Bilkent University

Department of Computer Engineering

LunarLander with Deep Reinforcement Learning

CS 464: Introduction to Machine Learning

Final Report

Muhammed Naci Dalkıran 21601736

Enes Duran 21601449

Muhammet Rafi Çoktalaş 21601537

Nurullah Sevim 21601102

Kasım Sarp Ataş 21502006

Teaching Assistant: İlayda Beyreli

Instructor: Abdullah Ercüment Çiçek

Table of Contents

Introduction **2**
Environment & Agent 2
Value Based Learning & Q-Learning 3
Function Approximation 4
On-Policy vs Off-Policy 4
Exploration and Exploitation 5
Problem Description **5**
Lunar Lander Environment 5
Discrete and Continuous Action Spaces 6
Deep Q-Network (DQN) [1] 6
Double Deep Q-Networks 8
Deep Deterministic Policy Gradient (DDPG) [8] 8
Results **10**
Discussion **16**
DQN vs Double DQN 16
DDPG 16
DDPG vs Double DQN 17
Human vs Models 17
Conclusions **18**
References **19**
Appendix **20**
Individual Contributions 20

1. Introduction

Reinforcement learning (RL) is one of the areas in machine learning to make a sequence of decisions for continuous and discrete action spaces. The agent tries to achieve an implicit goal by maximizing the cumulative rewards. RL has different applications such as the autonomous car industry, stock price prediction etc. One of the RL applications is game like circumstances. Our project is also related with RL application on game-like circumstances, namely LunarLander[1] and LunarLanderContinuous[2] environments provided by OpenAI.

1.1. Environment & Agent

Reinforcement Learning is composed of interactions between an environment and an agent. Simply, the agent takes an action and this action affects the environment. Consequently, the agent gets a negative or positive reward, considering if it is getting close to or further from achieving its goal, i.e., the environment reinforces the agent. Thereafter, the state of the agent is updated by the environment.

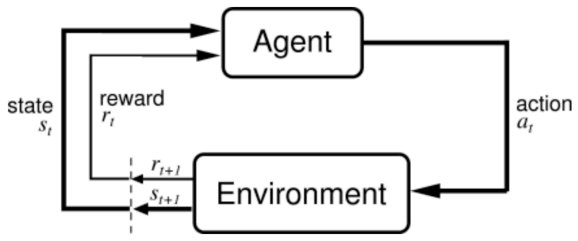


Figure 1: The Interaction of Agent and Environment

In our project, the agent is the lander and the environment is the Moon. The lander tries to land on the pad within a limited speed. In the discrete case the lander can take four actions which are moving up, right and left and doing nothing. If there is no action, the lander is solely under the effect of gravity. In the continuous case the agent returns an action vector of length two each element ranging from 0 to 1. The lander is rewarded according to the landing speed and its closeness to the landing pad. As a result of this reward, the RL algorithms optimize its further actions.

[1] https://gym.openai.com/envs/LunarLander-v2/
[2] https://gym.openai.com/envs/LunarLanderContinuous-v2/

1.2. Value Based Learning & Q-Learning

Value based learning is an RL approach in which the goal of an agent is to get the maximum expected future reward in each state. To achieve this goal, the agent tries to optimize the value function $v_\pi(s)$. The value function is the expected reward given the state the agent is currently in.

$$v_\pi(s) = [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]$$

Figure 2: The Value Function V(s)

Q-Learning is a value-based learning algorithm. In the Q-Learning, to optimize Q value, Q-function based on the Bellman equation (in Fig. 3) takes state s, and action a as inputs.

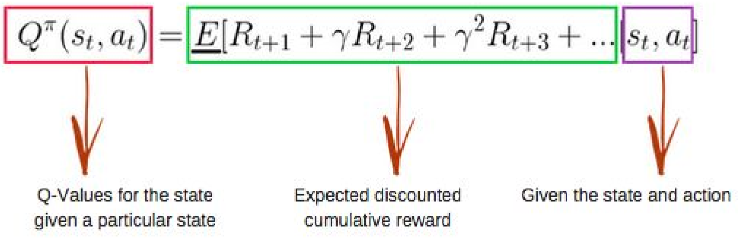
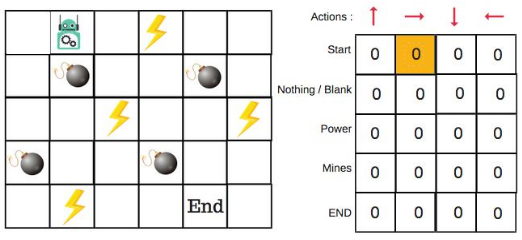

$$Q^\pi(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$
Q-Values for the state given a particular state
Expected discounted cumulative reward
Given the state and action

Figure 3: The Q-Function based on Bellman Equation

For the tabular case, the first process of the Q-learning algorithm is to initialize the Q-table. In the table, there are n columns where n is the number of actions and m rows where m is the number of states. The second process is the choosing and performing of the action. In this part of the process, the action has been taken and consequences of actions have been observed. Then, the next process is the evaluation of the action. Obtained values in the previous part are used to update Q-table and values in this part.



Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

Figure 4: The Q-Table

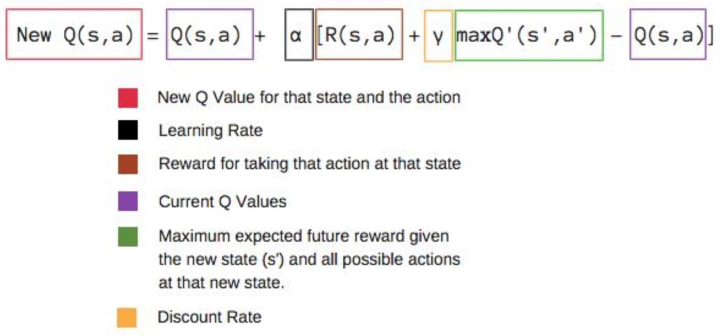

New Q(s,a) = Q(s,a) + α [R(s,a) + γ maxQ'(s',a') − Q(s,a)]
New Q Value for that state and the action
Learning Rate
Reward for taking that action at that state
Current Q Values
Maximum expected future reward given the new state (s') and all possible actions at that new state.
Discount Rate

Figure 5: Updating Q-Function Equation

In real world applications, the number of possible states is so much that it is infeasible to keep Q-values of each individual state. Therefore, we have to estimate the Q-value of each state with an approximator called function approximation.

1.3. Function Approximation

The aim of the function approximation is dealing with a vast state space. The process is that the features of each state are used to generalize the approximation of different states values; but include similar features. Actually, the approach does not find actual values; however, it approximates them. Thanks to this approach, the computation and generalization proceeds faster. The method using this approach is called function approximators. Function approximators are implemented utilizing neural networks.

1.4. On-Policy vs Off-Policy

To better explain on and off policies, target and behavior policies (exploitation and exploration) should be explained first. The agent updates Q values complying with target policy, $\pi(a|s)$. The behavior policy $b(a|s)$ indicates the policy used for data collection.

In this project the off-policy methodology is utilized as it is more sample efficient and more practical to implement. Also, if the algorithm approximates incorrectly, the data generated would also be incorrect if an on-policy algorithm was used; however, off-policy algorithms can deal with this circumstance thanks to using different target and behavior policies. Therefore, we will use two methods which use off-policy Q-learning algorithms: Deep Q-network (DQN) and Deep Deterministic Policy Gradient (DDPG).

1.5. Exploration and Exploitation

While exploring, the agent enhances its knowledge, exhausting the action space at every time. While exploiting, the agent greedily chooses the action that maximizes the reward. Exploitation may result in more reward in the short run but it may result in a sub-optimal behavior in the long run as it lacks the information to accurately estimate the reward thus converging in a local optima. The optimal approach is to balance the two in order to have an efficient and effective model.

2. Problem Description

2.1. Lunar Lander Environment

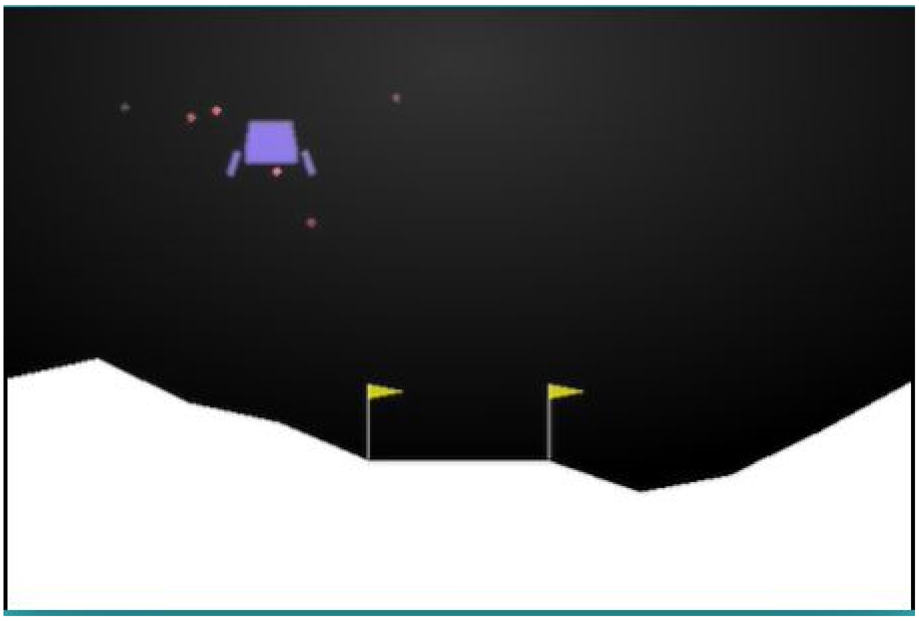


Figure 6: Lunar Lander Environment

Lunar Lander environment is provided by OpenAI. The environment contains a lander and a landing pad specified by two flags. The purpose in this environment is to land the lander in the landing pad with no velocity. Getting farther away from the landing pad causes a loss in the rewards that have been gained thus far. Landing on the pad results in between 100-140 points. There is an interval in this reward as the lander might have taken movements that move it away from the landing pad momentarily. If the lander crashes, it is given -100 reward points whereas having the landing results in rest is rewarded with 100 points. Moreover, the contact of each leg is rewarded with extra 10 points. The agent is equipped with 3 engines that work without any concern on fuel. There are two engines on the left and right sides of the agent and a main engine located at the bottom of it. Firing the main engine results in -0.3 reward per frame. A successful landing is a landing that has the total reward greater than 200.

2.2. Discrete and Continuous Action Spaces

When the action space is opted to be discrete, the agent can either use its engines at maximum capacity or at zero capacity. Therefore, the action space is limited to four actions including not taking an action, i.e., free fall.

When the action space is opted to be continuous, the agent can use its engines either at zero capacity or more than the engine's half capacity up to the maximum capacity. The agent decides the firing rate of main and side thrusters at each time step.

3. Methods

3.1. Deep Q-Network (DQN) [1]

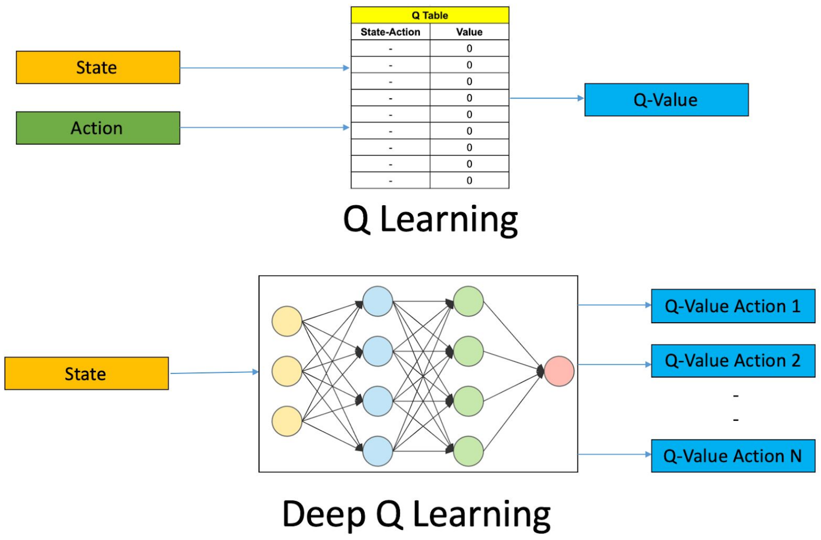


Figure 7: Illustration of DQN[2]

DQN is an off-policy Q-learning algorithm which is trained to estimate a Q-value function over a discrete action space. Estimation is done via the following function approximator:

- Behavior Q Network ($Q(s, a)$): Returns the Q value calculated using the state s and action a.

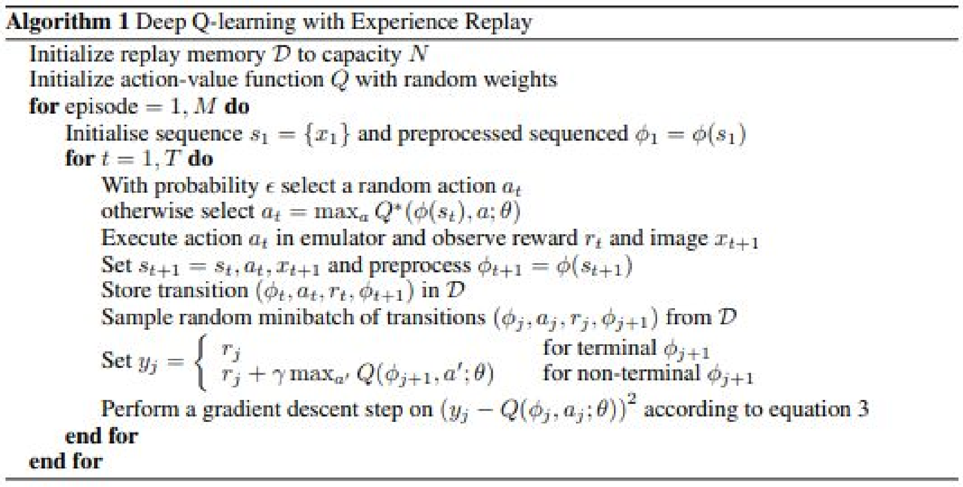
Behavior Q Network is used both for selection and evaluation of the next action.

The Q value update function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \bullet max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Action at a given time t, the next action is either the action that maximizes the Q value or any action. The former is associated with a probability of $1 - \varepsilon$ and the latter is associated with a probability of ε. This is to ensure that the agent balances exploitation and exploration. This is called greedy action selection [3].

Since the DQN algorithm uses Neural Network for function approximation, it requires a high number of experiments so as to perform at human-level [4]. To overcome this, the transitions are stored in a table for further use. This method is called experience replay [5]. A network based on the Q value update function is trained with mean squared error between predicted and target Q values. The label is provided via bootstrapping [6].



```
**Algorithm 1** Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

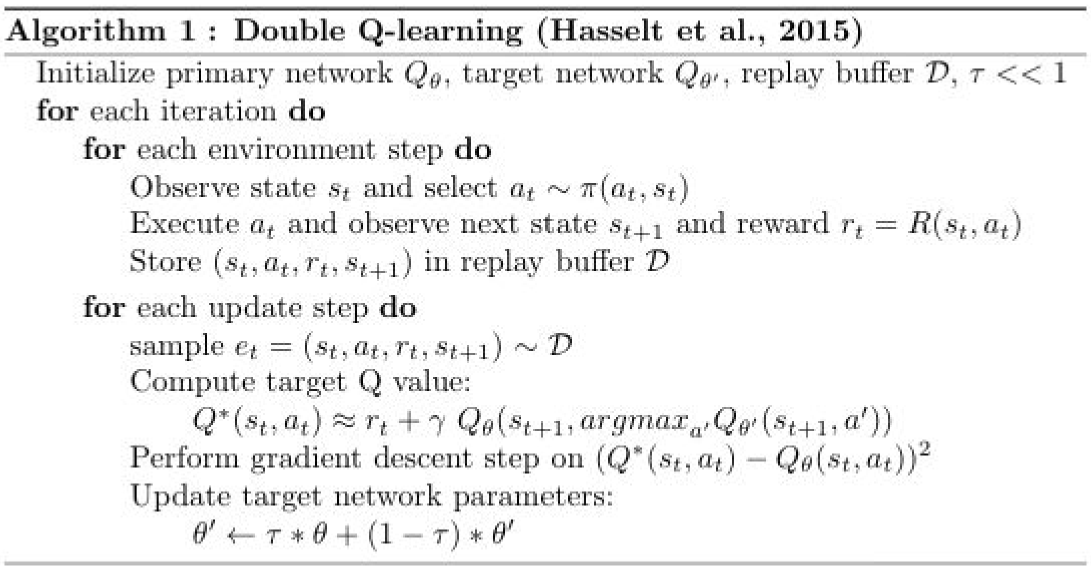
Figure 8: DQN Algorithm Pseudocode[4]

3.1.1. Double Deep Q-Networks

Double DQN is also an off-policy Q-learning algorithm that is highly similar to but an improved version of DQN. Estimation of Q function over a discrete action space is done via not one but two function approximators:

- Behavior Q Network ($Q(s,a)$): Returns the Q value calculated using the state s and action a.
- Target Q Network ($Q^*(s,a)$): It is used to control the estimation done by Behavior Q Network.

Unlike DQN, Behavior Q Network is used only for the selection of the next action. For the evaluation, Target Q Network is utilized.



Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

```
Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration do
    for each environment step do
        Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
        Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
        Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
    for each update step do
        sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
        Compute target Q value:
            $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
        Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
        Update target network parameters:
            $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$
```

Figure 9: Double DQN Algorithm Pseudocode[7]

The target update is done in each step according to a smoothing factor (a.k.a. rate of averaging) $\tau \in [0,1]$.

3.2. Deep Deterministic Policy Gradient (DDPG) [8]

DDPG algorithm is an off-policy policy based learning algorithm. In DDPG, actor-critic architecture consisting of actor and critic is used. Algorithm utilizes the actor to tune the parameter Θ to opt the best action for the current state via policy gradient. Algorithm utilizes the critic to evaluate the Q function approximated by the critic network by means of temporal difference (TD) errors.

Q values are approximated by the Critic network to be used to learn and update the policy of the Actor Network. The difference between DQN and DDPG is the type of action space. As mentioned before, In the DQN, action space is discrete; whereas, the action space is continuous in DDPG [3].

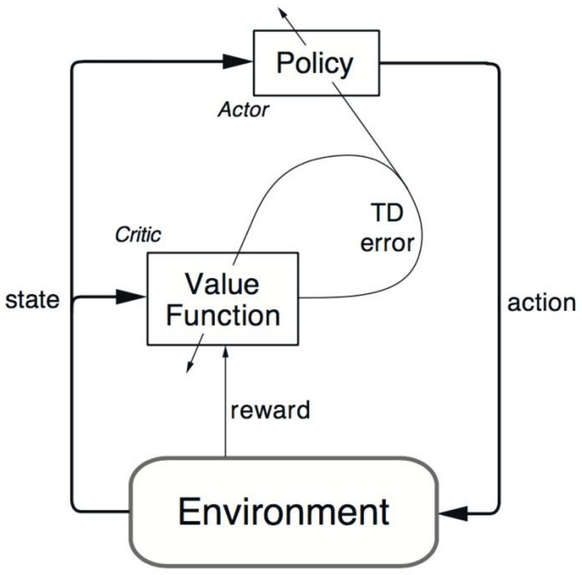
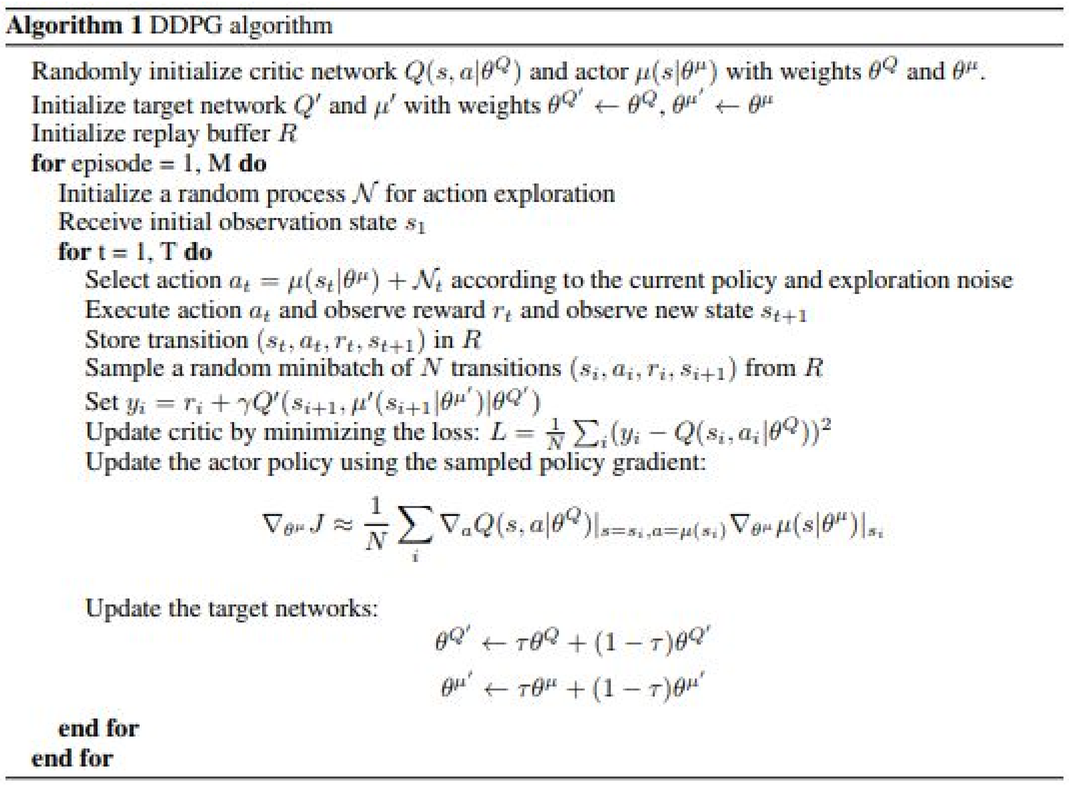


Figure 10: Actor-Critic Architecture [9]

Policy Function[9]: $\pi_{\Theta}(s,a) = \mathbb{P}[a|s,\Theta]$.

Temporal Differences Error [9]: $TD = r_{t+1} + \gamma V^{v}(s_{t+1}) - V^{v}(s_t)$

In TD , v indicates the policy actor has prefered. TD learning is very similar with Q-learning since it is a way to learn how value which depends on feature value of current state is predicted.



```
Algorithm 1 DDPG algorithm
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Figure 11: DDPG Algorithm Pseudocode[9]

In order for the agent to try more actions, i.e., explore better, a noise from a normal distribution is added. Additional noise and the fact that target networks are utilized make DDPG an off-policy algorithm. The target update is done in each step according to a smoothing factor (a.k.a. rate of averaging) $\tau \in [0, 1]$.

4. Results

Using the previously described algorithms, several experiments were conducted to answer the following questions:

- Can the agent exceed the average performance of a human?
- Does the agent behave consistently during the training?
- Can the agent perform better than the threshold performance defined by creators?

In order to have a sense of comparison among different algorithms, the experiments were conducted based on three Reinforcement Learning algorithms.

Each algorithm resulted in a distinct learning behaviour. The algorithms are DQN, Double DQN, and DDPG.

There are two modes in the environment that are activated distinctly during the development of the RL models: Train and Test modes. In the Train mode, the model weights are continuously being updated after each action taken by the agent. In this mode, the model is shaped according to the states and the amount of the rewards after any action. In the Test mode, the trained model is assessed where the weights are held constant i.e. do not get any update. The agent acts greedily in Test mode, meaning that it chooses the actions with maximum state action values. In addition to that the exploration noise is set to zero.

During the training process; the reward of each epoch, the failures and successes that the epochs resulted in were recorded. The weights were also recorded to use the best performing weights in the test mode. The training process had an early stop mechanism based on the following criterion: When the average episode reward of the last 30 episodes exceeds 220 points, the algorithm assessed to be successful and trained enough. Thus, after reaching the criterion, the training stops and the weights are recorded.

We also made a comparison between the trained models and average human performance. The average human performance is obtained by the group members. Each member controlled the lander manually and their actions got rewards based on the same parameters that the agent assessed. Note that, each member went through a tutorial process to simulate the training process of the agent for human subjects. The total reward curves for each group member have been obtained along with the agent’s curve and a visual comparison has been made based on the plot of these curves.

Firstly, we have conducted experiments on the DQN algorithm. The learning lasted for 600 episodes. The learning curves are depicted in Figure 9. As it can be seen in the figure, the rewards per episode is increasing for about 280 episodes. However, after this point the curve becomes unstable. The reward per episode drops suddenly at around episode 300 and after that point the algorithm behaves in a completely random manner. The standard deviation of the first 350 episodes turns out to be 216 where the standard deviation of the whole process is 278. This shows how unbalanced the algorithm becomes after a certain point.

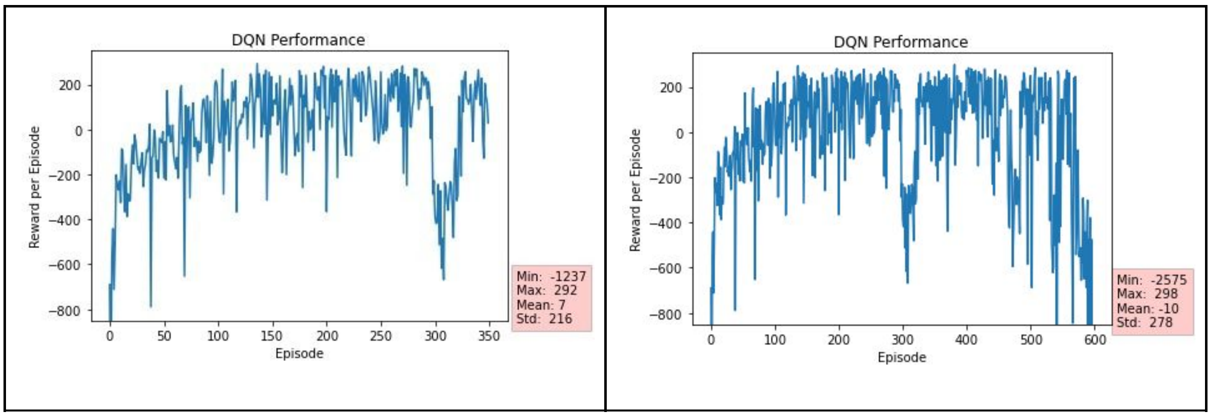


Figure 12: The learning curves of the DQN algorithm. The first 350 episodes depicted in the left plot and the whole process depicted in the right plot.

Considering the unstable behaviour of the DQN algorithm, we conducted the subsequent experiments on the Double DQN algorithm with the expectation of getting more consistent learning curves. The results for the Double DQN are depicted in Figure 10. The curves of this algorithm, as it can be seen as the figure, turn out to be more stable and the mean of the reward per episode increasing while the learning process continues. Meanwhile, the standard deviation shows no significant change as the agent performs better and better during the training.

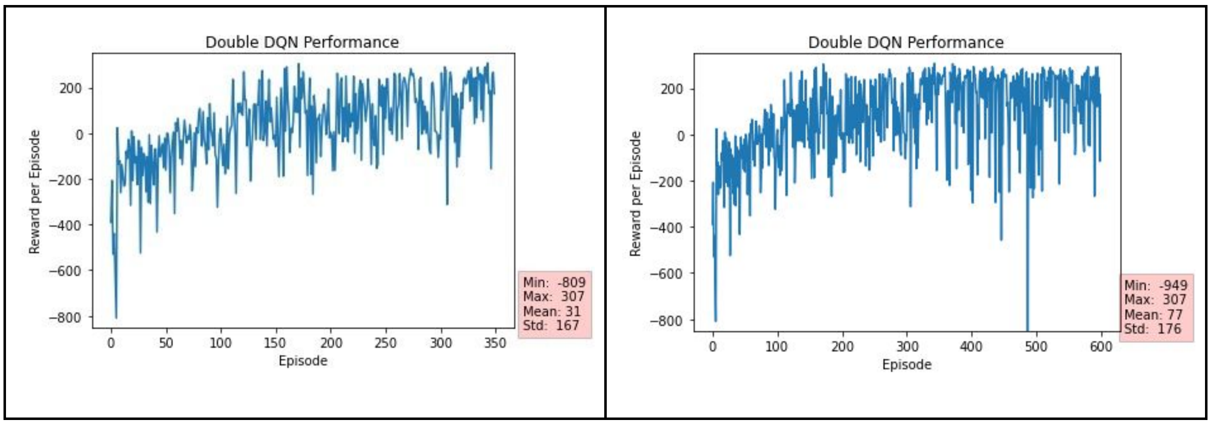


Figure 13: The learning curves of the Double DQN algorithm. The first 350 episodes depicted in the left plot and the whole process depicted in the right plot.

Lastly, an agent was trained with the DDPG algorithm. The DDPG algorithm was expected to be more successful than the other algorithms as the environment is more suitable for a DDPG algorithm. The learning curves are depicted in Figure 11. The initial rewards are worse than the other algorithms but overall the curve increases steadily and the eventual performance of the DDPG is the best one among all algorithms.

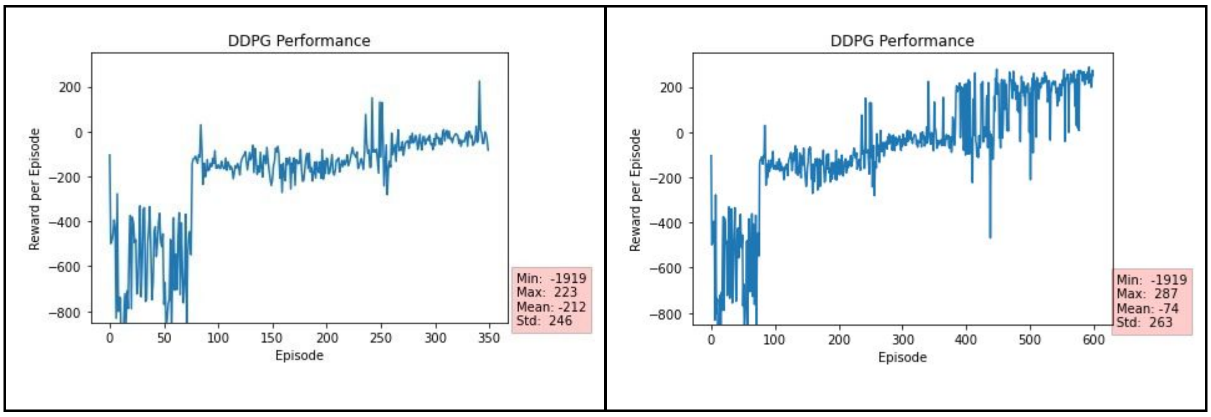


Figure 14: The learning curves of the DDPG algorithm. The first 350 episodes depicted in the left plot and the whole process depicted in the right plot.

Having completed the training process, with the best performing weights for all the algorithms, experiments for testing and evaluating the models were conducted. Note that, during the test process, the weights are not updated anymore. Since the DQN algorithm has failed to be stable, we used the Double DQN algorithm for testing purposes. In Figure 12, the reward per episode curve of the Double DQN algorithm was depicted. Test results in the figure seem to be quite variant, the standard deviation 96 which is very high.

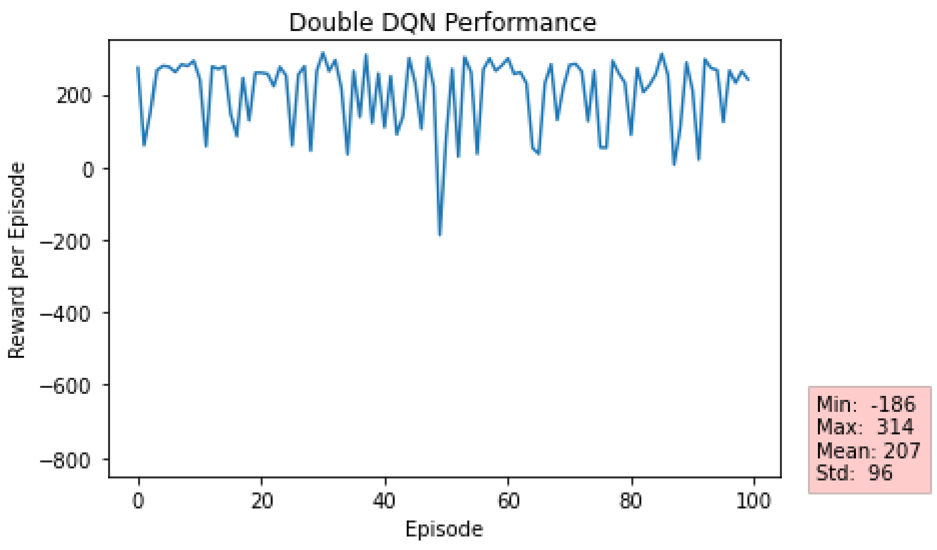


Figure 15: Reward per episode curve of Double DQN in Testing

Thereafter, the same test process was implemented for DDPG. The reward per episode curve is depicted in Figure 13. The results are more promising than the Double DQN's test results. Although there occurred some sudden drops during the testing process, the rewards are more stable than the previous test results with standard deviation of 69. Besides these, the mean of DDPG test results is higher than the Double DQN's even though the maximum reward for an episode is higher in Double DQN. Detailed results can be observed in the table on the next page.

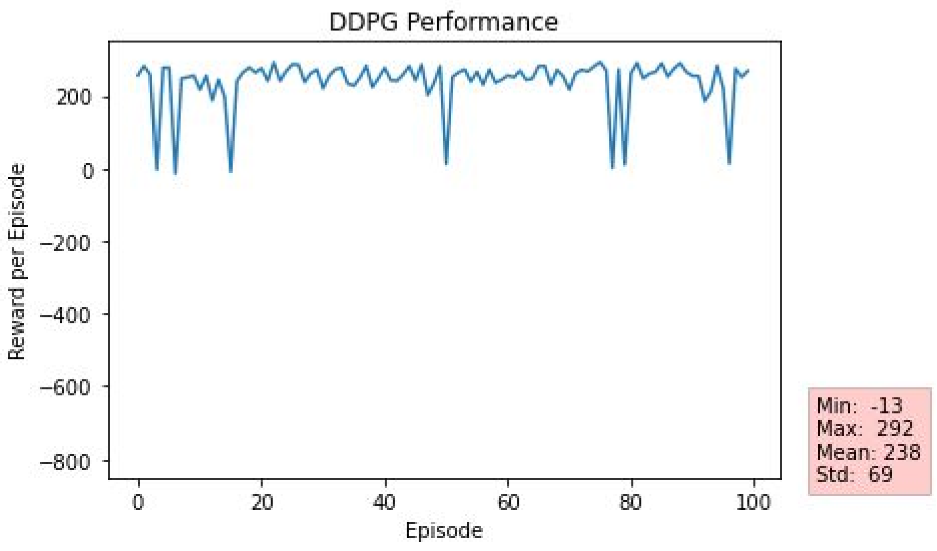


Figure 16: Reward per episode curve of DDPG in Testing

In Table 1 the performances of each group member are given with the RL algorithms' performances. Any human subject, during the experiments, could not succeed in any attempts. Furthermore, even the best performing group member's average reward is lower than the Double DQN's average by more than 200 points. The comparison between human and algorithms is also depicted in Figure 17. The performance difference is more clear from the figure where the algorithms' curves are higher than the best performing human subject's.

	Double-DQN	DDPG	Enes	Rafi	Nurullah	Sarp	Naci
Success Rate	0.70	0.86	0.0	0.0	0.0	0.0	0.0
Average Reward	207	238	-120	-65	-35	-10	-135

Table1: The success rate and average performances of RL algorithms along with each human subject

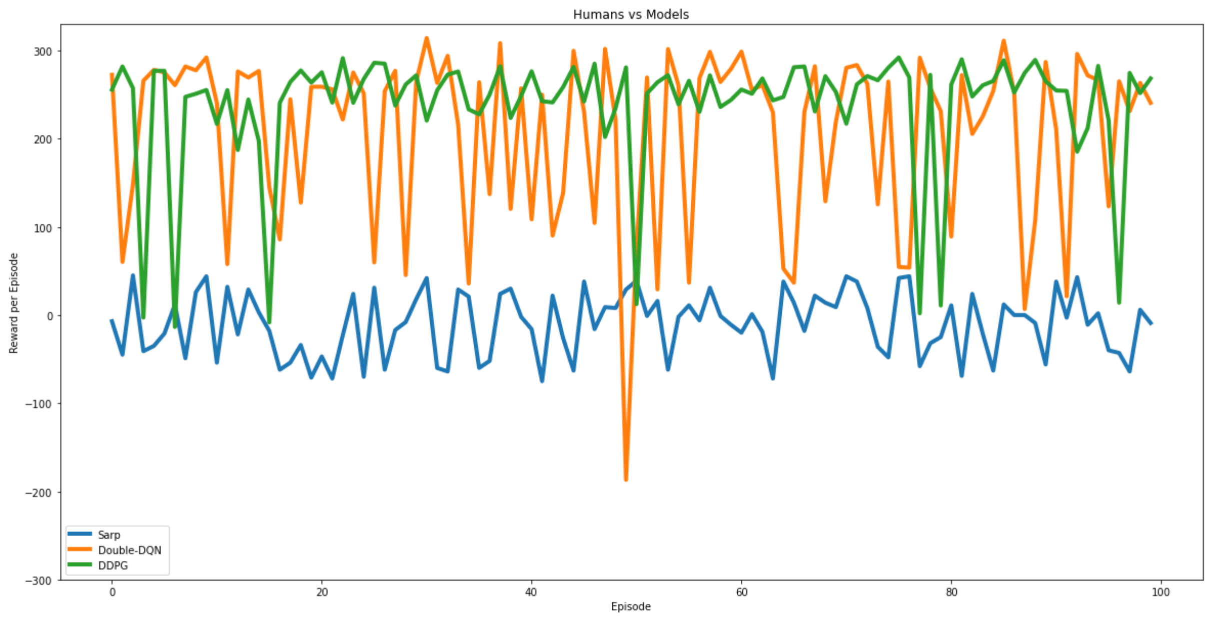


Figure 17: Reward per episode curves of RL models and the best performing human subject

5. Discussion

5.1. DQN vs Double DQN

After experimenting with DQN, as can be seen in figure 9, forgetting problems occur at around the 300th episode. The result of forgetting, its rewards drastically decreased and the model actually behaved like the initial episode. This is highly likely to be originated by the bootstrapping in the network update procedure. Therefore, we researched and decided to implement double DQN to deal with the forgetting problem.

After implementing Double DQN, the forgetting problem disappeared and the model worked more stable than DQN. Also, when looking at figure 9 and figure 10, Double DQN’s mean was much higher and the standard deviation was much lower compared to DQN.

Furthermore, first started with 350 episodes then after we saw both DQN’s and double DQN’s standard deviation, increased the episode number to understand how it will affect the result in the long run. Initially, DQN and Double DQN has minumum reward because of randomly determined weights; however, the interesting point is that when models reached 600 epoch, their minimum reward decreased -1237 to - 2575 for DQN and -809 to -949 for Double DQN. The environment's discrete action space might lead this circumstance since this does not happen in the DDPG model. Also, the minimum reward of DQN decreases twice; however, Double DQN is able to compensate for the decrease of its minimum reward.

As a result, Double DQN is more stable than DQN since it can deal with the forgetting problem and its standard derivation. Thus, we did not use DQN in the test process.

5.2. DDPG

Since environment action space is continuous, there is much more possibility to perform action than discrete action space. Therefore, initially, DDPG’s rewards, namely performance, are much worse than DQN and Double DQN. DDPG starts with a very low average reward but after the learning it goes more stable compared to both DQN and double DQN.

5.3. DDPG vs Double DQN

In figure 9&10, it can be seen that DDPG algorithm’s standard deviation is higher than Double-DQN; however, it is not reliable results since Initially, DDPG’s rewards are varying; however, after 100 episodes, its stability is much more better than DQN’s.

Also, Double DQN’s maximum and average reward is higher than DDPG’s. The reason for this circumstance might be associated with the continuous action space environment and low initial rewards of DDPG too. The reason for this prediction is based on testing process results.

In the figure 9 and 10, the rewards of models are shown until 350 and 600 episodes; however, when testing the models, each model was stopped after fulfilling the criteria which mentioned before. The plots in the figure 9 and 10 are important to see long run performance of the models not testing.

In table 1, which episode each model reaches the criteria is shown. According to the table, Double DQN fulfilled the criteria at 360 episodes; however, DDPG fulfilled the criteria at 610 and DQN at 601. After models , except DQN, fulfilled the criteria, Its weights were stored and testing process was conducted.

The results of the testing process are shown in the table 1, although Double DQN seems more suitable than DDPG for the task according to the training process, DDPG’s success rate is higher than DQN.

Another important aspect of this comparison is the sample efficiency of the algorithms. As seen from the graph, Double DQN learns visibly quicker than DDPG algorithm. This may be caused by the difference of the action spaces. Since Double DQN works in discrete action space the number of possible actions in a given state is restricted to four. However, in the continuous case the agent can take infinitely many actions. This makes DDPG less sample efficient.

5.4. Human vs Models

The game is very hard for humans. All team members play the games; but none of them get a success result (reward >= 200 for each episode). The reason for this circumstance is that even if the lunder is land between two poles, negative reward can be taken since the speed of the lunder, angle of the lunder, angle of the feet of the lunder etc affects the reward. Humans are not able to control all of the parameters for the reward; however, the models can.

6. Conclusions

RL is very suitable for basic video-game-like situations since It can deal with games better than humans. In our case, environment action space can be taken as continuous or discrete; therefore, we tried to successfully solve the task from both aspects. For discrete action space, DQN and Double DQN is used; for continuous one, DDPG is used.

The most challenging obstacle we encountered was the forgetting problem for DQN and we solve this problem by using Double DQN. Thanks to Double DQN, we can get better and more stable results for discrete action space. In this way, the model can compensate for reward if an unexpected situation, namely a large weight update, occurs.

In the training process DDPG is slower than others to fulfill the criteria; however, it gets a higher success rate than Double DQN in the testing process. Therefore, we can say that although training of DDPG is slower and its rewards in the training seem worse than Double DQN, DDPG is more suitable than Double DQN for the task after fullfill the criteria.

In implementing part, we learned the general structure of RL and neural networks; and also how we can deal with forgetting the problem. Also, we learned that RL models can go beyond humans' performance in the basic video games. Furthermore, we learned that although a model seems worse than others in the training, it can get better results in the test since to ensure the continuity of the rewards the model gets and stability is important for success of the system.

7. References

1. V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529--533, 2015.

2. Ankit Choudhary, “Deep Q-Learning: An Introduction To Deep Reinforcement Learning,” *Analytics Vidhya*, 27-Apr-2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. [Accessed: 15-Nov-2020].

3. Michel Tokic, “Adaptive ε-greedy Exploration in Reinforcement Learning Based on Value Differences,” *KI 2010: Advances in Artificial Intelligence.* 2010. [Online]. Available:http://tokic.com/www/tokicm/publikationen/papers/AdaptiveEpsilonGreedyExploration.pdf. [Accessed: 15-Nov-2020].

4. Mnih, Volodymyr *et al.*, “Playing Atari with Deep Reinforcement Learning,” 2013. [Online]. Available: https://arxiv.org/pdf/1312.5602.pdf. [Accessed: 15-Nov-2020].

5. D. Yang *et al.*, "Sample Efficient Reinforcement Learning Method via High Efficient Episodic Memory," in IEEE Access, vol. 8, pp. 129274-129284, 2020, doi: 10.1109/ACCESS.2020.3009329.

6. Kumar, Aviral *et al.*, “Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction,” 3-Jun-2019. [Online]. Available: https://arxiv.org/pdf/1906.00949.pdf. [Accessed: 15-Nov-2020].

7. Hasselt et al., “Deep Reinforcement Learning with Double Q-learning,” 22-Sep-2015. [Online]. Available: https://arxiv.org/pdf/1509.06461.pdf. [Accessed: 15-Nov-2020].

8. “Deep Deterministic Policy Gradient,” *Deep Deterministic Policy Gradient - Spinning Up documentation*. [Online]. Available: https://spinningup.openai.com/en/latest/ algorithms/ddpg.html.[Accessed:16-Nov-2020].

9. H. (S. Kung-Hsiang, “Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG),” *Medium*, 16-Sep-2018. [Online]. Available: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287. [Accessed: 16-Nov-2020].

10. H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” *arXiv.org*, 08-Dec-2015. [Online]. Available: https://arxiv.org/abs/1509.06461. [Accessed: 20-Dec-2020].

8. Appendix

8.1. Individual Contributions

Enes Duran: Worked on the implementation of DQN and DDPG algorithms together with their hyperparameter tunings and completed the implementation with parameter tuning and trained the RL models until the results are satisfactory. Evaluated the hyperparameters for both algorithms.

Muhammed Naci Dalkıran: Worked on implementation of the DQN algorithm and trained the RL model until the results were satisfactory. Evaluated the hyperparameters for both algorithms.

Muhammet Rafi Çoktalaş: Worked on implementation of the DQN algorithm and trained the RL model until the results were satisfactory. Benchmarked the performance metrics of the algorithms in the following criteria: success rate of the agent, learning pace and total reward value of the agent

Nurullah Sevim: Worked on the implementation of DDPG algorithm and completed the implementation with parameter tuning and trained the RL model until the results were satisfactory. Evaluated the hyperparameters for both algorithms.

Kasım Sarp Ataş: Worked on implementation of the DQN algorithm and trained the RL model until the results were satisfactory. Benchmarked the performance metrics of the algorithms in the following criteria: success rate of the agent, learning pace and total reward value of the agent.

All team members have worked on web-based literature review and selected the algorithms.