University of Sheffield

COM3240: Reinforcement Learning Assignment

Zaid Daghash

May 2024

Table of Contents

Introduction

With the upwards trajectory of technological advancements in today's modern era, machine learning has become quite popular at the height of these advancements [1]. Machine learning gives computer systems the ability to learn without being programmed by an external entity [2]. A machine learning algorithm can be defined by how it is trained, how it makes decisions, and how it uses data to learn. The main methods in how an algorithm is trained is through supervised, semi-supervised, unsupervised, and reinforcement learning. Unlike the former, reinforcement learning works in an environment where the agent learns by exploring and receiving feedback in the form of penalties or rewards [3]. Using this process, the agent will aim to achieve the maximum reward available.

In the first task of this assignment, a robot will navigate through a grid in search of the single target and avoid the punishments which are in the form of lava tiles. In the second task, the environment will contain two targets and the rewards are stochastic. The robot will then need to identify the reward with the highest value and navigate to it. In the final task, the objective is the same as the second task, however the transitions are now also stochastic. This implies that the robot might take an action that takes it to a different state other than the expected state.

The algorithms Q-learning and State-action-reward-state-action (SARSA) will be used to solve the tasks. SARSA is referred to as an on-policy algorithm since it takes actions and improves the actions taken based on the current policy. On the other hand, Q-learning is referred to as an off-policy algorithm since it takes actions based on the optimal policy (policy that yields the highest rewards) [4]. When deciding on an action to take, the SARSA algorithm utilises the epsilon-greedy policy which uses exploitation and exploration to determine the action that gives the highest reward. The SARSA algorithm uses the epsilon greedy policy to determine the action to take in a state, and then also uses it to determine the ideal actions to take that will give the best reward. Whereas Q-learning does not consider exploration and aims for exploitation by selecting the action that gives the highest reward, regardless of what the exploration actions are [4].

Task 1 – Deterministic Rewards

Figure 1 illustrates the average rewards accumulated across 5000 episodes and averaged against 50 runs when the algorithms were applied to the environment in task 1. Figure 2 shows the average steps taken per episode for both algorithms.

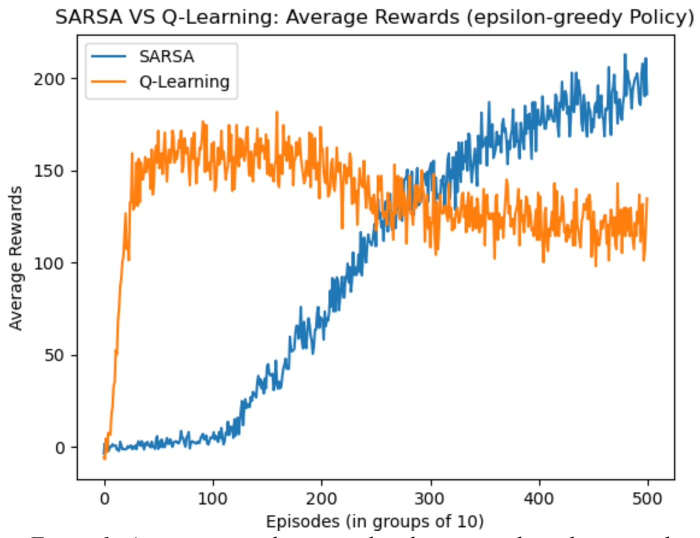


Figure 1: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the epsilon-greedy policy.. The algorithms were applied to the environment in task 1.

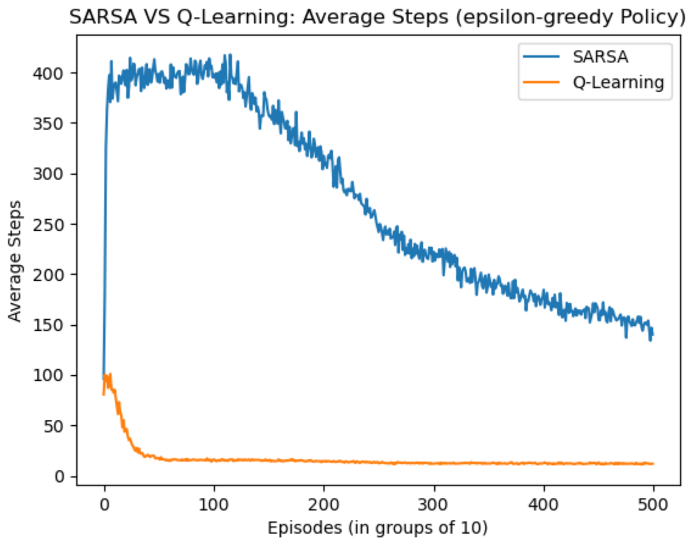


Figure 2: Average steps (averaged across 50 runs) taken by the Q-learning and SARSA algorithms when applied to the environment in task 1.

Based off of Figure 1, both algorithms eventually converge to the optimal policy. By observing Figure 1 it is evident that the SARSA algorithm performs better than the Q-learning algorithm as it achieves higher average rewards. The Q-learning algorithm initially achieves higher rewards since it is an off-policy algorithm. This allows it to update its Q-values using the maximum value of the next state-action pair, regardless of the action taken. SARSA on the other hand has a longer execution time due to its on-policy nature. This means that it updates its Q-values based on the action that will be taken, hence the algorithm spends more time exploring the environment. Since the SARSA algorithm spends more time exploring, the average steps taken by the SARSA algorithm are much higher than the average steps taken by the Q-learning algorithm. This is seen in Figure 2.

The hyperparameters for both algorithms were set as follows:

- Episodes = 5000
- γ = 0.99
- α = 0.1
- ε = 0.5

Figure 3 shows the comparison between the SARSA and Q-learning algorithms when implemented on the environment in task 1 with the exploration rate (ε) value set to 0.5 and 0.1.

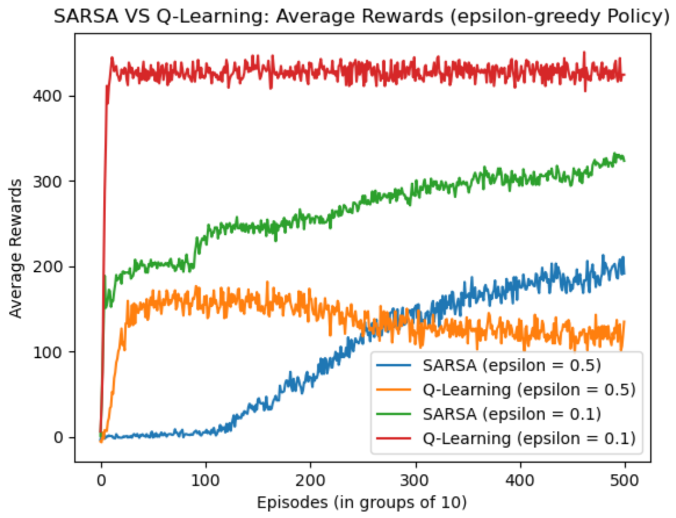


Figure 3: Comparison between the SARSA and Q-learning algorithms when implemented on the environment in task 1 with the exploration rate (ε) value set to 0.5 and 0.1.

Figure 3 clearly indicates that setting the exploration rate closer to 0 increases the average rewards achieved across the episodes for both algorithms. However, the number of episodes needs to be increased to allow the SARSA algorithm to converge towards the highest possible average reward it can achieve.

The learning rate (α) is used to determine to what extent new information replaces old information. A large learning rate allows the agent to adapt to new information, but it might forget old information. A low learning rate ensures the retention of old information, which will then lead to slow learning, but the convergence will be stable [5]. This is the reason why the learning rate was set to 0.1. The discount factor (γ) is used to highlight the importance of future rewards. A γ value close to 1 will enable the agent to value long-term rewards. Whereas a γ value close to 0 will put an emphasis on immediate rewards [6]. Since the given environment has a single fixed reward, and the objective is to maximise the number of rewards achieved, it is best to choose a γ value closer to 1. The exploration rate (ε) is used within the epsilon-greedy policy. It represents the probability of selecting a random action instead of selecting the action that will give the highest reward. Setting the exploration rate closer to 1 will favour exploration, whereas setting it closer to 0 will favour exploitation [7]. Once again since the aim is to maximise the average rewards, it is optimal to set the exploration rate closer to 0. The assignment briefing mentioned that the number of episodes should be more than 1000 in order to discover the location of the most precious artifact, hence why the number of episodes was set to 5000.

Figures 4 and 5 show the preferred actions taken in each grid location by both algorithms. The following defines what the coloured grids represent:

- Yellow grid = agent's starting location.
- Red grid = lava location (negative reward received and episode terminates).
- Black grid = wall location (agent cannot navigate to that location as its an obstacle).
- Blue grid = target location (positive reward received and episode terminates).

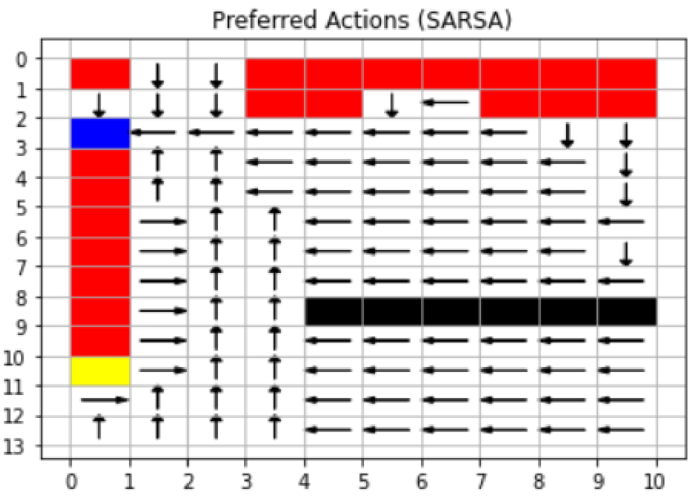


Figure 4: Preferred actions taken by the SARSA algorithm in the environment in task 1.

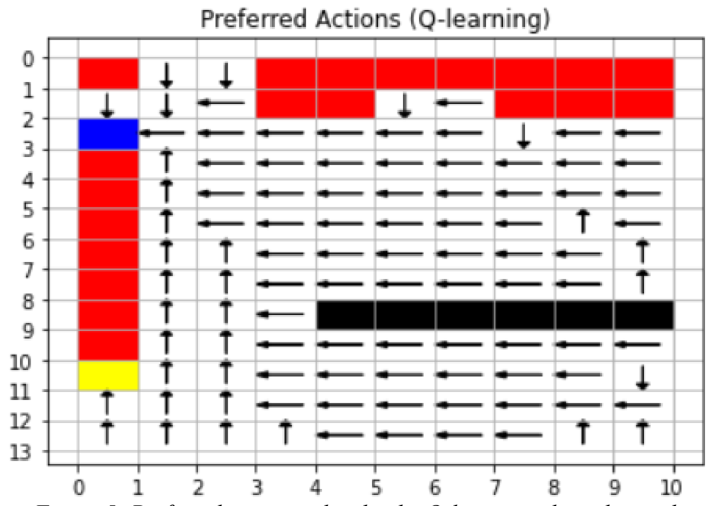


Figure 5: Preferred actions taken by the Q-learning algorithm in the environment in task 1.

It is evident from observing Figures 4 and 5 that both algorithms eventually converge to the target's location.

However, the Q-learning algorithm takes the optimal route since it learns the optimal action for each state-action pair regardless of the applied policy. SARSA on the other hand can be seen taking sub-optimal routes as it is an on-policy algorithm. Which means it learns the preferred action by following the applied policy. Therefore, the SARSA algorithm takes a longer time exploring the environment and hence takes sub-optimal routes.

Figure 6 illustrates the average rewards achieved across 5000 episodes and averaged across 50 runs when applying the Q-learning and SARSA algorithms to the environment in task 1 using the Softmax policy.

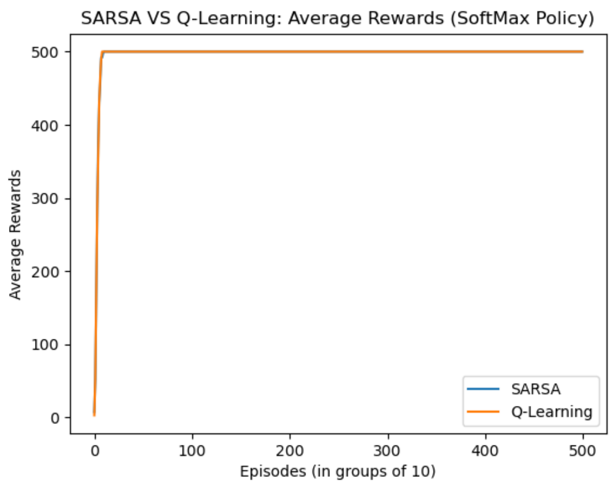


Figure 6: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the Softmax policy.. The algorithms were applied to the environment in task 1.

Based off Figures 6 and 1, it can be observed that the Softmax policy significantly improves the rate of learning for both Q-learning and SARSA algorithms. The average rewards achieved by the SARSA algorithm using the Softmax policy is overlapping the average rewards achieved by the Q-learning algorithm. It is also interesting to note that both algorithms do not have any fluctuations (oscillations) in the average rewards they are achieving. This means that when the agent discovers the location of the reward, it will remember the optimal route and proceed to take this route for the remainder of the runs. The Softmax policy performs better than the epsilon-greedy policy for the following reasons:

1. The Softmax policy balances between exploitation and exploration in a better manner by selecting actions with greater estimated rewards while allowing some exploration. Whereas the epsilon-greedy policy explores randomly with a fixed probability (ε) and exploits the best action otherwise [8].

2. The epsilon-greedy policy may proceed to explore less optimal routes even after learning the optimal route. This leads to slower convergence and explains the distortions seen in Figure 1. However, the Softmax policy is more sensitive to the acquired state-action values, therefore the convergence is much quicker since it favours the actions that lead to the target.

Task 2 – Dual Targets & Stochastic Rewards

Figures 7-9 compare the different hyperparameter values in order to determine which values result in a better performance in the learning curves when applying both algorithms to the environment in task 2. Once again the number of episodes was set to 5000.

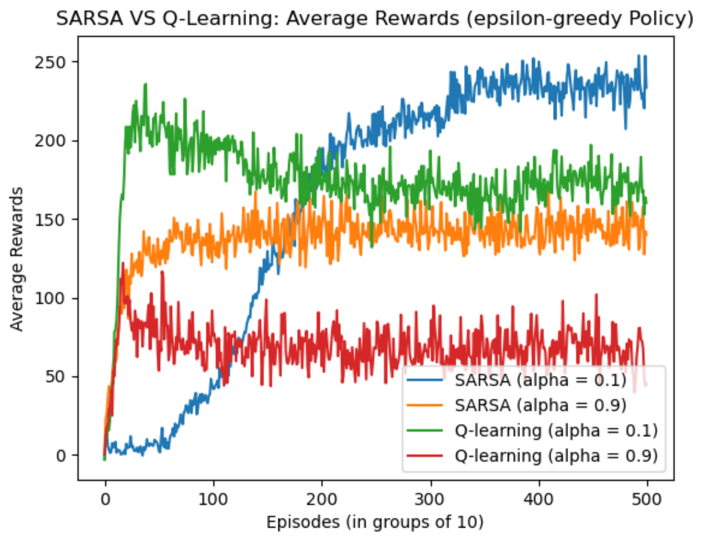


Figure 7: Average rewards achieved for different values of the learning rate for both algorithms when implemented on the environment in task 2 using the epsilon-greedy policy.

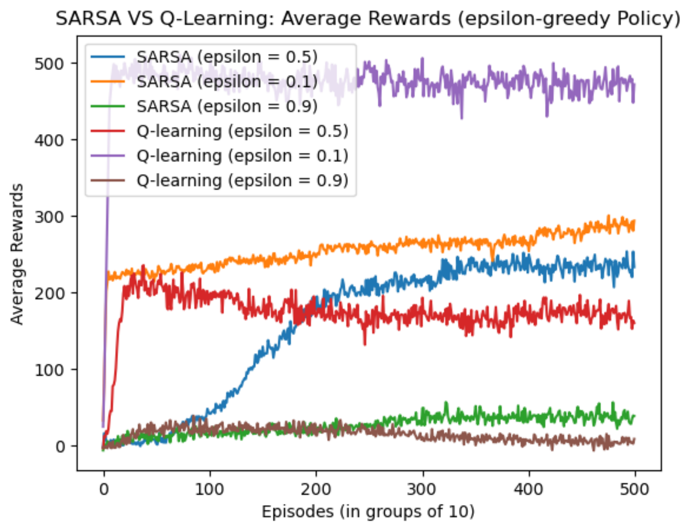


Figure 8: Average rewards achieved for different values of the exploration rate for both algorithms when implemented on the environment in task 2 using the epsilon-greedy policy.

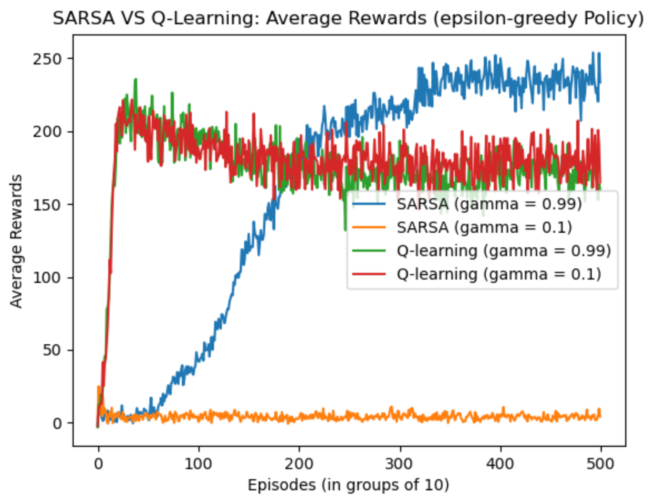


Figure 9: Average rewards achieved for different values of the discount factor for both algorithms when implemented on the environment in task 2 using the epsilon-greedy policy.

By analysing Figures 7-9, it is evident that the preferred hyperparameters for both algorithms is as follows:

- γ (Discount factor) = 0.99
- α (Learning rate) = 0.1
- ε (exploration rate) = 0.1

Using these parameters coupled with 5000 episodes and averaged across 50 runs, the performance of the SARSA and Q-learning algorithms when applied to the environment in task 2 can be seen in Figure 10.

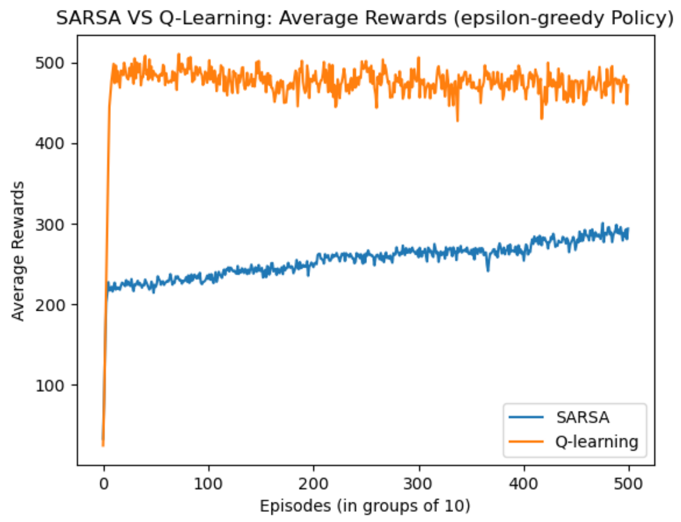


Figure 10: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the epsilon-greedy policy.. The algorithms were applied to the environment in task 2.

Based off Figure 10, it can be observed that the Q-learning algorithm outperforms the SARSA algorithm across 5000 episodes. However, based on the shape of the SARSA curve, it can be concluded that the SARSA algorithm has not yet converged to the best possible average reward. Therefore, given enough episodes, the SARSA algorithm will converge to the same (or even better) average rewards achieved by the Q-learning algorithm. This is similar to the performance observed in task 1 when the exploration rate for both algorithms was also set to 0.1 (Figure 3).

The agent needs to balance between exploring the environment and exploiting the known Q-values when presented with two target locations. The trade-off between exploitation and exploration becomes more complex and it will be more difficult for the agent to learn the location of the target with the highest value when the rewards are stochastic. Meaning that the value of each target location will fluctuate during training. The assignment of the rewards for each state-action pair in the Q-table will become more complex when there are multiple target locations, this will cause the agent to take actions that will result in longer routes (sub-optimal routes) to reach the location of the reward. The introduction of stochastic rewards causes the rewards achieved for each state-action pair values to change over time. This goes against the Markov property, which can then cause the convergence of the agent when using any of the algorithms to slow down, and hence converge to suboptimal policies.

The use of the Softmax policy instead of the epsilon-greedy policy can enhance the performance of the Q-learning and SARSA algorithms when implemented on an environment with multiple target locations and stochastic rewards. The Softmax policy provides smoother exploration when compared to the epsilon-greedy policy. The epsilon-greedy policy bases its exploration and exploitation strategies based on a fixed probability value (ε). This causes the algorithms to display erratic behaviour in the learning curves (distortions/oscillations) where the algorithms switch between exploration and exploitation abruptly. With the introduction of stochastic rewards, the Q-values stored in the Q-table will change over time, and the greedy action taken by the epsilon-greedy policy may change frequently. This abrupt change in actions taken can lead to slow convergence and instability. On the other hand, the Softmax policy enables the agent to adjust the action preferences incrementally based on the changing Q-values. This will cause the learning curves to be much smoother and the convergence will be much faster.

Figure 11 shows the average rewards achieved across 5000 episodes and averaged across 50 runs when applying the Q-learning and SARSA algorithms using the Softmax policy on the environment presented in task 2.

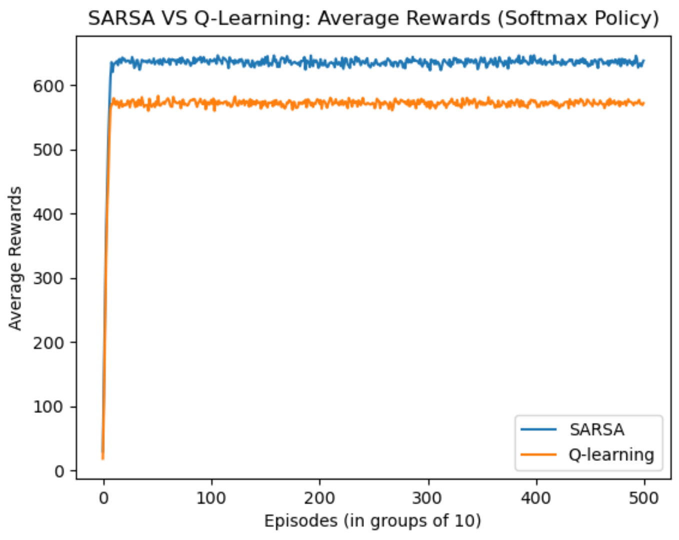


Figure 11: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the Softmax policy.. The algorithms were applied to the environment in task 2.

By observing Figure 11, it can be observed that the average rewards achieved by both algorithms when using the Softmax policy is much higher than when using the epsilon-greedy policy. This time the SARSA algorithm outperforms the Q-learning algorithm, and the convergence for both algorithms is much quicker.

Task 3 – Stochastic Transitions and Rewards

Figures 12-14 compare the different hyperparameter values in order to determine which values result in a better performance in the learning curves when applying both algorithms to the environment in task 3. Once again the number of episodes was set to 5000.

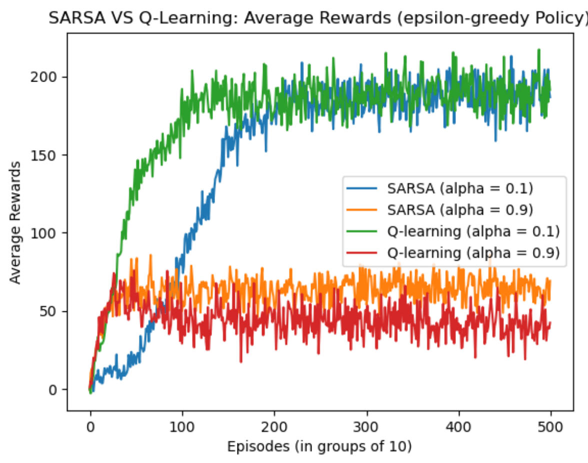


Figure 12: Average rewards achieved for different values of the learning rate for both algorithms when implemented on the environment in task 3 using the epsilon-greedy policy.

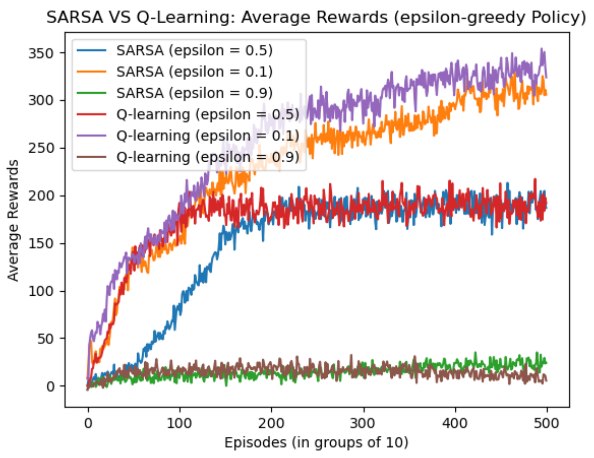


Figure 13: Average rewards achieved for different values of the exploration rate for both algorithms when implemented in the environment in task 3 using the epsilon-greedy policy.

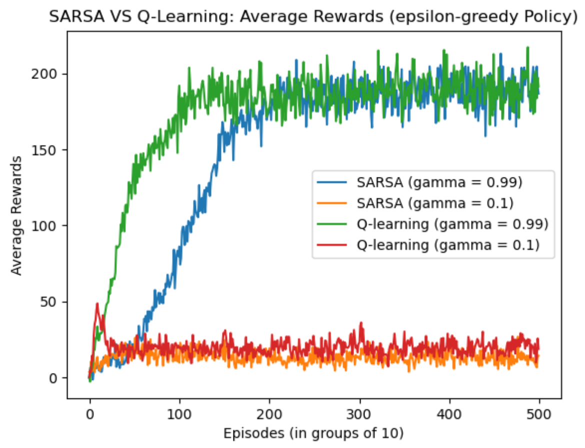


Figure 14: Average rewards achieved for different values of the discount factor for both algorithms when implemented on the environment in task 3 using the epsilon-greedy policy.

By analysing Figures 12-14, it is evident that the preferred hyperparameters for both algorithms are as follows:

- γ (Discount factor) = 0.99
- α (Learning rate) = 0.1
- ε (exploration rate) = 0.1

Using these parameters coupled with 5000 episodes, the performance of the SARSA and Q-learning algorithms when applied to the environment in task 3 can be seen in Figure 15.

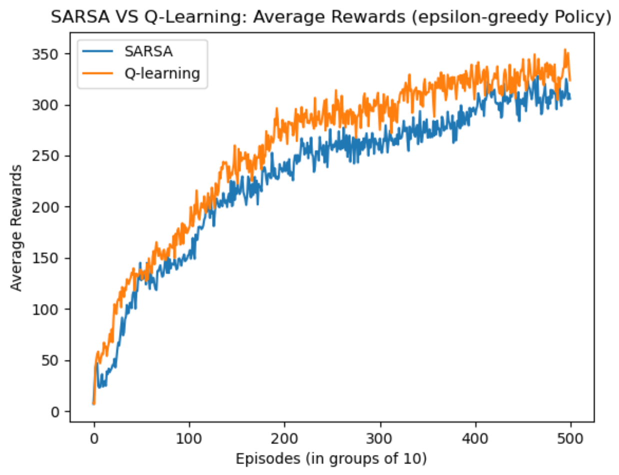


Figure 15: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the epsilon-greedy policy. The algorithms were applied to the environment in task 3.

In Figure 15, the performance of both algorithms is similar in terms of average rewards. Across 5000 episodes, both algorithms have still not converged towards the best possible average reward, meaning that more episodes are required to allow for the learning curves to stabilise. However, the performance of the algorithms in task 3 is much better than the performance seen in tasks 1 and 2 using the epsilon-greedy policy since the climbing rate in the learning curves is faster, and the average rewards received is higher.

The addition of stochastic rewards and stochastic transitions introduces several challenges for the SARSA and Q-learning algorithms. In reinforcement learning, an environment is assumed to be Markovian. Meaning that the next state depends on the current state and action, and not on previous state-action pairs [9]. With the presence of stochastic transitions, the Markov property gets violated since the next state is influenced by previous states and actions, making it difficult for the agent to predict the state-action values. With the introduction of stochastic transitions, the complexity in exploration increases. The agent might need to spend more time exploring in order to accurately estimate the state-action values. This is due to the same action taken in the same state resulting in a different next-state and different rewards achieved. This makes it difficult for the agent to identify the optimal policy.

One way to resolve the issues presented by the addition of stochastics rewards and transitions is by implementing an eligibility trace. An eligibility trace is a provisional record that stores the state-action pairs that have been visited by an agent in an environment [10]. It is used as a method to tackle the problems presented when assigning rewards to state-action pairs especially in environments with stochastic rewards and transitions. An eligibility trace can balance between exploitation and exploration. When assigning rewards to previous state-action pairs, the agent can continue to learn and re-evaluate its reward estimates for those pairs, even when exploiting the best policy. This will then lead to faster convergence and more efficient exploration. In task 3, the same state-action pair can lead to different next states. This is due to the presence of stochastic transitions. With the implementation of an eligibility trace, a record is kept of the visited state-action pairs, regardless of the transitions taken. Even though the transitions are stochastic, the information stored in an eligibility trace can be used to update the value estimates for those state-action pairs [10].

With that being said, Figure 16 shows the average rewards achieved across 5000 episodes and averaged across 50 runs when implementing the Q-learning and SARSA algorithms with the use of an eligibility trace on the environment in task 3. A decaying factor (lambda) of 0.85 was used for the eligibility trace.

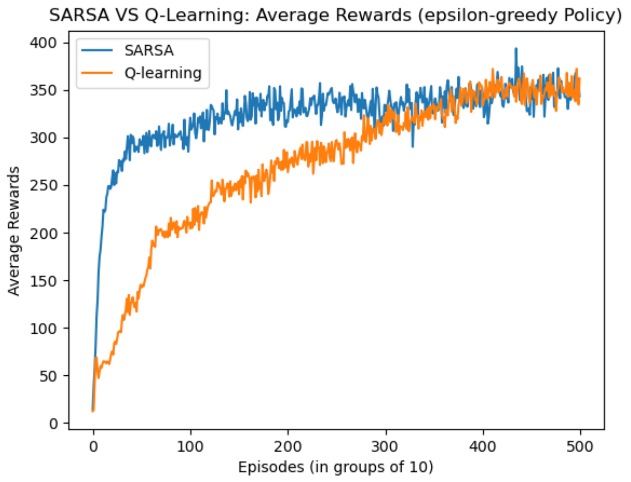


Figure 16: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the epsilon-greedy policy and an eligibility trace. The algorithms were applied to the environment in task 3.

Observing Figure 16 it is evident that the convergence has improved, and higher average rewards have been achieved. Alternatively, the Softmax policy can be used instead of the epsilon-greedy policy to improve the performance. This is presented in Figure 17.

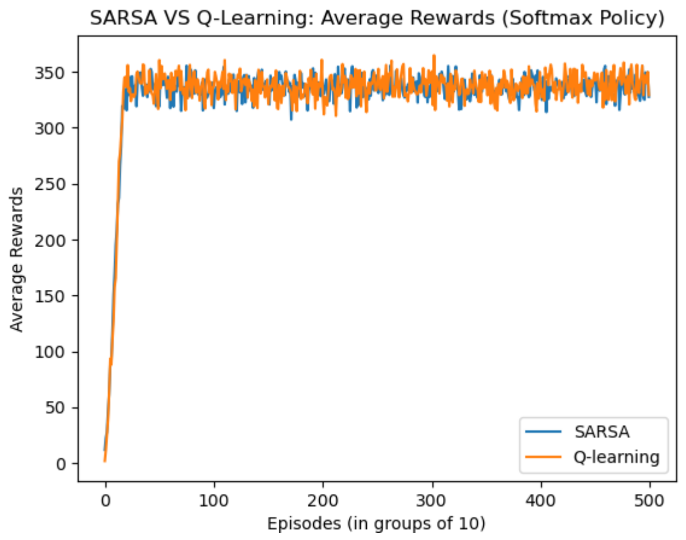


Figure 17: Average rewards accumulated per episode and averaged across 50 runs for the Q-learning and SARSA algorithms using the Softmax policy.. The algorithms were applied to the environment in task 3.

References

[1] "Why machine learning (ML) is so popular in the 21st century - GAMCO, SL." GAMCO. Accessed: Mar. 30, 2024. [Online]. Available: https://gamco.es/en/automatic-popular-popular-learning-xxi-century/

[2] V. Kanade. "What is machine learning? Understanding types & applications." Spiceworks. Accessed: Mar. 30, 2024. [Online]. Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-ml/

[3] "What is training algorithms concept and definition. glossary." GAMCO, SL. Accessed: Mar. 30, 2024. [Online]. Available: https://gamco.es/en/glossary/training-algorithms/#:~:text=Training%20algorithms%20can%20be%20supervised,patterns%20in%20an%20unlabelled%20dataset.

[4] A. Gautam. "SARSA reinforcement learning algorithm: A guide." Built In. Accessed: Mar. 30, 2024. [Online]. Available: https://builtin.com/machine-learning/sarsa

[5] M. Kwaśniak. "How to decide on learning rate." Medium. Accessed: Apr. 12, 2024. [Online]. Available: https://towardsdatascience.com/how-to-decide-on-learning-rate-6b6996510c98

[6] B. Or. "Penalizing the discount factor in reinforcement learning." Medium. Accessed: Apr. 12, 2024. [Online]. Available: https://towardsdatascience.com/penalizing-the-discount-factor-in-reinforcement-learning-d672e3a38ffe

[7] A. Makone. "Reinforcement learning 6: Exploration vs exploitation." Medium. Accessed: Apr. 12, 2024. [Online]. Available: https://ashutoshmakone.medium.com/reinforcement-learning-5-exploration-vs-exploitation-c1bae5a2ea42#:~:text=One%20way%20to%20keep%20a,good%20way%20to%20start%20initially.

[8] "Resmax: An alternative soft-greedy operator for reinforcement learning." Accessed: May 4, 2024. [Online]. Available: https://openreview.net/pdf?id=RjMtFbmETG#:~:text=ε-greedy%20explores%20in%20a,the%20value%20of%20the%20actions.

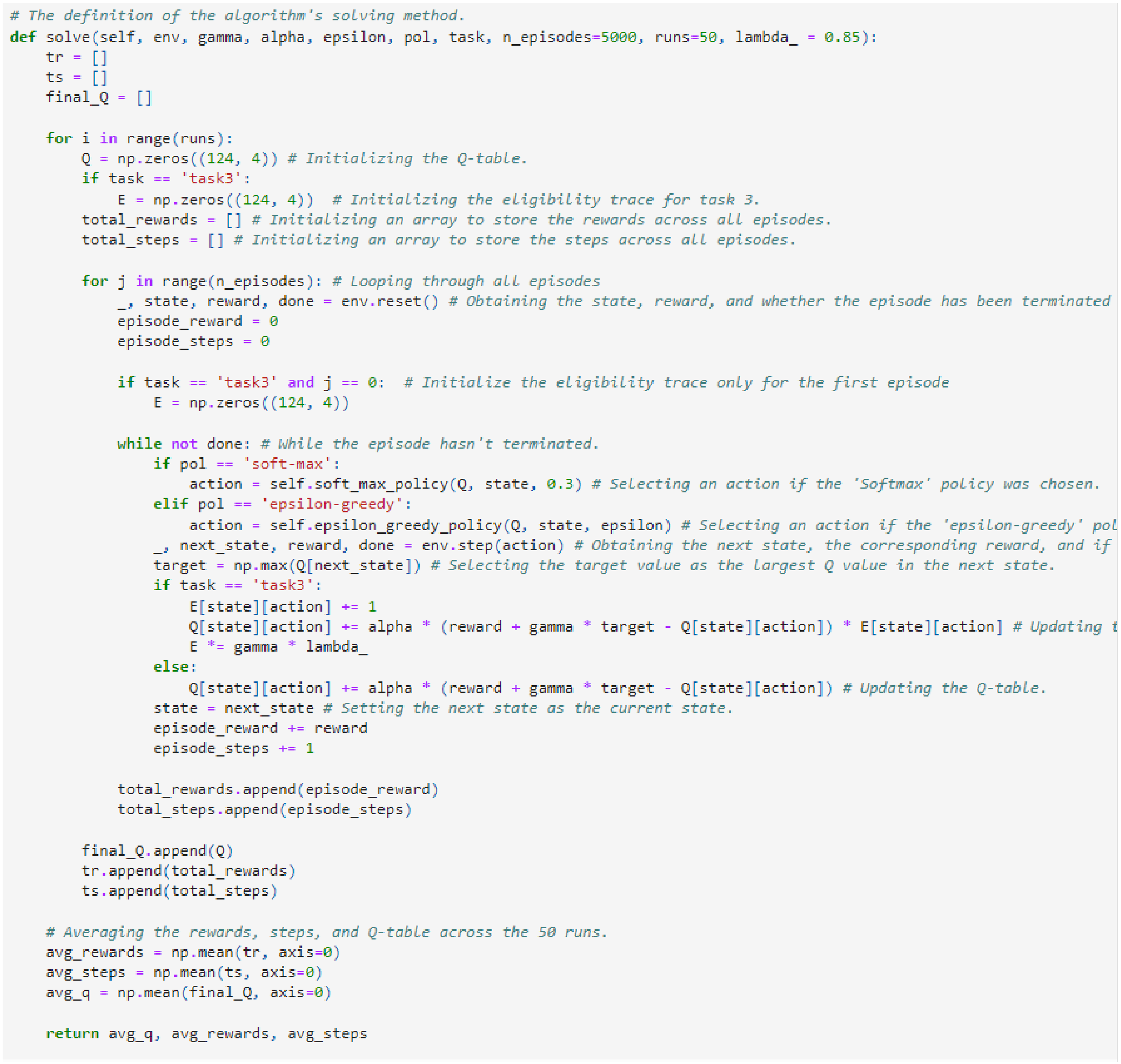
[9] D. Chruściński, S. Hesabi, and D. Lonigro, "On Markovianity and classicality in multilevel spin–boson models", *Scientific Rep.*, vol. 13, no. 1, Jan. 2023. Accessed: May 4, 2024. [Online]. Available: https://doi.org/10.1038/s41598-023-28606-z

[10] S. Mohan. "A brief overview of eligibility traces in reinforcement learning." Medium. Accessed: May 4, 2024. [Online]. Available: https://medium.com/nerd-for-tech/a-brief-overview-of-eligibility-traces-in-reinforcement-learning-c0a8326fa9f7

How to Reproduce the Figures in the Report

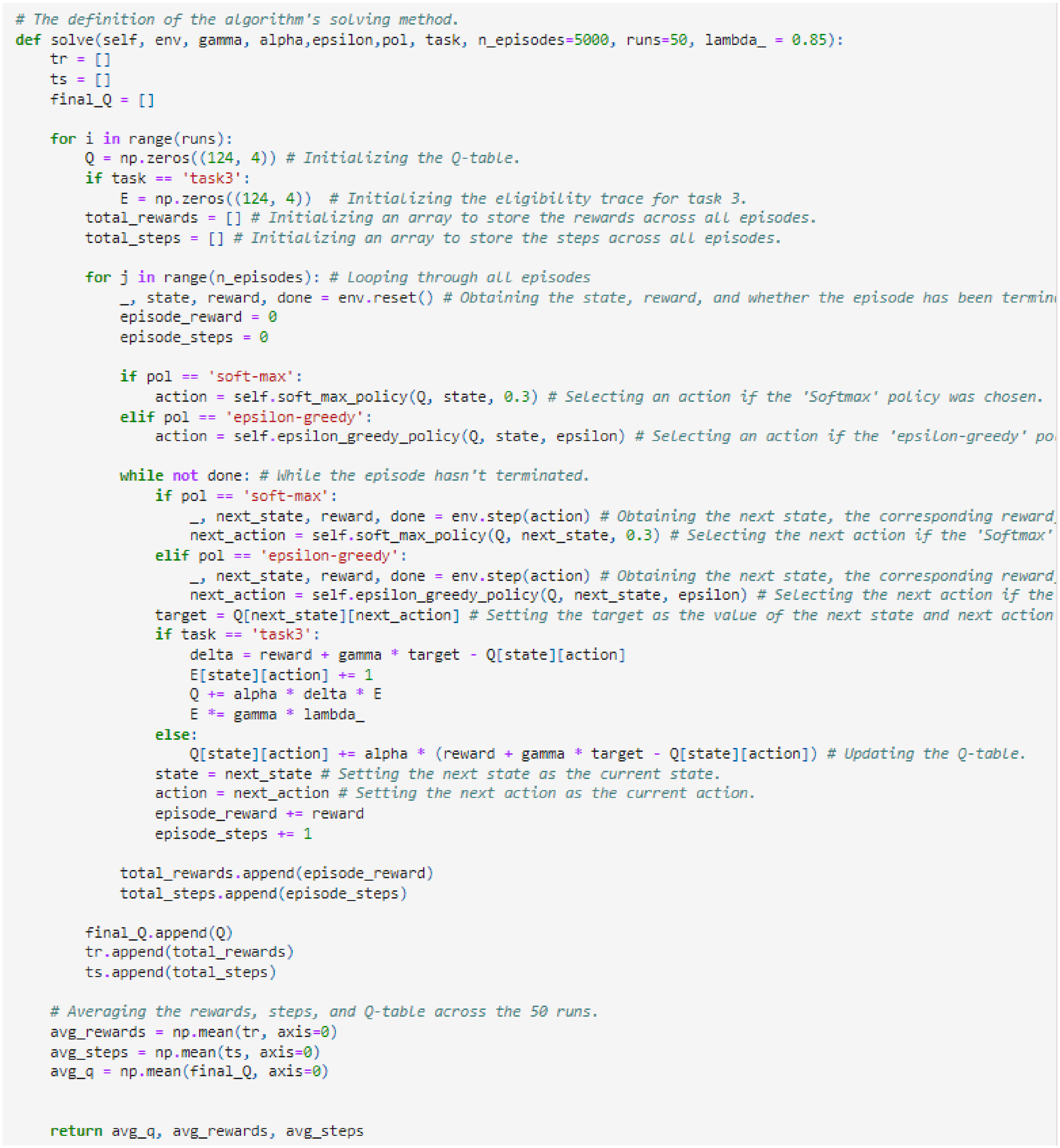
Please read the instructions found in the 'README.md' file to find out how to run the code and reproduce the figures found in the report. Make sure to read the documentation (comments) found in the code file ('Assignment_empty.ipynb') to understand the functionality of the code.

Appendix 1 – Training the agent using the Q-learning algorithm



```
# The definition of the algorithm's solving method.
def solve(self, env, gamma, alpha, epsilon, pol, task, n_episodes=5000, runs=50, lambda_ = 0.85):
    tr = []
    ts = []
    final_Q = []

    for i in range(runs):
        Q = np.zeros((124, 4)) # Initializing the Q-table.
        if task == 'task3':
            E = np.zeros((124, 4))  # Initializing the eligibility trace for task 3.
        total_rewards = [] # Initializing an array to store the rewards across all episodes.
        total_steps = [] # Initializing an array to store the steps across all episodes.

        for j in range(n_episodes): # Looping through all episodes
            _, state, reward, done = env.reset() # Obtaining the state, reward, and whether the episode has been terminated
            episode_reward = 0
            episode_steps = 0

            if task == 'task3' and j == 0:  # Initialize the eligibility trace only for the first episode
                E = np.zeros((124, 4))

            while not done: # While the episode hasn't terminated.
                if pol == 'soft-max':
                    action = self.soft_max_policy(Q, state, 0.3) # Selecting an action if the 'Softmax' policy was chosen.
                elif pol == 'epsilon-greedy':
                    action = self.epsilon_greedy_policy(Q, state, epsilon) # Selecting an action if the 'epsilon-greedy' pol
                _, next_state, reward, done = env.step(action) # Obtaining the next state, the corresponding reward, and if
                target = np.max(Q[next_state]) # Selecting the target value as the largest Q value in the next state.
                if task == 'task3':
                    E[state][action] += 1
                    Q[state][action] += alpha * (reward + gamma * target - Q[state][action]) * E[state][action] # Updating t
                    E *= gamma * lambda_
                else:
                    Q[state][action] += alpha * (reward + gamma * target - Q[state][action]) # Updating the Q-table.
                state = next_state # Setting the next state as the current state.
                episode_reward += reward
                episode_steps += 1

            total_rewards.append(episode_reward)
            total_steps.append(episode_steps)

        final_Q.append(Q)
        tr.append(total_rewards)
        ts.append(total_steps)

    # Averaging the rewards, steps, and Q-table across the 50 runs.
    avg_rewards = np.mean(tr, axis=0)
    avg_steps = np.mean(ts, axis=0)
    avg_q = np.mean(final_Q, axis=0)

    return avg_q, avg_rewards, avg_steps
```

Appendix 2 – Training the agent using the SARSA algorithm



```python
# The definition of the algorithm's solving method.
def solve(self, env, gamma, alpha,epsilon,pol, task, n_episodes=5000, runs=50, lambda_ = 0.85):
    tr = []
    ts = []
    final_Q = []

    for i in range(runs):
        Q = np.zeros((124, 4)) # Initializing the Q-table.
        if task == 'task3':
            E = np.zeros((124, 4))  # Initializing the eligibility trace for task 3.
        total_rewards = [] # Initializing an array to store the rewards across all episodes.
        total_steps = [] # Initializing an array to store the steps across all episodes.

        for j in range(n_episodes): # Looping through all episodes
            _, state, reward, done = env.reset() # Obtaining the state, reward, and whether the episode has been termin
            episode_reward = 0
            episode_steps = 0

            if pol == 'soft-max':
                action = self.soft_max_policy(Q, state, 0.3) # Selecting an action if the 'Softmax' policy was chosen.
            elif pol == 'epsilon-greedy':
                action = self.epsilon_greedy_policy(Q, state, epsilon) # Selecting an action if the 'epsilon-greedy' po

            while not done: # While the episode hasn't terminated.
                if pol == 'soft-max':
                    _, next_state, reward, done = env.step(action) # Obtaining the next state, the corresponding reward
                    next_action = self.soft_max_policy(Q, next_state, 0.3) # Selecting the next action if the 'Softmax'
                elif pol == 'epsilon-greedy':
                    _, next_state, reward, done = env.step(action) # Obtaining the next state, the corresponding reward
                    next_action = self.epsilon_greedy_policy(Q, next_state, epsilon) # Selecting the next action if the
                target = Q[next_state][next_action] # Setting the target as the value of the next state and next action
                if task == 'task3':
                    delta = reward + gamma * target - Q[state][action]
                    E[state][action] += 1
                    Q += alpha * delta * E
                    E *= gamma * lambda_
                else:
                    Q[state][action] += alpha * (reward + gamma * target - Q[state][action]) # Updating the Q-table.
                state = next_state # Setting the next state as the current state.
                action = next_action # Setting the next action as the current action.
                episode_reward += reward
                episode_steps += 1

            total_rewards.append(episode_reward)
            total_steps.append(episode_steps)

        final_Q.append(Q)
        tr.append(total_rewards)
        ts.append(total_steps)

    # Averaging the rewards, steps, and Q-table across the 50 runs.
    avg_rewards = np.mean(tr, axis=0)
    avg_steps = np.mean(ts, axis=0)
    avg_q = np.mean(final_Q, axis=0)


    return avg_q, avg_rewards, avg_steps
```

Appendix 3 – Applying the algorithms to the environment in task 1

```
# Setting the parameters.
env1 = get_task1_gridworld() # Definition of the gridworld in task 1.
gamma = 0.99
alpha = 0.1
epsilon = 0.5
epsilon2 = 0.1
```

```
# Note: This took 108 minutes to run on my PC.

agent1 = SARSA_agent() # Initializing the SARSA algorithm.

# Training the SARSA algorithm using the 'epsilon-greedy' policy and experimenting with different exploration rates.
Q_sarsa, rewards_sarsa, steps_sarsa = agent1.solve(env1, gamma, alpha, epsilon, 'epsilon-greedy', 'task1')
Q_sarsa2, rewards_sarsa2, steps_sarsa2 = agent1.solve(env1, gamma, alpha, epsilon2, 'epsilon-greedy', 'task1')

# Training the SARSA algorithm using the 'Softmax' policy.
Q_sarsa3, rewards_sarsa3, steps_sarsa3 = agent1.solve(env1, gamma, alpha, epsilon, 'soft-max', 'task1')
```

```
# Note: This took 11 minutes to run on my PC.

agent2 = Q_agent() # Initializing the Q-learning algorithm.

# Training the Q-learning algorithm using the 'epsilon-greedy' policy and experimenting with different exploration rates.
Q_qlearning, rewards_qlearning, steps_qlearning = agent2.solve(env1, gamma, alpha, epsilon, 'epsilon-greedy', 'task1')
Q_qlearning2, rewards_qlearning2, steps_qlearning2 = agent2.solve(env1, gamma, alpha, epsilon2, 'epsilon-greedy', 'task1')

# Training the Q-learning algorithm using the 'Softmax' policy.
Q_qlearning3, rewards_qlearning3, steps_qlearning3 = agent2.solve(env1, gamma, alpha, epsilon, 'soft-max', 'task1')
```

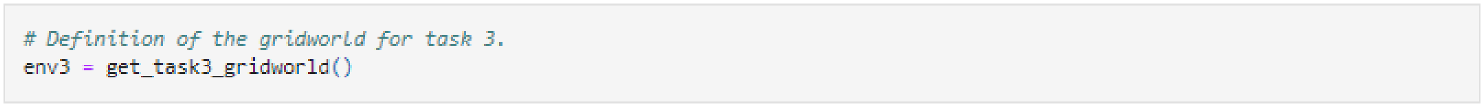
Appendix 4 – Applying the algorithms to the environment in task 2

```python
# Setting the parameters.
env2 = get_task2_gridworld() # Definition of the gridworld in task 2.
gamma1 = 0.99
gamma2 = 0.1
epsilon3 = 0.9
epsilon1 = 0.5
epsilon2 = 0.1
alpha1 = 0.1
alpha2 = 0.9
```

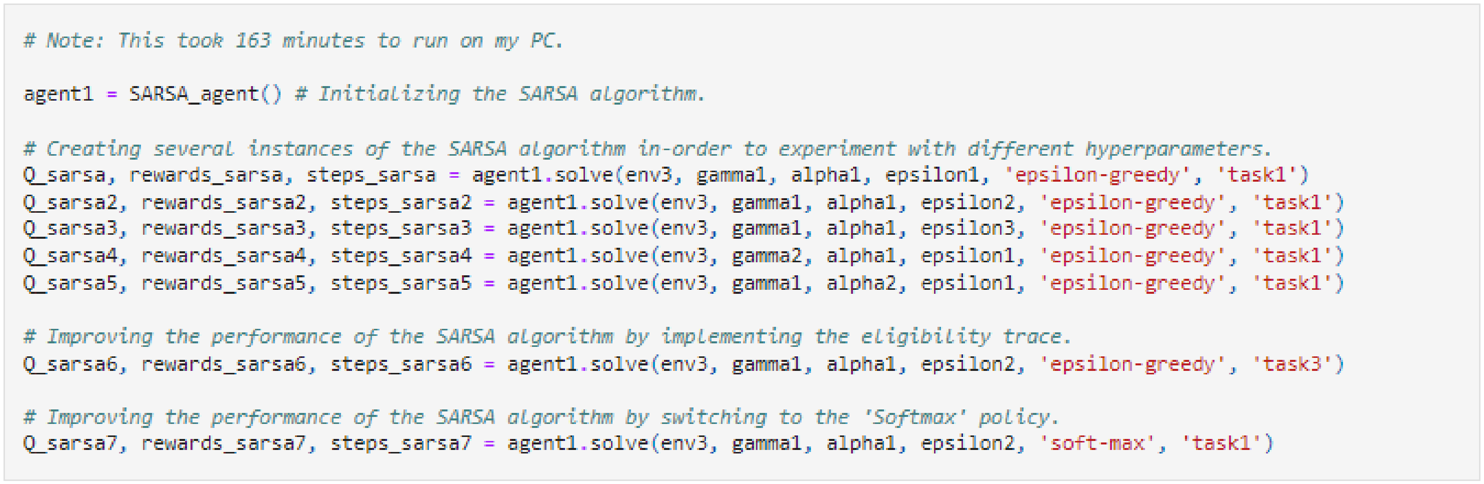
```python
# Note: This took 280 minutes to run on my PC.

agent1 = SARSA_agent() # Initializing the SARSA algorithm.

# Creating several instances of the SARSA algorithm in-order to experiment with different hyperparameters.
Q_sarsa, rewards_sarsa, steps_sarsa = agent1.solve(env2, gamma1, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_sarsa2, rewards_sarsa2, steps_sarsa2 = agent1.solve(env2, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task1')
Q_sarsa3, rewards_sarsa3, steps_sarsa3 = agent1.solve(env2, gamma1, alpha1, epsilon3, 'epsilon-greedy', 'task1')
Q_sarsa4, rewards_sarsa4, steps_sarsa4 = agent1.solve(env2, gamma2, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_sarsa5, rewards_sarsa5, steps_sarsa5 = agent1.solve(env2, gamma1, alpha2, epsilon1, 'epsilon-greedy', 'task1')

# Improving the performance of the SARSA algorithm by switching to the 'Softmax' policy.
Q_sarsa7, rewards_sarsa7, steps_sarsa7 = agent1.solve(env2, gamma1, alpha1, epsilon2, 'soft-max', 'task1')
```

```python
# Note: This took 28 minutes to run on my PC.

agent2 = Q_agent() # Initializing the Q-learning algorithm

# Creating several instances of the SARSA algorithm in-order to experiment with different hyperparameters.
Q_qlearning, rewards_qlearning, steps_qlearning = agent2.solve(env2, gamma1, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_qlearning2, rewards_qlearning2, steps_qlearning2 = agent2.solve(env2, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task1'
Q_qlearning3, rewards_qlearning3, steps_qlearning3 = agent2.solve(env2, gamma1, alpha1, epsilon3, 'epsilon-greedy', 'task1'
Q_qlearning4, rewards_qlearning4, steps_qlearning4 = agent2.solve(env2, gamma2, alpha1, epsilon1, 'epsilon-greedy', 'task1'
Q_qlearning5, rewards_qlearning5, steps_qlearning5 = agent2.solve(env2, gamma1, alpha2, epsilon1, 'epsilon-greedy', 'task1'

# Improving the performance of the Q-learning algorithm by switching to the 'Softmax' policy.
Q_qlearning7, rewards_qlearning7, steps_qlearning7 = agent2.solve(env2, gamma1, alpha1, epsilon2, 'soft-max', 'task1')
```

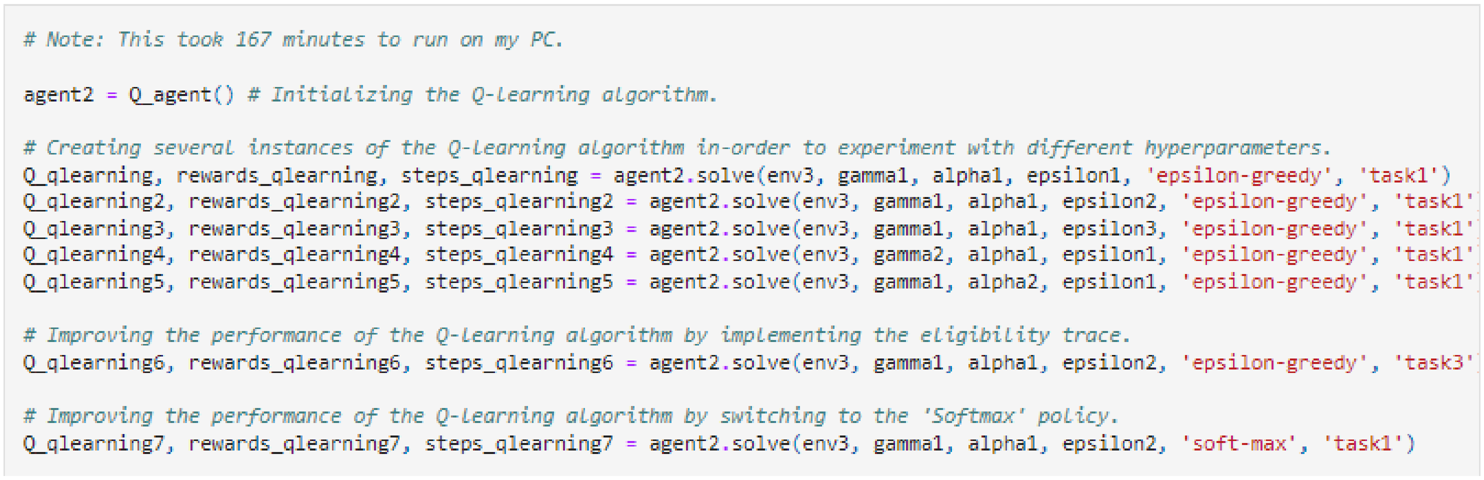
Appendix 5 – Applying the algorithms to the environment in task 3



```
# Definition of the gridworld for task 3.
env3 = get_task3_gridworld()
```



```
# Note: This took 163 minutes to run on my PC.

agent1 = SARSA_agent() # Initializing the SARSA algorithm.

# Creating several instances of the SARSA algorithm in-order to experiment with different hyperparameters.
Q_sarsa, rewards_sarsa, steps_sarsa = agent1.solve(env3, gamma1, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_sarsa2, rewards_sarsa2, steps_sarsa2 = agent1.solve(env3, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task1')
Q_sarsa3, rewards_sarsa3, steps_sarsa3 = agent1.solve(env3, gamma1, alpha1, epsilon3, 'epsilon-greedy', 'task1')
Q_sarsa4, rewards_sarsa4, steps_sarsa4 = agent1.solve(env3, gamma2, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_sarsa5, rewards_sarsa5, steps_sarsa5 = agent1.solve(env3, gamma1, alpha2, epsilon1, 'epsilon-greedy', 'task1')

# Improving the performance of the SARSA algorithm by implementing the eligibility trace.
Q_sarsa6, rewards_sarsa6, steps_sarsa6 = agent1.solve(env3, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task3')

# Improving the performance of the SARSA algorithm by switching to the 'Softmax' policy.
Q_sarsa7, rewards_sarsa7, steps_sarsa7 = agent1.solve(env3, gamma1, alpha1, epsilon2, 'soft-max', 'task1')
```



```
# Note: This took 167 minutes to run on my PC.

agent2 = Q_agent() # Initializing the Q-learning algorithm.

# Creating several instances of the Q-learning algorithm in-order to experiment with different hyperparameters.
Q_qlearning, rewards_qlearning, steps_qlearning = agent2.solve(env3, gamma1, alpha1, epsilon1, 'epsilon-greedy', 'task1')
Q_qlearning2, rewards_qlearning2, steps_qlearning2 = agent2.solve(env3, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task1'
Q_qlearning3, rewards_qlearning3, steps_qlearning3 = agent2.solve(env3, gamma1, alpha1, epsilon3, 'epsilon-greedy', 'task1'
Q_qlearning4, rewards_qlearning4, steps_qlearning4 = agent2.solve(env3, gamma2, alpha1, epsilon1, 'epsilon-greedy', 'task1'
Q_qlearning5, rewards_qlearning5, steps_qlearning5 = agent2.solve(env3, gamma1, alpha2, epsilon1, 'epsilon-greedy', 'task1'

# Improving the performance of the Q-learning algorithm by implementing the eligibility trace.
Q_qlearning6, rewards_qlearning6, steps_qlearning6 = agent2.solve(env3, gamma1, alpha1, epsilon2, 'epsilon-greedy', 'task3'

# Improving the performance of the Q-learning algorithm by switching to the 'Softmax' policy.
Q_qlearning7, rewards_qlearning7, steps_qlearning7 = agent2.solve(env3, gamma1, alpha1, epsilon2, 'soft-max', 'task1')
```