DA6400 : Reinforcement Learning
Programming Assignment #1
Report

Ritabrata Mandal
EE24E009

March 30, 2025

Contents

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

```
def Qtable(state_space, action_space, bin_size=100):

    bins = np.zeros((state_space.shape[0], bin_size))

    bins[0] = np.linspace(-4.8, 4.8, bin_size)
    bins[1] = np.linspace(-4, 4, bin_size)
    bins[2] = np.linspace(-0.42, 0.42, bin_size)
    bins[3] = np.linspace(-4, 4, bin_size)

    q_table = np.zeros((bin_size, bin_size, bin_size, bin_size,
     action_space.n))

    return q_table, bins
```

Listing 1: Q-table

```
def discretize_state(state_space, bins):

    state_discrete = np.zeros(state_space.shape)

    for i in range(state_space.shape[0]):
        state_discrete[i] = np.digitize(state_space[i], bins[i
    ])

    return state_discrete.astype(np.int32)
```

Listing 2: Discretized sates

```
class EpsilonGreedyPolicy:
    def __init__(self, epsilon, q_table, env):
        self.epsilon = epsilon
        self.q_table = q_table
        self.env = env

    def get_action(self, state):

        if np.random.rand() < self.epsilon:
            action = self.env.action_space.sample()
        else:
            action = np.argmax(self.q_table[state[0], state[1],
     state[2], state[3]])
        return action
```

Listing 3: epsilon-greedy action selection

```python
class SarasLearner:
    def __init__(self, alpha, gamma, epsilon, q_table, bins, env, seed):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.q_table = q_table
        self.env = env
        self.bins = bins
        self.seed = seed
        self.policy = EpsilonGreedyPolicy(self.epsilon, self.q_table, self.env)

    def compute_td_error(self, state, action, next_state, next_action, reward):

        return reward + self.gamma * self.q_table[next_state[0], next_state[1], next_state[2], next_state[3], next_action] - \
            self.q_table[state[0], state[1], state[2], state[3], action]

    def update_q_table(self, state, action, td_error):

        self.q_table[state[0], state[1], state[2], state[3], action] += self.alpha * td_error

    def learn(self, num_episodes, num_steps):
        reward_list = []
        for episode in range(num_episodes):
            state, _ = self.env.reset(seed=self.seed)
            state_discrete = discretize_state(state, self.bins)
            action = self.policy.get_action(state_discrete)
            total_reward = 0

            for step in range(num_steps):
                next_state, reward, done = self.env.step(action)[:3]
                next_state_discrete = discretize_state(next_state, self.bins)
                next_action = self.policy.get_action(next_state_discrete)  # Select next action

                td_error = self.compute_td_error(state_discrete, action, next_state_discrete, next_action, reward)
                self.update_q_table(state_discrete, action, td_error)

                state_discrete, action = next_state_discrete,
```

```
    next_action
                total_reward += reward

                if done:
                    break

            print(f"Seed: {self.seed} Episode: {episode + 1}/{num_episodes}, Total Reward: {total_reward}")
            reward_list.append(total_reward)

        return reward_list
```

Listing 4: SARSA implementation

```
class SoftmaxPolicy:
    def __init__(self, temperature, q_table, env):
        self.temperature = temperature
        self.q_table = q_table
        self.env = env

    def get_action(self, state):
        q_values = self.q_table[state[0], state[1], state[2], state[3]]
        max_q = np.max(q_values)
        exp_q = np.exp((q_values - max_q) / self.temperature)
        probabilities = exp_q / np.sum(exp_q)
        action = np.random.choice(len(q_values), p=probabilities)
        return action
```

Listing 5: Softmax action selection

```
class QLearner:

    def __init__(self, alpha, gamma, temperature, q_table, bins, env, seed):
        self.alpha = alpha
        self.gamma = gamma
        self.temperature = temperature
        self.q_table = q_table
        self.bins = bins
        self.env = env
        self.seed = seed
        self.policy = SoftmaxPolicy(self.temperature, self.q_table, self.env)

    def compute_td_error(self, state, action, next_state, reward):

```

```
        best_next_action_value = np.max(self.q_table[next_state[0], next_state[1], next_state[2], next_state[3], :])
        return reward + self.gamma * best_next_action_value - self.q_table[state[0], state[1], state[2], state[3], action]

    def update_q_table(self, state, action, td_error):

        self.q_table[state[0], state[1], state[2], state[3], action] += self.alpha * td_error

    def learn(self, num_episodes, num_steps):

        reward_list = []

        for episode in range(num_episodes):
            state, _ = self.env.reset(seed=self.seed)
            state_discrete = discretize_state(state, self.bins)
            total_reward = 0

            for step in range(num_steps):
                action = self.policy.get_action(state_discrete)
                next_state, reward, done = self.env.step(action)[:3]
                next_state_discrete = discretize_state(next_state, self.bins)

                td_error = self.compute_td_error(state_discrete, action, next_state_discrete, reward)
                self.update_q_table(state_discrete, action, td_error)

                state_discrete = next_state_discrete
                total_reward += reward

                if done:
                    break

            print(f"Episode: {episode + 1}/{num_episodes}, Total Reward: {total_reward}")
            reward_list.append(total_reward)

        return reward_list
```

Listing 6: Q-Learning implementation

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 regret.

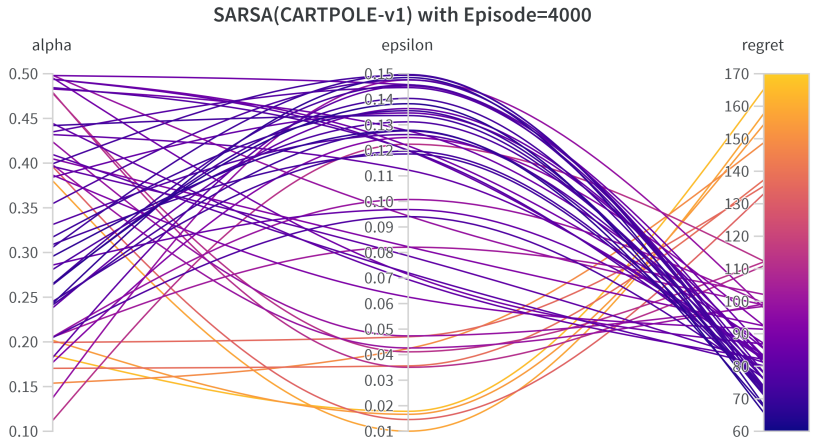
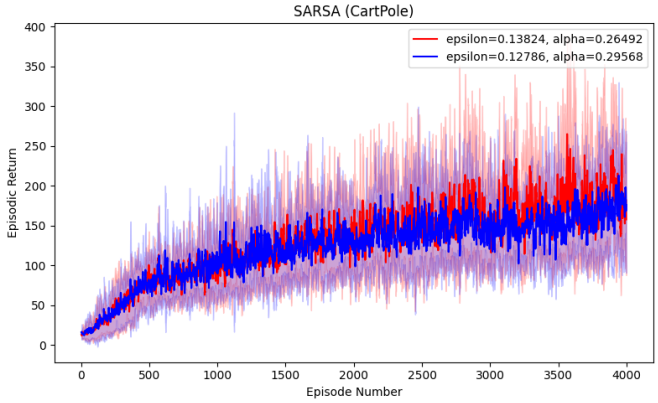


Figure 1: SARSA hyper-parameter sweeps for α and ϵ in CartPole-v1(4000 episodes), with lines color-coded by regret.

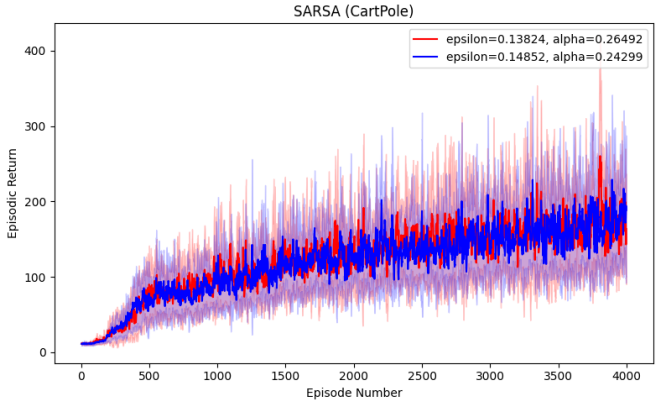
2.1.3 SARSA best 3 results

	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

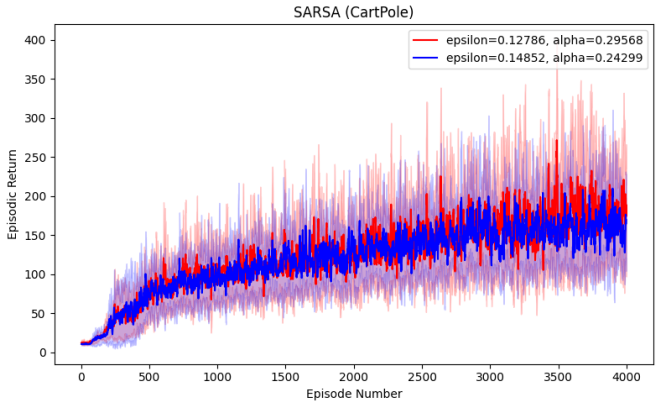
Figure 2 include the episode vs reward plot for best three hyper-parameter combination.



(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

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 4000 episodes while minimizing the regret, defined as $195-$ (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, τ, and the regret.

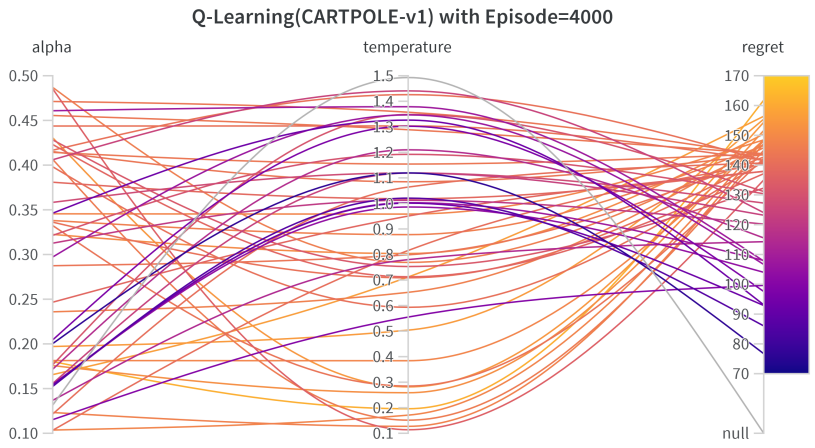
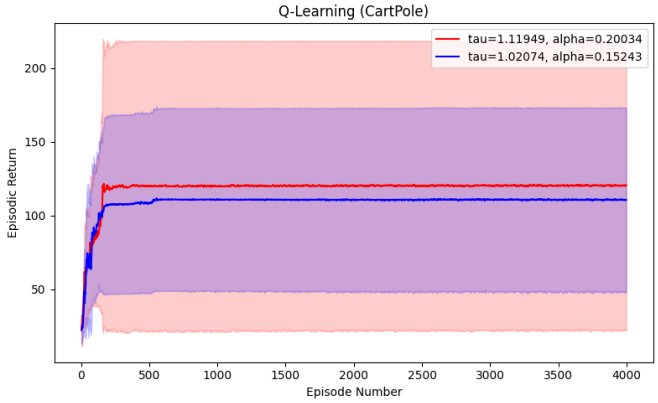


Figure 3: Q-Learning hyper-parameter sweeps for α and τ in Cartpole-v1(4000 episodes), with lines color-coded by regret.

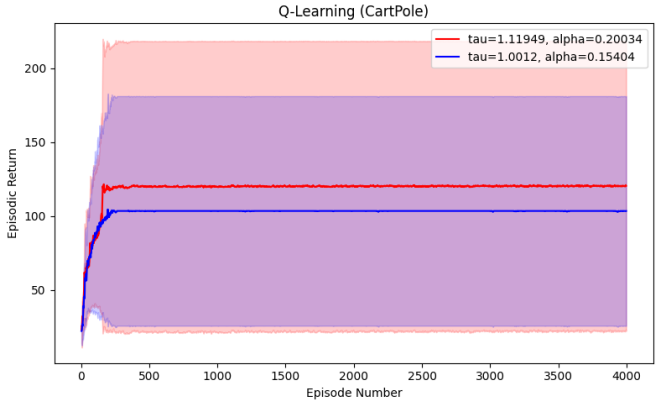
2.1.5 Q-Learning best 3 results

	α	τ	γ	regret
1	0.20034	1.11949	0.99	76.7096
2	0.15243	1.02074	0.9	86.03815
3	0.15404	1.0012	0.9	92.8918

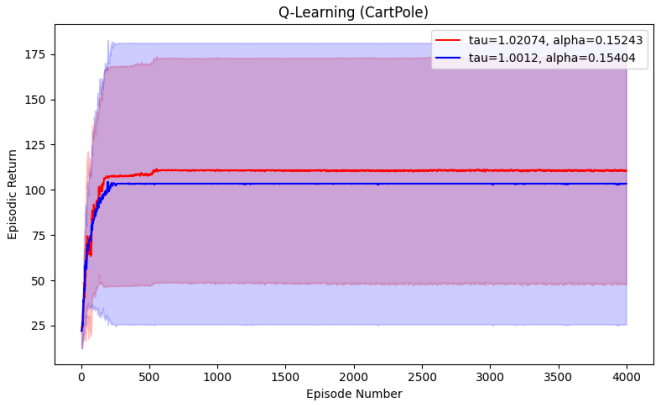
Figure 4 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

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

2.1.6 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. Hence SARSA with ϵ-greedy and Q-Learning both solves the environment and show a similar results.

2.2 MountainCar-v0

2.2.1 Code Snippets

```
            current_reward = 0
```

Listing 7: Q-table

```
def getState(state, env_low=env_low, env_high=env_high, bins=
    bins):
    """Returns the discretized position and velocity of an
    observation"""
    discretized_env = (env_high - env_low) / bins
    discretized_pos = int((state[0] - env_low[0]) /
    discretized_env[0])
    discretized_vel = int((state[1] - env_low[1]) /
    discretized_env[1])
    discretized_pos = np.clip(discretized_pos, 0, bins - 1)
    discretized_vel = np.clip(discretized_vel, 0, bins - 1)
    return discretized_pos, discretized_vel
```

Listing 8: Discretized sates

```
    """Choose action based on an epsilon greedy strategy"""
    if random.random() < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(q_table[pos][vel])
    return action
```

Listing 9: epsilon-greedy action selection

```
    print(f"\n=== Training with Seed: {seed} ===")
    np.random.seed(seed)
    random.seed(seed)
    env.reset(seed=seed)


    q_table_sarsa = np.zeros((bins + 1, bins + 1, env.
    action_space.n))

```

```
    rewards_sarsa = []

    for ep in range(episode):
        current_reward = 0
        done = False
        truncated = False
        state, _ = env.reset(seed=seed)
        pos, vel = getState(state)
        action = chooseAction(pos, vel, q_table_sarsa, epsilon)

        # while not (done or truncated):
        while not done:
            next_state, reward, done, truncated, _ = env.step(action)
            next_pos, next_vel = getState(next_state)
            next_action = chooseAction(next_pos, next_vel, q_table_sarsa, epsilon)

            if done:
                q_table_sarsa[pos][vel][action] += alpha * (reward - q_table_sarsa[pos][vel][action])
            else:
                q_table_sarsa[pos][vel][action] += alpha * (
                    reward + gamma * q_table_sarsa[next_pos][next_vel][next_action] - q_table_sarsa[pos][vel][action]
                )

            pos, vel = next_pos, next_vel
            action = next_action
            current_reward += reward
        rewards_sarsa.append(current_reward)
        print(f'seed {seed} Episode {ep+1}/{episode}, Reward: {current_reward}')
    all_rewards.append(rewards_sarsa)

env.close()
```

Listing 10: SARSA implementation

```
def softmax(q_values, temperature):
    """Returns a probability distribution over actions using softmax"""
    exp_q = np.exp(q_values / temperature)
    return exp_q / np.sum(exp_q)

def chooseAction(pos, vel, q_table, temperature):
    """Choose action based on Softmax exploration"""
    q_values = q_table[pos][vel]
    action_probs = softmax(q_values, temperature)
```

```
    action = np.random.choice(np.arange(len(q_values)), p=
action_probs)
    return action
```

Listing 11: Softmax action selection

```
    np.random.seed(seed)
    random.seed(seed)
    env.reset(seed=seed)

    q_table_qlearn = np.zeros((bins + 1, bins + 1, env.
action_space.n))
    rewards_qlearn = []

    for ep in range(episode):
        current_reward = 0
        done = False
        truncated = False
        state, _ = env.reset(seed=seed)
        pos, vel = getState(state)

        # while not (done or truncated):
        while not done:
            action = chooseAction(pos, vel, q_table_qlearn,
temperature)
            next_state, reward, done, truncated, _ = env.step(
action)
            next_pos, next_vel = getState(next_state)
            # if done or truncated:
            if done:
                q_table_qlearn[pos][vel][action] += alpha * (
reward - q_table_qlearn[pos][vel][action])
            else:
                max_next_q = np.max(q_table_qlearn[next_pos][
next_vel])
                q_table_qlearn[pos][vel][action] += alpha * (
                    reward + gamma * max_next_q -
q_table_qlearn[pos][vel][action]
                )

            pos, vel = next_pos, next_vel
            current_reward += reward

        rewards_qlearn.append(current_reward)
        print(f'Seed {seed} Episode {ep+1}/{episode}, Reward: {
current_reward}')

    all_rewards.append(rewards_qlearn)
```

Listing 12: 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 5 displays the results from 50 sweeps, illustrating the relationship between α, ϵ, and the regret.

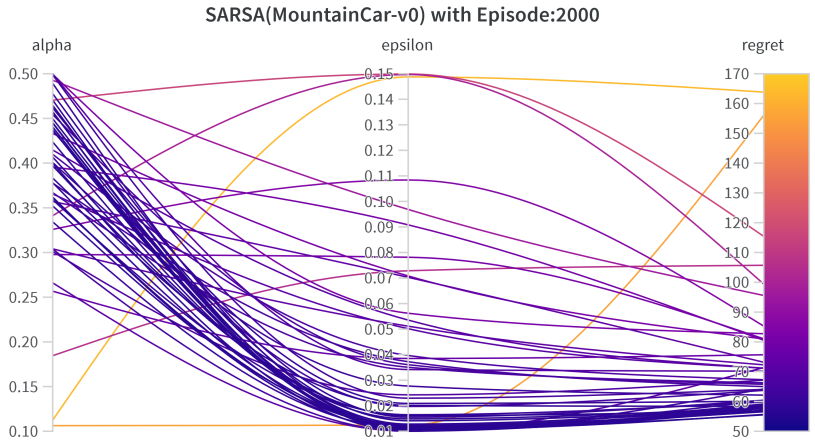
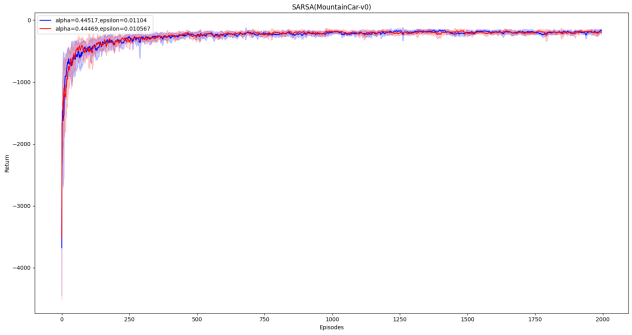


Figure 5: SARSA hyper-parameter sweeps for α and ϵ in MountainCar-v0(2000 episodes), with lines color-coded by regret.

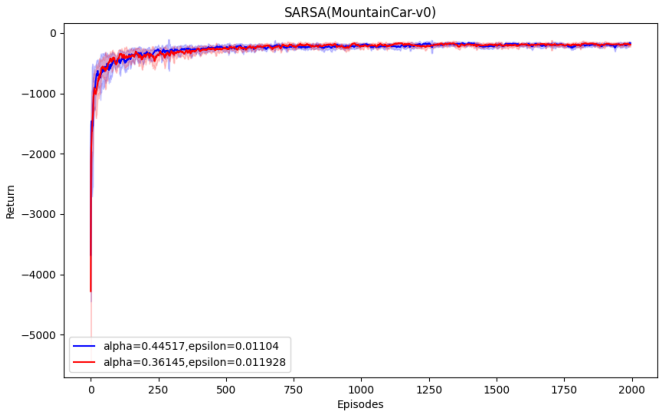
2.2.3 SARSA best 3 results

	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

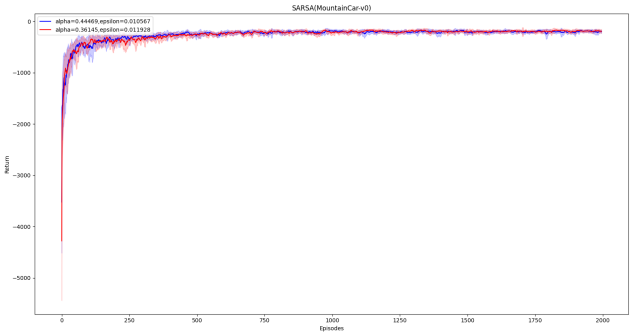
Figure 6 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 6: Performance comparison of the three best 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 7 displays the results from 50 sweeps, illustrating the relationship between α, τ, and the regret.

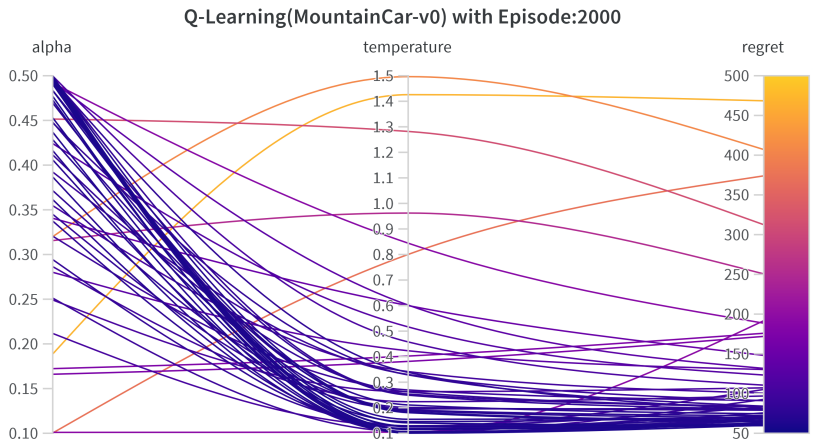
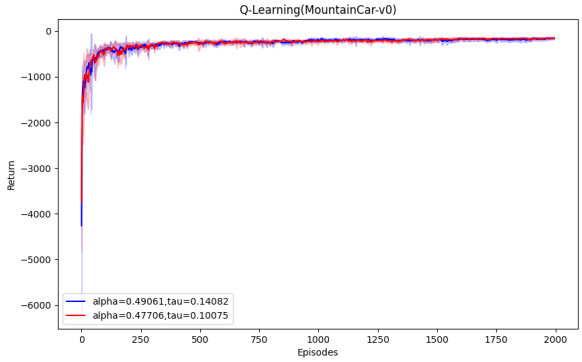


Figure 7: Q-Learning hyper-parameter sweeps for α and τ in MountainCar-v0(2000 episodes), with lines color-coded by regret.

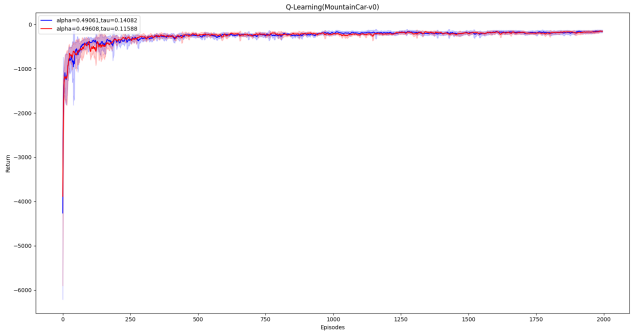
2.2.5 Q-Learning best 3 results

	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

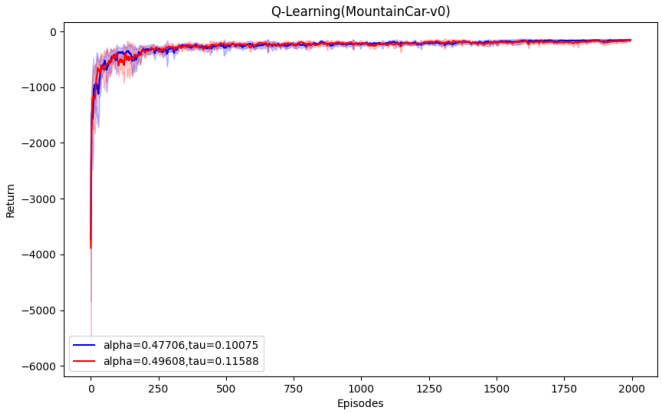
Figure 8 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 8: Performance comparison of the three best 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. Hence SARSA with ϵ-greedy and Q-Learning both solves the environment and show a similar results.

2.3 MiniGrid-Dynamic-Obstacles-5x5-v0

2.3.1 Code Snippets

Action selection using ϵ-greedy

```
def take_action(q_value, epsilon):
    if np.random.random() < epsilon:
        return np.random.randint(0, 3)
    return np.argmax(q_value)
```

Listing 13: epsilon-greedy action selection

```
        q_value = np.zeros((3, 25, 4, 2))
        total_reward = np.zeros(episodes)

        for ep in range(episodes):
            env.reset()
            terminated, truncated = False, False
            x1 = env.agent_pos[0] * 5 + env.agent_pos[1]
            x2 = env.agent_dir
            front_cell = env.grid.get(*env.front_pos)
            x3 = 1 if (front_cell and front_cell.type != "goal"
    ) else 0

            action = take_action(q_value[:, x1, x2, x3],
    epsilon)

            while not (terminated or truncated):
                observation, reward, terminated, truncated,
    info = env.step(action)
                new_x1 = env.agent_pos[0] * 5 + env.agent_pos
    [1]
                new_x2 = env.agent_dir
                front_cell = env.grid.get(*env.front_pos)
                new_x3 = 1 if (front_cell and front_cell.type
    != "goal") else 0
                new_action = take_action(q_value[:, new_x1,
    new_x2, new_x3], epsilon)

                q_value[action, x1, x2, x3] += alpha * (
```

```
                    reward + gamma * q_value[new_action, new_x1
    , new_x2, new_x3] - q_value[action, x1, x2, x3]
                )

                x1, x2, x3, action = new_x1, new_x2, new_x3,
    new_action
                total_reward[ep] += reward

            print(f"Seed: {seed} Episode: {ep+1} Reward: {
    total_reward[ep]}")

        all_rewards.append(total_reward)
        env.close()

    all_rewards = np.array(all_rewards)
    mean_rewards = np.mean(all_rewards, axis=0)
    all_time_avg_reward = np.mean(mean_rewards)
    return all_time_avg_reward, mean_rewards
```

Listing 14: SARSA implementation

```
def softmax_action(q_value, temperature):
    exp_values = np.exp(q_value / temperature)
    probabilities = exp_values / np.sum(exp_values)
    return np.random.choice(len(q_value), p=probabilities)
```

Listing 15: Softmax action selection

```
        env = gym.make('MiniGrid-Dynamic-Obstacles-Random-5x5-
    v0')
        env.reset(seed=seed)
        # Q-table dimensions: actions x (5x5 grid) x agent
    direction (4) x front cell flag (2)
        q_value = np.zeros((3, 25, 4, 2))
        total_reward = np.zeros(episodes)

        for ep in range(episodes):
            env.reset()
            terminated, truncated = False, False
            x1 = env.agent_pos[0] * 5 + env.agent_pos[1]
            x2 = env.agent_dir
            front_cell = env.grid.get(*env.front_pos)
            x3 = 1 if (front_cell and front_cell.type != "goal"
    ) else 0

            action = softmax_action(q_value[:, x1, x2, x3],
    temperature)

            while not (terminated or truncated):
```

```
                observation, reward, terminated, truncated,
    info = env.step(action)
                new_x1 = env.agent_pos[0] * 5 + env.agent_pos
    [1]
                new_x2 = env.agent_dir
                front_cell = env.grid.get(*env.front_pos)
                new_x3 = 1 if (front_cell and front_cell.type
    != "goal") else 0

                best_next_action = np.argmax(q_value[:, new_x1,
     new_x2, new_x3])
                q_value[action, x1, x2, x3] += alpha * (
                    reward + gamma * q_value[best_next_action,
    new_x1, new_x2, new_x3] - q_value[action, x1, x2, x3]
                )

                x1, x2, x3 = new_x1, new_x2, new_x3
                action = softmax_action(q_value[:, x1, x2, x3],
     temperature)
                total_reward[ep] += reward

            print(f"Seed: {seed} Episode: {ep+1} Reward: {
    total_reward[ep]}")

        all_rewards.append(total_reward)
        env.close()

    all_rewards = np.array(all_rewards)
    mean_rewards = np.mean(all_rewards, axis=0)
    all_time_avg_reward = np.mean(mean_rewards)
    return all_time_avg_reward, mean_rewards
```

Listing 16: 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 $1-$(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, ϵ, and the regret.

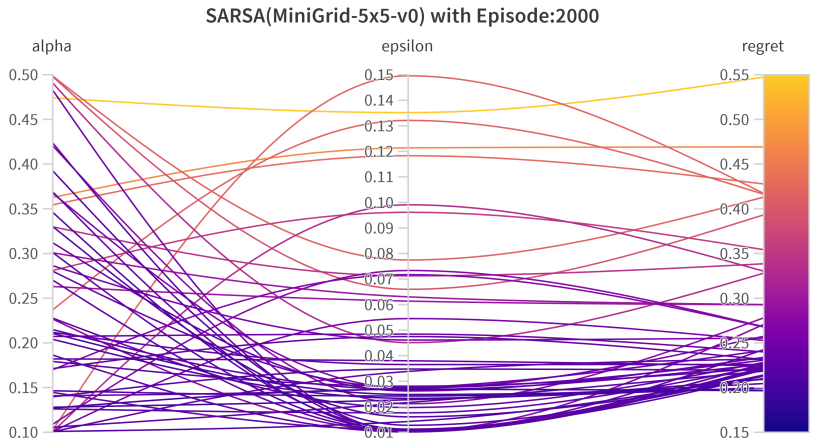
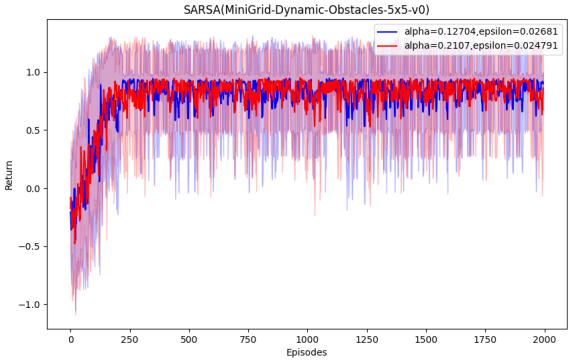


Figure 9: SARSA hyper-parameter sweeps for α and ϵ in MiniGrid-Dynamic-Obstacles-5x5-v0(2000 episodes), with lines color-coded by regret.

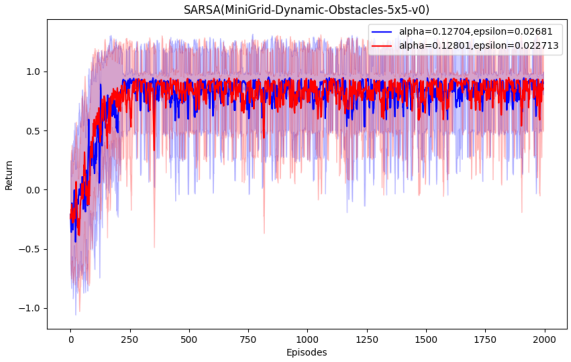
2.3.3 SARSA best 3 results

	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

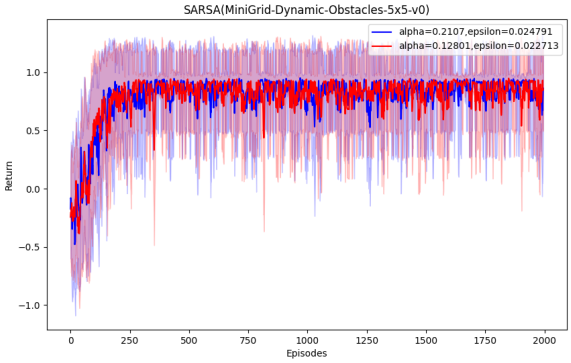
Figure 10 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 10: Performance comparison of the three best 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 $1-$ (all-time average return). See the wandb report on this environment here. Additionally, Figure 11 displays the results from 50 sweeps, illustrating the relationship between α, τ, and the regret.

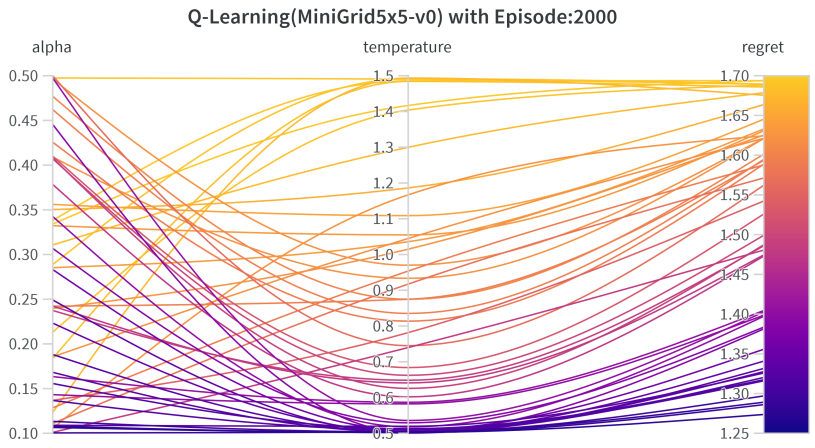
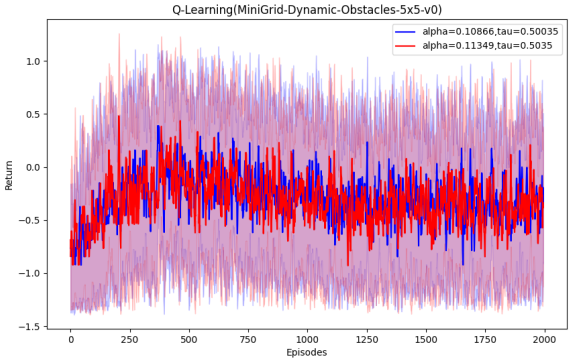


Figure 11: Q-Learning hyper-parameter sweeps for α and τ in MiniGrid-Dynamic-Obstacles-5x5-v0(2000 episodes), with lines color-coded by regret.

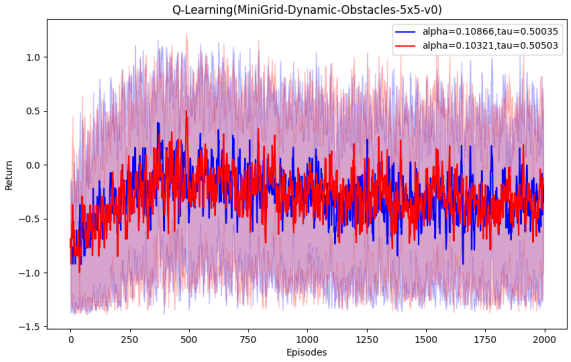
2.3.5 Q-Learning best 3 results

	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

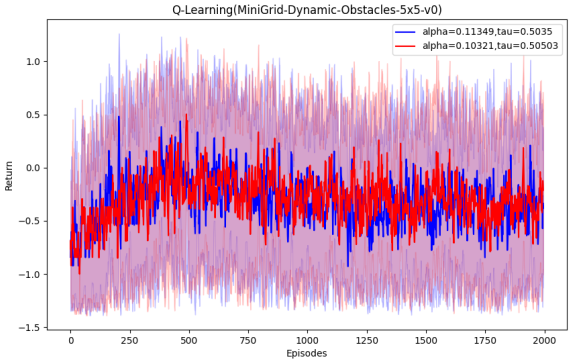
Figure 12 include the episode vs reward plot for best three hyper-parameter combination.



(a) 1 vs 2



(b) 1 vs 3



(c) 2 vs 3

Figure 12: 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. Hence SARSA with ϵ-greedy solves the environment better than Q-Learning with softmax exploration strategy.

3 Github link

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

4 Steps to recreate

1. for CartPole-v1
    - cd to cartpole-v1
    - for sarsa implementation run sarsa.py
    - for Q-Learning run qlearning.py
2. for Mountain_car-v0
    - cd mountain_car-v0
    - for sarsa implementation run sarsa.py
    - for Q-Learning implementation run q_learning.py
3. for MiniGrid-Dynamic-Obstacles-5x5-v0
    - cd minigrid_world
    - for sarsa implementation run sarsa_epsilon_greedy.py
    - for Q-Learning implementation run q_learning_softmax.py

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
```

```yaml
method: bayes
metric:
  name: regret
  goal: minimize
parameters:
  alpha:
    min: 0.1
    max: 0.5
  epsilon:
    min: 0.01
    max: 0.15
```

Listing 17: SARSA.yaml

```yaml
method: bayes
metric:
  name: regret
  goal: minimize
parameters:
  temperature:
    min: 0.5
    max: 1.5
  alpha:
    min: 0.1
    max: 0.5
```

Listing 18: Q-Learning.yaml