Getting Out of a Maze with a Reinforcement Learning Agent

defne.odabasi

May 2024

Contents

1 Introduction

2 Basic Questions

In this part reinforcement learning terms are compared with thir equivalent terms in supervised learning. For each term, definition is provided.

1. **Agent**

   - **RL**: The learner which uses a policy to maximize the expected gain from transitioning between states in a cumulative manner.

- **SL**: In supervised learning, the model is trained such that input leads to an output. Even though there is no direct equivalent of term agent, it may refer to the model since model makes predictions.

2. **Environment**

   - **RL**: The external system which agent interacts with. The environment can have features such as boundaries, traps, goals.
   - **SL**: In supervised learning the environment may be the equivalent of data set as it contains features and target labels.

3. **Reward**

   - **RL**: A received feedback after taking an action. It guides the agent toward the beneficial actions during learning. We can think of reward as a motivational factor for the agent so that it can take action towards it.
   - **SL**: The reward in supervised learning can be thought as the correct output for a given input. Therefore the loss function of the model is minimum.

4. **Policy**

   - **RL**: A strategy used by the agent, it performs a mapping from a state to an action. A policy can be stochastic for example. In this case the agent may act in an unexpected way which yield it way toward something unknown.
   - **SL**: In supervised learning, the model outputs a data with respect to the input data. Even though, there is no direct equivalent the policy can be the sturcture of the model such that it yields to an output.

5. **Exploration**

   - **RL**: New sequence of actions are taken to improve the knowledge of the agent on the environment. In exploration, unknown states can be discovered using the exploration function. Therefore, the knowledge of the agent toward the environment can be enhanced.
   - **SL**: New pairs of actions are trained in cross-validation technique to improve the model generalization. This is ,again, not a direct equivalent since in supervised learning the model has trained weights and layers.

6. **Exploitation**

   - **RL**: The known optimal policy is executed for high reward. In exploitation, an action which is known to yield the agent toward a reward based on previous experiments is taken. Therefore, it focuses on having the maximum reward. However the agent might end up not discovering states which are actually more beneficial for the agent.
   - **SL**: This is more similar to how supervised learning functions since the model is trained and thus makes predictions on the new data. Therefore it outputs what is trained before rather than improving the model itself.

3 Experimental Work

As shown in Table 1, variety of parameter configurations are experimented to determine their effect on the TD(0) and Q learning. The default values are highlighted as well.

Table 1: Parameter configurations to be experimented for both TD(0) and Q learning.

Parameter	Values				
α	0.001	0.01	**0.1**	0.5	1.0
γ	0.10	0.25	0.50	0.75	**0.95**
ϵ	0	**0.2**	0.5	0.8	1.0

To better understand the convergence graphs for different hyperparameters, the exponential moving average method is applied to the convergence data. This approach offers a clearer representation of the value function changes from one iteration to the next over time, making the convergence plots more interpretable. Therefore the scale might be different in the case of placing the convergence plot for iteration to iteration.

3.1 TD(0) learning

In Figure 1, bold parameters are the default parameters. The resuls for the TD(0) learning for bold parameters is shown Figure 1, 2, 3.

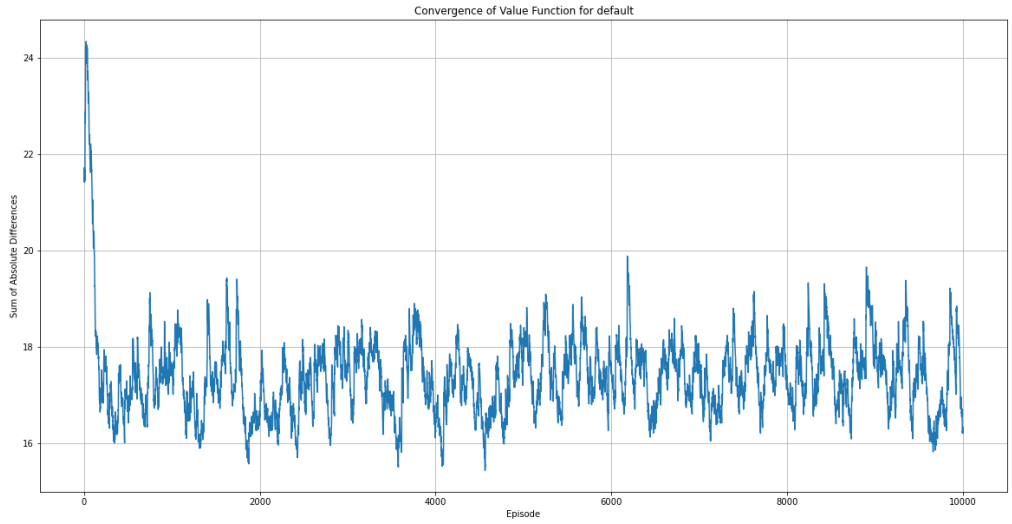

Figure 1: Convergence of the TD learning for default values.

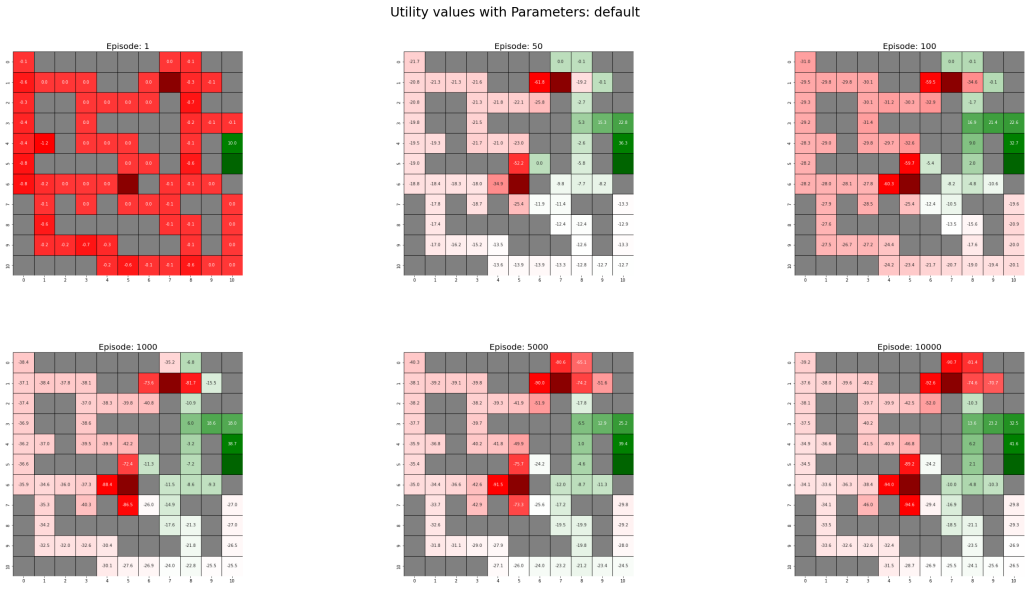

Figure 2: Utilities of the TD learning for default values.

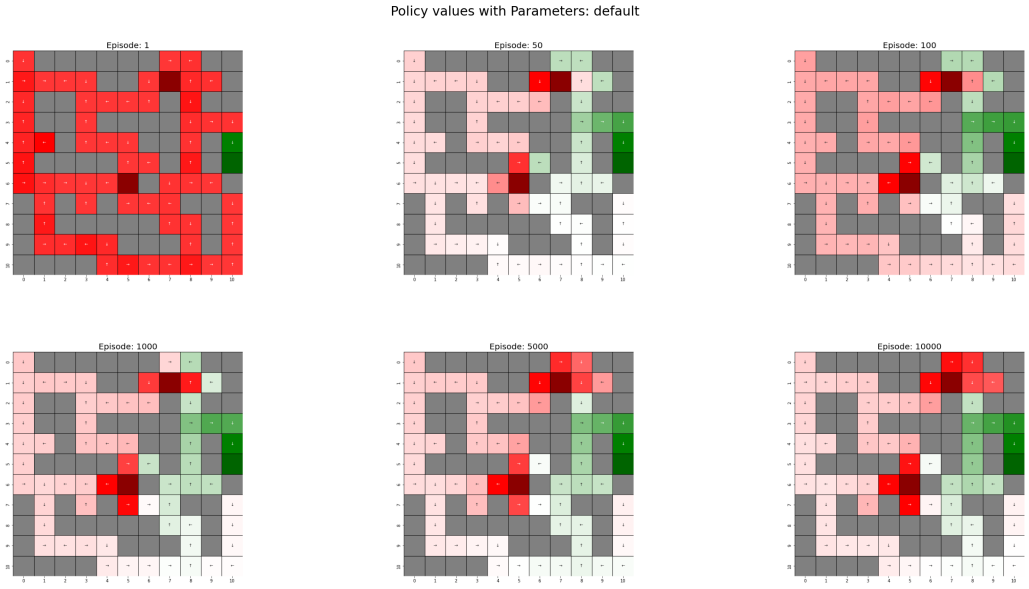

Figure 3: Policy of the TD learning for default values.

3.1.1 Learning Rate (α)

α = 0.001:

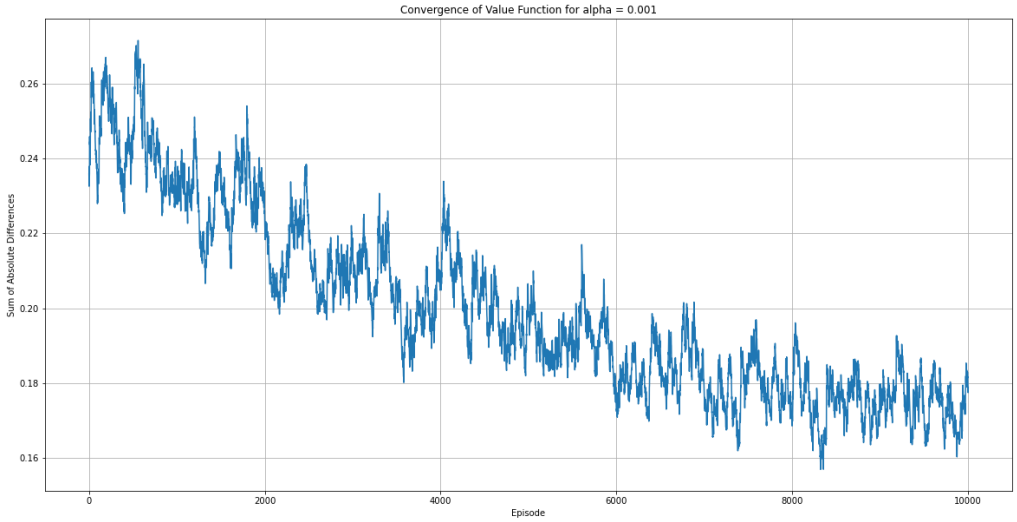

Figure 4: Convergence of the TD learning for $\alpha = 0.001$.

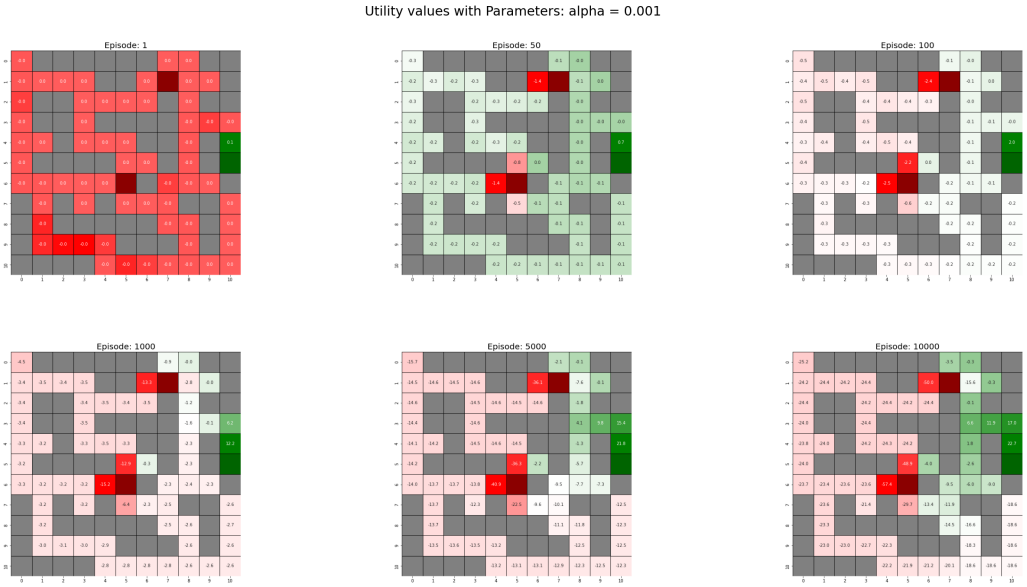

Figure 5: Utilities of the TD learning for $\alpha = 0.001$.

Figure 6: Policy of the TD learning for $\alpha = 0.001$.

$\alpha = 0.01$:

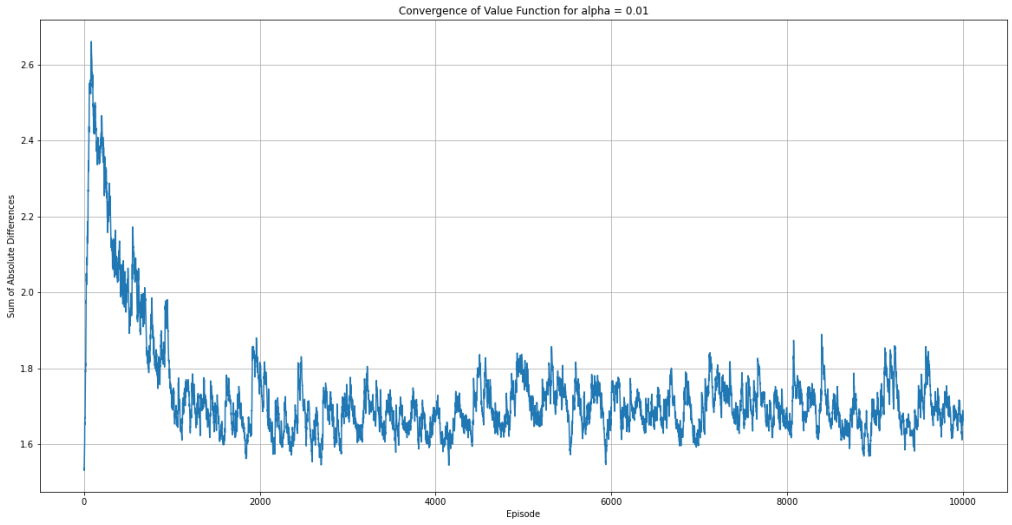

Figure 7: Convergence of the TD learning for $\alpha = 0.01$.

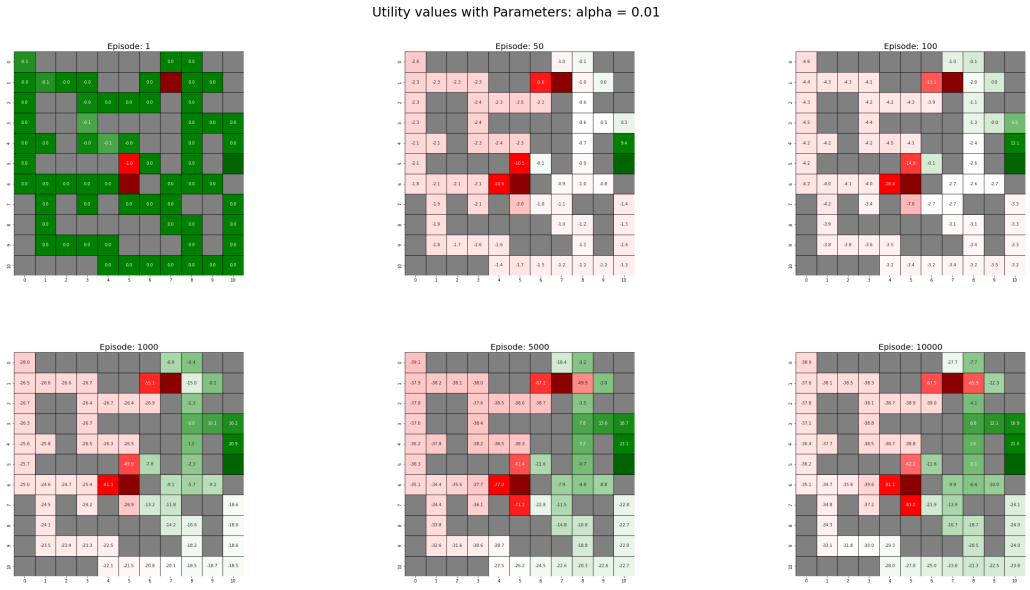

Figure 8: Utilities of the TD learning for $\alpha = 0.01$.

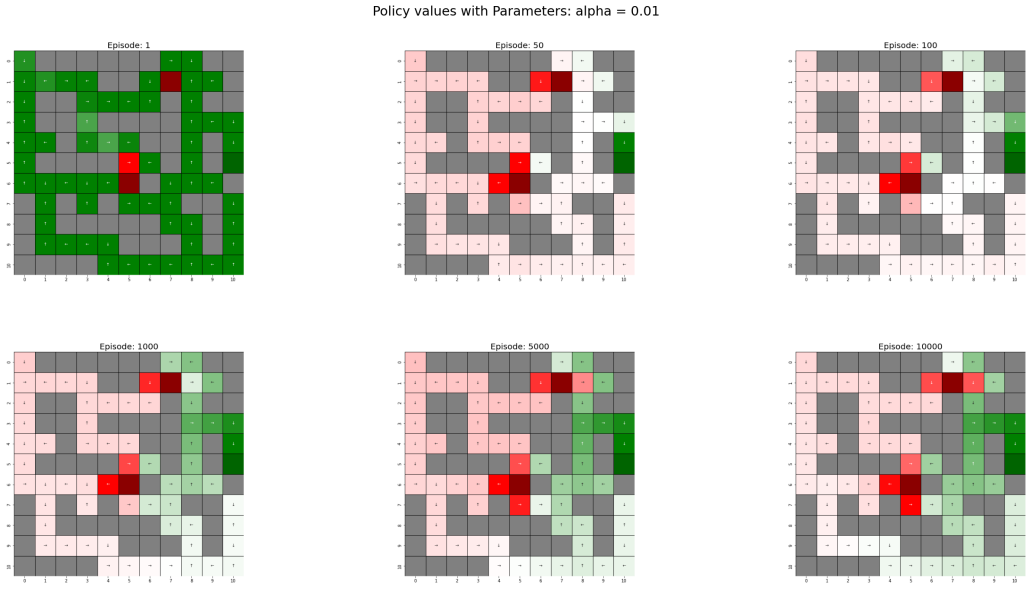

Figure 9: Policy of the TD learning for $\alpha = 0.01$.

$\alpha = \mathbf{0.5}$**:**

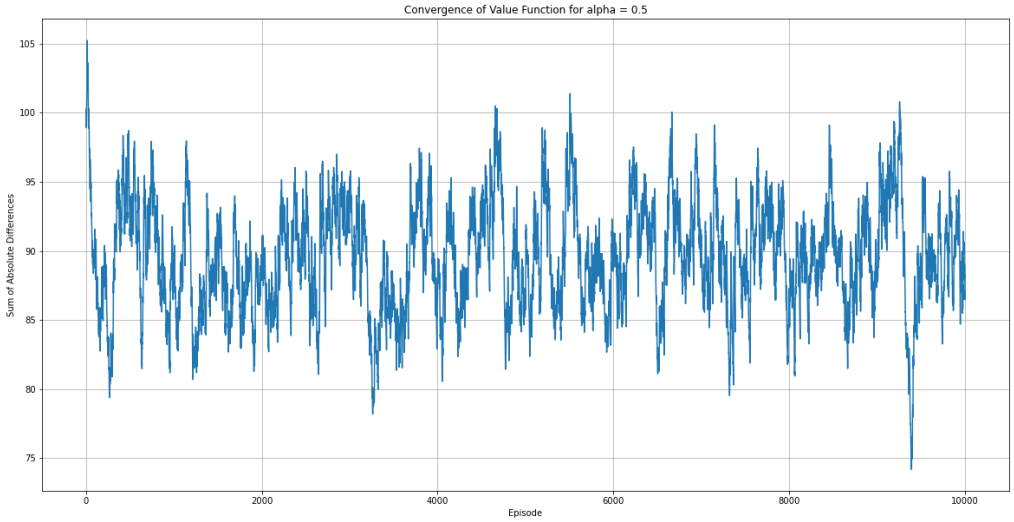

Figure 10: Convergence of the TD learning for $\alpha = 0.5$.

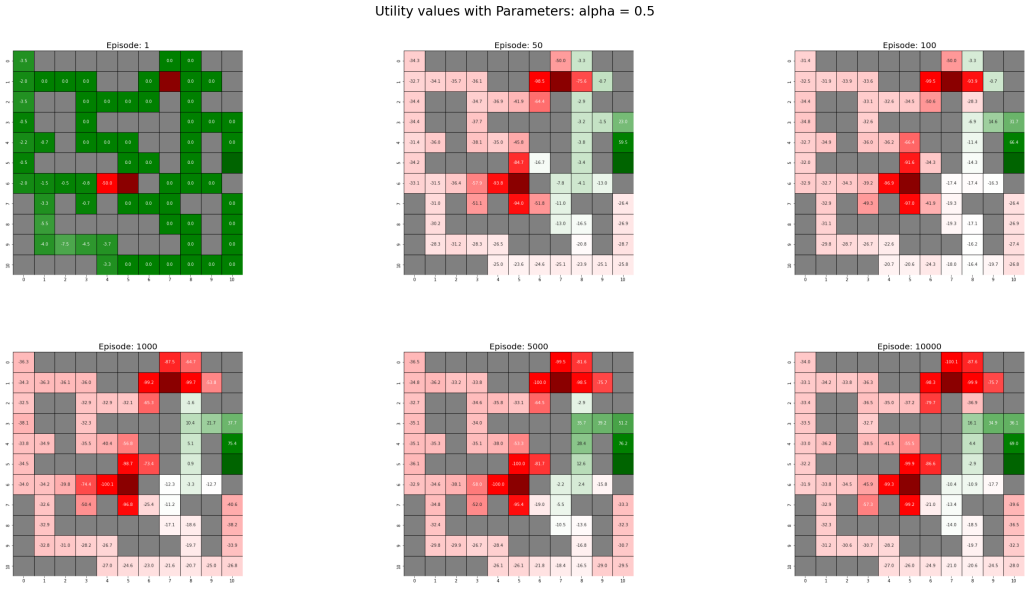

Figure 11: Utilities of the TD learning for $\alpha = 0.5$.

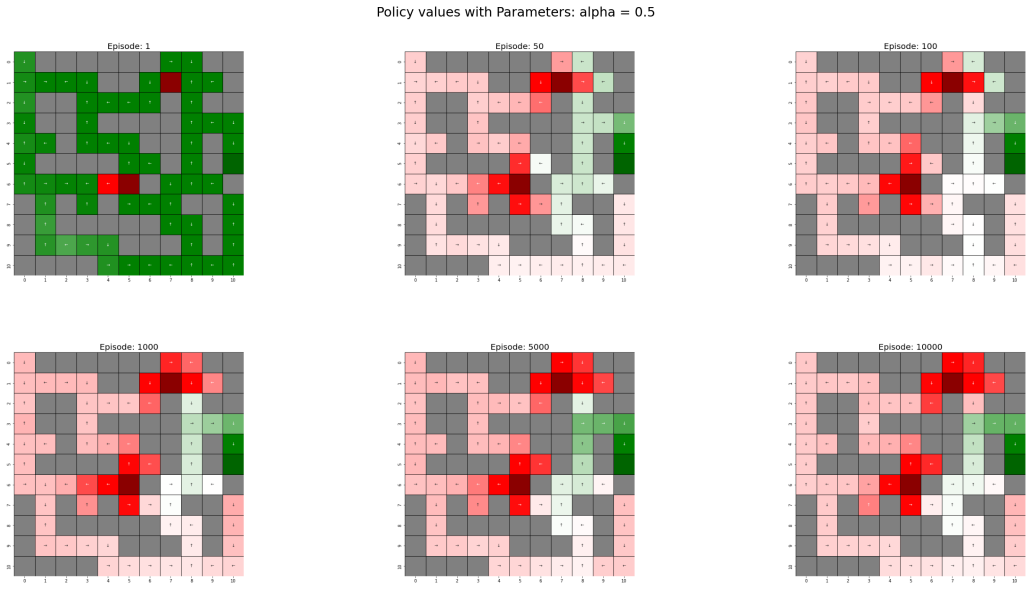

Figure 12: Policy of the TD learning for $\alpha = 0.5$.

$\alpha = 1$:

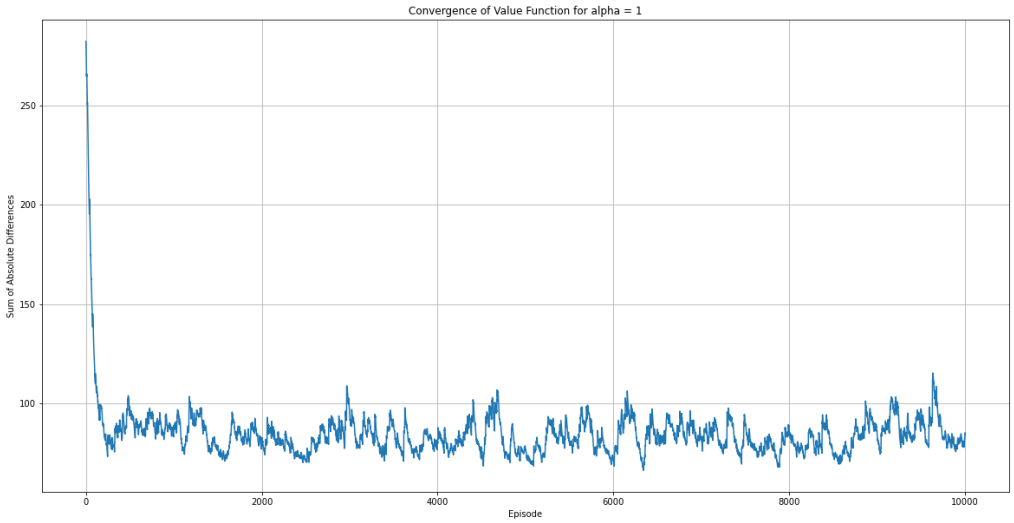

Figure 13: Convergence of the TD learning for $\alpha = 1$.

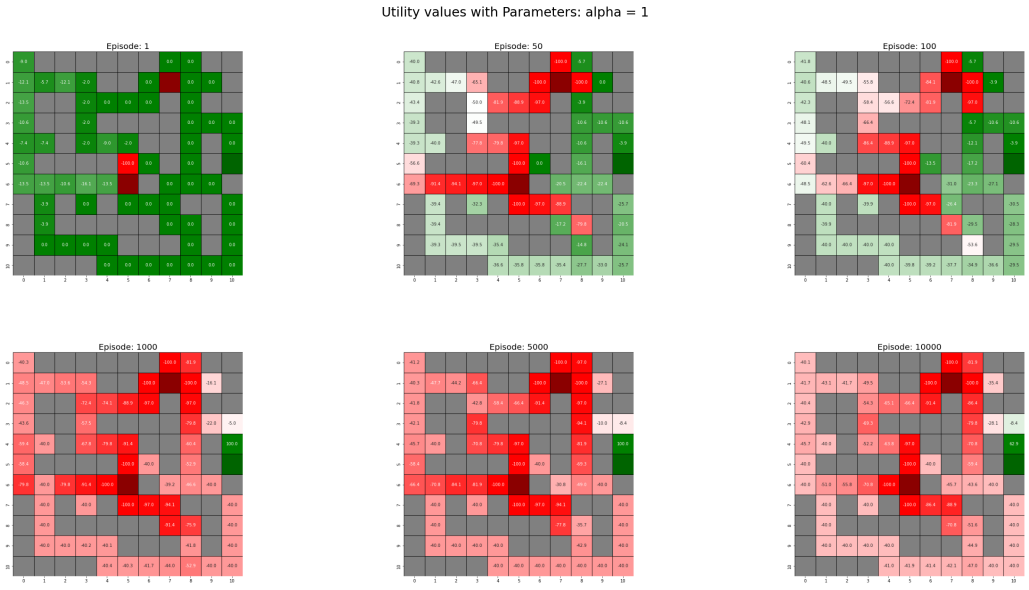

Figure 14: Utilities of the TD learning for $\alpha = 1$.

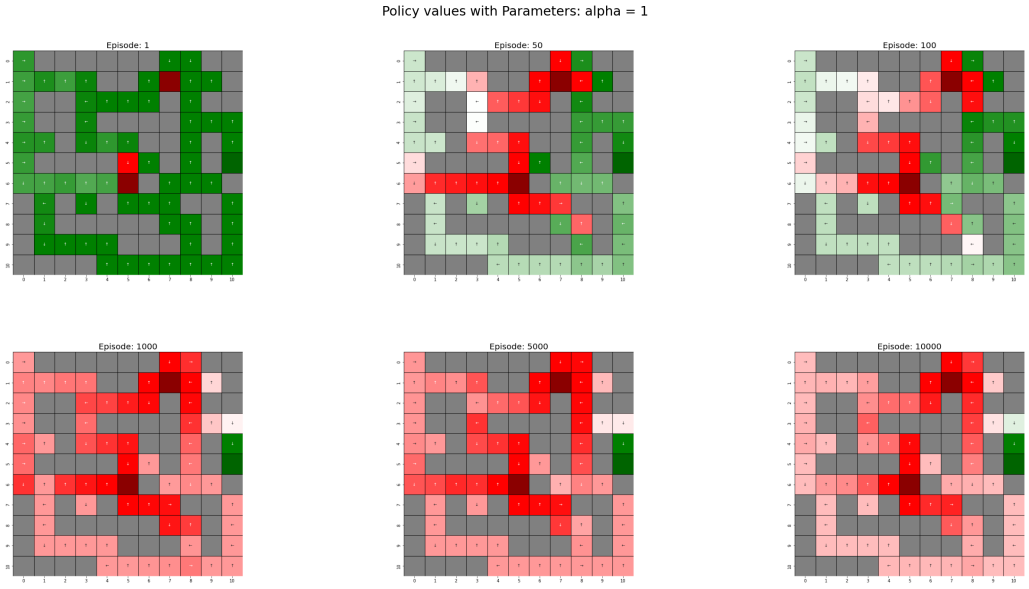

Figure 15: Policy of the TD learning for $\alpha = 1$.

3.1.2 Discount Factor (γ)

$\gamma = 0.1$:

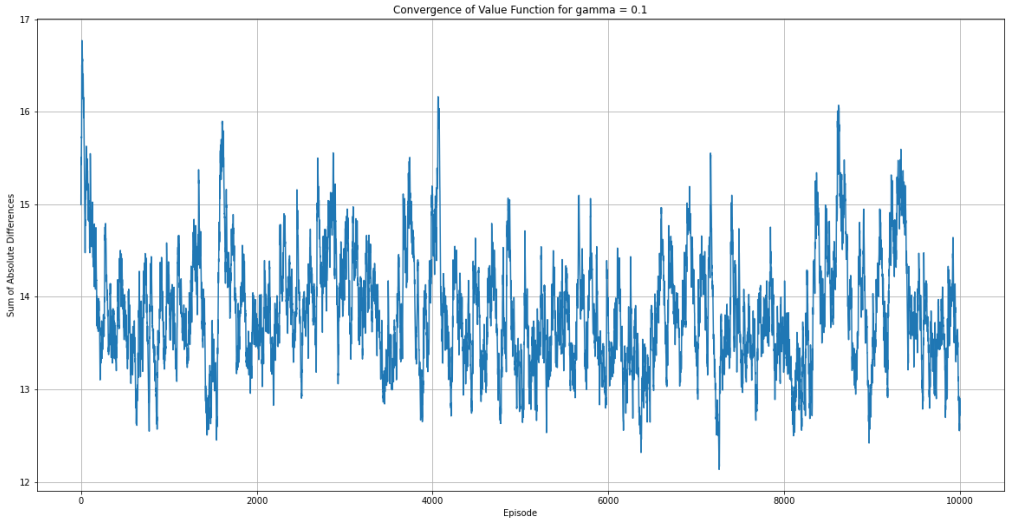

Figure 16: Convergence of the TD learning for $\gamma = 0.1$.

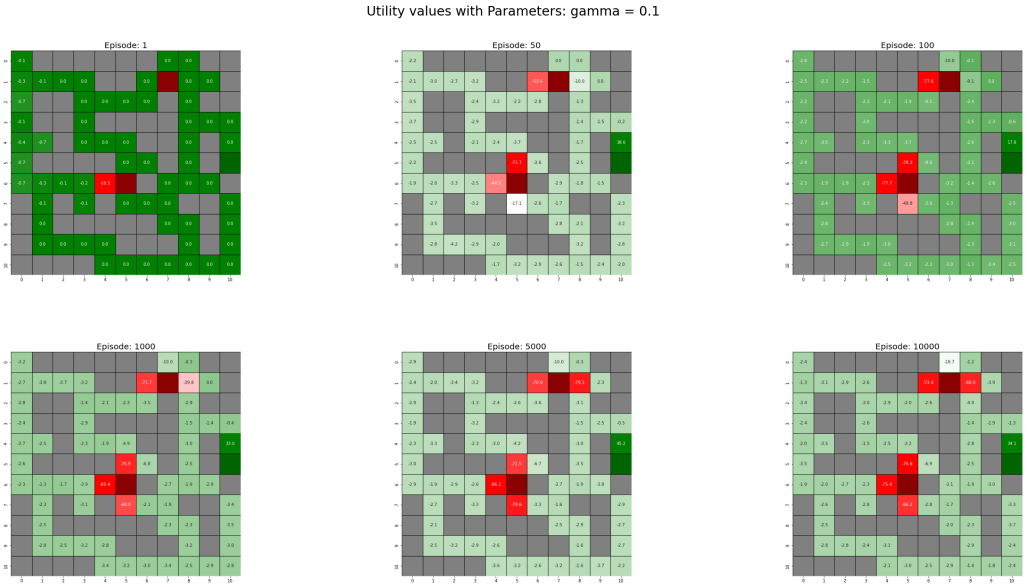

Figure 17: Utilities of the TD learning for $\gamma = 0.1$.

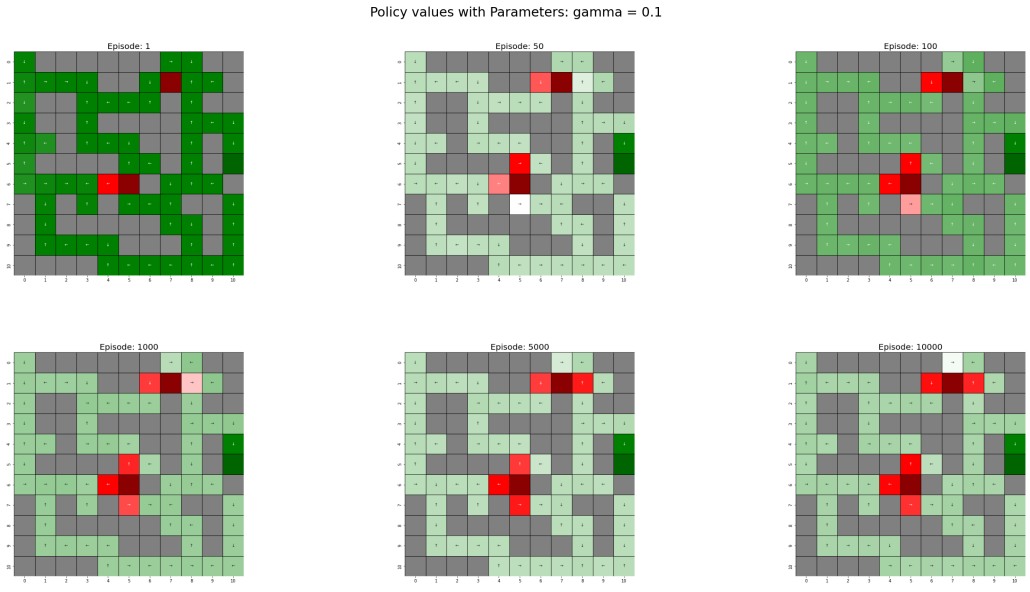

Figure 18: Policy of the TD learning for $\gamma = 0.1$.

$\gamma = 0.25$:

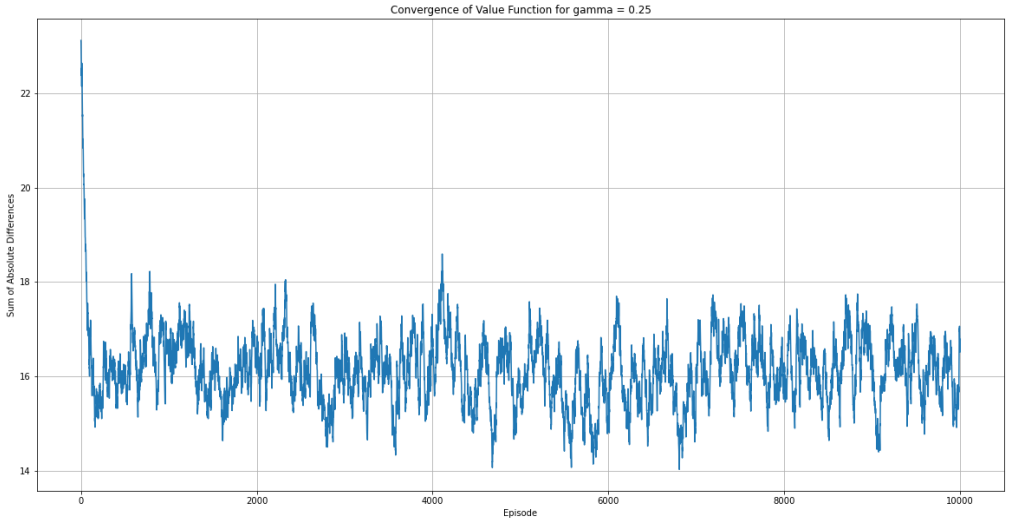

Figure 19: Convergence of the TD learning for $\gamma = 0.25$.

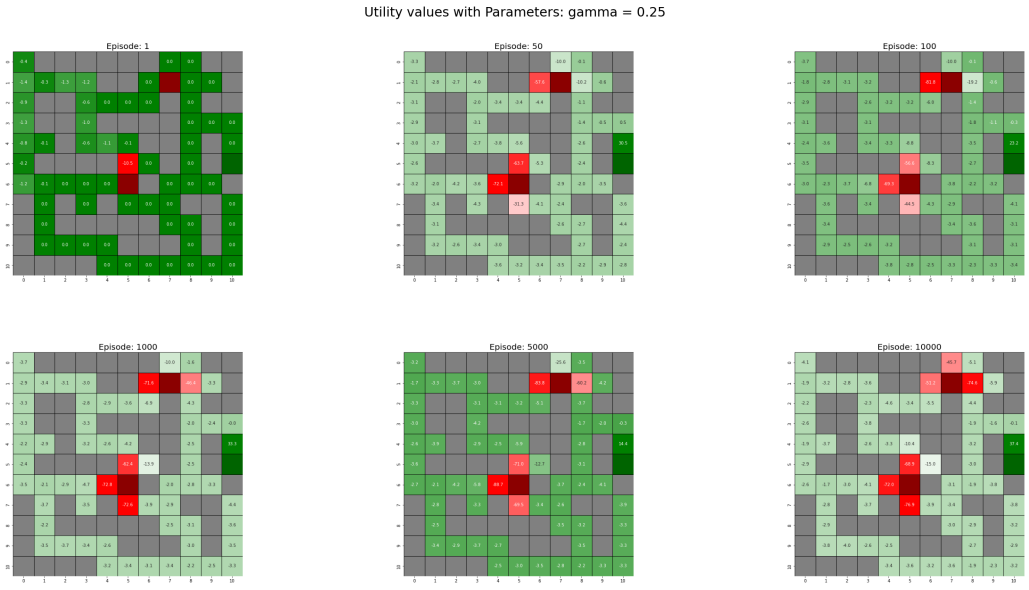

Figure 20: Utilities of the TD learning for $\gamma = 0.1$.

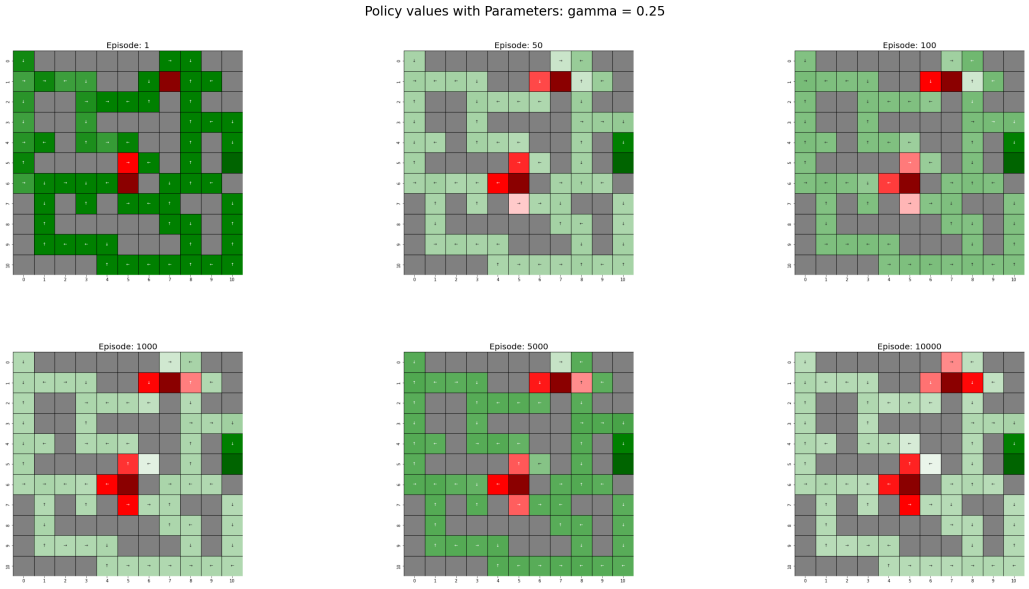

Figure 21: Policy of the TD learning for $\gamma = 0.25$.

$\gamma = 0.5$:

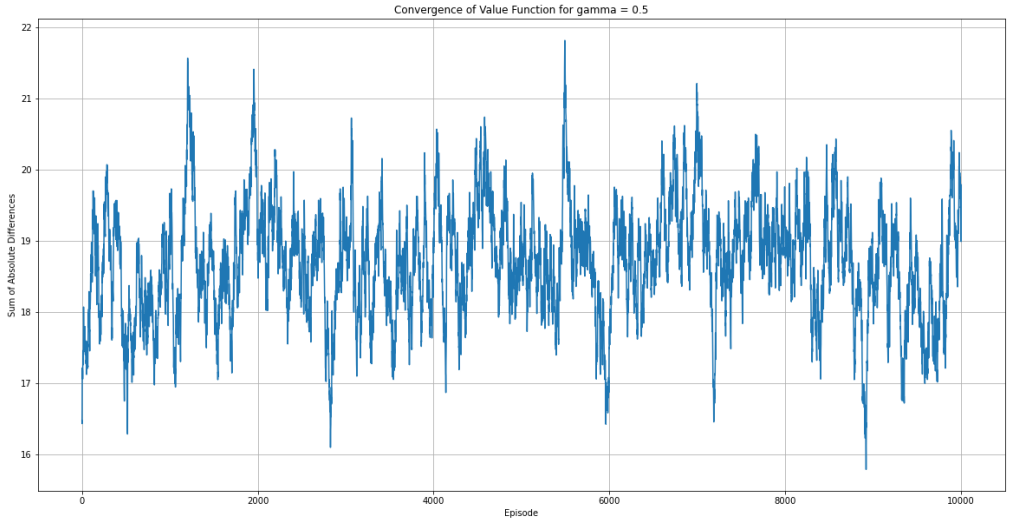

Figure 22: Convergence of the TD learning for $\gamma = 0.5$.

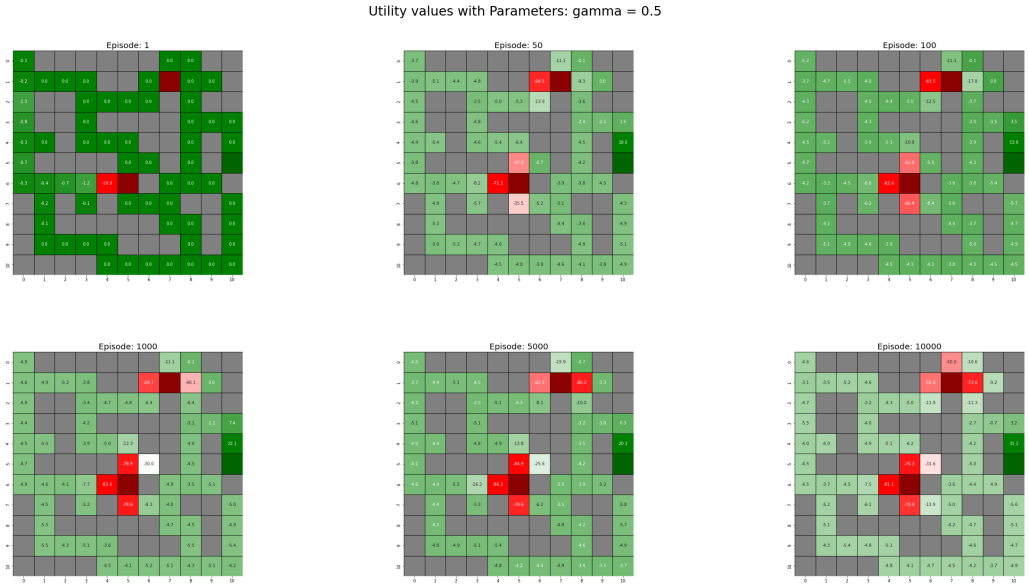

Figure 23: Utilities of the TD learning for $\gamma = 0.5$.

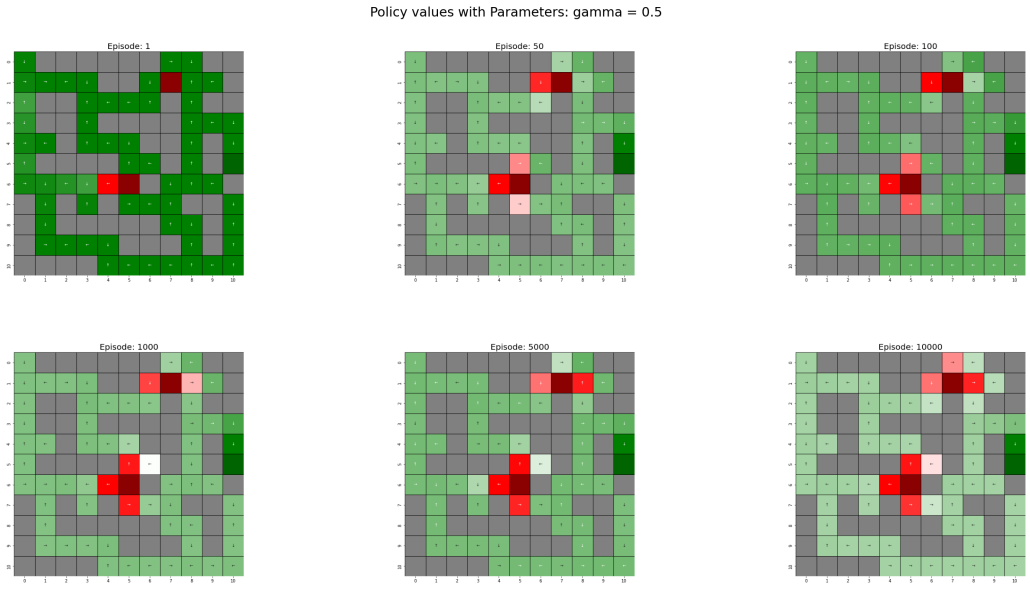

Figure 24: Policy of the TD learning for $\gamma = 0.5$.

$\gamma = 0.75$:

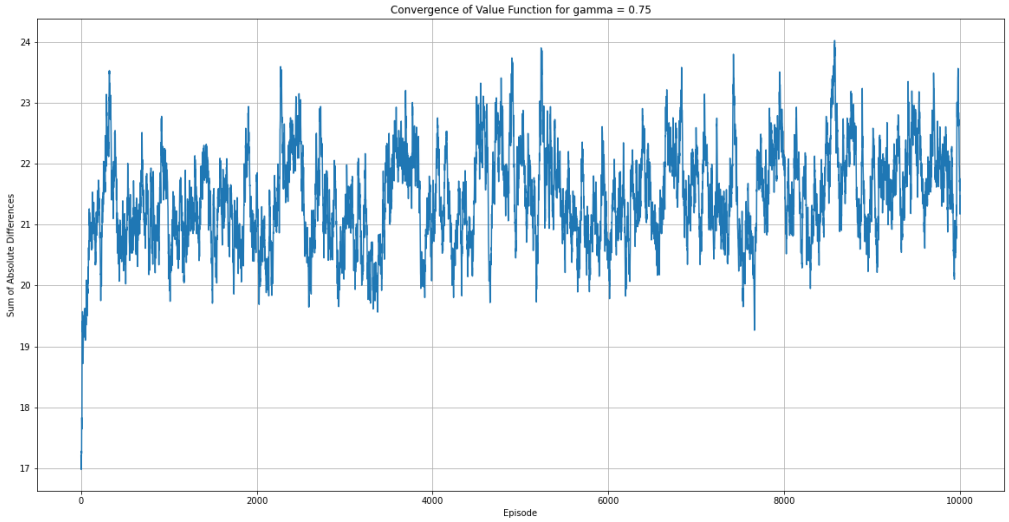

Figure 25: Convergence of the TD learning for $\gamma = 0.75$.

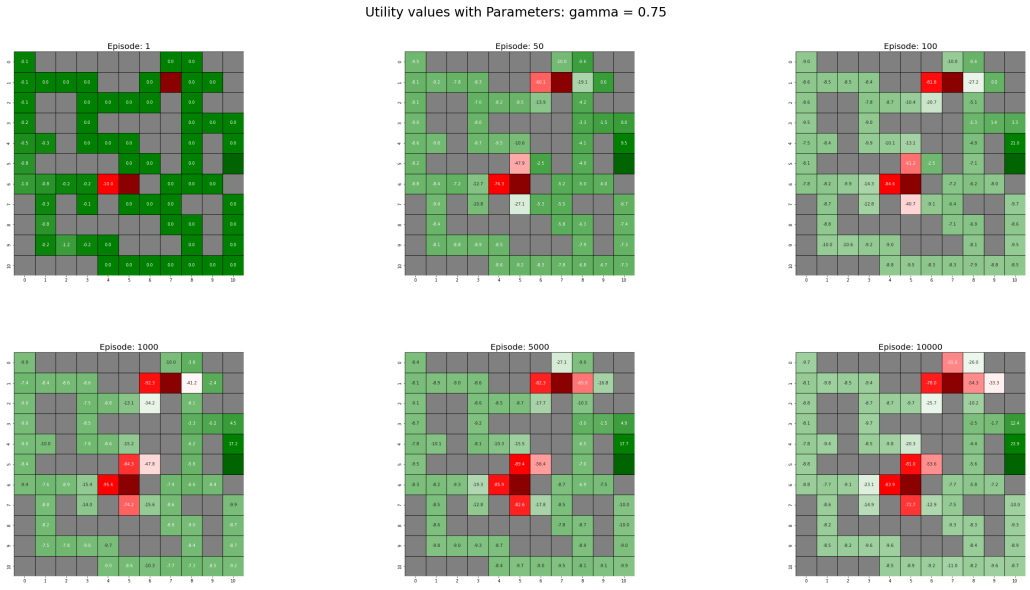

Figure 26: Utilities of the TD learning for $\gamma = 0.75$.

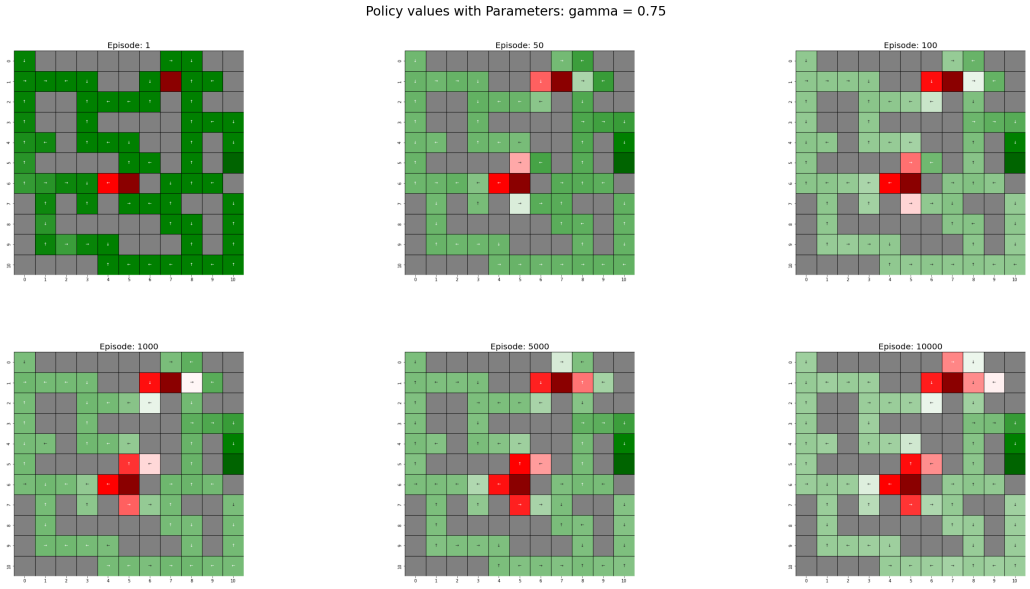

Figure 27: Policy of the TD learning for $\gamma = 0.75$.

3.1.3 Initial Exploration rate (ϵ)

$\epsilon = 0$:

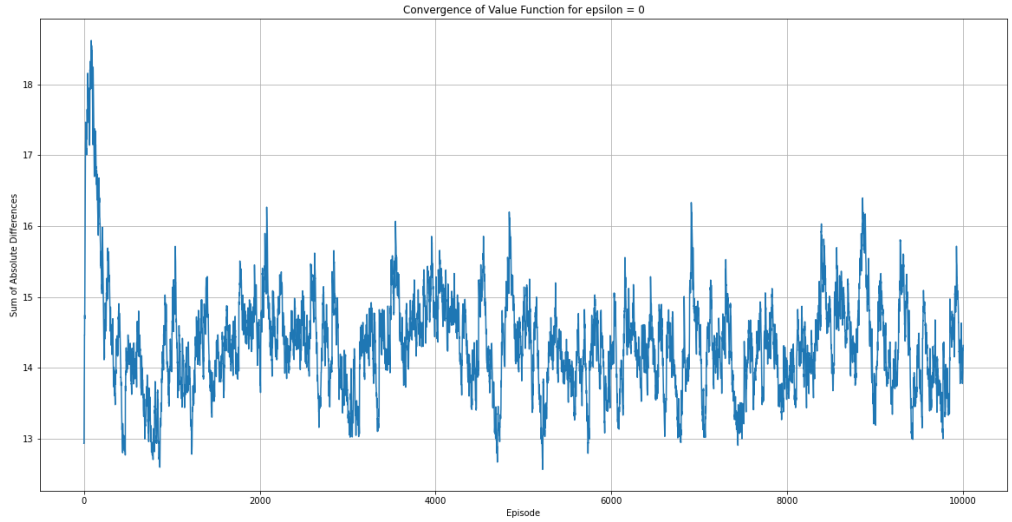

Figure 28: Convergence of the TD learning for $\epsilon = 0$.

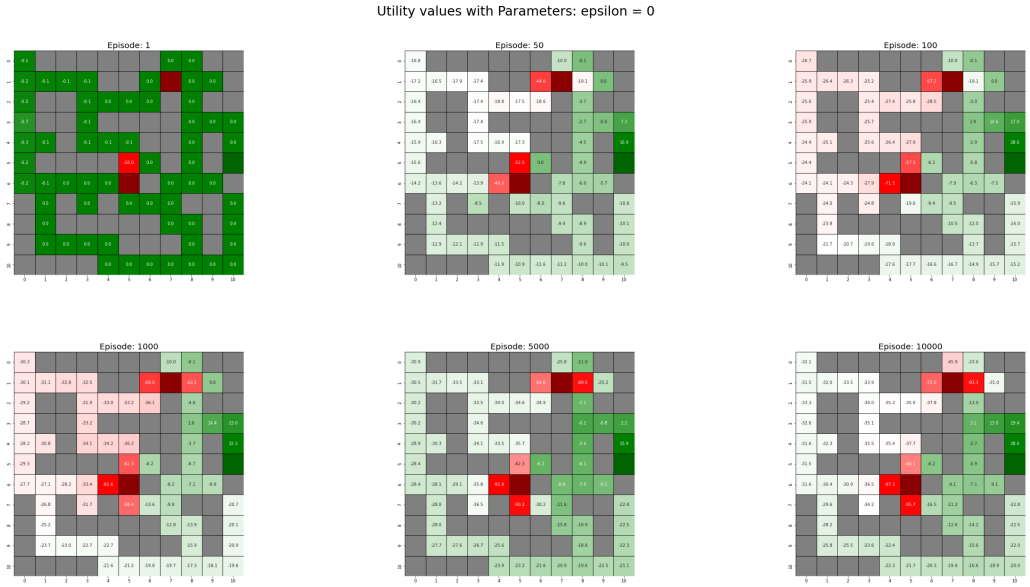

Figure 29: Utilities of the TD learning for $\epsilon = 0$.

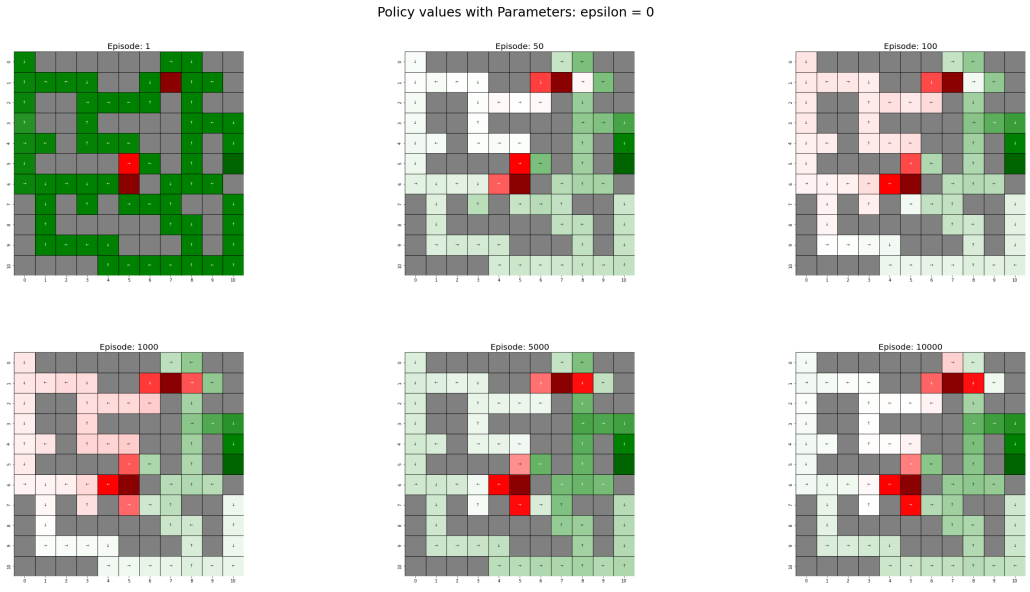

Figure 30: Policy of the TD learning for $\epsilon = 0$.

$\epsilon = 0.5$:

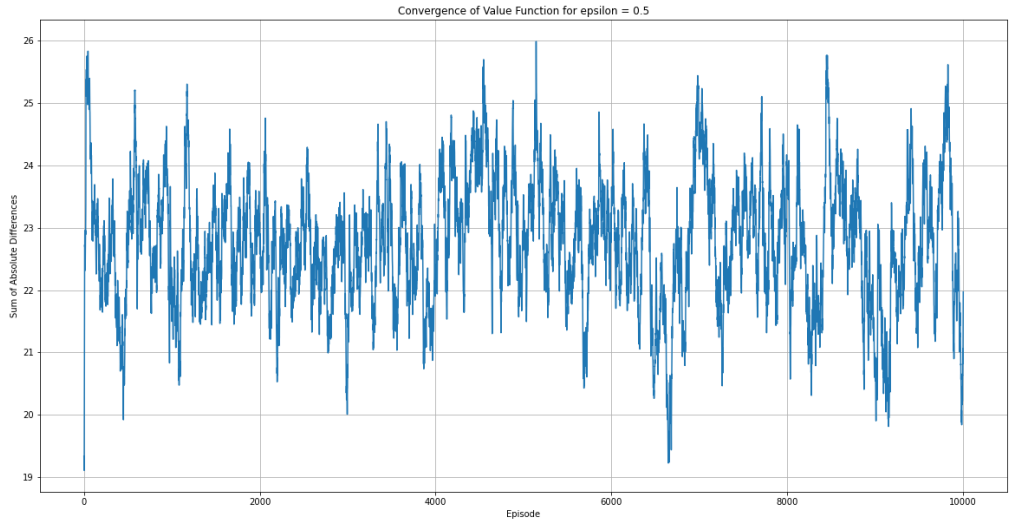

Figure 31: Convergence of the TD learning for $\epsilon = 0.5$.

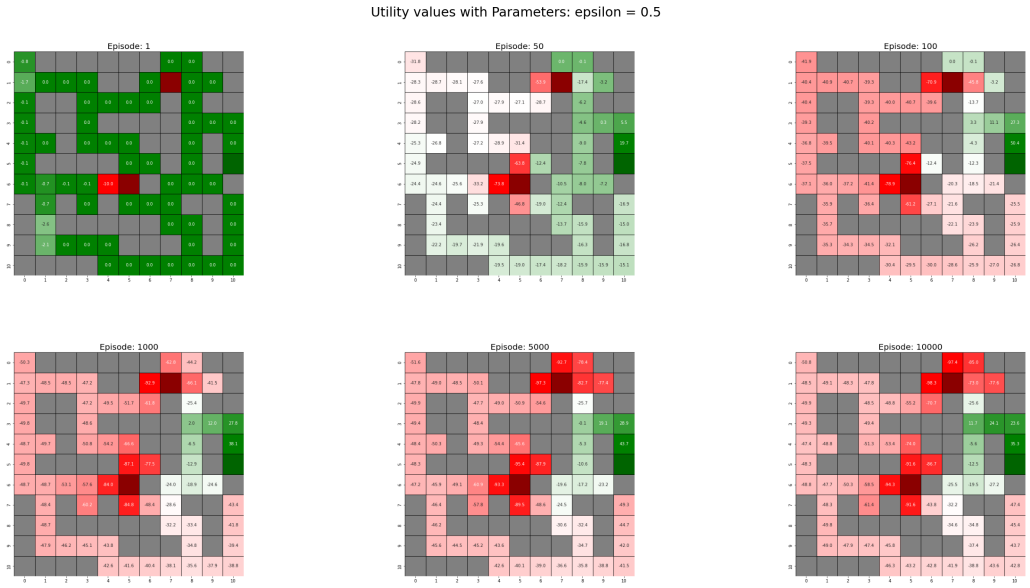

Figure 32: Utilities of the TD learning for $\epsilon = 0.5$.

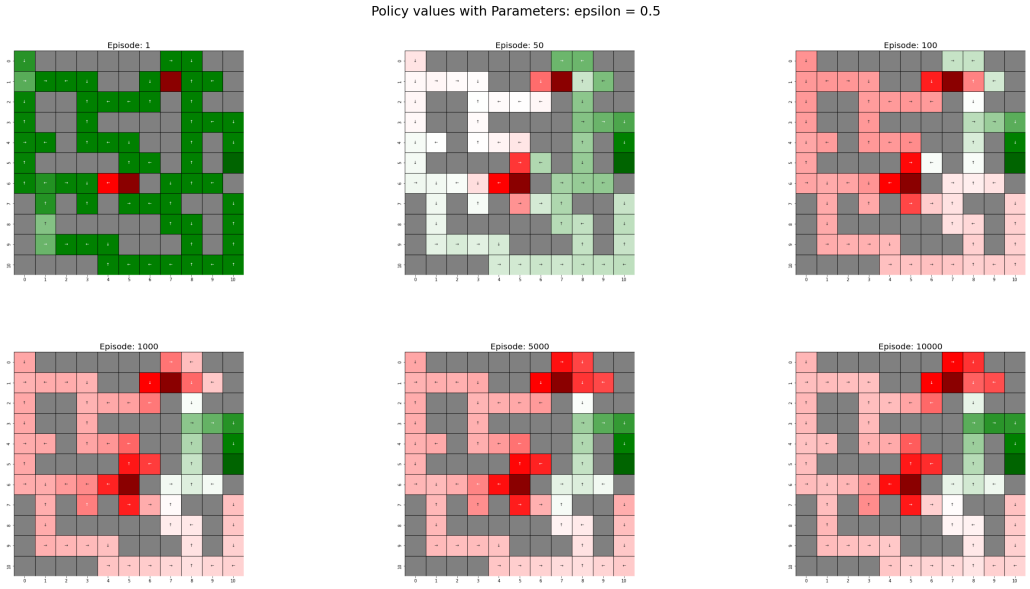

Figure 33: Policy of the TD learning for $\epsilon = 0.5$.

$\epsilon = 0.8$:

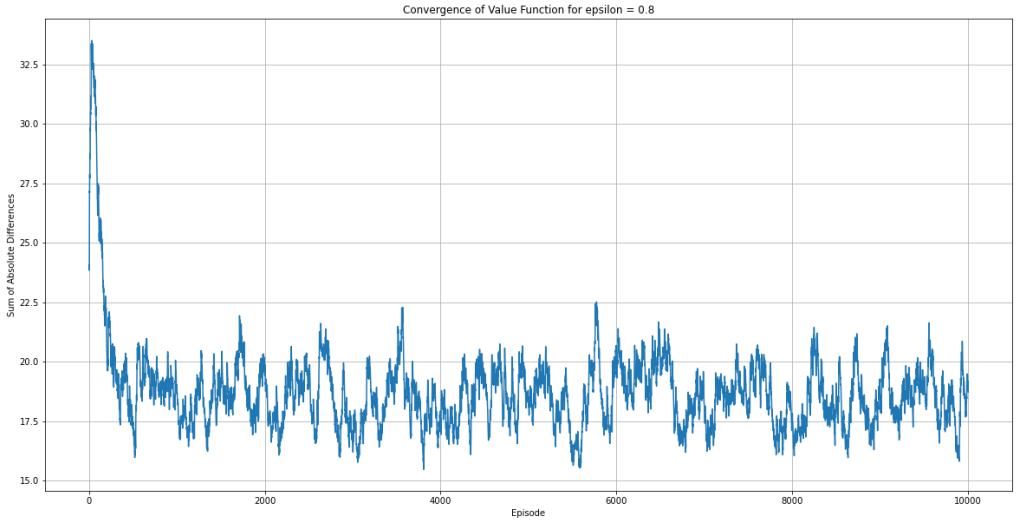

Figure 34: Convergence of the TD learning for $\epsilon = 0.8$.

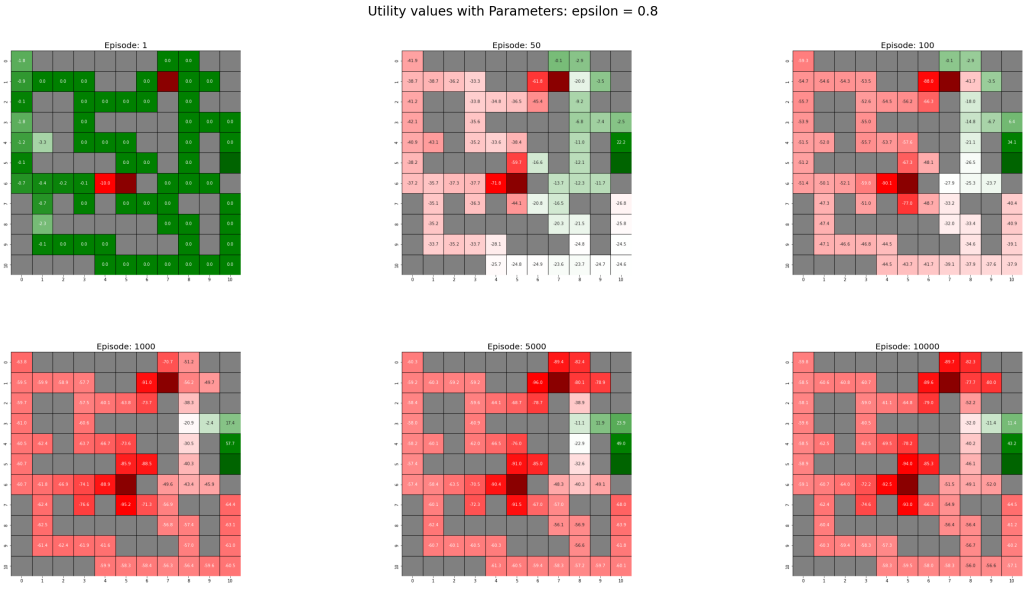

Figure 35: Utilities of the TD learning for $\epsilon = 0.8$.

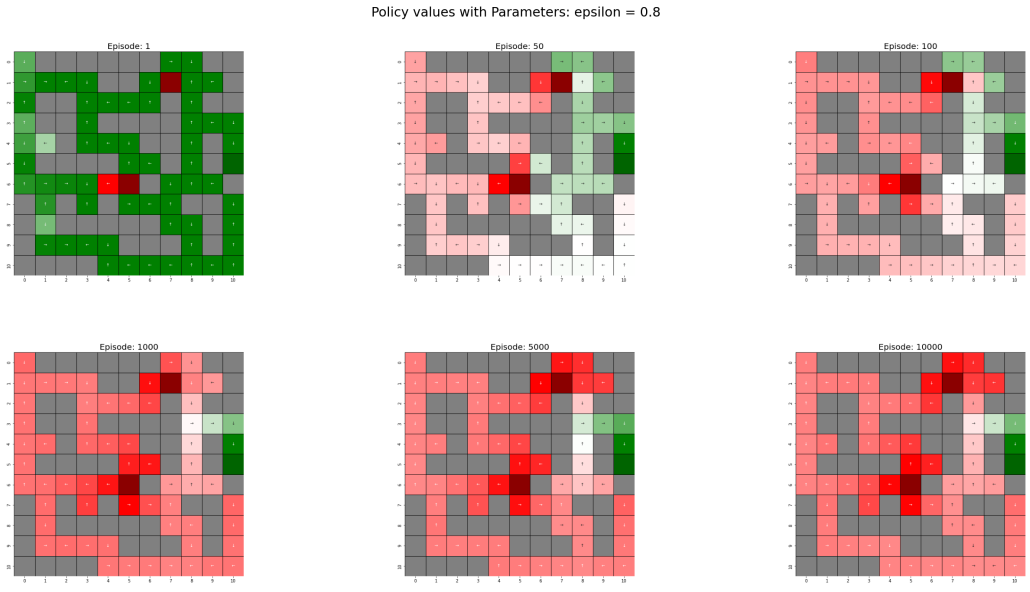

Figure 36: Policy of the TD learning for $\epsilon = 0.8$.

$\epsilon = 1$:

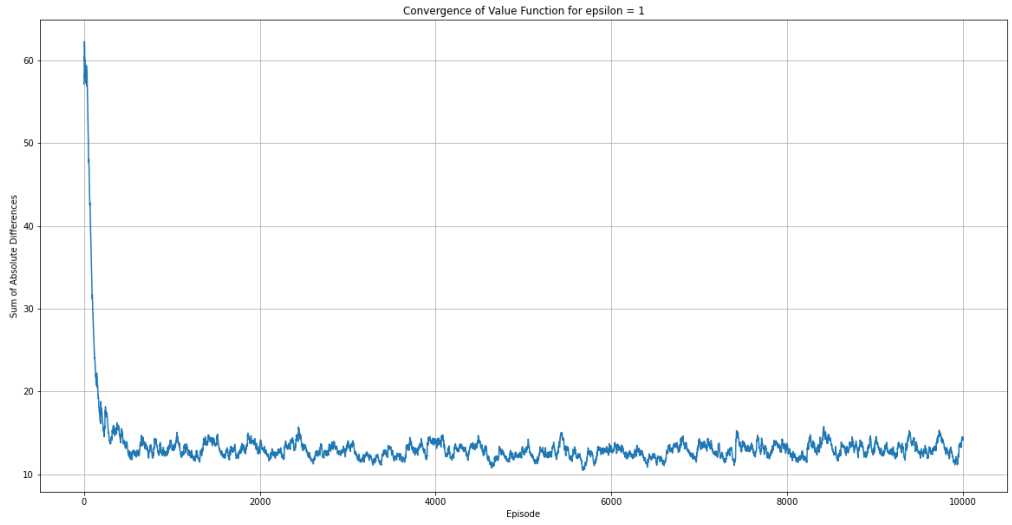

Figure 37: Convergence of the TD learning for $\epsilon = 1$.

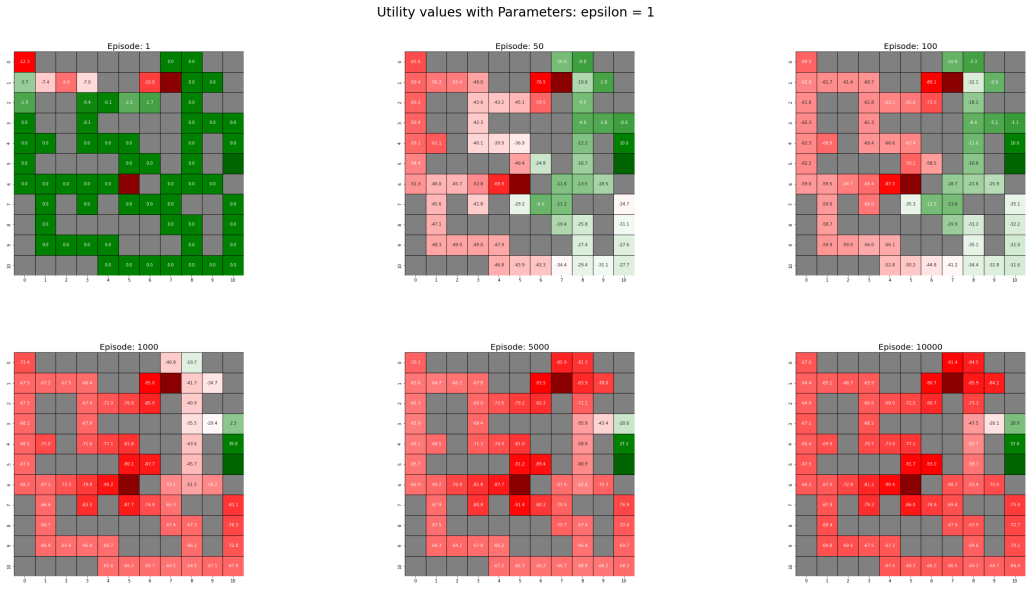

Figure 38: Utilities of the TD learning for $\epsilon = 1$.

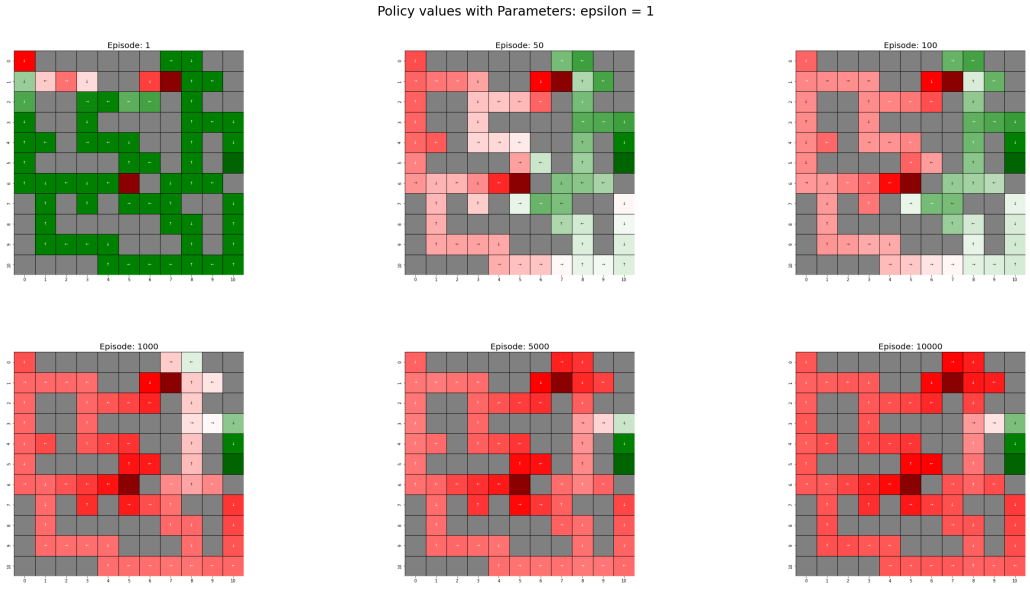

Figure 39: Policy of the TD learning for $\epsilon = 1$.

3.2 Q learning

In Figure 1, bold parameters are the default parameters. The resuls for the TD(0) learning for bold parameters is shown Figure 40, 41, 42.

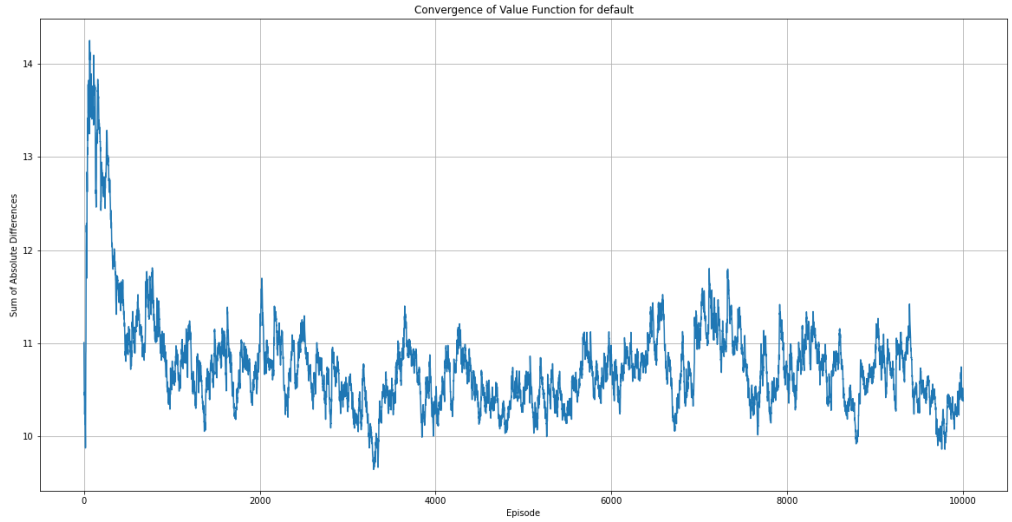

Figure 40: Convergence of the Q learning for default values.

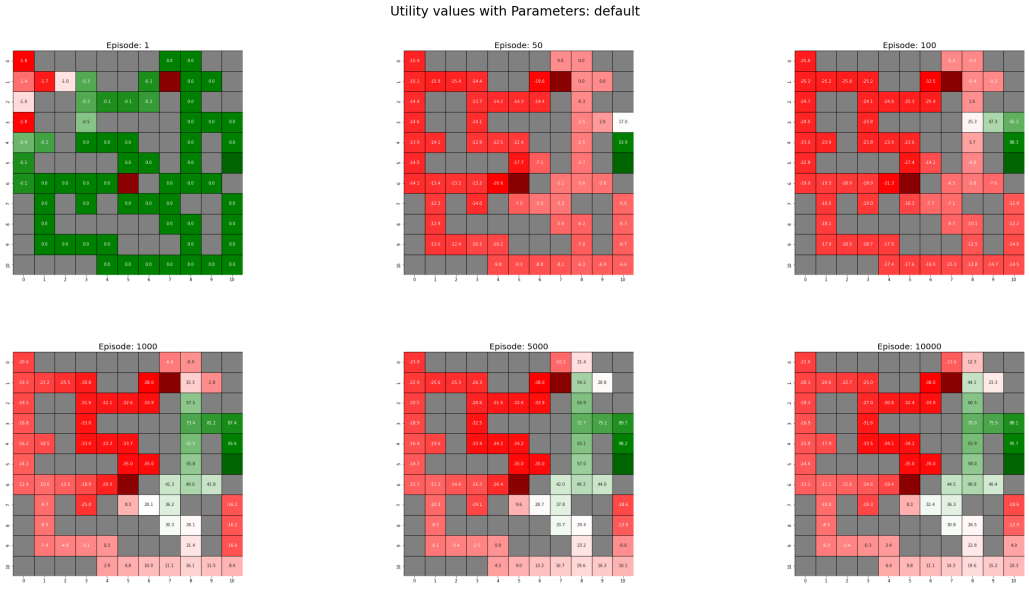

Figure 41: Utilities of the Q learning for default values.

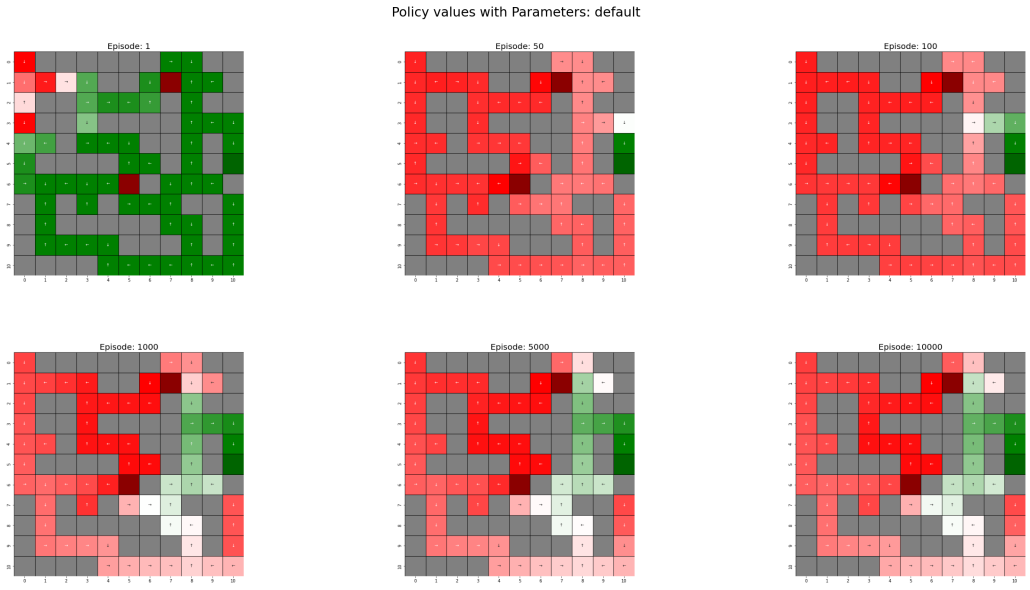

Figure 42: Policy of the Q learning for default values.

3.2.1 Learning Rate (α)

α = 0.001:

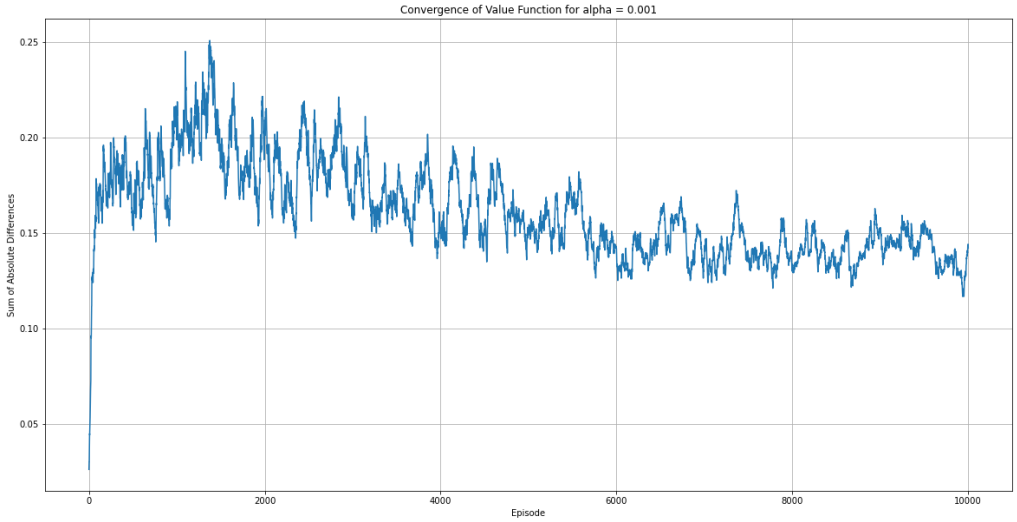

Figure 43: Convergence of the Q learning for $\alpha = 0.001$.

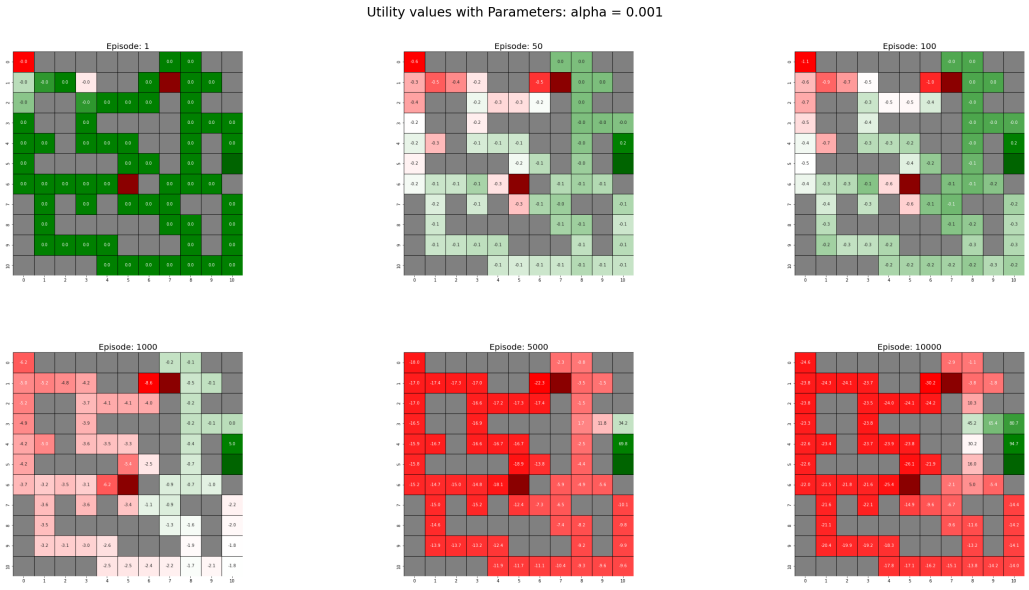

Figure 44: Utilities of the Q learning for $\alpha = 0.001$.

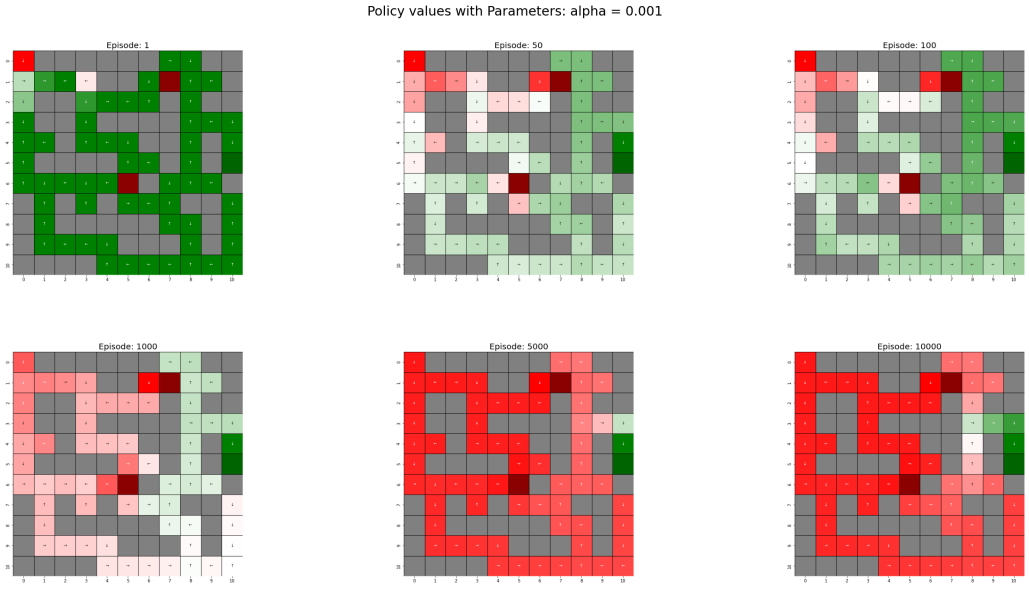

Figure 45: Policy of the Q learning for $\alpha = 0.001$.

$\alpha = 0.01$:

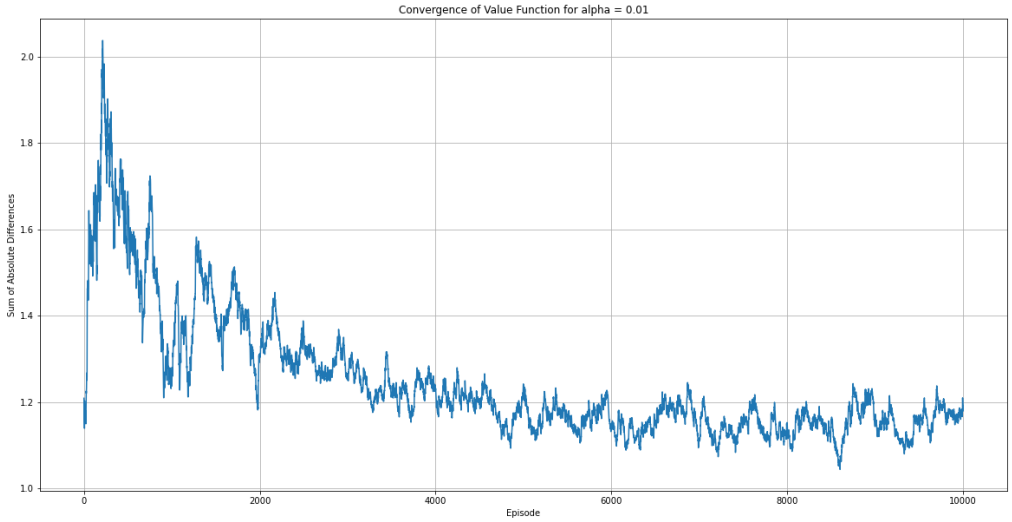

Figure 46: Convergence of the Q learning for $\alpha = 0.01$.

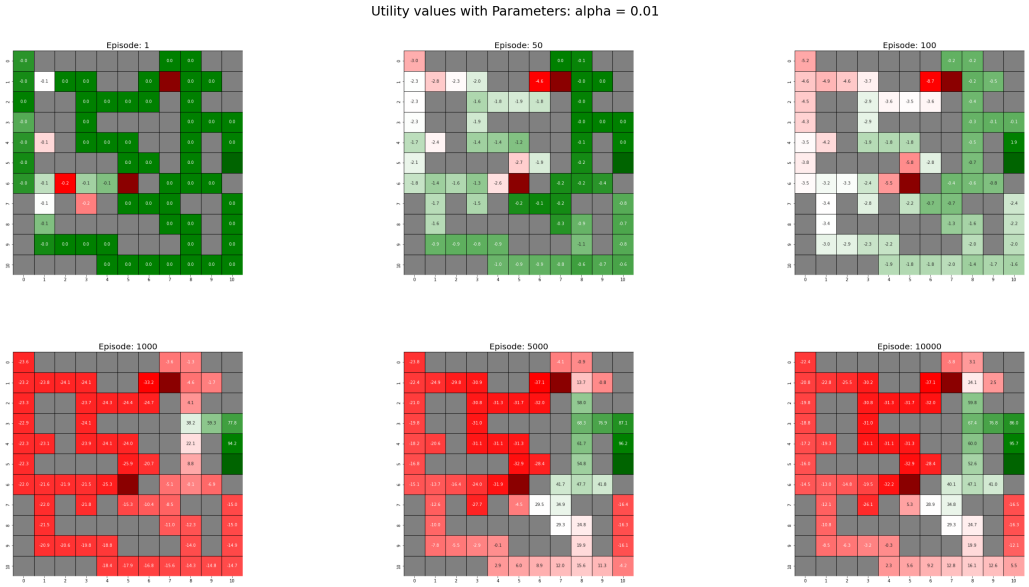

Figure 47: Utilities of the Q learning for $\alpha = 0.01$.

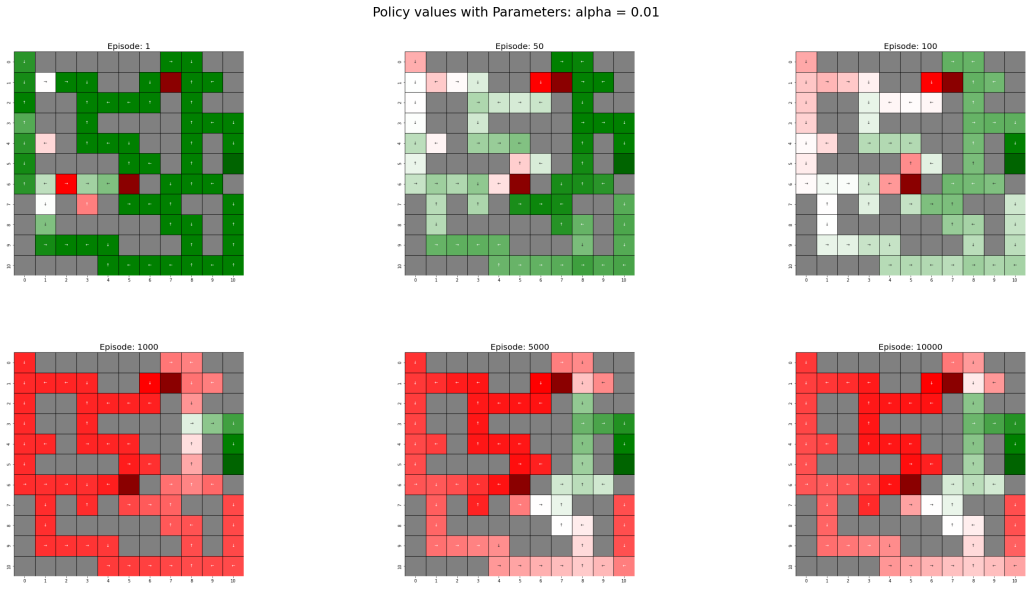

Figure 48: Policy of the Q learning for $\alpha = 0.01$.

$\alpha = 0.5$:

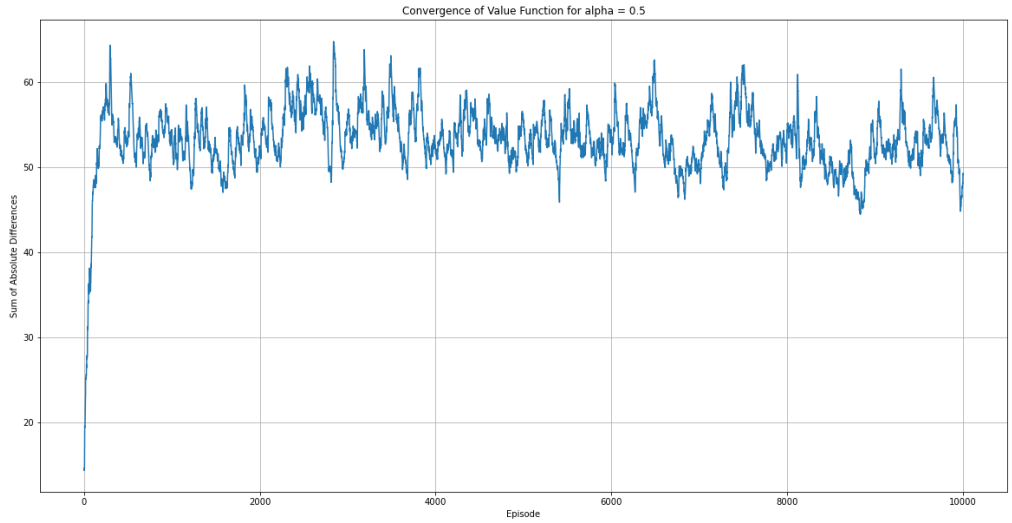

Figure 49: Convergence of the Q learning for $\alpha = 0.5$.

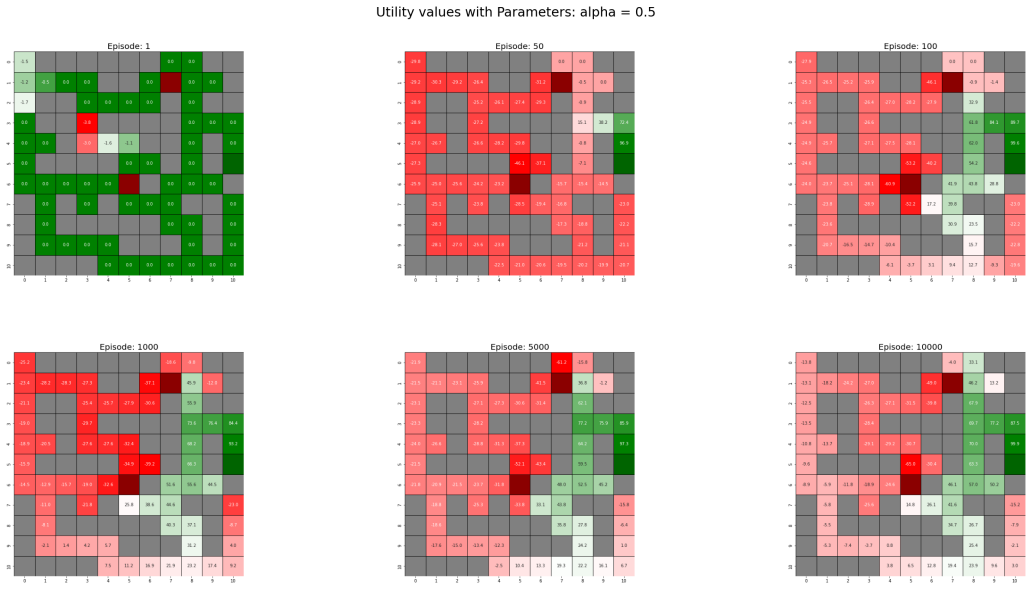

Figure 50: Utilities of the Q learning for $\alpha = 0.5$.

Figure 51: Policy of the Q learning for $\alpha = 0.5$.

$\alpha = 1$:

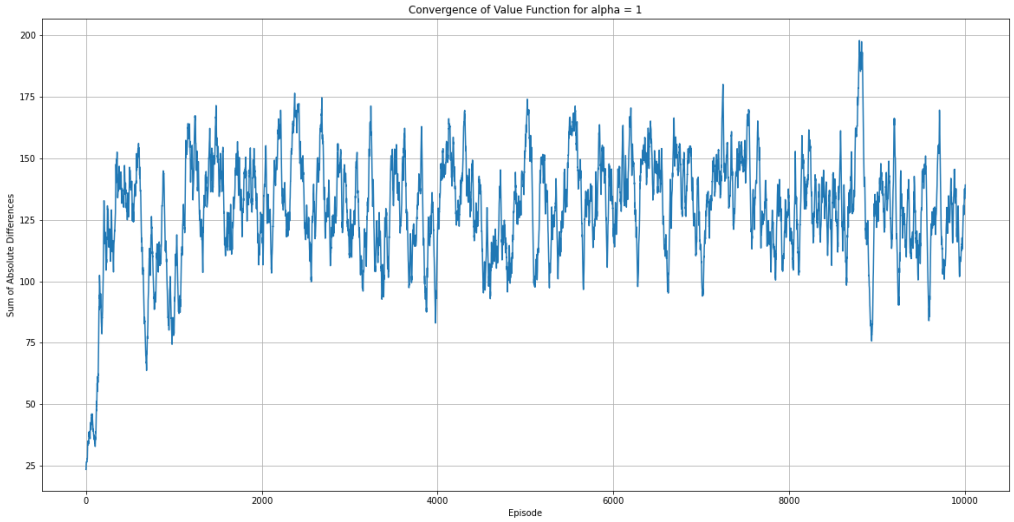

Figure 52: Convergence of the Q learning for $\alpha = 1$.

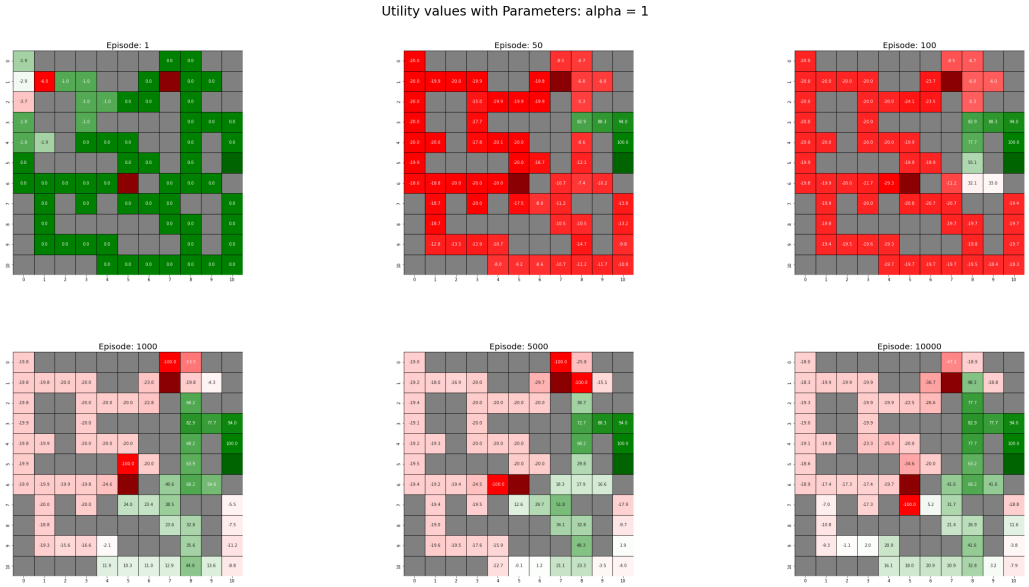

Figure 53: Utilities of the Q learning for $\alpha = 1$.

Figure 54: Policy of the Q learning for $\alpha = 1$.

3.2.2 Discount Factor (γ)

$\gamma = 0.1$:

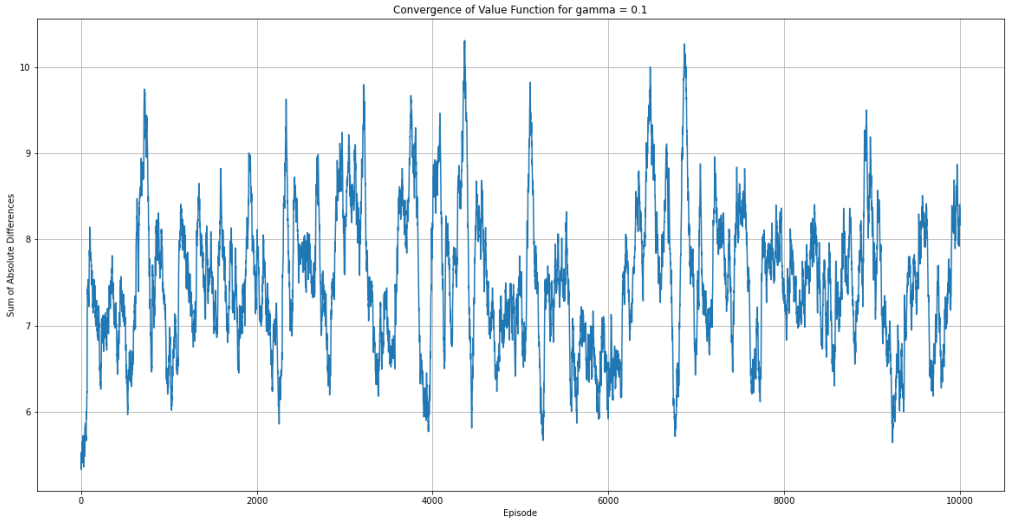

Figure 55: Convergence of the Q learning for $\gamma = 0.1$.

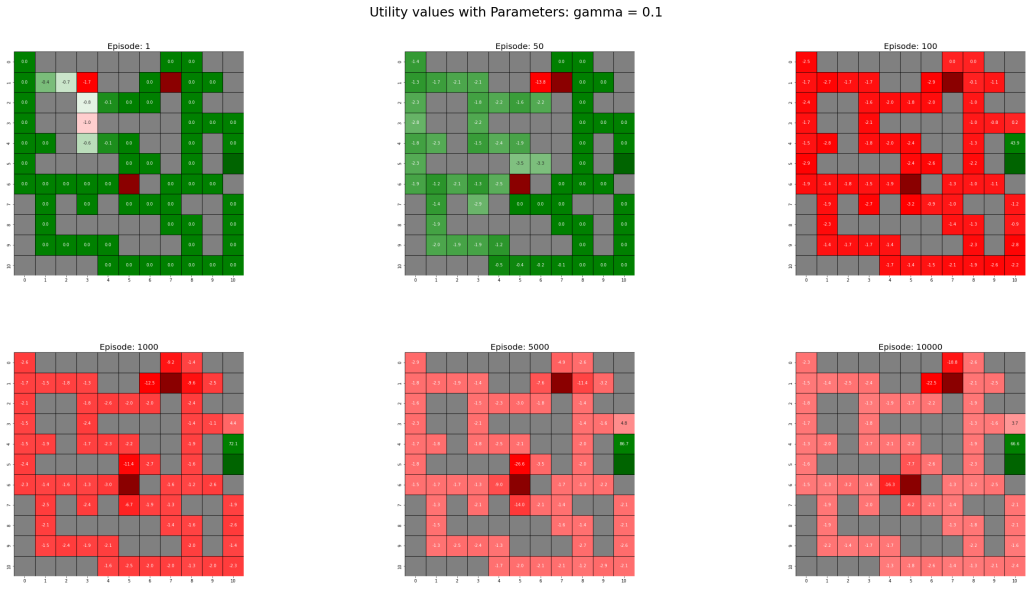

Figure 56: Utilities of the Q learning for $\gamma = 0.1$.

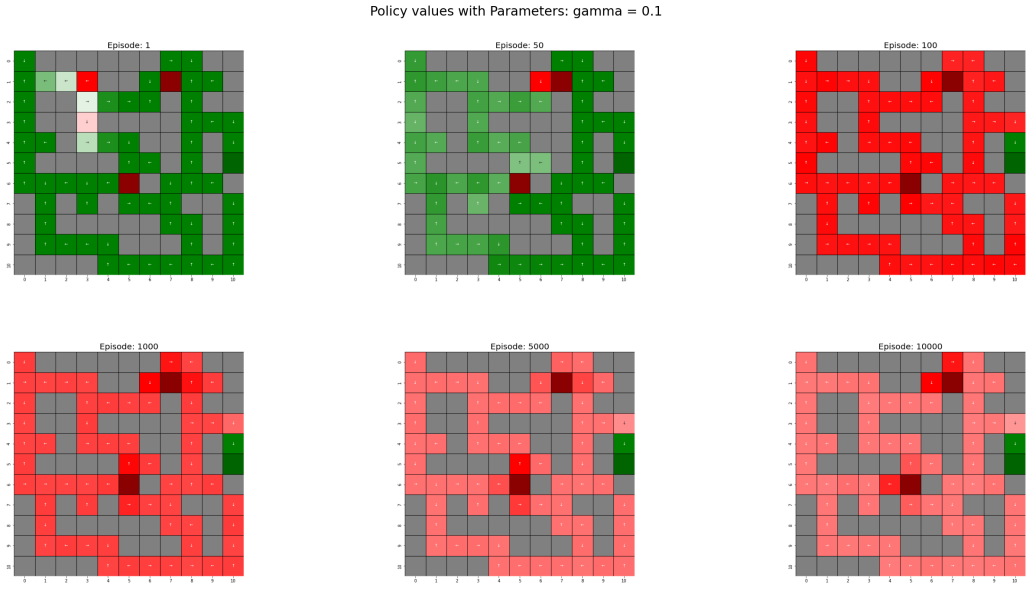

Figure 57: Policy of the Q learning for $\gamma = 0.1$.

$\gamma = 0.25$:

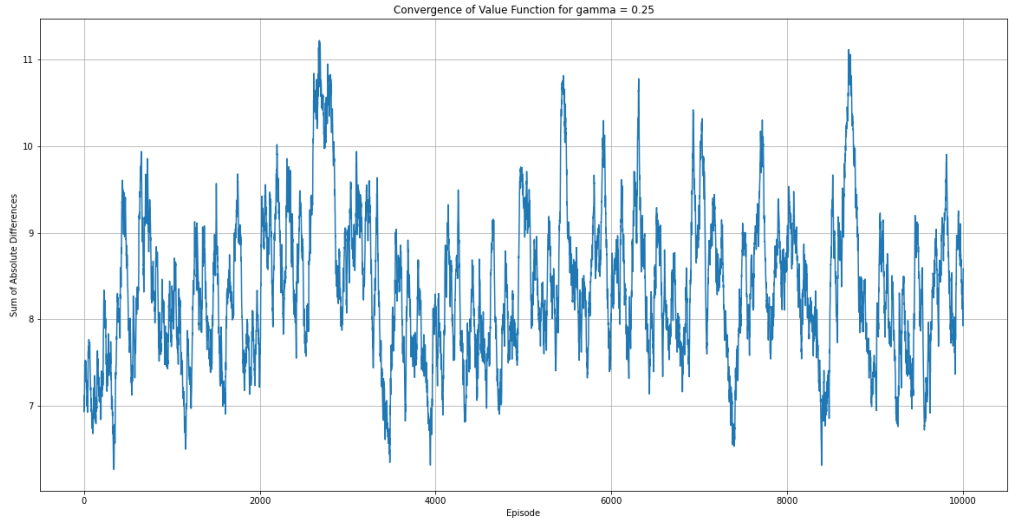

Figure 58: Convergence of the Q learning for $\gamma = 0.25$.

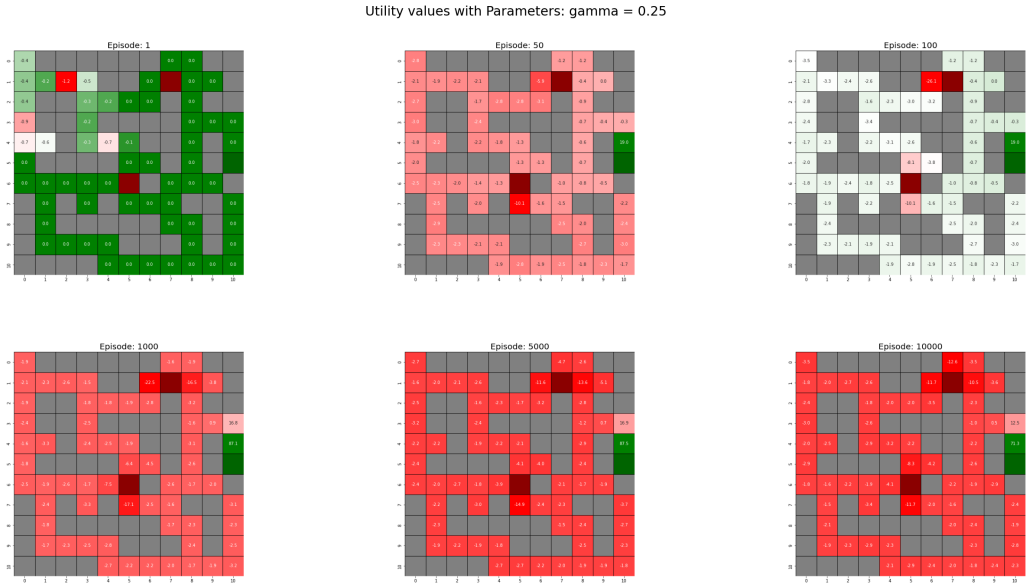

Figure 59: Utilities of the Q learning for $\gamma = 0.25$.

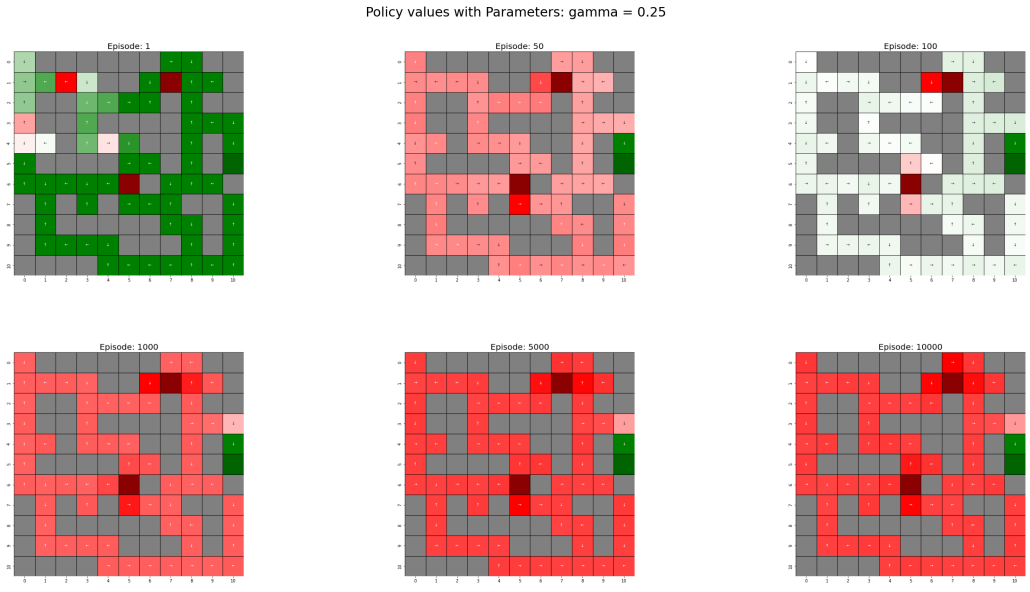

Figure 60: Policy of the Q learning for $\gamma = 0.25$.

$\gamma = 0.5$:

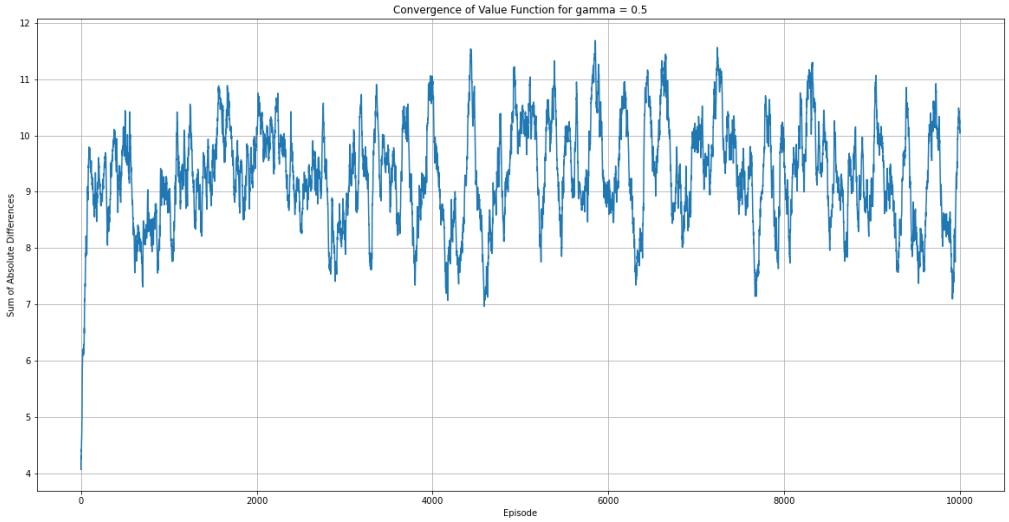

Figure 61: Convergence of the Q learning for $\gamma = 0.5$.

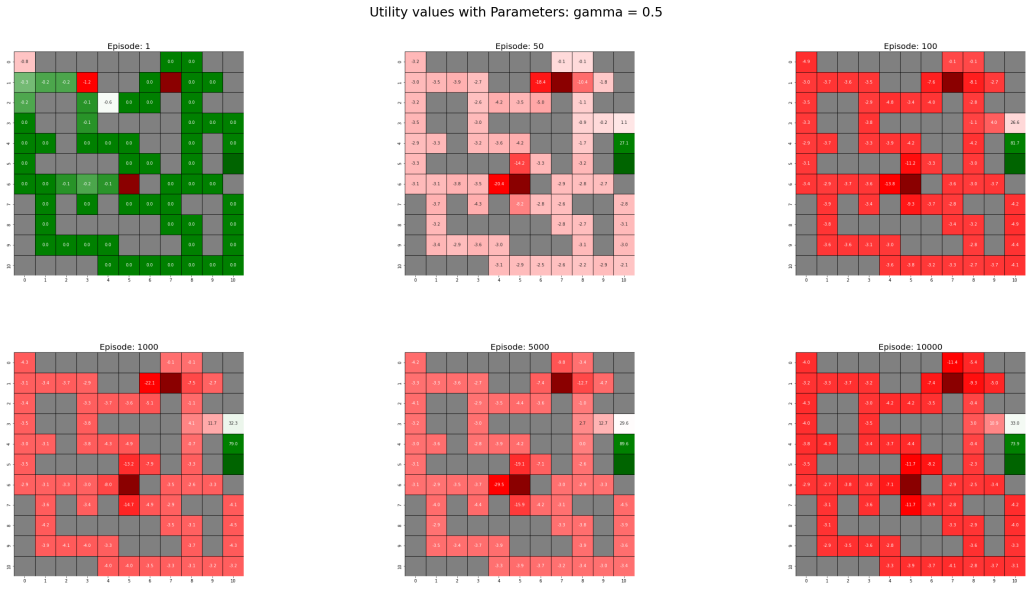

Figure 62: Utilities of the Q learning for $\gamma = 0.5$.

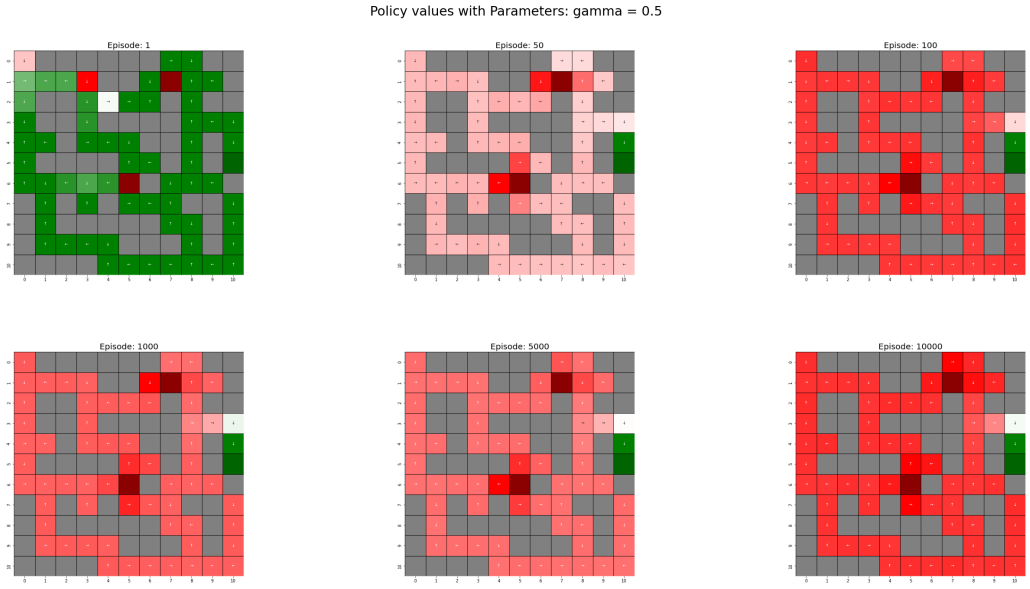

Figure 63: Policy of the Q learning for $\gamma = 0.5$.

$\gamma = 0.75$:

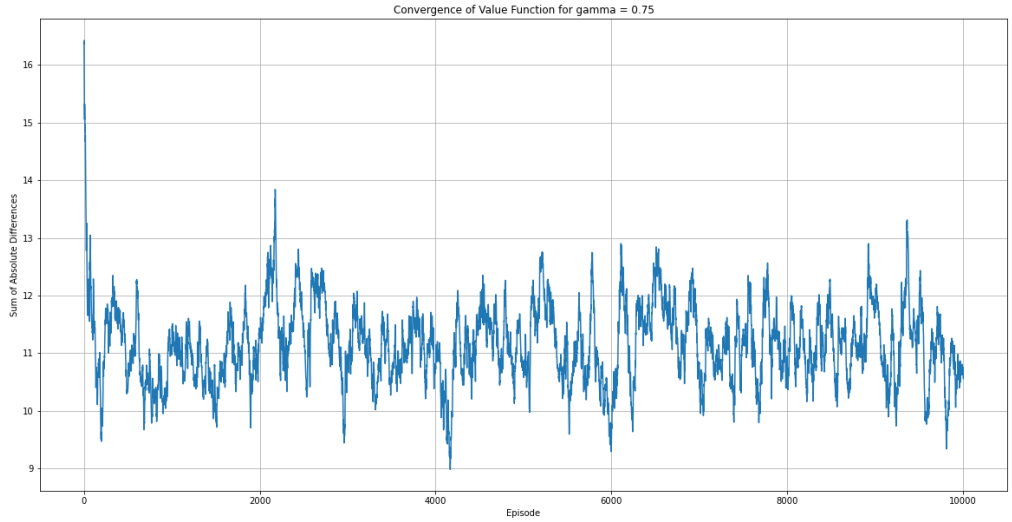

Figure 64: Convergence of the Q learning for $\gamma = 0.75$.

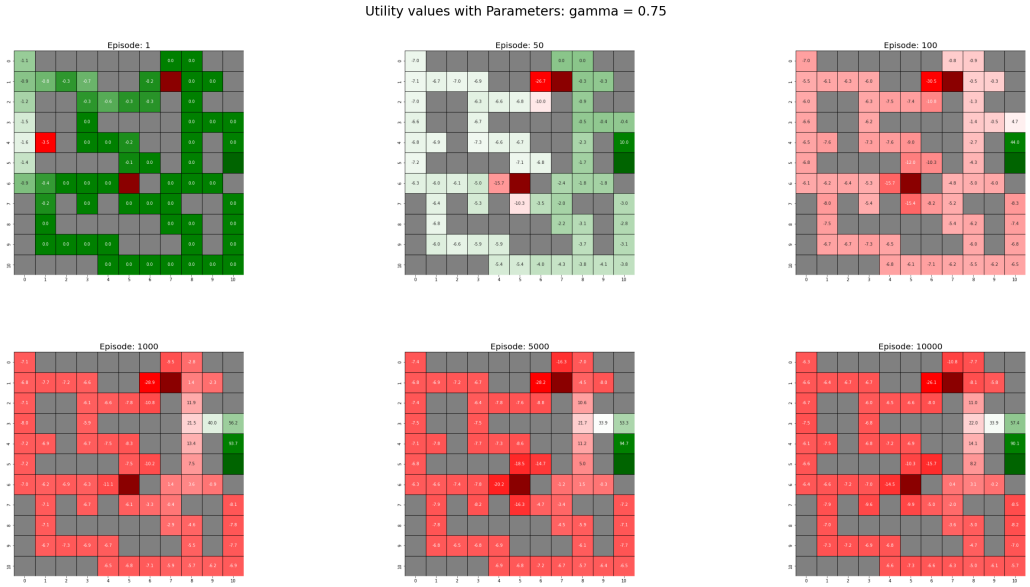

Figure 65: Utilities of the Q learning for $\gamma = 0.75$.

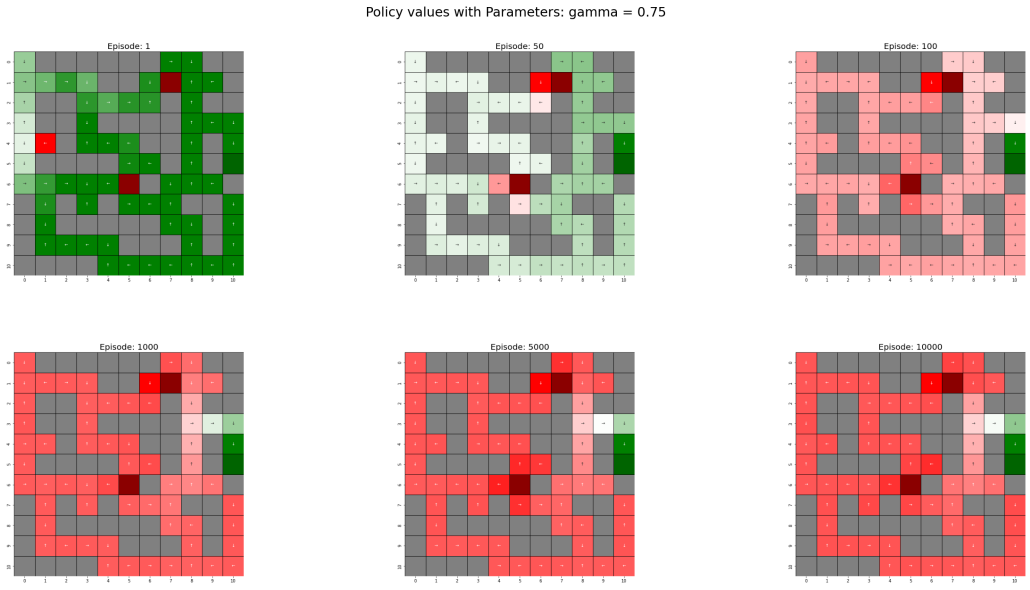

Figure 66: Policy of the Q learning for $\gamma = 0.75$.

3.2.3 Initial Exploration rate (ϵ)

$\epsilon = 0$:

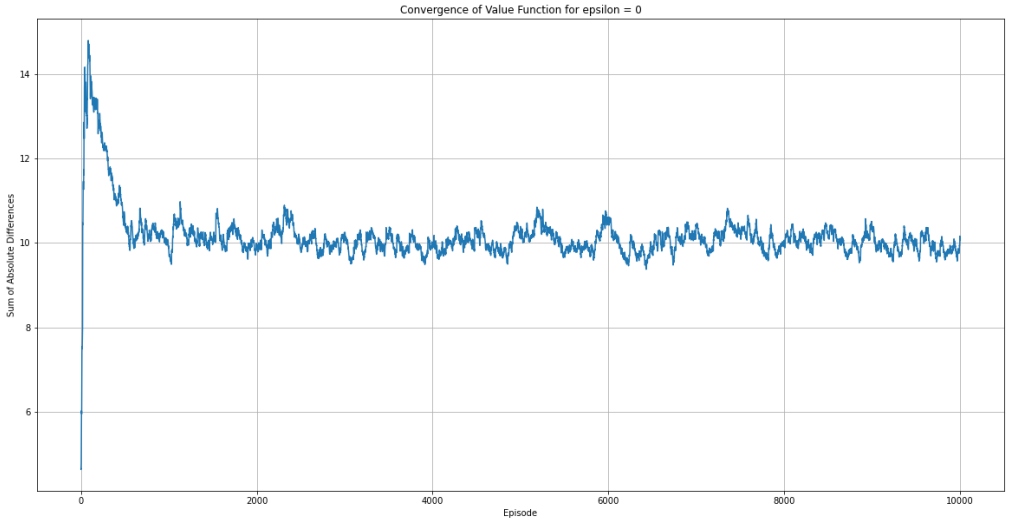

Figure 67: Convergence of the Q learning for $\epsilon = 0$.

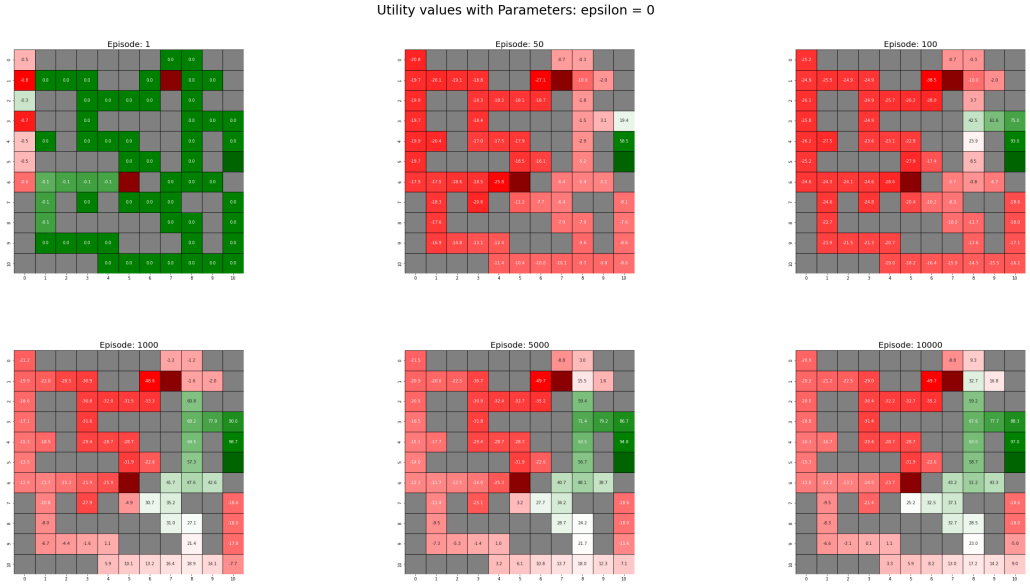

Figure 68: Utilities of the Q learning for $\epsilon = 0$.

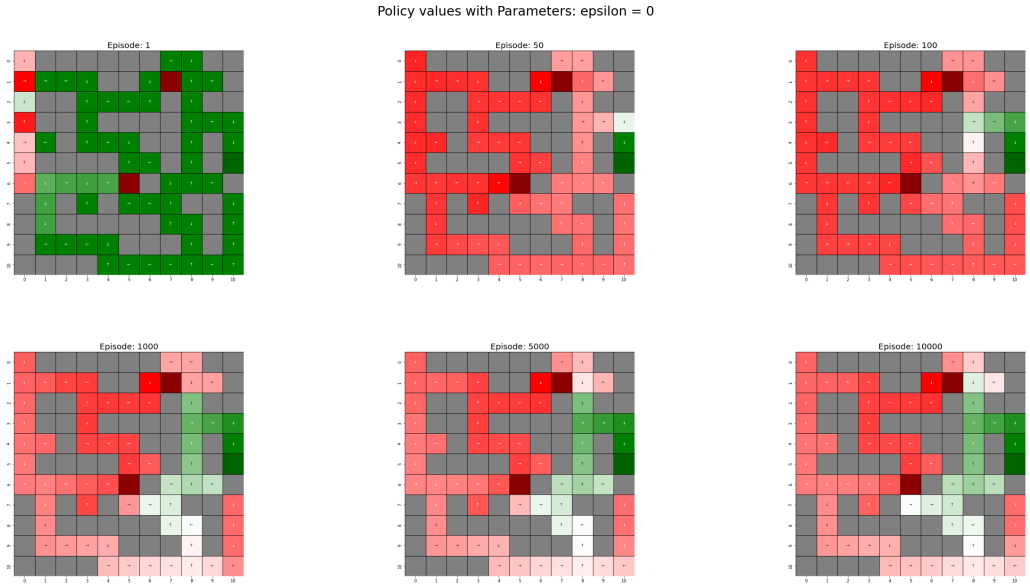

Figure 69: Policy of the Q learning for $\epsilon = 0$.

$\epsilon = 0.5$:

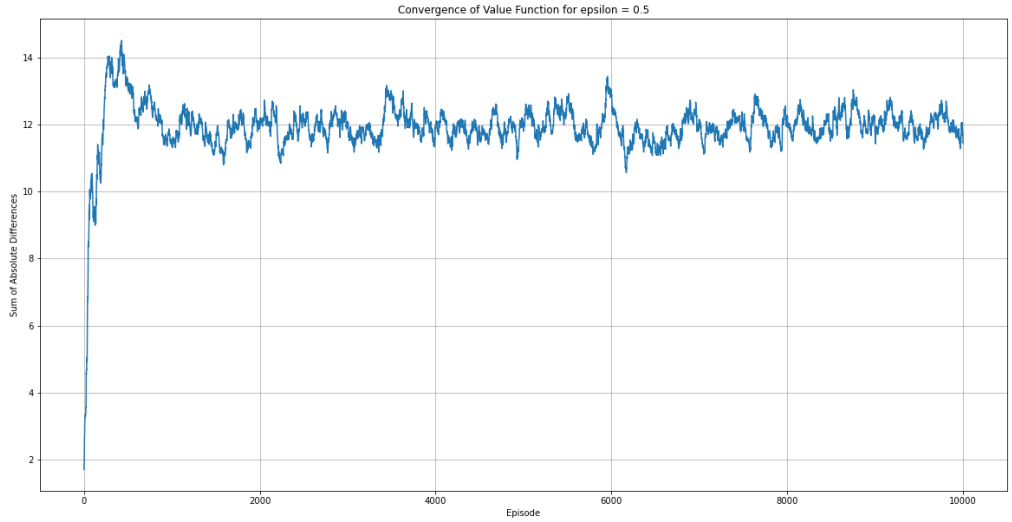

Figure 70: Convergence of the Q learning for $\epsilon = 0.5$.

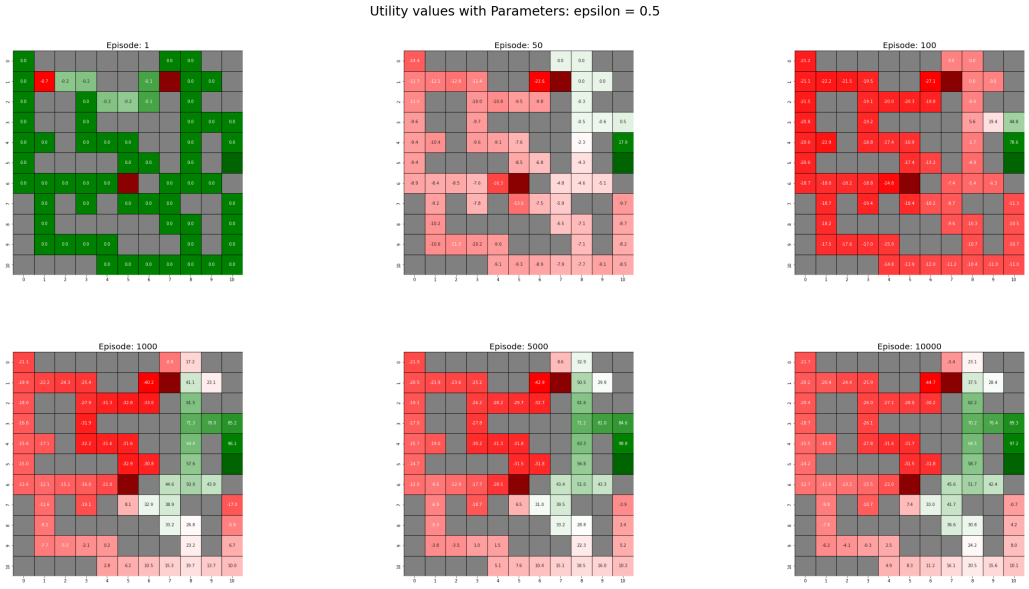

Figure 71: Utilities of the Q learning for $\epsilon = 0.5$.

Figure 72: Policy of the Q learning for $\epsilon = 0.5$.

$\epsilon = 0.8$:

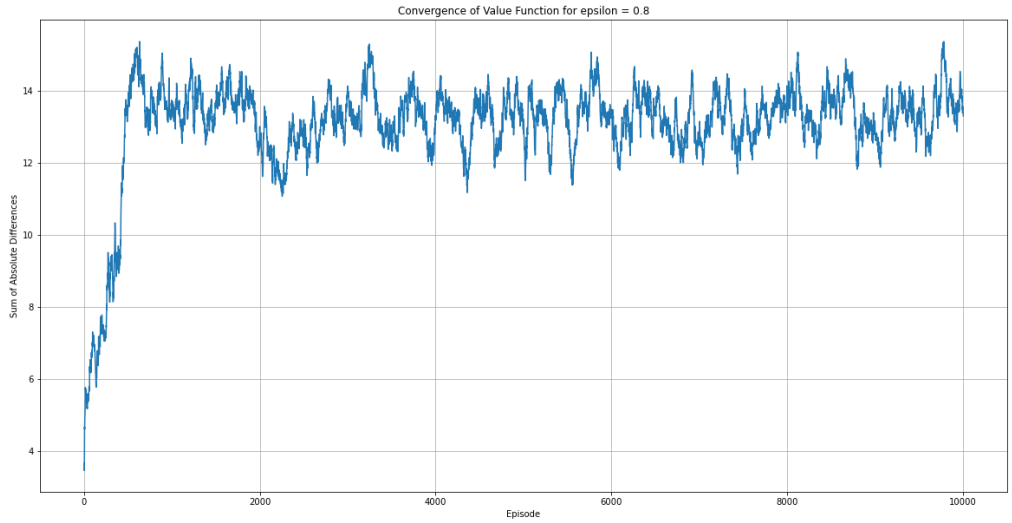

Figure 73: Convergence of the Q learning for $\epsilon = 0.8$.

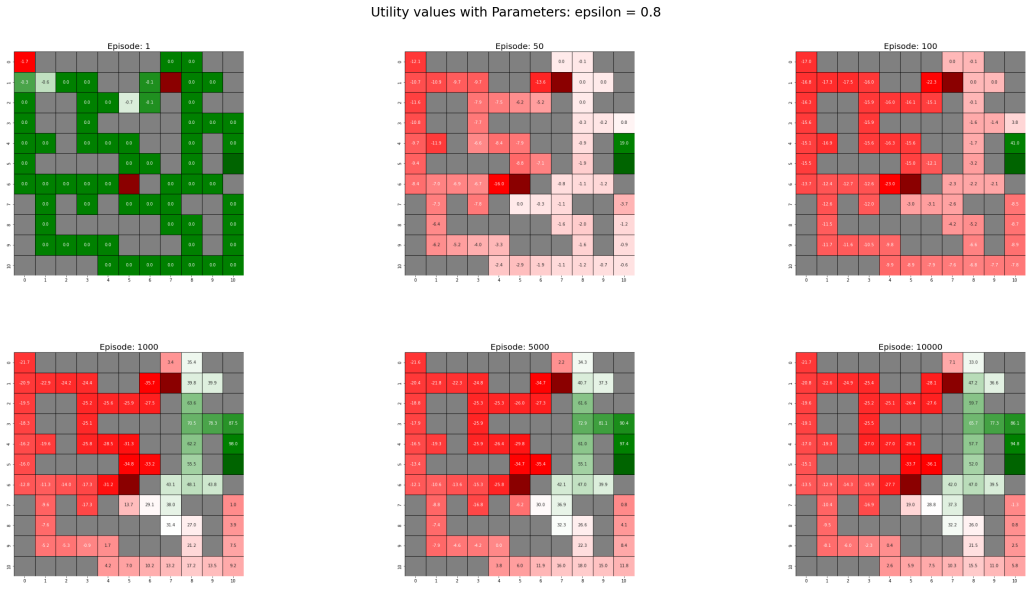

Figure 74: Utilities of the Q learning for $\epsilon = 0.8$.

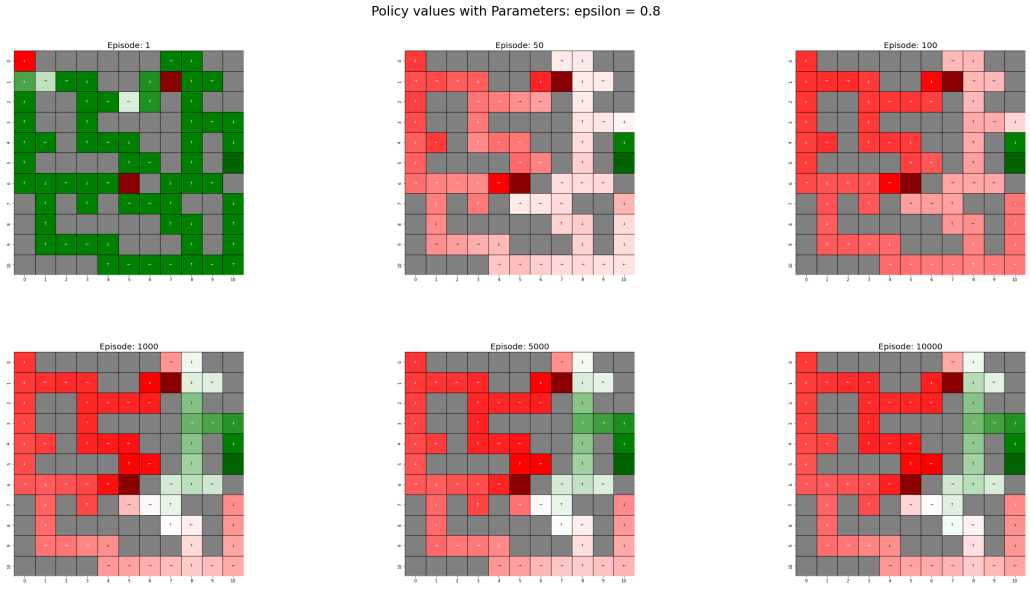

Figure 75: Policy of the Q learning for $\epsilon = 0.8$.

$\epsilon = 0.8$:

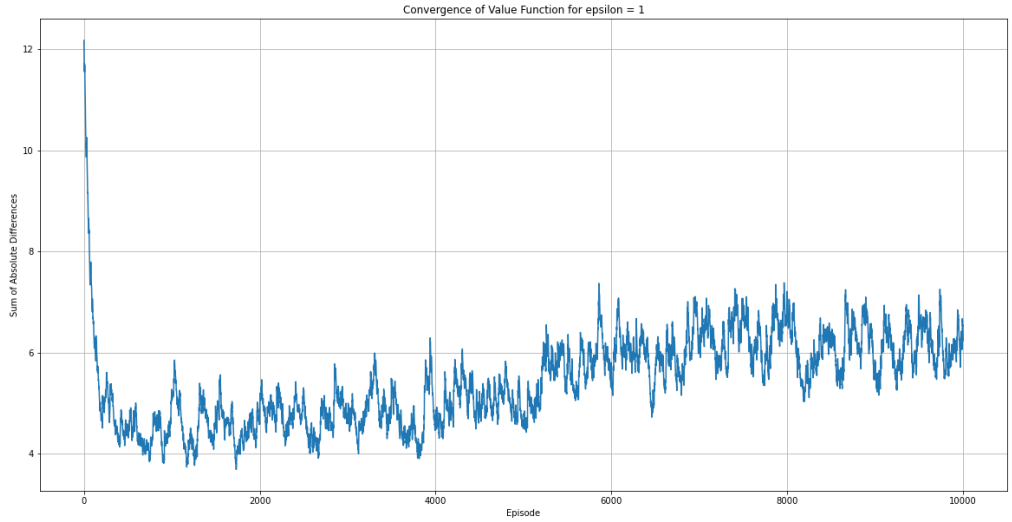

Figure 76: Convergence of the Q learning for $\epsilon = 1$.

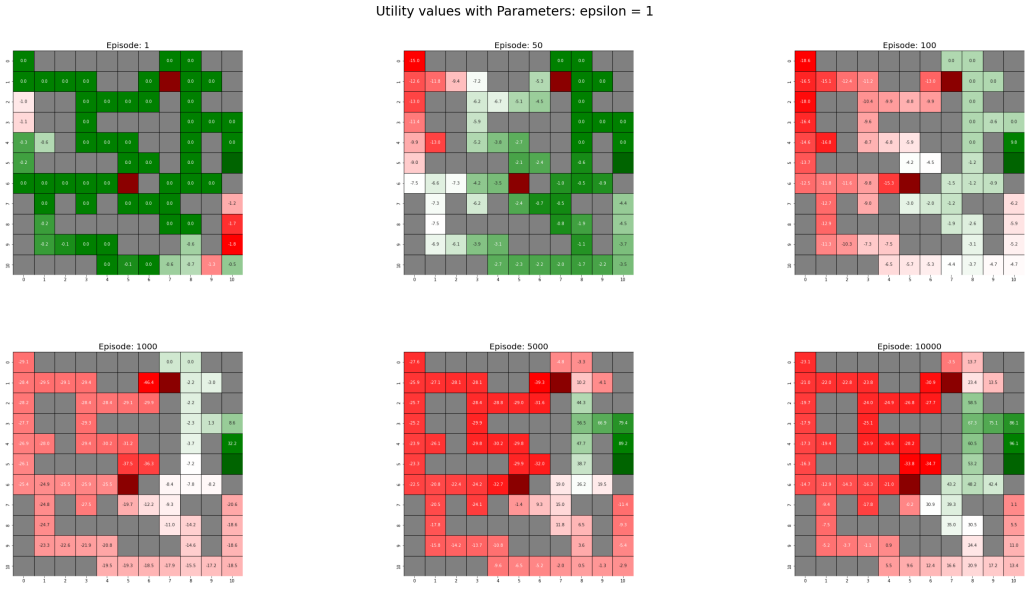

Figure 77: Utilities of the Q learning for $\epsilon = 1$.

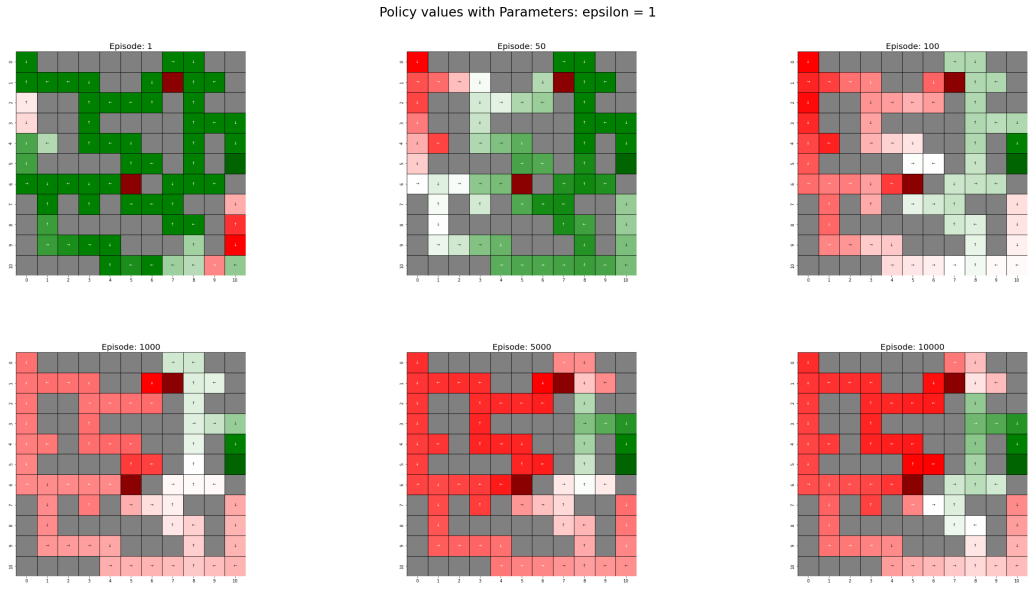

Figure 78: Policy of the Q learning for $\epsilon = 1$.

4 Discussion

In this part, answers to the questions in the Discussion part are given.

1. **Question 1:**

   Transition probabilities are essential in the sense that they simulate the stochastic nature as in real life scenario. In this sense, the outcome become more probabilistic rather than deterministic. For example, the highest probability is for the intended direction (0.75 in our case) and smaller probabilities for unintended directions.

   Reward function is also essential in the sense that the agent's moves result in with an either desirable or undesirable outcome. When the agent reaches the goal, it receives a high reward, when it falls into a trap it receives a high penalty and when it hits a wall or a boundary, the invalid move results in with a penalty. The rewards and penalties shape the learning behavior of the agent. As the agent experiment the environment, it can learn the shortest path so that the least amount of punishments received.

   Therefore, it can be said that the agents actions change over time as in a decision-making process and learning.

2. **Question 2**

   Inspecting from the Figures 2 and 3, the utility values in TD(0) in the first episode has almost arbitrary values throughout. This is because the agent has almost no information about the environment. As the episode number is increased, the agent starts to gain experience and adjust its utility values more accurately. This shows that the agent has a better understanding of the environment. From the policy figure

in 3, it can be seen by the episode 50 the agent can avoid traps and move along the goal state using the optimum path. After the episode 1000, the utility values change insignificantly. This is a sign that the agent's learning has stabilized and it has a good understanding of how to move around in the environment.

3. **Question 3**

   Yes, the utility function converges. It can be observed in 1 that the convergence plot has a very sharp fall at the first 250 episodes, indicating that learning is very intense that the utilities change significantly. After that, the utility change is not that intense. In the Figure, 2 we can see that the utility values are almost the same at the episode 5000 and episode 10000. In the episode 1000, some states near the trap are not yet converge to their final values. Therefore we may say that the convergence utilities converge around the episode 5000.

4. **Question 4**

   TD(0) learning is very sensitive to learning rate and discount factor. Firstly, in TD learning, utility values are updated through the process illustrated by the following equation, which represents online mean estimation:

$$\bar{X}_{n+1} = \bar{X}_n + \frac{1}{n+1}(X_{n+1} - \bar{X}_n)$$

   It can be observed from the equation that learning rate controls the rate of which new information is incorporated into the learning (utility values).In this manner, the learning rate (α) value is important since when it is too large, the control of the newly employed utility is prioritized ,this results in with a unstable utility map as may observed in Figure 14. Consequently, the utilities does not converge and the agent cannot learn the optimal path towards the goal. On the other learning with a low (α) value as shown in Figure **??** causes the agent to incorporate new information very slowly. This results in a slower convergence to the final utility values, requiring more episodes for the agent to find the optimal path. In this case, newer utility updates have minimal impact, increasing the learning process. Therefore, TD(0) learning is very sensitive to learning rate.

   Secondly, the effect of discount factor (γ) can be seen in the equation as follows:

$$U(S_t) \leftarrow U(S_t) + \alpha\left(R_{t+1} + \gamma U(S_{t+1}) - U(S_t)\right)$$

   Inspecting from the equation, the discount factor determines how much the future rewards are values compared to immediate rewards. Therefore when the discount factor (γ) is close small, the agent values immediate rewards over long-term benefits which results in with a faster learning. However it can be seen in Figure **??** that the agent does not learn the optimum path properly. On the other hand when discount factor is large, the agent values future rewards as much as immediate ones which encourages the agent to learn the optimum path with a possibly slower learning rate.

5. **Question 5:**

   The most significant challenge in implementing TD(0) learning arose when using a learning rate (α) of 1. This high learning rate led to an unstable utility map, causing the agent to frequently get stuck in certain parts of the maze, which significantly increased the running time. To address this issue, I introduced a penalty for the agent whenever it remained in the same state for an extended period. This adjustment effectively reduced the running time by encouraging to avoid repetitive loops.

6. **Question 6:**

   The Q learning converged faster (stabilized) than TD(0) learning as can be observed in the hyperparameter convergence graphs in this report. This is because in Q learning agent learns for each state-action pair. This results in with a quicker identification on the environment. On the other hand in TD(0) learning, the agent does not use the action-utility pair. Thus the convergence is slower because the agent does not always choose the optimal future action immediately.

7. **Question 7:**

   The ϵ-greedy strategy allows the agent to explore the environment by choosing random actions with a probability of ϵ and the best-known action with a probability of 1-ϵ. This strategy ensures that the agent does not become trapped in a local optimum early in the learning process by repeatedly choosing the same actions, and instead explores potentially better options that might initially appear not that optimal.

   The effect of exploration on TD(0) learning can be observed very easily. When the exploration has high values such as $\epsilon = 0.5$, 0.8, and 1, the agent cannot find the optimal path. On the other hand, in Q learning, the agent can still find the optimal path for low or high exploration rates, only the convergence occurs later for too high exploration rate.

   We prefer high exploration rates at the beginning and low exploration rates later. Therefore exploration effects the performance positively initially since it allows exploration of the environment. Then it effects the performance negatively later since the agents should move based on the learned path.

8. **Question 8:**

   In our maze, both TD(0) and Q learning can successfully find the optimal path with default hyperparameters since the maze is simple with only one goal state. However, I observed that Q learning has better performance in the hyperparameter tests than TD(0) learning. I would prefer Q learning since it's policy is based on action-state pairs, thus converges faster with better knowledge on the environment with small number of episodes. In more complex maze structures, Q learning be a robust choice and thus can outperform TD(0) learning. However for simpler maze structures TD(0) learning is still successful with the correct hyperparameters.

9. **Question 9:**

To enhance the learning process, additional rewards could be strategically placed along the path to the goal, thereby motivating the agent to reach its target faster and more efficiently.

For a more challenging setup, the maze complexity could be increased by introducing additional pathways to the goal along with potential traps. Expanding the maze size and incorporating dead ends would challenge the agent to learn navigation strategies. Introducing random starting positions within the maze can also increase the difficulty, requiring the agent to adapt its strategy based on varying initial conditions. These modifications would make the learning process more challenging and reflective of real-world navigation challenges.

10. **Question 10:**

    One way of increasing the performance and reliability of learning can be using adaptive ϵ rate strategy as discussed in lecture notes. This can lead to the maximum performance by adjusting the exploitation and exploration dynamically. In adaptive ϵ rate approach the random actions are prioritized at the beginning to encourage the agent to learn the environment more aggressively. Then as the utility values converge, the exploration rate can be dynamically decreased to encourage exploitation in the learning. Thus maximizing the rewards with the optimal path toward the goal.

5 Python code

```
1
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon May 13 16:59:21 2024
5
6  @author: defne
7  """
8
9  import utils
10 import numpy as np
11 from matplotlib import pyplot as plt
12 #from animation import animation_gif used to capture the movement of the agent
       along the maze, then commented
13 import cv2
14 import copy
15
16 class MazeEnvironment:
17
18     def __init__(self):
19         #Define the maze layout, rewards, action space (up, down, left, right)
20
21         self. maze = maze
22         self.start_pos = (0,0) # Start position of the agent
23         self.current_pos = self.start_pos
24
25         self.state_penalty = -1
```

```
        self.trap_penalty = -100
        self.goal_reward = 100

        self.actions = {0: (-1, 0), #Move up
                        1: (1, 0),  #Move down
                        2: (0,- 1), #Move left
                        3: (0, 1)   #Move right
                        }

    def reset(self):
        self.current_pos = self.start_pos
        return self.current_pos

    def step(self, action):
        '''
        Moving the agent according to an action.
        Due to the probabilistic nature of the environment the agent may move different than the chosen direction
        Probabilities:

        * Probability of going for the chosen direction: 0.75
        * Probability of going opposite of the chosen direction: 0.05
        * Probability of going each of perpendicular routes of the chosen direction: 0.10

        '''
        done = False
        reward = 0

        # Defining the possible moves
        action = self.actions[action] # Chosen action

        opposite = (-action[0], -action[1])
        perp_right = (action[1], -action[0])
        perp_left = (-action[1],  action[0])

        moves = [action, opposite, perp_right, perp_left]
        probabilities = [0.75, 0.05, 0.10, 0.10] # chosen , opposite, perpendicular respectively.

        chosen_action = moves[np.random.choice(len(moves), p=probabilities)]

        #New position is calculated
        new_position = (self.current_pos[0] + chosen_action[0], self.current_pos[1] + chosen_action[1])

        # Check boundaries and whether the new position is an obstacle
        if (0 <= new_position[0] < self.maze.shape[0] and  #Horizatonal boundaries
            0 <= new_position[1] < self.maze.shape[1] and  #Vertical boundaries
            self.maze[new_position[0], new_position[1]] != 1): #Not an obstacle

```

```
            self.current_pos = new_position
        #Next position is on the boundaries
        else:
            reward += -5 #for discouraging the agent to stay at its current state due to hitting the boundaries

        # If new position is out of bounds or an obstacle, stay in current position
        #current position is maintained

        #Reward is calculated
        cell = self.maze[self.current_pos[0], self.current_pos[1]]
        if cell == 0:
            reward += self.state_penalty
        # cell == 1 is a boundary
        if cell == 2:
            reward += self.trap_penalty
            done = True
        if cell == 3:
            reward += self.goal_reward
            done = True
        # Boundary or obstacle control
        return self.current_pos , reward, done

class MazeTD0(MazeEnvironment): #Inherited from MazeEnvironment
    def __init__(self, maze, alpha = 0.1, gamma = 0.95, epsilon = 0.2, episodes = 10000):
        super().__init__()

        self.maze = maze
        self.alpha = alpha #Learning rate
        self.gamma = gamma #Discount factor
        self.epsilon = epsilon #Exploration rate
        self.episodes = episodes
        self.utility =  np.zeros(self.maze.shape) #Encourage exploration

        # self.utility[maze == 2] = -1000 #trap
        # self.utility[maze == 3] = 1000 #goal
        self.utility[maze == 1] = -1000 #boundary

        self.convergence_history = []

    def choose_action(self, state):
        #Explore and Exploit: Choose the best action based on current utility values
        # Discourage invalid moves

        #Exploration:
        if np.random.rand() < self.epsilon: #any action
            return np.random.choice(list(self.actions.keys()))

        #Exploitation:
        else:
            #Geting the utilities of all possible actions from current state
```

```
            possible_actions = {}
            #Invalid moves are not considered in the exploitation
            for action, move in self.actions.items():
                next_pos = (state[0] + move[0], state[1] + move[1])
                if 0 <= next_pos[0] < self.maze.shape[0] and 0 <= next_pos[1]
    < self.maze.shape[1]:
                    possible_actions[action] = self.utility[next_pos]

            #Choose the action with the highest utility
            return max(possible_actions, key = possible_actions.get) if
    possible_actions else None

    def update_utility_value(self, current_state, reward, new_state):
        current_value = self.utility[current_state]

        new_value = self.utility[new_state]

        new_value = current_value + self.alpha * ((reward + self.gamma *
    new_value) - current_value)

        #TD(0) update formula
        self.utility[current_state] = new_value

    def run_episodes(self):

        selected_episodes = [0, 49, 99, 999, 4999, 9999]
        utility_filenames = []
        policy_filenames = []
        mask = (maze != 2) & (maze != 3)

        # starting_episode = 200
        # ending_episode = 201
        # i = 0

        for episode in range(self.episodes):
            print("Episode:",episode)
            state = self.reset()

            prev_utility = np.copy(self.utility) #before the update

            done = False
            while not done:
                # if (starting_episode <= episode <= ending_episode):
                    # i += 1
                    # plt.clf()
                    # plt.imshow(self.maze, cmap='hot', interpolation='nearest
    ')
                    # plt.scatter(self.current_pos[1], self.current_pos[0], c
    ='blue', s=100)  # Agent's position
                    # plt.title(f'Maze_{episode}')
                    # if i<=9:
                    #     plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024
    SpringSemester\\EE449\\HW3\\animation_plots\\figure_000{i}")   # save the
    figure to file
```

```
                    # elif 10 <= i <= 99:
                    #     plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024 SpringSemester\\EE449\\HW3\\animation_plots\\figure_00{i}")   # save the figure to file
                    # elif 100 <= i <= 999:
                    #     plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024 SpringSemester\\EE449\\HW3\\animation_plots\\figure_0{i}")   # save the figure to file
                    # else:
                    #     plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024 SpringSemester\\EE449\\HW3\\animation_plots\\figure_{i}")   # save the figure to file

                state = self.current_pos
                action = self.choose_action(state)
                next_state, reward, done = self.step(action)

                self.update_utility_value(state, reward, next_state)


            # Calculate the sum of absolute differences from the previous utility function
            diff = np.sum(np.abs(np.copy(self.utility[mask]) - prev_utility[mask]))
            self.convergence_history.append(diff)

            # if episode in selected_episodes:

            #     self.utility[maze == 2] = -1000
            #     self.utility[maze == 3] = 1000

            #     ax = utils_update.plot_value_function(self.utility, self.maze, episode)
            #     filename_1 = f'Utility_Episode_{episode + 1}_alpha_0_001.png',
            #     ax.figure.savefig(filename_1)
            #     utility_filenames.append(filename_1)
            #     plt.close()

            #     ax = utils_update.plot_policy(self.utility, self.maze, episode)
            #     filename_2 = f'Policy_Episode_{episode + 1}_alpha_0_001.png'
            #     ax.figure.savefig(filename_2)
            #     policy_filenames.append(filename_2)
            #     plt.close()

            #     self.utility[maze == 2] = 0
            #     self.utility[maze == 3] = 0

        plt.show()
        return self.utility, utility_filenames, policy_filenames

def plot_episodes(images, parameters, utility_or_policy):
    num_rows = 2
```

```
    num_cols = 3
    #plt.figure(fig_size=(24,12))
    fig, axs = plt.subplots(num_rows, num_cols, figsize=(40,20)) # Adjust the figsize based on your requirement

    for i, img_path in enumerate(images):
        img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
        row = i // num_cols  # Calculate row index
        col = i % num_cols   # Calculate column index
        axs[row, col].imshow(img)
        #axs[row, col].set_title(f'Episode: {episodes[i]}') # Title with generation
        axs[row, col].axis('off') # Hide axis

    plt.suptitle(f"{utility_or_policy} values with Parameters: {parameters}", fontsize = 30) # Main title with parameter
    plt.tight_layout()
    plt.show()

    return fig

def exponential_moving_average(history, alpha):
    ema = [history[0]]
    for x in history[1:]:
        ema.append(alpha * x + (1-alpha) * ema[-1])
    return ema

#maze layout
maze = np.array([
    [0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1],
    [0, 0, 0, 0, 1, 1, 0, 2, 0, 0, 1],
    [0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1],
    [0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0],
    [0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 3],
    [0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0],
    [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0],
    [1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
])

alpha = 1
gamma = 0.95
epsilon = 0.2
episodes = 10000

maze_td0 = MazeTD0(maze, alpha, gamma, epsilon, episodes)
final_values, utility_image, policy_image = maze_td0.run_episodes()

history = maze_td0.convergence_history
#history_smoothed = moving_average(history, 50)
history_smoothed = exponential_moving_average(history, 0.02)

```

```
parameter = 'alpha = 1'
plt.figure(figsize=(20, 10))
episodes = np.arange(1,10001) # Episodes from 1 to 10000
plt.plot(episodes,history_smoothed)
plt.xlabel('Episode')
plt.ylabel('Sum of Absolute Differences')
plt.title(f'Convergence of Value Function for {parameter}')
plt.grid(True)
plt.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3\\
    TD_learning\\alpha\\alpha_1\\Convergence_for_parameter_{parameter}.png')
plt.show()

fig1 = plot_episodes(utility_image, parameter, 'Utility')
fig1.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3
    \\TD_learning\\alpha\\alpha_1\\Episodes_utility_{parameter}.png')

fig2 = plot_episodes(policy_image, parameter, 'Policy')
fig2.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3
    \\TD_learning\\alpha\\alpha_1\\Episodes_policy_{parameter}.png')

utils.plot_value_function(final_values, maze)
utils.plot_policy(final_values, maze)

# animation_gif('C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\
    HW3\\animation_plots', 9998, 9999)

class MazeQLearning(MazeEnvironment): #Inherited from MazeEnvironment
    def __init__(self, maze, alpha = 0.1, gamma = 0.95, epsilon = 0.2,
    episodes = 10000):
        super().__init__()

        self.maze = maze
        self.alpha = alpha #Learning Rate
        self.gamma = gamma #Discount factor
        self.epsilon = epsilon #Exploration Rate
        self.episodes = episodes
        self.q_table = np.zeros((*maze.shape, 4))  # Assuming 4 actions: up,
    down, left, right

        self.convergence_history = []
    def choose_action(self,state):

        #Exploration:
        if np.random.rand() < self.epsilon: #any action
            return np.random.choice(list(self.actions.keys()))

        #Exploitation:
        else:
            #Geting the utilities of all possible actions from current state
            return np.argmax(self.q_table[state[0], state[1]])

    def update_q_table(self, action, current_state, reward, new_state):
        current_q = self.q_table[current_state[0],current_state[1], action]
        max_future_q = max(self.q_table[new_state[0], new_state[1]])
```

```
312         new_q = current_q + self.alpha * (reward + self.gamma * (max_future_q)
    - current_q)
313         self.q_table[current_state[0], current_state[1], action] = new_q
314 
315     def run_episodes(self):
316 
317         selected_episodes = [0, 49, 99, 999, 4999, 9999]
318         utility_filenames = []
319         policy_filenames = []
320         #mask = (maze != 2) & (maze != 3)
321 
322         for episode in range(self.episodes):
323 
324             print("Episode: ", episode)
325             # starting_episode = 9998
326             # ending_episode = 9999
327             # i = 0
328             prev_utility = np.copy(np.max(self.q_table, axis=2)) #before the
    update
329 
330             current_state = self.reset()
331             done = False
332             while not done:
333 
334                 #Animation purposes
335                 # if (starting_episode <= episode <= ending_episode):
336                 #     i += 1
337                 #     plt.clf()
338                 #     plt.imshow(self.maze, cmap='hot', interpolation='nearest
    ')
339                 #     plt.scatter(self.current_pos[1], self.current_pos[0], c
    ='blue', s=100)  # Agent's position
340                 #     plt.title(f'QL_Maze_{episode}')
341                 #     if i<=9:
342                 #         plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024
    SpringSemester\\EE449\\HW3\\animation_plots\\figure_000{i}")   # save the
    figure to file
343                 #     elif 10 <= i <= 99:
344                 #         plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024
    SpringSemester\\EE449\\HW3\\animation_plots\\figure_00{i}")   # save the
    figure to file
345                 #     elif 100 <= i <= 999:
346                 #         plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024
    SpringSemester\\EE449\\HW3\\animation_plots\\figure_0{i}")   # save the
    figure to file
347                 #     else:
348                 #         plt.savefig(f"C:\\Users\\defne\\Desktop\\2023-2024
    SpringSemester\\EE449\\HW3\\animation_plots\\figure_{i}")   # save the
    figure to file
349 
350 
351                 action = self.choose_action(current_state)
352                 new_state, reward, done = self.step(action)
353                 self.update_q_table(action, current_state, reward, new_state)
```

```
                current_state = new_state

                # #This part is for initialization of the goal and trap utility
                # if (self.maze[new_state] == 3) and (max(self.q_table[new_state[0], new_state[1]]) == 0):
                #     self.q_table[new_state[0], new_state[1]] = 1000 #goal
                # if (self.maze[new_state] == 2) and (max(self.q_table[new_state[0], new_state[1]]) == 0):
                #     self.q_table[new_state[0], new_state[1]] = -1000 #trap

            diff = np.sum(np.abs(np.copy(np.max(self.q_table, axis=2)) - prev_utility))
            self.convergence_history.append(diff)

            # if episode in selected_episodes:

            #     ax = utils_update.plot_value_function(copy.deepcopy(np.max(self.q_table, axis=2)), np.copy(self.maze), episode)
            #     filename_1 = f'Utility_Episode_{episode + 1}_alpha_0_001.png'
            #     ax.figure.savefig(filename_1)
            #     utility_filenames.append(filename_1)
            #     plt.close()

            #     ax = utils_update.plot_policy(copy.deepcopy(np.max(self.q_table, axis=2)), np.copy(self.maze), episode)
            #     filename_2 = f'Policy_Episode_{episode + 1}_alpha_0_001.png'
            #     ax.figure.savefig(filename_2)
            #     policy_filenames.append(filename_2)
            #     plt.close()

        return self.q_table, utility_filenames, policy_filenames

alpha = 0.1
gamma = 0.95
epsilon = 1

maze_q_learning = MazeQLearning(maze, alpha, gamma, epsilon, episodes=10000)
final_q_table, utility_image, policy_image = maze_q_learning.run_episodes()

final_utilities = np.max(final_q_table, axis=2) # Take max across the action dimension

history = maze_q_learning.convergence_history
#history_smoothed = moving_average(history, 50)
history_smoothed = exponential_moving_average(history, 0.02)

parameter = 'epsilon = 1'
plt.figure(figsize=(20, 10))
episodes = np.arange(1,10001) # Episodes from 1 to 10000
plt.plot(episodes,history_smoothed)
plt.xlabel('Episode')
plt.ylabel('Sum of Absolute Differences')
```

```
plt.title(f'Convergence of Value Function for {parameter}')
plt.grid(True)
plt.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3\\Q_learning\\epsilon\\epsilon_1\\Convergence_for_parameter_{parameter}.png')
plt.show()

fig1 = plot_episodes(utility_image, parameter, 'Utility')
fig1.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3\\Q_learning\\epsilon\\epsilon_1\\Episodes_utility_{parameter}.png')

fig2 = plot_episodes(policy_image, parameter, 'Policy')
fig2.savefig(f'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3\\Q_learning\\epsilon\\epsilon_1\\Episodes_policy_{parameter}.png')

final_utilities[maze == 1] = -1000 #boundaries
# animation_gif('C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW3\\animation_plots', 9998, 9999)
utils.plot_value_function(final_utilities, maze)
utils.plot_policy(final_utilities, maze)
```