Lab 1

Fredrick Carlsåker Hampus B Hallberg

December 2025

1 VI and DP

a)

Time horizon $T = 20$, as per given by the instructions.

$$s \in \Big\{x : \text{Player position}, m : \text{Minotaur position}\Big\} \cup \Big\{\text{Eaten, Win}\Big\} \tag{1}$$

Where player position can be all legal positions for the player, i.e not in a wall. The minotaurs position are all positions inside the maze, and Win/Eaten states are the terminal states.

$$a \in \Big\{\text{up, down, left, right, stay}\Big\} \tag{2}$$

These are actions taken by the player. The minotaur moves uniformly random across its possible moves, and cannot stay still. The transition probabilities are as follows:

$$p(s' \mid s, a) = p(x' \mid x, a) \cdot p(m' \mid m) \tag{3}$$

where the player moves deterministically

$$p(x' \mid x, a) = 1,$$

and the minotaur uniformly random, among the legal actions

$$p(m' \mid m) = \frac{1}{v}.$$

Here v is the number of valid moves for the minotaur. Furtermore, we have the special cases of

$$p(s' \mid x, m, a) = \begin{cases} 1, & \text{if } s' = \text{Eaten and } x = L,\ m = L, \\ 1, & \text{if } s' = \text{Win and } x = B,\ m \neq B, \end{cases} \tag{4}$$

where B is the goal position, and L is any legal position for the player. Lastly, the terminal states are absorbing, meaning that

$$p(s' \mid s, a) = \begin{cases} 1, & \text{if } s = \text{Eaten and } s' = \text{Eaten}, \\ 1, & \text{if } s = \text{Win and } s' = \text{Win}, \end{cases} \tag{5}$$

Three different reward functions were used throughout the report:

Normal reward

$$r(s, a) = \begin{cases} -1 & : \text{Step Reward} \\ 100 & : \text{Win Reward} \\ -10^6 & : \text{Impossible Reward} \\ -100 & : \text{Eaten Reward} \end{cases} \tag{6}$$

Sparse reward

$$r(s, a) = \begin{cases} 0 & : \text{Step Reward} \\ 1 & : \text{Win Reward} \\ 0 & : \text{Impossible Reward} \\ 0 & : \text{Eaten Reward} \end{cases} \tag{7}$$

And sparse reward with large impossible reward

$$r(s, a) = \begin{cases} 0 & : \text{Step Reward} \\ 1 & : \text{Win Reward} \\ -10^6 & : \text{Impossible Reward} \\ 0 & : \text{Eaten Reward} \end{cases} \tag{8}$$

The impossible reward is for when the player moves into a wall, aiming to make that "impossible" due to its large cost. The step reward was chosen to encourage leaving the maze as fast as possible, and the Win and Eaten reward were chosen to be larger and equally opposite, to strike a balance between avoiding being eaten and progressing toward the goal. Important to note, that for the dynamic programming especially, many different configurations of the rewards were possible and gave the same result. For others, different rewards gave different results. SARSA worked best with a sparse reward, while Q-learning worked better with the normal reward.

b)

Letting the minotaur stay still yields the same MDP definition, with one difference. Since every round, first the player moves, then the minotaur, the modification is as follows:

$$p(\text{Eaten} \mid x', m', a) = \begin{cases} 1 & \text{if } x' = m \text{ or } m' = x', \\ 0 & \text{otherwise}. \end{cases} \tag{9}$$

This means that the player can be eaten both if the player moves into the old position of the minotaur, or if the minotaur moves to the new position of the player. With everything else being equal, there are more ways to lose in the second formulation, than the first. Therefore, it is slightly more likely for the minotaur to catch the player.

c)

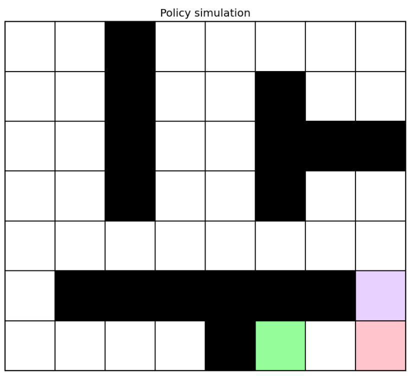
Using dynamic programming (DP) ensures that the agent always wins, if it is possible to do so. Simulating 10 000 games showed a 100% win rate. The minotaurs movements is limited and can therefore be exploited. The agent starts by moving straight to the goal, then when it nears the minotaur, it employes two main tactics. The agent can always move safely to the position that the minotaur is currently in, since the minotaur has to move every round. It can also safely move to any position diagonally of the current minotaur position, since the minotaur cannot move diagonally. If it is not possible to move to one of these safe positions, the player will wait. The different tactics can be seen in Figure 1.

d)

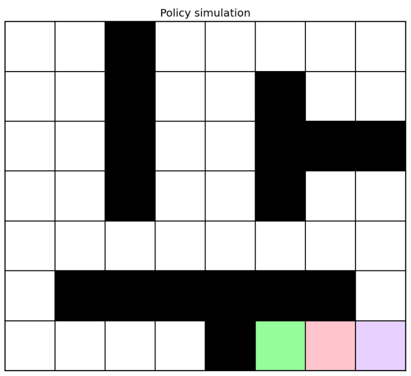
With horizon $T = 30$, the win rates for the situations where the mino is allowed to stay still or not can be seen in Table 1. These where calculated by simulating 10 000 games and keeping track of the number of wins n_{win}, with win rate $w_r = \frac{n_{win}}{n_{total}}$.

Using the sparse reward of 1 for Win, and 0 for all others, we can estimate the probability as a function of horizon, since the value of $V(s_{start})$ corresponds to the expected return, and the total return possible is 1. Therefore it can be seen as a rough estimate of the win probability. The probabilities for winning $T = 1, ..., 30$ can be seen in Figure 2. The probability is zero when the horizon is too small for the player to make it to the end. It rises dramatically around $T = 15$, since that is the minimum number of moves required to reach the goal.

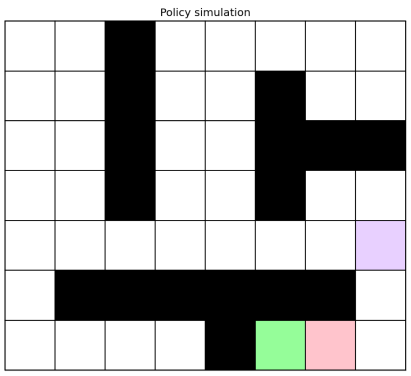
The difference between allowing the minotaur to stay still, is that the probability does not converge to 1, since there are some situations that could lead to a loss. Since the minotaur can stay still, the player is no longer guaranteed to be able to get past the minotaur, within the time frame. If the minotaur has to move, the player can always move into the minotaur to get past it. This is no longer true when the minotaur can stay still. The longer the horizon is, the less likely that the minotaur will stay in a position where it is blocking the player from advancing toward the goal, but nonetheless, there is always a chance that the minotaur would for example get to position $(4, 2)$ and continously roll a stay, for the rest of the game. Therefore, even though highly unlikely, the probability can never be exactly 1 as it is in the situation when the minotaur has to move every round.



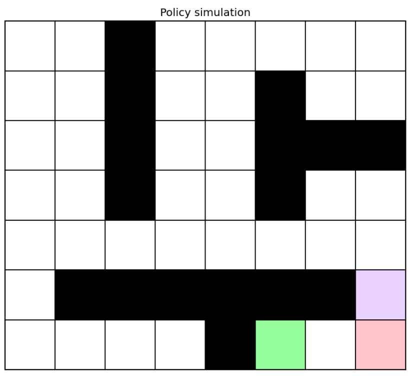
(a) Frame t, showing the player moving into the minotaurs position.



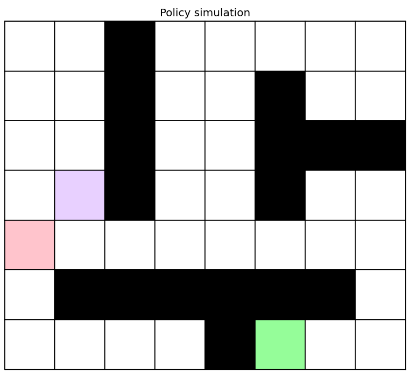
(b) Frame $t + 1$, showing the player moving into the minotaurs position.



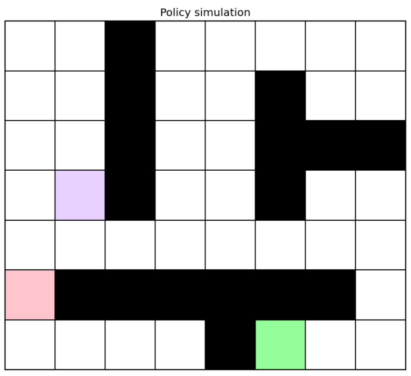
(c) Frame t, showing the player moving to the diagonal of the minotaur.



(d) Frame $t + 1$, showing the player moving to the diagonal of the minotaur.



(e) Frame t, showing the minotaur waiting. Obtained using starting position (6, 0) for minotaur.

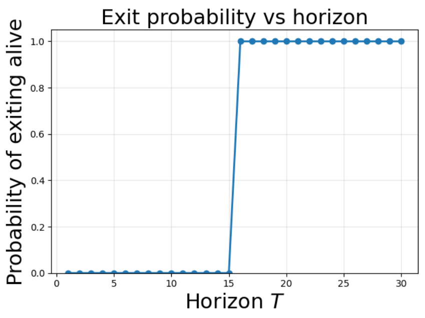


(f) Frame $t + 1$, showing the minotaur waiting. Obtained using starting position (6, 0) for minotaur.

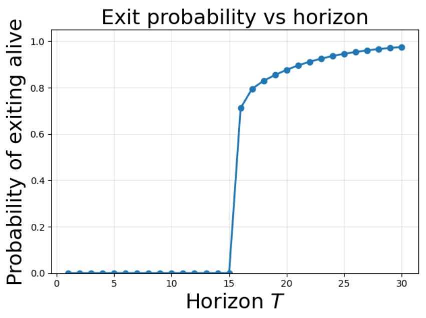
Figure 1: Illustrating three different types of agent behaviour under the learned policy.

Setting	**Win rate** $(T = 30)$ **Dynamic Programming**
Minotaur allowed to stay still	97%
Minotaur has to move	100%

Table 1: Estimated win rate for horizon $T = 30$ under different minotaur movement rules.



(a) Exit probabilities vs. horizon when the minotaur must move every round.



(b) Exit probabilities vs. horizon when the minotaur may stay still.

Figure 2: Exit probability curves for the two Minotaur movement rules.

e)

The players life being geometrically distributed with mean $\mu = 30$, means that at every time t, there is a chance $\frac{1}{\mu}$ that the player dies, i.e. every round there is a probability $\gamma = 1 - \frac{1}{\mu}$ of surviving, independently. This means that the $p(\text{Being alive at time } t) = p(\text{surviving round } 1) \cdots p(\text{surviving round } t) = \gamma^t$, the product of each rounds survival. We know that

$$V^\pi(s) = E_\pi\Big[\sum_{t=0}^{\infty} r_t p(\text{being alive at time } t)\Big] = E_\pi\Big[\sum_{t=0}^{\infty} r_t \gamma^t\Big] \tag{10}$$

This is exactly a discounted MDP.

$$V^\pi(s) = r(s,a) + \gamma \sum_j p(j \mid s, a) V^\pi(j) \tag{11}$$

f)

g)

On-policy means that the agent learns based on the policy it is currently following, meaning that updates to the policy are made based on the actions the

Setting	**Win rate ($\mu = \frac{1}{30}$) Value Iteration**
Minotaur allowed to stay still	99%
Minotaur has to move	100%

Table 2: Estimated win rate with discounted MDP from e), under different minotaur movement rules. Values obtained by simulating 10 000 games.

agent actually takes. Off-policy means that the agent learns based on a different policy than the one it is following, meaning that updates to the policy can be made based on actions other than those the agent actually performs.

Q-learning converges to Q^* with probability 1 if:

- $\sum_t \alpha_t = \infty$
- $\sum_t \alpha_t^2 < \infty$
- $\gamma \in (0, 1)$
- The agent visits every state–action pair infinitely many times over an infinite amount of time.
- Stationary transition probabilities
- Bounded rewards

SARSA converges to Q^π with probability 1 if:

- Same conditions as for Q-learning
- The policy is chosen using an ϵ-greedy strategy based on the Q-values at the current time step.

h)

To model that the agent can be poisoned, that it needs to pick up a key, and that the minotaur sometimes moves toward it, the state space must be expanded so that for every combination of the minotaur's and the agent's positions, it also includes whether the agent has the key or does not have the key.

$$s \in \Big\{x : \text{Player position},\ m : \text{Minotaur position},\ key\Big\} \cup \Big\{\text{Eaten, Win}\Big\} \quad (12)$$

With $key \in (0, 1)$ where $key = 1$ means that the agent is holding the key, and $key = 0$ means that the agent is not holding the key. key will therefore be 0 until the agent enters C, from which point it will be 1.

The transition probabilities also need to be modified so that there is a 35% chance that the minotaur moves in the direction of the agent, and so that the transition for picking up the key is handled. Defining the move in the direction of the agent as the move along the axis on which the minotaur is furthest away from the player, i.e. if the player is 4 steps above and 1 to the left, the minotaur will move up, we obtain the following:

$$p(x' \mid x, a) = 1,$$

but that

$$p(m'_l \mid m) = 0.65 \cdot \frac{1}{v}$$

$$p(m'_d \mid m) = 0.35$$

Here, m'_d represents the legal move in the direction of the player, and m'_l represents the remaining legal positions for the minotaur. v is the number of legal positions for the minotaur that are not in the direction of the player. For example, with four legal moves, each of the three, not in the direction of the player, will each have a probability of $0.65\frac{1}{3}$, and since there are three of them, to total probability of not moving toward the player is 0.65.

To incorporate that the agent's lifetime is geometrically distributed with a mean of 50, an additional terminal state could be added to represent that the agent has been poisoned, and at each step there would be a probability of $\frac{1}{50}$ that the agent transitions to this state regardless of the action taken. However, this can also be modeled using a discount factor in the learning algorithm, as shown in **e)**.

2 Q-Learning and Sarsa

i)

1

Pseudo code for the implementation of the Q-learning algorithm can be found in Algorithm 1.

2

In Figure 3, a plot of $V(s_0) = max_a(Q(s_0, a))$ over 50,000 episodes is shown, with $\epsilon = 0.01$ and $\epsilon = 0.5$. Here, the normal reward was used, and the plot has been normalized by the goal reward. For both $\epsilon = 0.01$ and $\epsilon = 0.5$, $V(s_0)$ converges towards a stable value around 0.6. With $\epsilon = 0.5$, the algorithm explores more, which in this case leads to a higher convergence value, and faster learning, especially in the beginning. Since Q-learning is off-policy, higher exploration often leads to the agent finding a more optimal strategy quicker. With $\epsilon = 0.01$, the agent mainly exploits its current estimates and visits fewer state–action pairs, which results in the agent tending to follow one path to the goal to a greater extent once it has found one. This causes the agent to reinforce the path to the goal more strongly, but it does not find new, better paths as quickly.

Algorithm 1 Q-learning

```
1:  Input: Number of episodes $N$, transition probability matrix, start position,
    $\gamma$, $\varepsilon \dots$
2:  procedure
3:      Initialize $Q$ values and other parameters
4:      for episodes $k = 1, 2, \dots, N$ do
5:          Initialize environment and read initial state $s_0$
6:          $t \leftarrow 0$
7:          while Episode $k$ is not finished do
8:              Select starting position - 50% split between random and start
9:              Select action $a_t$ $\varepsilon$-greedily
10:             Observe next state $s_{t+1}$ and reward $r_t$
11:             Calculate step size
12:             if Episode is finished then
13:                 Update Q without future rewards
14:             else
15:                 Update Q with reward, current Q and future maximum reward
16:             end if
17:             $t \leftarrow t + 1$
18:         end while
19:
20:     end for
21: end procedure
```

The choice of Q-initialization affects convergence speed. Optimistic initialization means that the initial values are larger than the expected reward. Since the normal reward was used, this was an optimistic initialization. Since every normal step gets a negative reward of -1, the agent is encouraged to explore actions it has not yet tried. This, combined with the fact that the learning rate decreases for state–action combinations that have been visited often, leads the agent to prioritize trying many different actions and thereby to optimize the path.

If instead pessimistic initialization had been used, convergence to the optimal policy would be slower because it does not explore as much, but once a path to the goal has been found, it is consolidated. Thus, ε and the initialization can have somewhat similar effects on convergence: high ε and optimistic initialization lead to a high update/visitation rate and an ambition to find a very good policy directly, but in combination they can cause it to take a long time to find any working policy at all. Low ε and pessimistic initialization instead lead to a low update/visitation rate and a tendency to find a policy quickly, but in combination they can cause a poor policy to be preserved and convergence to the optimal one to take a long time.

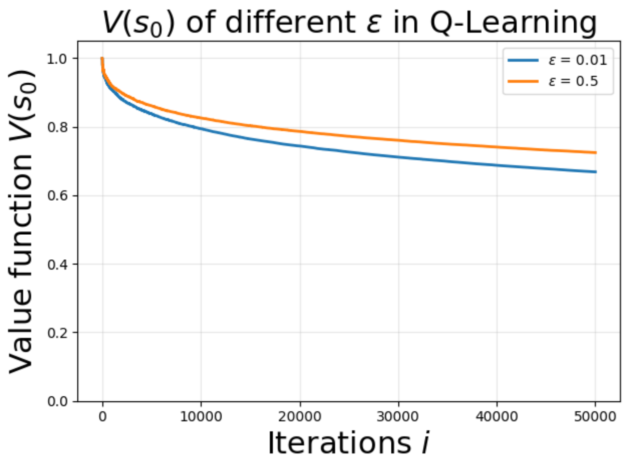


Figure 3: Plot of value function over episodes of the initial state, for varying ε

3

In Figure 5, a plot of $V(s_0) = max_a(Q(s_0, a))$ over 50,000 episodes is shown, with $\epsilon = 0.1$ and step size $\frac{1}{n(s,a)^\alpha}$ for $\alpha = 0.6$ respectively $\alpha = 0.9$ where $n(s, a)$ is the number of times that particular state–action pair has been visited. We know that the true $V^*(s_0)$ is somewhere around 0.6, so both curves decrease toward

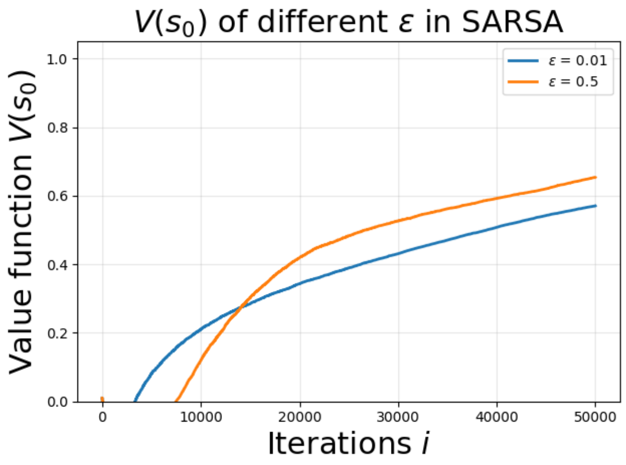


Figure 4: Plot of value function over episodes of the initial state, for varying ε

that value. $\alpha = 0.6$ means that the step sizes stay relatively larger for longer, so the algorithm adapts more aggressively and "forgets" the initialization faster. $\alpha = 0.9$ makes the step sizes shrink very quickly, so the algorithm becomes conservative: updates get tiny, and the Q-values remain closer to their initial value for a long time. Thus, a smaller α results in faster learning but higher variance, while a larger α yields earlier stability but slower learning.

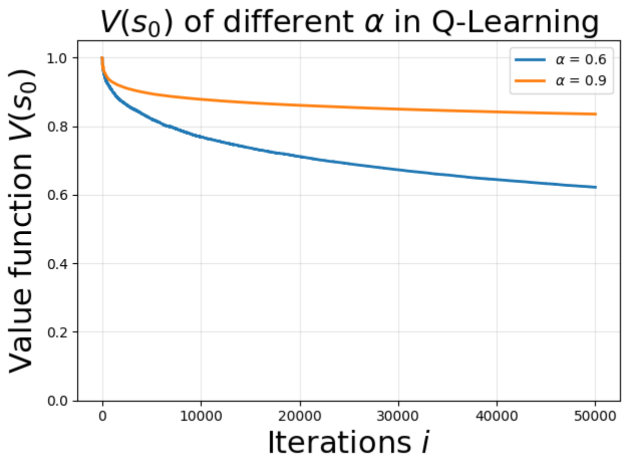


Figure 5: Plot of value function over epsiodes of the initial state, for varying α

(j)

1

The main difference between Q-learning and SARSA is that Q-learning is off-policy, updating $Q(s, a)$ towards the value of the greedy action, whereas SARSA is on-policy and updates towards the value of the actually executed action. In practice this means that Q-learning learns the value of a greedy policy even when behaviour is exploratory, while SARSA learns the value of the ε-greedy policy itself, i.e. it explicitly "expects" random actions.

2

Figure 6 shows $V(s_0)$ for SARSA with $\varepsilon = 0.1$ and $\varepsilon = 0.2$ when a large negative "impossible reward" is given for stepping into walls. Because SARSA is on-policy, the frequent random moves (every 10 or 5 steps on average) lead to many such collisions, especially in this maze where the optimal path passes very close to walls. As a consequence the value function is dominated by these large negative rewards, and both runs perform poorly, with $\varepsilon = 0.1$ still clearly better than $\varepsilon = 0.2$ since fewer random moves are taken. Initializing with zeros actually makes the value function not converge, while initializing with ones does. As previously discussed, this reflects the importance of biasing the agent toward trying new actions, which is crucial when the reward signal is very sparse.

When the impossible reward is removed (Figure 7), the value function is no longer dominated by very large penalties. The policy obtained with SARSA is still worse than the one obtained with Q-learning, but it now manages to solve the task in roughly 45% of the simulated episodes, again with $\varepsilon = 0.1$ outperforming $\varepsilon = 0.2$. This behaviour is consistent with SARSA learning the value of a policy that "expects mistakes". Even after convergence, the learned values reflect occasional random moves, which makes SARSA less suitable here since only the minotaur moves stochastically, while the agent itself is deterministic. The SARSA algorithm shines when the player's moves are also stochastic and less is known about the environment.

3

Finally, Figure 8 shows SARSA with a decaying exploration rate $\varepsilon_k = 1/k^\delta$ and step-size decay of $\alpha = 2/3$. In this case the performance improves dramatically and the resulting policy wins in about 95% of 10 000 simulations. As ε decays, random moves become rare and the behaviour approaches that of a greedy policy, so SARSA behaves more like Q-learning while still being on-policy. In practice it is desirable that ε decays somewhat faster than the step size (i.e. $\delta > \alpha$): this ensures sufficient exploration early on, but guarantees that the policy eventually becomes nearly greedy while the learning rate is already small

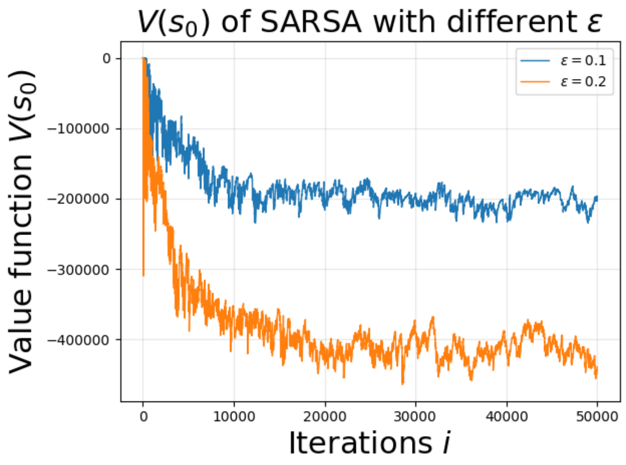


Figure 6: Plot of $V(s_0)$ vs. episodes for SARSA, with $\varepsilon = 0.1, 0.2$ and a large negative reward for stepping into walls.

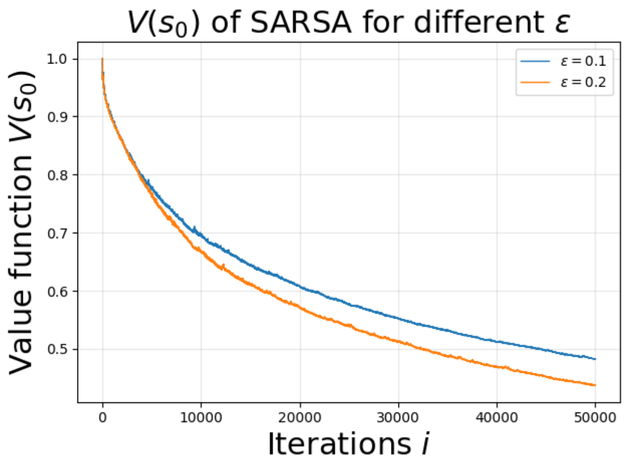


Figure 7: Plot of the $V(s_0)$ vs episodes, with $\varepsilon = 0.1, 0.2$ without the impossible reward of stepping into walls.

enough to stabilise the value estimates. In settings like this maze, where the environment dynamics are essentially known and the agent is deterministic, Q-learning is still preferable; SARSA is more advantageous when the dynamics or the agent behaviour are inherently noisy.

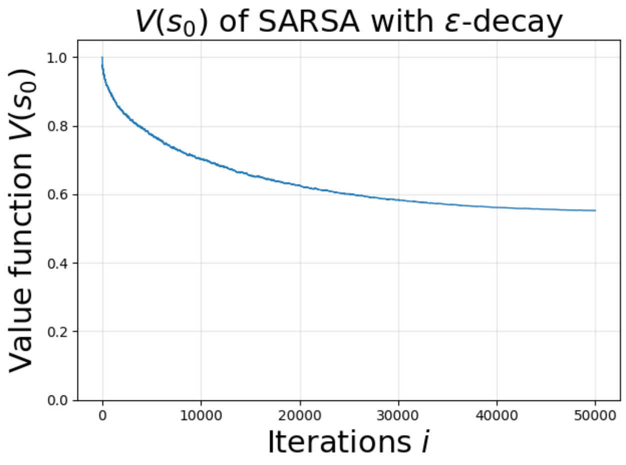
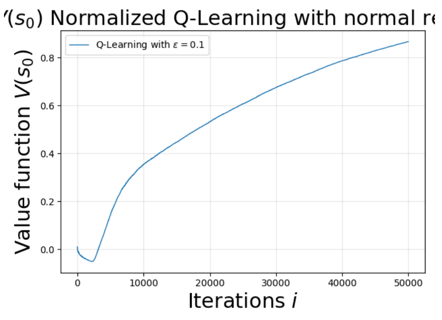


Figure 8: Plot of $V(s_0)$ vs. episodes for SARSA with decaying exploration parameter $\varepsilon_k = 1/k^\delta$.

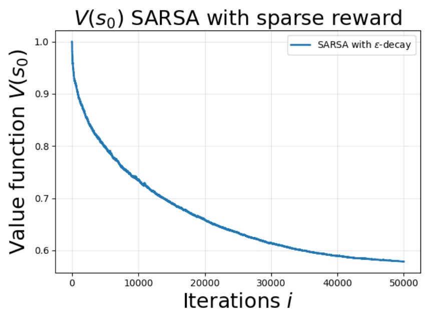
k)

Using Q-learning, with $\varepsilon = 0.1$, and SARSA with $\varepsilon_k = \frac{1}{k^\delta}$, with $\delta = 0.9$ and both functions using a step size of $\eta = \frac{1}{n(s,a)^\alpha}$ where $n(s,a)$ is the number of times the state action pair was visited, where for Q-learning $\alpha = \frac{2}{3}$ and for SARSA, $\alpha = 0.5$. Q-learning gets the best result when using the normal reward function, while SARSA performs better with the sparse. The value functions in Figure 9 cannot be seen as probabilities exactly, but it still shows something about the expected reward. The Q-learning plot has been normalized by the goal reward, to enable comparison. Simulating these policys for 10 000 games yields a 100% win rate for Q-learning and a 97% win rate for SARSA.

The value functions differ slightly. Q-learning, with a sparse reward more closely resembles a true probability, since it is trained off-policy. However, since it performed best with the normal reward, the value cannot be strictly seen as a probability (even when normalized), since the expected return includes, for example, the negative step rewards. The SARSA algorithm performed best with the sparse rewards, but since it learns an on-policy ε-greedy policy, it does not quite reflect the fully greedy policy that is actually used in simulation. Since the agent "expects" mistakes, the value function reflects that and that is why it converges to a value that is lower than the actual simulated win rate.



(a) Q-learning: $V(s_0)$ over training episodes with $\varepsilon = 0.1$ and $\alpha = \frac{2}{3}$ using the normal reward function. $V(s_0)$ has been normalized by the goal reward.



(b) SARSA: $V(s_0)$ over training episodes with $\varepsilon_k = 1/k^{0.9}$ and $\alpha = 0.5$ using the sparse reward function.

Figure 9: Comparison of value function evolution $V(s_0)$ for Q-learning (left) and SARSA (right) under different exploration and learning-rate schedules.

3 MountainCar

c)

Training procedure

We trained an on-policy Sarsa(λ) agent with linear function approximation based on a Fourier basis of order $p = 2$.

Episodes and hyperparameters. We trained the agent for $N_{\text{ep}} = 100$ episodes. Each episode started from an initial state s_0 sampled from the given set of initial states. We used

- Discount factor: $\gamma = 1$,
- Eligibility trace parameter: $\lambda = 0.7$),
- ε-greedy exploration $\varepsilon = 0.01$)
- momentum $m = 0.1$,
- Learning rates α_i of the form

$$\alpha_i(k) = \frac{\alpha}{\|\eta_i\|_2},$$

  With $\alpha = 0.01$.

  We used an α-decay, where if the total epsiode reward $R_{ep} > -145$, $\alpha_{t+1} = 0.7 \cdot \alpha_t$

  We used an ε-decay of $\varepsilon_{t+1} = \varepsilon_t \cdot 0.7$.

Fourier basis and function approximation. We used a Fourier basis of order $p = 2$. We tried both with and without the constant base vector $\eta_0 = (0, 0)$. It did not make a significant difference in the score, but a slightly smoother reward curve seems to be obtained when it is included.

Algorithm Given the current state s_t (normalized to be between $[0, 1]$) and action a_t chosen ε-greedily from $\hat{Q}(s_t, \cdot)$, we observe the reward r_t and the next state s_{t+1} (or a terminal state). We then:

1. Compute the TD-error

$$\delta_t = r_t + \gamma \, \hat{Q}(s_{t+1}, a_{t+1}; w) - \hat{Q}(s_t, a_t; w),$$

   where a_{t+1} is chosen according to the same ε-greedy policy (for non-terminal s_{t+1}).

2. Update eligibility traces for all actions a and add $\phi(s)$ to the selected action a_t:

$$z_a \leftarrow \gamma\lambda \, z_a, \quad z_{a_t} \leftarrow z_{a_t} + \phi(s_t).$$

3. Perform a stochastic gradient descent with momentum step on $\mathbf{w}$:

$$\mathbf{v} \leftarrow m\mathbf{v} + \alpha \, \delta_t \, \mathbf{z},$$

$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}\alpha \, \delta_t \, \mathbf{z},$$

   where α is the learning rate.

At the end of the episode (either after T steps or when a terminal state is reached), we start the next episode with reset traces, but we keep the updated weights $\mathbf{w}$.

SGD modifications We used stochastic gradient descent with momentum, and a decaying learning rate α as a function of the last episode reward. The momentum $m = 0.1$ with a starting learning rate of $\alpha = 0.01$ was used. The learning rate was multiplied by a factor 0.7, every time the total epsiode reward was higher than -145. This step proved to be very important for convergence. It is a bit strange that a very low momentum m proved to be ideal. Perhaps there are other configurations, with higher momentum and lower α that would be even better, but since it worked very well, we kept it.

d)

1. **Episodic return during training.**

   Figure 10 shows the total return as a function of the training episode index. The blue curve corresponds to the raw return of each episode, while the orange curve shows the running average over 10 episodes.

At the beginning of training the agent obtains returns close to -200, which means that it fails to reach the goal and simply accumulates the step penalty. After roughly 10–20 episodes the performance improves substantially and the average return quickly rises above -135. After this transient phase the average return remains relatively stable around -120, while the per-episode return continues to fluctuate due to the stochastic ε-greedy exploration. Overall, the plot indicates that the algorithm converges to a reasonably good policy within a relatively small number of episodes.

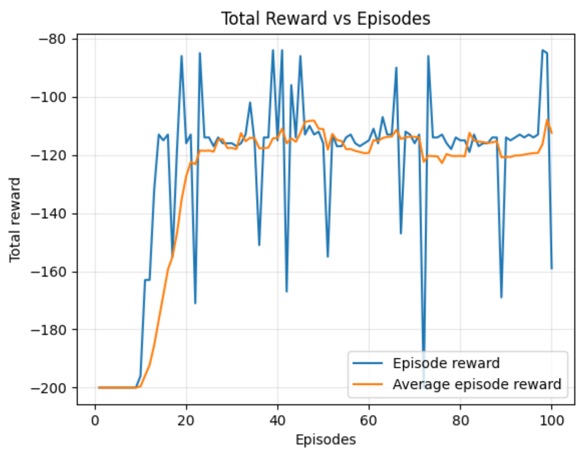


Figure 10: Episodic total reward during training. The blue curve shows the return of each episode, while the orange curve shows the running average over the last 10 episodes.

2. **Value function of the learned policy.**

   Figure 11 shows the approximate state-value function $V(s) = \max_a \hat{q}(s, a)$ of the learned policy over the continuous state space, parametrised by position and velocity. The plot has the shape of a smooth "bowl": states around the bottom of the valley (moderate position and low velocity) have the lowest value, since from these states many time steps are required to build up enough momentum to reach the goal. States towards the right-hand side of the domain, especially with positive velocity, have much higher value because the car is already close to the goal and can reach it in only a few steps. Symmetrically, states far on the left with sufficient negative velocity also have higher value, as they are good starting points for swinging back to the right with large momentum. This structure is consistent with the dynamics of the MountainCar environment and therefore the learned value function is reasonable.

3. **Optimal policy over the state space.**

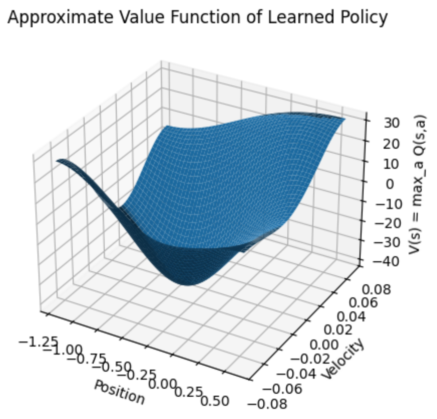


Figure 11: Approximate value function of the learned policy over the state space (position–velocity plane), computed as $V(s) = \max_a \hat{q}(s, a)$.

Figure 12 shows the greedy policy with respect to the learned action-value function, plotted over the state space. The three discrete actions are encoded on the vertical axis as 0 = left, 1 = no push, and 2 = right. We can observe three main regions:

- For positions far to the left, with negative velocity, the policy mostly chooses action “left” in order to continue climbing the left hill and thereby gain more potential energy that can later be converted into momentum.
- As the velocity nears zero, the policy switches direction and tends to choose the “right” action. This corresponds to the agent exploiting the previously built momentum to drive up the right hill towards the goal.
- Near the right-hand part of the state space the policy predominantly chooses the “right” action, as the goal can be reached directly from these states. But again, when the velocity approaches 0^+ it switches to left, to build momentum in the other direction.

The overall structure of the learned policy is therefore very intuitive: the car first swings back and forth to build momentum and then commits to driving up the right hill when the state is favourable.

4. **Effect of including the $\eta = [0, 0]$ basis vector.**

In the Fourier basis we used frequency vectors $\eta = (\eta_1, \eta_2)^\top$ with components in $\{0, 1, 2\}$. The special case $\eta = [0, 0]$ corresponds to the constant feature

$$\phi_{[0,0]}(s) = \cos\bigl(\pi\, [0, 0]^\top \tilde{s}\bigr) = 1,$$

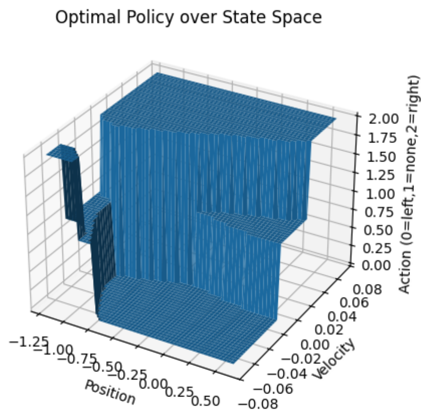


Figure 12: Greedy policy with respect to the learned action-value function, plotted over the state space. The vertical axis encodes the discrete action: 0 = left, 1 = no push, and 2 = right.

which acts as a bias term in the linear approximation. We repeated the experiments with and without this feature.

Qualitatively, the overall shape of the value function surface and the structure of the greedy policy did not change much when we excluded the $\eta = [0, 0]$ term: the agent still learned the swing-up behaviour and reached the goal reliably. However, including the constant feature slightly improved the numerical properties of the approximation. With $\eta = [0, 0]$ the value function could freely adjust its global offset, which led to slightly faster and smoother convergence of the episodic returns. Without this term the other Fourier components must compensate for the missing offset. This mainly manifests as a vertical shift of the value surface and somewhat larger temporal-difference errors during training, while the induced greedy policy remains essentially the same.

In summary, including the $\eta = [0, 0]$ feature does not fundamentally change the learned policy for MountainCar, but it does provide a useful bias term that improves the quality and stability of the value-function approximation.

e)

1)

We first analyse how the choice of learning rate α and eligibility trace λ affects the performance of the Sarsa(λ) agent. For each parameter value, we trained an agent for 100 episodes and computed the average total reward over 50 simulated episodes. The threshold for solving the problem is -135.

Figure 13 shows the average return as a function of the initial learning rate α_0. Very small step sizes (e.g. $\alpha_0 = 0.001$ or 0.01) yield stable learning and achieve average returns around -130, clearly above the task threshold. Increasing α_0 beyond 0.05 results in significantly worse performance. For $\alpha_0 = 0.1, 0.3$, the agent diverges and performs very poorly (average return around -200). This behaviour is expected because a large step size causes overly aggressive updates of the value function approximation, which destabilizes learning.

Figure 14 shows the effect of the eligibility trace parameter λ. Intermediate

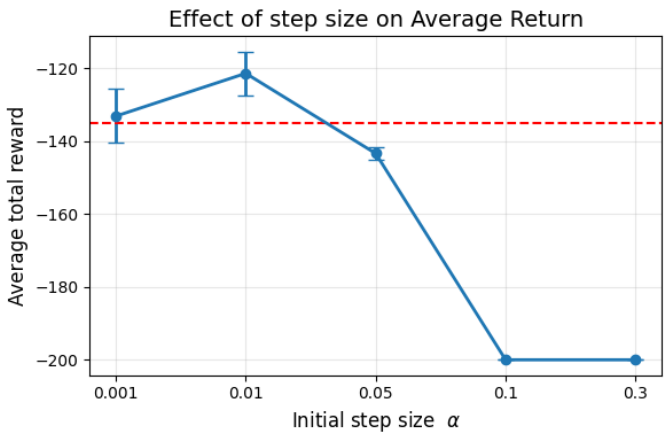


Figure 13: Effect of initial step size α_0 on the average return, with confidence intervals, over 50 episodes. The dashed red line marks the performance threshold -135.

values ($\lambda \in [0.5, 0.9]$) provide the best performance, with a peak in $\lambda = 0.5$, 0.7. Very short traces ($\lambda = 0.1$) under-utilise the benefits of temporal credit assignment and therefore, learn more slowly. On the other hand, very long traces ($\lambda = 0.9$) can propagate TD errors too aggressively, resulting in instability and poor performance. This reproduces the typical empirical trade-off: mid-range λ values offer a good balance between bias and variance.

2)

In value-based reinforcement learning, the initialisation of the value function (or, equivalently, the weight parameters in function approximation) can substantially influence early exploration and the overall convergence behaviour. We experimented with several common initialisation strategies:

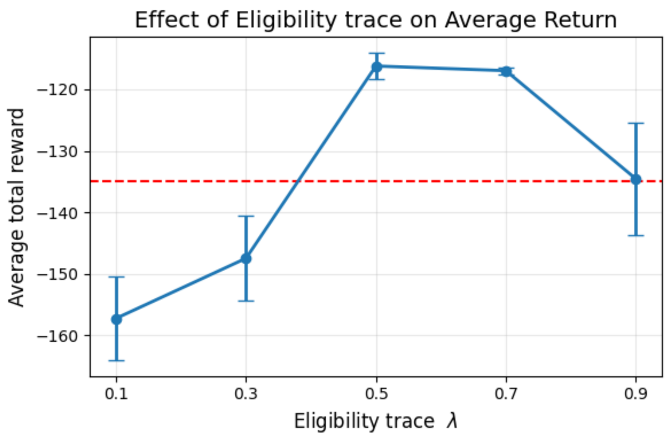


Figure 14: Effect of eligibility trace parameter λ on the average return, with confidence intervals, over 50 episodes. The dashed red line marks the performance threshold -135.

Initialising at 0 **or** 1 Setting all values to zero is a widely used and stable baseline. In the MountainCar environment, where rewards are negative at every step, initialising at 0 or 1 has a very similar effect: both choices are *optimistic* relative to the true value function, which is strictly negative. As a result, the agent is naturally encouraged to explore, since unseen state–action pairs appear more promising than those already experienced and updated with negative TD errors.

This behaviour contrasts with environments with sparse positive rewards (e.g., the maze problem). There, initialising at 0 may prematurely bias the agent toward the first action that yields a positive reward, reducing exploration, while initialising with a higher constant (e.g. 1) preserves optimism and encourages further exploration. In MountainCar, however, both 0 and 1 serve as optimistic initialisation, and thus their practical effect on learning is similar.

Random initialisation A second strategy is to initialise the value function (or weights) with small random values, for example from a distribution such as $\mathcal{N}(1, 0.01^2)$ or $\mathcal{N}(0, 0.01^2)$. This breaks symmetry between actions and produces a mild, stochastic form of optimism that encourages exploration without introducing large initial errors. Random initialisation can also help prevent the agent from repeatedly selecting actions that lead into bad or "trapping" regions of the state space, especially in environments where many transitions have similar negative rewards. Overall, it provides a more diverse exploratory behaviour than constant initialisation, while remaining stable. For this problem however, this proved worse.

Pessimistic initialisation Another strategy is to initialise the value function with negative values, biasing the agent to find the actions that lead to any reward very quickly and sticking to it. This demphasizes exploration, and is usually not a good idea. This resulted in no convergence and did not pass the MountainCar test. This might be good if we know there is only one single good set of actions, or in a system with dense rewards. However, in this situation, where rewards are sparse, this is not a good strategy.

3)

To compare alternative exploration mechanisms, we designed a custom **softmax (Boltzmann) exploration strategy**. Instead of ε-greedy, which selects all non-greedy actions uniformly, softmax chooses actions proportionally to their relative action-values:

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

where $\tau > 0$ is a temperature parameter. Lower temperatures make the policy close to greedy, while higher temperatures encourage exploration.

Figure 15 shows the episodic return over 100 episodes using softmax exploration, with a moderate temperature of 0.5. The agent quickly surpasses the performance threshold and converges to stable behaviour with average returns near -120. Compared to ε-greedy, the softmax strategy produces smoother learning and fewer severe drops in performance, because it continues to explore proportionally rather than uniformly.

However, this pattern was not entirely consistent. Due to random initialization, some runs displayed more similar behaviour to the ε-greedy strategy. This variability also makes direct comparisons between exploration strategies more difficult, as each run can evolve differently.

In summary, the softmax strategy provides a more principled form of exploration that tends to reduce variance in the return and leads to more stable training dynamics. Compared to the baseline ε-greedy agent, the softmax agent achieved faster convergence and more consistent performance above the task threshold.

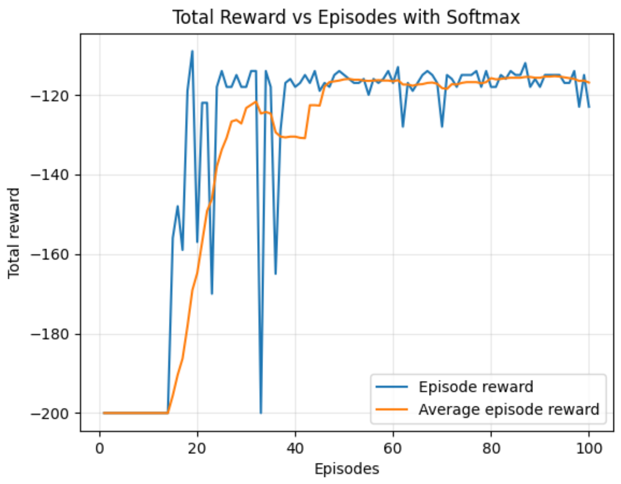


Figure 15: Total episodic reward over 100 episodes using softmax exploration. The orange line shows the running average reward.