DA6400: Introduction to Reinforcement Learning
Assignment 1

Jayagowtham J ME21B078, Lalit Jayanti ME21B096

Algorithms and Training

Agent

This is the common code block for both of our SARSA and Q Learning Agents. It encapsulates initializing discretizer, agent and agent environment interaction parameters. The functions *update_* override the default parameters giving us more control over our environment.

```
# scripts/agents.py
class SARSAAgent: / class QLearningAgent:
    def __init__(self, state_space, action_space, seed):
        ''' Intialises our agent with discretizer, agent and
        agent environment parameters
        '''

    def reset(self):
        '''Updates all the hyperparameters and related objects
        '''

    def update_hyperparameters(self, **kwargs):
        '''Updates hyperparameters at the start of training
        '''

    def update_agent_parameters(self):
        '''Updates agent training parameters based on decay type
        '''
```

SARSA

This code block deals with the state action value update step according to the SARSA algorithm. We execute the given action from our current state, observe the next state and reward and then choose the next action based on the ϵ-greedy policy. Now, we update the state action value function based on the next state and chosen action value function.

```
# scripts/agents.py
class SARSAAgent:
    .
    .
    .
    def step(self, state, action, reward, next_state, done):
        '''Updates Q-values according to the SARSA
        algorithm
        '''
        q_sa = self.q_table[state, action]

        next_action_values = self.q_table[next_state]

        # Choosing the next action
        self.next_action = epsilon_greedy(
            action_values=next_action_values,
            action_size=self.action_size,
            eps=self.eps
        )

```

```
        q_next_sa = next_action_values[self.next_action]

        # SARSA update
        self.q_table[state, action] = (
            q_sa + self.LR*(reward + self.GAMMA*q_next_sa - q_sa)
        )

    def act(self, state):
        '''Takes an action from current state based on
        epsilon-greedy policy according to current Q values.
        '''
        action_values = self.q_table[state]
        eps_action = self.next_action
        return eps_action, action_values
```

Q Learning

This code block deals with the state action value update step according to the Q Learning algorithm. We choose an action according to the softmax policy derived from our Q Table, observe the next state and reward. Now, we update the current state action value function based on the maximum possible state action value function.

```
# scripts/agents.py
class QLearningAgent:
    .
    .
    .
    def step(self, state, action, reward, next_state, done):
        '''Updates Q-values according to the Q-learning
        algorithm
        '''
        q_sa = self.q_table[state, action]

        # Maximization of Q-value over actions
        q_next_sa = np.max(self.q_table[next_state])

        # Q-Learning update
        self.q_table[state, action] = (
            q_sa + self.LR*(reward + self.GAMMA*q_next_sa - q_sa)
        )

    def act(self, state):
        '''Takes an action from current state based on
        softmax policy according to current Q values.
        '''
        action_values = self.q_table[state]
        softmax_action = softmax(
            action_values=action_values,
            action_size=self.action_size,
            tau=self.tau
        )
        return softmax_action, action_values
```

Tile Coding

We represent the states with their tile coordinates in multiple tilings hashed to give a single list of hashes. These hashes correspond to the indices in our Q Table which performs coarse coding (multiple updates for the same observation), resulting in a much better generalization. The following code blocks depict the algorithms used. We have based our implementation based on the description in section 9.5.4 of Reinforcement Learning: An Introduction. Further we have chosen to opt for asymmetrical tiling and used the offset vectors as the set of first n odd integers, where n is the number of dimensions of the state space, this choice is based on the work of Miller and Glanz (1996).

```
# scripts/tilecoding.py
class TileCoder:
    ''' Discretizes continuous states into discrete bins
```

```
    with multiple tilings for state aggregation
    Based on the description in section 9.5.4 of Reinforcement Learning: An Introduction
    '''
    def __call__(self, x):
        eps = 1e-10
        # Finding coordinates of states in different tilings
        tile_coords = ((x - self.lower_lim) / (self.tile_width + eps)
                       + self.offsets).astype(np.int32)

        # Hashing the tile coords into a single number per tiling, size (num_tilings,1)
        tile_coord_hash = (
            np.dot(self.base_hash, tile_coords.T)
        )
        # Return a list of hashes of the size (num_tilings,1)
        return list(self.base_tile_index + tile_coord_hash)

class QTable:
    '''Maintains a 2 dimensional Q Table to store multiple representations
    of a state from different tilings by hashing. Eases retrieval and
    update of Q-values.
    '''
    def __getitem__(self, key):
        """To get q_value = q_table[state] or q_value = q_table[state, action]
        """
        if isinstance(key, tuple) and len(key) == 2:
            state, action = key
            idx_s = self.tile_coder(state)
            return self.table[idx_s, action].mean()
        else:
            state = key
            idx_s = self.tile_coder(state)
            return self.table[idx_s].mean(axis=0)

    def __setitem__(self, key, value):
        """To set q_table[state, action] = value
        """
        if not (isinstance(key, tuple) and len(key) == 2):
            raise ValueError("Key must be a tuple (state, action)")

        state, action = key
        print(state, action)
        idx_s = self.tile_coder(state)
        self.table[idx_s, action] = value
```

Training

This code block consists of the main training loop for the agent, it handles the agent's actions, stepping the environment forward and also updating the agent's parameters for learning. Further this code block also takes care of logging the results which are subsequently used to perform analysis and also hyperparameter tuning.

```
# scripts/training.py
class Trainer:

    def training(self, env, agent, n_episodes=10000, process_training_info=lambda *args, **
    kwargs: (False, {})):
        """
        To train an agent in the given environment.

        Args:
            - env: The environment for training.
            - agent: An agent with `.step()`, `.act()`, and `.update_agent_parameters()`.
            - n_episodes (int, optional): Number of training episodes. Defaults to 10000.
            - process_training_info (function, optional): Runs after each episode.
                - First return value must be a `bool` for early stopping.
                - Second return value must be a `dict` to update the progress bar's postfix.

        Returns:
```

```
            - dict: Summary of the training process.
        """

        # Initialize variables for logging here

        progress_bar = tqdm(range(1, n_episodes+1), desc="Training")

        # Training loop
        for i_episode in progress_bar:
            state, _ = env.reset()
            score = 0
            total_reward = 0
            terminated, truncated = False, False
            episode_history = []

            # Running an episode
            while not (terminated or truncated):
                action, action_vals = agent.act(state)
                next_state, reward, terminated, truncated, _ = env.step(action)
                episode_history.append((state, action, reward, next_state))
                agent.step(state, action, reward, next_state, terminated)
                state = next_state
                score += self.compute_score(reward)
                total_reward += reward

            agent.update_agent_parameters()

            # Code for logging here

        return {
            "computation_time": end_time - begin_time,
            "scores": np.array(history_scores),
            "total_rewards": np.array(history_total_rewards)
        }

    def compute_score(self, reward):
        return reward
```

Metrics and Hyperparameters

To quantitatively compare the performance of the algorithms in an environment we have used the following two metrics:

- **Cumulative regret** : The sum of difference between return assuming optimal policy and the 5 run average return at each step. For this assignment we have tuned the hyperparameters to minimize the regret in all experiments.
- **Maximum rolling average score** : The maximum of the rolling average return over the past 100 episodes averaged over 5 runs

Further, based on the description in section 1, the agents are characterized by a few hyperparameters:

- **num_tilings_per_feature** : Determines the total discrete bins dividing the continuous state-space.
- **num_tilings** : Determines the total number of overlapping grids used in while using Tile Coding.
- **learning_rate** : The step size with which the Q-values are updated.
- **eps_start** : The starting value of ϵ in the ϵ-greedy policy.
- **eps_end** : The minimum value ϵ can take while training.
- **tau_start** : The starting value of τ in the softmax policy.
- **tau_end** : The minimum value τ can take while training.

- **decay_type** : Determines the rate at which the parameter ϵ or τ is decayed over episodes, this chosen either to be linear or exponential for this case.
- **frac_episodes_to_type** : Determines the fraction of total number of episodes where ϵ or τ is decayed, after which these parameters are held at their minimum values.

  We have tuned the hyperparameters using `wandb` to minimize regret in all experiments, by performing a `bayesian serach` over the search-space of hyperparameters. The exact configurations of the serach-space for the hyperparameters can be found in the `config/` folder.

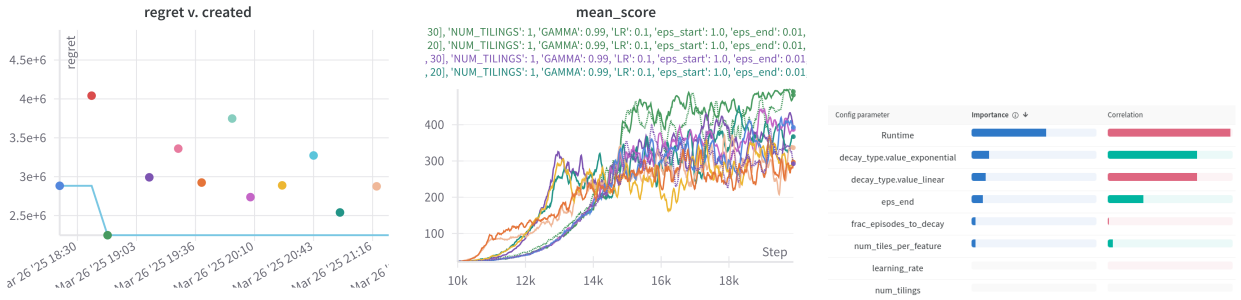
CartPole-v1

This environment's observation space consists of four continuous state variables, i.e. cart position, cart velocity, pole angle, and pole angular velocity. These variables range within $[-4.8, -\infty, -0.41887903, -\infty]$ to $[4.8, \infty, 0.41887903, \infty]$.

To address this environment, we process the state as follows. We first clip each variable to the range $[-5, 5]$ and then apply tile coding for discretization before passing it to the agent, as described in subsection (iv). We have observed empirically that the loss in information due to clipping does not lead to significant performance loss and yields satisfactory results in practice. We train the agent over a course of 10000 episodes.

Part (i): SARSA

To tune the hyperparameters we ran a parameter sweep consisting of 20 samples using `wandb` and performed a `bayesian serach` over the search-space of hyperparameters to minimize average cumulative regret, from these we selected the top 3 configurations, presented in Table 1.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 1: Tuning for CartPole-SARSA

Hyperparameters	Top	2nd Best	3rd Best
num_tiles_per_feature	30	20	30
num_tilings	1	1	1
learning_rate	0.1	0.1	0.1
eps_start	1	1	1
eps_end	0.01	0.01	0.01
decay_type	linear	linear	linear
frac_episodes_to_decay	0.3	0.5	0.5
Cumulative regret	2470134.6	2671750.8	2710446.4
Max rolling average score	443.366	444.438	449.752

Table 1: Top 3 Hyperparameter configurations for CartPole - SARSA

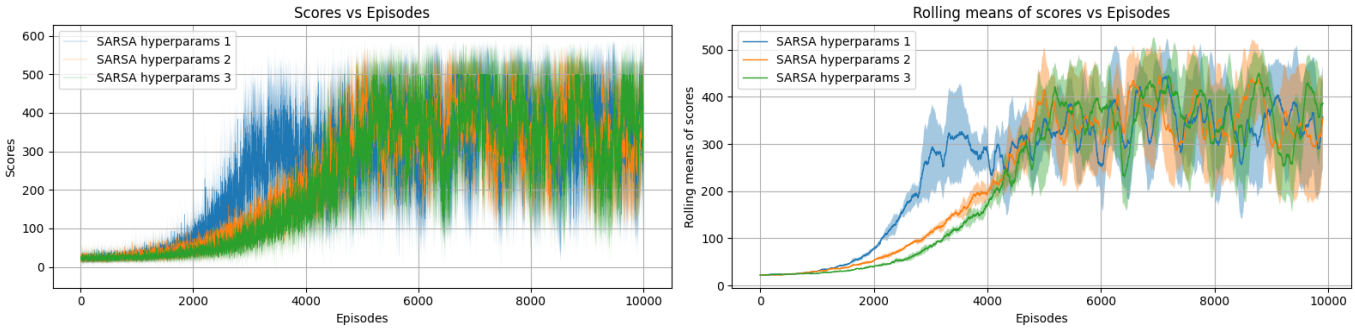
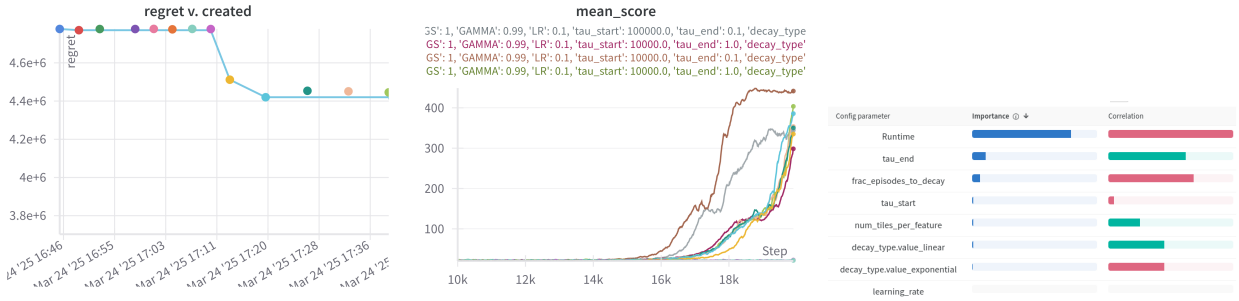


Figure 2: CartPole SARSA experiments for Top 3 Hyperparameter configurations

Part (ii): Q-Learning

Similar to SARSA case, we ran another sweep for Q-Learning and selected the top 3 configurations from 20 samples, see Table 2. For each sample we averaged the scores and cumulative regret over 5 runs.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 3: Tuning for CartPole-Q-Learning

Hyperparameters	Top	2nd Best	3rd Best
num_tiles_per_feature	30	20	30
num_tilings	1	1	1
learning_rate	0.1	0.1	0.1
tau_start	10000	10000	1000
tau_end	0.1	0.1	0.1
decay_type	exponential	exponential	exponential
frac_episodes_to_decay	0.7	0.7	1.0
Cumulative regret	2847478.6	3285443.4	3403292.0
Max rolling average score	432.358	340.840	453.070

Table 2: Top 3 Hyperparameter configurations for CartPole - Q-Learning

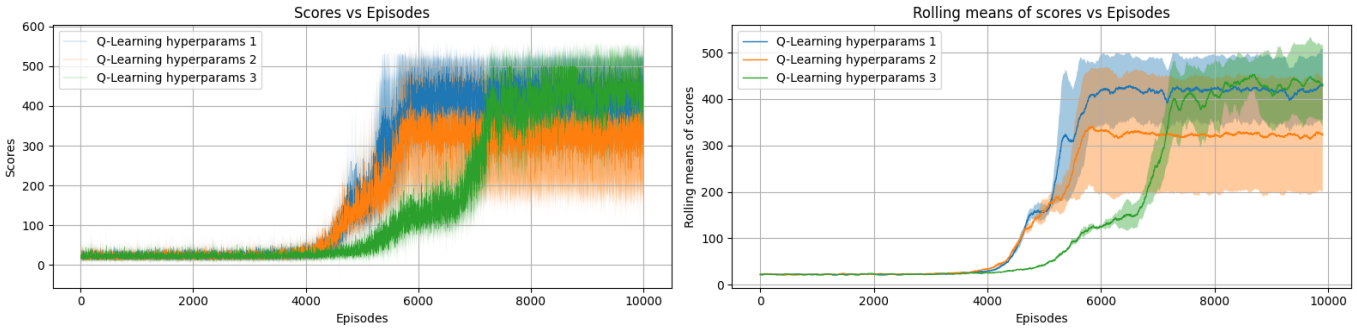


Figure 4: CartPole Q-Learning experiments for Top 3 Hyperparameter configurations

Part (iii): Comparison and Inferences

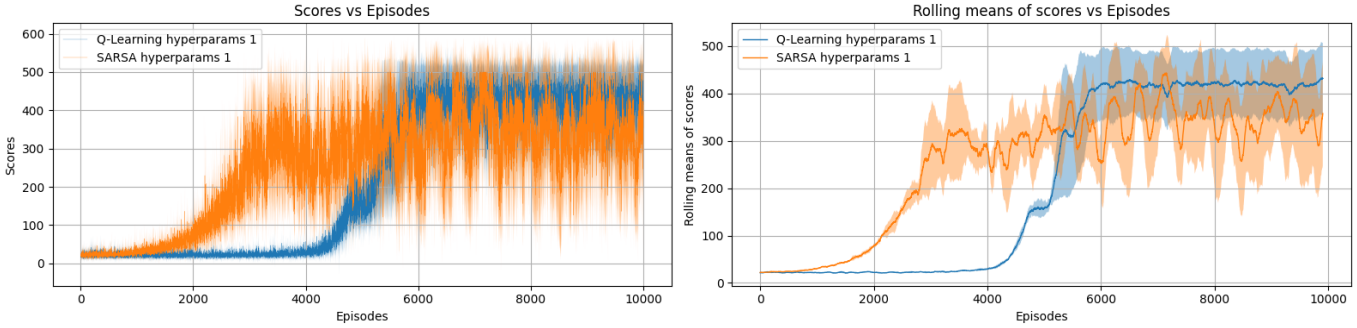


Figure 5: Comparing SARSA and Q-Learning for Top Hyperparameter configurations

In this experiment, we observe that SARSA initially outperforms Q-Learning. However, after approximately 7000 episodes, Q-Learning surpasses SARSA by a significant margin. This can be attributed to the fact that Q-Learning is an off-policy algorithm, where the exploratory nature of the softmax policy with a high τ leads to greater regret.

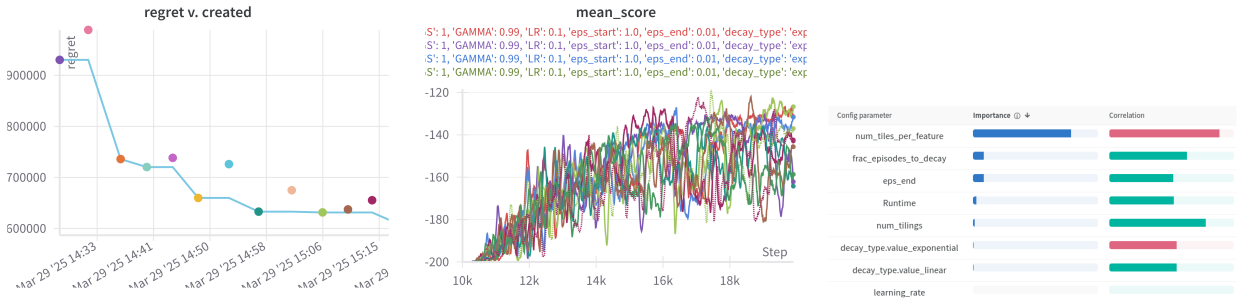
As τ decays to its final value (here 0.1) around 7000 episodes (as frac_episodes_to_decay $= 0.7$), the Q-Learning algorithm has effectively learned the optimal policy. At this point, it shifts toward exploitation, using the optimal Q-values to outperform SARSA.

MountainCar-v0

The environment's observation space consists of two continuous state variables: the car's position along the x-axis and its velocity. These values range from $[-1.2, -0.07]$ to $[0.6, 0.07]$. As before, we apply tile coding for discretization before passing the state to the agent, as described in subsection (iv).
To encourage exploration, we initialize all Q-values to 0, an optimistic value. This indirectly incentivizes the agent to explore previously un-explored actions since, in this experiment, all Q-values for states action pairs are negative, as all rewards are negative in this environment. We train the agent over a course of 10000 episodes.

Part (i): SARSA

Similar to the previous environment, to tune the hyperparameters we ran a parameter sweep consisting of 20 samples using `wandb` and performed a `bayesian serach` over the search-space of hyperparameters to minimize cumulative regret, from these we selected the top 3 configurations, presented in Table 3.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 6: Tuning for MountainCar-SARSA

Hyperparameters	Top	2nd Best	3rd Best
num_tiles_per_feature	20	20	20
num_tilings	1	1	4
learning_rate	0.1	0.1	0.1
eps_start	1	1	1
eps_end	0.01	0.01	0.01
decay_type	exponential	exponential	exponential
frac_episodes_to_decay	0.1	0.3	0.5
Cumulative regret	608150.6	618654.0	685039.0
Max rolling average score	-135.624	-135.514	-138.364

Table 3: Top 3 Hyperparameter configurations for MountainCar - SARSA

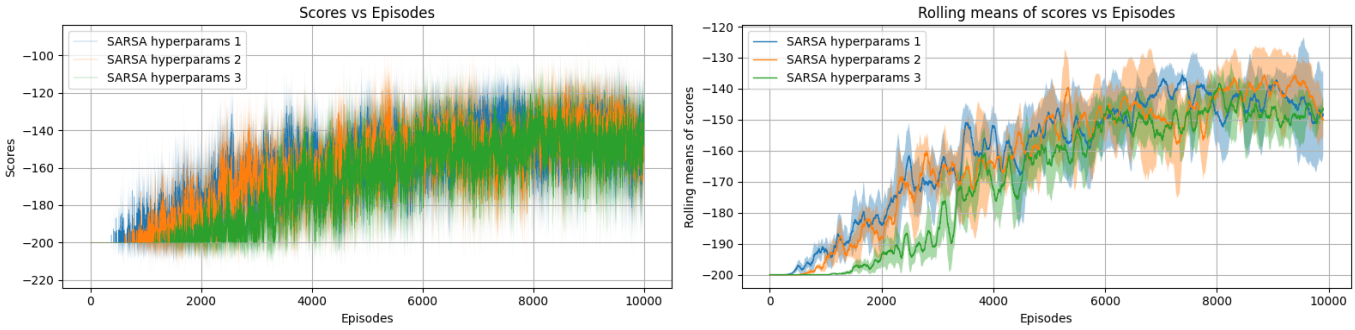
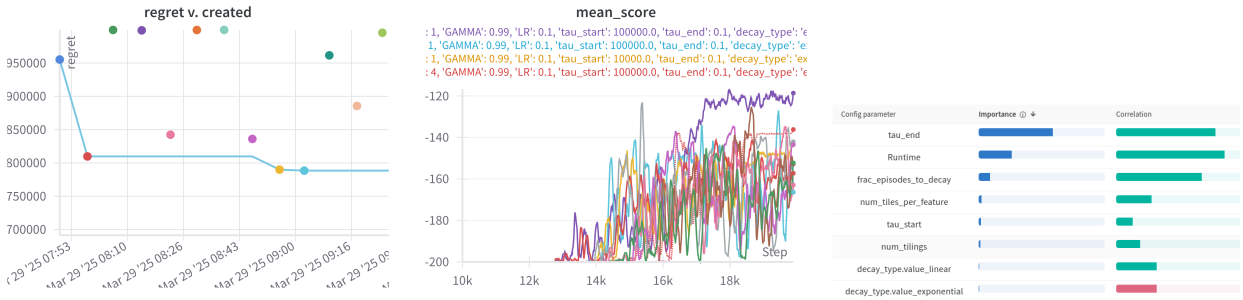


Figure 7: MountainCar SARSA experiments for Top 3 Hyperparameter configurations

Part (ii): Q-Learning

Similar to the SARSA case, we ran another sweep for Q-Learning and selected the top 3 configurations from 20 samples, see Table 4.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 8: Tuning for MountainCar-Q-Learning

Hyperparameters	Top	2nd Best	3rd Best
num_tiles_per_feature	20	10	20
num_tilings	1	1	4
learning_rate	0.1	0.1	0.1
tau_start	100000	100000	100000
tau_end	0.1	0.1	0.1
decay_type	exponential	exponential	exponential
frac_episodes_to_decay	0.3	0.5	0.3
Cumulative regret	758822.0	809055.8	784852.2
Max rolling average score	-130.074	-146.634	-144.930

Table 4: Top 3 Hyperparameter configurations for MountainCar - Q-Learning

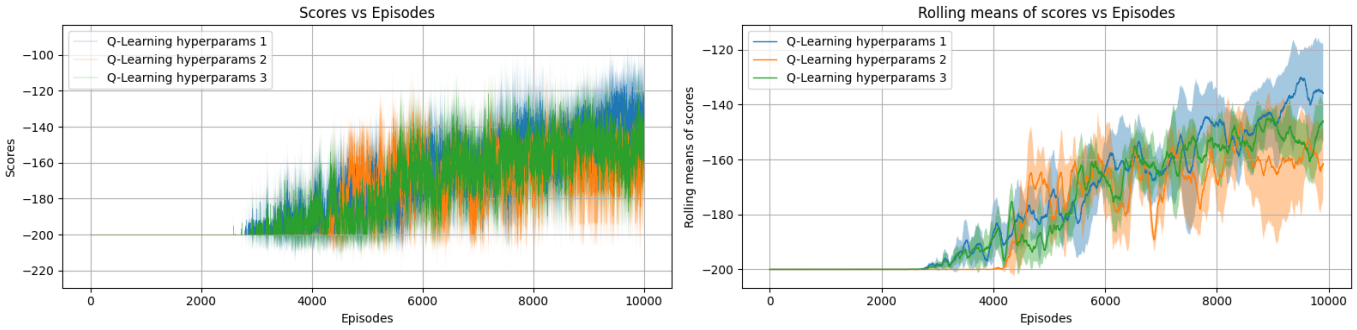


Figure 9: MountainCar Q-Learning experiments for Top 3 Hyperparameter configurations

Part (iii): Comparison and Inferences

While solving the MountainCar environment, we observed that the agent takes approximately 1500 episodes to explore and reach the goal for the first time. We infer from this although initializing Q-values with optimistic values encourages exploration, it may take a long time before the agent performs expected actions to reach the goal. After reaching the goal once, the agent's regret decreases as it learns to achieve the goal more efficiently.

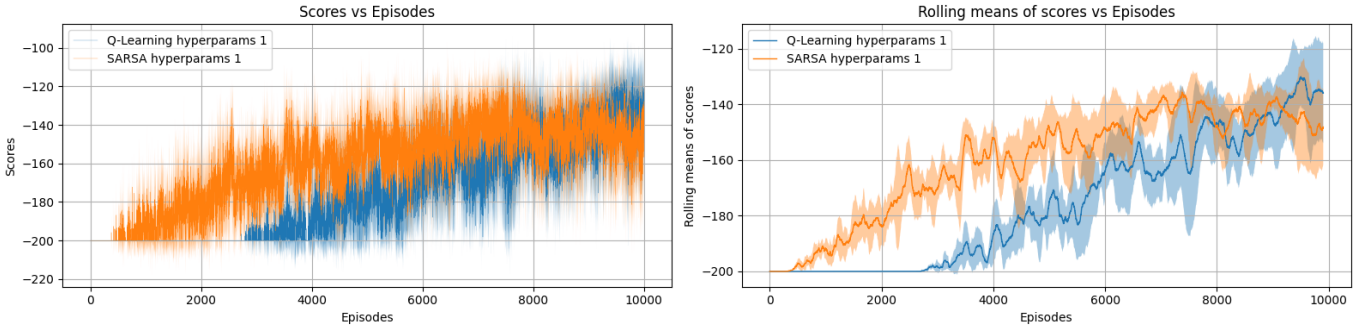


Figure 10: Comparing SARSA and Q-Learning for Top Hyperparameter configurations

Part (iv): Solving MountainCar with Reward Shaping

As discussed in the previous section, the agent takes a significant amount of time to reach the goal for the first time. To address this, we consider an alternative approach using reward shaping. Based on our experiments, we observed that the agent must be incentivized to accumulate higher velocities (or, informally, gain more momentum). This helps the agent escape the valley and reach the goal. Therefore, we introduce a new reward scheme:

$$R_{t+1} = R_v - 1$$

where R_v represents the reward for taking actions that accelerate the agent in its direction of motion, and R_{t+1} is the reward for taking an action at time step t. Here R_v is defined as:

$$R_v = \begin{cases} -k \cdot v_t, & \text{if action} = 0 \text{ (accelerate left)} \\ 0, & \text{if action} = 1 \text{ (no acceleration)} \\ k \cdot v_t, & \text{if action} = 2 \text{ (accelerate right)} \end{cases}$$

where k is a positive constant that determines the weight for incentivizing acceleration in the direction of motion, and v_t is the velocity of the agent at time step t. For this experiment we set k to be 100.
In a more compact form:

$$R_v = k \cdot (\text{action} - 1) \cdot v_t$$

For demonstration purposes, we use the same hyperparameters as described in Table 3 and Table 4.

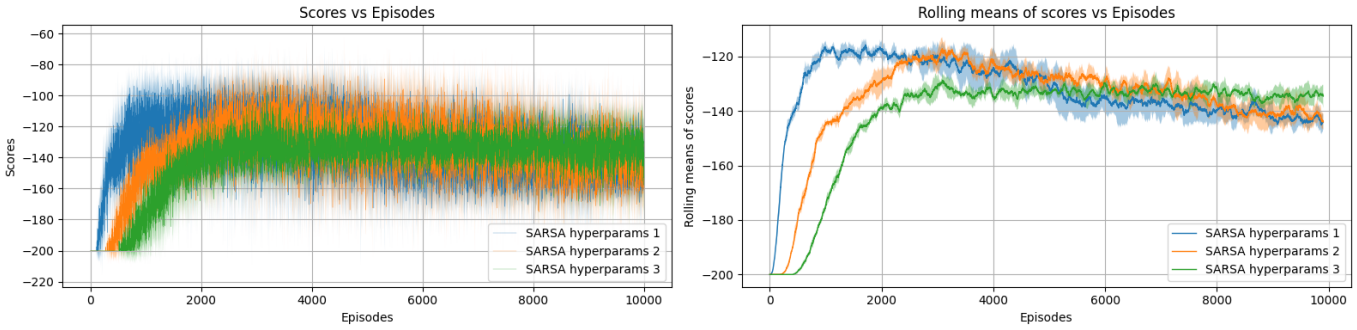


Figure 11: MountainCar SARSA experiments (with Reward shaping) for Top 3 Hyperparameter configurations

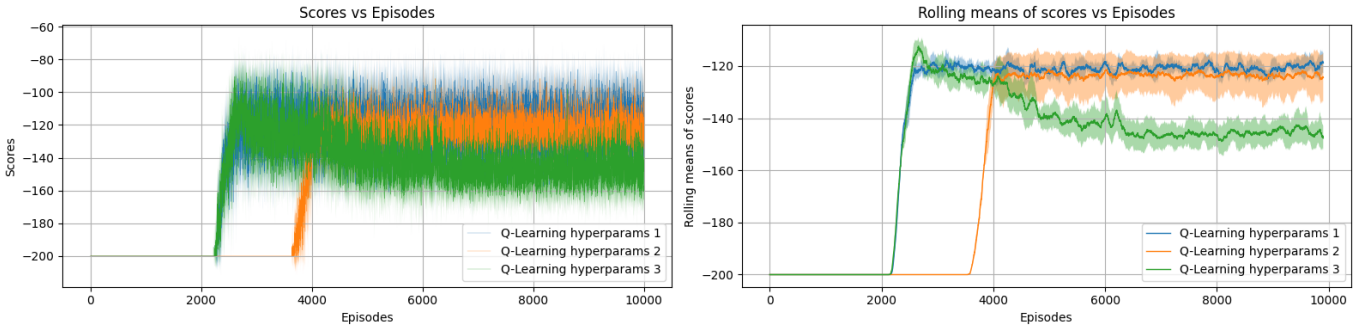


Figure 12: MountainCar Q-Learning experiments (with Reward shaping) for Top 3 Hyperparameter configurations

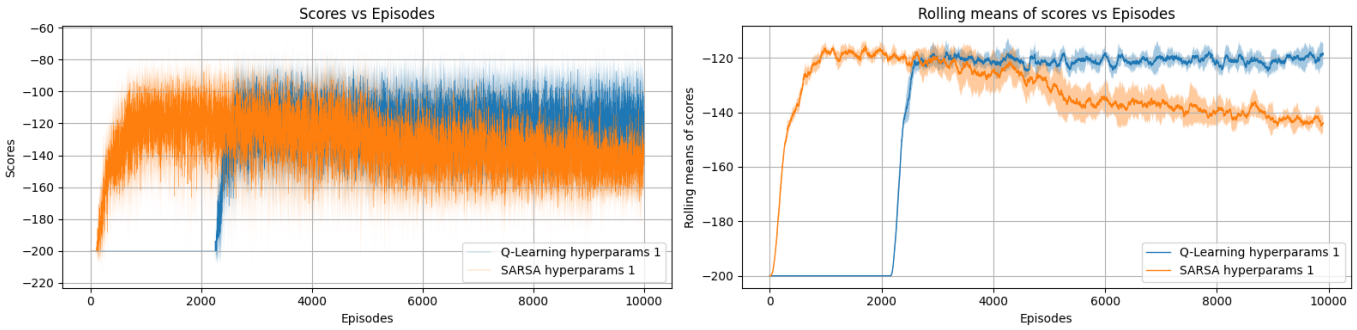
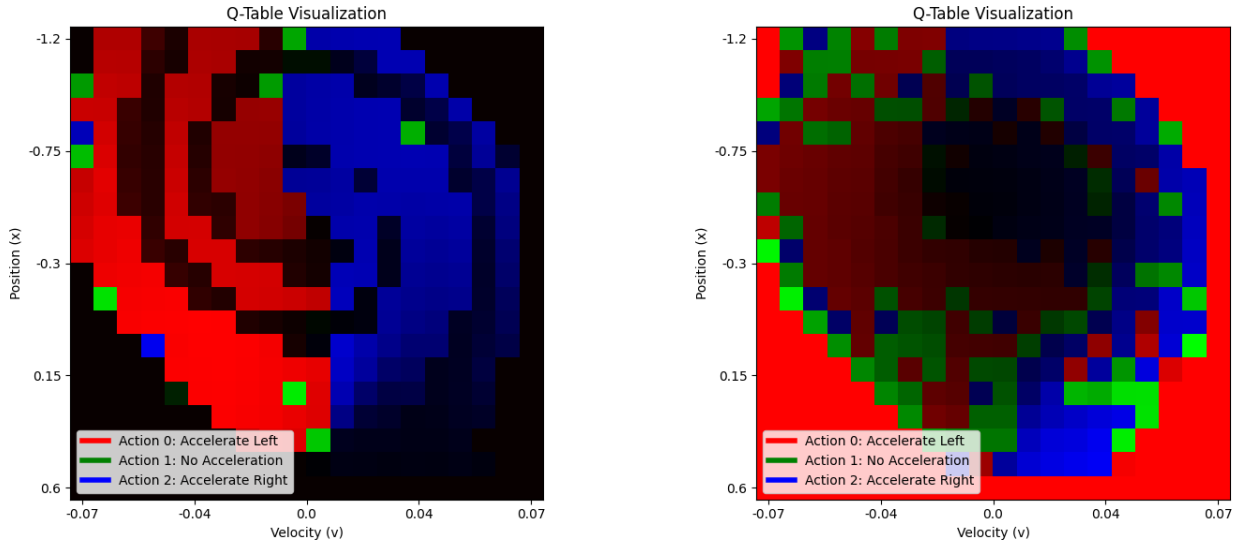


Figure 13: Comparing SARSA and Q-Learning (with Reward shaping) for Top Hyperparameter configurations

We observe the fact that the variance across five runs drastically decreases when the rewards are shaped. We can infer that the agent is consistently learning similar policies. We also observe a drastic improvement in rolling average performance, leading us to believe that the agent has learned a better policy than without shaping. SARSA seems to have a hold on the top policy for some time before it loses its grip. Q-Learning joins the race late, but clings to the top policy once it learns it. We believe that the drop in performance and stagnation at a sub-optimal score can be attributed to the agents optimizing for a different objective determined by R_v. As a result, an agent may spend more time oscillating in the valley to accumulate positive rewards before gaining enough velocity to escape, leading to sub-optimal behavior. We believe that this problem can be solved to some degree by tuning the weight k described earlier.



(a) Q-Table with reward shaping

(b) Q-Table with original reward scheme

Figure 14: Visualizing the Q-Tables

To further analyze the policies learned by the agents with and without reward shaping, we plotted the optimal action for each state (position, velocity of the car), as shown in Figure 14.

We observed a spiral-shaped structure in the Q-table, centered approximately at the starting state. Additionally, we inferred that the Q-table using the original reward scheme has a higher level of noise, which might explain the suboptimal performance. On the other hand, the Q-table with reward shaping displays less noise and a more structured pattern, where the agent follows the spiral outwards, while taking less suboptimal steps toward the goal.

MiniGrid-Dynamic-Obstacles-5x5-v0

The environment consists of the following observation space:

- **direction** : Represents the agent's current direction.
- **image**: A partially observed 7×7 grid, where each tile represents an object, which may be "wall", "empty", "ball", or "goal".
- **mission**: Specifies the environment's mission structure (as a string).

In this environment, the agent always starts from the same tile, and the goal is fixed at a specific location. To solve this environment, we apply state aggregation and extract three key features from the observation that we believe are sufficient for reaching the goal:

- **agent_direction**: The direction the agent is currently facing, represented as an integer in the range $[0, 3]$.
- **agent_pos_encoding**: The agent's position is inferred from the partially observed grid by counting the number of tiles in front of and to the right of the agent. This is then encoded as a single number in the range $[0, 8]$.
- **agent_path_clear**: Determines whether the path directly ahead is clear for movement (i.e., no "wall" or "ball"). A value of 0 indicates the path is blocked, while 1 indicates it is clear.

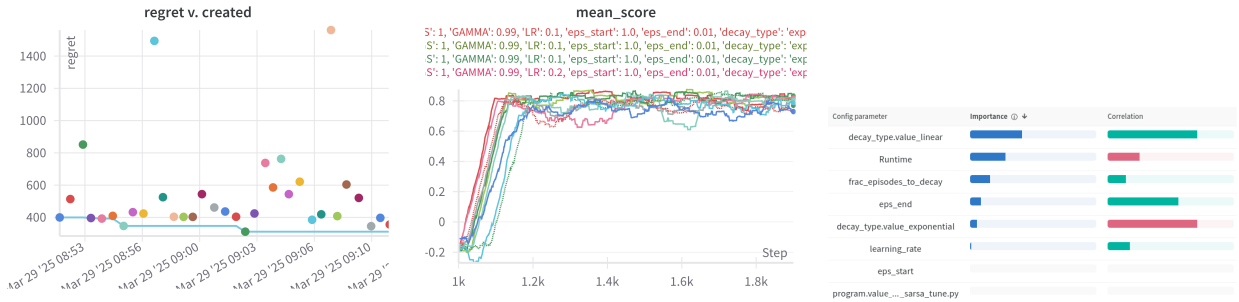
By aggregating states, we reduce the state space to:

$$4 \times 9 \times 2 = 72 \text{ states}$$

This allows for effective generalization over a much larger state space, consisting of all possible locations of obstacles and the agent. Since the goal is always in a fixed position, we do not explicitly compute its location, instead, the agent is left to learn it over episodes. We train the agent over a course of 1000 episodes.

Part (i): SARSA

Similar to the previous environment, to tune the hyperparameters we ran a parameter sweep consisting of 50 samples using `wandb` and performed a `bayesian serach` over the search-space of hyperparameters to minimize cumulative regret, from these we selected the top 3 configurations, presented in Table 5.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 15: Tuning for Minigrid-SARSA

Hyperparameters	Top	2nd Best	3rd Best
learning_rate	0.1	0.1	0.1
eps_start	1	1	1
eps_end	0.01	0.01	0.01
decay_type	exponential	exponential	exponential
frac_episodes_to_decay	0.1	0.3	1
Cumulative regret	379.268	433.254	854.887
Max rolling average score	0.770	0.836	0.791

Table 5: Top 3 Hyperparameter configurations for Minigrid - SARSA

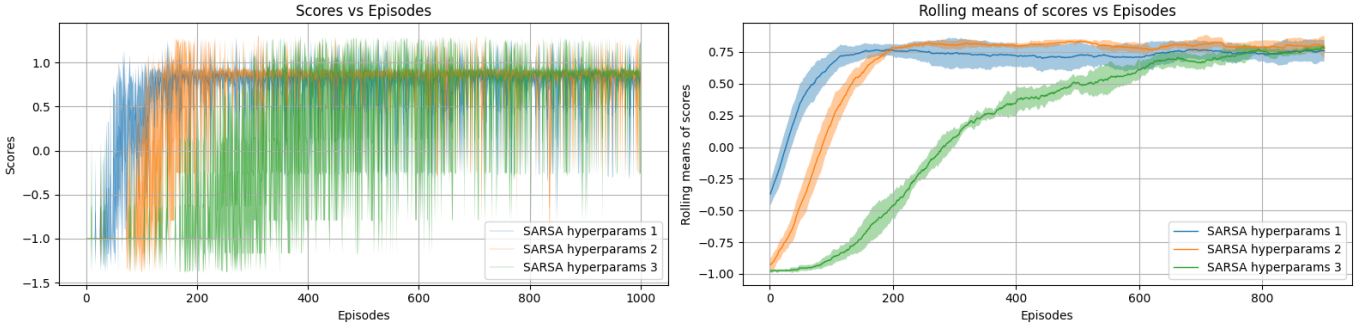
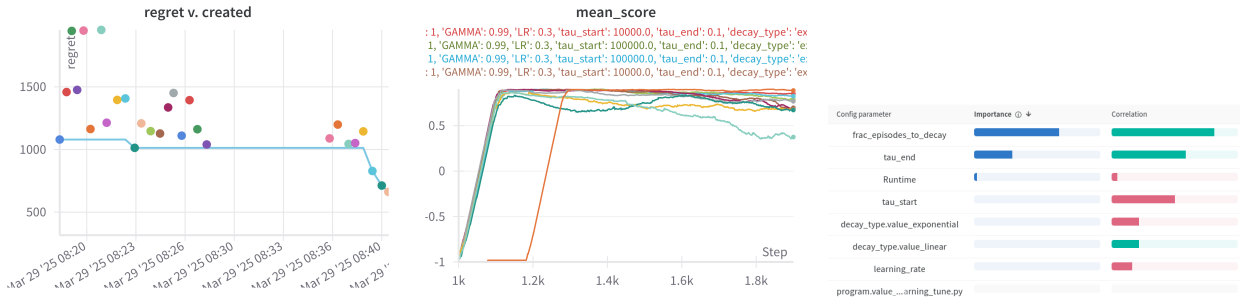


Figure 16: Minigrid SARSA experiments for Top 3 Hyperparameter configurations

Part (ii): Q-Learning

Similar to the SARSA case, we ran another sweep for Q-Learning and selected the top 3 configurations from 50 samples, see Table 6.



(a) Regret while tuning (b) Rolling average score (c) Hyperparameter correlation with regret

Figure 17: Tuning for Minigrid-Q-Learning

Hyperparameters	Top	2nd Best	3rd Best
learning_rate	0.3	0.3	0.3
tau_start	100000	10000	100000
tau_end	0.1	0.1	0.1
decay_type	exponential	exponential	exponential
frac_episodes_to_decay	0.1	0.1	0.3
Cumulative regret	382.606	412.651	735.958
Max rolling average score	0.872	0.875	0.844

Table 6: Top 3 Hyperparameter configurations for Minigrid - Q-Learning

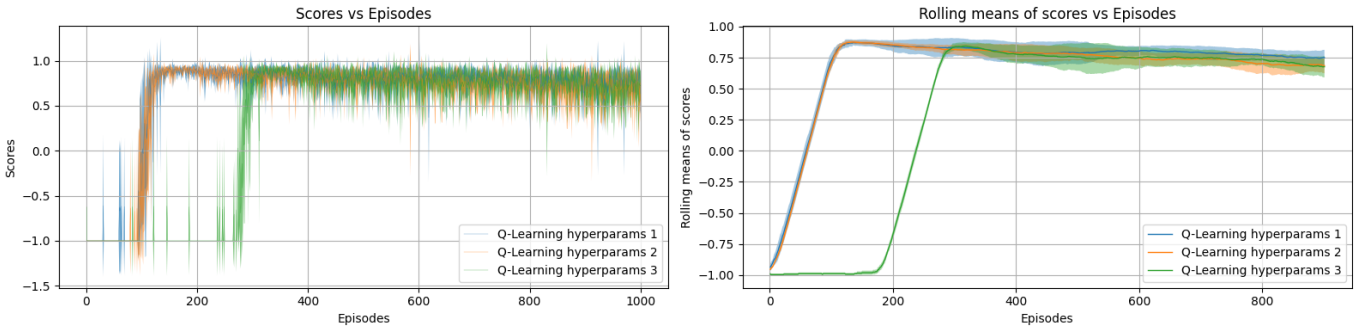


Figure 18: Minigrid Q-Learning experiments for Top 3 Hyperparameter configurations

Part (iii): Comparison and Inferences

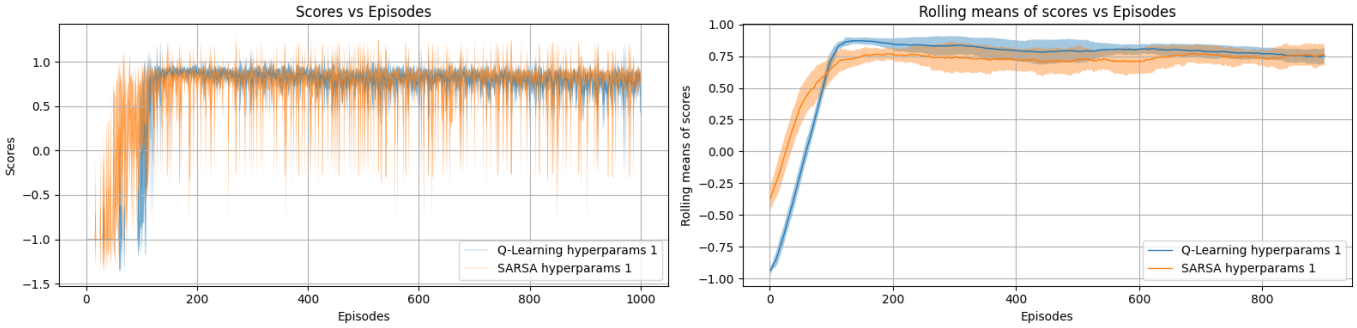


Figure 19: Comparing SARSA and Q-Learning for Top Hyperparameter configurations

It seems clear that the score we can expect is around 0.8. The speed which we can reach it is represented in the plots. The policy doesn't seem too complex to learn, as is evident from the plots and the corresponding *frac_episodes_to_decay*, we need to start exploitation as soon as possible. It is evident from the plot that in the initial stages, Q-Learning takes risky steps (as it evaluates a different policy) and hence converges a bit faster.