Reinforcement Learning EL2805 - Lab 1

Rodrigo Montero (20040421-T054)
Daniel De Dios Allegue (20041029-T074)

December 2, 2024

1 Problem 1

In this section, we formulate the Minotaur's maze problem as a Markov Decision Process (MDP) and solve it using Dynamic Programming (DP), Value Iteration (VI), Q-Learning, and SARSA (including the bonus part). Additionally, we address and answer the theoretical questions.

1.1 Task 1a - Basic maze

Providing state space, action space, reward function, and transition probabilities is sufficient to define a Markov decision process (MDP).

$$MDP = \{T, S, (A_s, p_t(\cdot, s, a), r_t(s, a), 1 \leq t \leq T, s \in S, a \in A_s)\}$$

In the rest of this section, we will describe our formulation of the Minotaur's maze problem as an MDP.

1.1.1 State Space

For defining the state space, we will consider the maze with dimensions (M_x, M_y) and the subset of special cells W consisting of the walls of the maze. The definition of the state space is done as the union of the set of non-terminal states, which we denote S_{nt}, and the set of terminal states, which we denote S_t.

$$S = S_{nt} \cup S_t$$

The non-terminal states are represented by tuples containing the player's position and the Minotaur's position on the map. We should also consider that the player cannot step on the wall cells contained in W. Thus, the invalid situations where the player is inside a wall are not included in the state space.

$$S_{nt} = \{(x_p, y_p, x_m, y_m) \mid 1 \leq x_p \leq M_x, 1 \leq y_p < M_y, 1 \leq x_m < M_x, 1 \leq y_m < M_y, (x_p, y_p) \notin W\}$$

The set of terminal states includes two cases:

1. The state s_{exit} where the player is in the exit cell (x_b, y_b) and the Minotaur is not in the exit cell, i.e., $x_p = x_b \wedge y_p = y_b \wedge (x_m \neq x_b \vee y_m \neq y_b)$, which means the player has exited alive.

2. The state s_{caught} where the player is in the same cell as the Minotaur, i.e., $x_p = x_m \wedge y_p = y_m$, which means the player has been caught, even at the exit.

Thus, the terminal states are:

$$S_t = \{s_{exit}, s_{caught}\}$$

We define two distinct terminal states ("exit" and "caught") rather than handling all states as general maze positions. This approach reduces the state space and simplifies both notation and implementation.

1.1.2 Action Space

The possible actions that the player can take after observing the state are moving one cell in any direction (not in a diagonal) or staying in the same cell. Therefore, the action set is:

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

It is necessary to limit the actions available in each state according to the player's valid possibilities. Denoted the exit coordinates as (x_b, y_b), we define the following restrictions:

- Out of the maze: In the states where the player is located at the border of the maze (i.e., $x_p = 1$ or $x_p = M_x$ or $y_p = 1$ or $y_p = M_y$), the action set is restricted in such a way that the player cannot choose to go outside of the maze.

- Inside a wall: In the states where the player is located adjacent to a wall $(x_w, y_w) \in W$, the action set is restricted so that the player cannot choose to go inside the wall.

- Terminal state: In the terminal states (i.e., the player has already exited or been caught by the Minotaur), the action set is restricted so that the player can only stay in the same position.

1.1.3 Transition Probabilities and Rewards

In the following, we denote with $s_p = (x_p, y_p)$ the player position and with $s_m = (x_m, y_m)$ the Minotaur position. We introduce $M(s_m)$ as the set of possible cells that the Minotaur can reach starting from the Minotaur position s_m. The cardinality $|M(s_m)|$ depends on the Minotaur position, as it is not possible to go out of the maze. Additionally, we restrict $M(s_m) = \emptyset$ in any terminal state.

For any of the two terminal states $s \in S_t$, we have:

$$P(s \mid s, stay) = 1, \forall s \in S_t$$

$$r(s, stay) = 0, \forall s \in S_t$$

Let us define a function f to denote the next deterministic cell the player can move to depending on the action. The function takes as input the current position of the player and the action and returns the next position of the player:

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

For any non-terminal state $s \in S_{nt}$, the next player position is entirely determined by the function f, while the Minotaur position can change according to the available next cells $M(s_m)$. Therefore, given a state s and an action a, we have the set of next states S' for which the transition probabilities are not null. Notice that the next state $s' \in S'$ can be a terminal state (exit alive or caught). To keep the notation clean, we do not write it explicitly. The transition probabilities are:

$$S' = \{(f(s_p, a), s_m) \mid s_m \in M(s_m)\}$$

$$P(s' \mid s, a) = \frac{1}{|M(s_m)|}, \forall s' \in S'$$

For the non-terminal states $s \in S_{nt}$, we design a reward that depends on the current state s and the next state s'. We also distinguish the cases of finite-horizon MDP and discounted MDP.

$$r(s, a, s_{exit}) = +1, \forall s \in S_{nt}, \forall a \in A_s$$

$$r(s, a, s_{caught}) = 0, \forall s \in S_{nt}, \forall a \in A_s$$

$$|T| < \infty \Rightarrow r(s, a, s') = -0.0001, \forall s \in S_{nt}, \forall a \in A_s, \forall s' \in S_{nt}$$

$$T \to \infty \Rightarrow r(s, a, s') = 0, \forall s \in S_{nt}, \forall a \in A_s, \forall s' \in S_{nt}$$

The positive exit-alive reward encourages the player to maximize the probability of exiting the maze alive, which is the main objective of the problem.

In the finite-horizon MDP, a small step penalty encourages the player to select shorter paths among those that lead to the exit with the same survival probability, reinforcing efficient movement as a secondary goal. The scale of this penalty is important: it must remain minor relative to the exit reward, so the main objective—maximizing the probability of exiting alive—remains the player's priority. If the cumulative step penalty were too large, the player might prioritize quick exits over safe paths, risking encounters with the Minotaur.

In both the finite-horizon and discounted MDPs, we apply an additional penalty for colliding with the Minotaur, discouraging risky moves that lead to capture. This collision penalty supports the primary objective of maximizing survival by reinforcing the need to avoid the Minotaur. In the discounted MDP, the discount factor already encourages shorter paths by prioritizing immediate rewards, so an extra step penalty is not necessary. The penalty for Minotaur encounters, however, applies consistently to ensure that players actively avoid high-risk paths, aligning with the survival objective without conflicting with the goal of safely reaching the exit.

1.2 Task 1b - Alternating Rounds: Does it affect the Minotaur catching the player?

In the original problem formulation, both the player and the Minotaur move simultaneously. However, if they were to alternate rounds, the dynamics of the problem would change. Specifically, the player would have more control over the position of the Minotaur, since they would be able to move before the Minotaur does. This increases the player's chances of avoiding the Minotaur, especially if they can predict the Minotaur's next move.

When the player and the Minotaur do not move simultaneously:
- The player could always move first, potentially avoiding getting caught by moving out of the Minotaur's immediate path.
- The Minotaur, on the other hand, would be constrained by the player's first move and would need to adapt afterwards.

From an MDP perspective, this alternating move structure changes the transition probabilities, as the player's first move affects the Minotaur's possible next positions. Although the reward structure remains the same, this dynamic indirectly improves the player's ability to reach high-reward states by avoiding capture more effectively.

Therefore, when the Minotaur and the player do not move simultaneously, it is less likely for the Minotaur to catch the player, as the player has more time to react to the Minotaur's movements. This would generally lead to a more favourable outcome for the player in terms of survival.

1.3 Task 1c - Dynamic Programming - Policy

Dynamic Programming (DP) can effectively solve finite-horizon Markov Decision Processes (MDPs), enabling the player to maximize rewards by avoiding the Minotaur and reaching the exit.

In this setup, the player aims to avoid being caught by the Minotaur, which would end the game prematurely. Additionally, given multiple paths with the same survival probability, the player prefers the shortest one to minimize step

penalties and reach the exit faster.

An optimal policy generally involves following the shortest path to the exit and moving continually, which reduces the chance of being caught by the Minotaur. This behavior aligns with the intuitive idea that continual movement lowers the risk of capture, especially if the Minotaur's position is unpredictable. Figure 1 shows a simulation where the player follows the shortest path to the exit while effectively avoiding the Minotaur.

Figure 2 demonstrates the actions chosen by the optimal policy for each player position. In this case, the Minotaur is fixed at the exit, and the player consistently moves towards the exit, minimizing their time on the board. With 20 moves left (at time t=1), the optimal policy guides the player along the shortest path, ensuring that they avoid staying still, reducing capture risk, and reaching the exit as quickly as possible.

If reaching the exit becomes impossible, the optimal strategy may shift toward maximizing survival time by avoiding unnecessary risks, such as staying in less exposed cells away from the Minotaur.

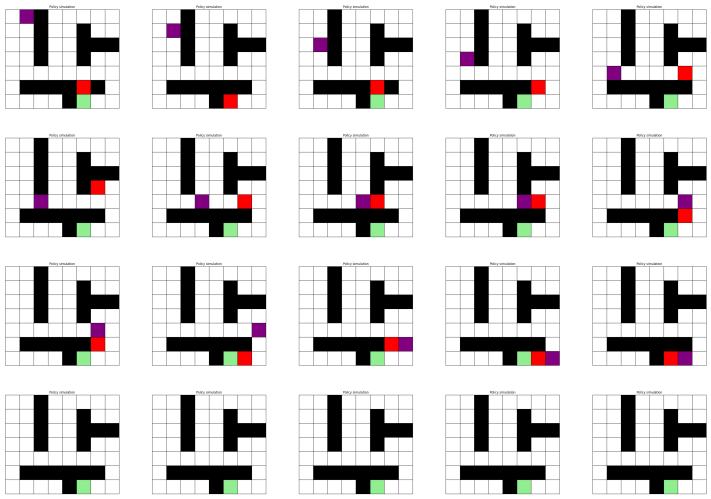

Figure 1: Simulation of a game played by the optimal policy with time horizon T=20. Notation: Purple Square = player, Red Square = Minotaur, Black Square = wall, White Square = free cell, Green Square = End.

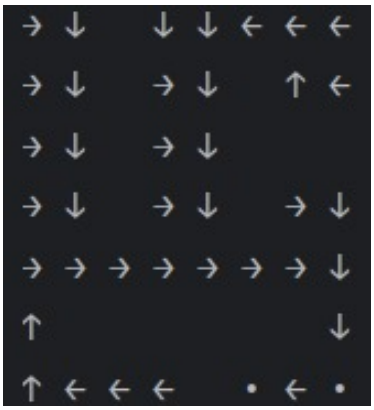

Figure 2: Optimal policy at time t=1 (20 moves left). Notation: move = arrow, stay = '·', wall = Blank space

1.4 Task 1d - Dynamic Programming pt2

As mentioned earlier, Dynamic Programming (DP) is effective for solving finite-horizon MDPs. The time horizon T is crucial because it impacts whether the player can reach the exit. If T is too short, the player may not have enough time to reach the goal. For small values of T, the information doesn't propagate well, causing the algorithm to fail in estimating the correct policy.

Figure 3 illustrates how the probability of exiting the maze depends on the time horizon. When T is smaller than the shortest path length (i.e., $T < 15$), the player cannot reach the exit in time, and the probability of exiting is 0. However, if there is enough time ($T \geq 15$), the player can eventually reach the exit.

The probability of survival also depends on the Minotaur's movement. If the Minotaur cannot stand still, the player can always escape for any $T \geq 15$. However, if the Minotaur can stand still, the probability of escaping decreases because the Minotaur becomes more dangerous. For example, if the Minotaur stands still at the exit or in a bottleneck, the player cannot escape at all. But if the Minotaur moves randomly, the longer the time horizon, the higher the probability the player has to escape, as the player can wait for the Minotaur to move away.

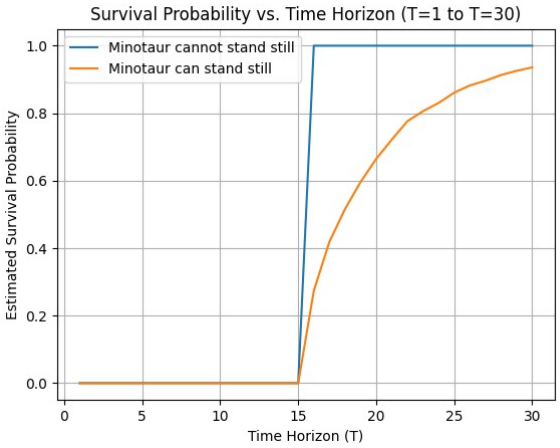

Figure 3: Probability of exiting the maze based on time horizon T, showing two scenarios: when the Minotaur can stay still and when the Minotaur must move each turn.

1.5 Task 1e - Value Iteration

In this problem, we need to adjust the MDP to account for the agent being poisoned, which affects its lifetime. The agent's survival follows a geometric distribution with a mean of 30 steps. Our goal is to maximize the probability of the agent exiting the maze before dying from the poison. This requires adjusting the reward structure and using an appropriate discount factor.

Unlike the previous model with a fixed time horizon, we now have a random time horizon. The agent's lifetime is governed by a geometric distribution, with $\mu = 30$ representing the average number of steps the agent can survive.

The mean lifetime μ is related to the discount factor λ, which shows how future rewards are discounted:

$$\mu = \frac{1}{1-\lambda} \quad \Rightarrow \quad \lambda = \frac{\mu - 1}{\mu}.$$

With $\mu = 30$, we get:

$$\lambda = \frac{29}{30}.$$

The discount factor λ reflects the agent's decreasing survival probability with each step.

We want to find the policy that maximizes the probability of the agent exiting the maze alive. Since the lifetime is uncertain, we need to adjust for the

random time horizon and rewards.

The expected total reward is:

$$E\left[\sum_{t=0}^{T}\lambda^t r(s_t,a_t)\right] = E\left[\sum_{t=0}^{\infty}\lambda^t r(s_t,a_t)\right],$$

where T is the random lifetime and $r(s_t, a_t)$ is the reward at each step.

To handle this, we use **Value Iteration**, which updates the value function until the changes are small enough. This approach helps find the optimal policy given the agent's uncertain lifetime.

1.6 Task 1f - Value Iteration pt2

After taking into account the poison probability of 1/30 and computing the optimal policy using Value Iteration, we simulated 10,000 games to estimate the probability of the agent successfully exiting the maze. The obtained probability of survival after these simulations is 58.89%.

1.7 Task 1g - Theoretical Questions

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

In reinforcement learning, the terms *on-policy* and *off-policy* refer to the relationship between the policy used to generate behavior and the policy being improved through learning.

- *On-policy*: In on-policy methods, the policy that is being improved (the target policy) is the same as the policy used to generate actions (the behavior policy). In other words, the agent learns from its own actions as guided by its current policy. An example of an on-policy method is SARSA (State-Action-Reward-State-Action), where the agent updates its policy based on the actions it actually takes.

- *Off-policy*: In off-policy methods, the behavior policy and the target policy are different. The agent can learn from actions taken by a different policy (e.g., exploratory behavior or actions taken by another agent). An example of an off-policy method is Q-learning, where the agent updates the Q-values based on the maximum possible future reward (using a greedy policy) rather than the policy it followed to collect the experiences.

2) State the convergence conditions for Q-learning and SARSA.

- *Q-learning convergence condition*: Q-learning is guaranteed to converge to the optimal Q-values (and thus an optimal policy) if the following conditions hold:

- The learning rate α must decay over time, i.e., $\sum_{t=1}^{\infty}\alpha_t = \infty$ and $\sum_{t=1}^{\infty}\alpha_t^2 < \infty$.

- Every state-action pair must be visited infinitely often, which implies that the agent must explore all possible state-action pairs sufficiently.
- The reward function must be bounded.

Under these conditions, Q-learning will converge to the optimal Q-values with probability 1, as long as the policy follows a greedy strategy after sufficient exploration.

- *SARSA convergence condition*: SARSA converges to the optimal policy under similar conditions, with the key difference being that it updates the Q-values based on the current action taken (the action chosen by the policy), rather than the optimal action:

- The learning rate α must decay as in Q-learning, satisfying $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$.
- Every state-action pair must be visited infinitely often, similar to Q-learning.
- The reward function must be bounded.

The key difference is that SARSA converges to the policy it is following (which may not be optimal if the agent is exploring). If the policy is not greedy or optimal, SARSA will converge to a suboptimal policy, but it still guarantees convergence as long as the exploration ensures every action in every state is visited infinitely often.

1.8 Task 1h - Secondary Scenario

To consider this new case, we need to modify our previously defined MDP. We can do such tasks as follows:

To modify our state space, we now introduce an additional dimension to represent whether the player has collected the keys. Previously, each state was represented by a 2-tuple. Now, we represent each state as a 3-tuple: $(\text{player\_position}, \text{minotaur\_position}, \text{has\_keys})$, where has_keys is a binary variable (0 or 1) indicating whether the player has acquired the keys.

The transition probabilities are now defined as follows: With 35% probability it moves one step closer to the agent and with 65% probability, it moves randomly (up, down, left, right). The transition probabilities for the player remain the same.

To model the new requirements we need to adjust the reward structure we defined before. To be able to escape the maze through cell B, we need to have previously collected the keys at C. Hence we define a new reward structure:

- **Goal Reward for reaching B:** This reward should only be provided if the agent has already visited C.

- **Step Reward:** Negative reward (e.g., -1) for each time step to encourage finding a solution quickly.
- **Poison Penalty:** We incorporate a probability that in each step the agent loses life due to poison, reflecting the geometric life expectancy.
- **Minotaur Capture Penalty:** Negative reward if the Minotaur catches the agent.
- **Impossible Move Penalty:** Negative reward for attempting to move into walls.

The rest of the MDP tuple defined in the first part can stay the same since it already complies with the requirements defined in this new problem.

1.9 Task 1i - Q-Learning

To solve Problem (h), we need to modify the standard Q-learning algorithm to accommodate the specifics of the scenario: First, the agent must reach the keys at position C before it can exit at position B. The agent's life is geometrically distributed with mean 50 (probability of dying from poison at each step is $p = \frac{1}{50}$). The minotaur moves towards the agent with probability 35% and randomly with probability 65% and the state space must include the agent's position, the minotaur's position, and whether the agent has the keys.

Hence, we introduce the pseudocode for the $\epsilon - greedy$ approach. Note that:

- N: Number of episodes
- ϵ: Exploration parameter
- α: Step size exponent (for $\alpha_t = \frac{1}{(n_{s,a})^\alpha}$)
- γ: Discount factor (typically $0 \leq \gamma < 1$)
- R_{goal}: Reward for successfully exiting the maze with the keys

The following code represents the pseudocode for the Q-learning implementation with the $\epsilon - greedy$ approach.

Algorithm Pseudocode

1. **For** each episode $k = 1$ to N
   (a) Initialize the episode
   (b) **While** episode k is not finished
      i. **Select action** a_t:
         - With probability ϵ, choose a_t uniformly at random from $A(s_t)$

- Else, choose $a_t = \arg\max_{a \in A(s_t)} Q(s_t, a)$

ii. **Execute action** a_t:
- Move agent to new position according to a_t (obeying maze constraints)
- Move minotaur:
  - With probability 0.35, minotaur moves towards the agent
  - With probability 0.65, minotaur moves randomly to an adjacent cell
- Update `key_status`:
  - If agent's position $= C$, set `key_status` $= 1$
- Check for poison:
  - With probability $p = \frac{1}{50}$, agent dies (episode ends)
- Check for minotaur encounter:
  - If agent's position = minotaur's position, agent dies (episode ends)
- Check for successful exit:
  - If agent's position $= B$ and `key_status` $= 1$, agent exits successfully (episode ends)

iii. **Observe reward** r_t:
- If agent exits successfully, $r_t = R_{\text{goal}}$
- If agent dies, $r_t = 0$ (or a negative reward if desired)
- Else, $r_t = 0$

iv. **Observe next state** $s_{t+1} = (\text{agent\_pos}, \text{minotaur\_pos}, \texttt{key\_status})$

v. **Update counts and Q-values:**
- $n(s_t, a_t) = n(s_t, a_t) + 1$
- $\alpha_t = \frac{1}{[n(s_t, a_t)]^{\alpha}}$
- $Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right)$

vi. **Update state and time:**
- $s_t = s_{t+1}$
- $t = t + 1$

vii. **If** episode ended (agent exited or died), **break** out of the loop

To answer part (2), we fixed a value of $\alpha = \frac{2}{3}$ and we tried 2 different values for $\epsilon = 0.1, 0.9$. The first value (i.e. $\epsilon = 0.1$) encourages the agent to explore more and exploit the best strategy less. On the other hand, with a larger value such as 0.9, we expect the agent to exploit the best strategy most of the time and hence it should converge faster towards the best path within the maze. In Figure 4 we show the results of the convergence of the value function over 5000 episodes. It is clear that with the value of 0.9, the agent learns how to traverse the maze most efficiently and therefore only on a few occasions it is not able to achieve the task. On the other hand, with a value of 0.1, the encouraged exploration causes the agent to fail on several occasions over later episodes and

thus we can conclude that higher epsilon values do affect the convergence speed of our agent. It is important to note that rewarding exploration to some extent benefits the agent; especially in the initial episodes.

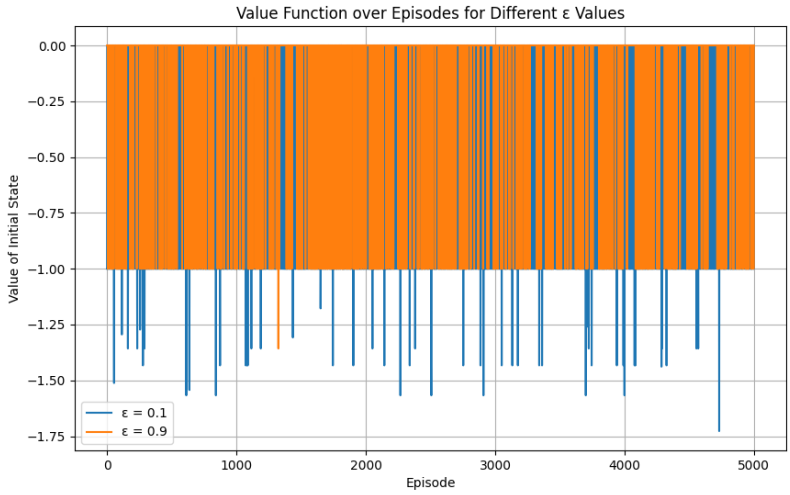

Figure 4: Value function over Episodes with varying ϵ values

To answer part (3), we fix the value $\epsilon = 0.1$ to allow the agent to explore and see the effect of different values of α in the step. In our case, we have made an experiment with $\alpha = 0.6$ and another one with $\alpha = 0.9$ and the results are shown in Figure 5. We should expect that with larger values for α, the convergence is faster but there is a risk that we find only suboptimal policies. On the other hand, smaller values tend to produce slower convergence agents but the learning process should be smoother due to the adaptability of the policies. In Figure 5 we see a larger variance in the values produced by $\alpha = 0.9$ than with the smaller value. Over a larger number of episodes, we see that this variance is reduced and the policies converge within a fixed range later on. However, for $alpha = 0.6$ the policy does not seem to converge optimally and therefore we see fluctuations in the values even within a large number of episodes. Hence, our results clearly illustrate our expected behaviour with varying values for α.

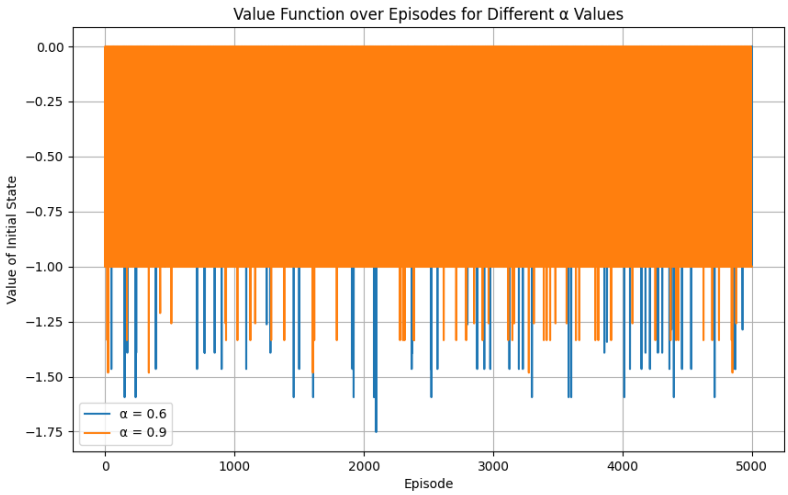

Figure 5: Value function over Episodes with varying α values

1.10 Task 1j - SARSA

To answer part (1), SARSA is an on-policy algorithm and hence it learns the optimal policy by learning from the actions that the agent chooses. In Q-Learning, we update the values with:

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

and in SARSA:

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

In Q-learning, the policy is learned independently from the agent's actions and hence we only take into account the action which maximizes the Q-value.

In part (2), we run SARSA over 5000 episodes with 2 different values of $\epsilon = 0.1, 0.2$ and a step size fixed with $\alpha = \frac{2}{3}$. We expect that with $\epsilon = 0.1$ the agent is encouraged to explore less and hence the convergence should be smoother and faster. On the other hand; with $\epsilon = 0.2$ we expect the agent to explore more and hence the value curve should have a larger variance and we expect it to reach the minimum slower since there is a higher risk of reaching a suboptimal minima. In Figure 6 we can see this effect. The curve representing $\epsilon = 0.2$ is less smooth showing the stochastic component of exploration.

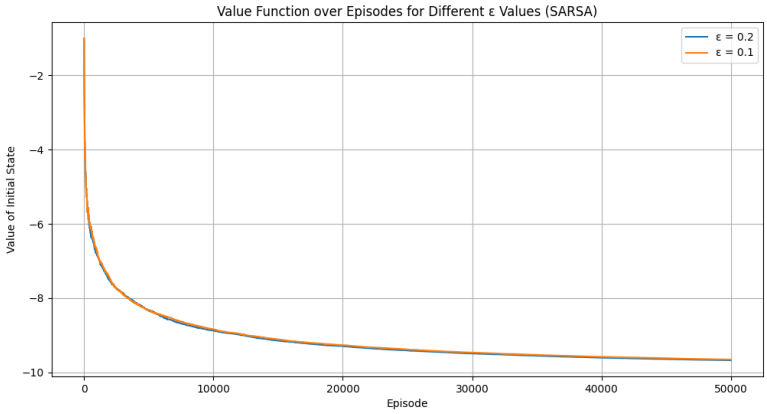

Figure 6: Value function over Episodes with varying ϵ values

In part (3), instead of fixing the value of the exploration parameter ϵ we decrease it every episode:

$$\epsilon_k = \frac{1}{k^\delta}, \delta \in [0.5, 1)$$

In our experiment, we ran 2 cases with $\delta = 0.7$ and $\alpha = 0.6, 0.8$ to test whether it is better to have $\alpha > \delta$ or vicerversa. The results are clear: it is better to have a value of $\alpha > \delta$ since there is a steady convergence with the value of $\alpha = 0.8$ but there is no convergence at all with $\alpha = 0.6$. Whenever $\alpha < \delta$, the learning rate decreases faster than the exploration rate ϵ_k and therefore the values are potentially stabilized sooner. In the other case, the learning rate decreases slower and therefore the exploration component is still an important part of the process, potentially causing the agent to reward exploration more than exploitation even within a larger number of episodes.

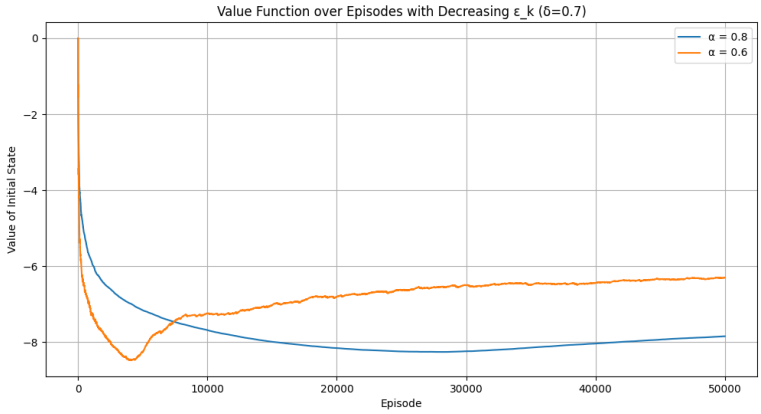

Figure 7: Convergence comparison with different α values and $\delta = 0.7$

1.11 Task 1k - Estimating Probabilities

In this task, we estimate the probability of exiting the maze using policies learned by Q-learning and SARSA, and compare these probabilities with the Q-values of the initial state.

For Q-learning, after training with $\alpha = \frac{2}{3}$, $\epsilon = 0.1$, and $\gamma = 0.9$ over 50,000 episodes, the estimated probability of exiting the maze was 0.0774, and the Q-value of the initial state was -4.5967.

For SARSA, under the same parameters, the estimated probability was 0.1525, and the Q-value of the initial state was -9.6564.

The differences in probabilities reflect the nature of the learning algorithms: Q-learning tends to focus more on maximizing future rewards, while SARSA learns from the agent's actual experience, leading to a more exploratory policy. The negative Q-values in both cases indicate suboptimal starting points, and the estimated probabilities align with the distinct learning dynamics of each method.

Thus, while Q-values provide insight into expected cumulative rewards, they don't directly predict success probabilities, which are influenced by the reward structure and exploration during training.

2 Part 2

2.1 Introduction

This part aimed to solve the MountainCar environment using linear function approximators and the Sarsa(λ) reinforcement learning algorithm. The MountainCar environment is characterized by continuous state spaces and discrete action spaces. The state vector $s = [s_1, s_2]$ consists of position (s_1) and velocity (s_2), and the available actions are: push left (0), no push (1), and push right (2). The task is episodic, ending when the agent reaches the goal position ($s_1 = 0.5$) or takes 200 actions.

The solution employs a Fourier basis for feature extraction and seeks to find a policy π that maximizes the episodic total reward. The environment is unknown, requiring a model-free approach.

2.2 Training Process

The training process employed the Sarsa(λ) reinforcement learning algorithm, leveraging a Fourier basis for linear function approximation. Key details of the training setup are as follows:

- **Fourier Basis Order:** $p = 2$.
- **Number of Episodes:** $N_{\text{episodes}} = 400$.
- **Learning Rate:** $\alpha = 0.001$.
- **Discount Factor:** $\gamma = 1.0$.
- **Eligibility Trace Parameter:** $\lambda = 0.5$.
- **Exploration Strategy:** Epsilon Greedy exploration.
- **Q-value Initialization:** Zero initialization.

2.2.1 Algorithm Description

The Sarsa(λ) algorithm operates by updating a weight matrix W that parameterizes the Q-function. For each episode:

1. The agent starts in a random initial state with the state variables scaled into the $[0, 1]^2$ range.
2. The Fourier basis transforms the state into features for linear function approximation.
3. The action is selected using the ϵ-greedy exploration strategy:

$$a = \begin{cases} \text{random action with probability } \epsilon \\ \text{argmax}_{a'} Q(s, a') \text{ with probability } 1 - \epsilon \end{cases}$$

The exploration parameter ϵ decays over time to balance exploration and exploitation, ensuring sufficient exploration during early episodes and greater exploitation of the learned policy in later stages.

4. The agent interacts with the environment, updating the eligibility traces E, which decay according to the parameters γ and λ and are clipped to avoid large updates:

$$E = \text{clip}(E \cdot \gamma \cdot \lambda, -5, 5)$$

5. Weight updates are performed using stochastic gradient descent with momentum:

$$v = m \cdot v + \alpha \cdot \delta \cdot E$$

$$W = W + v$$

where v is the momentum term, α is the learning rate, and δ is the temporal difference error.

6. The Q-values are updated using:

$$\delta = r + \gamma Q(s', a') - Q(s, a)$$

2.2.2 Stochastic Gradient Descent Modifications

The training process incorporated SGD enhancements:

- **Momentum:** Momentum $m = 0.9$ was used to accelerate convergence and smooth updates.
- **Eligibility Trace Clipping:** Traces were clipped to prevent excessively large updates, improving stability.

The training process converged within 400 episodes, achieving an average total reward of at least -135 over 50 episodes.

2.3 Analysis of the Policy

2.3.1 Episodic Total Reward During Training

Figure 8 shows how the episodic total reward changes across episodes during training. The reward converges rapidly within the first 50 episodes despite significant variance.

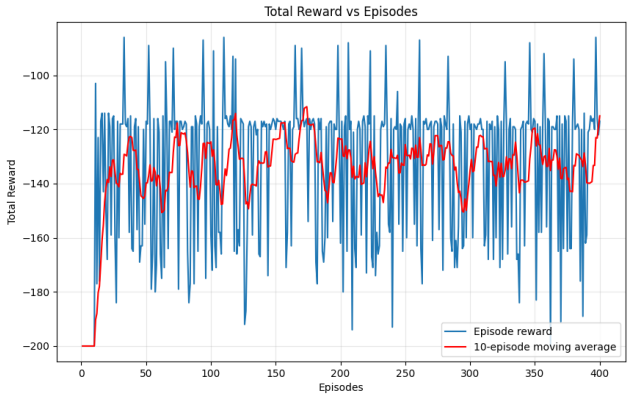

Figure 8: Episodic total reward over episodes during training.

2.3.2 3D Value Function of the Optimal Policy

The 3D plot of the value function is shown in Figure 9. The plot highlights how the agent assigns high values to states with high speeds (both positive and negative) and at the extremes of the state space, enabling it to maintain momentum.

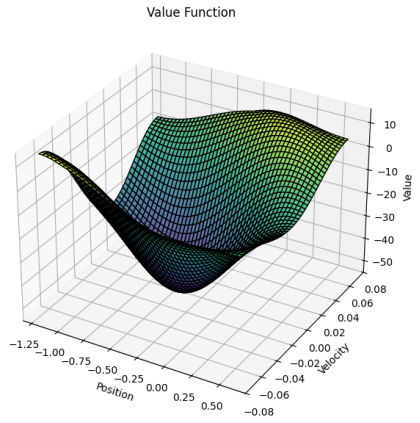

Figure 9: Value function of the optimal policy over the state space.

2.3.3 3D Optimal Policy

The optimal policy is illustrated in Figure 10. The policy primarily emphasizes maintaining the car's momentum, rarely opting for a stop action.

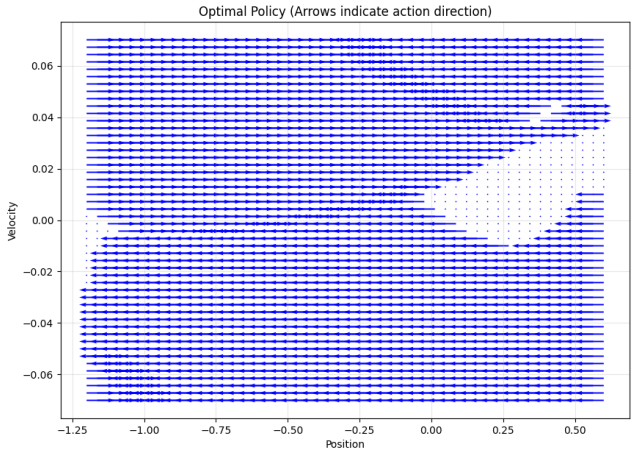

Figure 10: Optimal policy over the state space.

2.3.4 Including/Excluding $\eta = [0, 0]$

Including $\eta = [0, 0]$ in the Fourier basis resulted in smoother improvements during early training stages. Variance during training was smaller with $\eta = [0, 0]$. However, the final convergence level was not significantly affected when comparing the two scenarios.

2.4 Analysis of Training Process

2.4.1 Effect of Learning Rate α

The effect of α on the average total reward is shown in Figure 11. Only small values of α performed well, while large values showed that the agent was not able to improve from a score of -200.

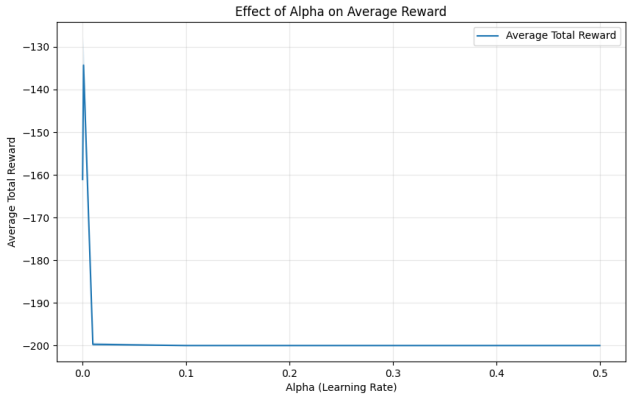

Figure 11: Average total reward as a function of learning rate α.

2.4.2 Effect of Eligibility Trace Parameter λ

The optimal λ was around 0.5, as shown in Figure 12. Lower values maintained a consistent level and small variance, while higher values increased variance in the training process and decreased performance.

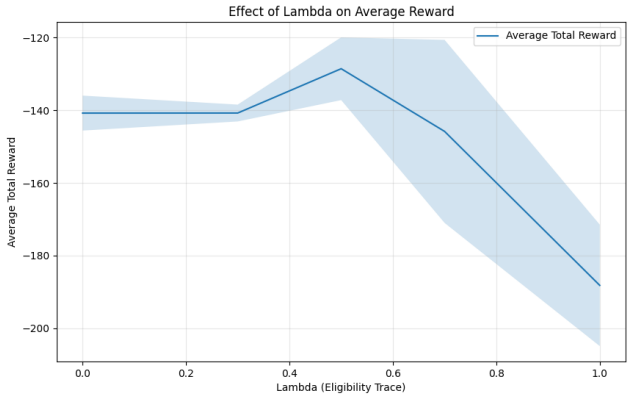

Figure 12: Average total reward as a function of eligibility trace parameter λ.

2.4.3 Q-value Initialization Strategies

Three Q-value initialization strategies were tested:

- **Zero Initialization:** Fastest to start improving.
- **Optimistic Initialization:** Produced the least variance upon convergence.

- **Random Initialization:** Results were inconsistent and varied across runs.

Overall, no significant differences were observed in the final performance.

2.4.4 Q-value Initialization Strategies

Three Q-value initialization strategies were tested to evaluate their impact on training dynamics:

- **Zero Initialization:** All Q-values were initialized to zero, which assumes no prior knowledge of the environment. This strategy showed the fastest improvement in performance during the early stages of training, as the agent quickly updates the Q-values based on observed rewards.
- **Optimistic Initialization:** Q-values were initialized to a high constant value (e.g., 10). This strategy produced the least variance upon convergence, leading to more stable results across runs.
- **Random Initialization:** Q-values were initialized randomly within a small range ($[-0.01, 0.01]$), representing uncertainty in the agent's knowledge of the environment. This strategy resulted in inconsistent performance across runs, likely due to the variability introduced by the random initialization.

While each strategy influenced the training dynamics differently, no significant differences were observed in the final performance achieved by the agent. On the other hand, Zero Initialization was the most consistent.

2.4.5 Exploration Strategies

Two exploration strategies were analyzed during training: **Epsilon-Greedy** and **Boltzmann**. Figures 13 and 14 illustrate the episodic total rewards as a function of training episodes for each strategy.

- **Epsilon-Greedy:** Actions are selected randomly with probability ϵ or greedily with probability $1 - \epsilon$.
- **Boltzmann:** Actions are chosen based on the softmax of their Q-values:

$$P(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)},$$

where τ is the temperature parameter.

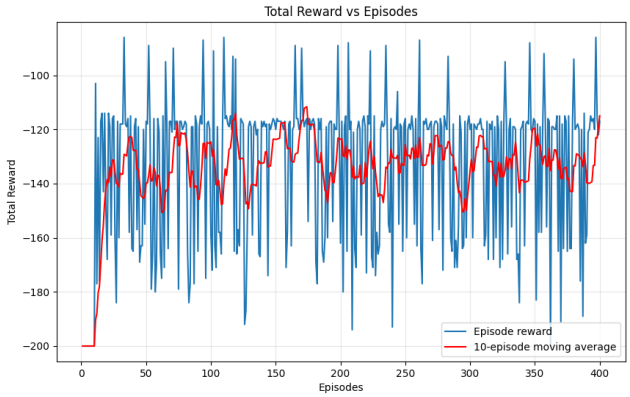

Figure 13: Episodic rewards for Epsilon-Greedy exploration.

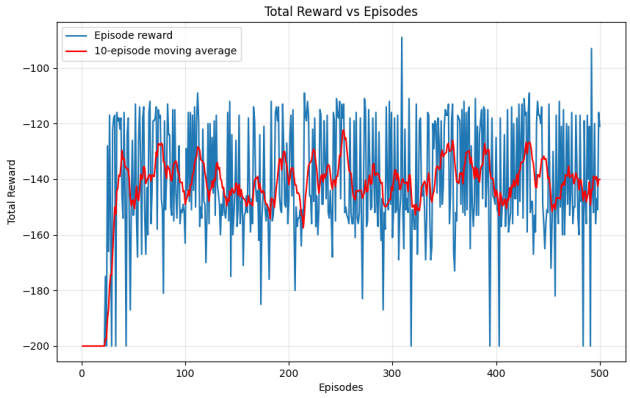

Figure 14: Episodic rewards for Boltzmann exploration.

The performance of the two strategies can be compared as follows:

- **Convergence Speed:** As shown in Figure 13, Epsilon-Greedy converges faster, making it more effective during the early stages of training.
- **Final Performance:** Both strategies achieve similar final performance after sufficient training episodes, as indicated in Figure 14.
- **Exploration Behavior:** The slower convergence of Boltzmann exploration is likely due to the dependence on tuning the temperature parameter τ, which requires careful adjustment to balance exploration and exploitation effectively.

Overall, Epsilon-Greedy demonstrated a slight advantage in terms of speed and simplicity, while Boltzmann exploration achieved comparable results with appropriate parameter tuning.

2.4.6 Exploration Strategies

Two exploration strategies were analyzed during training: **Epsilon-Greedy** and **Boltzmann**. Their performance is illustrated in Figures 13 and 14, which show the episodic total reward as a function of training episodes for each strategy.

2.5 Conclusion

The MountainCar environment was solved using the Sarsa(λ) algorithm with Fourier basis feature extraction. The training process, hyperparameter effects, and exploration strategies were systematically analyzed. The results demonstrate the effectiveness of linear function approximators in solving continuous-state reinforcement learning problems.