EL2805 Reinforcement Learning
Report of Laboratory 1

Franco Ruggeri
Division of Decision and Control Systems
KTH Royal University of Technology
Stockholm, Sweden
fruggeri@kth.se

Andres Felipe Cardenas Meza
KTH Royal University of Technology
Stockholm, Sweden
afcm2@kth.se

Abstract

In this laboratory, we study Markov Decision Processs (MDPs) and some algorithms to solve them. Firstly, we formulate a Maze problem, with stochastic transitions due to an adversary moving randomly, as an MDP and solve it with several variations using Dynamic Programming, Value Iteration, Q-learning and SARSA. Secondly, we show that SARSA algorithm with function approximation can be applied to the Mountain Car problem, a popular Reinforcement Learning benchmark available from OpenAI Gym. Using Stochastic Gradient Descent with Nesterov Acceleration we successfully train an agent able to solve the problem.

1 Problem 1

In this section, we formulate the Minotaur's maze problem as a Markov Decision Process (MDP) and we 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 Problem 1a - Basic maze

For defining a Markov Decision Process (MDP) it is sufficient to provide state space, action space, reward function and transition probabilities.

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

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

1.1.1 State Space

For defining the state space we will consider the maze with dimension (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 $\mathcal{S}_t$.

$$\mathcal{S} = \mathcal{S}_{nt} \cup \mathcal{S}_t \tag{2}$$

The non-terminal states are represented by using tuples containing the position of the player and the position of the Minotaur on the map. We should also consider that the player cannot step on the wall

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

cells contained in $\mathcal{W}$. Thus, the invalid situations where the player is inside a wall are not included in the state space.

$$S_{nt} = \{(x_p, y_p, x_m, y_m) | 1 \leq x_p \leq M_x, 1 \leq y_p < M_y, 1 \leq x_m < M_x, 1 \leq y_m < M_y, (x_p, y_p) \notin \mathcal{W}\} \tag{3}$$

The set of terminal states includes two cases: (i) 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., $x_p = x_b \wedge y_p = y_b \wedge (x_m \neq x_b \vee y_m \neq x_b)$), which means the player has exited alive, and (ii) 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.

$$\mathcal{S}_t = \{s_{exit}, s_{caught}\} \tag{4}$$

Notice that this is not the only possible representation. For example, one might consider all the possible positions on the map without defining special terminal states, and then define the transition probabilities and rewards accordingly. However, our definition reduces the cardinality of the state space and makes it easier to define transition probabilities and rewards. Furthermore, even though a unique terminal state would suffice, we define two separate terminal states to simplify implementation and notation in the following sections.

1.1.2 Action Space

The possible actions that the player can take, after observing the state, are moving 1 cell in any direction, not in a diagonal, or staying in the same cell. Therefore, the action set is:

$$A = \{stay, up, down, left, right\} \tag{5}$$

It is necessary to limit the actions available in each state according to the valid possibilities that the player has. 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 $(x_w, y_w) \in \mathcal{W}$, the action set is restricted in such a way that the player cannot choose to go *inside* the wall.
- *Terminal state*: In the terminal states (i.e., the player has already exited or has already been caught by the Minotaur), the action set is restricted such that the player can only *stay* in the same position.

1.2 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 $\mathcal{M}(s_m)$ as the set of possible cells that the Minotaur can reach starting from the Minotaur position s_m[1]. The cardinality $|\mathcal{M}(s_m)|$ depends on the Minotaur position, as it is not possible to go out of the maze. Additionally, we restrict $\mathcal{M}_m(s_m) = \emptyset$ in any terminal state[2].

For any of the two terminal states $s \in \mathcal{S}_t$, we have:

$$\mathbb{P}(s|s, stay) = 1, \forall s \in \mathcal{S}_t \tag{6}$$

$$r(s, stay) = 0, \forall s \in \mathcal{S}_t \tag{7}$$

Let us define a function f to denote the next deterministic cell that 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.

[1] $\mathcal{M}(s_m)$ does not include staying still in the original formulation of the problem.

[2] Essentially, in a terminal state, the game ends and nothing happens.

$$f(x_p, y_p, a) = \begin{cases} (x_p, y_p) & \text{if } a = \text{stay} \\ (x_p + 1, y_p) & \text{if } a = \text{down} \\ (x_p - 1, y_p) & \text{if } a = \text{up} \\ (x_p, y_p - 1) & \text{if } a = \text{left} \\ (x_p, y_p + 1) & \text{if } a = \text{right} \end{cases} \tag{8}$$

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

$$\mathcal{S}' = \{(f(s_p, a), s_m) | s_m \in \mathcal{M}(s_m)\} \mathbb{P}(s'|s, a) = \frac{1}{|\mathcal{M}(s_m)|}, \forall s' \in \mathcal{S}' \tag{9}$$

For the non-terminal states $s \in \mathcal{S}_{nt}$, 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.

$$r(s, a, s_{exit}) = +1, \forall s \in S_{nt}, \forall a \in \mathcal{A}_s \tag{10}$$

$$r(s, a, s_{caught}) = 0, \forall s \in S_{nt}, \forall a \in \mathcal{A}_s \tag{11}$$

$$|T| < \infty \implies r(s, a, s') = -0.0001, \forall s \in S_{nt}, \forall a \in \mathcal{A}_s, \forall s' \in \mathcal{S}_{nt} \tag{12}$$

$$T \to \infty \implies r(s, a, s') = 0, \forall s \in S_{nt}, \forall a \in \mathcal{A}_s, \forall s' \in \mathcal{S}_{nt} \tag{13}$$

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, the *very small step penalty* encourages the player to choose the shortest path among the ones that lead to the exit alive with the same probability, which is the *additional* objective of the problem. The key insight regards the scale of the rewards. Since the main objective is to maximize the probability of exiting the maze alive, it is important that the worst-case cumulative penalty (i.e., when the player does not exit) is much smaller than the exit-alive reward (i.e., $T \cdot step\_penalty \ll exit\_reward$). If this is not true, the player's main objective might become minimizing the average time to exit. In other words, the player might choose unsafe paths in order to reach the exit as soon as possible, at the cost of risking being caught.

In the discounted MDP, the additional objective is automatically achieved with the discount factor, since an exit-alive reward now is greater than an exit-alive reward in the future. Thus, in the discounted MDP we do not need a penalty for the time spent to reach the exit. Finally, we do not give any penalty for being caught. Indeed, avoiding the Minotaur is done naturally in order to collect the exit-alive reward. With a penalty for being caught, we would add another objective to minimize the probability of being caught. Such an objective could potentially conflict with the objective of reaching the exit, since being caught can be avoided also by staying alive and not reaching the exit.

1.3 Problem 1b - Alternating rounds

Our MDP formulation can be modified for alternating rounds by adding a restriction to the actions in the states where the player is adjacent to the Minotaur. In such cases, we can remove the action that makes the player move towards the Minotaur. Indeed, such an action would result in the player being caught. In general, for any maze layout and any starting state, the alternating-rounds version of the problem makes it more likely for the Minotaur to catch the player, as the player cannot move towards the Minotaur when adjacent to it. Imagine, for example, the state $(4, 1, 4, 2)$, with the Minotaur blocking the passage. If player and Minotaur move simultaneosly and the Minotaur cannot stand still, then the player can move safely towards the Minotaur. Instead, if they move in alternating rounds, the player cannot do that.

However, with this particular maze layout and initial state, denoted as s_0, we noticed that the Minotaur has no way to catch the player provided that the player never stays still in the same position. In fact,

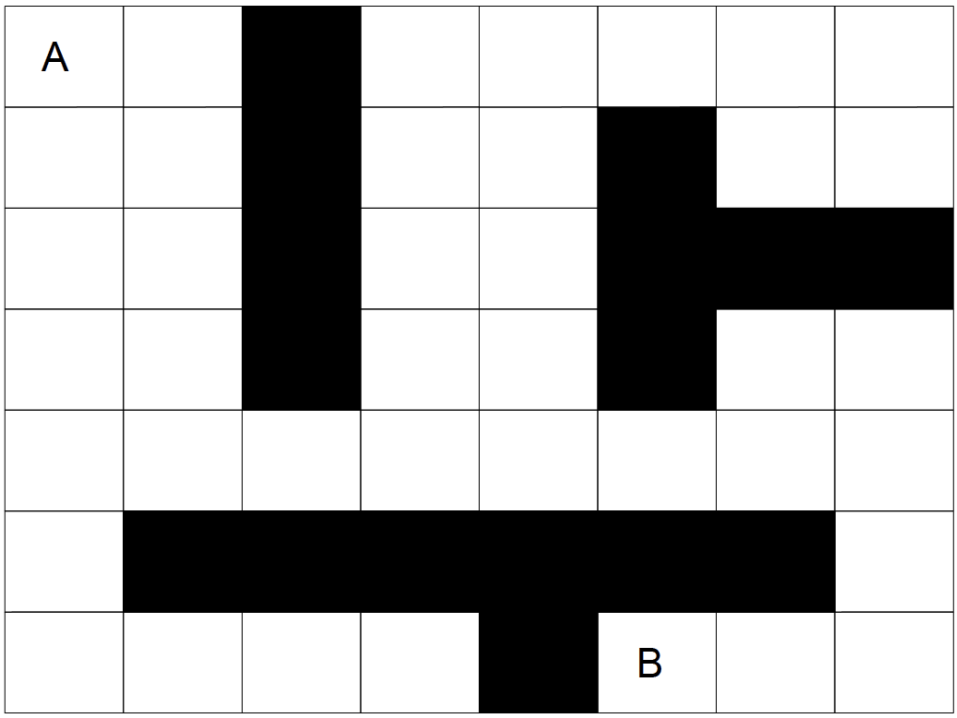


Figure 1: The minotaur's maze.

since player and Minotaur cannot move diagonally, they can reach each cell on the map in a number of moves which is either odd or even and, due to their initial positions, for each cell, if the Minotaur can reach the cell in an odd number of moves, then the player can reach the same cell only in an even number of moves by simply not standing still, and vice versa. For this reason, without the possibility of staying still, the Minotaur cannot break this behavior and cannot catch an optimal player (or even a non-optimal player that keeps moving).

1.4 Problem 1c - Dynamic Programming - Policy

DP can solve finite-horizon MDPs. The goal is to maximize the reward for the player in the maze. We expect the player to avoid being *caught* by the Minotaur in order to collect the reward at the exit. Additionally, we expect that, among paths with the same probability of exiting alive, the player will prefer the shortest one to avoid the step penalty.

For the considerations made in section 1.3, we know that the Minotaur cannot catch a player that keeps moving. Thus, we are sure that an optimal policy will always play by moving along the shortest path to the exit, regardless of the Minotaur's moves. Figure 2 illustrates the simulation of a game and agrees with our intuition. The player goes to the exit along the shortest path and never stays in the same position.

Figure 3 shows the action selected by the optimal policy in each player position with the Minotaur fixed at the exit (i.e., position B in fig. 1), at $t = 1$ (T moves left) and $t = T$ (only one move left). From fig. 3a, we can clearly see that the player follows the shortest path. Another interesting thing to notice is that, in the bottom-right corner, the agent stays still (for that move) in order not to risk to be caught. From fig. 3b, instead, we can notice that the player moves towards the exit only when the exit can be reached in one move, as there is only one move left. Otherwise, the player stays still[3] because there is no time to reach the exit.

[3]This is a consequence of the order of the moves in our implementation, *stay* is the first one. If we change the order, the first move will be picked. It is clear that all the moves have the same Q-value in this case.

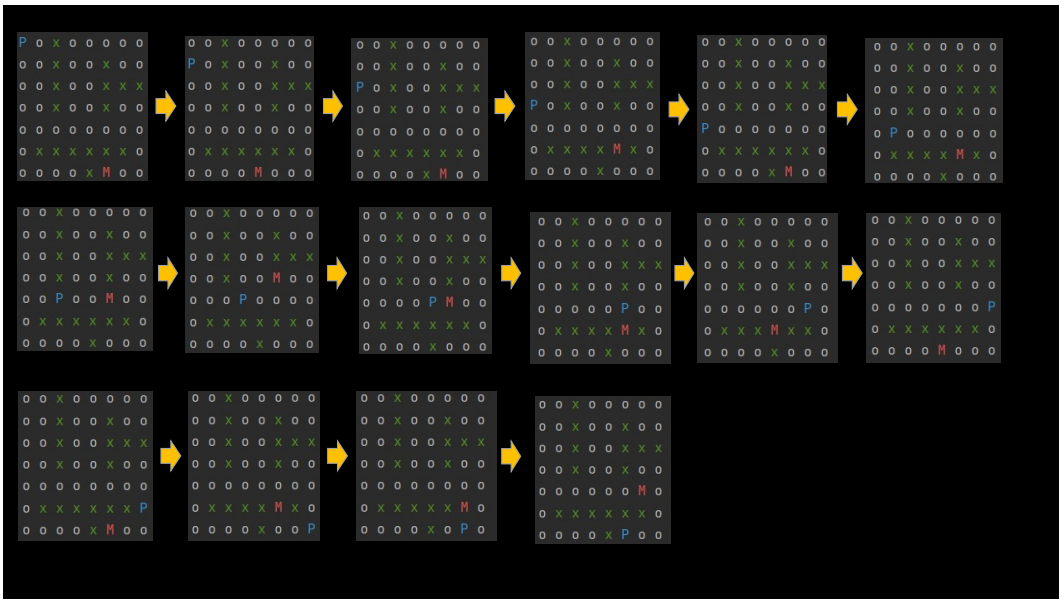
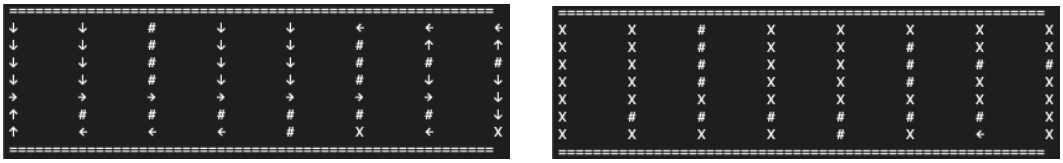


Figure 2: Simulation of a game played by an optimal policy obtained with dynamic programming and time horizon T=20. Notation: P = player, M = minotaur, X = wall, o = free cell.



(a) t=1 (b) t=20

Figure 3: Policy for the minotaur fixed at the exit (i.e., position B in figure fig. 1) at t=1 (20 moves left) and t=20 (1 move left). Notation: move = arrow, stay = X, wall = #.

1.5 Problem 1d - Dynamic Programming - Probability of exiting alive

As mentioned in the previous subsection, DP can solve *finite-horizon* MDPs. Indeed, the time horizon T is a critical parameter because it determines the ability of the player to reach the goal or not depends on it. In particular, with low values of T, the information does not "back-propagate" on time and the algorithm fails to estimate the policy that makes the player reach the goal. From the player's perspective, a too small T means there is no time to reach the exit and collect the reward, thus it does not make sense to go towards the exit.

Figure 4 shows the probability of exiting the maze, whose maximization is the main objective of the problem MDP. If the time horizon is smaller than the shortest path length ($T < 15$), then the player cannot reach the exit in time and the probability of exit is 0. When there is enough time ($T >= 15$), instead, the player can exit the maze alive and the probability depends on whether the Minotaur is allowed to stand still or not. In particular, if the Minotaur cannot stand still, the player can always exit because of the considerations done in section 1.3 (i.e., the Minotaur cannot catch the player with their initial positions) for any $T >= 15$. Otherwise, the Minotaur becomes more dangerous and the player needs more time to reach the exit. For explaining this increased difficulty, imagine in fig. 1 that the Minotaur stands still in $B = (6, 5)$ or in a bottleneck $(4, 2)$ or $(4, 5)$ (see fig. 1). In such cases, the player has no chance at all to exit. However, since the Minotaur moves randomly, the probability of exiting alive increases with the time for $T >= 15$. The intuition is that, with more time, the player can wait in the hope that the Minotaur moves away.

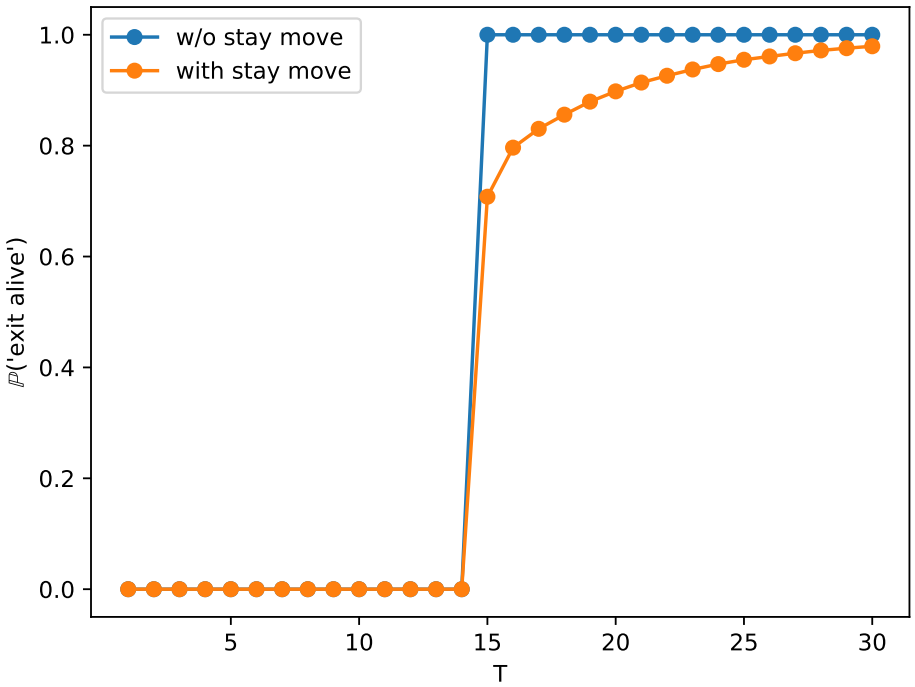


Figure 4: Probability of exiting the maze alive for varying time horizon T, for the cases in which the Minotaur can or cannot stay still. The results have been generated by simulating 10000 episodes for each value of T.

Table 1: Estimated probability of exiting alive from simulations of the game by using a $\epsilon = 0.01$-optimal policy computed with VI.

Number of episodes	$\mathbb{P}$('exit alive')
10000	0.616
100000	0.62016

1.6 Problem 1e - Poisoned Minotaur's maze

The poisoned version of the Minotaur's maze, where the player's life is geometrically distributed, can be seen as a MDP with random time horizon (i.e., the life is the time horizon). It can be proven that solving an infinite-horizon discounted MDP M is equivalent to solving a MDP M' with random time horizon $T \sim Geo(1-\lambda)$ (i.e., geometrically distributed with parameter $1-\lambda$) with $\mathbb{E}[T] = \frac{1}{1-\lambda}$, where λ is the discount factor in M. Thus, we can model the life geometrically distributed with $\mathbb{E}[T] = 30$ as an infinite-horizon discounted MDP with discount factor $\lambda = \frac{29}{30}$.

Infinite-horizon MDPs can be solved, among others, by VI. In the following subsection, we will show the results with VI.

1.7 Problem 1f - Value Iteration - Probability of exiting alive

We tested the ϵ-optimal policy computed with VI. The results are reported in table 1. We can also check that our estimate is correct by using the considerations made in section 1.3. Indeed, since the Minotaur cannot catch the player with this particular initial state (see fig. 1), the player can lose only in case of poison death. So, the probability of exiting alive is the same as the probability of not dying from poison for 14 steps, which is the number of moves necessary to reach the exit, without

considering the starting move. We assume that the condition at the exit does not matter. So, the true probability is:

$$\mathbb{P}('exit\ alive') = \mathbb{P}('no\ poison\ death')^{14} = \lambda^{14} = \left(\frac{29}{30}\right)^{14} = 0.622120345492098 \tag{14}$$

We also checked $V(s_0)$ computed by VI:

$$V(s_0) = 0.62212035 \tag{15}$$

Now, we know from what said in section 1.3 that the optimal policy is to go to the exit in 15 steps, and, according to section 1.2, the only reward present in the maze is the exit-alive reward at the exit, which is 1. Thus, $V(s_0)$ is the reward at the exit discounted by λ^{14}, which is exactly what VI computed.

1.8 Problem 1g - Theoretical questions

1.8.1 On-policy vs off-policy

On-policy learning methods learn the value of the policy that the agent uses to collect the data. The idea is to use this value to improve the policy used to interact with the environment and make it converge to the optimal policy. For example, SARSA is an off-policy method and learns the *Q-function of the policy used by the agent* to collect the observations.

Off-policy learning methods, instead, learn the value of the optimal policy independently of the policy that the agent uses to collect data, which is usually called *behavior policy*. Q-learning is an example of off-policy methods and learns the *Q-function* (of the optimal policy) from observations collected by a behavior policy.

1.8.2 Convergence of Q-learning and SARSA

The convergence conditions of Q-learning are:

- The step sizes α_t are square summable but not summable.
- The behavior policy visits every state-action pair infinitely times.

For example, the first condition can be guaranteed by using $\alpha(s, a) = 1/n(s, a)$, while the second one by using an ϵ-greedy policy.

The convergence conditions of SARSA are:

- The step sizes α_t are square summable but not summable.
- The policy is an ϵ-greedy policy.
- $\epsilon \to 0$ as $t \to \infty$.

For example, the last condition can be guaranteed by using $\epsilon = 1/k$, where k is the episode number.

Notice how, in both methods, *exploration* is crucial. The policy used to collect data should explore all state-action pairs.

1.9 Problem 1h - Minotaur's maze with keys

The introduction of keys and the new behavior of the Minotaur require some modifications to state space, rewards, and transition probabilities. The new MDP is defined in the rest of this section.

It is important to keep in mind that we want to model the problem as a *discounted* MDP, since we want a random time horizon geometrically distributed like the life of the player. As a consequence, the MDP needs to be *homogeneous* in order to admit an optimal stationary policy. The discount factor is set to $\lambda = \frac{49}{50}$ for the same considerations made in section 1.6.

1.9.1 State Space

We add an extra bit of information containing whether the player has or not the keys. Denoted S_{nk}[4] the state space without the keys defined in section 1.1, we can define the new state space after the addition of the keys as:

$$S_k = \{(x_p, y_p, x_m, y_m, k) | (x_p, y_p, x_m, y_m) \in S_{nk}, k \in \{0, 1\}\} \tag{16}$$

where $k = 1$ means that the player has already collected the key, while $k = 0$ means that the player has not. This definition of state space allows us defining a homogeneous MDP.

1.9.2 Transition probabilities and Rewards

We can define the transition probabilities by looking at the possible moves for the Minotaur. For making the definition easier, we first define:

$$\Delta_x(s) = x_p - x_m \tag{17}$$
$$\Delta_y(s) = y_p - y_m \tag{18}$$

The Minotaur approaches the player following the direction having minimum Δ, except for the following cases: (i) if $\Delta_x = \Delta_y$, the Minotaur moves with uniform probability in the two directions towards the player; (ii) if $\Delta_x = 0$ (i.e., the Minotaur is in the same row as the player), the Minotaur moves along the y direction; and (iii) if $\Delta_y = 0$ (i.e., the Minotaur is in the same column as the player), the Minotaur moves along the x direction. With probability 0.35 at each time step, the possible moves for the Minotaur are those we just defined. With probability 0.65, instead, the possible moves for the Minotaur are all the valid ones defined for the original problem. Another modification is that the transition to the terminal state s_{exit} can happen only when the player has the keys. If the player does not have the keys, being in (x_b, y_b) does not mean exit.

Regarding the rewards, we add only a positive reward for grabbing the keys (i.e., in the transition from a state without keys to a state with keys). We set this reward to $+1$. Even though not strictly necessary, such a reward helps the Reinforcement Learning (RL) algorithms to learn faster.

1.10 Problem 1i - Q-learning

The pseudo code for Q-learning (and SARSA, discussed in the next section) is shown in algorithm 1. It is worth highlighting that, in order to improve exploration, if several actions have the same maximum Q-value, the behavior policy *EpsilonGreedyPolicy* selects randomly among them. We also want to remark that, in our experiments, the behavior policy samples from an infinite-horizon MDP, i.e., the player cannot die from poison. Such a choice guarantees to reach a terminal state at each episode and improves the convergence. On the other hand, the RL policy still optimizes the poisoned version of the game because of the discount factor (see section 1.9.

Our first experiment regards the initialization of Q-values. We set $\alpha = \frac{2}{3}$ and $\epsilon = 0.2$ and tested different ways of initializing the Q-values of the non-terminal states. For the terminal states, instead, the Q-values were always initialized to 0, which is their true value by definition (the game terminates and no reward is received in the terminal states). The results are shown in fig. 5a. We found that the initialization to a value slightly greater than 0 (0.01) helps the convergence speed. The reason is that, with such an initialization, a *die move* (i.e., leading to die caught by the Minotaur) "pulls down" the corresponding Q-value more than another step move in the same state, thus the behavior policy (ϵ-greedy) will not pick the *die move* again the next time, unless for exploring. Furthermore, the initialization to 1 speeds up significantly the convergence speed and is the only one converging within 50000 episodes. Such a result is consistent with the theorem of convergence of the Robbins-Monro algorithm. Here, we report the result from the lecture slides:

$$e_{min}^{(k)} \leq \frac{||x^{(0)} - x^*||_2^2 + G^2 \sum_{i=0}^{k} \alpha_i^2}{2\beta \sum_{i=0}^{k} \alpha_i} \tag{19}$$

[4] nk stands for "no keys".

Algorithm 1: General training loop for Q-Learning or SARSA

Input: Environment env
Output: Reward

for $1 \leq episode \leq N$ **do**
  s ← Initial state
  a ← EpsilonGreedyPolicy(s)
  while *state is not terminal* **do**
    newState, reward ← ApplyAction(action)
    newAction ← EpsilonGreedyPolicy(s')

    s ← state
    s' ← newState
    a ← action
    a' ← nextAction
    n(s,a) ← n(s,a)+1 `// number of visits`

    if *is Q-learning* **then**
      $Q(s,a) \leftarrow Q(s,a) + \frac{1}{n(s,a)^\alpha} \left[r + \lambda \cdot \max_b Q(s',b) - Q(s,a)\right]$
    else if *is SARSA* **then**
      $Q(s,a) \leftarrow Q(s,a) + \frac{1}{n(s,a)^\alpha} \left[r + \lambda \cdot Q(s',a') - Q(s,a)\right]$
    end

    state ← nextState
    action ← nextAction
  end
end

where the upper bound of the minimum error up to episode k depends also on the distance between the initial estimate $x^{(0)}$ and the solution x^*. In our case, the initialization to 1 is closer to the solution than the other tested initialization values. In the other experiments, we decided to use the 0.01 initialization because it still manages to converge with a higher learning rate (see next) without requiring a guess of the true value.

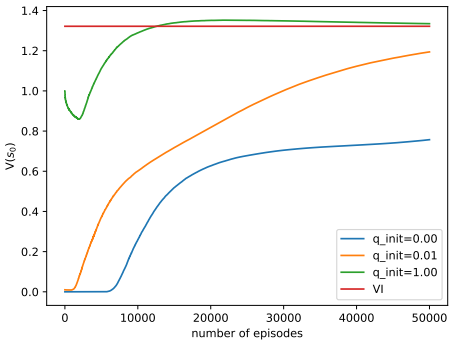
In our second experiment, we tested several values of ϵ by fixing $\alpha = 2/3$ and initialization to 0.01. The results, presented in fig. 5b, indicate that both too small ($\epsilon = 0.01$) and too large ($\epsilon = 0.5$) values slow down the convergence. The explanation for too small values is straightforward, as small values of ϵ explore the state-action pairs slowly. For too large values, instead, our intuition is that it leads to a too random exploration that is worse than a smarter exploration in promising areas of the state space reached through exploitation (as an analogy, we can think about breadth-first search and best-first search). We found the best value to be $\epsilon = 0.2$, and we used it for the next experiment.

Our last experiment was about α. We set $\epsilon = 0.2$ and initialization to 0.01. As shown in fig. 5c, the smaller value of α, the faster the convergence. For $\alpha = 0.55$, Q-learning achieves an accurate estimate within 50000 episodes. These results are intuitive, as smaller values of α imply higher step sizes.

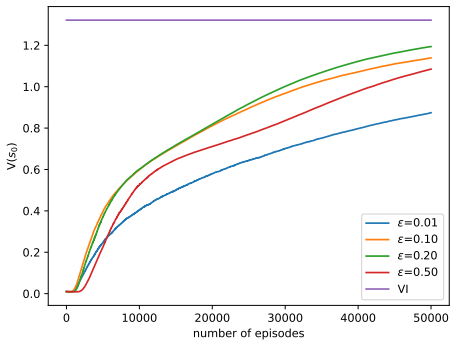
1.11 Problem 1j - SARSA

As shown in algorithm 1, the implementation of SARSA differs from Q-learning for the Q-function update. Q-learning estimates the Q-function (i.e., under the optimal policy) and so it contains the max operator (from Bellman's equations). SARSA, instead, uses also the next action to estimate the Q-function under the current policy (and then improve the policy by using it, on-policy algorithm), thus there is no max operator.

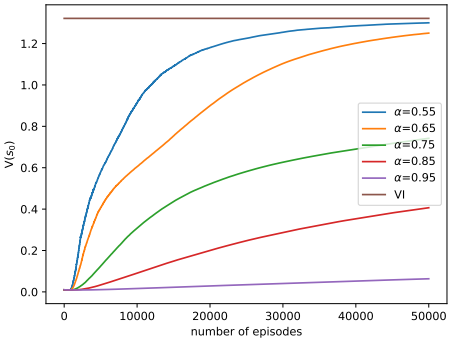
Firstly, as we did for Q-learning, we fixed $\epsilon = 0.1$ and $\alpha = 2/3$ and tried several initialization of Q-values for the non-terminal states. The reasoning is the same as the one for Q-learning (see



(a) Varying initial Q-values



(b) Varying ϵ values



(c) Varying α values

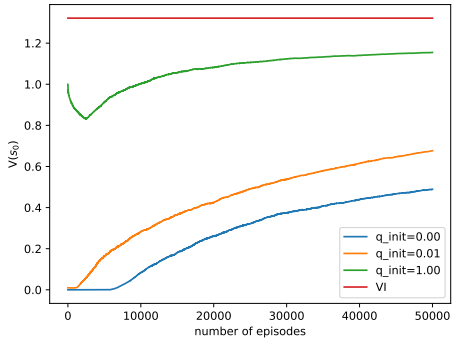
Figure 5: Learning curves of Q-learning showing $V(s_0)$ (value of the initial state) over the episodes for: (a) $\epsilon = 0.2$, $\alpha = 2/3$, and several initialization of Q-values for non-terminal states; (b) $\alpha = 2/3$, initialization to 0.01, and varying ϵ; (c) $\epsilon = 0.2$, initialization to 0.01 and varying α. The baseline true value is given by a 0.01-optimal value computed using VI.

section 1.10), and the conclusion from fig. 6a is again that the best initialization is 1, as it is the closest to the true value.

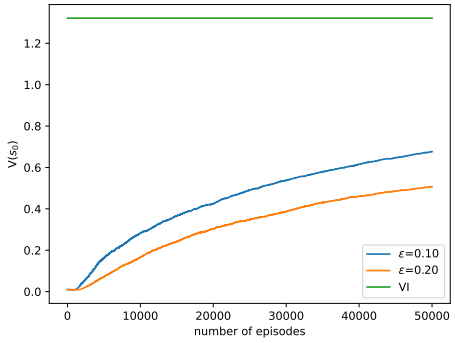
Secondly, we experimented with ϵ (fig. 6b) and we found that $\epsilon = 1$ provides a higher Q-value. This result can be explained by the fact that SARSA learns the Q-function under the policy used to collect data, which is ϵ-greedy. Thus, the higher the ϵ, the more exploration (potentially non-optimal actions), the smaller value under the ϵ-greedy policy.

Thirdly, we experimented with δ and α. The results are shown in figs. 6c and 6d for initialization of Q-values 0.01 and 1, respectively. We decided to experiment also for the better initial because, for initialization to 0.01, the convergence was not satisfying[5]. Regardless, for both initialization values, we can observe that the exploration decay is beneficial (compare with fig. 6b). Intuitively, the reason is that SARSA can have the benefit of exploring (a lot at the beginning) but still end up with an estimate of a Q-function under a policy with little exploration. More formally, we are satisfying the condition $\epsilon \to 0$ for $k \to \infty$ mentioned in section 1.8. Furthermore, the results indicate that it is better to have $\delta > \alpha$ (e.g., from fig. 6d, with $\delta = 0.75$, $\alpha = 0.65$ is better than $\alpha = 0.85$). Our intuition is that, with $\delta > \alpha$, when the exploration has decayed enough and SARSA is estimating the

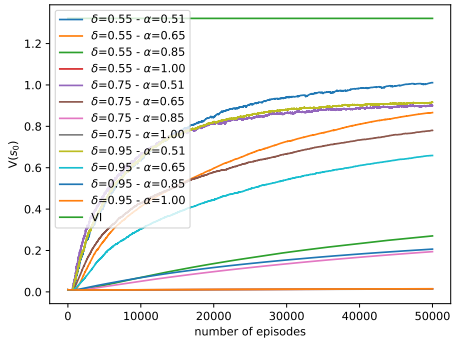
[5]The results reported here are the ones that perform best in terms of probability of exiting alive (problem 1k). We played extensively with SARSA to make it converge more closely to the true value. We managed to do it by modifying the reward (-1 when caught, $+0.1$ for exit, $+0.1$ for key, -0.01 for step). However, even though SARSA converged perfectly with such a reward ($\delta = 0.95$, $\alpha = 0.95$), the performance in terms of probability was worse, as the objective encoded by that reward was not exactly *maximizing the probability of exiting alive*. Thus, we decided not to report it.



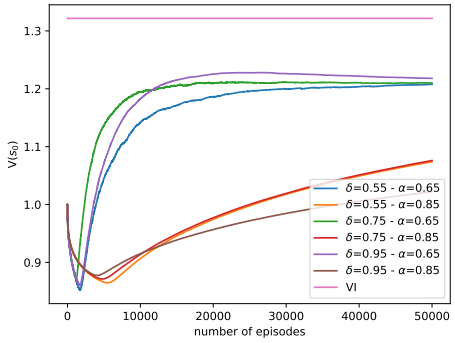
(a) Different initial Q-values



(b) Varying ϵ values



(c) Varying δ and α, initial Q-values 0.01



(d) Varying δ and α, initial Q-values 1

Figure 6: Learning curves of SARSA showing $V(s_0)$ (value of the initial state) over the episodes for: (a) $\epsilon = 0.1$, $\alpha = 2/3$, and several initialization of Q-values for non-terminal states; (b) $\alpha = 2/3$, initialization to 0.01, and varying ϵ; (c) initialization to 0.01 and varying δ and α; (d) initialization to 0.01 and varying δ and α. The baseline true value is given by a 0.01-optimal value computed using VI.

Table 2: Hyperparameters to train final policies with Q-learning and SARSA.

Algorithm	α	ϵ	δ	Initialization
Q-learning	0.55	0.2	-	0.01
SARSA	0.65	-	0.95	1

Q-function under a policy with little exploration, the learning rate is still high enough to impact the Q-values.

1.12 Problem 1k

The hyperparameters used to compute the final policies are reported in table 2. The corresponding performance, in terms of probability of exiting alive, are shown in fig. 7. We can observe that both algorithms achieve a nearly-optimal performance but Q-learning outperforms SARSA in our settings.

Since our reward function does not contain only one reward at the exit, the probabilities are not close to the value of the initial state. If we gave only a $+1$ reward at the exit, then the answer would be yes. Assume there is only a $+1$ reward at the exit. For each episode k, we might define a Bernoulli random variable:

$$Y_k = \begin{cases} 1 & \text{if player exited alive} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

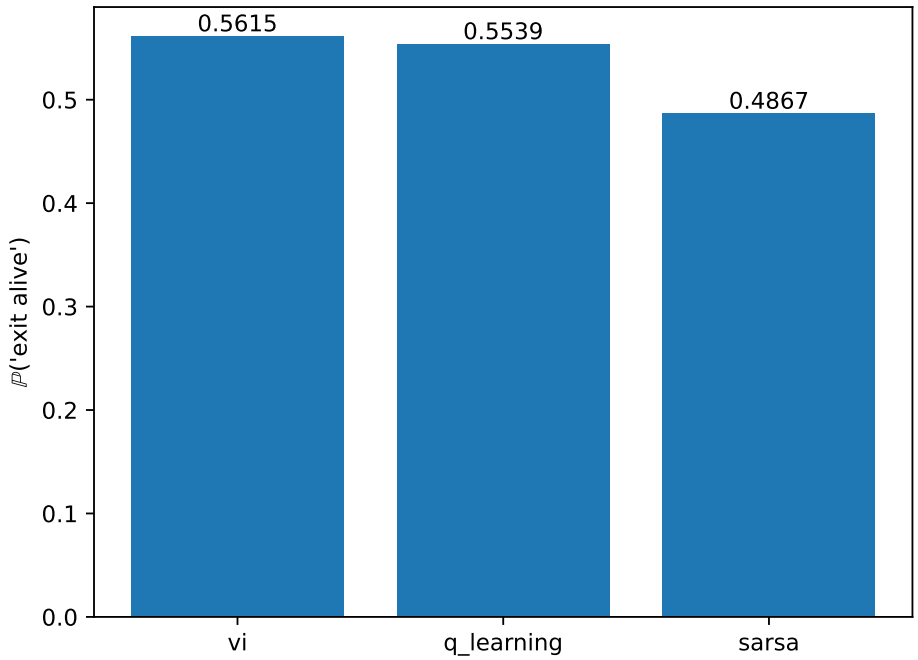


Figure 7: Probability of exiting the maze alive for policies computed with VI (baseline), Q-learning, and SARSA. The hyperparameters are in table 2.

Y_k clearly corresponds to the episode reward in episode k, and the random variables of different episodes are i.i.d., as episodes are independent trials. Then, the expected value $\mathbb{E}[Y_k]$ is the expected episode reward, which by definition is the value function of the initial state $V(s_0)$ (since the initial state is the same in all the episodes):

$$V(s_0) = \mathbb{E}[Y_k] \tag{21}$$

Furthermore, the probability of exiting alive in an episode k can be written as:

$$\mathbb{P}('exit\ alive') = \mathbb{P}(Y_k = 1) = \mathbb{E}[Y_k] \tag{22}$$

where the last relationship comes from the fact that Y_k has a Bernoulli distribution (eq. (20)). Thus, we have found that the probability of exiting alive would be equal to the value function of the initial state (eqs. (21) and (22)). Furthermore, for the law of large numbers, we know that:

$$\lim_{k \to +\infty} \frac{1}{k} \sum_{i=0}^{k} Y_i = p \tag{23}$$

which means that we can estimate the probability of exiting alive by simulating a large number of episodes.

2 Problem 2

In this section, we will solve the MountainCar problem from OpenAI gym, which presents an environment with a continuous state space. The state has two dimensions representing the position of the cart over the x-axis and its velocity. In this version of the problem the agent only has three possible actions:

- *Right*: Accelerate towards the right
- *Left*: Accelerate towards the left
- *Nothing*: Do not accelerate

Parameter	Values
η	η_1, η_2, η_3, η_4
λ	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
γ	0.95, 1
m	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Table 3: Hyperparameters for the grid search performed.

2.1 Technical Approach

For solving the problem we used SARSA(λ). Specifically, the Q-function is approximated as a linear combination of features under a Fourier basis, using also the eligibility traces. The eligibility traces help learning sparse rewards, as they allow having a backward view in order to update the weights proportionally to their contributions to recently visited states. We optimized the weights of the linear approximator using Stochastic Gradient Descent (SGD) with Nesterov acceleration.

2.1.1 Fourier Basis and Linear Approximator

For estimating the Q-function, we used a linear approximator with the Fourier basis. The key equations are the following:

$$Q_w(s,a) = \mathbf{w}_a^T \cdot \phi(\mathbf{s}) \tag{24}$$
$$\phi(\mathbf{s})_i = cos(\pi \eta_\mathbf{i} \cdot \mathbf{s}) \tag{25}$$

In the implementation of the above, it is efficient to consider it as vector and matrices operations rather than single-element operations. We can define the Fourier basis matrix as $\eta \in \{0, 1, .., p\}^{(n_b \times s_d)}$, where n_b denotes the number of elements in the basis and s_d denotes the state dimensionality. We can also introduce the matrix $W \in \mathbf{R}^{(n_a \times n_b)}$, where n_a denotes the number of actions, that represents a linear transformation from the basis space to the action space. The operation from the state space to the action space can be written as:

$$\mathbf{Q} = W \cdot cos(\pi \eta \cdot s) = W\phi(s) \tag{26}$$

where the cosine is applied element-wise. Given a state vector $s \in \mathcal{R}^{(s_d \times 1)}$, the vector $\mathbf{Q} \in \mathcal{R}^{(n_a \times 1)}$ will contain the approximated $Q(s,a)$ values. As we will see, using those values we can make the agent take decisions.

2.1.2 SGD and Eligibility Traces

For updating the weights of the agent, we used SGD with Nesterov acceleration and eligibility trace. The learning rate is scaled with respect to the norm of the basis. The rule for updating the eligibility trace is described in equation 27.

$$\mathbf{z}_a = \begin{cases} \gamma\lambda\mathbf{z}_a + \nabla_{w_a} Q_w(s_t, a) & \text{if } a = a_t \\ \gamma\lambda\mathbf{z}_a & \text{if otherwise} \end{cases} \tag{27}$$

We have that $Q = W \cdot \phi(\mathbf{s})$, therefore $\nabla_{w_a} Q_w(s_t, a) = \phi(\mathbf{s})$. Given a set of $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, obtained using an ϵ-greedy policy, the rule for updating the weights is described in the equation 28.

$$\begin{aligned} \delta_t &= r_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t) \\ \mathbf{v} &\leftarrow m\mathbf{v} + \alpha\delta_t\mathbf{z} \\ \mathbf{w} &\leftarrow \mathbf{w} + m\mathbf{v} + \alpha\delta_t\mathbf{z} \end{aligned} \tag{28}$$

2.2 Training and Evaluation

The model and training process shown before have several hyperparameters to be chosen.

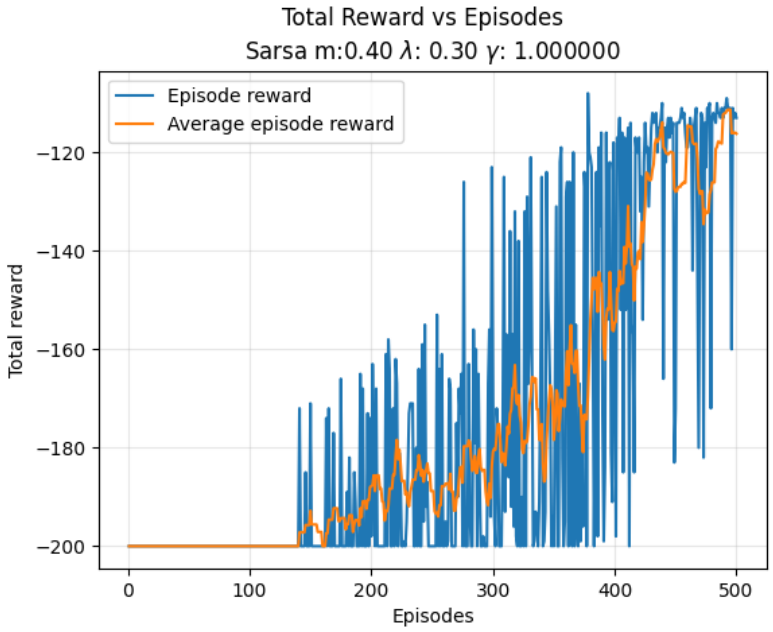


Figure 8: The episode reward and the moving average over 10 episodes during the training.

- η: The matrix of the Fourier basis.
- λ: The trace-decay parameter.
- γ: The discount factor.
- m: The momentum to be used with the SGD.

Since we did not have a good guess of such hyperparameters, we decided to perform a **grid search** for determining the best set of hyperparameters. In our grid search, we considered the hyperparameters reported in table 3, where the matrices η are the following:

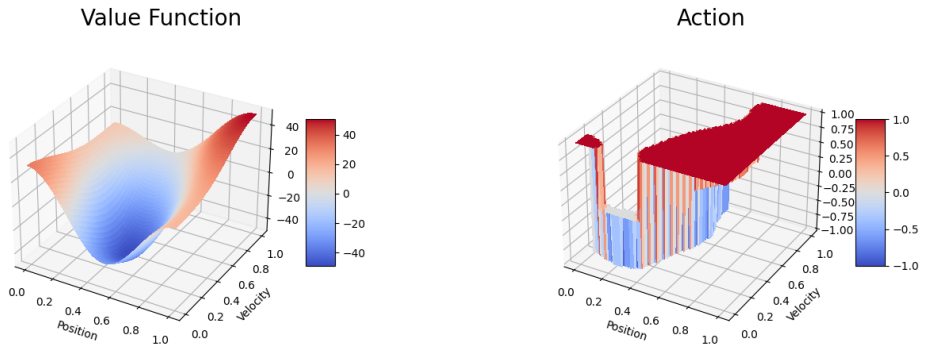
$$\eta_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{pmatrix}, \quad \eta_2 = \begin{pmatrix} 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{pmatrix}, \quad \eta_3 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad \eta_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{29}$$

It is worth highlighting that these matrices include the complete second order basis and the uncoupled basis (i.e., considering the features independent, which we know is not the case for this environment).

For training a given configuration we trained the agent on the environment for $N = 500$ episodes using an ϵ-*greedy* policy. For all models, we used a linear ϵ decay of $\frac{1}{N}$ and a unique learning rate decrease of $lr = lr * 0.7$ triggered when the episode reward was over a threshold $lr_{trigger} = -135$, that is the score considered as *passed*. The initial ϵ considered at the beginning of each episode was 0.8 and it linearly decayed towards 0 in the last episode for full-filling SARSA converging criteria.

For evaluating the resulting configuration, we tested the model on the same environment for $N_t = 50$ episodes and evaluated the average reward obtained by the policy as well as the confidence interval at 95% confidence. A policy is considered to pass if its complete confidence interval is over -135, which means $avg_{reward} - confidence > -135$. Among passing policies, we chose the one that consistently obtained higher results, thus the one obtaining higher $avg_{reward} - confidence$.

Our evaluation showed that the best hyperparameter configuration is $\eta = \eta_2$, $m = 0.4$, $\lambda = 0.3$, and $\gamma = 1$. As expected, the basis corresponding to feature uncoupling performed worse than the



(a) Value function

(b) Policy

Figure 9: Representation of (a) value function and (b) policy for the best agent. The x-y plane represents the state space, while the z-axis represents the value function or the action selected in each state. The action is encoded as: left = -1, nothing = 0, right = +1.

complete basis, as the features are highly coupled (velocity is the derivative of position). The learning curve for the best agent is shown in fig. 8. The estimated average reward of this agent by running 200 episodes is -111.94 ± 029. In the following, we describe its value function and policy.

The value function of the best agent, shown in fig. 9a, is intuitively correct. The value is high in situations where the agent can reach the goal in few moves. For example, when the car's position is on top of the left hill (position close to 0), there is enough potential energy to reach soon the goal by just accelerating to the right, so the value is high. Even higher is the value for positions close to the goal (position close to 1). On the other hand, when the car is in the middle, a high number of moves are needed to gain momentum and reach the goal, thus the value is low.

The policy of the best agent is represented in fig. 9b. We can observe some intuitive behavior confirming that the learned policy makes sense. For example, the closer the car is from the goal, the smaller (positive) velocity is required to decide to keep accelerating right (it can still reach the goal!). Equivalently, if the car is far from the goal and there is not enough positive velocity, the agent decides to accelerate to the left to go to the left hill and gain more momentum.

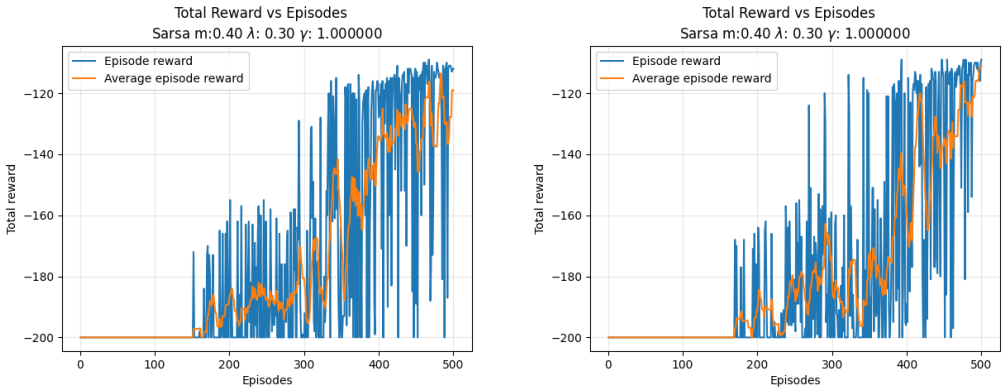
2.2.1 Effect of Hyperparameters

Among the hyperparameters values contained in our grid search, we can see that it contained both the η_1 and η_2, one of them having the null element in the basis while the other one not. At a theoretical level, we do not expect to see any change in their performances. We trained the model with the same hyperparameters listed before using both matrices. The results for η_1 and η_2 are shown , respectively, in figures 10a and 10b. As we can see from the figure, as well as from the result, is that there does not seem to be any significant difference between both basis.

2.2.2 Exploration Strategies

The learning ability of the agent highly depends on the parameters lr, λ, and γ. Fixing all other parameters to match the best policy found we modified these parameters one by one to evaluate their effect on the resulting average reward of the model and we re-trained the model using them. The effect of the learning rate and momentum can be seen in figure 12a and 12b.

We also tested different exploration strategies. We tried slight modifications of a regular ϵ-greedy policy as described in table 4. The comparison of the average reward obtained during training is shown in figure 13. As we can see from the results the best exploration strategy , in this case , is the standard ϵ-greedy However, the strategy that takes the *worst* action performs good ans allows the network to explore states. Also the "hard-coded" solution of exploring the domain according to the position performs well. Finally, the exploration strategy that with probability ϵ does not accelerae does not perform very good.



(a) η_1 (b) η_2

Figure 10: Training of model with best hyper-parameters using different basis

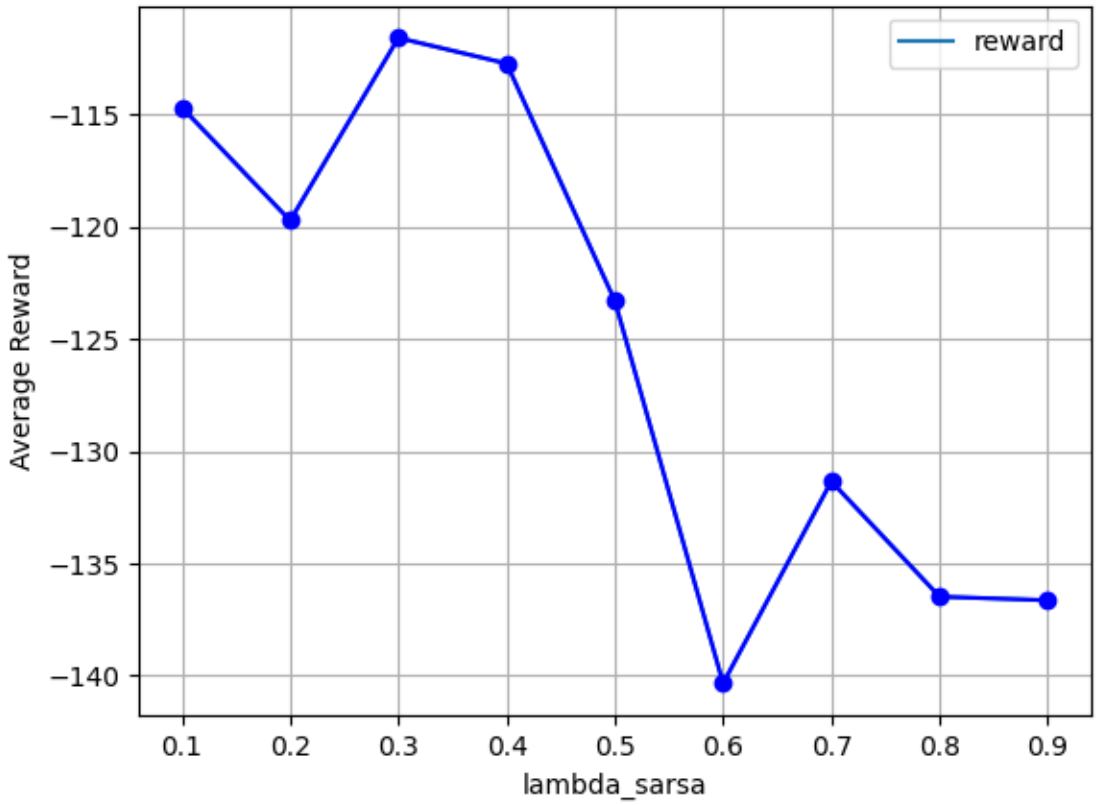
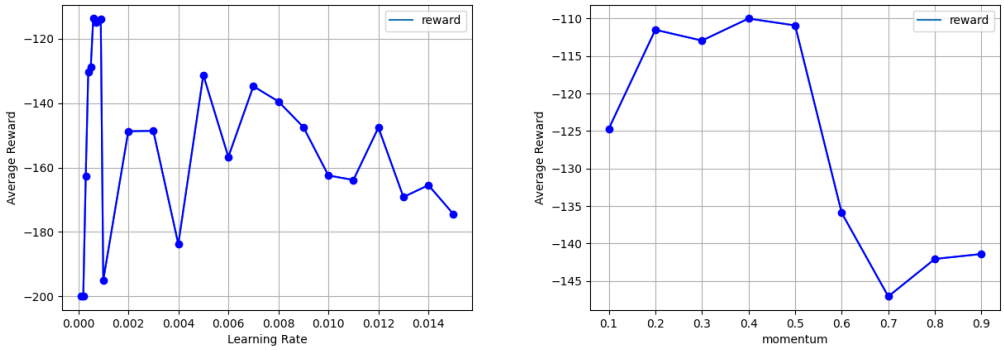


Figure 11: Average reward over 200 simulations using the policy obtained with the given eligibility trace λ. Other parameters were set to the best configuration.



(a) Learning rate effect

(b) Momentum effect

Figure 12: Average reward obtained after simulation of 200 episodes. Other parameters kept constant at best hyperparameter configuration found.

Policy	Description	Result
Standard	Regular ϵ-greedy policy	-115.12 ± 1.79
Opposite	Take the action with lowest Q value with probability ϵ	-126.26 ± 4.06
Still	Return "still" action with probability ϵ	-196.61 ± 0.38
State	Return right if relative position over half of the world. Return left otherwise.	-118.26 ± 1.7

Table 4: The proposed exploration strategies with their relative results at 95 % confidence trained over 500 episodes and tested over 200 episodes.

2.2.3 Weight Initialization

Finally, the effect of the initialization of the Q-function depends on the initialization of the weights. We tested two different types of initialization for the weights: random numbers uniformly distributed in $[0, 1]$ and random numbers normally distributed according to $\mathcal{N}(0, 1)$. As we found from experiments the results are equivalent and we could not find any significant difference. The training results are shown in Figure 14b and Figure 14a.

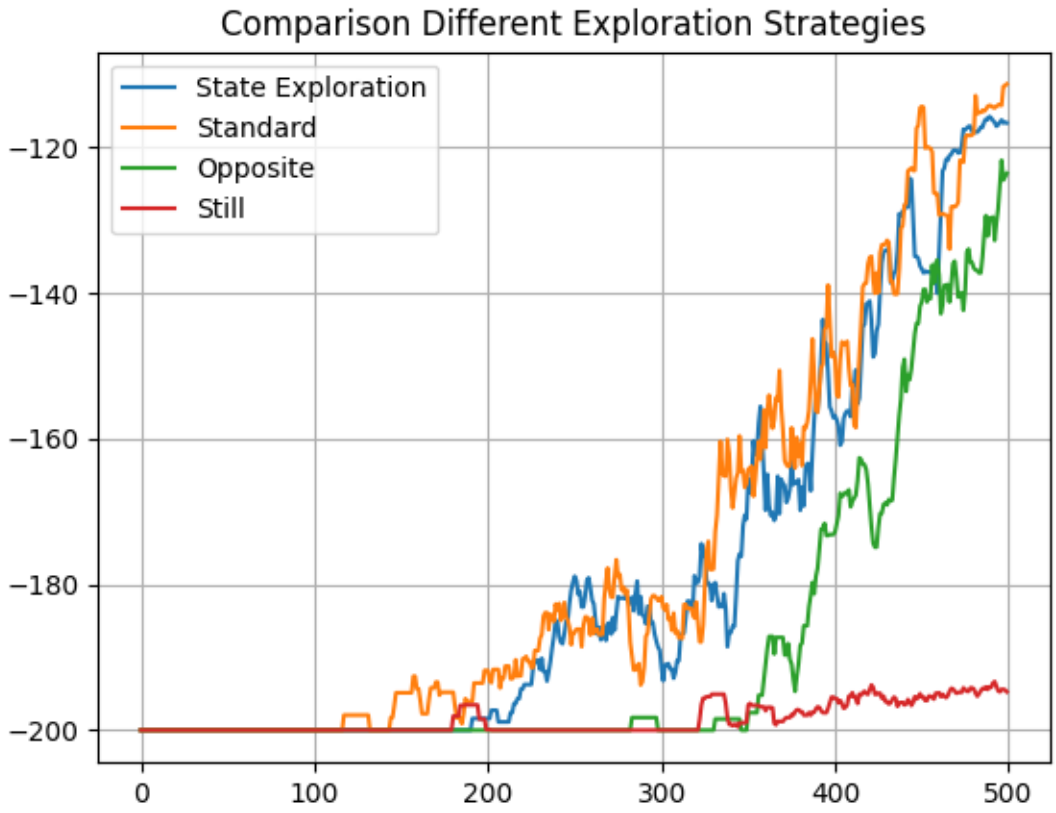
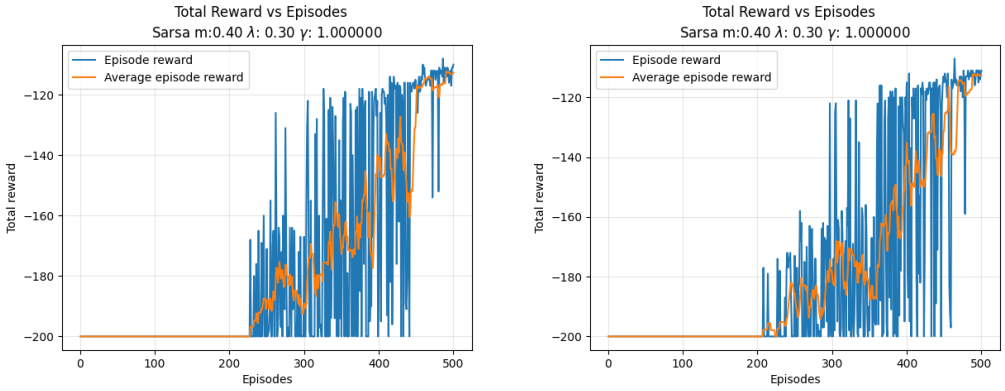


Figure 13: Comparison of the running average over 15 episodes of the obtained reward by different exploration strategies



(a) Uniformly distributed

(b) Normally distributed

Figure 14: Training of model with best hyper-parameters using different type of initialization