EL2805 Reinforcement Learning

Computer Lab 2

Luís Santos lmpss@kth.se

Simon Mello smello@kth.se

December 2020

Problem 1: Deep Q-Networks (DQN)

(b)

Learning from consecutive samples is inefficient due to the strong correlation between the samples. A way to combat this is to use a a replay buffer, which stores experiences and then is randomly sampled. With a large enough replay buffer, the sampled experiences can be expected to be further apart in time and, thus, have lower correlation.

Since the DQN algorithm uses a semi-gradient method, the gradient is not taken with respect to the "target" and it is essentially being treated as constant. Therefore, it makes sense to fix it as a stationary target, having its own network, and only updating it periodically. Furthermore, aiming for a stationary target is intuitively easier than aiming for a moving one.

(d)

Hyperparameters were chosen based on the "Tips and tricks" section of the instructions as a baseline, and further tuning was then done based on testing and theory.

For our DQN implementation we opted for a feedforward neural network with two hidden layers, where each hidden layer contains 64 neurons. The input

layer has 8 neurons, one for each continuous dimension of the state space, whereas the output layer has 4 neurons, one for each possible action in the discrete action space. While there are a a broad variety of optimizers to choose from, we went with the Adam optimizer coupled with gradient clipping of norm 1. Since Adam is an adaptive learning rate algorithm it maintains a learning rate per parameter and tends to work better with sparse gradients.

A smaller learning rate means taking smaller steps and should, generally, give more reliable training; however, optimizing takes a longer time. Striking a balance is important, and thus a learning rate of 0.0005 was chosen initially as the mid-point of the suggested range and gave good results, so it remained unchanged.

A larger buffer size should theoretically be an improvement, as having more experiences to sample will lead to lower correlation and more stable training. However, it can lead to slower training and might require being run for more episodes. Ultimately, we found that a buffer as large as possible was better as long as we also increased the amount of episodes. The largest buffer size suggest, 30000, proved to work well for the amount of episodes we tested, so we chose that value.

While we did see the network started to regress when run for a large amount of episodes, it would usually recover within a 100 additional episodes. This pattern was observed mainly in episodes beyond 500. So, while we were able to attain better models if run for 700-1000 episodes (especially with increasing the buffer size and amount of neurons simultaneously), the gains beyond 400 episodes were minimal. Given that the number of episodes was the biggest factor in training time, we decided to keep it at 400. However, note that training the network multiple times with the same parameters had lower variance with more episodes.

A good rule of thumb for the update frequency is $C \approx L/N$, where L is the buffer size and N is the batch size, so that is the formula we went with.

Computing the gradient separately for each experience in the entire replay buffer and averaging can take a long time, and thus slows down learning. Instead, we can estimate the true gradient by computing the gradient of a smaller, randomly sampled mini-batch from the replay buffer and then aver-

aging. If the mini-batch is sufficiently large, we can expect the average to be a reasonably good estimate of the average over the entire training data. The smaller the mini-batch is, the faster the computation but with worse accuracy. Since we only care about moving in a general direction in each step, we can get away with quite a small batch size, speeding up training quite a bit and giving better models in the end. While a batch size of 128 is on the upper end of the suggested range, we found that it worked better than lower batch sizes and was small enough to give good training speed nonetheless.

The criterion for convergence is that our agent visits every state-action pair infinitely often. This requires our agent to have a penchant for exploration. However, we also want our agent to visit state-action pairs that we know give good rewards. This dilemma, commonly known as the exploration/exploitation trade-off, is best dealt with by using an ϵ-greedy policy with a decaying epsilon. This ensure that our agent is exploring a lot early on and focusing on collecting good rewards later on. We tried both linear and exponential decay and found that linear decay gave better results, with parameters: $\epsilon_{max} = 0.99$; $\epsilon_{min} = 0.05$; and $Z = 0.9N$.

(e)

In Figure 1, we can see that the network is training slowly at first but sees a big spike in average reward around 250 episodes. If we look at the right graph, we are observing that the agent is taking more and more steps, indicating that it is learning not to crash. After around 250 episodes, we see that the amount of steps drastically goes down and this indicates the agent is learning to also land, instead of just trying not to crash. This drop coincides with the spike in average episode reward, which makes sense. In conclusion, the agent first learns not to crash and only after that starts learning how and where to land. The training slows down after around 350 episodes, and while it is still possible for the agent to perform better it is very minor and requires more tuning, as well as taking much longer to train.

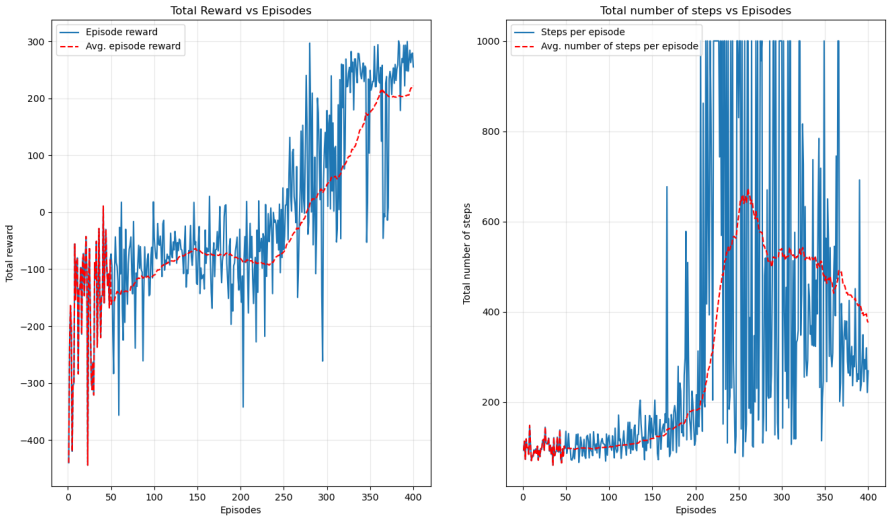

Figure 1: Total episodic reward and number of steps taken per episode during training for base parameters.

The discount factor, γ_0, used for the base network worked well. If we set the discount factor to $\gamma_1 = 1$ and $\gamma_2 = 0.05$ we obtain Figures 2 and 3, respectively. For γ_1, we see that the network is not learning. We want to treat this problem as an infinite horizon discounted MDP, however, with $\gamma_1 = 1$ there is no discount factor and we have an infinite horizon sum, which does not guarantee convergence to an optimal policy and is a poor optimization criteria. If we have infinite rewards, a poor policy might be viewed as equal to a good or optimal one. This can be seen in Figure 2, where this time the drop in steps per episode is due to crashing and not landing, accumulating very poor rewards. On the other hand, a discount factor which is very low, such as $\gamma_2 = 0.05$, discounts future rewards too heavily, and thus will only care about immediate rewards. This agrees with the results from Figure 3, where the entire training process looks similar to the initial training of the base network; we also observe that there is no big drop in steps per episode, indicating that the agent is neither crashing nor landing as much as the two other agents.

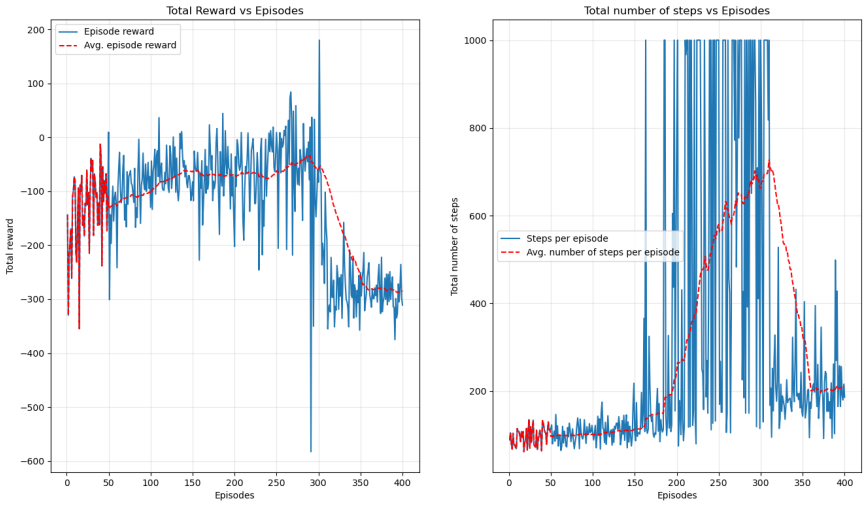

Figure 2: Total episodic reward and number of steps taken per episode during training for $\gamma_1 = 1$.

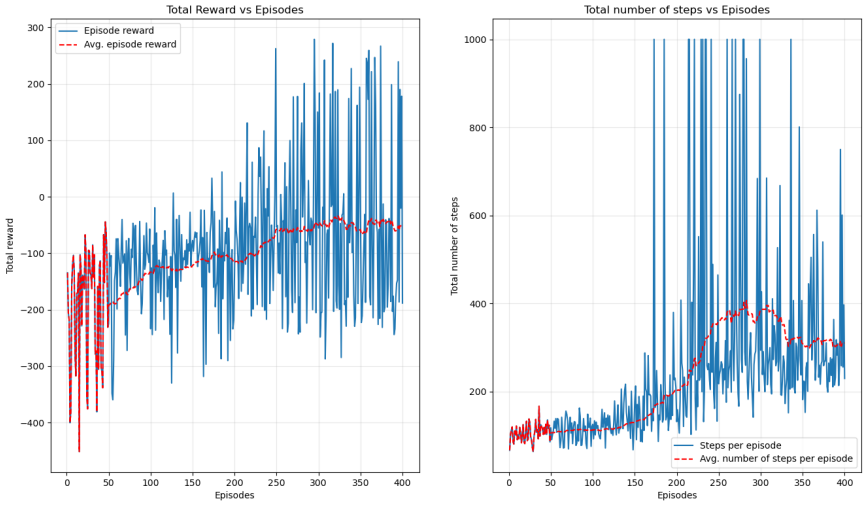

Figure 3: Total episodic reward and number of steps taken per episode during training for $\gamma_2 = 0.05$.

If we continue with the base parameters and instead alter the number of episodes, we see that in Figure 4 the network has not yet stopped learning and more episodes could be beneficial. If we increase the number of episodes to 400, which is our base network, we do see that it continues to learn but once again, it seems like it can still learn more. In Figures 5 and 6, we see that 500 episodes did not increase rewards and 600 episodes even started regressing. There is still a possibility to recover and learn more, so we also looked at 700 and at 1000 episodes in Figures 7 and 8, respectively. 700 episodes shows even more regression, a regression we see in the graph for 1000 episodes, as well. However, eventually the network starts learning again and shows a steady rise up to 1000 episodes. This is with the hyperparameters tuned for 400 episodes and it is possible for the network to start "forgetting" as the replay buffer is being filled with new experiences. When run for 1000 episodes, we can observe in the right-hand graph that the number of steps per episode goes up and down multiple times, indicating that the network is forgetting and re-learning better policies.

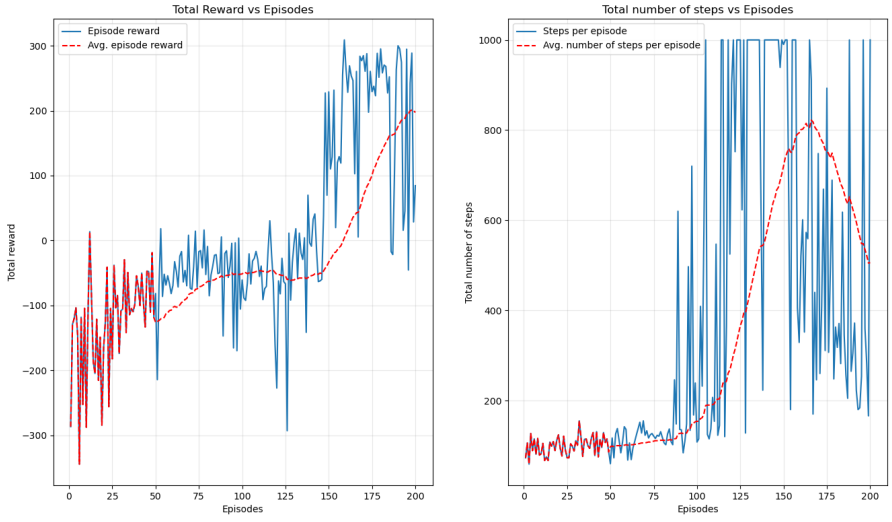

Figure 4: Total episodic reward and number of steps taken per episode during training for 200 episodes.

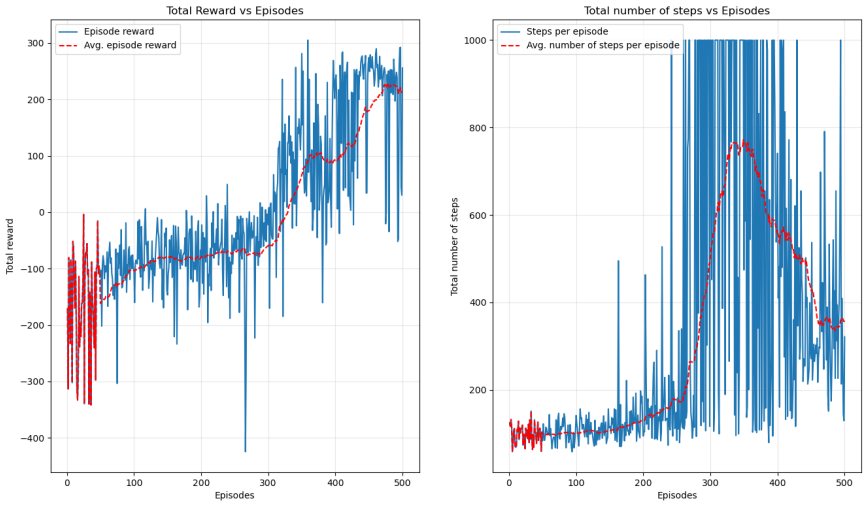

Figure 5: Total episodic reward and number of steps taken per episode during training for 500 episodes.

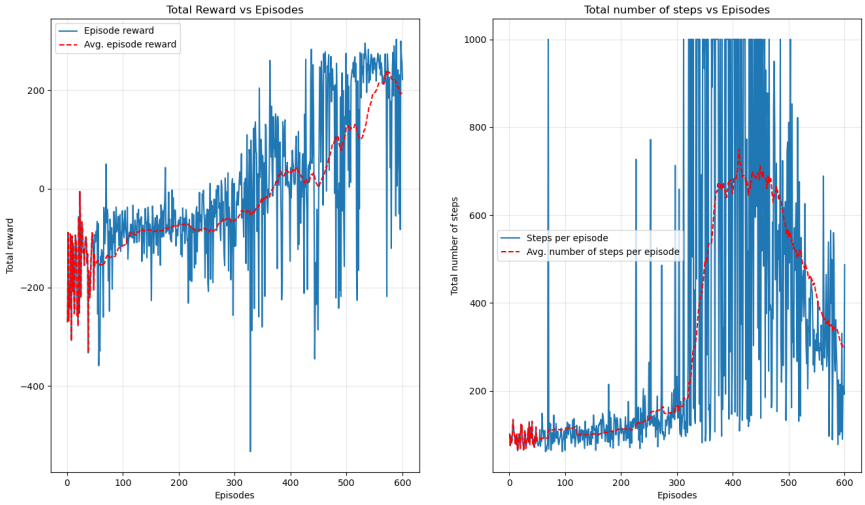

Figure 6: Total episodic reward and number of steps taken per episode during training for 600 episodes.

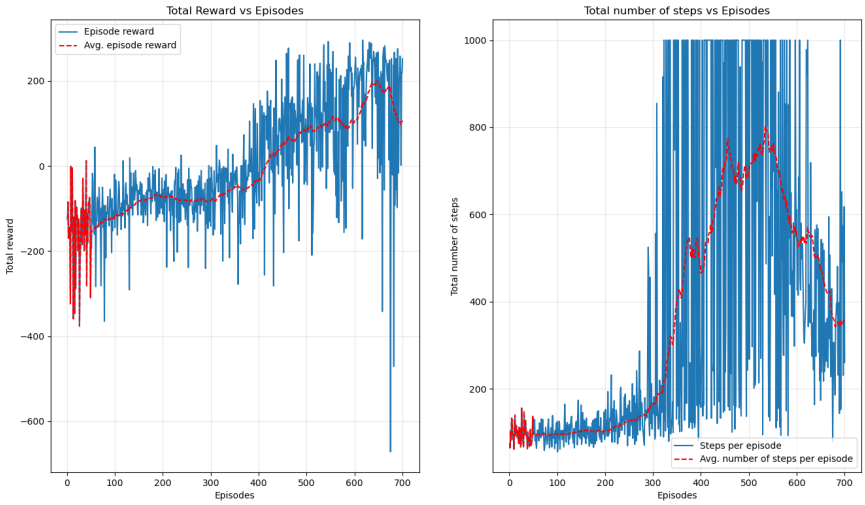

Figure 7: Total episodic reward and number of steps taken per episode during training for 700 episodes.

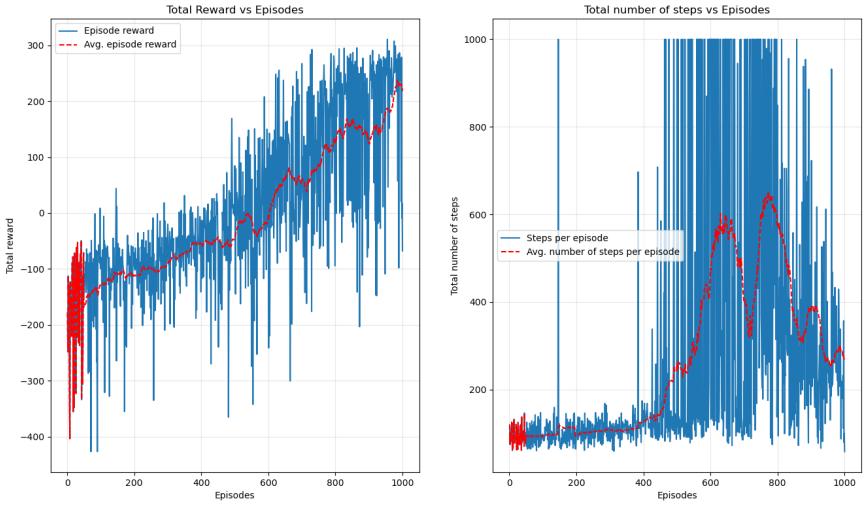

Figure 8: Total episodic reward and number of steps taken per episode during training for 1000 episodes.

It is important to note there is a decent amount of variance when training the networks even with the same parameters, and as such, it is not possible to say with certainty whether 400 episodes is better than 600 just from these graphs alone. However, the peak performance of any network, no matter the parameters, was never better than our base network; although some larger networks, with larger buffers, trained for more episodes these parameters seemed to be more consistent.

As hinted at before, the size of the replay buffer plays a role in how "forgetful" the network is and how stable the training is. In Figure 9, we see that the network has a hard time learning anything outside of an immature policy, only learning not to immediately crash. In the right-hand graph we see that it is hinting at possibly learning to land but it quickly goes back to learning not to crash: the replay buffer simply is not large enough to train a well-rounded agent.

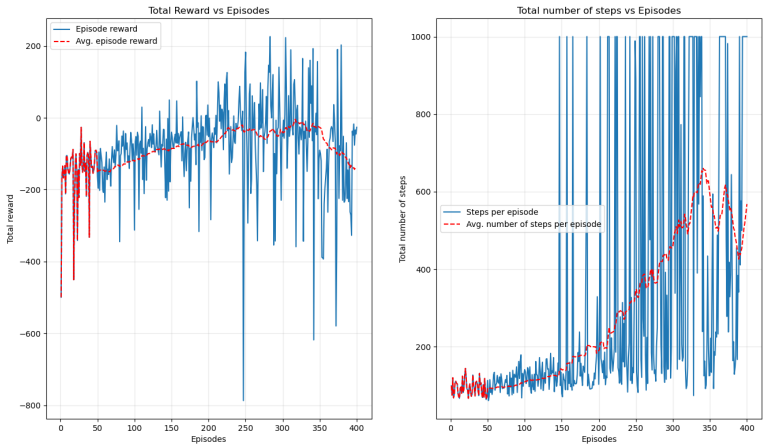

Figure 9: Total episodic reward and number of steps taken per episode during training for $L = 1000$.

In Figure 10, we see that the very large increase in buffer size does not lead to more rewards. In our previous section where we motivated our hyperparameter choices, we said that the larger the buffer the better, with the drawback being slower learning. This seems to be the case here, as the network is slowly but surely learning without regressing, but it just does not

have enough episodes to train and take advantage of its large memory.

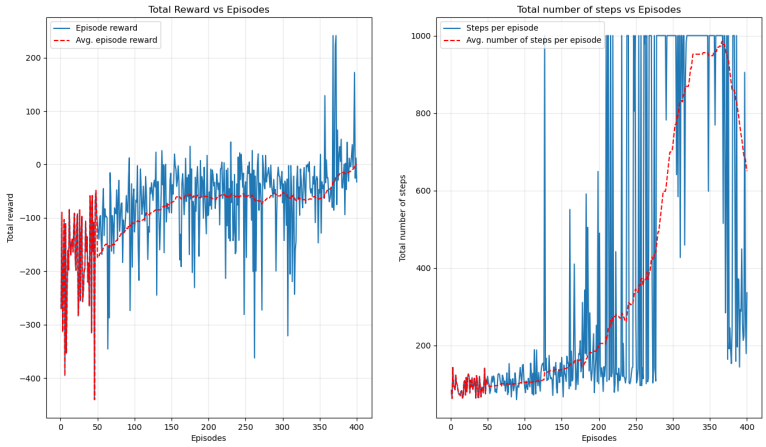

Figure 10: Total episodic reward and number of steps taken per episode during training for $L = 100000$.

(f)

The restriction $s(y, \omega) = (0, y, 0, 0, w, 0, 0, 0)$ gives the states where the lunar lander is right above the goal in any orientation, with no velocities. In Figure 11, we see how the maximum Q-value changes with parameters y and ω. We should expect to see symmetry about the ω-axis, however, instead we are seeing that one increases and the other decreases. In the paper published by DeepMind on Playing Atari with Deep Reinforcement Learning (Mnih et al., 2013), the authors mention that a potential problem is that if a maximizing action is to move left, then the training samples will be dominated by samples from the left-hand side. In our case, we should expect the distribution of firing the left and right engine to be roughly equal; however, due to this aforementioned problem, it might be the case that our agent is biased towards one side, and thus does not explore enough to choose both actions equally. Therefore, it rarely explores the extremes of one side. A replay buffer should remedy this problem, and for some networks we do see that it is equally distributed, but in this instance the agent is able to learn a very good policy, nonetheless. As mentioned previously, this was the network that accrued the most reward.

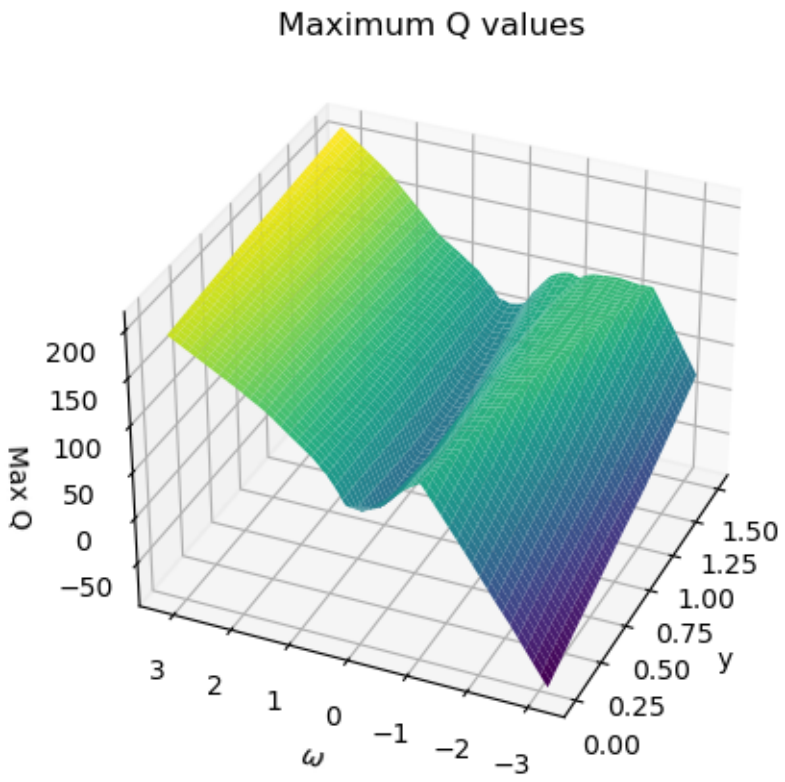

Figure 11: Max Q-values for a restricted state space.

In Figure 12, we see that the graph for the optimal policy is much more intuitive than the previous graph. For the limited values where $\omega = 0$ and its neighbourhood, the best action is to stay, which makes sense. Firing the main engine is rarely useful since it would just misalign the lander with the goal by gaining momentum in the x-direction, or incur unnecessary costs when the lander is already perfectly aligned. Even though the network is biased towards one side, as seen in Figure 11, it still manages to distinguish them policy-wise.

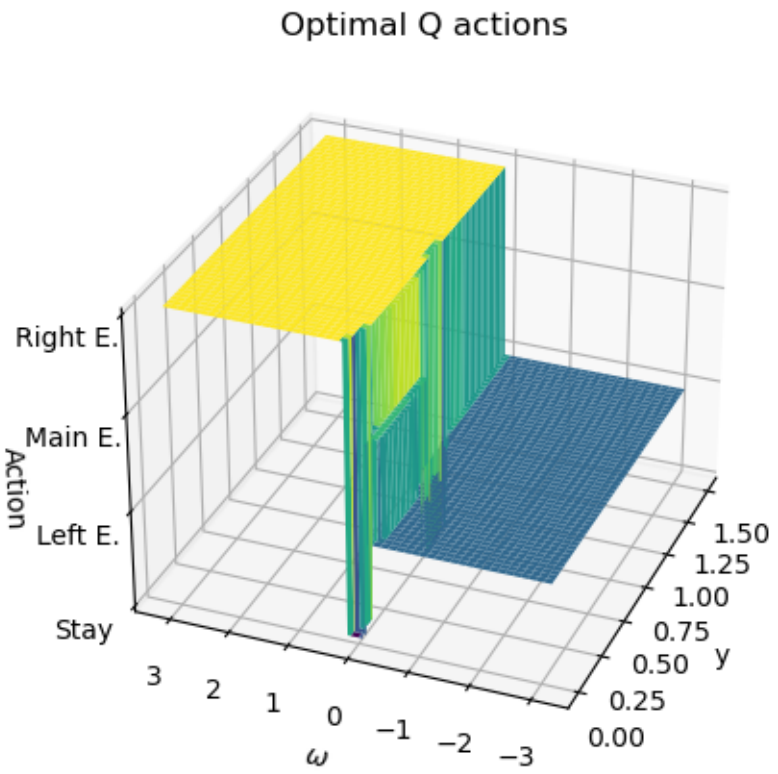

Figure 12: Optimal policy for a restricted state space.

(g)

In Figure 13, we observe that, as expected, the random agent never achieves higher average rewards as it is not learning a better policy. Our Q-network agent, however, quickly diverges as it learns better and better policies.

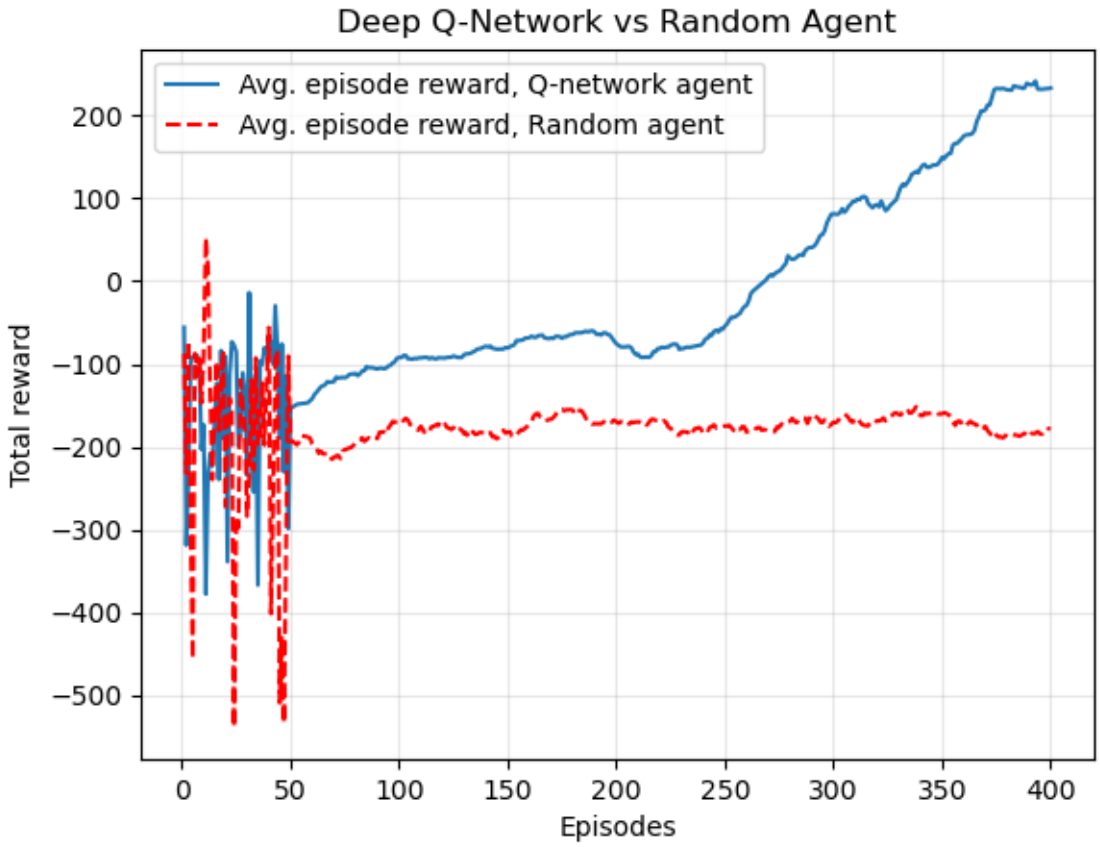

Figure 13: Total episodic reward averaged over last 50 episodes.

References

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.