Programming Assignment I

Reinforcement Learning DA6400

Yash Gawande ME21B062

Sameer Deshpande CH21B033

March 30, 2025

Contents

1 Code

- Here is the github repository link for the assignment.

2 Structure of the Assignment

2.1 File Structure Overview

- We implemented both SARSA and Q-learning algorithms for CartPole_v1 and MountainCar_v0 environments, organized in separate dedicated modules.
- The implementations utilize our custom agent classes and trainer classes for both QLearning and Sarsa.
- We have used **Wandb** for hyperparameter tuning. In our scripts to run wandb sweeps, we minimze Cumulative Regret over all episodes across the runs in sweep.
- We maintained fixed episode count = 10000 across runs to enable consistent performance averaging and comparative plotting of mean returns with standard deviation bands.

3 Hyperparameters and Tuning :

We implemented Wandb to help us analyse the effects of changing the hyper-parameters and tuning them. Wandb is imported as a library and the tuning script (detailed in the section, Code Snippets) runs **bayesian optimisation** to fine tune the declared HyperParameters. Wandb syncs and saves all of the runs and logs values to the cloud, making it easy to maintain it across a group and monitor remotely.

- The objective of the optimization process was to **minimize the cumulative regret** during training of the agents on the various environment.
- We defined an objective function to evaluate the performance of the agent under different hyperparameter settings and conducted a **hyperparameter sweep** having 30 runs.

- Each iteration of the sweep involved training the agent with a specific configuration, logging the resulting **episodic return, regret averaged over 5 runs** to overcome **stochasticity**, and iteratively refining the search space to converge towards optimal hyperparameters.

- Hyperparameter **search space reduction**: We systematically narrowed the search space from 200 to 75 configurations through iterative wandb sweeps. This focused exploration on high-performing regions while maintaining coverage of optimal parameter combinations.

3.1 Cartpole Q Learning

- **Number of Bins:** The discretization granularity affects state representation accuracy and learning efficiency. Finer bins (40) capture more state details but increase the Q-table size and training time, while coarser bins (10) miss important state distinctions.

- **Learning Rate:** Determines the step size during parameter updates. Higher rates (1) learn faster but may overshoot optimal values, while lower rates (0.001) provide stable convergence at the cost of slower learning progress.

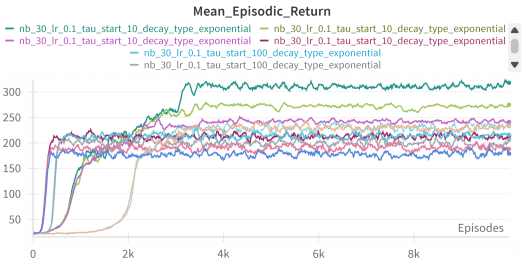
- **Initial Temperature (τ_{start}):** Governs exploration intensity in softmax policy - higher values promote exploration while lower values favor exploitation. Optimal values balance early exploration with later convergence to optimal policies.

- **Final Temperature (τ_{end}):** The value was fixed to 0.01 across all runs which indicated exploitation or greedy behaviour.

- τ **Decay Type:** Determines how exploration reduces over time - linear provides steady reduction while exponential allows rapid early exploration. The choice affects how quickly the agent transitions from exploration to exploitation behavior. Exponential Decay gives lower cumulative regret than linear decay as it starts exploiting earlier than linear decay.

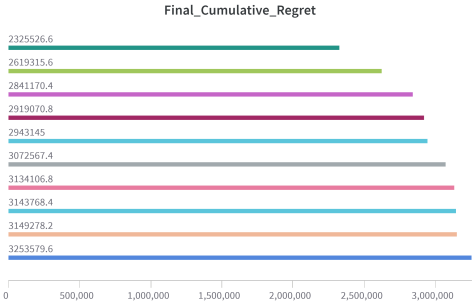
3.1.1 Plots from wandb after hyperparameter tuning

(a) Episodic Returns



(b) Moving Average (100 episode MA) Returns

(c) Episodic Regret



(d) Final Cumulative Regret

Figure 1: Cartpole Q learning Wandb Plots

3.1.2 Best Hyperparameters Obtained

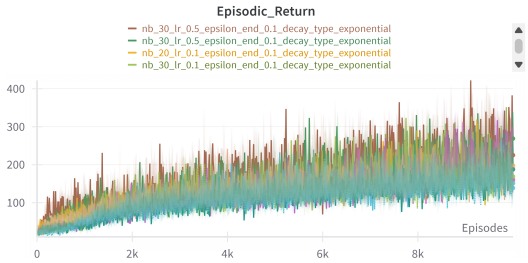
Hyperparameter	Values Tried	Best Value
Number of Bins	{10, 20, 30, 40}	30
Learning Rate (α)	{1, 1e-1, 1e-2, 1e-3}	0.1
Initial Temp (τ_{start})	{1, 1e1, 1e2, 1e3, 1e4, 1e5}	100
Final Temp (τ_{end})	0.01	0.01
Decay Type	{Linear, Exponential}	Exponential
Decay Rate	{1e-3, 1e-2, 1e-1}	0.01

Table 1: Hyperparameter Tuning Results for CartPole Q-learning

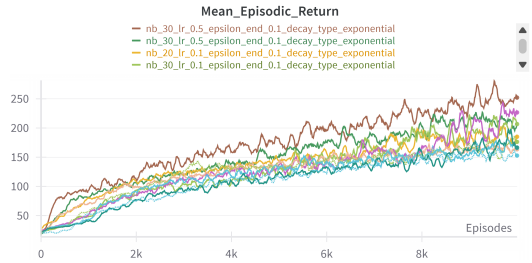
3.2 Cartpole SARSA

- **Number of Bins:** Based on the observations in Q learning 20, 30 values were used and 30 bins performed the best among them.
- **Learning Rate:** Based on earlier observations the values of 0.1, 0.5, 0.01 were used and learning rate = 0.1 performed the best among them.
- **Initial Epsilon (ϵ_{start}):** The value was fixed to 1 across all runs indicating that initially random actions will be taken for exploring.
- **Final Epsilon (ϵ_{end}):** Controls how much random exploration happens during later training episodes. The value of 0.1 performed best among the values of 0.1, 0.5, 0.01.
- ϵ **Decay Type:** Here also, exponential decay gives lower cumulative regret than linear decay as it starts exploiting earlier than linear decay.

3.2.1 Plots from wandb after hyperparameter tuning

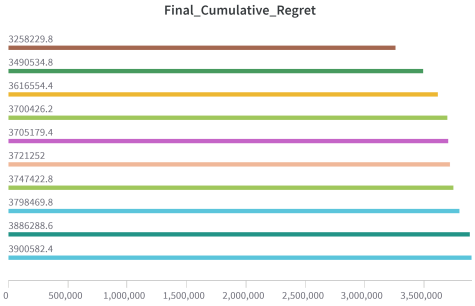


(a) Episodic Returns



(b) Moving Average (100 episode MA) Returns

(c) Episodic Regret



(d) Final Cumulative Regret

Figure 2: Cartpole Sarsa Wandb Plots

3.2.2 Best Hyperparameters Obtained

Hyperparameter	Values Tried	Best Value
Number of Bins	{20, 30}	30
Learning Rate (α)	{0.01, 0.1, 0.5}	0.5
Initial Epsilon (ϵ_{start})	1	1
Final Epsilon (ϵ_{end})	{0.01, 0.1, 0.5}	0.1
Decay Type	{Linear, Exponential}	Exponential
Decay Rate	{0.01, 0.1}	0.01

Table 2: Hyperparameter Tuning Results for CartPole SARSA

3.3 Mountain Car Q learning

For hyperparameter tuning of the Q learning algorithm, we use Bayesian optimization to minimize cumulative regret.

- **Alpha (α)**: Controls the learning rate, determining how much new information overrides old Q-values. A higher α leads to faster learning but may cause instability. *We tune it using a uniform distribution between 0.01 and 0.5 to balance stability and adaptability.*

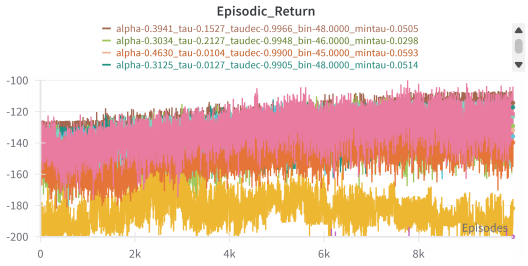
- **Tau (τ)**: Used in softmax exploration to control the randomness of action selection. A higher τ results in more exploration, while a lower τ promotes exploitation. *We tune it using a uniform distribution between 0.01 and 2.0 to ensure an appropriate exploration-exploitation trade-off.*

- **Tau Decay (τ_{decay})**: Determines the rate at which τ decreases over time, reducing exploration as learning progresses. *We tune it using a uniform distribution between 0.99 and 0.9999 to allow gradual annealing of exploration.*

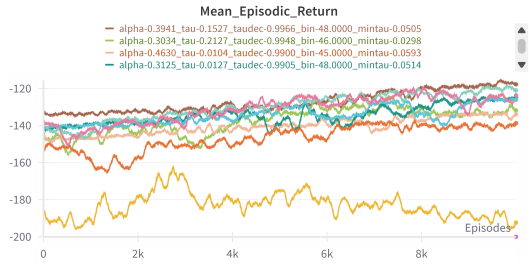
- **Bin Size**: Specifies the discretization level for continuous state/action spaces, impacting the granularity of value updates and generalization. *We tune it using a quantized log-uniform distribution between 10 and 50 to find an optimal discretization level.*

- **Minimum Tau (τ_{min})**: Defines the lowest possible value for τ, ensuring that the policy does not become entirely deterministic too early.
  We tune it using a uniform distribution between 0.01 and 0.1 to prevent premature convergence to a suboptimal policy.

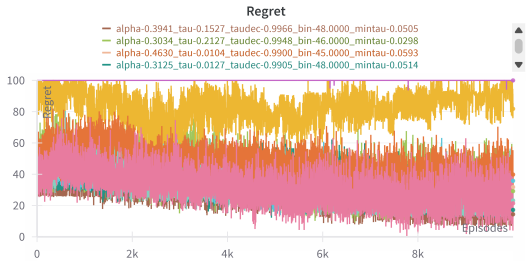
3.3.1 Plots from wandb after hyperparameter tuning



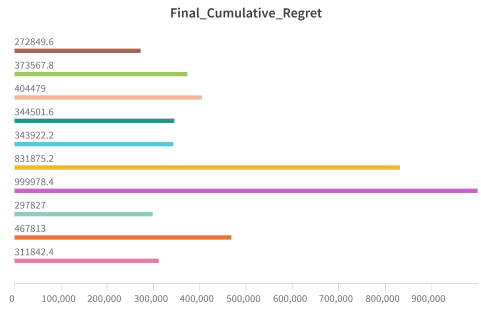
(a) Episodic Returns



(b) Moving Average (100 episode MA) Returns



(c) Episodic Regret



(d) Final Cumulative Regret

Figure 3: Mountain Car Q Learning Wandb Plots

3.3.2 Best hyperparameters obtained

Hyperparameter	Value
Alpha (α)	0.3941
Tau (τ)	0.1527
Tau Decay (τ_{decay})	0.9966
Bin Size	48
Minimum Tau (τ_{min})	0.0505

Table 3: Best Hyperparameters for Mountain Car Q-learning

3.4 Mountain Car SARSA

For hyperparameter tuning of the SARSA algorithm, we use Bayesian optimization to minimize cumulative regret.

- **Alpha (α)**: Controls the learning rate, determining how much new information updates the Q-values in SARSA. A higher α speeds up learning but may cause instability.
  We tune it over a uniform search space between 0.01 and 0.5 to balance stability and adaptability.

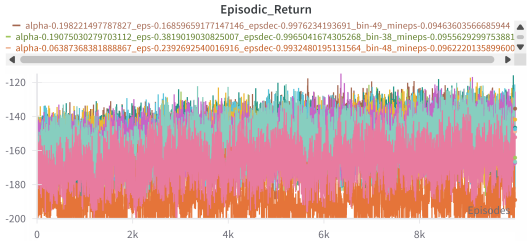
- **Epsilon (ϵ)**: Defines the initial probability of choosing a random action, encouraging exploration. A higher ϵ allows more exploration, which is crucial in early training.
  We tune it over a uniform search space between 0.1 and 1.0 to ensure effective exploration-exploitation trade-off.

- **Epsilon Decay (ϵ_{decay})**: Determines how fast ϵ decreases over time, allowing a gradual shift from exploration to exploitation in SARSA.
  We tune it over a uniform search space between 0.99 and 0.9999 to maintain sufficient exploration while converging efficiently.

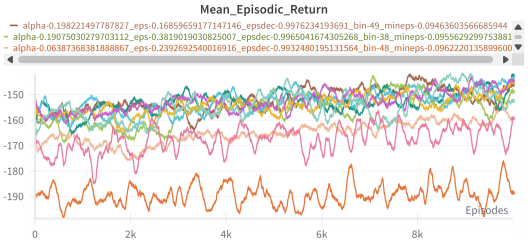
- **Bin Size**: Defines the discretization level for continuous state/action spaces, affecting the granularity of state representations in SARSA.
  We tune it over a quantized log-uniform search space between 10 and 50 to optimize learning precision.

- **Minimum Epsilon (ϵ_{min})**: Specifies the lowest possible value for ϵ, ensuring that the agent retains some exploration throughout training to avoid getting stuck in local optima.
  We tune it over a uniform search space between 0.01 and 0.1 to prevent excessive greediness in policy learning.

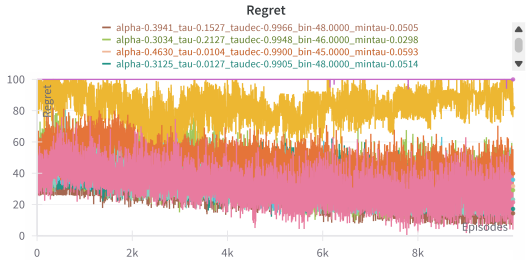
3.4.1 Plots from wandb after hyperparameter tuning



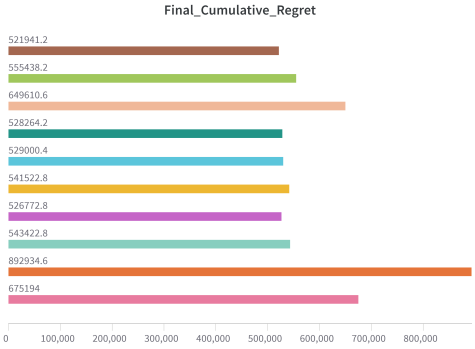
(a) Episodic Returns



(b) Moving Average (100 episode MA) Returns



(c) Episodic Regret



(d) Final Cumulative Regret

Figure 4: Mountain Car SARSA Wandb Plots

3.4.2 Best hyperparameters obtained

Hyperparameter	Value
Alpha (α)	0.198
Epsilon (ϵ)	0.168
Epsilon Decay (ϵ_{decay})	0.997
Bin Size	49
Minimum Epsilon (ϵ_{min})	0.094

Table 4: Hyperparameter values used in SARSA

4 Code-Snippets

4.1 Cartpole Q Learning

Q Learning Agent Class

```
class QLearningAgent:
    def __init__(self, state_space_shape, action_space_n):
        self.q_table = self._initialize_q_table(state_space_shape +
            (action_space_n,))
        self.exploration = ExplorationStrategy()

    @staticmethod
    def _initialize_q_table(shape):
        """Initialize the Q-table with zeros."""
        return np.zeros(shape)

    def get_action(self, state, tau=1.0):
        """Select action using softmax policy."""
        action_probs = self.exploration.softmax(self.q_table[state],
            tau)
        return np.random.choice(len(action_probs), p=action_probs)

    def update(self, state, action, reward, next_state, gamma,
        learning_rate):
        """Q-learning update rule."""
        best_next_action = np.argmax(self.q_table[next_state])
        td_target = reward + gamma *
            self.q_table[next_state][best_next_action]
        td_error = td_target - self.q_table[state + (action,)]
        self.q_table[state + (action,)] += learning_rate * td_error
```

Q Learning Episode Run

```
def _run_episode(self, tau):
    """Run a single episode."""
    state, _ = self.env.reset()
    state = self.discretizer.discretize(state)
    terminated = truncated = False
    total_score = 0

    while not (terminated or truncated):
        action = self.agent.get_action(state, tau)
        next_state, reward, terminated, truncated, _ =
            self.env.step(action)
        next_state = self.discretizer.discretize(next_state)
```

```
        self.agent.update(
            state, action, reward, next_state,
            self.config['gamma'], self.config['learning_rate']
        )

        total_score += reward
        state = next_state

    return total_score
```

4.2 Cartpole Sarsa

Sarsa Agent Class

```
class SARSAAgent:
    def __init__(self, state_space_shape, action_space_n):
        self.q_table = self._initialize_q_table(state_space_shape +
            (action_space_n,))
        self.exploration = ExplorationStrategy()

    @staticmethod
    def _initialize_q_table(shape):
        """Initialize the Q-table with zeros."""
        return np.zeros(shape)

    def get_action(self, state, epsilon):
        """Select action using epsilon-greedy policy."""
        return self.exploration.epsilon_greedy(self.q_table[state],
            epsilon)

    def update(self, state, action, reward, next_state, next_action,
        gamma, learning_rate):
        """SARSA update rule."""
        td_target = reward + gamma *
            self.q_table[next_state][next_action]
        td_error = td_target - self.q_table[state + (action,)]
        self.q_table[state + (action,)] += learning_rate * td_error
```

Sarsa Episode Run

```
def _run_episode(self, epsilon):
    """Run a single episode."""
    state, _ = self.env.reset()
    state = self.discretizer.discretize(state)
    terminated = truncated = False
```

```
    total_score = 0

    # Select initial action
    action = self.agent.get_action(state, epsilon)

    while not (terminated or truncated):
        next_state, reward, terminated, truncated, _ =
            self.env.step(action)
        next_state = self.discretizer.discretize(next_state)

        # Select next action (on-policy)
        next_action = self.agent.get_action(next_state, epsilon)

        # SARSA update
        self.agent.update(
            state, action, reward, next_state, next_action,
            self.config['gamma'], self.config['learning_rate']
        )

        total_score += reward
        state, action = next_state, next_action

    return total_score
```

4.3 Mountain Car Q learning

Q Learning Agent Class

```
# Agent Class
class Mountain_Car_Q_Learning_Agent:

    def __init__(self,
                 env,bin_size=[30,30],alpha=0.1,gamma=0.99,tau=0.5,
                 tau_decay=0.999,episodes=10000,min_tau=0.01):
        self.env = env
        self.bin_size = bin_size
        self.q_value = np.zeros((self.bin_size[0],
        self.bin_size[1],self.env.action_space.n))
        self.alpha = alpha
        self.gamma = gamma
        self.tau = tau
        self.episodes = episodes
        self.tau_decay = tau_decay
        self.min_tau = min_tau
        self.discretize_bins = self.create_discretize_bins()
```

```
    # Chooosing action using softmax policy
    def choose_action(self,state):

        q_max = max(self.q_value[state[0],state[1]])
        q_value_exp = np.exp((self.q_value[state[0],
        state[1]]-q_max)/self.tau)

        q_value_prob = q_value_exp/np.sum(q_value_exp)

        return np.random.choice(self.env.action_space.n,
           p=q_value_prob)

    # Creating bins
    def create_discretize_bins(self):

        grid = []

        for i in range(len(self.env.observation_space.high)):
            grid.append(np.linspace(self.env.observation_space.low[i],
            self.env.observation_space.high[i],
            self.bin_size[i]+1)[1:-1])

        return grid

    # Discretizing into bins
    def discretize_state(self,state):

        state_index = []

        for i in range(len(state)):
            state_index.append(np.digitize(state[i],
            self.discretize_bins[i]))

        return tuple(state_index)

    # Updating q value
    def update_q_value(self,state,action,reward,next_state):

        self.q_value[state[0],state[1],action] += self.alpha*(reward +
           self.gamma*np.max(self.q_value[next_state[0],
           next_state[1]]) - self.q_value[state[0],state[1],action])

    # Decaying the tau value
    def decay_tau(self):

        self.epsilon = max(self.tau*self.tau_decay, self.min_tau)
```

Q Learning Training Function

```
# Function that trains the Q learning agent
def train(env,bin_size,alpha,gamma,tau,tau_decay,episodes,min_tau):


    all_episodic_returns= []
    all_steps = []
    all_episodic_regrets = []

    no_runs = 5

    for run in range(no_runs):

        agent_Q_learning = Mountain_Car_Q_Learning_Agent(env,
        bin_size,alpha,gamma,tau,tau_decay,episodes,min_tau)

        episodic_returns = []
        steps  = []
        optimal_return_per_episode = -100
        episodic_regrets =  []

        for episode in range(episodes):

            state = agent_Q_learning.discretize_state(env.reset()[0])
            action = agent_Q_learning.choose_action(state)
            done = False

            return_per_episode = 0
            step_per_episode = 0

            while not done:

                # Acting in the environment
                next_state,reward,terminated,truncated,_ =
                    env.step(action)

                # Discretizing the next state
                next_state =
                    agent_Q_learning.discretize_state(next_state)

                # Choosing the next action
                next_action =
                    agent_Q_learning.choose_action(next_state)

                # Updating the Q-value
                agent_Q_learning.update_q_value(state, action, reward,
                    next_state)
```

```
                # Updating the state and action
                state = next_state
                action = next_action

                # Updating the episode reward
                return_per_episode += reward
                step_per_episode += 1

                done = terminated or truncated

            # Decaying the epsilon
            agent_Q_learning.decay_tau()

            # Print progress every 100 episodes
            if (episode+1) % 100 == 0:
                print(f"Run␣:␣{run+1},␣Episode␣{episode␣+␣
                    1}/{episodes},␣Episodic␣return:␣
                    {return_per_episode:.4f}")

            regret_per_episode = optimal_return_per_episode -
                return_per_episode

            episodic_regrets.append(regret_per_episode)
            episodic_returns.append(return_per_episode)
            steps.append(step_per_episode)


        all_episodic_returns.append(episodic_returns)
        all_episodic_regrets.append(episodic_regrets)
        all_steps.append(steps)


    return all_episodic_returns, all_episodic_regrets
```

4.4 Mountain Car SARSA

SARSA Agent Class

```
# Agent Class
class Mountain_Car_SARSA_Agent:

    def __init__(self,
                 env,bin_size=[30,30],alpha=0.1,gamma=0.99,epsilon=0.1,
                 epsilon_decay=0.999,episodes=10000,min_epsilon=0.01):
```

```
        self.env = env
        self.bin_size = bin_size
        self.q_value = np.zeros((self.bin_size[0],
        self.bin_size[1],self.env.action_space.n))
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.episodes = episodes
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = min_epsilon
        self.discretize_bins = self.create_discretize_bins()

    # Choosing action using epsilon-greedy approach
    def choose_action(self,state):

        if np.random.random() < self.epsilon:
            return np.random.choice(self.env.action_space.n)
        else:
            return np.argmax(self.q_value[state[0],state[1]])

    # Creating discrete bins
    def create_discretize_bins(self):

        grid = []

        for i in range(len(self.env.observation_space.high)):
            grid.append(np.linspace(self.env.observation_space.low[i],
            self.env.observation_space.high[i],
            self.bin_size[i]+1)[1:-1])

        return grid

    # Discretizing the states
    def discretize_state(self,state):

        state_index = []

        for i in range(len(state)):
            state_index.append(np.digitize(state[i],
            self.discretize_bins[i]))

        return tuple(state_index)

    # Updating the q value function.
    def update_q_value(self,state,action,reward,
    next_state,next_action):

        self.q_value[state[0],state[1],action] += self.alpha*(reward+
           self.gamma*self.q_value[next_state[0],next_state[1],
```

```
        next_action] - self.q_value[state[0],state[1],action])

    # Decaying the epsilon value
    def decay_epsilon(self):

        self.epsilon = max(self.epsilon*self.epsilon_decay,
           self.min_epsilon)
```

SARSA Training Function

```
# Function that trains the SARSA agent
def train(env,bin_size,alpha,gamma,epsilon,epsilon_decay,
episodes,min_epsilon):

    all_episodic_returns= []
    all_steps = []
    all_episodic_regrets = []

    no_runs = 5

    for run in range(no_runs):

        agent_SARSA = Mountain_Car_SARSA_Agent(env,bin_size,alpha,
        gamma,epsilon,epsilon_decay,episodes,min_epsilon)

        episodic_returns = []
        steps  = []
        optimal_return_per_episode = -100
        episodic_regrets =  []

        for episode in range(episodes):

            state = agent_SARSA.discretize_state(env.reset()[0])
            action = agent_SARSA.choose_action(state)
            done = False

            return_per_episode = 0
            step_per_episode = 0

            while not done:

                # Acting in the environment
                next_state,reward,terminated,truncated,_ =
                   env.step(action)

                # Discretizing the next state
                next_state = agent_SARSA.discretize_state(next_state)
```

```
                # Choosing the next action
                next_action = agent_SARSA.choose_action(next_state)

                # Updating the action-value function
                agent_SARSA.update_q_value(state,action,
                reward,next_state,next_action)

                # Updating the state and action
                state = next_state
                action = next_action

                # Updating the episode reward
                return_per_episode += reward
                step_per_episode += 1

                done = terminated or truncated

            # Decaying the epsilon
            agent_SARSA.decay_epsilon()

            # Print progress every 100 episodes
            if (episode+1) % 100 == 0:
                print(f"Run␣:␣{run+1},␣Episode␣{episode␣+␣
                    1}/{episodes},␣Episodic␣return:␣
                    {return_per_episode:.4f}")

            regret_per_episode = optimal_return_per_episode -
                return_per_episode

            episodic_regrets.append(regret_per_episode)
            episodic_returns.append(return_per_episode)
            steps.append(step_per_episode)


        all_episodic_returns.append(episodic_returns)
        all_episodic_regrets.append(episodic_regrets)
        all_steps.append(steps)

    return all_episodic_returns, all_episodic_regrets
```

5 Results of Top 3 hyperparameters

5.1 Cartpole Q Learning

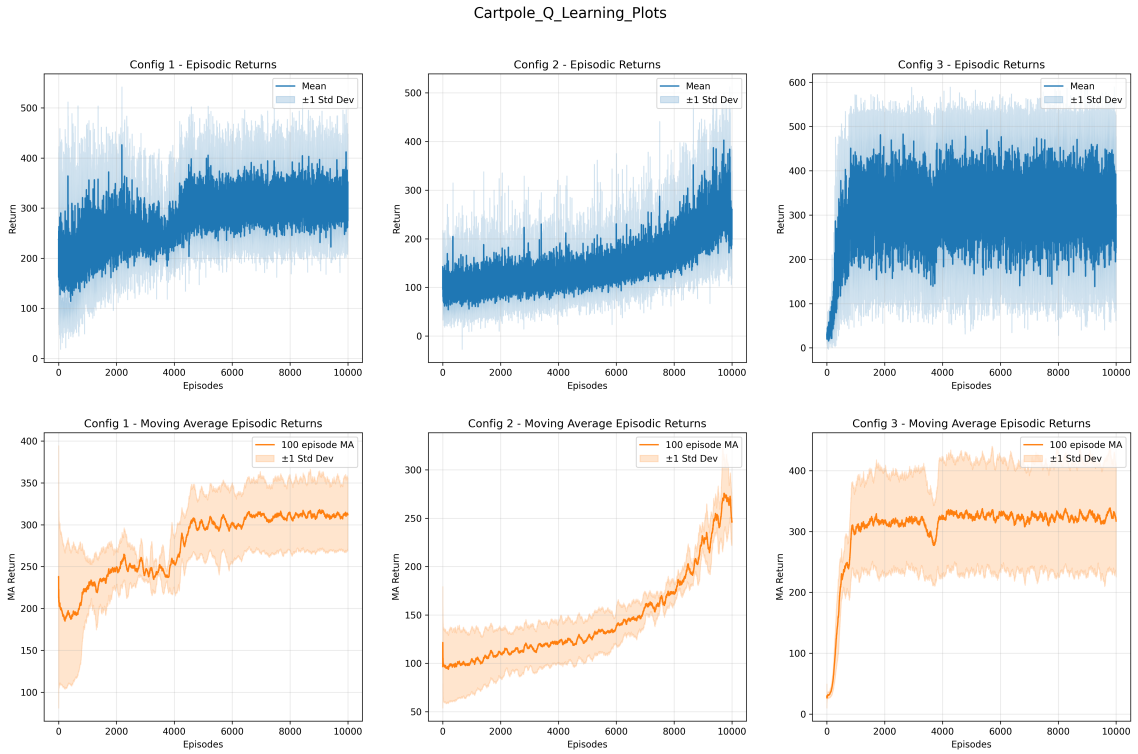


Figure 5: Plots for top 3 configurations

- τ **Decay Strategy:** The exponential decay configuration achieved faster initial learning compared to linear decay of τ. Hence, the final cumulative regret is lower for exponential decay as compared to linear decay.
- **Initial Temperature** τ_{start}**:** Higher initial temperature ($\tau_{start} = 100$) shows more exploration in early episodes, which is evident by its wider performance variance compared to $\tau_{start} = 10$ configuration.
- **Decay Rate:** The exponential decay with rate 0.1 showed the most rapid reduction in exploration and started behaving greedily after 1000 episodes whereas the 0.01 rate maintained moderate exploration.

- **Final Performance:** All three configurations achieved comparable final moving average returns (250-350), but the configuration 1 showed lower variance, indicating more stable learned policies during the exploitation phase.

Parameter	**Config 1**	**Config 2**	**Config 3**
Number of bins	30	30	30
Learning rate (α)	0.1	0.1	0.1
Discount factor (γ)	0.99	0.99	0.99
Initial temp (τ_{start})	10	10	100
Final temp (τ_{end})	0.01	0.01	0.01
Decay type	Exponential	Linear	Exponential
Decay rate	0.01	0.001	0.1
Training episodes	10,000	10,000	10,000
Runs (averaged over)	5	5	5

Table 5: Top 3 Q-learning hyperparameter configurations

5.2 Cartpole Sarsa

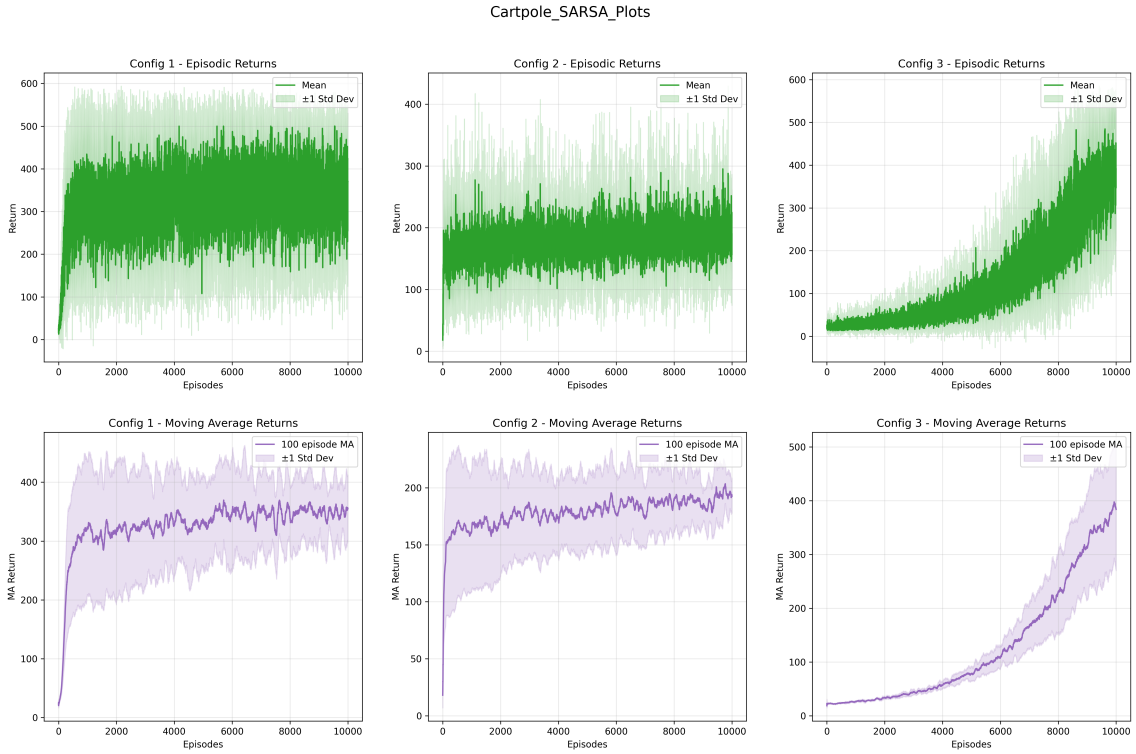


Figure 6: Plots for top 3 configurations

- **Learning Rate:** The higher learning rate $\alpha = 0.5$ configuration showed faster initial learning but greater variance in early episodes compared to $\alpha = 0.1$. However, both rates converged to similar final performance.
- ϵ **Decay Strategy:** Exponential decay achieved higher mean returns than linear decay in the earlier episodes. The linear decay explored more and achieved higher final episodic returns than exponential decay, but this also resulted in higher cumulative regret in linear decay.
- **Decay Rate:** The exponential decay rate 0.01 configuration showed more gradual performance improvement but ultimately achieved higher final returns than the faster decaying 0.1 rate configuration, suggesting benefits from prolonged moderate exploration.

- **Final Performance:** All three configurations achieved comparable final moving average returns (200-400). The configuration 2 showed lower variance, indicating more stable learned policies during the exploitation phase.

Parameter	**Config 1**	**Config 2**	**Config 3**
Number of bins	30	30	30
Learning rate (α)	0.5	0.1	0.1
Discount factor (γ)	0.99	0.99	0.99
Initial ϵ (ϵ_{start})	1	1	1
Final ϵ (ϵ_{end})	0.1	0.1	0.1
Decay type	Exponential	Exponential	Linear
Decay rate	0.01	0.1	0.1
Training episodes	10,000	10,000	10,000
Runs (averaged over)	5	5	5

Table 6: Top 3 SARSA hyperparameter configurations

5.3 Mountain Car Q Learning

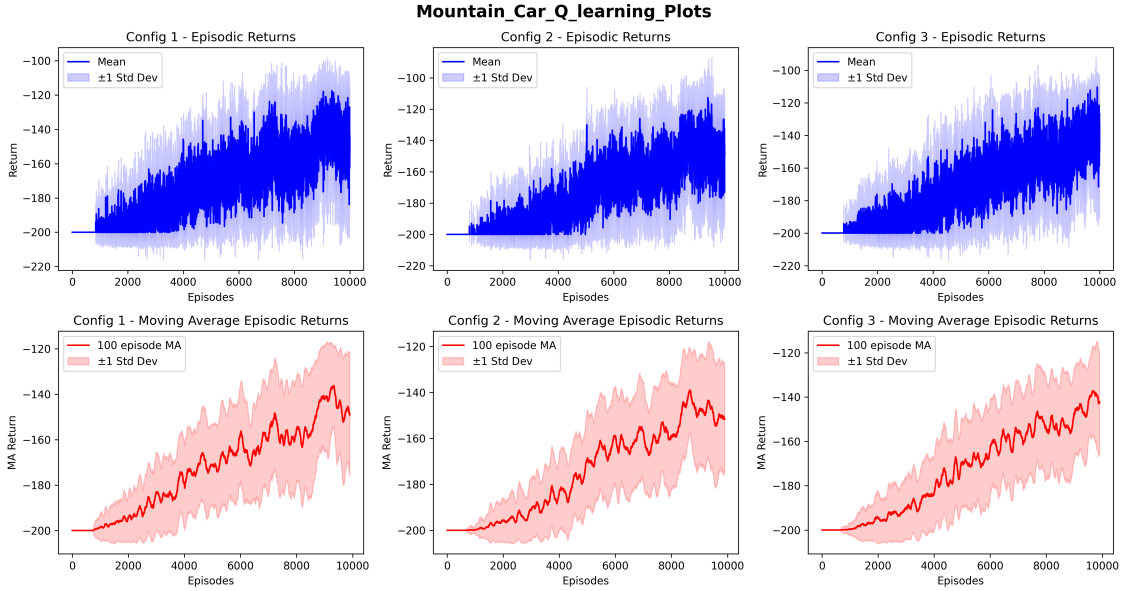


Figure 7: Plots for top 3 configurations

Hyperparameter	Config 1	Config 2	Config 3
Alpha (α)	0.3941	0.2634	0.2314
Tau (τ_{inital})	0.1527	0.1266	0.0121
Tau Decay (τ_{decay})	0.9966	0.9976	0.9945
Bin Size	48	46	45
Minimum Tau (τ_{min})	0.0505	0.0960	0.0814

Table 7: Top 3 Q-learning hyperparameter configurations

- **Learning Rate:** The higher learning rate $\alpha = 0.39$ configuration showed faster initial learning as compared to other learning rates. However, the high variance in the episodic returns suggests potential instability or overshooting.
- **Initial Temperature $\tau_{initial}$:** A moderate initial exploration rate allows the algorithm to explore the state-action space reasonably well from the start as compared to the other configurations.

- **Tau Decay:** Config 1 (0.9966) & Config 2 (0.9976) have very high decay rates, indicating a slow reduction in exploration over time. This allows the algorithm to continue exploring for a longer period, potentially leading to better coverage of the state-action space

- **Tau minimum:**The lowest minimum exploration rate ensures that the algorithm continues to explore even after a long period of learning, as seen in the config 1.

- **Bin Size:** The larger bin size results in the finest discretization of the state space. This can lead to a more accurate representation of the environment, potentially allowing the algorithm to learn more precise Q-values. However, it also increases the complexity of the Q-table, potentially slowing down learning.

- **Final Performance:** All three configurations achieved comparable final moving average returns (-150 to -140), but configuration 1 reached a higher episodic return, as can be seen from the episodic return plots.

5.4 Mountain Car SARSA

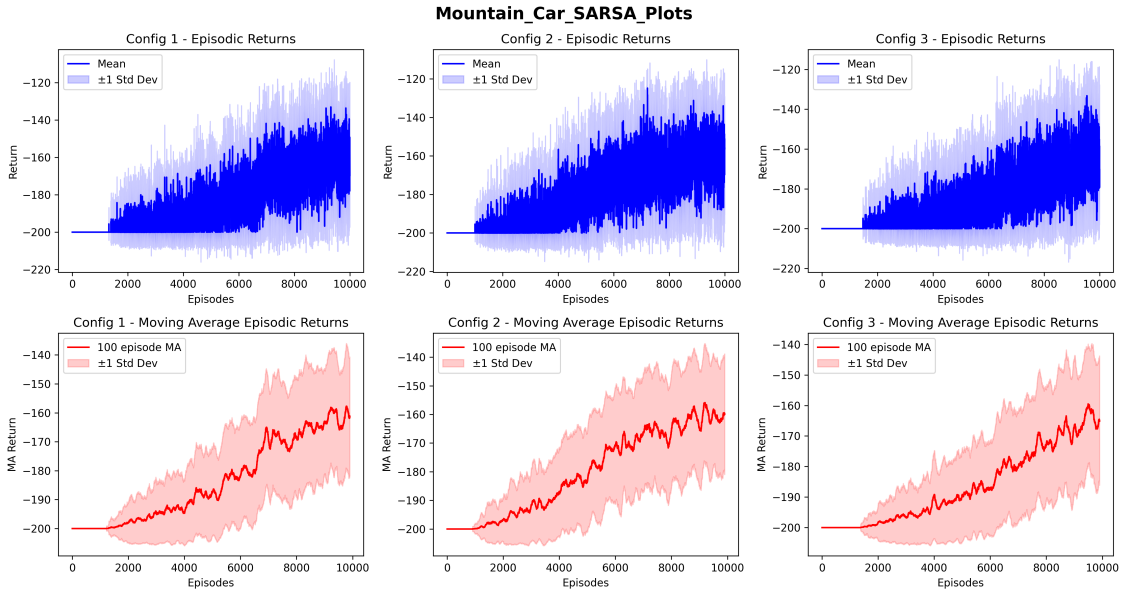


Figure 8: Plots for top 3 configurations

Hyperparameter	Config 1	Config 2	Config 3
Alpha (α)	0.198	0.205	0.144
Epsilon (ϵ_{inital})	0.168	0.637	0.214
Epsilon Decay (ϵ_{decay})	0.997	0.995	0.994
Bin Size	49	44	45
Minimum Epsilon (ϵ_{min})	0.094	0.099	0.097

Table 8: Top 3 SARSA hyperparameter configurations

- **Learning Rate:** Config 1 (0.198) and Config 2 (0.205) have similar moderate learning rates. They allow for reasonable updates to Q-values without being overly aggressive. Config 2's slightly higher alpha might lead to slightly faster learning but also a slightly higher risk of instability.

- ϵ **Initial:** Config 1 (0.168) has relatively low initial exploration suggests a focus on exploitation from the start. The config 2 (0.637) has the highest initial exploration

indicates a strong emphasis on exploring the state-action space early on. The Config 3 (0.214) has a moderate initial exploration balanceing exploration and exploitation.

- **Bin Size:** The larger bin size results in the finest discretization of the state space. This can lead to a more accurate representation of the environment, potentially allowing the algorithm to learn more precise Q-values. However, it also increases the complexity of the Q-table, potentially slowing down learning.

- **Epsilon Decay:** Config 1 (0.997) has the highest decay rate means exploration is reduced very slowly. This allows for prolonged exploration, potentially leading to better coverage of the state-action space. Config 2 (0.995) has a moderate decay rate reducing exploration at a moderate pace. Config 3 (0.994) has the lowest decay rate meaning exploration decreases relatively faster. This can lead to quicker exploitation but might miss out on better solutions if the initial exploration was insufficient.

- **Final Performance:** All three configurations achieved comparable final moving average returns (-160 to -150), but configuration 1 reached a higher episodic return, as can be seen from the episodic return plots.

6 Inference and Conclusion

Before moving on to the Inferences, it is important to state the importance of the **Termination condition** for Training and the experiments. We did all the runs for 10000 episodes minimizing cumulative regret. The number of episodes could be reduced while running experiments with tuned hyperparameters because we can observe that in case of exponential decay, the episodic return saturates in few episodes.

6.1 Cartpole

- Q-learning with softmax exploration demonstrated more stable policy convergence (lower variance) compared to SARSA's ϵ-greedy approach, particularly with exponential decay. However, SARSA showed faster initial learning (reaching higher mean return in fewer episodes) due to its more aggressive early exploration.
- **Final Performance:** Sarsa with linear decay achieved highest moving average return (400) but it had higher cumulative regret. Sarsa with exponential decay seems to perform better than Q learning giving lower cumulative regret.

6.2 MountainCar

- As the learning of the agent approached the end of 10,000 episodes, we observe that Q-learning algorithm reached higher episodic return (-120) than the SARSA algorithm (-140).
- Q-learning algorithm resulted in less cumulative regret over all the episodes than the SARSA algorithm, as it is visible in the plots. This suggests that Q-learning algorithm reaches near to the optimal return quite often than SARSA algorithm.
- We also observed that Q-learning showed faster initial learning than SARSA. This might be due to low $\epsilon_{\text{initial}}$ value taken in the ϵ-greedy algorithm in SARSA which forced it to do exploitation.