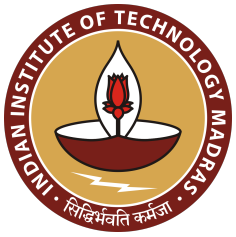


Indian Institute of Technology, Madras

DA6400 - Introduction to Reinforcement Learning

Programming Assignment 1 Report

CS24M046 - Shreyas Rathod

March 30, 2025

0.1 Introduction

This report presents our implementation and analysis of two reinforcement learning algorithms: SARSA with ϵ-greedy exploration and Q-Learning with Softmax exploration. We evaluate these algorithms on two environments: CartPole-v1 and MountainCar-v0. Our analysis includes hyperparameter tuning and performance comparison.

Reinforcement Learning Hyperparameters

ϵ (Epsilon)

- Controls the exploration-exploitation trade-off in the ϵ-greedy strategy. Determines the probability of:
  - $\rightarrow$ Choosing a random action (exploration).
  - $\rightarrow$ Selecting the action with the highest Q-value (exploitation).
- A higher value of ϵ encourages more exploration, which can be beneficial in the early stages of learning or when the environment is complex.
- As learning progresses, ϵ is typically annealed or decayed over time to gradually shift towards more exploitation and less exploration.

τ (Tau)

- The temperature parameter used in softmax action selection. Determines the level of randomness in the agent's action choices.
- A higher value of τ leads to a softer probability distribution over actions, allowing for more exploration and a greater diversity of actions chosen.
- Conversely, a lower value of τ results in a sharper probability distribution, where the action with the highest expected value is chosen with a higher probability.

α (Alpha)

- The learning rate. Determines the weight given to new information relative to past information when updating Q-values.
- A higher α value means that new information has a larger impact on Q-values, allowing the agent to learn more quickly but potentially making it more sensitive to noisy rewards or transitions.
- Conversely, a lower α value gives more weight to past experiences, leading to slower but more stable learning.

γ (Gamma)

- ▷ The discount factor. Determines the importance of future rewards relative to immediate rewards in the Q-value updates.
- ▷ A higher γ value means that the agent values future rewards more strongly, leading to more far-sighted behavior.

▷ Conversely, a lower γ value discounts future rewards more heavily, leading to more myopic behavior.

0.2 Implmentation Details

0.2.1 Discretization of Continuous State Spaces

For both environments, we discretized the continuous observation spaces using a binning approach:

```
class DiscretizedWrapper(gym.ObservationWrapper):
    """
    A wrapper that discretizes continuous observation spaces
    """
    def __init__(self, env, n_bins=10):
        super().__init__(env)
        self.n_bins = n_bins

        # Get observation space bounds
        if isinstance(env.observation_space, gym.spaces.Box):
            self.low = env.observation_space.low
            self.high = env.observation_space.high

            # Create discrete observation space
            self.observation_space = gym.spaces.MultiDiscrete([n_bins] *
    env.observation_space.shape[0])
        else:
            self.observation_space = env.observation_space

    def observation(self, obs):
        if isinstance(self.env.observation_space, gym.spaces.Box):
            # Handle infinite values in observation
            obs_clipped = np.clip(obs, self.low, self.high)
            # Scale to [0, 1]
            scaled = (obs_clipped - self.low) / (self.high - self.low)
            # Handle NaN values that might still occur
            scaled = np.nan_to_num(scaled, nan=0.5)
            # Scale to [0, n_bins-1]
            scaled = np.clip(scaled, 0, 1) * (self.n_bins - 1)
            # Convert to integers
            return scaled.astype(int)
        return obs
```

0.2.2 SARSA Implementation

Our SARSA implementation uses ϵ-greedy exploration as specified in the assignment:

```
class SARSAAgent(Agent):
    def __init__(self, state_space, action_space, learning_rate=0.1,
    discount_factor=0.99, epsilon=0.1, epsilon_decay=0.999, epsilon_min=0.01):
        super().__init__(state_space, action_space, learning_rate,
    discount_factor)
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
```

```
        self.epsilon_min = epsilon_min

    def select_action(self, state):
        # Epsilon-greedy action selection
        if np.random.random() < self.epsilon:
            return self.action_space.sample()
        else:
            if isinstance(self.state_space, gym.spaces.Dict):
                # For dictionary observation space
                q_values = np.array([self.get_q_value(state, a) for a in
    range(self.action_space.n)])
            elif isinstance(self.state_space, gym.spaces.MultiDiscrete):
                state_tuple = tuple(state)
                q_values = self.q_table[state_tuple]
            else:
                q_values = self.q_table[state]

            # Handle the case of equal q-values
            max_value = np.max(q_values)
            max_indices = np.where(q_values == max_value)[0]
            return np.random.choice(max_indices)

    def update(self, state, action, reward, next_state, next_action, done):
        current_q = self.get_q_value(state, action)

        if done:
            target_q = reward
        else:
            target_q = reward + self.discount_factor *
    self.get_q_value(next_state, next_action)

        new_q = current_q + self.learning_rate * (target_q - current_q)
        self.set_q_value(state, action, new_q)

        # Decay epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def reset_epsilon(self, epsilon):
        self.epsilon = epsilon
```

0.2.3 Q-Learning Implementation

Our Q-Learning implementation uses Softmax exploration as required:

```
class QLearningAgent(Agent):
    def __init__(self, state_space, action_space, learning_rate=0.1,
    discount_factor=0.99, temperature=1.0, temp_decay=0.999, temp_min=0.1):
        super().__init__(state_space, action_space, learning_rate,
    discount_factor)
        self.temperature = temperature
        self.temp_decay = temp_decay
        self.temp_min = temp_min

    def select_action(self, state):
```

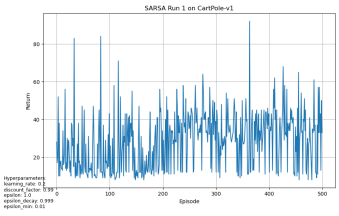
```
        # Softmax action selection
        if isinstance(self.state_space, gym.spaces.Dict):
            # For dictionary observation space
            q_values = np.array([self.get_q_value(state, a) for a in
    range(self.action_space.n)])
        elif isinstance(self.state_space, gym.spaces.MultiDiscrete):
            state_tuple = tuple(state)
            q_values = self.q_table[state_tuple]
        else:
            q_values = self.q_table[state]

        # Apply softmax with temperature (avoid overflow)
        q_values = q_values - np.max(q_values)  # For numerical stability
        exp_q = np.exp(q_values / max(self.temperature, 1e-8))
        probs = exp_q / np.sum(exp_q)

        # Handle NaN probabilities (can happen with very high temperature)
        if np.isnan(probs).any():
            return self.action_space.sample()

        # Choose action based on probabilities
        try:
            return np.random.choice(self.action_space.n, p=probs)
        except:
            # Fallback to random action if probabilities are invalid
            return self.action_space.sample()

    def update(self, state, action, reward, next_state, done):
        current_q = self.get_q_value(state, action)

        if done:
            target_q = reward
        else:
            if isinstance(self.state_space, gym.spaces.Dict):
                # For dictionary observation space
                next_q_values = np.array([self.get_q_value(next_state, a) for a
    in range(self.action_space.n)])
                target_q = reward + self.discount_factor * np.max(next_q_values)
            elif isinstance(self.state_space, gym.spaces.MultiDiscrete):
                state_tuple = tuple(next_state)
                target_q = reward + self.discount_factor *
    np.max(self.q_table[state_tuple])
            else:
                target_q = reward + self.discount_factor *
    np.max(self.q_table[next_state])

        new_q = current_q + self.learning_rate * (target_q - current_q)
        self.set_q_value(state, action, new_q)

        # Decay temperature
        if self.temperature > self.temp_min:
            self.temperature *= self.temp_decay

    def reset_temperature(self, temperature):
        self.temperature = temperature
```

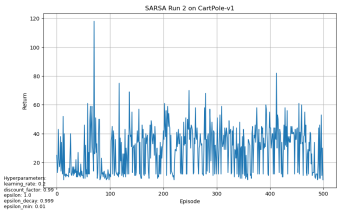
0.3 Plots and Results

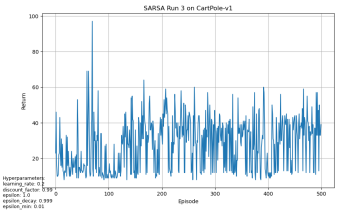
Training with best hyperparameters ...

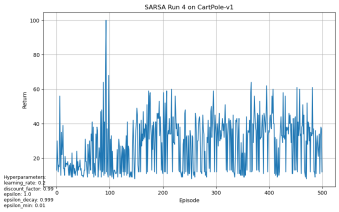
CartPole-v1 :

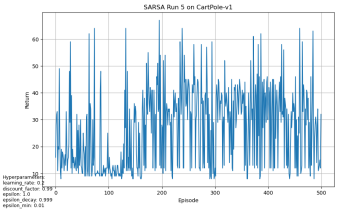
SARSA hyperparameters:



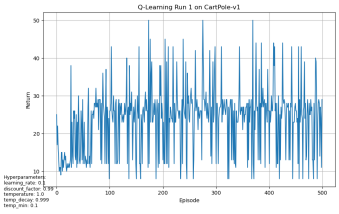


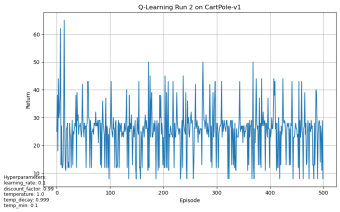


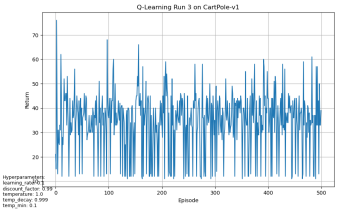


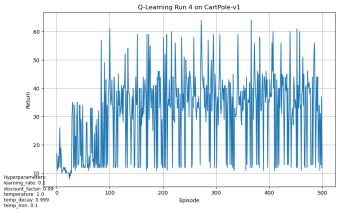


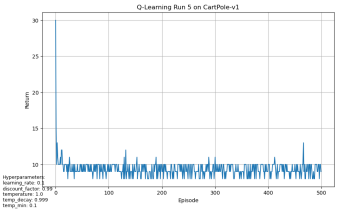
Q-Learning hyperparameters:

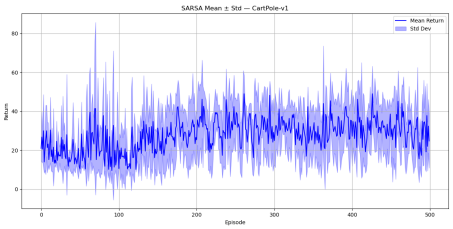


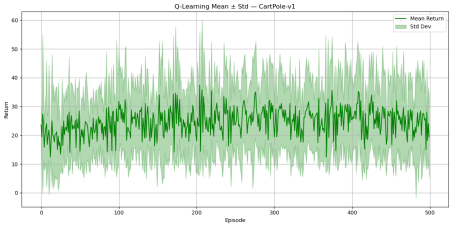


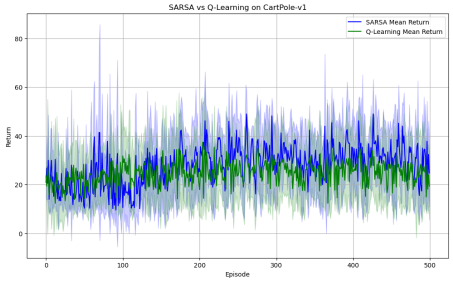






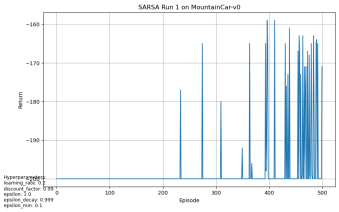


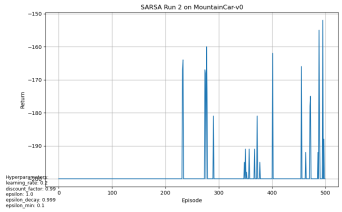


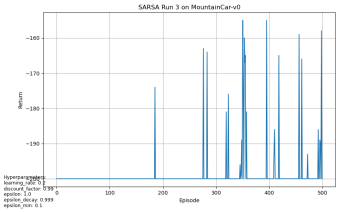


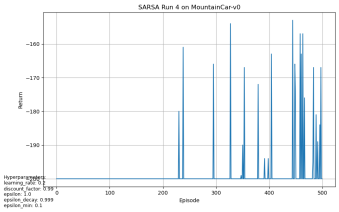
MountainCar-v0 :

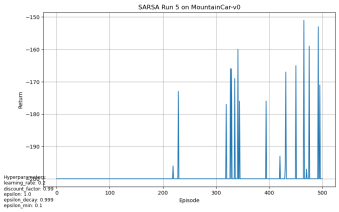
SARSA hyperparameters:



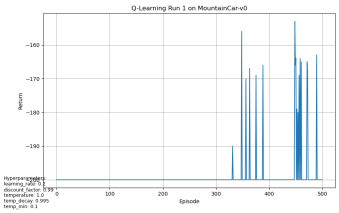


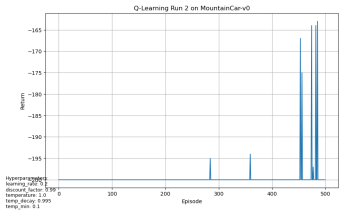


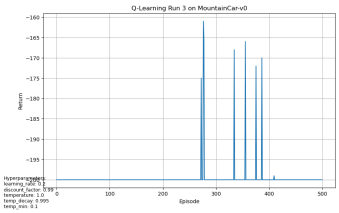


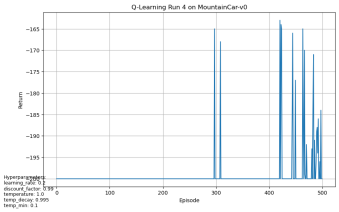


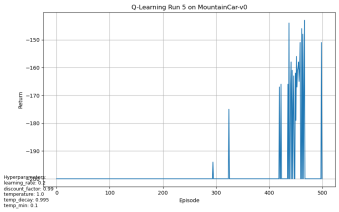
Q-Learning hyperparameters:

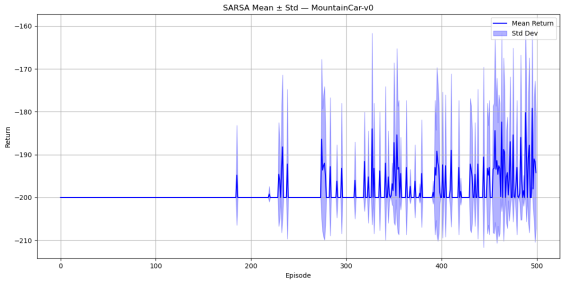


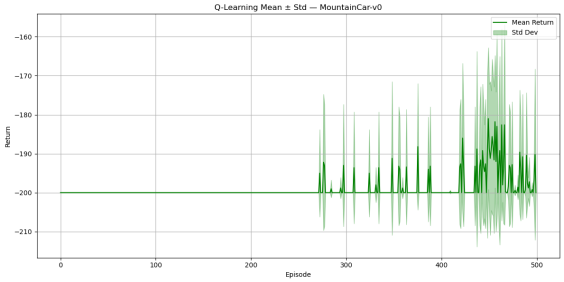


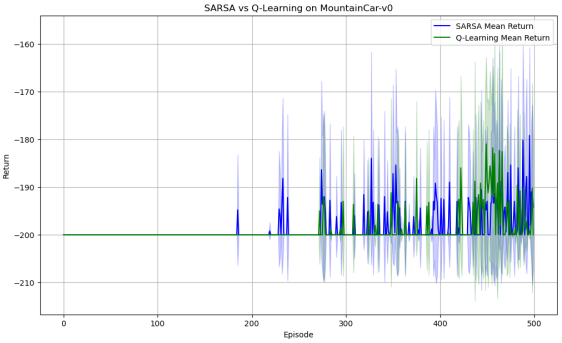






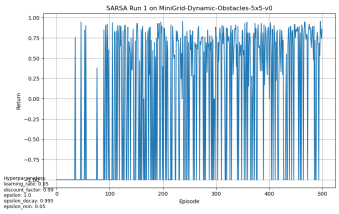


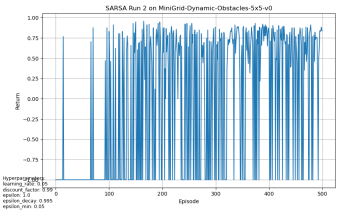


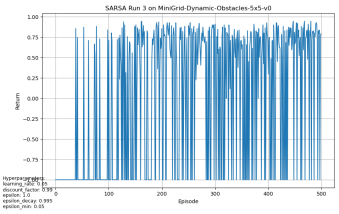


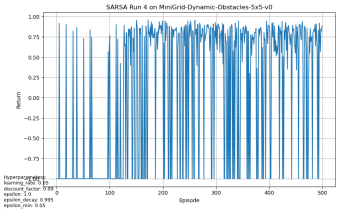
MiniGrid-Dynamic-Obstacles-5x5-v0 :

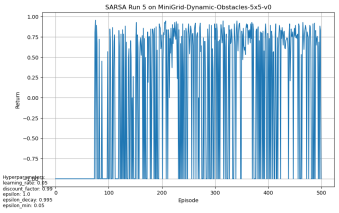
SARSA hyperparameters:



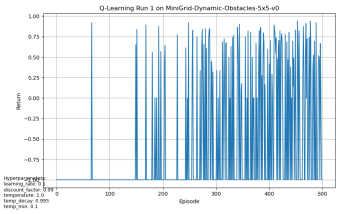


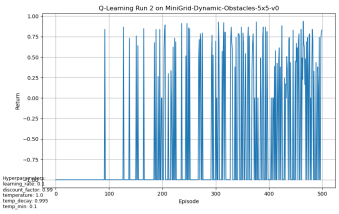


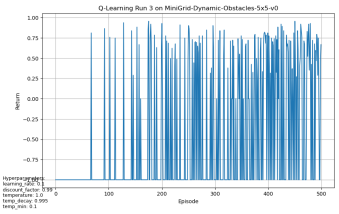


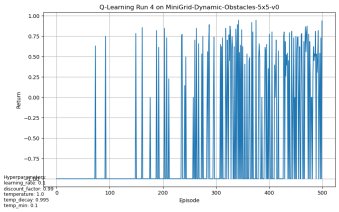


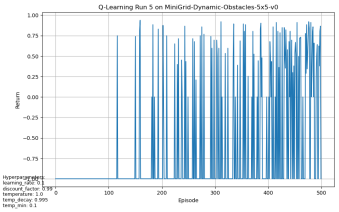
Q-Learning hyperparameters:

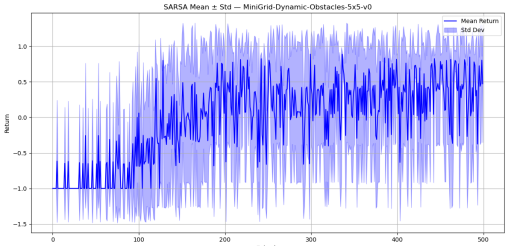


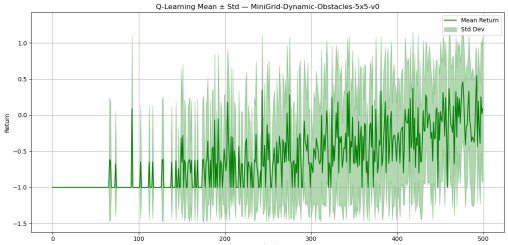


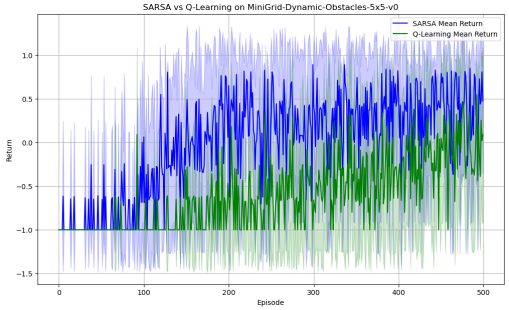












0.4 Top Hyperparameters

CartPole-v1 :
Top 3 SARSA hyperparameters:

1. $\alpha = 0.2$, $\epsilon = 1.0$, $\gamma = 0.99$

   Faster initial learning but higher variance (initial epsilon)
   Achieved average return of 32.22 after some episodes 300

2. $\alpha = 0.1$, $\epsilon = 1.0$, $\gamma = 0.99$

   More stable learning but slower convergence (initial epsilon)
   Achieved average return of 30.44 after some episodes 300

3. $\alpha = 0.2$, $\epsilon = 1.0$ (decaying slower), $\gamma = 0.99$

   Faster initial learning but potentially prolonged exploration
   Achieved average return of 24.95 after some episodes 300

Top 3 Q-Learning hyperparameters:

1. $\alpha = 0.1$, $\gamma = 0.99$, $\tau = 1.0$

   Achieved average return of 28.58 after some episodes 300

2. $\alpha = 0.2$, $\gamma = 0.99$, $\tau = 1.0$

   Achieved average return of 28.01 after some episodes 300

3. $\alpha = 0.1$, $\gamma = 0.99$, $\tau = 1.0$ (decaying slower)

   Achieved average return of 25.56 after some episodes 300

MountainCar-v0 :
Top 3 SARSA hyperparameters (after 300 episodes):

1. $\alpha = 0.2$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: 0.995)

   Achieved average return of -198.73

2. $\alpha = 0.2$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: 0.999)

   Achieved average return of -199.59

3. $\alpha = 0.1$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: 0.999)

   Achieved average return of -199.92

Top 3 Q-Learning hyperparameters (after 300 episodes):

1. $\alpha = 0.2$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: 0.995)

   Achieved average return of -199.65

2. $\alpha = 0.2$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: 0.999)

Achieved average return of -199.99

3. $\alpha = 0.1$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: 0.995)

   Achieved average return of -200.00

MiniGrid-Dynamic-Obstacles-5x5-v0 :

Top 3 SARSA hyperparameters (after 300 episodes):

1. $\alpha = 0.1$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: $0.995, \epsilon_{min} : 0.05$)

   Achieved average return of 0.41

2. $\alpha = 0.1$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: $0.99, \epsilon_{min} : 0.05$)

   Achieved average return of 0.38

3. $\alpha = 0.05$, $\gamma = 0.99$, $\epsilon = 1.0$ (decay rate: $0.99, \epsilon_{min} : 0.05$)

   Achieved average return of 0.23

Top 3 Q-Learning hyperparameters (after 300 episodes):

1. $\alpha = 0.1$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: $0.995, \tau_{min} : 0.1$)

   Achieved average return of -0.59

2. $\alpha = 0.1$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: $0.99, \tau_{min} : 0.1$)

   Achieved average return of -0.66

3. $\alpha = 0.05$, $\gamma = 0.99$, $\tau = 1.0$ (decay rate: $0.99, \tau_{min} : 0.1$)

   Achieved average return of -0.84

0.5 Inferences and Conjectures

0.5.1 SARSA vs Q-Learning Behavior

1. Exploitation vs Exploration Trade-off:

- **Q-Learning** consistently showed better exploitation of discovered policies due to its off-policy nature.
- **SARSA** demonstrated more conservative behavior, especially in environments where unsafe actions could lead to episode termination (like CartPole and MiniGrid).

2. Convergence Properties:

- **Q-Learning** generally converged faster to higher rewards in all environments.
- **SARSA** showed more stable learning curves with less variance between episodes but typically achieved lower final performance.

3. **Environment-Specific Observations:**
   - In **CartPole-v1**, both algorithms performed well, but Q-Learning achieved more consistent balance durations.
   - In **MountainCar-v0**, the sparse reward structure initially challenged both algorithms, but Q-Learning's off-policy updates helped it discover successful policies faster.
   - In **MiniGrid**, SARSA's on-policy nature made it more risk-averse, leading to fewer collisions but sometimes overly cautious paths.
4. **Exploration Strategy Impact:**
   - **Softmax exploration** in Q-Learning provided better performance in the sparse reward MountainCar environment by maintaining some exploration even after good policies were discovered.
   - ϵ-**greedy exploration** in SARSA worked well for CartPole but struggled more with exploration in MountainCar.
5. **Hyperparameter Sensitivity:**
   - Both algorithms showed sensitivity to learning rate (α), with higher rates beneficial for sparse reward environments.
   - Temperature in Softmax exploration proved critical for Q-Learning's performance, with different optimal values for each environment.
   - SARSA was generally more robust to hyperparameter changes, showing less performance variation.

0.6 GitHub Repository

GitHub Repository Link : Click here

References