BEN-GURION UNIVERSITY OF THE NEGEV

FACULTY OF ENGINEERING SCIENCES

# Deep Reinforcement Learning

Assignment 1: From Q-Learning to Deep Q-learning (DQN)

By: Or Simhon, Ofir Ben Moshe

ID: 315600486, 315923151

Mail: orsi@post.bgu.ac.il, ofirbenm@post.bgu.ac.il

Lecturer: Gilad Katz, Ph.D.

Nov 2021

# Contents

# 1. Section 1 – Tabular Q Learning

### 1.1 Why methods such as Value-Iteration cannot be implemented in such environments? Write down the main problem.

The value iteration method relies on achieving the optimal policy $\pi^*(s)$ directly from the optimal value function $v_*(s)$ (in a greedy way) which is obtained iteratively using the Bellman optimality equation. In order to use these equations, the model of the environment is needed, namely $p(s', r|s, a)$ as well as the reward function. Since these functions are not supplied in such environment, algorithms such Value-Iteration cannot be implemented properly.

### 1.2 How do model-free methods resolve the problem you wrote in previous question? Explain shortly.

To deal with the problem of unknowing information about the environment, model-free methods use the state-action value function $q(s, a)$ instead, by sampling the environment and exploring which action from which state will return higher return in the long run. Note that the value function $v(s)$ can be explored by sampling, but the optimal policy can only be derived from the state-action one.

### 1.3 What is the main difference between SARSA and Q-learning algorithms? Explain shortly the meaning of this difference.

The main difference between these algorithms is visible in the update statements for each technique. Q-Learning technique is an **Off Policy** and uses the greedy approach to update the Q-value. SARSA technique, on the other hand, is an **On Policy** and uses the action that will be performed in the next step by the current policy for computing the Q target.

### 1.4 Why is it better than acting greedily (choosing an action with $argmax_a Q(S', a)$)?

When the transition probability matrix is unknown (Model-Free RL), there is a need to explore the environment based on trial-and-error samples. In these cases, if using the greedy action selection method, the agent may be converging to a **non-optimal policy**, since he did not explore enough the environment. To prevent this situation, we want that the agent will balance between exploring the environment and trying new behaviors, to exploiting the knowledge gained from history. This can be achieved by choosing actions using $\epsilon - greedy$ way instead of greedily.

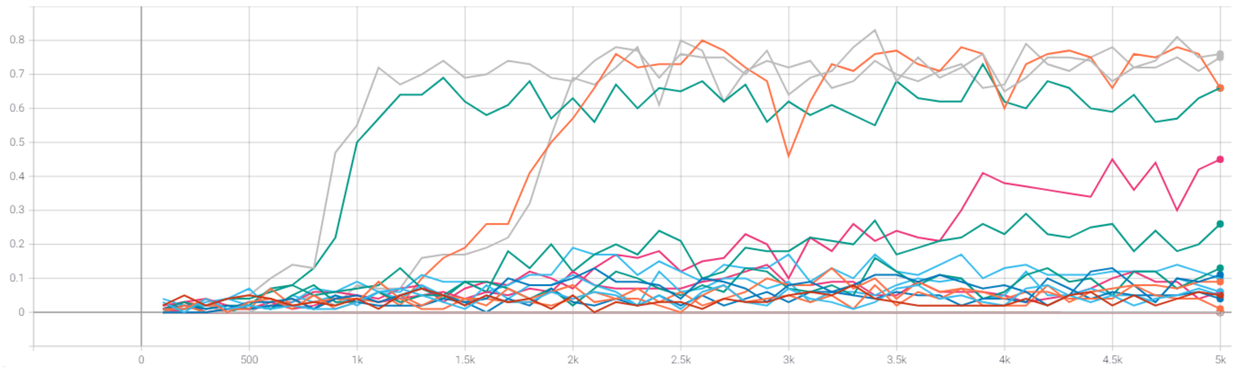
## 1.5 Python Implementation

In order to optimize the hyper-parameters for the optimal results, comparisons between different combinations were done which were studied using tensorboard graphs. The optimization function called "Optimize_Q_Learning" and the tested parameters are following.

- learning_rate $(\alpha) = [0.1, 0.01, 0.001, 0.0001]$
- discount_factor $(\gamma) = [0.9, 0.99, 0.999, 0.9995]$
- $epsilon\_decay\_schedule = [\text{linear}, \text{exp}]$
- $epsilon\_decay\_factor = [0.99, \ 0{,}999, 0.9995]$

In the figure below, the results of part of the checked trainings are shown.

Mean reward in the last 100 episodes



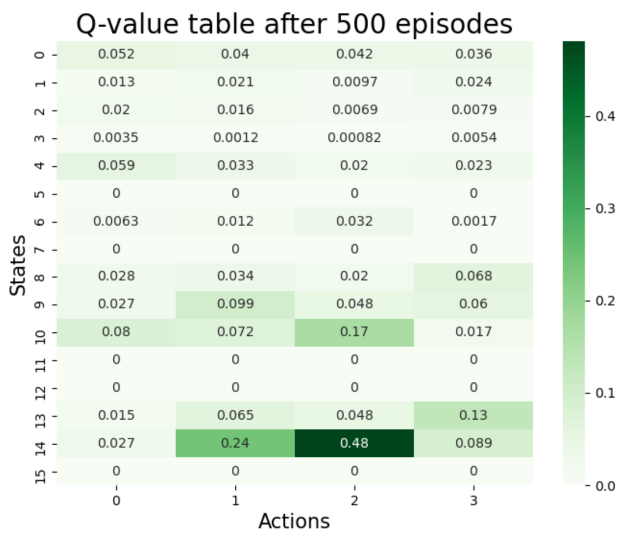
By checking all these combinations, the best results were achieved by the following values:

| Hyper-Parameter | Final |
|---|---|
| $\alpha$ | 0.1 |
| $\gamma$ | 0.999 |
| $\epsilon_{decay\_schedule}$ | linear |
| $\epsilon_{decay}$ | 0.999 |

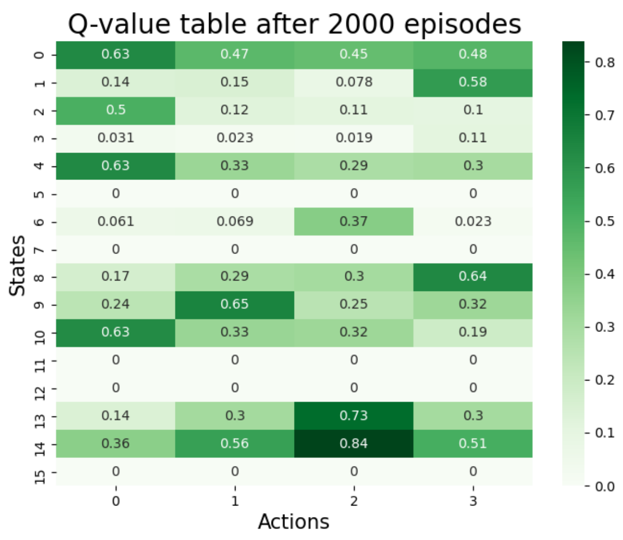
The linear decay rate for decaying epsilon-greedy probability was expressed this way:

$$\epsilon_{episode\_i} = \max\{0.001, 1 - (\epsilon_{decay} \cdot i)\} \qquad (1)$$

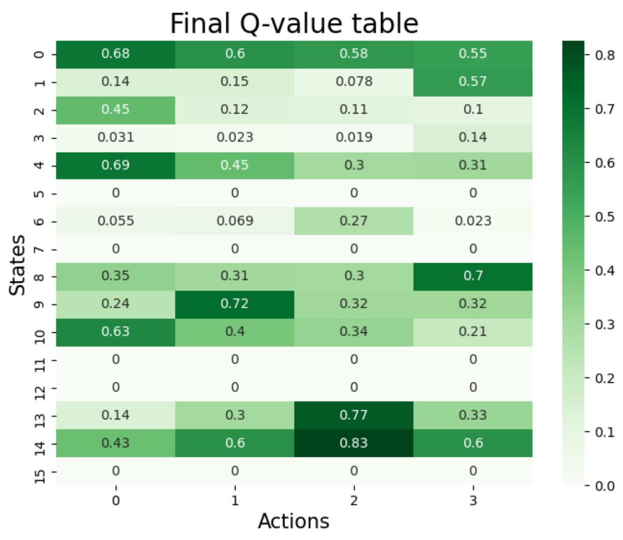
## Q – Learning Tables



| States | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0.052 | 0.04 | 0.042 | 0.036 |
| 1 | 0.013 | 0.021 | 0.0097 | 0.024 |
| 2 | 0.02 | 0.016 | 0.0069 | 0.0079 |
| 3 | 0.0035 | 0.0012 | 0.00082 | 0.0054 |
| 4 | 0.059 | 0.033 | 0.02 | 0.023 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0.0063 | 0.012 | 0.032 | 0.0017 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0.028 | 0.034 | 0.02 | 0.068 |
| 9 | 0.027 | 0.099 | 0.048 | 0.06 |
| 10 | 0.08 | 0.072 | 0.17 | 0.017 |
| 11 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 |
| 13 | 0.015 | 0.065 | 0.048 | 0.13 |
| 14 | 0.027 | 0.24 | 0.48 | 0.089 |
| 15 | 0 | 0 | 0 | 0 |

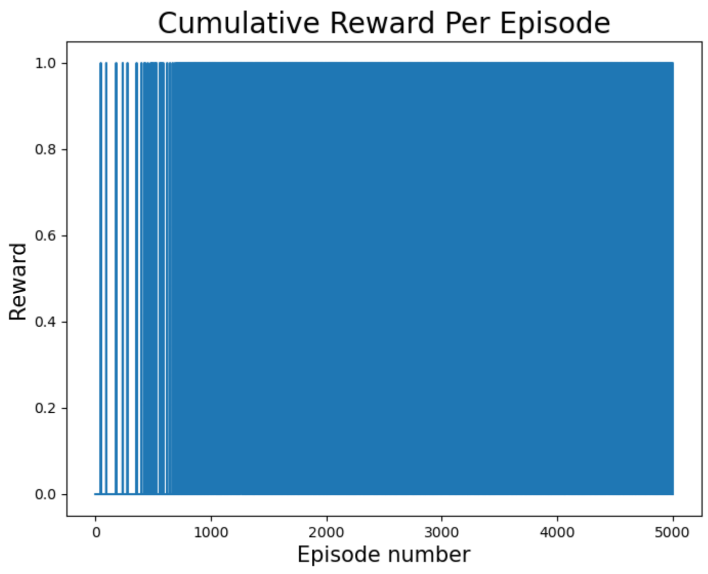


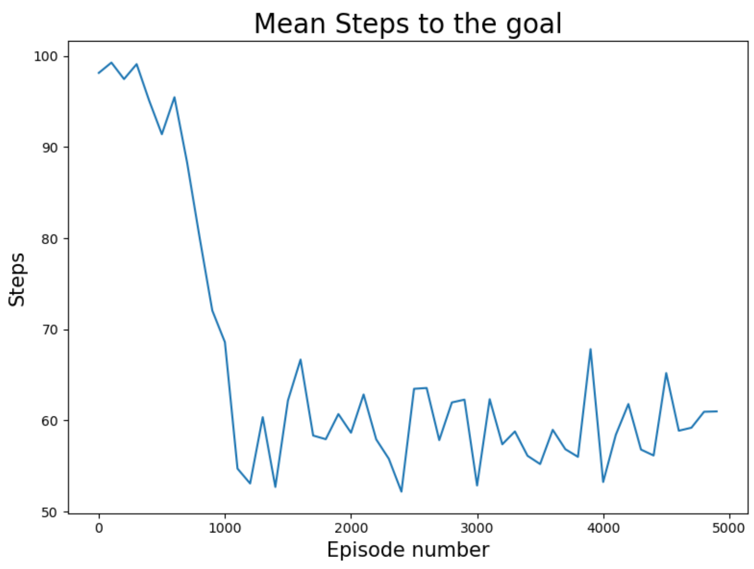
| States | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0.63 | 0.47 | 0.45 | 0.48 |
| 1 | 0.14 | 0.15 | 0.078 | 0.58 |
| 2 | 0.5 | 0.12 | 0.11 | 0.1 |
| 3 | 0.031 | 0.023 | 0.019 | 0.11 |
| 4 | 0.63 | 0.33 | 0.29 | 0.3 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0.061 | 0.069 | 0.37 | 0.023 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0.17 | 0.29 | 0.3 | 0.64 |
| 9 | 0.24 | 0.65 | 0.25 | 0.32 |
| 10 | 0.63 | 0.33 | 0.32 | 0.19 |
| 11 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 |
| 13 | 0.14 | 0.3 | 0.73 | 0.3 |
| 14 | 0.36 | 0.56 | 0.84 | 0.51 |
| 15 | 0 | 0 | 0 | 0 |

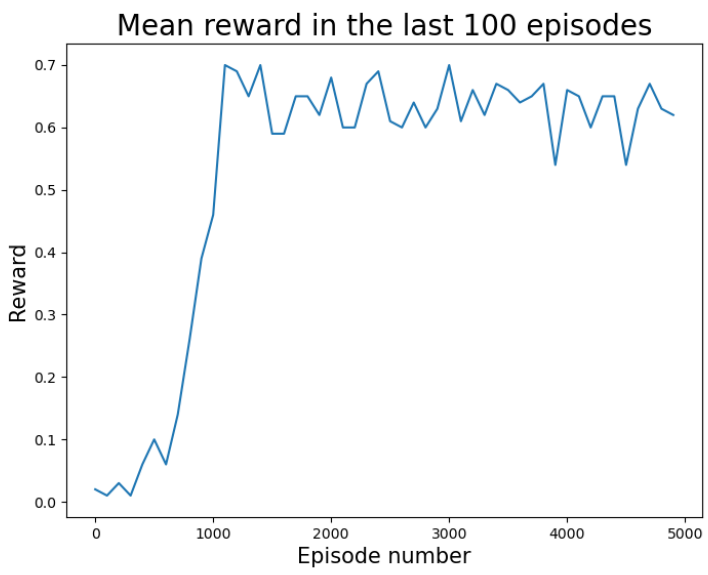


| States | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0.68 | 0.6 | 0.58 | 0.55 |
| 1 | 0.14 | 0.15 | 0.078 | 0.57 |
| 2 | 0.45 | 0.12 | 0.11 | 0.1 |
| 3 | 0.031 | 0.023 | 0.019 | 0.14 |
| 4 | 0.69 | 0.45 | 0.3 | 0.31 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0.055 | 0.069 | 0.27 | 0.023 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0.35 | 0.31 | 0.3 | 0.7 |
| 9 | 0.24 | 0.72 | 0.32 | 0.32 |
| 10 | 0.63 | 0.4 | 0.34 | 0.21 |
| 11 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 |
| 13 | 0.14 | 0.3 | 0.77 | 0.33 |
| 14 | 0.43 | 0.6 | 0.83 | 0.6 |
| 15 | 0 | 0 | 0 | 0 |

**Q-learning Figures**







## 2. Section 2 – Deep Q-learning

### 2.1 Experience replay - Why do we sample in random order?

The Experience replay technique is used to deal with two main problems. The first is forgetting past learned behavior, such a thing can happen if the environment changed and previous scenarios no more occur. The second problem is the correlation between states, which is naturally high when training sequentially. In summary, this technique helps the model to converge and generalize better.

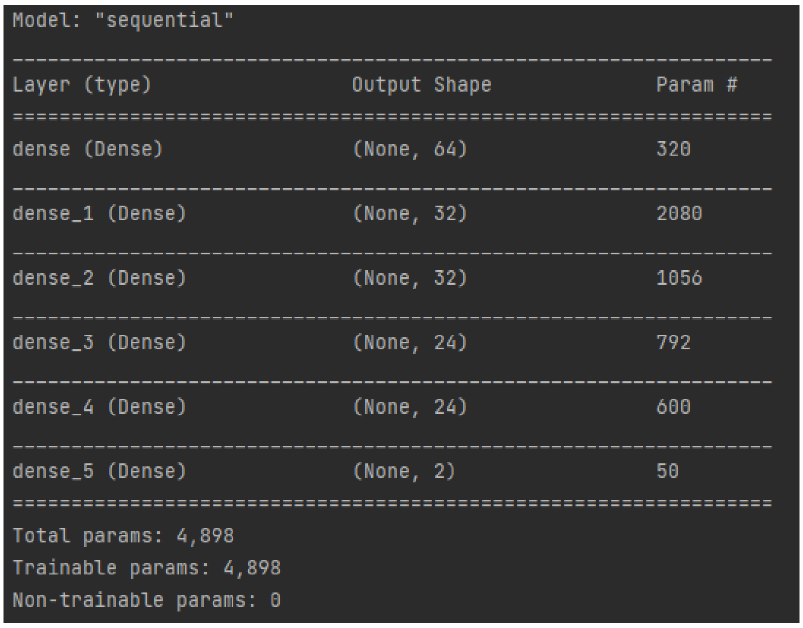
### 2.2 Use an older set of weights to compute the targets - How does this improve the model?

By freezing the target's network weights and optimizing the learned network, we prevent chasing a moving target, which can increase instability and add variance to the training process. Freezing the target's network weights gives the model time to improve the Q-values of the learned net, and then to estimate better the Q-values.

### 2.3 Hyper-parameter and number of hidden layers

In order to optimize the hyper-parameters (HP) for the optimal results, first different combinations of the HP were checked manually both for a model with 3 hidden layers and 5 hidden layers. Each training were constraint with a number of 1000 max episodes.

In terms of hidden layers, we saw that for the most the 5 hidden layers model converge faster. The final Neural Network Architecture consist of 5 hidden layers with ReLU activation function and linear activation. Below the summary from Keras.

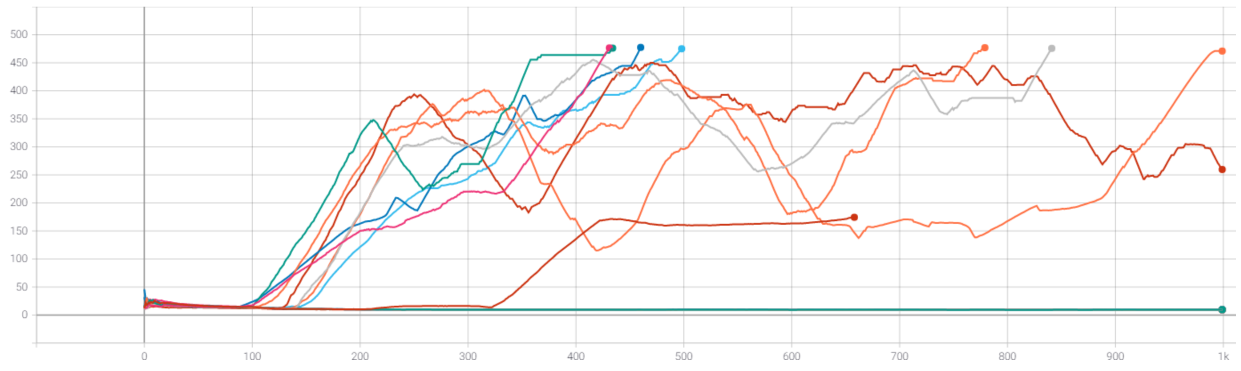


```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                320
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 24)                792
_________________________________________________________________
dense_4 (Dense)              (None, 24)                600
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 50
=================================================================
Total params: 4,898
Trainable params: 4,898
Non-trainable params: 0
```

In terms of HP, after finding values that supply a decent convergence, comparisons between different combinations were done and afterward studied using tensorboard graphs.
The tested HP combinations include all of the following combinations.

- learning_rate ($\alpha$) $= [0.0001, 0.00001]$
- discount_factor ($\gamma$) $= [\,0.99, 0.999]$
- $batch\_size = [64, 128]$
- $target\_update\_period = [100, 200]$
- $\epsilon_{max} = 1$
- $\epsilon_{min} = 0.01$
- $\epsilon_{decay} = 0.999$
- $Buffer\_Capacity = 10{,}000$

In the figure below we can see the mean reward in the last 100 consecutive episodes for all of the cases studied.

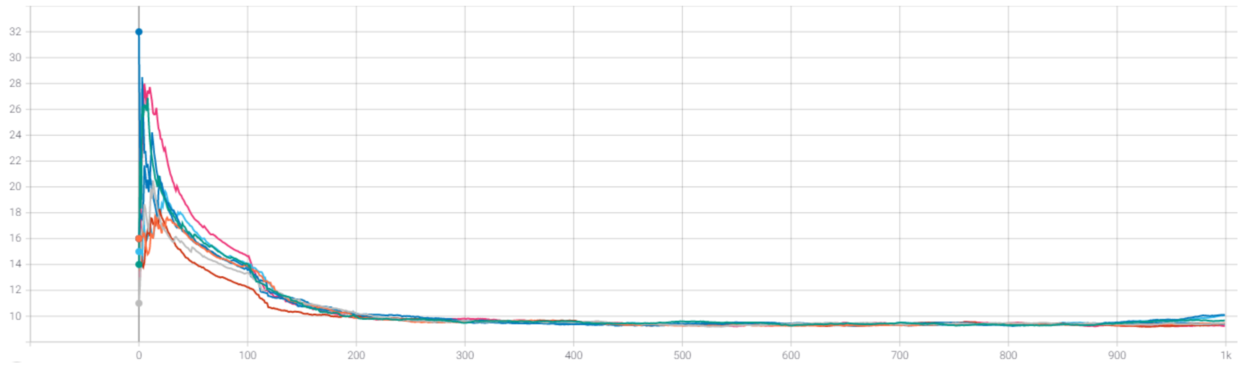


Now, we will discuss the parameters which affected the most the convergence time.

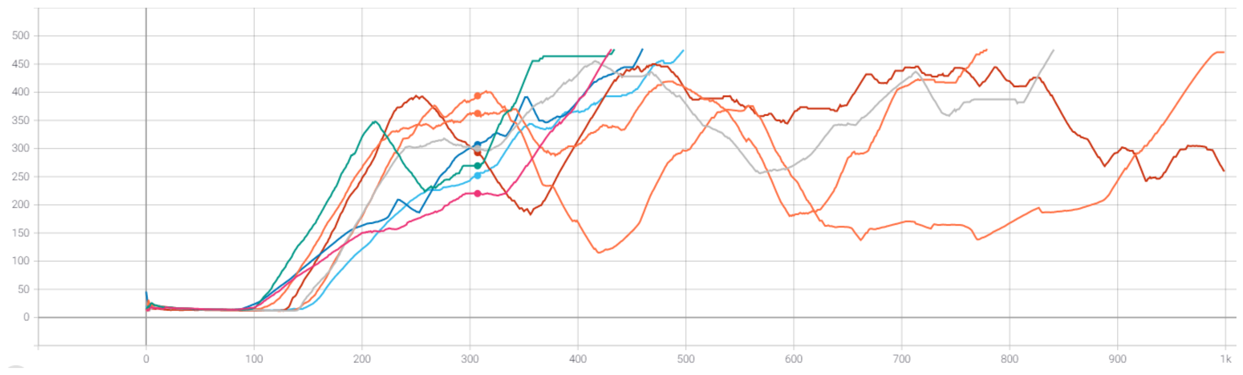
### 2.3.1 Learning rate ($\alpha$) comparison

From the results obtained, we can see that the learning rate has a large effect on convergence. It seems that $\alpha = 0.00001$ is too small a value, resulting in no one of the runs converging in that case. On the other hand, for the value of $\alpha = 0.0001$ all runs converged.
The results are shown in the figures below.

<u>Mean reward in the last 100 episodes - $\alpha = 0.00001$</u>



Mean reward in the last 100 episodes - $\alpha = 0.0001$

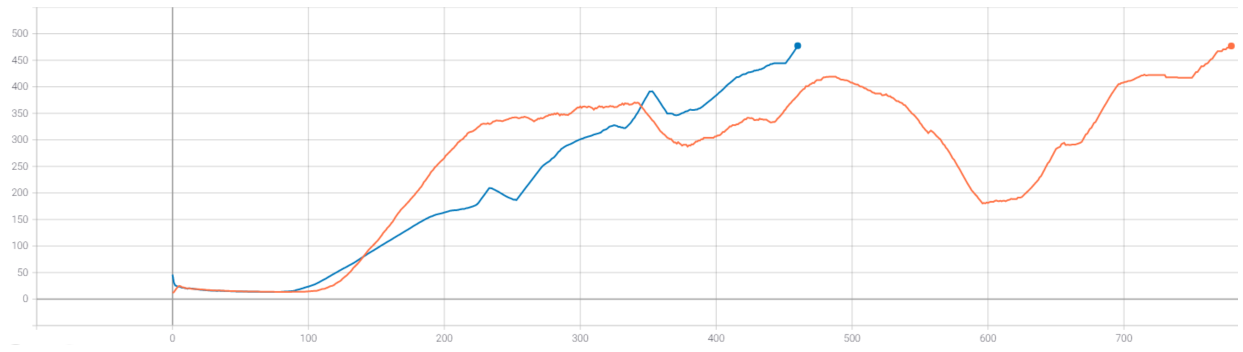


### 2.3.2 Discounted factor ($\gamma$) comparison

From the results obtained, we can see that this parameter has a large effect on convergence rate, although the algorithm converged for both of the cases. For the checked values, $\gamma = 0.99$ obtained better results.

Mean reward in the last 100 episodes - $\alpha = 0.0001$

- $\gamma = 0.99$ in blue
- $\gamma = 0.999$ in orange



## 2.4 Final Hyper-parameters

The Hyper-parameters which led us to the best execution is shown in the table below.

| Hyper-Parameter | Final |
|---|---|
| $\alpha$ | 0.0001 |
| $\gamma$ | 0.99 |
| $batch\_size$ | 128 |
| $target\_update\_period$ | 100 |
| $Buffer\_Capacity$ | 10,000 |
| $\epsilon_{max}$ | 1 |
| $\epsilon_{decay\_schedule}$ | Exponential |
| $\epsilon_{min}$ | 0.01 |
| $\epsilon_{decay}$ | 0.999 |

Average reward of at least 475.0 over 100 consecutive episode

For the selected Hyper-parameters, the challenge of getting an average reward of 475.0 at least in consecutive 100 episodes was accomplished after **431 episodes**

Figure of the average Reward in the previous 100 episodes

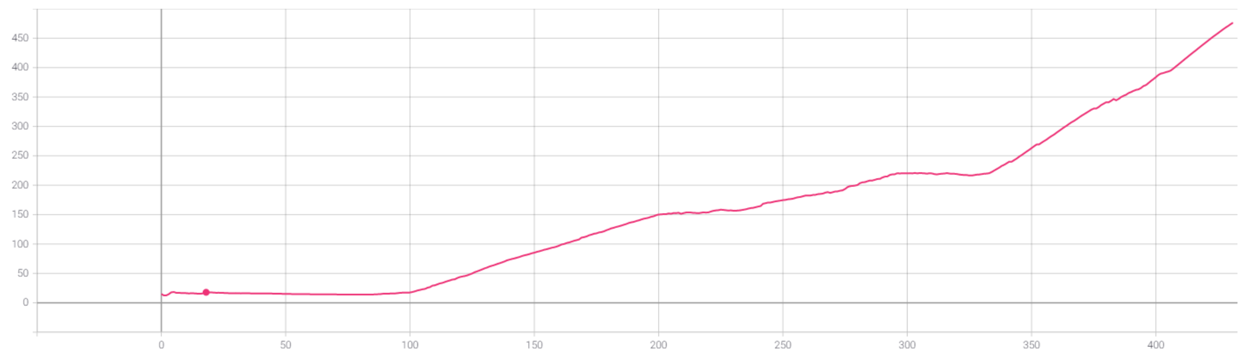


Figure of the Loss per steps

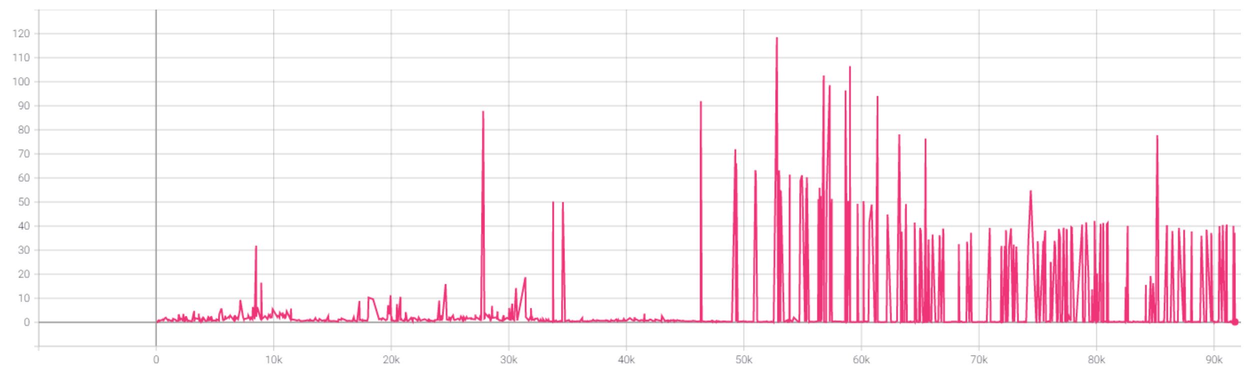
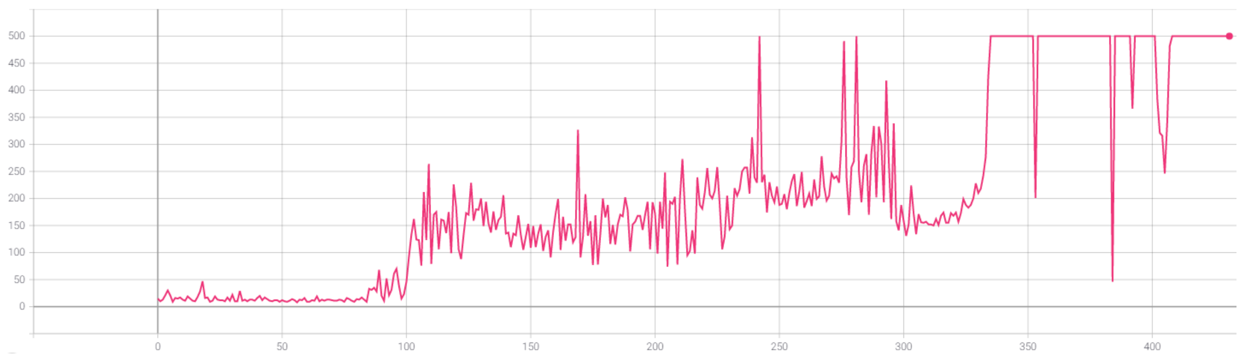


Figure of the Reward per episode



# 3. Section 3 – Improved DQN

## 3.1 Improved DQN – Double Dueling DQN

This algorithm is a combination of two separate algorithms that use to improve the traditional DQN, Double DQN, and Dueling DQN.

The Double DQN algorithm handles the problem of the overestimation of Q-values, which is occur when sub-optimal actions are initially given higher Q-values. The algorithm computes the Q target using two networks to decouple the action selected from the target Q value generation. When implementing we use our DQN network (A) to select the best action to take for the next state, and our target network (B) to calculate the target Q value of taking that action at the next state. The equation above describe it mathematically.

$$\underbrace{Q^A(s,a)}_{TD-target} = r(s,a) + \gamma \cdot Q^B\left(s', \underset{a'}{\operatorname{argmax}}(Q^A(s',a'))\right) \tag{2}$$

The Dueling DQN improvement deal with the idea that the importance of choosing the right action is not equal across all states. This method decouples the action from the state by defining the Action-Value function $Q(s,a)$ as a combination of the Value function $V(s)$ and the Advantage function $A(s,a)$ which in simple words, the last describe how much better is to take action a versus all other possible actions at some state. To implement this, there is a need to only change the way our NN's build. Mathematically, the Q function is calculated by the following equation.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right) \tag{3}$$

## 3.2 Hyper-parameters values in the final solution

In order to find the Hyper-parameters which lead to the best results, and to be able to compare this algorithm to DQN in a decent way, all of the presented HP combinations from section 2 were checked also here. After studying the results using tensorboard graphs, the following HP were selected.

| Hyper-Parameter | Final |
|---|---|
| $\alpha$ | 0.0001 |
| $\gamma$ | 0.999 |
| $batch\_size$ | 128 |
| $target\_update\_period$ | 100 |
| $Buffer\_Capacity$ | 10,000 |
| $\epsilon_{max}$ | 1 |
| $\epsilon_{decay\_schedule}$ | Exponential |
| $\epsilon_{min}$ | 0.01 |
| $\epsilon_{decay}$ | 0.999 |

Average reward of at least 475.0 over 100 consecutive episodes

Using the final Hyper-parameters from section 2, the challenge of getting an average reward of 475.0 at least in consecutive 100 episodes was accomplished after **181 episodes**

Figure of the average Reward in the previous 100 episodes

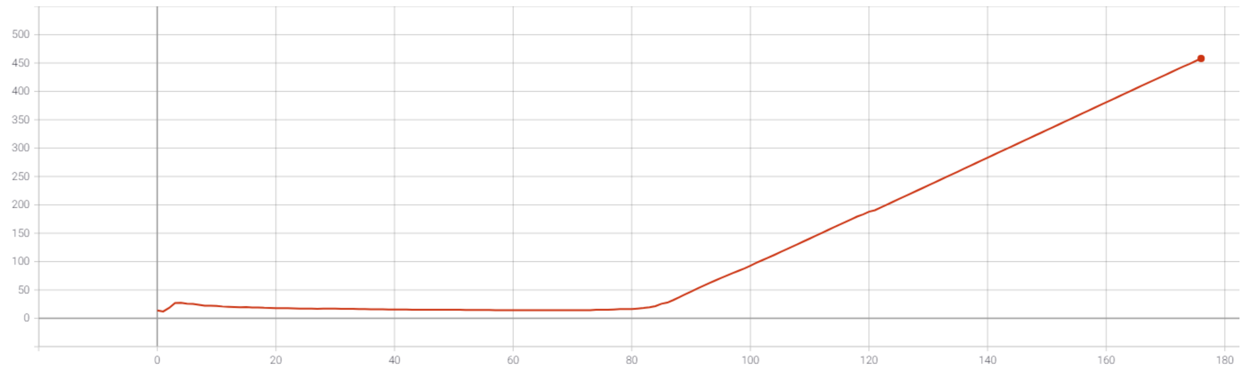


Figure of the Loss per step

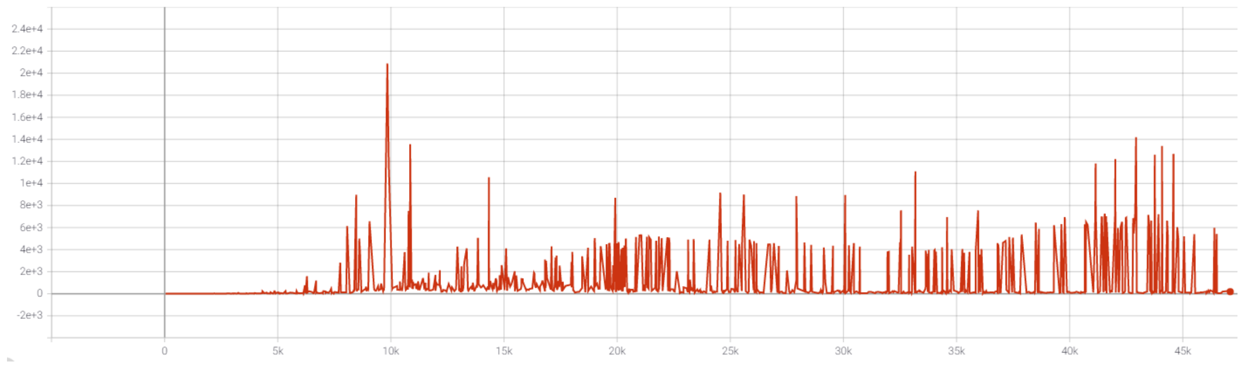
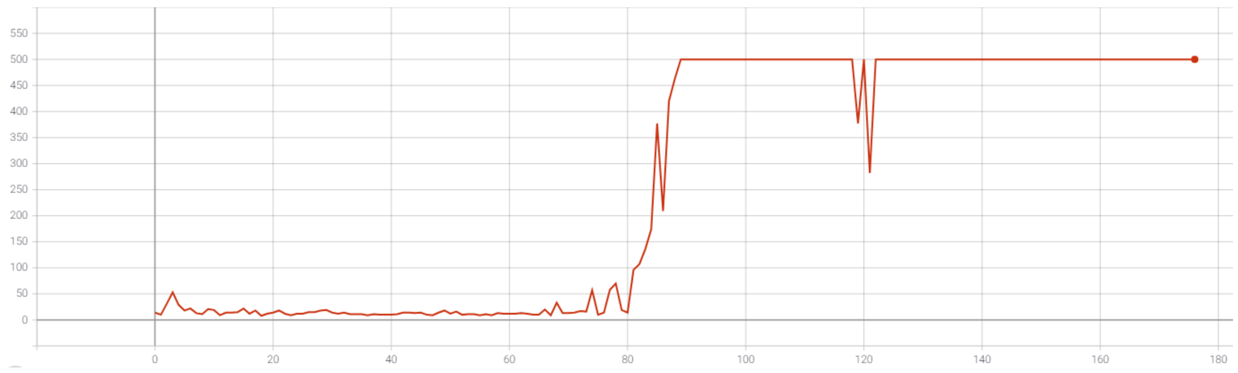


Figure of the Reward per episode



## 3.3 Comparison with the results of DQN

Figure of the average Reward in the previous 100 episodes

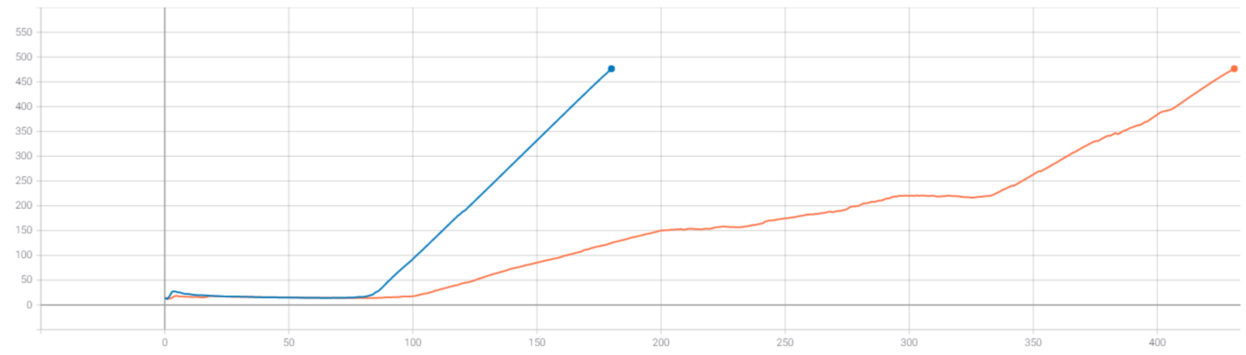


Figure of the Loss per step

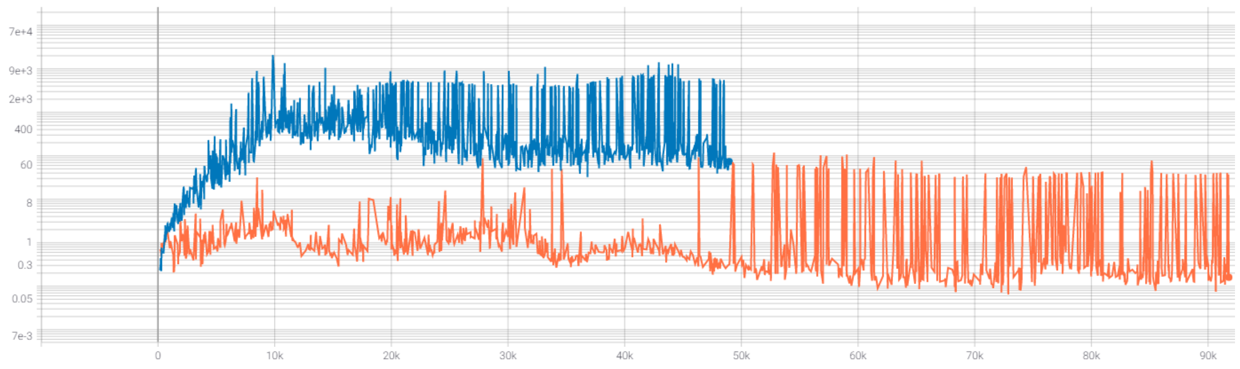
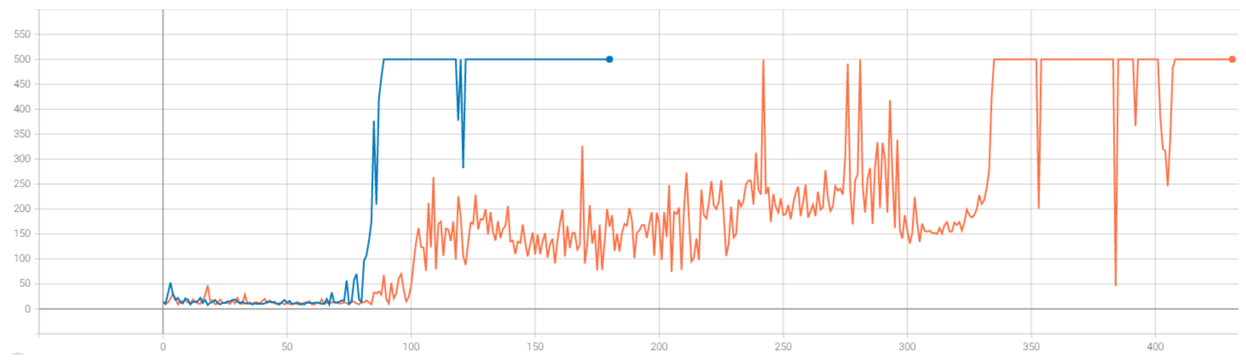


Figure of the Reward per episode



## 4. Code Execution Explanation

In order to run the .py files, there is a need to install all the relevant packages using the requirements.txt file. When located in the directory of this file in the Anaconda Prompt, you can use the following commands in your virtual environment (or to create new one) to do it. This project created in python 3.8.12.

If you want to create new virtual environment <u>(optional)</u>:

- **conda create --name TensorFlowEnv python=3.8.12**

Activate the virtual environment <u>(optional)</u>:

- **conda activate TensorFlowEnv**

Install all the required packages to the new virtual environment:

- **pip install -r requirements.txt**

Now after the environment is set up you can run the assignment files.

<u>For section 1 you should run:</u> Section1_Or_Ofir.py

<u>For section 2 you should run:</u> Section2_Or_Ofir.py

<u>For section 3 you should run:</u> Section3_Or_Ofir.py