DA6400 : Reinforcement Learning
Programming Assignment #1

Ritabrata Mandal
EE24E009

March 25, 2025

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 above 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 SARSA Code Snippets:

```python
import numpy as np
from q_table import discretize_state
from policy import EpsilonGreedyPolicy

class SarasLearner:
    """
        Description: This class implements the SARSA algorithm.
        Args:
            alpha     : The learning rate.
            gamma     : The discount factor.
            epsilon   : The exploration rate.
            q_table   : The q-table for the cartpole-v1 environment.
            bins      : The bins for discretizing the state space.
            env       : The cartpole-v1 environment.
    """
    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):
        """
            Description: This function computes the TD error.
            Args:
                state       : The current state.
                action      : The current action.
                next_state  : The next state.
                reward      : The reward.
            Returns:
                td_error    : The TD error.
        """
        return reward + self.gamma * self.q_table[next_state[0], next_state[1], next_state[2], next_state[3],
```

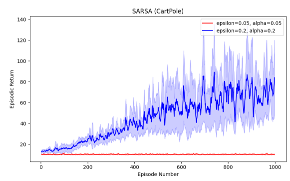
```python
next_action] - \
            self.q_table[state[0], state[1], state[2], state[3], action]

    def update_q_table(self, state, action, td_error):
        """
            Description: This function updates the q-table.
            Args:
                state       : The current state.
                action      : The current action.
                td_error    : The TD error.
                Equations:
                Q(s, a) <- Q(s, a) + alpha * (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)  # SARSA selects initial action
            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  # SARSA moves (s, a) -> (s', a')
                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
```

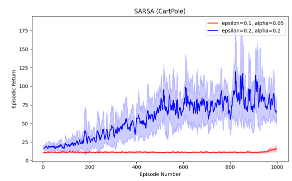
Listing 1: SARSA - Agent

2.1.2 SARSA Runs

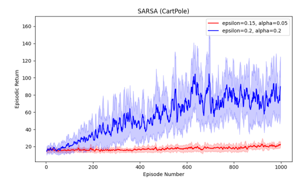
below are the graphs of runs of SARSA with different hyper-parameter



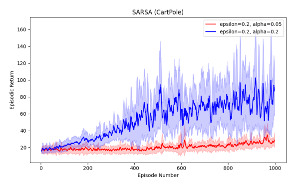
(a) $\alpha = 0.05$



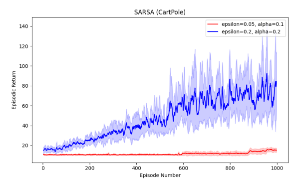
(b) $\epsilon = 0.1$



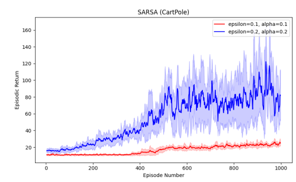
(c) $\epsilon = 0.15$



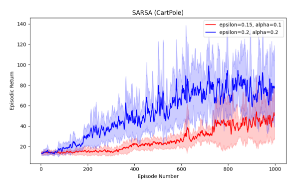
(d) $\epsilon = 0.2$



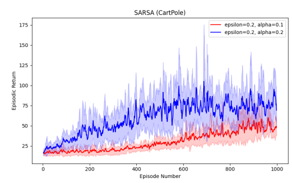
(e) $\epsilon = 0.25$



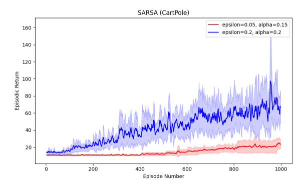
(f) $\epsilon = 0.3$



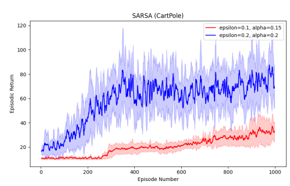
(g) $\epsilon = 0.35$



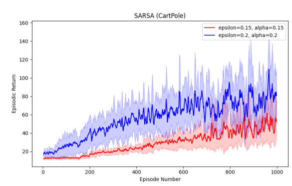
(h) $\epsilon = 0.4$



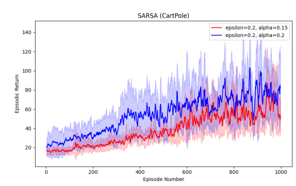
(i) $\epsilon = 0.45$



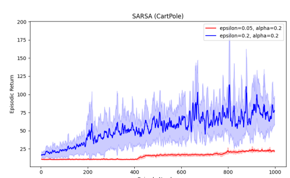
(j) $\epsilon = 0.5$



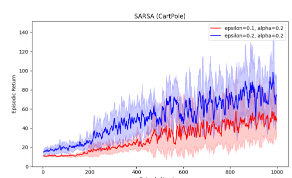
(k) $\epsilon = 0.55$



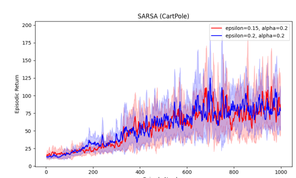
(l) $\epsilon = 0.6$



(m) $\epsilon = 0.65$



(n) $\epsilon = 0.7$



(o) $\epsilon = 0.75$

Figure 1: Phase portraits for different values of ϵ