DA6400 : Reinforcement Learning
Programming Assignment #1
Report

Ritabrata Mandal
EE24E009

March 28, 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**$\rightarrow$ with ϵ-**greedy exploration**

- **Q-Learning**$\rightarrow$ 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 2000 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.

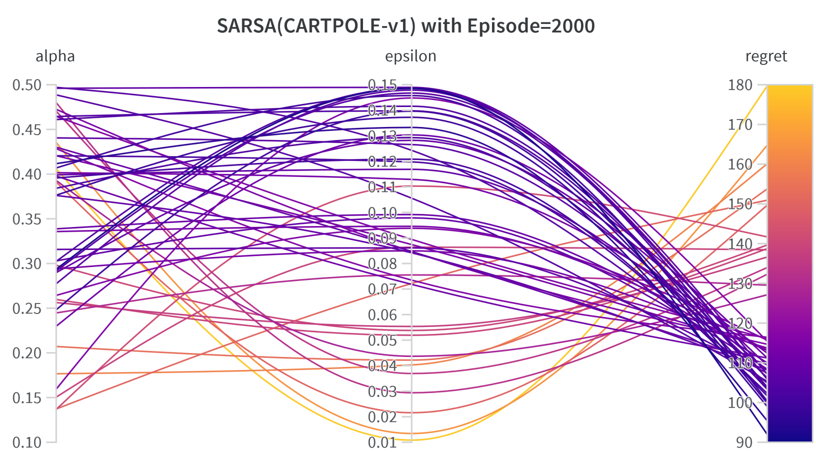

Figure 1: SARSA hyper-parameter sweeps for α and ϵ in CartPole-v1(2000 episodes), with lines color-coded by regret.

2.1.3 SARSA best 3 results

	α	ϵ	regret
1	0.29365	0.14854	91.977
2	0.4117	0.13327	95.411
3	0.3764	0.13724	98.9932

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

```
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])
    # Clip to avoid out-of-bound errors
    discretized_pos = np.clip(discretized_pos, 0, bins - 1)
    discretized_vel = np.clip(discretized_vel, 0, bins - 1)
    return discretized_pos, discretized_vel
```

Listing 1: Discretized sates

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

Listing 2: epsilon-greedy action selection

```
all_rewards = []
for seed in seeds:
    print(f"\n=== Training with Seed: {seed} ===")
    np.random.seed(seed)
    random.seed(seed)
    env.reset(seed=seed)

```

```python

    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)
```

Listing 3: SARSA implementation

```python
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"""
```

```python
    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 4: Softmax action selection

```python
for seed in seeds:
    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 5: Q-Learning implementation

2.2.2 SARSA hyper-parameter tuning

using wandb with sweep method bayes found the best performing hyper-parameter

2.2.3 SARSA best 3 results

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

2.2.4 Q-Learning hyper-parameter tuning

2.2.5 Q-Learning best 3 results

	α	τ	regret
1			
2			
3			

2.2.6 Result(SARSA vs Q-Learning)

2.3 MiniGrid-Dynamic-Obstacles-5x5-v0

2.3.1 Code Snippets

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

Listing 6: epsilon-greedy action selection

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

```python
        total_reward = np.zeros(episodes)

        for ep in range(episodes):
            env.reset()
            terminated, truncated = False, False
            # Calculate state indices
            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)

                # SARSA update rule
                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()
```

Listing 7: SARSA implementation

```python
def softmax_action(q_value, temperature):
    """Return an action selected via softmax exploration."""
    exp_values = np.exp(q_value / temperature)
    probabilities = exp_values / np.sum(exp_values)
```

```python
    return np.random.choice(len(q_value), p=probabilities)
```

Listing 8: Softmax action selection

```python
    for seed in seeds:
        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
            # Compute state indices
            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

            # Select action using softmax
            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

                # Q-learning update with softmax exploration
                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]
                )

                # Move to next state
                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()
```

Listing 9: Q-Learning implementation

2.3.2 SARSA hyper-parameter tuning

2.3.3 SARSA best 3 results

	α	ϵ	regret
1			
2			
3			

2.3.4 Q-Learning hyper-parameter tuning

2.3.5 Q-Learning best 3 results

	α	τ	regret
1			
2			
3			

2.3.6 Result(SARSA vs Q-Learning)

3 Conclusion

4 Github link

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

5 References