EE449 HW2*

*Note: This homework exaggerated and used LaTeX, I don't know why am I doing this.

Ömer Takkin
No.: 2516987
Electric and Electronic Engineer (EEE)
Middle East Technical University (METU)
Ankara, Turkey
omer.takkin@gmail.com

***Abstract*—This project explores fundamental reinforcement learning techniques by applying Temporal Difference (TD(0)) learning and Deep Q-Networks (DQN). For this, we implement to two tasks: a maze navigation problem and the LunarLander-v3 environment. In the first part, we model a stochastic maze and implement TD(0) algorithm to derive optimal navigation policies. We will analyze performance through utility heatmaps and convergence plots. In the second part, we will build a DQN agent in PyTorch to learn a landing policy for a simulated lunar lander. We will experiment with various hyperparameters and network architectures to evaluate their impact on learning stability and efficiency.**

***Index Terms*—METU, EEE, EE449, HW1, Reinforcement Learning, Help**

I. Temporal Difference (TD) Learning

A. Preliminary

In this part of the experiment, a maze is structured with a probabilistic move agent. The agent tries to reach the goal while avoiding traps.

B. Experiments

1) Value Iteration For Different Alpha Values: In this section, we will observe value iteration for different alpha values

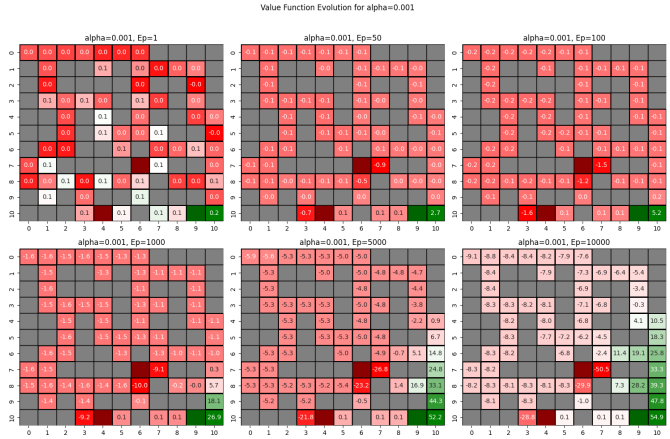

Fig. 1. Value Evaluation for α = 0.001

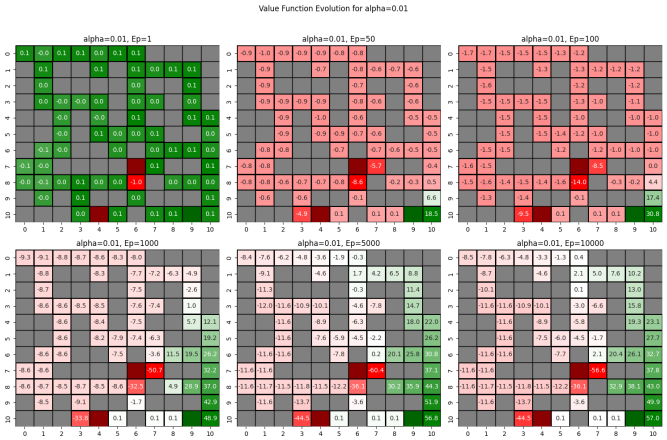

Fig. 2. Value Evaluation for α = 0.01

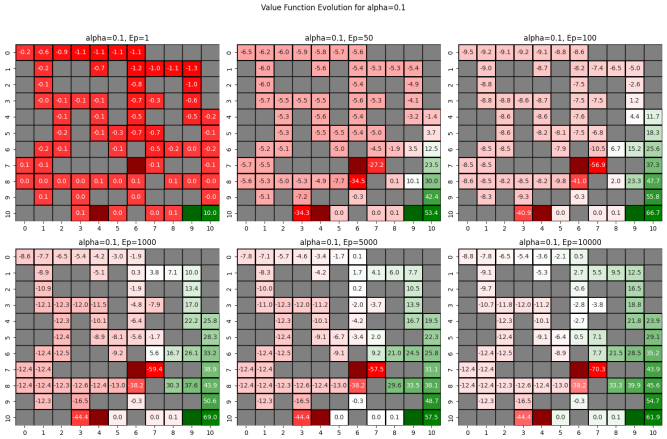

Fig. 3. Value Evaluation for α = 0.1

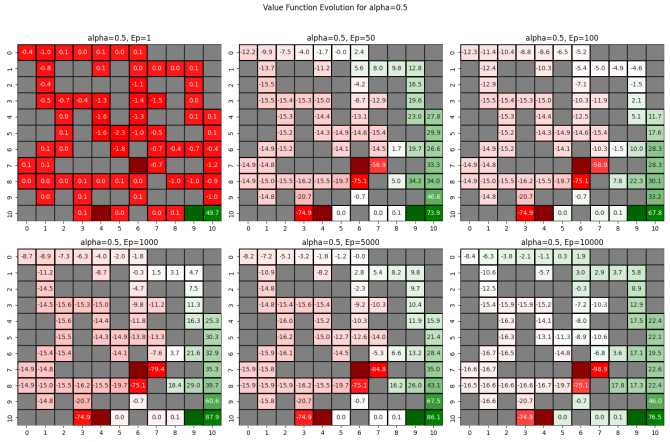

Fig. 4. Value Evaluation for $\alpha = 0.5$

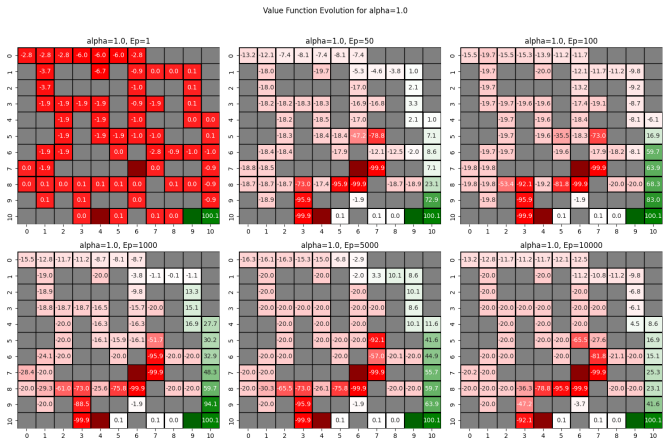

Fig. 5. Value Evaluation for $\alpha = 1.0$

2) Policy Iteration For Different Alpha Values: In this section, we will observe policy iteration for different alpha values

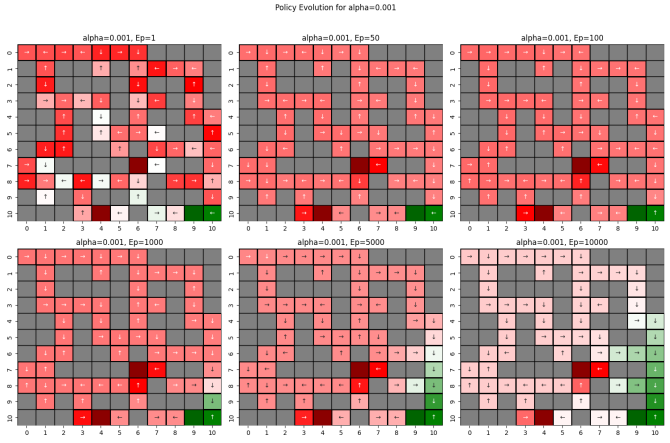

Fig. 6. Policy Evaluation for $\alpha = 0.001$

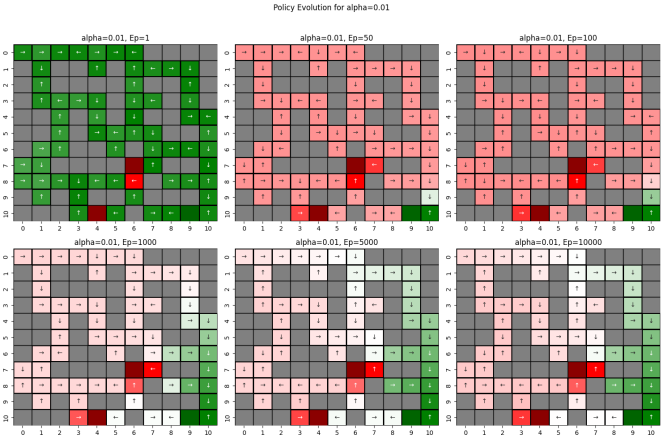

Fig. 7. Policy Evaluation for $\alpha = 0.01$

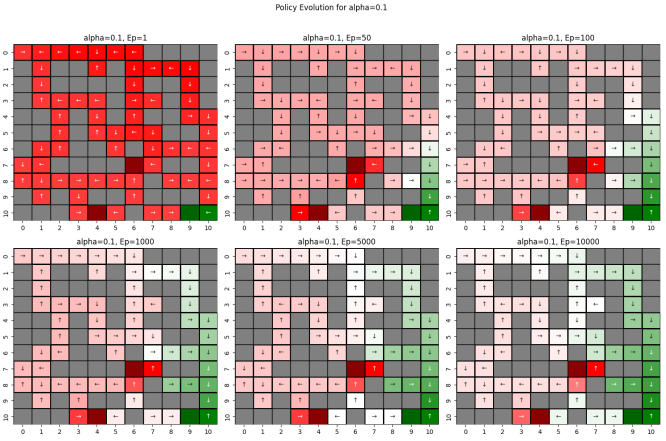

Fig. 8. Policy Evaluation for $\alpha = 0.1$

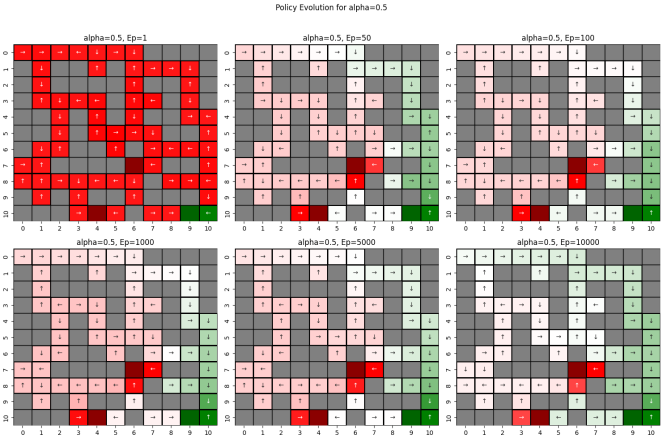

Fig. 9. Policy Evaluation for $\alpha = 0.5$

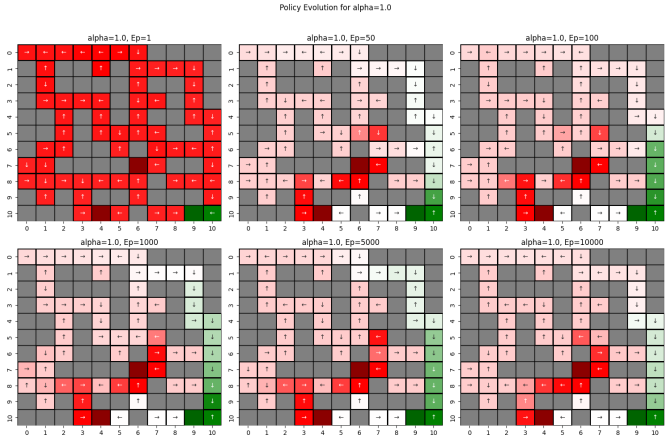

Fig. 10. Policy Evaluation for α = 1.0

3) Convergence For Different Alpha Values: In this section, we will observe convergence for different alpha values

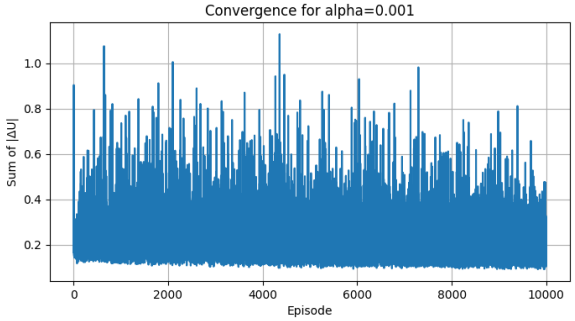

Fig. 11. Convergence for α = 0.001

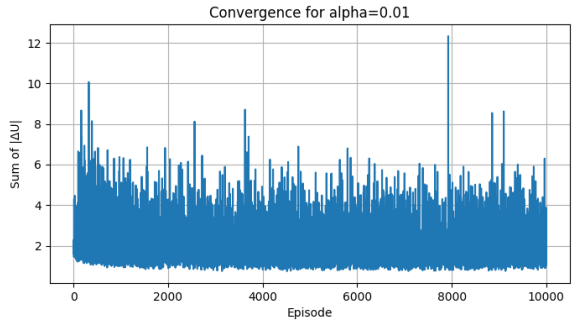

Fig. 12. Convergence for α = 0.01

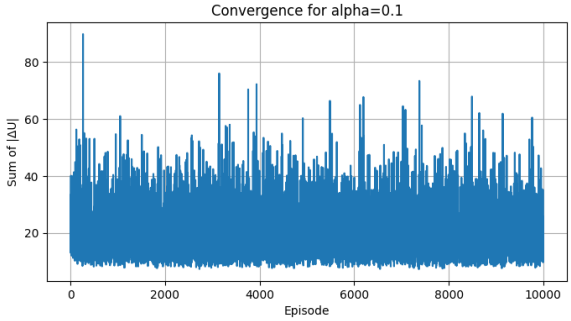

Fig. 13. Convergence for α = 0.1

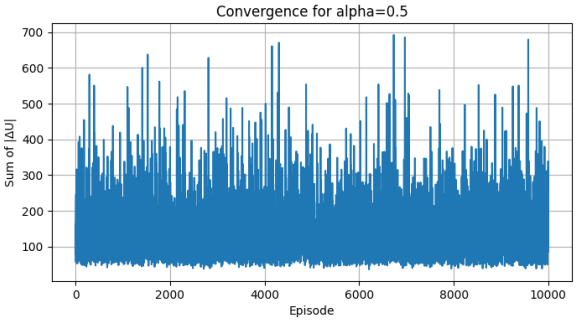

Fig. 14. Convergence for α = 0.5

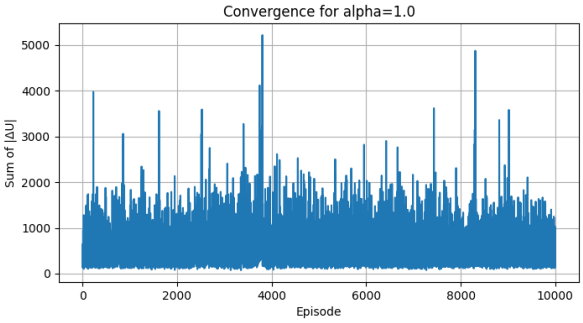

Fig. 15. Convergence for α = 1.0

4) Value Iteration For Different Gamma Values: In this section, we will observe value iteration for different gamma values

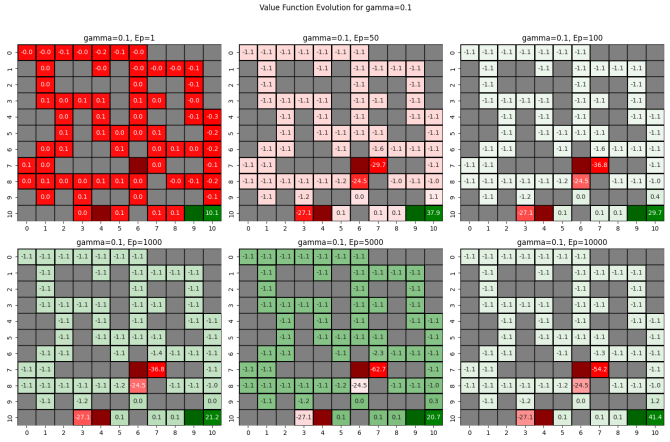

Fig. 16. Value Evaluation for $\gamma = 0.1$

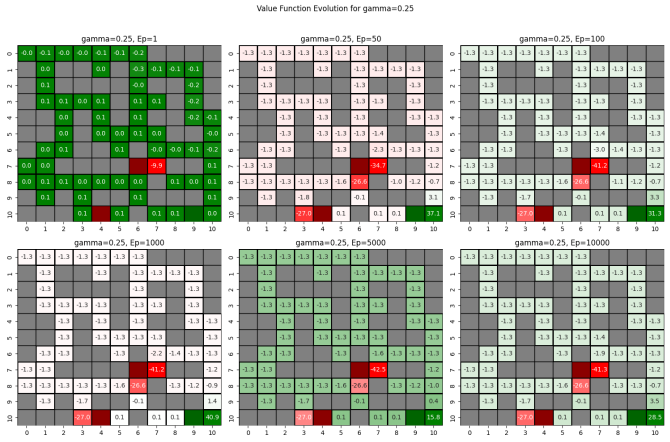

Fig. 17. Value Evaluation for $\gamma = 0.25$

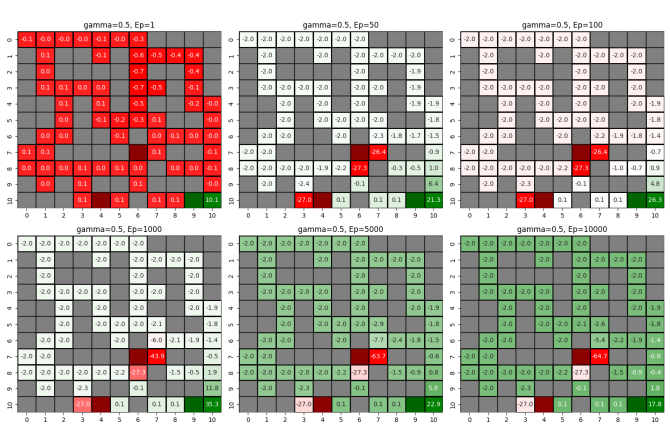

Fig. 18. Value Evaluation for $\gamma = 0.50$

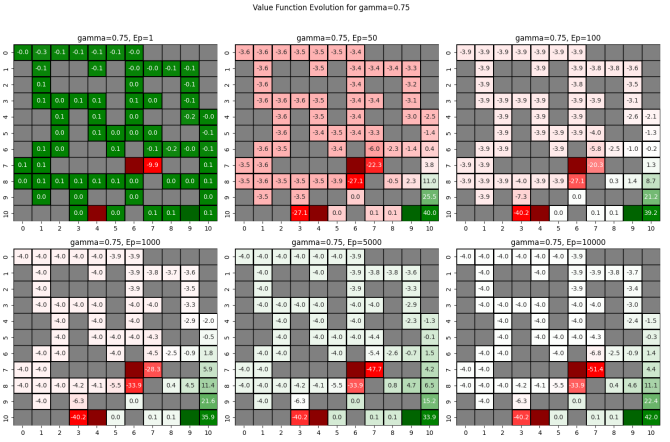

Fig. 19. Value Evaluation for $\gamma = 0.75$

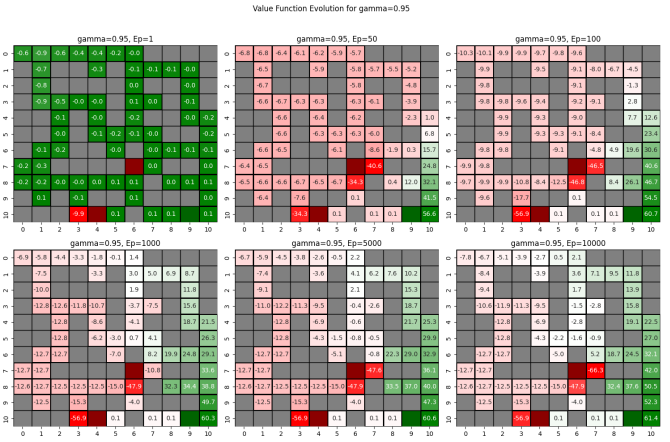

Fig. 20. Value Evaluation for $\gamma = 0.95$

5) Policy Iteration For Different Gamma Values: In this section, we will observe policy iteration for different gamma values

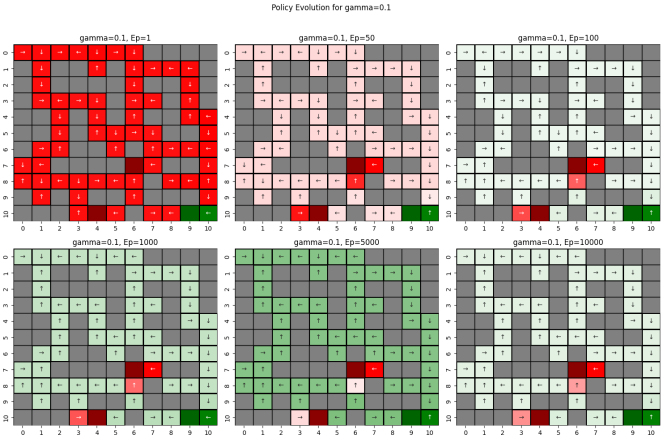

Fig. 21. Policy Evaluation for $\gamma = 0.1$

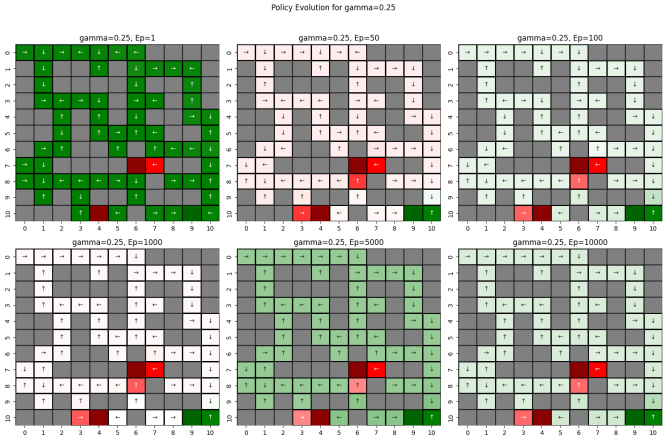

Fig. 22. Policy Evaluation for $\gamma = 0.25$

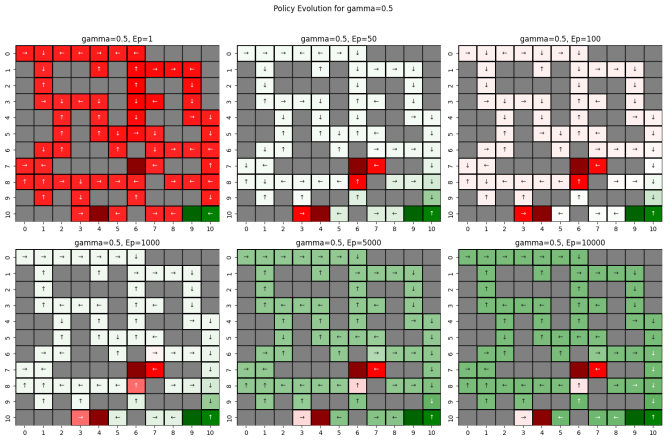

Fig. 23. Policy Evaluation for $\gamma = 0.50$

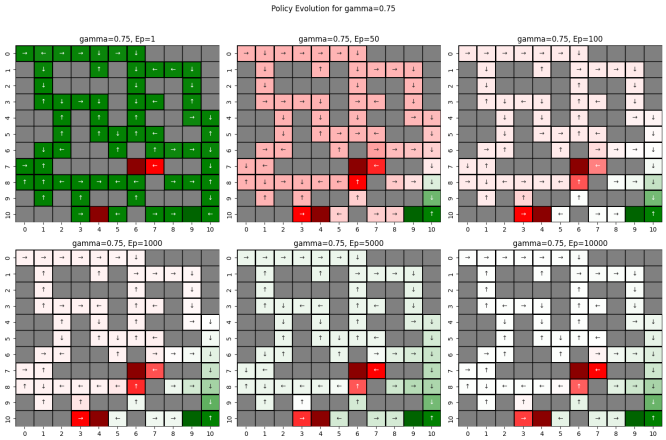

Fig. 24. Policy Evaluation for $\gamma = 0.75$

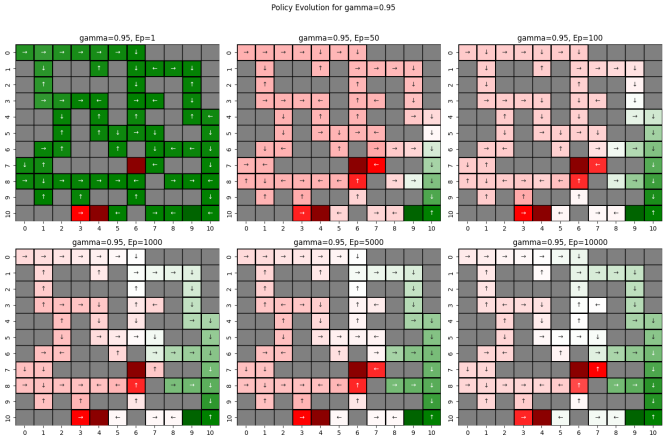

Fig. 25. Policy Evaluation for $\gamma = 0.95$

6) Convergence For Different Gamma Values: In this section, we will observe convergence for different gamma values

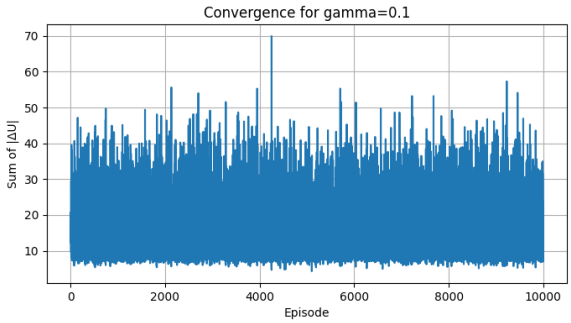

Fig. 26. Convergence for $\gamma = 0.1$

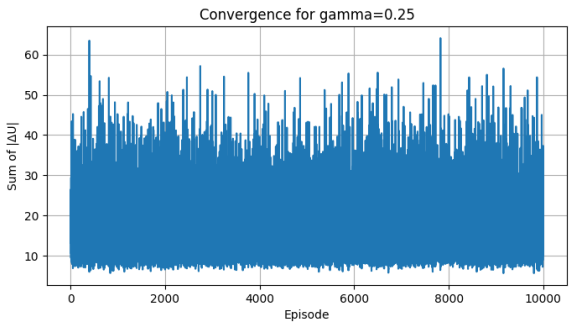

Fig. 27. Convergence for $\gamma = 0.25$

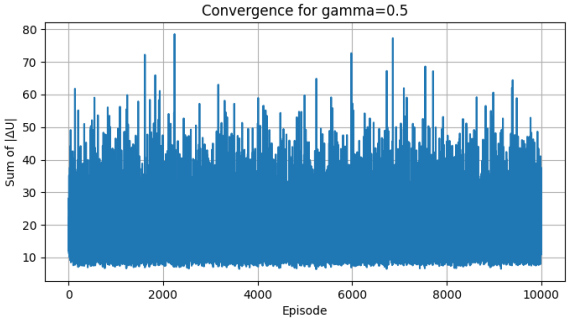

Fig. 28. Convergence for $\gamma = 0.50$

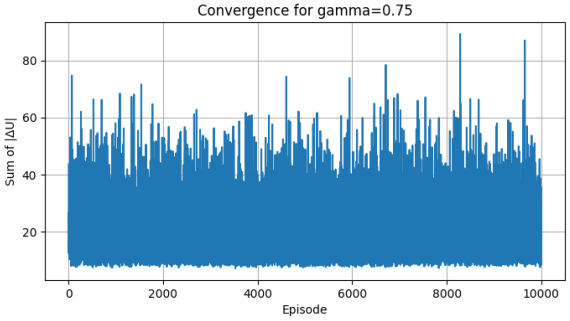

Fig. 29. Convergence for $\gamma = 0.75$

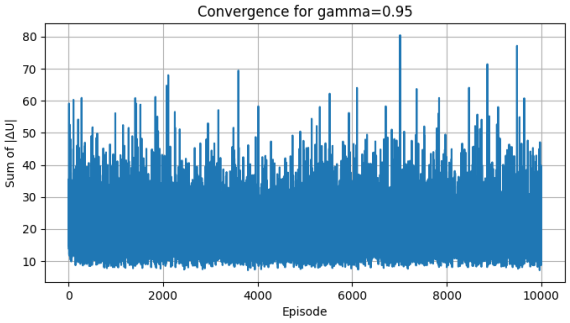

Fig. 30. Convergence for $\gamma = 0.95$

7) Value Iteration For Different Epsilon Values: In this section, we will observe value iteration for different epsilon values

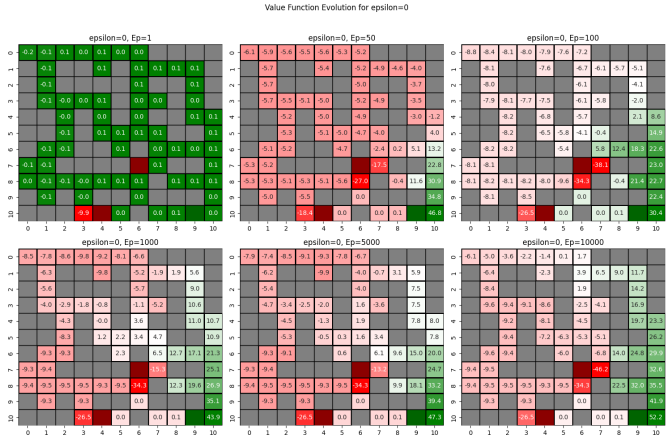

Fig. 31. Value Evaluation for $\epsilon = 0.0$

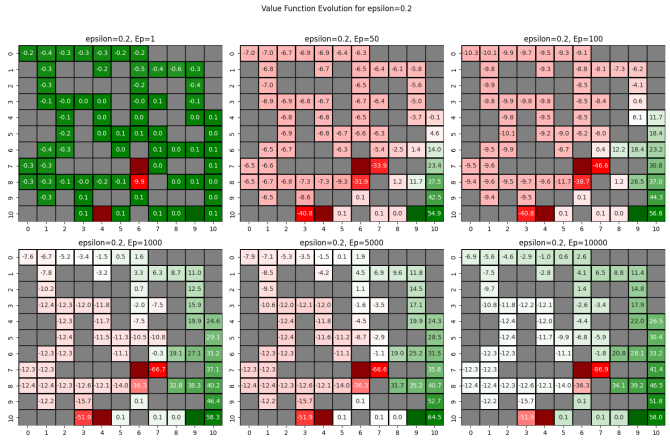

Fig. 32. Value Evaluation for $\epsilon = 0.2$

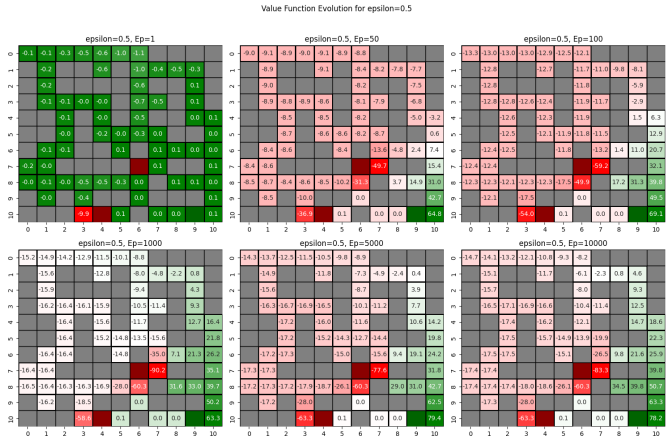

Fig. 33. Value Evaluation for $\epsilon = 0.50$

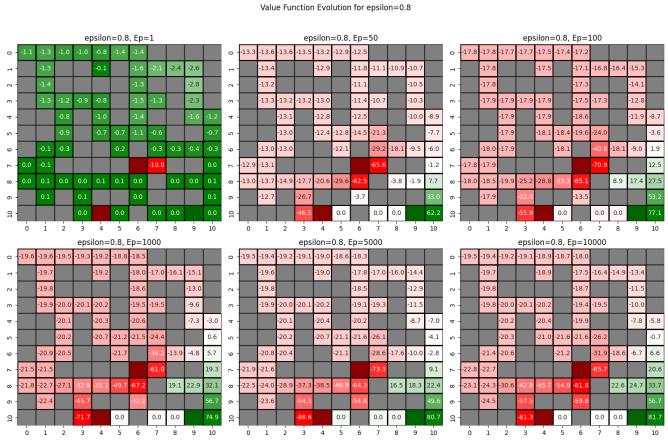

Fig. 34. Value Evaluation for $\epsilon = 0.80$

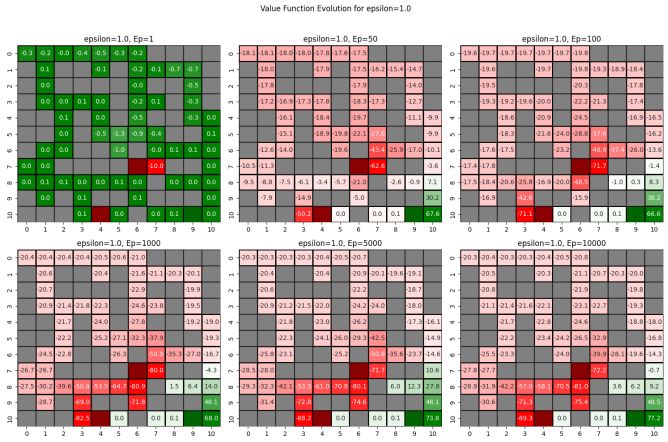

Fig. 35. Value Evaluation for $\epsilon = 1.0$

8) Policy Iteration For Different Epsilon Values: In this section, we will observe policy iteration for different epsilon values

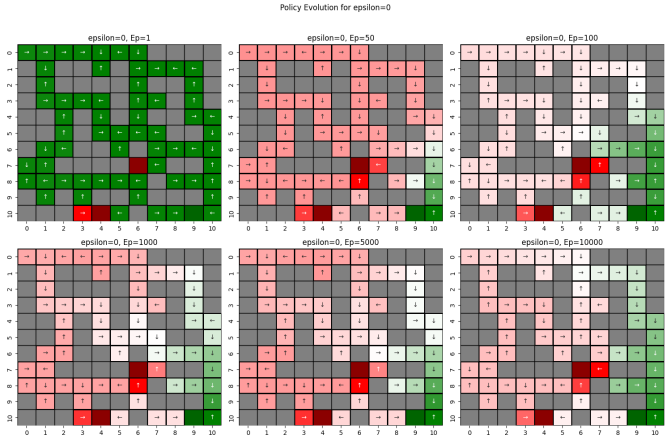

Fig. 36. Policy Evaluation for $\epsilon = 0.0$

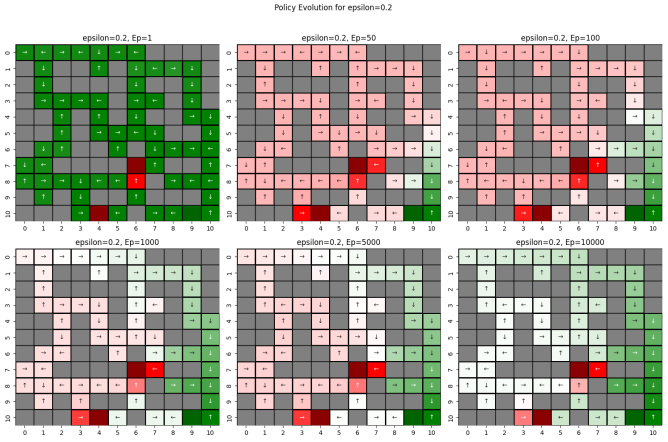

Fig. 37. Policy Evaluation for $\epsilon = 0.20$

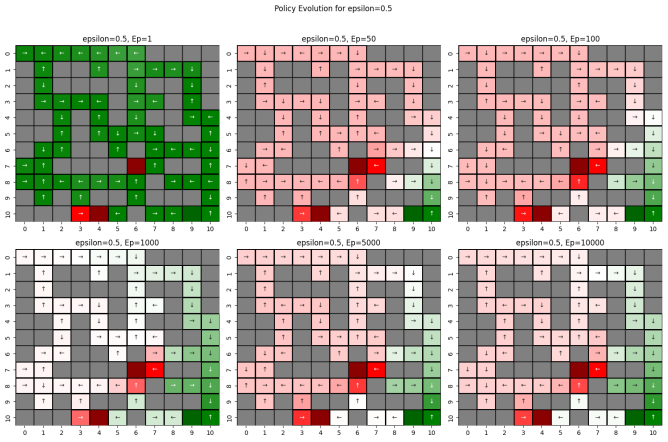

Fig. 38. Policy Evaluation for $\epsilon = 0.50$

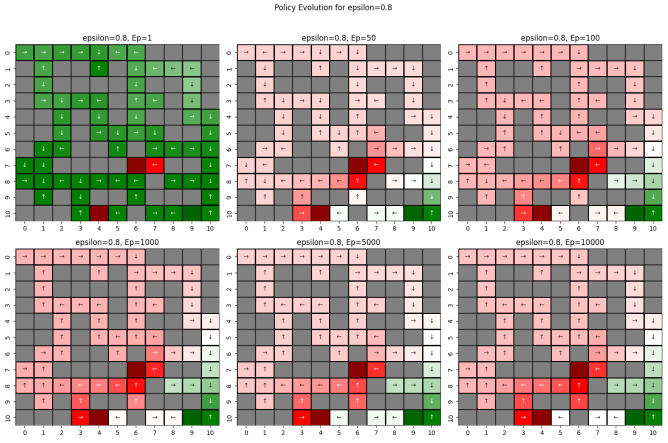

Fig. 39. Policy Evaluation for $\epsilon = 0.80$

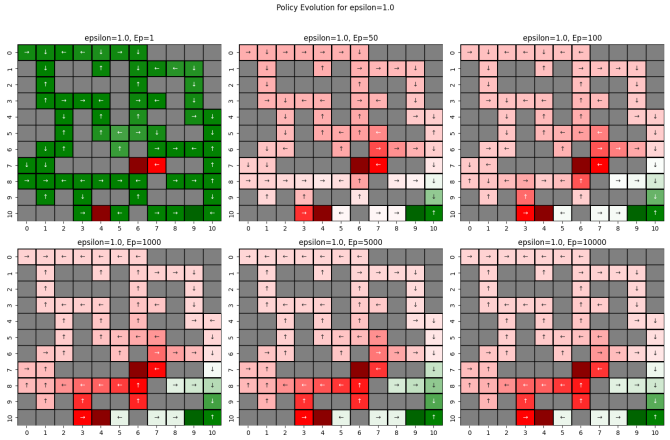

Fig. 40. Policy Evaluation for ϵ = 1.0

9) Convergence For Different Epsilon Values: In this section, we will observe convergence for different epsilon values

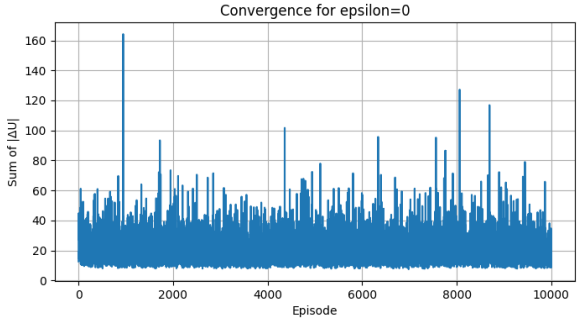

Fig. 41. Convergence for ϵ = 0.0

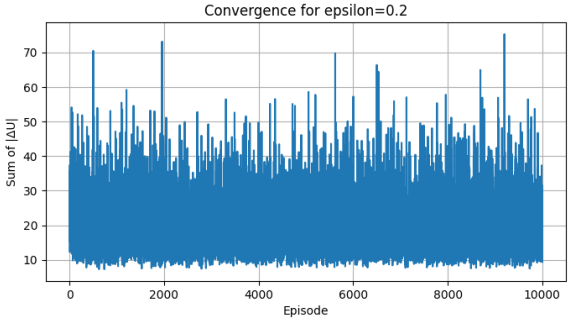

Fig. 42. Convergence for ϵ = 0.20

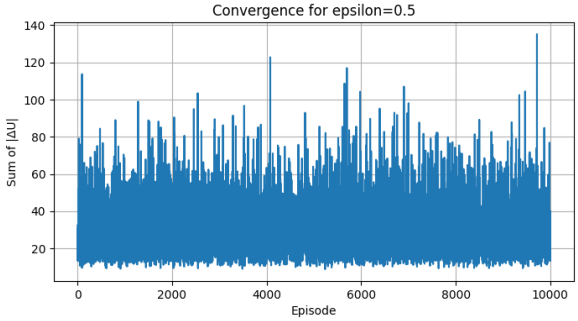

Fig. 43. Convergence for ϵ = 0.50

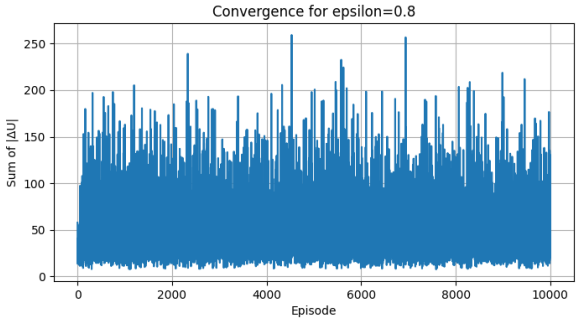

Fig. 44. Convergence for ϵ = 0.80

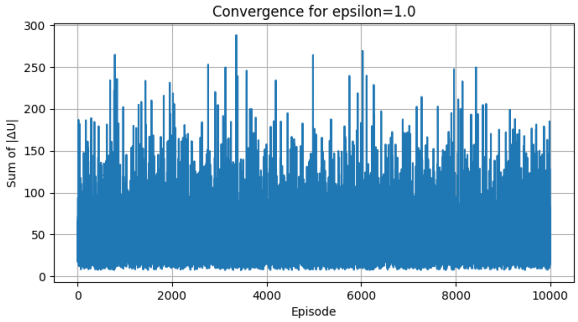

Fig. 45. Convergence for ϵ = 1.0

C. Discussion

The transition probabilities affect stochasticity in the agent's movements. It simulates real-world uncertainty. The reward function penalizes each move slightly so that the agent is encouraged for efficiency. Also, we heavily penalize traps to discourage the risk of getting trapt and highly reward reaching the goal. In that way we motivate the agent to find a safe and optimal path while learning to avoid traps and redundant steps.

Early in training, utility values are random. However, over time, these values. It begins to show the agent's learned expectations, higher values appear near the goal and lower near traps. Heatmaps show how the agent's understanding of the environment sharpens with experience, especially after 1000+ episodes.

For our cases, the utility values generally converged after around 5000 to 7000 episodes. This value depends on the parameter configuration. Convergence was calculated by observing minimal changes in utility values and a flattening in the convergence plots.

A very low learning rate, which is α is 0.001 slowed convergence. On the contrary, a very high one, which is α is 1.0 caused instability. Similarly, low discount factors (prioritized immediate rewards and discouraged planning, while high values encouraged long-term strategies but required more iterations to stabilize.

One of the challenges encountered is handling stochastic transitions and ensuring valid moves, especially near walls. I had to check in every move to ensure my move is valid.

Low ϵ causes to premature exploitation and poor exploration. Therefore, it causes suboptimal policies. On the other hand, high ϵ (like 1.0) causes exploration but delayed convergence. A balanced ϵ (e.g., 0.2) provided a good mix, enabling the agent to explore early and exploit later.

There are several ways to make the problem harder. There are Making the maze larger, increasing trap density, or introducing dynamic elements (e.g., moving obstacles) could make the problem more challenging. There are several ways to make the problem easier. There are simplifying the layout or increasing reward granularity could speed up learning.

Incorporating eligibility traces (TD(λ)) or switching to model-based methods could improve learning. Combining TD with supervised imitation learning or using function approximation (like neural networks) would also enhance performance and generalization in larger or more complex mazes.

II. Deep Q-Learning

A. Experiments

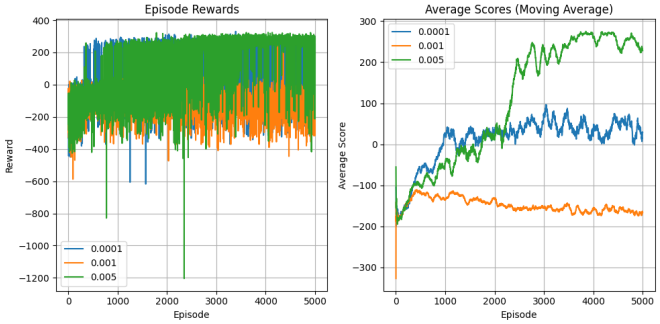

Fig. 46. Learning curves for learning rate

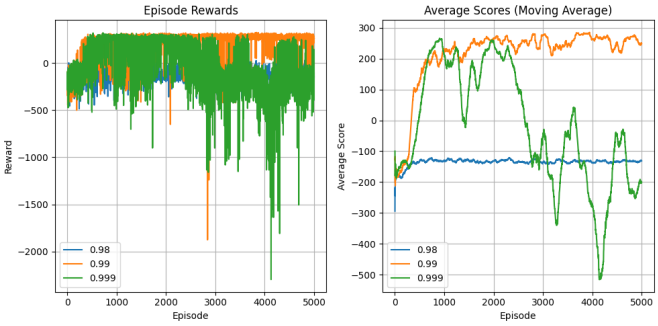

Fig. 47. Learning curves for discount factor

Fig. 48. Learning curves for epsilon decay

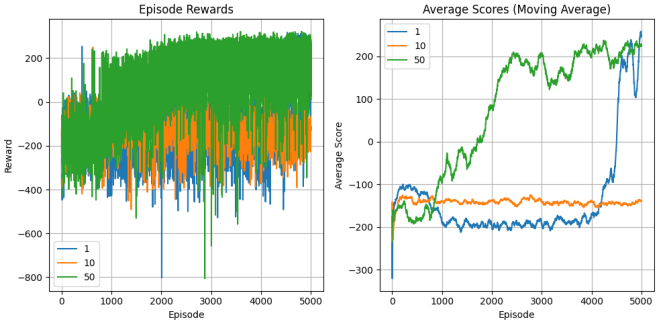

Fig. 49. Learning curves for target update frequency

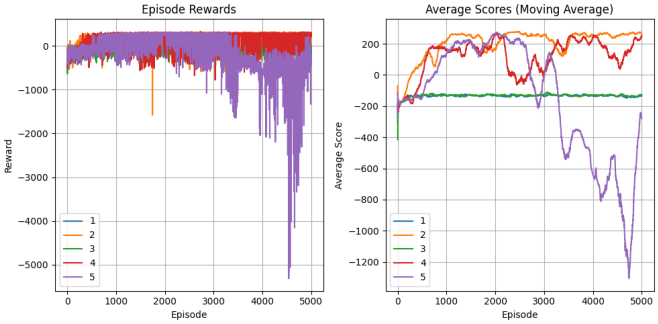

Fig. 50. Learning curves for net no

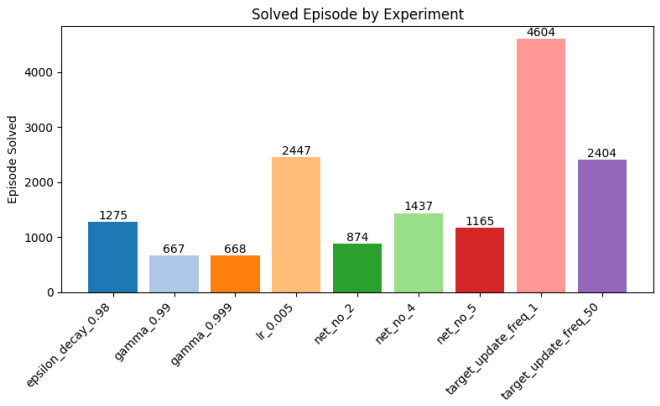

Fig. 51. Solved episodes for experiments

B. Discussion

The choice of learning rate (α) affects the training dynamics of the DQN agent. A lower learning rate generally leads to stable but slow convergence. This may be very beneficial in preventing divergence, but it may also prevent the agent from adapting quickly to new environmental patterns. On the other hand, a higher learning rate can lead to faster initial learning. However, most of the time it can introduce instability to the model due to overshooting optimal values. Among the tested values, 1e-3 offered the most balanced outcome. It achieved a better final performance. Moreover, it maintained a very good learning speed.

The discount factor (γ) affects the agent's preference for long-term versus short-term rewards. A smaller lower value prioritizes immediate rewards, which causes faster early improvements but may be bad for capturing the long-term strategies. On the other hand, a high value prioritizes future rewards, which causes better long-term planning. However, it may delay convergence due to the slower propagation of value updates. The best value for γ is 0.99. It allows for focusing on long-term goals without overriding or delaying learning progress.

The exploration decay rate (ϵ-decay) is very important for balancing exploration and exploitation. For small values, randomness reduces rapidly, which can cause the agent to settle prematurely on suboptimal policies. On the other hand, a slower decay causes the agent to sample a wider range of state-action pairs, and it may discover more optimal strategies. However, it may also slow down convergence. The experiments showed that slower decay rates tended to yield higher final performance due to more thorough policy refinement, although they required more episodes to reach the same level of stability.

The frequency of target network updates affects the stability and reliability of Q-value estimates. Frequent updates tend to increase instability because of shifting the target values rapidly. It can lead to oscillating behavior. On the other hand, infrequent updates cause the target network to lag behind the policy network. It means it results in outdated targets that slow learning progress. The default frequency of f = 10 is the most effective. It is stable in the learning process by periodically refreshing the target values without causing instability.

The model of the Q-network plays a very important role in the agent's ability to approximate the action-value function effectively. Networks with fewer parameters tend to converge quickly, but capacity can be shallow. It may result in suboptimal policies. Deeper or wider architectures offer greater function approximation capability and can capture more complex patterns. However, they may overfit. Among the tested models, Net 3 delivered the best performance by balancing expressiveness and stability. It converged efficiently and was able to reach the solution criterion faster than both smaller and larger networks in most cases.

The learning rate had the most noticeable impact on the agents' overall performance out of all the hyperparameters that were investigated. Compared to both lower and higher values, a carefully selected learning rate like 1e-3 allowed for faster and more stable convergence. Furthermore, the neural network architecture significantly affected the agent's ability to generalize from experiences. Additionally,y there seemed to be interplay among the hyperparameters. For example, deeper networks were more susceptible to rapid epsilon decay, while a higher learning rate necessitated more robust network architectures to preserve stability. Despite the independent testing of the parameters, these dependencies imply that tuning hyperparameters together rather than separately may be able to yield the best results.

Important information about the training dynamics under various configurations was revealed by the learning curves. Sharp increases and decreases in episode rewards, which signify instability in the learning process, are characteristic of volatile curves that are usually produced by configurations with high learning rates or very frequent target network updates. Smoother learning curves, on the other hand, were typically linked to a gradual epsilon decay schedule, steady update frequencies, and moderate learning rates. The more steady improvement in policy quality over time was represented by these smoother curves. The significance of choosing hyperparameters that strike a balance between learning speed and stability to guarantee stable performance and dependable convergence is highlighted by the variation in learning trajectories.

APPENDIX

A. Part 1

1) main.py:

```python
import sys
import os
sys.path.append(os.path.abspath('_given'))

import numpy as np
import matplotlib.pyplot as plt
from utils import plot_value_function, plot_policy, plot_convergence  # type: ignore

import gc

# Ensure results directory exists
os.makedirs('part1/results', exist_ok=True)

class MazeEnvironment:
    def __init__(self):
        # Define the maze layout, rewards, and action space
        self.start_pos = (0, 0)
        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)}
        # Transition probabilities
        self.prob_intended = 0.75
        self.prob_opposite = 0.05
        self.prob_perpendicular = 0.10

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

    def step(self, action):
        r = np.random.rand()
        if r < self.prob_intended:
            chosen = action
        elif r < self.prob_intended + self.prob_opposite:
            chosen = {0:1, 1:0, 2:3, 3:2}[action]
        elif r < self.prob_intended + self.prob_opposite + self.prob_perpendicular:
            chosen = {0:2, 1:3, 2:1, 3:0}[action]
        else:
            chosen = {0:3, 1:2, 2:0, 3:1}[action]

        move = self.actions[chosen]
        next_state = (self.current_pos[0] + move[0], self.current_pos[1] + move[1])
        if (0 <= next_state[0] < self.maze.shape[0] and 0 <= next_state[1] < self.maze.shape[1]
                and self.maze[next_state] != 1):
            self.current_pos = next_state

        cell_value = self.maze[self.current_pos]
        if cell_value == 2:
            return self.current_pos, self.trap_penalty, True
        elif cell_value == 3:
            return self.current_pos, self.goal_reward, True
        else:
            return self.current_pos, self.state_penalty, False

class MazeTD0(MazeEnvironment):
    def __init__(self, maze, alpha=0.001, gamma=0.95, epsilon=0.2, episodes=10000):
        super().__init__()
        self.maze = maze
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.episodes = episodes
        # Initialize utilities
        self.utility = np.random.rand(*self.maze.shape) * 0.1
        self.utility[maze == 1] = np.nan

    def choose_action(self, state):
        valid = []
```

```
        for a, move in self.actions.items():
            ns = (state[0] + move[0], state[1] + move[1])
            if 0 <= ns[0] < self.maze.shape[0] and 0 <= ns[1] < self.maze.shape[1] and self.maze[ns] != 1:
                valid.append(a)
        if not valid:
            return None
        if np.random.rand() < self.epsilon:
            return np.random.choice(valid)
        best, best_val = None, -np.inf
        for a in valid:
            move = self.actions[a]
            ns = (state[0] + move[0], state[1] + move[1])
            val = self.utility[ns]
            if val > best_val:
                best, best_val = a, val
        return best

    def update_utility_value(self, s, r, ns):
        curr = self.utility[s]
        new = self.utility[ns]
        self.utility[s] = curr + self.alpha * (r + self.gamma * new - curr)

    def run_episodes(self, snapshot_episodes=None, max_steps=5000000):
        if snapshot_episodes is None:
            snapshot_episodes = []
        snapshots = {}
        convergence = np.zeros(self.episodes)

        reached_step = 0 # checking if we hit max_steps

        for ep in range(1, self.episodes + 1):
            state = self.reset()
            diff = 0
            steps = 0
            done = False

            while not done and steps < max_steps:
                act = self.choose_action(state)
                if act is None:
                    break
                old = self.utility[state]
                next_state, reward, done = self.step(act)
                self.update_utility_value(state, reward, next_state)
                diff += abs(self.utility[state] - old)
                state = next_state
                steps += 1

            reached_step = max(reached_step , steps)
            convergence[ep - 1] = diff
            if ep in snapshot_episodes:
                snapshots[ep] = self.utility.copy()
        print(f"Reached maximum step is: {reached_step}")
        return snapshots, convergence



def save_results(param, val, snapshots, convergence):
    base = f"part1/results/{param}_{val}"
    os.makedirs(base, exist_ok=True)

    for ep, u in snapshots.items():
        np.save(f"{base}/utility_ep{ep}.npy", u)

    np.save(f"{base}/convergence.npy", np.array(convergence))



if __name__ == '__main__':
    # Define maze
    maze = np.array([
        [0,0,0,0,0,0,0,1,1,1,1],
        [1,0,1,1,0,1,0,0,0,0,1],
        [1,0,1,1,1,1,0,1,1,0,1],
        [1,0,0,0,0,1,0,0,1,0,1],
```

```
        [1,1,0,1,0,1,0,1,1,0,0],
        [1,1,0,1,0,0,0,0,1,1,0],
        [1,0,0,1,1,0,1,0,0,0,0],
        [0,0,1,1,1,1,2,0,1,1,0],
        [0,0,0,0,0,0,0,1,0,0,0],
        [1,0,1,0,1,1,0,1,1,1,0],
        [1,1,1,0,2,0,1,0,0,3,0]
    ])

    # Hyperparameter experiments
    experiments = {
        'alpha': {'values': [0.001, 0.01, 0.1, 0.5, 1.0], 'default': 0.1},
        'gamma': {'values': [0.10, 0.25, 0.50, 0.75, 0.95], 'default': 0.95},
        'epsilon': {'values': [0, 0.2, 0.5, 0.8, 1.0], 'default': 0.2}
    }
    episodes = 10000
    snapshots_to_keep = [1, 50, 100, 1000, 5000, 10000]

    for param, config in experiments.items():
        for val in config['values']:
            # Set parameters
            alpha = experiments['alpha']['default'] if param != 'alpha' else val
            gamma = experiments['gamma']['default'] if param != 'gamma' else val
            epsilon = experiments['epsilon']['default'] if param != 'epsilon' else val

            print(f"Selected values are: alpha = {alpha} | gamma = {gamma} | epsilon = {epsilon} ")

            # Run TD(0)
            agent = MazeTD0(maze, alpha=alpha, gamma=gamma, epsilon=epsilon, episodes=episodes)
            snapshots, convergence = agent.run_episodes(snapshots_to_keep)

                        # Plot and save value function snapshots
            fig, axes = plt.subplots(2, 3, figsize=(15, 10))
            axes = axes.flatten()
            for ax, ep in zip(axes, snapshots_to_keep):
                plot_value_function(snapshots[ep], maze, ax=ax, title=f"{param}={val}, Ep={ep}")
            fig.suptitle(f"Value Function Evolution for {param}={val}")
            plt.tight_layout(rect=[0, 0, 1, 0.96])
            fig.savefig(f"part1/results/{param}_{val}_value_evolution.png")
            plt.close(fig)

            # Plot and save policy snapshots
            fig, axes = plt.subplots(2, 3, figsize=(15, 10))
            axes = axes.flatten()
            for ax, ep in zip(axes, snapshots_to_keep):
                plot_policy(snapshots[ep], maze, ax=ax, title=f"{param}={val}, Ep={ep}")
            fig.suptitle(f"Policy Evolution for {param}={val}")
            plt.tight_layout(rect=[0, 0, 1, 0.96])
            fig.savefig(f"part1/results/{param}_{val}_policy_evolution.png")
            plt.close(fig)

            # Plot and save convergence
            fig, ax = plt.subplots(figsize=(8, 4))
            plot_convergence(convergence, ax=ax, title=f"Convergence for {param}={val}")
            fig.savefig(f"part1/results/{param}_{val}_convergence.png")
            plt.close(fig)
```

B. Part 2

1) main.py:

```python
import gymnasium as gym
import numpy as np
import torch
from collections import deque
import random
import json

from models import QNetwork_1, QNetwork_2, QNetwork_3, QNetwork_4, QNetwork_5

#  Define a replay memory to store experiences:
class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0
    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity
    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
    def __len__(self):
        return len(self.buffer)

#  Define the DQN Agent encapsulating the networks, memory, and training procedure:
class DQNAgent:
    def __init__(self,
                 state_dim,
                 action_dim,
                 Net_No=3,
                 memory_size=50000,
                 batch_size=64,
                 gamma=0.99,
                 alpha=1e-3,
                 epsilon_start=1.0,
                 epsilon_min=0.01,
                 epsilon_decay=0.995,
                 target_update_freq=10):

        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma
        self.alpha = alpha
        self.batch_size = batch_size
        self.epsilon = epsilon_start
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.memory = ReplayMemory(memory_size)

        match Net_No:
            case 1:
                self.policy_net = QNetwork_1(state_dim, action_dim)
                self.target_net = QNetwork_1(state_dim, action_dim)
            case 2:
                self.policy_net = QNetwork_2(state_dim, action_dim)
                self.target_net = QNetwork_2(state_dim, action_dim)
            case 3:
                self.policy_net = QNetwork_3(state_dim, action_dim)
                self.target_net = QNetwork_3(state_dim, action_dim)
            case 4:
                self.policy_net = QNetwork_4(state_dim, action_dim)
                self.target_net = QNetwork_4(state_dim, action_dim)
            case 5:
                self.policy_net = QNetwork_5(state_dim, action_dim)
                self.target_net = QNetwork_5(state_dim, action_dim)
            case _:
                print("Error on Net No: missing")


        self.target_net.load_state_dict(self.policy_net.state_dict())
```

```
        self.target_net.eval()
        self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=alpha)
        self.target_update_freq = target_update_freq
        self.solved_score = 200.0       #target average reward for    solved
        self.solved_window = 100        #number of episodes to average for solved check

    def get_action(self, state):
        # Select an action for the given state using epsilon-greedy policy.
        if random.random() < self.epsilon:
            return random.randrange(self.action_dim)
        else:
            state_v = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
            with torch.no_grad():
                q_values = self.policy_net(state_v)
            return int(q_values.argmax(dim=1)[0])

    def train_step(self):
        #Perform one training step.
        if len(self.memory) < self.batch_size:
            return  # Not enough samples to train yet.
        batch = self.memory.sample(self.batch_size)
        states = torch.tensor(np.array([exp[0] for exp in batch]), dtype=torch.float32)
        actions = torch.tensor(np.array([exp[1] for exp in batch]), dtype=torch.int64).unsqueeze(1)
        rewards = torch.tensor(np.array([exp[2] for exp in batch]), dtype=torch.float32).unsqueeze(1)
        next_states = torch.tensor(np.array([exp[3] for exp in batch]), dtype=torch.float32)
        dones = torch.tensor(np.array([exp[4] for exp in batch]), dtype=torch.float32).unsqueeze(1)
        curr_Q = self.policy_net(states).gather(1, actions)
        next_Q = self.target_net(next_states).max(1)[0].detach().unsqueeze(1)

        # Compute target Q
        target_Q = rewards + (1 - dones) * (self.gamma * next_Q)
        #target_Q = self.alpha * (rewards + self.gamma * next_Q - curr_Q)

        # Compute loss
        loss = torch.nn.functional.mse_loss(curr_Q, target_Q)

        # Optimize
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target(self):
        # Update target network to match policy network.
        self.target_net.load_state_dict(self.policy_net.state_dict())

    def decay_epsilon(self):
        # Decay exploration rate after each episode.
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)


def run_experiment(config, output_path):
    # Main training loop for a single experiment
    env = gym.make("LunarLander-v3")
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    #print(f"lr={config['lr']} | g={config['gamma']} | ed={config['epsilon_decay']} | f={config['
        target_update_freq']} | NN={config['net_no']}")

    agent = DQNAgent(state_dim,
                     action_dim,
                     Net_No=config['net_no'],
                     gamma=config['gamma'],
                     epsilon_decay=config['epsilon_decay'],
                     target_update_freq=config['target_update_freq'],
                    )
    num_episodes = config['num_episodes']

    rewards_history = []
    avg_scores = []
    solved_episode = None
    scores_window = deque(maxlen=agent.solved_window)

    for episode in range(1, num_episodes + 1):
```

```
        state, _ = env.reset()
        total_reward = 0
        done = False

        for t in range(1000): # limit max steps per episode to avoid long runs
            action = agent.get_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            agent.memory.push(state, action, reward, next_state, done)
            agent.train_step()
            state = next_state
            total_reward += reward
            if done:
                break

        # Episode end
        rewards_history.append(total_reward)
        scores_window.append(total_reward)
        # Compute moving average
        avg = np.mean(scores_window)
        avg_scores.append(avg)
        # Check solved
        if solved_episode is None and len(scores_window) == agent.solved_window and avg >= agent.
            solved_score:
            solved_episode = episode
        # Epsilon decay
        agent.decay_epsilon()
        # Update target network
        if episode % agent.target_update_freq == 0:
            agent.update_target()
        print(f"Episode␣{episode}␣|␣Reward:␣{total_reward:.2f}␣|␣Avg:␣{avg:.2f}␣|␣Epsilon:␣{agent.epsilon
            :.3f}")

    env.close()
    # Save results
    results = {
        'episode_rewards': rewards_history,
        'average_scores': avg_scores,
        'hyperparameters': config,
        'solved_episode': solved_episode
    }
    with open(output_path, 'w') as f:
        json.dump(results, f)
    print(f"Results␣saved␣to␣{output_path}")

# Example usage
if __name__ == "__main__":
    base_config = {
        'lr': 1e-3,
        'gamma': 0.99,
        'epsilon_decay': 0.995,
        'target_update_freq': 10,
        'net_no': 3,
        'num_episodes': 5000
    }

    param_grid = {
        'lr': [1e-4, 1e-3, 5e-3],
        'gamma': [0.98, 0.99, 0.999],
        'epsilon_decay': [0.98, 0.99, 0.995],
        'target_update_freq': [1, 10, 50],
        'net_no': [1, 2, 3, 4, 5]
    }

    for param, values in param_grid.items():
        for val in values:
            config = base_config.copy()
            config[param] = val
            save_path = f'part2/results/{param}_{val}.json'
            print(f"Running␣experiment␣with␣{param}={val}")
            run_experiment(config, save_path)
```

2) main.py:

```
import torch

# Q-Network FC-128, ReLU
class QNetwork_1(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(QNetwork_1, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return x


    # Q-Network FC-64, ReLU and FC-64, ReLU
class QNetwork_2(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim1=64, hidden_dim2=64):
        super(QNetwork_2, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim1)
        self.fc2 = torch.nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = torch.nn.Linear(hidden_dim2, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        return x


    # Q-Network FC-128, ReLU and FC-128, ReLU
class QNetwork_3(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim1=128, hidden_dim2=128):
        super(QNetwork_3, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim1)
        self.fc2 = torch.nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = torch.nn.Linear(hidden_dim2, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        return x


    # Q-Network FC-128, ReLU and FC-128, ReLU and FC-128, ReLU
class QNetwork_4(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim1=128, hidden_dim2=128, hidden_dim3=128):
        super(QNetwork_4, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim1)
        self.fc2 = torch.nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = torch.nn.Linear(hidden_dim2, hidden_dim3)
        self.fc4 = torch.nn.Linear(hidden_dim3, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = torch.relu(self.fc4(x))
        return x


    # Q-Network FC-256, ReLU and FC-256, ReLU
class QNetwork_5(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim1=256, hidden_dim2=256):
        super(QNetwork_5, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim1)
        self.fc2 = torch.nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = torch.nn.Linear(hidden_dim2, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
```

```
74          x = torch.relu(self.fc3(x))
75          return x
```

3) main.py:

```python
import sys
import os
sys.path.append(os.path.abspath('_given'))
from utils import plot_learning_curves, plot_solved_episodes # type: ignore


def main():
    # Directory containing experiment JSON results
    base_dir = os.path.join(os.path.dirname(__file__), 'results')

    # Hyperparameter experiments and their values (must match JSON filenames)
    experiments = {
        'lr': [0.0001, 0.001, 0.005],
        'gamma': [0.98, 0.99, 0.999],
        'epsilon_decay': [0.98, 0.99, 0.995],
        'target_update_freq': [1, 10, 50],
        'net_no': [1, 2, 3, 4, 5]
    }

    # Plot learning curves for each hyperparameter
    for param, values in experiments.items():
        # Construct list of JSON paths for this parameter
        json_paths = []
        labels = []
        for val in values:
            # Format value to match filename
            val_str = str(val)
            json_file = f"{param}_{val_str}.json"
            path = os.path.join(base_dir, json_file)
            if os.path.isfile(path):
                json_paths.append(path)
                labels.append(str(val))
            else:
                print(f"Warning: missing result file {path}")

        if not json_paths:
            print(f"No data for parameter '{param}', skipping.")
            continue

        # Output image filename
        out_png = f"part2/results_fig/learning_curves_{param}.png"
        # Generate and save figure
        plot_learning_curves(
            json_paths=json_paths,
            labels=labels,
            output_file=out_png
        )
        print(f"Saved learning curves for '{param}' at: {out_png}")

    # Plot bar chart of solved episodes for all experiments
    # Gather all JSON result files
    all_json = []
    for fname in os.listdir(base_dir):
        if fname.endswith('.json'):
            all_json.append(os.path.join(base_dir, fname))

    if not all_json:
        print("No JSON result files found, skipping solved-episode bar chart.")
        return

    all_labels = [
        os.path.splitext(os.path.basename(path))[0]
        for path in all_json
    ]

    # Output bar chart filename
    out_bar = 'part2/results_fig/solved_episodes.png'

    # Generate and save bar chart
    plot_solved_episodes(
        json_paths=all_json,
        labels = all_labels,
        output_file=out_bar
```

```
    )
    print(f"Saved␣solved␣episodes␣bar␣chart␣at:␣{out_bar}")


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