Lab2 Reinforcement Learning

Merlijn Sevenhuijsen — merlijns@kth.se — 200104073275
Hugo Westerg ård — hugwes@kth.se — 200011289659

December 2023

Exercise 1

1a) After inspecting the random agent we see that it always crashes, which is as expected as it does not have a good strategy of trying to land the spaceship. The random agent takes a random integer which corresponds to one of the actions. Within the simulations this does not lead to any good results. As this policy is stationary (always the same parameters) it does not improve over time, and on average it has the same reward each time.

1b) A replay buffer is used to make sure that each training sample is independent of one another. If we do not use a replay buffer, then the samples trained on are dependent on each other. As one state and action depend on the previous one which can potentially be learned on. By having a replay buffer, we add each sample of state, action, reward, next state and done randomly, and lowering the dependence on the previous state. This means that during training the order is not important anymore, which otherwise could be picked up by the model. Moreover, you can get a batch of data at the same time to use for learning.

A target network is used to lower problems that occur when learning a policy whilst also using it. First you test out the policy for a few iterations and then improve it, instead of improving it for each sample. We continuously update parameters and have one main set of parameters that is updated after every set number of updates according to the updated parameters. This leads to a bit of time where the set of parameters can be tested out and updated according to the losses, reducing the effect of oscillations of specific samples.

1c) Solve the problem. Please see the files problem_1.py and DQN_NN.py. For this we have used none of the DQN modifications. The file is saved in neural-network-1.pth.

1d) Network used: For the network we have looked at different layouts. Firstly, we started with two layers of 128 parameters. Once we saw that this performed good, we started reducing the size as the model does not need to be this large. This led to trying out layers of 64 and 32 parameters. 32 Parameters did perform well but it was less certain to converge to a stable position and would sometimes still train a lower reward. This has led to the layout:

Input layer: 8 (state) to 64 parameters

Hidden layer: 64 to 64 parameters

Output layer: 64 to 4 parameters

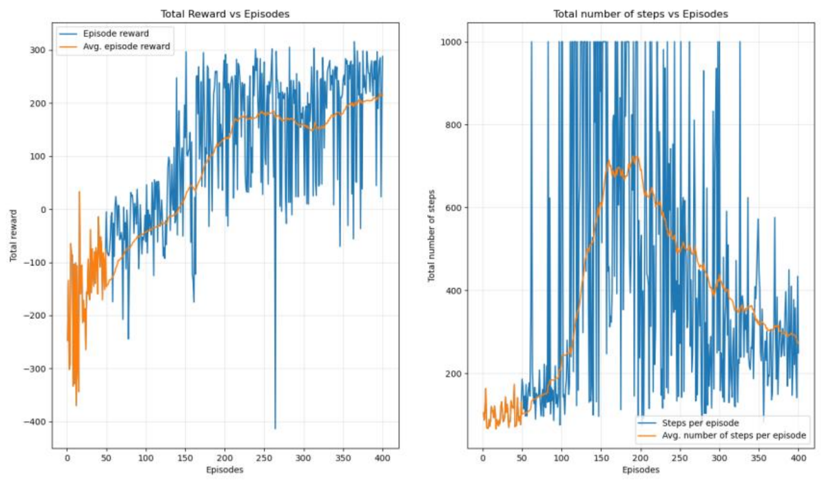
Choice of the optimizer: We have used the ADAM optimizer as it has shown that it can lead to good results for deep learning purposes. This optimizer uses stochastic gradient descent o update the weights with a fixed learning rate. The learning rate that was used is 0.0005 as this had the best tradeoff between learning the parameters and not updating too much leading to potentially missing better outcomes. This leads to updating the weights of the network in a slower way such that the changes are not too abrupt.

The following parameters were used:

- $\gamma = 0.99$. We started with a discount factor of 0.95, but as a single episode can have a thousand steps a lower discount factor did not necessarily have a positive effect on learning. Therefore, a higher discount factor has led to a better result, improving learning.
- L = 15000. We started at the maximal buffer size of 30000, and then looked at the influence of the buffer size. There was a limited amount of a decrease in performance at 15000 compared to 30000, so therefore we have decreased it to 15000 to save resources.
- Te = 400. We started with a total of 1000 episodes, but after 200 episodes we saw that the peak was already obtained, and therefore we had decreased this to 400 to still explore a bit and save the best network obtained. After 200 episodes the algorithm still explores more options to try and decrease the number of steps needed but makes sure to save the best performing model.
- C = int(L / N) = 15000/64 = 234: For this we tried out a few values as well, but eventually used $L\backslash N$ as the value for this.
- N = 64. After exploring 32, 64 and 128 as batch size numbers we have used 64 as this was a good tradeoff between running time and training samples used.
- εmin = 0.05 εmax = 0.95 using exponential decay. This was relatively good results were already quickly achieved.

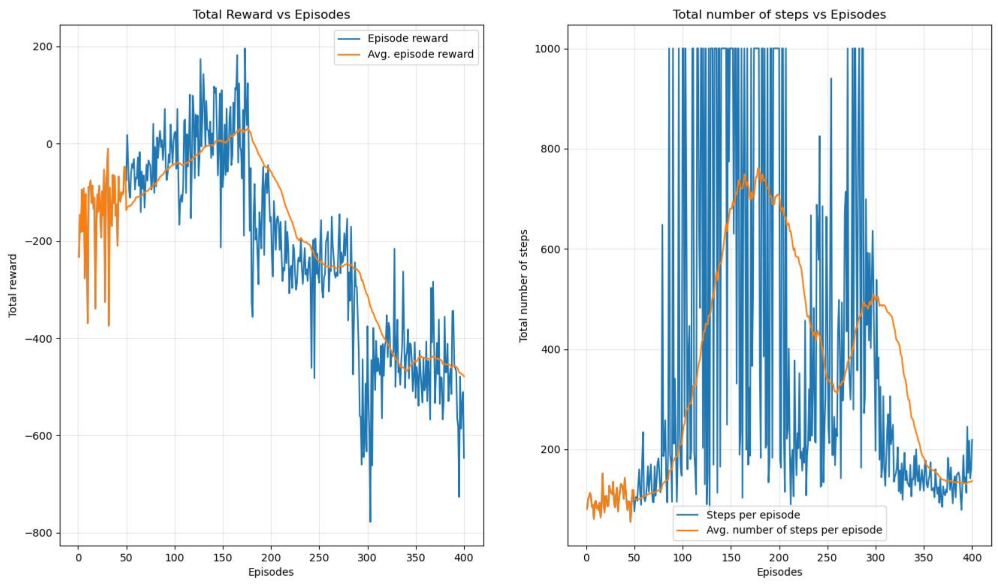
As mentioned for most of the parameters some exploration for fine tuning was done. It was mostly trial and error and seeing the effect of different parameters. The parameter that had the most effect was the discount factor. With lower discount factors the average reward was not 200, but far lower. Our search began with using large numbers, mostly everything on maximum and then after achieving a good performance trying to decrease used resources whilst still obtaining a good performance. This led to using a lower buffer size, number of episodes and batch size.

1e) 1) Plot of total episodic reward and total number of steps taken per episode during training.



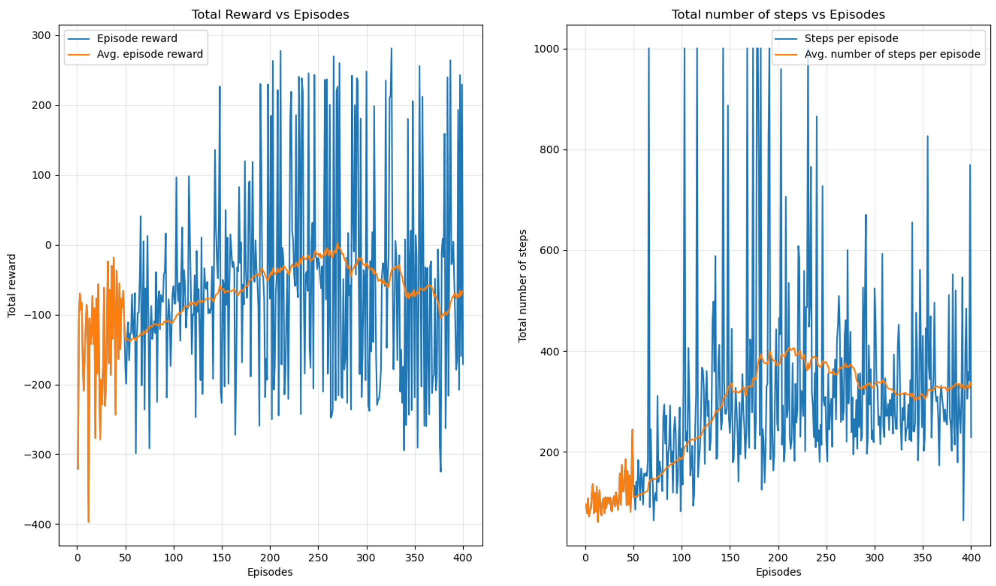
When looking at this plot we can see that a reward of 200 is obtained. And firstly, the model learns how to solve the environment using any number of steps. Once a decent amount of average reward is obtained, we can see that the model tries to do it with as few steps as possible to optimize the average reward in as little number of steps as possible. Therefore, the left graph by and large keeps going up (the average reward) whereas the number of steps taken initially increases, and then decreases due to a higher reward being possible with a lower discount factor.

1e) 2) Use discount factor γ = 1



When using a discount factor of one we see that initially the training looks similar. This is up to the point where the model already finds a good performance and reaches the limit of 1000 steps very frequently. Then it diverges whilst decreasing the number of steps and it is unable to converge to a good reward value. Thus a discount factor of one does not lead to good results.

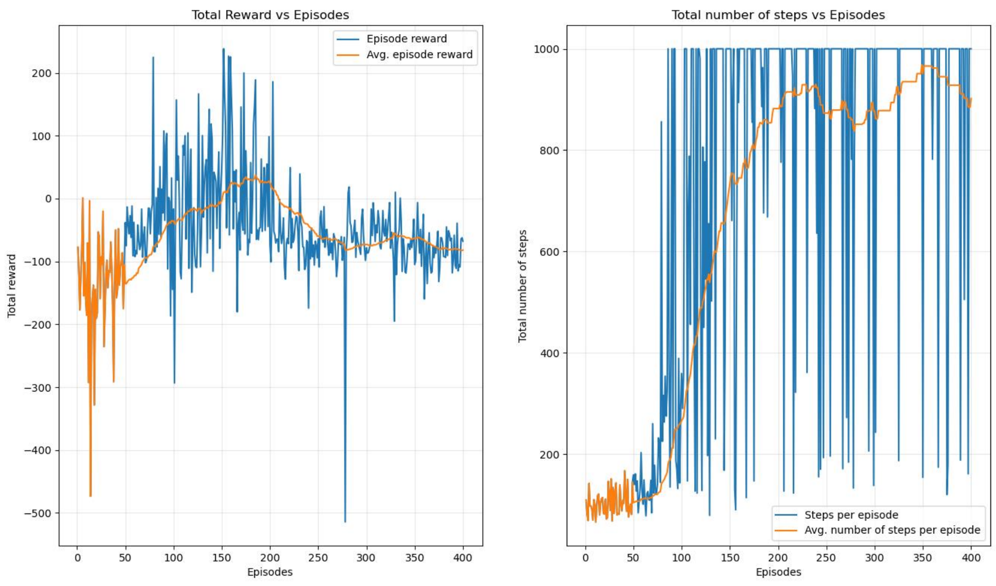
Use discount factor γ = 0.5



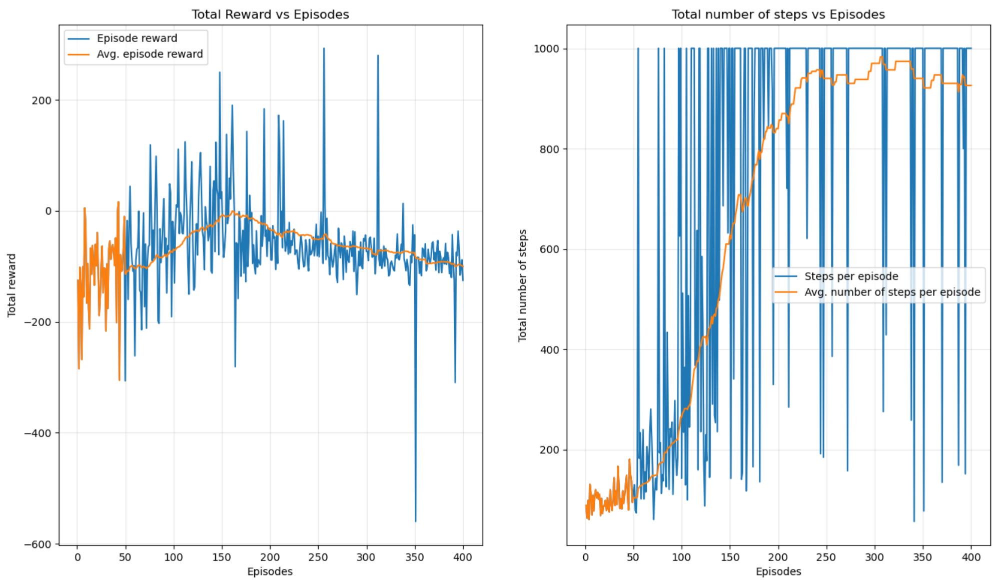
When inspecting these graphs using a discount factor of 0.5 we see that the training now increases to a reward around 0, but then diverges again as it is unable to learn a good policy. The steps are considerably less than the one that was obtained with a higher discount factor, it tries to directly learn a policy in few steps. But this is not possible if we look at the graphs. Therefore, a higher discount factor is needed than 0.5, but lower than 1. According to the tests a higher discount factor is very desirable.

1e) 3) Analyse the effect of decreasing and increasing the discount factor.

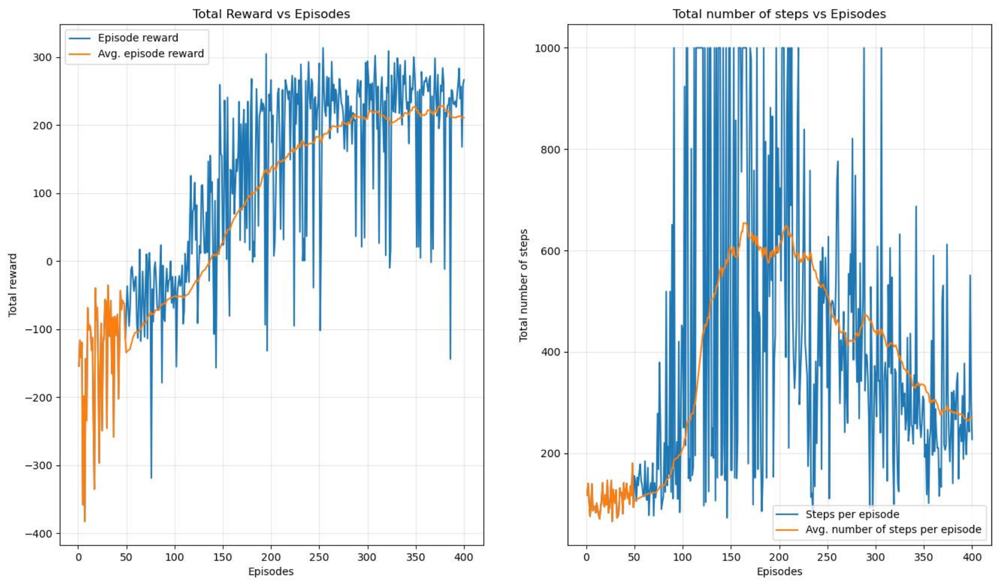
We start with a discount factor of 0.95.



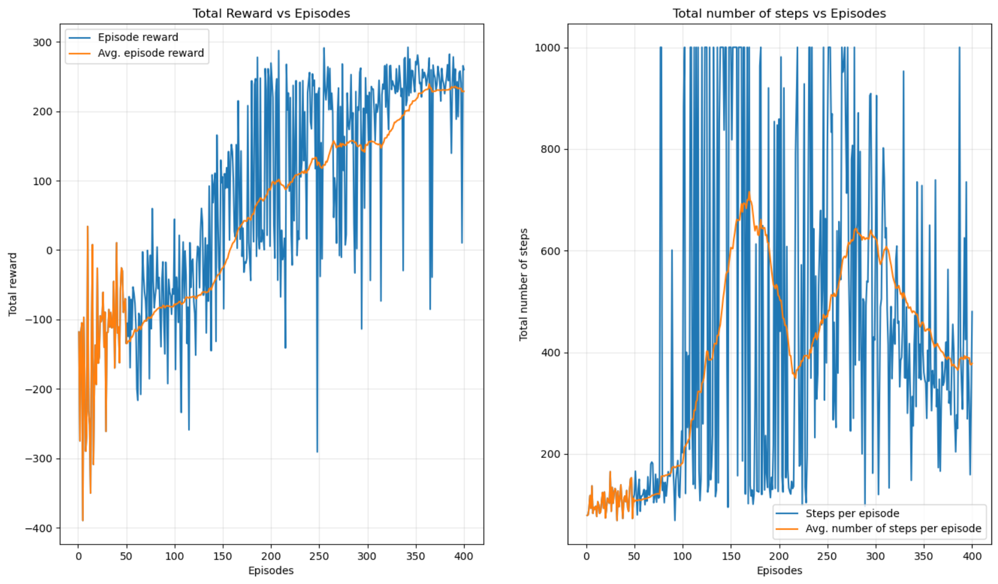
We see that for a discount factor of 0.95 learning starts out good with achieving a positive reward. However, at that point often the model reaches the maximum number of steps of 1000.After reaching the final steps very often it is unable to learn how to solve the problem with fewer steps, and the reward also decreases. The model does not pay a lot of attention to reaching a higher reward, therefore a higher discount factor could be desirable. Firstly, we will check out the discount factor of 0.9 to make sure that we indeed need a higher discount factor or a lower on.



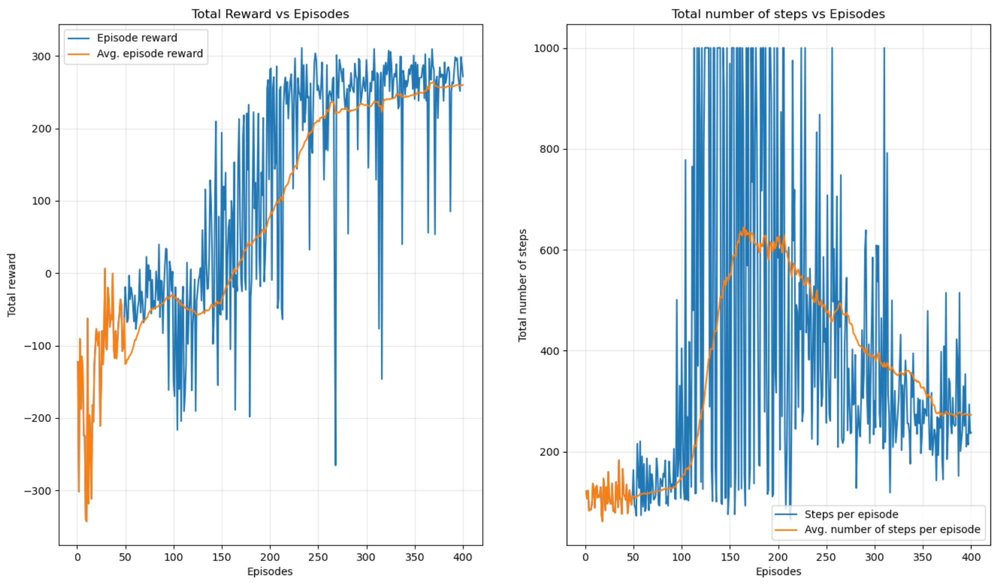
For the discount factor of 0.90 we see that it also diverges, and it is unable to find a good strategy to lands the drone in few numbers of steps. We seek a converging learning rate that is able to get a reward in a small number of steps. Next, we will try to increase the learning rate from 0.95, for example 0.96:



For the learning rate of 0.96 it performs far better than for 0.90. First it achieves a high reward and then the steps decrease over time. We will try to increase the lambda even more to 0.98:

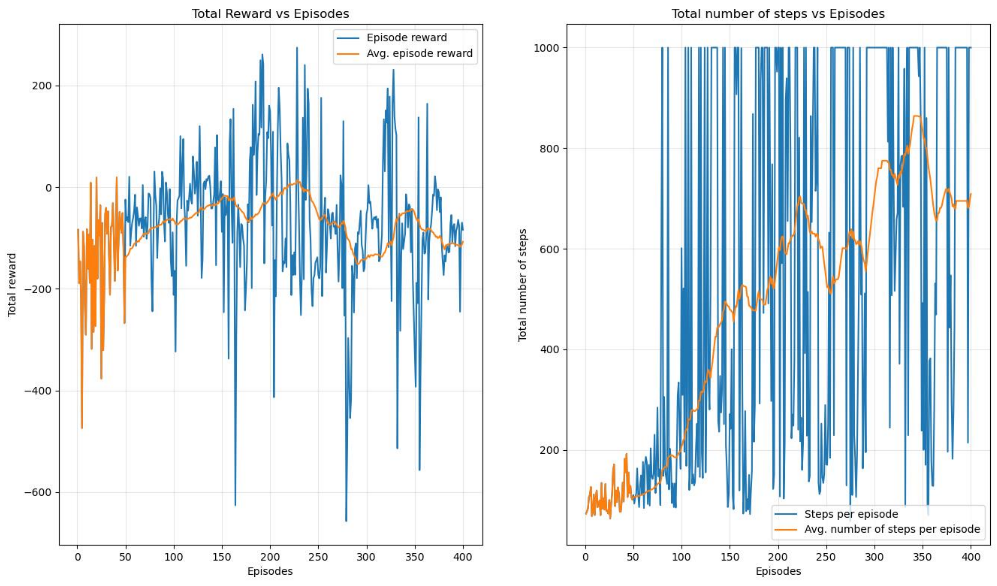


This also performs well, and we will analyze the learning rate of 0.99 as well to see the difference. We see that one time the number of steps increased again; we have two peaks in the amount of steps. Preferably we do not have this, one initial increase and one decrease whilst a steady increase in reward. We also analyse 0.99

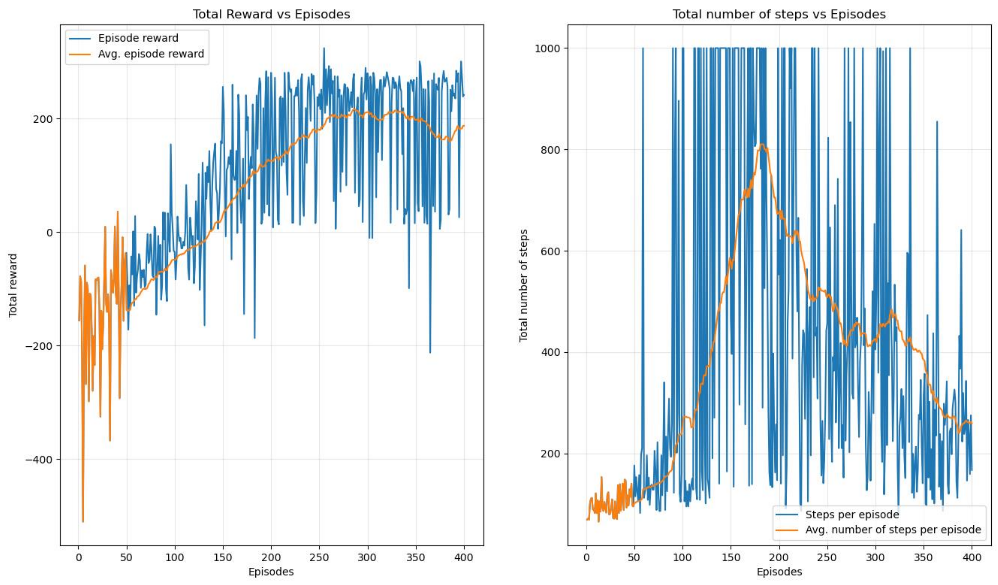


The learning rate for 0.99 also performs very well. As we have seen the learning rates from 0.96 to 0.99 perform well, where they learn the policy and decrease the number of steps to further increase the reward obtained. At 0.99 We can see almost a continuous increase in the average reward plotted against the episode, and one peak of number of steps. Therefore, this was the optimal parameter (0.99) for the learning rate.

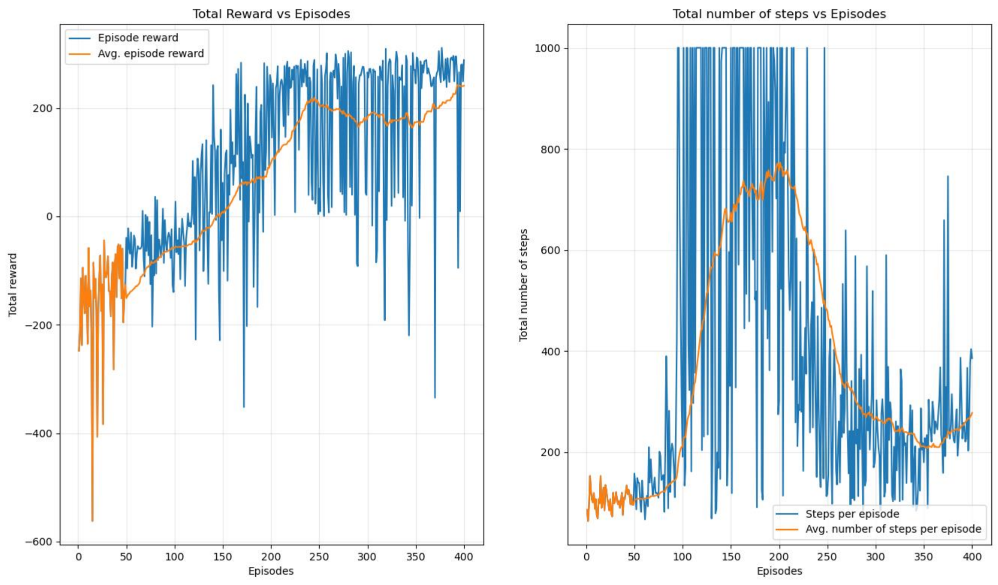
Next, we will briefly analyze the effect of the memory size. First, we will take a small buffer of 1000 and a buffer of 30000. Both use the learning rate of 0.99.



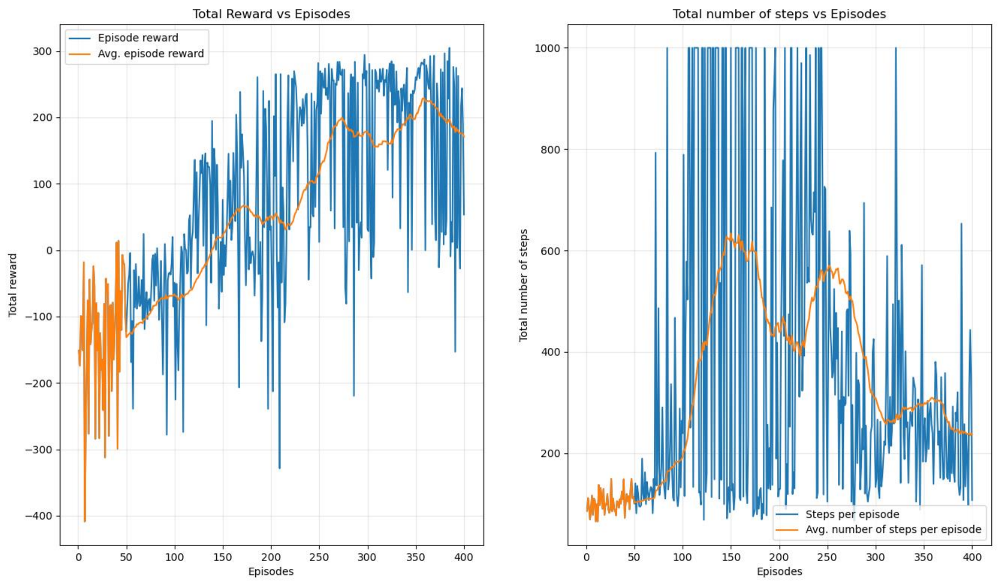
When using a small buffer of 1000 one can see (above) that the average reward does not converge. This can be due to the buffer consisting mainly of samples of the same episode, leading to no independent samples, which decreases the result. Next let us look at the large buffer size.



When using a large buffer of 30000 one can see (above) that the average reward does almost converge. Therefore, we will next try with 15000 to reduce the memory size as much as possible as the performance is already good.



We see that this indeed performs very well as well. However, it seems that at the end it tried to reduce the number of steps a bit too much so it adds a few more steps and the reward continues to rise again. We will try another smaller buffer to see if the result is better.



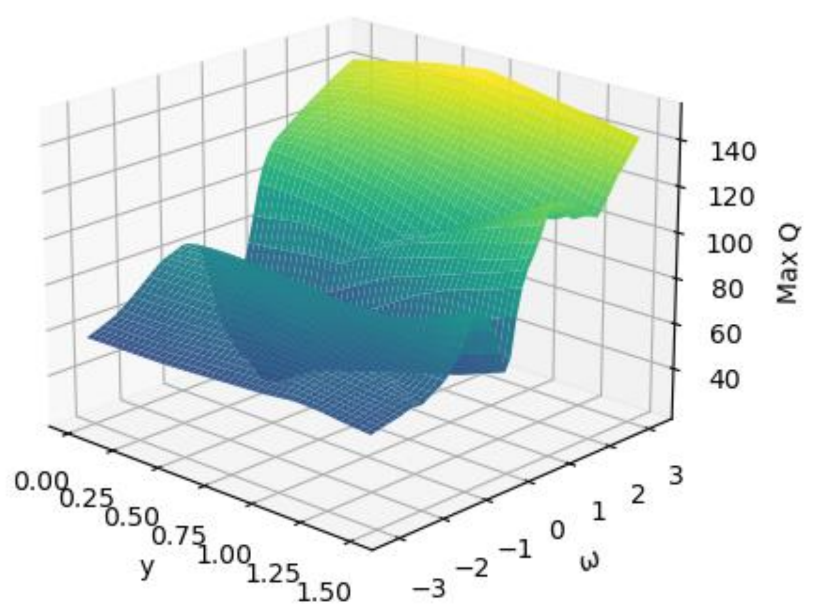
For a buffer size of 10000 we see that the number of steps is lower, but also the reward is not a steady increase, and the number of steps is not a smooth peak. Therefore, we decided to use 15000 to have a smooth converge to a result.

There is an effect between the number of steps and the memory size of the buffer. The larger the memory size is, the better the training in general as there is less dependence between the samples used for training from the buffer. However, a very large buffer size is not always desirable, as this uses up more resources.

1f) 1) Plot max$_a$ Q$_\theta$(s(y,ω),a)

Max$_a$ is the maximum Q-value indicating how good the best action is, evaluating how good the policy is.

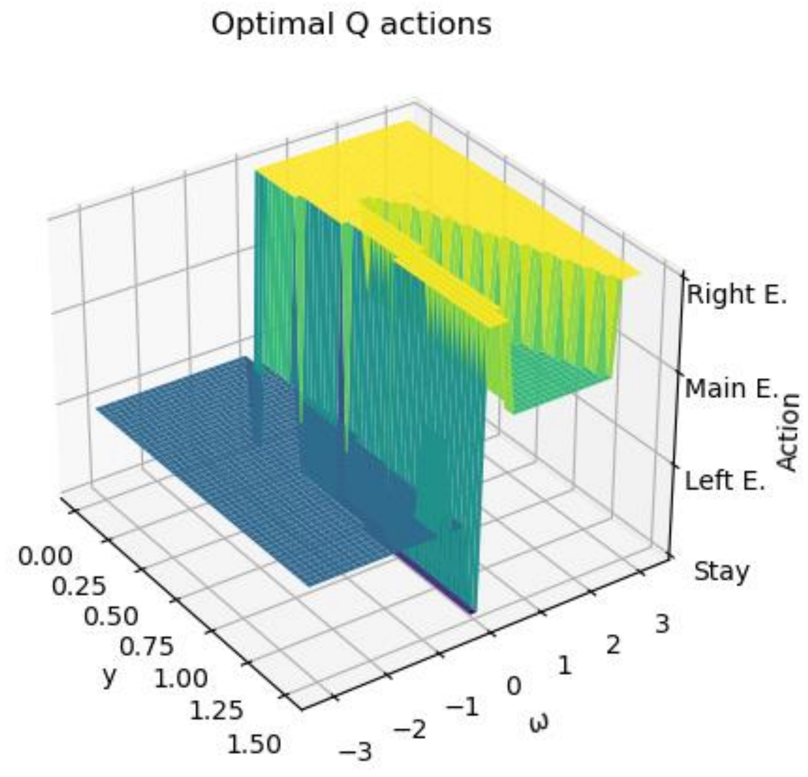
Maximum Q values



This graph shows the reward in regard to the angle and the height of the lunar lander. We can see that for high heights (y), we see less variability in the reward of the angle. This is because we have longer time to adjust the angle of the lander before we land. While as we get closer to the ground. The reward for adjusting the angle gets higher and higher. Furthermore, we see that the closer we are to angle 0, the lower the reward for the optimal action is. This is because since we are close to landing, we do not earn as much to make an adjustment as we would be if we were far from the perfect angle. When it comes to negative angles, it seems like it gets harder to land the spacecraft from this angle than a positive angle. This could be due to some kind of obstacle in the environment. The Q-network could also have learned a policy that has bias towards certain actions when the lander is tilted in a certain direction over another. If negative angles correspond to tilts that require more complex or less efficient maneuvers to correct, the Q-values might naturally be lower due to the increased difficulty in successfully landing from these states.

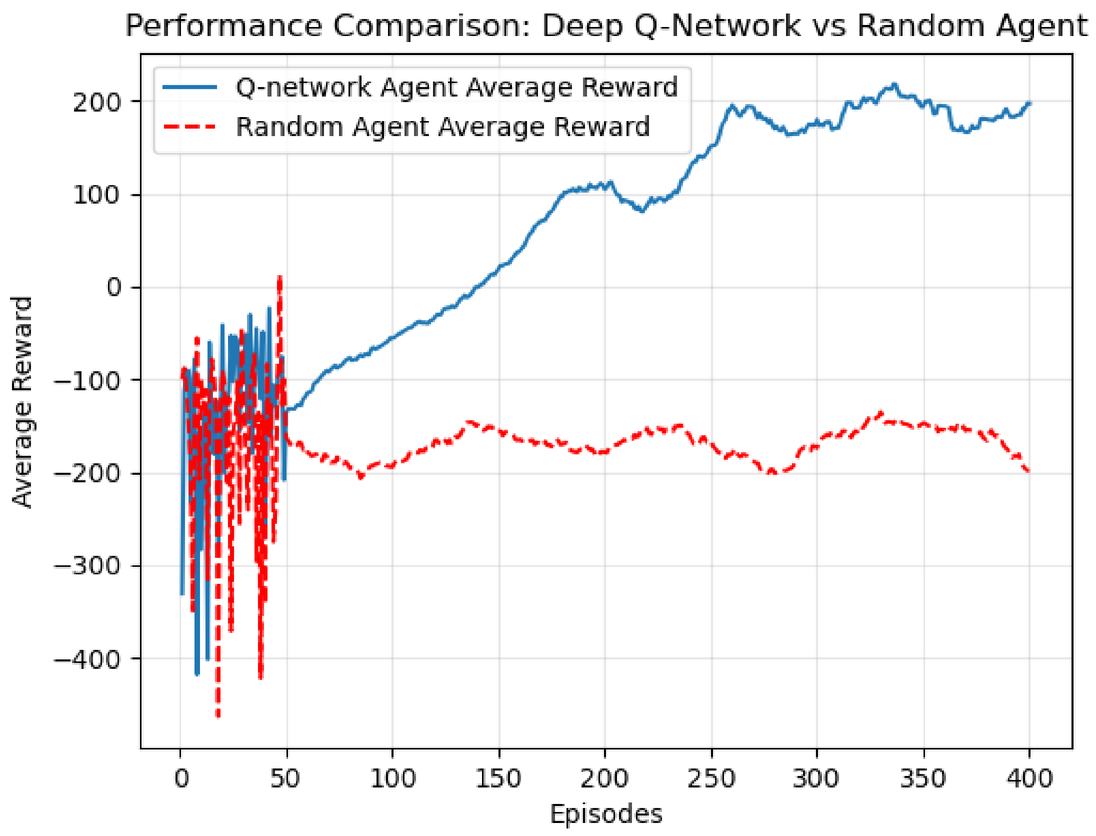
1f) 2) Plot argmax$_a$ Q(s(y,ω),a)

This task outputs the best action itself instead of how good the action is. In this way we can try to understand the model's chosen policy.



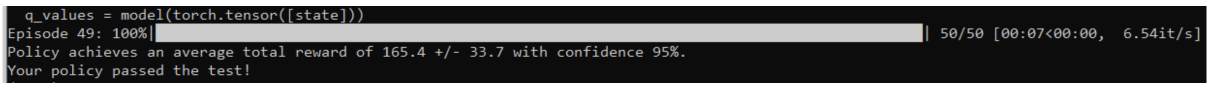
This graph also makes a lot of sense, since if the angle is too far to the right, we want to turn to the left and vice versa. And of course, if we are straight, we want to stay still or fire the main engine to reduce the speed of the lander. This is true regardless of the lander's height above the ground. Therefore, we see almost no variability due to the y-axis. When to fire the main engine is also interesting. It seems to be when we are high up from the ground and have a positive angle. This also makes sense since this is a way to move the lander sideways. So, if we are high up from the ground, we might want to move sideways to get closer to the landing platform.

1g)



As we can see, the random agent does not improve at all but rather stays around a -175 reward, while our Q-network agent improves rapidly until it plains out around a +200 reward.

1h) Our policy passed the test!



```
  q_values = model(torch.tensor([state]))
Episode 49: 100%|██████████████████████████████████████████████████| 50/50 [00:07<00:00,  6.54it/s]
Policy achieves an average total reward of 165.4 +/- 33.7 with confidence 95%.
Your policy passed the test!
```