DA6400: Reinforcement Learning
Programming Assignment 1 Report

Kush Shah CH21B053 Sai Teja MS ME21B171

March 30, 2025

Abstract

This report describes the implementation and experimental evaluation of SARSA (with ϵ-greedy exploration) and Q-Learning (with Softmax exploration) on the Gymnasium environments: CartPole-v1 and MountainCar-v0. We compare each algorithm with its own variant over 5 random seeds and report the mean and variance of the episodic returns. Hyperparameter tuning results and inferences are also discussed.

1 Introduction

In this assignment, we implemented two tabular RL algorithms: SARSA and Q-Learning. The primary goal was to train these algorithms on various environments (CartPole-v1, MountainCar-v0, and optionally MiniGrid-Dynamic-Obstacles-5x5-v0) and evaluate their performance. We adhered to the following guidelines:

- Use $\gamma = 0.99$ for all experiments.
- Tune hyperparameters to minimize regret.
- Use 5 random seeds per experiment and plot the mean and variance of episodic returns.

2 Environment Descriptions

2.1 CartPole-v1

CartPole-v1 consists of a cart moving along a frictionless track with a pole attached by an un-actuated joint. The goal is to balance the pole by applying forces left or right.

2.2 MountainCar-v0

MountainCar-v0 involves a car positioned in a valley with a sinusoidal track. The objective is to drive the car to the top of the right hill by carefully accelerating left or right.

2.3 MiniGrid-Dynamic-Obstacles-5x5-v0 (Bonus)

This environment is an empty room with moving obstacles. The agent must reach a goal square without colliding with obstacles.

3 Algorithm Implementations

We implemented SARSA and Q-Learning in separate Python modules. Below are snippets of the key parts of our code.

3.1 SARSA Implementation

```python
# Update rule for SARSA
Q[tuple(state)+(action,)] += alpha * (reward + gamma * Q[tuple(next_state)+(
    next_action,)] - Q[tuple(state)+(action,)])
```

Listing 1: Key SARSA Update Code

3.2 Our SARSA Code Implementation

```python
def sarsa(env, Q, episodes, gamma=0.99, alpha=0.1, epsilon=1.0, epsilon_decay
    =0.995, epsilon_min=0.05, print_freq=100, plot_heat=False, choose_action=
    choose_action_epsilon):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)


    for ep in tqdm(range(episodes)):
        tot_reward = 0
        steps = 0

        # Reset the environment and get the initial discretized state.
        state = env.reset()

        action = choose_action(Q, state, epsilon)

        done = False
        while not done:
            # Take a step in the environment.
            state_next, reward, done = env.step(action)

            # Select next action using the same policy.
            next_action = choose_action(Q, state_next, epsilon)

            # Sarsa update
            Q[tuple(state) + (action,)] += alpha * (reward + gamma * Q[tuple(
                state_next) + (next_action,)] - Q[tuple(state) + (action,)])

            tot_reward += reward
            steps += 1

            state, action = state_next, next_action

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep + 1) % print_freq == 0:
            avg_reward = np.mean(episode_rewards[ep-print_freq+1:ep+1])
            avg_steps = np.mean(steps_to_completion[ep-print_freq+1:ep+1])
            print(f"Episode {ep+1}: Avg Reward: {avg_reward:.2f}, Avg Steps: {
                avg_steps:.2f}, Qmax: {Q.max():.2f}, Qmin: {Q.min():.2f}")

        # Decay epsilon after each episode
        epsilon = max(epsilon * epsilon_decay, epsilon_min)

    return episode_rewards, steps_to_completion
```

Listing 2: Our SARSA Code Implementation

3.3 Q-Learning Implementation

```
# Update rule for Q-Learning
best_next_value = np.max(Q[tuple(next_state)])
Q[tuple(state)+(action,)] += alpha * (reward + gamma * best_next_value - Q[
    tuple(state)+(action,)])
```

Listing 3: Key Q-Learning Update Code

3.4 Our Q-Learning Code Implementation

```
def q_learning(env, Q, episodes, gamma=0.99, alpha=0.1,
               tau=1.0, tau_decay=0.995, tau_min=0.1,
               print_freq=100, plot_heat=False, choose_action=
                   choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)

    for ep in tqdm(range(episodes), desc="Q-learning"):
        tot_reward = 0
        steps = 0

        # Reset the environment and get the initial discretized state.
        state = env.reset()
        action = choose_action(Q, state, tau)
        done = False

        while not done:
            # Take a step in the environment.
            state_next, reward, done = env.step(action)

            # Q-learning update
            best_next_value = np.max(Q[tuple(state_next)])
            Q[tuple(state) + (action,)] += alpha * (reward + gamma *
                best_next_value - Q[tuple(state) + (action,)])

            tot_reward += reward
            steps += 1

            # Update state and select next action.
            state = state_next
            action = choose_action(Q, state, tau)

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep + 1) % print_freq == 0:
            avg_reward = np.mean(episode_rewards[ep - print_freq + 1 : ep + 1])
            avg_steps = np.mean(steps_to_completion[ep - print_freq + 1 : ep +
                1])
            print(f"Episode {ep+1}: Avg Reward: {avg_reward:.2f}, Avg Steps: {
                avg_steps:.2f}, Qmax: {Q.max():.2f}, Qmin: {Q.min():.2f}")

        tau = max(tau * tau_decay, tau_min)

    return episode_rewards, steps_to_completion
```

Listing 4: Our Q-Learning Implementation

3.5 Policy Functions

We used two policy functions:

- `choose_action_epsilon`: for ϵ-greedy exploration.
- `choose_action_softmax`: for softmax (Boltzmann) exploration.

```python
def choose_action_epsilon(Q, state, epsilon, rg=rg):
    q_values = Q[tuple(state)]
    # If all Q-values are zero or with probability epsilon, choose a random
        action
    if not q_values.any() or rg.rand()<epsilon:
        return rg.choice(Q.shape[-1])
    else:
        return np.argmax(q_values)

def choose_action_softmax(Q, state, tau, rg=rg):
    q_values = Q[tuple(state)]
    probs = softmax(q_values - np.max(q_values)) ** (1/tau)
    probs = probs / np.sum(probs)
    return rg.choice(Q.shape[-1], p=probs)
```

Listing 5: Our policy Function Code

4 Experimental Setup

We discretized the continuous state space using a custom function `create_bins()` and `discretize_state()` (see `utils.py`). The Q-table is built based on the number of discretization bins per state dimension.

4.1 State Discretization

To apply tabular reinforcement learning methods, we discretized the continuous state space using the following helper functions:

```python
def create_bins(env, n_bins):
    """
    Create bins for each state variable. This is required for discretizing the
        state for sarsa and q-learning.
    """
    if env.spec.id == "CartPole-v1":
        bins = []
        bins.append(np.linspace(-4.8, 4.8, n_bins))     # Cart position
        bins.append(np.linspace(-3.0, 3.0, n_bins))     # Cart velocity
        bins.append(np.linspace(-0.418, 0.418, n_bins))# Pole angle
        bins.append(np.linspace(-3.5, 3.5, n_bins))     # Pole angular velocity
        return bins
    else:
        obs_low = env.observation_space.low
        obs_high = env.observation_space.high
        bins = []
        for i in range(len(obs_low)):
            bins.append(np.linspace(obs_low[i], obs_high[i], n_bins))
        return bins

def discretize_state(state, bins):
    """
    Discretize the state.
    """
    indices=[]
    for i, val in enumerate(state):
        # Subtract 1 as digitize is 1-indexed
        index = np.digitize(val, bins[i]) - 1
        # Ensure the index is within the bounds
        index = min(max(index, 0), len(bins[i]) - 2)
        indices.append(index)
    return tuple(indices)
```

Listing 6: State Discretization Functions

The function `create_bins()` creates bins for each state variable, allowing us to convert continuous state values into discrete indices. The function `discretize_state()` maps an observed state to its corresponding discretized index using the bins. This discretization enables us to use a Q-table for SARSA and Q-learning.

Experiments were run over 5 random seeds. For each experiment, we recorded the episodic return and computed the mean and standard deviation across seeds.

5 Results and Plots

The main() function code is `run_experiment.py` is:

```
def main():
    parser = argparse.ArgumentParser(description="Run RL algorithms on a Gym
        environment.")
    parser.add_argument("--env", type=str, default="MountainCar-v0",
                        help="Gym environment to run (e.g., MountainCar-v0,
                            CartPole-v1)")
    parser.add_argument("--algorithm", type=str, default="sarsa",
                        choices=["sarsa", "qlearning", "both"],
                        help="Algorithm to run: sarsa, qlearning, or both")
    parser.add_argument("--episodes", type=int, default=12000, help="Number of
        episodes to run")
    parser.add_argument("--n_bins", type=int, default=20, help="Number of bins
        per state dimension")
    parser.add_argument("--tau_decay", type=float, default=0.999, help="
        Tau_decay")
    parser.add_argument("--epsilon_decay", type=float, default=0.995, help="
        Epsilon_decay")
    parser.add_argument("--alpha", type=float, default=0.1, help="Learning rate
        ")
    args = parser.parse_args()

    seeds = [42, 43, 44, 45, 46]
    gamma = 0.99

    epsilon = 1.0
    epsilon_min = 0.05

    # For softmax exploration (Q-learning), use tau (temperature)
    tau = 2.0
    tau_min = 0.1

    if args.algorithm in ["sarsa", "both"]:
        print("Running SARSA with epsilon-greedy exploration on", args.env)
        sarsa_rewards = run_experiment(args.env, sarsa, choose_action_epsilon,
            args.episodes, seeds,
                                       gamma, args.alpha, epsilon, epsilon_decay=args.
                                           epsilon_decay,
                                       epsilon_min=epsilon_min, n_bins=args.n_bins)
        plot_results(sarsa_rewards, f"SARSA on {args.env}")

    if args.algorithm in ["qlearning", "both"]:
        print("Running Q-Learning with softmax exploration on", args.env)
        q_rewards = run_experiment(args.env, q_learning, choose_action_softmax,
             args.episodes, seeds,
                              gamma, args.alpha, epsilon, tau=tau, tau_decay=args.
                                  tau_decay, tau_min=tau_min,
                              n_bins=args.n_bins)
        plot_results(q_rewards, f"Q-Learning on {args.env}")
```

The `run_experiment` function is implemented as follows:

```
def run_experiment(env_name, algorithm, policy_fn, episodes, seeds, gamma,
    alpha,
                   epsilon, epsilon_decay=None, epsilon_min=None,
                   tau=None, tau_decay=None, tau_min=None, n_bins=20):
    all_rewards = np.zeros((len(seeds), episodes))
    print_freq = 100

    for i, seed in enumerate(seeds):
        env = gym.make(env_name)
        env.reset(seed=seed)

        bins = create_bins(env, n_bins)
        # Discretize the environment
        d_env = DiscretizedEnv(env, bins)

        n_actions = env.action_space.n
        # Initialize Q-table
        state_shape = (n_bins-1,)*len(env.observation_space.low)
        Q = np.zeros(state_shape + (n_actions,))

        if policy_fn == choose_action_epsilon:
            rewards, _ = algorithm(d_env, Q, episodes, gamma=gamma, alpha=alpha
                , epsilon=epsilon,
                                   epsilon_decay=epsilon_decay, epsilon_min=
                                       epsilon_min,
                                   print_freq=print_freq, choose_action=
                                       policy_fn)
        else:
            rewards, _ = algorithm(d_env, Q, episodes, gamma=gamma, alpha=alpha
                , tau=tau,
                                   tau_decay=tau_decay, tau_min=tau_min,
                                   print_freq=print_freq, choose_action=
                                       policy_fn)
        all_rewards[i, :] = rewards

    return all_rewards
```

5.1 Episodic Returns

The results are plotted for the best hyperparameters. The best hyperparameters are found using hyperparameter tuning, which is in the next section. The method to reproduce these results is in the readme.md

Figure 1 shows the episodic return (mean ± variance) for SARSA on CartPole-v1.

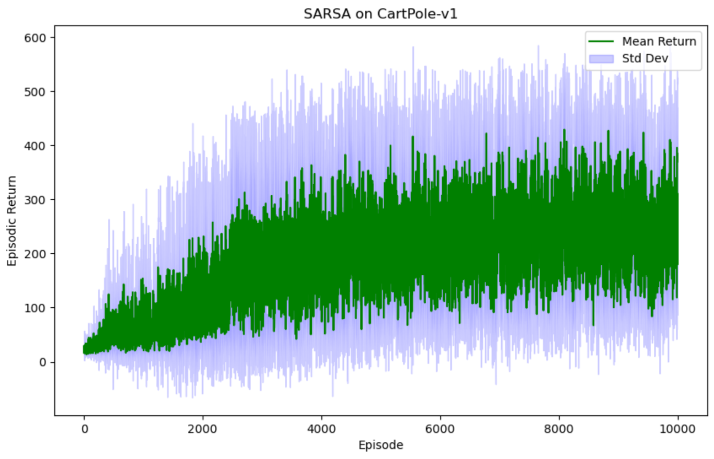

Figure 1: SARSA on CartPole-v1: Mean and standard deviation of episodic return over 5 seeds.

Figure 2 shows the corresponding plot for Q-Learning on Cartpole-v1.

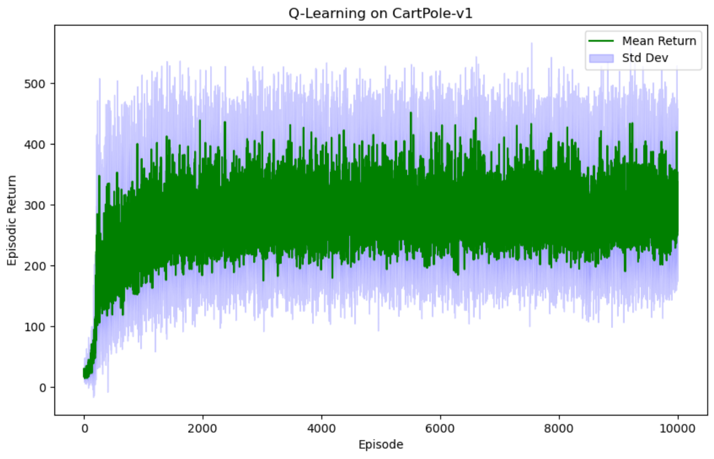

Figure 2: Q-Learning on CartPole-v1: Mean and standard deviation of episodic return over 5 seeds.

Figure 3 shows the episodic return (mean ± variance) for SARSA on MountainCar-v0.

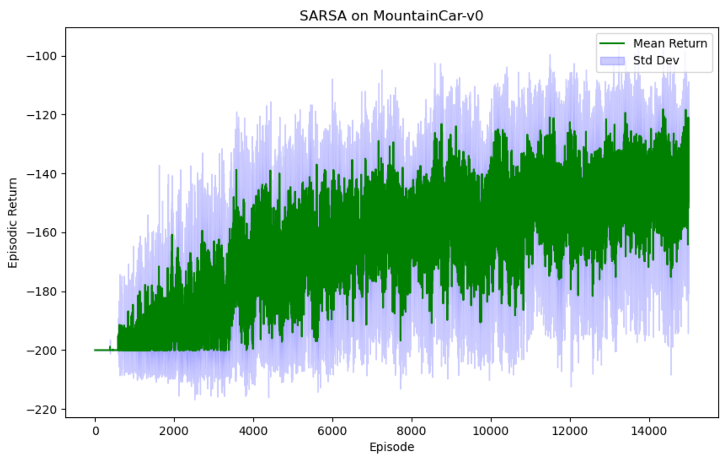

Figure 3: SARSA on MountainCar-v0: Mean and standard deviation of episodic return over 5 seeds.

Figure 4 shows the episodic return (mean ± variance) for Q-Learning on MountainCar-v0.

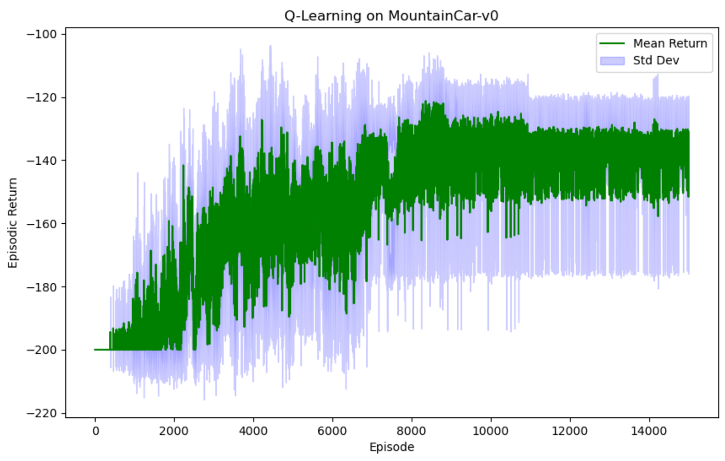

Figure 4: Q-Learning on MountainCar-v0: Mean and standard deviation of episodic return over 5 seeds.

6 Hyperparameter Tuning

We tested various hyperparameter settings. The top 3 settings for each algorithm were:

6.1 Parallel Coordinates Plots and Reward Images

The hyperparameters taken for SARSA on CartPole-v1 are:

- **Learning rate**
- **Epsilon Decay**
- **Number of bins**

Figure 5 shows the final parallel coordinates plot for SARSA on CartPole-v1.

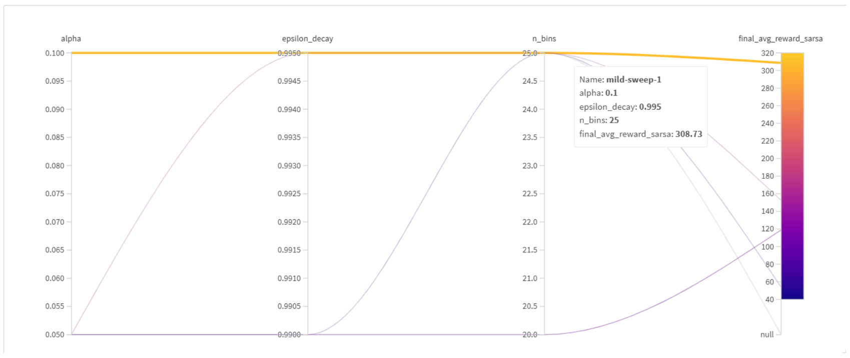

Figure 5: SARSA on CartPole-v1:Parallel coordinates plot.

Figure 6 shows the rewards for different runs for SARSA on CartPole-v1.

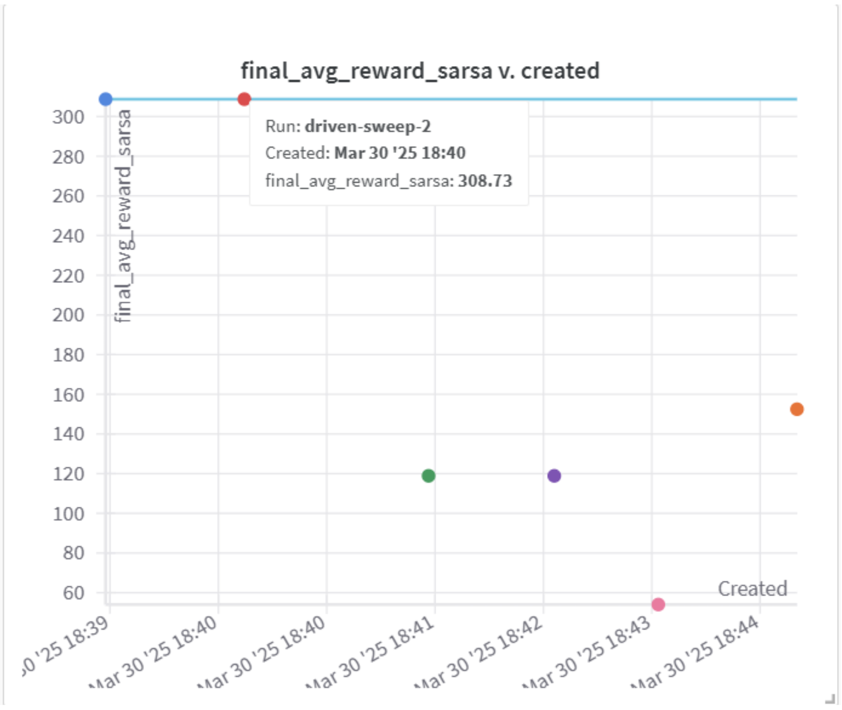

Figure 6: SARSA on CartPole-v1: Rewards for different runs.

The best hyperparameters for SARSA on CartPole-v1 from the parallel coordinates plot were found to be:

- **Learning rate:** 0.1
- **Epsilon Decay:** 0.995
- **Number of bins:** 25

The hyperparameters taken for Q-Learning on CartPole-v1 are:

- **Learning rate**
- **Tau Decay**
- **Number of bins**

Figure 7 shows the final parallel coordinates plot for Q-Learning on CartPole-v1.

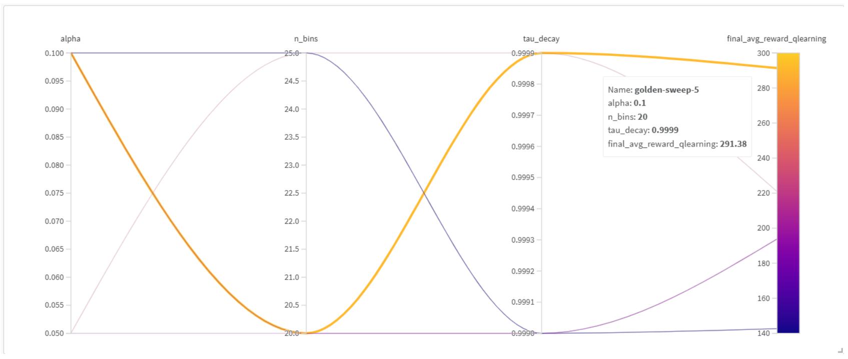

Figure 7: Q-Learning on CartPole-v1: Parallel coordinates plot.

Figure 8 shows the rewards for different runs for Q-Learning on CartPole-v1.

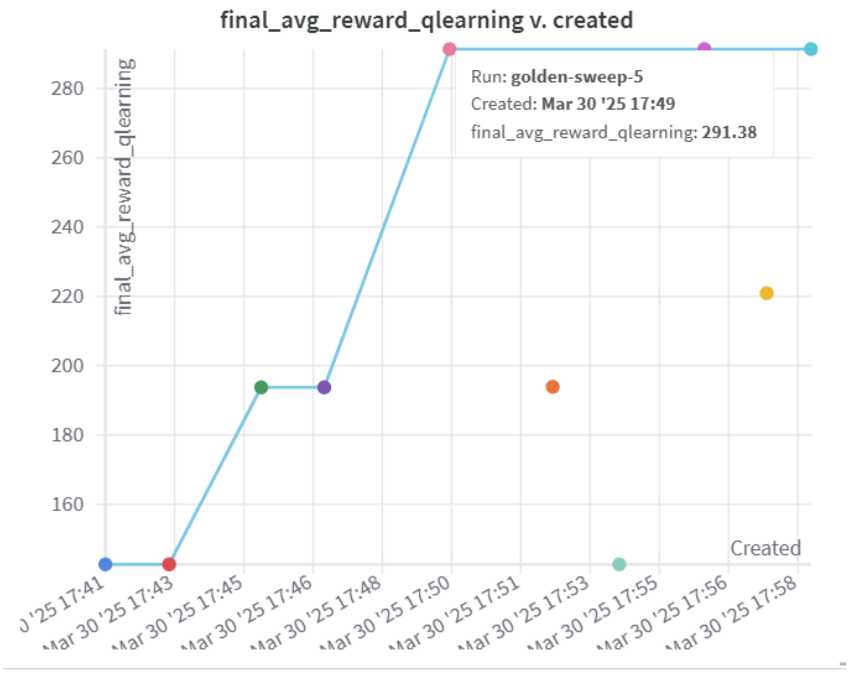

Figure 8: Q-Learning on CartPole-v1: Rewards for different runs.

The best hyperparameters for Q-Learning on CartPole-v1 from the parallel coordinates plot were found to be:

- **Learning rate:** 0.1
- **Tau Decay:** 20
- **Number of bins:** 0.9999

The hyperparameters taken for SARSA on MountainCar-v0 are:

- **Learning rate**
- **Epsilon Decay**

- **Number of bins**

Figure 9 shows the final parallel coordinates plot for SARSA on MountainCar-v0.

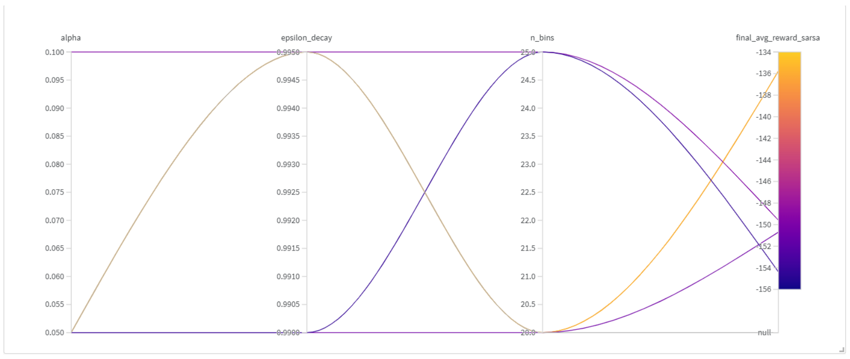

Figure 9: SARSA on MountainCar-v0: Parallel coordinates plot

Figure 10 shows the rewards for different runs for SARSA on MountainCar-v0 with.

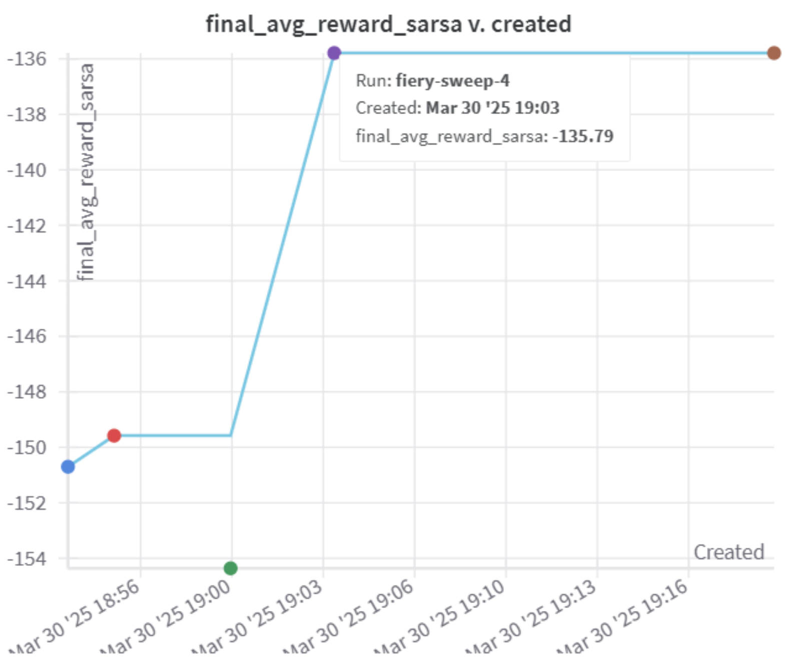

Figure 10: SARSA on MountainCar-v0: Rewards for different runs.

The best hyperparameters for SARSA on MountainCar-v0 from the parallel coordinates plot were found to be:

- **Learning rate:** 0.05
- **Epsilon Decay:** 0.995
- **Number of bins:** 20

The hyperparameters taken for Q-Learning on MountainCar-v0 are:

- **Learning rate**
- **Tau Decay**

- **Number of bins**

Figure 11 shows the final parallel coordinates plot for Q-Learning on MountainCar-v0.

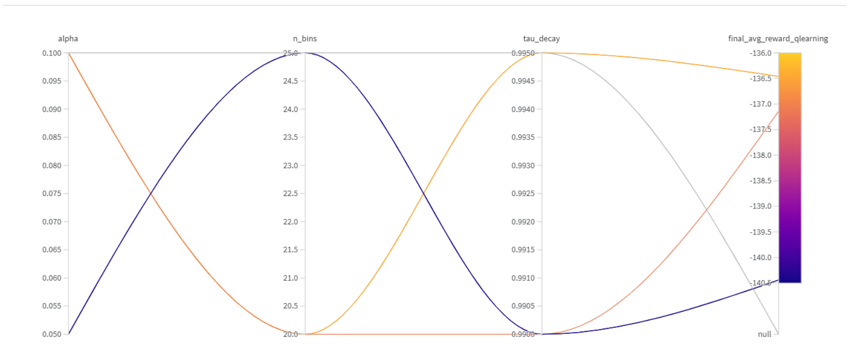

Figure 11: Q-Learning on MountainCar-v0: Parallel coordinates plot

Figure 12 shows the rewards for different runs for Q-Learning on MountainCar-v0.

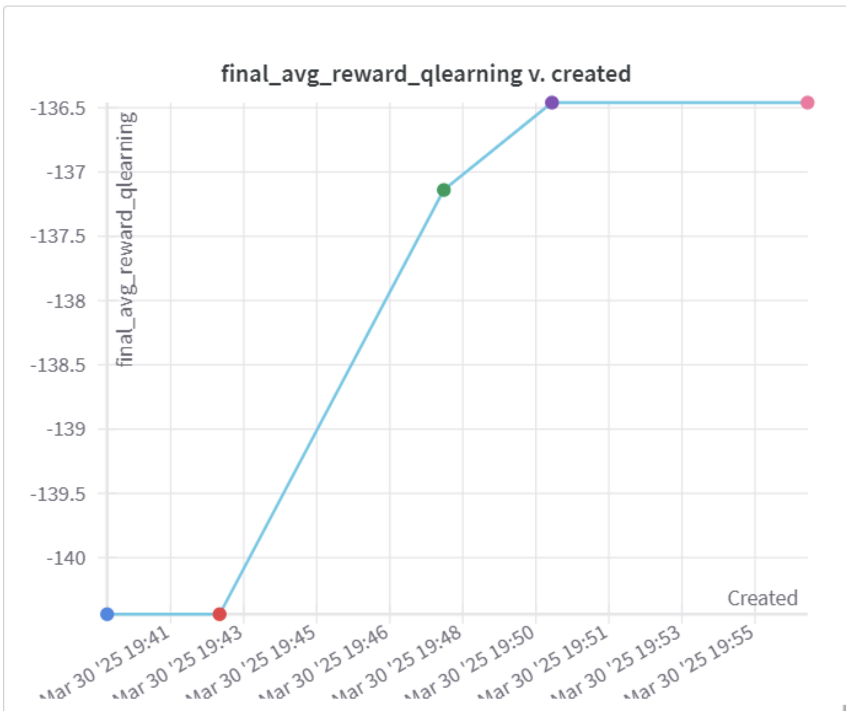

Figure 12: Q-Learning on MountainCar-v0:Rewards for different runs.

The best hyperparameters for Q-Learning on MountainCar-v0 from the parallel coordinates plot were found to be:

- **Learning rate:** 0.1
- **Tau Decay:** 20
- **Number of bins:** 0.995

7 Inferences and Conjectures

Based on our experiments, we observed the following:

- The number of bins can affect environmets like CartPole-v0 due to the definition of its states. A higher number of bins could help in better state representation as it can be seen in the hyperparameter tuning for CartPole-v0 using Sarsa.
- For both CartPole-v1 and MountainCar-v0, if the best hyperparameters are used, Q-Learning seems to give higher rewards than Sarsa.
- Q-Learning also lconverges quicker than Sarsa, as it can be seen from the episodic return plots that the Q-Learning plots flatten much earlier than the Sarsa Plots
- Sarsa tends to take lesser time per episode.
- The Parallel Coordinate Plots are used to find the best hyperparameters.

8 Code Repository

Our complete code is available on GitHub: `https://github.com/DA6400-RL-JanMay2025/programming-assignment-01-ch21b053_me21b171`

9 Bonus Task

We have also done the bonus task of evaluating the performance of SARSA and Q-Learning in the MiniGrid-Dynamic-Obstacles-5x5-v0.

The set of best hyperparameters of SARSA is: alpha0.134, epsilon 0.3117, decay 0.986, min 0.165 and seed 46.

And the set of best hyperparameters of Q-Learning is: alpha 0.1, decay 0.995, tau 1.0, min 0.01, seed 42.

Figure 13 shows the final reward plot distribution for Sarsa on MiniGrid-Dynamic-Obstacles5x5-v0.

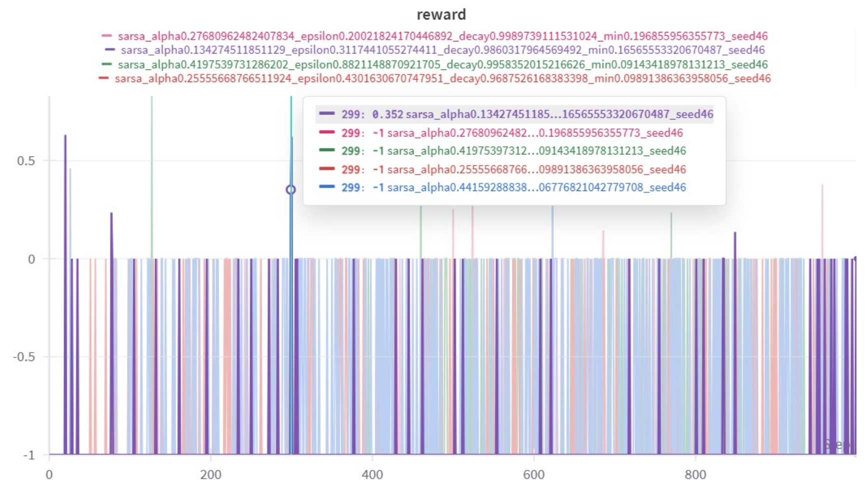

Figure 13: Sarsa on MiniGrid-Dynamic-Obstacles5x5-v0: Reward Plot Distribution.

Figure 14 shows the final reward plot distribution for Q-Learning on MiniGrid-Dynamic-Obstacles5x5-v0.

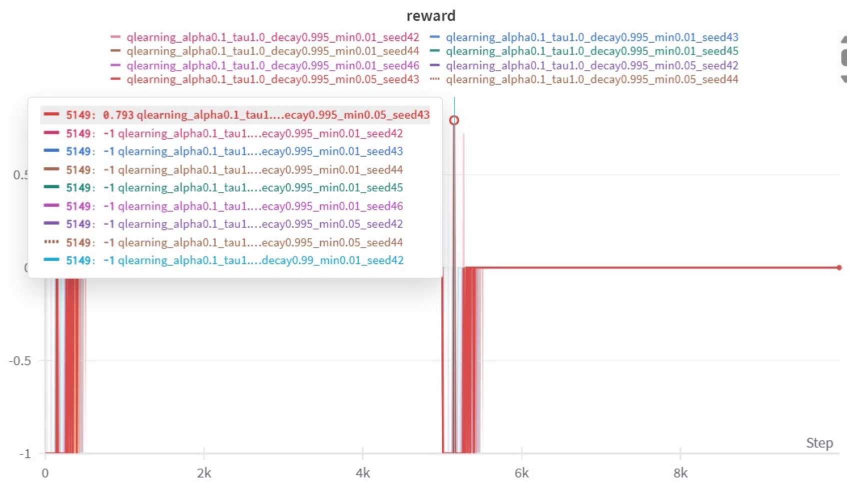

Figure 14: Q-Learning on MiniGrid-Dynamic-Obstacles5x5-v0: Reward Plot Distribution.

10 Conclusion

In this assignment, we implemented and compared SARSA and Q-Learning on multiple Gymnasium environments. Our experiments and hyperparameter tuning demonstrate the importance of appropriate discretization and exploration strategies in tabular reinforcement learning.