Reinforcement Learning

Part I: Define an RL Environment

Theme: Treasure Island Grid World with map and key as positive rewards and snakes and volcano as negative rewards.

States: {S1 = (0,0), S2 = (0,1), S3 = (0,2), S4 = (0,3), S5 = (1,0), S6 = (1,1), S7 = (1,2), S8 = (1,3), S9 = (2,0), S10 = (2,1), S11 = (2,2), S12 = (2,3), S13 = (3,0), S14 = (3,1), S15 = (3,2), S16 = (3,3)}

Actions: {Up, Down, Right, Left}

Rewards: {-5, -6, +5, +10}

Reaching goal rewards 30

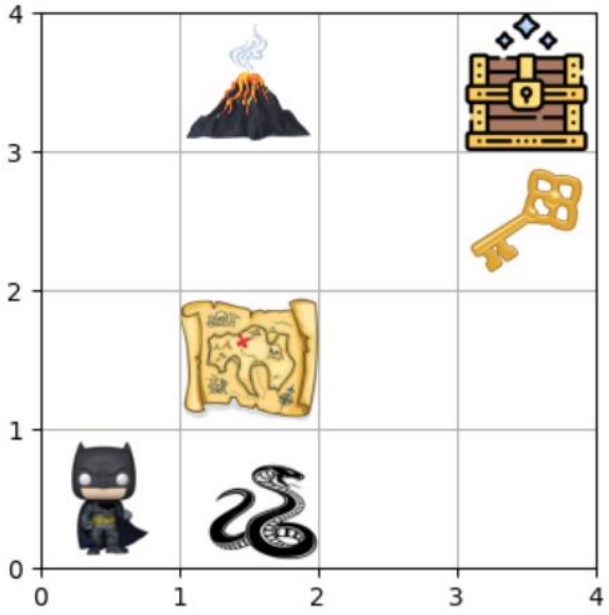
Objective: Reach the goal state with maximum reward

Brief Description

The environment that we chose for this part of the assignment is a Treasure Hunt. Batman is our agent, and his goal is to reach the treasure (Goal), which gives him a reward of +30. There are two rewards for him on the way (a map and a key), which give him a positive reward +5 and +10 respectively. He needs to avoid obstacles while trying to reach the treasure on the way, which is a snake and a volcano, which give him negative rewards -5 and -6 respectively.

Visualization of the environment

Below is a representation of our environment. Batman (agent) is at [0,0], the treasure (goal) is at [3,3]. The positive rewards (map and key) are at [1,1] and [3,2] respectively. The negative rewards (snake and volcano) are at [2,0] and [1,3] respectively.



Safety in AI

The Batman (Agent) is allowed to move only through observation space (i.e., the grid world). Below is how we ensure it -

Action space constraints: We constrain the agent's action space to only allow actions that keep it within the defined space. When the agent is navigating the observation space, his actions are limited to moving up, down, right, or left. We did this by providing the list of actions as [0,1,2,3], which confines the agent to make a choice only from this list of available actions instead of a discrete space.

State space representation: We have defined the state space in a way that encourages the agent to stay within the defined space. The environment is a 4X4 space, which means there are 16 possible states the agent can be. It means, the agent must always be at a state between [0,0] and [3,3]. For this, we have used the clip statement to ensure that the agent navigates only within the defined observation space.

Reward shaping: We have used 2 positive rewards (map and key), and 2 negative rewards (snake and volcano) to make sure the agent is inclined to take the path with higher reward.

Part II and Part III: Solving the environment using SARSA and Q-Learning Algorithm

SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm that learns the optimal value function for a given policy. The key idea behind SARSA is to learn the Q-values for each state-action pair and use them to guide the agent's behavior towards the optimal policy.

Update Function

In this algorithm, the agent follows an epsilon-greedy policy, where it selects the action with the highest Q-value with probability (1-e), and a random action with probability (e).

The update function for SARSA is:

Q(s,a) <- Q(s,a) + alpha * (reward + gamma * Q(s',a') - Q(s,a))

where Q(s,a) is the Q-value for taking action **a** in state **s**, **alpha** is the learning rate, **reward** is the reward received for taking that action, **gamma** is the discount factor, **s'** is the next state, and **a'** is the action taken in the next state according to the current policy.

Features

1. SARSA learns the value function for the policy it follows, so it is an on-policy algorithm.
2. SARSA updates its Q-values based on the difference between the predicted Q-value and the actual Q-value.
3. SARSA explores the environment by selecting a random action with probability epsilon.

Advantages

1. SARSA algorithm is simple to implement.
2. It can handle small to moderate sized state and action spaces.
3. It converges to optimal policy under some conditions

Disadvantages

1. SARSA algorithm can take very long to converge in complex environments

2. It might not find the optimal policy if exploration rate is very low

Q-Learning Algorithm

Q-learning is an off-policy reinforcement learning algorithm that tries to find the best action to take for the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions. More specifically, q-learning seeks to learn a policy that maximizes the total reward. When q-learning is performed we create a q-table with same shape of [state, action] and we initialize our values to zero. We then update and store our q-values after each episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

Update Function

The update function is based on the Bellman equation, which incorporates the immediate reward and the maximum Q-value of the next state. The Q-value update equation in Q-learning is as follows:

Q(s,a) <- Q(s,a) + alpha * (reward + gamma max(a) * Q(s', a') - Q(s,a))

Q(s, a) represents the Q-value of taking action **a** in state **s**. **alpha** is the learning rate, determining the impact of the new information compared to the existing Q-value. **reward** is the immediate reward received after acting **a** in state **s**. **gamma** is the discount factor, determining the importance of future rewards. **max(Q(s', a'))** is the maximum Q-value among all possible actions **a'** in the next state **s'**.

Features

1. Q-learning is a model-free algorithm, meaning it does not require explicit knowledge or assumptions about the underlying dynamics of the environment. It learns directly from interactions with the environment, making it applicable in situations where the dynamics are unknown or complex.
2. Q-learning is a value-based method that focuses on estimating the action-value function (Q-function) to determine the best action to take in each state.
3. Q-learning employs an exploration-exploitation trade-off by balancing the exploration of unknown state-action pairs and the exploitation of learned information.
4. Under certain conditions, Q-learning is proven to converge to the optimal Q-values and policy

Advantages

1. Q-learning does not require prior knowledge or assumptions about the underlying dynamics of the environment. It learns directly from interactions with the environment, making it applicable in situations where the model is unknown or complex.
2. Q-learning is a general-purpose algorithm that can be applied to a wide range of problems without specific modifications.
3. Q-learning is relatively simple to understand and implement, especially for problems with small state and action spaces. The Q-table serves as a lookup table for storing and updating Q-values, making it intuitive and easy to follow.
4. Q-learning is robust to noisy rewards or delayed rewards, as it estimates the long-term value of actions based on the cumulative rewards over time.

Disadvantages

1. Q-learning faces challenges when dealing with high-dimensional or continuous state and action spaces. As the number of states and actions increases, the size of the Q-table grows exponentially, making it computationally expensive or even infeasible to store and update all Q-values.
2. Q-learning relies on the exploration-exploitation trade-off to balance between trying new actions and exploiting the learned information. While epsilon-greedy exploration is commonly used, it can struggle to effectively explore large state spaces or find optimal policies.

3. The performance of Q-learning can be sensitive to the initial values in the Q-table.

Solving Part I using SARSA

Hyperparameters

alpha = 0.15

gamma = 0.90

epsilon = 1

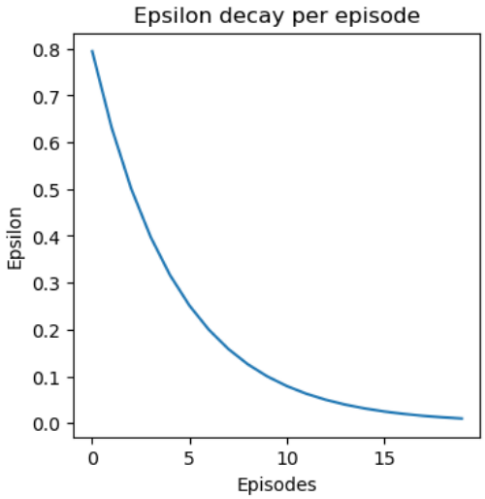
num_episodes = 20

total_steps = 20

epsilon_decay = (0.01/1) ** (1/num_episodes)

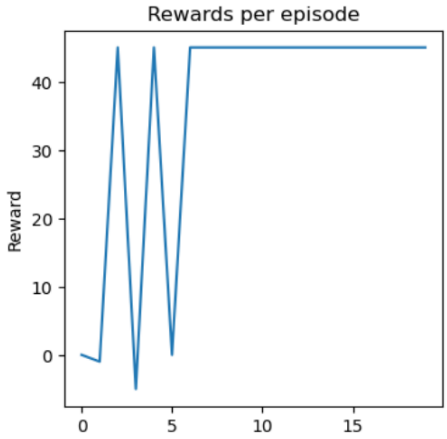
Visualizations

1. Epsilon decay over episodes



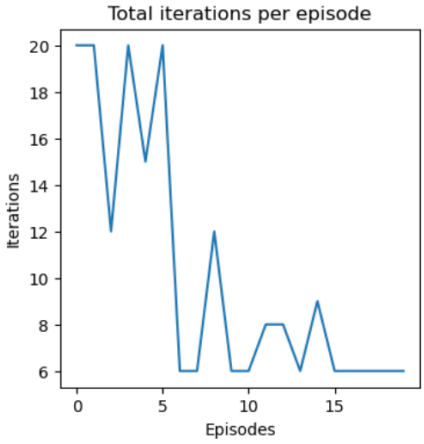
We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode



We see that out of 20 total episodes, the agent has learned to accumulate more positive rewards over time. We observe that, after 6 episodes, the agent has been successful in collecting the highest reward in each episode (i.e., the goal).

3. Total iterations per episode
   From the below graph, we observe that the agent has learned to get to the treasure (goal) very quickly as the number of episodes pass. We see that, after episode 5, the agent has taken only 6-12 steps to reach the goal compared to 16-20 at the beginning of the training.



Hyperparameter Tuning

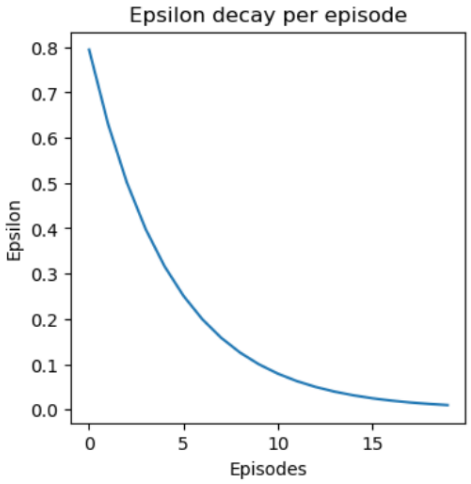
Tuning the gamma value

In this part, we only change the value of gamma while keeping every other hyperparameter same as the initial model.

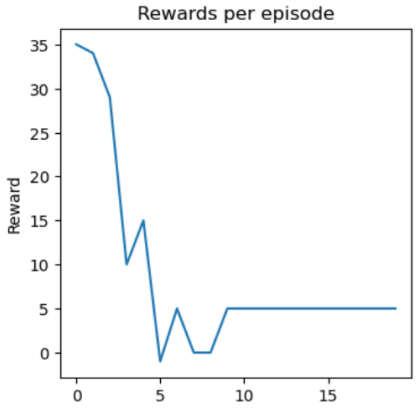
Model 1: gamma = 0.92

Visualizations

1. Epsilon decay over episodes

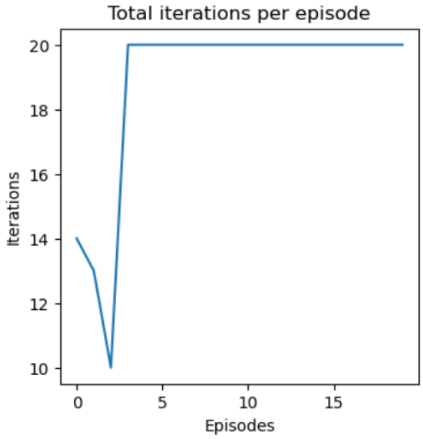


   We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01
2. Total reward per episode



Though the agent managed to get a very high reward in the beginning, the number of rewards per episode gradually reduced. We see that, after the 5th episode, the total reward per episode is 5 throughout till the end of 20 episodes.

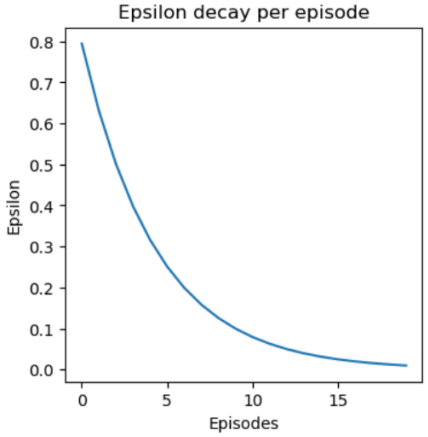
3. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) as the number of episodes pass. We see that, after episode 3, the agent has utilized all the 20 steps. In conjunction with the above "total reward per episode" graph, we can see that, even after 20 steps, the agent has not managed to reach the goal.



Model 2: gamma = 0.95

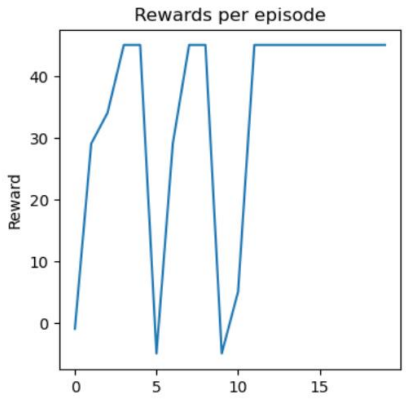
Visualizations

1. Epsilon decay over episodes

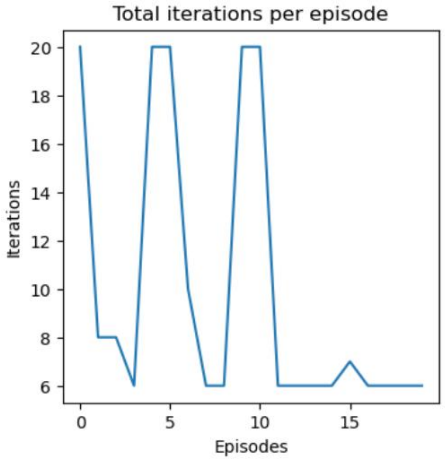


We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode
   We see that through each episode, the agent has learned to gain rewards. If we observe the plot, the agent has gained maximum reward 4 out of first 10 episodes, and managed to reach the goal every time after the 10^{th} episode.



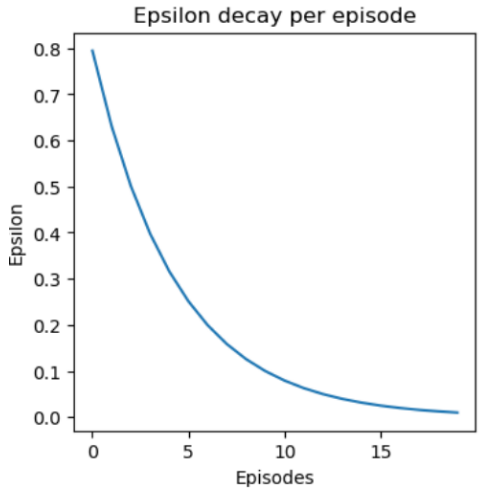
3. Total iterations per episode
   From the below graph, we observe that the agent has learned to get to the treasure (goal) as the number of episodes pass. We see that the agent took very little steps to reach the goal once every 3 episodes. In conjunction with the above "total reward per episode" graph, we can see that, the agent has managed to reach the goal every time after 10 episodes, within just 6 steps.



Model 3: gamma = 0.98

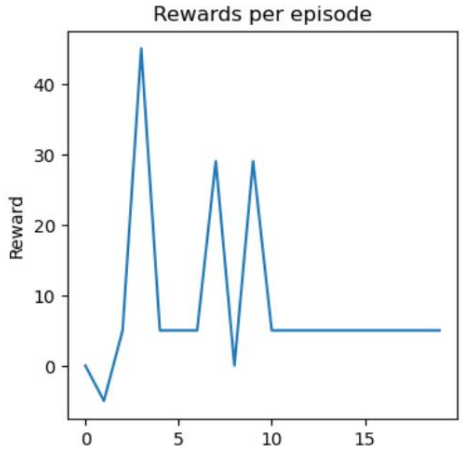
Visualizations

1. Epsilon decay over episodes

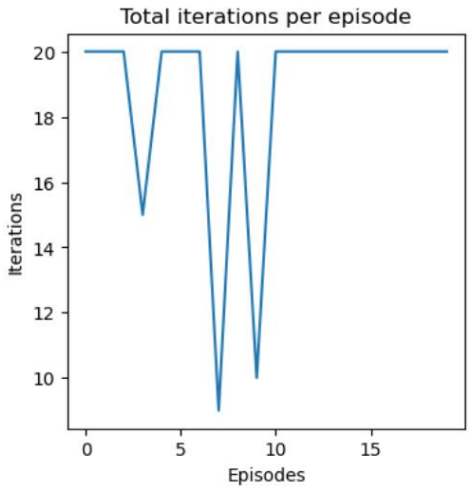


We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode
   We see that through each episode, the agent has learned to gain rewards. If we observe the plot, the agent has gained atleast 5 reward points throughout 20 episodes, managing to reach the goal only once.



3. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the agent took the maximum steps in most of the episodes. In conjunction with the above "total reward per episode" graph, we can see that, the agent has not managed to reach the goal most of the times.



Comparing Model 1, Model 2, and Model 3 – we see that Model 2 has performed better than the other two models in terms of reward collection per episode and number of iterations taken per episode.

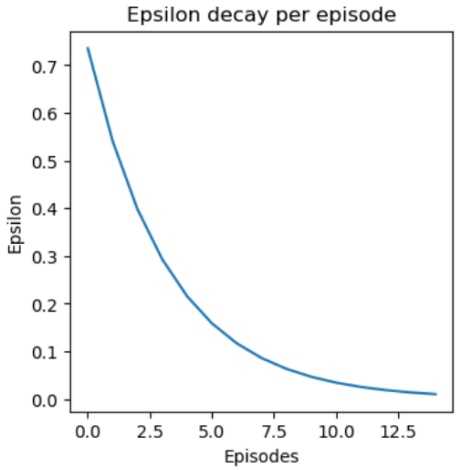
Tuning the number of episodes

In this part, we only change the number of episodes while keeping every other hyperparameter same as the initial model.

Model 4: Total episodes = 15

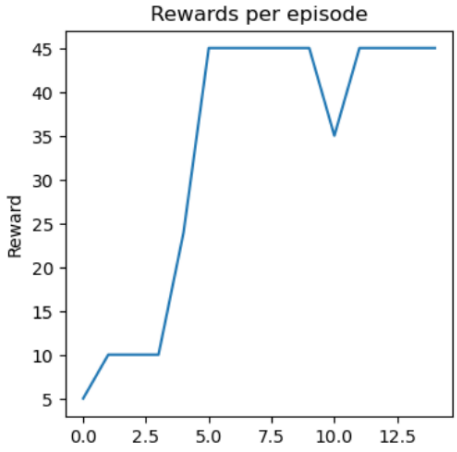
Visualizations

1. Epsilon decay over episodes

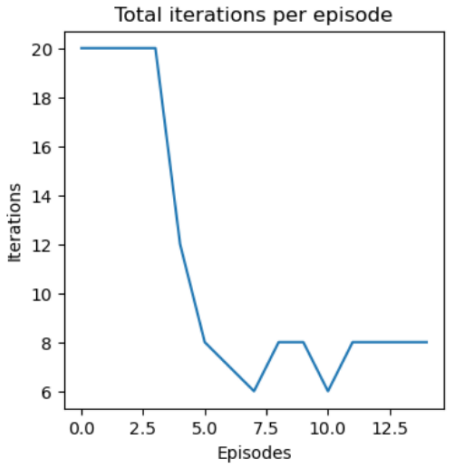


   We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 2 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode
   We see that through each episode, the agent has learned to gain rewards gradually. The agent has managed to reach the goal and collect the maximum reward most of the time after 2nd episode.



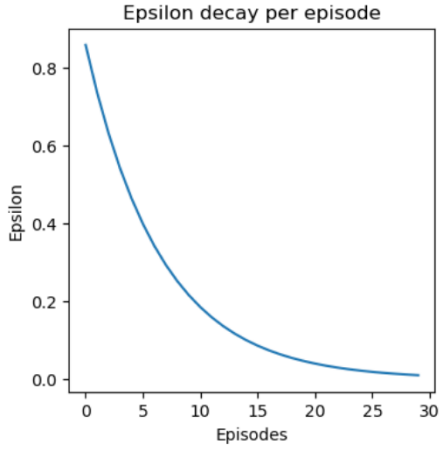
3. Total iterations per episode
   From the below graph, we observe that the agent has learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the agent took the minimum steps in most of the episodes. In conjunction with the above "total reward per episode" graph, we can see that, the agent has managed to reach the goal most of the times in just 6-8 steps.



Model 5: Total episodes = 30

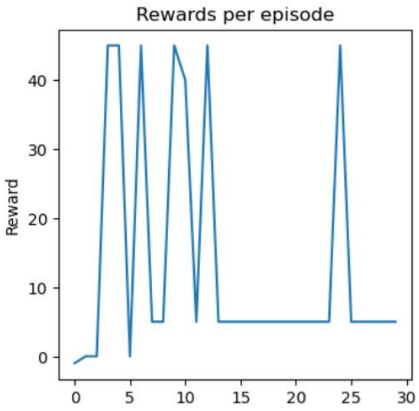
Visualizations

1. Epsilon decay over episodes

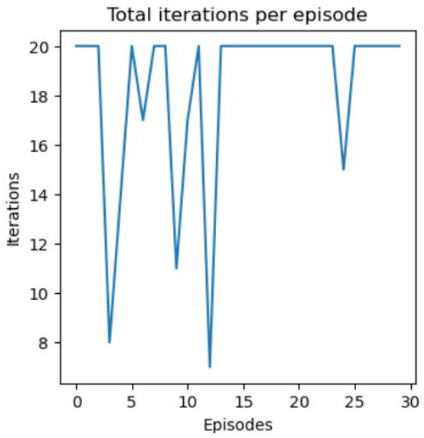


We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 2 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode
   We see that through each episode, the agent has learned to gain rewards gradually. The agent has managed to reach the goal and collect the maximum reward most of the time after 2nd episode. Though the agent hasn't claimed maximum reward continuously as episodes progress, we see spike in rewards once every 2 episodes.



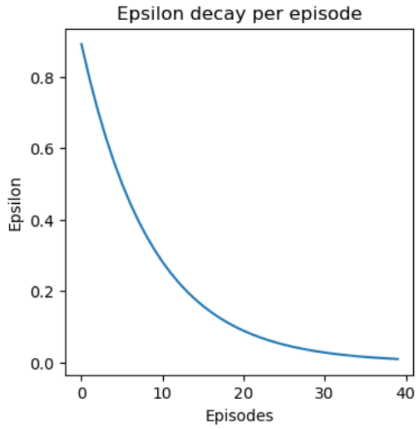
3. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the agent took the maximum steps in most of the episodes. In conjunction with the above "total reward per episode" graph, we can see that, the agent has not managed to reach the goal most of the time.



Model 6: Total episodes = 40

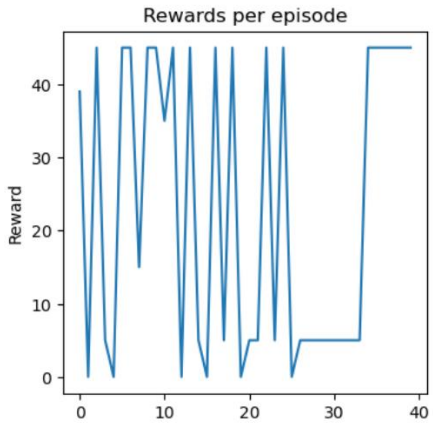
Visualizations

1. Epsilon decay over episodes

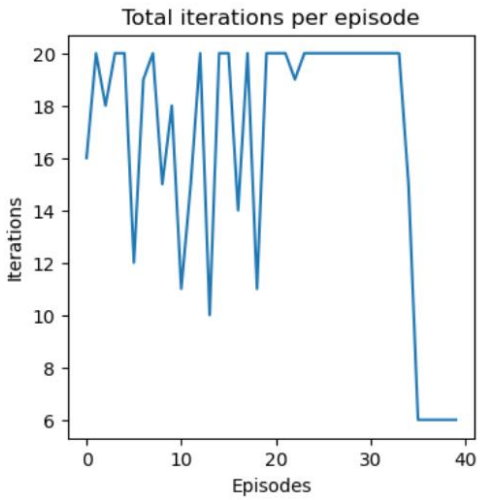


We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 5 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode
   We see that through each episode, the agent has learned to gain rewards gradually. We can't identify a specific pattern in the rewards claimed per episode. However, we see that the agent has managed to get the maximum reward in the last 5 episodes.



3. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) effectively as the number of episodes pass. We see multiple dips in the number of steps taken per episode, but most of the time, it appears the agent took the maximum steps. Only after 35th episode, the agent seems to have used minimum steps, which is in agreement with the above plot as well.



Comparing Model 4, Model 5, and Model 6 – we see that Model 4 has performed better than the other two models in terms of reward collection per episode and number of iterations taken per episode.

Solving Part I using Q-Learning

Hyperparameters

alpha = 0.15

gamma = 0.90

epsilon = 1

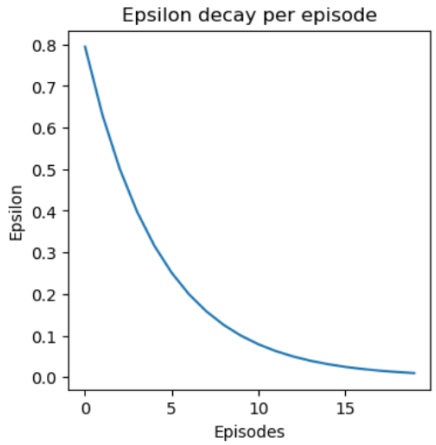
num_episodes = 20

total_steps = 20

epsilon_decay = (0.01/1) ** (1/num_episodes)

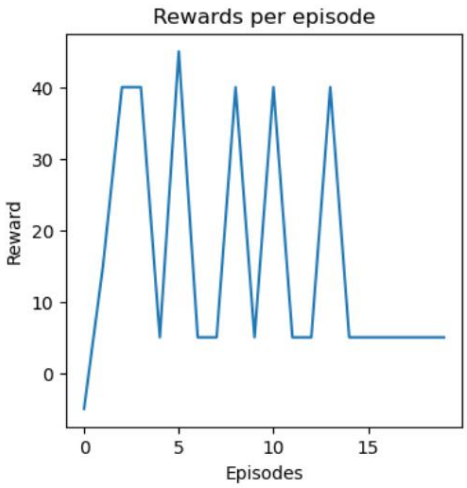
Visualizations

1. Epsilon decay over episodes



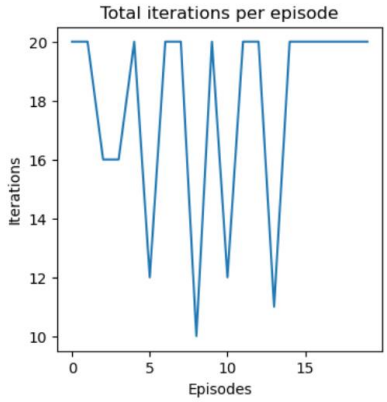
   We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

2. Total reward per episode



We see that out of 20 total episodes, the agent has learned to accumulate more positive rewards over time. We observe that, there are many spikes in the plot indicating that the agent was successful in capturing maximum reward multiple times during the episodes.

3. Total iterations per episode
   From the below graph, we observe that the agent has utilized a smaller number of iterations before termination atleast 5 times out to 20 episodes. The episodes where the agent took less than maximum iterations indicate that the agent has achieved the goal.



Hyperparameter Tuning

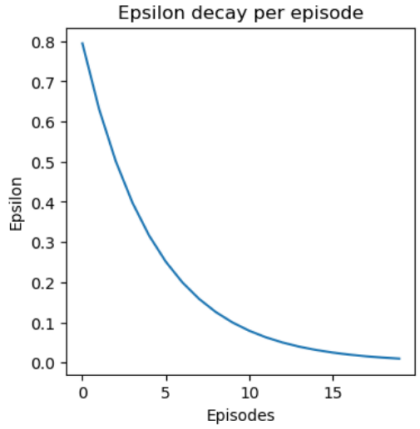
Tuning the gamma value

In this part, we only change the value of gamma while keeping every other hyperparameter the same as the initial model.

Model 1: gamma = 0.92

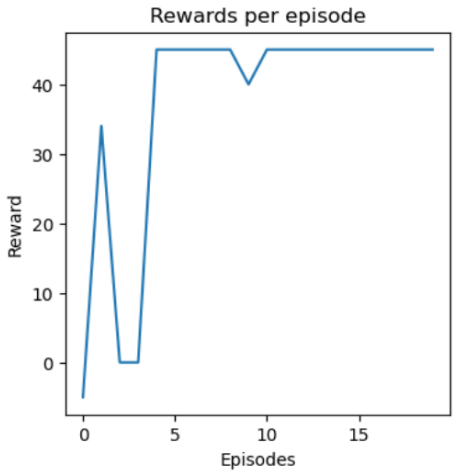
Visualizations

1. Epsilon decay over episodes



We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

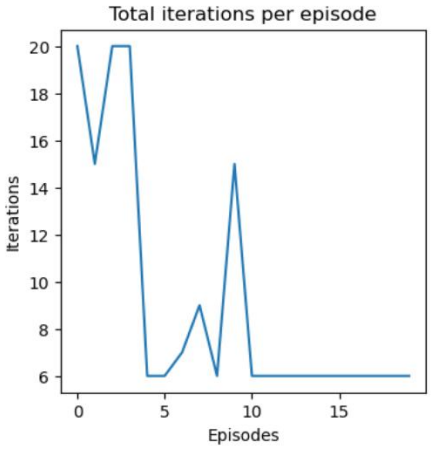
2. Total reward per episode



We see that the agent has achieved maximum reward each time after the 4th episode, except for once. This indicates that the agent has learned well.

3. Total iterations per episode

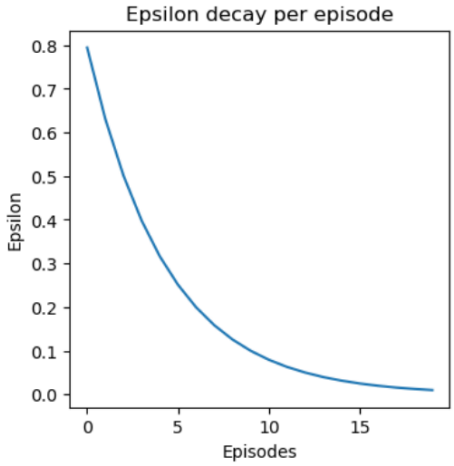
From the below graph, we observe that, though the agent took maximum iterations at the beginning, he has completely learned to achieve the maximum reward after the 10th episode.



Model 2: gamma = 0.95

Visualizations

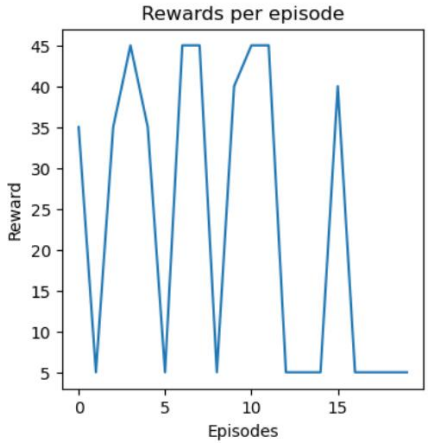
4. Epsilon decay over episodes



We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

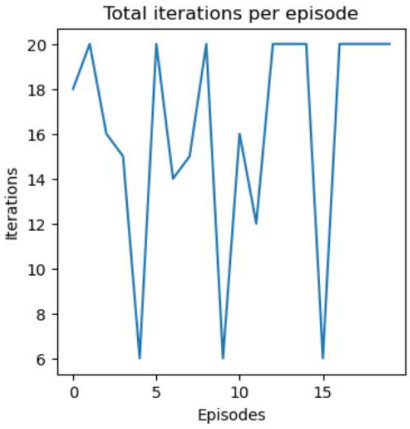
5. Total reward per episode

We see that through each episode, the agent was able to get maximum reward 6 times out of 20 episodes. There is a lot of fluctuation in the reward collected per episode, which indicates that the agent is not learning very well. However, 6 times, the agent has managed to achieve maximum reward and terminate the episode.



6. Total iterations per episode

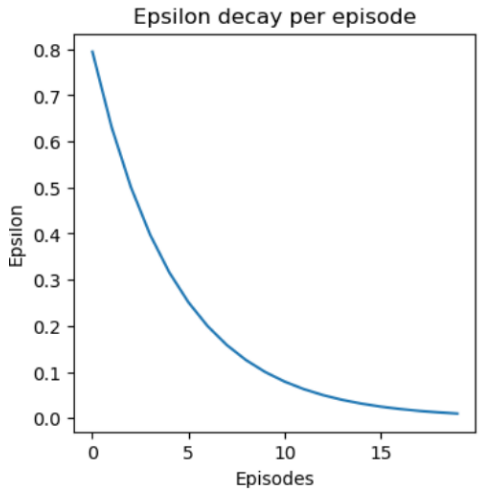
From the below graph, we observe that the agent has taken less than maximum iterations per episode most of the time, except for 2-3 episodes. This is a good indicator that the agent is trying to maximize the rewards gained in each episode and terminate the episode faster.



Model 3: gamma = 0.98

Visualizations

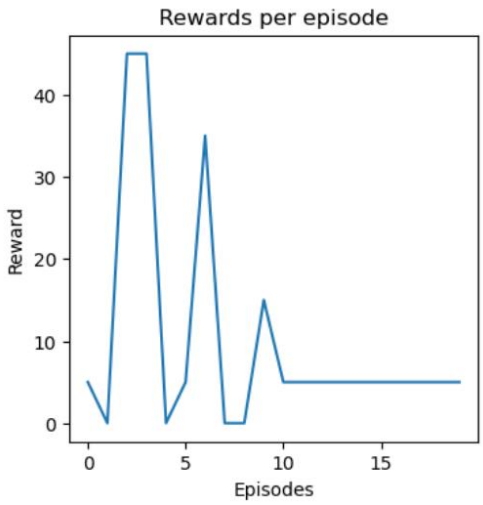
4. Epsilon decay over episodes



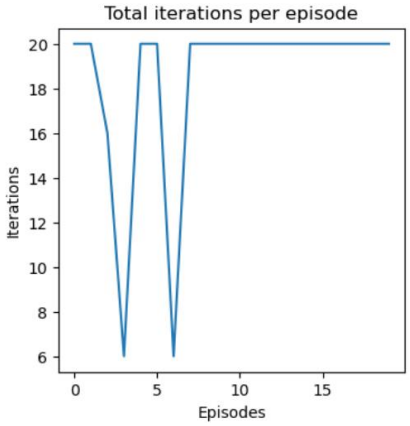
We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 20 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

5. Total reward per episode

We see that through each episode, the agent has learned to gain rewards. If we observe the plot, the agent has gained atleast 5 reward points throughout 20 episodes, managing to reach the goal only twice.



6. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the agent took the maximum steps in most of the episodes. In conjunction with the above "total reward per episode" graph, we can see that the agent has not managed to reach the goal most of the times.



Comparing Model 1, Model 2, and Model 3 – we see that Model 1 has performed better than the other two models in terms of reward collection per episode and number of iterations taken per episode.

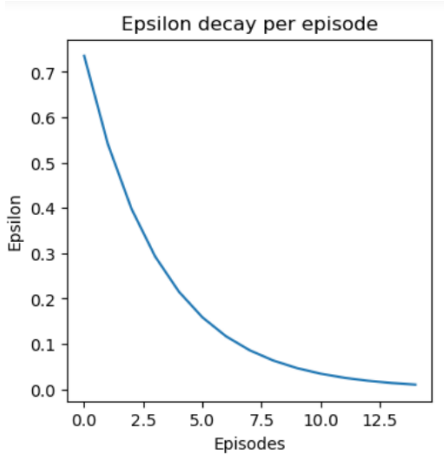
Tuning the number of episodes

In this part, we only change the number of episodes while keeping every other hyperparameter same as the initial model.

Model 4: Total episodes = 15

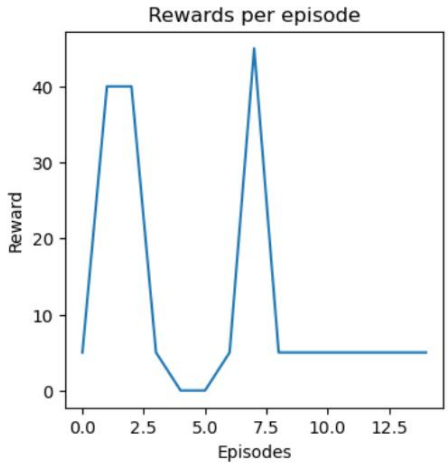
Visualizations

4. Epsilon decay over episodes

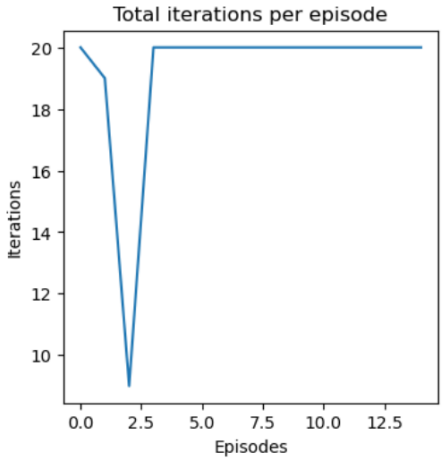


We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 2 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

5. Total reward per episode
   We see that through each episode, the agent has not learned to gain rewards. The agent has managed to reach the goal and collect the maximum reward only once.



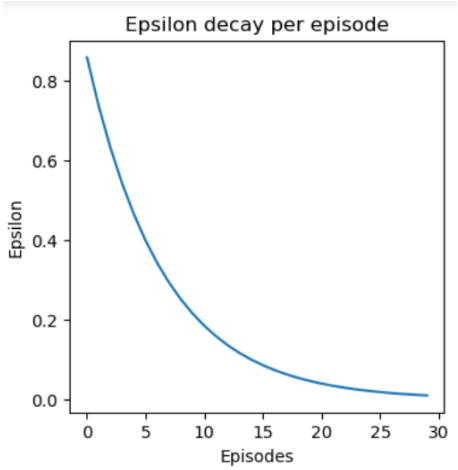
6. Total iterations per episode
   From the below graph, we observe that the agent has not learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the agent took the maximum steps in most of the episodes. In conjunction with the above "total reward per episode" graph, we can see that the agent has not managed to reach the goal most of the times.



Model 5: Total episodes = 30

Visualizations

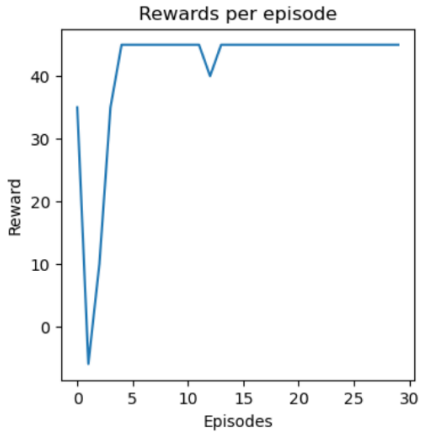
4. Epsilon decay over episodes



We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 2 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

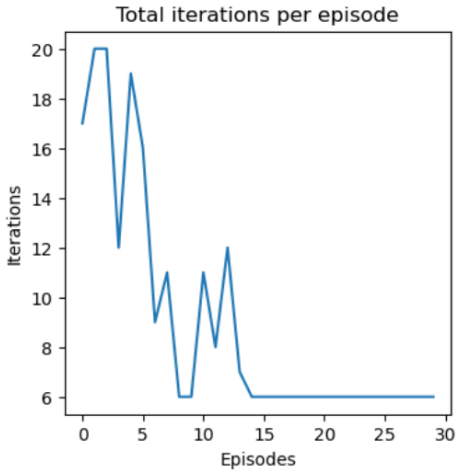
5. Total reward per episode

We see that through each episode, the agent has learned to gain rewards gradually. The agent has managed to reach the goal and collect the maximum reward each time after episode 5, except at episode 12.



6. Total iterations per episode

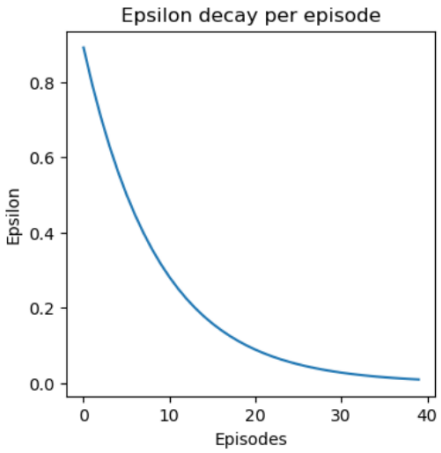
   From the below graph, we observe that the agent has learned to get to the treasure (goal) effectively as the number of episodes pass. We see that the steps taken by the agent as each episode passes are gradually reduced. After episode 15, the agent was able to achieve maximum reward with only 6 steps.



Model 6: Total episodes = 40

Visualizations

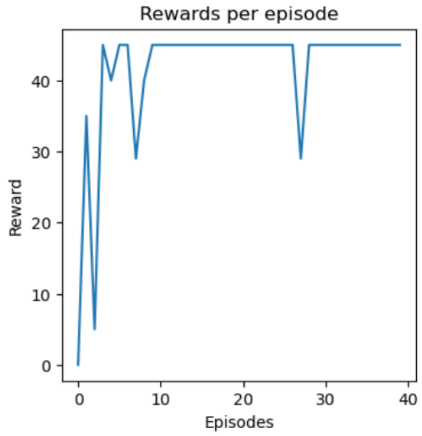
1. Epsilon decay over episodes



We start with epsilon = 1. Based on the number of episodes, the decay factor is calculated and applied to epsilon after each episode. After 5 episodes of 20 steps each, we see that the epsilon value has gradually reduced to 0.01

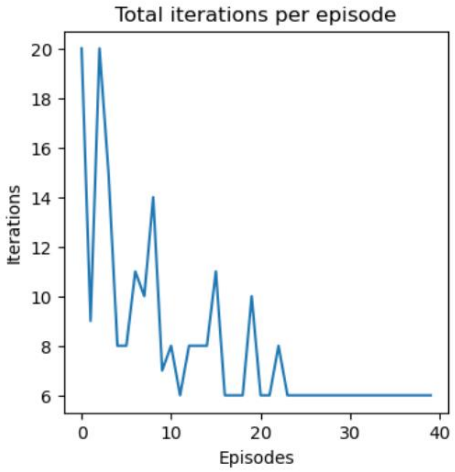
2. Total reward per episode

We see that through each episode, the agent has learned to gain rewards gradually. We see that most times after episode 10, the agent has managed to achieve maximum reward, indicating that the agent has learned well.



3. Total iterations per episode

From the below graph, we observe that the agent has learned to get to the treasure (goal) effectively as the number of episodes pass. After the first couple of episodes, the agent has never utilized his maximum iterations before terminating an episode. This indicates that the agent has learned well.



Comparing Model 4, Model 5, and Model 6 – we see that Model 5 has performed better than the other two models in terms of reward collection per episode and number of iterations taken per episode.

Comparing SARSA vs Q-Learning

Let us now compare the performance of SARSA vs Q-Learning algorithm for the same set of hyperparameters. The hyperparameters we choose are:

alpha = 0.15

gamma = 0.92

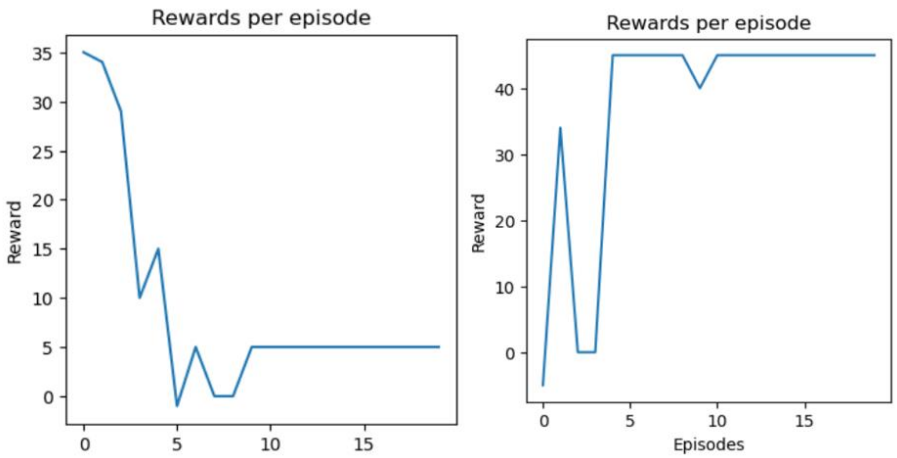
epsilon = 1

num_episodes = 20

total_steps = 20

epsilon_decay = (0.01/1) ** (1/num_episodes)

For comparison, we will use the total rewards per episode plots for two algorithms.



From the above plot, it is evident that for the given hyperparameters, Q-learning algorithm performed much better. We see that using SARSA algorithm, the agent couldn't learn well and hence the rewards per episode dropped significantly after 5th episode. In contrast, after 5th episode, the agent has learned to achieve maximum reward each time using Q-learning algorithm.

References

1. https://towardsdatascience.com/reinforcement-learning-implement-grid-world-from-scratch-c5963765ebff
2. https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/3-21_value_iteration.pdf
3. https://medium.com/reinforcement-learning-a-step-by-step-implementati/reinforcement-learning-a-step-by-step-implementation-using-sarsa-1cfd3e64775a
4. https://analyticsindiamag.com/a-complete-intution-on-sarsa-algorithm/
5. https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
6. https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html