Ahmet Akman 2442366
Middle East Technical University
Electrical and Electronics Engineering

May 26, 2024

HOMEWORK 3 — Report

1 Questions

1.1 Agent:

Agent corresponds to the entity that interacts with the environment. It is responsible for taking action, observing the environment, and receiving rewards. The agent is the entity that learns the optimal policy to maximize the cumulative reward. When compared to supervised learning, the agent is not provided with the correct output, but it learns the optimal policy through trial and error. That is, the concept of error is not exactly available in a reinforcement setting.

1.2 Environment:

Environment means the world that the agent interacts with. It is the entity that the agent observes and takes action. The environment is responsible for providing the agent with the current state, reward, and possible actions. The environment is also responsible for updating the state of the agent based on the action taken by the agent. When compared to supervised learning, the environment corresponds to both the dataset and loss function that the system trained and runs on.

1.3 Reward:

The reward is the feedback that the agent receives from the environment. It is the scalar value that the agent receives after taking an action. The reward is used to evaluate the action taken by the agent. The agent aims to maximize the cumulative reward by learning the optimal policy. When compared to supervised learning, the reward corresponds to the loss function that the system tries to minimize.

1.4 Policy:

Policy is the strategy that the agent uses to take action. It is the mapping from the state to the action. If we compare it with supervised learning, the policy corresponds to the model that the system uses to predict the output.

1.5 Exploration:

The exploration corresponds to the process of trying different actions to learn the optimal policy. The agent explores the environment by taking different actions and observing the reward. The exploration is necessary to learn the optimal policy. When we try to map exploration to the supervised learning step, we may encounter more than one phenomenon. For example, we can say that exploration corresponds to the training process of the model. On the other hand, we can say the exploration corresponds to the data augmentation process or other randomization steps taken during training.

1.6 Exploitation:

Exploitation corresponds to the process of taking the best action based on the learned policy. The agent exploits the environment by taking the action that maximizes the reward. Exploitation is necessary to maximize the cumulative reward. When we try to map exploitation to the supervised learning step, we can say that exploitation corresponds to the inference process of the model.

2 Experimental Work

The proper implementation for maze, temporal difference learning and Q-learning are implemented and provided in appendix.
The experimental work section is divided into four main part where each part has TD learning and Q learning related experiments seperately. First the default parameter outputs are presented. Then the effect of alpha parameter is investigated. After that the effect of gamma parameter is investigated. Lastly the effect of epsilon parameter is investigated.

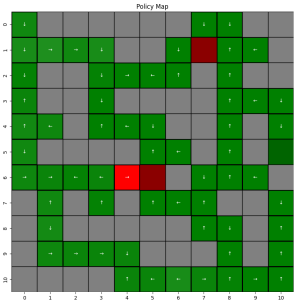
2.1 Temporal Difference Learning Default Parameters

The default parameters for the temporal difference learning are set as follows:

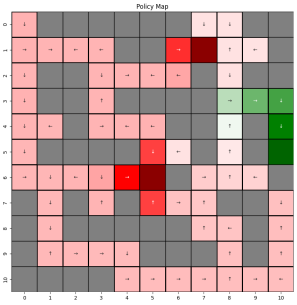
- Alpha: 0.1
- Gamma: 0.95
- Epsilon: 0.2
- Episodes: 10000

According to these settings the training is done. The policy maps are provided in Figure 1. The value function plots are provided in Figure 2. The convergence plots are provided in Figure 3.

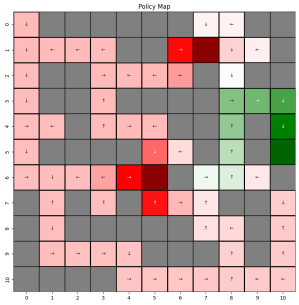
Basically we can say that the agent learns the optimal policy and value function.



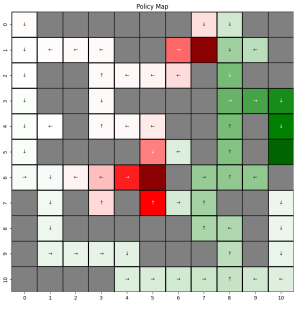
(a) Episode 1.



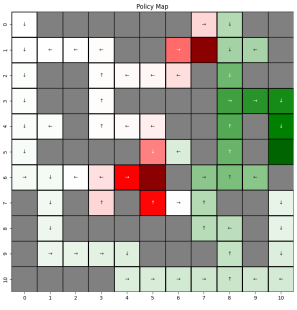
(b) Episode 50



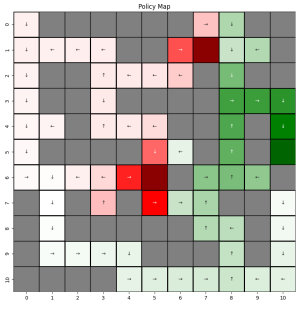
(c) Episode 100



(d) Episode 1000.

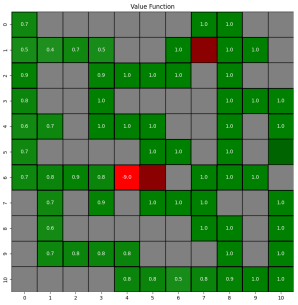


(e) Episode 5000

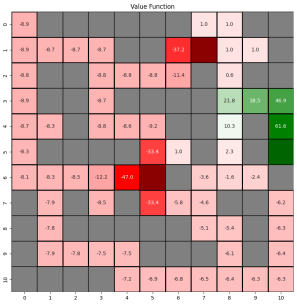


(f) Episode 10000

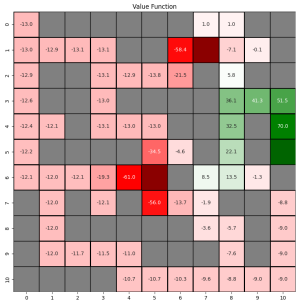
Figure 1: Evolution of policy maps throughout episodes.



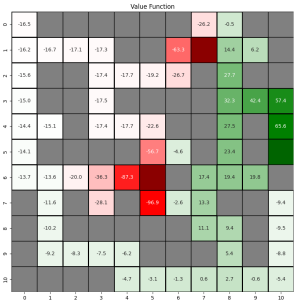
(a) Episode 1.



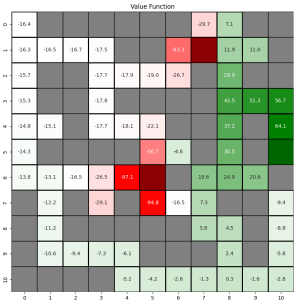
(b) Episode 50



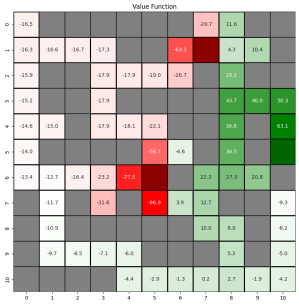
(c) Episode 100



(d) Episode 1000.

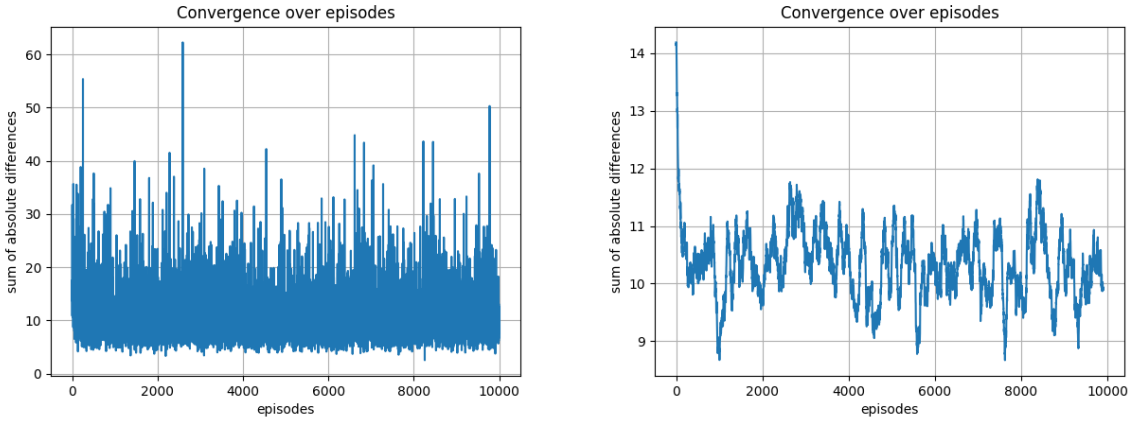


(e) Episode 5000



(f) Episode 10000

Figure 2: Evolution of value function throughout episodes.



(a) Converge raw.

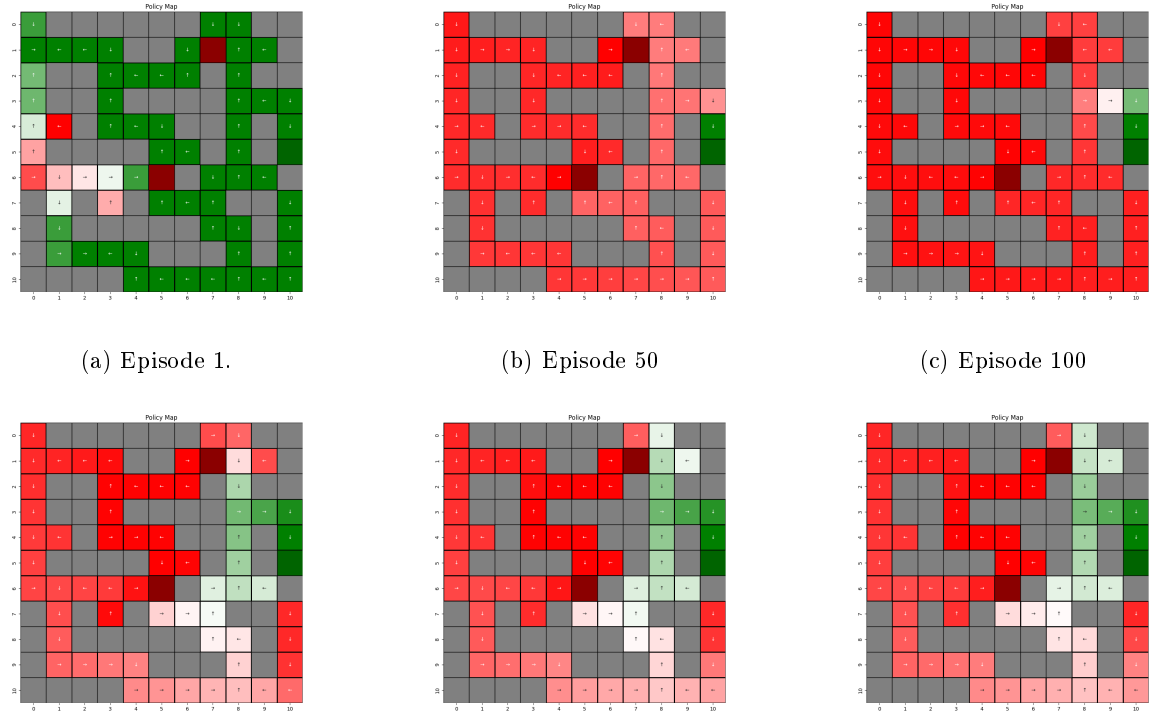
(b) Smoothed convergence.

Figure 3: Converge of value function.

2.2 Q-Learning Default Parameters

As same as the temporal difference learning, the default parameters for the Q-learning are set. Then the training is done. The policy maps are provided in Figure 4. The value function plots are provided in Figure 5. The convergence plots are provided in Figure 6.

So, again we can say that the agent learns the optimal policy and value function at the end.



(a) Episode 1.

(b) Episode 50

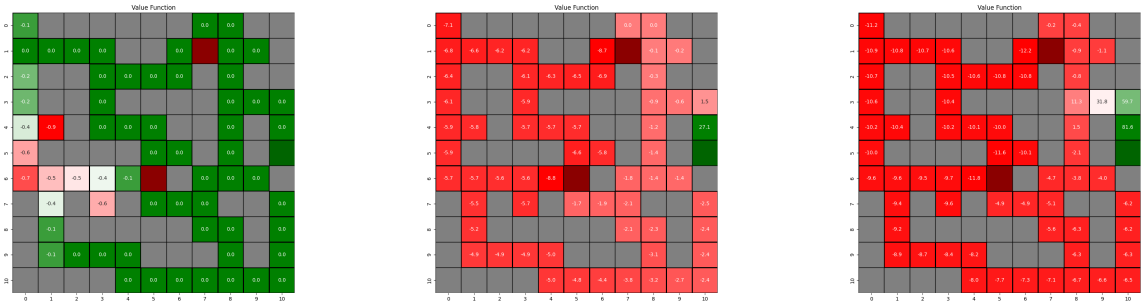
(c) Episode 100

(d) Episode 1000.

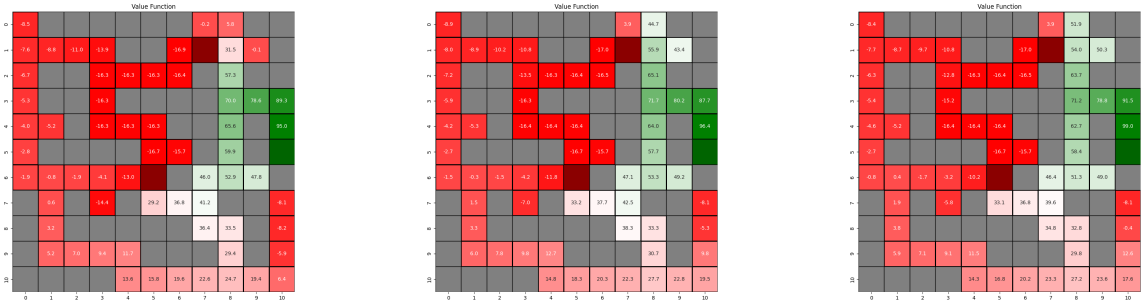
(e) Episode 5000

(f) Episode 10000

Figure 4: Evolution of policy maps throughout episodes.

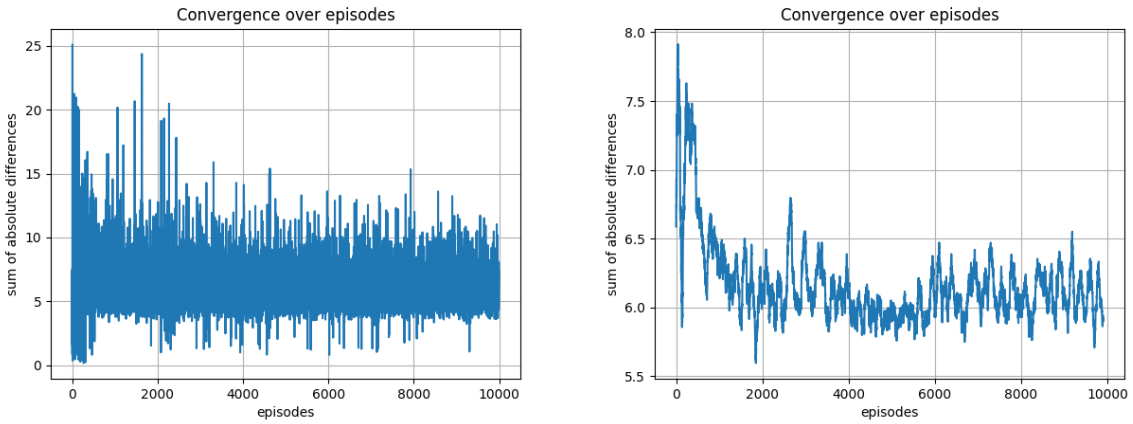


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 5: Evolution of value function throughout episodes.



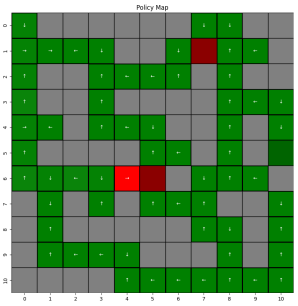
(a) Converge raw. (b) Smoothed convergence.

Figure 6: Converge of value function.

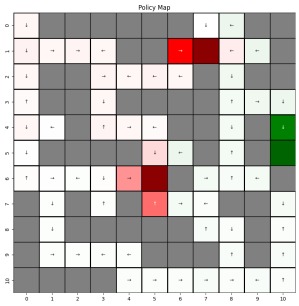
2.3 Effect of Alpha in Temporal Difference Learning

Let us first provide the necessary output for each alpha parameter and then discuss the results at the end of this section.

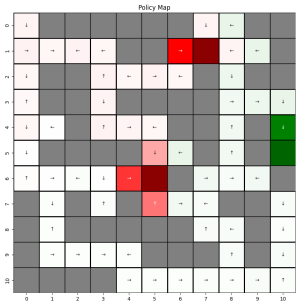
Figure 7 provides the policy maps for the alpha parameter set to 0.001. Figure 8 provides the value function plots for the alpha parameter set to 0.001. Figure 9 provides the convergence plots for the alpha parameter set to 0.001.



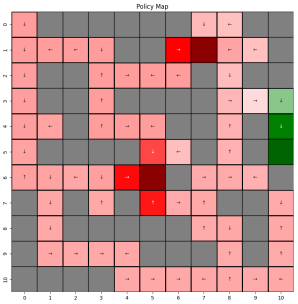
(a) Episode 1.



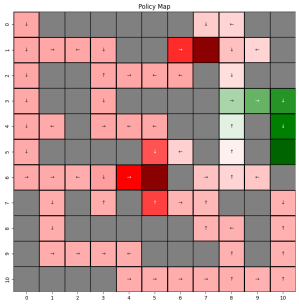
(b) Episode 50



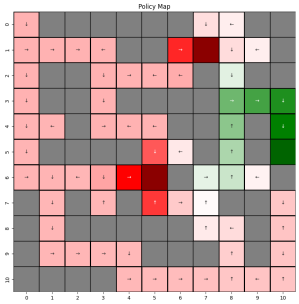
(c) Episode 100



(d) Episode 1000.

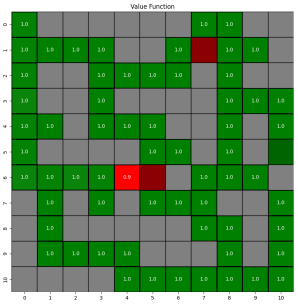


(e) Episode 5000

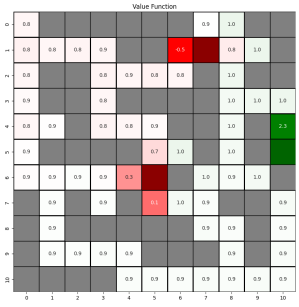


(f) Episode 10000

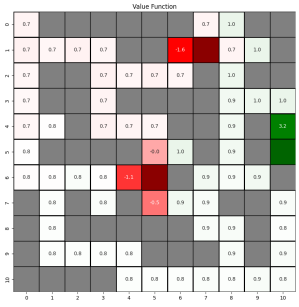
Figure 7: Evolution of policy maps throughout episodes.



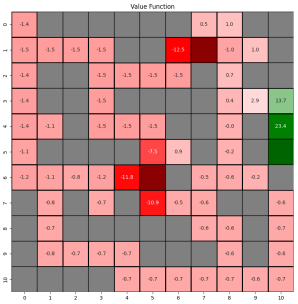
(a) Episode 1.



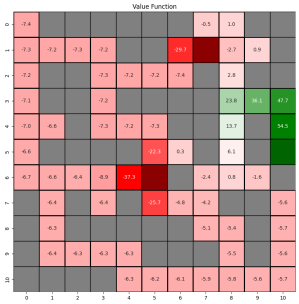
(b) Episode 50



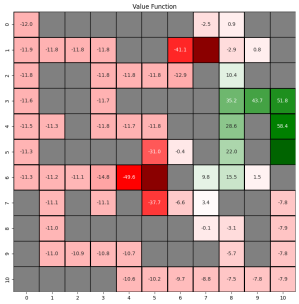
(c) Episode 100



(d) Episode 1000.

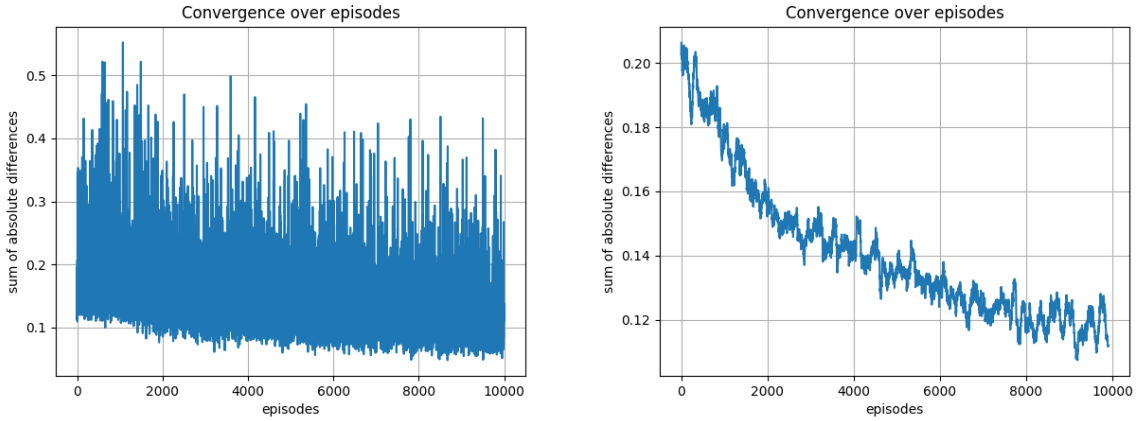


(e) Episode 5000



(f) Episode 10000

Figure 8: Evolution of value function throughout episodes.

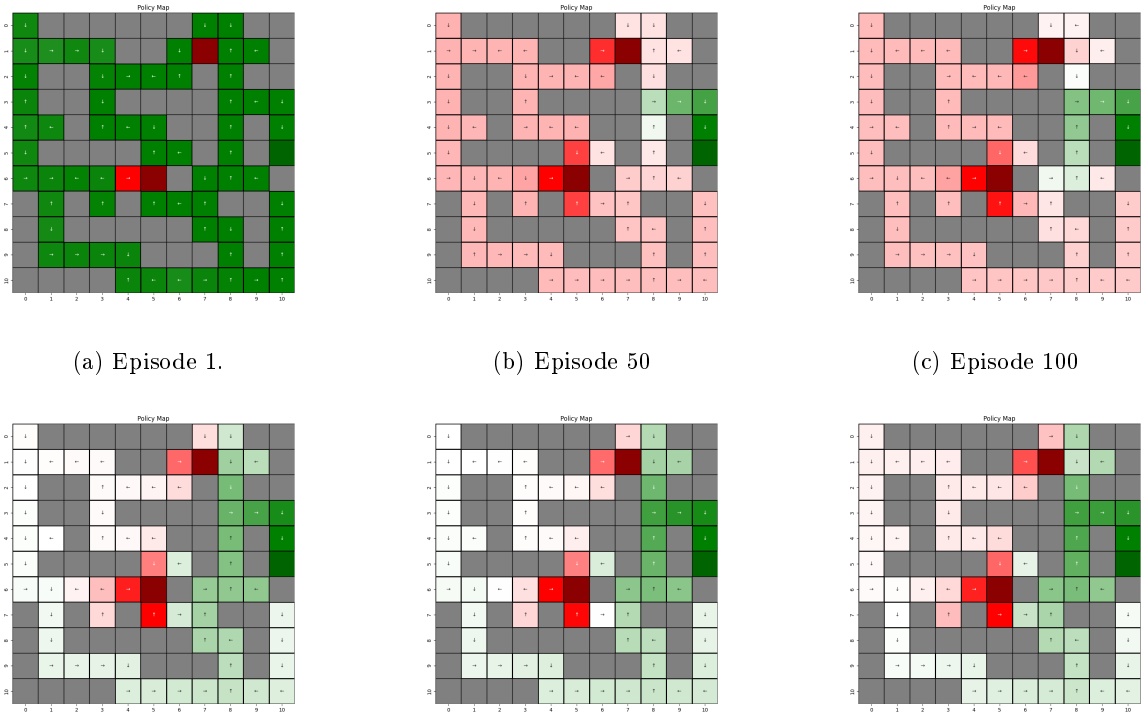


(a) Converge raw.

(b) Smoothed convergence.

Figure 9: Converge of value function.

Figure 10 provides the policy maps for the alpha parameter set to 0.1. Figure 11 provides the value function plots for the alpha parameter set to 0.1. Figure 12 provides the convergence plots for the alpha parameter set to 0.1.



(a) Episode 1.

(b) Episode 50

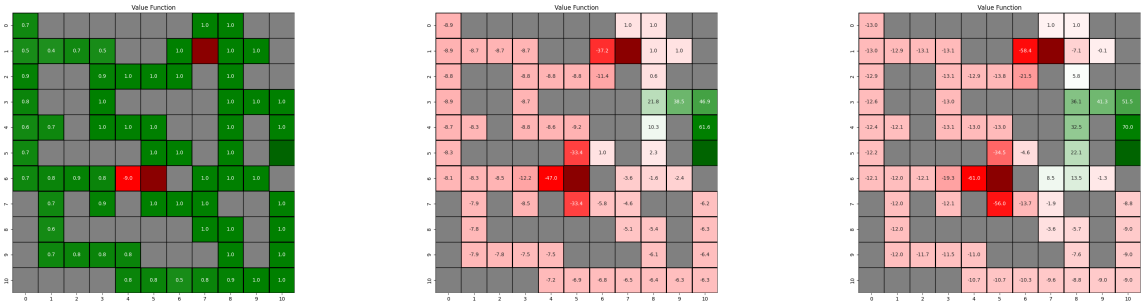
(c) Episode 100

(d) Episode 1000.

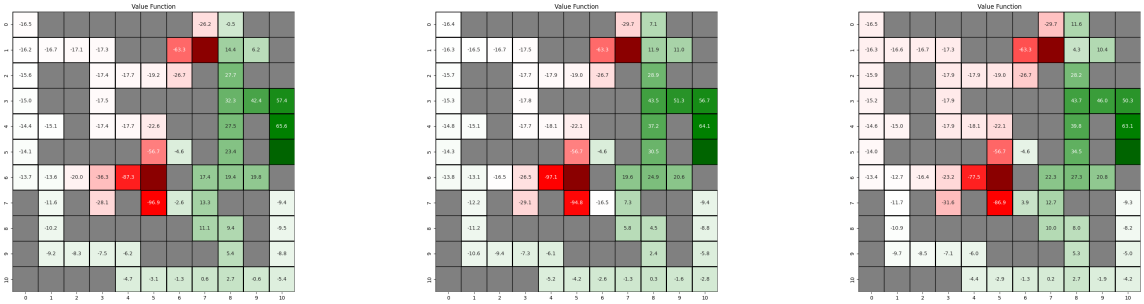
(e) Episode 5000

(f) Episode 10000

Figure 10: Evolution of policy maps throughout episodes.

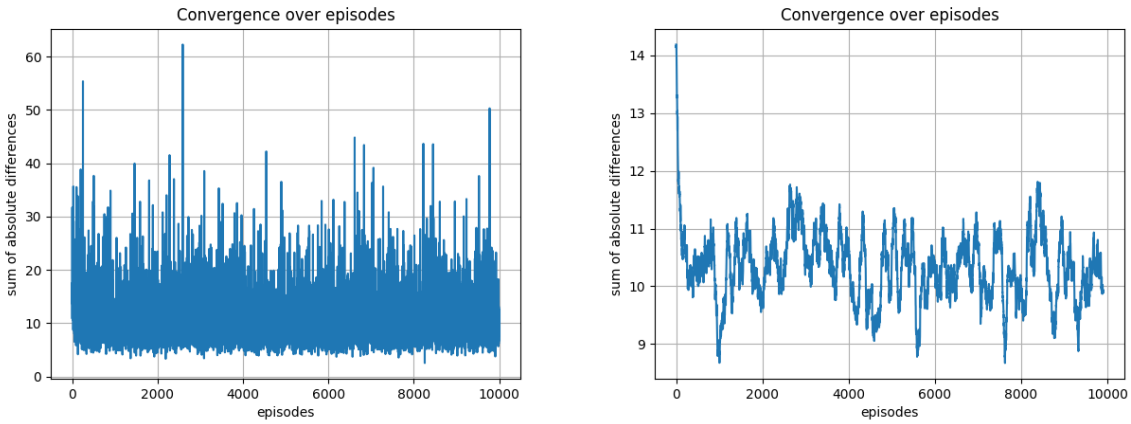


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

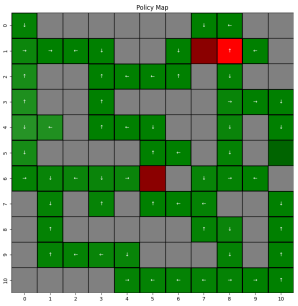
Figure 11: Evolution of value function throughout episodes.



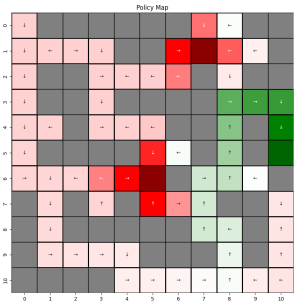
(a) Converge raw. (b) Smoothed convergence.

Figure 12: Converge of value function.

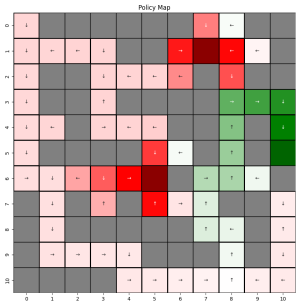
Figure 13 shows the policy maps for the alpha parameter set to 0.5. Figure 14 illustrates the value function plots for the alpha parameter set to 0.5. Figure 15 provides the convergence plots for the alpha parameter set to 0.5.



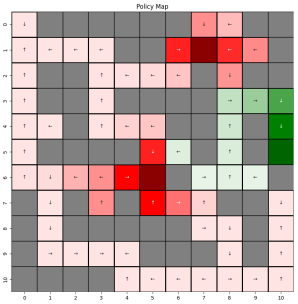
(a) Episode 1.



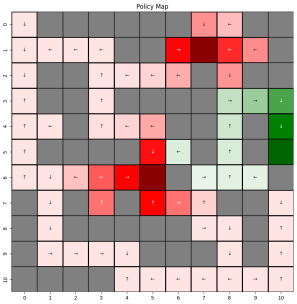
(b) Episode 50



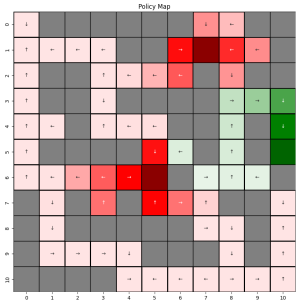
(c) Episode 100



(d) Episode 1000.

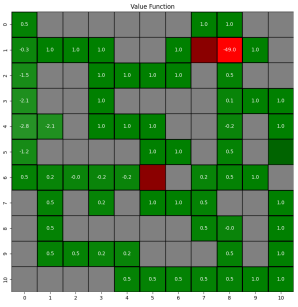


(e) Episode 5000

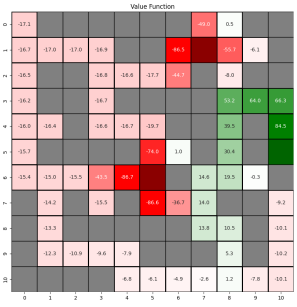


(f) Episode 10000

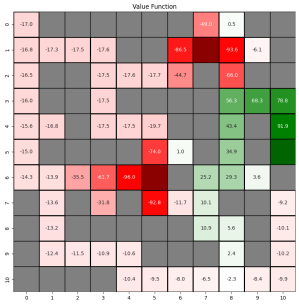
Figure 13: Evolution of policy maps throughout episodes.



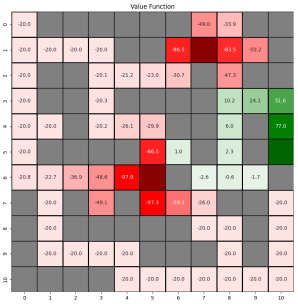
(a) Episode 1.



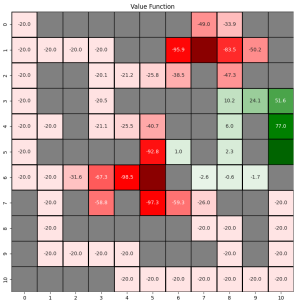
(b) Episode 50



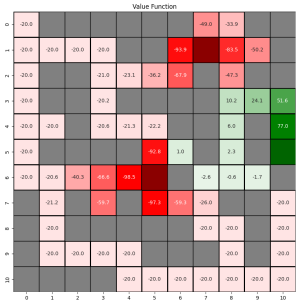
(c) Episode 100



(d) Episode 1000.

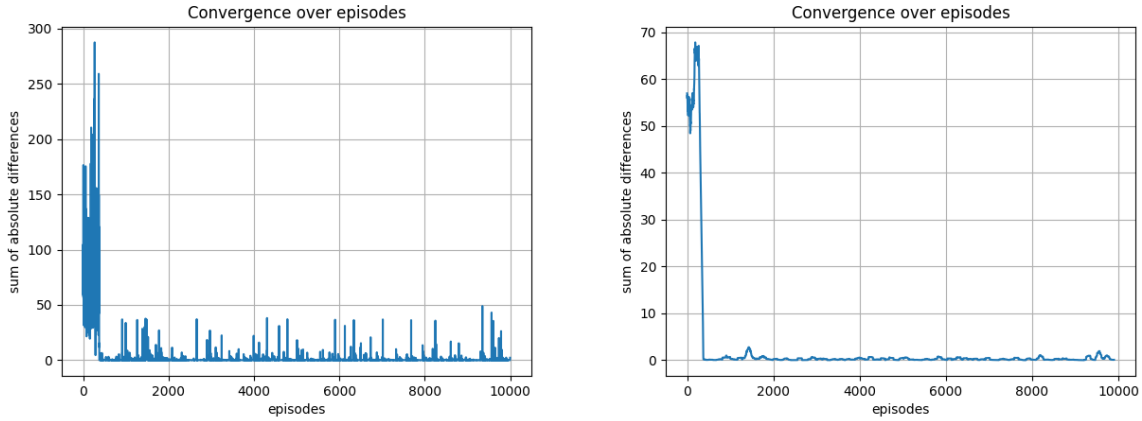


(e) Episode 5000



(f) Episode 10000

Figure 14: Evolution of value function throughout episodes.

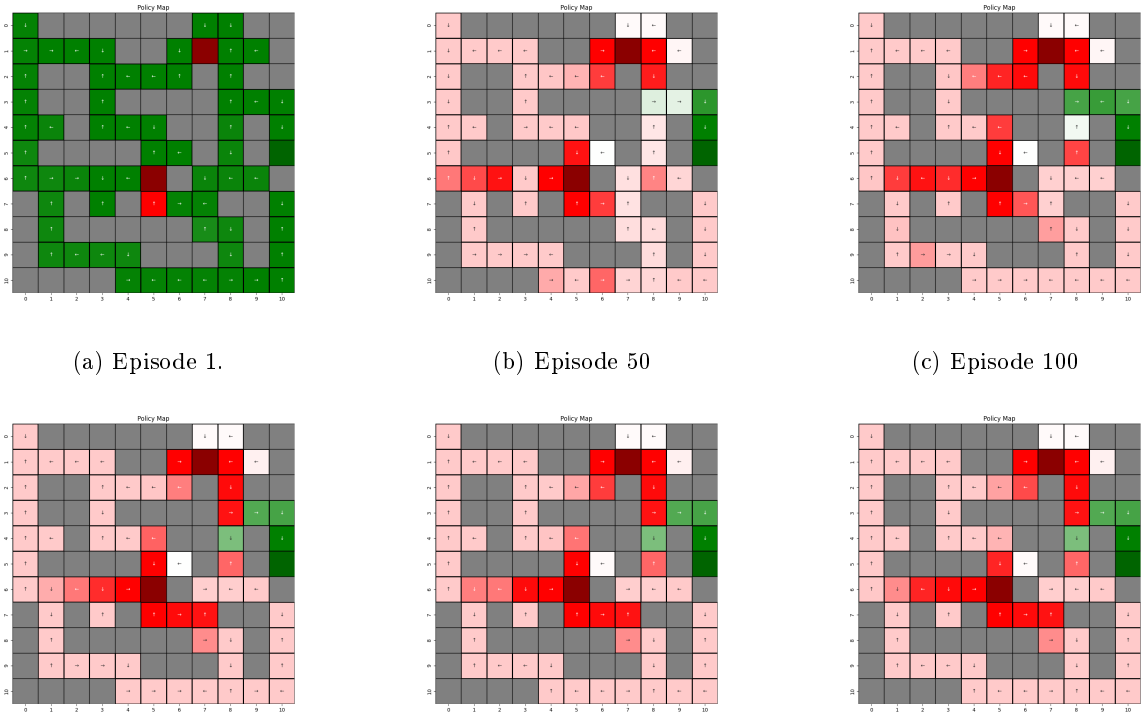


(a) Converge raw.

(b) Smoothed convergence.

Figure 15: Converge of value function.

Lastly, Figure 16 shows the policy maps for the alpha parameter set to 1. Figure 17 illustrates the value function plots for the alpha parameter set to 1. Figure 18 provides the convergence plots for the alpha parameter set to 1.



(a) Episode 1.

(b) Episode 50

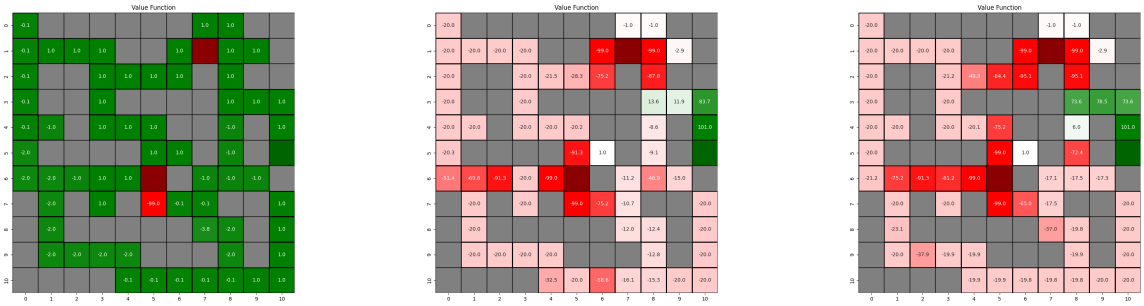
(c) Episode 100

(d) Episode 1000.

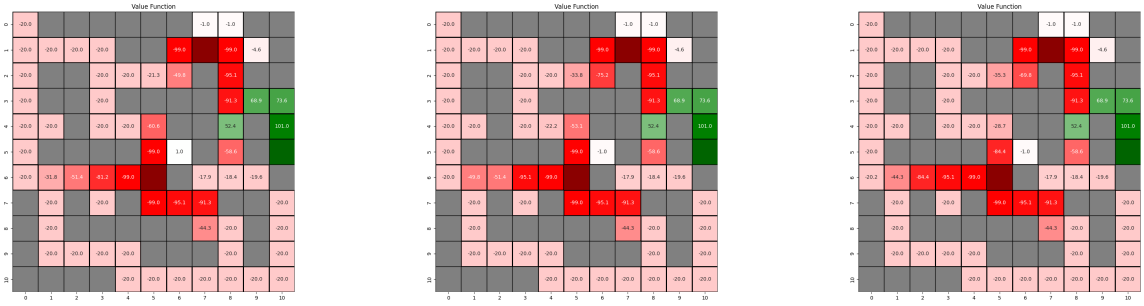
(e) Episode 5000

(f) Episode 10000

Figure 16: Evolution of policy maps throughout episodes.

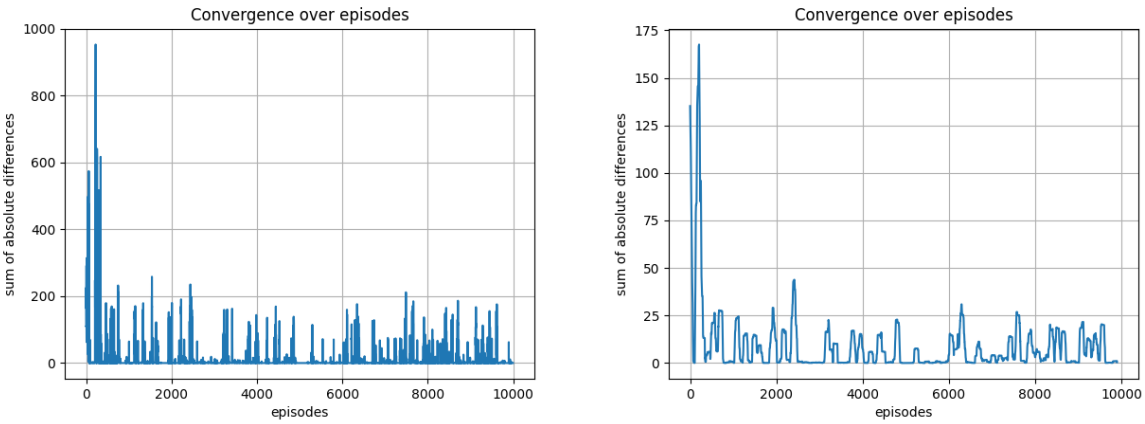


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 17: Evolution of value function throughout episodes.



(a) Converge raw. (b) Smoothed convergence.

Figure 18: Converge of value function.

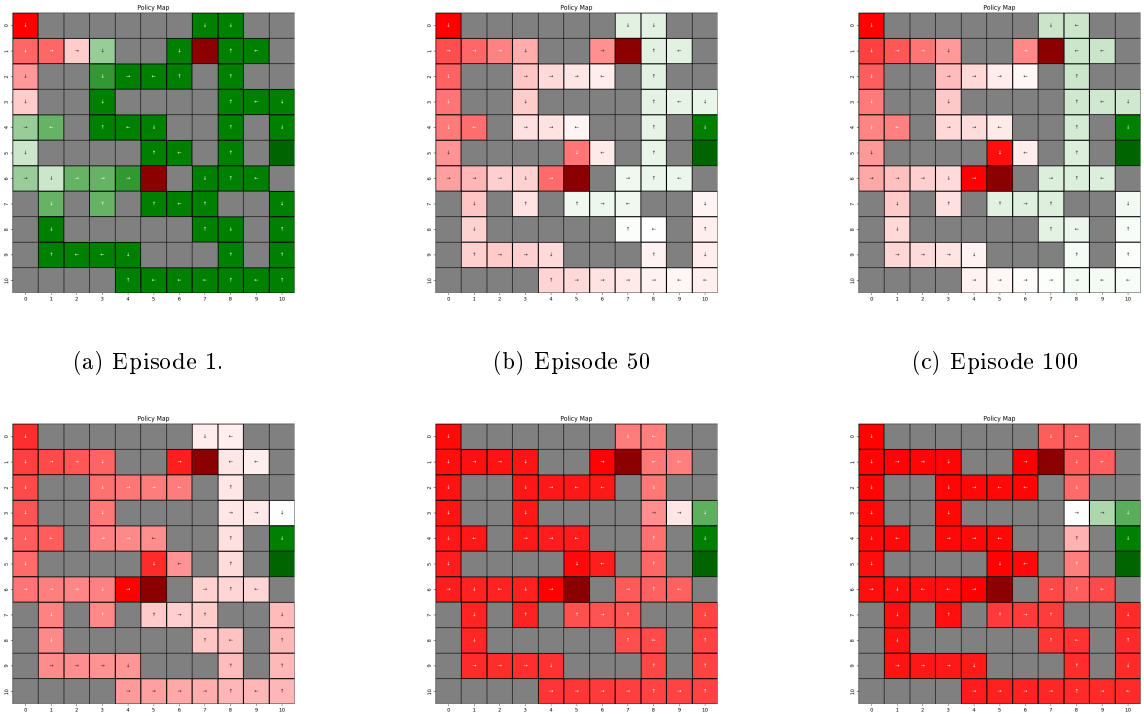
What we can interpret from the sweep of alpha value in temporal difference learning as follows. When the alpha value is set to 0.001, the agent try learns the optimal policy and value function as expected. However, the convergence is slower compared to the other alpha values and proper convergence is not observed. When the alpha value set to 0.01 convergence observed to optimal policy. When the alpha value is set to 0.5, the agent learns the optimal policy and value function as expected. The convergence is faster compared to the alpha value set to 0.01. Lastly, when the alpha value is set to 1, the agent can not learn the optimal policy and value function since the learning rate overshoots. The convergence is faster compared to the alpha value set to 0.5 but it is hard to say this is a stable convergence. The reason

is, the convergence is not as smooth as the alpha value set to 0.5. So the defaul value and 0.5 are better choices for the alpha parameter in temporal difference learning.

2.4 Effect of Alpha in Q-Learning

Similarly, we can analyze the effect of alpha parameter in Q-learning. The results are provided below.

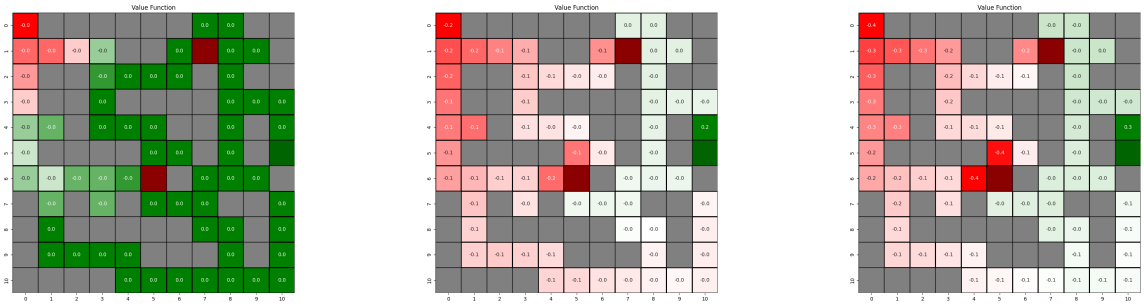
Figure 19 shows the policy maps for the alpha parameter set to 0.001. Figure 20 illustrates the value function plots for the alpha parameter set to 0.001. Figure 21 presents the convergence plots for the alpha parameter set to 0.001.



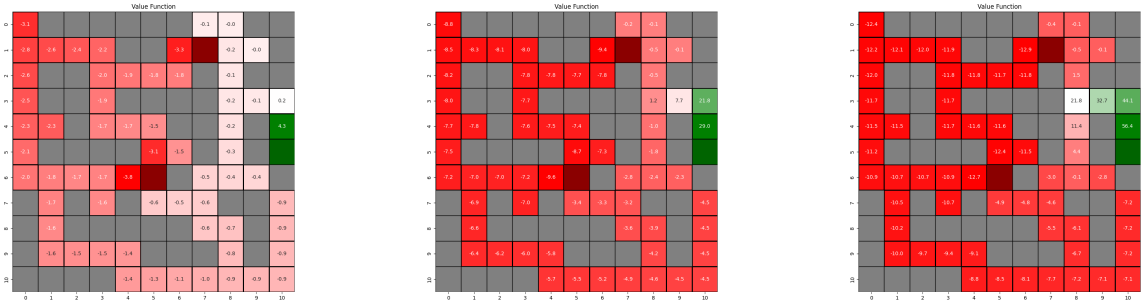
(a) Episode 1. (b) Episode 50 (c) Episode 100

(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 19: Evolution of policy maps throughout episodes.

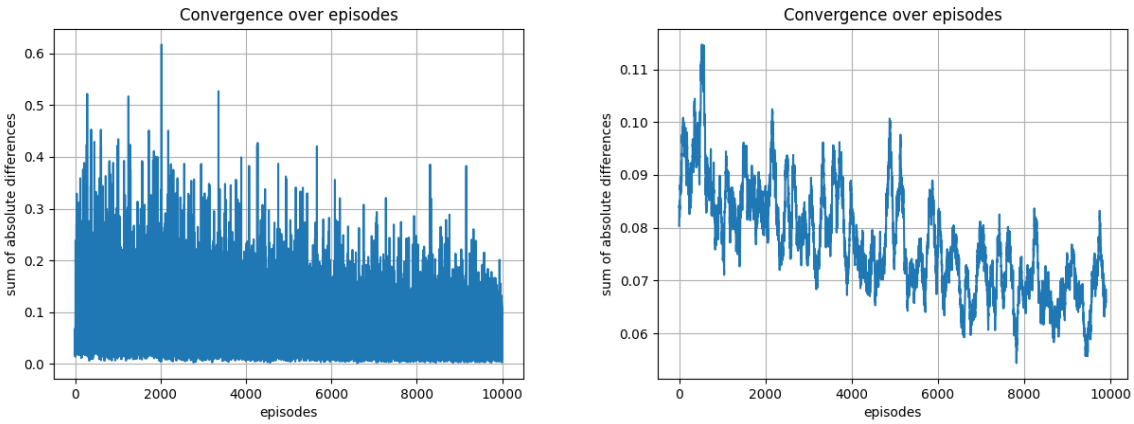


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

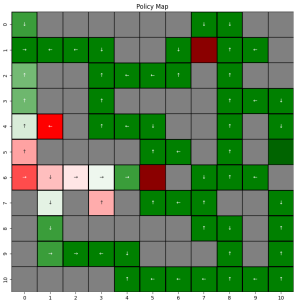
Figure 20: Evolution of value function throughout episodes.



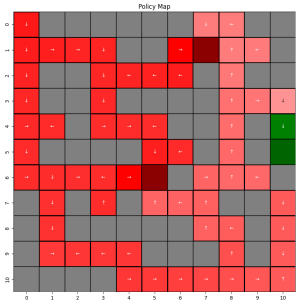
(a) Converge raw. (b) Smoothed convergence.

Figure 21: Converge of value function.

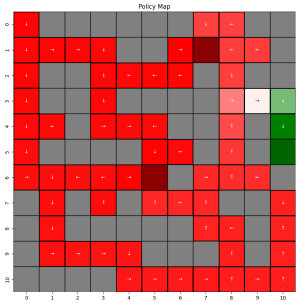
Figure 22 shows the policy maps for the alpha parameter set to 0.1. Figure 23 illustrates the value function plots for the alpha parameter set to 0.1. Figure 24 provides the convergence plots for the alpha parameter set to 0.1.



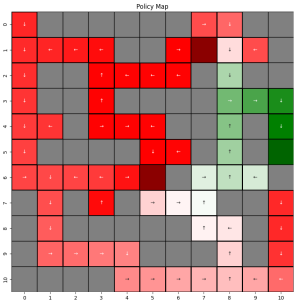
(a) Episode 1.



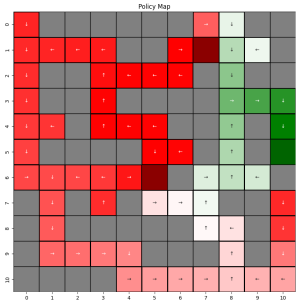
(b) Episode 50



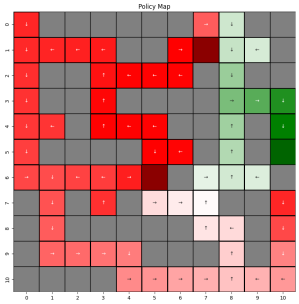
(c) Episode 100



(d) Episode 1000.

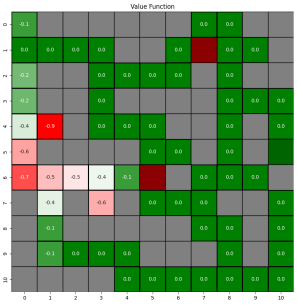


(e) Episode 5000

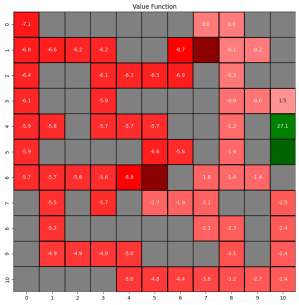


(f) Episode 10000

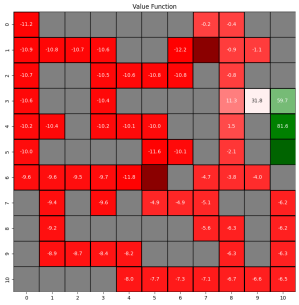
Figure 22: Evolution of policy maps throughout episodes.



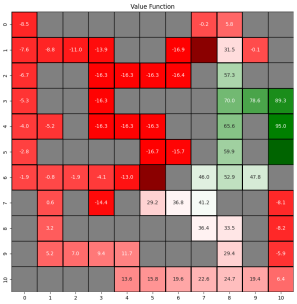
(a) Episode 1.



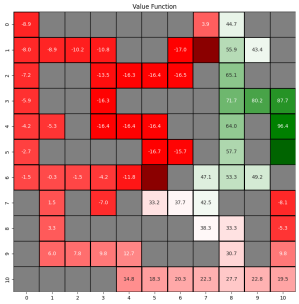
(b) Episode 50



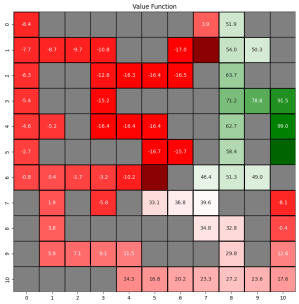
(c) Episode 100



(d) Episode 1000.

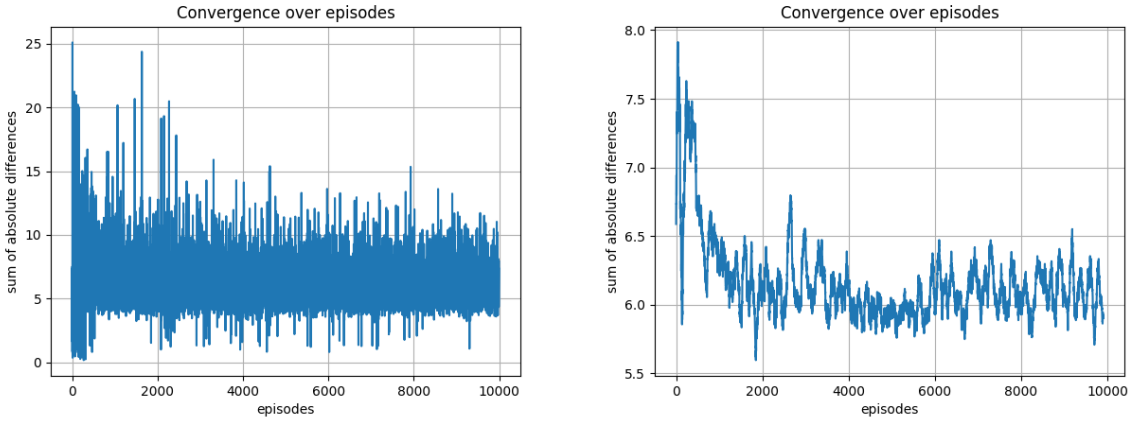


(e) Episode 5000



(f) Episode 10000

Figure 23: Evolution of value function throughout episodes.

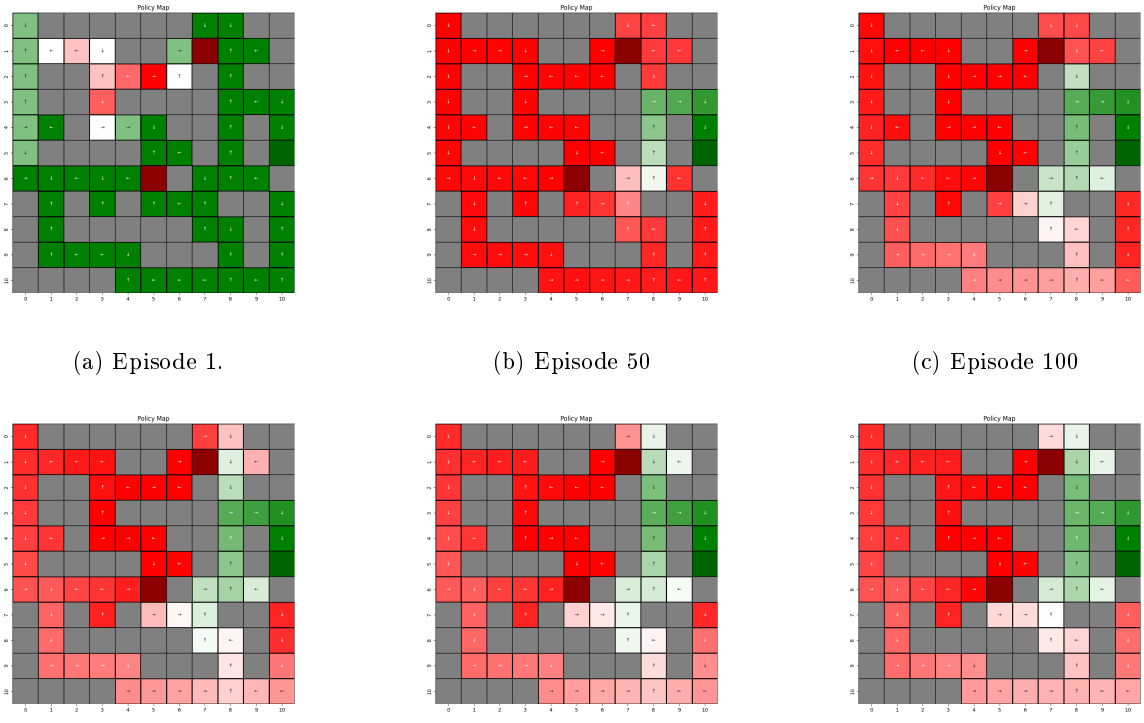


(a) Converge raw.

(b) Smoothed convergence.

Figure 24: Converge of value function.

Figure 25 shows the policy maps for the alpha parameter set to 0.5. Figure 26 illustrates the value function plots for the alpha parameter set to 0.5. Figure 27 provides the convergence plots for the alpha parameter set to 0.5.



(a) Episode 1.

(b) Episode 50

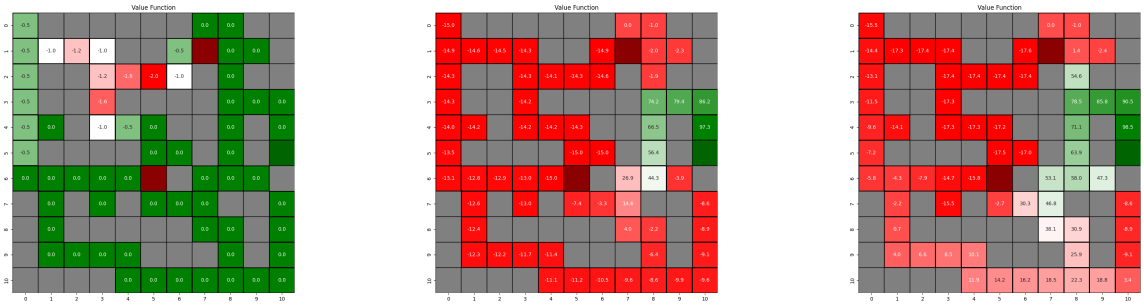
(c) Episode 100

(d) Episode 1000.

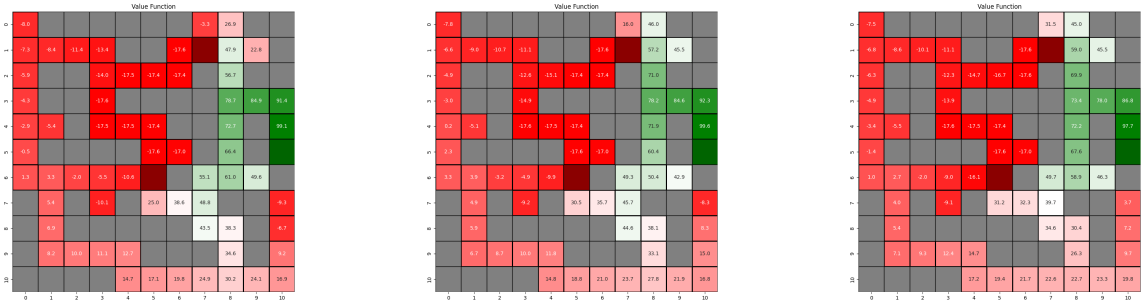
(e) Episode 5000

(f) Episode 10000

Figure 25: Evolution of policy maps throughout episodes.

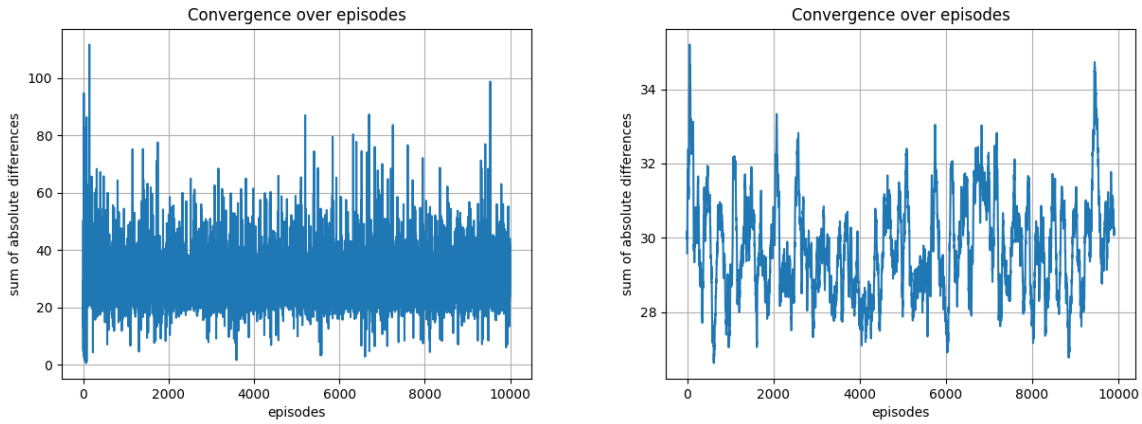


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

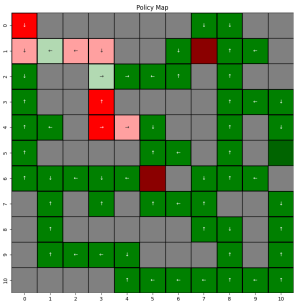
Figure 26: Evolution of value function throughout episodes.



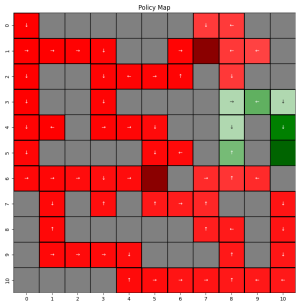
(a) Converge raw. (b) Smoothed convergence.

Figure 27: Converge of value function.

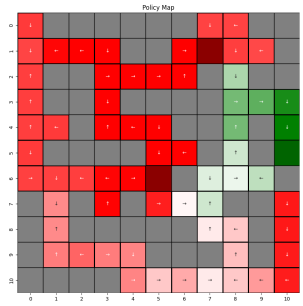
Figure 28 shows the policy maps for the alpha parameter set to 1. Figure 29 illustrates the value function plots for the alpha parameter set to 1. Figure 30 provides the convergence plots for the alpha parameter set to 1.



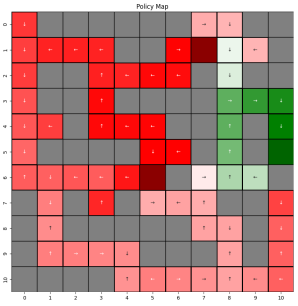
(a) Episode 1.



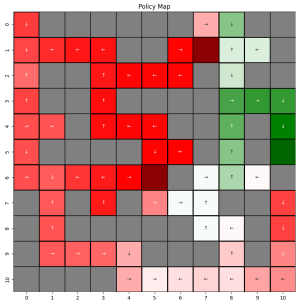
(b) Episode 50



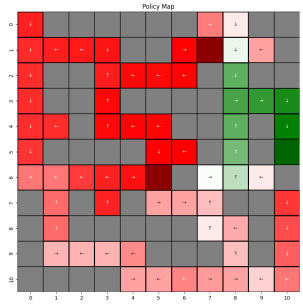
(c) Episode 100



(d) Episode 1000.

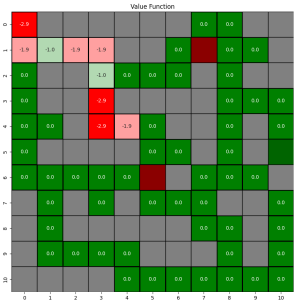


(e) Episode 5000

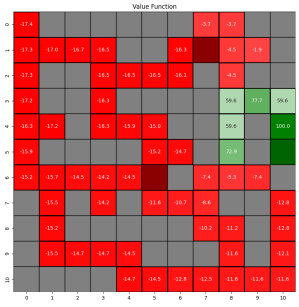


(f) Episode 10000

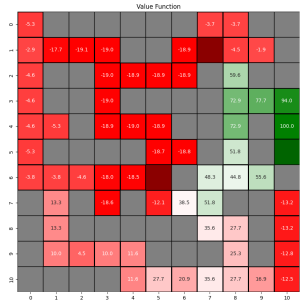
Figure 28: Evolution of policy maps throughout episodes.



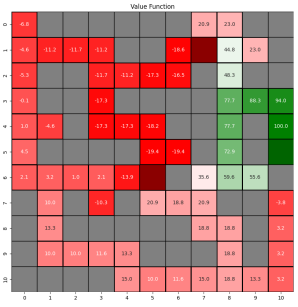
(a) Episode 1.



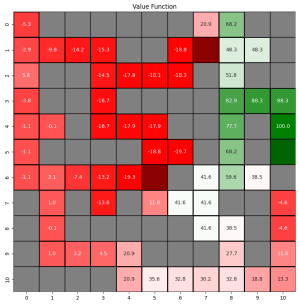
(b) Episode 50



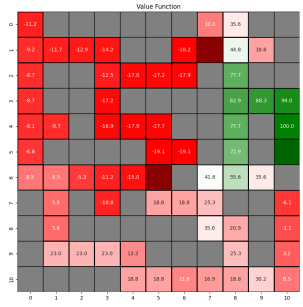
(c) Episode 100



(d) Episode 1000.

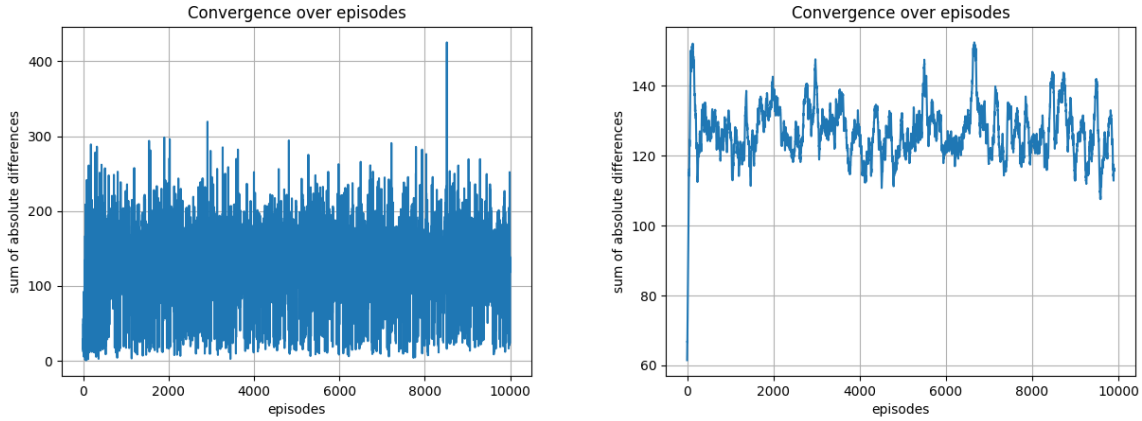


(e) Episode 5000



(f) Episode 10000

Figure 29: Evolution of value function throughout episodes.



(a) Converge raw.

(b) Smoothed convergence.

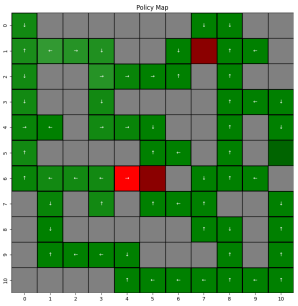
Figure 30: Converge of value function.

As a result we can say that the for small alpha values Q learning is an effective strategy as we see convergence is possible. However, for the alpha value set to 1, the agent can not learn the optimal policy and value function since the learning rate is too much again. For these values the trend of fast convergence is observed from 0.001 to 0.5. However, the convergence is not stable for the alpha value set to 1. So the values up to 0.5 are better choices for the alpha parameter in Q learning.

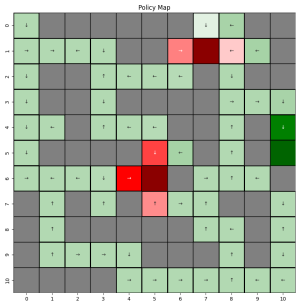
2.5 Effect of Gamma in Temporal Difference Learning

Here, we will have a look at the effect of gamma parameter in temporal difference learning. The results are provided below. The parameter set used for this experiment is as follows: $\alpha = 0.1$, $\epsilon = 0.2$, and the number of episodes is set to 10000. The gamma parameter is varied from 0.1 to 0.95.

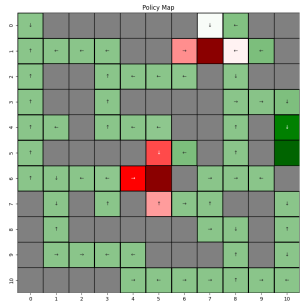
Figure 31 shows the policy maps for the gamma parameter set to 0.1. Figure 32 illustrates the value function plots for the gamma parameter set to 0.1. Figure 33 provides the convergence plots for the gamma parameter set to 0.1.



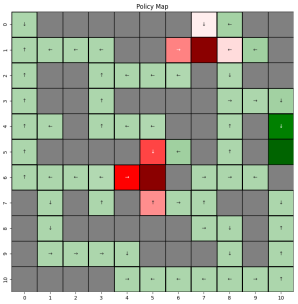
(a) Episode 1.



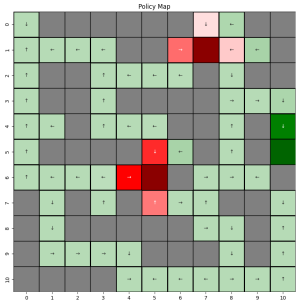
(b) Episode 50



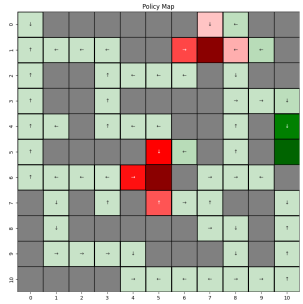
(c) Episode 100



(d) Episode 1000.

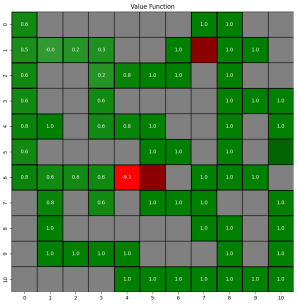


(e) Episode 5000

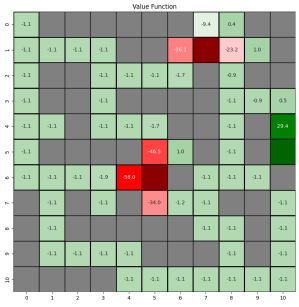


(f) Episode 10000

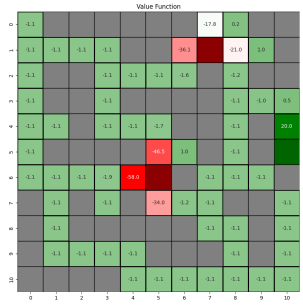
Figure 31: Evolution of policy maps throughout episodes.



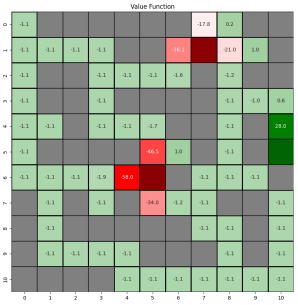
(a) Episode 1.



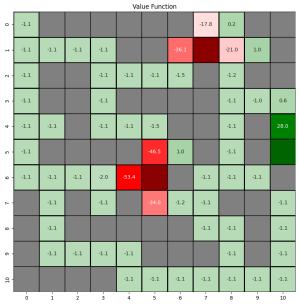
(b) Episode 50



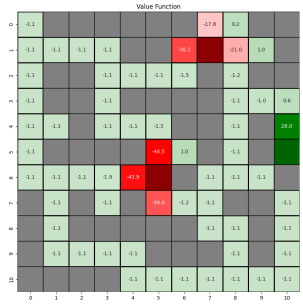
(c) Episode 100



(d) Episode 1000.

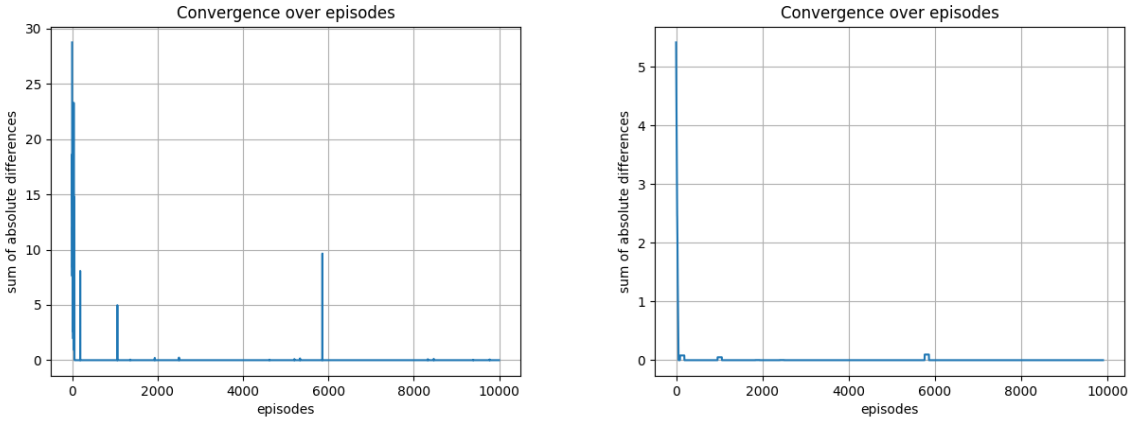


(e) Episode 5000



(f) Episode 10000

Figure 32: Evolution of value function throughout episodes.

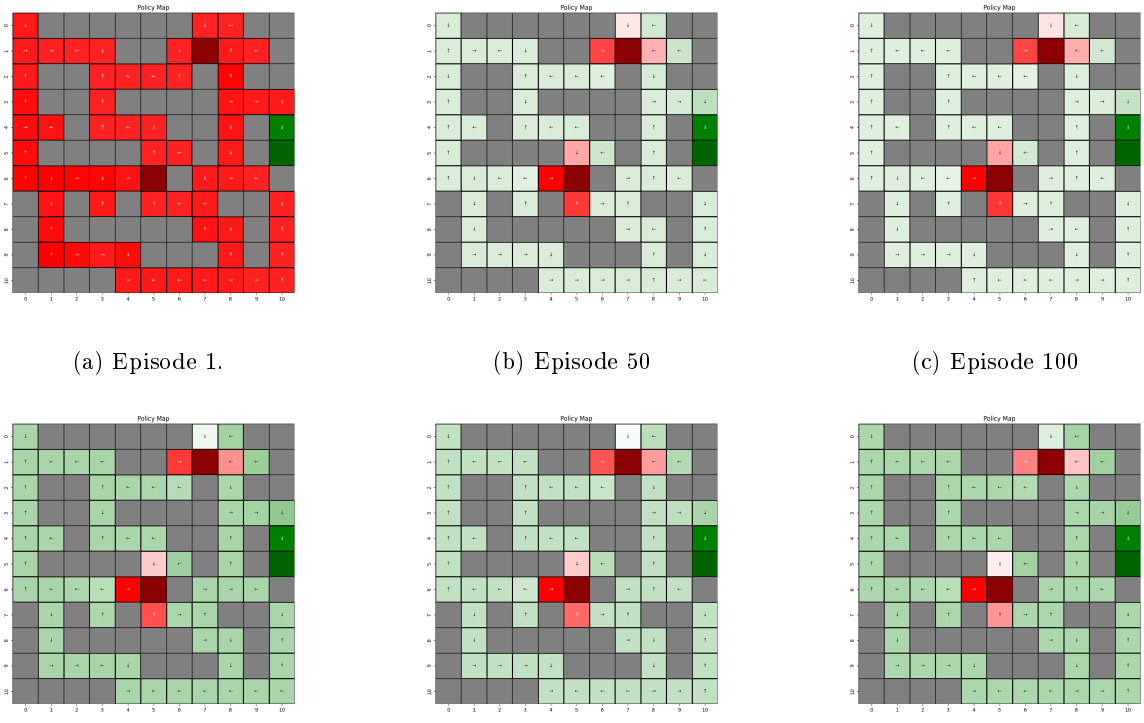


(a) Converge raw.

(b) Smoothed convergence.

Figure 33: Converge of value function.

Figure 34 is provided as the policy maps for the gamma parameter set to 0.25. Figure 35 shows the value function plots for the gamma parameter set to 0.25. Figure 36 presents the convergence plots for the gamma parameter set to 0.25.



(a) Episode 1.

(b) Episode 50

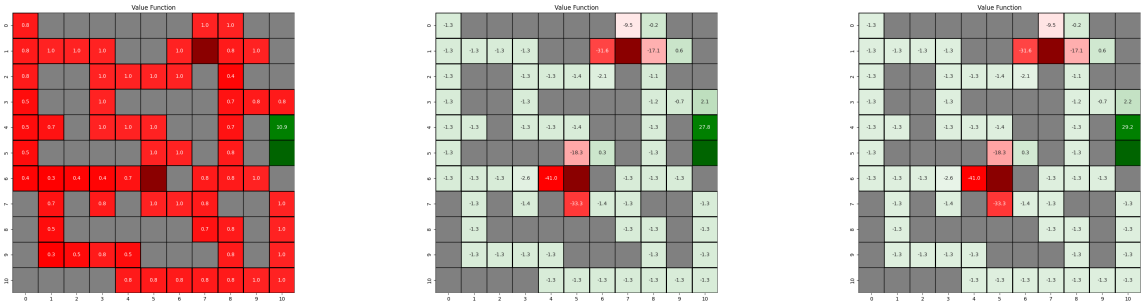
(c) Episode 100

(d) Episode 1000.

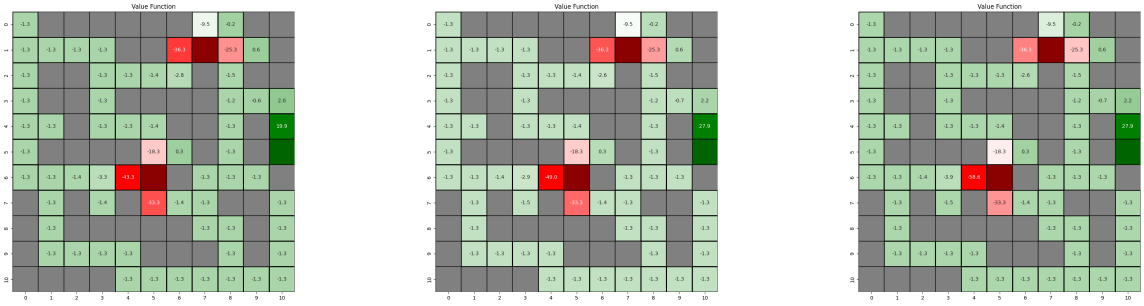
(e) Episode 5000

(f) Episode 10000

Figure 34: Evolution of policy maps throughout episodes.

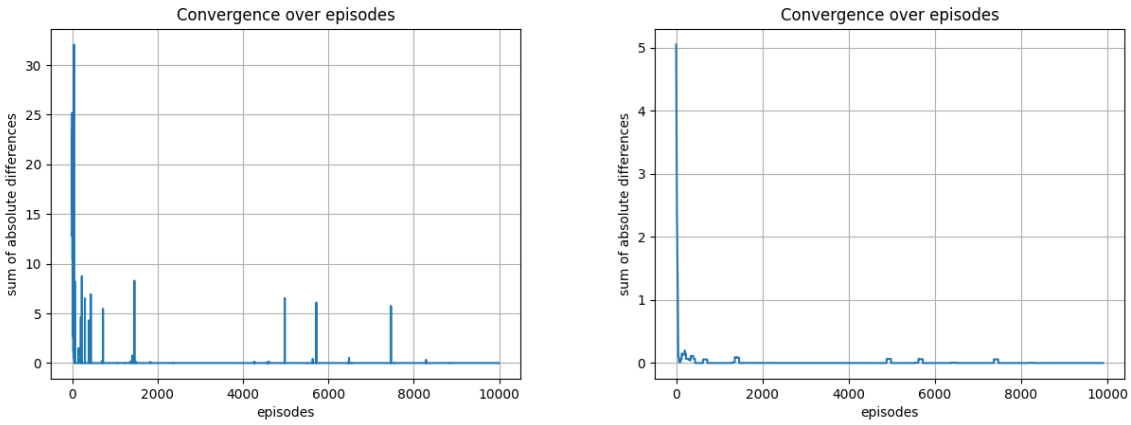


(a) Episode 1. (b) Episode 50 (c) Episode 100



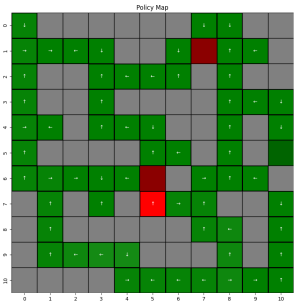
(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 35: Evolution of value function throughout episodes.

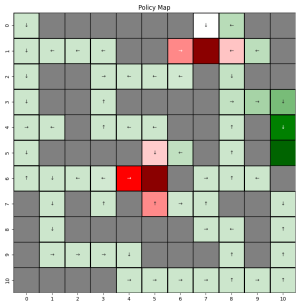


(a) Converge raw. (b) Smoothed convergence.

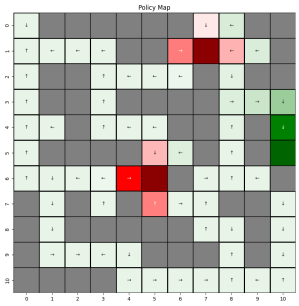
Figure 36: Converge of value function.



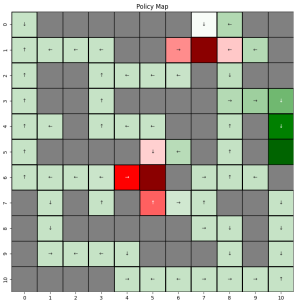
(a) Episode 1.



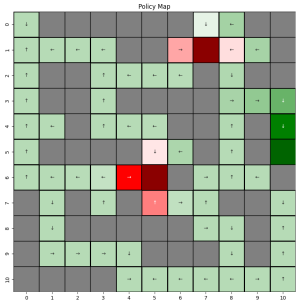
(b) Episode 50



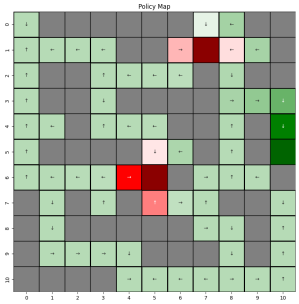
(c) Episode 100



(d) Episode 1000.

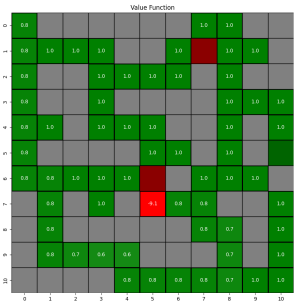


(e) Episode 5000

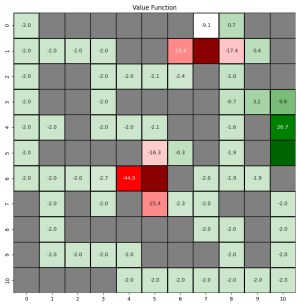


(f) Episode 10000

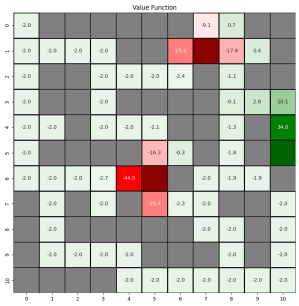
Figure 37: Evolution of policy maps throughout episodes.



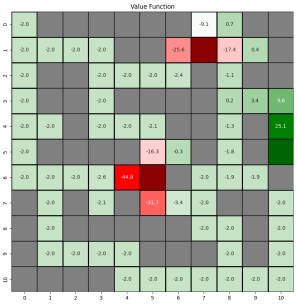
(a) Episode 1.



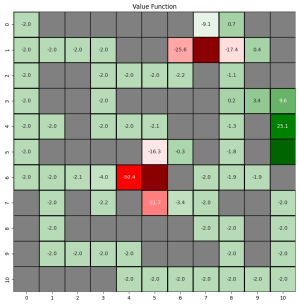
(b) Episode 50



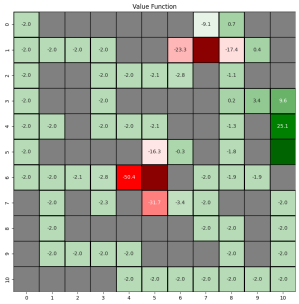
(c) Episode 100



(d) Episode 1000.

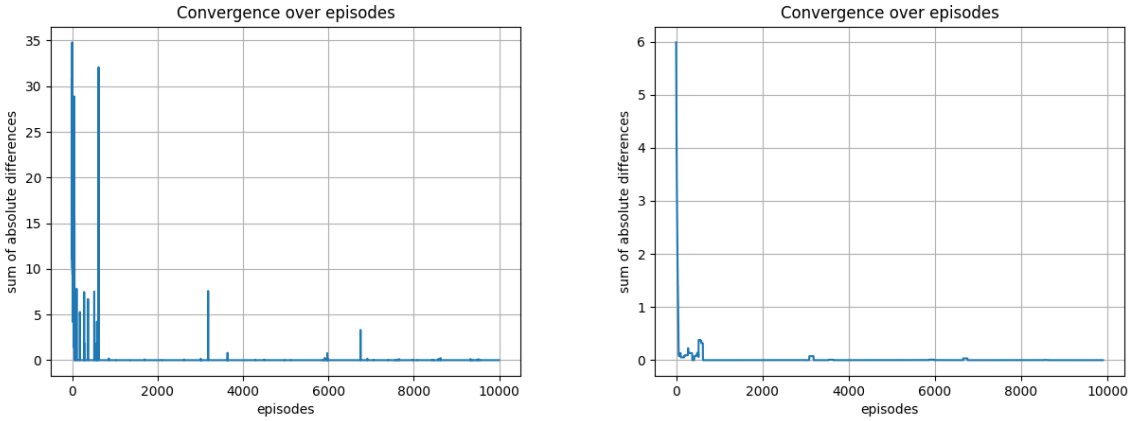


(e) Episode 5000



(f) Episode 10000

Figure 38: Evolution of value function throughout episodes.

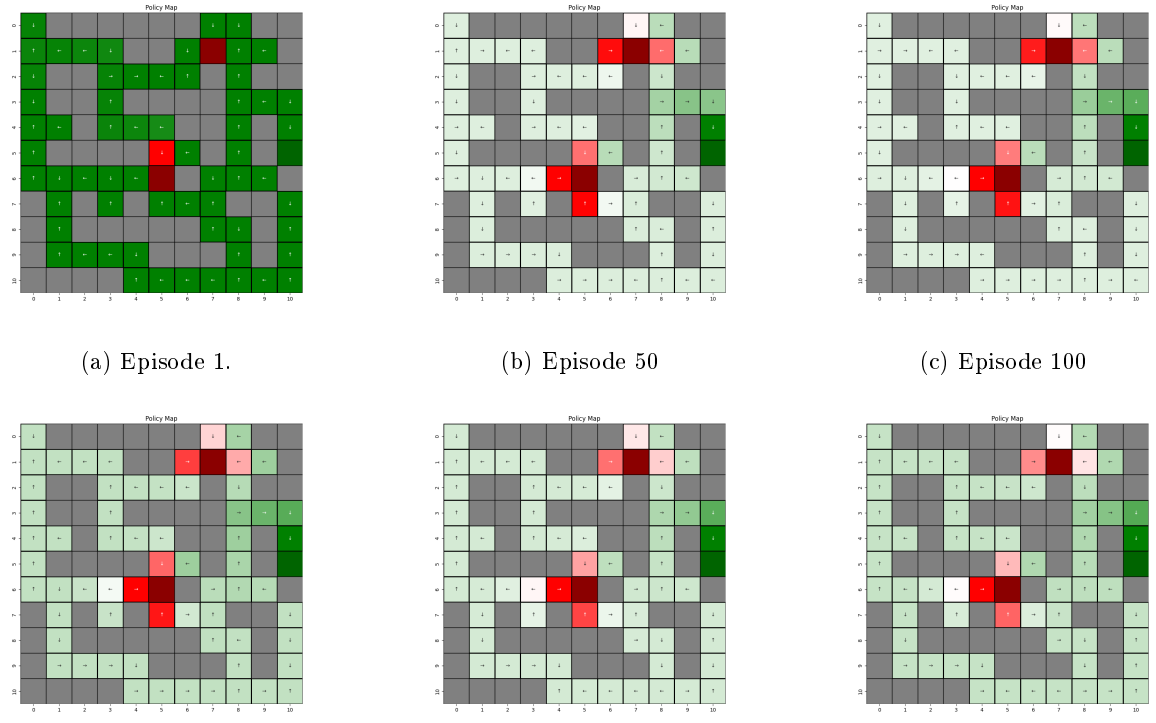


(a) Converge raw.

(b) Smoothed convergence.

Figure 39: Converge of value function.

Figure 40 shows the policy maps for the gamma parameter set to 0.75. Figure 41 illustrates the value function plots for the gamma parameter set to 0.75. Figure 42 provides the convergence plots for the gamma parameter set to 0.75.



(a) Episode 1.

(b) Episode 50

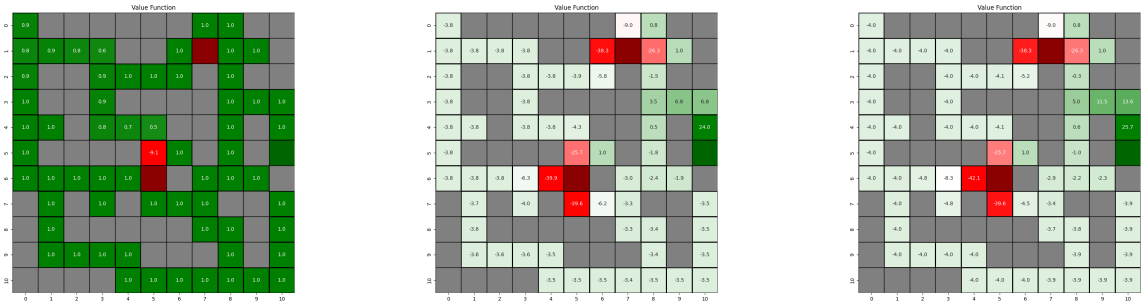
(c) Episode 100

(d) Episode 1000.

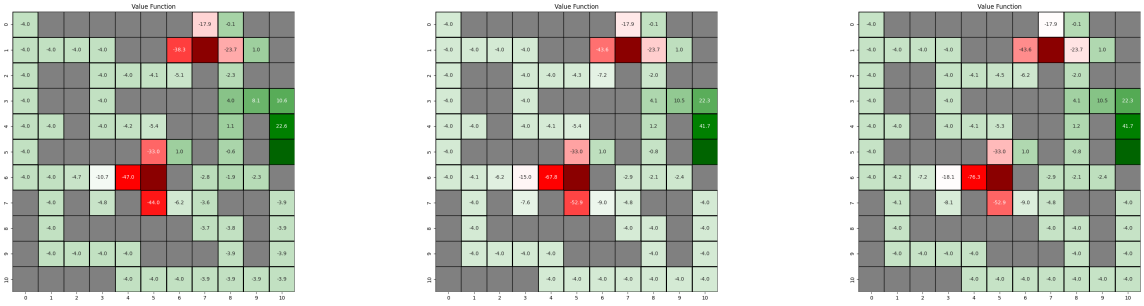
(e) Episode 5000

(f) Episode 10000

Figure 40: Evolution of policy maps throughout episodes.

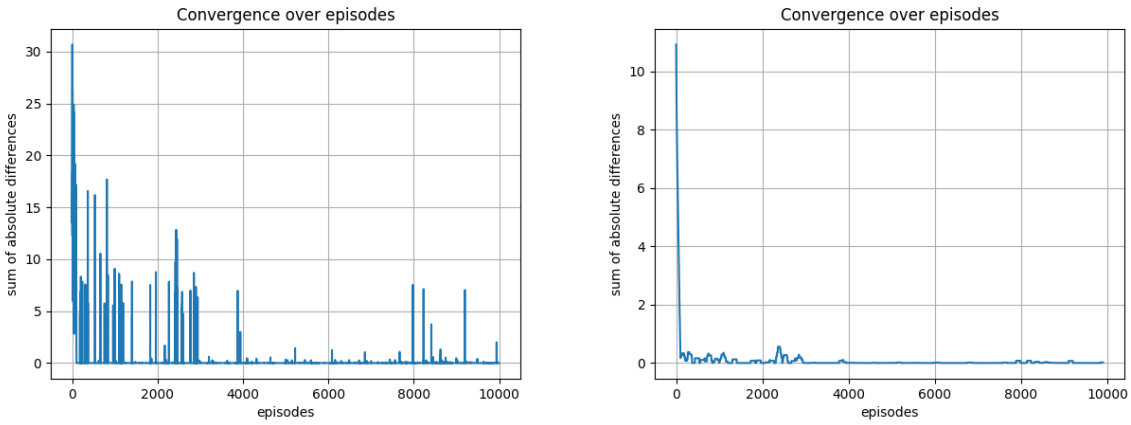


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 41: Evolution of value function throughout episodes.



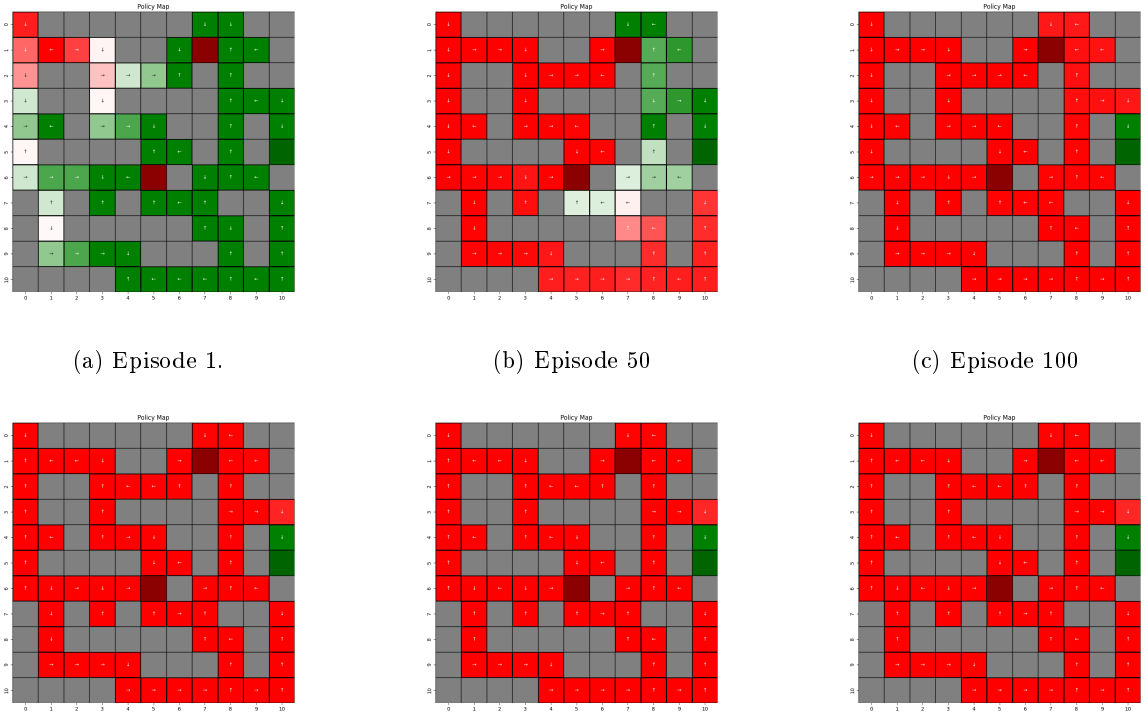
(a) Converge raw. (b) Smoothed convergence.

Figure 42: Converge of value function.

What we can interpret from the results is that the gamma parameter has a significant effect on the convergence of the value function. As we can see from the results, the gamma values up to 0.95 are not feasible choices since the agent rarely changes its policy. Also, since the agents can get stuck at exploitation we see quite long training process where having small discount factor does not help. In order to prevent too much computation time a certain limit is set for maximum number of steps. We can say that the gamma value set to default is the best choice for the gamma parameter amongst the others in temporal difference learning.

2.6 Effect of Gamma in Q-Learning

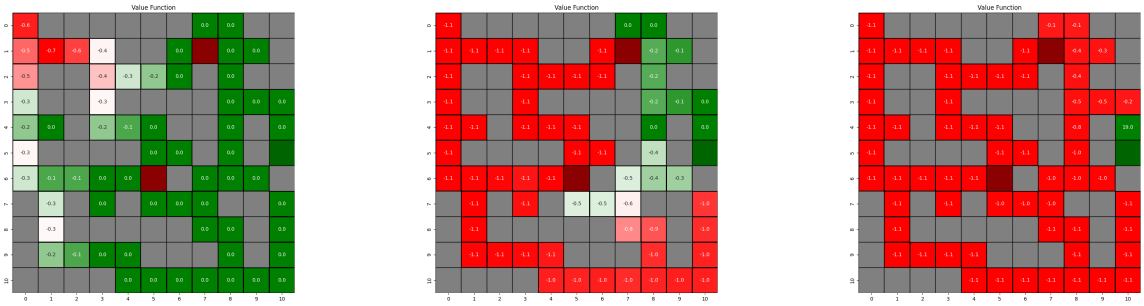
The results for the effect of gamma parameter in Q learning are provided below. The parameter set used for this experiment is as follows: $\alpha = 0.1$, $\epsilon = 0.2$, and the number of episodes is set to 10000. The gamma parameter is varied from 0.1 to 0.95. Figure 43 shows the policy maps for the gamma parameter set to 0.1. Figure 44 illustrates the value function plots for the gamma parameter set to 0.1. Figure 45 provides the convergence plots for the gamma parameter set to 0.1.



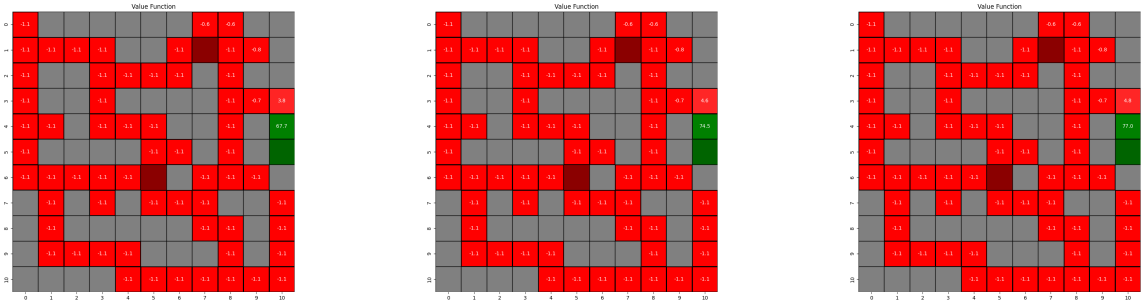
(a) Episode 1. (b) Episode 50 (c) Episode 100

(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 43: Evolution of policy maps throughout episodes.

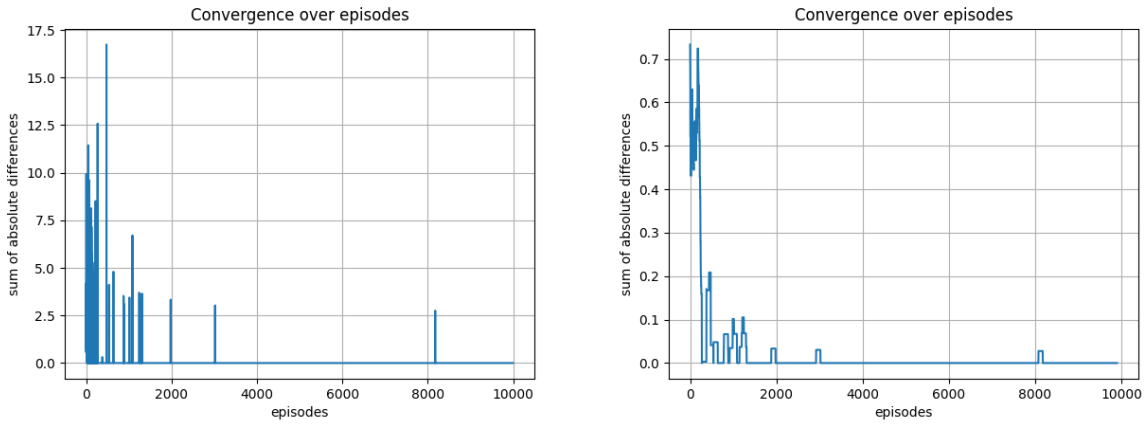


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

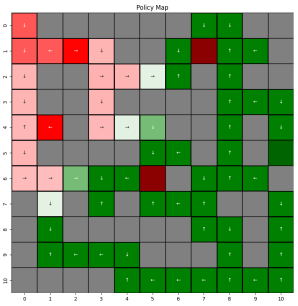
Figure 44: Evolution of value function throughout episodes.



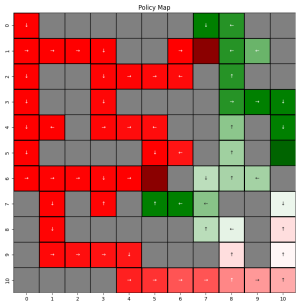
(a) Converge raw. (b) Smoothed convergence.

Figure 45: Converge of value function.

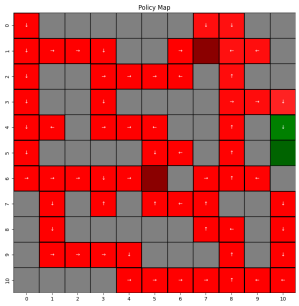
Figure 46 is provided as the policy maps for the gamma parameter set to 0.25. Figure 47 shows the value function plots for the gamma parameter set to 0.25. Figure 48 presents the convergence plots for the gamma parameter set to 0.25.



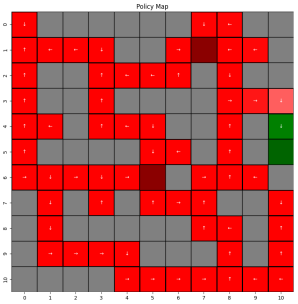
(a) Episode 1.



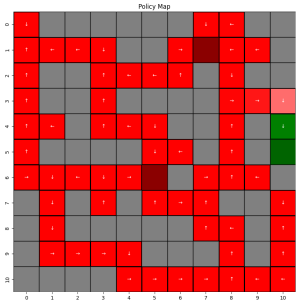
(b) Episode 50



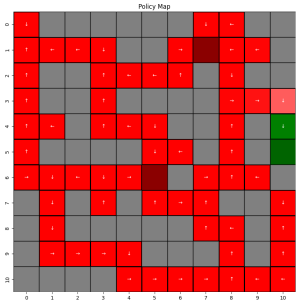
(c) Episode 100



(d) Episode 1000.

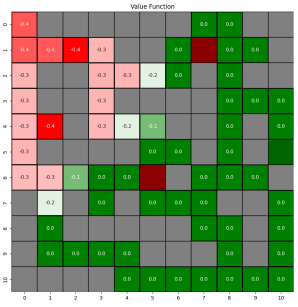


(e) Episode 5000

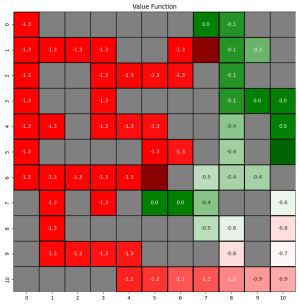


(f) Episode 10000

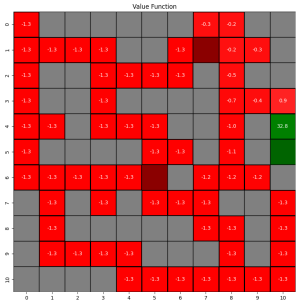
Figure 46: Evolution of policy maps throughout episodes.



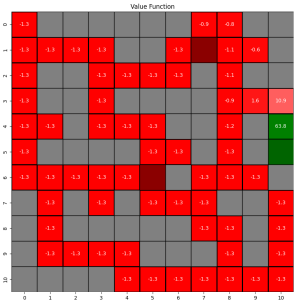
(a) Episode 1.



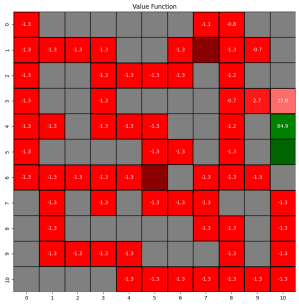
(b) Episode 50



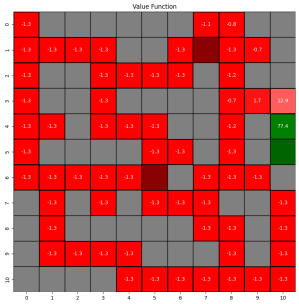
(c) Episode 100



(d) Episode 1000.

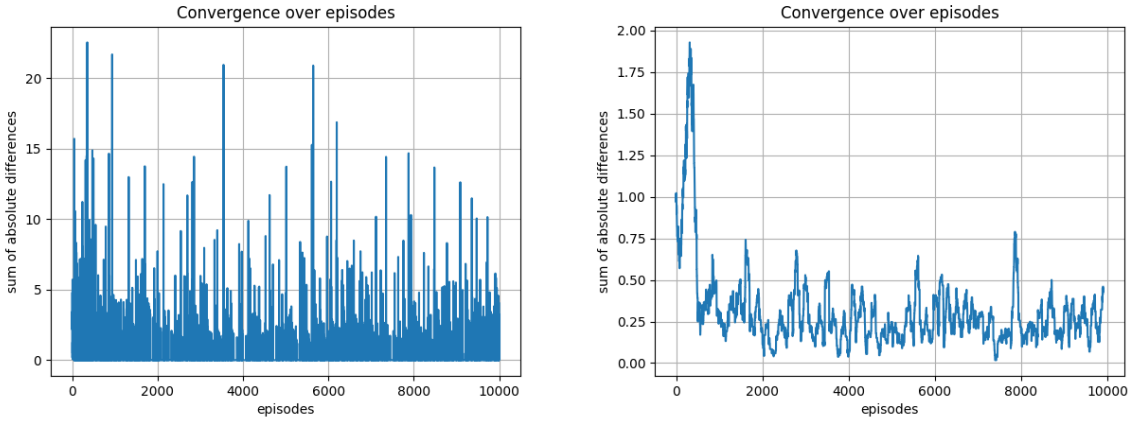


(e) Episode 5000



(f) Episode 10000

Figure 47: Evolution of value function throughout episodes.

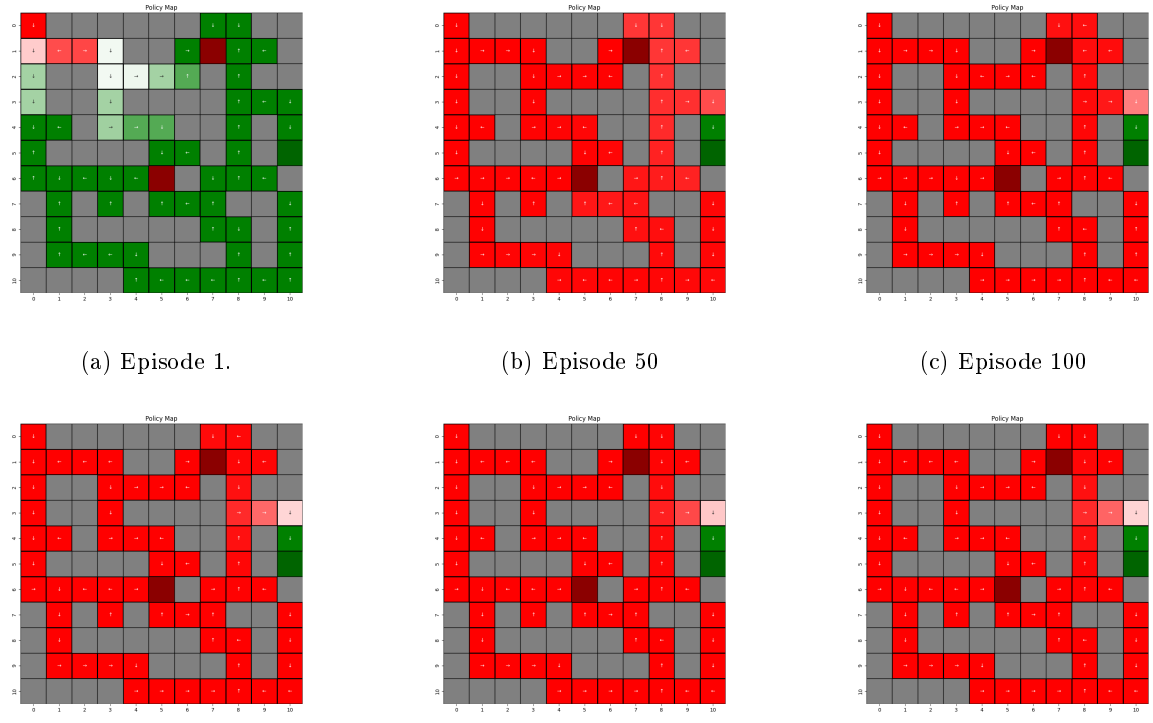


(a) Converge raw.

(b) Smoothed convergence.

Figure 48: Converge of value function.

Figure 49 shows the policy maps for the gamma parameter set to 0.5. Figure 50 illustrates the value function plots for the gamma parameter set to 0.5. Figure 51 provides the convergence plots for the gamma parameter set to 0.5.



(a) Episode 1.

(b) Episode 50

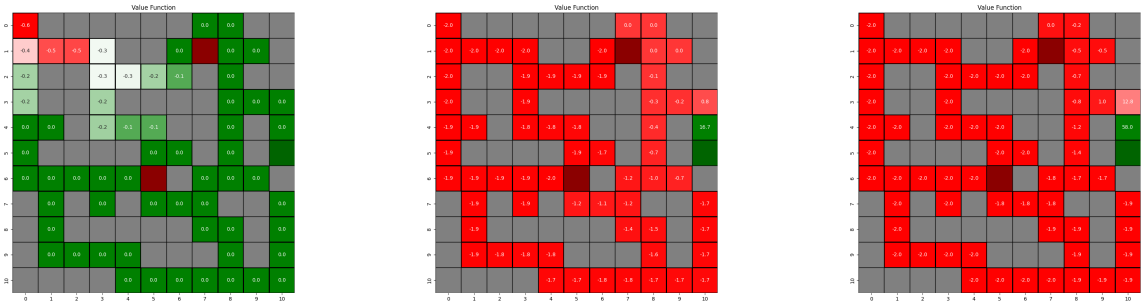
(c) Episode 100

(d) Episode 1000.

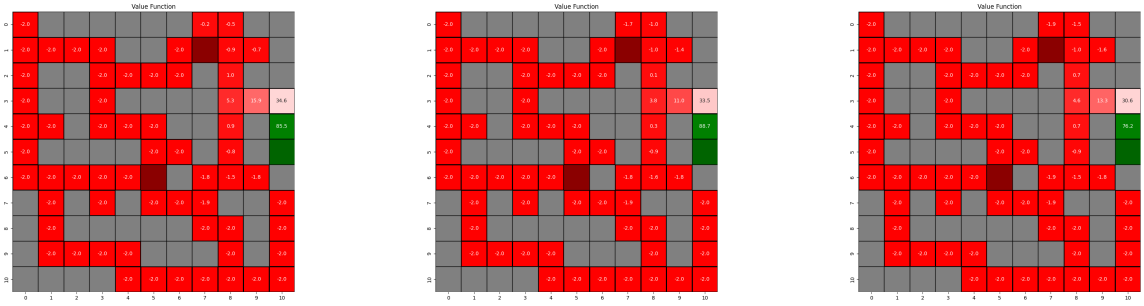
(e) Episode 5000

(f) Episode 10000

Figure 49: Evolution of policy maps throughout episodes.

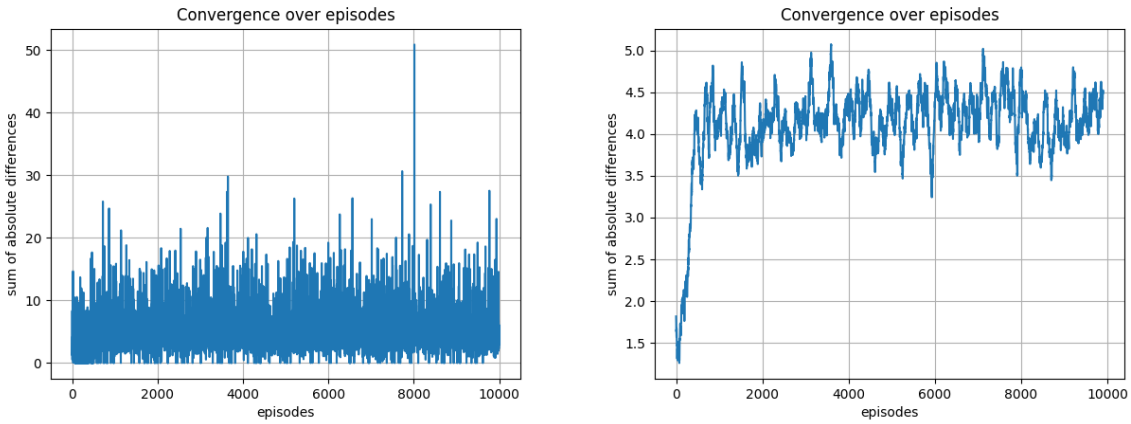


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

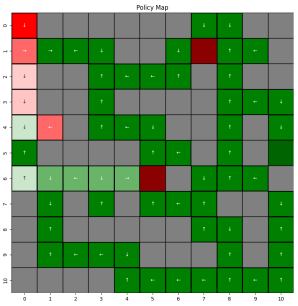
Figure 50: Evolution of value function throughout episodes.



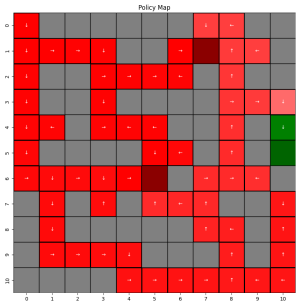
(a) Converge raw. (b) Smoothed convergence.

Figure 51: Converge of value function.

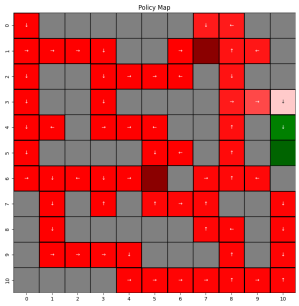
Figure 52 shows the policy maps for the gamma parameter set to 0.75. Figure 53 illustrates the value function plots for the gamma parameter set to 0.75. Figure 54 provides the convergence plots for the gamma parameter set to 0.75.



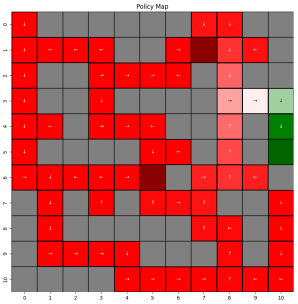
(a) Episode 1.



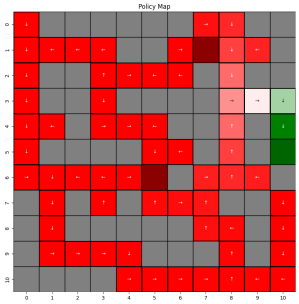
(b) Episode 50



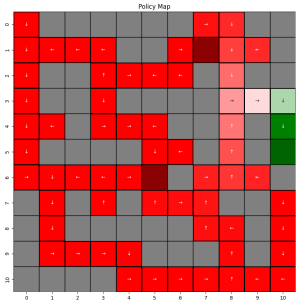
(c) Episode 100



(d) Episode 1000.

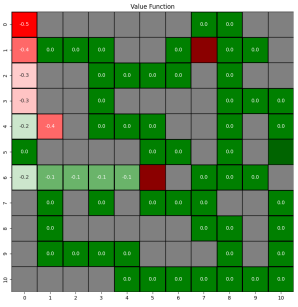


(e) Episode 5000

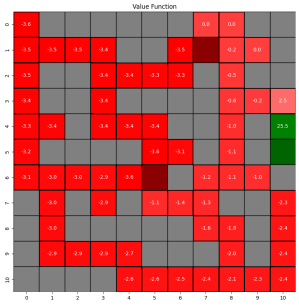


(f) Episode 10000

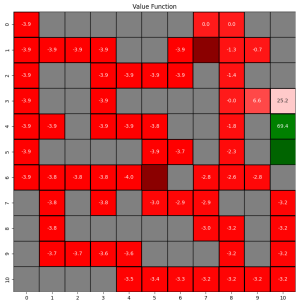
Figure 52: Evolution of policy maps throughout episodes.



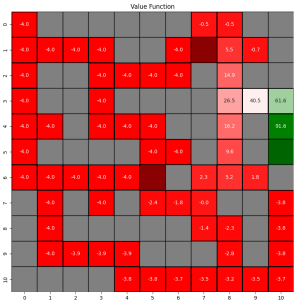
(a) Episode 1.



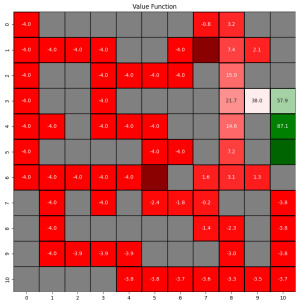
(b) Episode 50



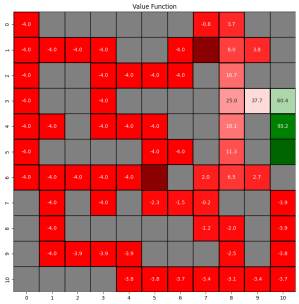
(c) Episode 100



(d) Episode 1000.

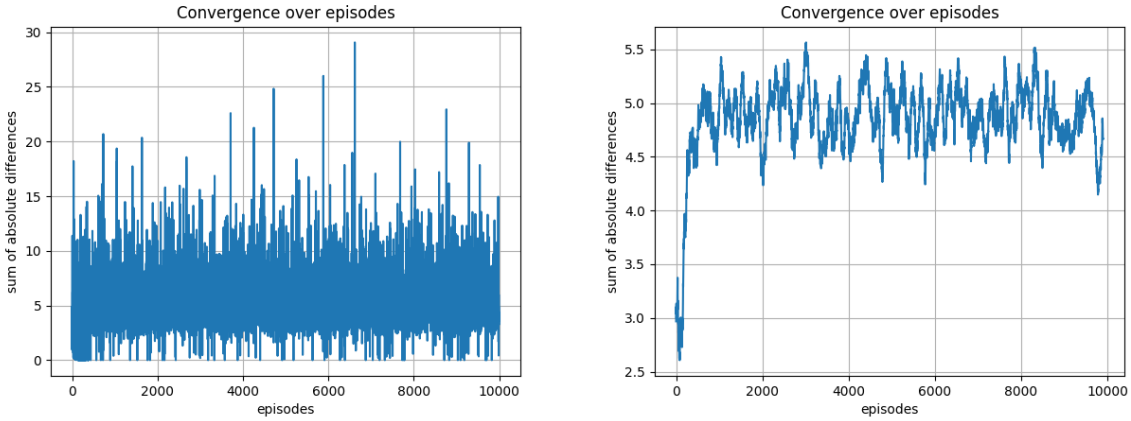


(e) Episode 5000



(f) Episode 10000

Figure 53: Evolution of value function throughout episodes.



(a) Converge raw.

(b) Smoothed convergence.

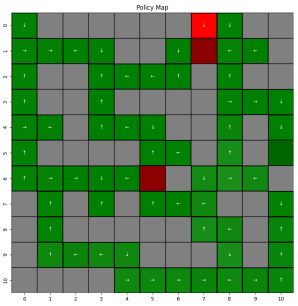
Figure 54: Converge of value function.

Contrary to temporal difference learning, we can say that smaller gamma values can yield to optimal policy for Q learning. From the results, we can see that the gamma value set to above 0.5 can be choosen, otherwise the agent can get stuck at exploitation.

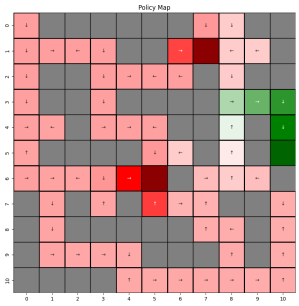
2.7 Effect of Epsilon in Temporal Difference Learning

Here the sweep of epsilon resuls are presented. The parameters used for these experiments are as follows: $\alpha = 0.1$, $\gamma = 0.95$, and the number of episodes is set to 10000. The epsilon parameter is varied from 0.0 to 1.0.

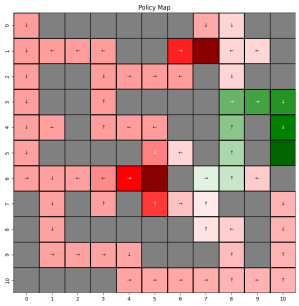
Figure 55 shows the policy maps for the epsilon parameter set to 0.0. Figure 56 illustrates the value function plots for the epsilon parameter set to 0.0. Figure 57 provides the convergence plots for the epsilon parameter set to 0.0.



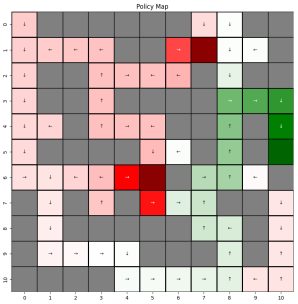
(a) Episode 1.



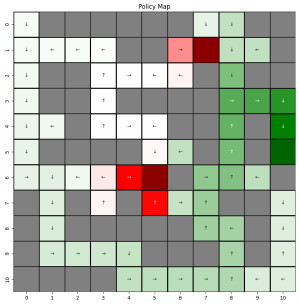
(b) Episode 50



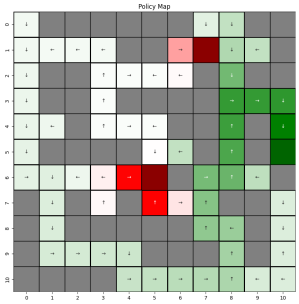
(c) Episode 100



(d) Episode 1000.

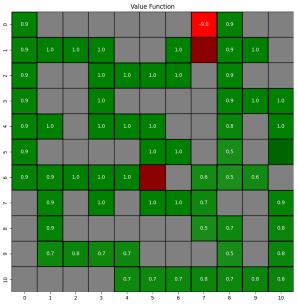


(e) Episode 5000

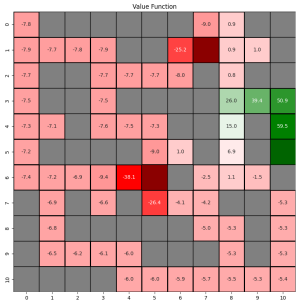


(f) Episode 10000

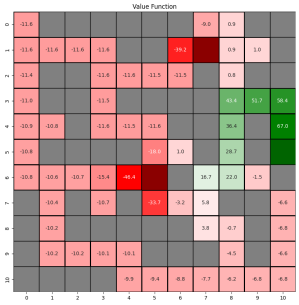
Figure 55: Evolution of policy maps throughout episodes.



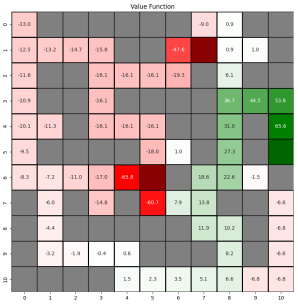
(a) Episode 1.



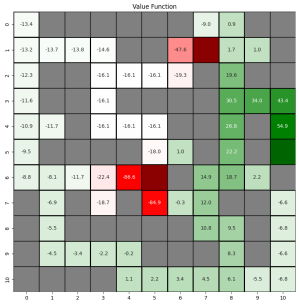
(b) Episode 50



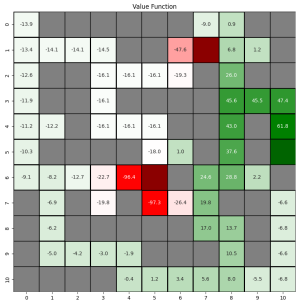
(c) Episode 100



(d) Episode 1000.

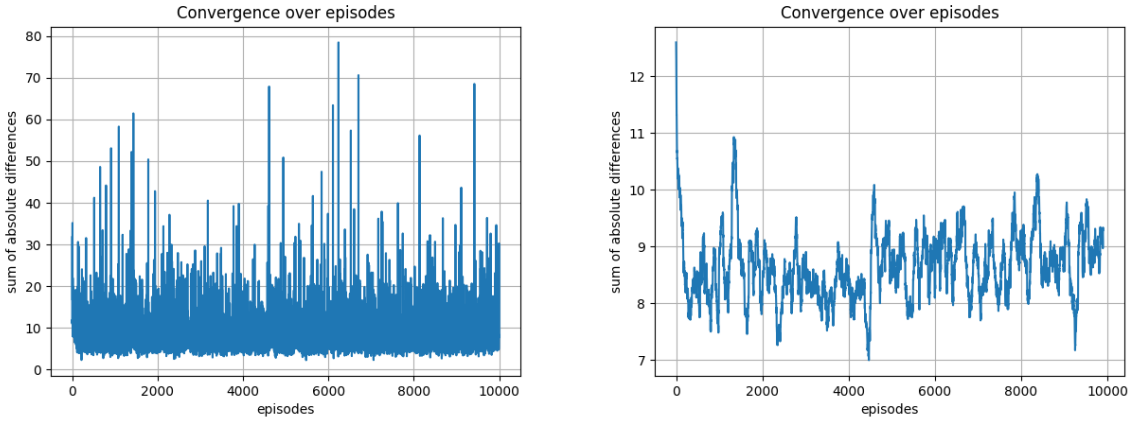


(e) Episode 5000



(f) Episode 10000

Figure 56: Evolution of value function throughout episodes.

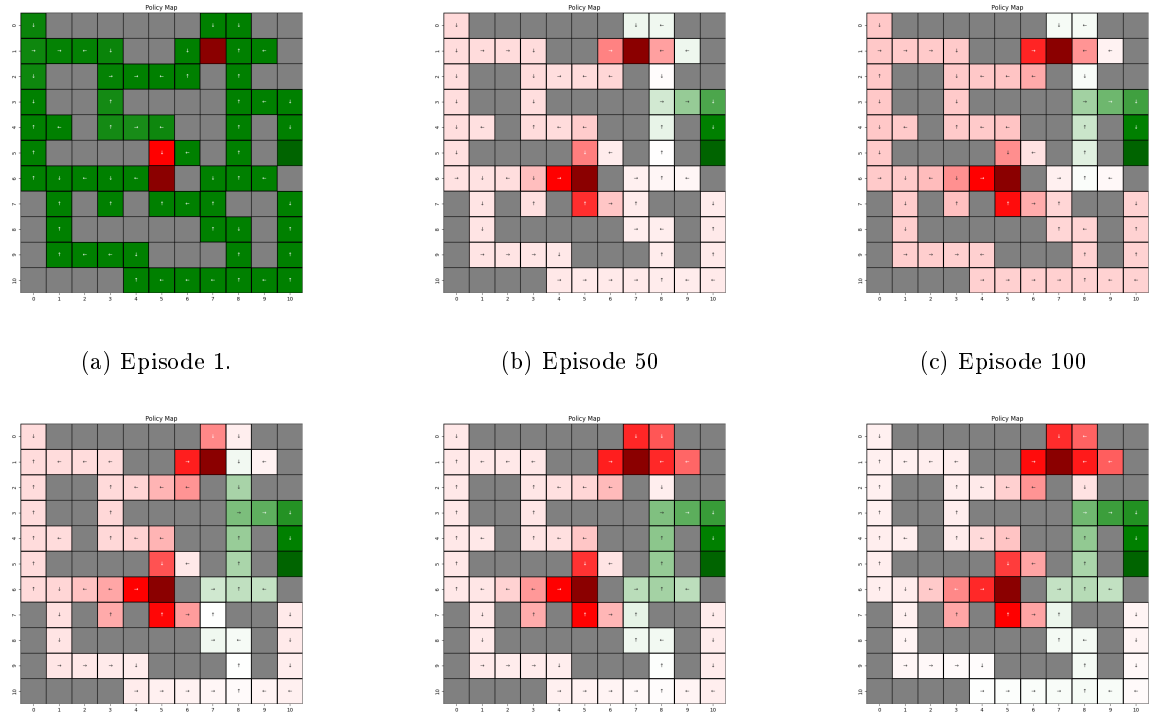


(a) Converge raw.

(b) Smoothed convergence.

Figure 57: Converge of value function.

Figure 58 shows the policy maps for the epsilon parameter set to 0.5. Figure 59 illustrates the value function plots for the epsilon parameter set to 0.5. Figure 60 provides the convergence plots for the epsilon parameter set to 0.5.



(a) Episode 1.

(b) Episode 50

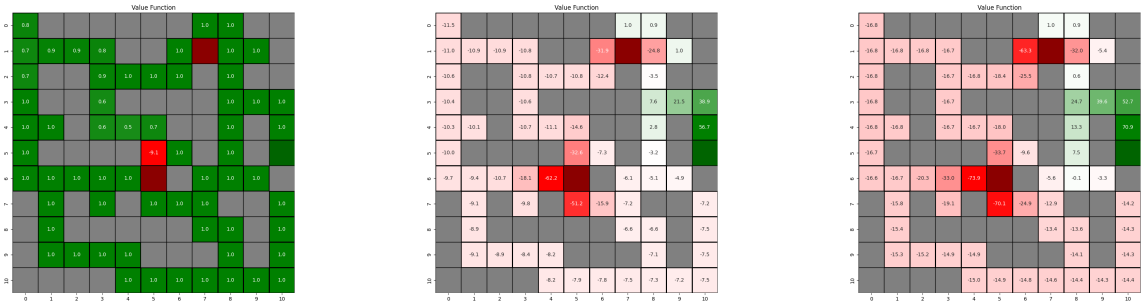
(c) Episode 100

(d) Episode 1000.

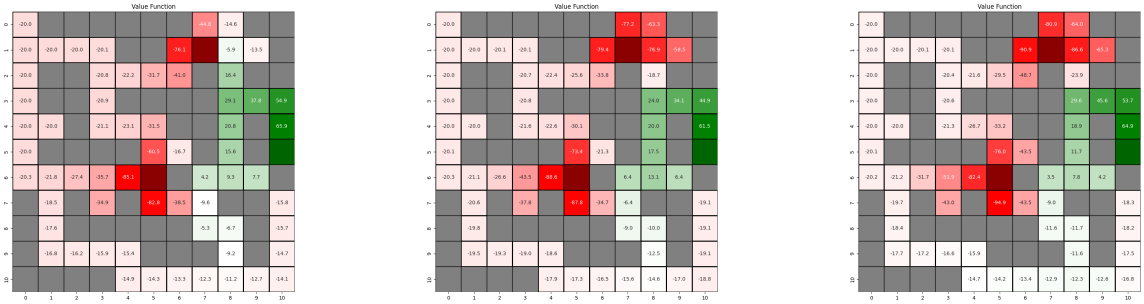
(e) Episode 5000

(f) Episode 10000

Figure 58: Evolution of policy maps throughout episodes.

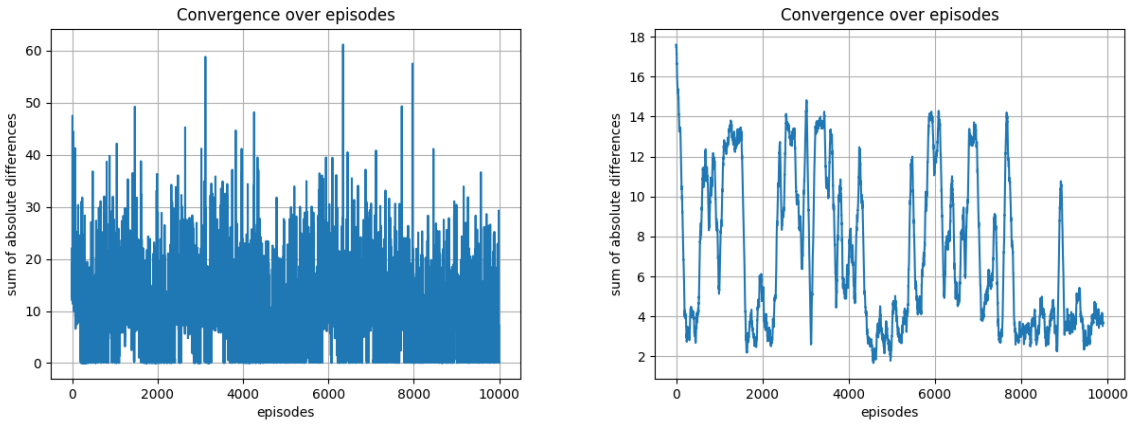


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

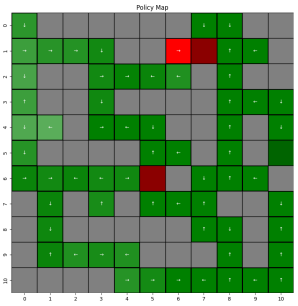
Figure 59: Evolution of value function throughout episodes.



(a) Converge raw. (b) Smoothed convergence.

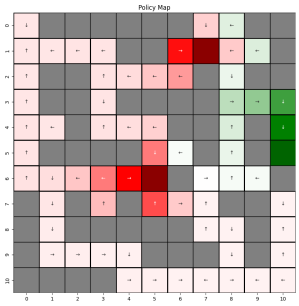
Figure 60: Converge of value function.

Figure 61 shows the policy maps for the epsilon parameter set to 0.8. Figure 62 illustrates the value function plots for the epsilon parameter set to 0.8. Figure 63 provides the convergence plots for the epsilon parameter set to 0.8.

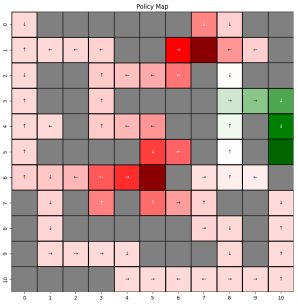


(a) Episode 1.

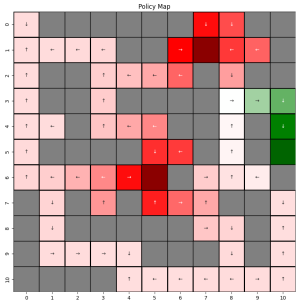
(b) Episode 50



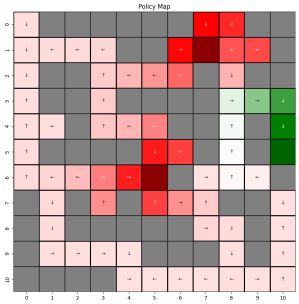
(c) Episode 100



(d) Episode 1000.

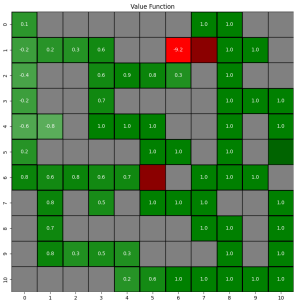


(e) Episode 5000

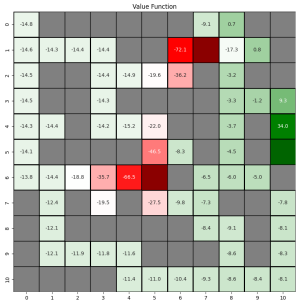


(f) Episode 10000

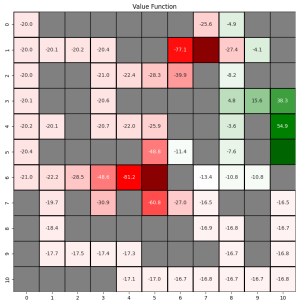
Figure 61: Evolution of policy maps throughout episodes.



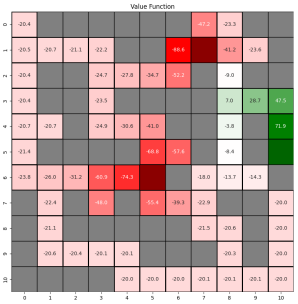
(a) Episode 1.



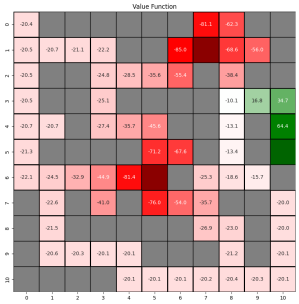
(b) Episode 50



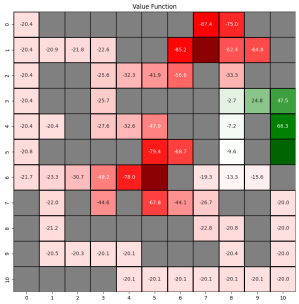
(c) Episode 100



(d) Episode 1000.

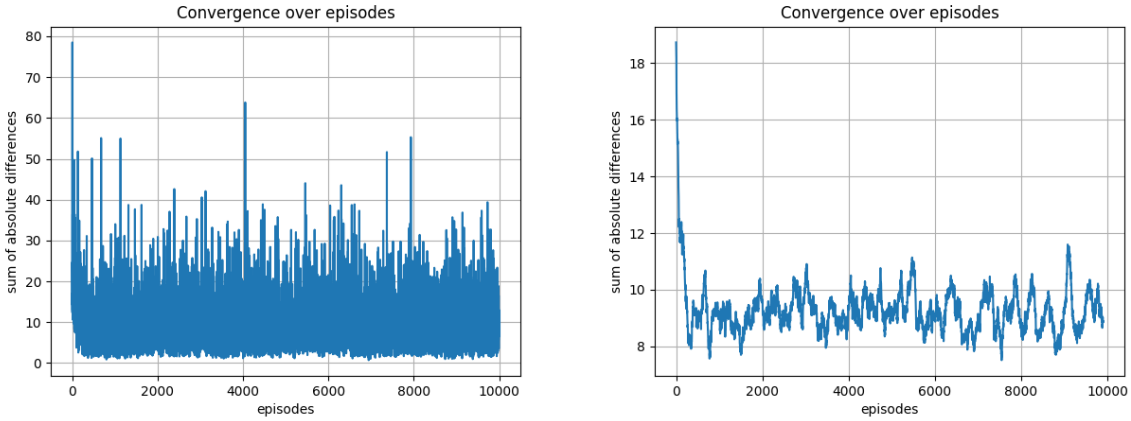


(e) Episode 5000



(f) Episode 10000

Figure 62: Evolution of value function throughout episodes.

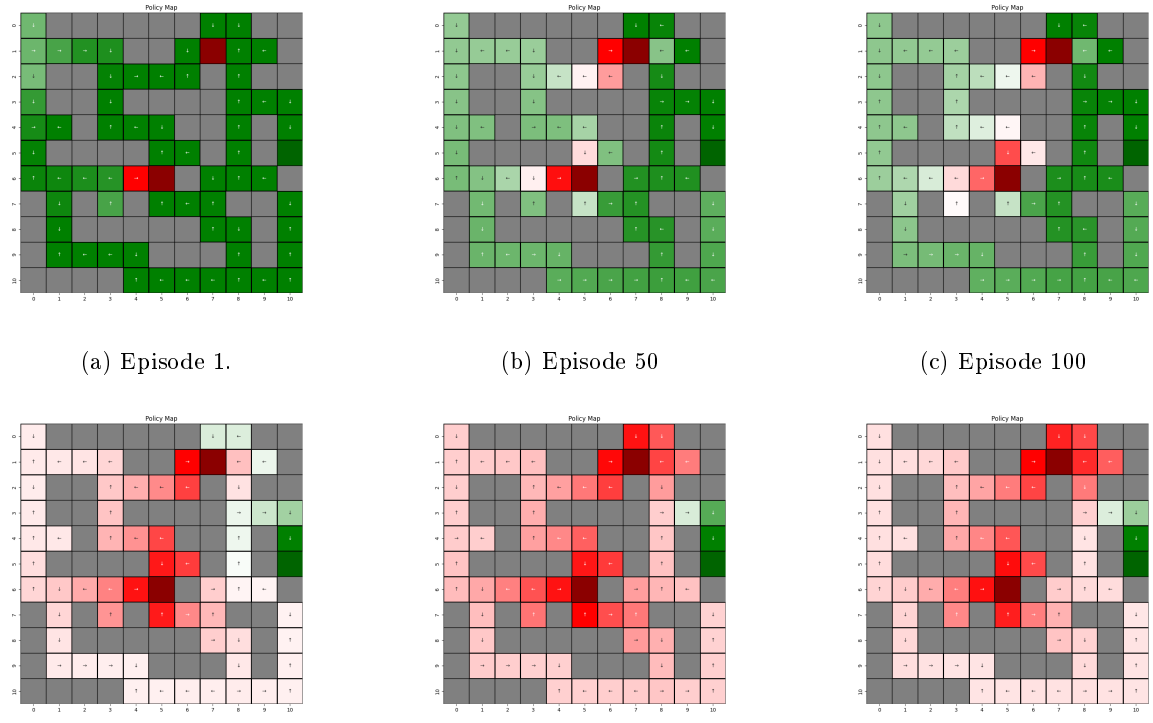


(a) Converge raw.

(b) Smoothed convergence.

Figure 63: Converge of value function.

Figure 64 shows the policy maps for the epsilon parameter set to 1.0. Figure 65 illustrates the value function plots for the epsilon parameter set to 1.0. Figure 66 provides the convergence plots for the epsilon parameter set to 1.0.



(a) Episode 1.

(b) Episode 50

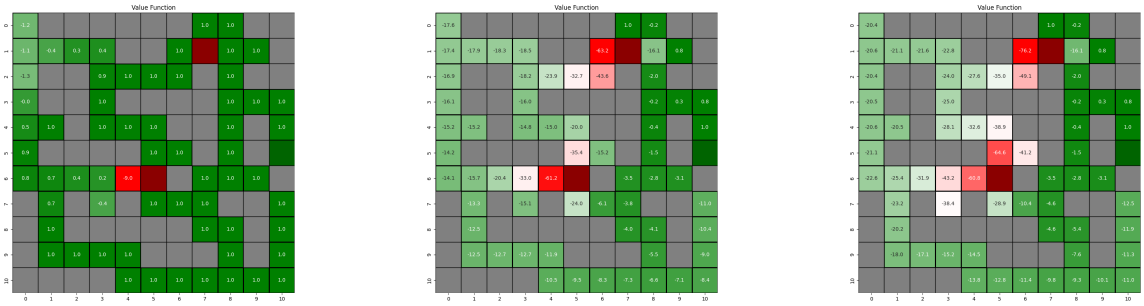
(c) Episode 100

(d) Episode 1000.

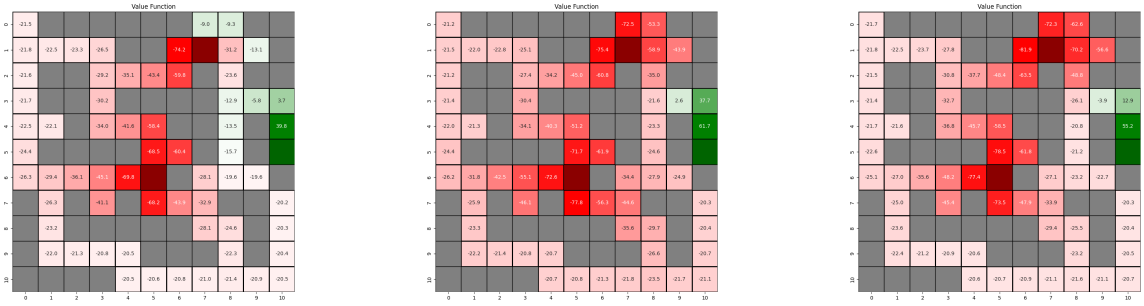
(e) Episode 5000

(f) Episode 10000

Figure 64: Evolution of policy maps throughout episodes.

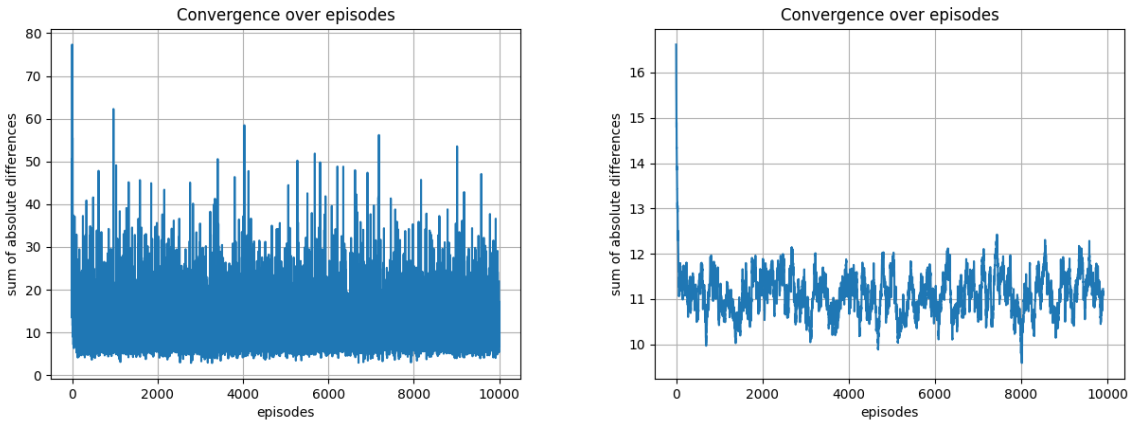


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 65: Evolution of value function throughout episodes.



(a) Converge raw. (b) Smoothed convergence.

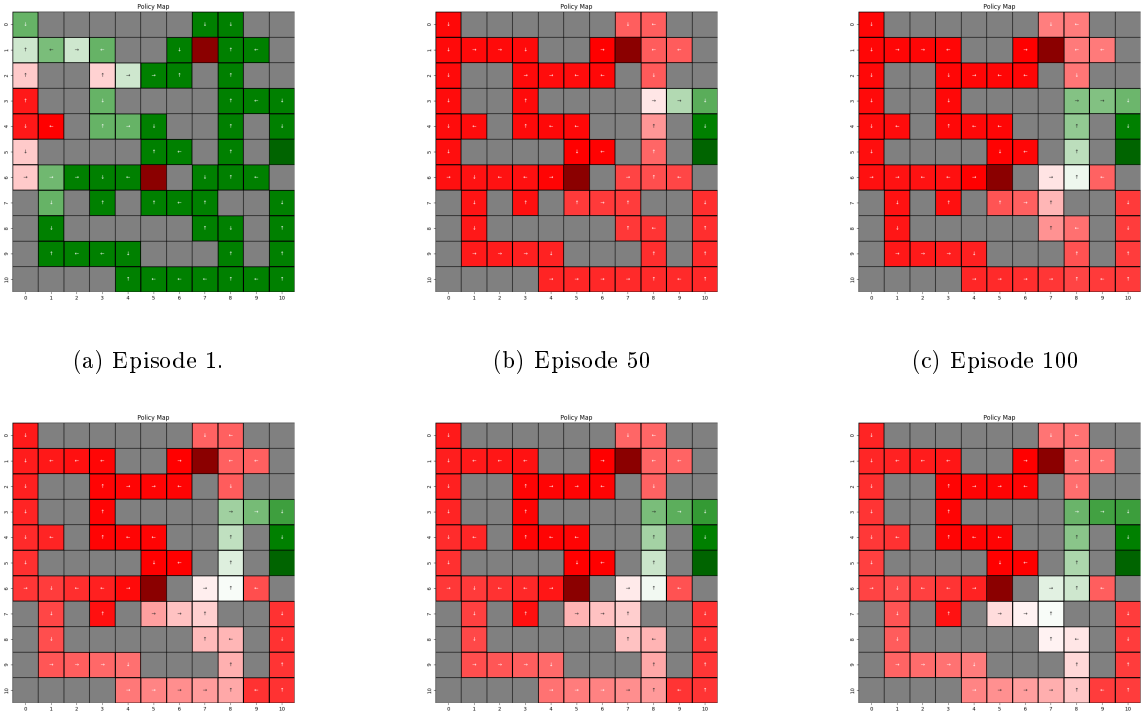
Figure 66: Converge of value function.

What we have observed from the results is that the epsilon parameter is another crucial one for the convergence of the value function. The epsilon parameter set to 0.0 can lead to the optimal policy than the other epsilon values. Even though the epsilon greater that the 0.2 can lead to a converged value function, the optimal policy can not be achieved since exploration-exploitation balance is exceedingly towards exploration.

2.8 Effect of Epsilon in Q-Learning

Using same set of parameters, we have conducted the experiments for Q learning. The epsilon parameter is varied from 0.0 to 1.0.

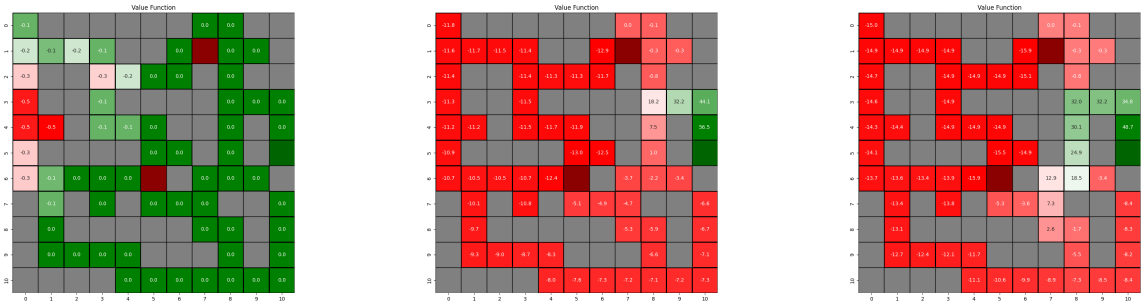
Figure 67 shows the policy maps for the epsilon parameter set to 0.0. Figure 68 illustrates the value function plots for the epsilon parameter set to 0.0. Figure 69 provides the convergence plots for the epsilon parameter set to 0.0.



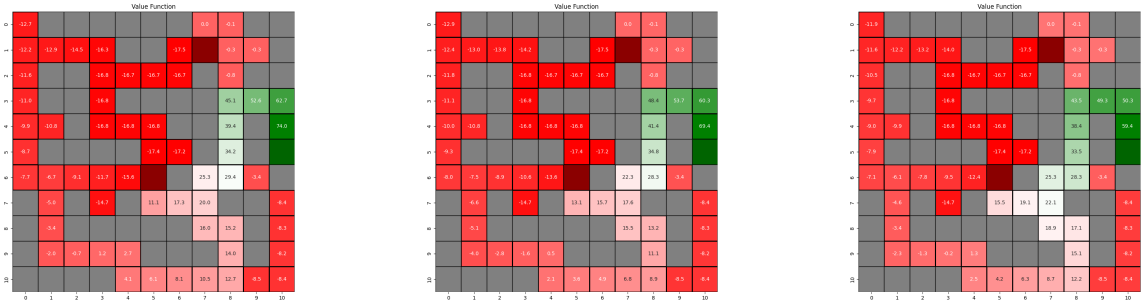
(a) Episode 1. (b) Episode 50 (c) Episode 100

(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

Figure 67: Evolution of policy maps throughout episodes.

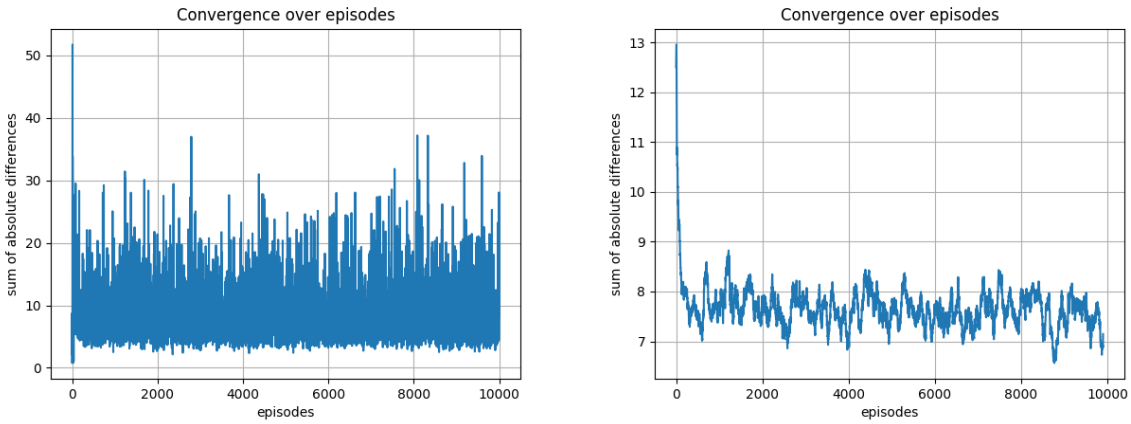


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

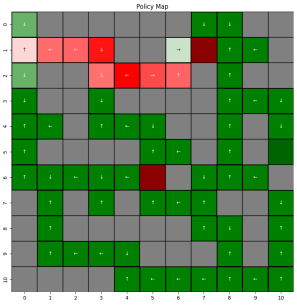
Figure 68: Evolution of value function throughout episodes.



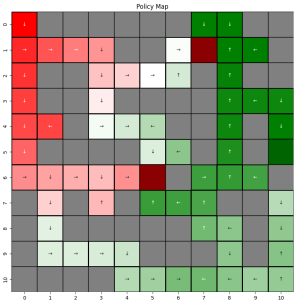
(a) Converge raw. (b) Smoothed convergence.

Figure 69: Converge of value function.

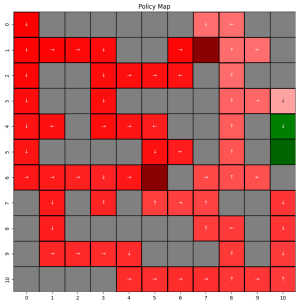
Figure 70 shows the policy maps for the epsilon parameter set to 0.5. Figure 71 illustrates the value function plots for the epsilon parameter set to 0.5. Figure 72 provides the convergence plots for the epsilon parameter set to 0.5.



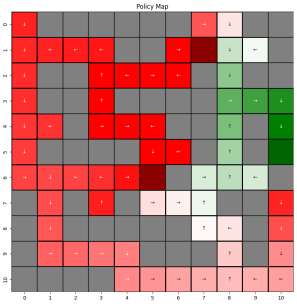
(a) Episode 1.



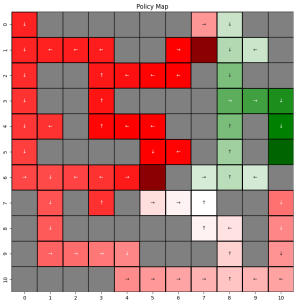
(b) Episode 50



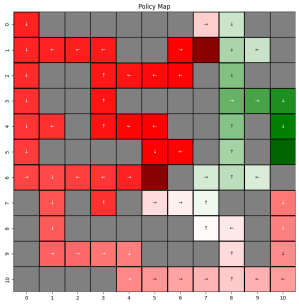
(c) Episode 100



(d) Episode 1000.

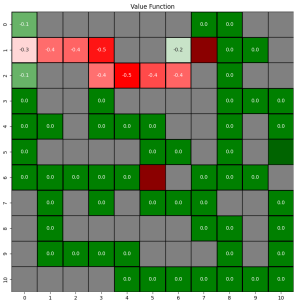


(e) Episode 5000

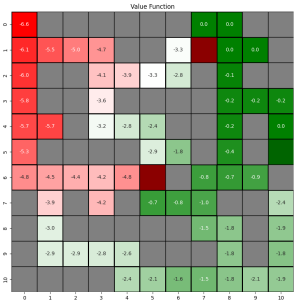


(f) Episode 10000

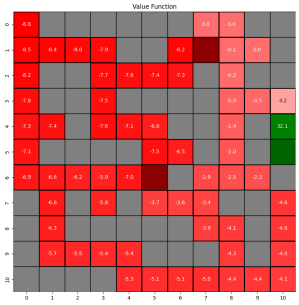
Figure 70: Evolution of policy maps throughout episodes.



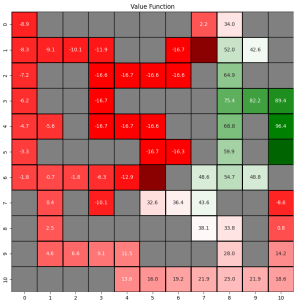
(a) Episode 1.



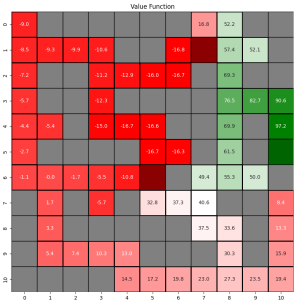
(b) Episode 50



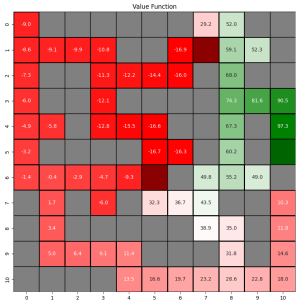
(c) Episode 100



(d) Episode 1000.

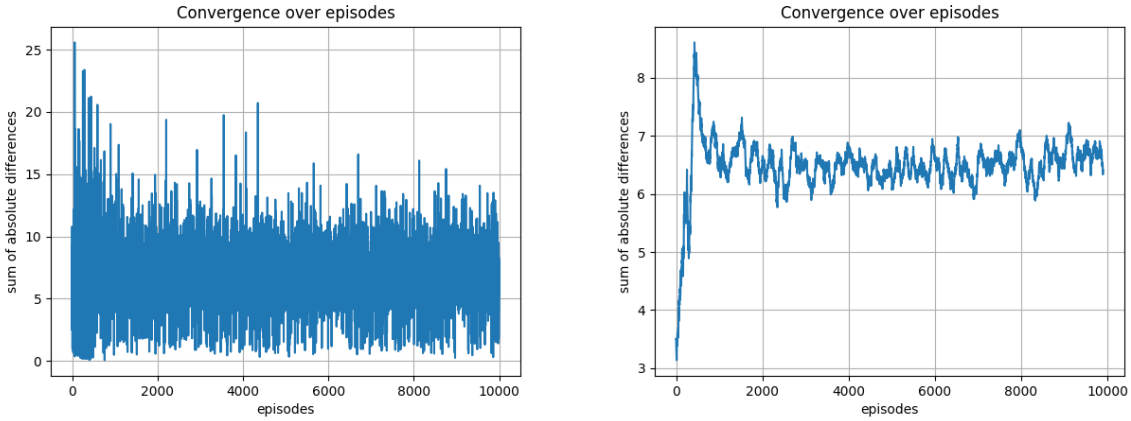


(e) Episode 5000



(f) Episode 10000

Figure 71: Evolution of value function throughout episodes.

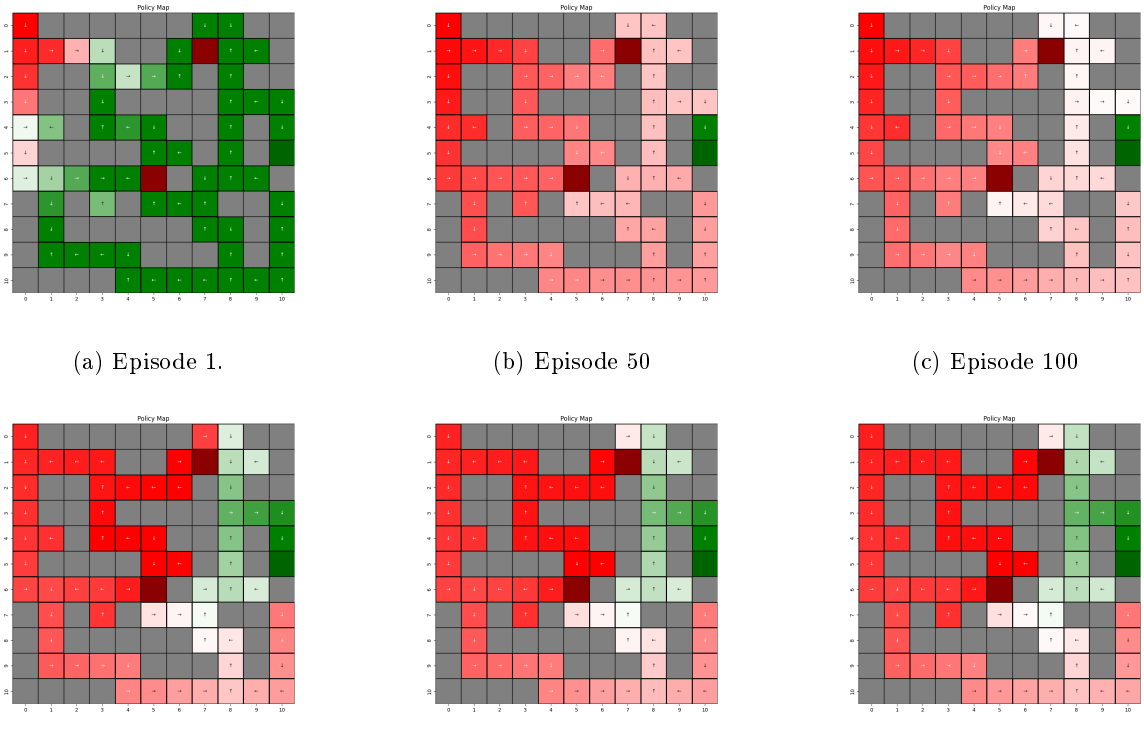


(a) Converge raw.

(b) Smoothed convergence.

Figure 72: Converge of value function.

Figure 73 shows the policy maps for the epsilon parameter set to 0.8. Figure 74 illustrates the value function plots for the epsilon parameter set to 0.8. Figure 75 provides the convergence plots for the epsilon parameter set to 0.8.



(a) Episode 1.

(b) Episode 50

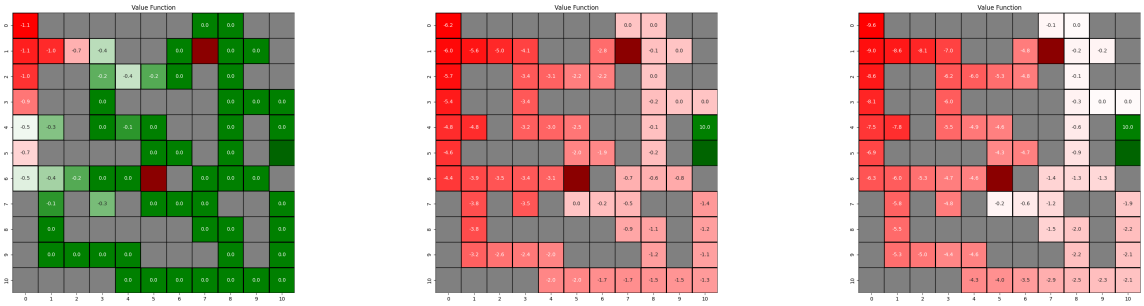
(c) Episode 100

(d) Episode 1000.

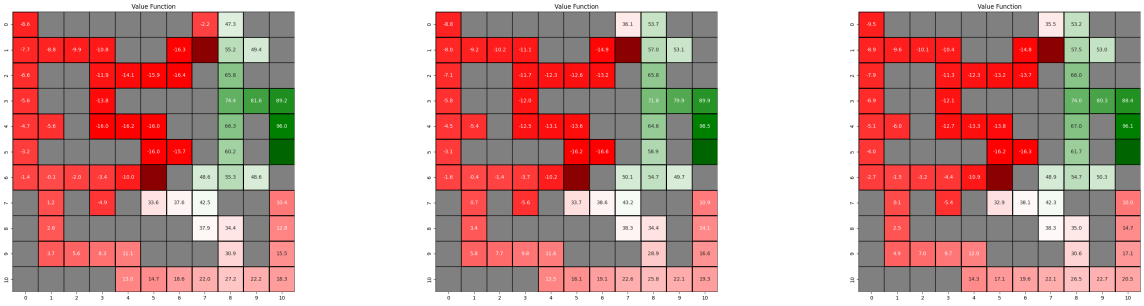
(e) Episode 5000

(f) Episode 10000

Figure 73: Evolution of policy maps throughout episodes.

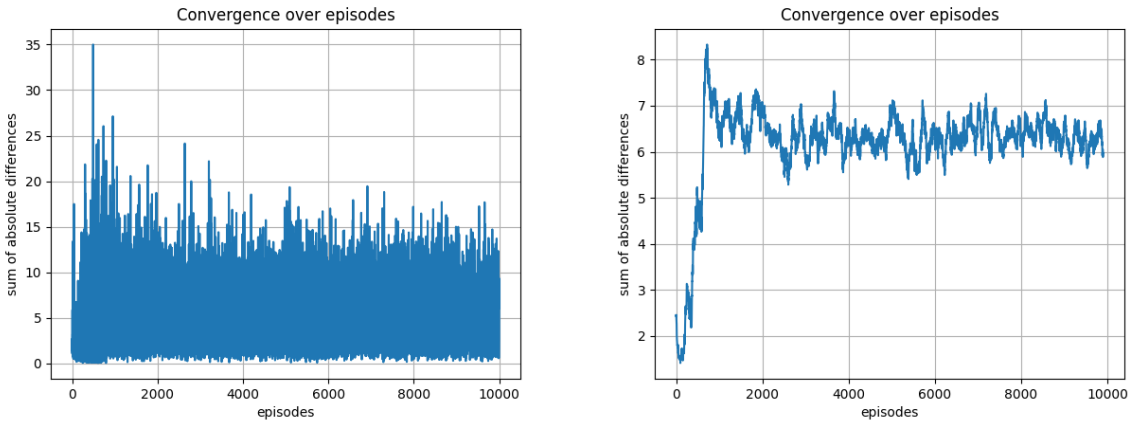


(a) Episode 1. (b) Episode 50 (c) Episode 100



(d) Episode 1000. (e) Episode 5000 (f) Episode 10000

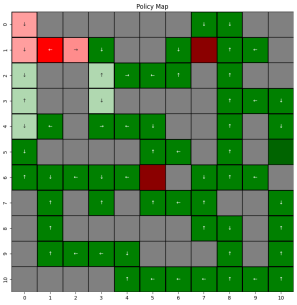
Figure 74: Evolution of value function throughout episodes.



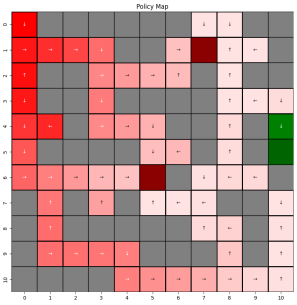
(a) Converge raw. (b) Smoothed convergence.

Figure 75: Converge of value function.

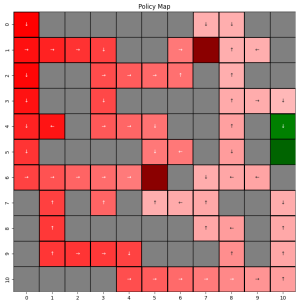
Figure 76 shows the policy maps for the epsilon parameter set to 1.0. Figure 77 illustrates the value function plots for the epsilon parameter set to 1.0. Figure 78 provides the convergence plots for the epsilon parameter set to 1.0.



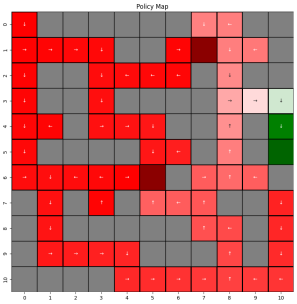
(a) Episode 1.



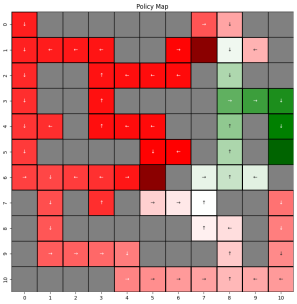
(b) Episode 50



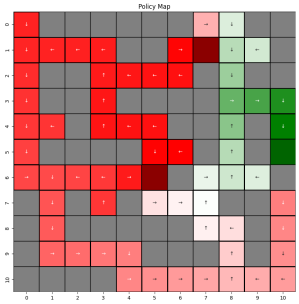
(c) Episode 100



(d) Episode 1000.

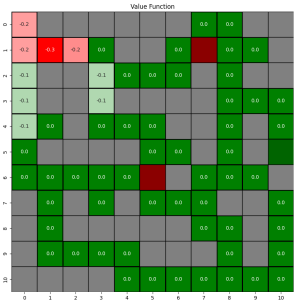


(e) Episode 5000

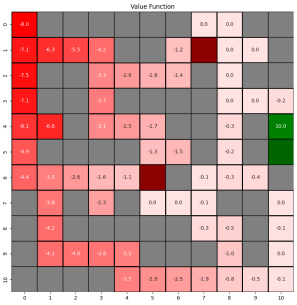


(f) Episode 10000

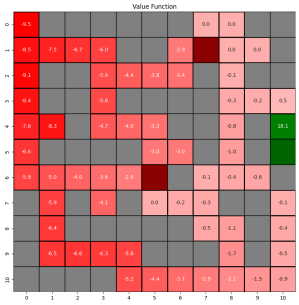
Figure 76: Evolution of policy maps throughout episodes.



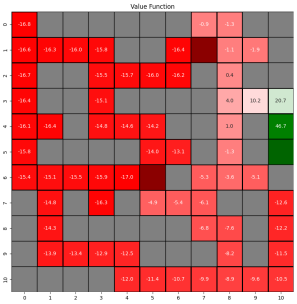
(a) Episode 1.



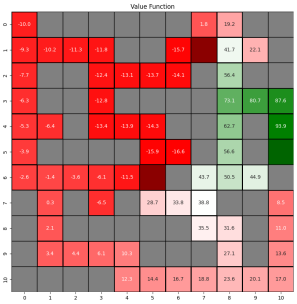
(b) Episode 50



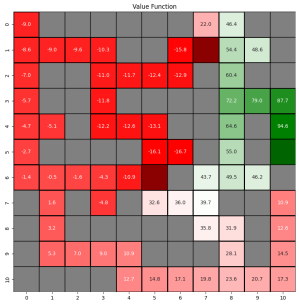
(c) Episode 100



(d) Episode 1000.

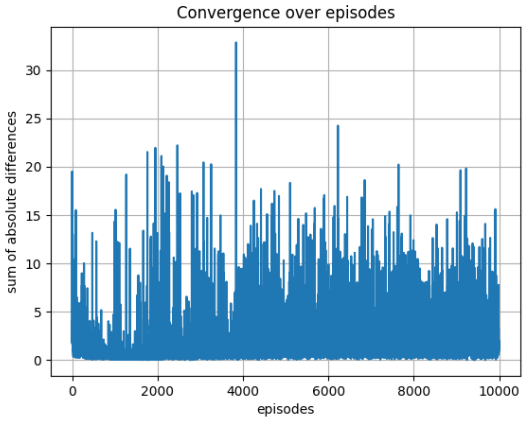


(e) Episode 5000

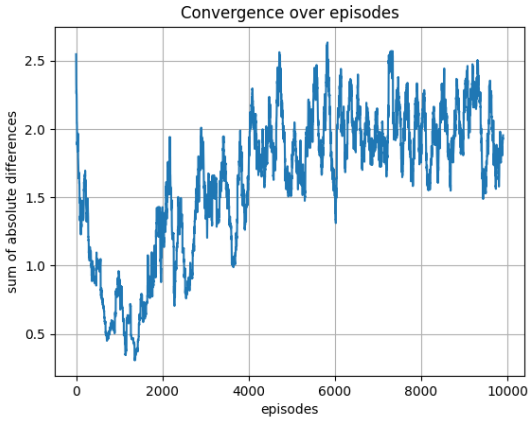


(f) Episode 10000

Figure 77: Evolution of value function throughout episodes.



(a) Converge raw.



(b) Smoothed convergence.

Figure 78: Converge of value function.

What we observed from the Q-learning experiments that all of them have converged to the optimal policy. The epsilon parameter does not have a significant effect on the eventual convergence of the value function, where the epsilon parameter associated with the exploration side of the exploration-exploitation trade-off.

3 Discussions

3.1 Q1

The rationale behind transition probabilities and reward function can be explained as follows. The transition probabilities are the probabilities of transitioning from one state to another state, given an action. The stochastic setting is used in this experiment where the chosen action may not be the action that the agent will take. This allows the agent to explore the environment and learn the optimal policy. The reward function is the function that provides the reward for the agent when it takes an action in a state. So, the reward function is the feedback mechanism that allows the agent to learn the optimal policy.

3.2 Q2

By looking at the value maps in default settings, we can describe the evolution of the value function as follows. The value function is initialized with zeros. The value function is updated by the Bellman equation. The value function is updated by the sum of the reward and the discounted value of the next state. The value function is updated iteratively until convergence occurs. Here, at the first iteration, exploration is more dominant since most of the values are zeros. At first trials, the agent hits traps, and episodes proceed. Then, the agent starts to explore the regions close to the goal. When the agent hits the goal states, the values of the nearby states start to increase, and, roughly speaking, positive reinforcement starts to propagate as the optimal path gets higher and higher values from goal to start state episode by episode.

3.3 Q3

Utility value function converged for TD(0) learning in cases convergence is said to be available. Approximately the first 1000 episodes were the time the values more or less converged.

3.4 Q4

As explained, TD(0) learning is sensitive to learning rate (alpha) and discount factor (gamma) parameters. As the learning rate increases, the value function converges faster. However, if the learning rate is too high, the value function may not converge. The discount factor is another crucial parameter for the convergence of the value function. The discount factor determines the importance of future rewards. If the discount factor is too high, the value function may not converge. If the discount factor is too low, the value function may converge to the suboptimal policy.

3.5 Q5

I encountered two important points while implementing TD(0) learning. First, the borders of the map need to be handled properly, as in invalid states. I set an additional layer of invalid states, making the map (12, 12), which makes the map borders shared (like torus-shaped periodicity). This allowed us to have an equally generic algorithm in every state. Second, utility values in invalid states were initially set to the same value as others. This caused the agent to get stuck in the invalid states while exploitation was dominant. I set the utility values of invalid states to quite negative values so that the policy maps also do not deteriorate.

3.6 Q6

We can see from the experiments that the Q-learning stabilizes much faster than the temporal difference learning algorithm. The reason is that the Q-learning algorithm is an off-policy algorithm, which means that the Q-learning algorithm learns the optimal policy while following another policy. The TD (0) only evaluates one possible action, whereas the Q-learning evaluates all possible actions.

3.7 Q7

The epsilon parameter (exploration rate) allows us to set a balance between exploration and exploitation. If the epsilon parameter is set to 0.0, the agent will always exploit the environment. If the epsilon parameter is set to 1.0, the agent will always explore the environment. Here, we also see from the experiments that if we select the exploration rate too high, the agent will not be able to learn the optimal policy. However, it is important to note that the epsilon parameter is ineffective for the Q-learning algorithm. The reason is that the Q-learning algorithm is an off-policy algorithm, which means that the Q-learning algorithm learns the optimal policy while following another policy. The TD (0) only evaluates one possible action, whereas the Q-learning evaluates all possible actions.

3.8 Q8

Comparing TD(0) and Q-learning, we can say that the Q-learning algorithm is more stable and converges faster than the TD(0) learning algorithm. So, I would prefer using the Q-learning algorithm for the given problem and other similar problems.

3.9 Q9

As the number of steps needed to be taken to reach the goal increases, the number of episodes needed to converge also increases. The reason is that the agent needs to explore more states to learn the optimal policy. The number of episodes needed to converge is directly proportional to the number of steps needed to be taken to reach the goal. Another factor would be the proximity of traps and goals. If the traps are close to the goal, the agent needs to explore more states to learn the optimal policy. So, in our case,

there was only one bottleneck, and the agent had to overcome it to reach the goal. For example, if the agent was surrounded by more traps, the number of episodes needed to converge would increase. The opposite of these cases would yield improved learning. For problem setup, if the invalid states were not subject to exploration, the learning process would be much faster. Another point would be assigning very high negative utility values to traps and very high positive utility values to goal states. This would make the agent learn the optimal policy faster; in other words, the propagation of positive and negative reinforcement would be faster.

3.10 Q10

A very basic adjustment to the algorithm would be scheduling the epsilon parameter where exploration is encouraged in the first episode and exploitation is favored in later episodes. Another optimization would be adding a "number of steps to reach the goal" parameter to the algorithm as something to be optimized. This way, the agent would be able to learn the optimal policy faster.

Appendix

The code set implemented throughout this homework is provided as follows.

```
# This file will include the implementation of reinforcement learning homework to solve maze problem using temporal difference learning and q-learning

import numpy as np
import matplotlib.pyplot as plt
import random
from utils import plot_value_function, plot_policy

from tqdm import tqdm


class MazeEnvironment:
    def __init__(self):
        # Define the maze layout, rewards, action space (up, down, left, right)
        self.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],
            ]
        )
        self.start_pos = (0, 0)  # Start position of the agent
        self.current_pos = self.start_pos
        self.state_penalty = -1
        self.trap_penalty = -100
        self.goal_reward = 100
        self.actions = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)}

```

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

    def step(self, action):
        # Simulate the action based on stochastic environment
        random_val = np.random.uniform(0, 1)
        if random_val <= 0.75:  # Probability of going for the chosen direction
            dx, dy = self.actions[action]
        elif (
            0.75 < random_val <= 0.80
        ):  # Probability of going opposite of the chosen direction
            dx, dy = tuple([-1 * x for x in self.actions[action]])
        elif (
            0.80 < random_val <= 0.90
        ):  # Probability of going each of perpendicular routes
            dy, dx = self.actions[action]  # Swap the x and y coordinates
        else:
            dy, dx = self.actions[action]
            dx, dy = -dy, -dx

        new_x, new_y = self.current_pos[0] + dx, self.current_pos[1] + dy

        # Check if the new position is valid
        if (
            (0 <= new_x < self.maze.shape[0])
            and (0 <= new_y < self.maze.shape[1])
            and (self.maze[new_x, new_y] != 1)
        ):
            self.current_pos = (new_x, new_y)
        else:  # If the new position is invalid, stay in the current position
            self.current_pos = self.current_pos

            return self.current_pos, self.state_penalty

        # Determine reward based on new position
        if self.maze[self.current_pos] == 2:
            reward = self.trap_penalty
        elif self.maze[self.current_pos] == 3:
            reward = self.goal_reward
        else:
            reward = self.state_penalty

        return self.current_pos, reward


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.ones((12,12))
        self.utility[np.where(maze.maze == 1)] = -1000
        self.utility[11,:] = -1000
        self.utility[:,11] = -1000
        self.valid_actions = list(maze.actions.keys())
```

```
95
96          self.convergence_data = []
97          self.episodes_to_plot = [1, 50, 100, 1000, 5000, 10000]
98          self.previous_values = np.zeros((11, 11))
99
100         self.wander_limit = 1000
101     def choose_action(self, state):
102         if random.random() < self.epsilon:
103             # Explore: Randomly choose an action
104             return random.choice(self.valid_actions)
105         else:
106             # Exploit: Choose the best action based on current utility values
107             return np.argmax(
108                 [
109                     self.utility[state[0] + dx, state[1] + dy]
110                     for dx, dy in self.maze.actions.values()
111                 ]
112             )
113
114     def update_utility_value(self, current_state, reward, new_state):
115         current_value = self.utility[current_state[0], current_state[1]]
116         new_value = self.utility[new_state[0], new_state[1]]
117         td_target = reward + self.gamma * new_value
118         self.utility[current_state[0], current_state[1]] = current_value + self.alpha * (
    td_target - current_value)
119
120     # for TD learning
121     def value_function_from_utility(self):
122         utility_values = self.utility[0:11, 0:11].copy()
123         # update value function of the target
124         utility_values[np.where(self.maze.maze == 3)] = 3000
125         return utility_values
126
127
128     def run_episodes(self):
129         for episode in tqdm(range(self.episodes)):
130             self.maze.reset()
131             wander_count = 0
132             while True:
133                 current_state = self.maze.current_pos
134
135                 action = self.choose_action(current_state)
136                 new_state, reward = self.maze.step(action)
137                 self.update_utility_value(current_state, reward, new_state)
138                 current_state = new_state
139                 # print(current_state)
140                 if self.maze.maze[current_state] == 3 or self.maze.maze[current_state] ==
     2:
141                     break
142                 wander_count += 1
143                 # to limit too much wandering
144                 if wander_count > self.wander_limit:
145                     break
146             utility_vals = self.value_function_from_utility()
147
148             self.convergence_data.append(np.sum(np.abs(np.subtract(utility_vals, self.
    previous_values))))
149             self.previous_values = utility_vals.copy()
150
```

```
            if (episode+1) in self.episodes_to_plot:
                plot_value_function(utility_vals, self.maze.maze, self.alpha, self.gamma, self.epsilon, episode+1)
                plot_policy(utility_vals, self.maze.maze, self.alpha, self.gamma, self.epsilon, episode+1)
        # plot the convergence data
        plt.figure()
        plt.plot(self.convergence_data[1:])
        plt.xlabel("episodes")
        plt.ylabel("sum of absolute differences")
        plt.title("Convergence over episodes")
        plt.grid()
        # plt.show()
        plt.savefig("hw3/output/convergence_TD_alpha_{}_gamma_{}_epislon_{}.png".format(self.alpha, self.gamma, self.epsilon))
        plt.close()
        # smoothed version of it
        plt.figure()
        plt.plot(np.convolve(self.convergence_data[1:], np.ones(100)/100, mode='valid'))
        plt.xlabel("episodes")
        plt.ylabel("sum of absolute differences")
        plt.title("Convergence over episodes")
        plt.grid()
        # plt.show()
        plt.savefig("hw3/output/convergence_TD_smoothed_alpha_{}_gamma_{}_epislon_{}.png".format(self.alpha, self.gamma, self.epsilon))
        plt.close()
        return self.utility


# Create an instance of the Maze with TD(0) and run multiple episodes
# maze = MazeEnvironment()
# maze_td0 = MazeTD0(maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000)
# final_values = maze_td0.run_episodes()
# plot_policy(maze_td0.value_function_from_utility(), maze.maze, maze_td0.alpha, maze_td0.gamma, maze_td0.epsilon, maze_td0.episodes)
############ Q Learning ###################

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.maze.shape[0], maze.maze.shape[1], 4))  # Initialize Q-table
        self.valid_actions = list(maze.actions.keys())

        self.convergence_data = []
        self.previous_values = np.zeros((11, 11))
        self.episodes_to_plot = [1, 50, 100, 1000, 5000, 10000]

        self.wander_limit = 10000
    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:  # Explore
            return random.choice(self.valid_actions)
        else:  # Exploit
```

```
            state_q_values = self.q_table[state[0], state[1], :]
            return np.argmax(state_q_values)

    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 = np.max(self.q_table[new_state[0], new_state[1], :])
        new_q = current_q + self.alpha * (reward + self.gamma * max_future_q - current_q)
        self.q_table[current_state[0], current_state[1], action] = new_q


    # for Q-learning output
    def value_function_from_q_table(self):
        # convert q_table to value function
        value_function = np.max(self.q_table.copy(), axis=2)
        # make the invalid moves -1000 in value function
        value_function[np.where(self.maze.maze == 1)] = -1000
        value_function[np.where(self.maze.maze == 3)] =  3000
        return value_function

    def run_episodes(self):
        for episode in tqdm(range(self.episodes)):
            state = self.maze.reset()  # Assuming reset initializes and returns the start
     state

            wander_count = 0

            while True:
                action = self.choose_action(state)
                new_state, reward = self.maze.step(action)  # Assuming step executes
    action and returns new_state, reward, and done

                self.update_q_table(action, state, reward, new_state)
                state = new_state

                if self.maze.maze[state] == 3 or self.maze.maze[state] == 2:
                        break
                wander_count += 1
                # to limit too much wandering
                if wander_count > self.wander_limit:
                    break


            utility_vals = self.value_function_from_q_table()

            self.convergence_data.append(np.sum(np.abs(np.subtract(utility_vals, self.
    previous_values))))
            self.previous_values = utility_vals.copy()

            if (episode+1) in self.episodes_to_plot:
                plot_value_function(utility_vals, self.maze.maze, self.alpha, self.gamma,
     self.epsilon, episode+1)
                plot_policy(utility_vals, self.maze.maze, self.alpha, self.gamma, self.
    epsilon, episode+1)
        # plot the convergence data
        plt.figure()
        plt.plot(self.convergence_data[1:])
        plt.xlabel("episodes")
        plt.ylabel("sum of absolute differences")
        plt.title("Convergence over episodes")
```

```
        plt.grid()
        # plt.show()
        plt.savefig("hw3/output/convergence_Q_alpha_{}_gamma_{}_epislon_{}.png".format(self.alpha, self.gamma, self.epsilon))
        plt.close()
        # smoothed version of it
        plt.figure()
        plt.plot(np.convolve(self.convergence_data[1:], np.ones(100)/100, mode='valid'))
        plt.xlabel("episodes")
        plt.ylabel("sum of absolute differences")
        plt.title("Convergence over episodes")
        plt.grid()
        # plt.show()
        plt.savefig("hw3/output/convergence_Q_smoothed_alpha_{}_gamma_{}_epislon_{}.png".format(self.alpha, self.gamma, self.epsilon))
        plt.close()
        return self.q_table


# maze = MazeEnvironment()# Use 0 = free space, 1 = obstacle, 2 = goal
# maze_q_learning = MazeQLearning(maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000)
# q_table = maze_q_learning.run_episodes()
# plot_policy(maze_q_learning.value_function_from_q_table(), maze.maze, maze_q_learning.alpha, maze_q_learning.gamma, maze_q_learning.epsilon, maze_q_learning.episodes)

# plot_value_function(value_function, maze.maze)
# plot_policy(value_function, maze.maze)



parameters_alpha_sweep = [{"alpha": 0.001, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.001, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.5, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 1, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000}]


parameters_gamma_sweep = [{"alpha": 0.1, "gamma": 0.1, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.25, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.50, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.75, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 95, "epsilon": 0.2, "episodes": 10000}]


parameters_epsilon_sweep = [{"alpha": 0.1, "gamma": 0.95, "epsilon": 0.0, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.95, "epsilon": 0.2, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.95, "epsilon": 0.5, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.95, "epsilon": 0.8, "episodes": 10000},
                {"alpha": 0.1, "gamma": 0.95, "epsilon": 1.0, "episodes": 10000}]

# # alpha sweep TD learning
# for parameters in parameters_alpha_sweep:
#     maze_td0 = MazeTD0(maze, alpha=parameters["alpha"], gamma=parameters["gamma"], epsilon=parameters["epsilon"], episodes=parameters["episodes"])
#     final_values = maze_td0.run_episodes()
```

```
309
310 # # alpha sweep Q learning
311 # for parameters in parameters_alpha_sweep:
312 #     maze_q_learning = MazeQLearning(maze, alpha=parameters["alpha"], gamma=parameters["
    gamma"], epsilon=parameters["epsilon"], episodes=parameters["episodes"])
313 #     q_table = maze_q_learning.run_episodes()
314
315 # # gamma sweep TD learning
316 # for parameters in parameters_gamma_sweep:
317 #     maze_td0 = MazeTD0(maze, alpha=parameters["alpha"], gamma=parameters["gamma"],
    epsilon=parameters["epsilon"], episodes=parameters["episodes"])
318 #     final_values = maze_td0.run_episodes()
319
320 # # gamma sweep Q learning
321 # for parameters in parameters_gamma_sweep:
322 #     maze_q_learning = MazeQLearning(maze, alpha=parameters["alpha"], gamma=parameters["
    gamma"], epsilon=parameters["epsilon"], episodes=parameters["episodes"])
323 #     q_table = maze_q_learning.run_episodes()
324 maze = MazeEnvironment()
325
326 # epsilon sweep TD learning
327 # for parameters in parameters_epsilon_sweep:
328 #     maze_td0 = MazeTD0(maze, alpha=parameters["alpha"], gamma=parameters["gamma"],
    epsilon=parameters["epsilon"], episodes=parameters["episodes"])
329 #     final_values = maze_td0.run_episodes()
330
331 # epsilon sweep Q learning
332 for parameters in parameters_epsilon_sweep:
333     maze_q_learning = MazeQLearning(maze, alpha=parameters["alpha"], gamma=parameters["
    gamma"], epsilon=parameters["epsilon"], episodes=parameters["episodes"])
334     q_table = maze_q_learning.run_episodes()
```