Assignment 1 - Defining & Solving RL Environments

Status	Reinforcement Learning

CSE 546: Reinforcement Learning

Part 1 [Total: 30 points] - Defining RL Environments

1.1 Deterministic Environment [15 points]

```
class FrozenLakeEnv(gym.Env):
    metadata = {'render.modes': []}
    def __init__(self, max_timestamp=10):
                ...
        ## STATES SET #############
        self.state = np.zeros((4, 4))
        self.myskater = np.asarray([0, 0])
        self.goal_loc = np.asarray([3, 3])
        self.gem_loc = [np.asarray([0, 2]), np.asarray([3, 2])]
        self.hole_loc = [np.asarray([1, 3]), np.asarray([2, 0])]
```

```
        self.state[tuple(self.myskater)] = 0.2
        self.state[tuple(self.goal_loc)] = 0.8
        for pos in self.gem_loc:
            self.state[tuple(pos)] = 0.5
        for pos in self.hole_loc:
            self.state[tuple(pos)] = 0.4
        ############################
                ...

    def step(self, action):
                ...
        ## ACTIONS SET #############
        if action == 0:    # moves right
            self.myskater[0] += 1
        elif action == 1:   # moves left
            self.myskater[0] -= 1
        elif action == 2:   # moves up
            self.myskater[1] += 1
        elif action == 3:    # moves down
            self.myskater[1] -= 1
        ############################

        # If the agent is in the same position as the previous step, choose a different action
        prev_state_positions = np.argwhere(self.prev_state == 0.2)
        if len(prev_state_positions) > 0 and np.array_equal(self.myskater, prev_state_position
s[0]):
            while action == self.prev_action:
                action = self.action_space.sample()
            self.flag_out_grid = 1
                ...
        return self.state.flatten(), reward, terminated, truncated, info

    def reset(self, **kwargs):
        self.state = np.zeros((4, 4))
        self.myskater = np.asarray([0, 0])
        self.state[tuple(self.myskater)] = 0.2
        self.state[tuple(self.goal_loc)] = 0.8

        for pos in self.gem_loc:
            self.state[tuple(pos)] = 0.5
        for pos in self.hole_loc:
            self.state[tuple(pos)] = 0.4

        self.prev_state = np.zeros((4, 4))
        self.prev_action = None
        self.flag_out_grid = 0
                ...
        return obs, info

    def calculate_reward(self):
                ...
        ## REWARDS SET #############
        if np.array_equal(self.myskater, self.goal_loc):
            reward = 10  # Positive reward for reaching goal
```

```
        elif np.array_equal(self.myskater, self.hole_loc[0]):
            reward = -5  # negative reward for reaching holes 1
        elif np.array_equal(self.myskater, self.hole_loc[1]):
            reward = -6   # negative reward for reaching holes 2
        elif np.array_equal(self.myskater, self.gem_loc[0]):
            reward = 5   # positive reward for reaching gems 1
        elif np.array_equal(self.myskater, self.gem_loc[1]):
            reward = 6  # positive reward for reaching gems 2
        elif current_distance_to_goal < prev_distance_to_goal:
            reward = 1  # Positive reward for moving closer to goal
        elif current_distance_to_goal > prev_distance_to_goal:
            reward = -1  # Negative reward for moving away to goal
        else:
            reward = -0.1  # Slight negative reward for no change
        ##########################

        return reward

    def render(self):
        fig, ax = plt.subplots()
        plt.title('Frozen Lake Environment')

        skater_img = plt.imread('icons8-skateboard-100.png')
        hole_img = plt.imread('icons8-hole-100.png')
        gem_img = plt.imread('icons8-gems-100.png')
        goal_img = plt.imread('icons8-flag-100.png')
        skater_hole_drown_img = plt.imread('agent_hole_drown.png')
        skater_gem_lottery_img = plt.imread('agent_gems_lottery.png')
        agent_flag_winner_img = plt.imread('agent_flag_winner.png')
        agent_grid_cross_img = plt.imread('agent_grid_cross.png')

        # Plot Skater
                ...
        # Plot Holes
                ...
        # Plot Gems
                ...
        # Plot goal
                ...
        plt.xticks(np.arange(-0.5, 4.5, 1))
        plt.yticks(np.arange(-0.5, 4.5, 1))
        plt.gca().set_xticklabels(np.arange(-0.5, 4.5, 1))
        plt.gca().set_yticklabels(np.arange(-0.5, 4.5, 1))
        plt.show()
```

1.2 Stochastic Environment [15 points]

```
class StochasticFrozenLakeEnv(gym.Env):
    metadata = {'render.modes': []}

    def __init__(self, max_timestamp=10):
                ...
```

```
        self.state = np.zeros((4, 4))
        self.myskater = np.asarray([0, 0])
        self.goal_loc = np.asarray([3, 3])
        self.gem_loc = [np.asarray([0, 2]), np.asarray([3, 2])]
        self.hole_loc = [np.asarray([1, 3]), np.asarray([2, 0])]

        self.state[tuple(self.myskater)] = 0.2
        self.state[tuple(self.goal_loc)] = 0.8
        for pos in self.gem_loc:
            self.state[tuple(pos)] = 0.5
        for pos in self.hole_loc:
            self.state[tuple(pos)] = 0.4
                ...

    def step(self, action):
                ...
        # Define a probability distribution for actions
        randomness = [0.35, 0.15, 0.35, 0.15]  # probability for each action

        # Sample an action based on the probability distribution
        action = np.random.choice(4, p=randomness)

        if action == 0:
            self.myskater[0] += 1
        elif action == 1:
            self.myskater[0] -= 1
        elif action == 2:
            self.myskater[1] += 1
        elif action == 3:
            self.myskater[1] -= 1
                ...
        reward = self.calculate_reward()
                ...
        return self.state.flatten(), reward, terminated, truncated, info

    def reset(self, **kwargs):
        self.state = np.zeros((4, 4))
        self.myskater = np.asarray([0, 0])
        self.state[tuple(self.myskater)] = 0.2
        self.state[tuple(self.goal_loc)] = 0.8

        for pos in self.gem_loc:
            self.state[tuple(pos)] = 0.5
        for pos in self.hole_loc:
            self.state[tuple(pos)] = 0.4

        self.prev_state = np.zeros((4, 4))
        self.prev_action = None
        self.flag_out_grid = 0

        obs = self.state.flatten()
        self.timestep = 0
        info = {}
        self.penalty_counter = 0
```

```
        return obs, info

    def calculate_reward(self):
            ...
        ## REWARDS SET #############
        if np.array_equal(self.myskater, self.goal_loc):
            reward = 10  # Positive reward for reaching goal
        elif np.array_equal(self.myskater, self.hole_loc[0]):
            reward = -5  # negative reward for reaching holes 1
        elif np.array_equal(self.myskater, self.hole_loc[1]):
            reward = -6   # negative reward for reaching holes 2
        elif np.array_equal(self.myskater, self.gem_loc[0]):
            reward = 5   # positive reward for reaching gems 1
        elif np.array_equal(self.myskater, self.gem_loc[1]):
            reward = 6  # positive reward for reaching gems 2
        elif current_distance_to_goal < prev_distance_to_goal:
            reward = 1  # Positive reward for moving closer to goal
        elif current_distance_to_goal > prev_distance_to_goal:
            reward = -1  # Negative reward for moving away to goal
        else:
            reward = -0.1  # Slight negative reward for no change
        ###########################

        return reward

    def render(self):
        fig, ax = plt.subplots()
        plt.title('Frozen Lake Environment')

        skater_img = plt.imread('icons8-skateboard-100.png')
        hole_img = plt.imread('icons8-hole-100.png')
        gem_img = plt.imread('icons8-gems-100.png')
        goal_img = plt.imread('icons8-flag-100.png')
        skater_hole_drown_img = plt.imread('agent_hole_drown.png')
        skater_gem_lottery_img = plt.imread('agent_gems_lottery.png')
        agent_flag_winner_img = plt.imread('agent_flag_winner.png')
        agent_grid_cross_img = plt.imread('agent_grid_cross.png')

        # Plot Skater
        ...
        # Plot Holes
        ...

            # Plot Gems
        ...
        # Plot goal
        ...

        plt.xticks(np.arange(-0.5, 4.5, 1))
        plt.yticks(np.arange(-0.5, 4.5, 1))
        plt.gca().set_xticklabels(np.arange(-0.5, 4.5, 1))
        plt.gca().set_yticklabels(np.arange(-0.5, 4.5, 1))
        plt.show()
```

1. **Describe the deterministic and stochastic environments, which were defined (set of actions/states/rewards, main objective, etc).**

- In our environment, we have a skater as our agent that has to reach the goal destination with maximum rewards.
- We have added two gems locations which counts for positive rewards and two holes locations which drowns the skater as negative rewards.

To understand the locations in terms of grid indices, see the below code:

```
# start position of our skater
start_pos = [0, 0]

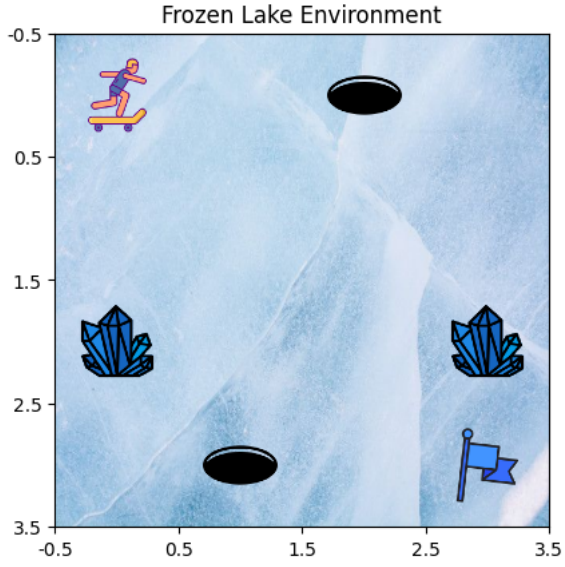
# goal position of our skater
goal_pos = [3, 3]

# Gems positions
gem1 = [0, 2]
gem2 = [3, 2]

# Holes positions
hole1 = [1, 3]
hole2 = [2, 0]
```

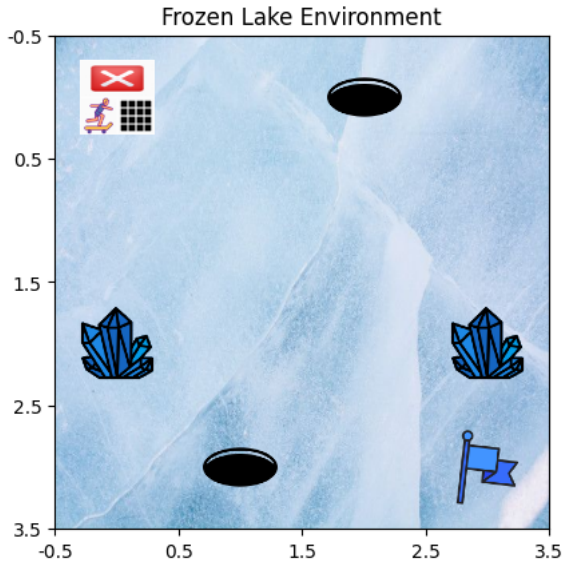
- **Set of States:**
  - We have 6 different states that a skater agent can end up in:

    1) Agent Initial State: This is where the skater starts his race to reach the goal position.
    - Initially he is at [0, 0] on the bottom left corner. The goal is at the top right corner of the grid, as shown below:



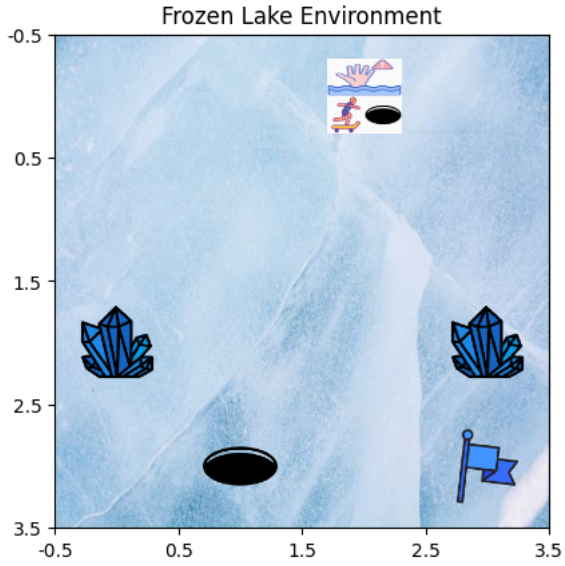
2) Agent going out of the Grid State: If the skater tries to go out of the grid he ends up in the state.

- Consider from the initial position, if the agent tries to go from [0,0] to [-1, 0] or [0, -1] then the agent ends up in this going out of the grid state.



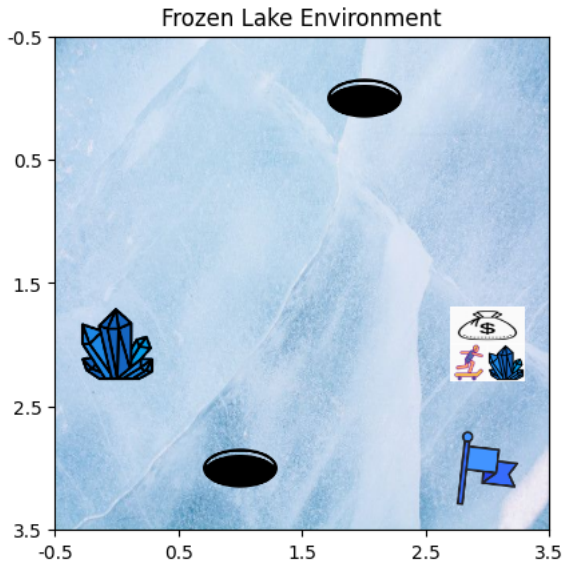
3) In Hole State: If the agent ends up in a hole then he is in this state.

- Agent slips in the hole and drowns if he lands on the locations of the rocks are [2, 0] and [2, 3]



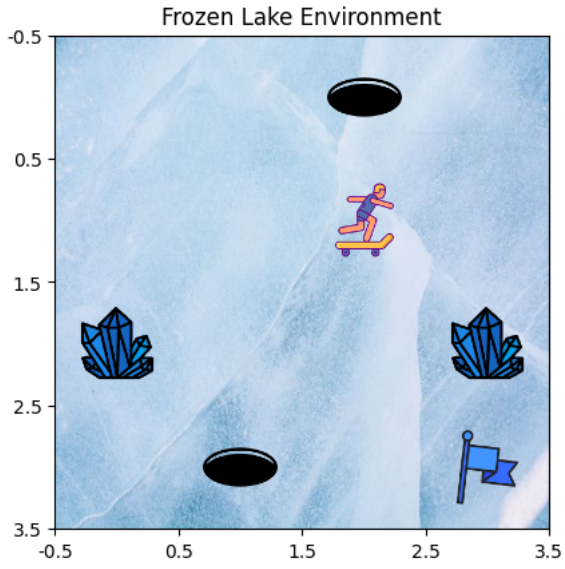
4) On Gems States: If the skater reaches the Gems he has won a lottery himself and then reach the goal with some gems.

- The locations where the gems are present is: [0, 2] and [3, 2]



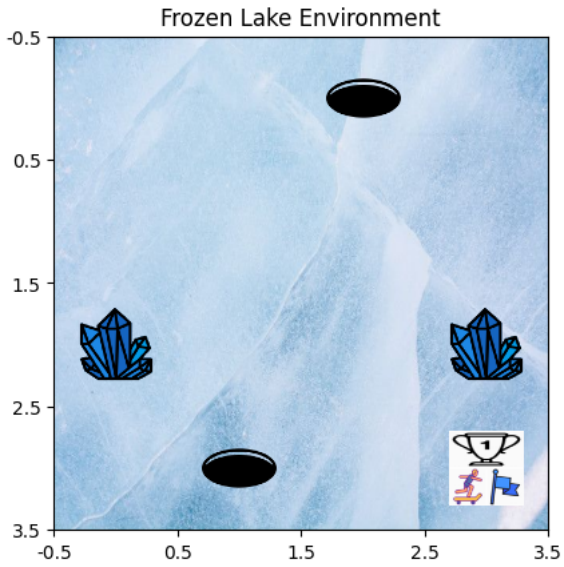
5) Intermediate State: Any other random location that the agent skates the lake grid is the intermediate state.

- Let he be present at [2, 1], then it is an intermediate state.



6) Goal State: The final goal state indicates the termination of the agents race as he has reached the goal.

- The goal state is [3,3] in the frozen grid:



- **Set of Actions** defined in our case:
  - We have 4 main actions which include: Left, Right, Up and Down.
  - We also ensure to keep the agent within the grid by clipping over the boundaries and restoring his location from where he tried to move out of the grid.
  - If the agent is in the same position as the previous step, choose a different action.
  - The following code handles the direction movement and clipping:

```
````````## ACTIONS SET #############
        if action == 0:   # moves right
            self.myskater[0] += 1
        elif action == 1:   # moves left
            self.myskater[0] -= 1
        elif action == 2:   # moves up
            self.myskater[1] += 1
        elif action == 3:   # moves down
            self.myskater[1] -= 1
        ##########################

        self.myskater = np.clip(self.myskater, 0, 3)

        # If the agent is in the same position as the previous step, choose a different act
ion
        prev_state_positions = np.argwhere(self.prev_state == 0.2)
        if len(prev_state_positions) > 0 and np.array_equal(self.myskater, prev_state_posit
ions[0]):
            while action == self.prev_action:
                action = self.action_space.sample()
            self.flag_out_grid = 1
```

- **Set of Rewards**
```

- Goal Flag Reached (Positive Reward):
  If the goal position is reached, the agent receives a reward of 10. This encourages the agent to reach the goal.
- Skating on Hole (Negative Rewards):
  If the agent skates on the first hole, it receives a penalty of -5. If the agent slides on the second hole, the agent receives a penalty of -6. These negative rewards discourage the agent from skating over holes as it will drown.
- Gem Collections (Positive Rewards):
  If the agent reaches the location of the first gem, the agent receives a reward of 5. If the agent reaches the location of the second gem, the agent receives a reward of 6. These positive rewards encourage the agent to collect gems and win a lottery along the way.
- Distance to Goal (Dynamic Rewards):
  If the current distance to the goal is less than the previous distance to the goal, the agent receives a reward of 1. This provides a positive reinforcement for moving closer to the goal.

  If the current distance to the goal is greater than the previous distance to the goal, the agent receives a penalty of -1. This penalizes the agent for moving away from the goal.

  If there is no change in distance, the agent receives a slight negative reward of -0.1. This also handles the case when he is trying to go out of the grid.
- The following snippet handles the rewards given to agent.

```
        ## REWARDS SET #############
    if np.array_equal(self.myskater, self.goal_loc):
        reward = 10  # Positive reward for reaching goal
    elif np.array_equal(self.myskater, self.hole_loc[0]):
        reward = -5  # negative reward for reaching holes 1
    elif np.array_equal(self.myskater, self.hole_loc[1]):
        reward = -6   # negative reward for reaching holes 2
    elif np.array_equal(self.myskater, self.gem_loc[0]):
        reward = 5   # positive reward for reaching gems 1
    elif np.array_equal(self.myskater, self.gem_loc[1]):
        reward = 6  # positive reward for reaching gems 2
    elif current_distance_to_goal < prev_distance_to_goal:
        reward = 1  # Positive reward for moving closer to goal
    elif current_distance_to_goal > prev_distance_to_goal:
        reward = -1  # Negative reward for moving away to goal
    else:
        reward = -0.1  # Slight negative reward for no change
    ############################
```

- **Main Objective:**
  - The primary objective is for the agent to learn a policy that maximizes the cumulative reward over time. The agent should navigate the grid, avoid obstacles (holes), reach the goal efficiently, and collect gems from two locations for winning a lottery. The dynamic rewards based on distance encourage the agent to find an optimal path to the goal.
  - In our environment, the maximum reward that can be achieved in a single episode is determined by the structure of the reward system.

    Reaching the Goal:

    - The agent receives a reward of 10 for reaching the goal position (`[3, 3]`).

    Collecting Gems:

- The agent receives rewards for collecting gems. Based on the reward structure, there are two gems with rewards 5 and 6, respectively.

Avoiding Holes:

- The agent should avoid holes, as touching them results in negative rewards (-5 and -6).

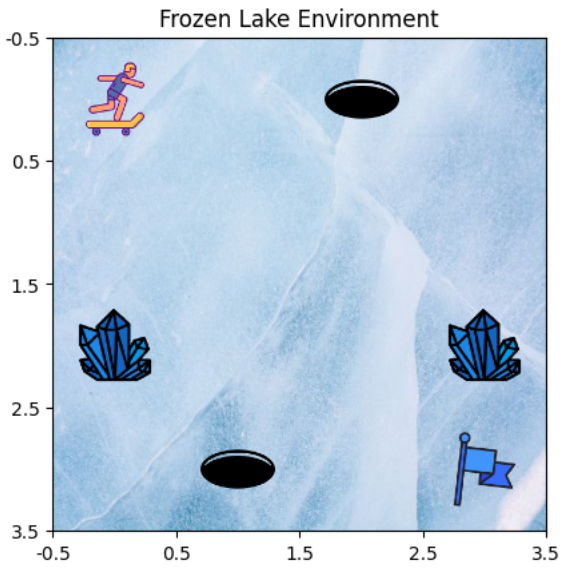
Moving Towards the Goal:

- Each step towards the goal could potentially earn a reward of 1. However, the exact reward depends on whether each step objectively decreases the distance to the goal.

Based on this, the maximum reward in a single episode would be achieved by collecting both gems and reaching the goal, while also making every move directly towards the goal.

2. **Provide visualization of your environment.**

- Initially we defined our grid as shown below:



3. **How did you define the stochastic environment?**

The stochasticity is introduced in the agent's actions. Instead of having a deterministic action, we defined a probability distribution over the possible actions. The list `randomness` represents the probability distribution for each action. In my case, the probabilities are assigned to each action `[0.35, 0.15, 0.35, 0.15]` favoring going right and top. This means that when the agent selects an action, it will follow this probability distribution. Note that the sum of probabilities is equal to 1 for a valid probability distribution. This approach allows to add randomness in the agent's decision-making process, making the environment stochastic.

4. **What is the difference between the deterministic and stochastic environments?**

In the deterministic `FrozenLakeEnv` environment, given a specific state and action, the next state and reward are uniquely determined. Each action directly leads to a specific outcome. In the `step` method of the environment, for a given action, there is a deterministic mapping to the next state. The skater's movement (up, down, left, or right) is determined without any randomness. We explicitly check the action and update the skater's position accordingly. There is no randomness introduced in the action selection process.

For the environment to be stochastic, we would introduce randomness in the action selection process. This means that, instead of the skater always taking the intended action, there is a probability distribution over actions, and the skater

samples an action based on that distribution, adding an element of uncertainty to the skater's behavior.

5. **Safety in AI: Write a brief review (~ 5 sentences) explaining how you ensure the safety of your environment. E.g. how do you ensure that the agent choose only actions that are allowed, that agent is navigating within defined state-space, etc**

There are several measures implemented to ensure the safety of the environment and make sure the agent behaviur is under control.

- Firstly agent actions are restricted to a 4 moves that are up, down, left and right so it cant make any other actions like jump up-left corner or anything. Here's the example code snipped used

```
          if action == 0:    # moves right
            self.myskater[0] += 1
        elif action == 1:   # moves left
            self.myskater[0] -= 1
        elif action == 2:   # moves up
            self.myskater[1] += 1
        elif action == 3:    # moves down
            self.myskater[1] -= 1
```

- Second, we also implemented state-space constraints to ensure that our agent moves inside the grid and if the agent moves out or an attempt is made it sets a flag to indicate that agent is attempting to move out of the grid and negative penalty occurs of -0.1 because if he attempts to move out, he is put back to the same place and negative reward of -0.1 is applied.

```
self.myskater = np.clip(self.myskater, 0, 3)

terminated = True if np.array_equal(self.myskater, self.goal_loc) else self.timestep >= self.m
ax_timestamp
        truncated = True if np.any((self.myskater < 0) | (self.myskater > 3)) else False
        if truncated:
            self.flag_out_grid = 1
```

- There's also a penalty for skating over holes, if it goes into hole 1, a `reward = -5` or hole 2 a `reward = -6` and this is handled in the `calculate_reward` method.

```
penalty = any(np.array_equal(self.myskater, pos) for pos in self.hole_loc)
        if penalty:
            self.penalty_counter += 1
```

- We are trying to reward him with positive points when the agent reaches gems.

```
elif np.array_equal(self.myskater, self.gem_loc[0]):
            reward = 5   # positive reward for reaching gems 1
        elif np.array_equal(self.myskater, self.gem_loc[1]):
            reward = 6  # positive reward for reaching gems 2
```

- We are giving him positive rewards if he is taking steps towards the goal by verifying the distance from current position to goal position.

```
# Calculating distance to goal before and after the step
        prev_distance_to_goal = np.linalg.norm(self.goal_loc - prev_myskaterition)
        current_distance_to_goal = np.linalg.norm(self.goal_loc - self.myskater)

elif current_distance_to_goal < prev_distance_to_goal:
            reward = 1  # Positive reward for moving closer to goal
        elif current_distance_to_goal > prev_distance_to_goal:
            reward = -1  # Negative reward for moving away to goal
```

- To ensure that the task is completed by the agent and appreciated with 10 points as he reaches the goal.

```
terminated = True if np.array_equal(self.myskater, self.goal_loc) else self.timestep >= self.m
ax_timestamp
```

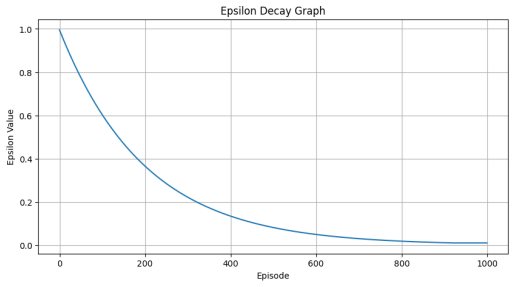
Part 2 [Total: 40 points] - Applying Tabular Methods

Deterministic Q Learning vs Stochastic Q Learning

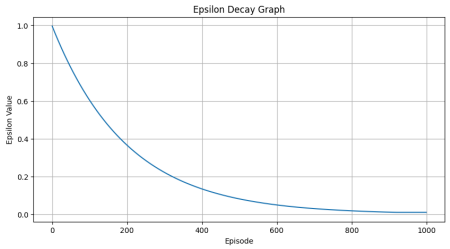
1. **Show and discuss the results after:**
   • Applying Q-learning to solve the deterministic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.
   • Applying Q-learning to solve the stochastic environment defined in Part 1. Plots should include epsilon decay and total reward per episode.
   • Provide the evaluation results. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.

Deterministic

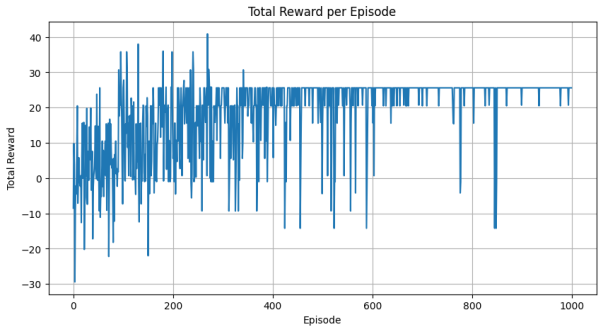
Epsilon Decay Comparison



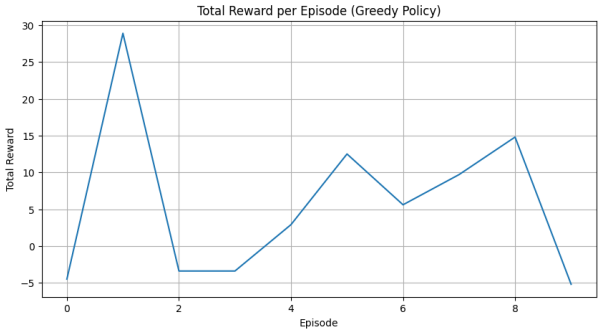
Stochastic



Total Rewards per Episode Comparison



Total Rewards for 10 Episodes Comparison



Discussion of comparison for Q Learning

As the stochasticity is added, the total rewards over time is not constant over 10 episodes. It fluctuates as the randomness is added. When it comes to rewards per episode both of the cases, it increases over the epochs however in the stochasticity even after 500+ episodes there is no guarantee that the agent will start receiving higher rewards, for the deterministic environment the rewards have gradually started to increase and the number almost remains constant towards the end of epochs. For the epsilon decay curve, both of the cases show a similar trend of exponential decrease.

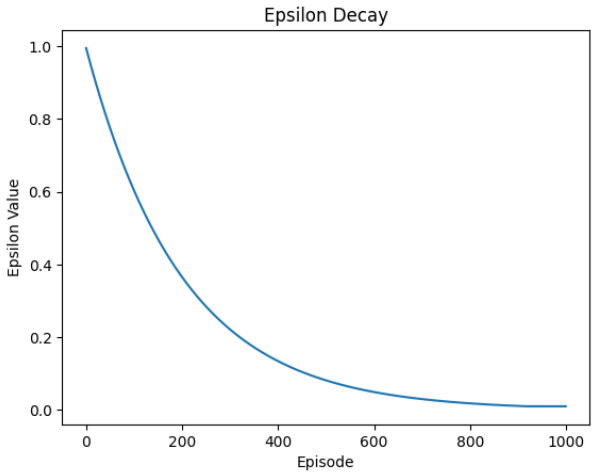
- **Applying any other algorithm of your choice to solve the deterministic environment defined in Part 1. Plots should include total reward per episode.**

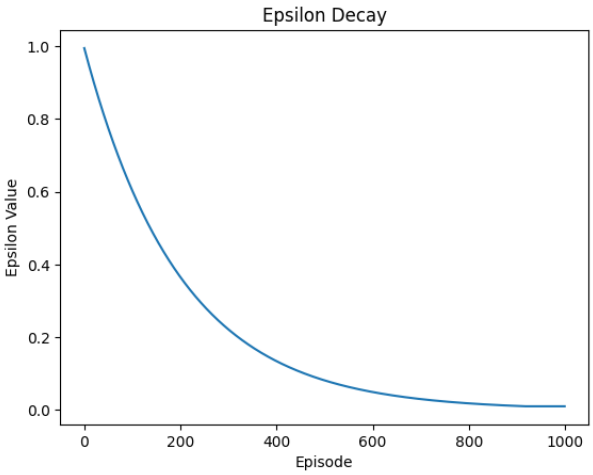
- **Applying any other algorithm of your choice to solve the stochastic environment defined in Part 1. Plots should include total reward per episode.**
- **Provide the evaluation results. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.**

Deterministic Double Q Learning vs Stochastic Double Q Learning

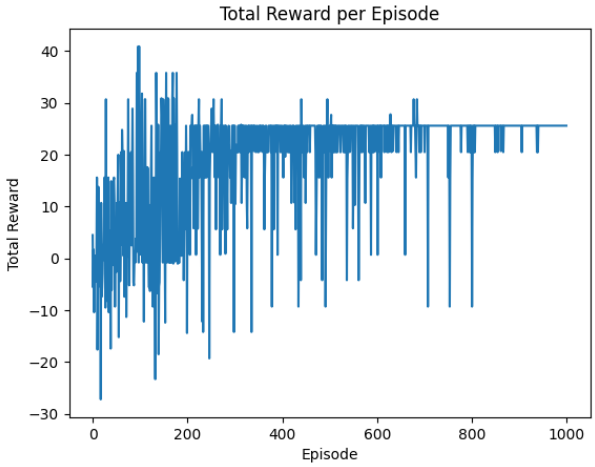
Deterministic **Stochastic**

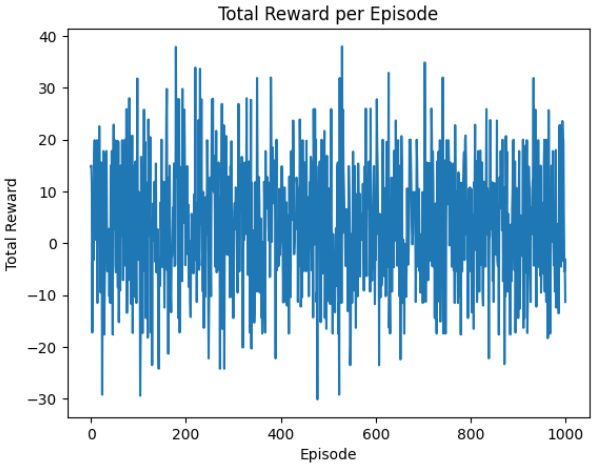
Epsilon Decay Comparison



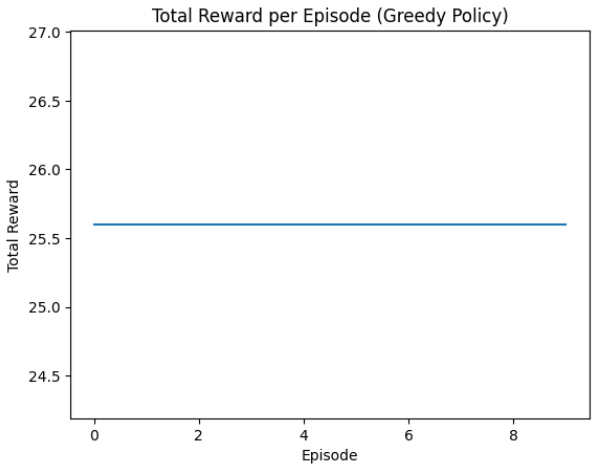


Total Rewards per Episode Comparison





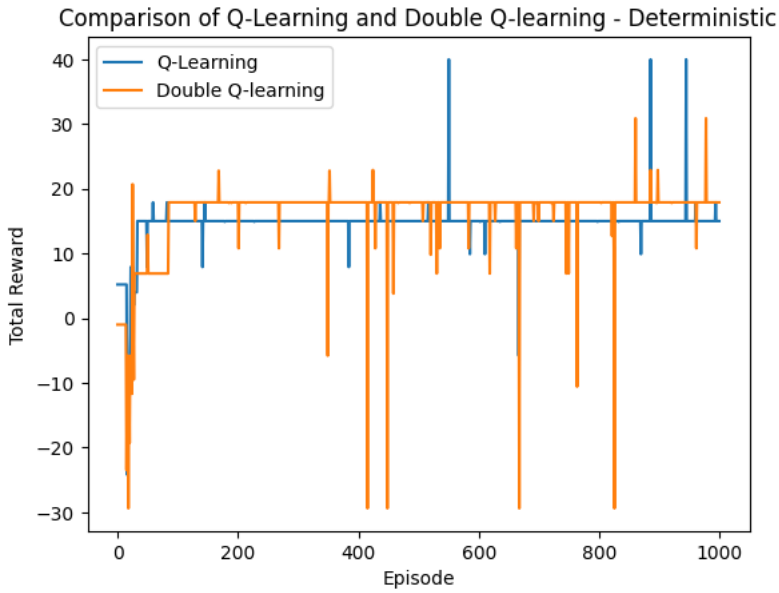
Total Rewards for 10 Episodes Comparison



Discussion of comparison for Double Q Learning

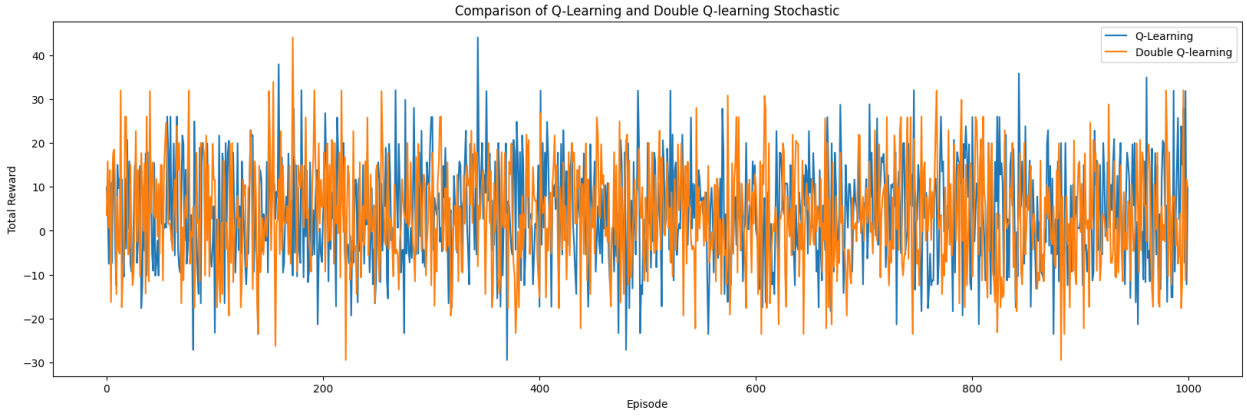
Similar to Q learning, for double Q learning as well there are very similar graphs like the above. As the stochasticity is added, the total rewards over time is not constant over 10 episodes. It fluctuates as the randomness is added. When it comes to rewards per episode both of the cases, it increases over the epochs however in the stochasticity even after 500+ episodes there is no guarantee that the agent will start receiving higher rewards, for the deterministic environment the rewards have gradually started to increase and the number almost remains constant towards the end of epochs. For the epsilon decay curve, both of the cases show a similar trend of exponential decrease.

2. **Compare the performance of both algorithms on the same deterministic environment (e.g. show one graph with two reward dynamics) and give your interpretation of the results.**



Comparing both the algorithms reward dynamics we can notice that both double Q learning and Q learning have performed very similar. The reward over the episodes is slightly higher in the double Q process for most of the time, however it is also negative more number of times compared to Q learning.

3. **Compare how both algorithms perform in the same stochastic environment (e.g. show one graph with two reward dynamics) and give your interpretation of the results.**



As the stochasticity is added, both the algorithms show that the rewards fluctuate more than the deterministic settings. As the name stochastic, it has higher randomness ranging all the way to -50 to +50 rewards. There is no constant reward for even a small number of continuous epochs.

4. **Briefly explain the tabular methods, including Q-learning, that were used to solve the problems. Provide their update functions and key features.**

Q - Learning

- **Update Function:**

The Q-value update function in Q-learning is given by the following equation:

$[Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_a Q(s',a))]$

  - $(Q(s,a))$ represents the Q-value for taking action (a) in state (s).
  - (α) is the learning rate, controlling the weight of new information.
  - (r) is the immediate reward obtained after taking action (a) in state (s).
  - (γ) is the discount factor, determining the importance of future rewards.
  - (s') is the next state after taking action (a) in state (s).
  - $(\max_a Q(s',a))$ is the maximum Q-value for the next state (s').

- **Key Features:**
  - Q-learning balances exploration and exploitation. It is a model-free approach, meaning it doesn't require knowledge of the environment's dynamics. It learns directly from experiences. Under certain conditions, Q-learning is guaranteed to converge to the optimal Q-values. It is an off-policy algorithm, meaning it learns from experiences generated by a different policy than the one being improved. It maintains a table of Q-values for each state-action pair, making it suitable for problems with a manageable number of discrete states and actions.

Double Q-Learning

- **Update Function:**

The update function for Double Q-learning involves two sets of Q-values, \(Q_1\) and \(Q_2\), and alternates between them during updates. The update equation is as follows:

$[Q_1(s,a) \leftarrow (1-\alpha) \cdot Q_1(s,a) + \alpha \cdot (r + \gamma \cdot Q_2\left(s', \text{argmax}_{a'} Q_1(s',a')\right))]$

$[Q_2(s,a) \leftarrow (1-\alpha) \cdot Q_2(s,a) + \alpha \cdot (r + \gamma \cdot Q_1\left(s', \text{argmax}_{a'} Q_2(s',a')\right))]$

  - $(Q_1(s,a))$ and $(Q_2(s,a))$ are two sets of Q-values for taking action (a) in state (s).
  - (α) is the learning rate.
  - (r) is the immediate reward obtained after taking action (a) in state (s).
  - $(\gamma)$is the discount factor.
  - (s') is the next state after taking action (a) in state (s).

- **Key Features:**
  - Double Q-learning helps reduce the overestimation bias present in traditional Q-learning by using two sets of Q-values. The algorithm alternates between the two sets of Q-values during updates, reducing the correlation between the action selection and evaluation. Double Q-learning tends to provide more accurate estimates of the true Q-values, leading to improved learning and decision-making in certain scenarios. Its principles have also been extended to deep reinforcement learning settings, where neural networks are used to represent Q-values.

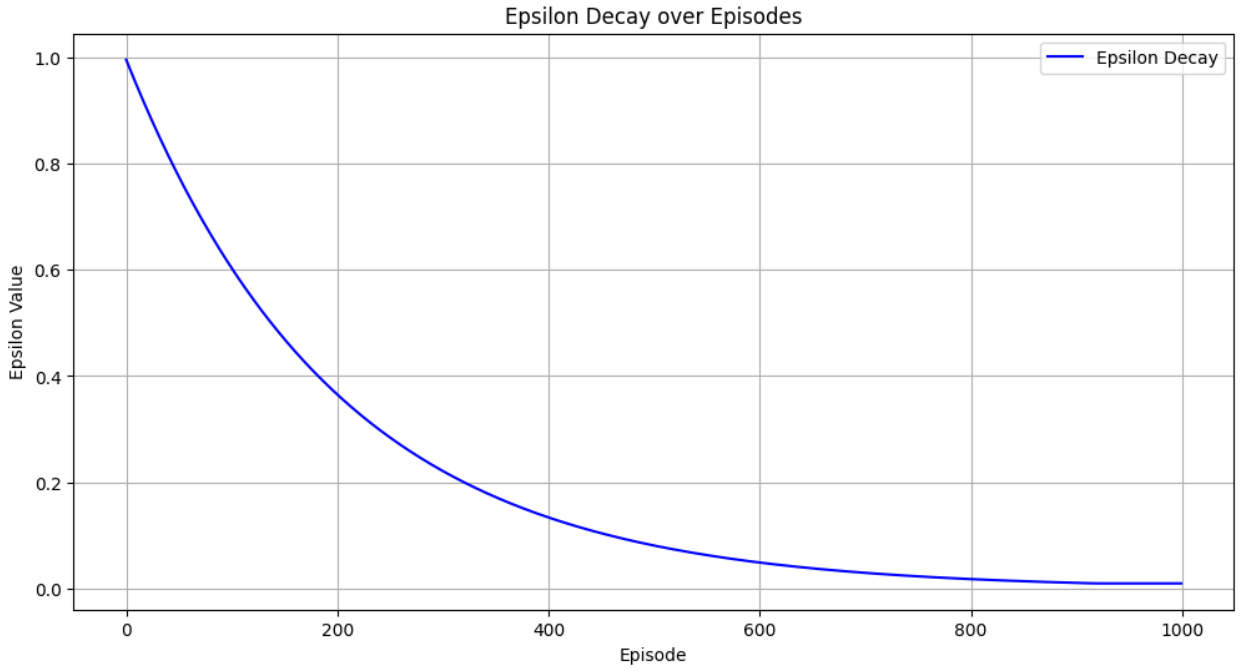
5. **Briefly explain the criteria for a good reward function. If you tried multiple reward functions, give your interpretation of the results.**

- The reward function should be aligned with the desired objectives of the task. It should encourage the agent to take actions that lead to successful task completion. We ensure that as the process has to terminate with maximum rewards, we have added 10 points upon reaching the goal state.
- It's often an iterative process of refining the reward function based on observations and insights gained during experimentation, analyzing the agent's performance under each reward setting, the agent has showcased having negative rewards at the end of the episode, when it always reached the hole locations. However, the number of times it reached the hole gradually decreased in the deterministic setting when it also reached the positive rewards constantly. However in the stochastic environment, the agent received negative rewards even after several epochs.

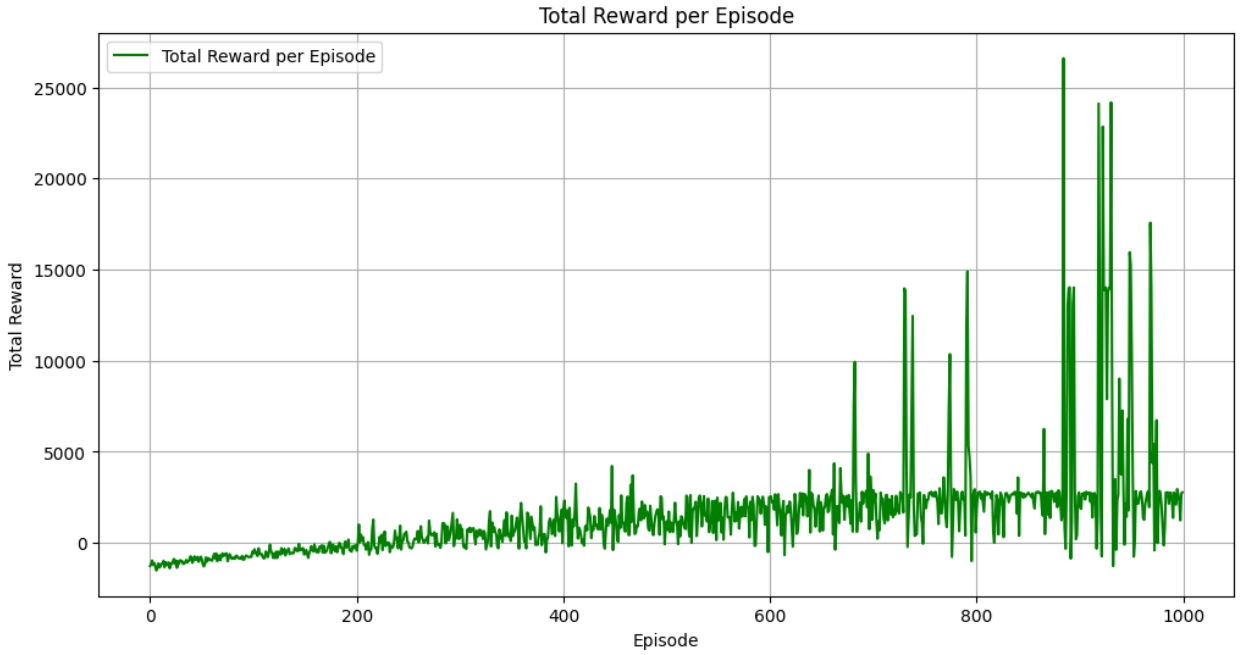
Part 3 [Total: 30 points] - Solve Stock Trading Environment

1. **Show and discuss the results after applying the Q-learning algorithm to solve the stock trading problem. Plots should include epsilon decay and total reward per episode.**

Epsilon Decay



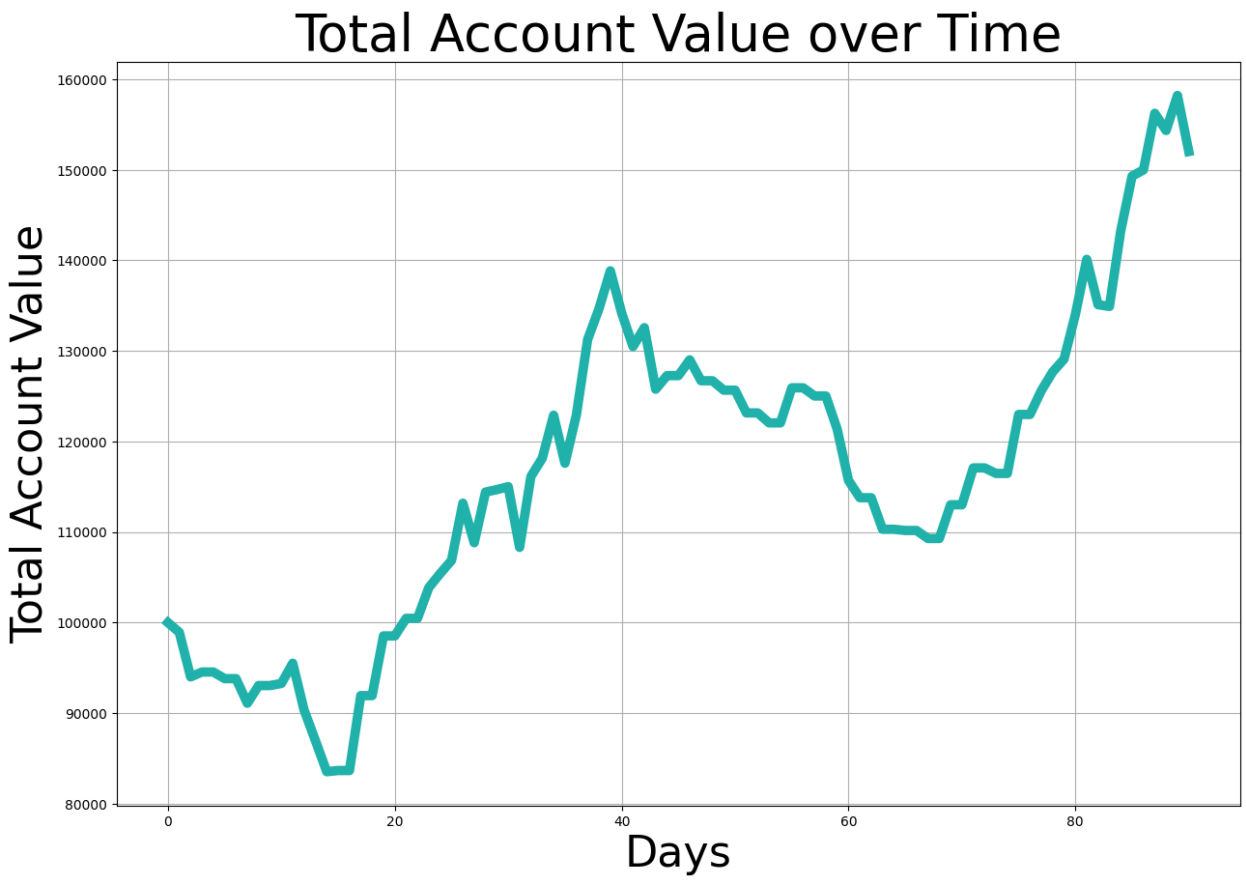
Total Rewards per Episode



Discussion

- Since the epsilon decay curve is smooth and gradual, it suggests that the exploration-exploitation balance is well-tuned and the agent is learning effectively. Even when epsilon reaches its minimum value, there is still a small chance of the agent choosing a random action. This helps to prevent the agent from getting stuck in local optima and ensures continued exploration, which is important for adapting to changes in the market.
- There is a slight improvement in the total number of rewards as we go towards higher number of episodes. Highly fluctuating after 700 episodes. This upward trend in the total reward per episode suggests that the Q-learning agent is learning and improving its performance over time. these fluctuations in the reward is to be expected in the complex and stochastic environment of the stock market. From episode 200 to 600, the reward starts to increase more consistently. After episode 600, the reward seems to fluctuate around a certain level, with occasional spikes. This suggests that the agent has found a strategy that is profitable on average, but it is still exploring and occasionally finding better opportunities.

2. **Provide the evaluation results. Evaluate your trained agent's performance (you will have to set the train parameter set to False), by only choosing greedy actions from the learnt policy. Plot should include the agent's account value over time. Code for generating this plot is provided in the environment's render method. Just call environment.render after termination.**



Discussion

- The `stock_trading_environment.train = False` line sets the environment to evaluation mode, meaning the agent won't learn from its experiences during this run.
- There is an upward trend in the total account value over time suggests the agent is making profitable trades on average.
- There are fluctuations throughout which is expected in the dynamic and stochastic nature of the stock market.
- Sudden spikes could represent particularly successful trades or periods of favorable market conditions.
- Even good trading strategies can have setbacks so there are dips or plateaus in the plot indicating periods of losses or corrections.

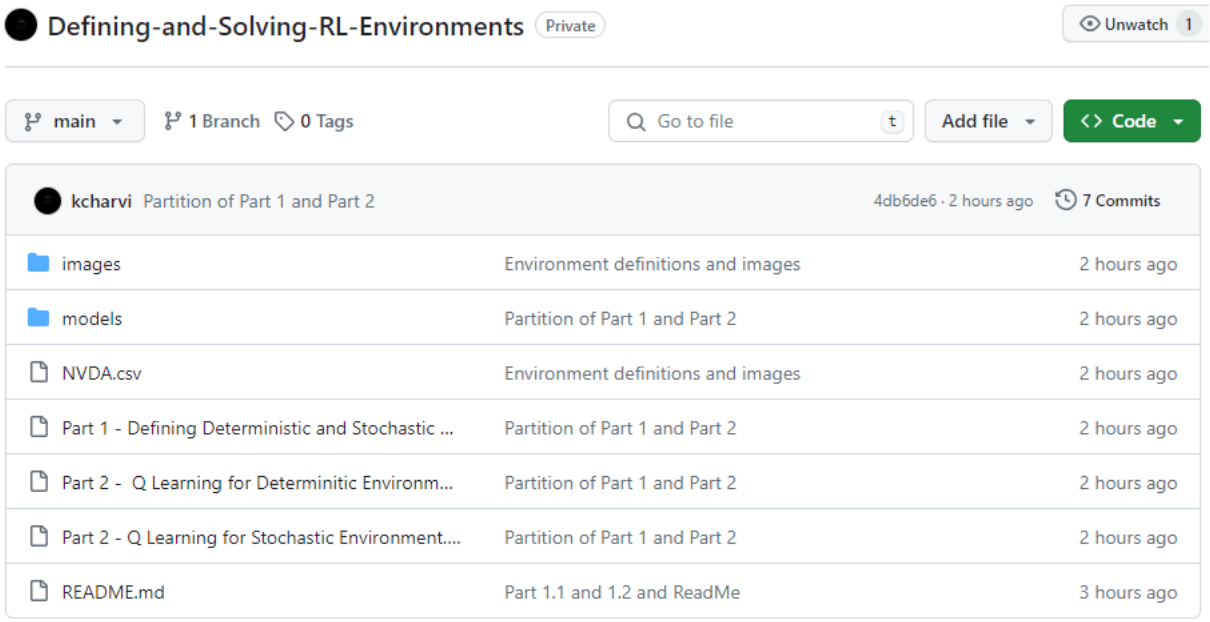
Extra Points [max +10 points]

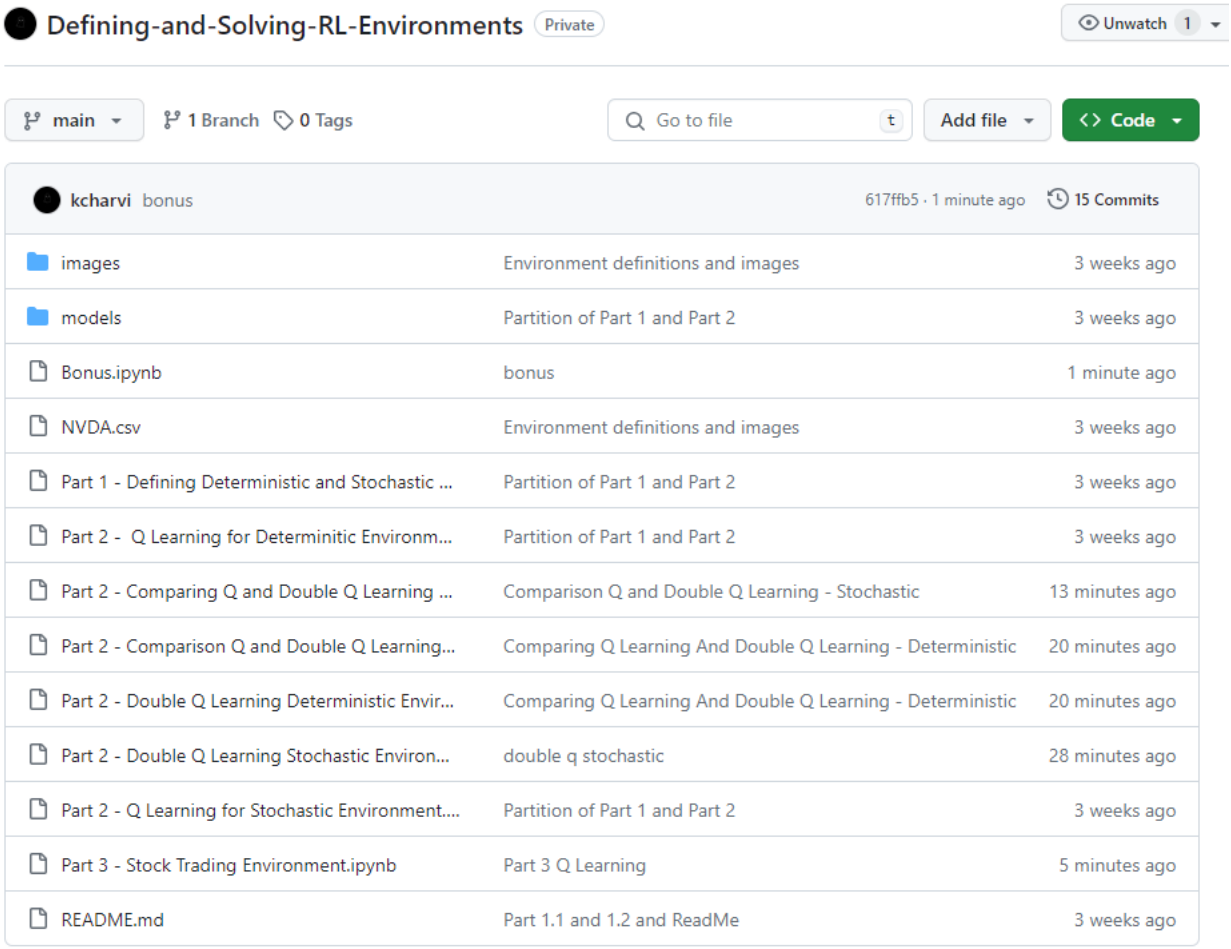
Git Expert [2 points]

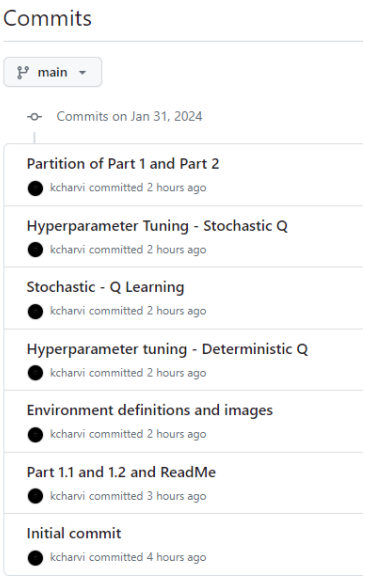
In your report include a link to your private GitHub project and a snapshot of your commits history. Reports without the link and snapshot provided, will not be considered for this bonus.

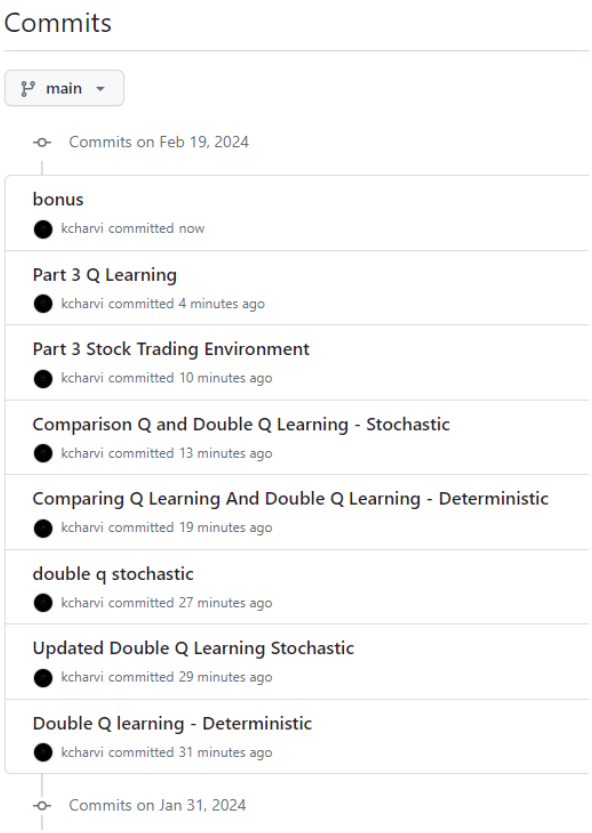
Github Repo: https://github.com/kcharvi/Defining-and-Solving-RL-Environments

Snapshot of Commits


Defining-and-Solving-RL-Environments
Private
Unwatch 1
main
1 Branch
0 Tags
Go to file
t
Add file
Code
kcharvi Partition of Part 1 and Part 2
4db6de6 · 2 hours ago
7 Commits
images Environment definitions and images 2 hours ago
models Partition of Part 1 and Part 2 2 hours ago
NVDA.csv Environment definitions and images 2 hours ago
Part 1 - Defining Deterministic and Stochastic ... Partition of Part 1 and Part 2 2 hours ago
Part 2 - Q Learning for Determinitic Environm... Partition of Part 1 and Part 2 2 hours ago
Part 2 - Q Learning for Stochastic Environment.... Partition of Part 1 and Part 2 2 hours ago
README.md Part 1.1 and 1.2 and ReadMe 3 hours ago


Defining-and-Solving-RL-Environments
Private
Unwatch 1
main
1 Branch
0 Tags
Go to file
t
Add file
Code
kcharvi bonus
617ffb5 · 1 minute ago
15 Commits
images Environment definitions and images 3 weeks ago
models Partition of Part 1 and Part 2 3 weeks ago
Bonus.ipynb bonus 1 minute ago
NVDA.csv Environment definitions and images 3 weeks ago
Part 1 - Defining Deterministic and Stochastic ... Partition of Part 1 and Part 2 3 weeks ago
Part 2 - Q Learning for Determinitic Environm... Partition of Part 1 and Part 2 3 weeks ago
Part 2 - Comparing Q and Double Q Learning ... Comparison Q and Double Q Learning - Stochastic 13 minutes ago
Part 2 - Comparison Q and Double Q Learning... Comparing Q Learning And Double Q Learning - Deterministic 20 minutes ago
Part 2 - Double Q Learning Deterministic Envir... Comparing Q Learning And Double Q Learning - Deterministic 20 minutes ago
Part 2 - Double Q Learning Stochastic Environ... double q stochastic 28 minutes ago
Part 2 - Q Learning for Stochastic Environment.... Partition of Part 1 and Part 2 3 weeks ago
Part 3 - Stock Trading Environment.ipynb Part 3 Q Learning 5 minutes ago
README.md Part 1.1 and 1.2 and ReadMe 3 weeks ago

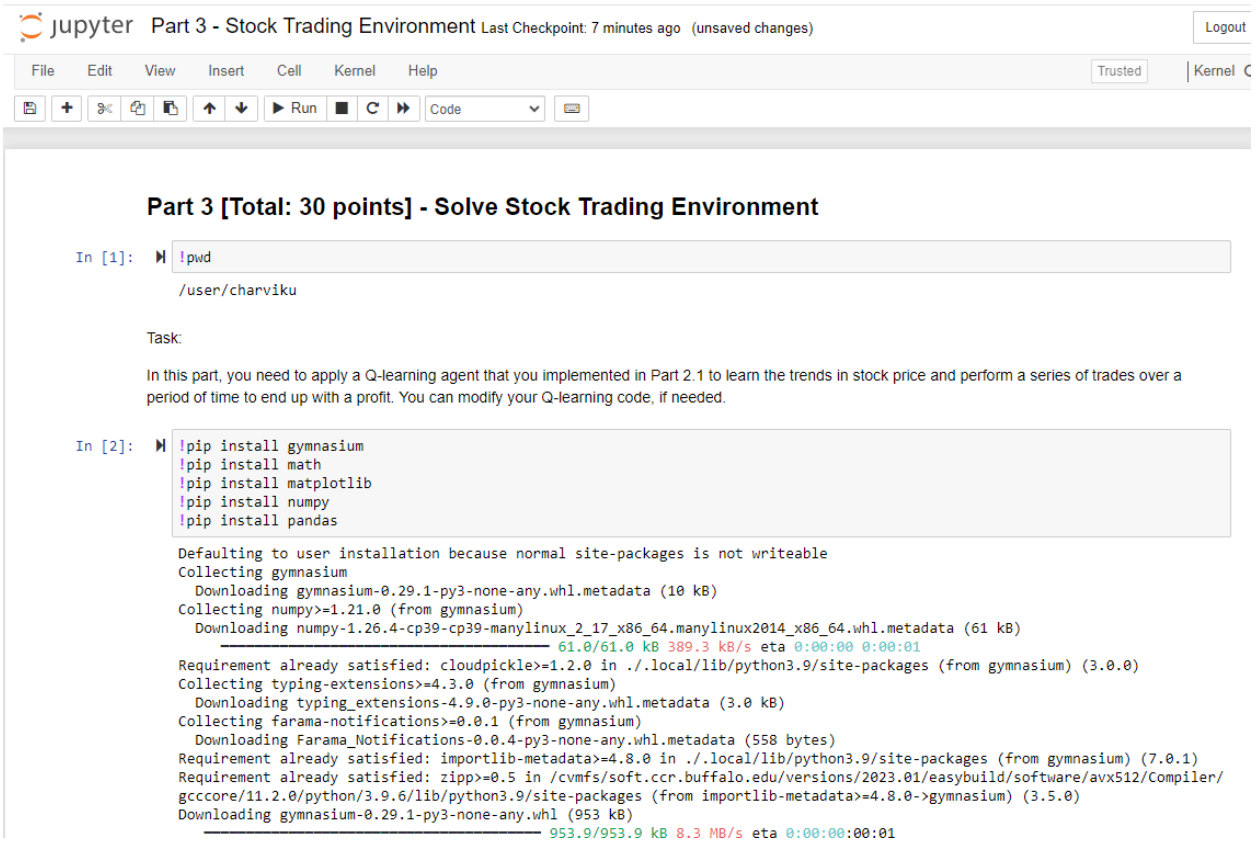




CCR Submission [3 points]

Submit a screenshot of successful execution of your code on CCR. Depending on which method you are using to access CCR resources, requirements for proof of successful use of CCR are as follows:

- Jupyter Notebook Session:-
  - Since there is no terminal here, print your working directory (!pwd) showing that you are in your own home folder. It should have your ubitname somewhere in the path.
  - Send a screenshot of the entire screen showing the output of this command and successful execution of your code (just the final results will suffice).


jupyter Part 3 - Stock Trading Environment Last Checkpoint: 7 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Kernel
Run
Code
Part 3 [Total: 30 points] - Solve Stock Trading Environment
In [1]:
!pwd
/user/charviku
Task:
In this part, you need to apply a Q-learning agent that you implemented in Part 2.1 to learn the trends in stock price and perform a series of trades over a period of time to end up with a profit. You can modify your Q-learning code, if needed.
In [2]:
!pip install gymnasium
!pip install math
!pip install matplotlib
!pip install numpy
!pip install pandas
Defaulting to user installation because normal site-packages is not writeable
Collecting gymnasium
Downloading gymnasium-0.29.1-py3-none-any.whl.metadata (10 kB)
Collecting numpy>=1.21.0 (from gymnasium)
Downloading numpy-1.26.4-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (61 kB)
61.0/61.0 kB 389.3 kB/s eta 0:00:00 0:00:01
Requirement already satisfied: cloudpickle>=1.2.0 in ./.local/lib/python3.9/site-packages (from gymnasium) (3.0.0)
Collecting typing-extensions>=4.3.0 (from gymnasium)
Downloading typing_extensions-4.9.0-py3-none-any.whl.metadata (3.0 kB)
Collecting farama-notifications>=0.0.1 (from gymnasium)
Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Requirement already satisfied: importlib-metadata>=4.8.0 in ./.local/lib/python3.9/site-packages (from gymnasium) (7.0.1)
Requirement already satisfied: zipp>=0.5 in /cvmfs/soft.ccr.buffalo.edu/versions/2023.01/easybuild/software/avx512/Compiler/gcccore/11.2.0/python/3.9.6/lib/python3.9/site-packages (from importlib-metadata>=4.8.0->gymnasium) (3.5.0)
Downloading gymnasium-0.29.1-py3-none-any.whl (953 kB)
953.9/953.9 kB 8.3 MB/s eta 0:00:00:00:01

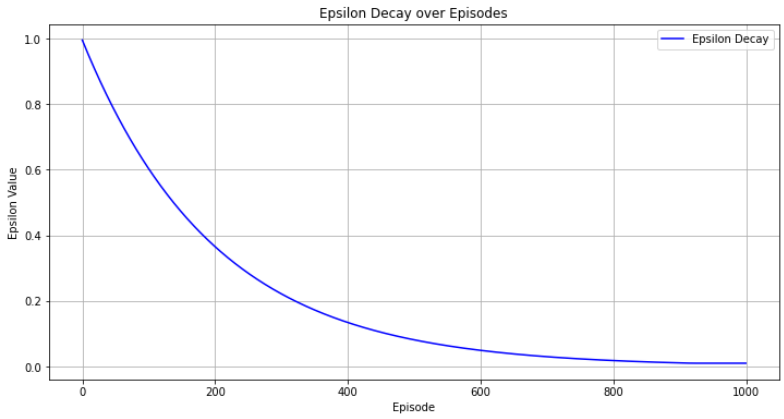
jupyter Part 3 - Stock Trading Environment Last Checkpoint: 8 minutes ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Help Trusted Kernel

Run Code

1. Show and discuss the results after applying the Q-learning algorithm to solve the stock trading problem. Plots should include epsilon decay and total reward per episode.
2. Provide the evaluation results. Evaluate your trained agent's performance (you will have to set the train parameter set to False), by only choosing greedy actions from the learnt policy. Plot should include the agent's account value over time. Code for generating this plot is provided in the environment's render method. Just call environment.render after termination.

In [8]:
```
# Plot epsilon decay
plt.figure(figsize=(12, 6))
plt.plot(epsilon_values, label='Epsilon Decay', color='blue')
plt.xlabel('Episode')
plt.ylabel('Epsilon Value')
plt.title('Epsilon Decay over Episodes')
plt.legend()
plt.grid()
plt.show()
```

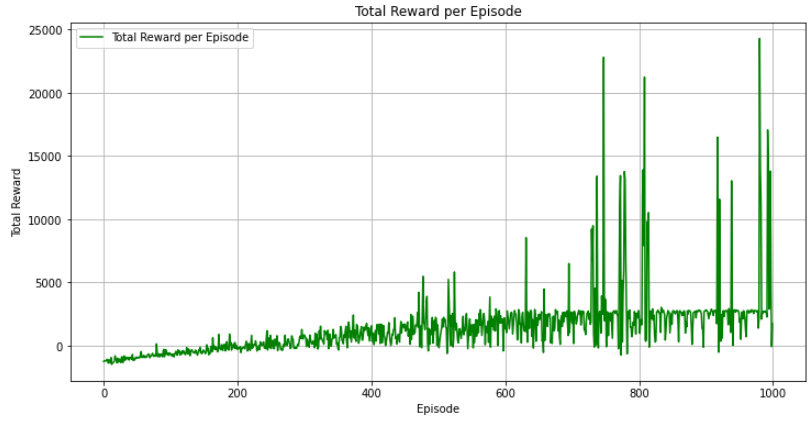


jupyter Part 3 - Stock Trading Environment Last Checkpoint: 8 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Help Trusted Kernel

Run Code

In [9]:
```
# Plot total reward per episode
plt.figure(figsize=(12, 6))
plt.plot(rewards_epi, label='Total Reward per Episode', color='green')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Total Reward per Episode')
plt.legend()
plt.grid()
plt.show()
```



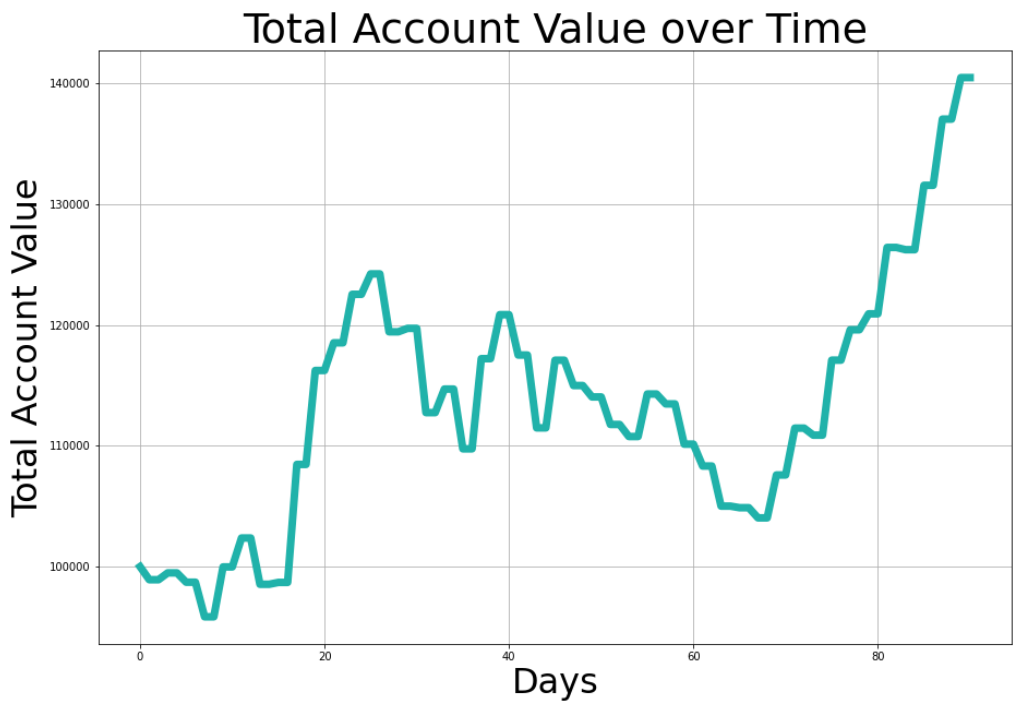
jupyter Part 3 - Stock Trading Environment Last Checkpoint: 9 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help Trusted Kernel

Run Code

```
while True:
    action = np.argmax(qt[state_index])
    next_state, reward, terminated, truncated, _ = stock_trading_environment.step(action)
    next_strt_idx = next_state
    state_index = next_strt_idx
    total_account_value_list_eval.append(stock_trading_environment.total_account_value)
    if terminated or truncated:
        stock_trading_environment.render()  # Call render to generate the plot
        break
```



Grid-World Scenario Visualization [5 points]

Your grid has to contain different images to represent:

- Agent - At least two images dependent on what the agent is doing.
- Appropriate background for your scenario (Not the default one)
- Images representing each object in your scenario.

```
def render(self):
        fig, ax = plt.subplots()
        plt.title('Frozen Lake Environment')

        # Load and display the background image
        background_img = plt.imread('frozen_lake.jpg')
        ax.imshow(background_img, extent=(-0.5, 3.5, -0.5, 3.5), origin='upper')


        skater_img = plt.imread('icons8-skateboard-100.png')
```

```
        hole_img = plt.imread('icons8-hole-100.png')
        gem_img = plt.imread('icons8-gems-100.png')
        goal_img = plt.imread('icons8-flag-100.png')
        skater_hole_drown_img = plt.imread('agent_hole_drown.png')
        skater_gem_lottery_img = plt.imread('agent_gems_lottery.png')
        agent_flag_winner_img = plt.imread('agent_flag_winner.png')
        agent_grid_cross_img = plt.imread('agent_grid_cross.png')

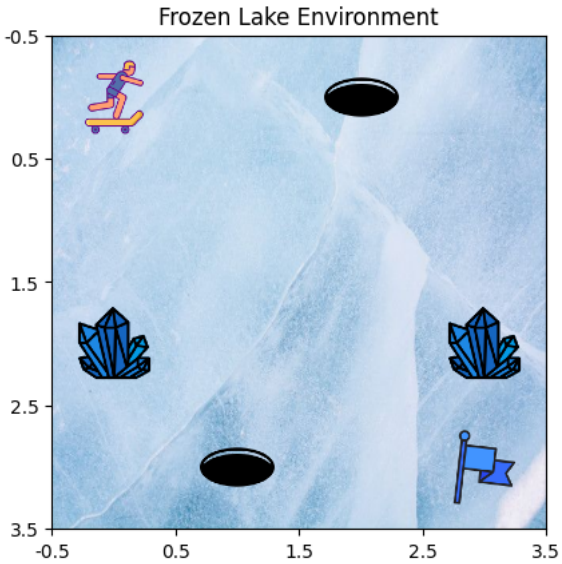
        # Plot Skater
        myskater = self.myskater
        if self.flag_out_grid:
            skater_img = agent_grid_cross_img
        agent_box = AnnotationBbox(OffsetImage(skater_img, zoom=0.4), myskater, frameon=False)
        ax.add_artist(agent_box)

        # Plot Holes
        for hole_loc in self.hole_loc:
            hole_loc = hole_loc
            if np.array_equal(self.myskater, hole_loc):
                hole_img = skater_hole_drown_img
            else:
                hole_img = plt.imread('icons8-hole-100.png')
            rock_box = AnnotationBbox(OffsetImage(hole_img, zoom=0.4), hole_loc, frameon=Fals
e)
            ax.add_artist(rock_box)

        # Plot Gems
        for gem_loc in self.gem_loc:
            gem_loc = gem_loc
            if np.array_equal(self.myskater, gem_loc):
                gem_img = skater_gem_lottery_img
            else:
                gem_img = plt.imread('icons8-gems-100.png')
            battery_box = AnnotationBbox(OffsetImage(gem_img, zoom=0.4), gem_loc, frameon=Fals
e)
            ax.add_artist(battery_box)

        # Plot goal
        goal_loc = self.goal_loc
        goal_loc = self.goal_loc
        if np.array_equal(self.myskater, goal_loc):
            goal_img = agent_flag_winner_img
        else:
            goal_img = plt.imread('icons8-flag-100.png')
        goal_box = AnnotationBbox(OffsetImage(goal_img, zoom=0.4), goal_loc, frameon=False)
        ax.add_artist(goal_box)

        plt.xticks(np.arange(-0.5, 4.5, 1))
        plt.yticks(np.arange(-0.5, 4.5, 1))
        plt.gca().set_xticklabels(np.arange(-0.5, 4.5, 1))
        plt.gca().set_yticklabels(np.arange(-0.5, 4.5, 1))
        plt.show()
```



References

- Part I and II is based on Charvi Kusuma Assignment 3 submission for CSE 574, Fall 2023.

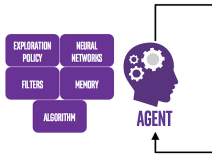
Build Smarter AI, Faster: A Beginner's Guide to Reinforcement Learning (Part 1)

AI learns by doing: Dive into Reinforcement Learning with MDP, SARSA, & Q-Learning.🤖

https://medium.com/predict/build-smarter-ai-faster-a-beginners-guide-to-reinforcement-learning-part-1-1ff1e0b2c173

https://medium.com/ai³-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf

What is Reinforcement Learning and how is it used? Find out in 5 minutes!



https://medium.com/ai³-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf