Reinforcement Learning: Lab 2

Martin Schuck, Damian Valle
mschuck@kth.se, damianvs@kth.se

1 Problem 1: The Lunar Lander

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

Since the environment state space is large, the agent needs to learn from a collection of samples that properly represents the state space.

The target network is used to have a slow moving target for the SGD algorithm in order to improve stability.

1.2 Network layout, optimizer and hyperparameters

The network layout is a dueling network with 128 nodes in the first layer and then a split into a scalar representing the value of the input state and a second branch for the action values.

- Adam optimizer
- Number of episodes: 600
- Discount factor (γ): 0.99
- Learning rate (α): 0.001
- Clipping value: 1.3
- Replay buffer size: 30000
- Batch size: 64

As for modifications, we implemented CER (Combined Experience Replay), Dueling DQN, Double DQN and exponential decay for the ϵ parameter.

1.3 Training process

Figure 1 shows the total episodic reward for the chosen hyperparameters. We can see that as the algorithm learns, the reward consistently increases. Regarding the steps per episode; at first the agent crashes very early which means very few steps, then the agent learns to hover for the maximum amount of time (1000 steps) and lastly begins exploring how to land which decreases the amount of steps taken per episode.

Our chosen discount factor was $\gamma = 0.99$, now we will analyze it's effect on the training process by changing it to extreme values. Figure 2 shows how a $\gamma = 1$ discount factor training process looks like. Intuitively, an agent that has a discount factor equal to one values every single reward the same without taking into account how long into the future such reward might come. With a discount factor of 1, the algorithm essentially becomes a Monte Carlo learning procedure. Since our state space is huge, the agent fails to learn anything with this approach. On the other hand, a lower discount factor favours rewards that come early instead of long term ones. The agent therefore rushes towards the landing zone without any control, since it does not account for any long term rewards and tries to minimize its losses in a short period of time.

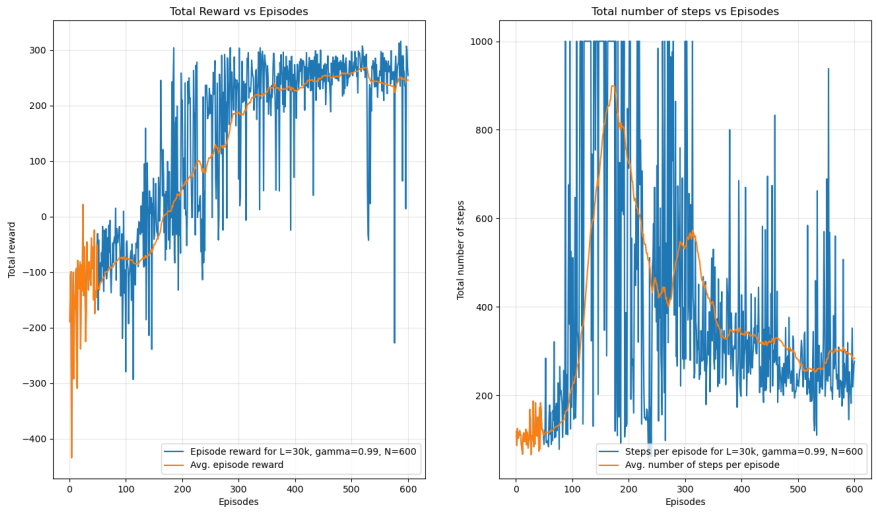

Figure 1: Total episodic reward (left) and steps per episode (right) in the training process.

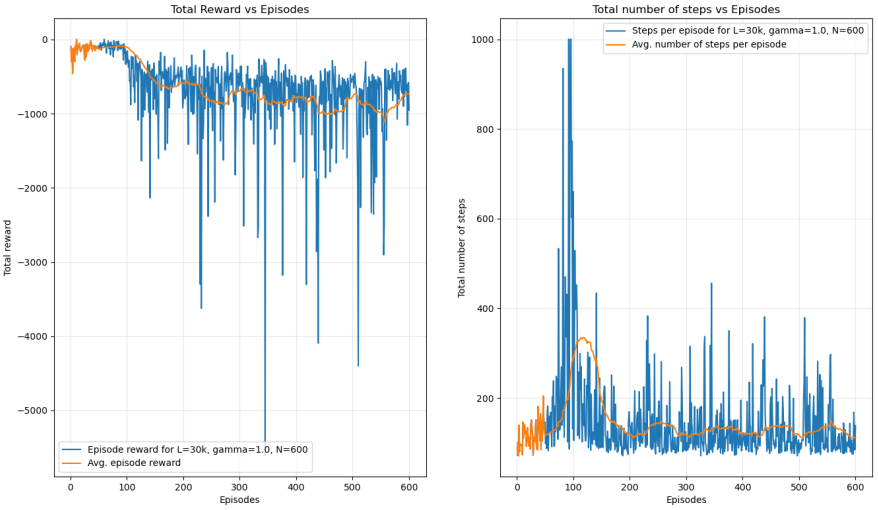

Figure 2: Training process with $\gamma = 1$.

Fixing all the parameters and changing the number of episodes we obtain Figure 4 and Figure 5. We see that for 1000 episodes the agent converged and was able to stay collecting high rewards. For 200 episodes the agent has clearly nor converged yet as it stopped before getting 100 reward.

If we now change the memory size we obtain Figure 6 and Figure 7. We see that if the buffer is too big (300k) then the algorithm takes longer to learn since it's looking at samples from early in the training when it it still had a bad policy. Conversely, if we use a buffer that is too small (3k), the agent at one point unlearned because of catastrophic interference, that is, after the agent chose a bad policy it was only able to learn from samples generated under such policy hence worsening performance. This is most probably because the state space was misrepresented by consecutive batches that only showed a fraction of the covered space.

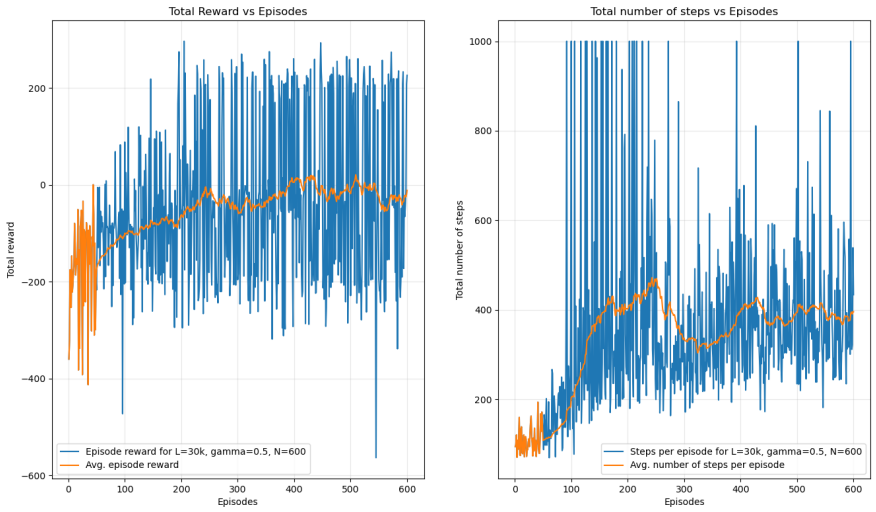

Figure 3: Training process with $\gamma = 0.5$.

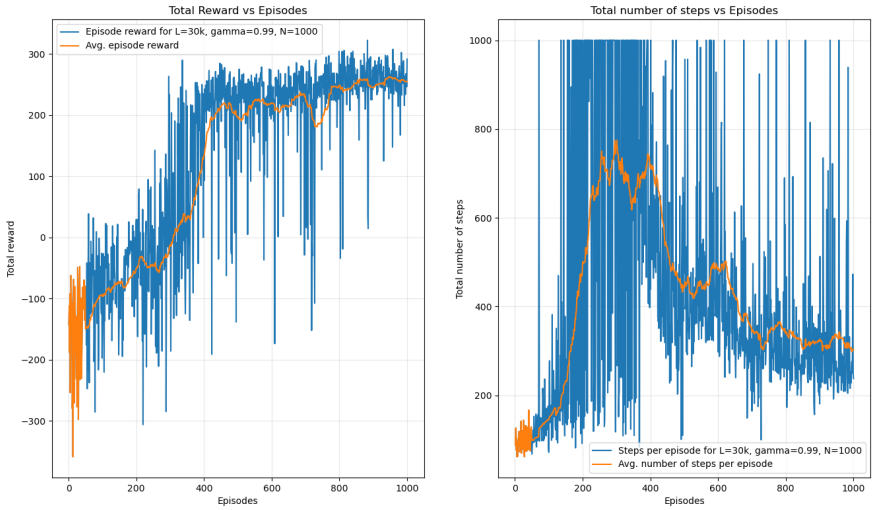

Figure 4: Training process with 1000 episodes.

1.4 Q-network Q_θ

When restricting the state such that the spacecraft is strictly above the landing spot, without linear or angular velocity and without having already landed, we can plot the value of each state as a function of the two degrees of freedom left: $s(y, w) = (0, y, 0, 0, w, 0, 0)$. In Figure 8 we plot $\text{argmax}_a Q_\theta((y, w), a)$. The value of the optimal policy does not make sense for many regions of the state space. We think that is due to the fact that our agent never or hardly ever visited those states. For example, values of w that are close to π or $-\pi$ mean that the aircraft is upside down. This is hardly ever the case, since the agent learns early on to avoid those cases and episodes end after drifting too far to the right/left (which is the case for repeated firings of the left/right thruster).

At first, we intuitively thought that states with $w = 0$ would have the biggest rewards since the aircraft just needs to fire the main engine and land perfectly. However, this is not the case for our optimal

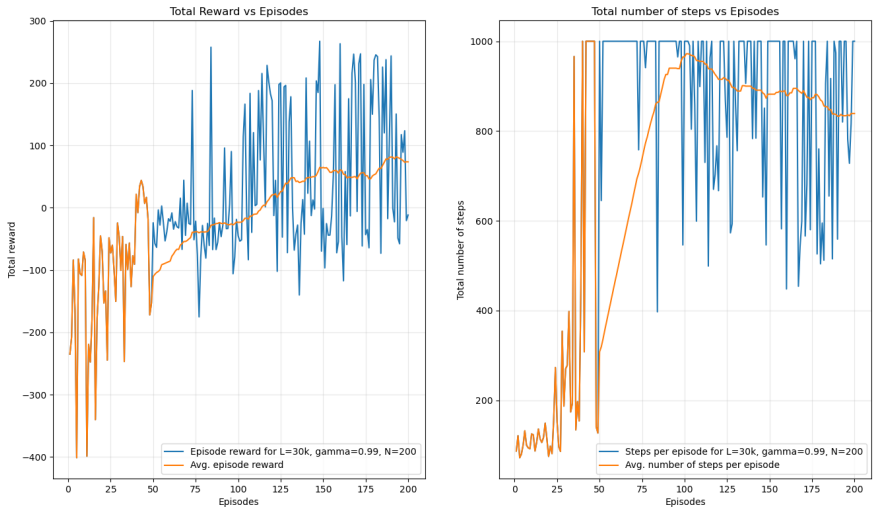

Figure 5: Training process with 200 episodes.

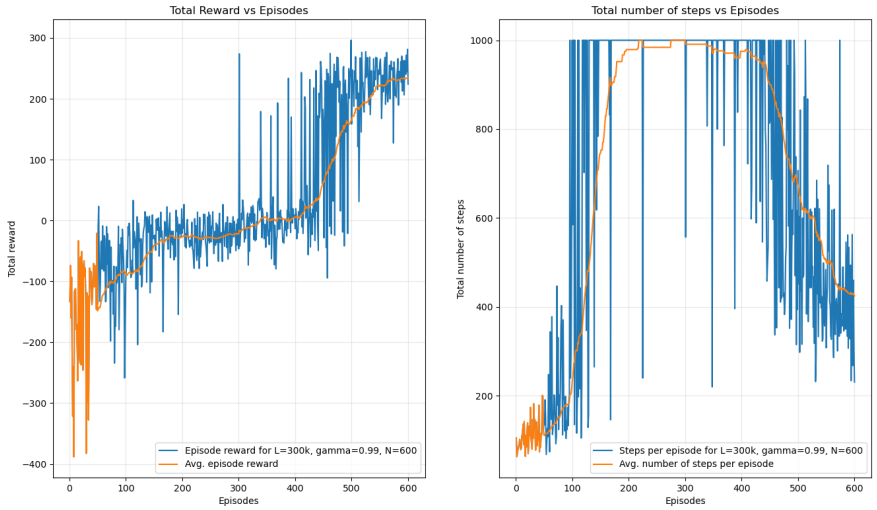

Figure 6: Training process with 300k buffer.

value function. We came to the conclusion that there are two possible explanations for this. This first one is the agent prefers to have a small angle in order to enable its more powerful main engine to control its x position. The second one is an exploration argument: The agent simply never explores these regions, since we are assuming 0 velocities for all states and x = 0. The slight preference for angled positions was confirmed by both the continuous policies as well though, so we think this might be more than just an artifact of the function approximator.

Figure 9 shows the behaviour of our trained optimal policy makes sense for values of w close to zero because it has a step function with actions left and right around the $w = 0$ line. For values of w close to π or $-\pi$ the policy does not make much sense since those states are not explored during training.

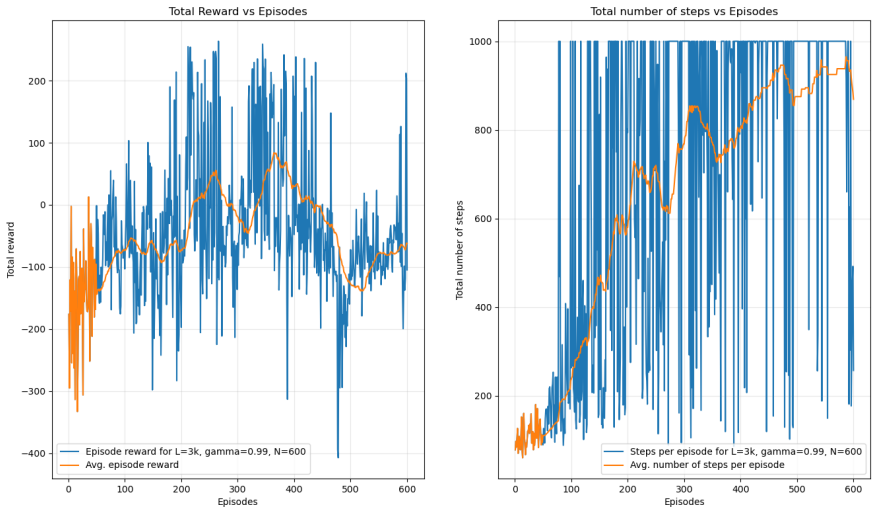

Figure 7: Training process with 3k buffer.

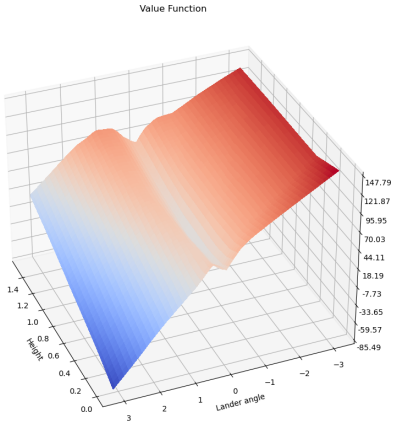

Figure 8: Value function.

1.5 Random agent vs DQN

Figure 10 shows the total episodic reward comparison between the random agent and the DQN agent we trained. It is clear that the DQN consistently outperforms the random agent.

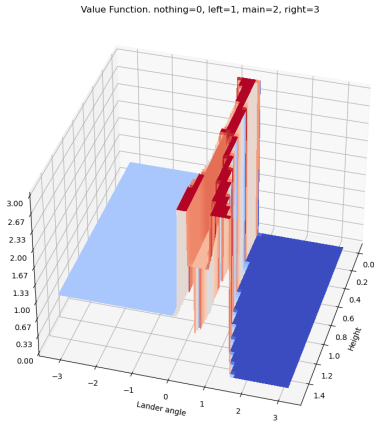

Figure 9: Optimal policy behaviour for our trained agent.

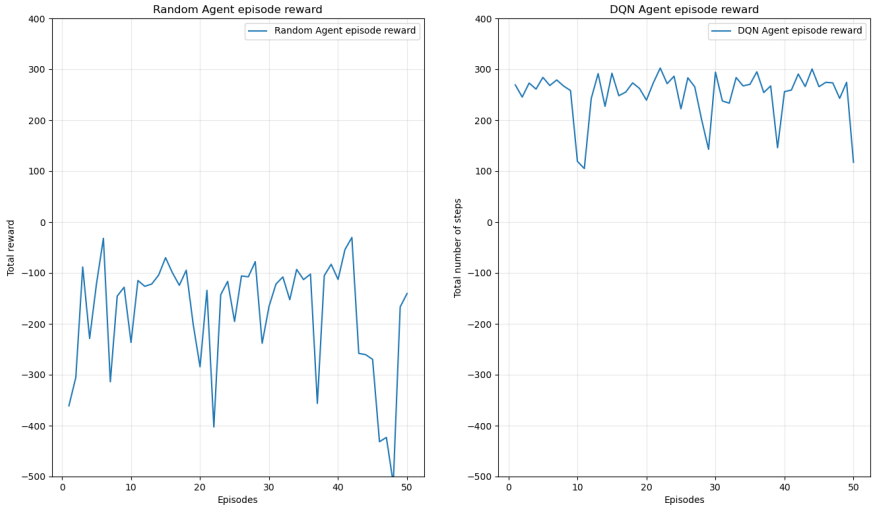

Figure 10: Random agent vs DQN: episodic rewards.

2 Problem 2: Deep Deterministic Policy Gradient (DDPG)

2.1 DDPG properties

For the default update like DQN, in order to get a smooth convergence it is useful to freeze the target and only update the Q-function for the current state. In this case, the Q-function is used to estimate the gradient of theta. Since the current Q-function is our best estimate of the real Q-function, it also results in the best estimated gradient for π_θ. Furthermore, the algorithm relies on a "nimble" Q-function estimate that quickly incorporates the shift in focus under the new policy π. Using the target Q-function would defeat the purpose behind that part of the algorithm.

DDPG uses noise on top of its policy pi to enforce exploration when choosing actions, but the estimate of the Q-function only uses π for its policy, therefore the policies differ making DDPG an off-policy algorithm.

Regarding sample complexity, off-policy algorithms perform better since storing samples allows them to reuse and learn from a sample multiple times.

2.2 Network layout

The continuous action space of the task is considerably more complex. Therefore we use networks with a higher amount of nodes. The exact network architecture was chosen according to the report instructions.

The critic network has two inputs in order to be able to process the state before taking into account the action that was chosen. It does not have an activation function in the output layer in order to not constrain the value estimate.

The actor network on the other hand takes only the state as input and is restricted to the environment's action values by choosing a $tanh(\cdot)$ activation function for the output.

2.3 Optimizer and hyperparameters

The hyperparameters used were chosen according to the instructions. We used a standard Adam optimizer. The discount factor (γ) is 0.99 because we want to penalize later rewards but at the same time we want the agent to think long term.

In general, you want a larger learning rate for the critic than for the actor. Large changes to the behavioral policy are problematic, since the gradient of our policy is stochastic and a large change to theta in combination with a bad batch might lead to a policy from which the agent is not able to recover anymore.

For the critic, the algorithm relies on a "nimble" Q-function estimate that quickly incorporates the shift in focus under the new policy pi. A larger learning rate is therefore desirable, as long as the optimization is still stable.

2.4 Training process

Figure 11 shows the total episodic reward for the chosen hyperparameters. We can see that as the algorithm learns, the reward consistently increases. Regarding the steps per episode; at first the agent crashes very early which means very few steps, it quickly learns how to hover and then consistently lowers the number of steps per episode. It still has cases where it takes 1000 steps, mostly because it lands near the landing pad and continues using the side engines to correct its position.

Our chosen discount factor was $\gamma = 0.99$, now we will analyze it's effect on the training process by changing it to extreme values. Figure 12 shows how a $\gamma = 1$ discount factor training process looks like. Intuitively an agent that has a discount factor equal to one values every single reward the same without taking into account how long into the future such reward might come. The agent therefore keeps hovering and isn't able to learn how to land properly. This is reflected in the number of steps per episode. On the other hand, a very low discount factor favours rewards that come early instead of long term ones. The agent, before properly learning to hover, therefore rushes to the landing spot without ever mastering a controlled descent. The resulting strategy yields a poor reward with short episode times.

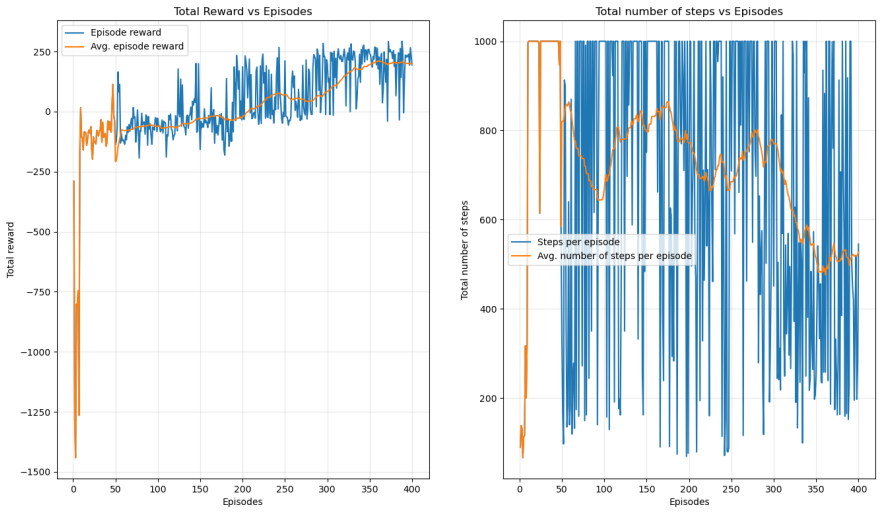

Figure 11: Total episodic reward (left) and steps per episode (right) in the training process.

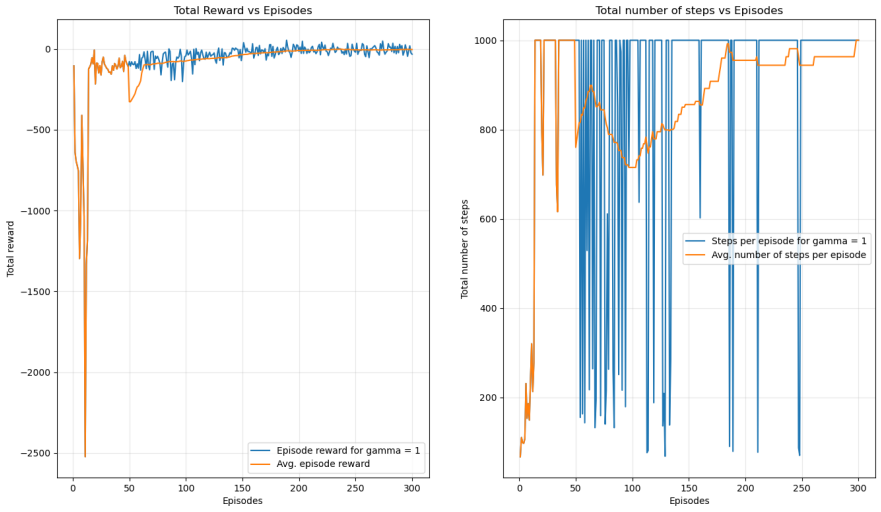

Figure 12: DDPG training process with $\gamma = 1$.

Fixing the rest of the parameters and changing the memory size we obtain Figure 14 and Figure 15. We see that if we use a buffer that is too small (3k) training becomes less stable since the agent at one point unlearned because of catastrophic interference, that is, after the agent chose a bad policy it was only able to learn from samples generated under such policy hence worsening performance. A buffer too big might make the algorithm learn slower since it uses samples back at time steps where it hadn't learnt.

2.5 Policy evaluation

The value function depicted in Figure 16 is similar to the previous one. Once again values of w that are close to π or $-\pi$ mean that the aircraft is upside down. As those states always yield bad rewards they are avoided and therefore it's values not updated.

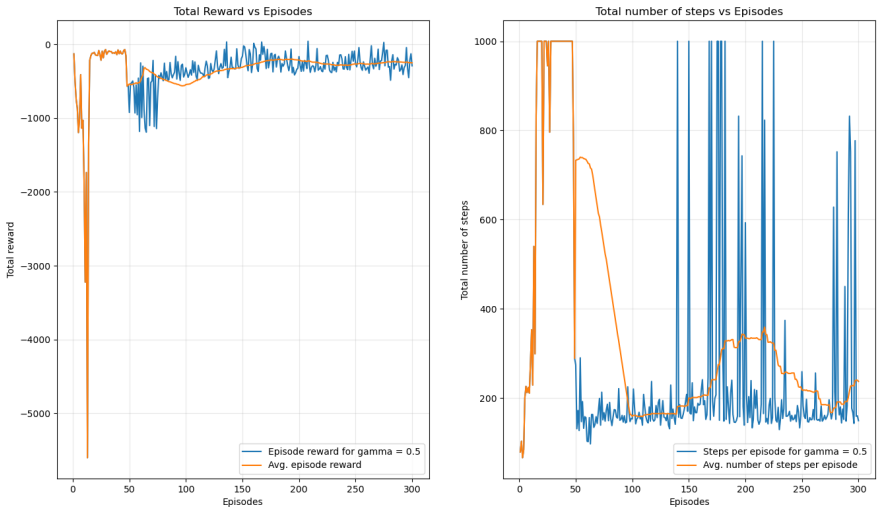

Figure 13: DDPG training process with $\gamma = 0.5$.

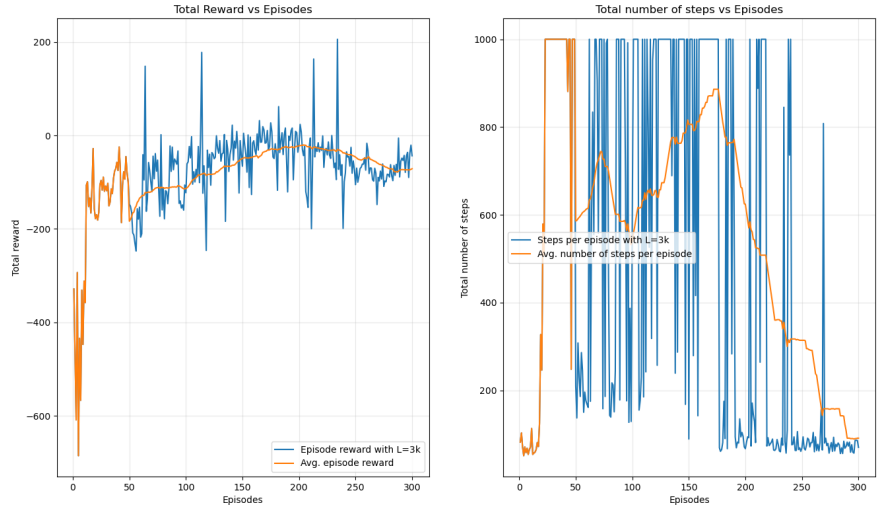

Figure 14: DDPG training process with $L = 3k$.

States with $w = 0$ are again valued less than the neighboring ones. This confirms our previous assumption that the agent prefers to have a small angle as to be able to control its x position with the main engine which is more powerful, and maybe even get some fine-level control on its vertical velocity with the side thrusters.

Figure 17 shows the policy behavior for firing the side engines. The step function is even more pronounced now, clearly showing the desire of the agent to stabilize its orientation.

2.6 Random agent vs DDPG

Figure 18 shows the total episodic reward comparison between the random agent and the DDPG agent we trained. It is clear that the DDPG consistently outperforms the random agent.

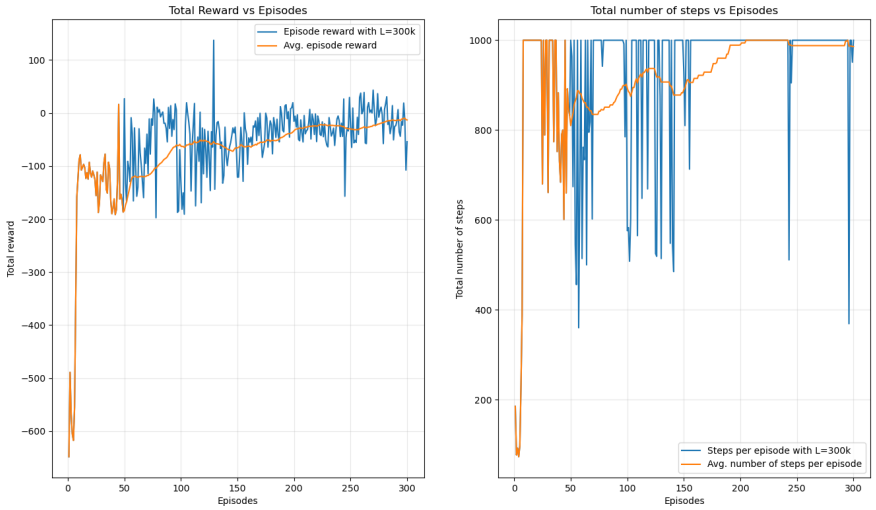

Figure 15: DDPG training process with $L = 300k$.

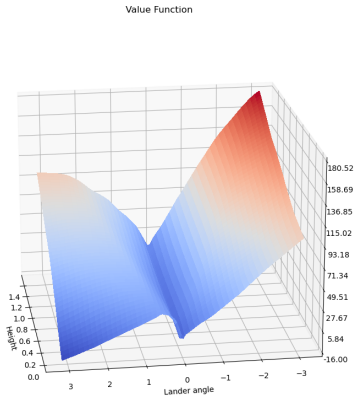

Figure 16: DDPG trained policy value function.

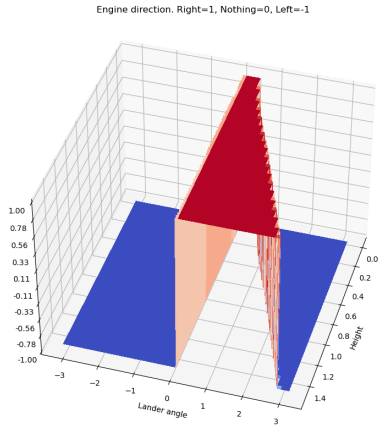

Figure 17: DDPG Optimal policy behaviour of side engine for our trained agent.

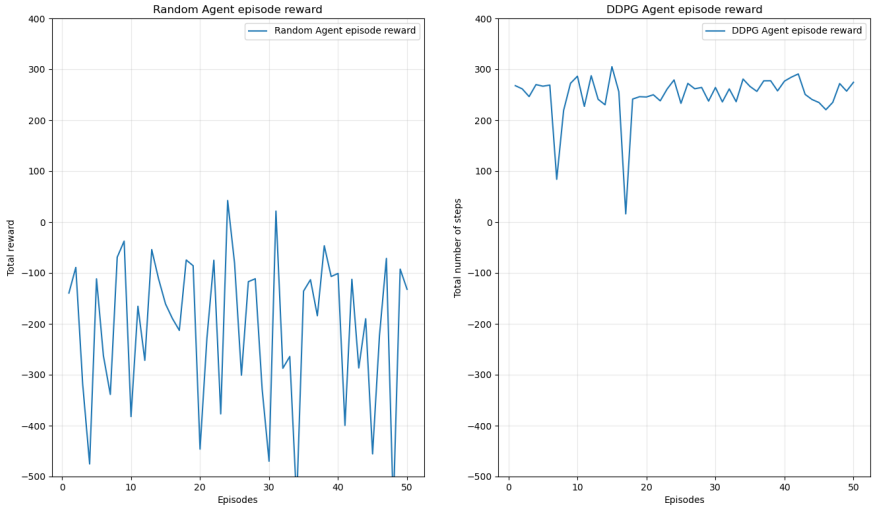

Figure 18: Random agent vs DDPG: episodic rewards.

3 Problem 3: Proximal Policy Optimization (PPO)

3.1 PPO properties

PPO updates the value function with a Monte Carlo estimate instead of a network as the target. This is only possible because it doesn't sample at random from a replay buffer, but always has access to a complete trajectory τ. This complete history of rewards enables a direct computation of G without networks to estimate the value for the next state (instead, its value can also be inferred with the reward history).

PPO uses the same policy that it samples actions from for the parameter update. Also, the single trajectory that is used for the current update step is always generated by the most recent policy. Therefore it is an on-policy algorithm.

Regarding sample complexity, on-policy algorithms perform worse since they cannot store samples which would allow them to reuse and learn from old trajectories. PPO lessens the effect of sample complexity by reusing the same trajectories M times and making sure that the optimization step is still within reasonable bounds.

3.2 Network Layout, optimizer and hyperparameters

The hyperparameters used were chosen according to the instructions as well. We used a standard Adam optimizer. Again, the network size is increased compared to the discrete case, with the same argument of complexity as for DDPG. We found that it was beneficial to train for more than 1600 episodes though and combine that with early stopping if the agent exceeded a reward threshold over the last 50 episodes.

Updating the actor less frequently would indeed make training more stable, since the estimate of the advantage function would be more accurate for the policy update. However, the whole point of PPO with clipping was to ensure a stable policy update by restricting the gradient to trusted regions. Instead of arbitrarily changing the frequency, it therefore makes more sense in PPO to adjust ϵ, which also directly effects the stability of the training process.

3.3 Training process

We found that the performance of our agent during training had a fairly high variance. Some runs would result only in average rewards of ~ 100, whereas other runs managed to achieve ~ 240. The results shown here are achieved by running multiple trials and using early stopping on training success (an average reward of > 200 over the last 50 episodes). The agent consistently improves, although it has a large plateau during its training. It should be noted that this is by chance, on other training runs, the agent converged faster (<2000 episodes) and sometimes even slower.

As in DDPG, both a lowering the discount factor to 0.5 (Figure 21) and increasing it to 1 (Figure 20) resulted in unsuccessful training runs. The explanations remain the same: For a low discount factor, the agent rushes its moves, and for $\gamma = 1$ it is missing an incentive to learn how to land within its limited episode time.

PPO proved to be surprisingly stable with respect to changes of ϵ. We expected to see a slow, yet stable learning process for small ϵ values (0.05) and large, unstable changes for larger values (0.95). For small values, our assumption proved to be correct (Figure 22). However, even for larger values, the training was still reasonably stable(Figure 23). Both agents failed to achieve outstanding results, but improved their policy considerably compared to random behavior. We discovered that the similarity is due to the fact that our probability changes rarely ever reach the clipping values for M=10, making both cases essentially the same.

3.4 Policy evaluation

The value function (Figure 24) shows the same peculiar shape as for DQN and DDPG, further confirming our hypothesis. PPO also found the same policy as DDPG for its side engine direction (Figure 25). It is worth a note that this time, the policy is completely stable on the right hand side as well. This might be due to a more thorough exploration, or the function approximation artifacts in unexplored regions are more fitting by pure chance.

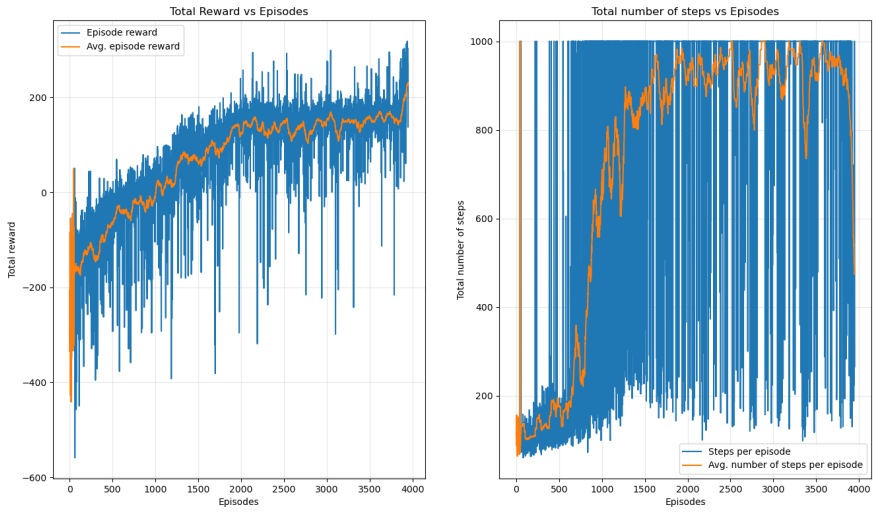

Figure 19: Total episodic reward (left) and steps per episode (right) in the training process.

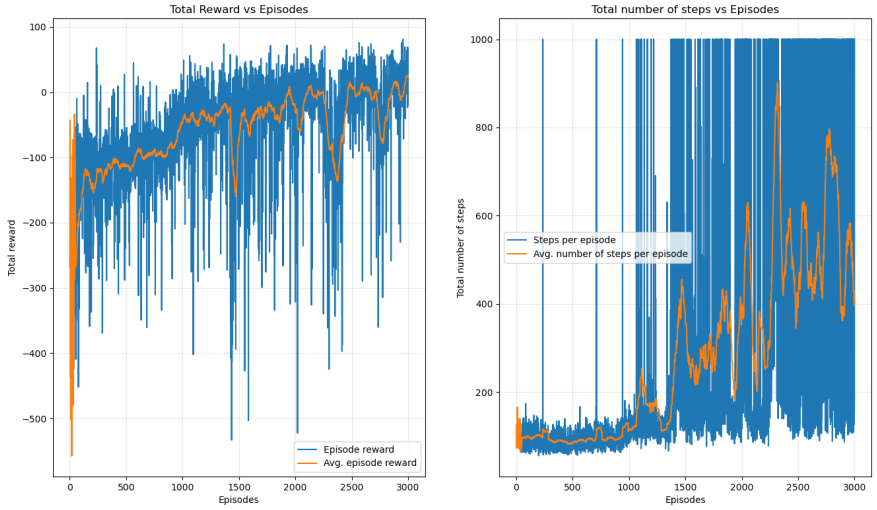

Figure 20: Training process with $\gamma = 1$.

Figure 25 shows the policy behavior for the side engine, here we clearly see a step function around the $w = 0$ value which is intuitively correct.

3.5 Random agent vs PPO

Figure 26 shows the total episodic reward comparison between the random agent and the PPO agent we trained. It is clear that the PPO consistently outperforms the random agent.

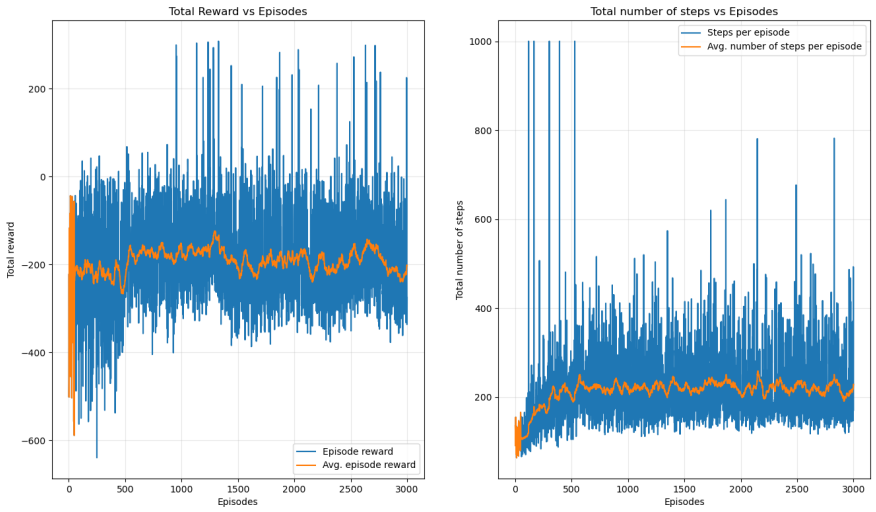

Figure 21: Training process with $\gamma = 0.5$.

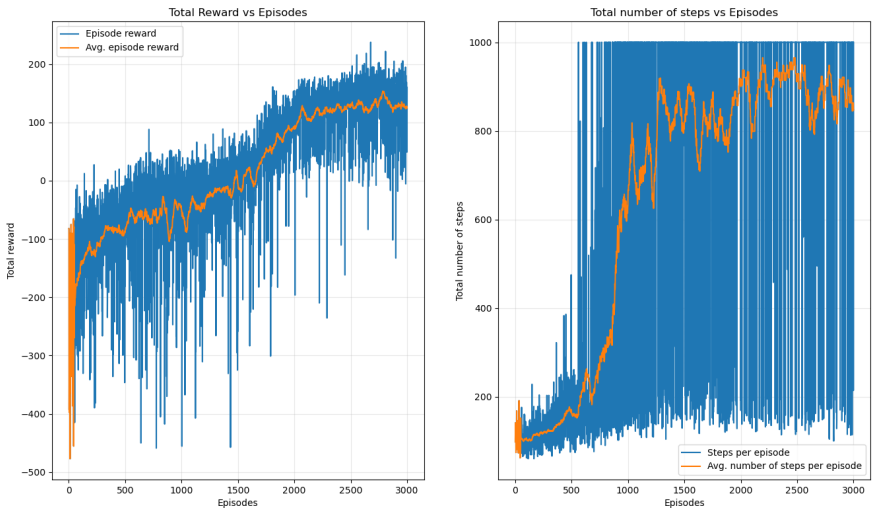

Figure 22: Training process with $\epsilon = 0.05$.

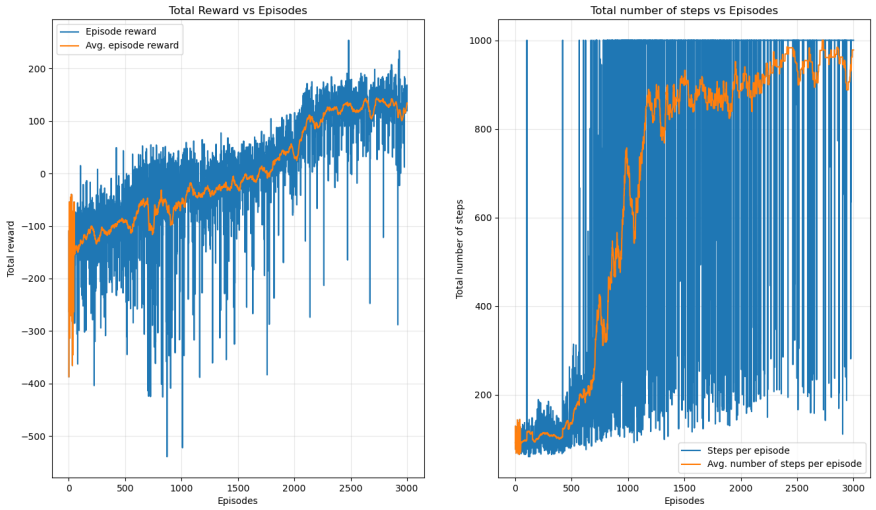

Figure 23: Training process with $\gamma = 0.95$.

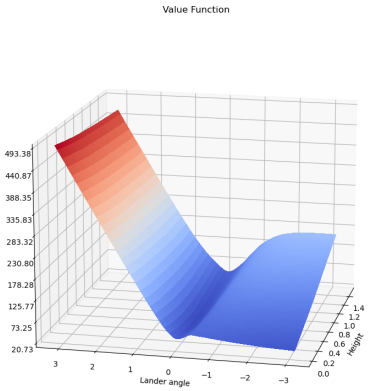

Figure 24: Trained policy value function.

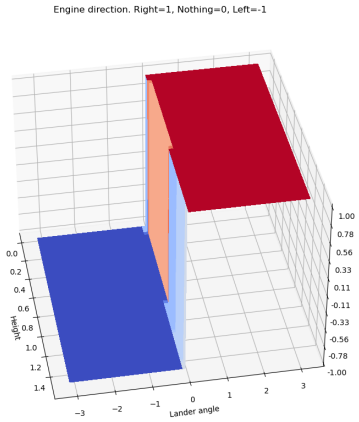

Figure 25: Optimal policy behaviour of side engine for our trained agent.

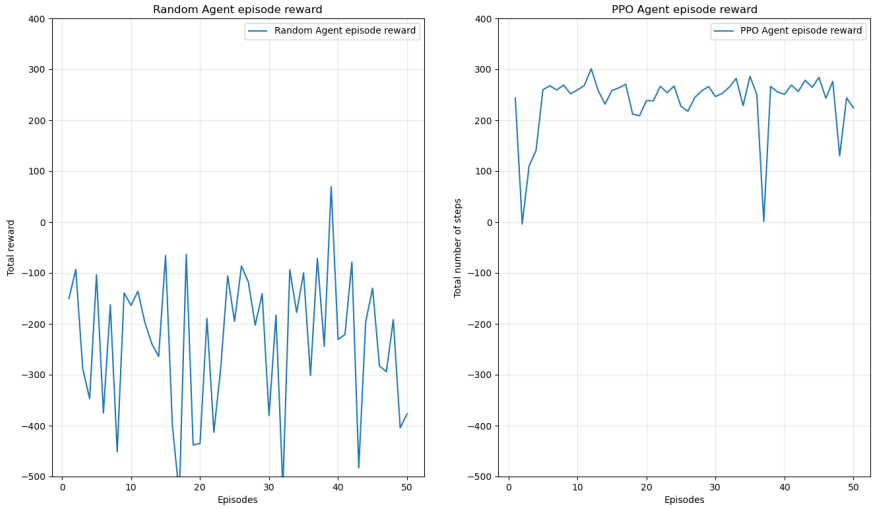

Figure 26: Random agent vs PPO: episodic rewards.