Reinforcement Learning (EL2805) Lab 2

EL2805, Autumn 2023

Oscar Eriksson	Philip Ahrendt
oscer@kth.se	pcah@kth.se
001130-1991	960605-R119

December 2023

Problem 1: Deep Q-Networks (DQN)

Tasks

(a) Random Agent Baseline

Familiarize yourself with the code in `DQN problem.py`. Check the agent taking actions uniformly at random (implemented in `DQN agent.py`). Use this random agent as a baseline for comparison with your final agent.

(b) Replay Buffer and Target Network in DQN

Explain the purpose of using a replay buffer and target network in DQN.

If we have a replay buffer of length 30000, and the average episode will run for about 200 steps before terminating. If we fill the replay buffer and randomly sample mini-batches the network will be more agnostic to the order/time/context that a state/action/next state tuple occurs. This means that the network will more closely approximate an optimal one-step policy.

A target network is a good idea since the standard parameter update formulation will successively move the target which produces instabilities in training. If we instead produce the target using a fixed target network, we see better stability. After C iterations (C being rather large) using the fixed target, we then update the target network parameters to match the network parameters.

(c) Problem Solving

Implement the experience replay buffer. Define the Neural Network architecture and the optimizer. Implement DQN to solve the problem. Your algorithm is considered successful if it achieves an average total reward of at least 50 points over 50 episodes. Save your Q_θ network after solving the problem (do not save the target network).

We record the following output from the `DQN_check_solution.py` script

```
Policy achieves an average total reward of 231.5 +/- 19.8 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required.

(d) Network Layout and Parameters

Explain the layout of the network you used.

The following pseudo code details the structure of the network

```
input_size = 8
hidden_layer_size = 64
output_size = 4
...
nn.Sequential(
    nn.Linear(input_size, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, output_size),
)
```

The input size and output sizes correspond to the dimensions of the state and the actions respectively. The input layer is a 8×64 fully connected layer with a relu activation, the middle layer is a 64×64 fully connected layer with a relu activation and the output layer is a 64×4 fully connected layer with no activation. In order to select the action from the unconstrained outputs, we simply select the arg max, same as for ordinary Q-learning.

Justify the choice of the optimizer.

The following pseudo code details the optimizer

```
learning_rate = 0.0005
...
optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate)
```

Specify the parameters used: γ, L, TE, C, N, ε.

The following pseudo code details the use of parameters

```
ERB_size = 30000 # L
epsilon0 = 0.99 # initial varepsilon
N_episodes = 300 # TE
discount_factor = 0.99 # gamma
batch_size = 128 # N
C = ERB_size // batch_size # rate of target network update approx 234
```

We decay ε using

$$\varepsilon_t = \varepsilon_0 e^{-3t/TE}$$

which means that we start out with $\varepsilon_0 = 0.99$ and end up at $\varepsilon_{TE} \approx 0.05$.

Mention any modification implemented (if applicable) and provide reasons for your choices.

As mentioned above, an exponentially decaying epsilon was chosen. It appears to make it so that the first few hundred episodes are spent rapidly finding good candidate policies, and the last few hundred episodes are spent doubling down on some policy and making it perform really well.

(e) Analysis after Problem Solving

Plot the total episodic reward and the total number of steps taken per episode during training. Discuss observations regarding the training process.

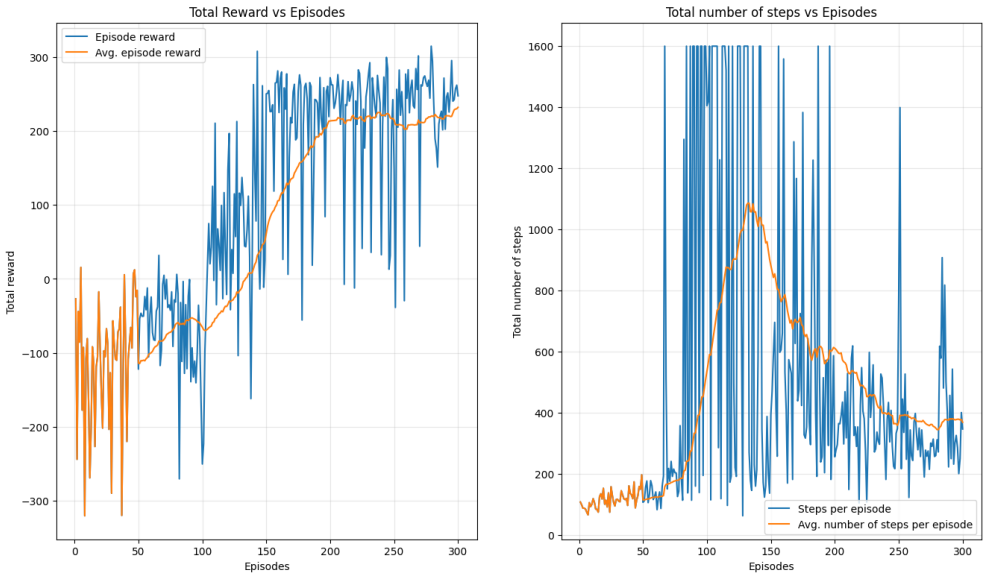

Figure 1: Training data for DQN with $\gamma = 0.99$

In figure 1 we see that it initially doesn't take very many steps before terminating, since it doesn't have a good policy and is very exploratory. The poor reward at the corresponding episodes hint towards the fact that the lander is crashing. Around 100 episodes in, we see that the policy is getting better and better, since it is staying around for longer and longer (it has learned how to fly without crashing!) and is breaking even in terms of reward. It is, however, too exploratory to actually land on the ground. Towards the end we see that it starts terminating earlier and earlier, meaning that it is getting quicker and quicker terminating, and since we have such high rewards at the corresponding episodes it is fair to assume that we are landing the lunar lander!

Choose a discount factor γ_0 that solves the problem. Set $\gamma_1 = 1$ and $\gamma_2 < \gamma_0$. Redo the plots for γ_1 and γ_2. Analyze the impact of the discount factor on the training process.

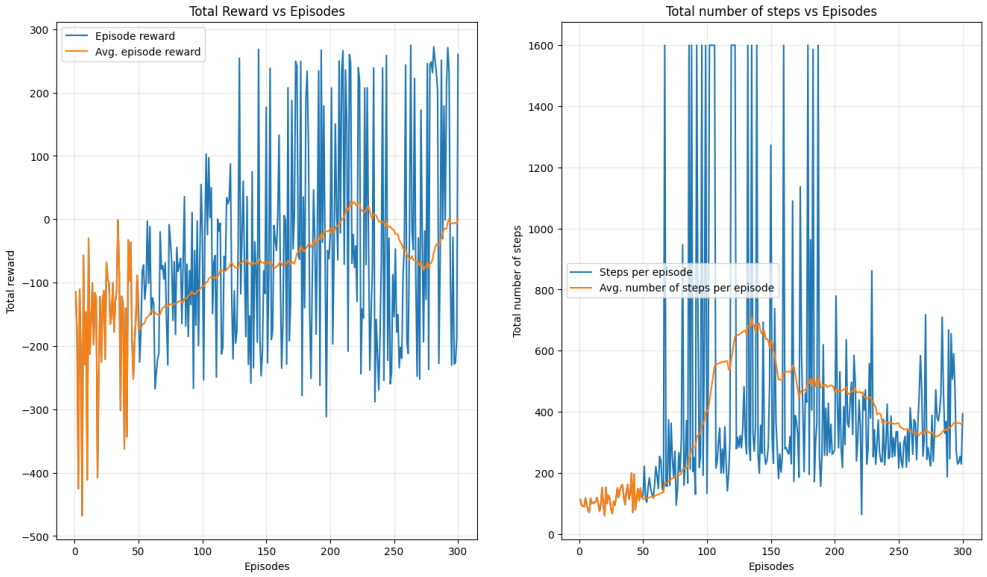

Figure 2: Training data for DQN with $\gamma_2 = 0.20$

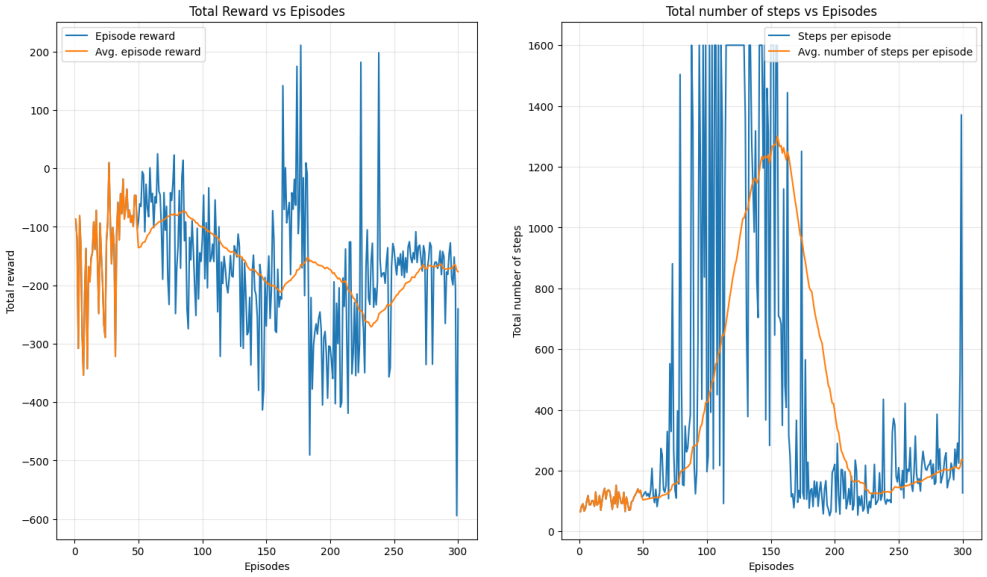

Figure 3: Training data for DQN with $\gamma_1 = 1.00$

We see in figure 2 that for a small discount factor γ_2 we barely break even in terms of total episode reward, but we follow the same trends as for γ_0 just less exaggerated. One could say that we are more eager to start landing the lander once we think we learned how to fly it, but we are too bad at flying to reliably land it. In Figure 3 the discount fact of $\gamma_1 = 1$ means that future reward are valued equally to future rewards. With this approach the agent fails to learn how to land in the episode time limit.

Investigate the effect of varying the number of episodes and memory size for your initial choice of the discount factor γ_0. Document findings with representative plots.

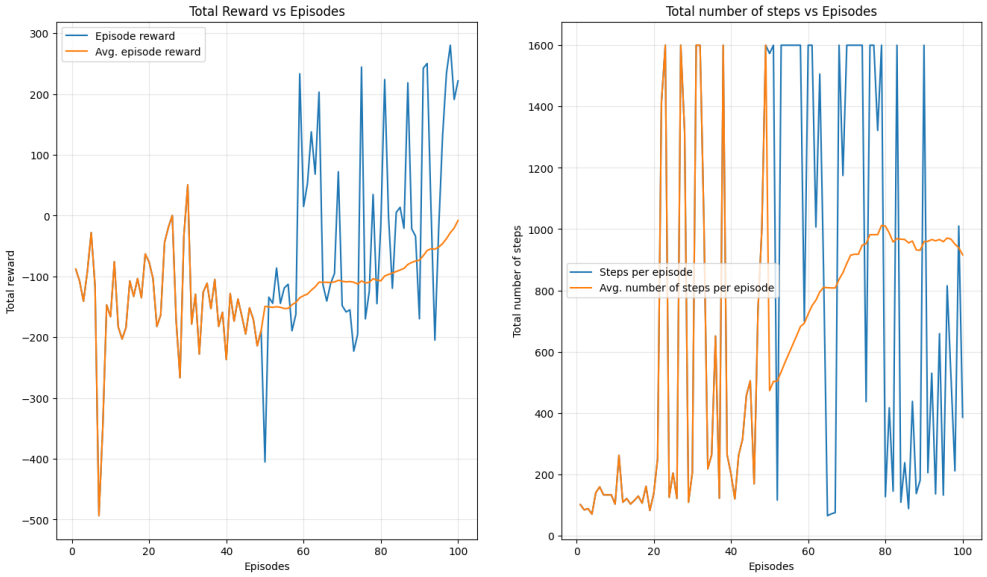

Figure 4: Training data for DQN for 100 episodes

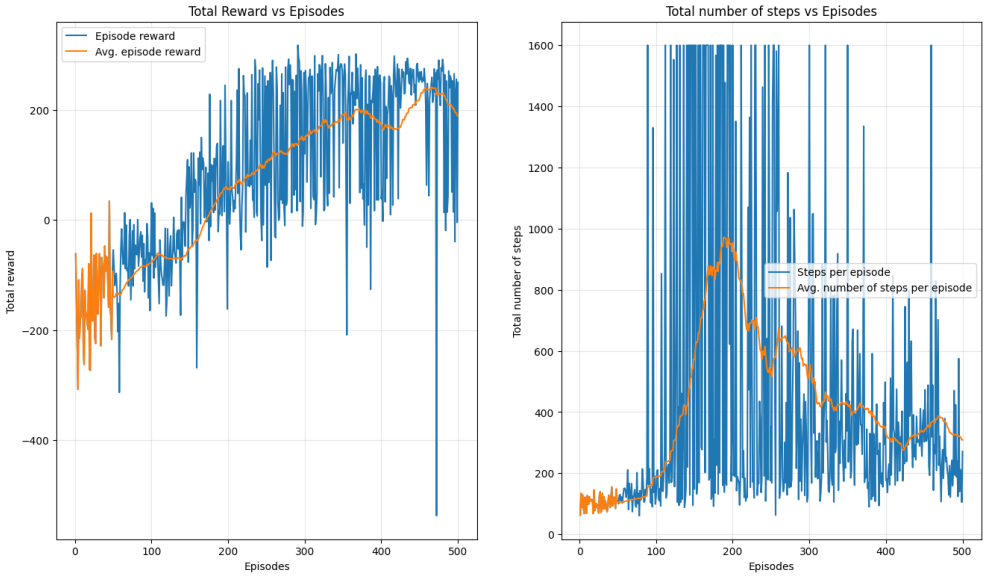

Figure 5: Training data for DQN for 500 episodes

In figures 4 and 5 we see what happens when we train for 100 and 500 episodes respectively (using the same exponential decay in ε). The shorter training doesn't learn to fly very robustly, and never manages to break even in terms of total episodic reward. The longer training network has the same characteristics as the 300 episode one (our baseline), but does reach a higher total episodic reward. There does however appear to be some diminishing returns when one trains for that long.

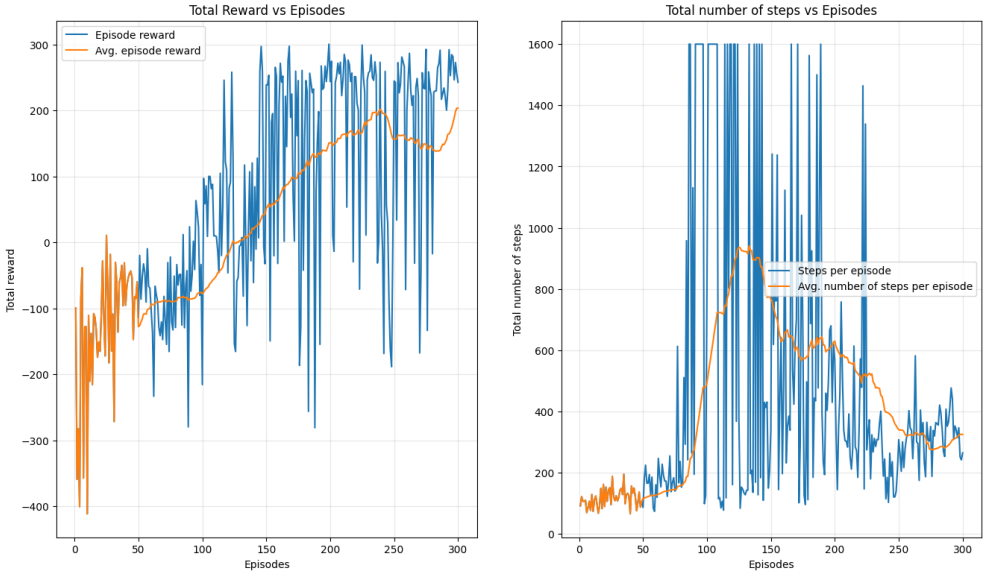

Figure 6: Training data for DQN with $TE = 10000$

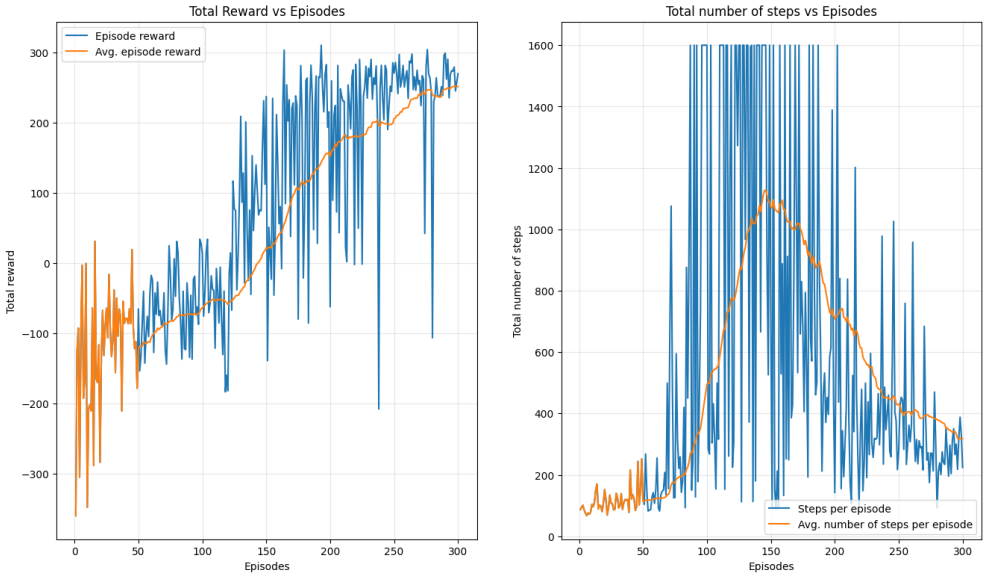

Figure 7: Training data for DQN with $TE = 50000$

We see in figure 6 that a relative decrease in TE of 1/3 from 1 (our baseline) emulates the effect of decreasing the discount cost, because the available experiences to sample from are more recent and less dis-correlated in time.
We see in 7 that a relative increase in TE of 5/3 from 1 emulates the effect of increasing the discount cost. Here one could argue that a substantial amount of the experiences from he last 50 episodes were produced with a relatively high ε, and as such the network is having an easier time translating from it's exploratory phase towards it's optimal policy. It doesn't suffer the severe dip in total episodic reward towards the last episodes like all previous examples, which could indicate that it is much safer to terminate training early once sufficiently high average total episodic rewards are achieved.

(f) Q-Network Analysis

For the Q-network that solves the problem, plot $\max_a Q_\theta(s(y,\omega),a)$ for varying $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$. Does the value of the optimal policy you found make sense? Explain it.

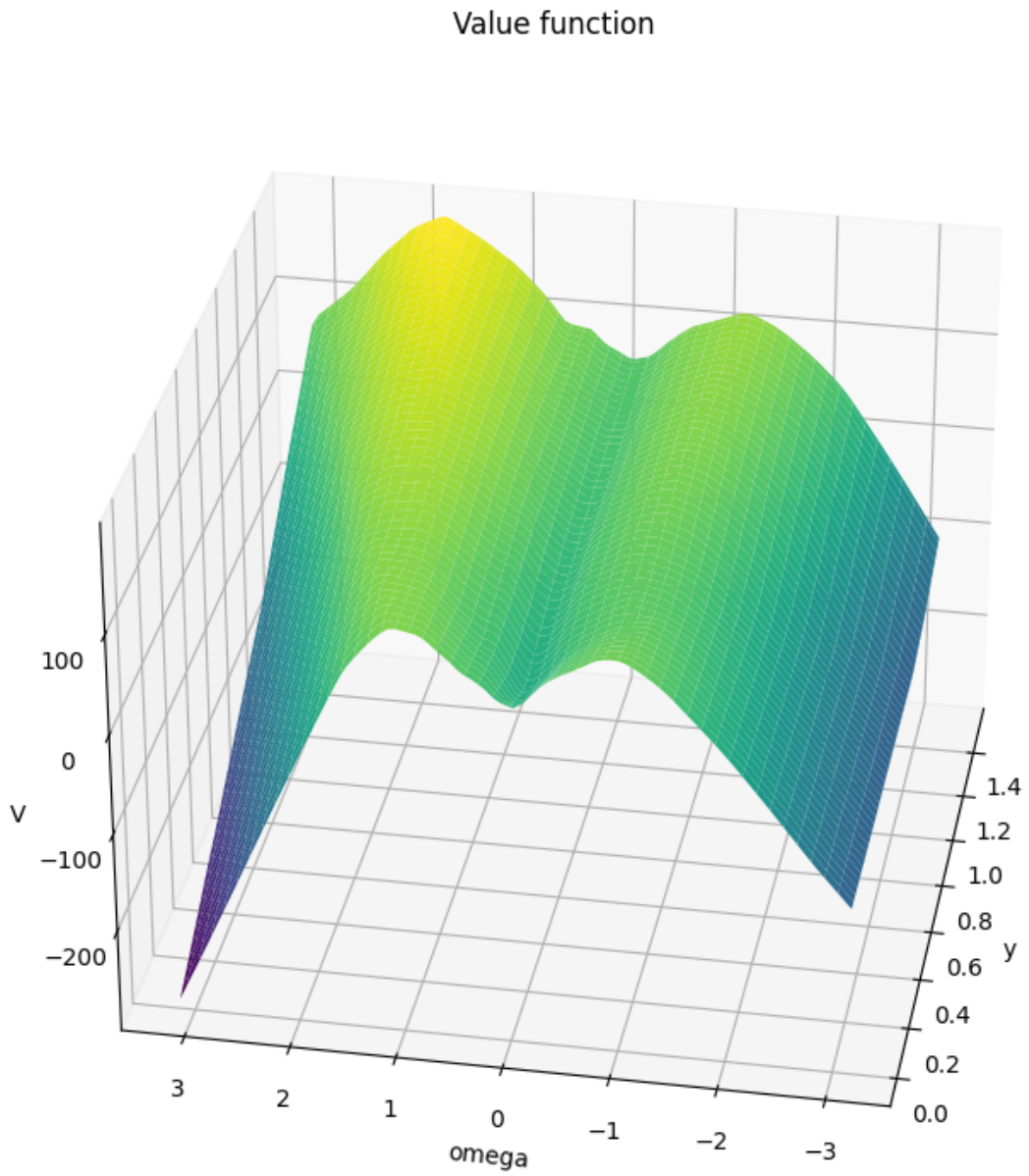

Figure 8: A plot over $\max_a Q_\theta(s(y,\omega),a)$ for varying y and ω.

We see in 8 that our policy is similar for all y-values (heights) as it is always beneficial to be close to perpendicular to the ground. For higher heights, we more often occupy wider ω, that is we are more okay with being less perpendicular to the ground since we have more time to manoeuvre towards the landing site before descending. We correlate a higher Q-value with a willingness to act any specific way. The policy doesn't know much about big ω since it probably doesn't operate much in those zones, which is especially

true closer to the ground.

For the same Q-network, plot argmax$_a Q(s(y,\omega),a)$ for varying y and ω. Does the behaviour of the optimal policy make sense? Explain it.

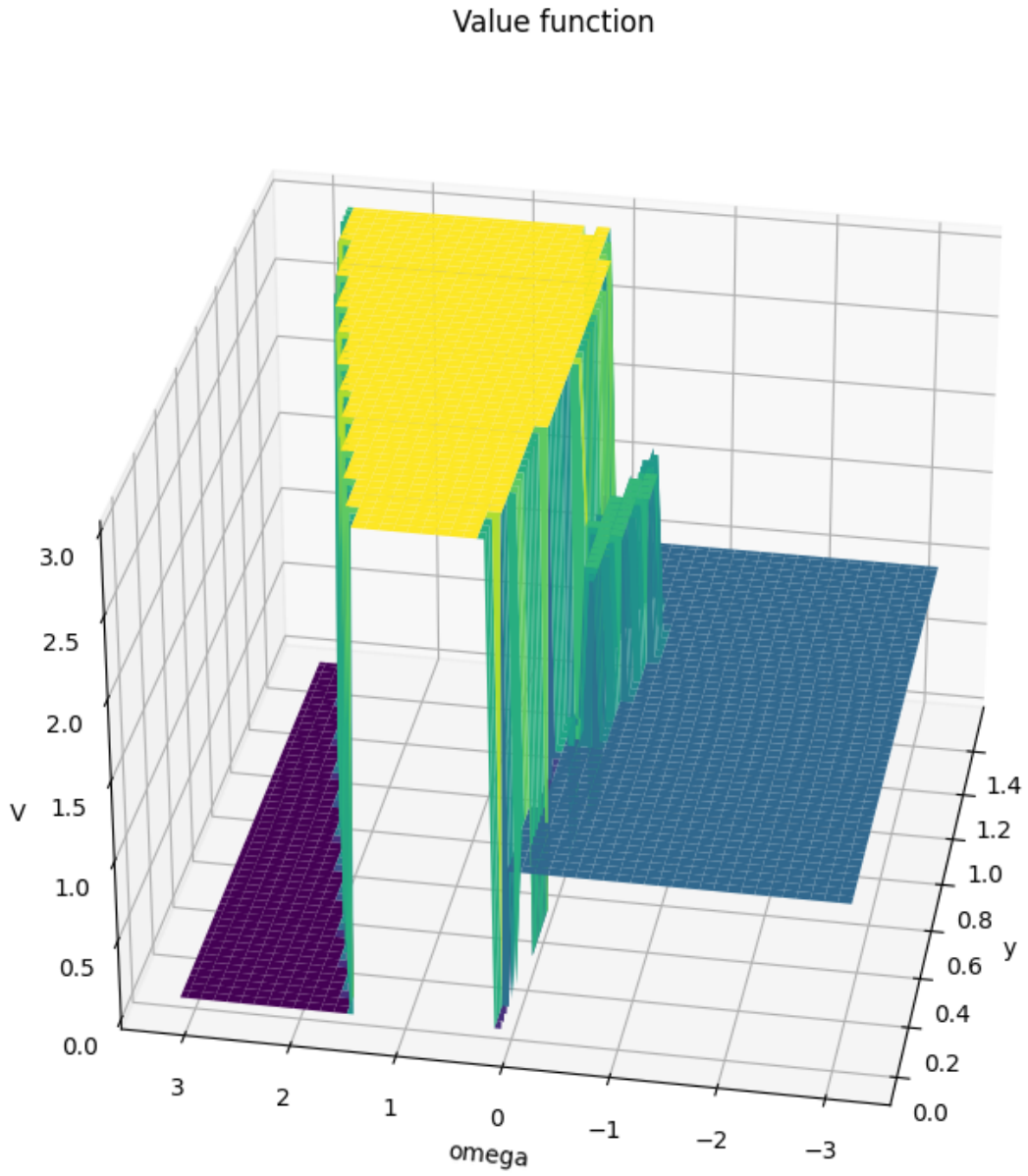

Figure 9: A plot over arg max$_a Q_\theta(s(y,\omega),a)$ for varying y and ω.

We see in 9 that if $\omega > 0$ that the ship angles itself towards perpendicularity, and if $\omega < 0$ it orients itself the opposite way, towards perpendicularity (with the ground). There is a thin sliver in the middle of actions corresponding to no action, as we are perfectly perpendicular in those cases. For behaviour around $\omega \approx \pm 3$ we assume that the network thinks its more beneficial to crash than to try and recover.

(g) Comparison with Random Agent

Compare the Q-network you found with the random agent from task (a). Show the total episodic reward over 50 episodes for both agents.

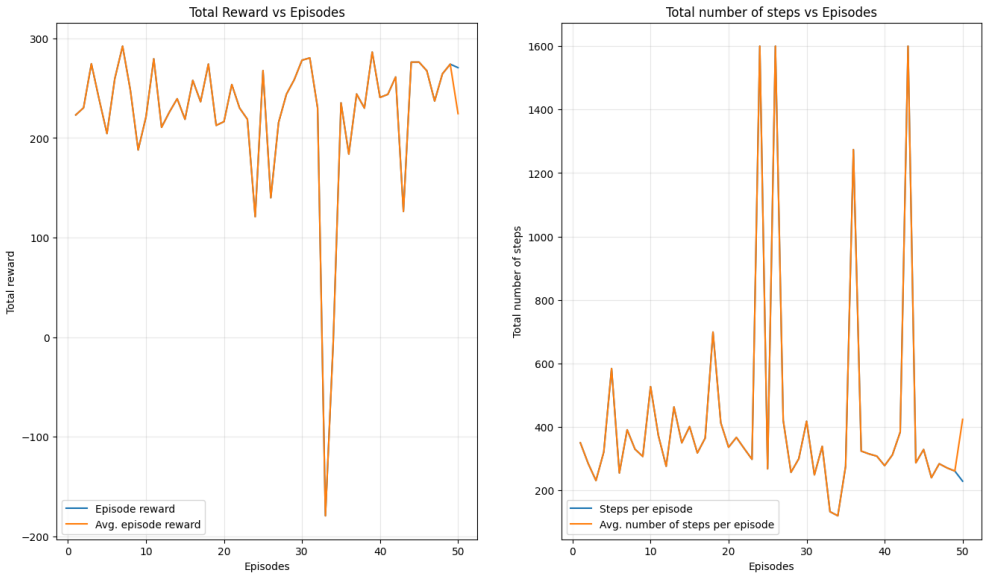

Figure 10: 50 episodes with our policy

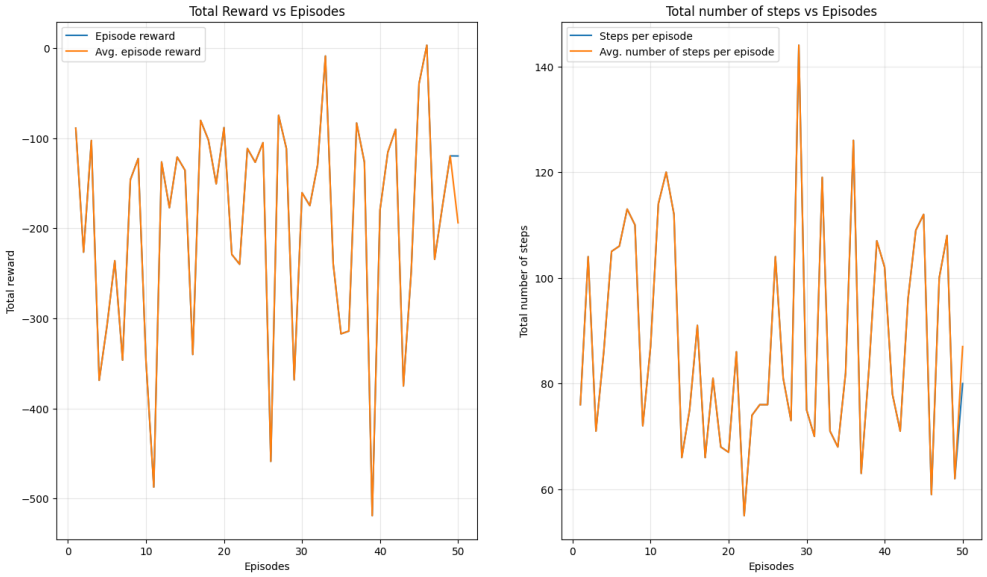

Figure 11: 50 episodes with random policy

We see that our policy vastly outperforms the random agent by comparing figures 10 with 11. It does however appear to have one unlucky accident.

(h) Save Final Q-Network

Save the final Q-network to a file named `neural-network-1.pth` **using the command** `torch.save(your_neural_network, 'neural-network-1.pth')`**. Verify the validity of your policy using** `python DQN_check_solution.py`**.**

We record the following output from the `DQN_check_solution.py` script

```
Policy achieves an average total reward of 231.5 +/- 19.8 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required.

Problem 2: Deep Deterministic Policy Gradient (DDPG)

Task

(a) Random Agent Baseline

Familiarize yourself with the code. Check the agent taking actions uniformly at random (implemented in DDPG agent.py). You will use this agent as a baseline for comparison with your final agent. Notice that the actions are clipped between -1 and 1.

(b) Some Questions

Why don't we use the critic's target network when updating the actor network? Would it make a difference?

The actor's policy update in DDPG directly relies on the gradient of the Q-value function (critic) with respect to actions. Using the critic's target network for this would mean basing the actor's update on an outdated version of the Q-value function, potentially leading to suboptimal policy updates. This is why in DDPG the critic's target network is used for calculating the target Q-values but not for updating the actor network. This separation ensures more stable and reliable gradient estimates for the actor, as the critic's target network provides a more stable benchmark for policy improvement.

Is DDPG off-policy? Is sample complexity an issue for off-policy methods (compared to on-policy ones)?

DDPG is an off-policy algorithm, which means it can learn from experiences generated from a different policy than the current one. This characteristic allows the use of a replay buffer to reuse past experiences, enhancing learning efficiency. Sample complexity is less of an issue for off-policy methods compared to on-policy ones, as they can leverage past experiences more effectively.

(c) Implement Modified DDPG

Implement the modified version of DDPG (shown in Algorithm 2) and solve the problem. You are not allowed to change the environment or the reward signal. To verify that your policy solves the problem, check question (h).

We record the following output from the `DDPG_check_solution.py` script

```
Policy achieves an average total reward of 221.8 +/- 50.4 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required.

(d) Answer the Following

Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used. Motivate why you made those choices. In general, do you think it is better to have a larger learning rate for the critic or the actor? Explain why.

The following pseudocode details the network layouts, the parameters and the optimizers

```
# parameters
input_size = 8
output_size = 2
ERB_size = 30000
epsilon0 = 0.99
batch_size = 64
```

```
N_episodes = 300
discount_factor = 0.99
learning_rate = 0.0005
tau = 0.001
d = 2
mu = 0.15
sigma = 0.2
hidden_layer_size = 64

...

# optimizers
optimizer_critic = torch.optim.Adam(network_critic.parameters(), lr=learning_rate)
optimizer_actor = torch.optim.Adam(network_actor.parameters(), lr=learning_rate)

...

# actor
nn.Sequential(
    nn.Linear(input_size, 400),
    nn.ReLU(),
    nn.Linear(400, 200),
    nn.ReLU(),
    nn.Linear(200, output_size),
    nn.Tanh()
)

# critic
nn.Sequential(
    nn.Linear(input_size + output_size, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, 1)
)
```

The structure of the actor network was given to us in the problem description, and the choice of parameters were barely adapted from the given ones, if at all. These are however, very similar to the DQN parameters, so there are no wild discrepancies.
The actor network takes in the state and outputs an action. There are several fully connected linear layers with relu activations to introduce nonlinearities, and the output is put trough a hyperbolic tangent so that the network only outputs normalized values in $[-1, 1]$.
The structure of the critic network is such that it takes in a state and action (input + output corresponding to the actor) and uses some fully connected network with relu activations to introduce nonlinearities to capture complex behaviour. It all boils down to one number, $Q(s, a)$, which of course doesn't use an activation function as Q-values are in general unconstrained.
The optimizers were chosen to be Adam, as these have empirically worked really well so far. The alternative is perhaps SGD, but that optimizer never fails to disappoint.
If any optimizer should have a smaller/slower learning rate it ought to be the actor. The quality of the critic depends somewhat on the quality of the actor, and if the actor is really erratic/incomprehensible due to a too great learning rate, then the critic will internalize the erratic behaviour of the actor.

(e) Analysis after Problem Solving

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

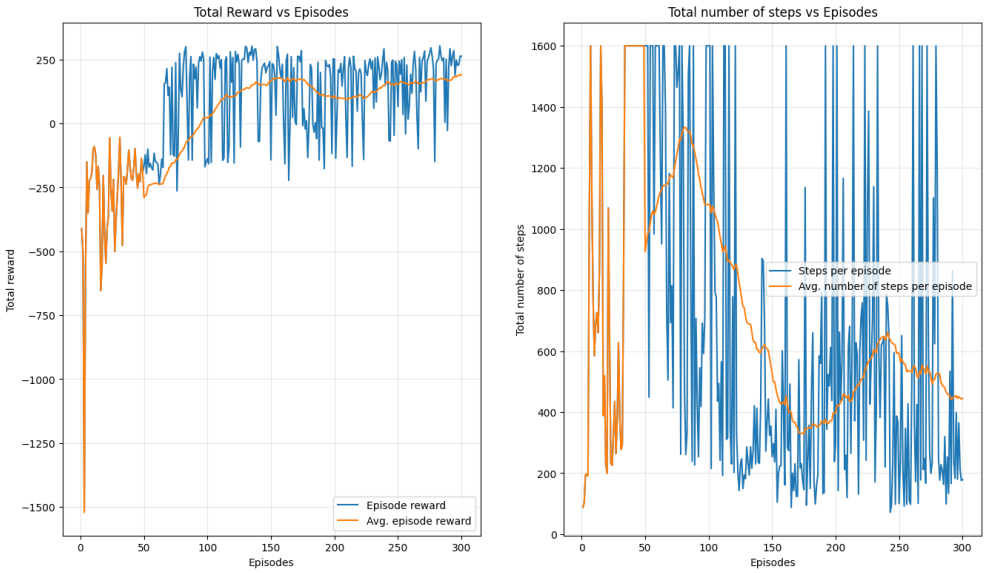

Figure 12: Training data for DDPG with $\gamma = 0.99$

In 12 we see that the network is exceedingly fast with learning a policy which allows it to fly. It is then also really quick to learn how to land, but has some crashes along the way as one can spy a dip in the total episodic reward. But in the end it churns out a really good policy.

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 $\gamma_2 < \gamma_0$. 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?

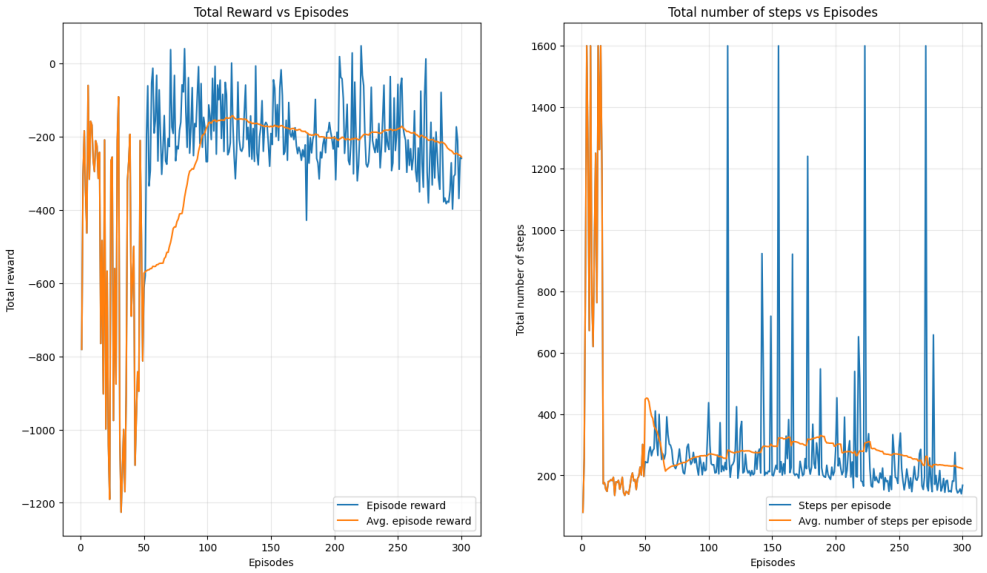

Figure 13: Training data for DDPG with $\gamma_2 = 0.20$

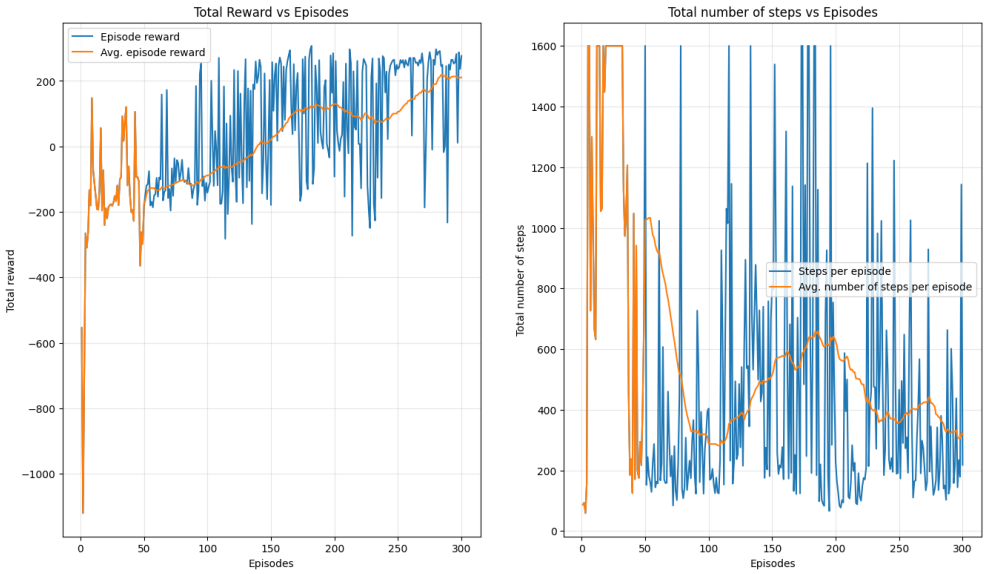

Figure 14: Training data for DDPG with $\gamma_1 = 1.00$

In figure 13 we see that a too exploratory network appears to learn to fly early, but when it tries to start landing it fails, and then it is never able to recover from the really poor policy it has developed. This network has no redeeming qualities.
In figure 14 we see the same trends as in 12, but it is much slower to start landing near the landing zone, as can be discerned from the consistently lower total episodic reward. It does however result in a good policy

in the end.

For your initial choice of the discount factor γ_0, investigate the effect of increasing/decreasing the memory size (show results for 2 different values of L). Does training become more/less stable? Document your findings with relevant plots.

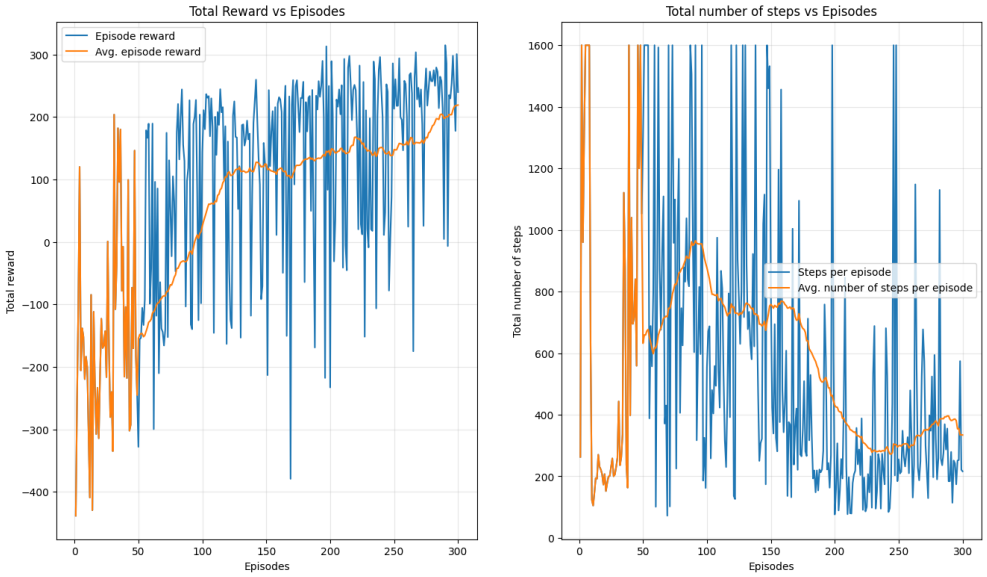

Figure 15: Training data for DDPG with $TE = 10000$

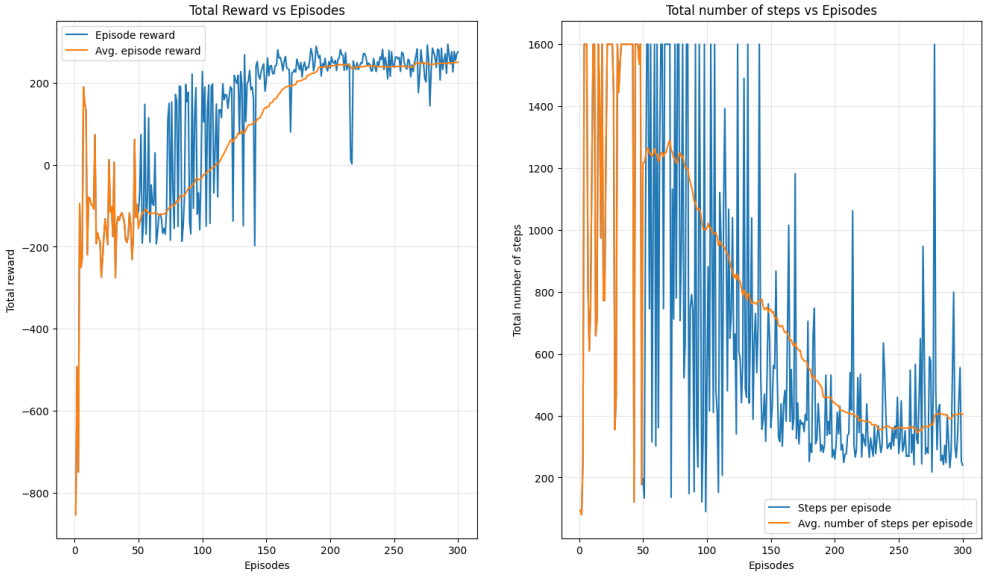

Figure 16: Training data for DDPG with $TE = 50000$

Similar too our analysis of the DQN under the same alterations (relative increase/decrease of TE by 1/3 and 5/3 respectively in figures 15 and 16), we see that the prior network is a lot more exploratory, while the latter network doesn't necessarily sacrifice it's exploratory nature, but does land in the final optimal policy

more easily. We also see that the variance in the total episodic reward grows really small, so we could have terminated at episode 200 with an excellent policy.

(f) Q-Network Analysis

For the networks π_θ, Q_ω that solve the problem, generate the following two plots: 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 $Q_\omega(s(y,\omega), \pi_\theta(s(y,\omega)))$ 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.

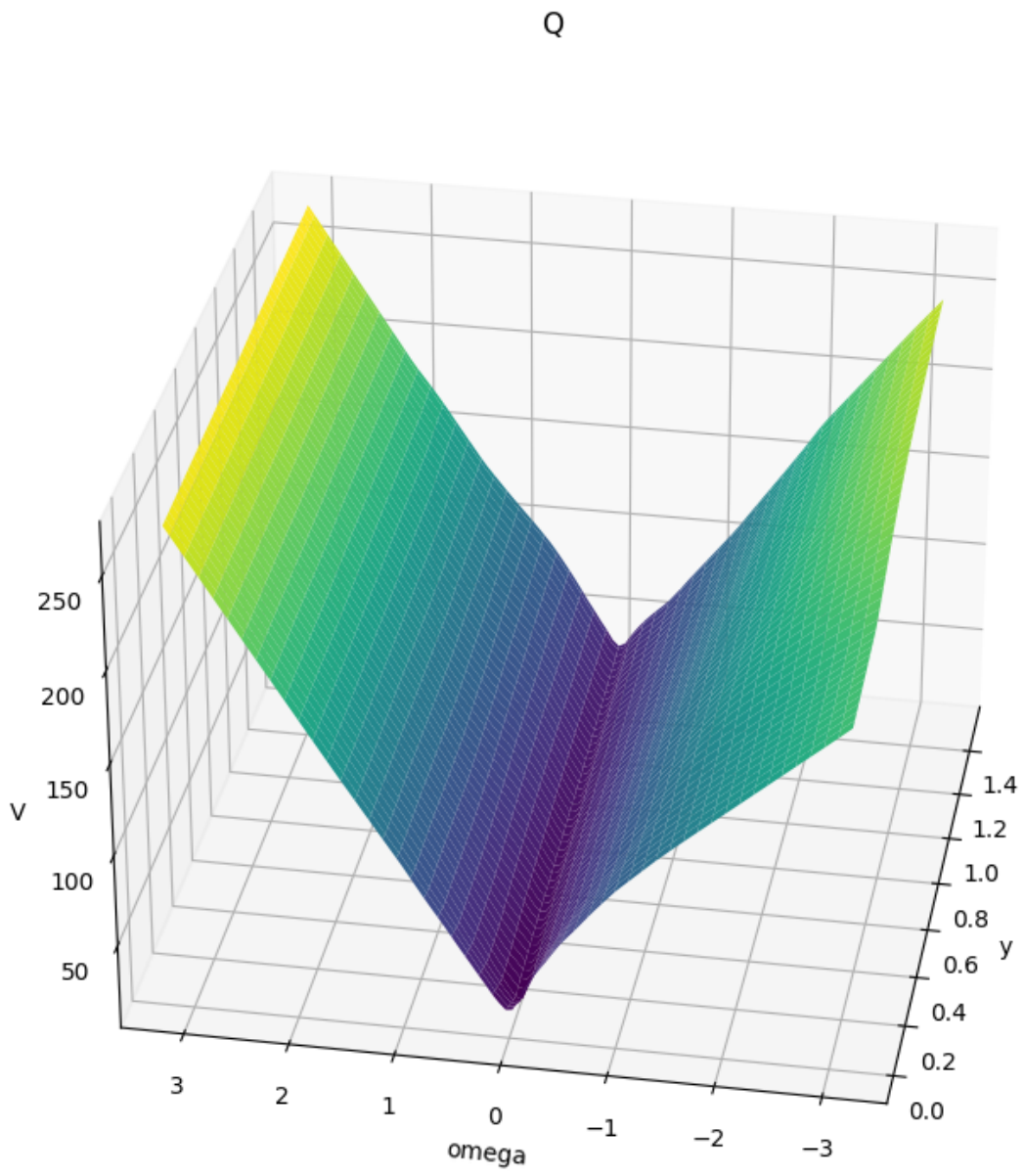

Figure 17: A plot over $\max_a Q_\omega(s(y,\omega), \pi_\theta(y,\omega))$ for varying y and ω.

In 17 we can again borrow much of our analysis from DQN, but we can state that this network is a lot better at taking the entire state space into consideration, and does really appropriate extrapolations as to how important it is to act at even really high/low ω. This network would probably be able to recover from these states.

Let s be the same as the previous question. Remember that the action is a bi-dimensional vector, where the second element denotes the engine direction. Plot the engine direction $\pi_\theta(s(y,\omega)))_2$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behavior of the optimal policy make sense? Explain it.

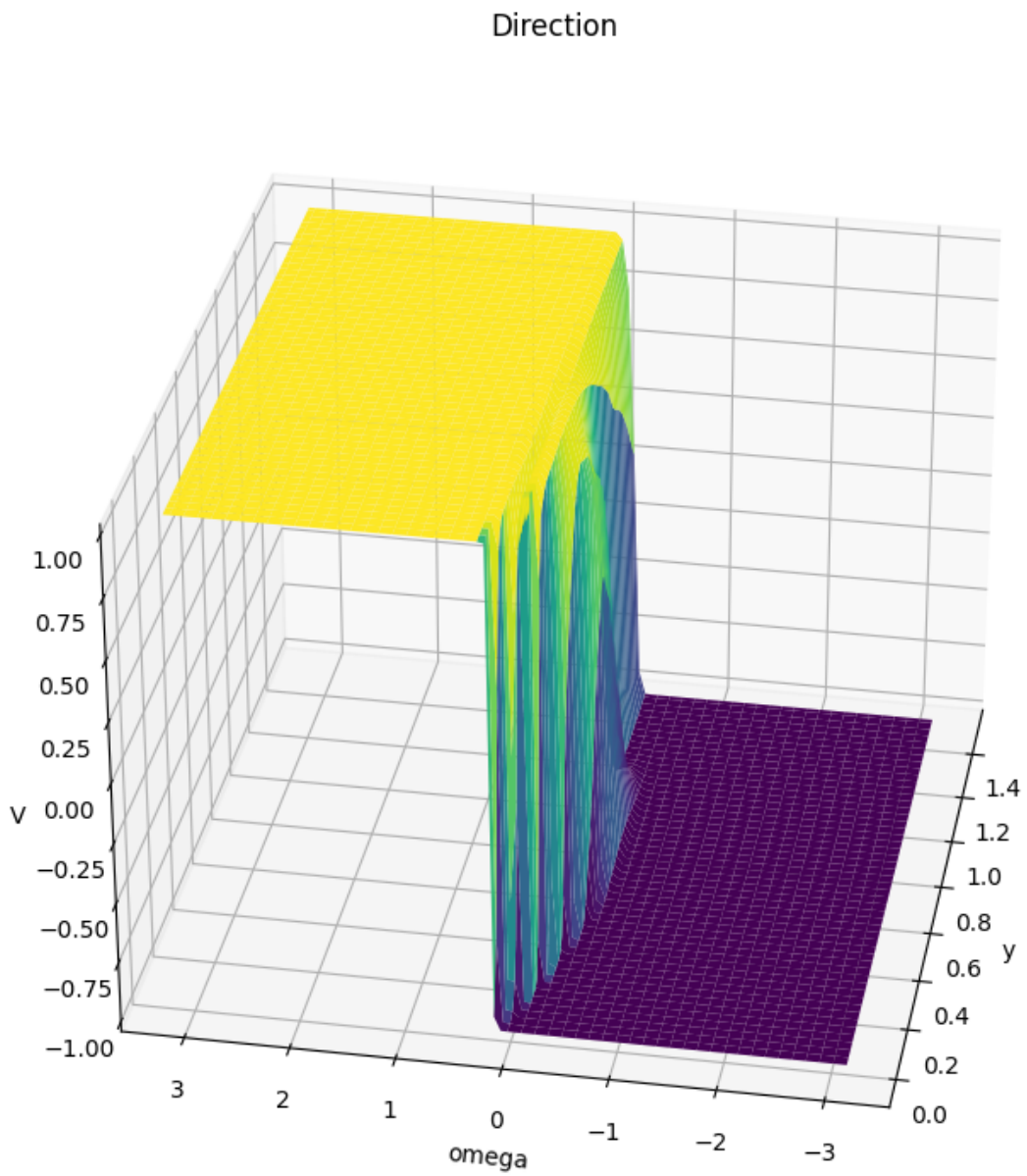

Figure 18: A plot over $\pi_\theta(s(y,\omega), \pi_\theta(y,\omega))_2$ (engine direction) for varying y and ω.

n 18 it is really easy to discern that the the controls always aim to maintain perpendicularity to the ground, regardless of height this time. There is however strange rippling action around $\omega = 0$, which hints towards the fact that this controller might be pretty erratic, though effective.

(g) Comparison with Random Agent

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

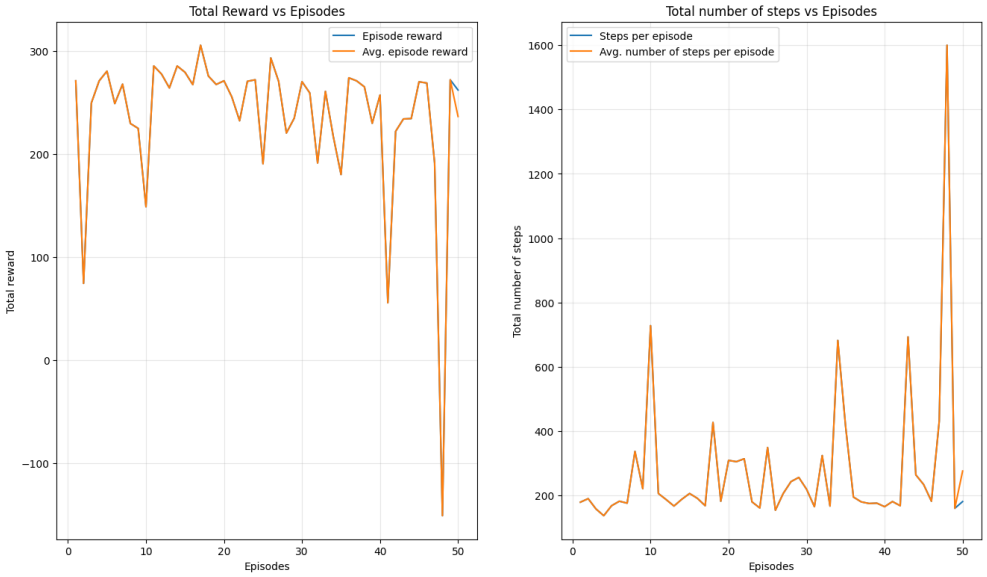

Figure 19: 50 episodes with our policy

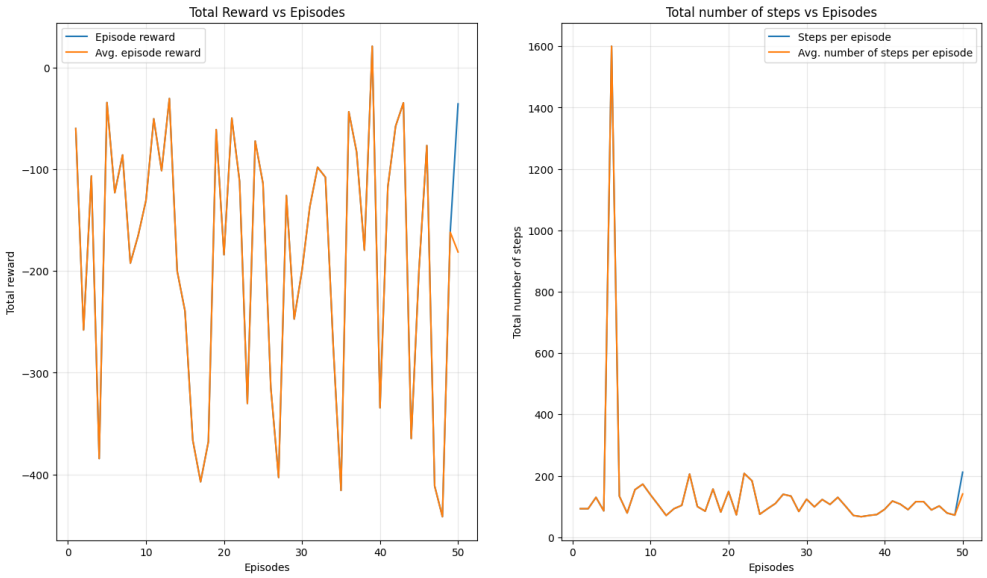

Figure 20: 50 episodes with random policy

We see that our policy vastly outperforms the random agent by comparing figures 19 with 20. It does however appear to have one unlucky accident, again.

(h) Save Networks

Save the actor and critic network from (c) to two different files `neural-network-2-actor.pth` **and** `neural-network-2-critic.pth`**. You can use the command** `torch.save(neural_net, path)` **to save a neural network. You can execute the command** `python DDPG check solution.py` **to verify the validity of your policy.**

We record the following output from the `DDPG_check_solution.py` script

```
Policy achieves an average total reward of 221.8 +/- 50.4 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required.

Problem 3: Proximal Policy Optimization (PPO)

Task

(a) Random Agent Baseline

Familiarize yourself with the code. Check the agent taking actions uniformly at random (implemented in PPO agent.py). You will use this agent as a baseline for comparison with your final agent. Notice that the actions are clipped between -1 and 1.

(b) Some Questions

Why don't we use target networks in PPO?

PPO avoids the use of target networks by employing a different approach for stabilizing policy updates. It uses a clipped objective function, ensuring that updates are not too far from the current policy, thus maintaining stability without needing a target network.

Is PPO an on-policy method? If yes, explain why. Furthermore, is sample complexity an issue for on-policy methods?

PPO is an on-policy algorithm, requiring data to be collected under the current policy for effective learning. This requirement often leads to higher sample complexity since experiences collected under an old policy are less useful, necessitating more frequent data collection for training.

(c) Implement the PPO Algorithm

Implement the PPO algorithm (shown in Algorithm 3) and solve the problem:

Implement the actor and critic network. Your algorithm solves the problem if you get an average total reward that is at least 125 points on average over 50 episodes. You are not allowed to change the environment or the reward signal. To verify that your policy solves the problem, check question (h).

We record the following output from the `PPO_check_solution.py` script

```
Policy achieves an average total reward of 212.3 +/- 31.7 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required, even though it had a really high bar this time around!

(d) Answer the Following

Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used. Motivate why you made those choices.

We state the network layout, used parameters and the choice of optimizer from the below pseudocode

```
ep = 0.2
M = 10
N_episodes = 1600
discount_factor = 0.99
critic_learning_rate = 1e-3
actor_learning_rate = 1e-5
...
optimizer_critic = torch.optim.Adam(network_critic.parameters(), lr=critic_learning_rate)
optimizer_actor = torch.optim.Adam(network_actor.parameters(), lr=actor_learning_rate)
```

```
...

# actor, input layer
input_layer = nn.Sequential(
    nn.Linear(input_size, 400),
    nn.ReLU()
)

# actor, mean value
mu = nn.Sequential(
    nn.Linear(400, 200),
    nn.ReLU(),
    nn.Linear(200, output_size),
    nn.Tanh()
)

# actor, covariance value
cov = nn.Sequential(
    nn.Linear(400, 200),
    nn.ReLU(),
    nn.Linear(200, output_size),
    nn.Sigmoid()
)

# critic
nn.Sequential(
    nn.Linear(input_size, 400),
    nn.ReLU(),
    nn.Linear(400, 200),
    nn.ReLU(),
    nn.Linear(200, 1)
)
```

Again we are mostly following the parameters that were given, and did not deviate from them as they appear to have worked really well.
The optimizers were again chosen to be Adam for both the critic and the actor, again because it works so well empirically. This time however we choose two different learning rates (due to instruction), but we accept that it makes sense to have the actor learn slower for robustness (as argued previously).
The layout of the networks was supplied to us again, with one big change being the complexity of the critic. Perhaps the rippling effect in the DDPG implementation would not have been as severe if it had more neurons. But DDPG may also be unfit for complex critics for unknown reasons. We use the common input layer between the mean and convariance estimators, as instructed. It makes sense that they would share some intrinsic qualities and logic, but also that they should deviate towards their respective heads. μ is constrained to $[-1, 1]$ appropriately since the actions space is again $[-1, 1]^2$, and the covariances of the outputs are appropriately positive.

Do you think that updating the actor less frequently (compared to the critic) would result in a better training process (i.e., more stable)?

No, we don't think updating the actor less frequently would result in a better training process. The simultaneous update strategy is a core part of PPO's design, focusing on balancing the policy improvement and stability. The clipped objective function in PPO inherently provides the stability needed in the learning process, making the need for staggered updates of the actor and critic less critical compared to DDPG.

(e) Once you have solved the problem, do the Following Analysis

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

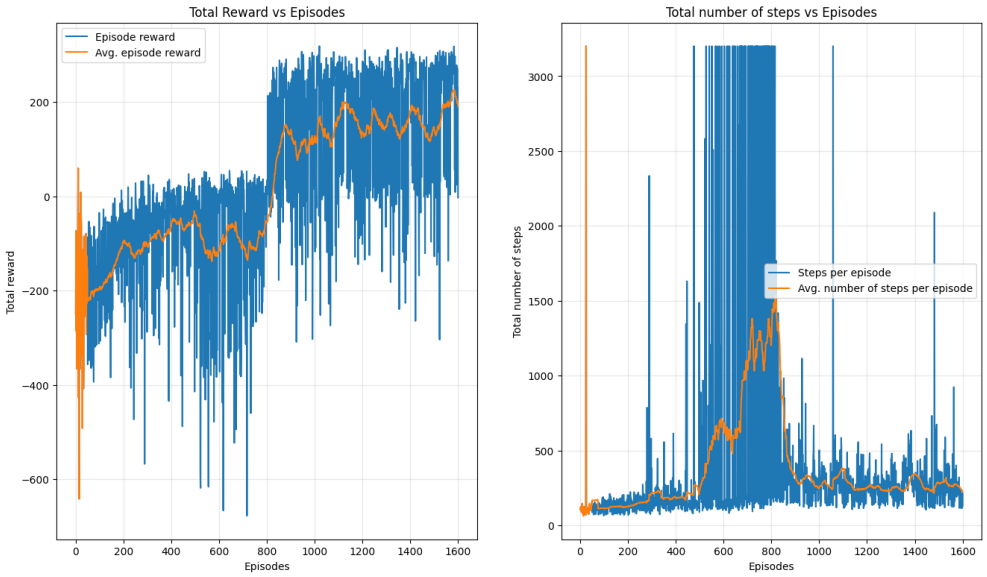

Figure 21: Training data for PPO with $\gamma = 0.99$

Similar to the previous two models we can see in Figure 21 that our agent is definitely learning seeing as the total reward increases with time. The steps per episode also follows a similar trend as previously, namely terminating early in the first episodes, then having increasing step size, suggesting that the agent is exploring the environment until the number of steps decrease and stabilizes again as the training progresses indicating that the agent is learning to achieve the goal more effectively.

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 $\gamma_2 < \gamma_0$. 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?

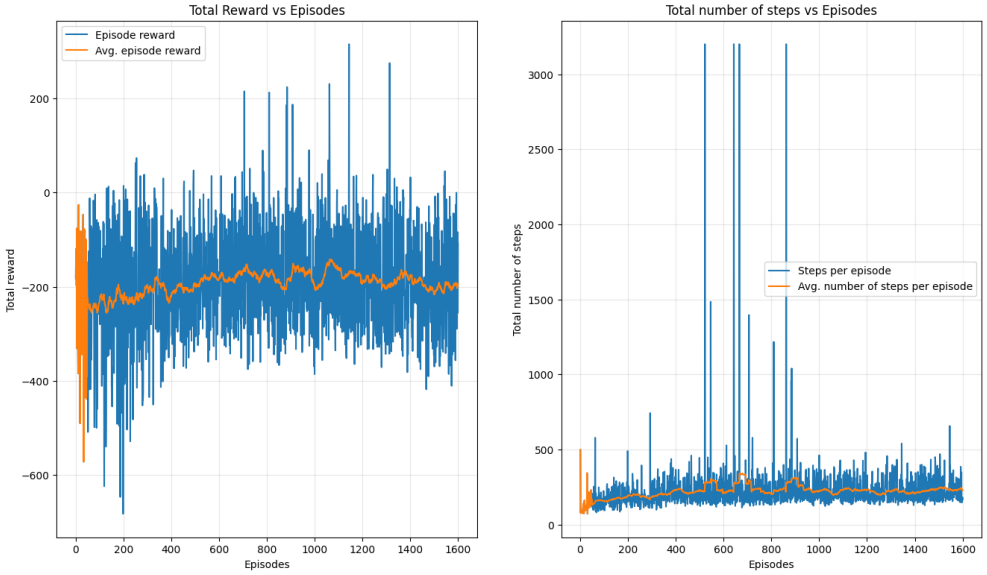

Figure 22: Training data for PPO with $\gamma_2 = 0.20$

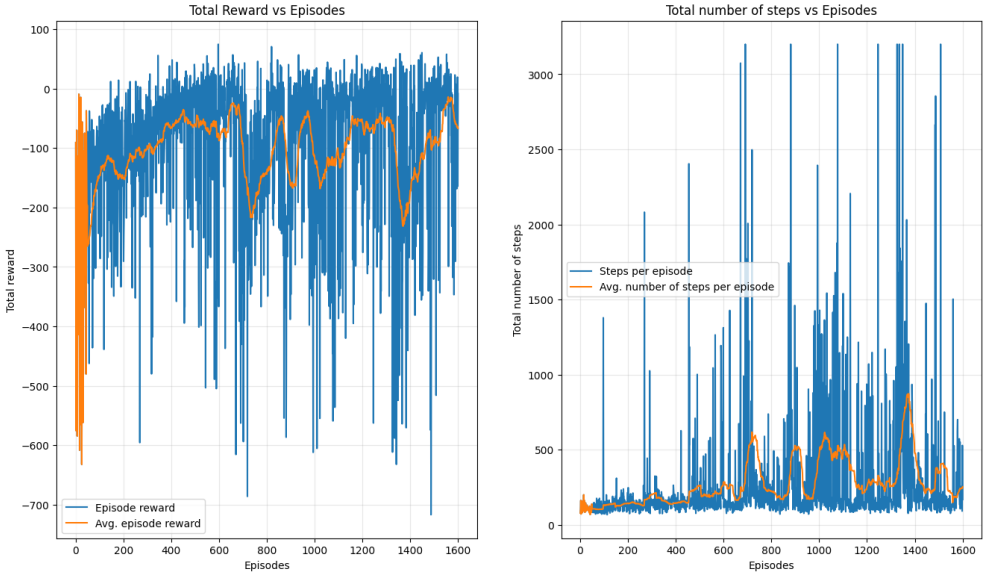

Figure 23: Training data for PPO with $\gamma_1 = 1.00$

Setting γ to both $\gamma_2 = 0.20$ and $\gamma_1 = 1.00$ results in unsuccessful training.
We can see that with a low discount factor of $\gamma_2 = 0.20$ does not seem to learn anything as it fails to explore the environment, indicated by the smaller number of average steps per episode. Combined with the consistent negative reward at approximately -200 the agent seems to continuously crash early on without exploring the space.

With a discount factor of $\gamma_1 = 1.00$ we can see that the agent also fails to achieve its goal. Similar reasoning as in the previous tasks this discount factor values immediate rewards as high as future rewards. This leads to the agent likely not learning how to land in time.

For your initial choice of the discount factor γ_0, investigate the effect of increasing/decreasing ϵ (show results for 2 different values of ϵ). Does training become more/less stable? Document your findings with relevant plots.

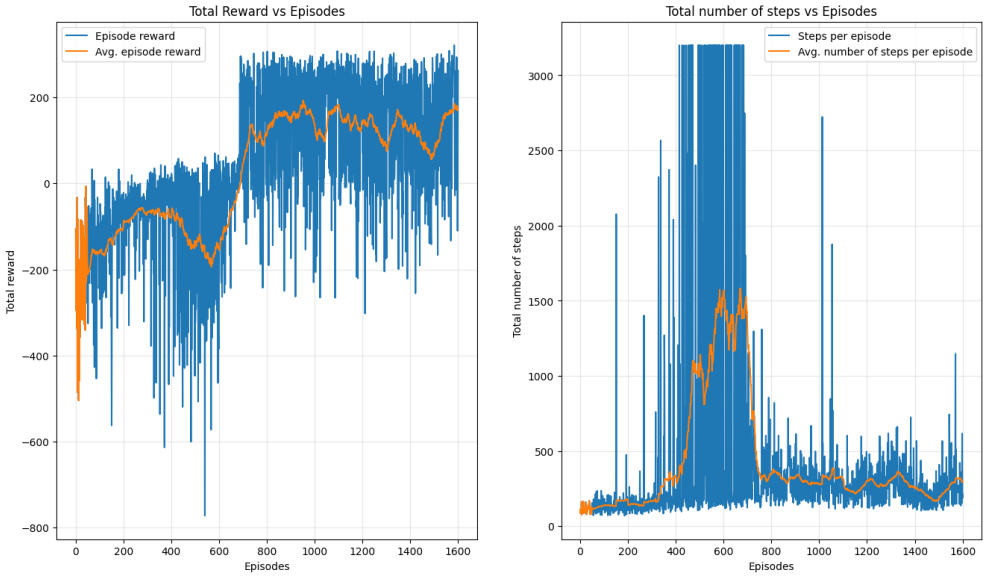

Figure 24: Training data for PPO with $\epsilon_1 = 0.20$

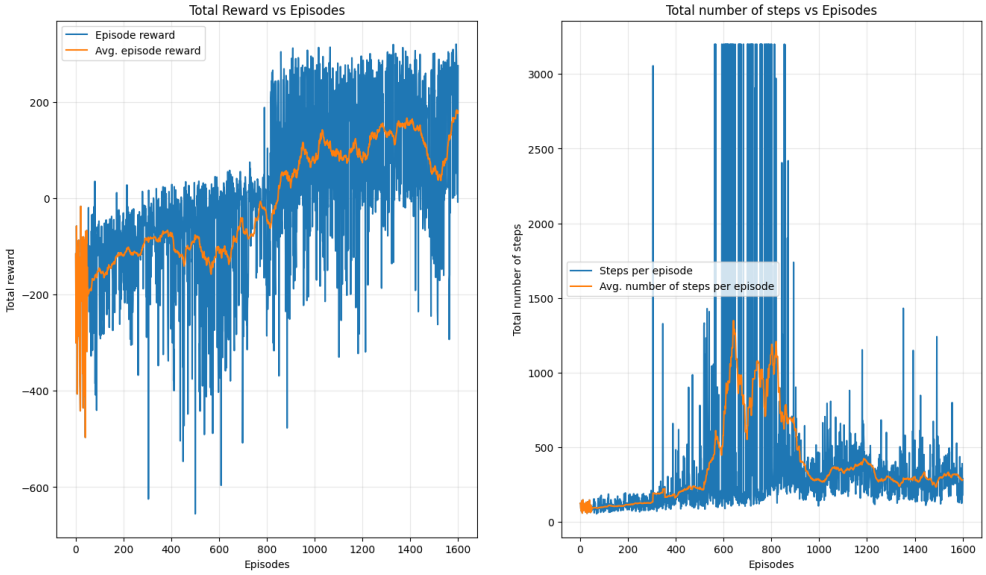

Figure 25: Training data for PPO with $\epsilon_2 = 0.50$

Varying the ϵ has less influence than we would have thought. As can be seen in Figure **??**. The plots look similar to our original one. The agent learns how to fly and land. This similarity might be because the

clipping values are rarely reached probability deltasß.

(f) For the Networks π_θ, V_ω that Solve the Problem

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 $V_\omega(s(y,\omega))$ 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.

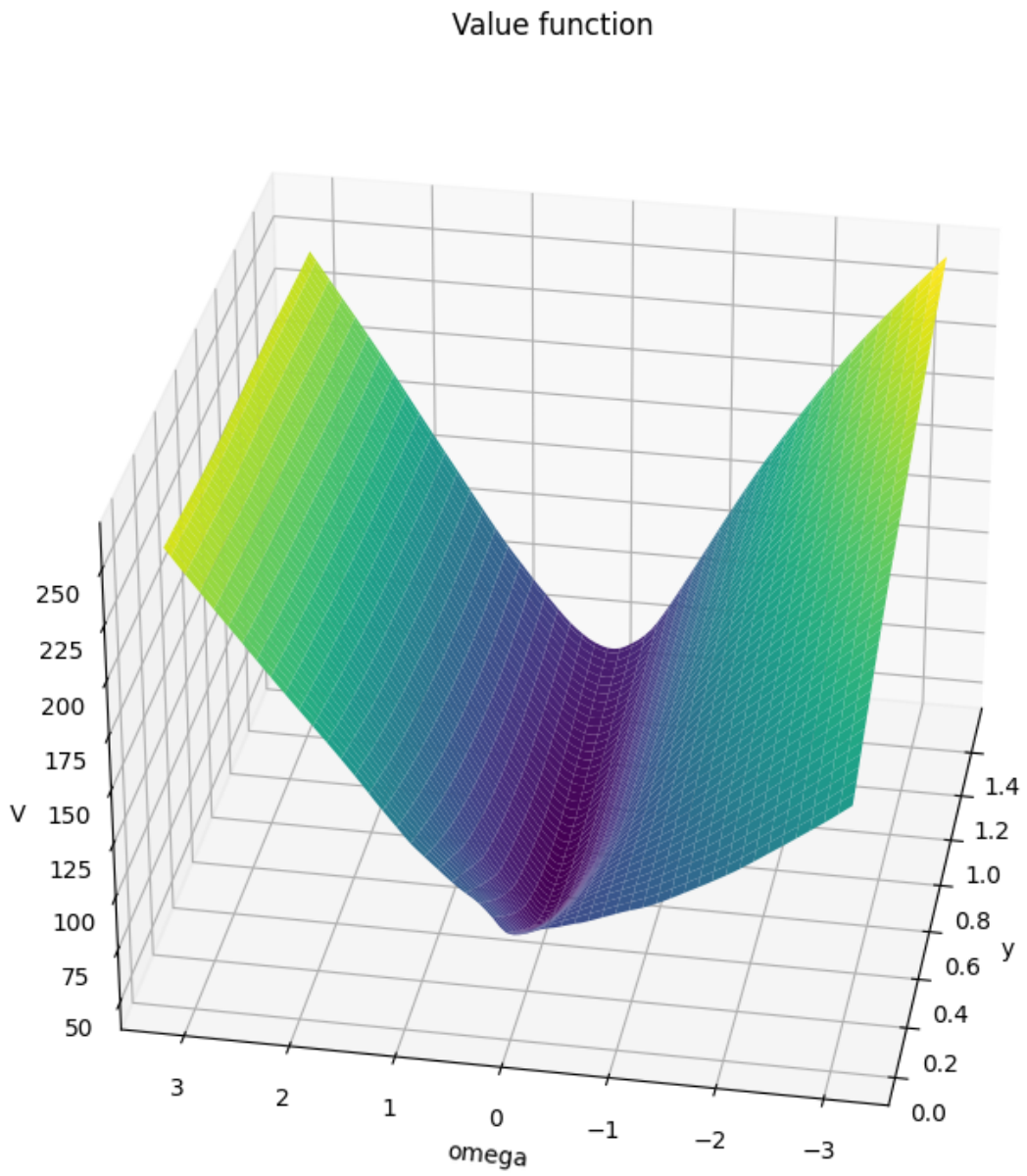

Figure 26: A plot over $V_\omega(s(y,\omega))$ for varying y and ω.

As can be assessed from 26, this network also has well defined behaviour across it's state space entire domain. It appears to have learned some trick around $\omega = 0$, $y = 0.4$, perhaps inaction as a virtue. Over-correcting close to the ground may result in a lot more accidents than necessary.

Let s be the same as the previous question. Remember that the action is a bi-dimensional vector, where the second element denotes the engine direction. Let the mean value of $\pi_\theta(s)$ be $\mu_\theta(s)$: plot the engine direction $\mu_\theta(s(y,\omega)))_2$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behavior of the optimal policy make sense? Explain it.

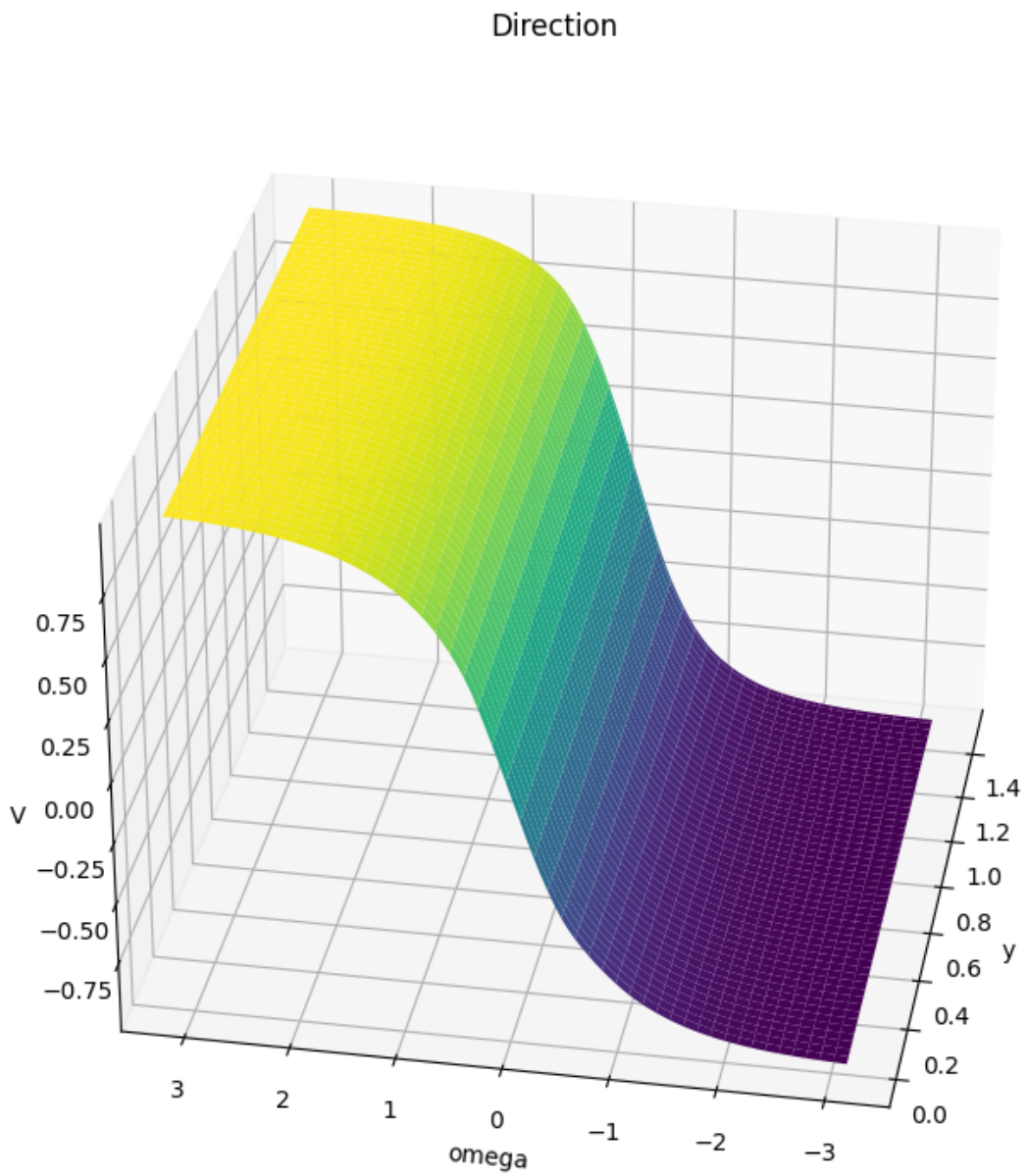

Figure 27: A plot over $\mu_\theta(s(y,\omega))_2$ for varying y and ω.

We see the engine direction as a function of the angle and height of the lunar lander in 27. This plot makes the most sense of any produced so far, as it is a perfectly smooth hyperbolic tangent, which makes for the most appropriate rule to make the lunar lander be perpendicular to the ground.

(g) Compare the Policy

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

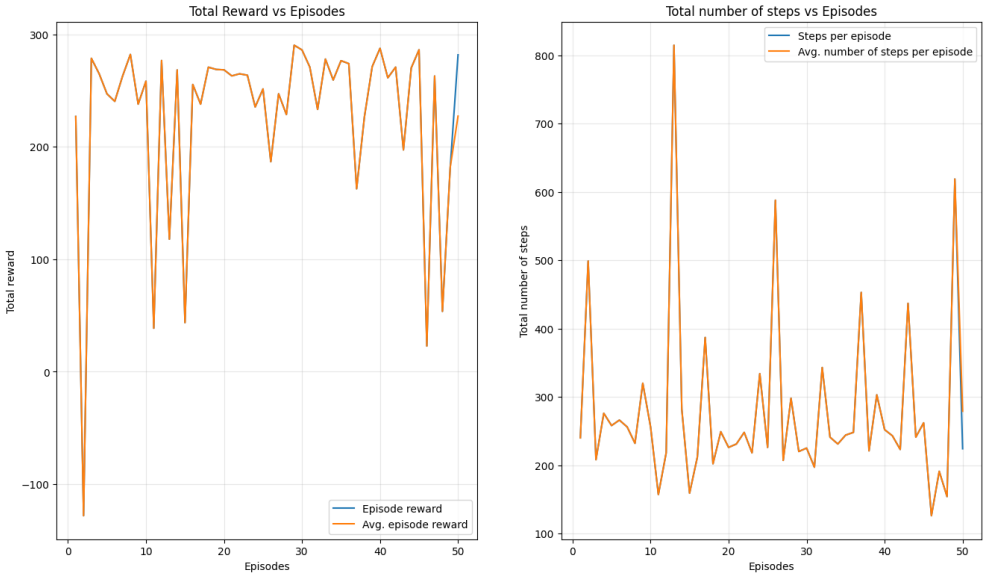

Figure 28: 50 episodes with our policy

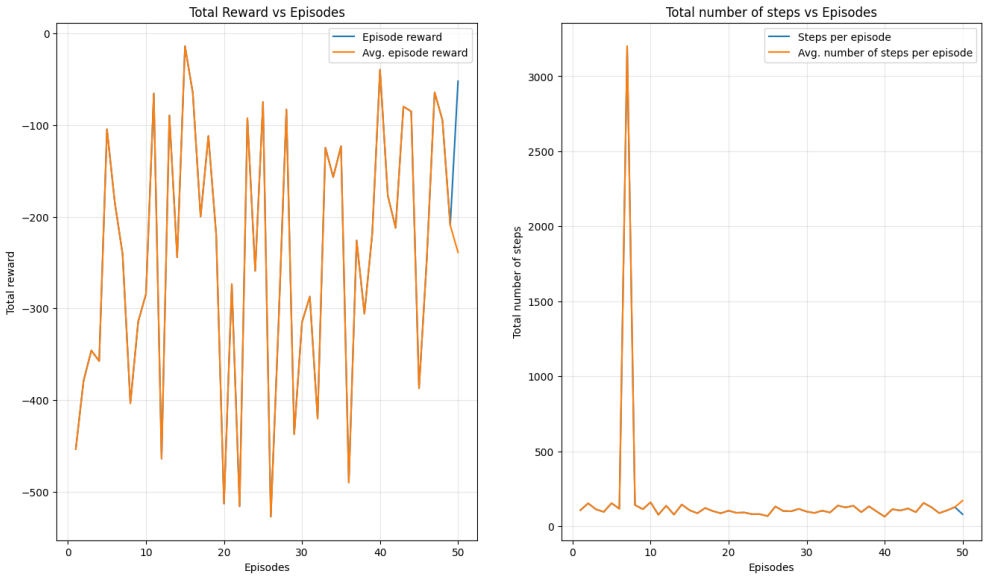

Figure 29: 50 episodes with random policy

We see that our policy vastly outperforms the random agent by comparing figures 28 with 29. It does however appear to have one unlucky accident, again (again).

Save the actor and critic network from (c) to two different files `neural-network3-actor.pth` **and** `neural-network-3-critic.pth`**. You can use the command** `torch.save(neural_net, path)` **to save a neural network. You can execute the command** `python PPO check solution.py` **to verify the validity of your policy.**

We record the following output from the `PPO_check_solution.py` script

```
Policy achieves an average total reward of 212.3 +/- 31.7 with confidence 95%.
Your policy passed the test!
```

We see that it goes above and beyond what is required, even though it had a really high bar this time around!

References