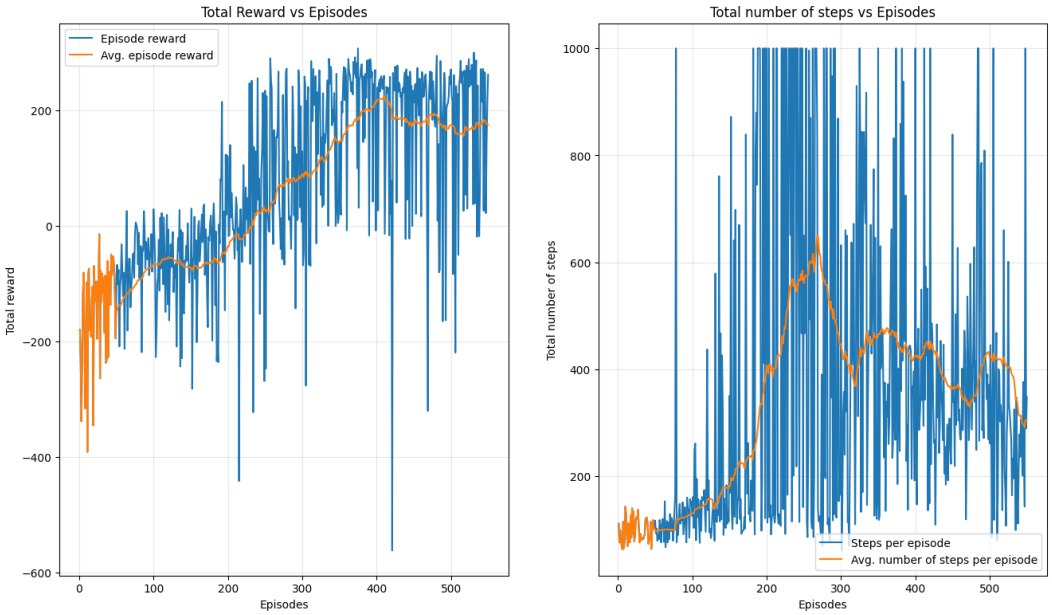

Figure 3: Final Network Training Process

Figure 3, shows the process of training of the Dueling DQN. At the beginning we have an high variance, in fact the memory is initialized with experience generated randomly. We can see that the average reward increases steadily over time and it converges around 400 episodes, however if we train for longer we see that also the number of steps per episode decreases, this is probably due to the decreasing exploration parameter. In fact, in our case ϵ decreases exponentially over time, especially at the end of the training (since $Z = 0.9 * T_E$). Notice that we setted a limit to the number of steps (to 1000) because we noticed that sometimes the lunar lander could get stuck somewhere and this stopped the training.

(2) Let γ_0 be the discount factor you chose that solves the problem. Now choose a discount factor $\gamma_1 = 1$, and a discount factor γ_2 « γ_0. Redo the plots for γ_1 and γ_2 (don't change the other parameters): what can you say regarding the choice of the discount factor? How does it impact the training process?

The initial discount factor was $\gamma_0 = 0.99$. We tested first $\gamma_1 = 1$, this value does not discount future rewards, that could be due to wrong trajectories. In fact, in Figure 4 is possible to see that the reward quickly decrease over time.

For the second discount factor we choose a value much lower than the initial one, $\gamma_2 = 0.2$, in this case we discount a lot future rewards and the learnt (state,action) depends more on the actual rewards. This is also not optimal, in Figure 5 and 6 we can see that the algorithm is learning but it is very slow to converge and the variance increases over time.

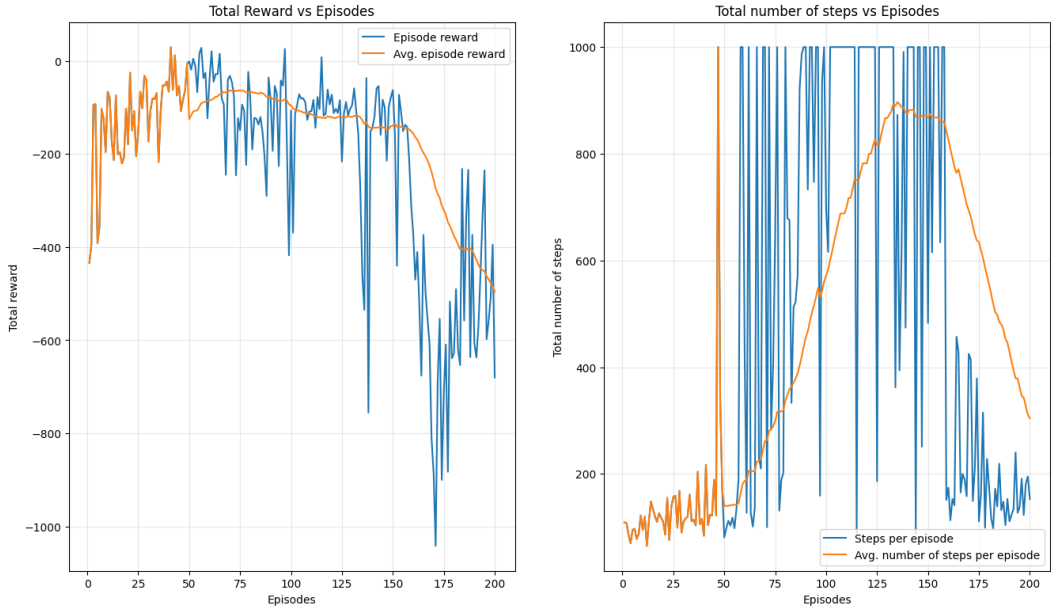

Figure 4: $\gamma_1 = 1$

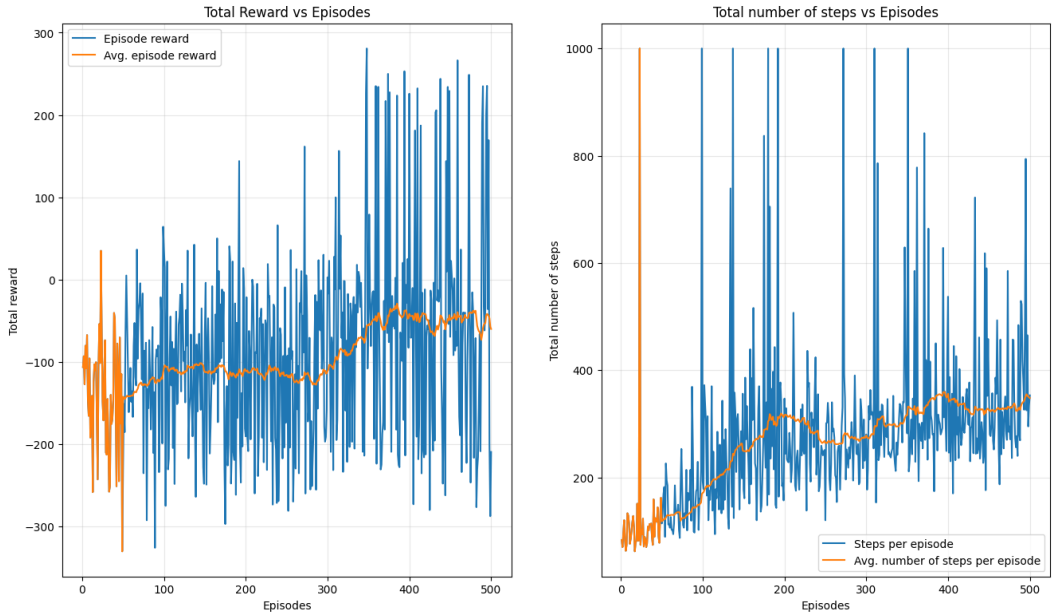

Figure 5: $\gamma_2 = 0.2$

(3) For your initial choice of the discount factor γ_0 investigate the effect of decreasing (or increasing) the number of episodes. Also investigate the effect of reducing (or increasing) the

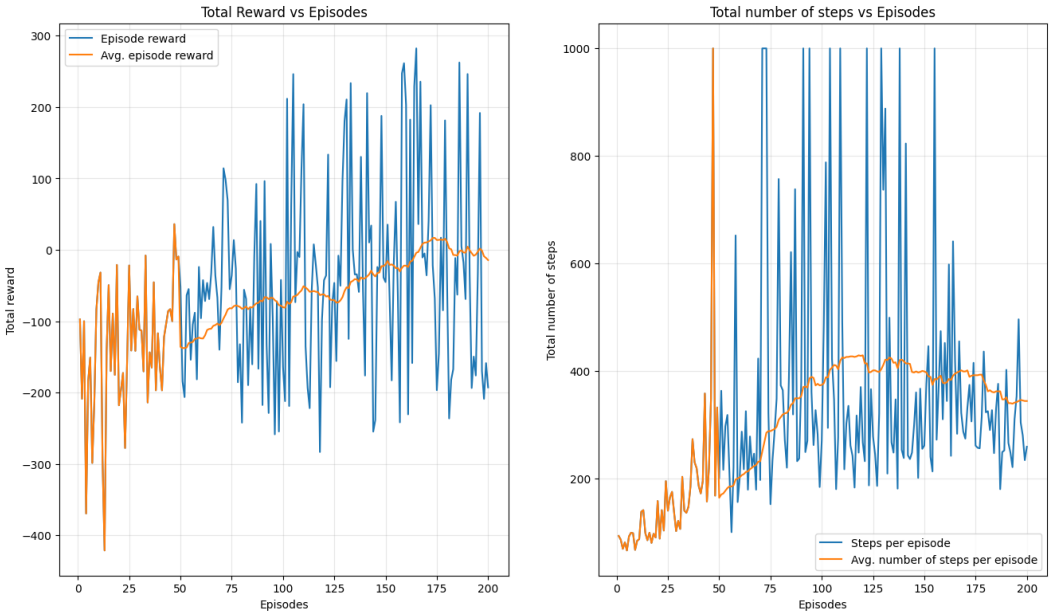

Figure 6: $\gamma_2 = 0.5$

memory size. Document your findings with representative plots.

We tested the network with different number of episodes, we found out that if we train for more than 600 episodes the network starts to forget, see Figure 7. Based on our findings wee should stop around 550 episodes, however we implemented an early stopping of our training when the average reward over the last 50 episodes was above a certain threshold. Training for less than 500 episode is not enough for our network to converge and learn an optimal policy, i.e. see Figure 8.

We also investigate the effect of the memory size on the training performance, we tested the following sizes: 5000 (Figure 9), 10000 (Figure 10), 20000 (Figure 11), 30000 (Figure 12) and 40000 (Figure 13). We found that a large memory size is necessary for the learning process, in fact with our model, both the buffer sizes of 30000 and 40000 converged to an "optimal" policy.

f) For the Q-network Q_θ that solves the problem, generate the following two plots:

(1) Consider the following restriction of the state $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, where y is the height of the lander and ω is the angle of the lander. Plot $max_a Q\theta(s(y, \omega), a)$ for varying $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$. You should obtain a 3D plot. Does the value of the optimal policy you found make sense? Explain it.

(2) Let s be the same as the previous question, and plot $argmax_a Q\theta(s(y, \omega), a)$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behaviour of the optimal policy make sense? Explain it.

Restricting the state to $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, is an equivalent of restricting the space to states that are directly above the landing strip, with different angles. This results in a two degree of freedom system, as shown in Figure 14.

While analysing the value function, shown in Figure 15 on the left side, we can see that it does not always make sense. For example, in regions close to $y = 1.4$ and $\omega = -3$, which corresponds to the spacecraft upside down at the top of the environment, the value is high compared to the other regions. This should not be the case, since the agent should learn to avoid being upside down. However, we think this is the case, due to the fact that these kind of regions are hardly explored during training, which results in the Q-function not being able to converge to its true value for said regions.

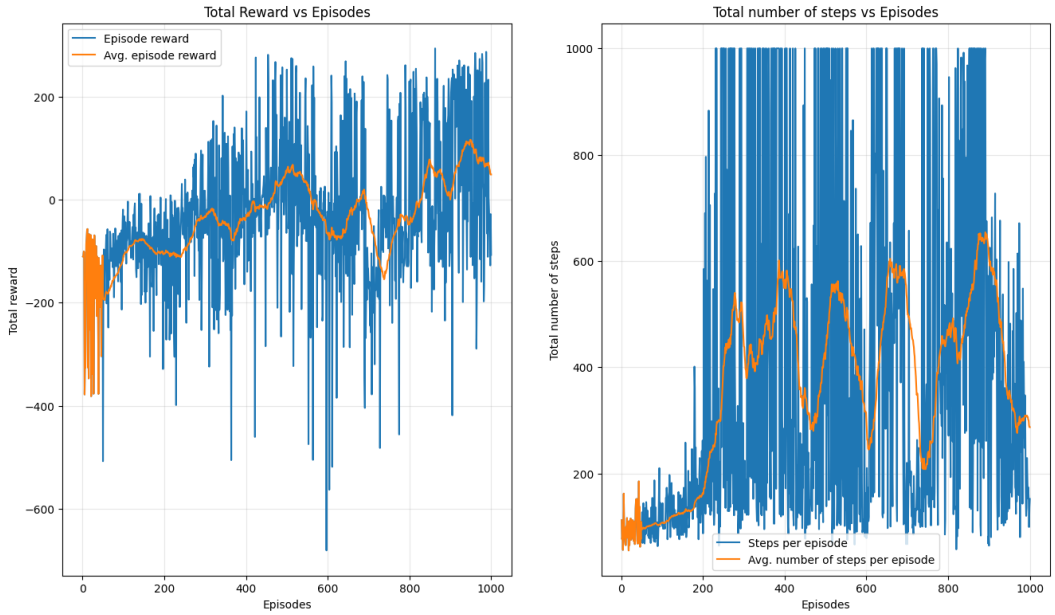

Figure 7: Training for 1000 episodes

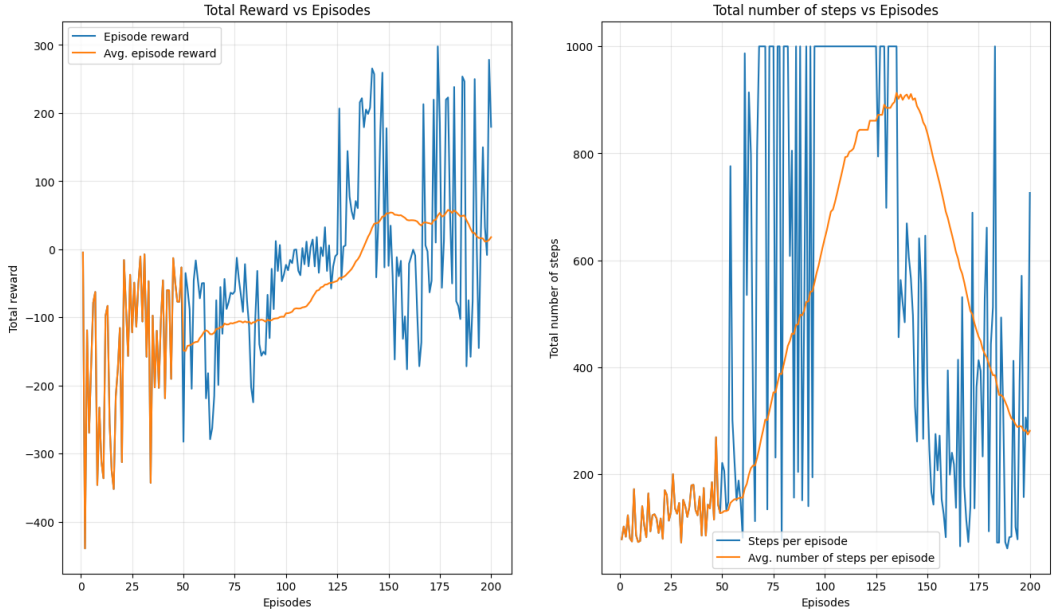

Figure 8: Training for 200 episodes

While analysing the optimal policy for different y and ω values, we observe that the actions mostly make sense. For example if we take a look at $y \in [0.8, 1.0]$ and $\omega \in [0, \pi]$, we can see that the agent chooses action 3, i.e. fire right engine, in order to turn the spacecraft the right way up. However, for $y = 1.4$ and $\omega \in [-\pi, -\frac{\pi}{2}]$, the agent also chooses to fire the right engine. This action is not necessarily the optimal action, the agent does not see the situation a often where the spacecraft is upside down. The corresponding plot can be seen in Figure 15, on the right side.

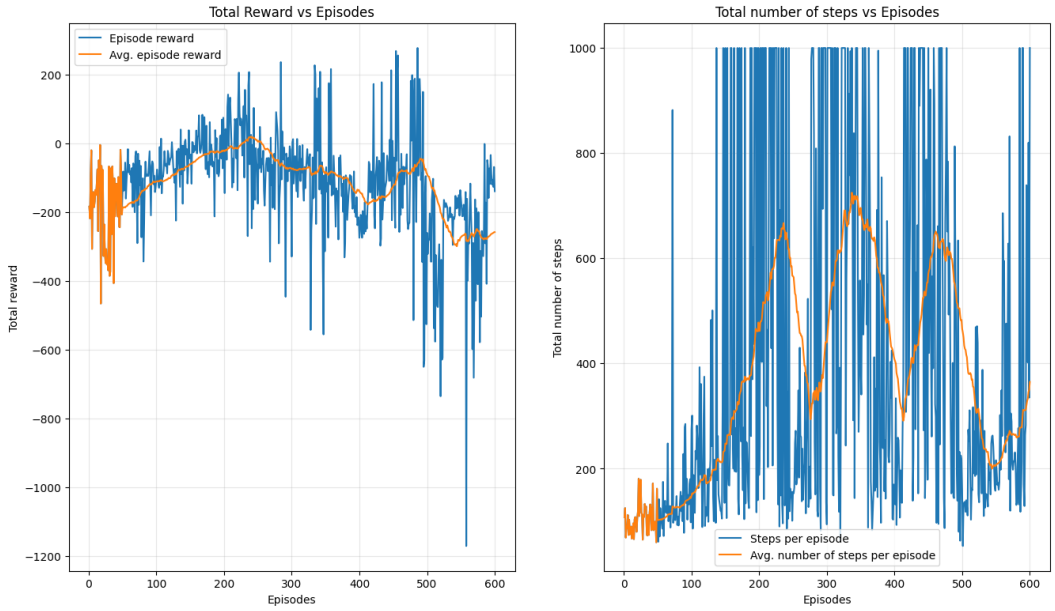

Figure 9: Training with memory size of dimension 5000

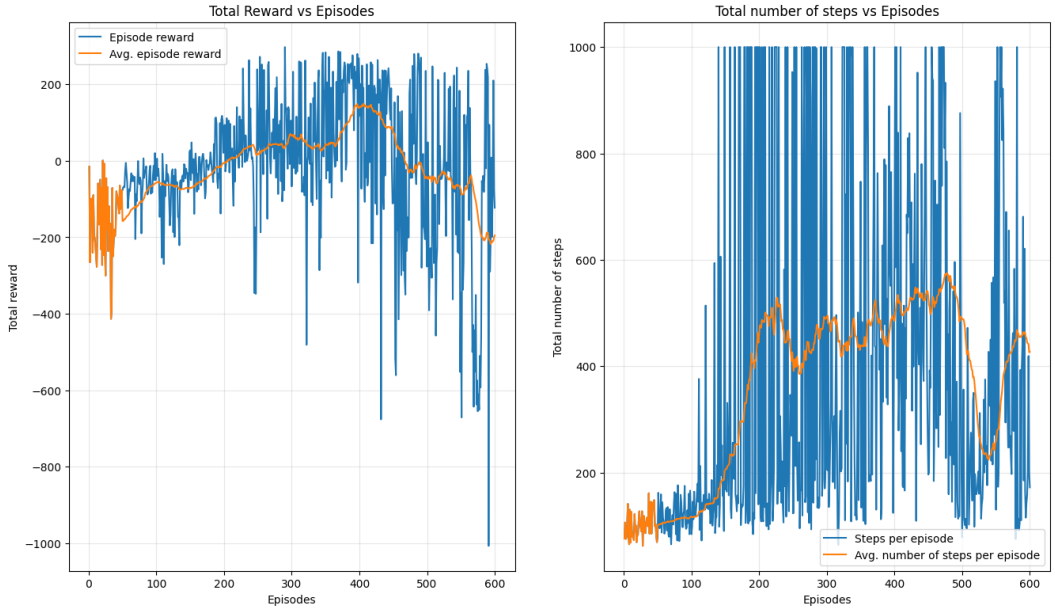

Figure 10: Training with memory size of dimension 10000

g) Compare the Q-network you found with the random agent in (a). Show the total episodic reward over 50 episodes of both agents.

In Figure 16 is shown the behaviour of the Random agent over 50 episodes. We can notice an high variance and the scores that are almost never positive. Instead, Figure 17 shows the behaviour of the agent controlled by the trained network, we can see a lower variance and an average reward around 250.

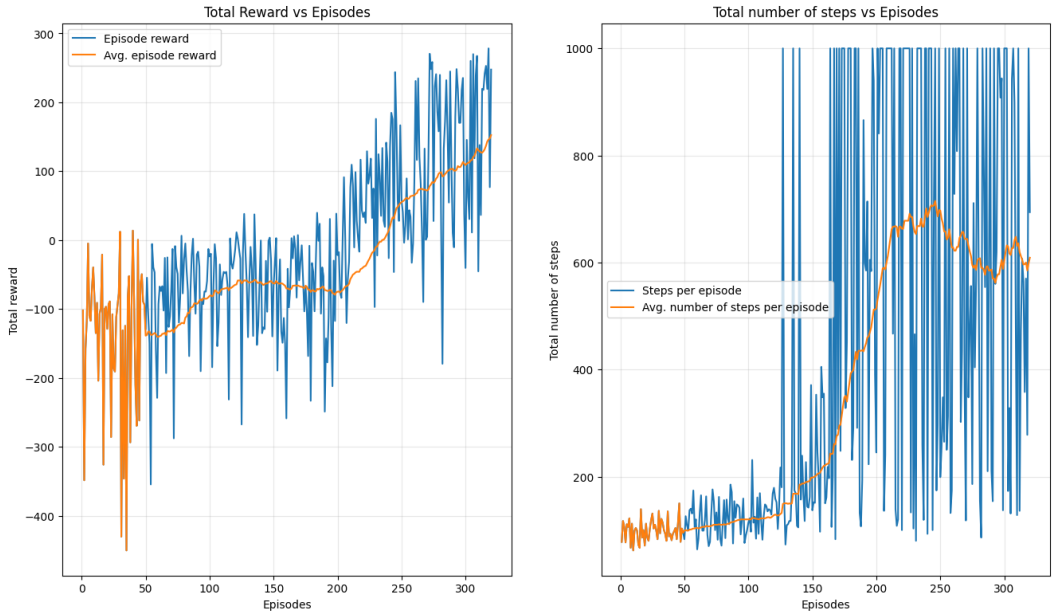

Figure 11: Training with memory size of dimension 20000

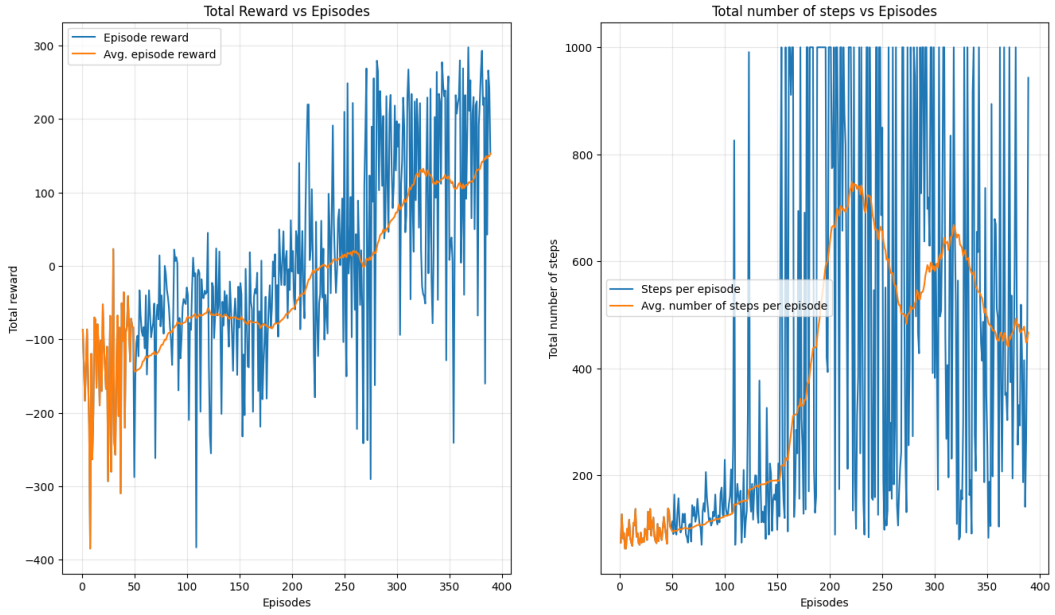

Figure 12: Training with memory size of dimension 30000

h) Save the final Q-network to a file `neural-network-1.pth`**. You can use the command** `torch.save(your neural network, 'neural-network-1.pth')`**. You can execute the command python** `DQN_check_solution.py` **to verify validity of your policy.**

The output of the file is shown in Figure 18. Our network passed the test with a confidence of 0.95% and an average reward of 242.2 +/- 25.2.

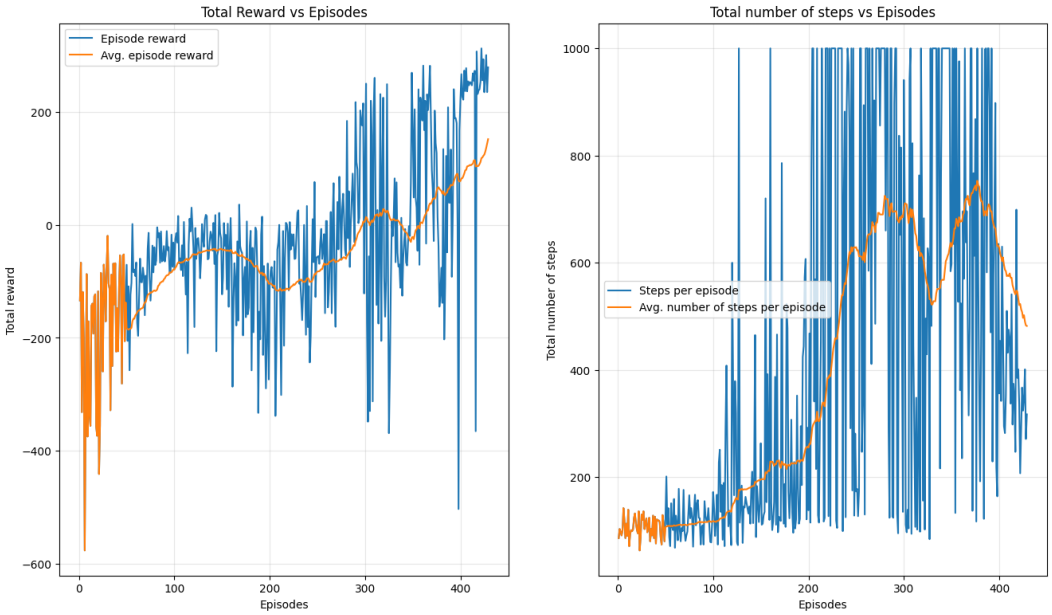

Figure 13: Training with memory size of dimension 40000

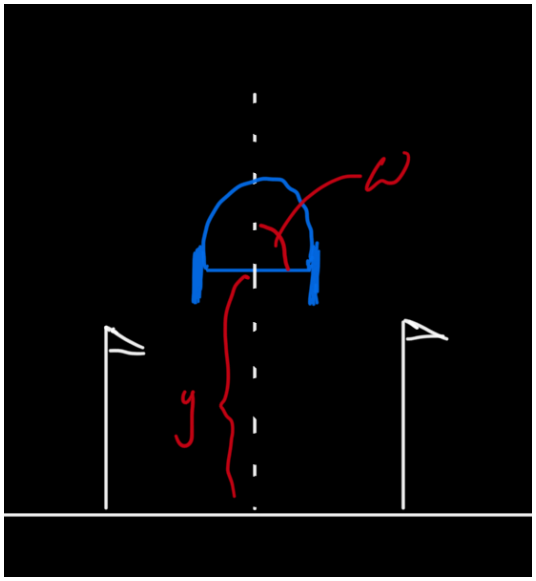

Figure 14: System with restricted state-space $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$

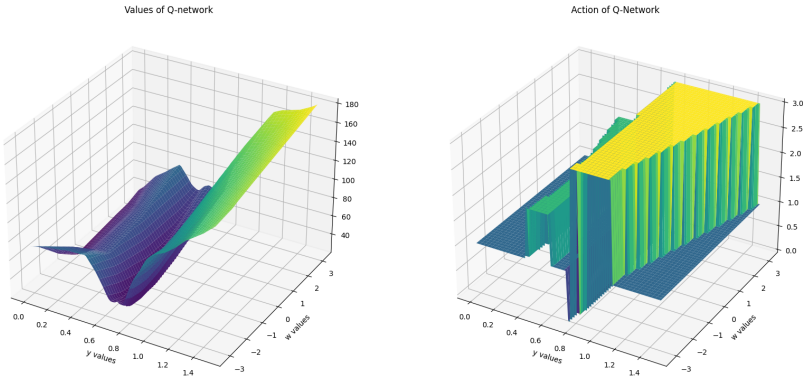

Figure 15: Q-values and corresponding actions

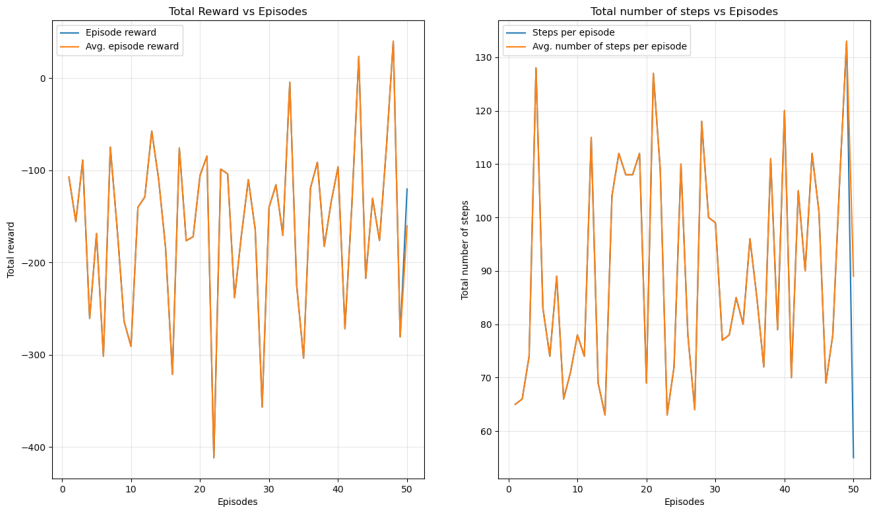

Figure 16: Random Agent over 50 episodes

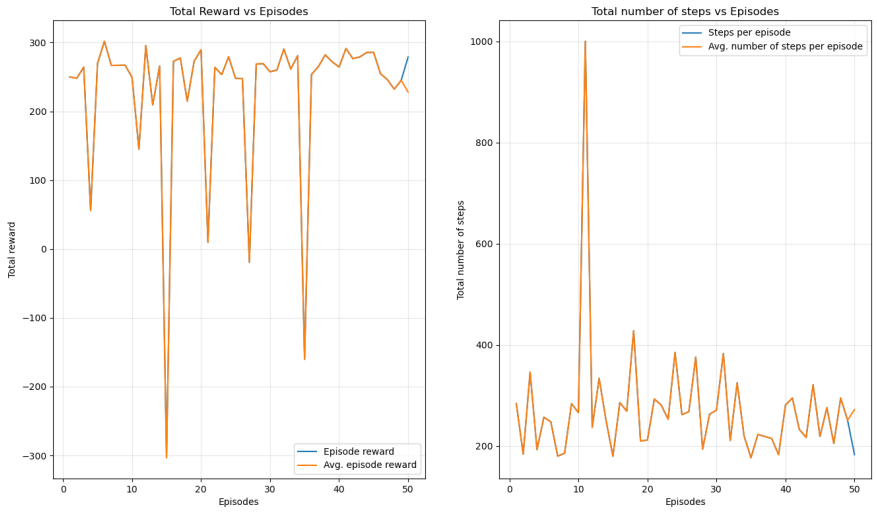

Figure 17: DQN Agent over 50 episodes

```
Episode 49: 100%| 50/50 [00:09<00:00,  5.51it/s]
Policy achieves an average total reward of 242.2 +/- 25.2 with confidence 95%.
Your policy passed the test!
```

Figure 18: Output of `DQN_check_solution.py` with `neural-network-1.pth` as input

References

[1] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., PETERSEN, S., BEATTIE, C., SADIK, A., ANTONOGLOU, I., KING, H., KUMARAN, D., WIERSTRA, D., LEGG, S., AND HASSABIS, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (Feb. 2015), 529–533.