RL Computer Assignment 4

Max Brehmer & Florian John

March 19, 2024

1 Introduction

In this report, we implement Q-learning on a MDP gridworld task. We present plots for different parameter values and analyze their impact on model performance while we explain how we implemented our model. We also discuss the results of different parameter tests and descussfuture improvements for our model. An attempt at visualizing our environment and how q-values and rewards can be interpreted.

1.1 Task description

We implement MDP on a gridworld environment where x and y coordinates represent different states and agents can perform 5 different tasks which include waiting, going north, east, south and westo on the grid. The goal is to reach the goal which is positioned in the gridworld. The starting position for the agent is always randomly selected.

1.2 Grid world states (distance to goal)

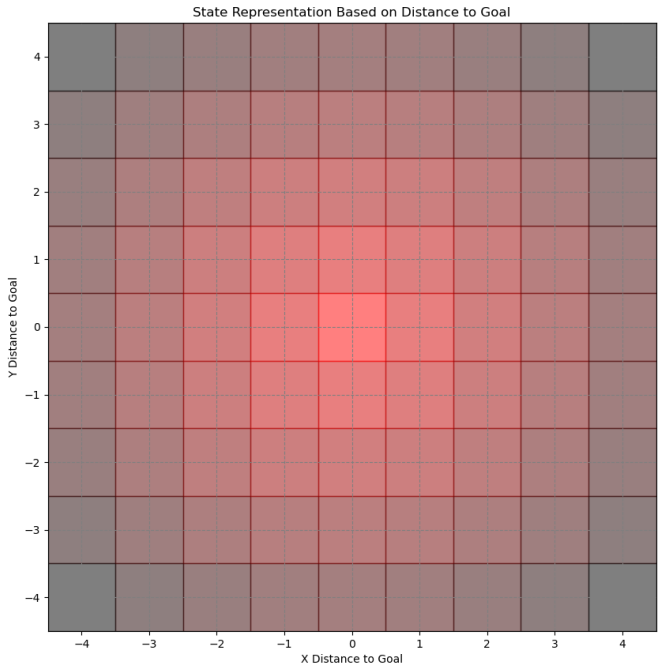

Figure 1: Here we see an illustration of the Q-table in which we store the Q-values relating to each tile.

We expect the algorithm to optimize toward granting higher Q-values (expected rewards) when one is fewer tiles from the position of the goal. In our code, we have actually mapped the negative values to positive values starting at the width or height of the environment. This is done so that we can use the indexes of our Q-table as the state values mapping directly to the position of the agent.

The following function is used to map the values:

```
state = env.distance_to_goal[agent] # Current state
                if state[0] < 0:
                    state[0] = (env.width-1) + np.abs(state[0])
                if state[1] < 0:
                    state[1] = (env.height-1) + np.abs(state[1])
```

Listing 1: Map negative state values to positive values

2 Background

2.1 Markov Decision Process

A Markov Decision Process (MDP) is a type of stochastic control process in discrete time, commonly used in Reinforcement Learning problems. An MDP consists of an Environment in which there are $s \in \mathcal{S}$ states and possible actions $a \in \mathcal{A}$. For each state-action pair (s, a) there is a reward process associated with it. The goal of an MDP is to maximize the return G, which is a function of the rewards r.

As our project is uses multiple agents we must consider the parameters:

- Number of agents k.
- Joint state $\vec{s} \in \vec{\mathcal{S}}$, consisting of states s_t^i at time t for agent i.
- Joint action $\vec{a} \in \vec{\mathcal{A}}(\vec{s})$, consisting of actions a_t^i at time t for agent i.
- Immediate reward $\vec{R}(s', \vec{a}, s)$. A vector consisting of rewards r_t^i at time t for agent i based on state transition $(s \to s')$ and action taken.
- Joint policy $\vec{\pi}$ consisting of policies for each agent.
- Transition probabilities $p(s', a|s, a)$ for transitioning from state $s \to s'$ using action a.

We can model this task as an MDP where the environment is an $(n \times m)$ grid-world with each discrete coordinate representing a position in the environment. The joint state $\vec{s}$ takes into account the position of each agent in each time step. As each agent $i = 1, ..., k$ find itself in state s^i, the action taken can be chosen from the action set $\mathcal{A}^i(s)$. At most these actions include moving (*east*, *west*, *north*, *south*), but limitations can appear in case an agent is adjacent to a wall. There are three different rewards involved. One (large) positive reward r_{win} for reaching the target coordinate, which results in a terminal state s_T^i. The other rewards are negative, one a small negative reward for each time step an agent spends in a non-terminal state and a larger punishment for attempting to move to the same coordinate as another agent (crash). This does not cause the agents to terminate, but rather stay in their current position.

2.2 Method background

Based on the description of the MDP, we have concluded that Independent Q-learning is an intuitive and easy to implement algorithm which gets the job done. However there are significant limitations in treating a multi-agent task as simply a dynamic environment.

In order to better take into account the joint state-actions, i.e. the states and actions of other agents, an adaptation of the base Q-learning algorithm would be a logical next step. One such algorithm with multi-agent cooperation is CQ-learning, an adapted form of Q-learning.

2.2.1 Independent Q-learning

When treating each agent individually, meaning no information is transferred between agents, the process becomes relatively simple as each agents' state only needs to take into account the distance to it's own goal for the given agent.

The independent Q-learning algorithm is one of the most fundamental RL algorithms, and is presented as the following:

$$Q(s^i, a^i) \leftarrow (1-\alpha)Q(s^i, a^i) + \alpha\left[r(s^i, a^i) + \gamma \max_{a'^i} Q(s'^i, a'^i)\right].$$

The problem with using independent Q-learning is that each agent is "blind" to each other, meaning that they have no way of knowing where to expect a collision. One might expect a convergence toward a very inefficient path where the agents take safe but slow paths toward the goals, but to get a more efficient algorithm we consider CQ-learning to be a more efficient method.

2.2.2 CQ-learning

CQ-learning, short for Coordinating Q-Learning is based on independent Q-learning for each agent with the adaptation that in certain marked states, agents cooperate to find the optimal joint action. For each state, we assume that each agent has learnt the optimal policy and thus the expected rewards for each agent in any given state can be computed following this policy. This works as expected when no interactions between the agents occur. However, when there are collisions between two or more agents, the observed reward is lower than the expected reward in the given state-action pair. In this case the algorithm marks the state-action pair as *dangerous*, for both agents. A student's t-test is used to determine whether a change in received immediate rewards is significant enough to explore the joint state space and thus determine the joint state-actions that result in a collision.

When encountering one of these marked states an observation is made in order to determine whether the joint state encourages cooperation, i.e there are actions that lead to collisions. The update rule used in this case is

$$Q^i_{s=j}(\vec{s}, a^i) \leftarrow (1-\alpha)Q^i_{s=j}(\vec{s}, a^i) + \alpha\left[r(\vec{s}, a^i) + \gamma \max_{a'^i} Q(\vec{s'}, a'^i)\right].$$

where Q^i is the vector of Q-values for agent i containing independent states, while $Q^i_{s=j}$ is the Q-values using the joint states. The Q-values of the joint states are constructed by bootstrapping the Q-values for the independent states.

3 Results

In this section we present comparisons of different parameter values and how they perform in regards to mean rewards, as well as the results and Q-values our model converges towards.

3.1 Optimizing for alpha

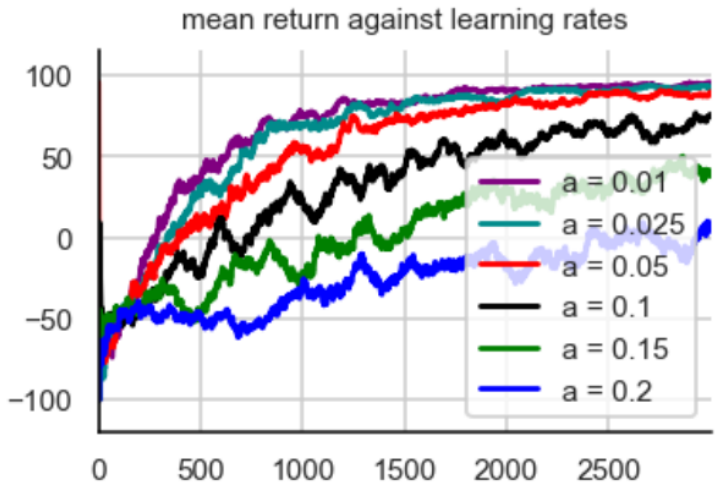

Figure 2: We now plot moving mean for previous 100 episodes, and we plot this for 3000 episodes.

In this figure we observe a higher moving mean reward for different α values. We observe that a lower choice for α is preferred.

3.2 Optimizing for gamma

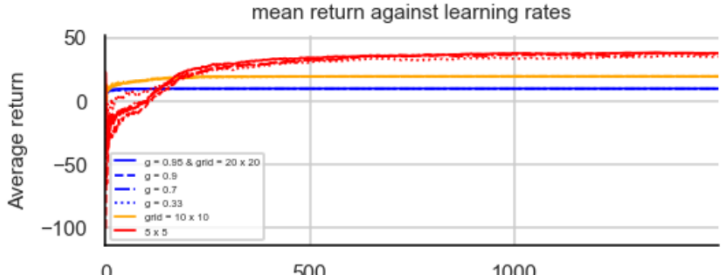

Figure 3: Now for different γ discount factors we observe moving mean for different grid sizes for 1500 episodes.

We observe that for the different values of γ we observe no significant difference for each grid size. To note, the red values (5x5) have a higher avg reward value only due to minimal total penalty for steps is lower for smaller grids.

3.3 Moving average for optimal model

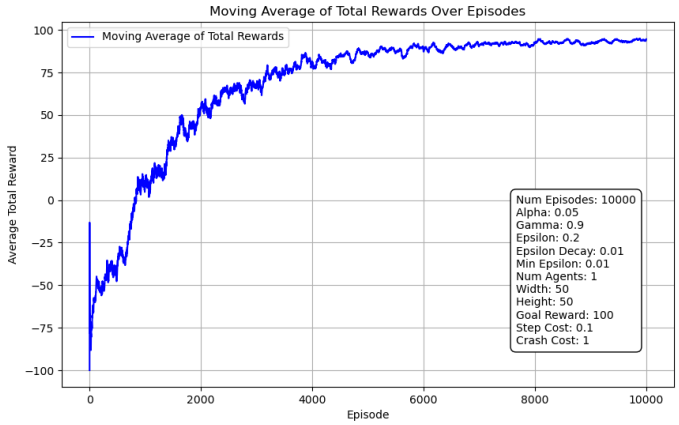

Figure 4: Now for preferred parameters we plot 10000 episodes.

We observe in the above plot that the moving average converges to a value approaching $92-94$.

3.4 Heatmap of Q-values

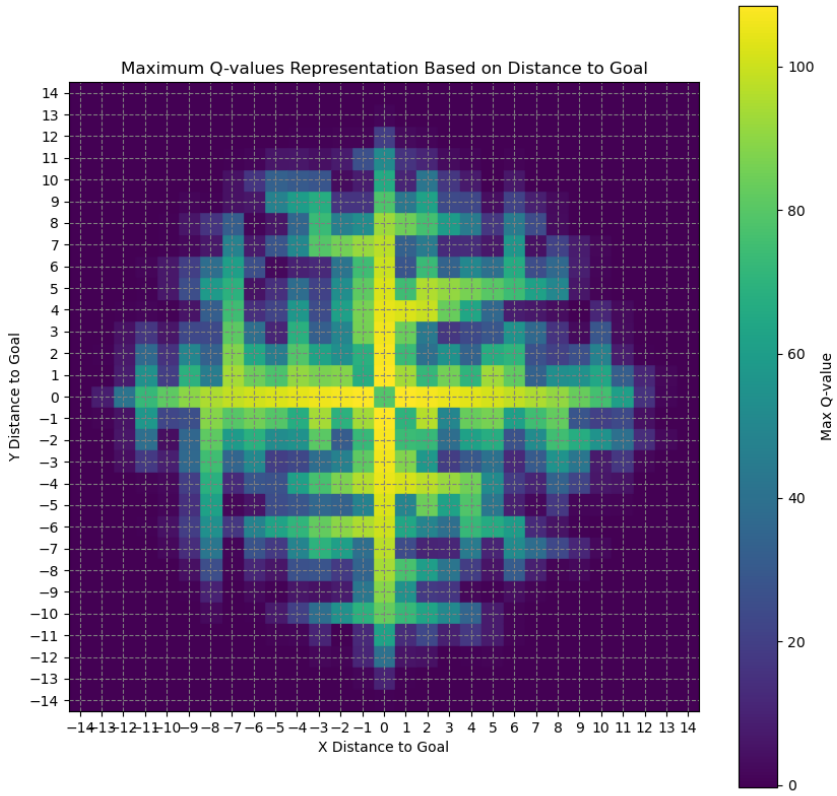

Figure 5: Q-value heatmap using a (15×15) grid, $\alpha = 0.05$, $\gamma = 0.995$ over 10000 episodes

This heatmap shows that the Q-values optimize toward following straight lines. The agents appear to prefer following rows/columns that cross the goal tile. This is intuitive since these rows/columns have a clear optimal strategy as there is only one optimal action. When on diagonals there are more than one way to walk to the goal tile. The reason the values are so low at the edges is that the amount of steps it takes to get from there to the goal is very many thus leading to a very small effect on the reward (which is only obtained at the goal tile). A higher discount factor γ would make the tiles at the edges higher, but "smooth out" the maximum state-action values so that the path might be less deterministic.

4 Discussion

4.1 Results

As we observe in results, selecting a smaller α value (learning rate) is preferred. This result is expected since a larger learning rate leads to non greedy policy due to overshooting minimum and discarding global minima. A γ seems not to have a large effect regradless of gridsize. This may be due to a flaw in tested parameters but the result may be due to the grid environment being free from obstacles, symmetric

4.2 Future improvements

One limitation to this method is that the agents learn the policies in each joint state independently, meaning that similar situations occurring in different areas of the environment or even the exact same situation occurring for different agents (or simply just the two agents swapping positions) are not affected by each other. Thus the convergence of this algorithm scales very poorly as the size of the grid-world or number of agents grows. To avoid such a large number of conflicting policies among agents and among areas of the environment, one could implement a generalization in marked states to rapidly decrease computation time. [1][CQ-learning, De Hauwere, p.718] propose implementing a simple neural network for each joint state in which conflicts appear by taking into account the *relative* positions of each agent.

References

[1] Yann-Michaël De Hauwere, Peter Vrancx, and Ann Nowé. Learning multi-agent state space representations. In *Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent Systems*, page 715–722, 2010.

5 Appendix

5.1 Class definition code

```
from pettingzoo import ParallelEnv
from gymnasium import spaces
import numpy as np

NUM_AGENTS = 1
WIDTH = 15
HEIGHT = 15
GOAL_REWARD = WIDTH + HEIGHT
STEP_COST = 0.1
CRASH_COST = 1

class GridWorld(ParallelEnv):
    metadata = {"render.modes": ["human"], "name": "GridWorld_v0"}

    def __init__(self, render_mode=None):
        super().__init__()

        # Define agents
        self.possible_agents = ["car_" + str(i) for i in range(NUM_AGENTS)]
        # Mapping between agent name and ID
        self.agent_name_mapping = dict(
            zip(self.possible_agents, list(range(len(self.possible_agents))))
        )

        self.width = WIDTH
        self.height = HEIGHT
        self.goals = {agent: np.array([np.random.randint(0, self.width-1), np.random.randint(0, self.height-1)]) for agent in self.possible_agents}     # Random goal positions
        #self.goals = {agent: np.array([np.floor(self.width/2).astype(int), np.floor(self.height/2).astype(int)]) for agent in self.possible_agents}    # Fixed goal positions
        self.render_mode = render_mode
        self.action_space = {agent: spaces.Discrete(5) for agent in self.possible_agents} # 0=wait, 1=north, 2=east, 3=south, 4=west
        self.observation_space = spaces.Dict({
            "distance_to_goal": spaces.Tuple((spaces.Discrete(self.width), spaces.Discrete(self.height))), # x, y coordinates distance to goal
        })

    def reset(self):
        self.agents = self.possible_agents.copy() # Reset agents
```

```
        self.timestep = 0

        # Reset agent positions
        self.agent_positions = {agent: np.array([
            np.random.randint(0, self.width-1), np.random.randint(0, self.height-1)
            ]) for agent in self.agents}
        #self.goals = self.goals
                                    # Goal positions set at initialization
        self.goals = {agent: np.array([np.random.randint(0, self.width-1),
                                       np.random.randint(0, self.height-1)]) for
    agent in self.agents}              # Random goal positions

        self.distance_to_goal = {agent: np.array([self.goals[agent][0] - self.
    agent_positions[agent][0],
                                                  self.goals[agent][1] - self.
    agent_positions[agent][1]]) for agent in self.agents}
        self.rewards = {agent: 0 for agent in self.agents}      # Reset rewards
        self.dones = {agent: False for agent in self.agents}    # Reset dones
        self.infos = {agent: {} for agent in self.agents}       # No additional
    information

    def step(self, actions):
        self.rewards = {agent: 0 for agent in self.agents} # Reset rewards
        tentative_positions = {}
        collisions = set()

        # Check if agents have reached their goals already
        for agent in self.agents:
            if np.array_equal(self.distance_to_goal[agent], np.array([0, 0])) and
    not self.dones[agent]:
                self.rewards[agent] += GOAL_REWARD
                self.dones[agent] = True

        # Step 1: Determine tentative next positions, ignoring done agents
        for agent in self.agents:
            if self.dones[agent]:  # Skip agents that are done
                continue
            action = actions[agent]
            current_position = self.agent_positions[agent]
            # Determine next position based on action
            if action == 1:  # North
                next_position = (current_position[0], min(current_position[1] + 1,
    self.height - 1))
            elif action == 2:  # East
                next_position = (min(current_position[0] + 1, self.width - 1),
    current_position[1])
            elif action == 3:  # South
                next_position = (current_position[0], max(current_position[1] - 1,
    0))
            elif action == 4:  # West
                next_position = (max(current_position[0] - 1, 0), current_position
    [1])
            else:              # Wait
                next_position = current_position
            tentative_positions[agent] = next_position

        # Step 2: Detect collisions among non-done agents
        for agent, position in tentative_positions.items():
            # Convert position to a tuple for comparison
            position_tuple = tuple(position)
            if sum(1 for pos in tentative_positions.values() if tuple(pos) ==
    position_tuple) > 1:
                collisions.add(agent)
                self.rewards[agent] -= CRASH_COST
                # Reset to current position due to collision
                tentative_positions[agent] = self.agent_positions[agent]

        # Step 3: Apply valid moves and update states
        for agent in self.agents:
            if self.dones[agent]:  # Do not update position or rewards for done
```

```
    agents
96                  continue
97
98              if agent not in collisions:
99                  self.agent_positions[agent] = tentative_positions[agent]  # Apply
    move
100             # Update distance to goal
101             self.distance_to_goal[agent] = np.array([self.goals[agent][0] - self.
    agent_positions[agent][0],
102                                                      self.goals[agent][1] - self.
    agent_positions[agent][1]])
103
104             # Agents entering the goal receive a reward and are set to done
105             if np.array_equal(self.distance_to_goal[agent], np.array([0, 0])):
106                 self.rewards[agent] += GOAL_REWARD
107                 self.dones[agent] = True
108
109             self.rewards[agent] -= STEP_COST  # Apply step cost for all agents that
     are not done
110
111         self.timestep += 1
112         return self.distance_to_goal, self.rewards, self.dones, self.infos
```

Listing 2: GW_env.py

5.2 Code for running simulation

```
1  import GW_env
2  import numpy as np
3  import pickle
4
5  # Environment setup
6  env = GW_env.GridWorld()
7  num_episodes = 10000     # Number of episodes
8  alpha = 0.05             # Learning rate
9  gamma = 0.995            # Discount factor
10 epsilon = 0.2            # Exploration rate
11 epsilon_decay = 0.01     # Exploration decay rate
12 min_epsilon = 0.01       # Minimum exploration rate
13
14 # Q-table setup
15 # Shared Q-table for all agents with one element for each distance-to-goal
       combination and action
16 q_tables = np.zeros((
17     (env.width)*2, (env.height)*2, 5 # 2 directions on 2 axes, 5 actions
18     ))
19
20 # Function to select a random index from the maximum values in an array
21 def select_random_max(array):
22     max_value = np.max(array)
23     max_indices = np.where(array == max_value)[0]
24     random_index = np.random.choice(max_indices)
25     return random_index
26
27 # Initialize total rewards and moving averages
28 total_rewards = {episode: 0 for episode in range(num_episodes)}
29 moving_averages = {episode: 0 for episode in range(num_episodes)}
30
31 def main(env=env, num_episodes=num_episodes, alpha=alpha, gamma=gamma, epsilon=
       epsilon, epsilon_decay=epsilon_decay, min_epsilon=min_epsilon, q_tables=
       q_tables, total_rewards=total_rewards, moving_averages=moving_averages):
32     # Q-learning algorithm
33     for episode in range(num_episodes):
34         terminate = False
35         total_reward = 0
36         env.reset()
37         #print(f"Agent positions: {env.agent_positions}")
38
39         while not terminate:
```

```
            action_dict = {}
            for agent in env.agents:
                state = env.distance_to_goal[agent] # Current state
                if state[0] < 0:
                    state[0] = (env.width-1) + np.abs(state[0])
                if state[1] < 0:
                    state[1] = (env.height-1) + np.abs(state[1])

                if np.random.uniform(0, 1) > epsilon and np.sum(np.abs(q_tables[
    state[0], state[1]])) > 0:
                    #action = np.argmax(q_tables[state[0], state[1]]) # Exploit
                    action = select_random_max(q_tables[state[0], state[1]]) #
    Exploit
                else:
                    action = np.random.choice([0, 1, 2, 3, 4]) # Explore
                action_dict[agent] = action
                current_dist = env.distance_to_goal[agent] # Current distance to
    goal

            # Take action
            next_state, reward, done, _ = env.step(action_dict)

            # Update Q-table
            for agent in env.agents:
                next_dist = next_state[agent] # Next distance to goal
                if current_dist[0] < 0:
                    current_dist[0] = (env.width-1) + np.abs(current_dist[0])
                if current_dist[1] < 0:
                    current_dist[1] = (env.height-1) + np.abs(current_dist[1])
                if next_dist[0] < 0:
                    next_dist[0] = (env.width-1) + np.abs(next_dist[0])
                if next_dist[1] < 0:
                    next_dist[1] = (env.height-1) + np.abs(next_dist[1])

                old_q_value = q_tables[current_dist[0], current_dist[1],
    action_dict[agent]]       # Previous Q-value
                td_target = reward[agent] + gamma * np.max(q_tables[next_dist[0],
    next_dist[1]])   # TD target value
                td_error = td_target - old_q_value
                    # TD error
                q_tables[current_dist[0], current_dist[1], action_dict[agent]] +=
    alpha * td_error # Update Q-value using Q-learning update rule

            # Update total reward
            total_reward += sum(reward.values())

            # Check if episode is done
            terminate = all(done.values()) or env.timestep >= 1000

            # Print positions and actions for each agent as well as the Q-values
    and the total reward
            #print(f"Agent actions: {action_dict}")
            #print(f"Agent positions: {next_state}")
            #print(f"Agent Q-values: \n{q_tables}")
            #print(f"Total reward: {total_reward}")

        total_rewards[episode] = total_reward
        # Moving average of total reward over the last 100 episodes or the
    available episodes
        moving_average = np.mean(list(total_rewards.values())[max(0, episode-100):
    episode+1])

        # Store moving average of total reward
        moving_averages[episode] = moving_average

        # Decay exploration rate
        epsilon = max(min_epsilon, epsilon * np.exp(-epsilon_decay * episode))
        #epsilon = max(min_epsilon, epsilon * epsilon_decay)

        # Print episode results and moving average of total reward
```

```
        print(f"Episode {episode+1}/{num_episodes} - Average total reward: {moving_average:.1f} - Epsilon: {epsilon:.2f}")
        #print()

        # Save moving averages and Q-tables
        with open('moving_averages.pkl', 'wb') as f:
            pickle.dump(moving_averages, f)

        with open('q_tables.pkl', 'wb') as f:
            pickle.dump(q_tables, f)

if __name__ == "__main__":
    main()
```

Listing 3: myminigrid.py

5.3 Blue-line convergence plot

```
# Plot values
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.patches as mpatches
import numpy as np
import pickle
from myminigrid import num_episodes, alpha, gamma, epsilon, epsilon_decay, min_epsilon
from GW_env import NUM_AGENTS, WIDTH, HEIGHT, GOAL_REWARD, STEP_COST, CRASH_COST

with open('moving_averages.pkl', 'rb') as f:
    moving_averages = pickle.load(f)

# Extract episodes and moving average values
episodes = list(moving_averages.keys())
average_rewards = list(moving_averages.values())

# String of parameter values
parameters_str = f"""Num Episodes: {num_episodes}
Alpha: {alpha}
Gamma: {gamma}
Epsilon: {epsilon}
Epsilon Decay: {epsilon_decay}
Min Epsilon: {min_epsilon}
Num Agents: {NUM_AGENTS}
Width: {WIDTH}
Height: {HEIGHT}
Goal Reward: {GOAL_REWARD}
Step Cost: {STEP_COST}
Crash Cost: {CRASH_COST}"""

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(episodes, average_rewards, color='blue', label='Moving Average of Total Rewards')

# Labeling the plot
plt.xlabel('Episode')
plt.ylabel('Average Total Reward')
plt.title('Moving Average of Total Rewards Over Episodes')
plt.legend()

# Display parameters in a neat box
plt.figtext(0.7, 0.2, parameters_str, bbox=dict(boxstyle="round,pad=0.5", fc="white", ec="black", lw=1))

# Display the plot
plt.grid(True)
plt.show()
```

Listing 4: GW_plots.py

5.4 Grid illustration plot

```python
import matplotlib.pyplot as plt
import numpy as np

WIDTH, HEIGHT = 5, 5

# Adjusted ranges to include from -4 to +4, aligning with observation space
x_range = np.arange(-WIDTH + 1, WIDTH)  # From -4 to 4
y_range = np.arange(-HEIGHT + 1, HEIGHT)  # From -4 to 4
xx, yy = np.meshgrid(x_range, y_range)
distance = np.sqrt(xx**2 + yy**2)

# Normalize distances for color intensity
max_distance = np.sqrt((WIDTH - 1)**2 + (HEIGHT - 1)**2)
norm_distance = 1 - (distance / max_distance)

fig, ax = plt.subplots(figsize=(10, 10))
for i in range(len(x_range)):
    for j in range(len(y_range)):
        # Color intensity based on distance
        color_value = norm_distance[j, i]
        rect = plt.Rectangle((x_range[i]-0.5, y_range[j]-0.5), 1, 1, color=(
    color_value, 0, 0, 0.5))  # Semi-transparent
        ax.add_patch(rect)

# Adjust the axis limits and aspect ratio
ax.set_xlim([-WIDTH + 0.5, WIDTH - 1.5+1])  # From -4 to 4
ax.set_ylim([-HEIGHT + 0.5, HEIGHT - 1.5+1])  # From -4 to 4
ax.set_aspect('equal')

# Customizing the grid
ax.set_xticks(np.arange(-WIDTH + 1, WIDTH, 1))
ax.set_yticks(np.arange(-HEIGHT + 1, HEIGHT, 1))
ax.grid(True, which='both', linestyle='--', color='grey')

ax.set_xlabel('X Distance to Goal')
ax.set_ylabel('Y Distance to Goal')
ax.set_title('State Representation Based on Distance to Goal')

plt.show()
```

Listing 5: Grid_illustraiton.py

5.5 Q-value plot

```python
import matplotlib.pyplot as plt
import numpy as np
import pickle
from GW_env import WIDTH, HEIGHT

with open('q_tables.pkl', 'rb') as f:
    q_tables = pickle.load(f)

# Calculate the maximum Q-value for each state
max_q_values = np.max(q_tables, axis=2)

fig, ax = plt.subplots(figsize=(10, 10))

# Create a color map to represent Q-values
cmap = plt.cm.get_cmap('viridis')

# Normalize the Q-values for color mapping
norm = plt.Normalize(np.min(max_q_values), np.max(max_q_values))

# Plotting each state with its corresponding max Q-value
for x in range(0, 2*WIDTH - 1):
    for y in range(0, 2*HEIGHT - 1):
```

```
        if x < WIDTH and y < HEIGHT:  # No conversion needed
            i, j = x, y
        elif x < WIDTH and y >= HEIGHT:  # Convert y to negative
            i, j = x, (y-HEIGHT+1) * (-1)
        elif x >= WIDTH and y < HEIGHT:  # Convert x to negative
            i, j = (x-WIDTH+1) * (-1), y
        else:  # Convert x and y to negative
            i, j = (x-WIDTH+1) * (-1), (y-HEIGHT+1) * (-1)

        # Color intensity based on the max Q-value
        color = cmap(norm(max_q_values[x, y]))
        rect = plt.Rectangle((i-0.5, j-0.5), 1, 1, color=color)
        ax.add_patch(rect)

# Adjust the axis limits and aspect ratio
ax.set_xlim([-WIDTH + 0.5, WIDTH - 0.5])
ax.set_ylim([-HEIGHT + 0.5, HEIGHT - 0.5])
ax.set_aspect('equal')

# Customizing the grid
ax.set_xticks(np.arange(-WIDTH + 1, WIDTH, 1))
ax.set_yticks(np.arange(-HEIGHT + 1, HEIGHT, 1))
ax.grid(True, which='both', linestyle='--', color='grey')

ax.set_xlabel('X Distance to Goal')
ax.set_ylabel('Y Distance to Goal')
ax.set_title('Maximum Q-values Representation Based on Distance to Goal')

# Adding a color bar to understand the values
sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])
plt.colorbar(sm, ax=ax, label='Max Q-value')

plt.show()
```

Listing 6: q_value_plot.py