Computer Lab 1

Benet Ramió i Comas 20031026T177
Laura Solà Garcia 20031119T225
Reinforcement Learning

December 4, 2024

1 The Maze and the Random Minotaur

1.1 Basic maze

1.1.1 MDP definition

State Space: The state space is defined as:

$$S = \{((x_p, y_p), (x_m, y_x))|\textit{valid positions}\} \cup \{Dead\} \cup \{Win\}$$

where (x_p, y_p) and (x_m, y_m) denote the positions of the player and the Minotaur, respectively.

Firstly, the set of states $\{((x_p, y_p), (x_m, y_x))|\textit{valid positions}\}$ can be defined as follows:

$$S_{\textit{valid pos.}} = \{((x_p, y_p), (x_m, y_m)) \mid 1 \leq x_p \leq M_x, 1 \leq y_p \leq M_y, 1 \leq x_m \leq M_x, 1 \leq y_m \leq M_y, (x_p, y_p) \notin W\}$$

- W is the set of wall cells, which the player cannot occupy. In addition, he must stay within the limits of the maze (shape M_x x M_y)
- The Minotaur can occupy any cell within the maze boundaries, including walls.

On the other hand, there are the two terminal states in the state space:

- Win: The player reaches the exit position (x_{exit}, y_{exit}) and the Minotaur is not in the exit cell:
$$x_p = x_{exit},\ y_p = y_{exit} \text{ and } (x_m \neq x_{exit} \text{ or } y_m \neq y_{exit}).$$
- Dead: The player and the Minotaur occupy the same cell:
$$x_p = x_m \text{ and } y_p = y_m.$$

Action Space: The actions that can be taken by the person are:

$$A = \{\text{stay}, \text{up}, \text{down}, \text{left}, \text{right}\}.$$

However, not all of them are available at any given state and time, there are some limitations:

- The player cannot move outside the maze boundaries. For example, if $x_p = 1$, the action "up" is invalid.
- The player cannot move into a wall cell $(x_w, y_w) \in W$.

Transition Probabilities: These are determined by the player's deterministic movement and the Minotaur's stochastic random walk.
Let $f(x_p, y_p, a)$ denote the deterministic next position of the player given the action a:

$$f(x_p, y_p, a) = \begin{cases} (x_p, y_p), & \text{if } a = \text{stay}, \\ (x_p - 1, y_p), & \text{if } a = \text{up}, \\ (x_p + 1, y_p), & \text{if } a = \text{down}, \\ (x_p, y_p - 1), & \text{if } a = \text{left}, \\ (x_p, y_p + 1), & \text{if } a = \text{right}. \end{cases}$$

The next player position is determined by:

$$(x'_p, y'_p) = f(x_p, y_p, a).$$

The Minotaur moves randomly to any valid adjacent cell (including walls). Let $M(x_m, y_m)$ denote the set of valid cells the Minotaur can move to. The size of this set is $|M(x_m, y_m)|$, which depends on the Minotaur's current position.
Therefore, for a state (x_p, y_p, x_m, y_m) and an action a, the set of possible next states S' is:

$$S' = \{(f(x_p, y_p, a), (x'_m, y'_m)) \mid (x'_m, y'_m) \in M(x_m, y_m)\}.$$

and the transition probability can be defined as:

$$P(s' \mid s, a) = \begin{cases} \frac{1}{|M(x_m, y_m)|}, & \text{if } s' \in S', \\ 0, & \text{otherwise.} \end{cases}$$

Note that $s' =$ *Dead* in case that $(x'_p, y'_p) = (x'_m, y'_m)$.

In addition, to account for the special conditions involving the *Win* state:

- If $s' = (x_{exit}, y_{exit}, x'_m, y'_m)$ and $(x_{exit}, y_{exit}) \notin M(x_m, y_m)$, then the transition leads to the *Win* state with probability 1.
- If $s' = (x_{exit}, y_{exit}, x'_m, y'_m)$ and $(x_{exit}, y_{exit}) \in M(x_m, y_m)$, then:
  - The transition leads to the *Dead* state with probability $\frac{1}{|M(x_m, y_m)|}$.
  - Otherwise, the transition leads to the *Win* state.

On the other hand, we must define the transition probabilities from terminal states:

$$P(s' \mid \text{Win}, a) = \begin{cases} 1, & \text{if } s' = \text{Win}, \\ 0, & \text{otherwise.} \end{cases} \qquad P(s' \mid \text{Dead}, a) = \begin{cases} 1, & \text{if } s' = \text{Dead}, \\ 0, & \text{otherwise.} \end{cases}$$

Rewards: The rewards are defined as follows:

$$r(s,a) = \begin{cases} 1, & \text{if } s = \text{Win}, \\ -1, & \text{if } s = \text{Dead}, \\ 0, & \text{otherwise.} \end{cases}$$

This formulation penalizes being caught by the Minotaur, encouraging the optimal policy to avoid capture. Simultaneously, it rewards reaching the maze exit as quickly as possible, as the reward accumulates for each time step the player remains in the *win* state until the time horizon T is reached. Thus, the faster the player reaches the exit, the greater the total reward.

1.1.2 Movement in Alternating Rounds

When the player and the Minotaur move in alternating rounds rather than simultaneously, the Markov Decision Process (MDP) is modified. While the state space, action space, and reward structure remain unchanged, the transition probabilities are altered due to the sequential nature of the moves.

In the alternating rounds setup, the transition probabilities depend on the timestep. Assuming the player moves first at $t = 0$, we can describe the transitions as follows:

- At even timesteps ($t = 0, 2, 4, \ldots$), the player moves while the Minotaur remains stationary. In this case, the next state is determined solely by the player's action.
- At odd timesteps ($t = 1, 3, 5, \ldots$), the Minotaur moves while the player remains stationary. The Minotaur transitions to one of its available adjacent cells with equal probability.

Formally and using the same notation as in the previous section, given a state $s = (((x_p, y_p), x_m, y_m))$ and an action $a \in A$, the transition probabilities are defined as:

$$P_t(s' \mid s, a) = \begin{cases} 1, & \text{if } t \text{ is even and } s' = (f(x_p, y_p, a), (x_m, y_m)), \\ \frac{1}{|M(x_m, y_m)|}, & \text{if } t \text{ is odd and } s' \in \{((x_p, y_p), (x'_m, y'_m)) \mid (x'_m, y'_m) \in M(x_m, y_m)\}, \\ 0, & \text{otherwise,} \end{cases}$$

where $f(x_p, y_p, a)$ denotes the player's position after taking action a, and $M(x_m, y_m)$ represents the set of valid moves for the Minotaur from its position (x_m, y_m).

The change in movement order introduces both advantages and disadvantages for the player, making the likelihood of being caught by the Minotaur dependent on the situation.

One could argue that the Minotaur is less likely to catch the player under the alternating rounds setup. In this configuration, the player gains an informational advantage by observing the Minotaur's movement before deciding on their next action. This reduces uncertainty,

especially when the player is in close proximity to the Minotaur. Knowing the Minotaur's exact position allows the player to react strategically and maintain at least one cell of distance, minimizing the immediate risk of being caught. This advantage is particularly useful in confined regions of the maze where the player has fewer movement options.

Conversely, there are scenarios where the Minotaur becomes more likely to catch the player in the alternating rounds setup. In the simultaneous movement case, if the player and the Minotaur are in adjacent cells, the player could theoretically avoid being caught indefinitely by moving into the Minotaur's position, exploiting the rule that the Minotaur cannot remain stationary. This dynamic provided a safe escape route in situations where the player's available actions were otherwise constrained. In the alternating rounds setup, this escape mechanism is no longer possible, as the Minotaur's moves occur in alternating rounds of the player's action.

The alternating rounds setup alters the dynamics of the game in a way that changes the strategic considerations for both the player and the Minotaur. While the reduced uncertainty provides the player with a tactical advantage, the inability to exploit simultaneous movement mechanics may increase the likelihood of being caught in certain scenarios. Overall, whether the Minotaur is more or less likely to catch the player depends on the specific configuration of the maze and the relative positions of the two agents.

1.2 Dynamic Programming

1.2.1 Solve the problem for $T = 20$

With the MDP defined as in the first section, we have used dynamic programming to find the optimal policy of the person in order to maximize the probability of exiting the maze alive with a time horizon of 20. The dynamic programming algorithm returns the optimal policy as a matrix of sizes SxT, the optimal time-varying policy at every state.

If we simulate a game we observe how the optimal policy takes the shortest path to the exit without staying still at any given time. This can be seen in Figure 1.

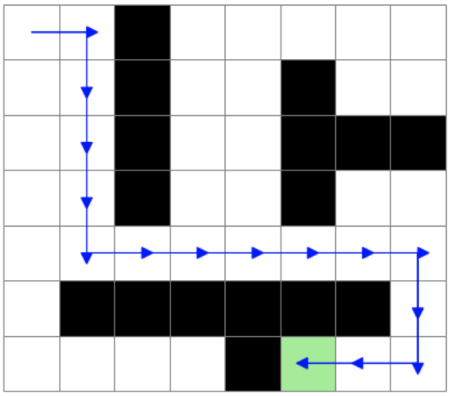


Figure 1: Movement of the person in a simulated game following the optimal policy

1.2.2 Estimate probability of exiting alive for $T \in \{1...30\}$

To estimate the probability of the player successfully exiting the maze alive, we simulated 10,000 games for time horizons $T = 1, \ldots, 30$. Figure 2 illustrates these probabilities under two scenarios: (1) the Minotaur is not allowed to stay still, and (2) the Minotaur is allowed to stay still.

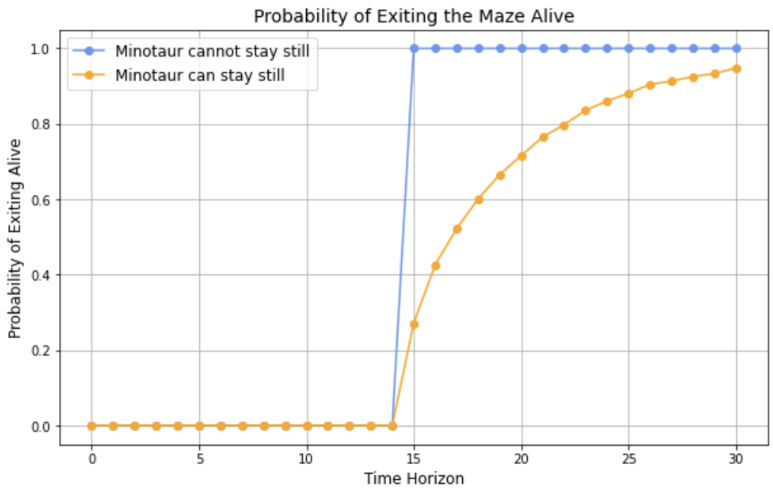


Figure 2: Probability of escaping alive depending on whether the minotaur can stay still or not, for each time horizon $T \in \{1...30\}$

Firstly, when the Minotaur is required to move every step, the probability of escaping alive is:

$$P(\text{escape alive}) = \begin{cases} 0, & \text{if } T < 15, \\ 1, & \text{if } T \geq 15. \end{cases}$$

This behavior arises because the shortest path to the exit requires 15 steps. For $T < 15$, escape is impossible as there is insufficient time to reach the exit. For $T \geq 15$, the optimal policy ensures the player takes the shortest path without stopping, guaranteeing escape. The player's starting position (even) and the Minotaur's starting position (odd) ensure that their positions alternate in parity, making it impossible for the Minotaur to catch the player.

In contrast, if the Minotaur is allowed to stand still, the probability of the player exiting the maze alive decreases. While the even-odd parity advantage previously prevented encounters, allowing the Minotaur to stay still removes this safety mechanism. On a more general level, this decrease in survival probability occurs because the Minotaur can control key positions in the maze by pausing, reducing the player's safe paths. Even with random motion, the ability to pause allows the Minotaur to align with the player's path and catch them.

As the time horizon T increases, the probability of exiting the maze alive improves because the player has more opportunities to wait and adjust their movements in response to the Minotaur's random actions. Longer time horizons provide greater flexibility for the player to avoid the Minotaur and reach the exit safely, even if the Minotaur's random walks occasionally bring it dangerously close.

1.3 Value iteration: poisoning

1.3.1 MDP modelling

The poison imposes a random time horizon for survival, modeled as a geometric distribution with a mean of 30 steps. This transforms the problem into a Markov Decision Process (MDP) with a random time horizon. This type of MDPs can be modelled as an MDP with discounted expected reward, taking a discount factor such that

$$E[T] = \frac{1}{1-\lambda}.$$

The geometric distribution with mean $E[T] = 30$ can be expressed as $T \sim \text{Geo}(1-\lambda)$, where λ is the discount factor of the equivalent infinite-horizon MDP. Substituting $E[T] = 30$, we solve for λ:

$$30 = \frac{1}{1-\lambda} \implies 1-\lambda = \frac{1}{30} \implies \lambda = \frac{29}{30}.$$

Thus, the original problem with a geometrically distributed time horizon can be reformulated as an infinite-horizon discounted MDP with a discount factor $\lambda = \frac{29}{30}$.

The state space, action space, and reward structure of the problem remain unchanged.

By modeling the poisoned scenario as an infinite-horizon discounted MDP we can use algorithms like Value Iteration to compute the optimal policy. This policy maximizes the probability of exiting the maze before the random time horizon determined by the poison elapses.

1.3.2 Estimate probability of getting out alive

After implementing the value iteration algorithm and simulating 10,000 games, we have estimated that the person has approximately 0.625% probability of getting out alive by following the optimal policy (taking the shortest path without stops).

If we analyze the problem formulation and the previously obtained results, we can see how the estimated probability makes sense. Firstly, the life-span of the person is distributed following a geometric distribution Geo(p) where $p = 1 - \frac{29}{30}$. The probability of the person's survival depends on whether they have enough time to reach the exit. Specifically, we compute the probability of $T \geq 15$, where 15 represents the time steps required to reach the exit along the shortest path:

$$P(T \geq 15) = (1-p)^{14},$$

Substituting p into the formula:

$$P(T \geq 15) = \left(\frac{29}{30}\right)^{14} \approx 0.624\%.$$

This result aligns with our simulation, where the estimated probability of survival is approximately 0.624%. This tells us that the person survives when they have enough time to reach

the exit, and if they do have enough time, they have a survival rate of 100%. This conclusion matches our earlier findings in Section 1.2, where the optimal policy guarantees survival in such scenarios.

1.4 Additional questions

What does it mean that a learning method is on-policy or off-policy?

In reinforcement learning, on-policy and off-policy methods differ in how data is collected and used to improve the agent's policy. While an off-policy learner learns the value of the optimal policy independently of the agent's actions, an on-policy learner learns the value of the policy being carried out by the agent.

On-policy learning updates the policy that the agent is actively following, meaning that the agent collects data under its current policy and uses this data to refine and improve the same policy. This approach leads to a direct cycle where the behavior policy (the one used for data collection) is the same as the target policy (the one being optimized).

In contrast, off-policy learning allows the agent to learn about a target policy independently of the actions it actually takes to gather data. This means data can be collected using a different behavior policy. This separation between behavior and target policies makes off-policy methods more flexible, as they can leverage data from other sources or past experiences.

State the convergence conditions for Q-learning and SARSA:

The theorem for Q-learning convergence states that $\lim_{n\to\infty} Q^{(t)} = Q$ almost surely if the following conditions are met:

- The step sizes α_t satisfy $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 \leq \infty$
- The behaviour policy π_b visits every (state, action) pairs infinitely often.
- The discount factor $\lambda \in (0, 1)$

This conditions are met if $\alpha_t = \frac{1}{t+1}$ and if the behaviour policy yields an irreducible Markov chain. They are also met for the ϵ-greedy behavior policy.

The theorem for SARSA convergence states that $\lim_{n\to\infty} Q^{(t)} = Q^{\pi}$ almost surely if the following conditions are met:

- The step sizes α_t satisfy $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 \leq \infty$
- The policy π_t is ϵ-greedy with relation to $Q^{(t)}$
- The discount factor $\lambda \in (0, 1)$

In addition, Q^{π} tends to Q as ϵ tends to 0^+. An example of such case is when $\epsilon = \frac{1}{t}$, in which case $Q^{(t)}$ tends to Q as $t \to \infty$.

1.4.1 Maze with keys:

To adapt the MDP to this new scenario, we incorporate the requirement that the agent must first collect the keys at position C before proceeding to the exit at position B. This introduces a new dimension to the state space to account for whether the agent has collected the keys. The modified components of the MDP are as follows:
The state space now includes the information about whether the agent has collected the keys. Each state is represented as:

$$(x_p, y_p, x_m, y_m, \text{keys}),$$

where (x_p, y_p) is the player's position, (x_m, y_m) is the Minotaur's position, and keys is a Boolean variable indicating whether the player has collected the keys. We mantain the terminal states Win and Dead, representing successful exit and being caught, respectively.

The reward function remains unchanged[1]:

$$r(s,a) = \begin{cases} 1, & \text{if } s = \text{Win}, \\ -1, & \text{if } s = \text{Dead}, \\ 0, & \text{otherwise.} \end{cases}$$

The action space also remains the same:

$$A = \{\text{stay}, \text{up}, \text{down}, \text{left}, \text{right}\}.$$

The transition probabilities are now adjusted to account for the following factors: 1) need to collect the keys 2) improved movement of the minotaur (with 35% probability, the minotaur moves toward the player and with 65% probability, the minotaur moves randomly to any neighboring cell).

For the transitions before or after having collected the keys, for a state $(x_p, y_p, x_m, y_m, \text{keys})$, the set of possible next states S' is:

$$S' = \{(f(x_p, y_p, a), (x'_m, y'_m), \text{keys}') \mid (x'_m, y'_m) \in M(x_m, y_m), \text{ and keys}' = \text{keys}\}.$$

Here, the "keys" boolean remains unchanged during these transitions.

We introduce (x^*_m, y^*_m) as the position that minimizes the Euclidean distance between the minotaur and the player:

$$(x^*_m, y^*_m) = \underset{(x'_m, y'_m) \in M(x_m, y_m)}{\arg\min} ||(x'_p, y'_p) - (x'_m, y'_m)||.$$

[1]For our implementation and experiments in the following sections we changed the rewards adding a small reward when getting the keys to help the algorithm converge: 10 for *Win*, 1 when getting the keys and -100 for *Dead*.

The transition probabilities are then defined as follows:

$$P(s' \mid s, a) = \begin{cases} 0.35 + 0.65 \cdot \frac{1}{|M(x_m, y_m)|}, & \text{if } s' \in S' \text{ and } (x'_m, y'_m) = (x^*_m, y^*_m), \\ 0.65 \cdot \frac{1}{|M(x_m, y_m)|}, & \text{if } s' \in S' \text{ and } (x'_m, y'_m) \neq (x^*_m, y^*_m), \\ 0, & \text{otherwise.} \end{cases}$$

Note that $s' =$ *Dead* in case that $(x'_p, y'_p) = (x'_m, y'_m)$.

We also need to account for the moment in which the player collects the keys. We do this by modifying S' when $(x'_m, y'_m) = (x_{keys}, y_{keys})$. In such case, the boolean $keys$ of the next possible states should be set to $True$ instead of $False$ (in case that the keys have not been previously collected).

In addition, to account for the special conditions involving the *Win* state:

- If $s' = (x_{exit}, y_{exit}, x'_m, y'_m, true)$ and $(x_{exit}, y_{exit}) \notin M(x_m, y_m)$, then the transition leads to the *Win* state with probability 1.
- If $s' = (x_{exit}, y_{exit}, x'_m, y'_m, true)$ and $(x_{exit}, y_{exit}) \in M(x_m, y_m)$, then:
  - The transition leads to the *Dead* state with probability $0.35 + 0.65 \frac{1}{|M(x_m, y_m)|}$.
  - Otherwise, the transition leads to the *Win* state.

For the terminal states, the transitions remain unchanged:

$$P(s' \mid \text{Win}, a) = \begin{cases} 1, & \text{if } s' = \text{Win}, \\ 0, & \text{otherwise.} \end{cases} \qquad P(s' \mid \text{Dead}, a) = \begin{cases} 1, & \text{if } s' = \text{Dead}, \\ 0, & \text{otherwise.} \end{cases}$$

This modification essentially creates two "parallel mazes": one where the player has not yet collected the keys (keys = false) and one where the player has collected the keys (keys = true). The player transitions between these mazes upon collecting the keys, and the overall objective remains to maximize the probability of reaching *Win* while avoiding *Dead*.

1.5 Q-Learning and SARSA

1.5.1 Q-learning

Algorithm 1 provides the pseudocode for both Q-learning and SARSA algorithms. In the case of Q-learning, the key idea is to use the maximum Q-value of the next state-action pair, which leads to an off-policy learning process. This allows the agent to learn an optimal policy independent of the behavior policy it follows during exploration. For our implementation, we note that the reward for reaching terminal states (such as *Win* or *Dead*) is preassigned to the corresponding states, and the Q-values for these terminal states are not updated during the training process. We also ensure that the environment is modeled with an infinite-horizon Markov Decision Process (MDP) to prevent early termination (e.g., by poison), which guarantees that the agent will reach a terminal state in each episode. This choice accelerates convergence since the agent consistently reaches a terminal state, but the RL policy still optimizes the poisoned version of the game due to the discount factor λ.

Algorithm 1 Q-learning and SARSA pseudocode implementation

Require: maze, start, algorithm, ϵ, λ, α, num_episodes, init, delta

env $\leftarrow$ Maze(maze)
agent $\leftarrow$ RL_agent(env, ϵ, λ, α, init, delta)

for $i = 1$ **to** num_episodes **do**
  $s \leftarrow$ start
  while *state is not terminal* **do**
    $a \leftarrow$ Epsilon_greedy_policy(s)
    $r, s' \leftarrow$ Observations(s, a)
    $n(s, a) \leftarrow n(s, a) + 1$ ▷ Number of visits to state-action pair (s, a)
    if *algorithm is 'Q_learning'* **then**
      $Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s,a)^\alpha}(r + \lambda \cdot \max_b Q(s', b) - Q(s, a))$
    else if *algorithm is 'SARSA'* **then**
      $a' \leftarrow$ Epsilon_greedy_policy(s') ▷ Choose next action for SARSA
      $Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s,a)^\alpha}(r + \lambda \cdot Q(s', a') - Q(s, a))$
    end if
    $s \leftarrow s'$ ▷ Move to next state
  end while
end for
policy $\leftarrow$ agent.get_policy() ▷ Extract the final policy
return policy

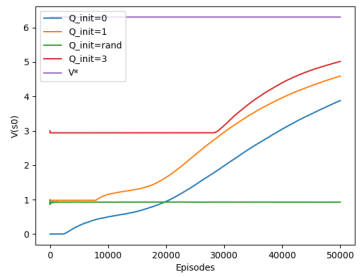
To determine optimal hyperparameters, we conducted several experiments, training the agent for 50,000 episodes in each case. The optimal value $V^*(s_0)$ was precomputed analytically using the defined rewards and optimal policy to serve as a benchmark for comparison. The following subsections summarize the results of our experiments.

In the first experiment, we tested four initialization strategies for the Q-values of all non-terminal states: zeros, ones, random values between 0 and 1, and a constant value of three. The hyperparameters $\alpha = \frac{2}{3}$ and $\epsilon = 0.2$ were fixed for all cases. Figure 3a presents the results. Random initialization failed to produce meaningful learning, while optimistic initializations (ones and threes) demonstrated superior performance. Interestingly, higher initializations delayed initial learning but achieved a better $V(s_0)$ after sufficient training. The results align with the theoretical intuition that optimistic initial values encourage exploration by overestimating Q-values, accelerating convergence as they approach their true values.

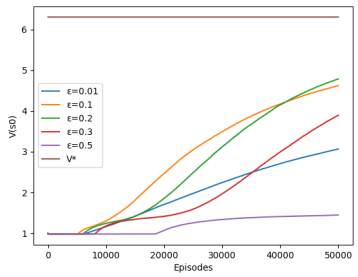
In the second experiment, we explored the impact of different exploration probabilities, ϵ. We tested $\epsilon \in \{0.01, 0.1, 0.2, 0.3, 0.5\}$, using the same $\alpha = \frac{2}{3}$ and the two best-performing initializations (ones and threes) from the previous experiment. Figures 3b and 3c show the results for initializations to ones and threes, respectively. For both initializations, $\epsilon = 0.2$ emerged as the best choice, closely followed by $\epsilon = 0.1$. Lower values of ϵ led to insufficient exploration, violating the convergence theorem's assumption of visiting each (state, action)

pair infinitely often. On the other hand, higher values resulted in excessive exploration, impeding the agent from effectively refining its policy.

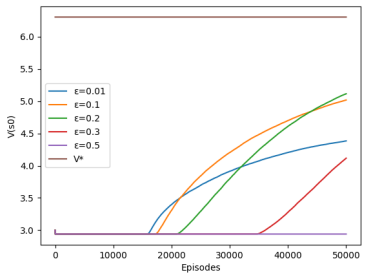
Finally, we evaluated the impact of the step size decay α on performance. Fixing $\epsilon = 0.2$, we tested $\alpha \in \{0.55, 0.65, 0.75, 0.85, 0.95\}$ for both the ones and threes initializations. Figures 3d and 3e present the results. For $\alpha = 0.55$ and $\alpha = 0.65$, both initializations exhibited stable learning, with $\alpha = 0.55$ providing the best results. Smaller decay rates allowed the agent to refine Q-values more effectively, leading to a $V(s_0)$ closer to the optimal value after 50,000 episodes. In contrast, higher values of α led to an excessively rapid decay in the learning rate, resulting in insufficient Q-value updates, particularly for the initialization to threes.



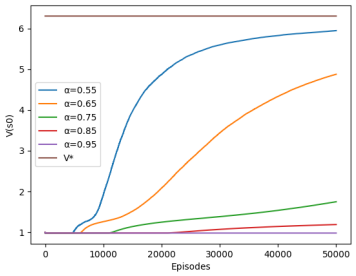
(a) Q-value Initialization



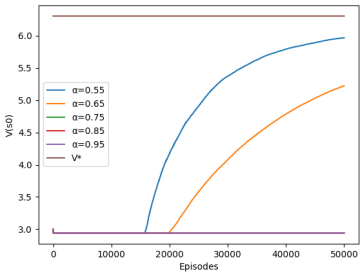
(b) ϵ - Initialization to Ones



(c) ϵ - Initialization to Threes



(d) α - Initialization to Ones



(e) α - Initialization to Threes

Figure 3: Performance analysis of Q-learning for different hyperparameter settings: Q-value initialization, exploration parameter (ϵ) and step size decay (α).

1.5.2 SARSA

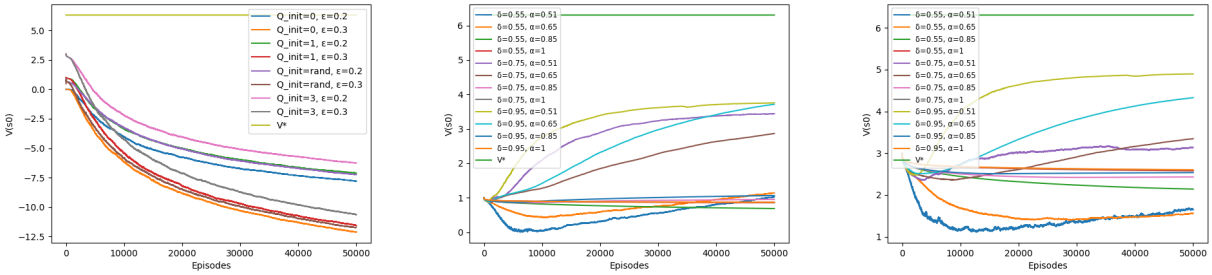
The pseudocode in Algorithm 1 also represents the SARSA algorithm, which differs from Q-learning in the way the Q-values are updated. Unlike Q-learning, which uses the maximum Q-value over the next state, SARSA uses the Q-value corresponding to the action actually taken in the next state (i.e., it uses the action selected by the epsilon-greedy policy). This makes SARSA an on-policy algorithm, as the policy used to select actions during training is the same policy that is evaluated and improved. SARSA tends to learn more conservative policies compared to Q-learning, as it takes into account the current behaviour policy's

actions rather than aiming for the optimal next state-action pair. However, this can make SARSA more challenging to converge, as the agent may explore suboptimal actions more frequently.

To determine the optimal hyperparameters for SARSA, we followed a procedure similar to the one used for Q-learning. The agent was trained for 50,000 episodes in each experiment, and since the reward structure remained unchanged, the optimal value $V^*(s_0)$ served as a consistent benchmark.

In the first experiment, we explored the same Q-value initialization strategies (zeros, ones, random values between 0 and 1, and threes) while testing $\epsilon \in \{0.2, 0.3\}$. Figure 4a presents the results. Regardless of the initialization or ϵ value, the agent failed to learn effectively, as evidenced by the consistently decreasing $V(s_0)$. Higher initializations combined with $\epsilon = 0.2$ performed slightly better, but all results were negative. This outcome highlights SARSA's sensitivity to the exploration strategy. Because SARSA learns the Q-function under the ϵ-greedy policy used for data collection, the policy does not always take the action that maximizes the Q function. This underscores the necessity of reducing ϵ over time to allow the agent to converge. According to theoretical insights, SARSA's Q-function converges to the optimal solution as $\epsilon \to 0^+$.

To address the issue of persistent exploration, we introduced an ϵ decay strategy: $\epsilon_k = \frac{1}{k^\delta}$, where δ controls the rate of decay. We combined $\delta \in \{0.55, 0.75, 0.95\}$ with $\alpha \in \{0.51, 0.65, 0.85, 1.0\}$ for both the ones and threes initializations. The results are shown in Figures 4b and 4c, respectively. The introduction of ϵ decay allowed the agent to converge when using appropriate combinations of α and δ. Specifically, the four configurations with $\alpha \in \{0.51, 0.65\}$ and $\delta \in \{0.75, 0.95\}$ showed successful convergence, with the optimal performance achieved using $\alpha = 0.51$ and $\delta = 0.95$. This suggests that having $\delta > \alpha$ is beneficial. Intuitively, when exploration decays sufficiently, SARSA begins to estimate the Q-function under a near-greedy policy. At this stage, a higher step size (which occurs when α is small) ensures that updates have a meaningful impact on the Q-values, facilitating convergence.



(a) Q-value Initialization and ϵ (b) Exploration Decay - Initialization to Ones (c) Exploration Decay - Initialization to Threes

Figure 4: Performance analysis of SARSA under different initialization and exploration strategies.

1.5.3 Probability of Leaving the Maze Alive

In this final experiment, we evaluated the performance of the Q-learning and SARSA algorithms using the best hyperparameters identified in the previous sections. To estimate the probability of successfully leaving the maze alive, we simulated 10,000 episodes for each algorithm. The hyperparameters and results of these simulations are summarized in Table 1.
The results demonstrate notable differences in the performance of the two algorithms. Q-learning achieved a higher probability of leaving the maze alive, approximately 57.61%, compared to SARSA, which achieved 41.96%.

The performance difference can be attributed to the inherent characteristics of the two algorithms:

- *Q-learning*, an off-policy algorithm, learns the optimal policy by directly maximizing the Q-values for the best possible actions. This enables Q-learning to perform well in ideal evaluation conditions.

- *SARSA*, an on-policy algorithm, evaluates and improves the policy used during training, incorporating exploratory actions into its Q-value estimates. This results in more conservative strategies that are robust in environments requiring strict adherence to the behavior policy but may lead to suboptimal performance in purely evaluative scenarios.

In this maze environment, Q-learning outperformed SARSA, but this outcome may vary depending on the task and specific setup.

Algorithm	Initialization	α	ϵ	δ	$V(s_0)$	Prob. of Exiting Alive
Q-learning	3	0.55	0.2	-	5.89	0.5761
SARSA	3	0.51	-	0.95	4.49	0.4196

Table 1: Comparison of Q-learning and SARSA algorithms for the probability of successfully exiting the maze alive.

The probabilities achieved by the algorithms do not directly correspond to the Q-value of the initial state $V(s_0)$. This is because the Q-value depends on the reward function, which in our case was manually set and could take arbitrarily large values. Thus, $V(s_0)$ does not necessarily lie between 0 and 1, nor does it inherently represent the probability of successfully exiting alive.

If we redefine the reward function to assign:

- A reward of +1 for successfully exiting the maze alive,

- A reward of 0 for all other outcomes,

then the relationship between $V(s_0)$ and the probability of exiting alive becomes apparent. Specifically, for each episode k, we define a Bernoulli random variable:

$$X_k = \begin{cases} 1, & \text{if the player exits alive,} \\ 0, & \text{otherwise.} \end{cases}$$

Here, X_k represents the reward for episode k. Since episodes are independent, the sequence $\{X_k\}$ constitutes an i.i.d. random variable. The expected value of X_k, is the expected episode reward, which by definition equals the value function of the initial state, so $V(s_0) = \mathbb{E}[X_k]$. Furthermore, the probability of exiting alive, $P(\text{exit alive})$, can be expressed as:

$$P(\text{exit alive}) = P(X_k = 1) = \mathbb{E}[X_k].$$

Therefore, if we use the proposed reward structure, the Q-value of the initial state directly corresponds to the probability of exiting alive. Using the law of large numbers, this probability can also be estimated through simulation:

$$\lim_{k\to\infty} \frac{1}{k} \sum_{i=1}^{k} X_i = P(\text{exit alive}),$$

where the summation represents the fraction of episodes where the agent successfully exits the maze alive. Therefore, simulating a large number of episodes allows us to estimate this probability empirically.

2 RL with function approximation

Note on Experimental Variability: During the experiments conducted in this section, we encountered significant variability in the results. To address this, we applied smoothing techniques to many of the plots to better visualize the underlying trends. Additionally, confidence intervals were included wherever possible to provide a sense of the results' robustness. However, in cases where confidence intervals caused excessive occlusion, they were omitted to maintain clarity.

2.1 Description of the training process

We are solving the Mountain Car environment using SARSA(λ) algorithm with function approximation.

SARSA(λ) is an on-policy reinforcement learning algorithm that extends SARSA by incorporating eligibility traces to balance between short-term and long-term learning. At each step, an experience tuple (s, a, r, s', a') is sampled according to an exploration policy, such as ε-greedy, to ensure sufficient exploration of the state-action space. This experience is used to update the action-value function parameters incrementally. The parameter λ determines the weight given to past experiences, with $\lambda = 0$ focusing on immediate rewards and $\lambda = 1$ fully incorporating the episode's history. The main equations for updating the parameters are the following ones:

$$\mathbf{z}_a = \begin{cases} \gamma\lambda\mathbf{z}_a + \nabla_{\mathbf{w}_a} Q_{\mathbf{w}}(s_t, a), & \text{if } a = a_t \\ \gamma\lambda\mathbf{z}_a, & \text{otherwise} \end{cases} \quad \text{for all } a \in \{a_1, \ldots, a_A\}.$$

$$\omega \leftarrow \omega + \alpha\delta_t\mathbf{z} = \omega + \alpha(r_t + \gamma Q_{\mathbf{w}}(s_{t+1}, a_{t+1}) - Q_\omega(s_t, a_t))\mathbf{z}$$

Instead of implementing the traditional SARSA(λ) algorithm, we have implemented it with linear function approximators. This means that instead of having SxA parameters to define the Q-function, we parametrize it by a linear combination of the features of the states using a set of basis functions $\phi_1, ..., \phi_m$. Therefore, $Q_w(s, a) = w_a T\phi(s)$. We have done this by means of a Fourier Basis $\phi_i(s) = cos(\phi\eta_t^T s)$. After different experiments (section 2.2), we found the following coefficients (which represent the complete basis) to be the best performing basis:

$$\eta = [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]$$

As for the training process, we implemented Stochastic Gradient Descent (SGD) with momentum. In addition, to improve the performance we performed hyperparameter tuning, leading to the following combination (more detailed description in Section 2.3):

- Learning rate $\alpha = 0.001$ with reduction factor of 0.7
- Discount factor $\gamma = 1$
- For the eligibility trace, $\lambda = 0.6$

- Momentum = 0.4
- For the exploration strategy ($\epsilon - greedy$), $\epsilon = 0.1$ with decaying factor of 0.9999
- Number of episodes = 200
- Zero initialization of the weights

In addition, in order to stabilize the learning process, different techniques were implemented. Firstly, as it has already been mentioned, instead of the classical SGD we implemented SGD with momentum to help reduce oscillations in the parameter update (SGD with Nesterov acceleration is also an alternative that was tested but did not present any significant improvements on performance). Secondly, we scaled the learning rate for each basis function ϕ_i (ie. $\alpha_i = \alpha/||\eta_i||_2$). Finally, as has been said when defining the hyperparameters, both the learning rate α and the exploration's ϵ were reduced during training by a decaying factor.

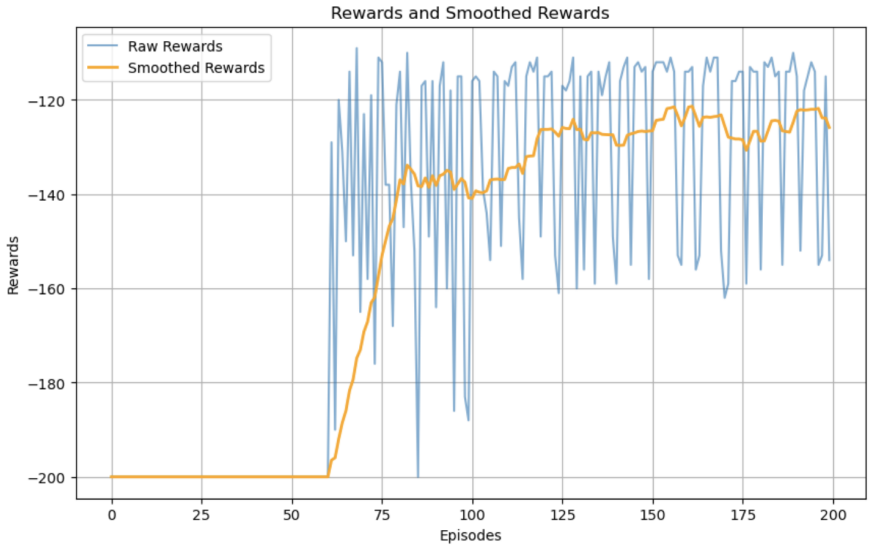


Figure 5: Rewards and smoothed rewards obtained throughout the training process

Figure 5 shows the episodic total reward across episodes during training. As it can be seen, in the first ≈ 60 episodes there is no increase in the episodic reward, as the model still has a lot to explore and learn until he is able to reach the goal in less than 200 steps. However, there is a turning point in which the reward rapidly rises to around -140, and from there it steadily continues to increase until it reaches approximately -125 episodic reward. It is worth-noting the elevated variability in the rewards between consecutive episodes, that is why we also plotted the smoothed rewards over a window of size 50.

Finally, test performance over 50 episodes results in an average total reward of -118.5 +/- 3.5 with confidence 95%.

2.2 Design of the Fourier Basis Coefficients

To optimize the performance of the Fourier basis, we experimented with different sets of coefficients to identify the most effective design. The coefficients we tested are as follows:

$$\eta_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \eta_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad \eta_3 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad \eta_4 = \begin{bmatrix} 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{bmatrix}, \quad \eta_5 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{bmatrix}$$

The first design, η_1, is the smallest basis, assuming independence between position and velocity, with no feature coupling. The second design, η_2, introduces feature coupling by adding the coefficient $[1, 1]$ but excludes $[0, 0]$. The third design, η_3, extends η_2 by including $[0, 0]$. Moving to larger and more complete bases, η_4 represents a fully coupled basis but excludes $[0, 0]$, while η_5 is the most complete basis, including all possible coefficients such as $[0, 0]$. The results of training the agent with these different Fourier basis designs are shown in Figure 6.

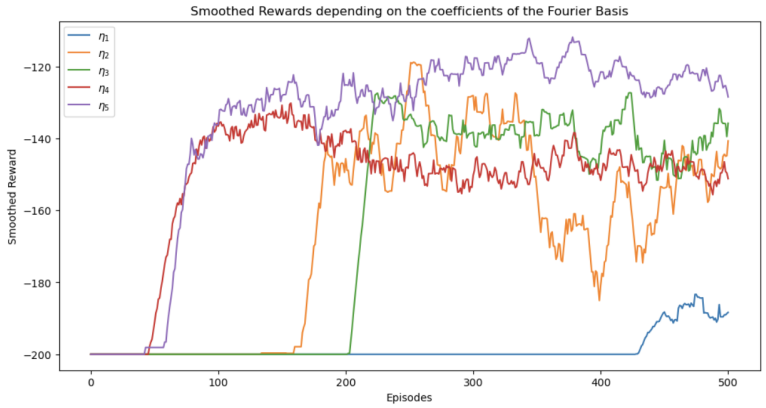
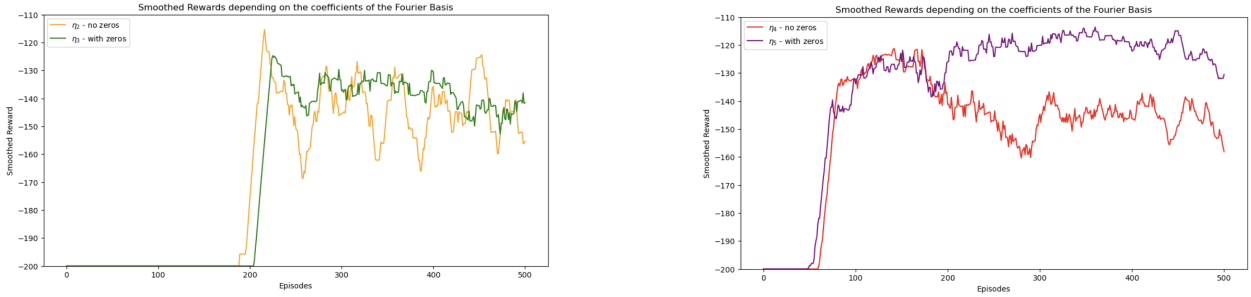


Figure 6: Smoothed rewards (window size = 20) through the training process with varying coefficients of the Fourier basis.

From the results, we observe that η_1 is the worst-performing basis. This is because position and velocity are not independent in the Mountain Car environment, and a lack of feature coupling prevents the agent from learning optimal policies. In contrast, η_5, the most complete basis, performs the best. By using the full set of coefficients, the model can better approximate the Q-function, as it can capture finer details in the environment's dynamics by combining more cosine terms. The intermediate bases, η_2 and η_3, show weaker performance compared to η_5, likely due to their smaller number of coefficients, which limits the approximation capacity of the Fourier basis.

To further analyze the impact of including the $[0, 0]$ coefficient, we separately compared the pairs η_2 vs. η_3 and η_4 vs. η_5. These comparisons, shown in Figure 7, highlight the differences in performance.



(a) η_2 vs. η_3 (b) η_4 vs. η_5

Figure 7: Comparison of smoothed rewards with and without $[0, 0]$ in the coefficients of the Fourier Basis.

From Figures 7a and 7b, we observe that including the $[0, 0]$ coefficient leads to more stable training, as seen in the reduced oscillations in the average total reward. Furthermore, its inclusion is even more important when the base is bigger. This coefficient corresponds to the constant term in the Fourier basis, representing the global average of the Q-function.

$$\eta_0 = [0, 0] \rightarrow \phi_0(s) = cos(\pi \mathbf{0}^T s) = cos(0) = 1$$

$$Q_w(s, a) = w_a^T \phi(s) = w_{a,0} + \sum_{i=1}^{m} w_{a,i}\phi_i(s) = bias + \sum_{i=1}^{m} w_{a,i}\phi_i(s)$$

Including this term allows the model to establish a baseline for value approximation, which improves the overall stability and convergence of the learning process.

2.3 Hyperparameter tuning

2.3.1 Learning rate α

The learning rate (α) is a critical hyperparameter that governs how much the weights are updated during each iteration. We began by testing several fixed α values throughout the agent's training.

From Figure 8, we observe that $\alpha = 0.001$ performs the best, as it steadily approaches the optimal reward and maintains stability around the found solution. Smaller learning rates, such as $\alpha = 0.0001$, take significantly longer to show improvements, as the weight updates are minimal. For example, rewards do not improve until approximately episode 450. While these rates may eventually converge to a better solution, the slow learning process makes them impractical. Conversely, larger learning rates, such as $\alpha = 0.01$ or $\alpha = 0.1$, lead to instability. For $\alpha = 0.1$, the agent fails to increase its reward beyond -200, while $\alpha = 0.01$ shows initial progress but diverges later due to excessively large updates that prevent convergence.

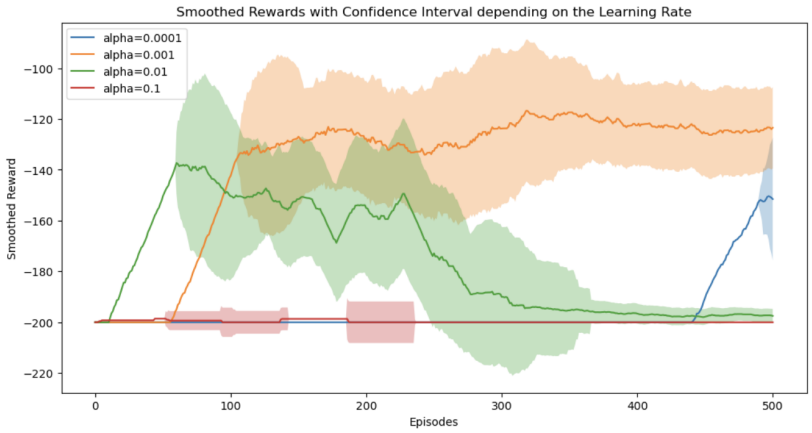


Figure 8: Smoothed rewards (window size = 50) with confindence intervals through the training process with different learning rates

This behavior can also be seen at test phase (Figure 9), where $\alpha = 0.001$ demonstrates significantly better performance than the other learning rates that have been tried

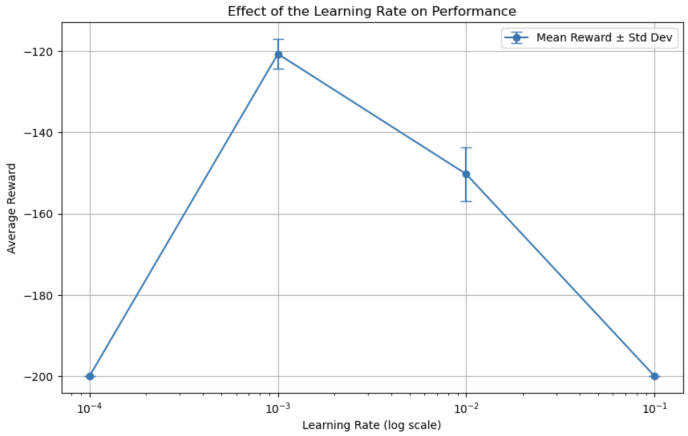


Figure 9: Average total reward and standard deviation over 50 episodes in test phase with different α values

Next, we experimented with a decaying learning rate to further optimize performance at $\alpha = 0.001$. The rationale is that as the agent approaches a good solution, smaller learning rates can fine-tune the weights for greater accuracy. We defined a threshold where the average reward over the last 50 episodes exceeds -135, at which point the learning rate is updated using $\alpha = \alpha \times \textit{reduction factor}$. Additionally, the reduction factor is reapplied whenever the average reward improves by increments of 5 (e.g., -135, -130, -125, etc.).

Figure 10 demonstrates that a reduction factor of 0.7 yields the best balance. Higher reduction factors do not sufficiently lower the learning rate, limiting the agent's ability to fine-tune the solution. Conversely, smaller reduction factors decrease the learning rate too quickly, preventing adequate weight updates to improve the solution. Thus, a reduction factor of 0.7 strikes an ideal balance between stability and adaptability, resulting in the best performance.

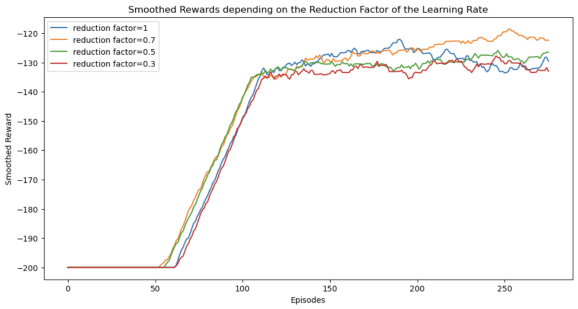


Figure 10: Smoothed rewards (window size = 50) through the training process with different reduction factors of the learning rate $\alpha = 0.001$

2.3.2 Eligibility trace λ

The λ parameter in eligibility traces plays a critical role in determining how updates are distributed across the actions that contributed to the total reward. In contrast to the classic SARSA algorithm (SARSA(0)), where at each time step only the Q-value of the current (s, a) pair is updated, eligibility traces allow credit to propagate to past actions. This makes them particularly valuable in environments with sparse rewards.

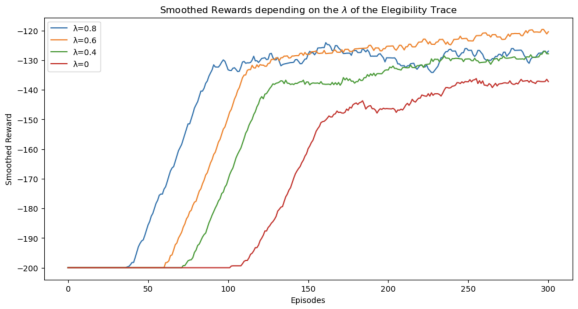


Figure 11: Smoothed rewards through the training process with varying λ values of the eligibility trace

From Figure 11 it can be seen that smaller λ values result in slower learning during the initial episodes. When λ is close to zero, only the weights associated with the current action a_t are updated, limiting the scope of updates. This delays the propagation of useful information to other state-action pairs, requiring more training steps for significant performance improvement. Conversely, as λ approaches 1, the algorithm updates the weights of multiple actions simultaneously, using information from the entire trajectory. This results in faster learning as credit is more broadly assigned to actions that contributed to the observed rewards.

In terms of the obtained solution, from Figure 12 it can be observed that the best performance is achieved with $\lambda = 0.6$, which provides a balance between updating multiple actions and focusing updates on the most relevant ones (recently visited and with higher Q-values). Larger λ values lead to slightly worse solutions as they distribute credit to less relevant actions. On the other hand, smaller λ values miss information from past time steps, resulting in slightly slower and less effective learning.

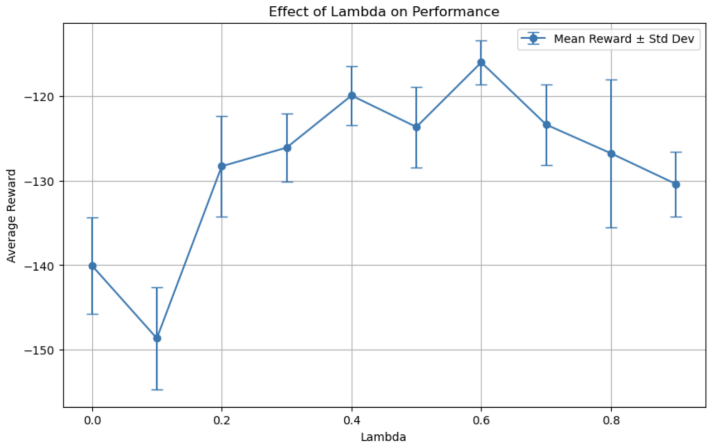


Figure 12: Average total reward and standard deviation over 50 episodes in test phase with different λ values

In summary, for this environment, $\lambda = 0.6$ achieves the optimal balance, efficiently sharing credit across actions while maintaining sufficient specificity to converge to an effective policy.

2.3.3 Parameter initialization

To analyse the impact of Q-value initialization on training performance, we tested three different initialization strategies for the weights w used to estimate the Q-function:

- *Zero initialization*: All weights are initialized to zero.
- *Uniform initialization*: Weights are sampled uniformly from the range $[-1, 1]$.
- *Normal initialization*: Weights are sampled from a normal distribution with mean 0 and standard deviation 1.

Through experimentation, we observed that the uniform and normal initialization strategies led to unstable training behavior, with significant variability in performance depending on the specific initial weights. To mitigate this variability, we trained 10 agents for each initialization strategy and averaged their results to enable a meaningful comparison. The average episodic rewards achieved by each initialization method over the course of training are shown in Figure 13.

Zero initialization clearly outperformed the other strategies, stabilizing at an average episodic reward of around -130. In contrast, both uniform and normal initialization methods converged to a reward of around -150, which is significantly worse. The superior performance of zero initialization can be attributed to its stability and predictability, which allow the agent to systematically learn the Q-function without the noise introduced by random initializations. In comparison, random initializations with either uniform or normal distributions often result in erratic initial Q-value estimates, causing the agent to prioritize suboptimal actions during the early stages of training. This erratic behaviour delays convergence and decreases overall performance.

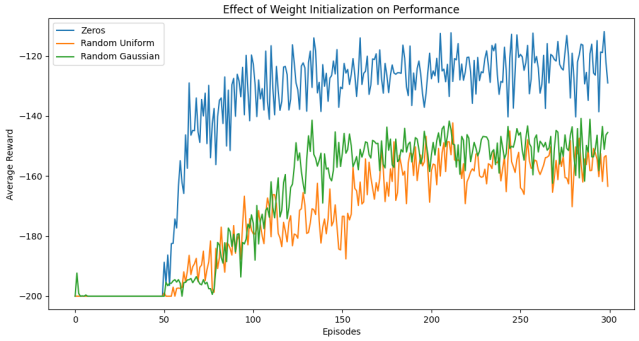


Figure 13: Average episodic reward as a function of training episodes for zero, uniform, and normal initialization weight methods.

In conclusion, zero initialization proved to be the most effective strategy in this environment, providing consistent performance across agents and avoiding the instability introduced by random initialization.

2.3.4 Exploration strategy

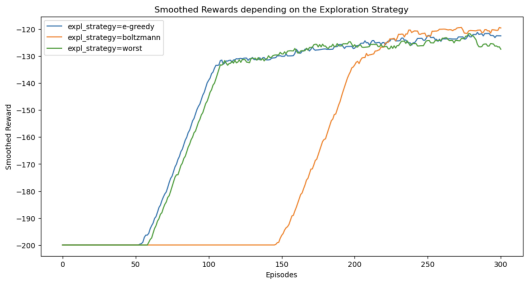
Until now, the current and next actions in a (s, a, r, s', a') experience were selected using an ϵ-greedy policy. This policy chooses a random action with probability ϵ and selects the action that maximizes the current estimate of the Q-function with probability $1 - \epsilon$. Formally:

$$a = \arg\max_{b \in A} Q_t(s, b).$$

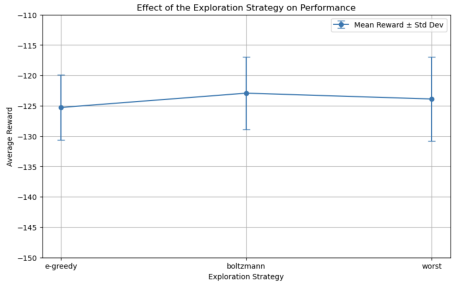
However, there are alternative exploration strategies that the agent can use. We tested the following:

- **ϵ-worst:** With probability $1-\epsilon$, the strategy selects the action that maximizes $Q(s, a)$. With probability ϵ, there is a 50% chance of selecting a random action and a 50% chance of selecting the action that minimizes $Q(s, a)$. The rationale is that an action minimizing $Q(s, a)$ might not have been explored enough, rather than being inherently suboptimal. Thus, this approach adds a slight bias toward exploring such actions.
- **Boltzmann exploration:** This strategy treats $Q(s, a)$ as a probability distribution over actions through the softmax function and samples an action accordingly. As a result, there is randomness at every time step, but actions with higher Q-values are more likely to be chosen.

It is worth-noting that hyperparameter tuning was performed in the ϵ-greedy strategy to find the optimal value of ϵ. We found that the best performing combination was $\epsilon = 0.1$ with a decay factor of 0.9999 at each time step ($\epsilon \leftarrow \epsilon * \textit{decay factor}$). These values provided a balance between sufficient exploration in the early episodes and selecting the best action as the agent gained knowledge of the Q-function. Smaller decay factors led to insufficient exploration, while larger ones reduced performance in later episodes due to an increased likelihood of random actions.



(a) Smoothed reward (window size = 50) during training



(b) Average reward and standard deviation over 50 test episodes

Figure 14: Effect of exploration strategies on training and performance.

From Figure 14a, we observe that both ϵ-greedy and ϵ-worst strategies perform similarly during the training process. However, in the final episodes, ϵ-worst shows a slight decrease in performance. Initially, selecting a random action or the "worst" action is comparable, as unexplored actions are likely to have low Q-values. Over time, as the agent learns which actions are suboptimal, ϵ-worst selects these poor actions with higher probability than ϵ-greedy, leading to a slightly degraded performance.

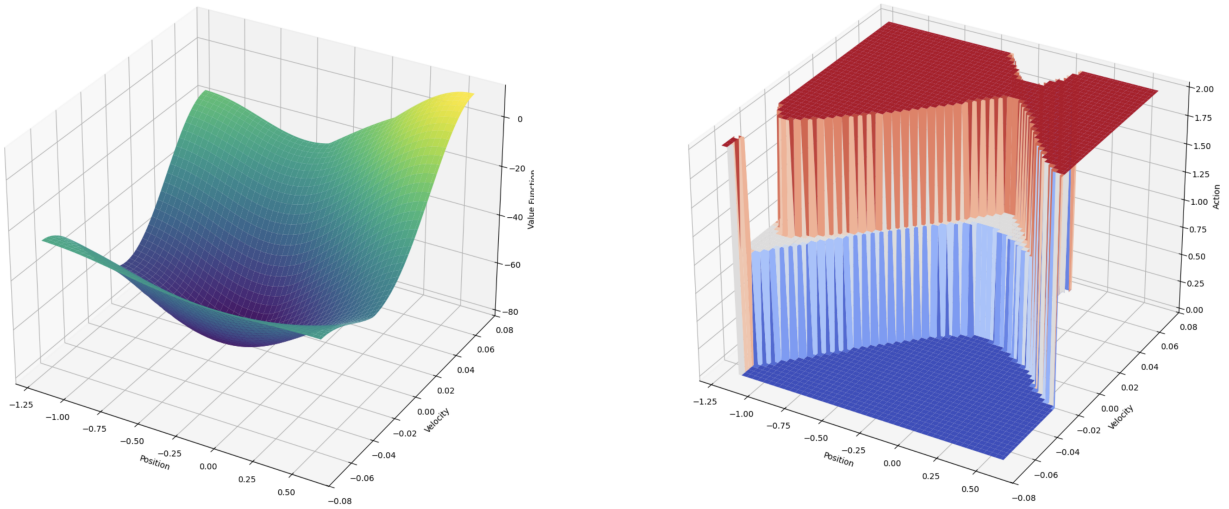
In contrast, as can bee seen in Figure 14a, the Boltzmann exploration strategy behaves differently. It takes longer for the agent to learn initially but achieves slightly better results once learning stabilizes, as shown in Figure 14b. Early in training, the softmax probability distribution over Q-values is nearly uniform because the agent lacks sufficient knowledge about the environment. As training progresses, the probability mass shifts toward optimal actions. This approach reduces the likelihood of selecting clearly suboptimal actions, unlike ϵ-greedy, where random actions may still be chosen even when they are evidently poor. However, due to the Boltzmann exploration strategy having more variance between different trainings and the performance not having a big improvement we decided to keep the $\epsilon-greedy$ as the exploration strategy.

2.4 Analysis of the Solution: Value Function and Optimal Policy

In Figure 15 we can see the value function and the optimal policy achieved with the best hyperparameters found.

By analyzing the **value function**, we observe that it accurately captures the dynamics of the Mountain Car environment. Higher values indicate fewer steps are required to reach the goal, as each step incurs a reward of -1. Therefore, higher values correspond to a greater total reward due to fewer pushes being necessary. The value function peaks in regions where the car has sufficient velocity to reach the goal, such as when it is near the goal with positive velocity.

Conversely, the value function is lowest in the valley between $x = -0.6$ and $x = -0.4$ when the car's velocity is small or zero. In this region, the car requires many pushes to gain sufficient momentum, resulting in a low value and consequently a lower total reward.



(a) Value Function (b) Optimal Policy

Figure 15: Representation of the value function and optimal policy obtained using the SARSA(λ) algorithm.

The **optimal policy** derived from the value function shows the appropriate action to take at each state to maximize total rewards. It reflects the learned strategy for efficiently navigating the environment, leveraging the car's momentum to minimize the number of steps required to reach the goal.

2.5 Analysis of the Learned Policy

When analyzing the **policy** learned by the agent, we observe that it aligns well with intuitive strategies for solving the MountainCar problem. The actions available to the agent are defined as follows:

- $a = 0$: push left
- $a = 1$: no push
- $a = 2$: push right

The learned policy primarily suggests pushing right ($a = 2$) when the car has a positive velocity. This allows the car to increase its speed and make further progress toward the goal. Interestingly, the policy also advises pushing right in scenarios where the car has negative velocity but is either far from or very close to the goal. However, as the car approaches the position $x \approx -0.1$, the policy shifts. Despite having a small negative or zero velocity, the agent recommends pushing left or taking no action.

This behavior occurs because attempting to push right near $x \approx -0.1$ with insufficient velocity would leave the car unable to climb the hill toward the goal. Instead, the policy correctly advises moving back down the hill to gain momentum for a stronger ascent. By prioritizing

building momentum over immediate progress, the agent ensures that the car can reach the goal efficiently without getting stuck mid-climb.

The learned policy demonstrates an understanding of the environment's dynamics. It avoids pushing in the direction opposite to the car's movement except in specific cases, such as steeper regions of the hill. This strategy prevents the car from losing valuable momentum and ensures sufficient velocity is maintained to overcome the steep inclines.

Furthermore, the policy exhibits a nuanced approach near the left hill. When the car is near the top of the left hill with a high negative velocity, the agent recommends pushing right ($a = 2$). This action prevents the car from reaching the summit of the hill or going out of bounds, ensuring it begins to gain velocity for the ascent toward the goal. The optimal strategy is to allow the car to climb as high as possible on the left hill without reaching the summit. Reaching the summit would require an extra push to descend, reducing the total reward by wasting valuable actions. By stopping just short of the peak, the car can gather maximum momentum for a more effective descent and subsequent climb.