EL2805 Reinforcement Learning
Computer Lab 1

Abdessamad Badaoui & Nasr Allah Aghelias
20011228-T118 & 20010616-T318

1 The Maze and the Random Minotaur

1.1 Basic Maze

1.1.1

We model the state space $\mathcal{S}$ as a tuple of all possible positions of the player (x_p, y_p) (excluding walls), and all possible positions of the Minotaur (x_m, y_m) :

$$\mathcal{S} = \{((x_p, y_p), (x_m, y_m)) : \text{such that } (x_p, y_p) \text{ is not an obstacle and } 0 \leq x_p, x_m \leq 7 \text{ and } 0 \leq y_p, y_m \leq 6\} \tag{1}$$

We let the player choose any action from the action space $\mathcal{A}$ defined by :

$$\mathcal{A} = \{\text{up, down, right, left, stay}\} \tag{2}$$

The objective of the player is to identify a strategy maximizing the probability of exiting the maze (reaching B) before time T :

$$r(s, a) = p_M * R_{mp} + (1 - p_M) * R_{step} \tag{3}$$

where p_M is the probability of the Minotaur being at the same position as the new deterministic state of the player after taking action a.
$R_{mp} = -\infty$: represents the reward of being eaten by the Minotaur.
R_{step} : equals to -1 when taking action a, leads the player to some position in the maze other than the exit, the walls or the Minotaur's position, 0 if it leads to the exit, and $-\infty$ if it leads to a wall.

The player transitions are deterministic, whereas the Minotaur's transitions are random, and we can express that formally as follows :

$$\mathbb{P}((p_t, m_t)|(p_{t-1}, m_{t-1}, a)) = \frac{1}{N} \tag{4}$$

where N equals to the number of Minotaur's possible actions ($2 \leq N \leq 4$) and p_t, m_t are the possible positions of the player and the Minotaur respectively after taking the action a from the state (p_{t-1}, m_{t-1}), which means that the probability of the other states is 0.

1.1.2

To model this case where the player and the Minotaur do not move simultaneously, we need just to modify the reward function as follows:

- If at state s, taking action a, leads the player to a wall or to the Minotaur's position then $r(s,a) = -\infty$
- If at state s, taking action a, leads the player to some position in the maze other than the exit, the walls or the Minotaur's position then $r(s,a) = -1$
- if at state s, taking action a, leads the player to the exit then $r(s,a) = 0$

1.2 Dynamic Programming

1.2.1

For a time horizon $T = 20$, we compute the policy that maximizes the probability of leaving the maze alive using dynamic programming. Then we simulate the Minotaur's behavior randomly and we compute the player's next move according to the best policy. The following figures show the player and Minotaur's movement in the maze.

1.2.2

We plotted the probability of the player exiting the maze alive for all time horizons $T \in \{1, \dots, 30\}$, while taking into account that the Minotaur can or can not stand still during rounds. The result can be observed in the following figure.

Allowing the Minotaur to remain stationary introduces an extra potential location for the Minotaur at each time step. This additional option adds complexity to the player's decision-making process, potentially reducing the likelihood of successfully navigating the maze and exiting alive.

1.3 Value Iteration

1.3.1

Now, in this scenario, we are dealing with a random time horizon following a geometric distribution. Consequently, we discount the importance of future rewards using the parameter λ. This parameter can be determined using the following relationship, which links the expectation of the time horizon with λ:

$$\mathbb{E}(T) = \frac{1}{1-\lambda} = 30 \iff \lambda = \frac{1-30}{30} = \frac{29}{30} \tag{5}$$

Here we solve the discounted infinite-horizon MDP problem using value iteration, the objective here is to find a stationary policy π that minimizes the infinite horizon objective with a discount factor λ

$$\mathbb{E}\Big[\sum_{t=0}^{\infty} \lambda^t r\big(s_t, \pi(s_t)\big)\Big].$$

Bellman equations in the case of a stationary policy π :

$$\forall s \in \mathcal{S} \qquad V^*(s) = \max_{\pi} \Big\{ r(s,\pi(s)) + \lambda \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s,\pi(s)) V^*(s') \Big\} \tag{6}$$

or equivalently in terms of the Bellman operator $\mathcal{L}$

$$V^* = \mathcal{L}(V^*) \tag{7}$$

where

$$\forall s \in \mathcal{S} \qquad \mathcal{L}(V)(s) = \max_{\pi} \Big\{ r(s,\pi(s)) + \lambda \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s,\pi(s)) V(s') \Big\}. \tag{8}$$

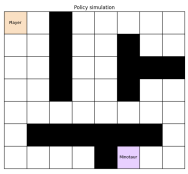

Figure 1: State at instant 0

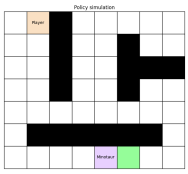

Figure 2: State at instant 1

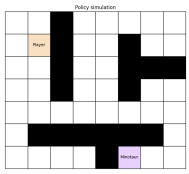

Figure 3: State at instant 2

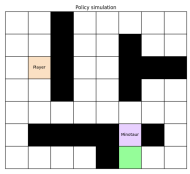

Figure 4: State at instant 3

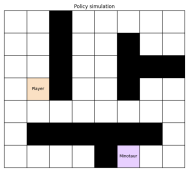

Figure 5: State at instant 4

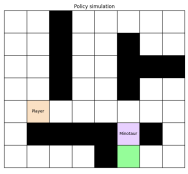

Figure 6: State at instant 5

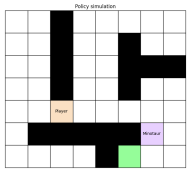

Figure 7: State at instant 6

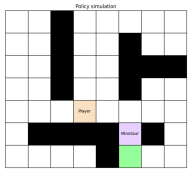

Figure 8: State at instant 7

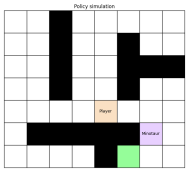

Figure 9: State at instant 8

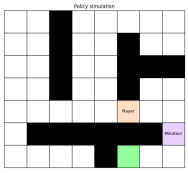

Figure 10: State at instant 9

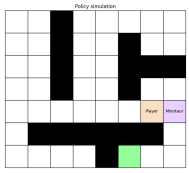

Figure 11: State at instant 10

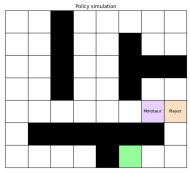

Figure 12: State at instant 11

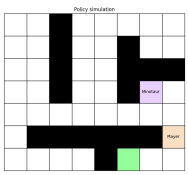

Figure 13: State at instant 12

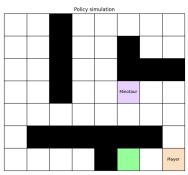

Figure 14: State at instant 13

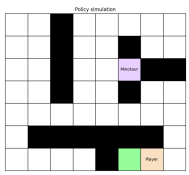

Figure 15: State at instant 14

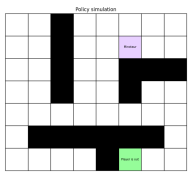

Figure 16: State at instant 15

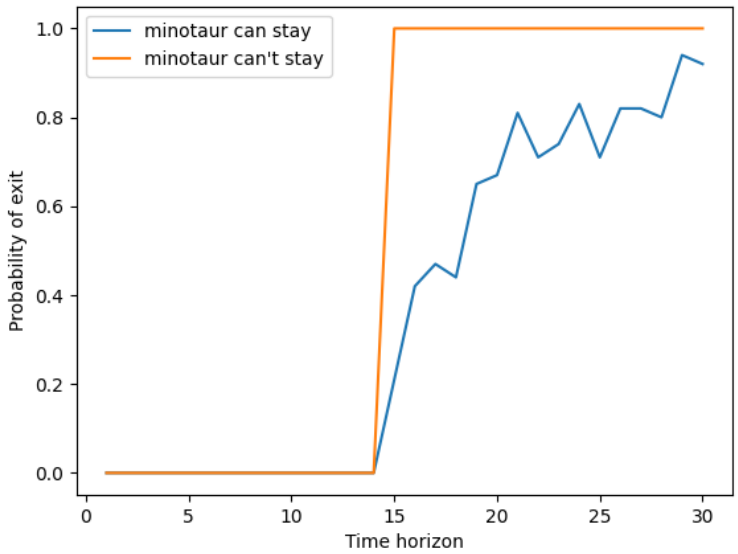

Figure 17: The probability of successful exit with respect to time horizon

1.3.2

Based on the graphics, we observe that the player needs at least T=15 to reach the exit. To calculate the probability of the player successfully reaching the exit alive, we just need to calculate the probability of T being greater than 15. This gives us:

$$\mathbb{P}(T \geq 15) = 1 - p(T \leq 14) = 1 - F_T(15) = 1 - 1 + \left(\frac{29}{30}\right)^{14} = 0.622 \tag{9}$$

By simulating 10,000 games, we obtained an estimate of the probability of successfully exiting using the optimal policy: 0.619. This value is close to the theoretical result.

1.4 Additional questions

1.4.1

In off-policy learning, the algorithm learns the value of the optimal policy independently of the actions taken by the agent. The policy employed by the agent is often referred to as the behavior policy, denoted as π_b. On the contrary, in on-policy learning, the algorithm learns the value of the policy being executed by the agent. This policy is determined from previously collected data, making it an active learning method where the gathered data is controlled and used for further learning.

1.4.2

For this new problem, the state space and action space remain the same as the previous MDP. We modify the reward function as follows :

$$r(s,a) = p_M * R_{mp} + (1 - p_M) * R'_{step} \tag{10}$$

where R'_{step} equals to -1 when taking action a, leads the player to some position in the maze other than the exit, the walls, the Minotaur's position or the first visit of C. 50 if it leads to the first visit of C, 0 if it leads to the exit and $-\infty$ if it leads to a wall.

$R_{mp} = -100$: represents the reward of being eaten by the Minotaur.
p_M is the probability of the Minotaur being at the same position as the new deterministic state of the player after taking action a.

The Minotaur's transitions are no longer 100% random. There is 35% chance that the Minotaur will move towards the player and 65% chance he will move randomly in all possible directions. Let us consider d_{Man} the Manhattan distance between two positions. We can express the transition probabilities as follows :

$$\mathbb{P}((p_t, m_t)|(p_{t-1}, m_{t-1}, a)) = \begin{cases} 0.35 \cdot \frac{1}{M} + 0.65 \cdot \frac{1}{N} & \text{if } d_{Man}(m_t, p_{t-1}) < d_{Man}(m_{t-1}, p_{t-1}) \\ 0.65 \cdot \frac{1}{N} & \text{if } d_{Man}(m_t, p_{t-1}) \geq d_{Man}(m_{t-1}, p_{t-1}) \end{cases} \tag{11}$$

where N equals to the number of Minotaur's possible actions ($2 \leq N \leq 4$) and p_t, m_t are the possible positions of the player and the Minotaur respectively after taking the action a from the state (p_{t-1}, m_{t-1}).
M is equal to the number of the Minotaur's possible moves that minimize the Manhattan distance between him and the player (i.e. the moves that take him to the player) ($1 \leq M \leq 2$).
M is equal to 1 when the player and the Minotaur are in the same column or row at instant $t-1$.
M is equal to 2 when the player and the Minotaur are in a different column and a different row at instant $t-1$

1.5 Q-Learning and Sarsa [Bonus questions]

(i)

1.5.1

The pseudo-code is given in Algorithm 1 :

1.5.2

We plot figures (See Figures 20) of the value function for the initial states using different initial values for the Q-function. It is observed that our Q-function converges to -50. Therefore, starting with an initial value of the Q-function that is close to -50 accelerates the convergence process.

1.5.3

When the exploration parameter is held constant, the algorithm converges more rapidly as the value of alpha approaches 0.5, in contrast to a slower convergence observed as alpha approaches 1 as shown in figure 21

Algorithm 1 Q-learning

procedure Q_LEARNING(env, number_episodes, initial_state, epsilon, gamma, alpha)
    $N \leftarrow$ number_episodes
    $n\_states \leftarrow$ env.n_states
    $n\_actions \leftarrow$ env.n_actions
    $Q \leftarrow$ initialize_Q$(n_states, n_actions)$ ▷ Initialize Q-values
    $visits\_state\_action \leftarrow$ zeros$(n_states, n_actions)$ ▷ Initialize visit counts
    $V \leftarrow []$
    for $k \leftarrow 1$ **to** N **do**
        $state \leftarrow$ initial_state
        $cond \leftarrow$ False
        while not cond **do**
            $action \leftarrow$ epsilon_greedy$(Q, \epsilon, state)$
            $visits\_state\_action[state, action] \leftarrow$ visits_state_action[state, action] $+ 1$
            $next\_state, \_ \leftarrow$ env.move$(state, action)$
            $reward \leftarrow$ env.reward$(state, action, next_state)$
            $Q[state, action] \leftarrow Q[state, action] + \frac{1}{\text{visits\_state\_action[state, action]}^{\alpha}} \cdot (reward + \gamma \cdot \max(Q[next\_state]) - Q[state, action])$
            $state \leftarrow$ next_state
            if (env.maze[env.states[next_state][0]] $==$ 2 **and** env.states[next_state][2]) **or** env.states[next_state][1] == env.states[next_state][0] **then**
                $cond \leftarrow$ True
            end if
        end while
        V.append$(\max(Q[\text{initial_state}]))$
    end for
    $policy \leftarrow$ argmax$(Q, 1)$
    return $V, policy$
end procedure

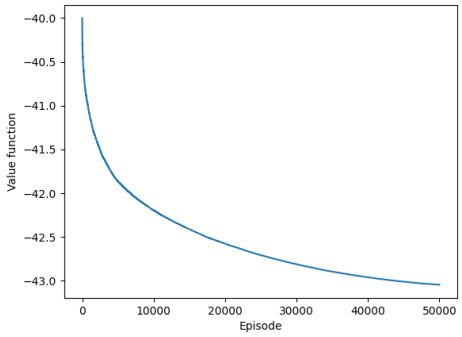

Figure 18: Q-values initialized to -40

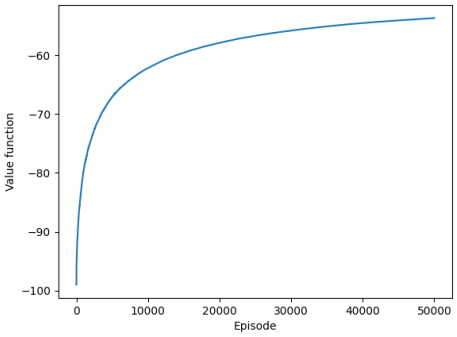

Figure 19: Q-values initialized to -100

Figure 20: Value Function Of The Initial State

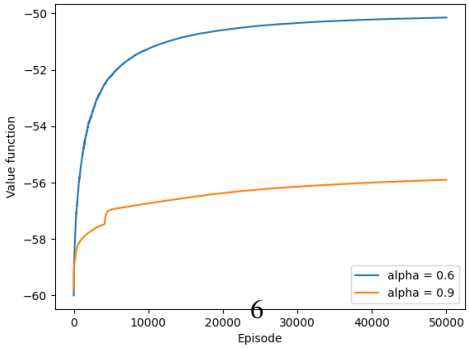

Figure 21: Value Function with Step Sizes $1/n(s,a)^{\alpha}$, $\alpha = 0.6$ and $\alpha = 0.9$

(j)

1.5.4

The pseudo-code of the SARSA algorithm is given in Algorithm 2. The main difference between Q-learning and SARSA is that SARSA uses an additional action. This is necessary to obtain an estimate of the state-action value function. Thanks to the policy improvement theorem, SARSA can approximate the optimal policy when epsilon is very small (epsilon decreasing over time towards 0). On the other hand, Q-learning attempts to learn the Q function directly, and ultimately converges to the true optimal Q function.

Algorithm 2 SARSA

function SARSA(env, number_episodes, initial_state, $\epsilon, \gamma, \alpha, \delta$)
    $N \leftarrow$ number_episodes
    $n\_states \leftarrow$ env.n_states
    $n\_actions \leftarrow$ env.n_actions
    $Q \leftarrow Q_{init}$
    $visits\_state\_action \leftarrow$ np.zeros($(n\_states, n\_actions)$)
    $V \leftarrow []$
    for $k \leftarrow 0$ **to** N **do**
        $state \leftarrow$ initial_state
        cond $\leftarrow$ False
        $\epsilon_k \leftarrow \frac{1}{(k+1)^\delta}$
        while not cond **do**
            $action \leftarrow$ epsilon_greedy$(Q, \epsilon_k, state)$
            $visits\_state\_action[state, action] \leftarrow visits\_state\_action[state, action] + 1$
            $next\_state, \_ \leftarrow$ env.move$(state, action)$
            $next\_action \leftarrow$ epsilon_greedy$(Q, \epsilon_k, next_state)$
            $reward \leftarrow$ env.reward$(state, action, next_state)$
            $Q[state, action] \leftarrow Q[state, action] + \frac{1}{(visits\_state\_action[state,action])^\alpha} \times (reward + \gamma \times Q[next\_state, next\_action] - Q[state, action])$
            $state \leftarrow next\_state$
            if (env.maze[env.states[next_state][0]] $==$ 2 **and** env.states[next_state][2]) **or** env.states[next_state][1] == env.states[next_state][0] **then**
                cond $\leftarrow$ True
            end if
        end while
        V.append(np.max(Q[initial_state]))
    end for
    $policy \leftarrow$ np.argmax$(Q, 1)$
    return V, policy
end function

1.5.5

For $\varepsilon = 0.1$ and $\varepsilon = 0.2$ and a step size of $1/n(s, a)$, with $\alpha = 2/3$. We obtain the figures (22) and (23)

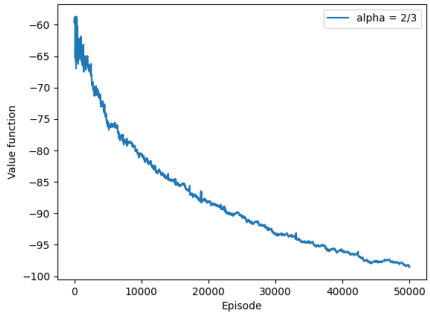

Figure 22: Value function of the SARSA algorithm for $\varepsilon = 0.1$ wto. episodes

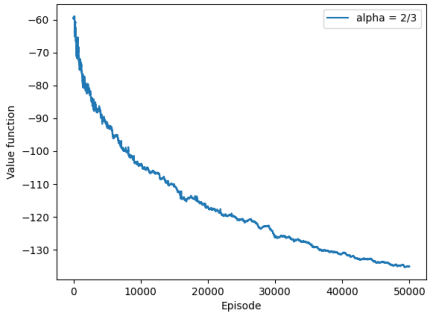

Figure 23: Value function of the SARSA algorithm for $\varepsilon = 0.2$ wto. episodes

SARSA will converge faster using an epsilon equal to 0.2 for exploration compared to epsilon equal to 0.1. This can be explained by the fact that setting ε to a relatively high value leads to discover the optimal policy faster in that case.

Proper initialization of Q-values in SARSA can contribute to faster convergence. Initializing Q-values appropriately sets the initial estimates for the expected cumulative rewards, influencing the exploration-exploitation trade-off. Careful initialization can help balance exploration and exploitation, potentially leading to quicker convergence to optimal policies. Initializing with small random values or prior knowledge can be effective, but it's crucial to consider the specific characteristics of the environment and the learning task.

1.6

For $\varepsilon = 0.1$ and $\varepsilon = 0.2$ and a step size of $1/n(s, a)$, with $\alpha = 2/3$. We obtain the figures (22) and (23)

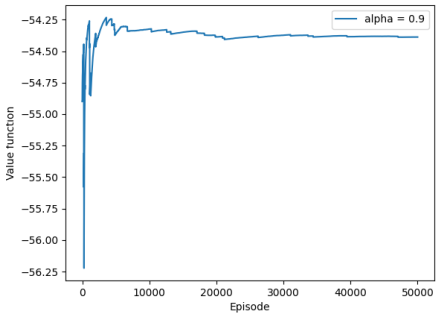

Figure 24: Value function of the SARSA algorithm for $\alpha = 0.9$ and $\delta = 0.6$ wto. episodes

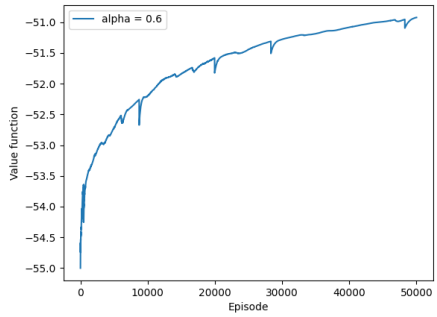

Figure 25: Value function of the SARSA algorithm for $\alpha = 0.6$ and $\delta = 0.8$ wto. episodes

A larger α implies a smaller step size and consequently a slower update.
A larger δ implies a smaller ε_k and consequently a smaller exploration.
$\alpha > \delta$ means we're going for more exploration and a slower update whereas $\alpha < \delta$

means we're going for less exploration and a faster update.
In our particular case, we witnessed a better performance when $\alpha < \delta$ i.e. when we explored less and updated faster.

2 RL with linear approximators : MountainCar

2.1 Question c :

We have trained our model for 200 episodes using the following hyperparameter values :

- $\lambda = 0.5$
- $\alpha = 0.01$
- $\epsilon = 0.1$
- $m = 0.01$

SARSA(λ) with Fourier basis and SGD with momentum equal to m for the MountainCar problem involves approximating the state-action value function using Fourier basis functions (We have used $\eta_i = [k, l] \quad \forall k, l \in \{0, ..., 2\}$, except $[0, 0]$). The algorithm initializes weights of the linear function approximator and eligibility traces. It iteratively selects actions (ϵ-greedy selection with respect to the current estimation of Q_w), updates eligibility traces, and employs SGD with momentum to adjust weights based on temporal difference errors. The exploration strategy guides action selection, and the algorithm repeats this process for a number of episodes equal to 200. The combination of function approximation, eligibility traces, and momentum-based optimization allows SARSA() to efficiently learn and update the state-action value function, balancing exploration and exploitation to solve the MountainCar problem. Fine-tuning hyperparameters was crucial for achieving optimal performance.

2.2 Question d :

2.2.1

Here is the figure 26 showing the convergence of our algorithm. We observe an initial phase during which the average total reward is -200, indicating scenarios where the car fails to reach the flag. Once the car successfully reaches the flag (upon exploration), the actual learning process begins. Subsequently, the learning algorithm converges towards the optimal state-action value function, gradually improving the average performance.

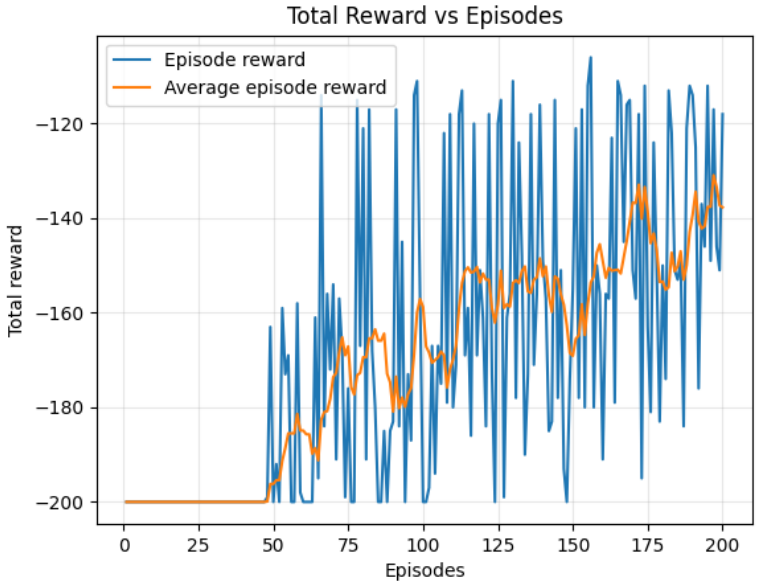

Figure 26: Average episode reward converging to ~ -135

2.2.2

Here is the 3D plot in figure 2.2.2 of the value function of the optimal policy over the state space domain :

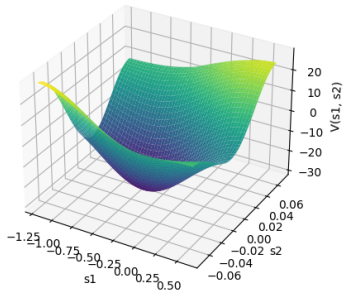

Figure 27: 3D plot of the optimal policy value function

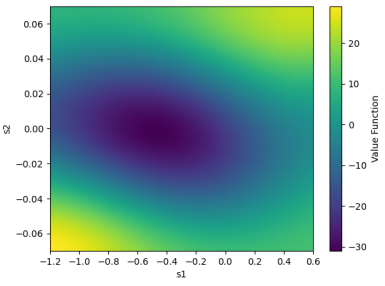

Figure 28: Temperature plot of the value function of the optimal policy

The plots 2.2.2 reveal that the value function, under the optimal policy that was identified, exhibits a global minimum around the state where the position is -0.4, and the velocity is equal to 0. This state corresponds to a location close to the initial state used during training and symbolizes the bottom of the hill. Consequently, it is evident that the value function for this state is very low, indicating that it requires numerous steps to reach the flag. As we increase both the position and the velocity, the value function starts to rise until we reach the position of the flag, where it attains its maximum value.

2.3

Here is the 3D plot in figure 29 and 30 of the optimal policy :

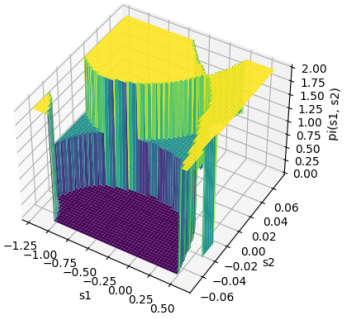

Figure 29: 3D plot of the optimal policy

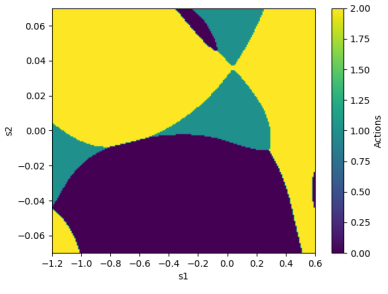

Figure 30: Temperature plot of the optimal policy

In examining these figures, a notable trend is apparent: generally, when the velocity is negative, the predominant action is moving left. Conversely, the usual action is to move right when the velocity is positive (though not always, as indicated in the plots), with occasional instances of choosing not to move.

2.4

Figure 31 shows the evolution of both the episode reward and average episode reward when adding $\eta = [0, 0]$ to our Fourier basis.

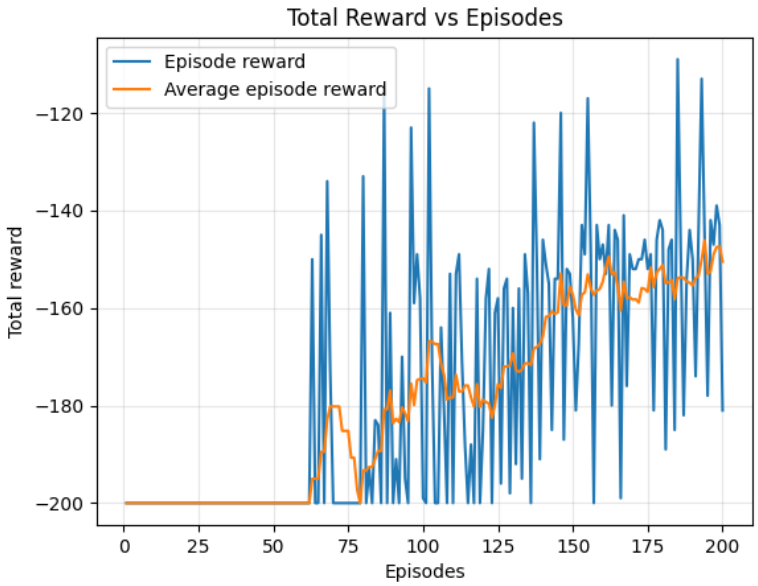

Figure 31: Average episode reward in the case of using $\eta = [0, 0]$

Compared to (26) where $[0, 0]$ is excluded from the basis, the average episodic reward plots are fairly similar.

The following plots 2.4 and **??** show the optimal policy and the value function in that case :

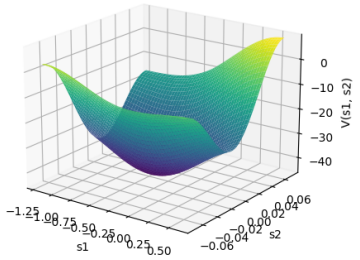

Figure 32: 3D plot of the optimal policy value function

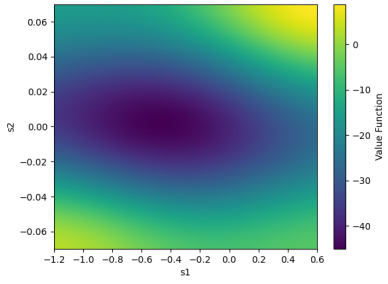

Figure 33: Temperature plot of the value function of the optimal policy

Same as before, the above figures are visually similar to the ones obtained where $[0, 0]$ (27 and 28). The quantitative difference is the order of magnitude of the value function that appears higher in the former case.

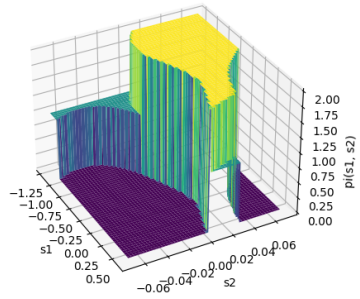

Figure 34: 3D plot of the optimal policy

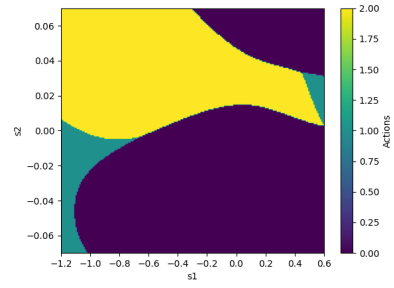

Figure 35: Temperature plot of the optimal policy

In the case of the optimal policy, there is a notable difference between the above figures and (29) & (30), namely that the optimal policy towards when s_1 is closer to 0.6 is to almost certainly *push right* in (30) whereas it is not necessarily the case in (35) depending on the speed s_2. This might be due to the fact that including the constant term in the basis allows the function approximator to capture the mean or offset of the value function. This can be important if your value function has a non-zero mean or if there is a constant component that needs to be represented.

2.5 Question e :

2.5.1

The figure 36 shows the average total reward of the policy as a function of α, whereas figure 37 shows the average total reward of the policy as a function of λ :

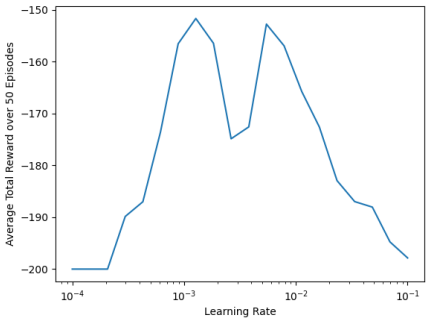

Figure 36: Average reward with respect to α

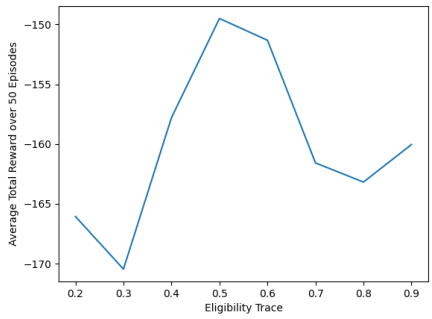

Figure 37: Average reward with respect to λ

Both the figures above have a quasi-bell shape, (37) more than (37).
When λ is close to 0, the algorithm considers only the immediate rewards and neglects the influence of future rewards. This leads to a myopic behavior, where the agent focuses on short-term gains.
As λ increases, the eligibility trace incorporates information about more distant future rewards, allowing the agent to make decisions that consider a longer time horizon.
If the learning rate is excessively low, it may cause the algorithm to converge slowly or become trapped in suboptimal solutions.
Conversely, an excessively high learning rate can induce oscillations or divergence, preventing the algorithm from converging to an optimal policy.

2.5.2

Here are some strategies for initializing Q-values:

Zero Initialization: Initialize all Q-values to zero. This is a straightforward and commonly used approach. However, it may lead to slow convergence or convergence to suboptimal solutions, especially if the true Q-values have a non-zero mean.

Random Initialization: Initialize Q-values with small random values. This introduces exploration and helps break symmetries. However, the impact of randomness may make it challenging to reproduce results.

Optimistic Initialization: Set Q-values to a high constant value. This encourages exploration in the early stages of learning, as the agent believes all actions have high potential rewards. Over time, Q-values are updated based on the observed rewards.

Prior Knowledge Initialization: If there is prior knowledge about the problem, initialize Q-values based on that knowledge. This can provide a head start and accelerate learning. However, it requires domain expertise.

In my opinion, a combination of optimistic initialization and exploration strategies (such as epsilon-greedy) often works well. Optimistic initialization provides a bias towards exploration in the early stages, and exploration strategies ensure ongoing exploration throughout the learning process.

2.5.3

Our exploration strategy is a decreasing ε. At the beginning of the training, our ε is large enough to allow exploration of all possible actions independently on the q-value. ε decreases with respect to the number of episodes as the training allows the algorithm to converge.
After implementing this strategy, our average episodic reward (38) increases faster compared to a fixed value of epsilon (26)

Figure 38: Average reward with a decreasing epsilon wto. episodes