KTH Royal Institute of Technology
Course: EL2805 - Reinforcement Learning

Saga Tran | 19991105 – 2182
Anh Do | 20020416 – 2317

Computer Lab 1

Date: 2025/12/05 18.00

Contents

1 Problem 1

1.1 Basic Maze

1.1.1 (a)

The problem is modeled as a finite horizon Markov Decision Process (MDP) defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$.

1.1.1.1 State Space ($\mathcal{S}$)

The state space is defined by the joint position of the Player (P) and the Minotaur (M) within the grid.

$$\mathcal{S} = \{(p, m) \mid p \in \mathcal{C}, m \in \mathcal{D}\} \tag{1}$$

where $\mathcal{D}$ represents the set of all discrete cells in the maze grid, while $\mathcal{C}$ represents the set of all discrete non-wall cells in the maze grid (e.g., p and m have each coordinates (x, y) and $\mathcal{C} \subset \mathcal{D}$). The terminal states are:

- **Win State:** $s_{win} = \{(p, m) \in \mathcal{S} \mid p = B, m \neq B\}$. The player reaches exit B without being caught.
- **Loss State:** $s_{loss} = \{(p, m) \in \mathcal{S} \mid p = m\}$. The Minotaur occupies the same cell as the player (caught).

Once in a terminal state (e.g., s_{win} or s_{loss}) the game transitions to the absorbing state s_{done}. This is mainly for handling the accumulation of rewards at s_{win}.

1.1.1.2 Action Space ($\mathcal{A}$)

At each time step t, the player can choose to move to an adjacent cell or stay still.

$$\mathcal{A} = \{\text{Stay, Up, Down, Left, Right}\} \tag{2}$$

Note: Moves are constrained by the maze walls for the player.

1.1.1.3 Transition Probabilities ($\mathcal{P}$)

The state transitions from $s_t = (p_t, m_t)$ to $s_{t+1} = (p_{t+1}, m_{t+1})$ occur simultaneously. The transition probability factorizes as:

$$P(s_{t+1} \mid s_t, a_t) = P(p_{t+1} \mid p_t, a_t) \times P(m_{t+1} \mid m_t) \tag{3}$$

- **Player Dynamics:** Deterministic. p_{t+1} is the result of action a_t applied to p_t. If the move hits a wall or is invalid, $p_{t+1} = p_t$.
- **Minotaur Dynamics:** Random Walk. The Minotaur moves to any adjacent cell within the grid boundaries with uniform probability. It cannot stand still and ignores internal walls.

$$P(m_{t+1} \mid m_t) = \frac{1}{|N(m_t)|} \quad \text{if } m_{t+1} \in N(m_t) \tag{4}$$

where $N(m_t)$ is the set of valid neighbors (Up, Down, Left, Right) for the Minotaur at position m_t, restricted only by the exterior borders of the maze.

Observe that some transitions result in a winning or losing state based on the conditions defined in 1.1.1.1. In these cases, if an action leads to multiple winning outcomes, we must sum their probabilities.

To handle the transitions to the absorbing states correctly, we define:

$$P(s_{done} \mid s_t, a_t) = \begin{cases} 1 & \text{if } s_t \in s_{win} \cup s_{loss} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

1.1.1.4 Reward Function ($\mathcal{R}$)

To maximize the probability of exiting the maze, we define the reward to be 1 only upon being at the "Win" state, and 0 otherwise. This transforms the expected cumulative reward into the probability of success.

$$R(s_t, a_t) = \begin{cases} 1 & \text{if } s_t = s_{win} \text{ (Reached B, not eaten)} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

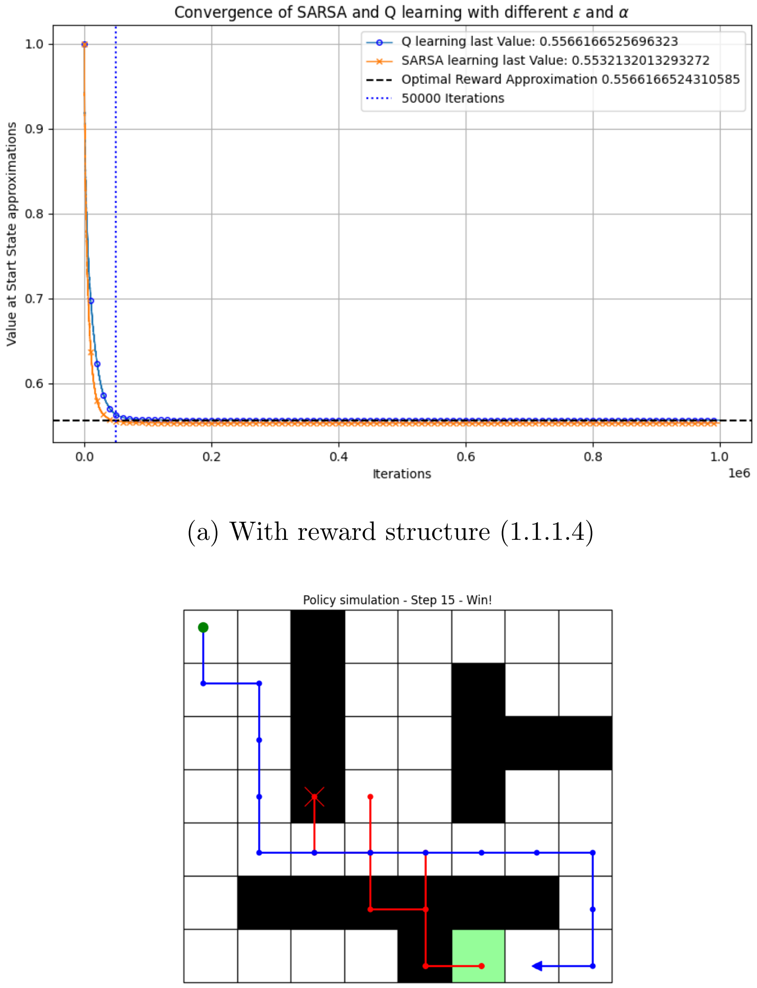
1.1.2 (b)

Yes, introducing the possibility for the minotaur to stand still makes the problem more difficult. This is because the agent must now account for an additional action of the minotaur, which increases the complexity of predicting its movements and planning a safe path to the goal. In the original problem, it is clear that we will never be eaten as long as we keep moving, due to the even steps between the player and the minotaur. However, allowing the minotaur to stand still breaks this parity, making it harder to find an optimal policy. In the extended problem, standing still could now become an optimal action in certain situations, further complicating the model.

1.2 Dynamic Programming

1.2.1 (c)

Using Dynamic Programming, we computed the optimal policy for the basic maze setup.



(a) With reward structure (1.1.1.4)

(b) With penalized stepping

Figure 1: Optimal policies for the basic maze using Dynamic Programming. The green circle indicates the starting position of the player, while the red cross marks the Minotaur's starting position. The blue triangle points to the exit point B. (a) Policy under the standard sparse reward. (b) Policy when a step penalty is introduced to discourage loitering.

Figure 1a shows that Dynamic Programming successfully finds the optimal policy to reach the exit. The model sometimes stops or moves randomly, which might look strange, but it makes sense given the 20-step horizon. The agent only needs 15 steps to exit and 16 steps to claim the reward, so the extra steps do not affect the total reward. The reward function in 1.1.1.4 depends only on reaching the exit and does not penalize time. Therefore, the policy can make unnecessary moves in the beginning as long as it reaches the goal safely.

In contrast, Figure 1b illustrates the optimal policy when we modify the reward function to penalize each step taken by the player. We observe that the agent now takes a more direct path to the exit, avoiding unnecessary moves and reaching the goal in exactly 15 steps.

1.2.2 (d)

Utilizing the reward structure defined in 1.1.1.4, the probability of escaping the maze is calculated directly from the value function at the start state at initial time ($t = 0$) that was obtained via Dynamic Programming.

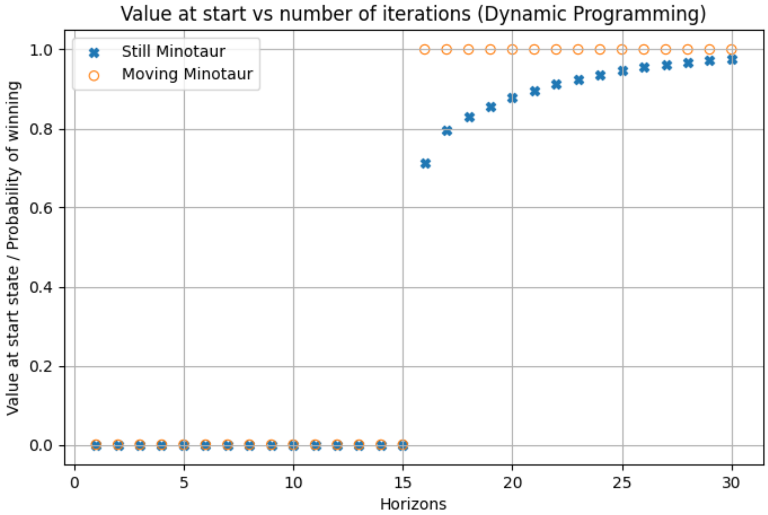


Figure 2: Probability of escaping the maze under different time horizons (T) using Dynamic Programming. The plot compares the scenarios where the Minotaur can and cannot stand still.

As shown in Figure 2, when the Minotaur cannot stand still, the escape probability reaches exactly 1 for time horizons $T \geq 16$. This sharp jump indicates that a deterministic winning policy exists. The strict movement constraints of the Minotaur allow the agent to exploit even step parity and guarantee an escape given sufficient time.

In contrast, when the Minotaur is allowed to stand still, the escape probability asymptotically approaches 1 but does not reach certainty within the observed finite horizons. Since the Minotaur retains the option to remain stationary, there always exists a non-zero probability that it will block the exit path for the duration of the episode, making a guaranteed escape impossible within a fixed time frame.

1.3 Value Iteration

1.3.1 (e)

We are given that the agent's life span L follows a geometric distribution with mean $\mu = 30$. The probability of dying at any time step t, denoted as p_{die}, is derived from the mean:

$$\mathbb{E}[L] = \frac{1}{p_{die}} = 30 \implies p_{die} = \frac{1}{30} \tag{7}$$

Consequently, the probability of surviving a single time step is $\gamma = 1 - p_{die} = \frac{29}{30}$.

To maximize the probability of exiting the maze alive, we must maximize the probability of reaching the goal state B at some time T multiplied by the probability of surviving the poison until time T. In the MDP framework, this survival probability acts identically to a discount factor. Therefore, we modify the MDP from problem 1.1.1 by introducing a discount factor $\gamma < 1$.

The tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ remains largely the same, with the following addition:

- **Discount Factor (γ):** We set $\gamma = \frac{29}{30} \approx 0.9667$.

Since the reward is only received once upon exiting (at time T), the expected return becomes $\mathbb{E}[\gamma^T \cdot 1]$. Since γ^T is exactly the probability of surviving T steps of poison, maximizing this value maximizes the probability of exiting alive. In this case the optimal $T^* = 15$. So the probability of exiting alive with the optimal policy would thus be $\gamma^{T^*} = \left(\frac{29}{30}\right)^{15} \approx 0.6014$.

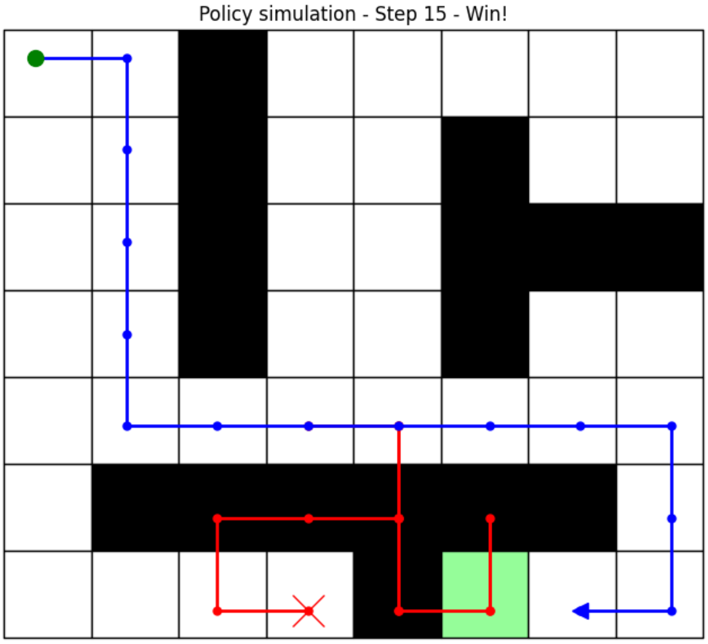


Figure 3: Optimal policy for the basic maze with poison using Value Iteration. The green circle indicates the starting position of the player, while the red cross marks the Minotaur's starting position. The blue triangle points to the exit point B.

As shown in Figure 3, the Value Iteration algorithm successfully computes the optimal policy for the environment with poison. We observe that the agent now strictly follows the most direct path to the exit, minimizing the number of steps taken to reduce the cumulative risk of succumbing to the poison. This behavior is quantitatively supported by the computed Value function at the start state when $t = 0$, which is 0.6014. This aligns with the theoretical survival probability for the minimum required

1.3.2 (f)

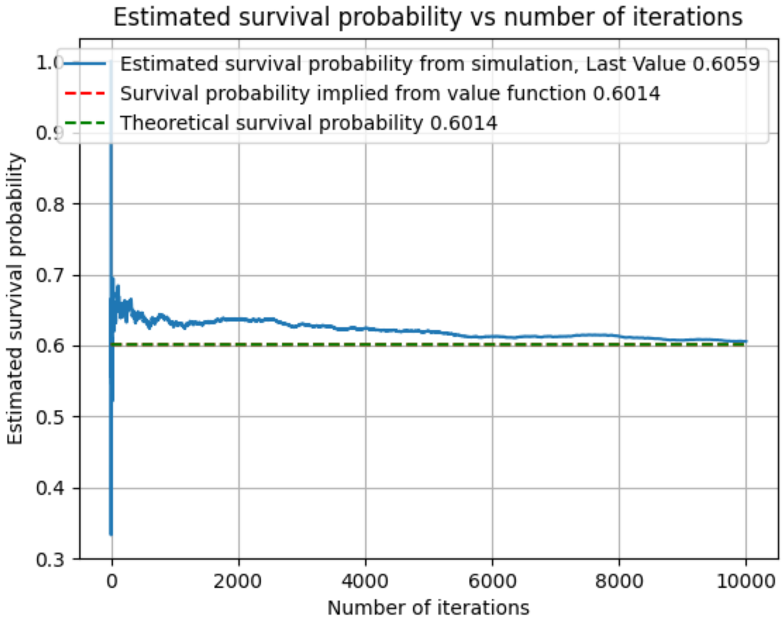


Figure 4: Estimated survival probability vs. number of iterations using Value Iteration and Monte Carlo simulation (cumulative sum averaging). The dashed lines represent the theoretical survival probabilities derived from the value function and direct calculation.

We can clearly see that this simulation estimate converges to the value function result of 0.6014, which confirms that our Value Iteration implementation is correct and that the derived optimal policy effectively maximizes the probability of exiting the maze alive.

1.4 Additional questions

1.4.1 (g)

1.4.1.1 On-policy vs. Off-policy Learning

The distinction between these learning methods lies in the relationship between the *behavior policy* (used to explore) and the *target policy* (being learned).

- **On-policy methods** (e.g., SARSA) estimate the value of the policy currently being used by the agent. The update rule depends on the action actually taken:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

  where a' is the action executed by the current policy, the same we used to get a.

- **Off-policy methods** (e.g., Q-learning) estimate the value of an optimal target policy independently of the agent's actions. The update rule uses the best possible next action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

This allows the agent to learn an optimal strategy even while exploring randomly, which means a' is chosen greedily instead and not by the current policy.

1.4.1.2 Convergence Conditions

For both algorithms to converge to Q^* with probability 1, the step sizes $\alpha_t(s,a)$ must satisfy the Robbins-Monro conditions:

$$\sum_{t=0}^{\infty} \alpha_t(s,a) = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2(s,a) < \infty \tag{8}$$

Additionally, specific exploration conditions are required:

- **SARSA (On-policy)** Requires all state-action pairs must be visited infinitely often, and the policy must become greedy in the limit (e.g., $\epsilon \to 0$).
- **Q-learning (Off-policy)** Requires only that all state-action pairs are visited infinitely often. The behavior policy does not need to become greedy.

1.4.2 (h)

To accommodate the new requirements (keys, smarter Minotaur, and poison), we modify the MDP tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ as follows:

1.4.2.1 State Space Expansion

We introduce a binary variable $k \in \{0,1\}$ to track possession of the keys. The new state is $s = (p, m, k)$, where p is the player's position, m is the Minotaur's position, and $k = 1$ implies the key has been retrieved from C. We also need to modify the winning state to require the key aswell as not being eaten:

$$s_{win} = \{(p, m, k) \in \mathcal{S} \mid p = B, m \neq B, k = 1\} \tag{9}$$

1.4.2.2 Discount Factor (Poison)

To model the geometric life expectancy of 50 steps, we set the discount factor γ to the survival probability:

$$\gamma = 1 - \frac{1}{50} = \frac{49}{50} = 0.98 \tag{10}$$

1.4.2.3 Transition Probabilities

The transition $P(s_{t+1}|s_t, a_t)$ now accounts for the Minotaur's aggressive behavior. Let $\mathcal{N}(m_t)$ be the set of valid neighbors for the Minotaur. Let $\mathcal{N}^*(m_t, p_t) \subset \mathcal{N}(m_t)$ be the subset of neighbors that minimizes the Manhattan distance to the player p_t, which is given by:

$$m_{t+1} \in \mathcal{N}^*(m_t, p_t), \quad d(m_{t+1}, p_t) = \min_{m \in \mathcal{N}(m_t)} d(m, p_t) \tag{11}$$

The probability of the Minotaur moving to a specific neighbor $m' \in \mathcal{N}(m)$ is given by:

$$P(m_{t+1} \mid m_t, p_t) = \frac{0.65}{|\mathcal{N}(m_t)|} + \mathbb{I}(m_{t+1} \in \mathcal{N}^*(m_t, p_t)) \cdot \frac{0.35}{|\mathcal{N}^*(m_t, p_t)|} \tag{12}$$

Here, the first term represents the random behavior (uniform over all directions), and the second term adds probability mass to the optimal "chasing" moves.

The key transitions to 1 deterministically as long as $s = (p = C, m \neq C, k = 0)$ for the rest it is inherited from the previous state:

$$k_{t+1} = \begin{cases} 1 & \text{if } p_t = C,\ m_t \neq C,\ k_t = 0 \\ k_t & \text{otherwise} \end{cases} \tag{13}$$

1.4.2.4 Reward Expansion

Although not explicitly needed, expanding the rewards to include key retrieval could help convergence. However, to keep in line with the reward structure in 1.1.1.4, we simply set this term to 0.

1.5 Q-Learning and Sarsa

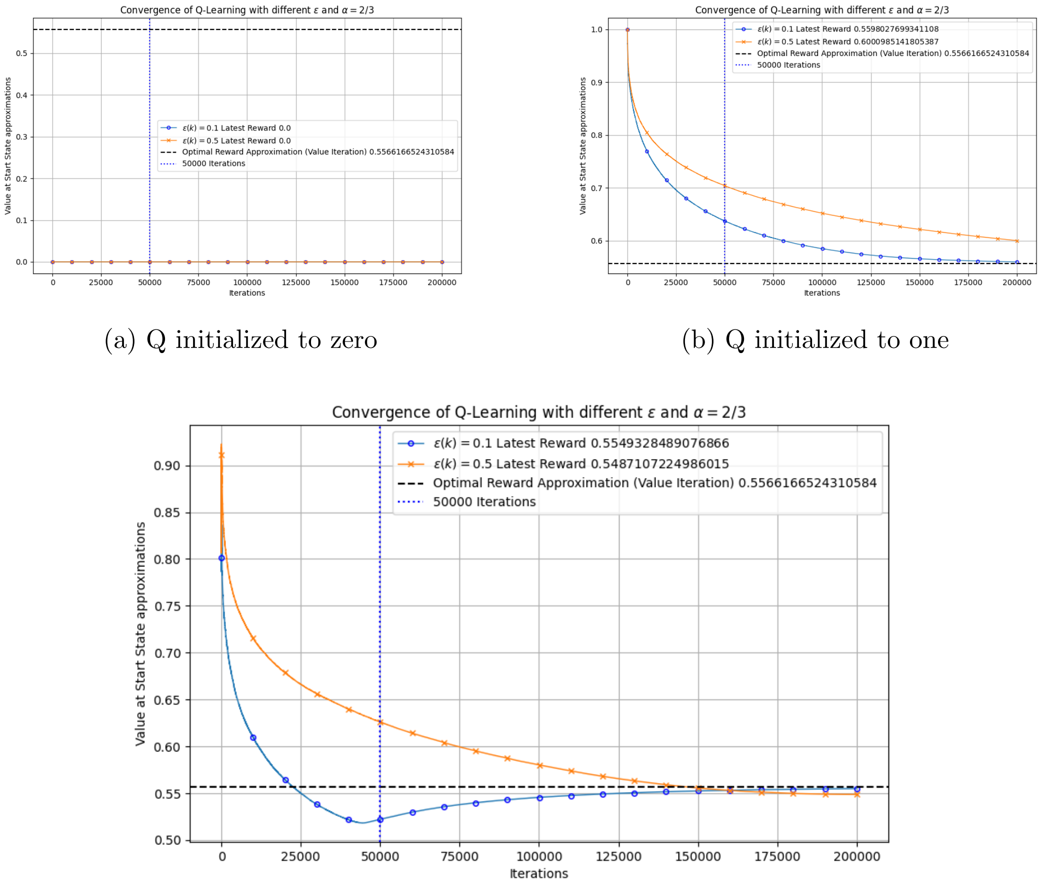
1.5.1 (i) BOUNS

1.5.1.1 1)

Algorithm 1 Q-learning

```
Require: Environment $\mathcal{E}$, Start State $s_{start}$, Discount $\gamma$, Episodes $K$
Initialize $Q(s,a) \sim U(0,1)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
Set $Q(\text{absorbing}, \cdot) \leftarrow 0$
Initialize visit counters $N(s,a) \leftarrow 0$ for all $s, a$
Initialize array $V_{starts}$ of size $K$
for episode $k = 1, 2, \ldots, K$ do
    $s \leftarrow s_{start}$
    $\varepsilon \leftarrow \epsilon(k)$                                ▷ Get current exploration rate
    while $s$ is not absorbing do
        Select action $a$:
        if random() $< \varepsilon$ then
            $a \leftarrow$ random action from $\mathcal{A}$
        else
            $a \leftarrow \arg\max_{a'} Q(s,a')$
        end if
        Observe reward $r(s,a)$ and next state $s'$
        $N(s,a) \leftarrow N(s,a) + 1$
        $\alpha \leftarrow$ learning_rate($N(s,a)$)
        Update Q-value:
        $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
        $s \leftarrow s'$
    end while
    $V_{starts}[k] \leftarrow \max_a Q(s_{start}, a)$
end for
return $Q, N, V_{starts}$
```

1.5.1.2 2)



(a) Q initialized to zero

(b) Q initialized to one

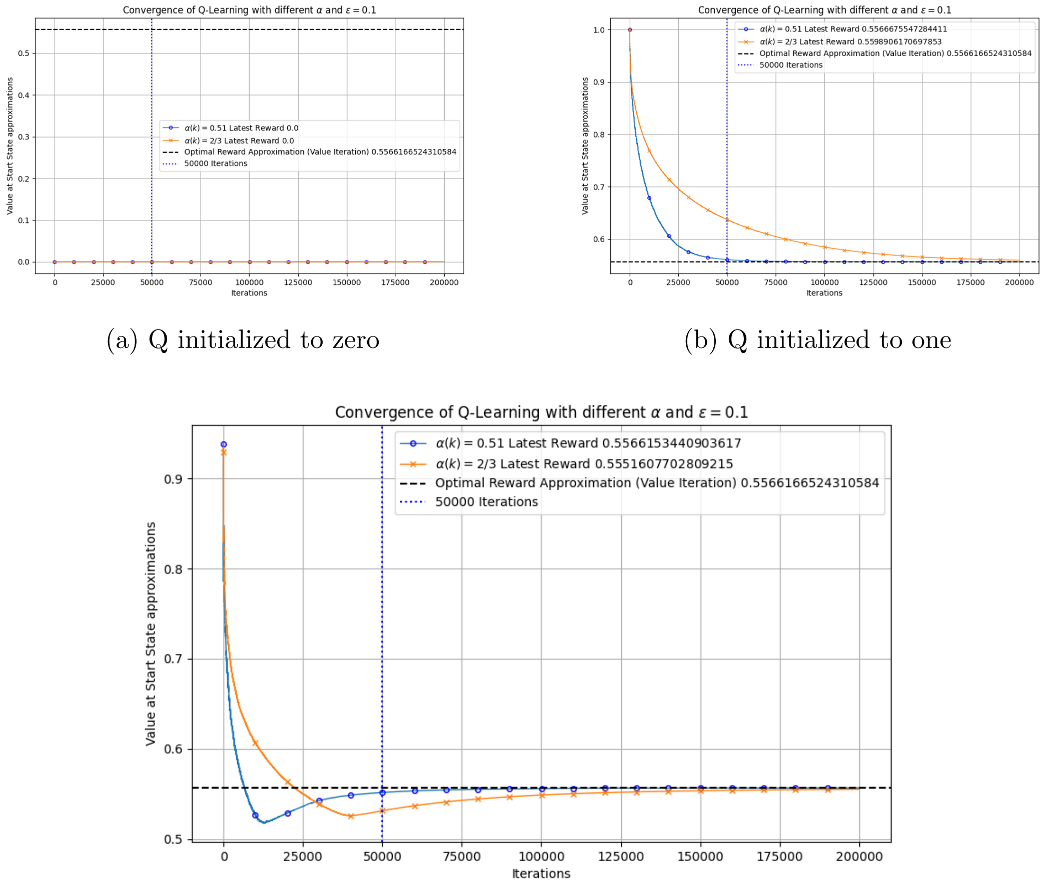
(c) Q initialized uniformly

Figure 5: Convergence of the value function for different Q-table initializations with diffrent ϵ, where $\alpha = 2/3$.

It is evident from Figure 5 that initializing the Q-table uniformly between 0 and 1 leads to the fastest convergence of the value function at the start state. Random initialization within a reasonable bound encourages exploration by introducing stochastic optimism, as some values start high by chance without being uniformly overly optimistic. This allows the model to converge faster than when initialized to all ones, where the agent must systematically lower the Q-values for almost every state-action pair, requiring significantly more updates. Conversely, initializing to zero results in a pessimistic model that relies entirely on the ϵ-greedy mechanism for exploration, failing to naturally incentivize visiting unknown states and in this case not even converging within the definied episodes.

We can also observe that a higher ϵ (more exploration) generally leads slower convergence across all initializations. This is because excessive exploration prevents the agent from sufficiently exploiting learned knowledge to refine the Q-values.

1.5.1.3 3)



(a) Q initialized to zero

(b) Q initialized to one

(c) Q initialized uniformly

Figure 6: Convergence of the value function for different Q-table initializations with different α where $\epsilon = 0.1$.

Figure 6 confirms that uniform random initialization yields the fastest convergence. We observe that a learning rate parameter of $\alpha = 0.51$ outperforms $\alpha = 2/3$, as the latter decays too quickly to sustain learning. Optimistic initialization (to one) performs better here than in previous experiments because the high initial learning rate rapidly corrects the over-estimated values. Although the random initialization suffers a slight initial overcorrection, it ultimately achieves higher precision (5 correct decimals vs. 4) after 200,000 episodes.

Note that the episode count was increased to 200,000 to highlight these asymptotic differences.

1.5.2 (j) BOUNS

1.5.2.1 1)

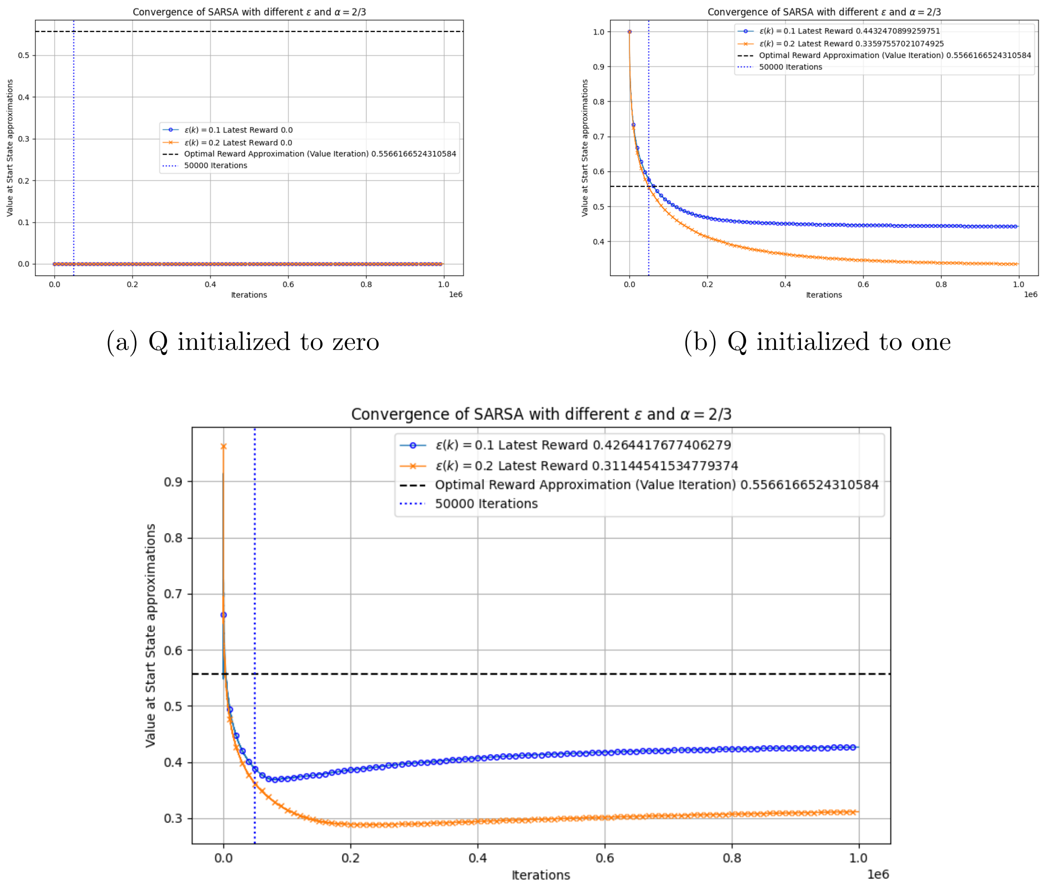
The only thing that changes in the SARSA algorithm compared to Q-learning is the update rule. Instead of using the maximum Q-value of the next state, we use the Q-value of the next action a', selected according to the current policy. To be more precise, instead of the following from Algorithm 1:

```
1: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
```

We now have:

```
1: Select next action $a'$:
2: if random() $< \varepsilon$ then
3:     $a' \leftarrow$ random action from $\mathcal{A}$
4: else
5:     $a' \leftarrow \arg\max_{a'} Q(s',a')$
6: end if
7: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma Q(s',a') - Q(s,a))$
```

1.5.2.2 2)



(a) Q initialized to zero

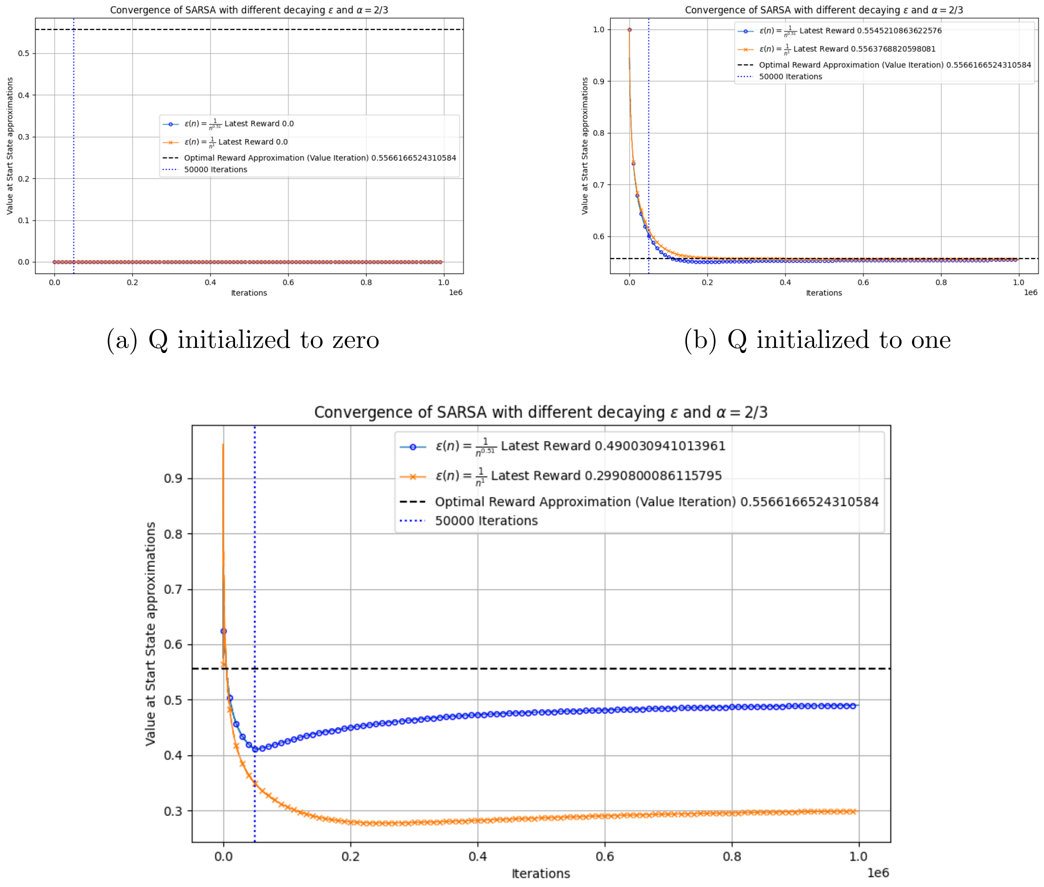
(b) Q initialized to one

(c) Q initialized uniformly

Figure 7: Convergence of the value function for different Q-table initializations with diffrent ϵ, where $\alpha = 2/3$.

A fixed $\epsilon = 0.2$ hinders convergence (Figure 7) because SARSA updates Q-values using the actual next action, including suboptimal exploratory moves. This prevents the value estimates from stabilizing on the theoretical optimum, as the agent effectively learns the value of a policy that acts randomly ϵ of the time. Random initialization remains superior to other methods, though no configuration reaches the optimal value within 1,000,000 episodes due to the fixed exploration rate.

1.5.2.3 3)



(a) Q initialized to zero

(b) Q initialized to one

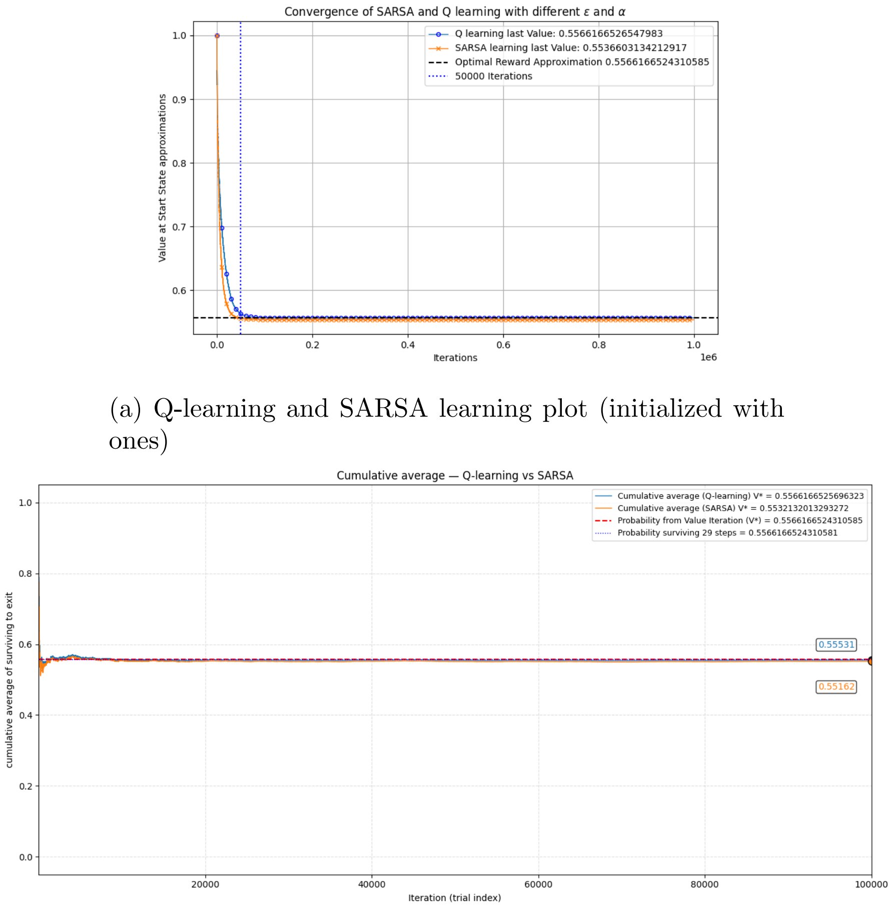
(c) Q initialized uniformly

Figure 8: Convergence of the value function for different Q-table initializations with different α where $\epsilon = 0.1$.

With a decaying ϵ, the model finally converges to the optimal value (Figure 8b). In this setting, optimistic initialization (ones) outperforms random initialization because the high initial learning rate rapidly corrects the over-estimations. regarding parameter tuning, maintaining $\alpha > \delta$ is not necessarily beneficial when convergence is smooth, as a faster decaying ϵ encourages the model to exploit the optimal policy sooner. However, for random initialization (Figure 8c), a higher learning rate proves superior, as it allows for significant updates later in training, helping the agent escape local optima caused by insufficient early exploration.

We iterated for 1,000,000 episodes to ensure convergence across all configurations. To remain in line with the problem formulation, we have also marked the 50,000-episode threshold for reference.

1.5.3 (k) Bonus



(a) Q-learning and SARSA learning plot (initialized with ones)

(b) 100,000 trial simulation of Q-learning and SARSA from Figure 9a

Figure 9: Performance of Q-learning ($\alpha = 0.501$, $\epsilon = 0.2$) and SARSA ($\alpha = 0.501$, $\delta = 1$) initialized with ones.

We can clearly see from Figure 9 that the probabilities of survival for Q-learning and SARSA converge in this environment to the Q-value of the initial state. This is due to the reward structure defined in 1.1.1.4.

We also observe that both Q-learning and SARSA attempt to converge to the theoretical optimal survival probability, with Q-learning slightly outperforming SARSA. This is expected because Q-learning is an off-policy method that learns the optimal policy directly. In contrast, SARSA is on-policy and learns the value of the policy being followed, which may include suboptimal exploratory actions, from the non zero ϵ.

2 Problem 2

2.0.1 (a)

(see code).

2.0.2 (b)

(see code).

2.0.3 (c)

The agent was trained for 200 episodes using the SARSA(λ) algorithm. Due to the continuous nature of the Mountain Car state space (position and velocity), tabular methods are infeasible. We therefore employed Linear Function Approximation, approximating the value function as $Q(s, a) \approx \mathbf{w}^\top \phi(s)$.

2.0.3.1 Feature Engineering
We utilized an order-2 Fourier Basis ($p = 2$) to construct the feature vector $\phi(s)$. This method maps the 2-dimensional state vector into a higher-dimensional feature space using cosine functions. With order $p = 2$, we generate coefficients for all combinations of frequencies $f \in \{0, 1, 2\}$ for both state dimensions, resulting in $(p + 1)^2 = 9$ distinct features per action. Specifically, a feature $\phi_i(s)$ corresponds to a coefficient vector $\mathbf{c}_i \in \{0, 1, 2\}^2$ and is calculated as $\phi_i(s) = \cos(\pi \cdot \mathbf{c}_i^\top \mathbf{s})$.

2.0.3.2 Optimization and Stability
Standard Stochastic Gradient Descent (SGD) was replaced with Nesterov Momentum. This modification accelerates convergence by incorporating a ”velocity” term, allowing the agent to navigate the optimization landscape with reduced oscillation. To further ensure stability, we implemented:

1. **Gradient Clipping:** Eligibility traces were clipped to the range $[-5, 5]$ to prevent gradient explosion.
2. **Adaptive Learning Rate:** The learning rate for each feature was scaled inversely by the norm of its Fourier basis coefficients ($1/||\eta||_2$). This ensures smaller update steps for high-frequency features (which are sensitive to noise) and larger steps for low-frequency features.

2.0.3.3 Hyperparameters
The specific hyperparameters used during training were:

- **Discount factor (γ):** 1 (Undiscounted task).
- **Trace decay (λ):** ≈ 0.854.

- **Momentum:** ≈ 0.998.
- **Learning Rate (α):** Follows a polynomial decay schedule:

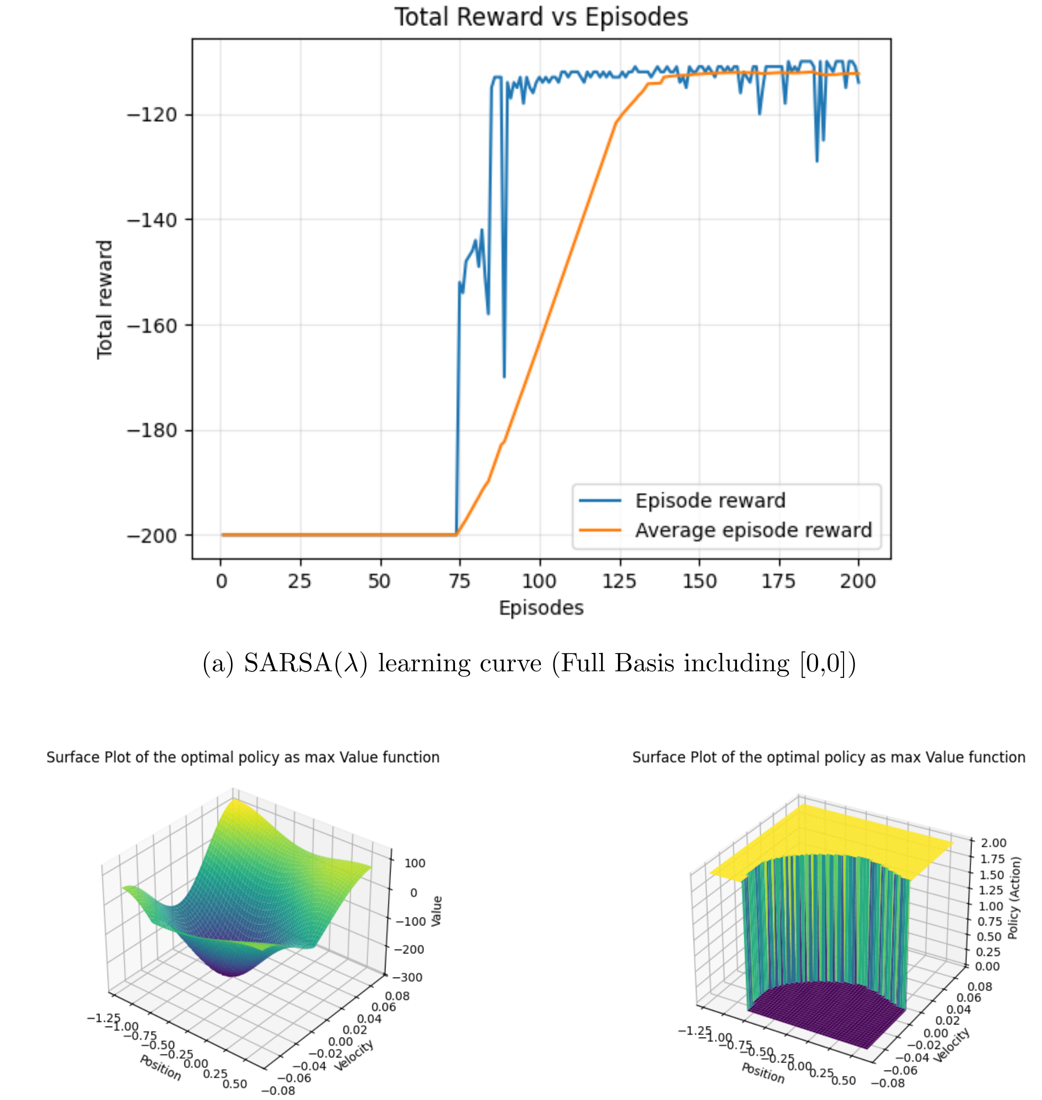
  $$\alpha_k = \frac{\alpha_{init}}{(k/\text{scale})^{\text{power}}}, \quad k \geq 1$$

  where $\alpha_{init} \approx 6.64 \times 10^{-5}$, scale ≈ 21.5, and power ≈ 0.674. OBS k is the episodes count.
- **Exploration (ϵ):** Follows an exponential decay schedule:

  $$\epsilon_k = \epsilon_{init} \cdot (\epsilon_{decay})^k, \quad k \geq 1$$

  where $\epsilon_{init} \approx 0.002$ and $\epsilon_{decay} \approx 0.934$. OBS k is the episodes count.

2.0.4 (d)



(a) SARSA(λ) learning curve (Full Basis including [0,0])

(b) Approximated Value Function $V(s)$

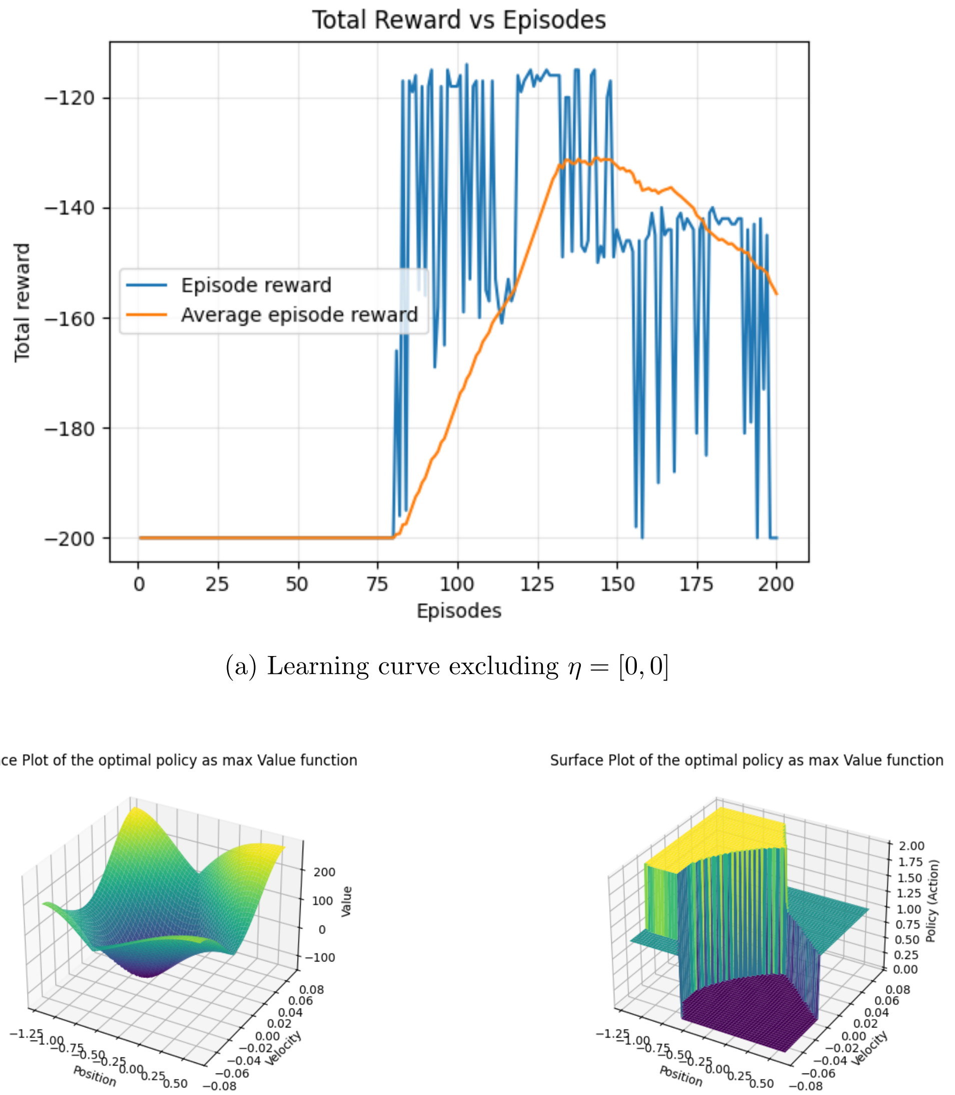
(c) Learned Policy $\pi(s)$

Figure 10: Analysis of the agent trained with the full Fourier Basis ($p = 2$).

Figure 10a illustrates the learning curve over 200 episodes. The agent demonstrates a consistent increase in total reward (decrease in steps), indicating effective convergence. The 3D value function plot (Figure 10b) aligns with the expected physics of the environment. The value peaks near the goal state (position ≈ 0.5, velocity ≈ 0), reflecting the high expected return of reaching the flag. Conversely, the value is lowest in the valley where the agent has no momentum.

The learned policy (Figure 10c) exhibits the necessary ”swinging” strategy. To reach the goal, the agent does not greedily push right; instead, it pushes left when velocity is negative and right when velocity is positive. This builds the mechanical energy required to overcome gravity and reach the top of the hill.

We investigated the importance of the basis vector $\eta = [0, 0]$. As shown in Figure 11, excluding this term severely hinders performance.



(a) Learning curve excluding $\eta = [0, 0]$

(b) Value Function $V(s)$ (No Bias)

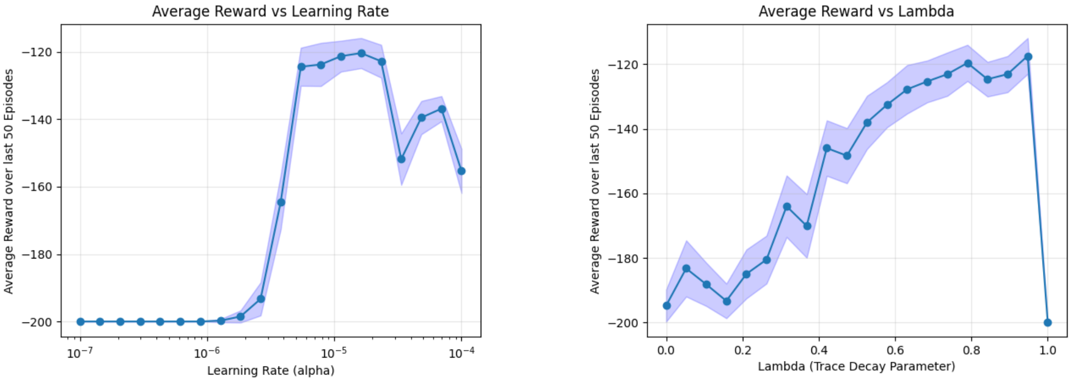
(c) Policy $\pi(s)$ (No Bias)

Figure 11: Performance analysis when the constant basis feature $\eta = [0, 0]$ is excluded.

Without $\eta = [0, 0]$, the agent fails to converge effectively (Figure 11a). The mathematical reason is that the feature corresponding to $\eta = [0, 0]$ is $\cos(\pi \cdot [0, 0]^\top s) = \cos(0) = 1$. This acts as a bias term (or intercept) for the linear approximator, allowing the value function to shift its global mean up or down. Without it, the function $Q(s, a)$ is forced to oscillate around zero (or have a mean determined solely by the higher frequencies), which makes approximating the true value function extremely difficult. Consequently, the value landscape (Figure 11b) is distorted, leading to an incoherent policy (Figure 11c).

2.0.5 (e)

2.0.5.1 1)



(a) Impact of learning rate α (with 95% CI)

(b) Impact of eligibility trace λ (with 95% CI)

Figure 12: Sensitivity analysis of the average total reward (last 50 episodes) with respect to learning rate and eligibility trace decay.

We can see that the learning rate α has a significant impact on performance (Figure 12a), with values around 10^{-5} yielding the best results. Lower values result in learning that is too slow to solve the task within the episode limit, while excessively high values likely cause instability or divergence in the weight updates.

Similarly, the eligibility trace parameter λ performs better with higher values (Figure 12b). This is because the Mountain Car environment has sparse rewards (the agent receives a penalty of -1 at every step until the goal). A higher λ increases the decay horizon, allowing the algorithm to propagate the delayed reward signal further back in time to the early actions that helped build the necessary momentum. Conversely, lower λ values rely more on bootstrapping (TD), which propagates reward information much slower across the long trajectories required to reach the top.

2.0.5.2 2)

We observed that zero initialization of the weights $\mathbf{w}$ yields the best performance. This success is attributed to the principle of *Optimistic Initial Values.*

In the Mountain Car environment, the reward structure is strictly negative ($r = -1$ per step). Consequently, the true value function $V^{\pi}(s)$ is always negative. By initializing weights to zero, the initial value estimates start at $Q(s, a) \approx 0$, which is strictly greater than the true values. This ”optimism” drives exploration: whenever the agent visits a state and receives a reward of -1, the value estimate decreases. However, unvisited or less-visited state-action pairs remain close to 0 (higher value), effectively pulling the agent towards exploring new trajectories to find the non-existent ”zero reward” path, eventually leading it to the goal.

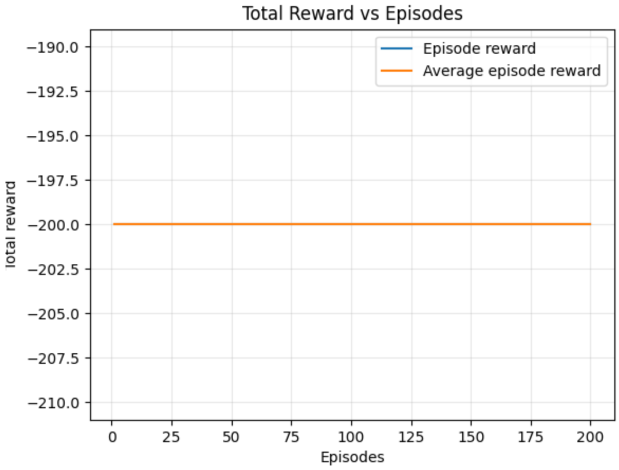


Figure 13: Learning curve using Random Initialization ($\mathbf{w} \sim U[-140, 0]$). All other hyperparameters remain identical to Figure 10a.

In contrast, random initialization (Figure 13) hinders convergence. As shown in the plot, initializing with arbitrary negative values (e.g., mimicking the range of expected returns) introduces high variance. This can result in "pessimistic" estimates where the agent falsely believes certain useful state-actions are worse than they are, or "arbitrary" biases where the agent favors a direction purely due to initialization noise rather than experience. In the context of Function Approximation with momentum, these erratic initial gradients make it difficult for the optimizer to settle into a stable descent direction, significantly delaying learning.

2.0.5.3 3)

We implemented an Upper Confidence Bound (UCB) action selection strategy. This method balances exploration and exploitation by augmenting the action-value estimates (Q-values) with an uncertainty bonus proportional to the rarity of the state-action pair [?].

Since the Mountain Car state space is continuous, exact visitation counts are impossible. We addressed this by discretizing the state space into a grid of 20×20 bins for position and velocity, respectively. As our implementation normalizes the state variables to the range $[0, 1]$, the bin indices (i, j) for a given normalized state $\bar{s} = (\bar{p}, \bar{v})$ are computed as:

$$i = \lfloor 20 \cdot \bar{p} \rfloor, \quad j = \lfloor 20 \cdot \bar{v} \rfloor$$

We maintain a visitation count table $N(i, j, a)$ which tracks how many times action a has been taken in bin (i, j). The policy then selects the action a that maximizes the combined value:

$$a = \underset{\alpha \in \mathcal{A}}{\operatorname{argmax}} \left(Q(s, \alpha) + c \sqrt{\frac{\ln \left(\sum_{x,y,k} N(x, y, k) \right)}{N(i, j, \alpha) + 10^{-5}}} \right)$$

Here, the term inside the logarithm represents the total number of steps taken across all episodes. The square root term acts as the exploration bonus. It is large for actions rarely visited in the current region (low local N) relative to the total training time, effectively guiding the agent to explore unknown regions.

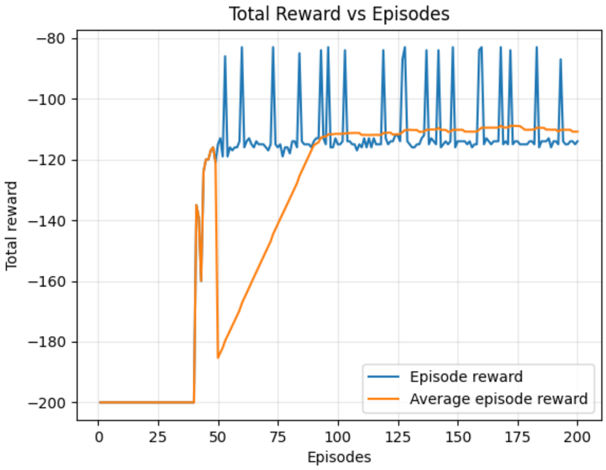


Figure 14: Learning curve using UCB action selection with zero weight initialization.

We observed that UCB action selection (Figure 14) accelerates learning compared to ϵ-greedy exploration. UCB converges within approximately 50 episodes, whereas ϵ-greedy required around 80 episodes to reach similar performance (Figure 10a). Furthermore, UCB appears to converge to a more stable optimal policy, achieving a consistently higher average reward in the final episodes compared to the downward spikes observed in the ϵ-greedy approach.

To ensure a robust comparison, we utilized Optuna to tune the hyperparameters for both models using identical search spaces for shared variables (differing only in exploration parameters c vs. ϵ). Notably, we found that the optimal learning rate for UCB ($\approx 3\times10^{-3}$) was significantly higher than that required for ϵ-greedy ($\approx 5 \times 10^{-5}$). This suggests that UCB's directed exploration leads to more stable updates, allowing the agent to learn more aggressively without diverging.

2.0.6 (f)

For the final submission, we selected the policy trained using the UCB exploration strategy described in Section 2.0.5.3. This model was chosen because it demonstrated superior stability and convergence speed compared to the standard ϵ-greedy approach, while successfully satisfying the performance criterion (average reward ≥ -135).