PART I: DEFINING RL ENVIRONMENT

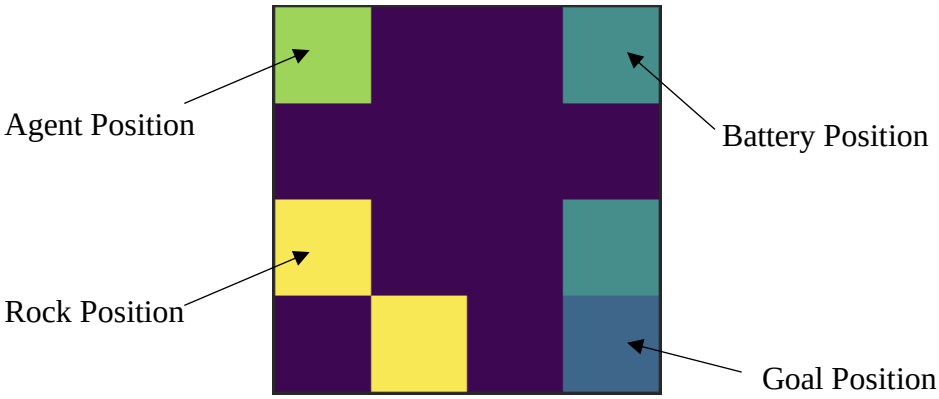
Environment Description

The environment represents a Lawnmower Grid World. It can be thought of as a two-dimensional space that has been broken up into 16 possible positions (or states) in the form of grid squares. The agent that interacts with the environment is a robot/person who is mowing the lawn with a lawnmower. The agent needs to get from the bottom right grid square from the top right grid square. In certain grid squares, there are items. The two kinds of items in the environment are batteries and rocks. The batteries can be thought of as positive rewards that are required by the lawnmower to run, and rocks can be thought of as negative rewards which damage the lawnmower. The objective of the agent is to reach the bottom-right grid square by accumulating the most rewards (batteries).

A more formal definition of the environment is given below:

- **Theme**: Lawnmower Grid World with batteries as positive rewards and rocks 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:** {-6, -6, +6, +6, +10} (Battery: +6, Rock: -6, Goal State: +10)
- **Objective:** Reach the goal state with maximum reward.

Environment Visualization



The image shown above is a visual representation of the Lawnmower Grid World. As described, we can see the environment as a representation of a 4x4 grid. Each grid square has a color representing the objects present on the grid square. The description of these colors is given below:

1) **Green:** Represents grid square with agent's current position.
2) **Yellow:** Represents grid square with rock on it.
3) **Teal:** Represents grid square with battery on it.
4) **Blue:** Represents goal position that agent needs to get to.
5) **Deep Purple:** Represents generic grid square with no items present.

Safety in AI

To ensure safety of AI in the context of our environment, we have put up some features in place. The first way of ensuring security is by making sure that the agent does not go outside of the environment. To ensure this, we have forced the agent to bounce back into the environment in case it tries to go outside. To make sure that the agent is only choosing from the possible actions, we have set a fixed action space that is not modified. This means that whenever an agent is attempting to make a move, it will be confined to our fixed action space. Another step we have taken to ensure AI safety is by balancing the reward system. We have made sure to set a reward system such that the agent is rewarded for reaching the goal state. This prevents the agent from deviating from the goal and ensures safety in AI through restricting agent.

PART II: SOLVING THE ENVIRONMENT USING SARSA

SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is a reinforcement learning algorithm that can be used to help agents solve Markov Decision Processes (MDP). It is known as a tabular algorithm since it works by referencing a table of values, called Q-values. The algorithm works by updating Q-values (expected discount future reward) and referencing them to take an action in a certain state of an MDP. The Q-values are updated for each state-action pair using the following formula:

$$Q(s,a) = Q(s,a) + \alpha * (r + \gamma * Q(s',a') - Q(s,a))$$

where:

1. Q(s,a) is the Q-value for the current state-action pair (s,a)
2. α is the learning rate
3. r is the immediate reward
4. γ is the discount factor
5. s' and a' are the next state and action taken after the current state-action pair (s,a)

Key Features:

- It is a form of temporal difference learning.
- It is an on-policy algorithm.

Advantages:

- Since it learns the value function for the current policy that is being used, it is more stable during the exploration process as it allows the agent to learn from the experience without deviating greatly from current policy.
- The algorithm guarantees convergence, assuming that the learning rate is sufficiently small, and the agent has a chance to explore the state-action pairs infinitely often.
- It is a highly flexible algorithm that can be used extensively in a multitude of episode-based tasks, continuing tasks, and function approximating methods.

Disadvantages:

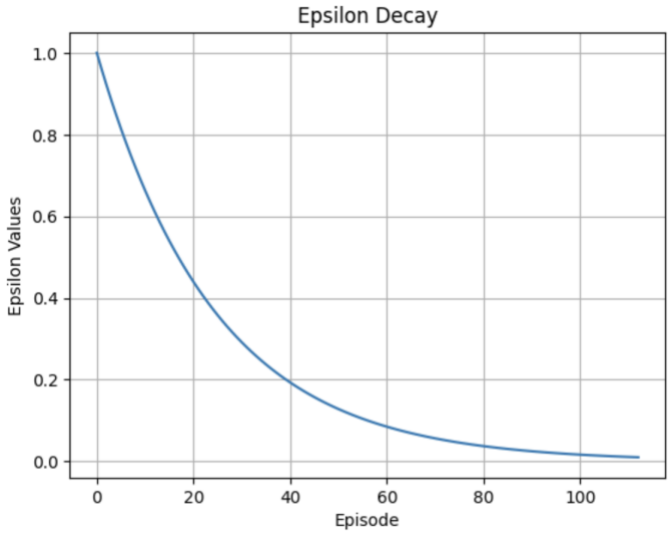
- The convergence, although guaranteed, is very slow as compared to on-policy algorithms like Q-Learning.

- If the exploration-exploitation of the algorithm is not correctly balanced, then exploration might be risky, and the agent may get stuck in a sub-optimal policy.
- SARSA may result in overfitting or underfitting to Q-values, leading to sub-optimal performance. This is especially the case in function approximation use cases.

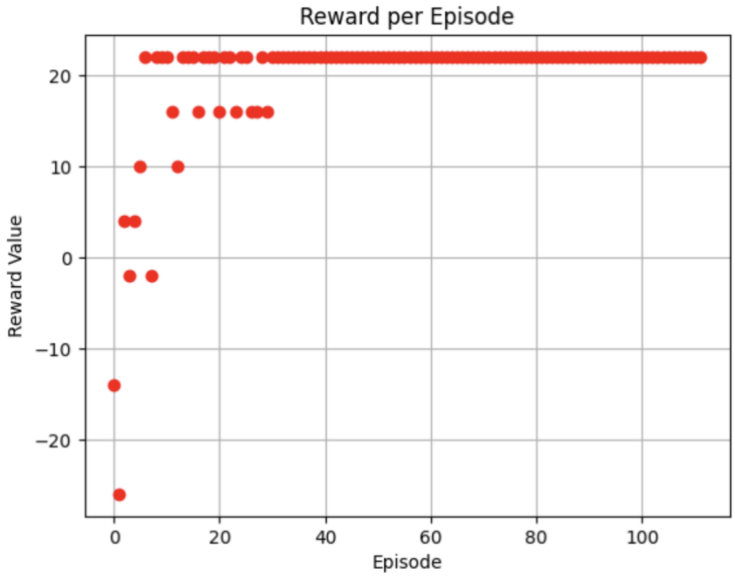
Solving Environment

After using the algorithm to solve the environment, we can gauge the performance of the algorithm using the following plots:

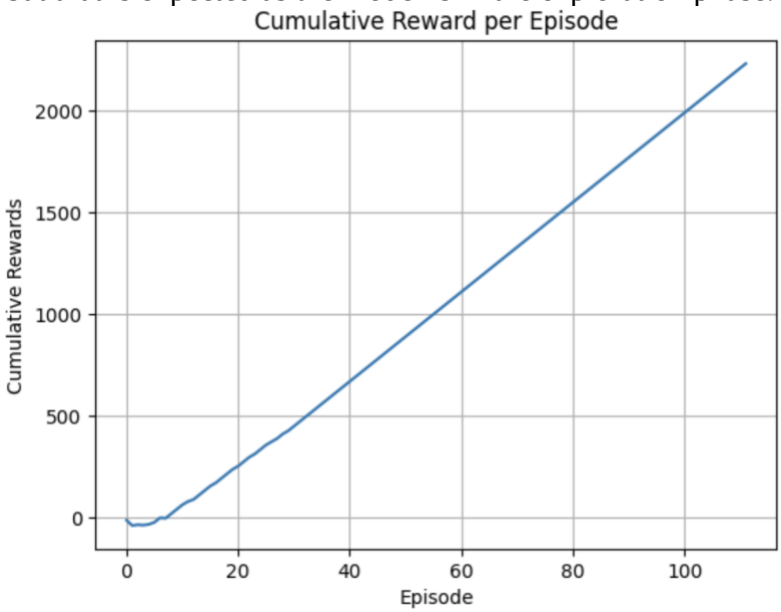
1) Epsilon-Decay Plot:
   Initially we want to have a high rate of exploration, but as we run the agent over more episodes, we would like to use the knowledge we have gained from exploration to make more efficient choices. The discount factor controls the exploration-exploitation of the algorithm, and by gradually decaying it from a value of 1 (always exploration) to 0.01 (almost never exploration), the agent can converge to an optimal solution. The graph below shows the decay of discount factor.



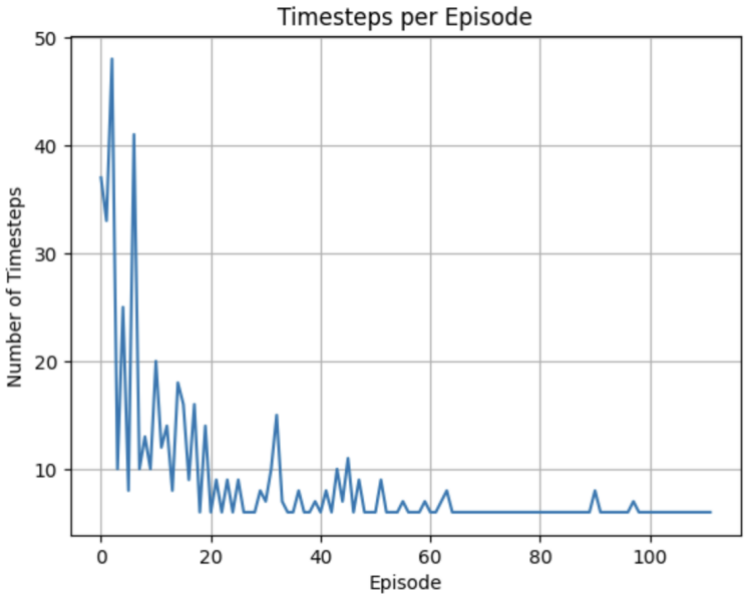
2) Total Rewards per Episode:
   This plot acts as a proof of convergence. If the algorithm has correctly converged, we should see that as more training episodes go by, the model will eventually converge to a state in which it obtains the maximum reward that is possible through the environment. The maximum reward that can be obtained in our environment is 22, and we see that over episodes, the reward values stabilize to the maximum reward.



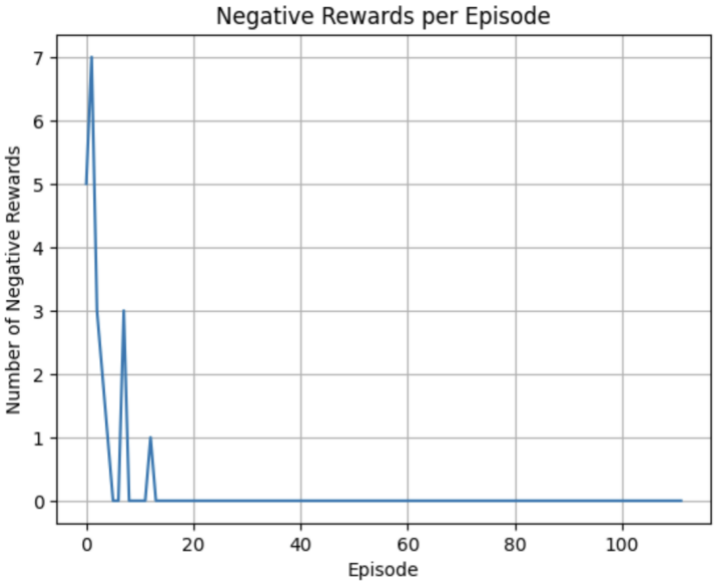
3) Cumulative Rewards per Episode:
This plot also acts as a proof of convergence. If the algorithm has correctly converged, we should expect to see the graph increase linearly over episodes. This is because as the number of episodes increases, the model will pick up constant maximum rewards. We see from the graph below that our algorithm has, converged correctly. There is a slight dip in the beginning, but that is expected as the model is in the exploration phase.



4) Timesteps vs Number of Episodes:
This graph can show us the speed of convergence in a sense. Over time we expect the agent to use the values in the Q-table to optimally arrive at the goal state. By exploiting the values in the Q-table, the agent will converge to the goal state with fewer timesteps. In the plot below, we see that the algorithm does take fewer timesteps as the number of episodes increases.

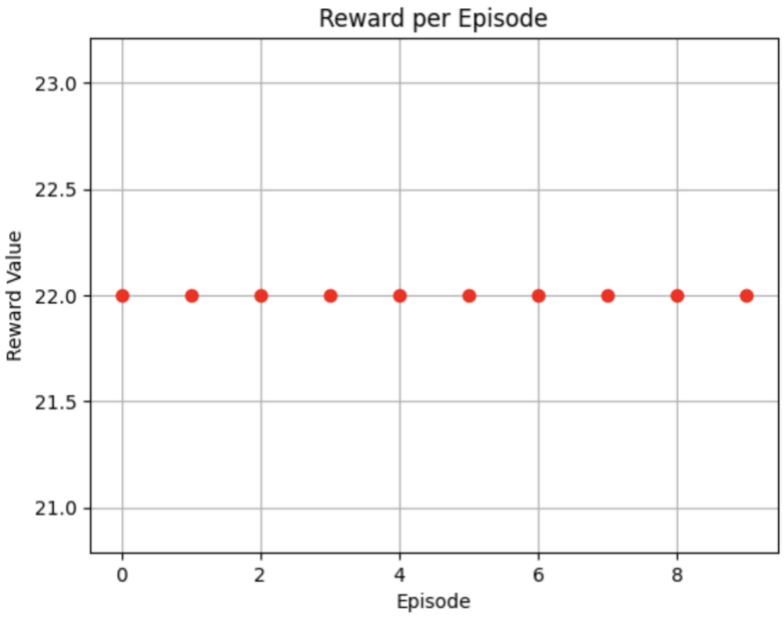


5) Number of Negative Rewards per Episode:
This graph should ideally reduce as the number of timesteps increases. This is because as the agent moves from exploration to exploitation over the episodes, if the algorithm is converging properly, we should expect that it does not take decisions that would land it in a state of facing penalty. We see from the below graph that this is correctly happening in case of our model.



Validation:

We will now validate our reinforcement model by running the algorithm greedily for 10 timesteps.



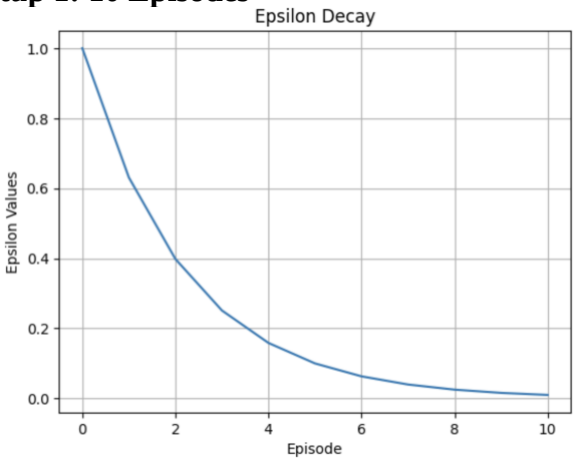
Looking at the graph, we see that the rewards obtained at every episode in the Reward per Episode graph are the maximum value possible for our grid world. The rewards also remain constant, which serves as a validation that the Q-table has been filled with optimal values.

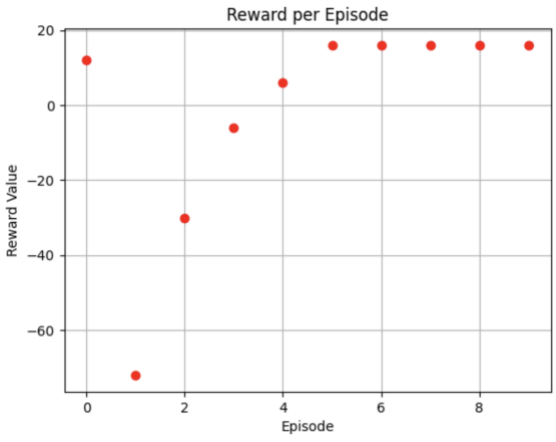
Hyper Parameter Tuning:

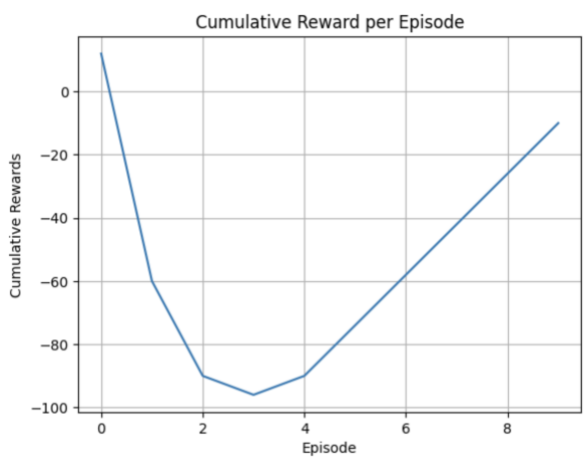
We will be tuning the hyper parameters to help our model converge to the optimal state. We will manually tune two hyper parameters – the number of episodes, and the discount factor.

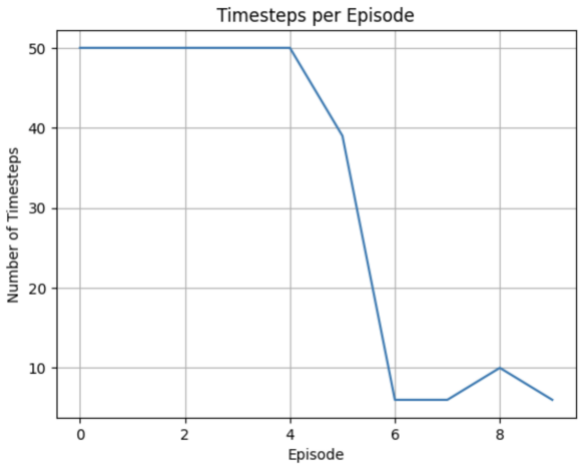
Parameter 1: Number of Episodes

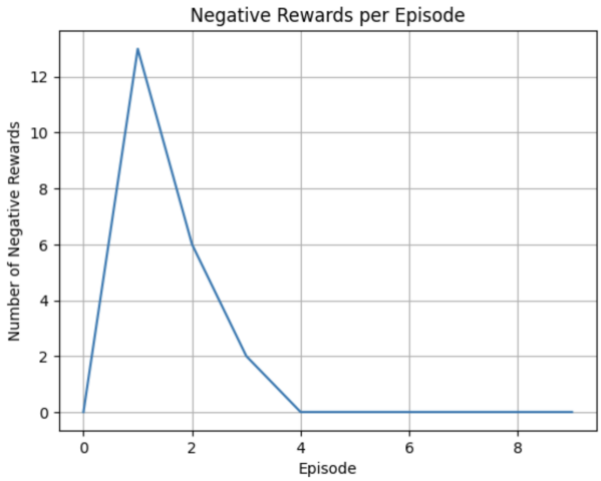
Setup 1: 10 Episodes





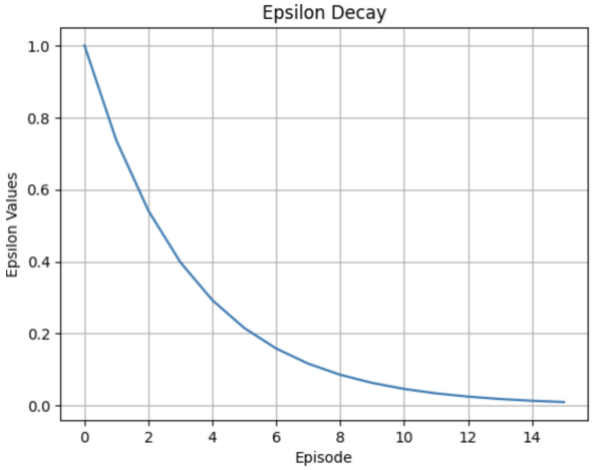


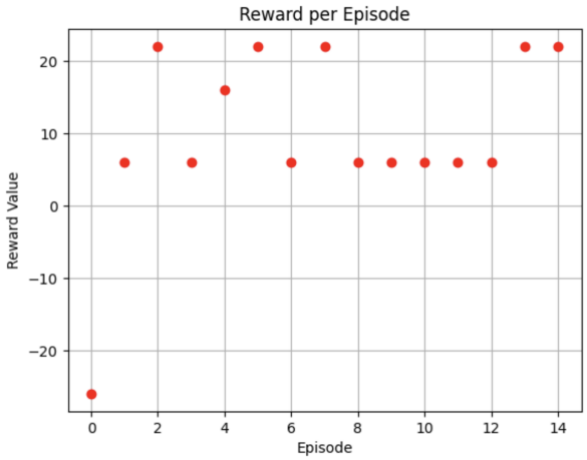


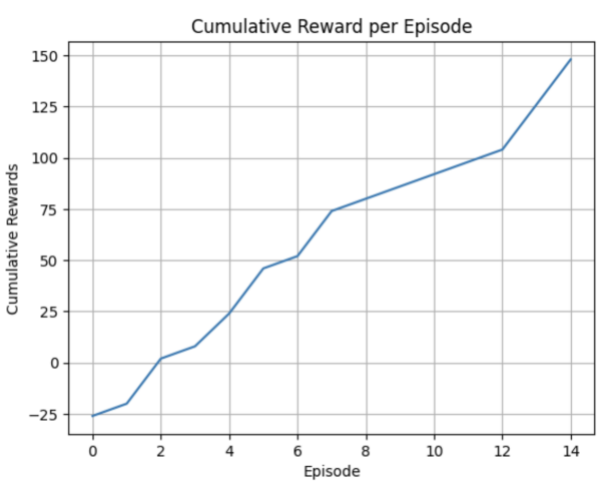


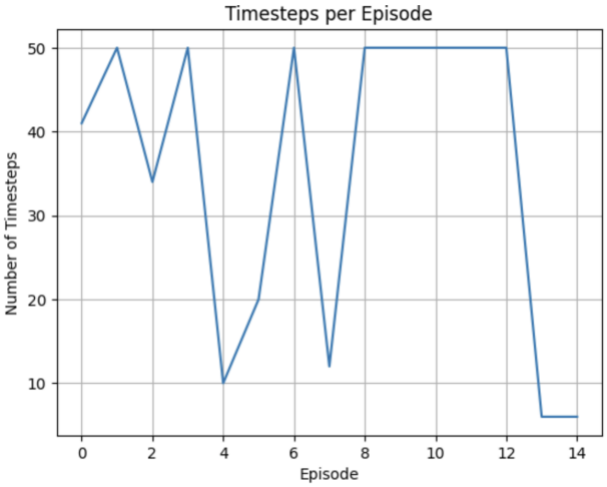
We see from the Reward per episode graph, that the algorithm has converged to a sub-optimal state. There is also not a linear relationship between cumulative reward per episode graph.

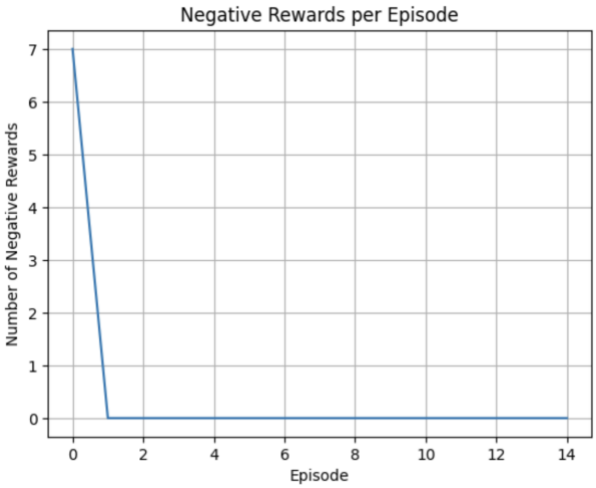
Setup 2: 15 Episodes





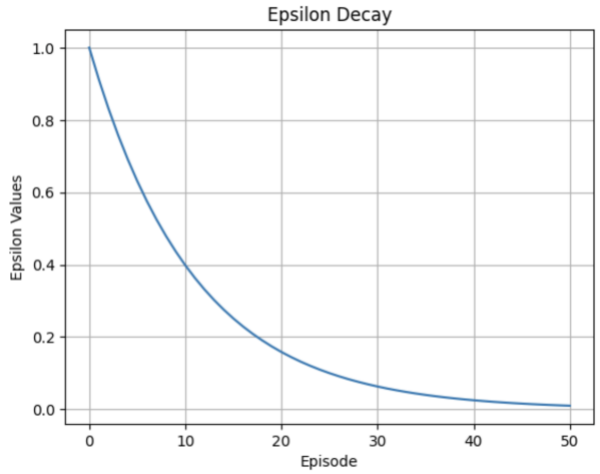


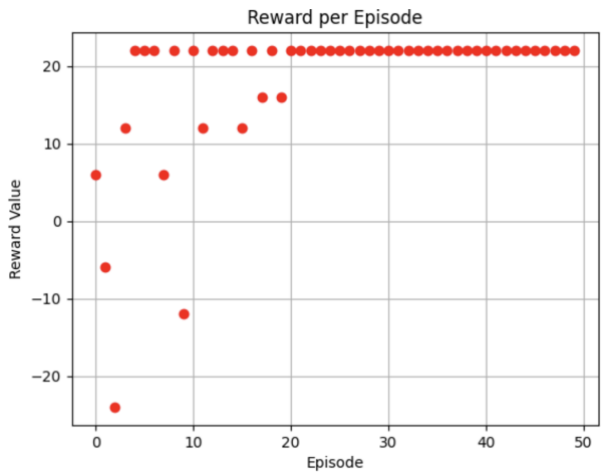


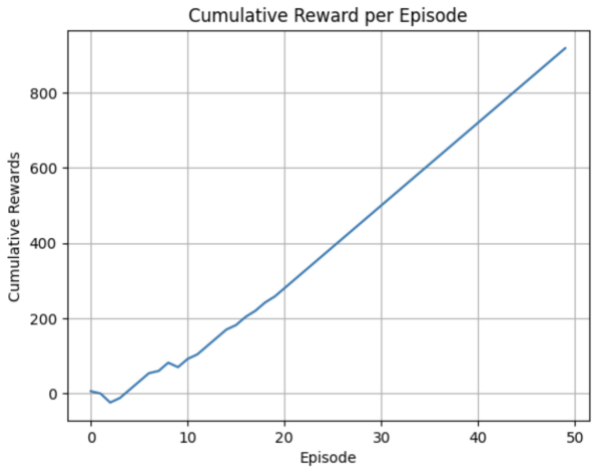


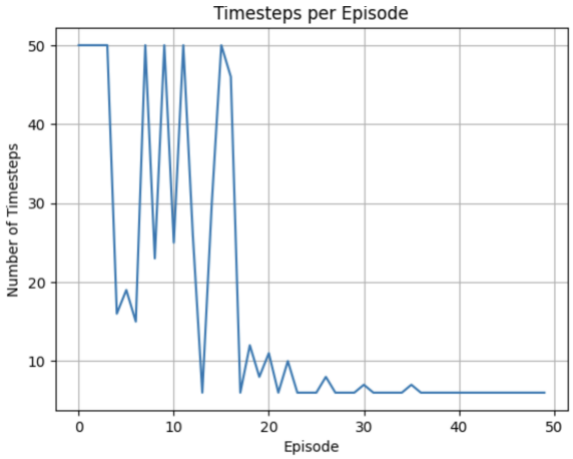
We see that there is some improvement in the cumulative rewards per episode graph, but as we can see from the Reward per Episode graph, the algorithm does not always constantly converge to the optimal value of maximum reward (22).

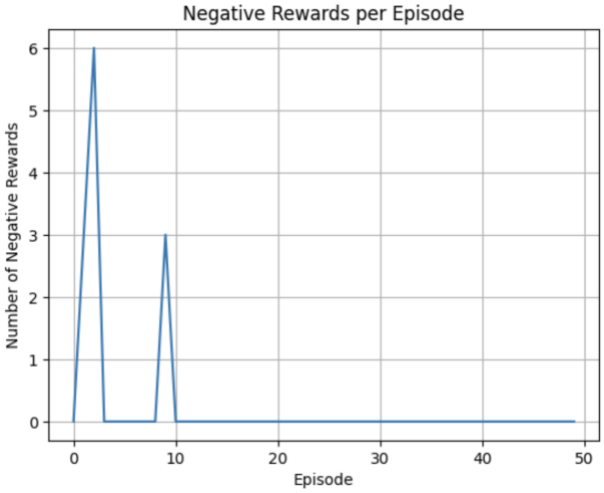
Setup 3: 30 Episodes







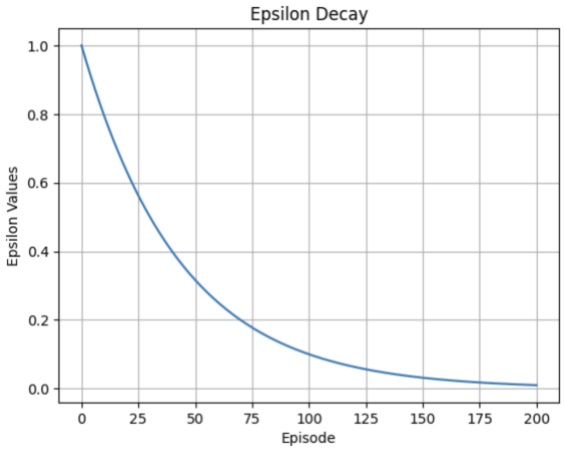


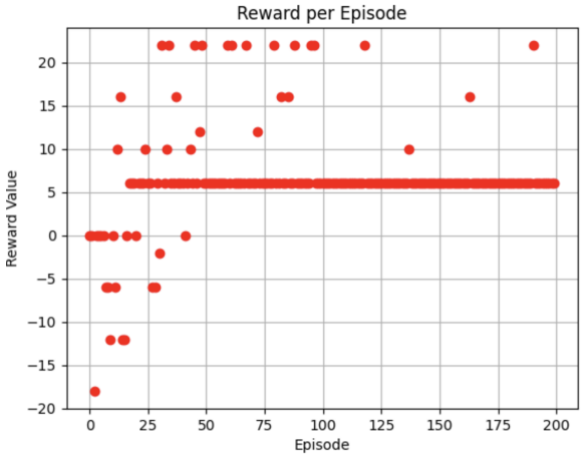


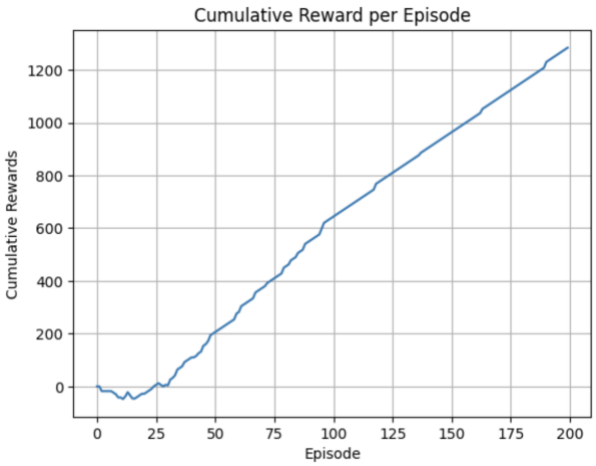
We see that both the Cumulative Rewards per Episode has improved and that the Reward per Episode graph shows convergence to the maximum value of 22. Therefore, it can be concluded that increasing the number of episodes improves the model.

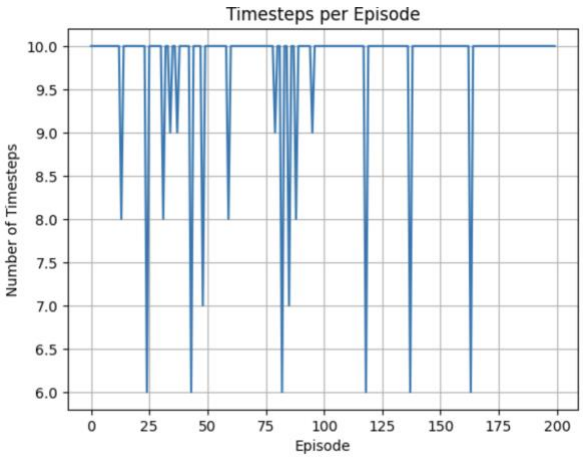
Parameter 2: Timesteps per Episode

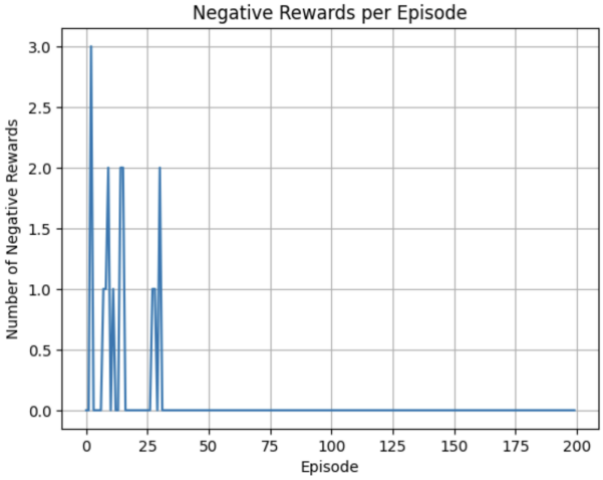
Setup 1: 10 Timesteps per Episode





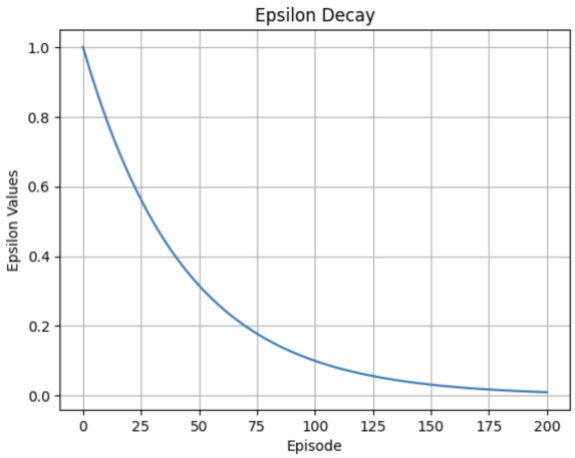


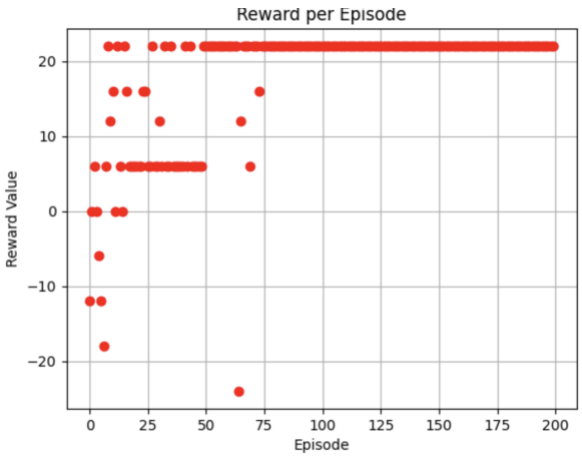


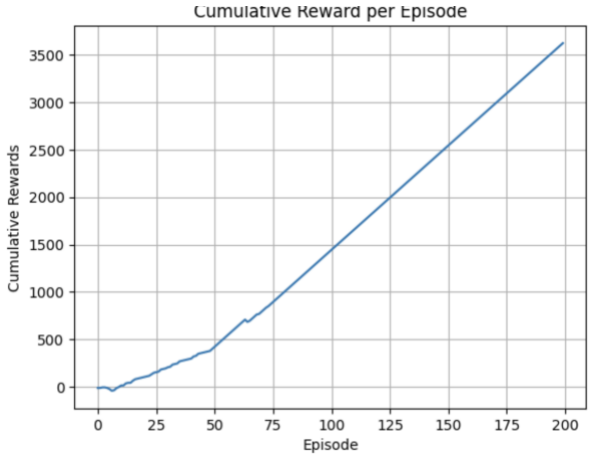


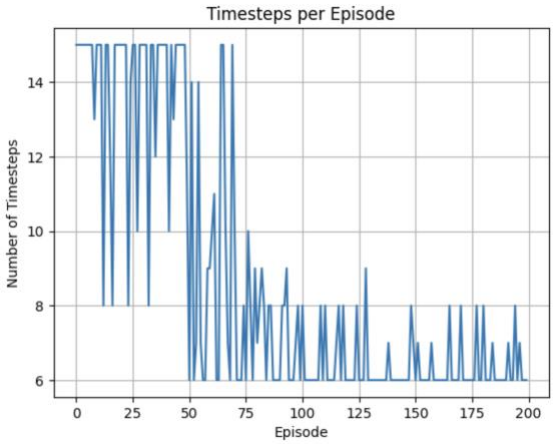
We see that the Rewards per Episode graph has converged to a suboptimal value. Also, the Timestamps per Episode does not reduce with the increase in episodes. This is expected as the algorithm needs more timesteps to explore.

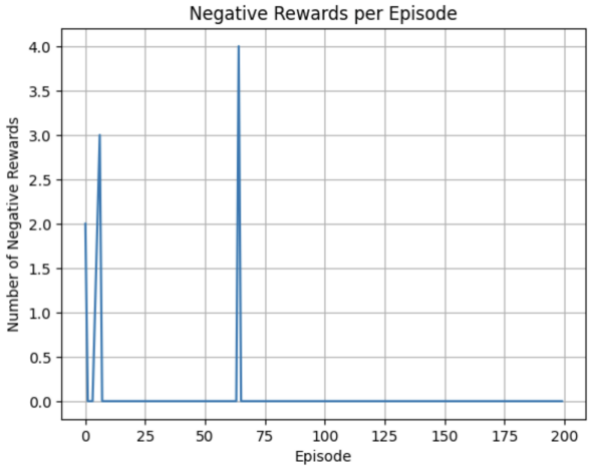
Setup 2: 15 Timesteps per Episode





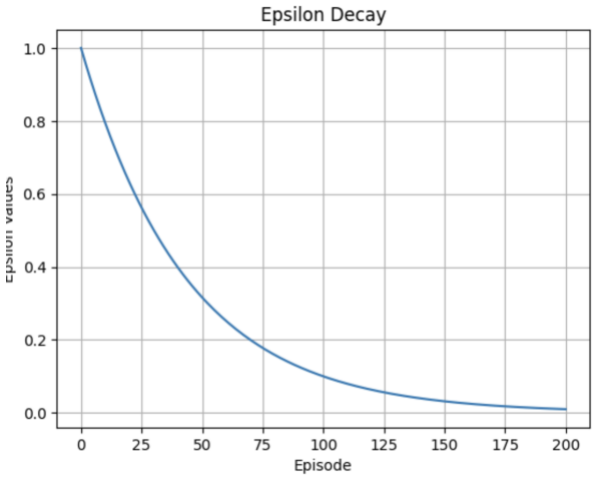


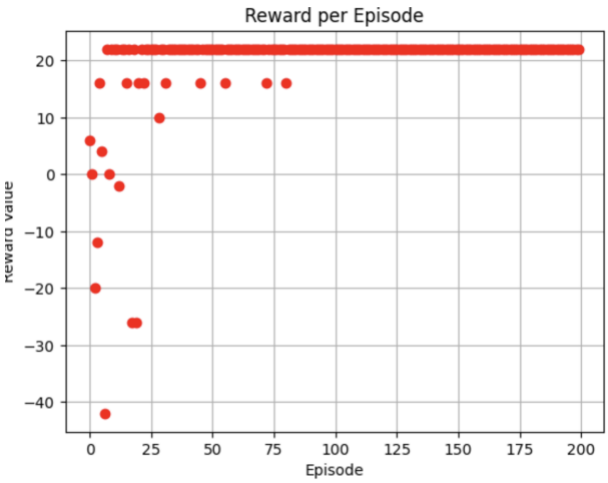


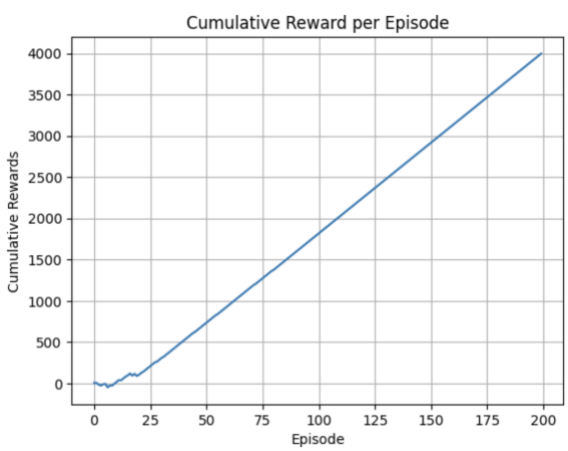


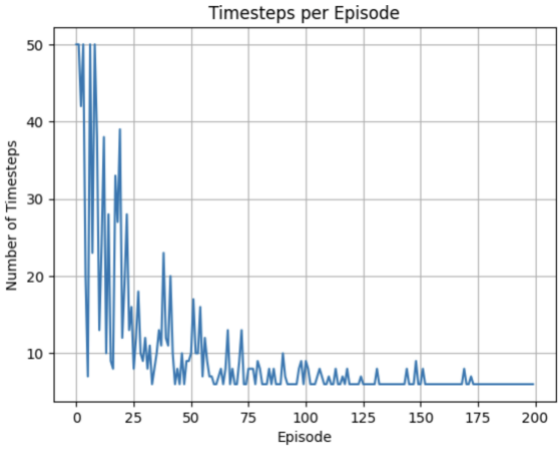
We see that the algorithm has converged, but it takes a long time to do so. Also, intermediately, it gets stuck in a suboptimal state. Another thing to note is that the number of timesteps has not become constant.

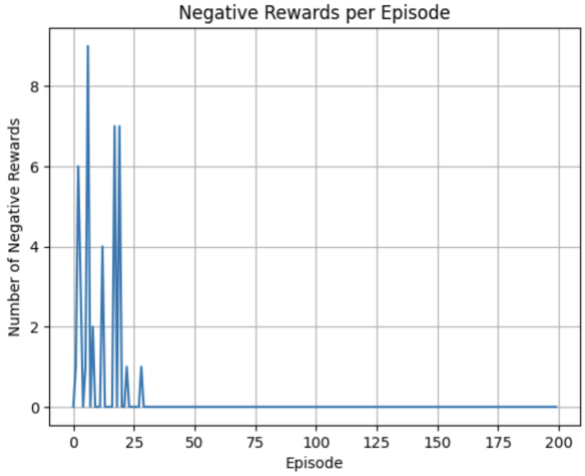
Setup 3: 50 Timesteps per Episode











We see that the Reward per Episode graph shows that the algorithm does not get stuck at a sub optimal state even intermediately. The Timesteps per Episode graph also shows significant improvement. The Cumulative Rewards per Episode graph is also almost exactly linear.

Using Optuna to Obtain the best Hyper Parameters

We will use Optuna to obtain the best hyper parameters for the algorithm. We will choose to maximize the value of the average rewards over all the episodes as a metric to tune the hyperparameters to. Another improvement we will make is to tune a total of 4 hyperparameters:

1. Learning Rate in the range of (0.1 – 0.2)
2. Discount Factor in the range of (0.9 – 0.99)
3. Number of episodes in the range of (20 – 1000)
4. Timesteps per episode in the range of (25 – 100)

By optimally searching for and running 300 unique combinations of these four hyper parameters, we obtain the following best values that maximize our value of average reward:

```
{
        'total_episodes': 208,
        'timesteps_per_episode': 98,
        'learning_rate': 0.17130025023718856,
        'discount_factor': 0.9066040050840016
}
```

The graphs for these values of hyper parameters have been shown in the first part of the report. These are the best values of hyper parameters that are to be reported to maximize our objective in the Lawnmower Grid World by SARSA.

PART III: SOLVING THE ENVIRONMENT USING Q-LEARNING

Q-Learning

Q-Learning is a reinforcement learning algorithm that can be used to help agents solve Markov Decision Processes (MDP). It is known as a tabular algorithm since it works by referencing a table of values, called Q-values. The algorithm works by updating Q-values (expected discount future reward) and referencing them to take an action in a certain state of an MDP. The Q-values are updated for each state-action pair using the following formula:

$$Q(s,a) = Q(s,a) + \alpha * (r + \gamma * max(Q(s',a')) - Q(s,a))$$

where:

1. Q(s,a) is the Q-value for the current state-action pair (s,a)
2. α is the learning rate
3. r is the immediate reward
4. γ is the discount factor
5. s' is the next state after taking action a in state s, and a' is the action that maximizes the Q-value for the next state s'

Key Features:

- It is a model free algorithm.
- It is an off-policy algorithm.
- It is a form of temporal difference learning.

Advantages:

- Performs well in environments where the transition probabilities and rewards are unknown.
- Can learn the optimal policy even when the agent is following a different policy during exploration.
- Very effective at handling delayed rewards.

Disadvantages:

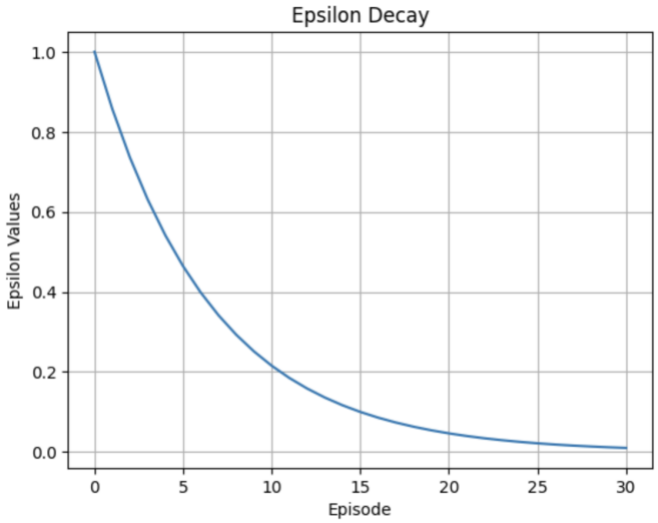
- May suffer from overestimation bias.
- Very sensitive to poor values of learning rate. Algorithm may diverge if tuned improperly.
- May not perform well where the environments are dynamic in nature.
- Sometimes does not work well in environments with noisy rewards.

Solving Environment

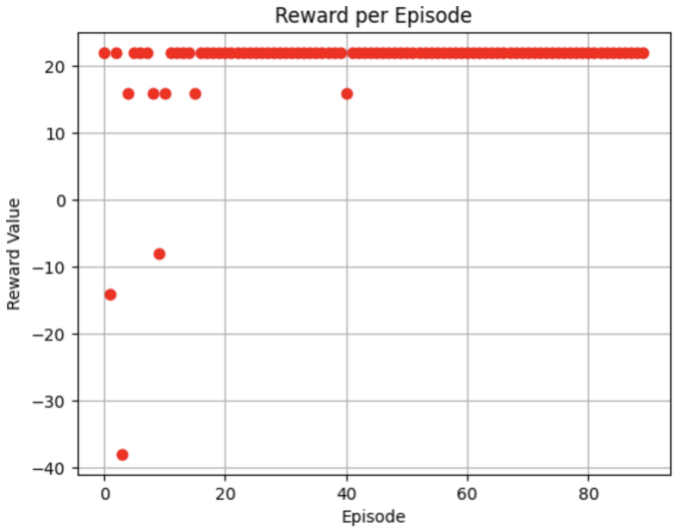
After using the algorithm to solve the environment, we can gauge the performance of the algorithm using the following plots:

1) Epsilon-Decay Plot:

Initially we want to have a high rate of exploration, but as we run the agent over more episodes, we would like to use the knowledge we have gained from exploration to make more efficient choices. The discount factor controls the exploration-exploitation of the algorithm, and by gradually decaying it from a value of 1 (always exploration) to 0.01 (almost never exploration), the agent can converge to an optimal solution. The graph below shows the decay of discount factor.

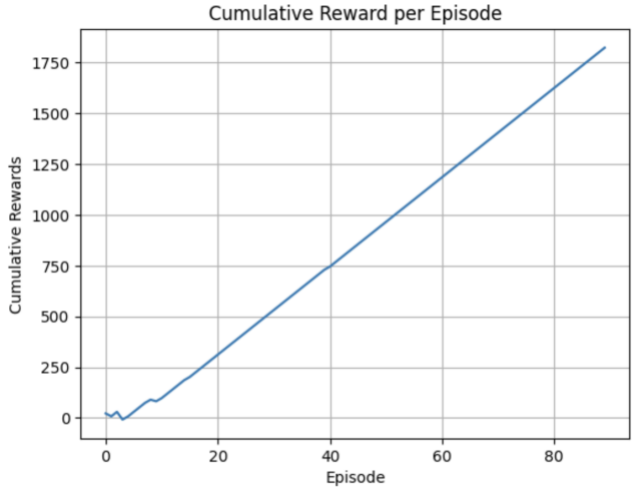


2) Total Rewards per Episode:
This plot acts as a proof of convergence. If the algorithm has correctly converged, we should see that as more training episodes go by, the model will eventually converge to a state in which it obtains the maximum reward that is possible through the environment. The maximum reward that can be obtained in our environment is 22, and we see that over episodes, the reward values stabilize to the maximum reward.

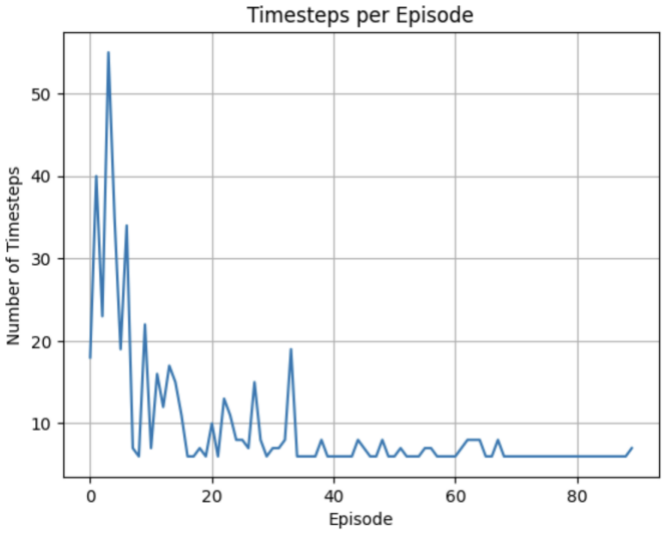


3) Cumulative Rewards per Episode:
This plot also acts as a proof of convergence. If the algorithm has correctly converged, we should expect to see the graph increase linearly over episodes. This is because as the number of episodes increases, the model will pick up constant maximum rewards. We see

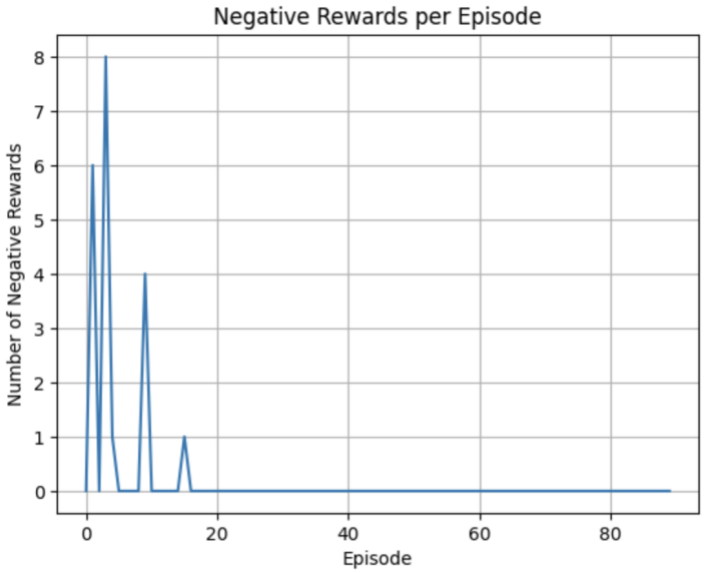
from the graph below that our algorithm has, converged correctly. There is a slight irregularity in the beginning, but that is expected as the model is in the exploration phase.



4) Timesteps vs Number of Episodes:
This graph can show us the speed of convergence in a sense. Over time we expect the agent to use the values in the Q-table to optimally arrive at the goal state. By exploiting the values in the Q-table, the agent will converge to the goal state with fewer timesteps. In the plot below, we see that the algorithm does take fewer timesteps as the number of episodes increases.

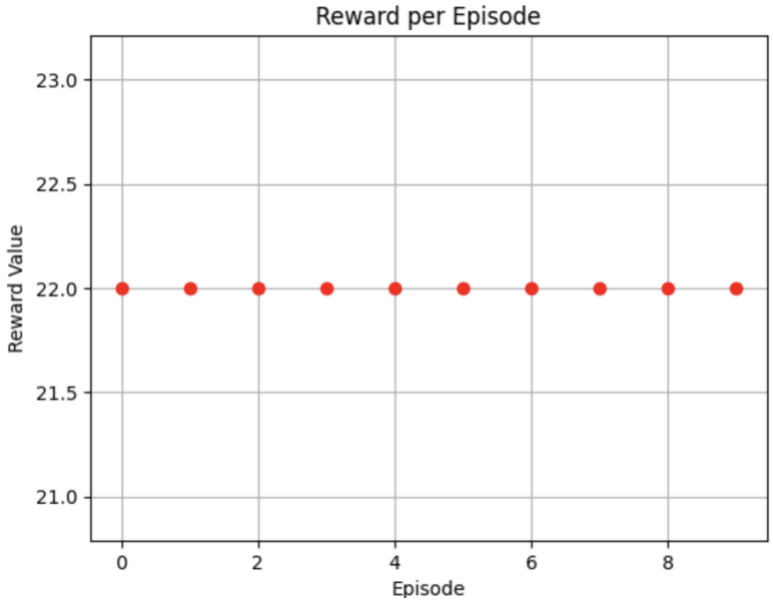


5) Number of Negative Rewards per Episode:
This graph should ideally reduce as the number of timesteps increases. This is because as the agent moves from exploration to exploitation over the episodes, if the algorithm is converging properly, we should expect that it does not take decisions that would land it in a state of facing penalty. We see from the below graph that this is correctly happening in case of our model.



Validation:

We will now validate our reinforcement model by running the algorithm greedily for 10 timesteps.

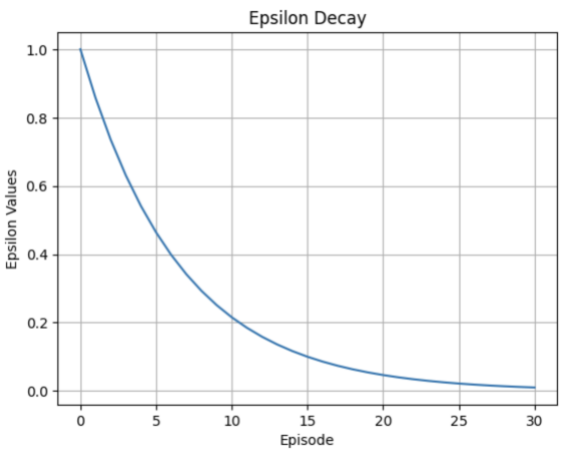


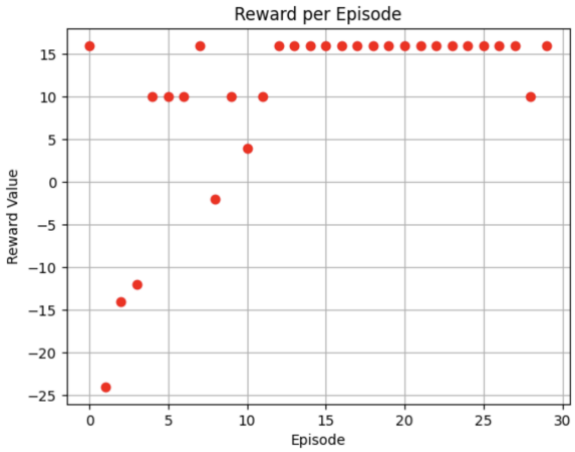
Looking at the graph, we see that the rewards obtained at every episode in the Reward per Episode graph are the maximum value possible for our grid world. The rewards also remain constant, which serves as a validation that the Q-table has been filled with optimal values.

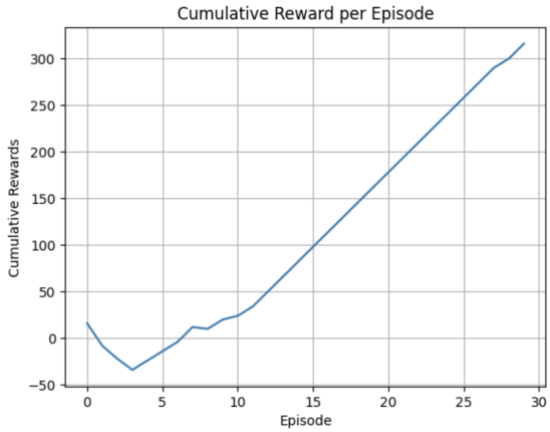
Hyper Parameter Tuning:

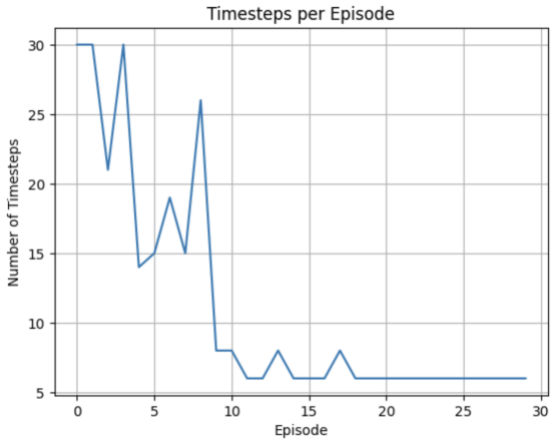
Parameter 1: Learning Rate

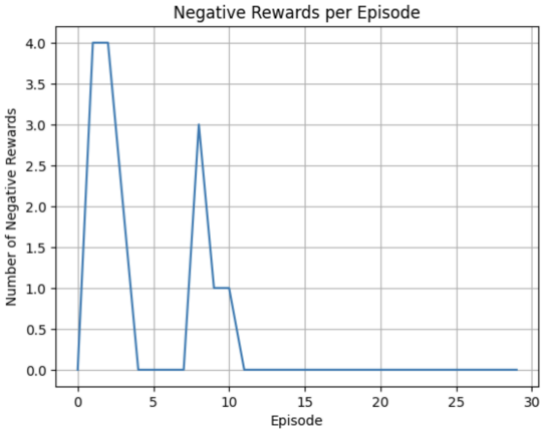
Setup 1: Learning Rate – 0.1





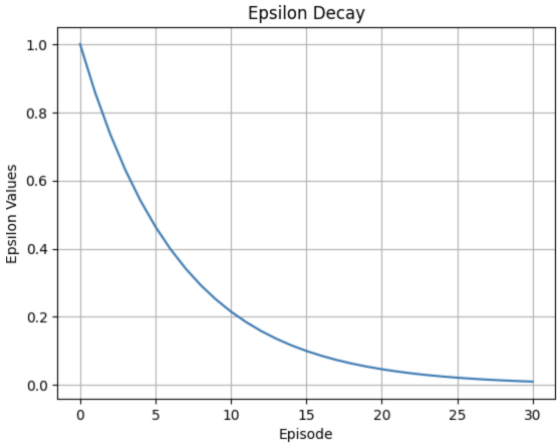


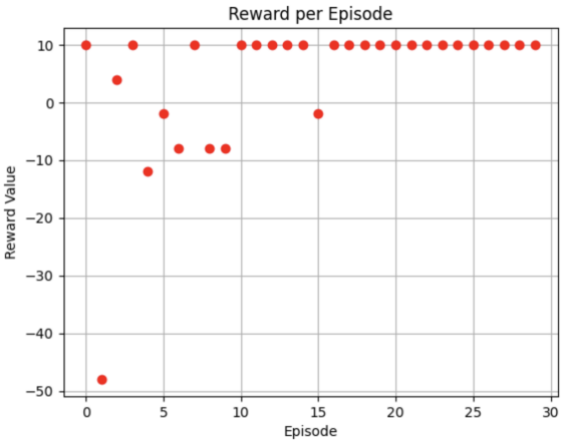


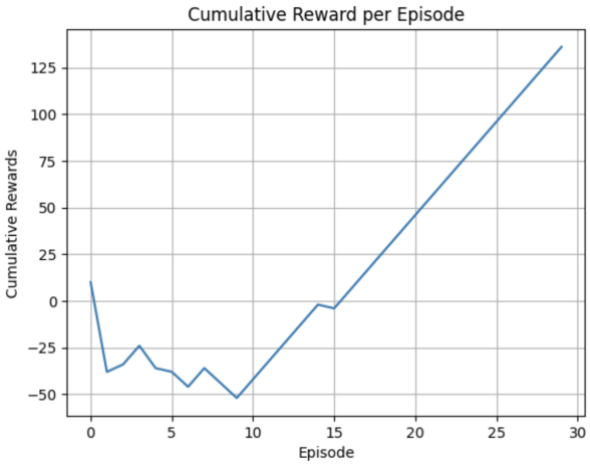


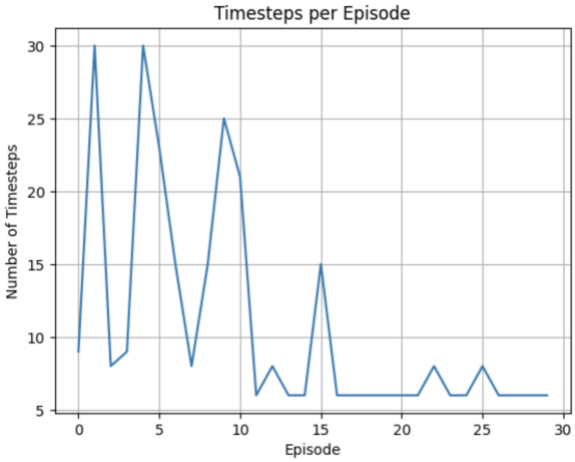
As we can see from the reward per episode graph, using a small learning rate has made our algorithm to converge to a suboptimal state where the reward is not the maximum reward that is attainable. This is because the agent does not drastically explore the environment, making it converge to a smaller reward.

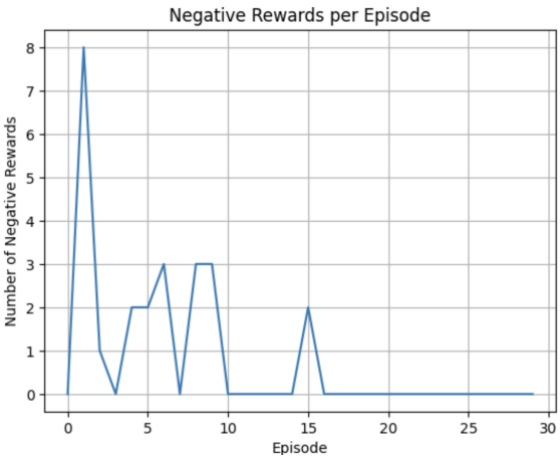
Setup 2: Learning Rate – 0.2





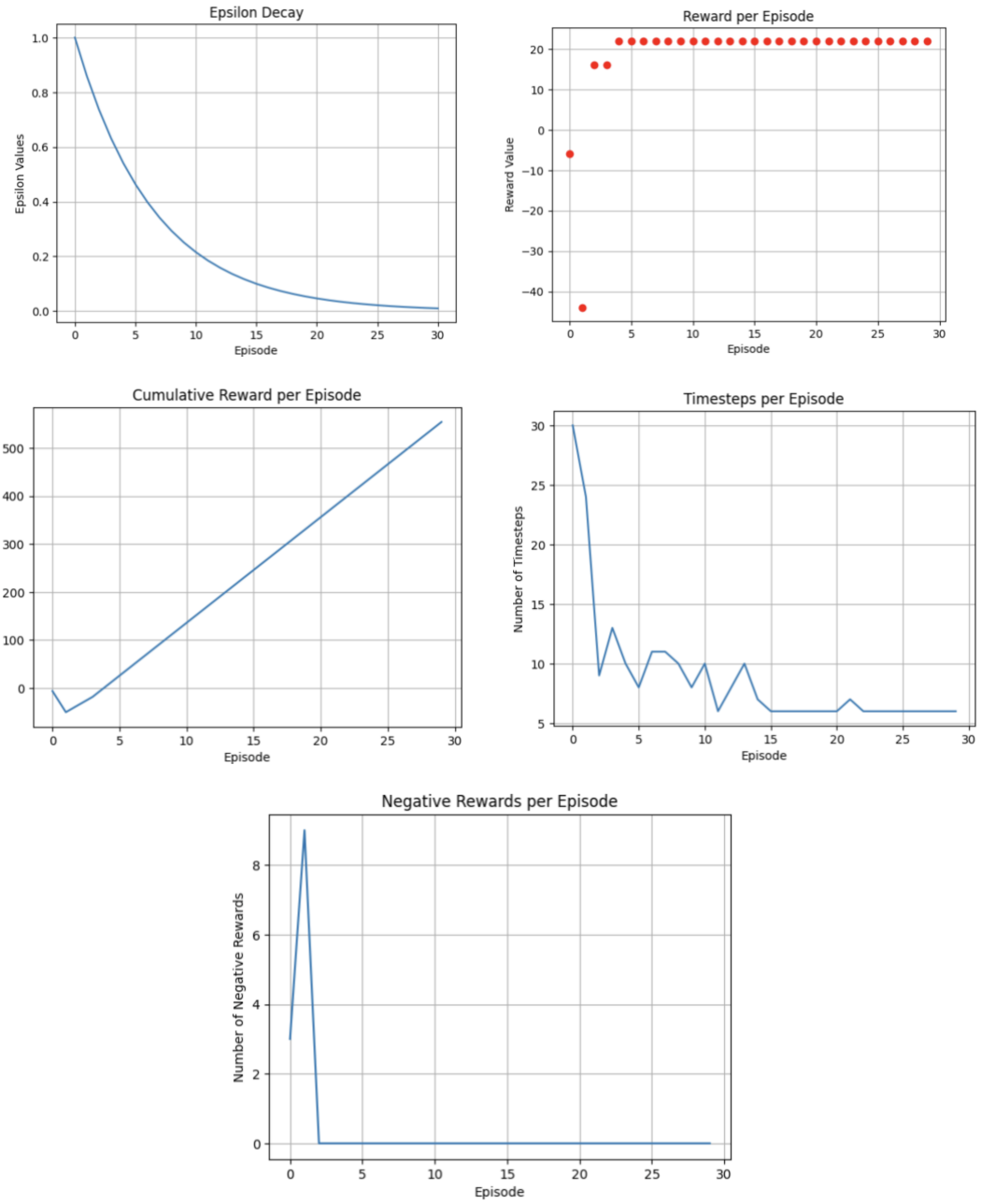






As we can see from the Rewards per Episode graph, taking a high learning rate has also made the exploration happen very quickly, which is also making the algorithm to converge at a suboptimal state where the reward is 10.

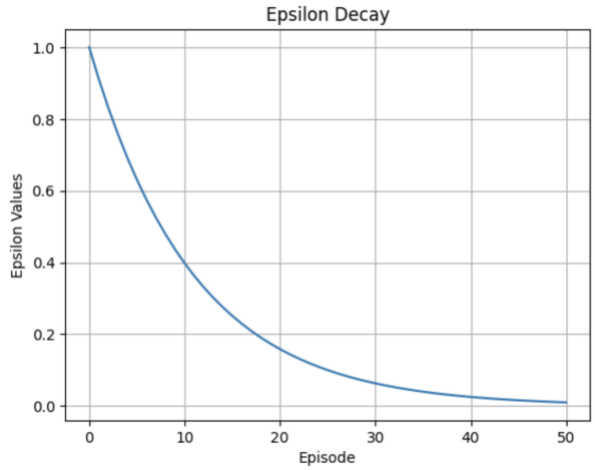
Setup 3: Learning Rate – 0.15

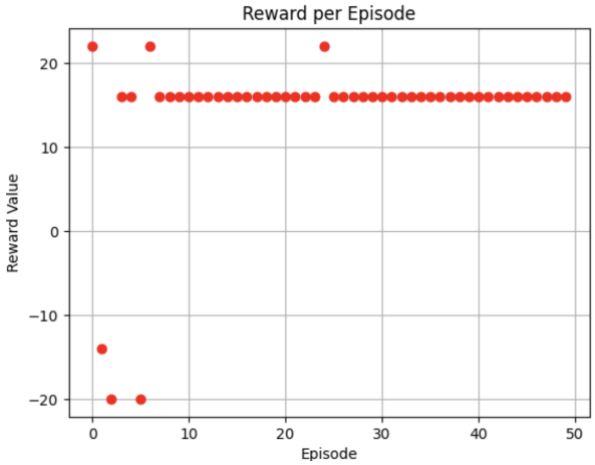


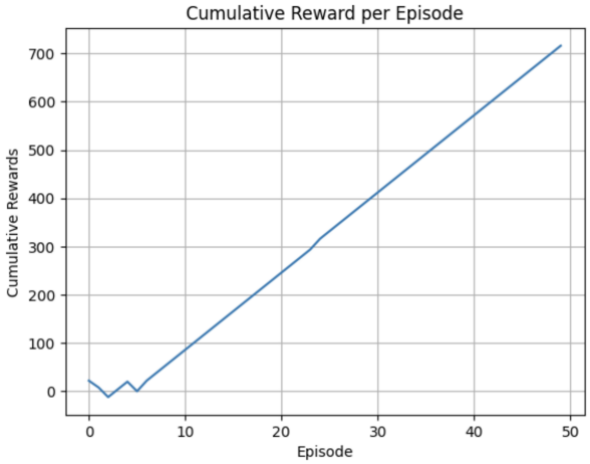
The graphs that are plotted that take a learning rate of 0.15 which is not too high, nor too low, show that the agent converges to the maximum rewards state. The negative rewards graph is also much smoother, indicating better performance.

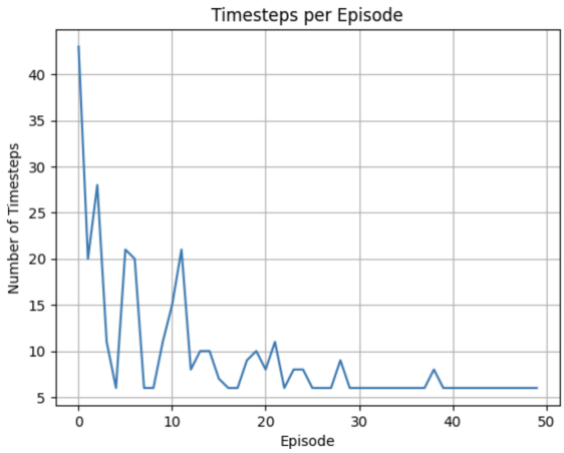
Parameter 1: Discount Factor

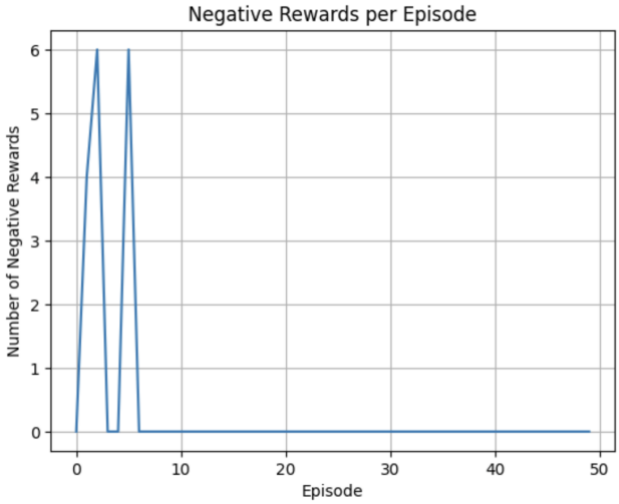
Setup 1: Discount Factor – 0.9





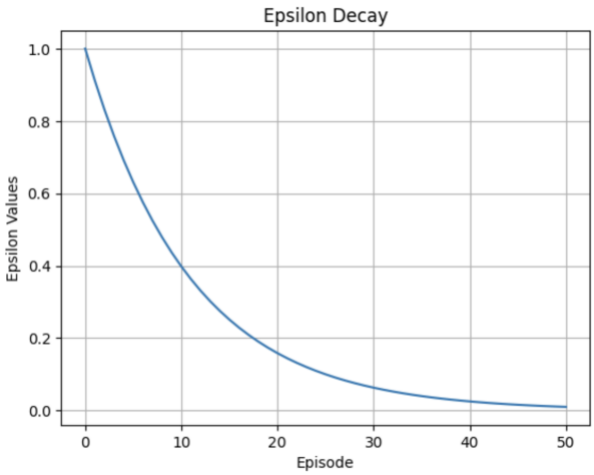


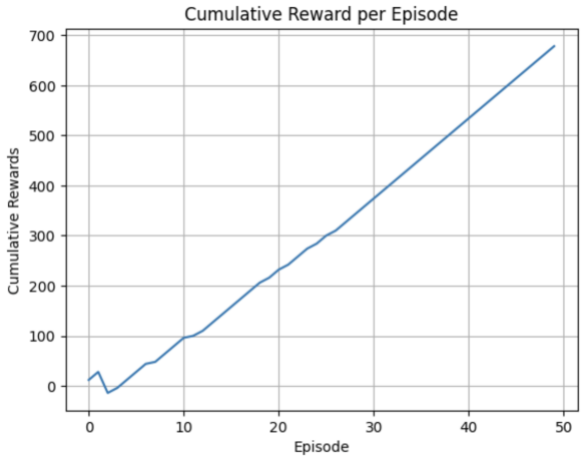


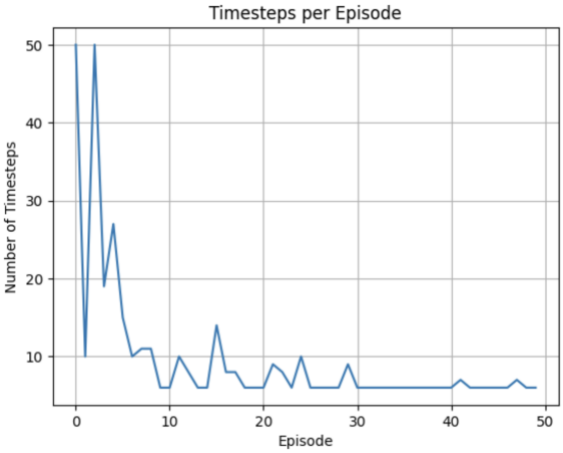


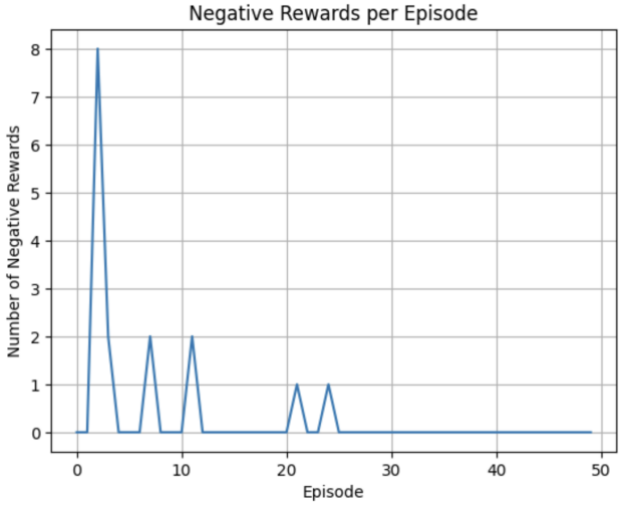
As we can see from the Reward per Episode graph, taking a relatively low value of discount factor results in convergence to a sub optimal state. This is because the agent has favored the short terms rewards that have been found within the grid world.

Setup 2: Discount Factor – 0.99



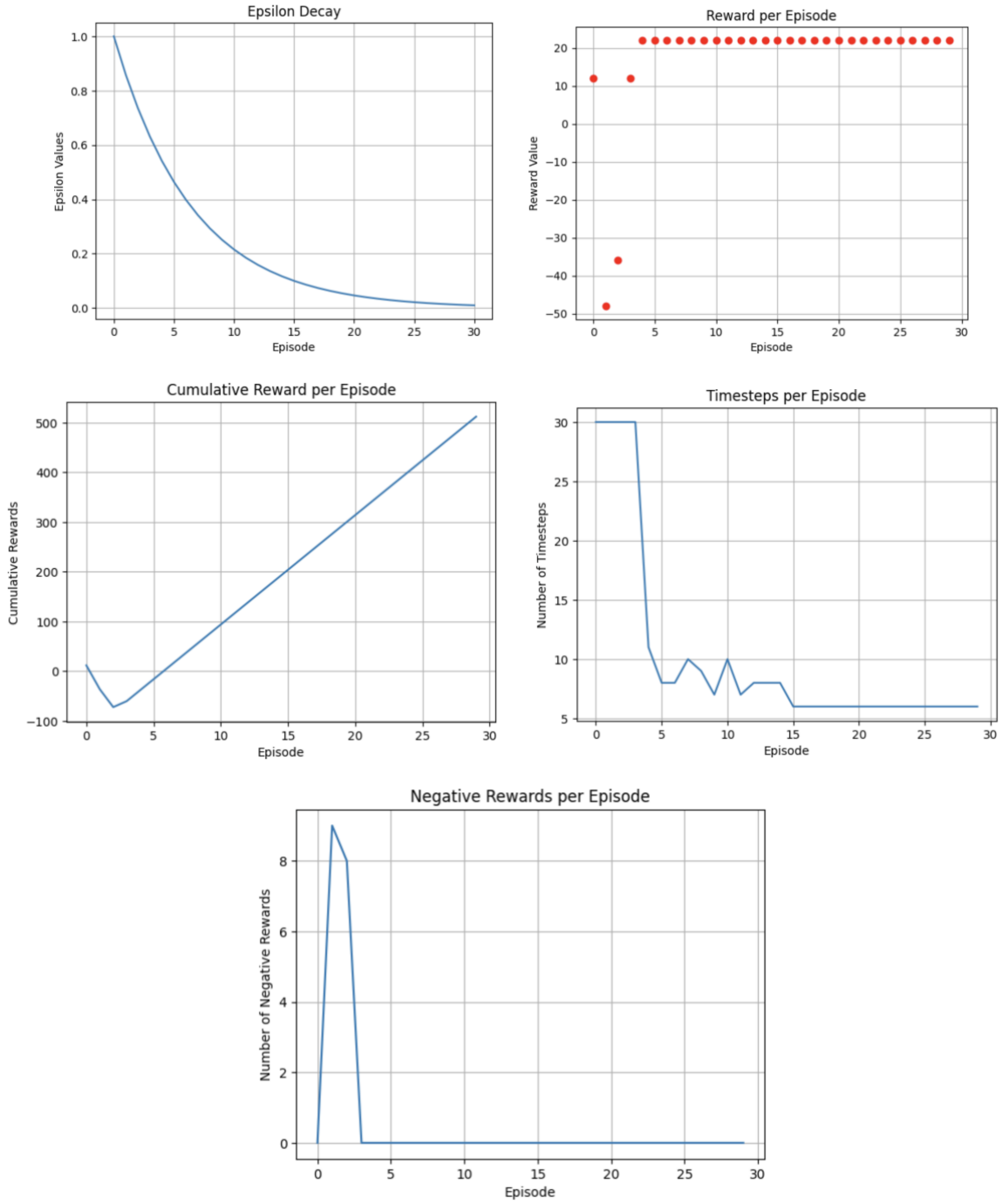






As we can see fromt the Reward per Epsisode graph, the algorithm has once again converged to a sub optimal state. This is because the agent prefers future rewards which may not be present and disregards the knowledge it has gained for the rewards it has found.

Setup 3: Discount Factor – 0.955



As we can see from the Rewards per Episode graph, the algorithm has converged to the optimal state to obtain maximum rewards. This is because taking a value of discount factor which is not too high or too low results in the agent being able to balance current and long term rewards and finally arrive at an optimal solution.

Using Optuna to Obtain the best Hyper Parameters

We will use Optuna to obtain the best hyper parameters for the algorithm. We will choose to maximize the value of the average rewards over all the episodes as a metric to tune the hyperparameters to. Another improvement we will make is to tune a total of 4 hyperparameters:

1. Learning Rate in the range of (0.1 – 0.2)
2. Discount Factor in the range of (0.9 – 0.99)
3. Number of episodes in the range of (20 – 1000)
4. Timesteps per episode in the range of (25 – 100)

By optimally searching for and running 300 unique combinations of these four hyper parameters, we obtain the following best values that maximize our value of average reward:

```
{
        'total_episodes': 95,
        'timesteps_per_episode': 73,
        'learning_rate': 0.10899292064011003,
        'discount_factor': 0.9341142673694802
}
```

The graphs for these values of hyper parameters have been shown in the first part of the report. These are the best values of hyper parameters that are to be reported to maximize our objective in the Lawnmower Grid World by Q-Learning.

SARSA vs. Q-Learning Performance Analysis

In this section, we will compare the performance of the SARSA and Q-Learning algorithms. We first define an identical environment to compare the algorithms. The environment parameters have **50 episodes and 50 timesteps per episode**. We then run the SARSA and Q-Learning algorithms with their **respective best parameters** for such an environment. Since we keep the environment definitions constant, we also use the **standard epsilon decay rate between both** for 50 episodes. That leaves us with the hyper parameters learning rate and discount factor for which we need to find optimal values. Using Optuna, we find the optimal hyper parameters to be the following:

- SARSA (Learning Rate: 0.13681219259059596, Discount Factor: 0.9293139448701037)
- Q-Learning (Learning Rate: 0.1759780044971019, Discount Factor: 0.9350914494434022)

The best mode for visualizing rewards would be the cumulative rewards obtained by each, since if we look at just rewards, we will see a graph that shows both algorithms obtaining rewards after a certain number of episodes.

We have shown the plot of cumulative rewards of both the algorithms in a single graph below. From the graph, we see that Q-Learning converges faster, but there is a slight indication that SARSA may perform better, given the increase in number of episodes.

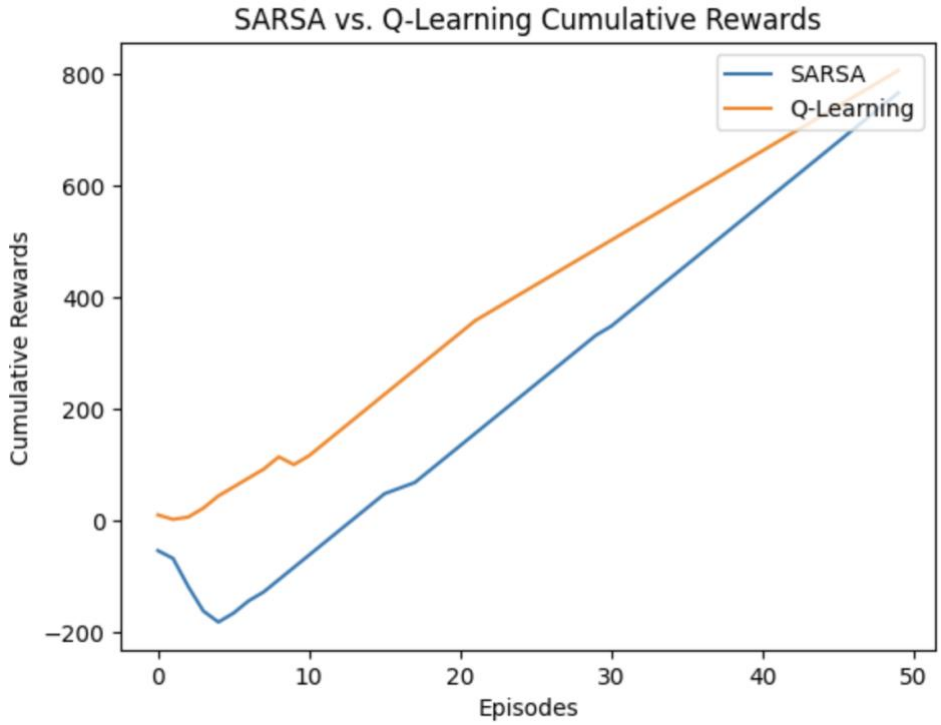


Figure 1. SARSA vs. Q-Learning in an Environment for 50 Episodes

To see what happens at a larger number of episodes, let's plot a graph for the same environment, but the only change being the increase in number of episodes.

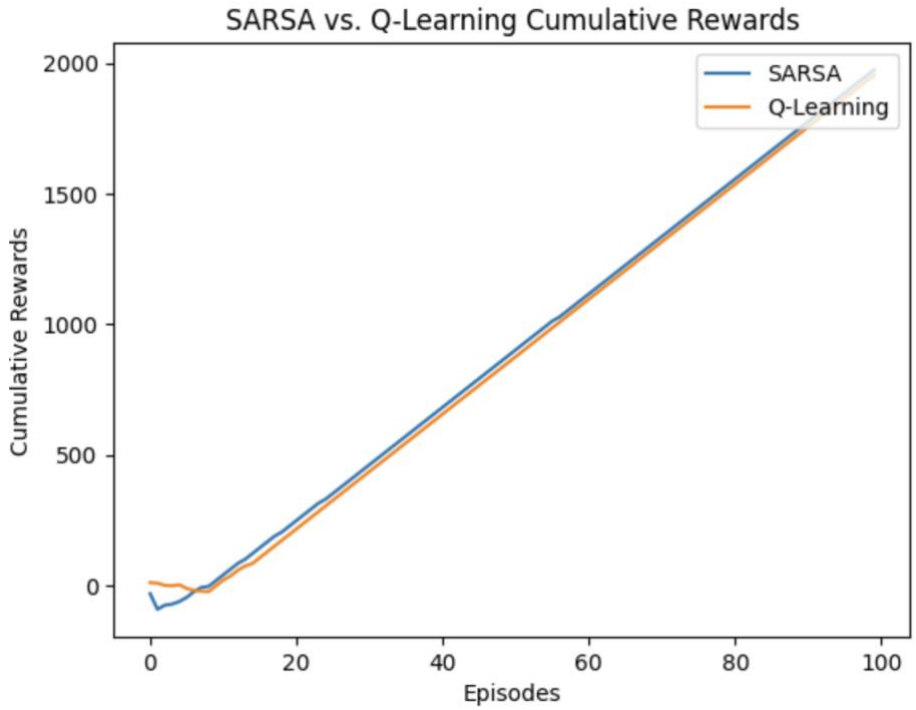


Figure 2. SARSA vs. Q-Learning in an Environment for 100 Episodes

Looking at the plot above which was drawn for 100 episodes, we see that our hypothesis is true, since after a point of time, the cumulative rewards for SARSA surpass the cumulative rewards for Q-Learning. This means that the q-values obtained in SARSA are more stable.