

DA6400 : Reinforcement Learning
Programming Assignment #1
Report

Ritabrata Mandal
EE24E009

March 28, 2025

Contents

1	Introduction	3
1.1	Environments	3
1.2	Algorithms	3
2	Implementation	4
2.1	CartPole-v1	4
2.1.1	Code Snippets	4
2.1.2	SARSA Hyper-Parameter Tuning	4
2.1.3	SARSA best 3 results	4
2.1.4	Q-Learning hyper-parameter tuning	6
2.1.5	Q-Learning best 3 results	6
2.1.6	Result(SARSA vs Q-Learning)	6
2.2	MountainCar-v0	6
2.2.1	Code Snippets	6
2.2.2	SARSA hyper-parameter tuning	9
2.2.3	SARSA best 3 results	9
2.2.4	Q-Learning hyper-parameter tuning	11
2.2.5	Q-Learning best 3 results	11
2.2.6	Result(SARSA vs Q-Learning)	13
2.3	MiniGrid-Dynamic-Obstacles-5x5-v0	13
2.3.1	Code Snippets	13
2.3.2	SARSA hyper-parameter tuning	15
2.3.3	SARSA best 3 results	16
2.3.4	Q-Learning hyper-parameter tuning	18
2.3.5	Q-Learning best 3 results	18
2.3.6	Result(SARSA vs Q-Learning)	20
3	Conclusion	20
4	Github link	20

1 Introduction

1.1 Environments

In this programming task, we are utilize the following [Gymnasium environments](#) for training and evaluating your policies. The links associated with the environments contain descriptions of each environment.

- [CartPole-v1](#) : A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.
- [MountainCar-v0](#) : The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill. There are two versions of the mountain car domain in gymnasium: one with *discrete actions* and one with *continuous*. This version is the one with discrete actions.
- [MiniGrid-Dynamic-Obstacles-5x5-v0](#) : This environment is an empty room with moving obstacles. The goal of the agent is to reach the green goal square without colliding with any obstacle. A large penalty is subtracted if the agent collides with an obstacle and the episode finishes. This environment is useful to test Dynamic Obstacle Avoidance for mobile robots with Reinforcement Learning in Partial Observability.

1.2 Algorithms

Training each of the below algorithms and assessing their comparative performance.

- **SARSA**→ with ϵ -greedy exploration
- **Q-Learning**→ with Softmax exploration

2 Implementation

2.1 CartPole-v1

2.1.1 Code Snippets

2.1.2 SARSA Hyper-Parameter Tuning

Using Weights & Biases (wandb) with a sweep method based on Bayesian optimization, we identified the best-performing hyper-parameters. Specifically, we set $\alpha \in [0.1, 0.5]$ and $\epsilon \in [0.01, 0.15]$, and ran 4000 episodes while minimizing the regret, defined as $195 -$ (all-time average return). See the wandb report on this environment here. Additionally, Figure 1 displays the results from 50 sweeps, illustrating the relationship between α , ϵ , and the reward.

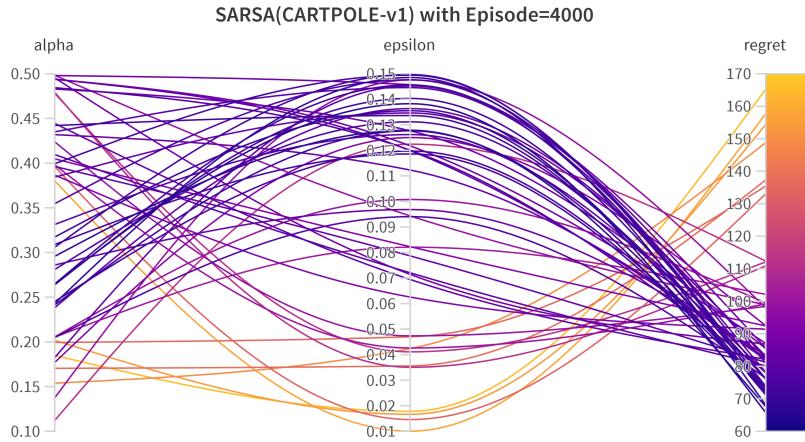
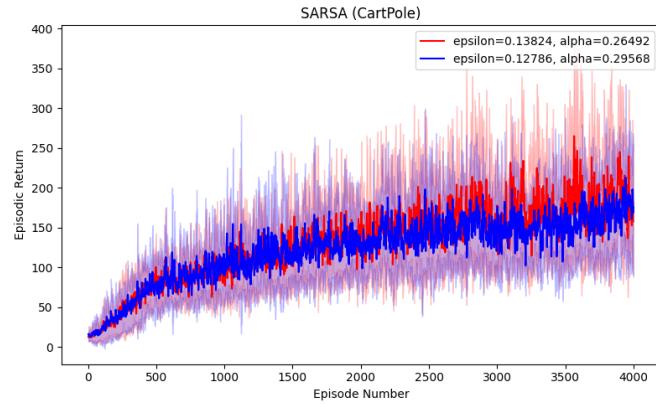


Figure 1: SARSA hyper-parameter sweeps for α and ϵ in CartPole-v1(4000 episodes), with lines color-coded by regret.

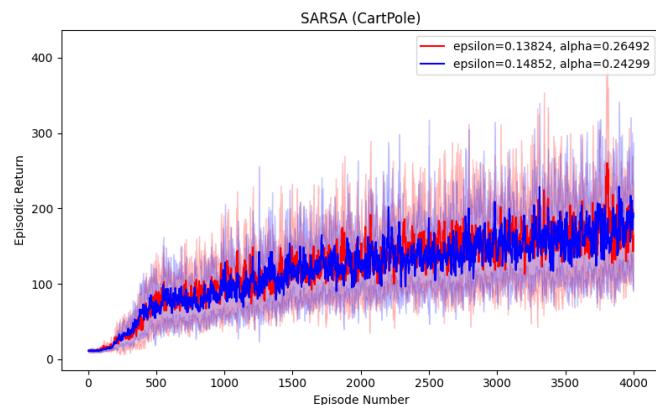
2.1.3 SARSA best 3 results

	α	ϵ	γ	regret
1	0.26492	0.13824	0.99	65.7379
2	0.29568	0.12786	0.99	67.5396
3	0.24299	0.14852	0.99	68.09645

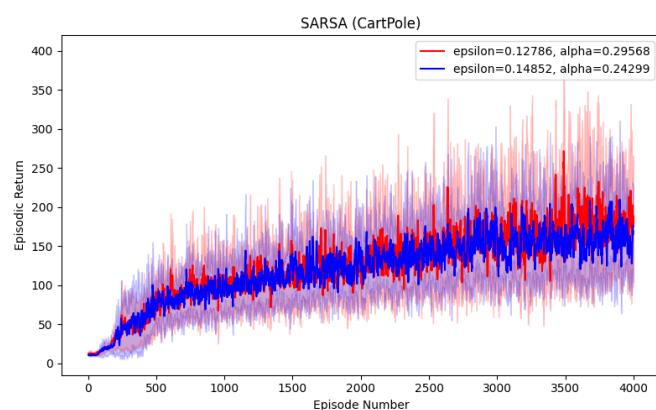
Figure 2 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 2: Performance comparison of the three best SARSA parameter configurations: (a) 1 vs 2, (b) 1 vs 3, and (c) 2 vs 3.

2.1.4 Q-Learning hyper-parameter tuning

2.1.5 Q-Learning best 3 results

	α	τ	regret
1			
2			
3			

2.1.6 Result(SARSA vs Q-Learning)

2.2 MountainCar-v0

2.2.1 Code Snippets

```
1 def getState(state, env_low=env_low, env_high=env_high, bins=bins):
2     """Returns the discretized position and velocity of an
3     observation"""
4     discretized_env = (env_high - env_low) / bins
5     discretized_pos = int((state[0] - env_low[0]) /
6                           discretized_env[0])
7     discretized_vel = int((state[1] - env_low[1]) /
8                           discretized_env[1])
9     # Clip to avoid out-of-bound errors
10    discretized_pos = np.clip(discretized_pos, 0, bins - 1)
11    discretized_vel = np.clip(discretized_vel, 0, bins - 1)
12    return discretized_pos, discretized_vel
```

Listing 1: Discretized states

```
1 def chooseAction(pos, vel, q_table, epsilon):
2     """Choose action based on an epsilon greedy strategy"""
3     if random.random() < epsilon: # Explore
4         action = env.action_space.sample()
5     else: # Exploit
6         action = np.argmax(q_table[pos][vel])
7     return action
```

Listing 2: epsilon-greedy action selection

```
1 all_rewards = []
2 for seed in seeds:
3     print(f"\n--- Training with Seed: {seed} ---")
4     np.random.seed(seed)
5     random.seed(seed)
6     env.reset(seed=seed)
7
```

```

8
9     q_table_sarsa = np.zeros((bins + 1, bins + 1, env.
10    action_space.n))
11
12    rewards_sarsa = []
13
14    for ep in range(episode):
15        current_reward = 0
16        done = False
17        truncated = False
18        state, _ = env.reset(seed=seed)
19        pos, vel = getState(state)
20        action = chooseAction(pos, vel, q_table_sarsa, epsilon)
21
22        # while not (done or truncated):
23        while not done:
24            next_state, reward, done, truncated, _ = env.step(
25                action)
26            next_pos, next_vel = getState(next_state)
27            next_action = chooseAction(next_pos, next_vel,
28                q_table_sarsa, epsilon)
29
30            if done:
31                q_table_sarsa[pos][vel][action] += alpha * (
32                    reward - q_table_sarsa[pos][vel][action])
33            else:
34                q_table_sarsa[pos][vel][action] += alpha * (
35                    reward + gamma * q_table_sarsa[next_pos][
36                    next_vel][next_action] - q_table_sarsa[pos][vel][action]
37                    )
38
39            pos, vel = next_pos, next_vel
40            action = next_action
41            current_reward += reward
42            rewards_sarsa.append(current_reward)
43            print(f'seed {seed} Episode {ep+1}/{episode}, Reward: {current_reward}')
44            all_rewards.append(rewards_sarsa)

```

Listing 3: SARSA implementation

```

1 def softmax(q_values, temperature):
2     """Returns a probability distribution over actions using
3     softmax"""
4     exp_q = np.exp(q_values / temperature)
5     return exp_q / np.sum(exp_q)
6
7 def chooseAction(pos, vel, q_table, temperature):
8     """Choose action based on Softmax exploration"""

```

```

8     q_values = q_table[pos][vel]
9     action_probs = softmax(q_values, temperature)
10    action = np.random.choice(np.arange(len(q_values)), p=
11        action_probs)
12    return action

```

Listing 4: Softmax action selection

```

1 for seed in seeds:
2     np.random.seed(seed)
3     random.seed(seed)
4     env.reset(seed=seed)
5
6     q_table_qlearn = np.zeros((bins + 1, bins + 1, env.
7         action_space.n))
8     rewards_qlearn = []
9
10    for ep in range(episode):
11        current_reward = 0
12        done = False
13        truncated = False
14        state, _ = env.reset(seed=seed)
15        pos, vel = getState(state)
16
17        # while not (done or truncated):
18        while not done:
19            action = chooseAction(pos, vel, q_table_qlearn,
20                temperature)
21            next_state, reward, done, truncated, _ = env.step(
22                action)
23            next_pos, next_vel = getState(next_state)
24            # if done or truncated:
25            if done:
26                q_table_qlearn[pos][vel][action] += alpha * (
27                    reward - q_table_qlearn[pos][vel][action])
28            else:
29                max_next_q = np.max(q_table_qlearn[next_pos][
30                    next_vel])
31                q_table_qlearn[pos][vel][action] += alpha * (
32                    reward + gamma * max_next_q -
33                    q_table_qlearn[pos][vel][action]
34                )
35
36            pos, vel = next_pos, next_vel
37            current_reward += reward
38
39            rewards_qlearn.append(current_reward)
40            print(f'Seed {seed} Episode {ep+1}/{episode}, Reward: {current_reward}')

```

```

35
36     all_rewards.append(rewards_qlearn)

```

Listing 5: Q-Learning implementation

2.2.2 SARSA hyper-parameter tuning

Using Weights & Biases (wandb) with a sweep method based on Bayesian optimization, we identified the best-performing hyper-parameters. Specifically, we set $\alpha \in [0.1, 0.5]$ and $\epsilon \in [0.01, 0.15]$, and ran 2000 episodes while minimizing the regret, defined as $-200 -$ (all-time average return). See the wandb report on this environment here. Additionally, Figure 3 displays the results from 50 sweeps, illustrating the relationship between α , ϵ , and the reward.

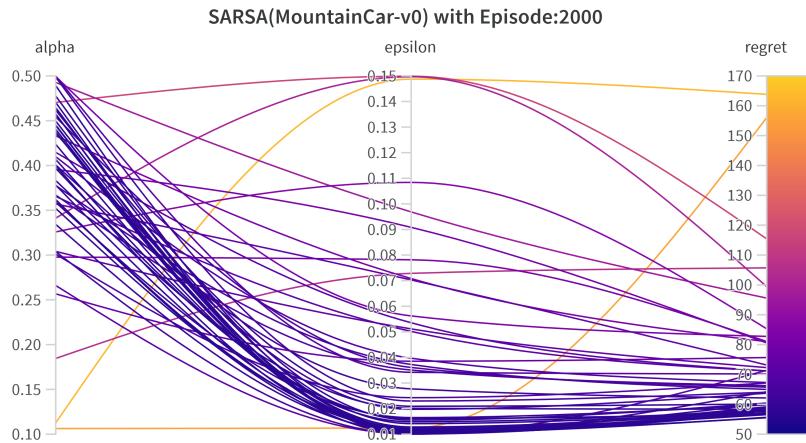
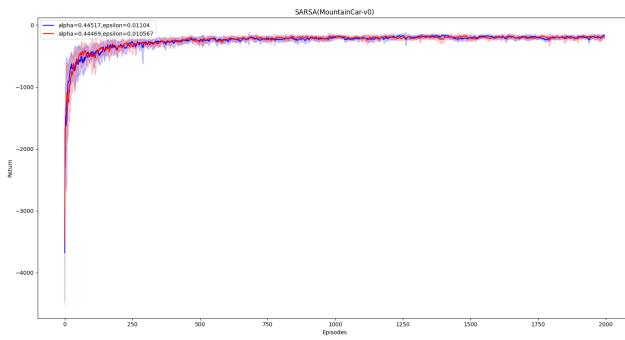


Figure 3: SARSA hyper-parameter sweeps for α and ϵ in MountainCar-v0(2000 episodes), with lines color-coded by regret.

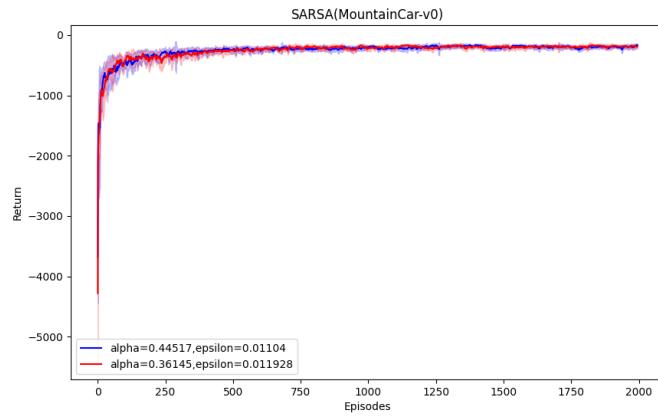
2.2.3 SARSA best 3 results

	α	ϵ	γ	regret
1	0.44517	0.01104	0.99	55.5099
2	0.44469	0.010567	0.99	55.6604
3	0.36145	0.011928	0.99	56.547

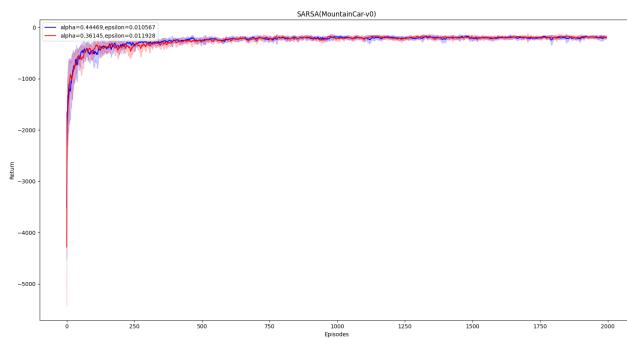
Figure 4 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 4: Performance comparison of the three best SARSA parameter configurations: (a) 1 vs 2, (b) 1 vs 3, and (c) 2 vs 3.

2.2.4 Q-Learning hyper-parameter tuning

Using Weights & Biases (wandb) with a sweep method based on Bayesian optimization, we identified the best-performing hyper-parameters. Specifically, we set $\alpha \in [0.1, 0.5]$ and $\tau \in [0.1, 1.5]$, and ran 2000 episodes while minimizing the regret, defined as $-200 -$ (all-time average return). See the wandb report on this environment here. Additionally, Figure 5 displays the results from 50 sweeps, illustrating the relationship between α , τ , and the reward.

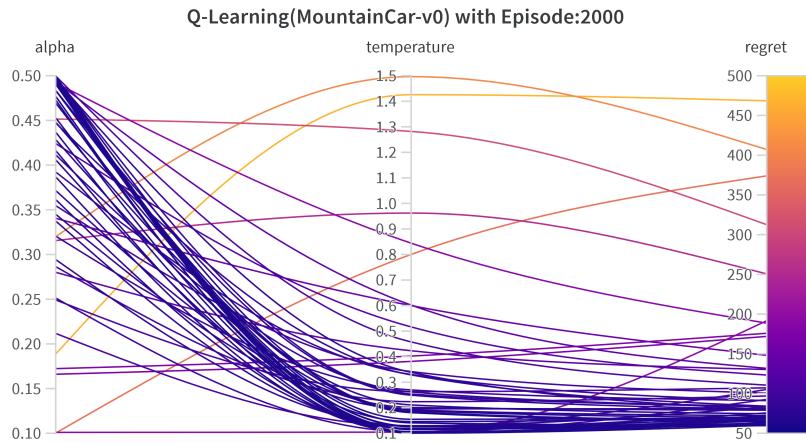
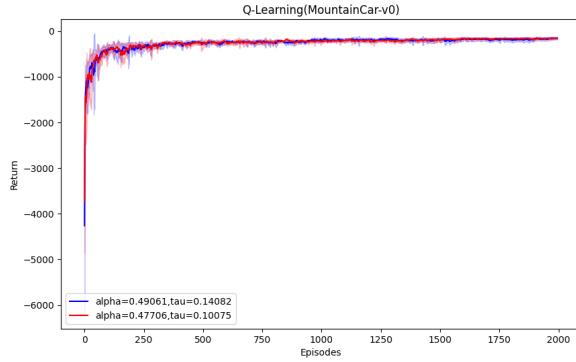


Figure 5: Q-Learning hyper-parameter sweeps for α and τ in MountainCar-v0(2000 episodes), with lines color-coded by regret.

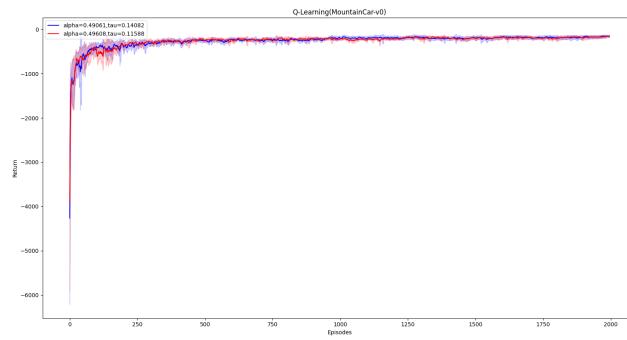
2.2.5 Q-Learning best 3 results

	α	τ	γ	regret
1	0.49061	0.14082	0.99	59.6092
2	0.47706	0.10075	0.99	60.0469
3	0.49608	0.11588	0.99	60.4235

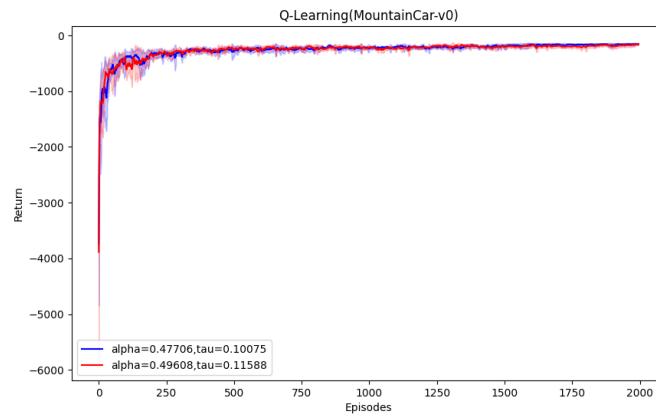
Figure 6 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 6: Performance comparison of the three best Q-Learning parameter configurations: (a) 1 vs 2, (b) 1 vs 3, and (c) 2 vs 3.

2.2.6 Result(SARSA vs Q-Learning)

The SARSA implementation achieved a best average reward of -255 over 2000 episodes, whereas Q-learning achieved -259 over the same number of episodes.

2.3 MiniGrid-Dynamic-Obstacles-5x5-v0

2.3.1 Code Snippets

```
1 def take_action(q_value, epsilon):
2     """Choose an action using an epsilon-greedy strategy."""
3     if np.random.random() < epsilon:
4         return np.random.randint(0, 3) # Explore
5     return np.argmax(q_value) # Exploit
```

Listing 6: epsilon-greedy action selection

```
1     for seed in seeds:
2         env = gym.make('MiniGrid-Dynamic-Obstacles-Random-5x5-
v0')
3         env.reset(seed=seed)
4         # Q-table dimensions: action x (5x5 grid) x agent
5         # direction (4) x front cell flag (2)
6         q_value = np.zeros((3, 25, 4, 2))
7         total_reward = np.zeros(episodes)
8
8     for ep in range(episodes):
9         env.reset()
10        terminated, truncated = False, False
11        # Calculate state indices
12        x1 = env.agent_pos[0] * 5 + env.agent_pos[1]
13        x2 = env.agent_dir
14        front_cell = env.grid.get(*env.front_pos)
15        x3 = 1 if (front_cell and front_cell.type != "goal"
16    ) else 0
17
17        action = take_action(q_value[:, x1, x2, x3],
18        epsilon)
18
19        while not (terminated or truncated):
20            observation, reward, terminated, truncated,
21            info = env.step(action)
22            new_x1 = env.agent_pos[0] * 5 + env.agent_pos
23            [1]
23            new_x2 = env.agent_dir
23            front_cell = env.grid.get(*env.front_pos)
```

```

24         new_x3 = 1 if (front_cell and front_cell.type
25             != "goal") else 0
26         new_action = take_action(q_value[:, new_x1,
27             new_x2, new_x3], epsilon)
28
29             # SARSA update rule
30             q_value[action, x1, x2, x3] += alpha * (
31                 reward + gamma * q_value[new_action, new_x1
32                 , new_x2, new_x3] - q_value[action, x1, x2, x3]
33             )
34
35             x1, x2, x3, action = new_x1, new_x2, new_x3,
36             new_action
37             total_reward[ep] += reward
38
39             print(f"Seed: {seed} Episode: {ep+1} Reward: {total_reward[ep]}")
40
41             all_rewards.append(total_reward)
42             env.close()

```

Listing 7: SARSA implementation

```

1 def softmax_action(q_value, temperature):
2     """Return an action selected via softmax exploration."""
3     exp_values = np.exp(q_value / temperature)
4     probabilities = exp_values / np.sum(exp_values)
5     return np.random.choice(len(q_value), p=probabilities)

```

Listing 8: Softmax action selection

```

1 for seed in seeds:
2     env = gym.make('MiniGrid-Dynamic-Obstacles-Random-5x5-
v0')
3     env.reset(seed=seed)
4     # Q-table dimensions: actions x (5x5 grid) x agent
5     # direction (4) x front cell flag (2)
6     q_value = np.zeros((3, 25, 4, 2))
7     total_reward = np.zeros(episodes)
8
9     for ep in range(episodes):
10         env.reset()
11         terminated, truncated = False, False
12         # Compute state indices
13         x1 = env.agent_pos[0] * 5 + env.agent_pos[1]
14         x2 = env.agent_dir
15         front_cell = env.grid.get(*env.front_pos)
16         x3 = 1 if (front_cell and front_cell.type != "goal"
17         ) else 0

```

```

16
17         # Select action using softmax
18         action = softmax_action(q_value[:, x1, x2, x3],
19         temperature)
20
21         while not (terminated or truncated):
22             observation, reward, terminated, truncated,
23             info = env.step(action)
24             new_x1 = env.agent_pos[0] * 5 + env.agent_pos
25             [1]
26             new_x2 = env.agent_dir
27             front_cell = env.grid.get(*env.front_pos)
28             new_x3 = 1 if (front_cell and front_cell.type
29             != "goal") else 0
30
31             # Q-learning update with softmax exploration
32             best_next_action = np.argmax(q_value[:, new_x1,
33             new_x2, new_x3])
34             q_value[action, x1, x2, x3] += alpha * (
35                 reward + gamma * q_value[best_next_action,
36             new_x1, new_x2, new_x3] - q_value[action, x1, x2, x3]
37             )
38
39             # Move to next state
40             x1, x2, x3 = new_x1, new_x2, new_x3
41             action = softmax_action(q_value[:, x1, x2, x3],
42             temperature)
43             total_reward[ep] += reward
44
45             print(f"Seed: {seed} Episode: {ep+1} Reward: {total_reward[ep]}")
46
47             all_rewards.append(total_reward)
48             env.close()

```

Listing 9: Q-Learning implementation

2.3.2 SARSA hyper-parameter tuning

Using Weights & Biases (wandb) with a sweep method based on Bayesian optimization, we identified the best-performing hyper-parameters. Specifically, we set $\alpha \in [0.1, 0.5]$ and $\epsilon \in [0.01, 0.15]$, and ran 2000 episodes while minimizing the regret, defined as $1 - (\text{all-time average return})$. See the wandb report on this environment here. Additionally, Figure 7 displays the results from 50 sweeps, illustrating the relationship between α , ϵ , and the reward.

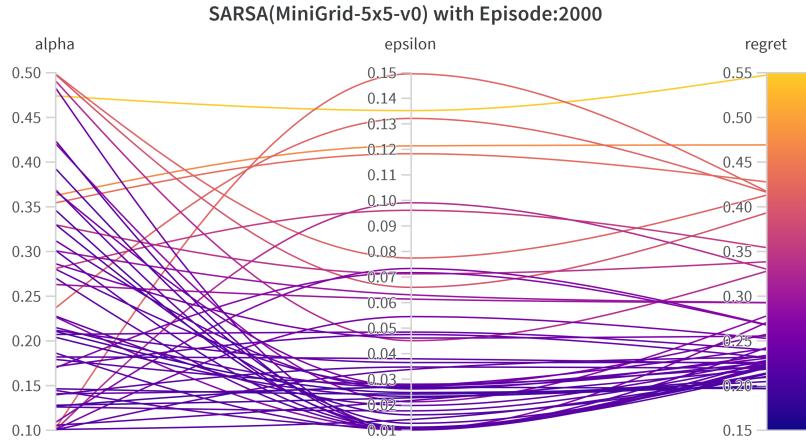
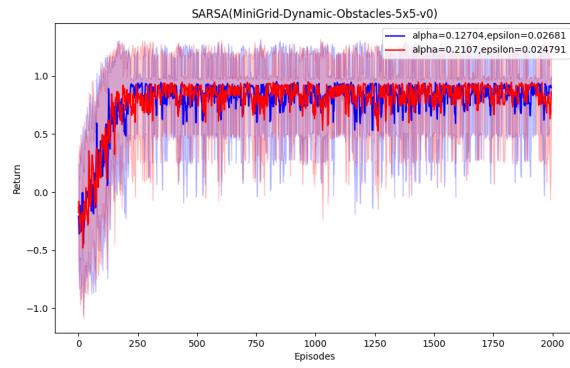


Figure 7: SARSA hyper-parameter sweeps for α and ϵ in MiniGrid-Dynamic-Obstacles-5x5-v0(2000 episodes), with lines color-coded by regret.

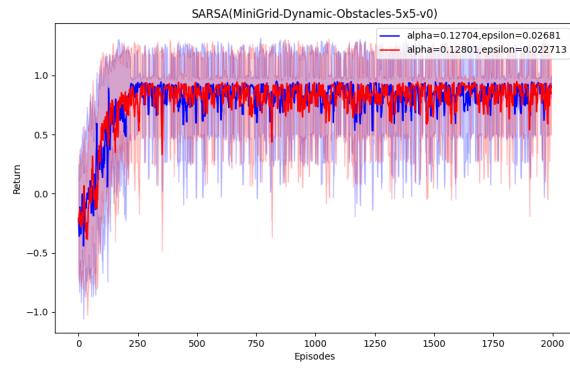
2.3.3 SARSA best 3 results

	α	ϵ	γ	regret
1	0.12704	0.02681	0.99	0.19653
2	0.2107	0.024791	0.99	0.19896
3	0.12801	0.022713	0.99	0.199997

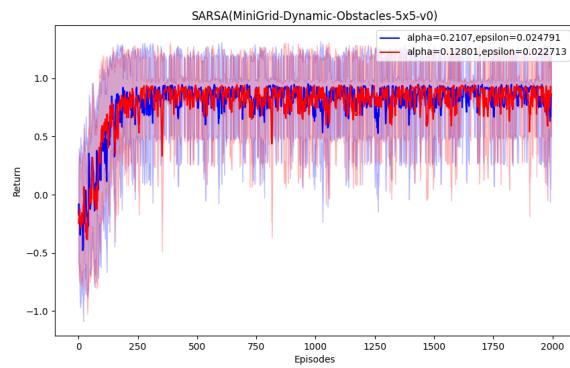
Figure 8 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 8: Performance comparison of the three best SARSA parameter configurations: (a) 1 vs 2, (b) 1 vs 3, and (c) 2 vs 3.

2.3.4 Q-Learning hyper-parameter tuning

Using Weights & Biases (wandb) with a sweep method based on Bayesian optimization, we identified the best-performing hyper-parameters. Specifically, we set $\alpha \in [0.1, 0.5]$ and $\tau \in [0.1, 1.5]$, and ran 2000 episodes while minimizing the regret, defined as $1 - \text{all-time average return}$. See the wandb report on this environment here. Additionally, Figure 9 displays the results from 50 sweeps, illustrating the relationship between α , τ , and the reward.

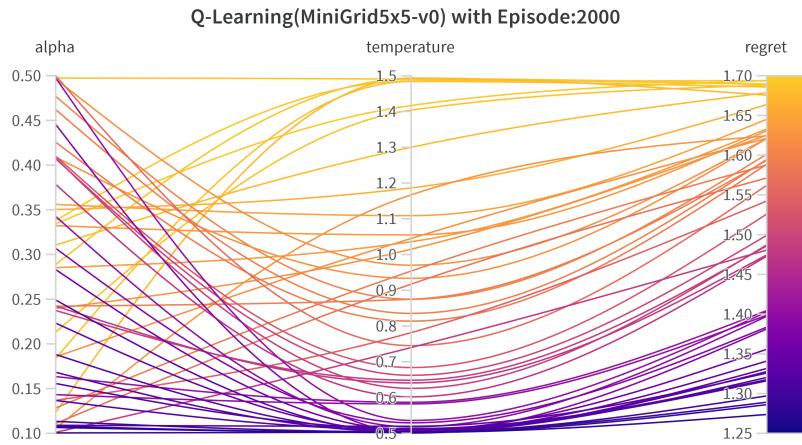
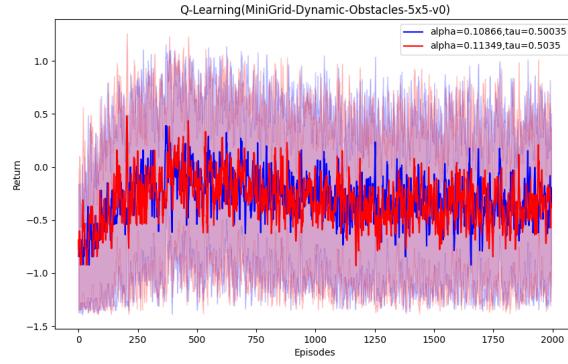


Figure 9: Q-Learning hyper-parameter sweeps for α and τ in MiniGrid-Dynamic-Obstacles-5x5-v0(2000 episodes), with lines color-coded by regret.

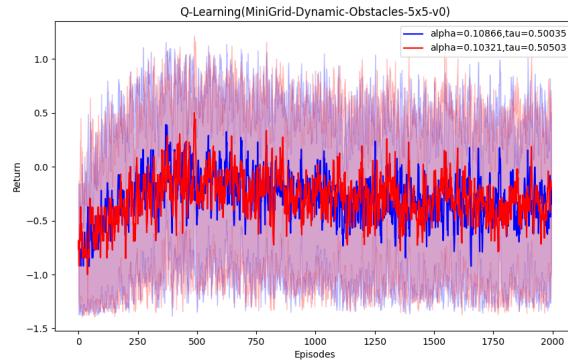
2.3.5 Q-Learning best 3 results

	α	τ	γ	regret
1	0.10866	0.50035	0.99	1.22505
2	0.11349	0.5035	0.99	1.25063
3	0.10321	0.50503	0.99	1.25763

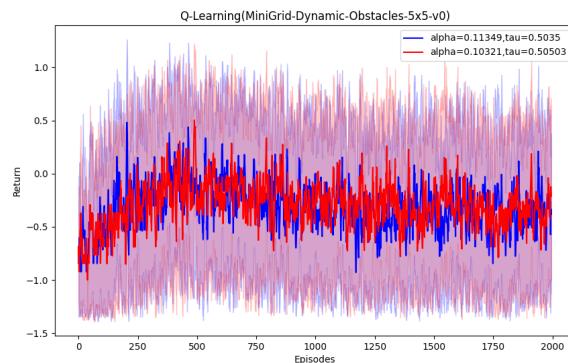
Figure 10 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 10: Performance comparison of the three best Q-Learning parameter configurations: (a) 1 vs 2, (b) 1 vs 3, and (c) 2 vs 3.

2.3.6 Result(SARSA vs Q-Learning)

The SARSA implementation achieved a best average reward of 0.8 over 2000 episodes, whereas Q-learning achieved -0.2 over the same number of episodes.

3 Conclusion

4 Github link

https://github.com/RitabrataMandal/RL-DA6400-assignment_1

requirements.txt

```
gymnasium==0.29.1
numpy==1.26.4
pygame==2.5.2
matplotlib==3.8.4
argparse==1.1
wandb==0.19.8
```

```
1 method: bayes
2 metric:
3   name: regret
4   goal: minimize
5 parameters:
6   alpha:
7     min: 0.1
8     max: 0.5
9   epsilon:
10    min: 0.01
11    max: 0.15
```

Listing 10: SARSA.yaml

```
1 method: bayes
2 metric:
3   name: regret
4   goal: minimize
5 parameters:
6   temperature:
7     min: 0.5
8     max: 1.5
9   alpha:
```

```
10      min: 0.1
11      max: 0.5
```

Listing 11: Q-Learning.yaml