Assignment 3 - Defining and Solving Reinforcement Learning Task

Madhu Babu Sikha msikha@buffalo.edu
Deepak Raj Mohan Raj dmohanra@buffalo.edu

Team Member	Assignment Part	Contribution (%)
Madhu Babu Sikha	Part I, Part II, Part III, Bonus	50
Deepak Raj Mohan Raj	Part I, Part II, Part III, Bonus	50

Abstract

The goal of this assignment is to acquire experience in defining and solving a reinforcement learning (RL) environment, following Gym standards. The assignment consists of three parts. The first part focuses on defining an environment that is based on a Markov decision process (MDP). In the second part, we will apply a tabular method SARSA to solve an environment that was previously defined. In the third part, we apply the Q-learning method to solve a grid-world environment.

1 Part I: Define an RL Environment

In this part, we have defined a grid-world reinforcement learning environment as an MDP. While building an RL environment, we have defined possible states, actions, rewards and other parameters.

1.1 Grid World

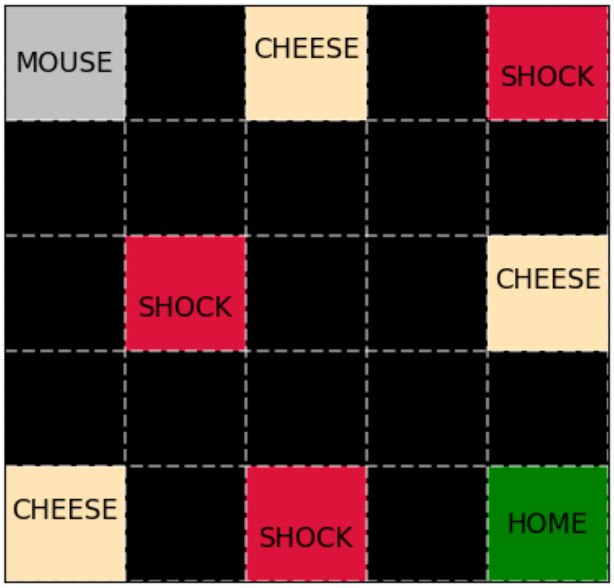

Figure 1: Grid World

Theme: Grid world environment with a mouse, cheese, home, and obstacles.

States: The environment is a 5x5 grid where the mouse, cheese, home, and obstacles are located.

Actions: The mouse can move in four directions: up, down, left, and right.

Rewards: The mouse receives a negative reward if it steps on a shock obstacle and a positive reward if it steps on a cheese square.

Objective: The mouse's objective is to reach the home located at the bottom right corner of the grid while avoiding the shocks and collecting the cheese pieces to maximize its total reward. The episode terminates after a fixed maximum number of time steps.

1.2 RL Environment

States: The environment is a 5x5 grid where the mouse, cheese, home, and shock obstacles are located. The state space is defined as a discrete space of 25 states, where each state is represented by a unique integer value ranging from 0 to 24.

Actions: The mouse can take one of four actions, which are moving right, left, up, or down. The action space is defined as a discrete space of four actions, where each action is represented by a unique integer value ranging from 0 to 3.

Rewards: The mouse receives a positive reward of 3, 4, or 5 if it steps on a cheese square at position [0, 2], [2, 4], or [4, 0], respectively. The mouse receives a negative reward of -3, -4, or -5 if it steps on a shock obstacle at position [4, 2], [0, 4], or [2, 1], respectively. The mouse receives a positive reward of 10 if it reaches the home located at [4, 4].

Objective: The mouse's objective is to reach the home while avoiding the shocks and collecting the cheese pieces to maximize its total reward. The episode terminates after a fixed maximum number of time steps, which is 100.

1.3 Environment Safety

In the given GridEnvironment, safety is ensured in various ways. First, the actions chosen by the agent are validated to ensure they are within the allowed range of actions. This is done using the assert statement in the step method. Second, the environment is designed such that the agent can only navigate within the defined state-space of the grid, which is 5x5. This is achieved by constraining the agent's movement based on the current position and the action taken. Third, the agent is penalized for stepping on shocks, which serves as a disincentive for the agent to take actions that are not allowed. Finally, the maximum number of timesteps that the agent can take is predefined, which ensures that the agent does not get stuck in an infinite loop or take too long to complete the task.

2 Part II: Define an RL Environment

In this part, we implemented SARSA (State-Action-Reward-State-Action) algorithm and applied it to solve the environemnt defined in Part 1. SARSA is an on-policy reinforcement learning algorithm. The agent updates its Q-values based on the current state, action, reward, and next state, action pair. It uses an exploration-exploitation strategy to balance between exploring new actions and exploiting the knowledge gained so far.

2.1 SARSA

SARSA (State-Action-Reward-State-Action) is a reinforcement learning algorithm that learns a policy by iteratively interacting with an environment. The algorithm updates its policy based on the state, action, reward, and next state it receives from the environment. SARSA is an on-policy learning algorithm, which means that it learns the value of the policy that it is currently using to interact with the environment.

2.1.1 Update Function

The update function for SARSA is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{1}$$

where

- $Q(S_t, A_t)$ is the estimated value of taking action A_t in state S_t
- R_{t+1} is the reward received after taking action A_t in state S_t and transitioning to the next state S_{t+1}
- γ is the discount factor that determines the importance of future rewards
- α is the learning rate that controls the step size of the updates.

2.1.2 Key Features

The key features of SARSA are:

- It is an on-policy learning algorithm that learns the value of the policy that it is currently using to interact with the environment.
- It uses the same policy to select actions during training and evaluation.
- It is a model-free algorithm that does not require knowledge of the environment's dynamics or transition probabilities.
- It is suitable for problems where the agent interacts with the environment in a sequential manner and where the state and action spaces are not too large.

2.1.3 Advantages

- It can handle problems with continuous state and action spaces.
- It is an online algorithm that can learn from experience as it interacts with the environment.
- It can learn optimal policies in environments with stochastic rewards.

2.1.4 Disadvantages

- It can be slow to converge and may require a large amount of data to learn optimal policies.
- It can suffer from the problem of overestimation of action values in certain scenarios.

2.2 Applying SARSA

1. We have initialized some hyperparameters such as the number of episodes, learning rate (α), discount factor (γ), and exploration-exploitation trade-off (ϵ).
2. We have set up a Q-table, which is used to store the state-action values.
3. Then we executed a loop for a specified number of episodes. At the start of each episode, the environment is reset, and the initial state is obtained. The agent takes an action based on the current state using an epsilon-greedy policy, where it chooses a random action with a probability of epsilon, and otherwise chooses the action with the highest Q-value for that state. The environment returns a new state and a reward based on the action taken. The agent uses the new state and reward to update the Q-value of the previous state and action using the SARSA update rule.
4. The epsilon value is then updated according to the epsilon decay rate. The total rewards obtained during the episode, epsilon, and the number of steps taken to complete the episode are then recorded.

5. After the loop finishes executing, the code uses matplotlib to plot the rewards, epsilon, and timesteps taken for each episode. The Q-table is also printed and displayed as a heatmap using seaborn. Finally, we have implemented a function to perform hyperparameter tuning for the SARSA algorithm.

Table 1: Hyperparameters Setup

Hyperparameters	**Setup**
num_episodes	1000
alpha	0.2
gamma	0.95
maximum_epsilon	1
minimum_epsilon	0.01
epsilon_decay_rate	0.9954

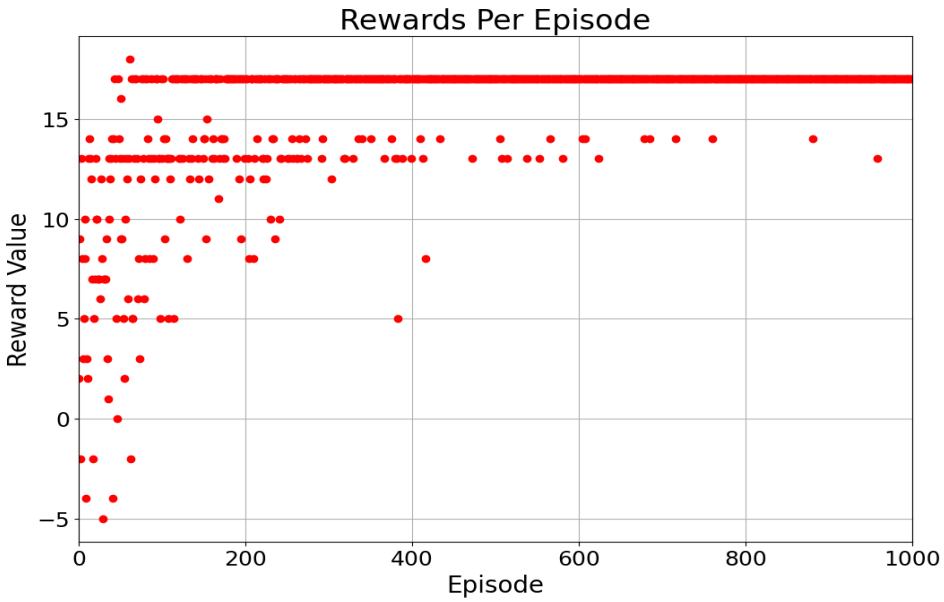

Figure 2: Rewards vs Episode

The depicted graph illustrates the aggregate reward acquired by the agent in each episode. As the epsilon value is progressively lowered, the agent engages in exploratory behavior in the initial stages, leading to lower rewards. However, with simultaneous q-table updates and decreasing epsilon, the agent endeavors to maximize rewards, which explains the increasing trend of total rewards over the course of numerous episodes.

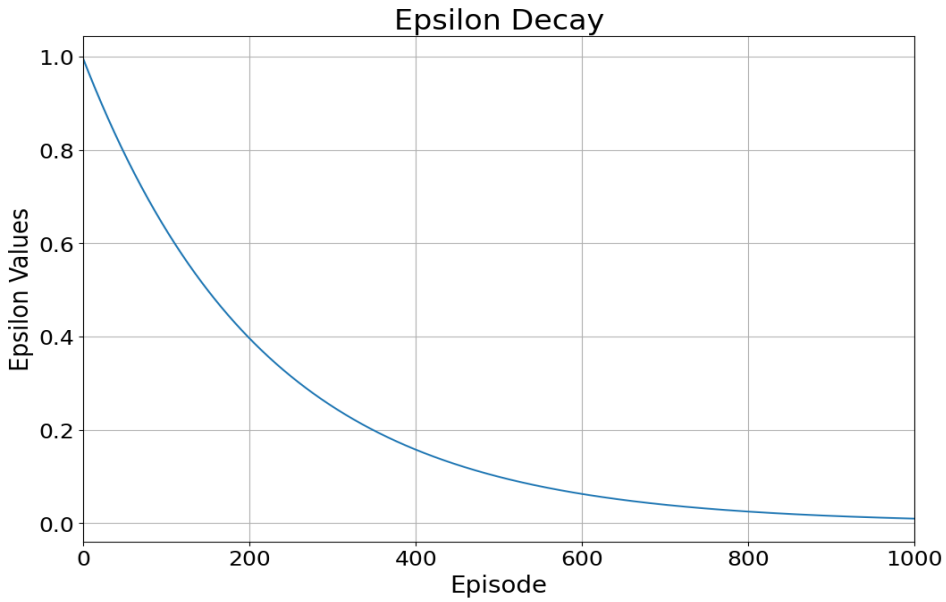

Figure 3: Epsilon vs Episode

The presented graph indicates a gradual decay of episode with respect to epsilon value.

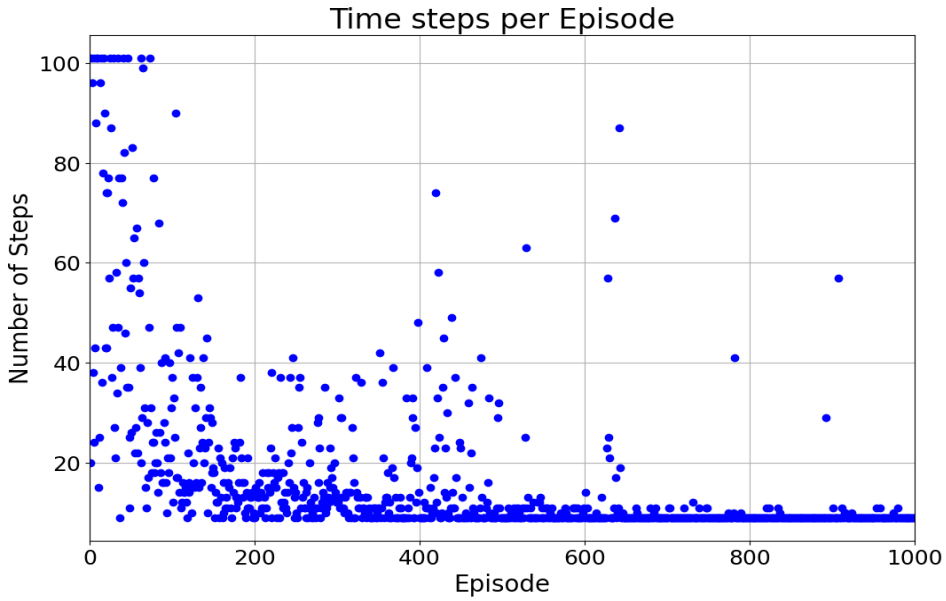

Figure 4: Number of Steps vs Episode

The depicted graph illustrates the agent's timestep count for achieving the goal. In the early training phase, the q-values remain unaltered, and the epsilon value is high, thereby compelling the agent to explore further. Consequently, the agent requires a greater number of timesteps to reach the goal, whereas as the episodes progress, the agent's timestep count reduces.

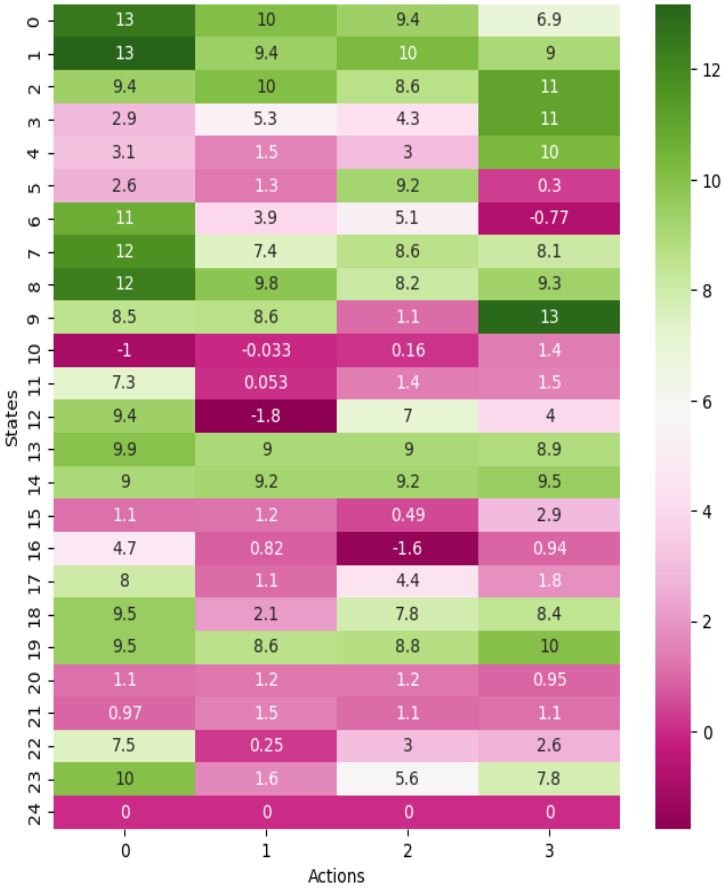

Figure 5: Q-matrix

This is the final q-matrix that resulted from training the agent within the environment for a total of 1000 episodes. To illustrate, let's take into account state 23, which corresponds to location 4,3 within the grid environment. When moving towards the left direction from this particular state, a negative reward is incurred at location 4,2, resulting in a relatively lower q-value for this state-action pair when compared to the q-values of the remaining actions possible from the same state.

The state values corresponding to the 24th state are 0's because it is a terminal state.

If the agent is at state 15(location 3,0) and takes action as down, it reaches state 20(location 4,0) which has a maximum q-value for that state because there is a positive reward at state 20.

Figure 6: Reward vs Epsode for evaluation

The SARSA algorithm was evaluated by plotting the agent's performance over 10 episodes, during which it was instructed to take only greedy actions. The resulting graph indicates that the agent achieved a maximum reward of 17 in each episode.

2.3 Hyperparameter Tuning

In each iteration, the values of num_episodes, alpha, gamma, and epsilon were kept constant, while only the maximum_epsilon, minimum_epsilon, gamma and number_episodes parameters were altered.

2.3.1 Min/Max Epsilon

Table 2: Hyperparameters Setup

Hyperparameters	**Setup**
maximum_epsilon	0.9
minimum_epsilon	0.2
epsilon_decay_rate	0.9984

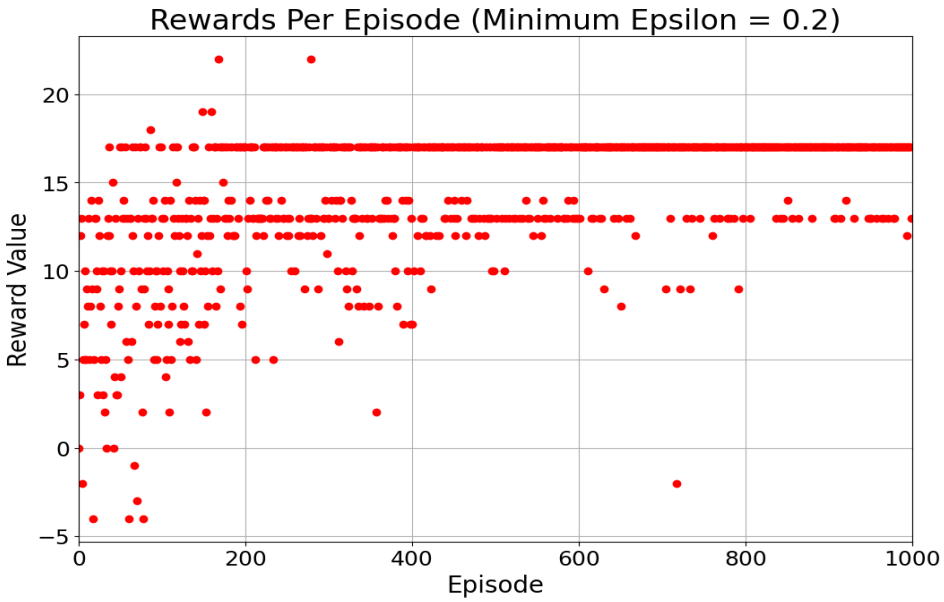

Figure 7: Rewards vs Episode

From the plot, it is evident that as the episode count increases, the rewards begin to diverge from the initial configuration.

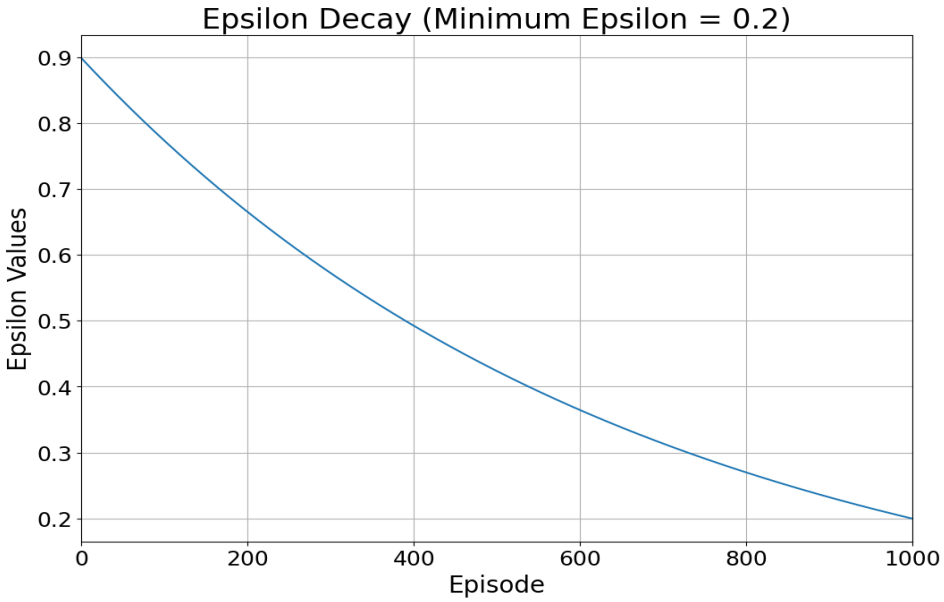

Figure 8: Epsilon vs Episode

The plot indicates that the epsilon's elbow becomes linear with an increase in the number of episodes.

Table 3: Hyperparameters Setup

Hyperparameters	Setup
maximum_epsilon	0.8
minimum_epsilon	0.3
epsilon_decay_rate	0.9990

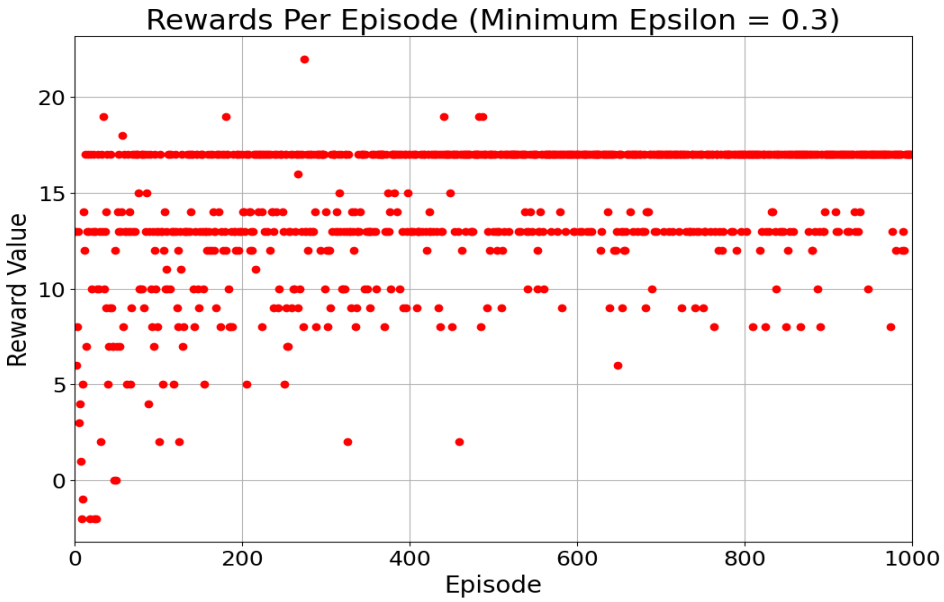

Figure 9: Rewards vs Episode

It is clear from the plot that decreasing the maximum/minimum epsilon results in a divergence of rewards from the previous configuration as the number of episodes increases.

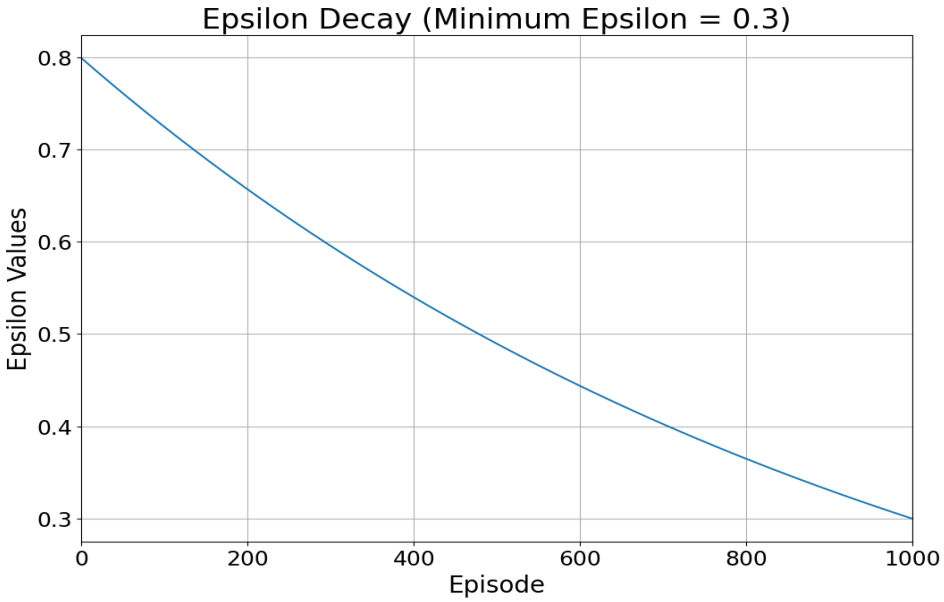

Figure 10: Epsilon vs Episode

It can be inferred from the plot that as the maximum/minimum epsilon value decreases, the elbow of the epsilon curve shows a tendency towards linearity with an increase in the number of episodes.

Table 4: Hyperparameters Setup

Hyperparameters	**Setup**
maximum_epsilon	0.7
minimum_epsilon	0.4
epsilon_decay_rate	0.9994

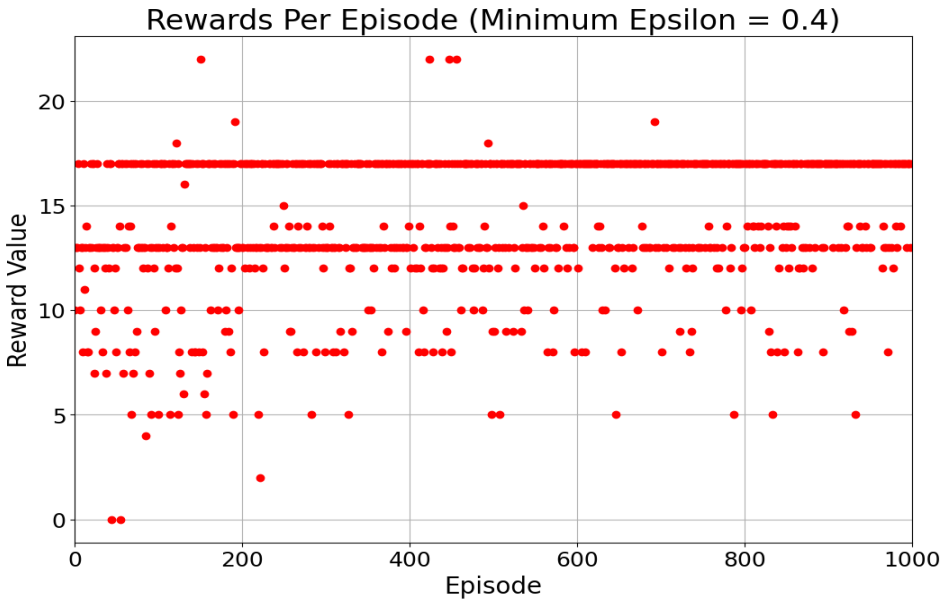

Figure 11: Rewards vs Episode

As the number of episodes increases, the plot clearly shows that further decreasing the maximum/minimum epsilon leads to greater divergence of rewards compared to the previous configuration. As we increase the minimum epsilon and decreasing the maximum epsilon values, we are restricting the agent to use its knowledge (exploiting) and exploring.

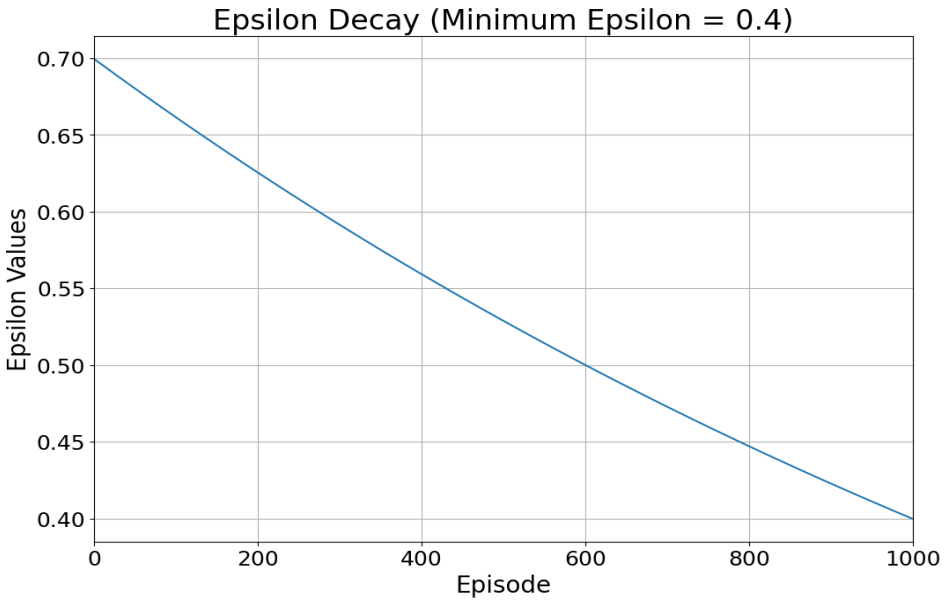

Figure 12: Epsilon vs Episode

As the maximum/minimum epsilon value decreases further, the elbow of the epsilon curve demonstrates a marked inclination towards linearity with a rise in the number of episodes, which can be deduced from the plot.

The trials show that the choice of episode count and maximum/minimum epsilon values significantly affects the agent's performance in reinforcement learning. Increasing the episode count leads to improved policy and greater cumulative rewards. However, decreasing the maximum/minimum epsilon values can result in a divergence of rewards from the initial configuration, emphasizing the importance of finding optimal epsilon values.

2.3.2 Gamma

Table 5: Hyperparameters Setup

Hyperparameters	**Setup**
gamma	0.7

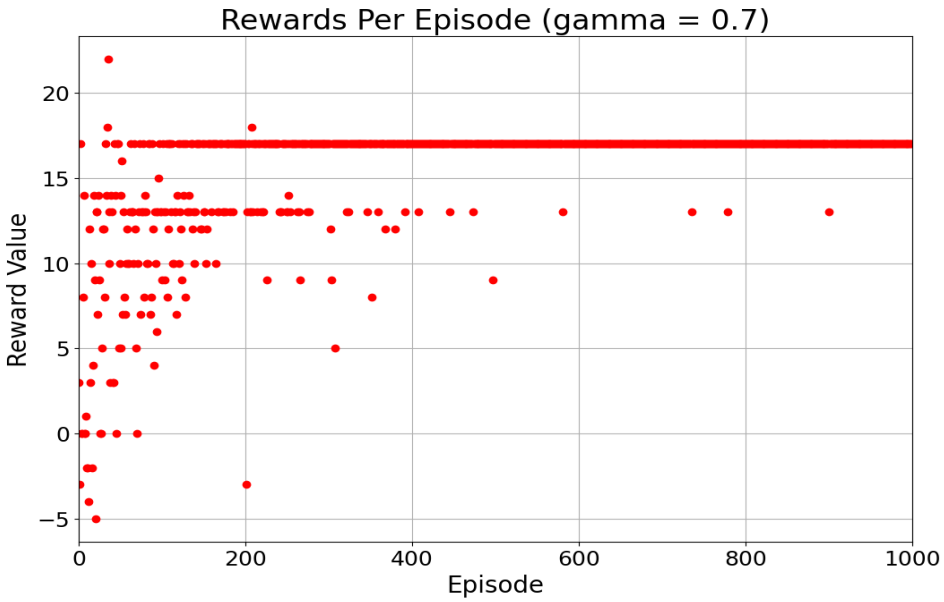

Figure 13: Rewards vs Episode

For the graph with a gamma value of 0.7, the agent appears to be more farsighted, considering future rewards more heavily. This is reflected in the increasing trend of cumulative rewards over episodes, indicating that the agent is gradually learning to maximize its long-term returns.

Table 6: Hyperparameters Setup

Hyperparameters	**Setup**
gamma	0.4

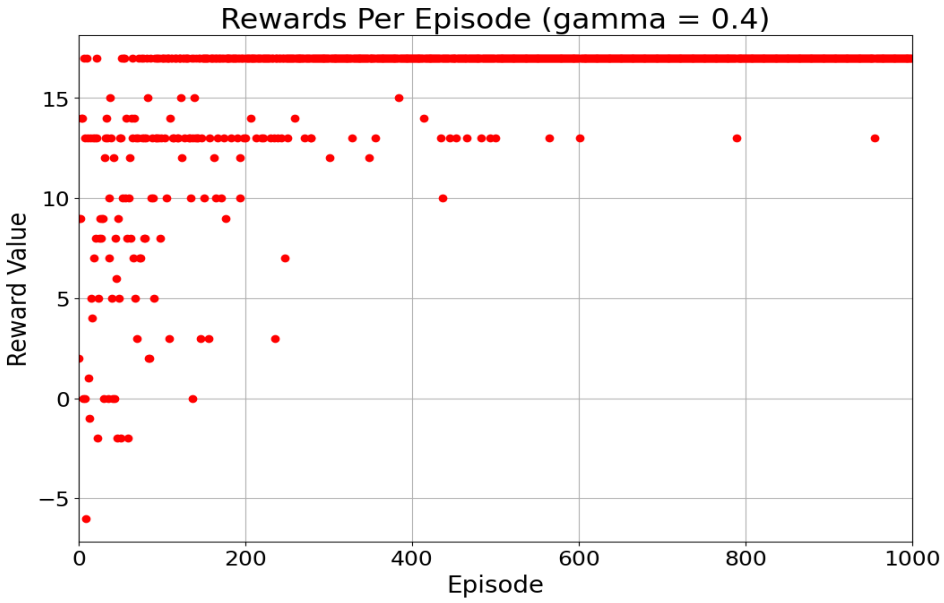

Figure 14: Rewards vs Episode

In contrast, the graph with a gamma value of 0.4 shows a more balanced approach, where the agent is neither too short-sighted nor too farsighted. The cumulative rewards gradually increase over episodes, but at a slower pace compared to the higher gamma value.

Table 7: Hyperparameters Setup

Hyperparameters	Setup
gamma	0.2

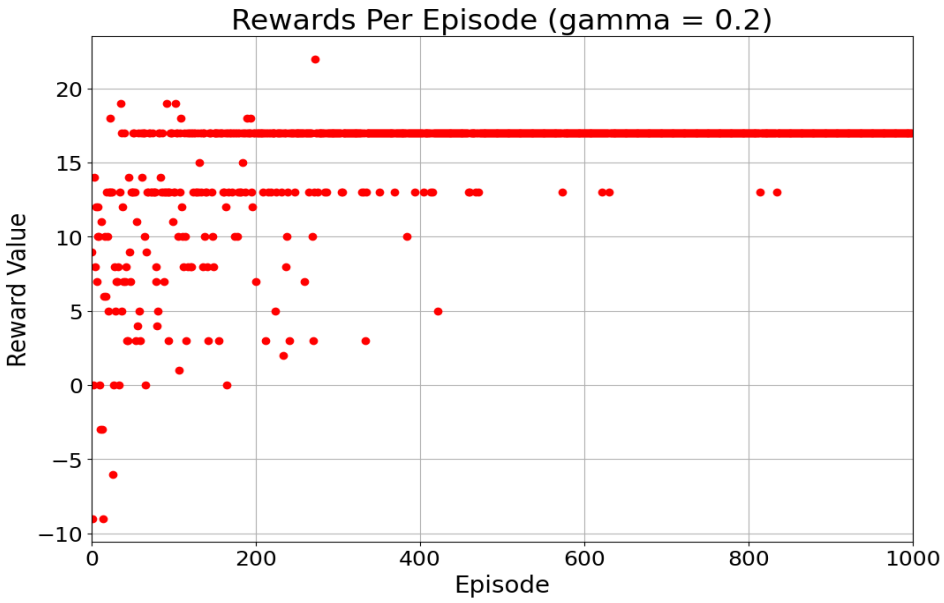

Figure 15: Rewards vs Episode

Lastly, the graph with a gamma value of 0.2 shows that the agent is more short-sighted, prioritizing immediate rewards over long-term ones. This is reflected in the steep increase of cumulative rewards in the initial episodes, followed by a plateau in the later episodes. This policy may result in faster convergence, but it may also lead to a suboptimal policy that overlooks future rewards. The higher gamma value lead to higher cumulative rewards in the long run, but may also result in slower convergence during training.

2.3.3 Episodes

Table 8: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	700

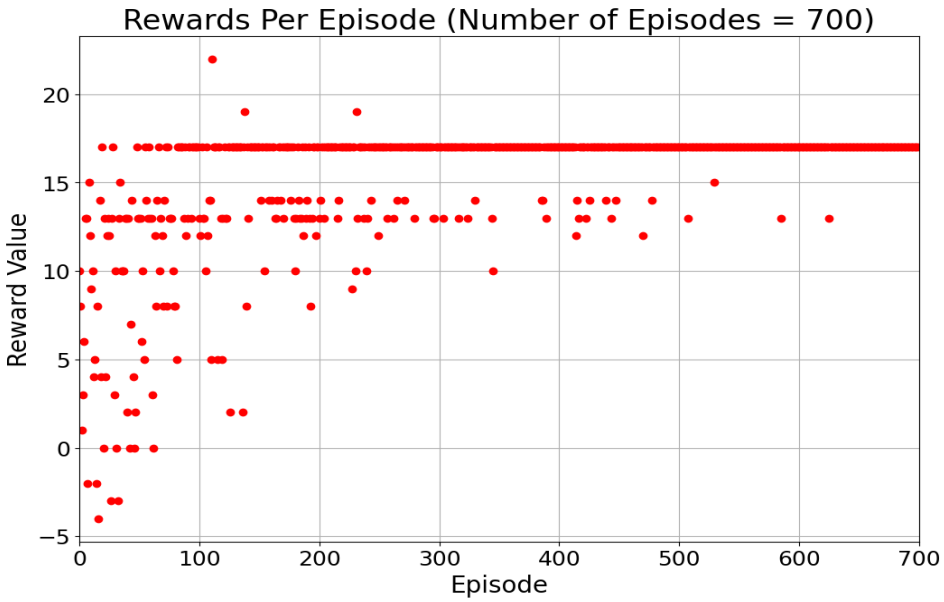

Figure 16: Rewards vs Episode

For the graph with 700 num_episodes, the agent's performance gradually improves over time, with a noticeable increase in cumulative rewards in the initial episodes, followed by a steady rise in the later episodes. This suggests that the agent needs more episodes to fully converge to an optimal policy and maximize its long-term returns.

Table 9: Hyperparameters Setup

Hyperparameters	**Setup**
num_episodes	400

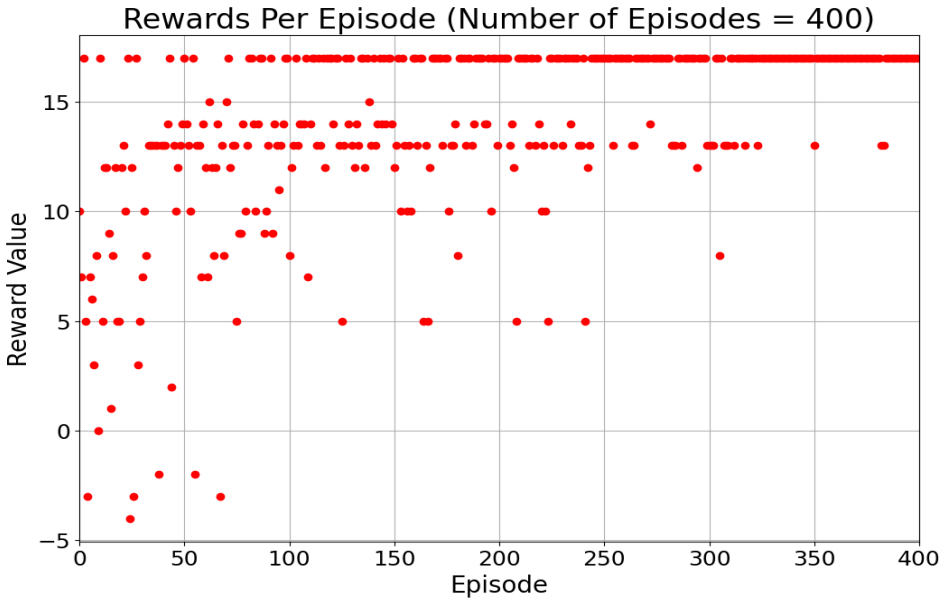

Figure 17: Rewards vs Episode

In contrast, the graph with 400 num_episodes shows a quicker convergence, with the cumulative rewards reaching a plateau at around 200 episodes. This suggests that the agent is able to learn the optimal policy faster, but may not be able to fully explore the state-action space and may miss out on potentially higher rewards.

Table 10: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	200

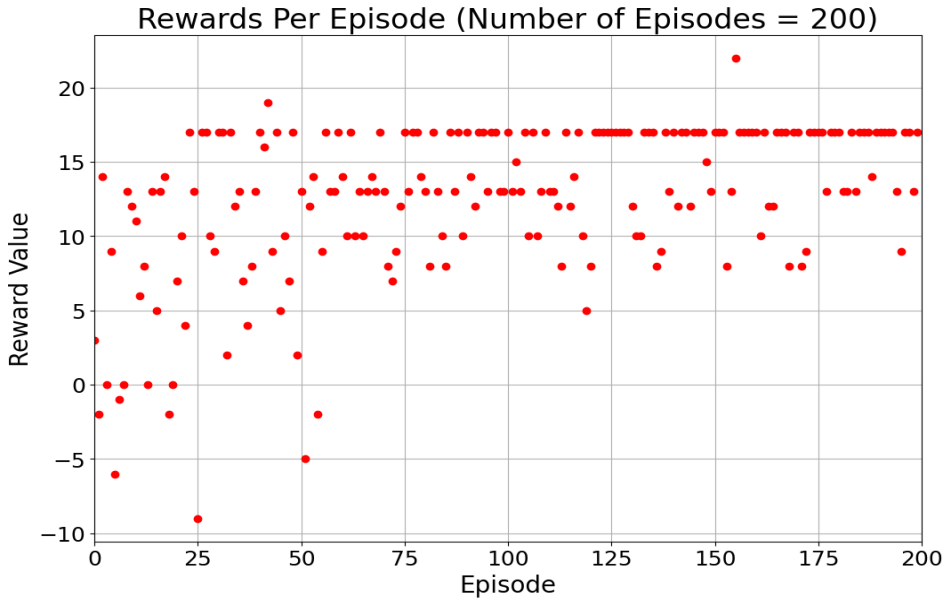

Figure 18: Rewards vs Episode

Lastly, the graph with only 200 num_episodes shows a relatively unstable learning curve, with a large variance in the cumulative rewards across different episodes. This suggests that the agent does not have enough episodes to explore the state-action space and learn the optimal policy, leading to a suboptimal policy that may prioritize short-term rewards over long-term ones.

2.4 Efficient Hyper-parameters

Upon adjusting and fine-tuning all the parameters, we conclude that the original model performed the best overall, using the following values for its parameters:

Table 11: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	1000
alpha	0.2
gamma	0.95
maximum_epsilon	1
minimum_epsilon	0.01
epsilon_decay_rate	0.9954

3 Part-III: Solving Environment using Q-Learning

In this section, we implement the Q-learning algorithm to address the environment specified in Part 1. Q-learning is a reinforcement learning algorithm that is off-policy, and it shares numerous similarities with SARSA. The procedure involves taking an action in the current state, observing the reward and next state, and updating the Q-value estimate of the previous state-action pair by considering the maximum anticipated Q-value of the next state.

3.1 Q-learning

Q-learning is an off-policy reinforcement learning algorithm that can learn optimal policies by interacting with an environment. The algorithm updates its policy based on the maximum expected Q-value of the next state, regardless of the action taken in that state. This makes it different from SARSA, which updates its policy based on the action taken in the next state.

3.1.1 Update Function

The update function for Q-learning is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{2}$$

where

- $Q(S_t, A_t)$ is the estimated value of taking action A_t in state S_t
- R_{t+1} is the reward received after taking action A_t in state S_t and transitioning to the next state S_{t+1}
- γ is the discount factor that determines the importance of future rewards
- α is the learning rate that controls the step size of the updates.

3.1.2 Key Features

The key features of Q-learning are:

- It is an off-policy learning algorithm that can learn optimal policies even when the policy used for exploration is different from the optimal policy.
- It uses the maximum Q-value of the next state to update the Q-value of the current state, which can lead to faster convergence to the optimal policy.
- It is a model-free algorithm that does not require knowledge of the environment's dynamics or transition probabilities.
- It is suitable for problems where the agent interacts with the environment in a sequential manner and where the state and action spaces are not too large.

3.1.3 Advantages

- It can handle problems with continuous state and action spaces.
- It is an online algorithm that can learn from experience as it interacts with the environment.
- It can learn optimal policies in environments with stochastic rewards.

3.1.4 Disadvantages

- It can be sensitive to the initial values of the Q-table and may require exploration to converge to the optimal policy.
- It may suffer from the problem of overestimation of Q-values in certain scenarios, especially when the agent encounters rare events.

3.2 Applying Q-learning

1. First, the Q-table is initialized with zeros for all state-action pairs.

2. The Q-learning algorithm executes a loop for a specified number of episodes, where each episode represents a full interaction between the agent and the environment.

3. At the start of each episode, the environment is reset, and the initial state is obtained.

4. Inside the loop, the agent takes an action based on the current state using an epsilon-greedy policy. This means that the agent chooses a random action with a probability of epsilon, and otherwise chooses the action with the highest Q-value for that state.

5. The environment returns a new state and a reward based on the action taken by the agent.

6. The Q-value of the previous state-action pair is then updated using the Q-learning update rule, which is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{3}$$

   where $Q(S_t, A_t)$ is the Q-value for state S_t and action A_t, α is the learning rate, R_{t+1} is the reward obtained for the action taken, γ is the discount factor, $\max_{a'} Q(S_{t+1}, a')$ is the maximum Q-value for the new state S_{t+1}, and a' represents all possible actions from state S_{t+1}.

7. The total rewards obtained during the episode, epsilon, and the number of steps taken to complete the episode are then recorded.

8. After the loop finishes executing, the code uses matplotlib to plot the rewards, epsilon, and timesteps taken for each episode.

9. Finally, the Q-table is printed and displayed as a heatmap using seaborn.

10. A list of Q_REWARDS, EPS, TIMESTEPS, and the updated Q-table are returned as the output of the function.

Note that the Q-learning algorithm is similar to the SARSA algorithm described earlier, with the main difference being in the update rule used to update the Q-value for each state-action pair. SARSA updates the Q-value based on the next state and action chosen by the agent, while Q-learning updates the Q-value based on the maximum Q-value for the next state and all possible actions.

Table 12: Hyperparameters Setup

Hyperparameters	**Setup**
num_episodes	1000
alpha	0.2
gamma	0.95
maximum_epsilon	1
minimum_epsilon	0.01
epsilon_decay_rate	0.9954

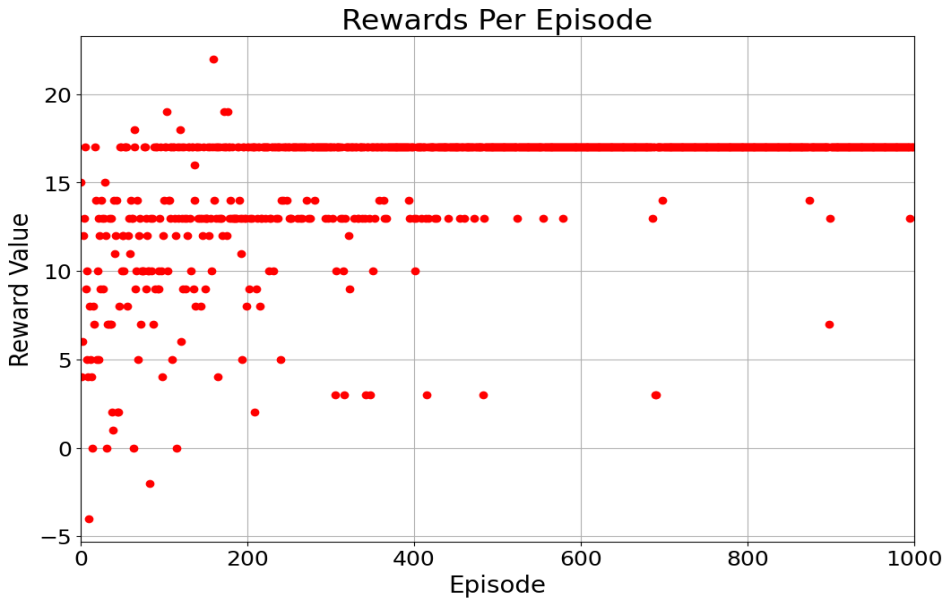

Figure 19: Rewards vs Episode

The graph displays the total rewards obtained by the agent in each episode. Initially, as the agent explores the environment, its exploratory behavior results in lower rewards. However, as the agent's Q-table gets updated with experience and epsilon value is reduced, the agent shifts towards maximizing rewards, which explains the increasing trend of total rewards over time.

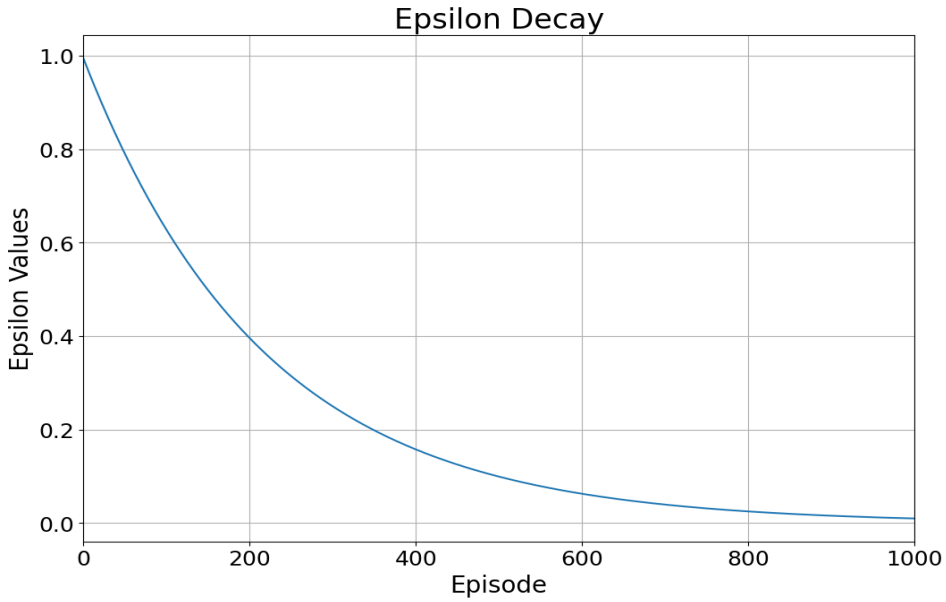

Figure 20: Epsilon vs Episode

The graph shows a gradual decrease in the number of episodes as the epsilon value decreases.

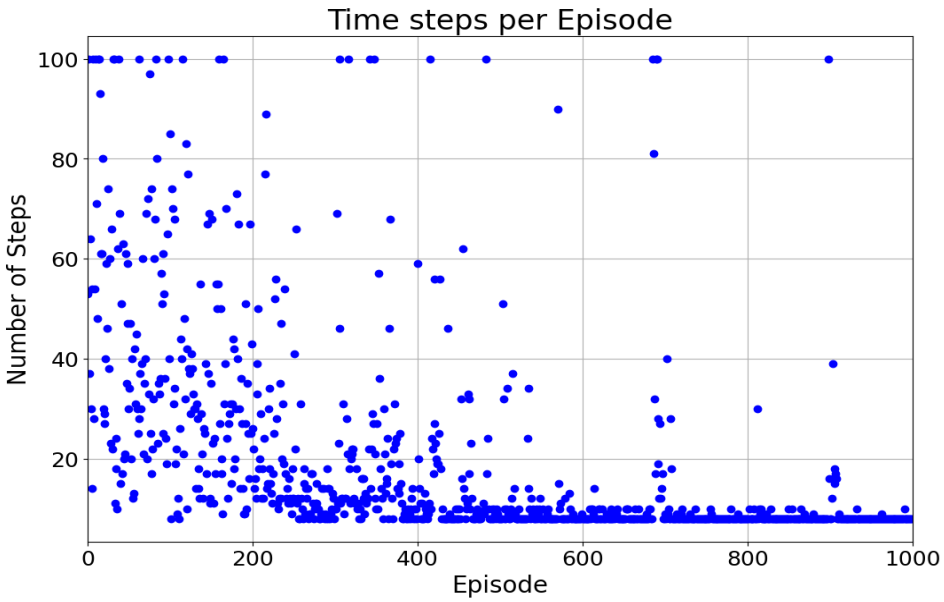

Figure 21: Number of Steps vs Episode

The graph shows the number of timesteps taken by the agent to reach the goal. Initially, during the early stages of training, the q-values remain unchanged and the exploration rate, represented by the epsilon value, is high. This forces the agent to explore more, resulting in a higher number of timesteps required to reach the goal. However, as training progresses, the agent's timestep count decreases as it gains more knowledge about the environment and learns to exploit this knowledge to reach the goal more efficiently.

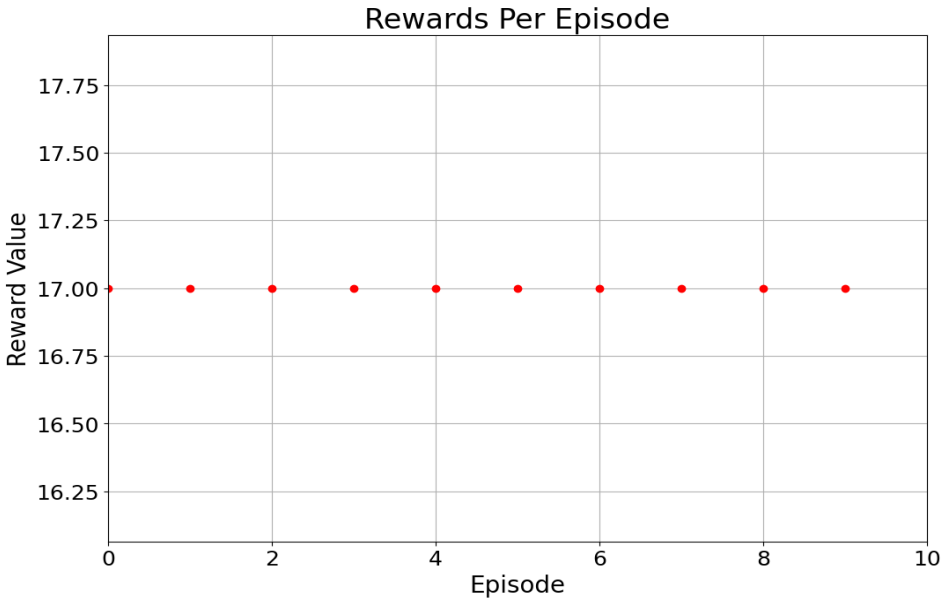

Figure 22: Rewards vs Episode

The Q-learning algorithm was evaluated by plotting the agent's performance over 10 episodes, during which it was instructed to take only greedy actions. The resulting graph indicates that the agent achieved a maximum reward of 17 in each episode.

3.3 Hyperparameter Tuning

During each iteration, the values of num_episodes, alpha, gamma, and epsilon remained constant, while only the parameters of maximum_epsilon, minimum_epsilon, gamma, and number_episodes were modified.

3.3.1 Min/Max Epsilon

Table 13: Hyperparameters Setup

Hyperparameters	Setup
maximum_epsilon	0.9
minimum_epsilon	0.2
epsilon_decay_rate	0.9984

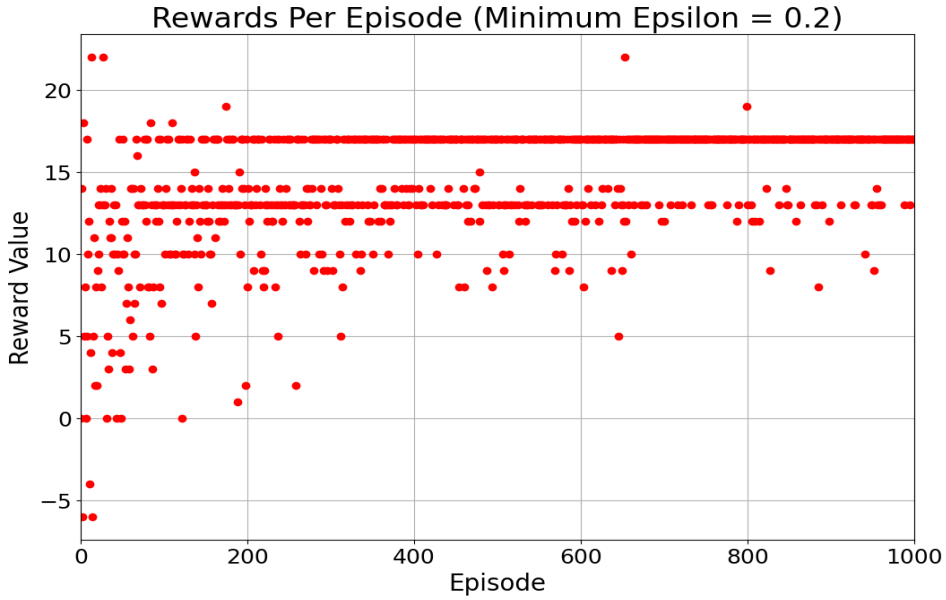

Figure 23: Rewards vs Episode

As the number of episodes increases, the rewards tend to deviate from the initial configuration, which is apparent from the plotted results.

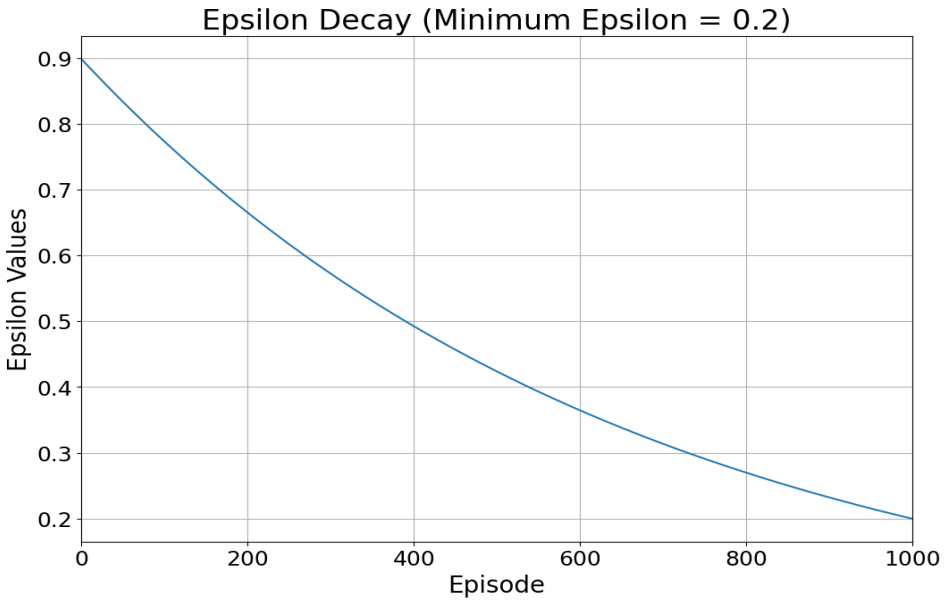

Figure 24: Epsilon vs Episode

The plot shows that as the number of episodes increases, the change in epsilon becomes more linear at the elbow point.

Table 14: Hyperparameters Setup

Hyperparameters	**Setup**
maximum_epsilon	0.8
minimum_epsilon	0.3
epsilon_decay_rate	0.9990

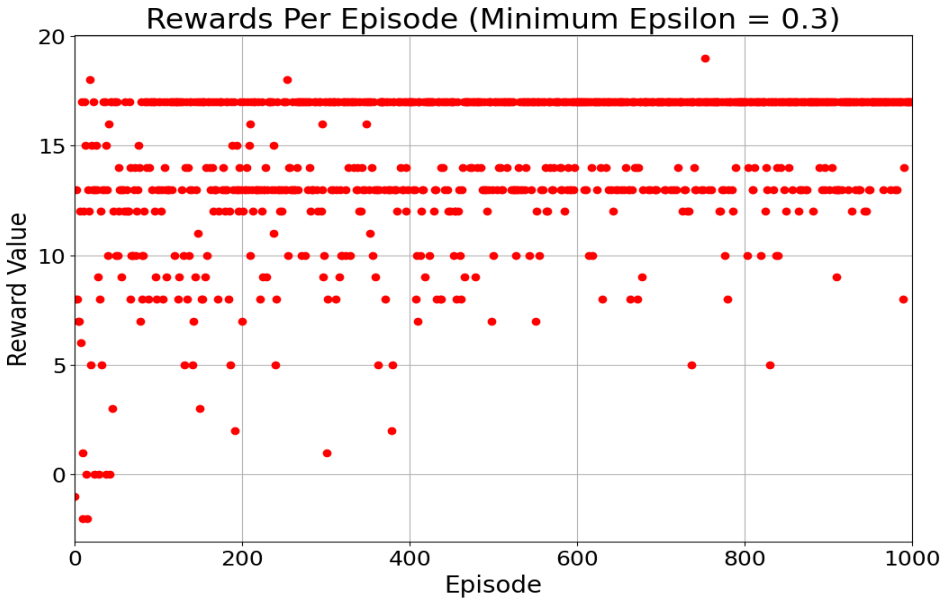

Figure 25: Rewards vs Episode

The plot clearly shows that as the number of episodes increases, reducing the maximum/minimum value of epsilon leads to a significant divergence of rewards from the previous configuration.

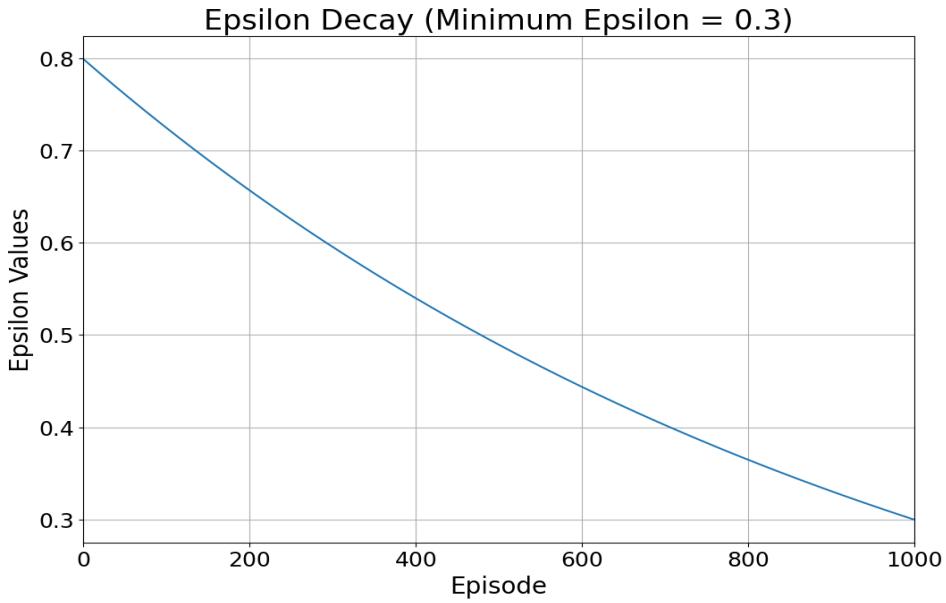

Figure 26: Epsilon vs Episode

The plot suggests that as the maximum and minimum epsilon values decrease, the elbow in the epsilon curve tends to become more linear as the number of episodes increases.

Table 15: Hyperparameters Setup

Hyperparameters	**Setup**
maximum_epsilon	0.7
minimum_epsilon	0.4
epsilon_decay_rate	0.9994

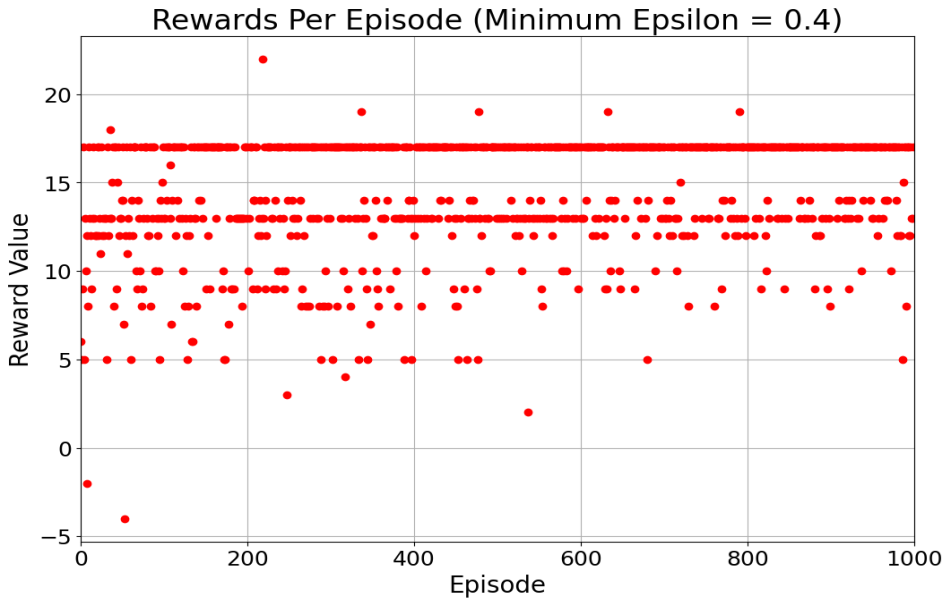

Figure 27: Rewards vs Episode

The plot indicates that as the number of episodes increases, reducing the maximum and minimum epsilon values leads to a greater divergence of rewards compared to the previous settings. By increasing the minimum epsilon value and decreasing the maximum epsilon value, we are limiting the agent's ability to use its knowledge (exploit) and explore more.

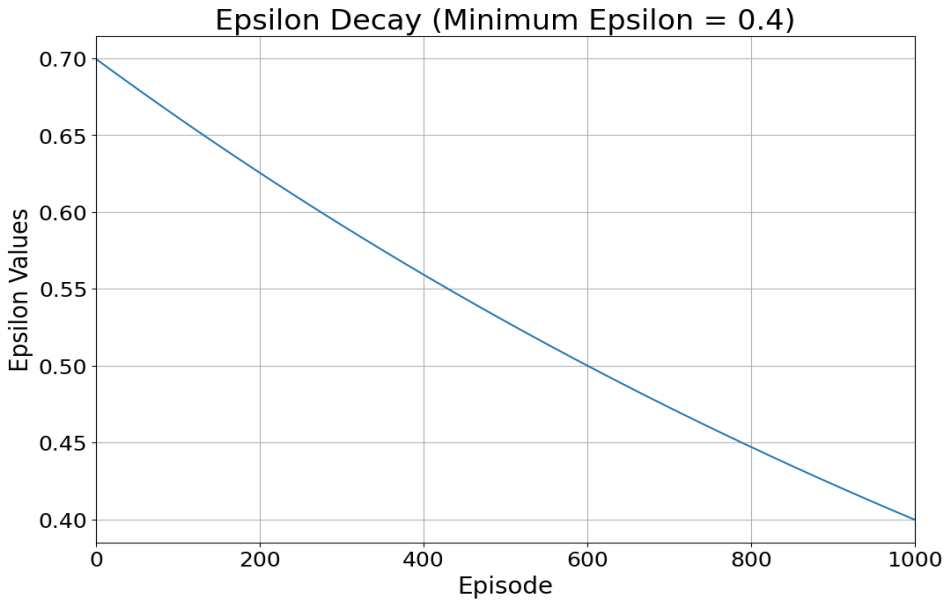

Figure 28: Epsilon vs Episode

As the maximum/minimum value of epsilon decreases, the epsilon curve shows a more linear inclination towards a rise in the number of episodes, as observed in the plot.

The results indicate that the performance of the reinforcement learning agent is significantly influenced by the choice of episode count and maximum/minimum epsilon values. Increasing the

number of episodes results in a better policy and higher cumulative rewards. However, reducing the maximum/minimum epsilon values can lead to a divergence of rewards from the initial configuration, highlighting the importance of selecting optimal epsilon values.

3.3.2 Gamma

Table 16: Hyperparameters Setup

Hyperparameters	Setup
gamma	0.7

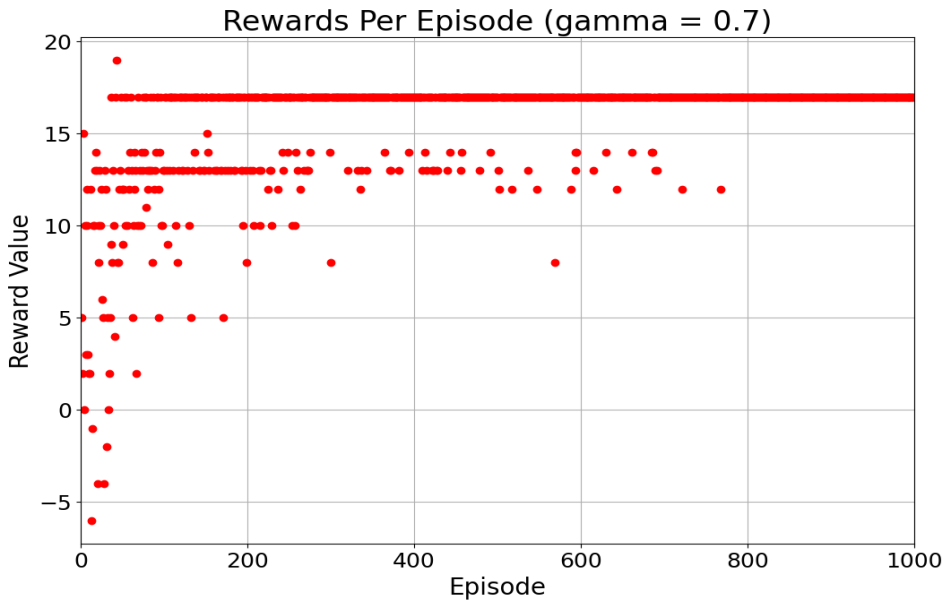

Figure 29: Rewards vs Episode

When the gamma value is set to 0.7, the agent seems to prioritize future rewards more compared to the other gamma values. This can be observed in the upward trend of cumulative rewards over episodes, indicating that the agent is progressively improving its ability to maximize its long-term returns.

Table 17: Hyperparameters Setup

Hyperparameters	Setup
gamma	0.4

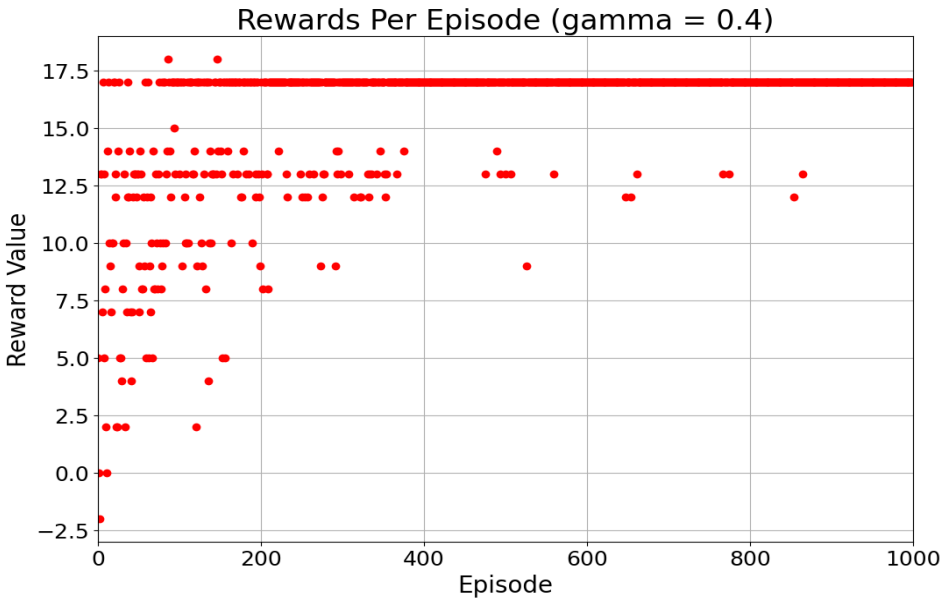

Figure 30: Rewards vs Episode

On the other hand, the graph for a gamma value of 0.4 demonstrates a more moderate approach where the agent neither prioritizes short-term gains nor long-term gains excessively. In this case, the cumulative rewards increase steadily over episodes, albeit at a slower rate compared to the graph with a higher gamma value.

Table 18: Hyperparameters Setup

Hyperparameters	**Setup**
gamma	0.2

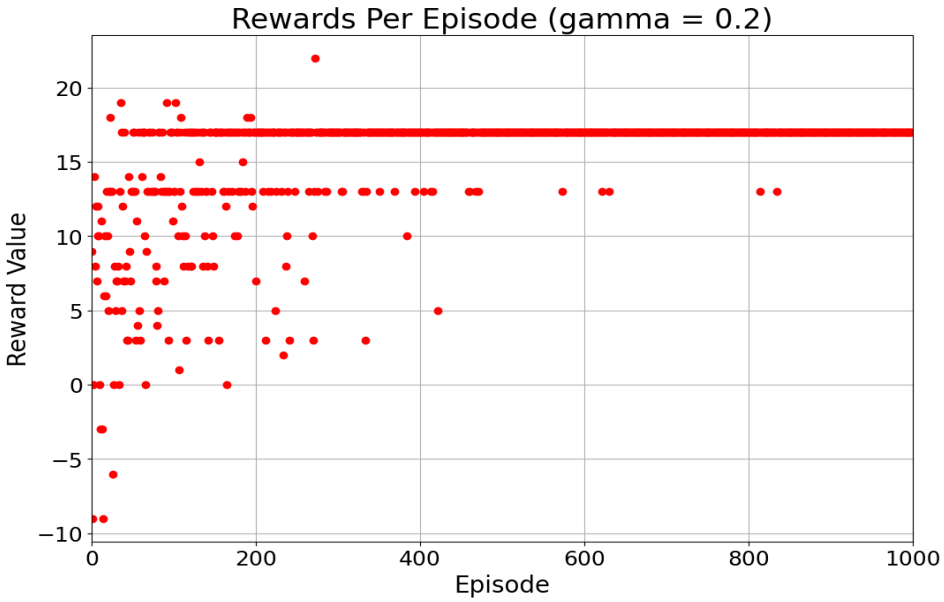

Figure 31: Rewards vs Episode

The graph for the agent with a gamma value of 0.2 indicates that it is more focused on immediate rewards and tends to overlook long-term rewards. As shown in the graph, there is a rapid increase in cumulative rewards in the early episodes, followed by a plateau in the later episodes. Although this strategy may lead to faster convergence, it could result in a suboptimal policy that ignores future rewards. On the other hand, a higher gamma value may lead to higher cumulative rewards in the long run, but it may also slow down the convergence during training.

3.3.3 Episodes

Table 19: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	700

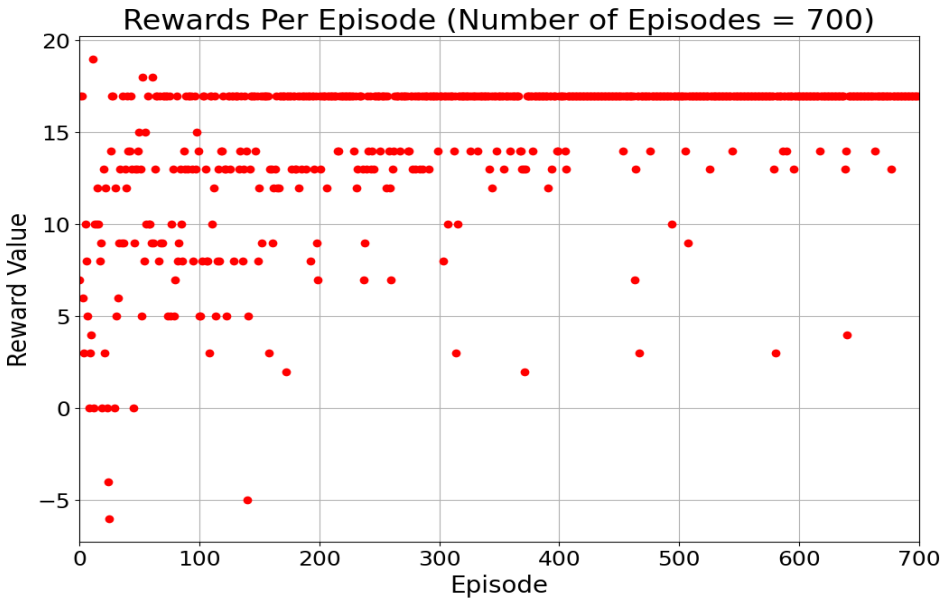

Figure 32: Rewards vs Episode

The graph with 700 num_episodes shows that the agent's performance improves gradually over time. The cumulative rewards increase noticeably in the initial episodes, followed by a steady rise in the later episodes. This indicates that the agent requires more episodes to converge to an optimal policy and achieve maximum long-term returns.

Table 20: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	400

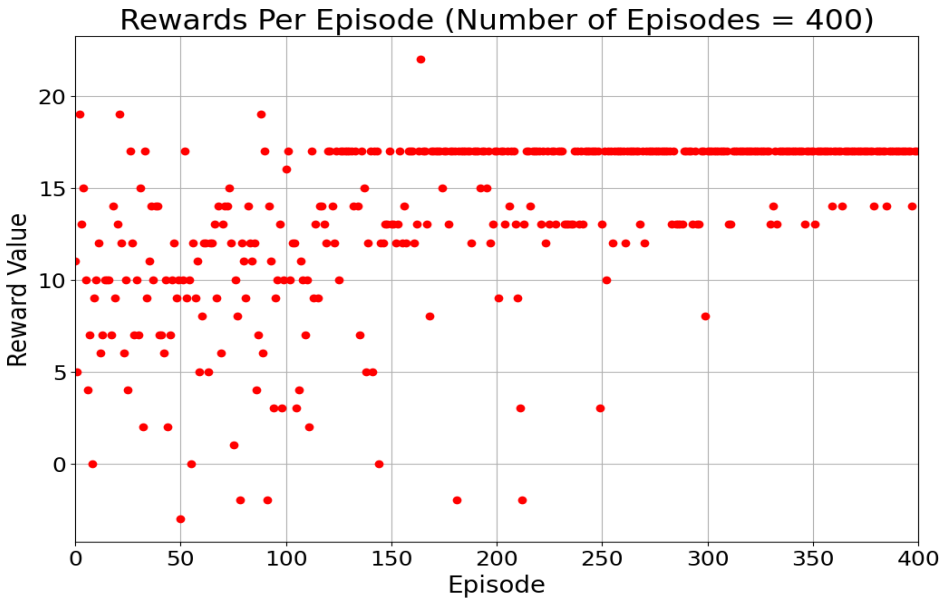

Figure 33: Rewards vs Episode

On the other hand, the graph with 400 num_episodes demonstrates a more rapid convergence, where the cumulative rewards reach a plateau after around 200 episodes. This implies that the agent can learn the optimal policy more quickly, but it may not be able to explore the state-action space entirely, which could lead to missed opportunities for potentially greater rewards.

Table 21: Hyperparameters Setup

Hyperparameters	**Setup**
num_episodes	200

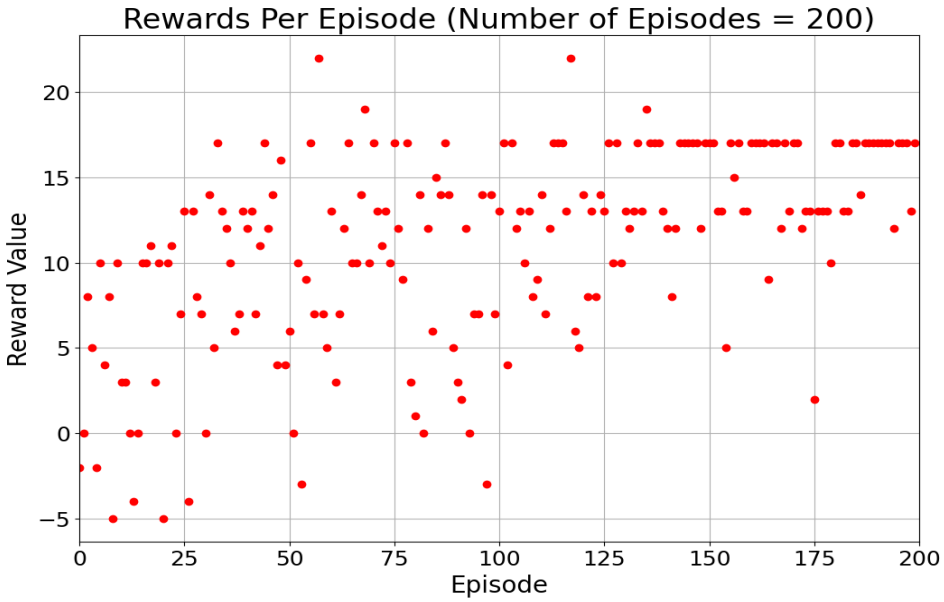

Figure 34: Rewards vs Episode

Finally, the plot generated with only 200 episodes demonstrates an unstable learning curve, characterized by significant fluctuations in cumulative rewards across different episodes. This observation indicates that the agent did not have enough opportunities to explore the state-action space and learn an optimal policy. Consequently, the learned policy may prioritize immediate rewards over long-term ones, resulting in a suboptimal policy.

3.4 Efficient Hyper-parameters

After making adjustments and fine-tuning all the parameters, we have determined that the original model outperformed the others in overall performance, with the following parameter values:

Table 22: Hyperparameters Setup

Hyperparameters	Setup
num_episodes	1000
alpha	0.2
gamma	0.95
maximum_epsilon	1
minimum_epsilon	0.01
epsilon_decay_rate	0.9954

3.5 Comparison

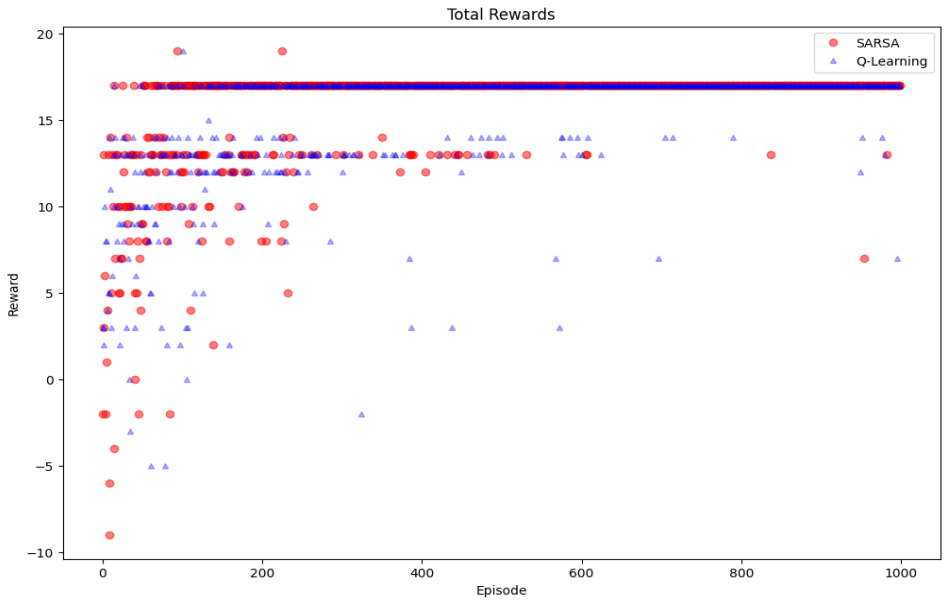

Figure 35: Rewards vs Episode

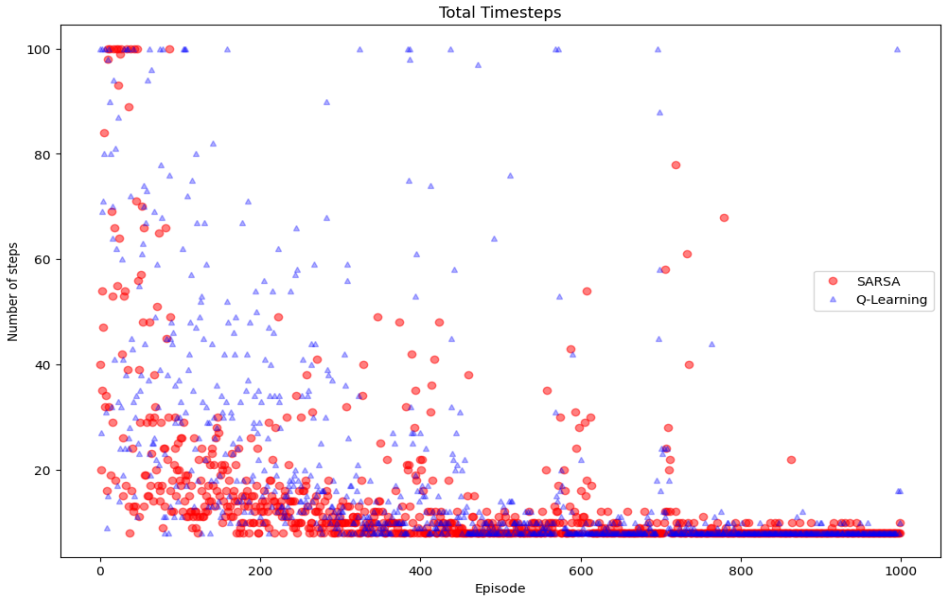

Figure 36: Number of steps vs Episode

In our environment, SARSA performs slightly better than Q-learning because of the nature of the environment. SARSA is an on-policy learning algorithm, which means it learns the value of the policy being used to make decisions while Q-learning is an off-policy learning algorithm, which learns the value of the optimal policy. In our environment, the agent (mouse) needs to explore and learn the optimal policy, which might not be possible with an off-policy learning algorithm like Q-learning.

For example, suppose the agent initially starts exploring and chooses an action that results in a negative reward. In that case, SARSA will update its policy according to the same exploration policy, leading to potentially better exploration of the environment. In contrast, Q-learning will update its policy based on the maximum Q-value of the next state, which might not always result in the optimal policy in this environment.

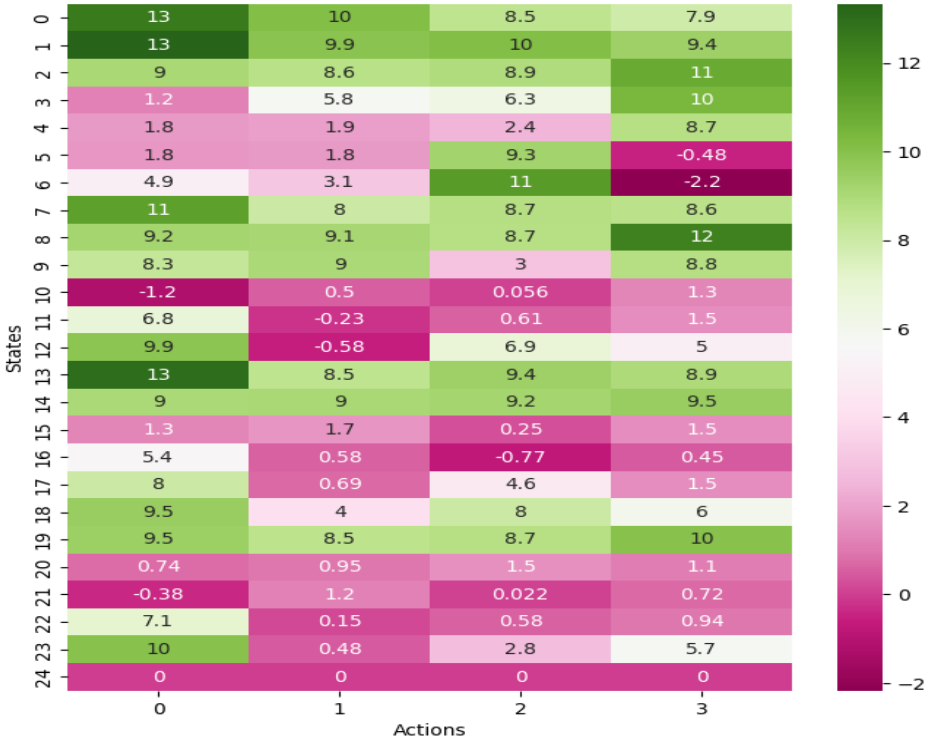

Figure 37: Q-matrix of SARSA

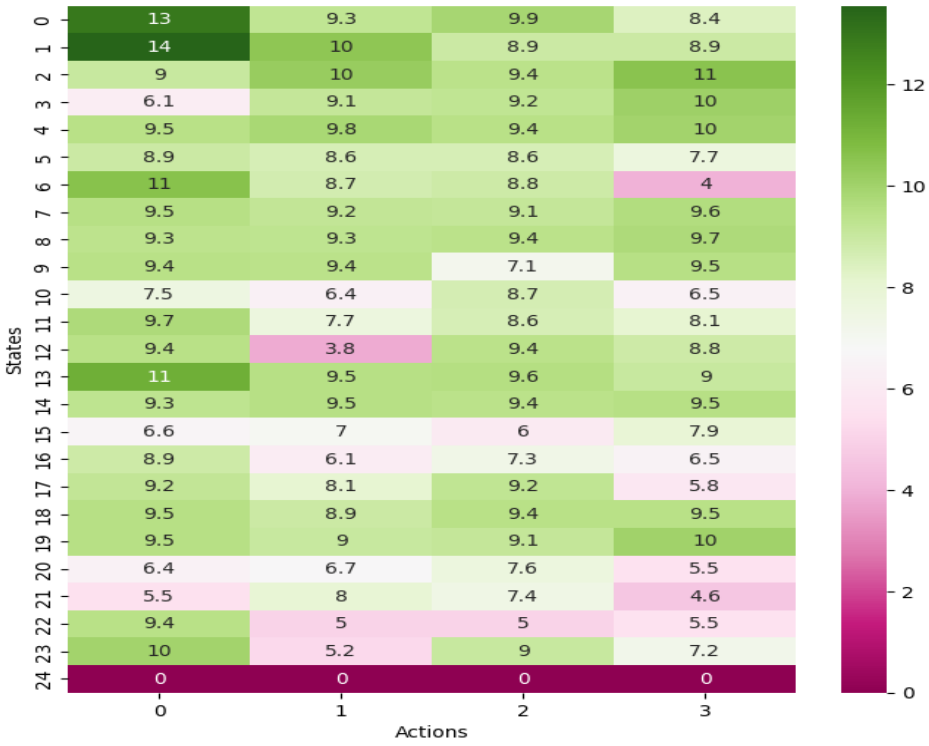

Figure 38: Q-matrix of Q-learning algorithm

The Q-values in SARSA are smaller compared to Q-learning because SARSA follows an on-policy learning approach, while Q-learning follows an off-policy learning approach.

In SARSA, the agent updates its Q-values based on the actions it actually takes during the learning process. The Q-values represent the expected return when starting from a state, taking a specific action, and then following the current policy for action selection. Since SARSA updates Q-values based on the actions it takes, the Q-values tend to be more conservative and reflect the expected return under the current policy. This can result in smaller Q-values overall.

On the other hand, Q-learning is an off-policy learning algorithm that updates Q-values based on the maximum Q-value of the next state, regardless of the action actually taken. Q-learning estimates the optimal Q-values regardless of the policy being followed. As a result, Q-learning can potentially converge to higher Q-values because it considers the maximum possible return in each state, even if it is not the action chosen by the current policy.

The smaller Q-values in SARSA do not necessarily indicate poorer performance. SARSA is more conservative and can be more suitable for situations where the agent needs to be cautious. Q-learning, on the other hand, focuses on maximizing the long-term return and can be more suitable when the agent's exploration is decoupled from its policy.

4 Bonus task

We have made modifications to the SARSA algorithm and implemented a 2-step bootstrapping SARSA. We then compared the results of this modified algorithm with the original SARSA algorithm. This comparison involved evaluating the performance of both algorithms and analyzing the results obtained.

Figure 39: Rewards vs Episode

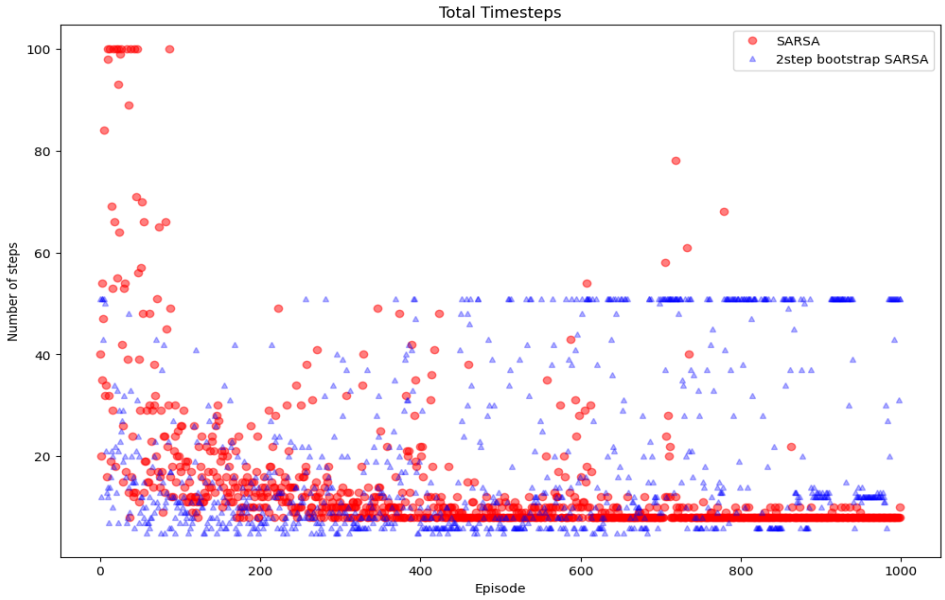

Figure 40: Number of steps vs Episode

In our environment, SARSA has performed better because it considers the immediate next state and action to update its Q-values. On the other hand, 2-step bootstrap SARSA considers the next two steps ahead, which is not advantageous in our case.

The rewards in the environment are sparse and have a high magnitude, which means that immediate rewards play a significant role. SARSA tends to have a cautious exploration strategy and is better suited to handle sparse rewards compared to 2-step bootstrap SARSA.

However, it's important to note that the performance of reinforcement learning algorithms are influenced by several factors, such as the exploration rate, learning rate, discount factor, and the number of training episodes.

References

1. https://stackoverflow.com/
2. RL Environment Visualization by Nitin Kulkarni Link
3. https://www.gymlibrary.dev/content/environment_creation/