Istanbul Technical University

Artificial Intelligence

BLG 435E

Homework 2 Report

Mete Alp Potuk
150190013

December 27, 2023

Table of Contents

1 Introduction

In the assigned homework, I had the opportunity to apply optimization methods and reinforcement deep learning techniques to specific challenges. The initial task involved addressing the Traveling Salesman Problem by implementing either simulated annealing or a genetic algorithm. The second question required the implementation of a Q-learning approach, with a focus on observing changes in the Q-table across predefined scenarios. Lastly, the third question tasked me with developing a Flappy Bird-playing agent using Deep Q-Learning.

For this assignment, I organized the code into separate folders for each problem and aimed to maintain a similiar project structure as much as possible. Informative comments are placed throughout the code to enhance reader appreciation. The central component, *'main.py'* serves as the primary code to execute. All the necessary classes are initialized in this file, and their parameters are specified. If you intend to experiment with the code or make modifications, I recommend you to start with adjustments in 'main.py'. Additionally, I've included comments in this file detailing potential parameters and their respective effects.

2 Challanges and Their Solutions

2.1 Problem 1

In this part, we were challanged with the Traveling Salesman Problem involving 12 cities, and we were tasked with finding a solution using either simulated annealing or a genetic algorithm. I chose the genetic algorithm since I found its concept more exciting. You can explore and experiment with the code yourself by opening a terminal in the "problem-1" folder and executing the code using Python.

```
python .\main.py
```

2.1.1 Data and Class Structures

I started the project by creating a **'City'** class designed to store the provided x and y coordinates. Each city within this class has a distinct index and name, aiding in understanding the algorithm and facilitating visualization. Secondly, I introduced a **'DistanceDict'** to manage the distances between cities. This approach simplified the process of accessing the desired distance by requiring just a single line of code, as demonstrated below.

```
distance = self.distance_dict[city][other_city]
```

Finally, I implemented the **'TSPGA'** class, short for for 'Travelling Salesman Problem Genetic Algorithm.' This class encapsulates both the collection of cities and the distance dictionary. Additionally, it involves key parameters such as mutation probability, crossover probability, elitism rate and etc... All essential functions for a genetic algorithm are integrated into this class, serving as a centralized controller for these operations.

2.1.2 Genetic Algorithm Implementation

The primary function called from the 'TSPGA' object in 'main.py' is **run_algorithm()**. This function, taking the desired algorithm version (basic, elitism, or advanced) and the number of generations as parameters, configures essential settings based on the given algorithm version. It then invokes the **GA()** function with the appropriate parameters. Notably, parameters such as elitism rate and the activation of advanced functions are adjusted according to the specified algorithm mode. Here are the modes and their parameter effects:

- **Basic:** This employs the most fundamental genetic functions. New generations are created by mating the fittest 50%. There is no elitism, and only basic one-point crossover and two-point exchanging mutation are utilized.
- **Elitism:** The elitism rate is set to 10%, allowing some of the fittest solutions to

directly pass to the next generation. The remaining solutions are generated by mating the fittest 50%. No advanced functions are applied.

- **Advanced:** With a higher elitism rate of 15%, advanced functions such as two-point crossover and a 20% mutation rate are introduced.

After the configuration is set up, the **GA()** function is executed with these parameters. This function implements a genetic algorithm, a widely-used approach for optimization problems. The fundamental concept involves assessing solutions using a fitness score and, over successive generations, converging to maximize this score. In my implementation, I used formula below for the fitness score calculation:

```
1 def calculate_fitness(self, solution):
2         return 1 / self.calculate_total_distance(solution)
```

This method ensures that a higher fitness score corresponds to a minimization of the total distance. Once the fitness scores for all solutions are computed, they are sorted to identify the fittest 50%, who will serve as parents for the next generation. Then, the algorithm determines the number of children to produce based on the elitism rate. If the mode is set to basic (with an elitism rate of 0), no elitism occurs. In this case, children are generated by sampling two parents and applying crossovers and mutations in a loop. After all children are generated, the new generation is ready, and the generation loop continues. This algorithm converges toward a solution by consistently selecting the fittest individuals (principle of survival of the fittest) and evolving them.

2.1.3 Algorithm Mode Comparison

To determine the relative accuracy of each algorithm mode, I conducted 100 runs for each mode over 1000 generations, providing a rough estimation. The results showcase the best solutions calculated during these runs:

- **Basic:**
  - Maximum total distance: 846.270

  - Minimum total distance: 655.039
  - Average total distance: 739.862
  - Optimal solution count: 0
  - Accuracy: 0%

- **Elitism:**
  - Maximum total distance: 722.480
  - Minimum total distance: 544.447
  - Average total distance: 583.281
  - Optimal solution count: 20
  - Accuracy: 20%

- **Advanced:**
  - Maximum total distance: 544.447
  - Minimum total distance: 544.447
  - Average total distance: 544.447
  - Optimal solution count: 100
  - Accuracy: 100%

From these results, it becomes evident that the advanced mode consistently performs well over 1000 generations, achieving optimal solutions. In contrast, the basic mode falls short of hitting optimality entirely. Elitism mode, on the other hand, appears to function decently, and given enough generations it tends to eventually reach an optimal solution. This suggests that having a modestly higher rate of elitism and utilizing advanced functions contributes significantly to the accuracy of genetic algorithms.

I've also developed a solution visualizer using Pygame to provide a clear visual representation of the results. Here is some figures to showcase example results from different algorithm modes:

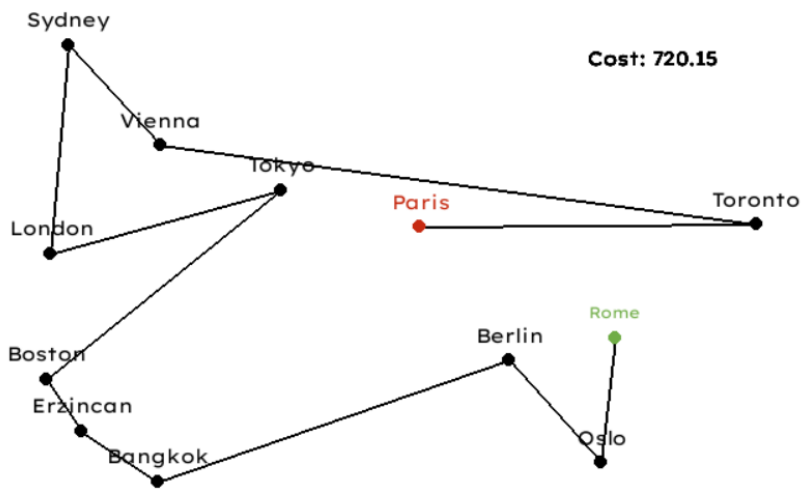

Figure 1: Basic Mode Result

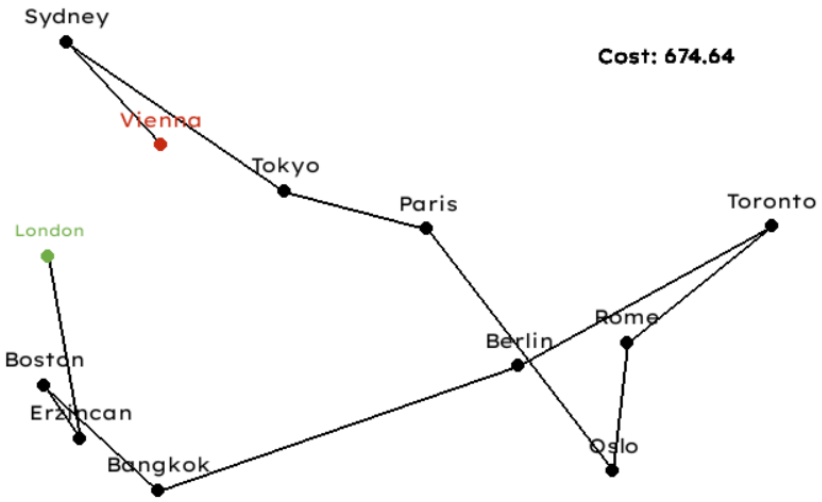

Figure 2: Elitism Mode Result

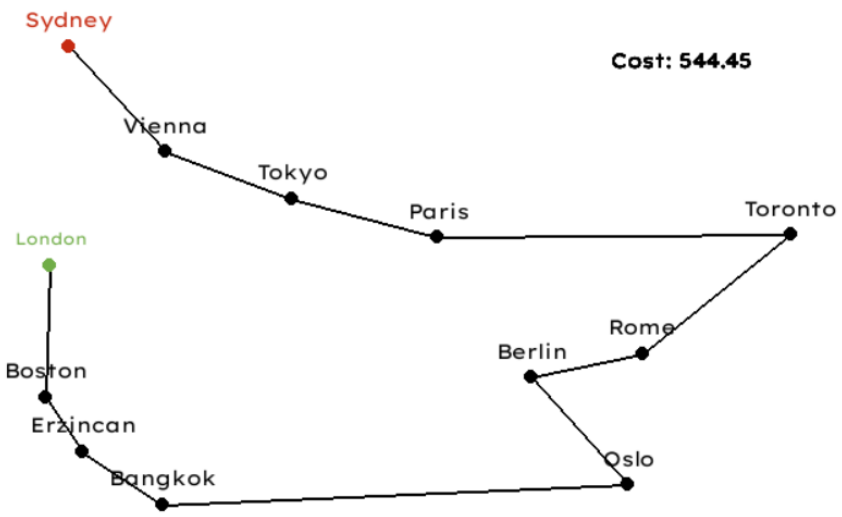

Figure 3: Advanced Mode Result

Note: Since I implemented a single genetic algorithm and manage algorithm modes by only changing parameters, there is minimal to no difference in their time complexities. Therefore, their performances are similar and I have not included time-cost graphs.

2.2 Problem 2

We encountered a path planning problem within a grid world environment in this segment. The environment is formed up of a 5x5 grid, with some grids colored blue to represent water. Our agent's goal is to navigate through the environment safely and to find a path from the starting grid to the goal grid while avoiding the water, which is unsafe to the agent. You can explore and experiment with the code yourself by opening a terminal in the "problem-2" folder and executing the code using Python.

```
python .\main.py
```

In this section, I developed a Pygame environment to help me observe and understand the agent's movements and Q-table updates. Environment and agent parameters for experimentation can be configured in the'main.py' file. Beyond that, it's a simple Q-learning agent implementation. I set the agent's Q-table to 3D because the grid was 2D,

with four possible actions to consider for each grid. Based on the current epsilon value, which decays over time, the environment directs the agent's movements and calculates states and rewards accordingly. The agent is then given this information, which is used to update its Q-table with the given parameters.

2.2.1 Part a

Q-learning is a reinforcement learning algorithm that enables an agent to explore and learn optimal actions in a given environment. It evaluates actions and computes Q-values stored in a Q-table using information such as the current state, action taken, next state, and the corresponding reward. As it balances exploration and exploitation, the agent adjusts its policy, gradually improving decision-making through cumulative rewards.

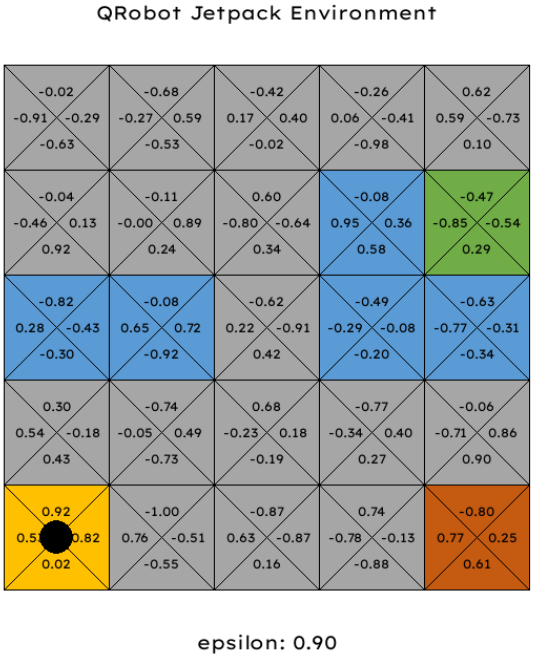

Figure 4: Q-Table Before Trajectory 1

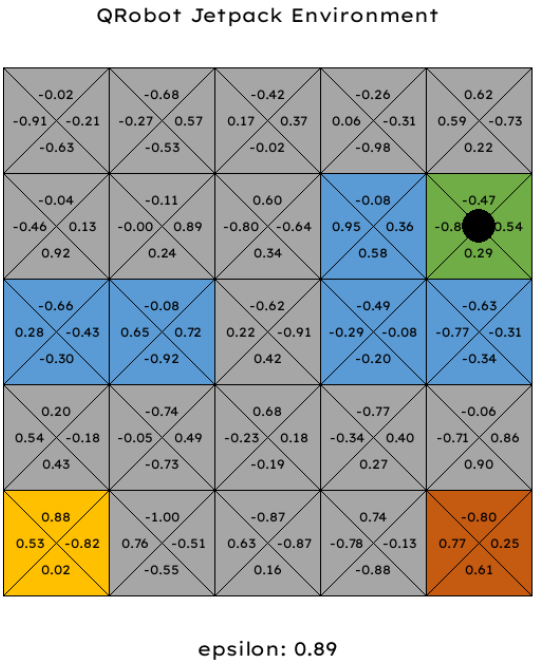

Figure 5: Q-Table After Trajectory 1

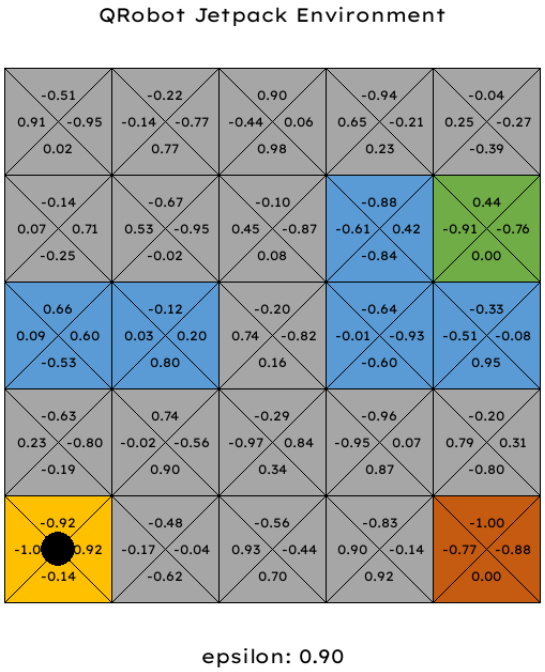

Figure 6: Q-Table Before Trajectory 2

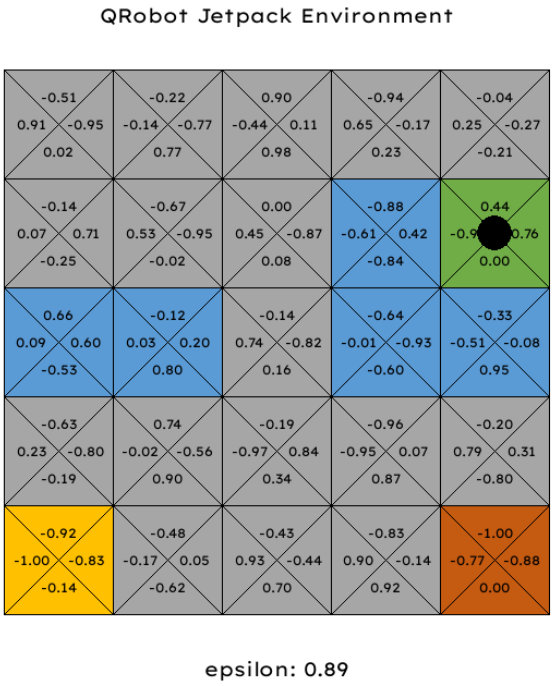

Figure 7: Q-Table After Trajectory 2

As shown in the figures above, the agent iteratively updates Q-values along its trajectory using the formula given below this paragraph. The agent updates its values based on these random entries and associated rewards, if any, as the Q-table is initialized with random values that correspond with the assignment's request. The main distinction between the two trajectories is that the first traverses a water grid, resulting in evaluations with negative rewards, while the second traverses without encountering water and depends entirely on pre-defined Q-values. Additionally, a rule is implemented to penalize the agent if its action results in no movement (if the state is equal to the next state), aiming to prevent the agent from getting stuck at edges.

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(R + \gamma \cdot \max_{a} Q(s',a)\right)$$

2.2.2 Part b

As stated earlier in the Problem 2 section, I developed a simple Q-learning agent that was tailored to the given environment. I ran experiments by modifying the parameters

stored in 'config.py'. It became clear that the decay rate has a significant impact on the delicate balance of exploration and exploitation. Furthermore, the learning rate (represented in the formula as alpha) determines the strength of the updates. The parameters I used to highlight the learned path in the Figure 8 are listed below.

```
DECAY_RATE = 0.99995
EPS_MIN = 0.1
EPS_MAX = 0.9
ALPHA = 0.1
GAMMA = 0.9
```

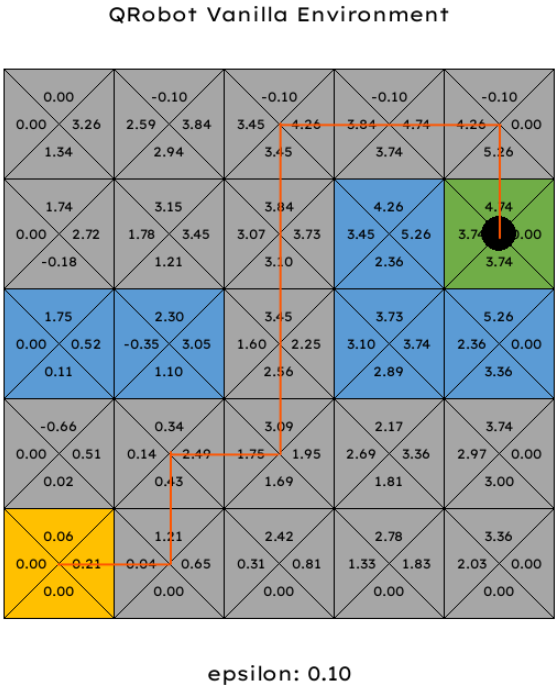

Figure 8: Learnt Path Vanilla

2.2.3 Part c

In this section, a jetpack grid is introduced, when taken making the waters become for the agent and it can move up to 2 tiles at a time. Upon reaching the this grid, the agent's original color from black to the orange color of the jetpack grid to represent jetpack is taken. During experiments comparing the vanilla and jetpack environment modes, I noticed significant differences in the learning curve and Q-values when the agent

acquires the jetpack in the exploration phase. As a result, the agent's early acquisition of the jetpack emerges as a huge factor, influencing the learning process. I observed that the agent tends to learn more rapidly in the jetpack environment, where obstacles are removed, and positive feedback from reaching the goal grid expands more freely compared to the vanilla environment. The learned trajectory of the agent in the jetpack environment is shown in the Figure

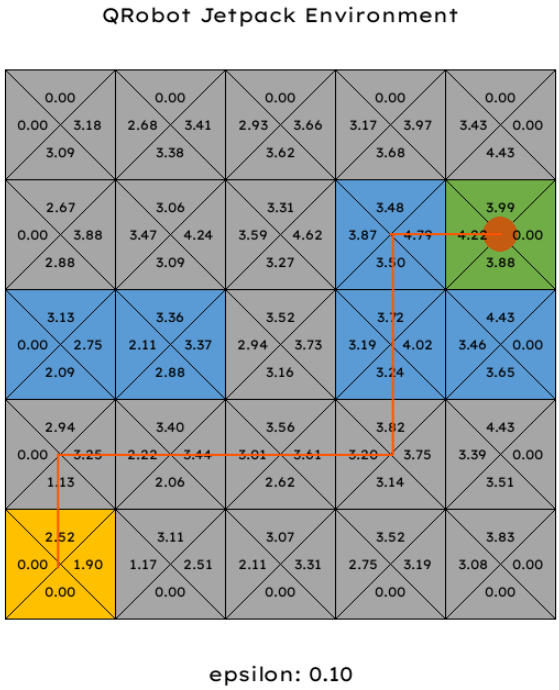

Figure 9: Learnt Path Jetpack

Given the dynamic nature of the agent's movement after acquiring the jetpack (allowing movement of up to 2 tiles at a time), a modification in the q-table structure becomes necessary. Initially, before the jetpack is obtained, the q-table is initialized as follows:

```
self.q_table =  np.zeros((rows, columns, 4)) # 4 -> up down left right
```

This configuration allows the agent's ability to move only one step at a time, resulting in a four-element action space. However, after acquiring the jetpack, the agent evolves, causing an adaptation in the q-table structure as shown below:

```
self.q_table =  np.zeros((rows, columns, 8))
# 8 -> one_up, one_down, one_left, one_right,
# two_up, two_down, two_left, two_right
```

Since these changes necessitate significant changes in visualization and structure itself, I've not included it directly and implemented double moves as a sequence of single moves. Despite being aware of the risks of not implementing this change, I believe this approach has provided valuable insights into various aspects of q-learning.

2.3 Problem 3

In this problem, the objective was to create a DQN agent utilizing PyTorch and design a reinforcement learning environment to train the agent to play a well known game, Flappy Bird. DQN is basically a variant of the Q-learning algorithm that uses a neural network to store and estimate Q-values instead of a traditional Q table. This neural network-based approach allows for a more flexible and scalable representation of Q-values, facilitating improved handling of complex state-action spaces. I developed this part with guidance from a PyTorch tutorial [1], a resource recommended during our course practice sessions. To facilitate code execution as intended, it's necessary to open the terminal at the project's root directory and run the code with the following command:

```
python .\solution\main.py
```

As it can be seen from the command, I organized my code within a directory named 'solution' Within this directory, there is a 'model' folder containing relevant classes and a 'config.py' file containing parameters such as epsilon, alpha, gamma, etc. Additionally, a 'plots' folder is included where I stored some of the experiment results in the form of PNGs. The primary structure and execution code included in 'main.py' where I provided comments explaining the additional lines beyond the tutorial. Nevertheless, I will be explaining some key ideas and noteworthy experiments for further clarity.

2.3.1 Reward Offset

While experimenting with various parameters to understand how my agent learns to play Flappy Bird, I observed instances where it directly falls to the ground, resulting in notably short duration scores. To address this issue, I introduced a logic to evaluate the episode time score relative to all episode time scores. Based on the deviation from the overall average, I adjusted the negative feedback the agent receives when it hits an obstacle (when given at the end of a game) using a reward offset. This adjustment initiates after 10 episodes, as a clear average time score is not available before that point. In my latest implementation, I constrained this offset within a range of -2.5 to 2.5 to balance it with the original negative feedback of -5. I managed these boundaries using a variable named offset_aggression. I also wanted to give extra points if it overcame any obstacles, so I began counting and adding them up to reward. I experimented with this parameters a lot to determine the most effective value. While I initially tried to lessen the direct falling problem by making essential changes to parameters, including large numbers, I eventually decided to allow the agent to learn and adapt with minimal changes. Although this approach did not completely solve the directly falling issue, I believe it contributed to a slight improvement in performance. The reward offset calculation and evaluation section can be found below in the 'main.py' file, accompanied by detailed explanatory comments but here is the essential ideas from it:

```
if p.game_over():
    if episode > 10:
        episode_min = min(episode_durations)
        episode_max = max(episode_durations)
        episode_avg = sum(episode_durations) / len(episode_durations)
        offset_aggression = -0.5 * -5
        reward_offset = max(min(((frame - episode_avg) / ((episode_max - episode_min) / 2))
            * offset_aggression, 2.5), -2.5)
        # reward_offset += passed_obstacles
        reward += reward_offset
        passed_obstacles = 0
    next_state = None
```

2.3.2 Different Epsilon Ending Values

This section was requested in the assignment pdf file as well. I've included a section where I saved result plots along with the parameters used, such as epsilon starting and ending values, learning rate, decay rate, etc. This enables a comparison of different parameter setups to understand their impact on the agent's performance. Varying epsilon endings plays a crucial role in determining the balance between exploration and exploitation. Related plot pictures can be found under './solution/plots' directory.

During my experiments, I noticed an interesting observation: regardless of the parameter combinations used, achieving a successful agent with consistent performance proved to be a difficult task. When experimenting with an epsilon end value of 0.5, I didn't notice significant performance changes, as half of the agent's moves remained randomized. However, setting epsilon end values to 0.1 and 0.01 revealed that the agent predominantly relies on its learned knowledge.

I also noticed that allowing the agent to play more games, lowering the learning rate, and keeping epsilon decay high seems to return a better performance. I believe that with sufficient time and fine-tuning of parameters, the agent has the potential to learn and perform optimally.

Some plots showcasing the results are presented in the figures below.

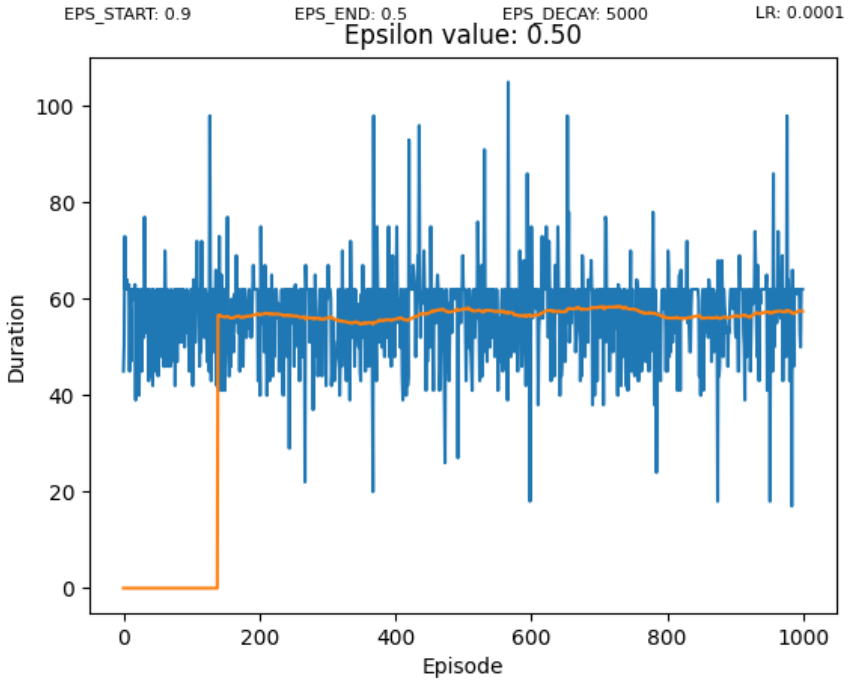

Figure 10: Epsilon End Value 0.5

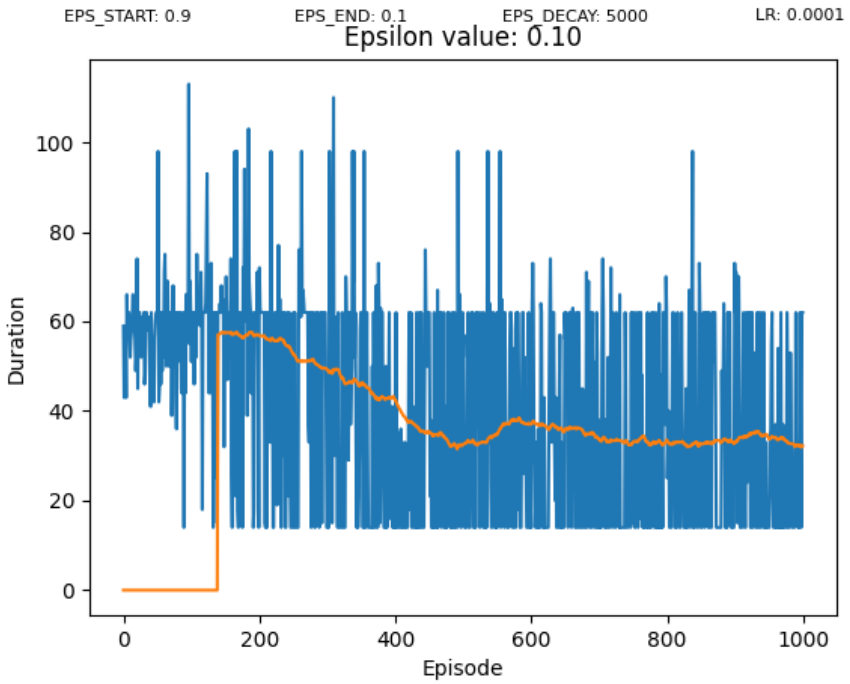

Figure 11: Epsilon End Value 0.1

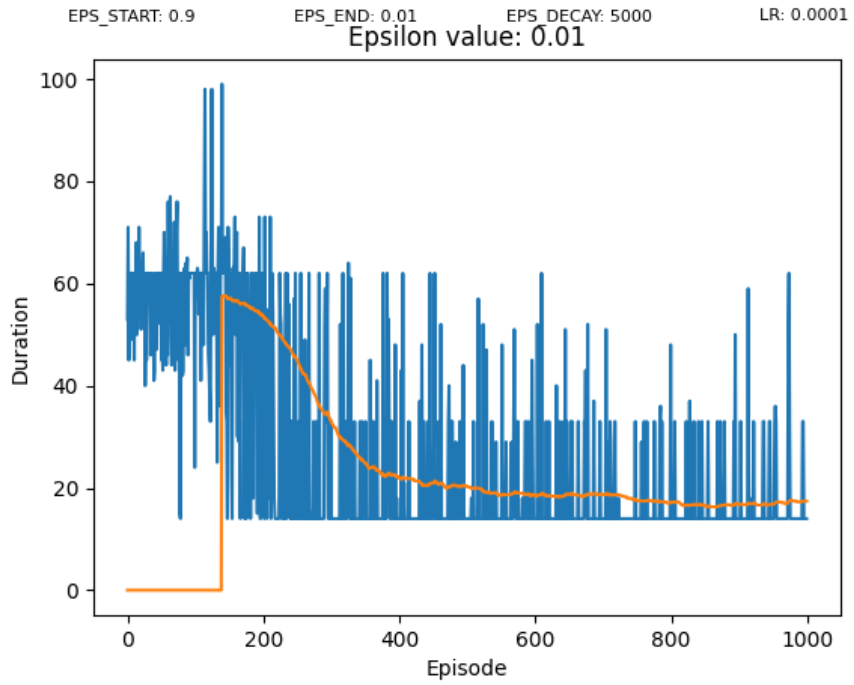

Figure 12: Epsilon End Value 0.0.1

3 Conclusion

In this assignment, we had the opportunity to apply optimization algorithms such as genetic algorithms and reinforcement learning techniques like Q-learning and DQN to specific problems. Through this process, I discovered the significant impact of elitism on the performance of genetic algorithms. Additionally, I learned that careful setup of the environment, defining rewards and penalties, and tuning parameters like epsilon end and learning rate are extremely crucial factors influencing the success of reinforcement learning agents. The systematic development of these algorithms not only enhanced my understanding but also allowed me to improve my Python skills. Overall, this experience provided valuable insights into the intricate aspects of optimization and reinforcement learning.

References

[1] Paszke, A. and Towers, M. *Reinforcement Learning (DQN) tutorial with PyTorch.* https://pytorch.org/tutorials/intermediate/reinforcement$_{q}$*learning.html*, *Accessed* : 24.12.2023.