Machine Learning – Assignment 4

Author: Mohamed Amine DASSOULI

Table of Contents

1. Introduction

In this report, we are going to explore Markov Decision Processes (*MDP*). We are going to work on two very different MDPs: a grid world problem (*Frozen Lake*), and a non-grid one (*Mountain Car*). To solve them, we are going to use three reinforcement learning techniques: value iteration, policy iteration, and Q-learning.

More details are provided in the next parts.

2. Presentation of the problems

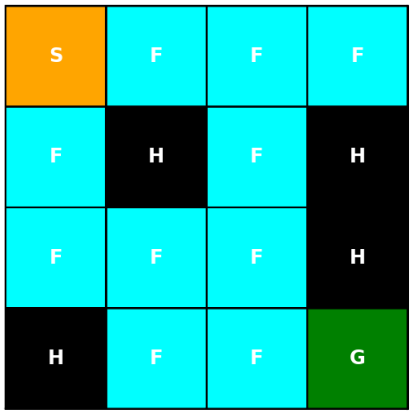


Fig 2.1: Frozen Lake map

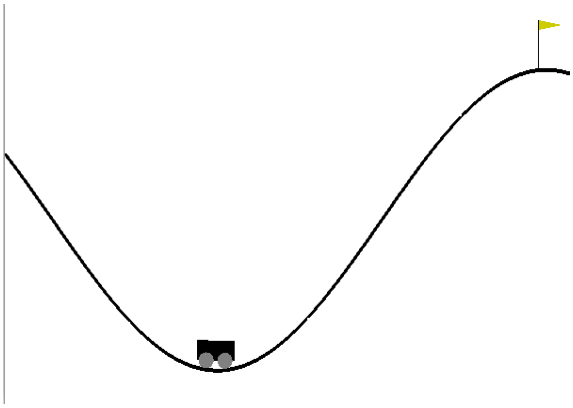


Fig 2.2: Mountain Car instance

2.1. The Frozen Lake

The problem environment is a lake, its surface is described by a grid **(Fig 2.1)**. Most of the parts are frozen (*F*), but there are some others where the ice is very thin (*H*). The goal of the agent is to go from a starting point (*S*) and reach a goal (*G*) without falling into a hole (*H*).

An episode starts with the agent being at the position (*S*), and it finishes when the agent either reaches the goal (*G*) or falls into a hole (*H*). The problem is represented as following:

- **States:** Each cell in the map represents one (there are 16 states in Fig 2.1).
- **Actions:** *Left*, *Right*, *Up*, *Down*.
- **Rewards:** 0 if the agent falls into water through a hole (*H*), or 1 if the goal is reached.

This MDP is **interesting** first because it is a world grid problem. In addition to that, we can increase/decrease the difficulty and see how the algorithms will perform under the different circumstances. In the next three parts, we are going to work simultaneously with 3 maps of different sizes: 4x4 (*Small - 16 states*), 8x8 (*Medium - 64 states*) and 16x16 (*Large - 256 states*). In the last part, we are going to discuss more experiments: add more holes, add a slippery factor...

2.2. The Mountain Car

The problem is about a car stuck in a mountain valley. The gravity being strong, the engine cannot simply drive up the hill. Instead, it needs to leverage potential energy by driving forward and backward, until it reaches the goal at the top.

An episode starts with the car being at a random position between -0.4 and -0.6, and having a velocity null. And it finishes when the engine either reaches the flag (*position 0.5*), or spends a total number of 200 steps. The problem is represented as following:

- **States:** Each different situation of the car, *(position, velocity)*, represents one.
- **Actions:** *Drive left*, *Do nothing*, *Drive right*.
- **Rewards:** -1 for each step the car does not reach the flag.

The range of possible positions are [-1.2, 0.6], and the range of possible velocities are [-0.07, 0.07]. They are continuous ranges, but to convert them to a MDP, we are going to discretize them. For each dimension, we choose a number of discrete values, then we round all the continuous values to their respective closest discrete one. In addition to making comparison between a grid world and a non-grid world problem, this is the part which makes the Mountain Car an **interesting** problem: Which discretization will work better? Is it better to work with a large value of discrete values? In the next three parts, we are going to work on two different discretizations: 10 discrete values for each dimension (*Small - 100 states*), and 25 discrete values for each dimension (*Large - 625 states*). In the last part, more experiments are going to be done.

3. Value Iteration

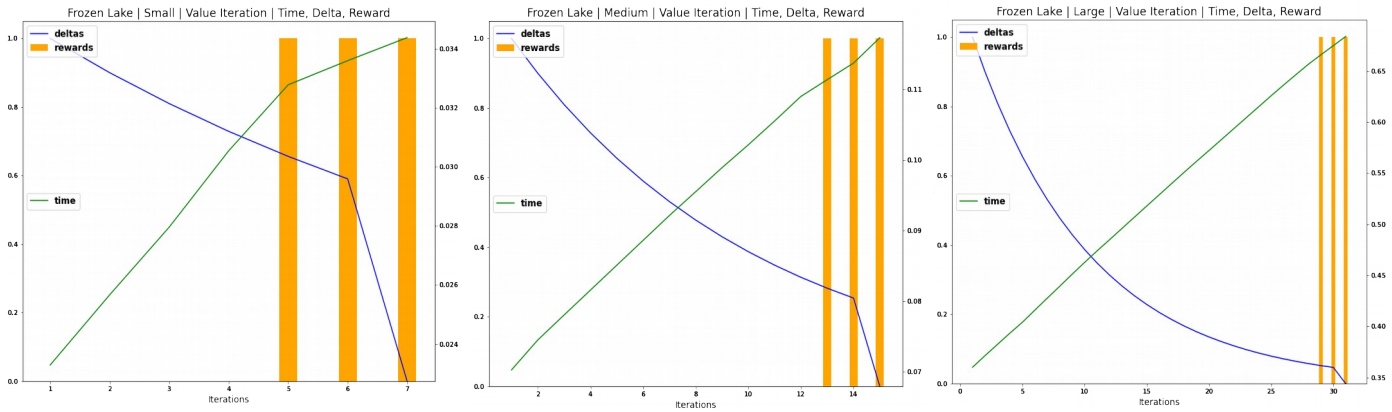
3.1. Definition

It is an algorithm that knowing the transition probabilities and rewards of a MDP, computes the optimal policy, through the estimation of the value function V. In the beginning, for each state, V(s) is initialized to an arbitrary value, then Q(s, a) and V(s) are repeatedly updated until the difference between two consecutive computed values V(s) is small compared to a threshold, which we call convergence. Value Iteration is guaranteed to converge to the optimal values.

3.2. Frozen Lake

For our experiments, we chose an arbitrary value of 0.9 for the discount factor γ in order to give a high importance to the future rewards. Concerning the *convergence threshold*, we chose a value of 10^{-3} because the problem is relatively simple (more details later).

The figures below show for each of the three grids, and for each iteration of the algorithm, the values of the Wall Clock Time (*green curve*), the V-delta (*blue curve*), which is the maximum difference on the value function between two consecutive iterations, and the reward (*orange bars*). The left y-axis corresponds to the values of V-delta, and the right one for the time values.

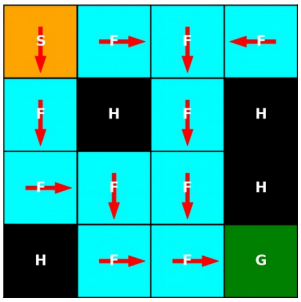


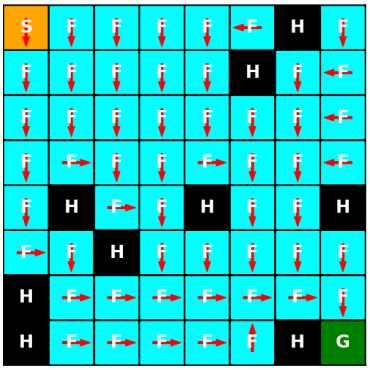
■ The bigger is the number of states, the higher is the number of iterations needed to converge, and the running time. For the small problem (*16 states*), it took *0.034 seconds* for 7 iterations. For the medium problem (*64 states*), it took *0.118 seconds* for 15 iterations. For the large problem (*256 states*), it took *0.683 seconds* for 31 iterations. We can also notice that the time per iteration gets higher when the problem gets bigger because there are more states to evaluate for V.

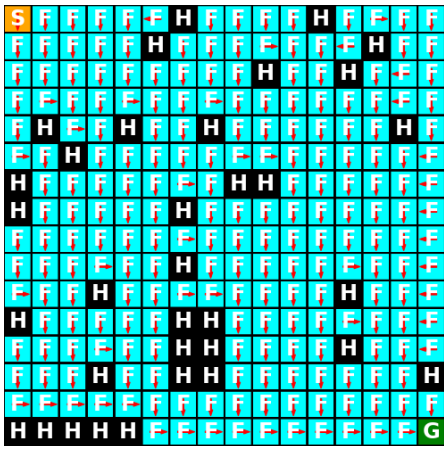
■ For the three maps, the reward starts null and becomes 1 in the three last iterations of the algorithm. This means that the policy extracted from the value function (*V*), gives a correct path to the agent (from *S* to *G*) at those steps. But even though the reward is reached, the algorithm does not directly converge (V-delta gets under the convergence threshold only at the last step), it takes two more iterations. This comes from the nature of the grid world problem, where there are many correct paths. The two last iterations change the policy, but just from one correct path to another.

■ From the curves of V-delta above, we can notice that its value decreases drastically on the last iteration compared to the one just before. This is because the problem environment is finite (discrete states, finite number of possible actions/result), and therefore V(s) can only change with a relatively high value. As a result, the only restriction on the convergence threshold is to be lower than approximately 0.02 (from the graph), and any value under that is acceptable (including 10^{-3}).

Here are the final policies, resulting from the Value Iteration algorithm:



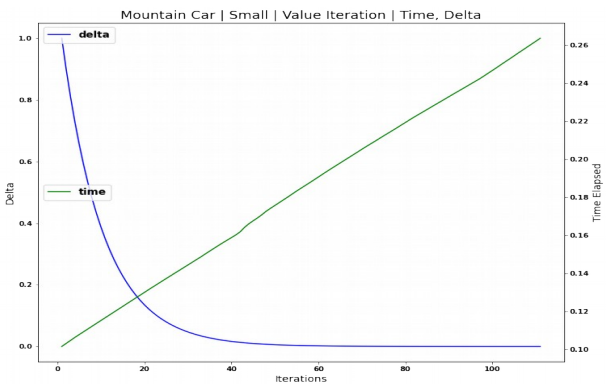


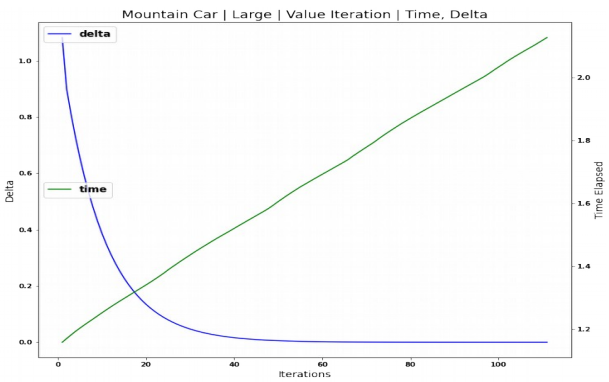


■ All the policies seem correct, they give correct actions to the agent at each state. And since there is no penalty related to the number of steps needed to reach the goal, we can visually confirm that there are a lot of optimal policies, which explains the two last iterations.

3.3. Mountain Car

Here are the figures for the Value Iteration algorithm on the Small instance (*100 discrete states*) and the large instance (*625 discrete states*) of the Mountain Car, with γ*=0.9*, *and* 10^{-5} for threshold:

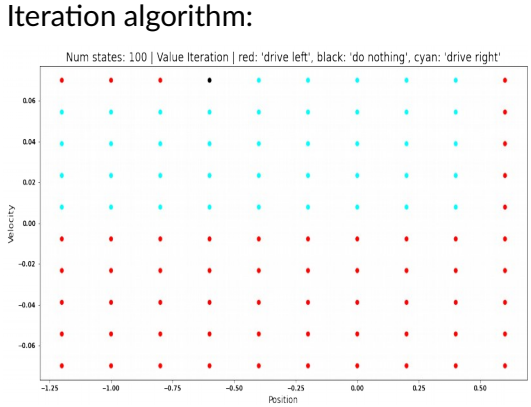


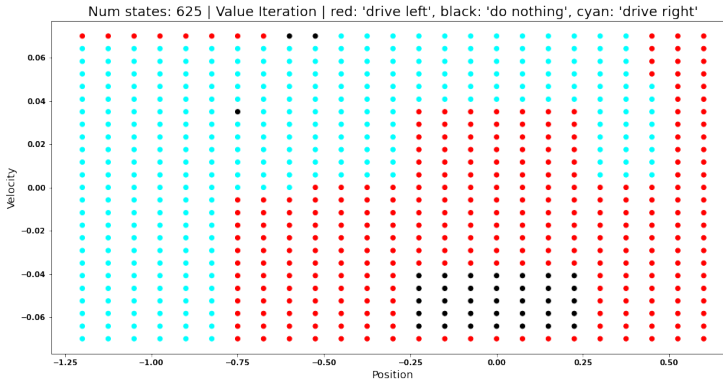


■ For the Small instance, the algorithm took *0.264 seconds* and 111 iterations to converge. For the Large instance, it took 2.*128 seconds* and 111 iterations as well. Once again, the time per iteration is bigger for the largest problem in term of discrete values because there are more states to evaluate each time.

■ Because we are confronted to a continuous problem (infinite number of discrete states in reality), the value function V keeps getting improved infinitely, with smaller variations at each iteration (we can notice that the blue curve is smooth), and the convergence is only related to the convergence threshold. The lowest is the threshold, the more precise gets V, and the higher is the number of iterations needed. For a value of 10^{-4}, it takes 89 iterations. For a value of 10^{-6}, it takes 133 iterations... Therefore, the choice of the convergence threshold is critical in this problem.

Here is a representation of the final policies for the Mountain Car, resulting from the Value Iteration algorithm:





The *x-axis* represents the positions, the y-axis represents the velocities, and the actions following the policy are represented with the colors (red for '*drive left*', black for '*do nothing*' and cyan for '*drive right*').

■ The overall structure of the two representations is similar: when the velocity is positive, it is more likely to '*drive left*' and when it is negative, it is more likely to '*drive right*'. But the policy of the Large instance (*625 discrete states*) seems more precise. It was expectable because it is closer to the real model.

■ While running 100 episodes using each policy, the average reward for the Small instance was ***-156***, while the average reward for the Large instance was ***-138*** ! This confirms that the higher is the number of discrete states, the better is the modelisation.

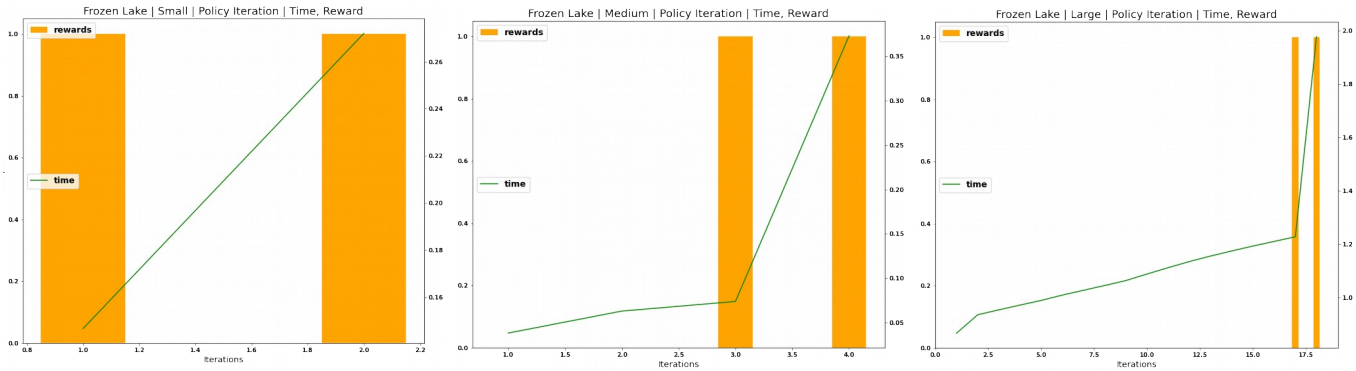
4. Policy Iteration

4.1. Definition

It is an algorithm to solve MDP and find an optimal policy. Just like Value Iteration, this second algorithm requires the transition probabilities and the rewards of the MDP. However to operate, it directly manipulates the policy, instead of using the value function. An arbitrary policy is chosen in the beginning. Then, Policy evaluation is performed, which is an inner value iteration. The next step is Policy Improvement, where a new policy is computed using the resulting value function of the previous step. The process is then repeated until the policy converges.

4.2. Frozen Lake

In this part, we are going to use Policy Iteration, once again on the three maps of Frozen Lake. Here are the results found with γ=*0.9*, *and* 10^{-5} for the inner convergence threshold:

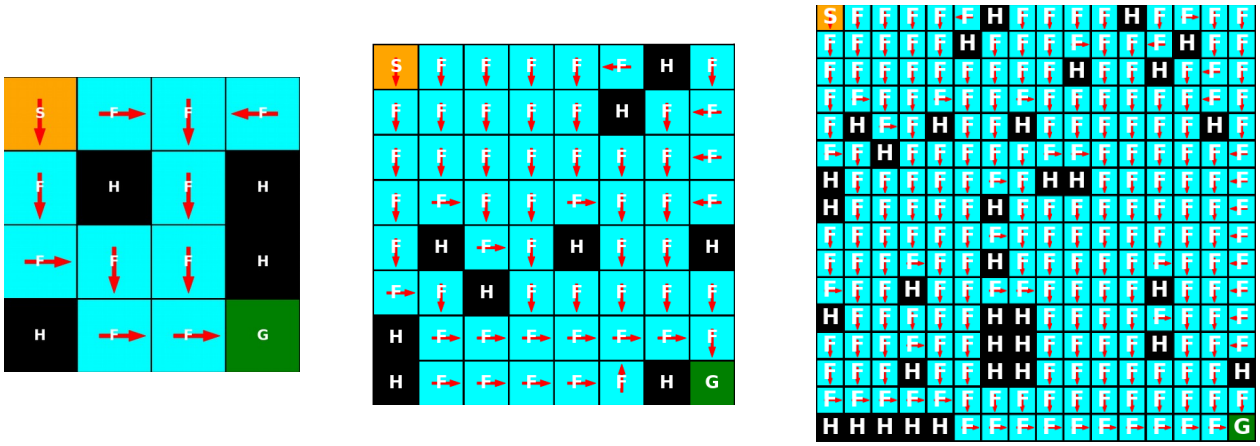


■ The running time and the number of iterations get higher while increasing the map size. It goes from *0.27 seconds* - 2 iterations, for the small instance (*16 states*), to 1.*98 seconds* - 18 iterations, for the large instance (*256 states*).

■ Comparing to the Value Iteration, Policy Iteration converges faster in terms of number of iterations (*4 iterations* for the medium instance, versus *15 iterations* for the first algorithm), but slower in terms of running time (*0.37 seconds* versus *0.12 seconds*). It is normal for the iterations of Policy Iteration to take a lot of time because in each iteration, Policy Evaluation is launched, and it is almost equivalent to an entire Value Iteration algorithm, which makes it computationally expensive. But this is also the reason why it converges in few iterations: the computation and optimization of the value function in the inner loop of Policy Iteration, guides the algorithm and makes it reach quickly the optimal policy.

■ Similarly to the Value Iteration, there is one more iteration for convergence after the reward of 1 is reached. And concerning the convergence threshold, we can chose it arbitrary just like in the previous part as long as it is small enough, thanks to the nature of Frozen Lake.

Now, let's take a look at the the final policies, resulting from the Policy Iteration algorithm:

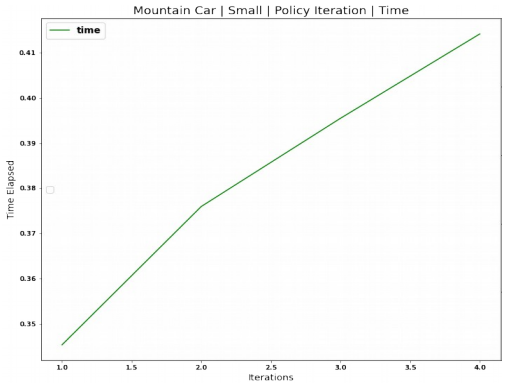


■ The policies are correct, they give the right actions to reach the goal and get the reward.

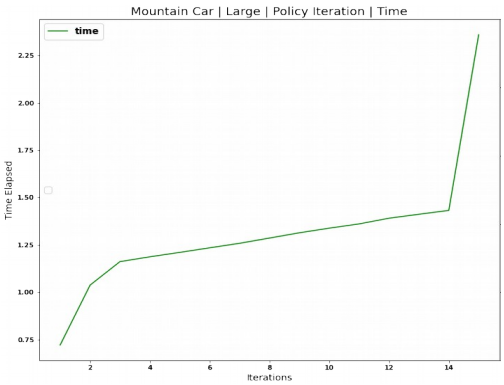
■ Value iteration and Policy iteration converge to the same answer for the Frozen Lake.

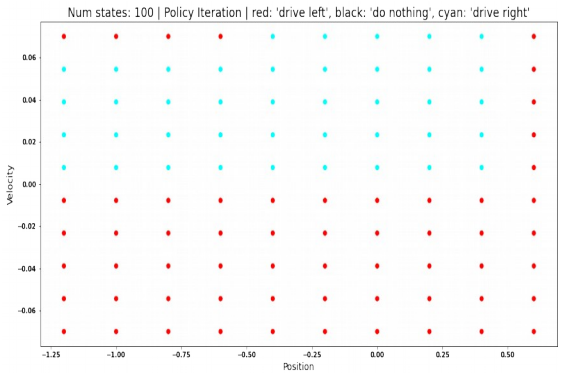
4.3. Mountain Car

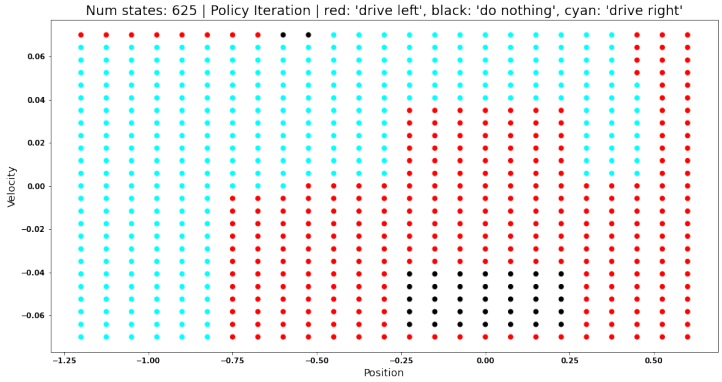
In this part, we apply Policy Iteration to the small and large instances of the Mountain Car. Here are the results found with $\gamma=0.9$, *and* 10^{-5} for the inner convergence threshold:



Average rewards on 100 episodes	
Small	-159
Large	-137







■ The running time and the number of iterations are higher for the Large instance (*625 states*).

■ For Policy Iteration, the convergence is defined by two criteria. The first one is the convergence regarding the Policy Evaluation in the inner loop (which is related to the convergence threshold), and the second one is about the policy improvement (whether the actions defined by the policy in two consecutive iterations of the algorithm are the same for each state, or not). Therefore, the choice of the convergence threshold is very important, just as in Value Iteration. It affects the running time, as the optimization of the value function in Policy Evaluation takes more time to convergence. In addition to that, the convergence threshold also affects directly the number of iterations for convergence: a smaller threshold leads the value function to more precision, which makes the policy closer to the optimal, resulting in lower number of iterations overall (for large instance, it takes 19 iterations with a value of 10^{-3}, and 27 iterations with 10^{-2}).

■ The policies are almost the same between Value Iteration (VI) and Policy Iteration (PI). There is a single dot (state) in each of the two instances which is black ('*do nothing*') in VI figures, and different in PI ('*drive left*' for small| '*drive right*' for large). One reason that can explain this difference is the fact that those states are never reached during an episode, they never participate in getting the reward and their best action comes only from the backpropagating reward.

■ Since the policies are the same, the average rewards of PI is expected to be the same as the one for VI. There is just a low difference (*-156* versus *-159*, and *-138* versus *-137*), due to the randomness of the episodes (the car starts at a random position between *-0.4* and *-0.6*).

5. Q-learning

5.1. Definition

It is a Reinforcement Learning technique used to solve MDPs and find optimal policies. The algorithm is very different from the two first ones, as it's model-free. While Value Iteration and Policy Iteration use the transition probabilities and the rewards, defining the model, Q-learning does not have access to it. It only relies on the scenarios (episodes) and the interaction with the environment to compute the value of each (state, action), in its own function Q.

5.2. Hyperparameters

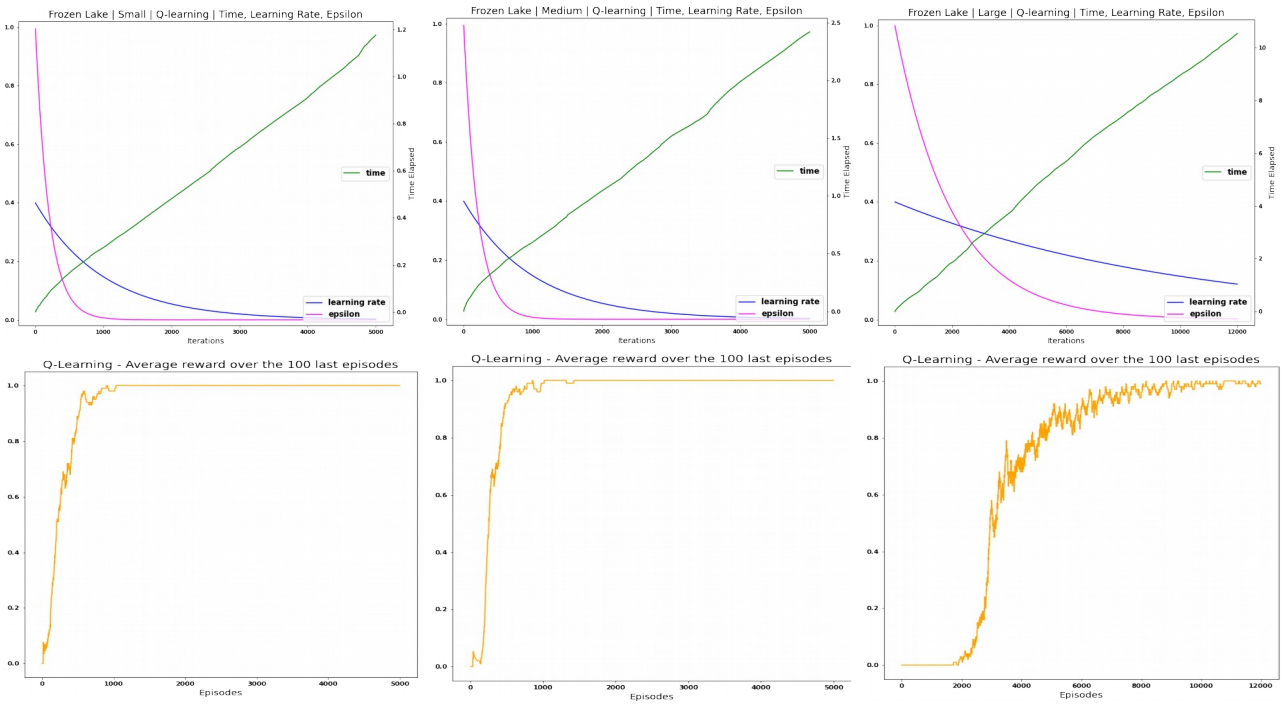
- **Learning rate (LR):** *float between 0 and 1*, representing the importance given to newly acquired information, in contrast with old information. A value of 0 makes the agent no longer learn. A value of 1 makes the agent consider only recent information, ignoring prior knowledge.

- **Discount factor:** *float between 0 and 1*, it represents the importance given to future rewards. A value of 0 makes the agent short-sighted (only considering instant rewards), while a value of 1 makes it strive for long-term rewards and may lead to a divergence of the action values.

- **Epsilon:** *float between 0 and 1*, it is the exploration factor. In our greedy-epsilon method, an agent has a probability of ε to take a random action (*exploration*), and a probability of *(1- ε)* to take the best action based on prior knowledge (exploitation).

5.3. Frozen Lake

We run Q-learning on the three maps of the Frozen Lake with a discount factor *γ=0.98*, and a total number of iterations equal to *5000* for the small/medium maps, and *12000* for the large maze. The figures below show the values, during the training process, of the learning rate (*blue*), epsilon (*magenta*), time (*green*), and the average reward over 100 last episodes (*orange*).



■ For the **discount factor γ**, we chose a value of *0.98* instead of *0.9* like in the previous two algorithms because Q-learning does not have knowledge of the environment, and as a result the agent can take a huge number of steps before getting a reward. So, he needs to be seeking for long-term rewards in order for the algorithm to work well. With *γ=0.9*, the algorithm takes more iterations to converge for small/medium maps, and does not converge for the large map.

■ We set initially the value of **epsilon** *ε* at *1*, and we use a geometric decay (*new_ε = ε * decay)* to decrease it over the iterations until it reaches 0. The main purpose is to focus on exploration in the first steps, and slowly switch to more exploitation. Because Q-learning has a model-free aspect, the agent can improve only if it gets a reward (reaches the goal), that's why it is very important to do more exploration in the beginning.

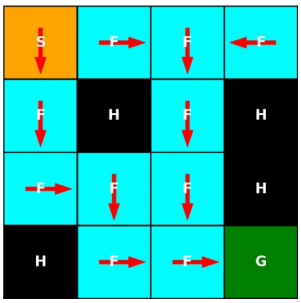
■ The initial **learning rate** is set at *0.4*, and a geometric decay is used here too to decrease its value over the iterations, with a clipping at 10^{-4} until the end of the training. The idea behind it is to learn a lot from the acquired information in the beginning, and as the time goes by, to keep benefiting from new information but with less importance so that it does not override prior knowledge, and then in the end, to completely depend on prior knowledge and stop the training.

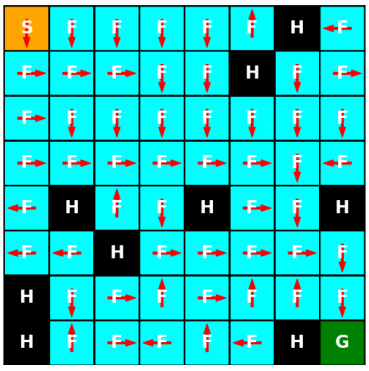
■ From the **running time** curves, the higher is the number of states, the higher is the time per iteration (*0.2 ms* for small | *0.5 ms* for medium | *0.9 ms* for large). The time per iteration remains almost constant during the training.

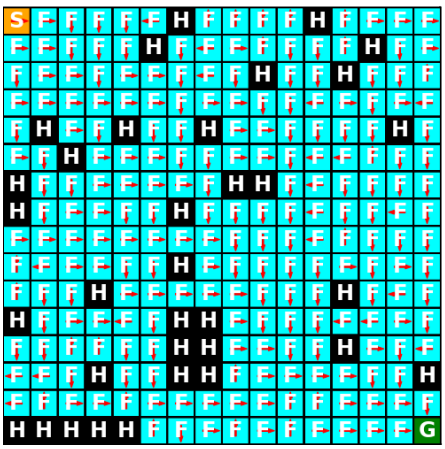
■ For the small/medium maps, we chose the **epsilon decay** in such a way that we explore 10% to 20% of the training time, it corresponded to *ε_decay = 0.995*. And we chose a **learning rate decay** in such a way that it covers the entire training time, just like described above, it corresponded to a value of *0.999*. For the large maze, those values did not work (reward remained 0 all over the training) because the problem gets more complicated with the big number of states, and the agent needs to do more exploration in order to find an episode with a reward of 1. To make it work, we increased the time of exploration (we set epsilon decay equal to *0.9995)*, we increased the number of maximum iterations (because the training is expected to last longer), and we increased the learning decay to cover all the training time (by setting it to *0.9999*).

■ From the curve of rewards (*yellow*), we can notice that the average rewards is null during the first steps, it is because the agent is in the exploration phase, and he has not played an episode with a reward of 1 yet (It takes more time as the number of states gets higher, because the probability of randomly getting the reward is lower). Then, it gets higher as the agent is improving until it finally converges as 1. This also takes more time and iterations for the maps with more states (approximately *1000* for small, *1300* for medium and *12000* for large).

Here are the final policies resulting from the Q-learning algorithm:



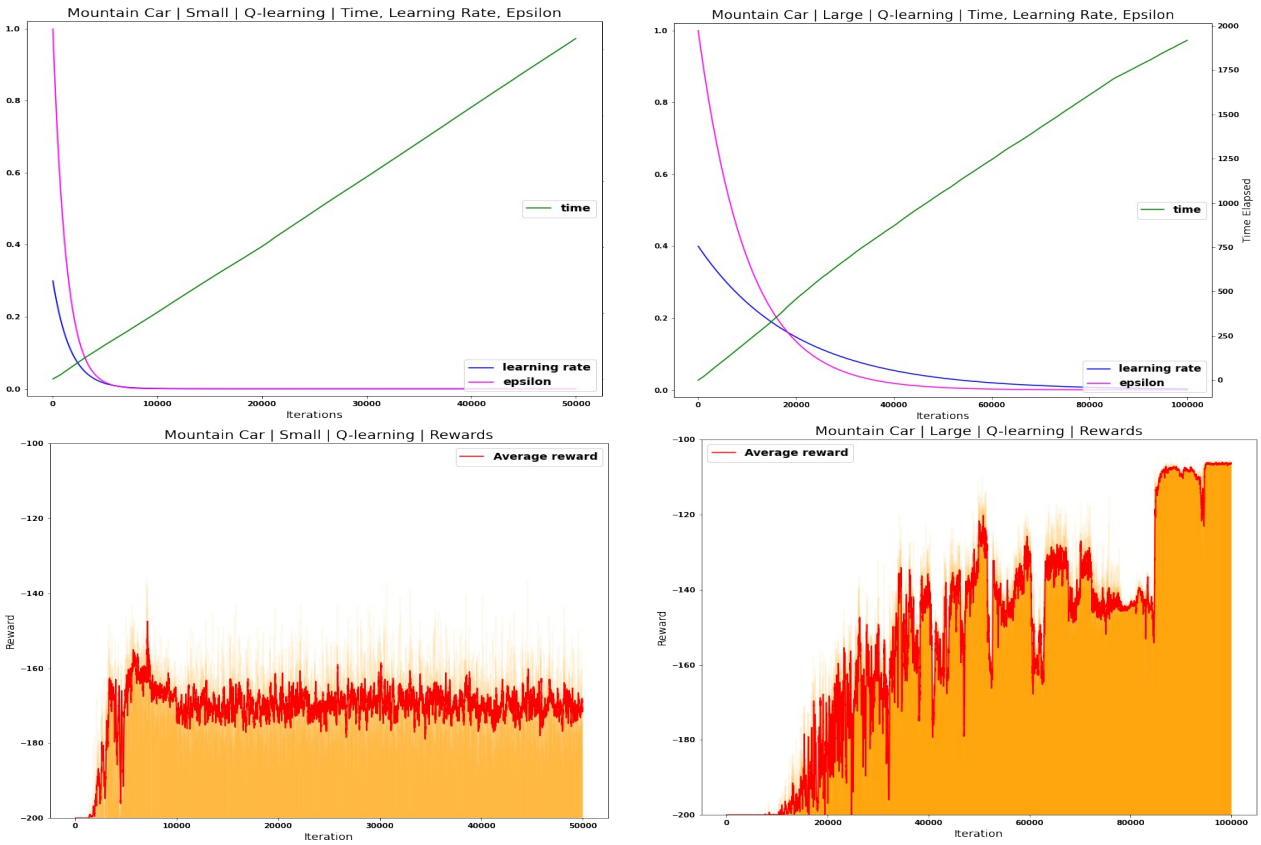




■ The policy of the small instance is the same as the one given by Value Iteration and Policy Iteration, however the policies given by Q-learning for the medium and the large maps are different from the ones found before. From the starting point to the goal, the path given by the policy is the same (it's an optimal one which gives a reward of 1). However, for some other states, the policy given by Q-learning is incorrect (For example in the medium instance, line 2, last case, the policy dictates to go right, to the wall...). This is explainable by the fact that the agent learns only when reaching a reward, and it rarely (or maybe never) happens that he goes through that state, so it did not get improved. But the policy is still optimal for our problem ($S \rightarrow G$).

5.4. Mountain Car

Now, we run Q-learning on the two instances of the Mountain Car problem with a discount factor of γ=*0.98*, and a total number of iterations equal to *100 000*. Here are the training figures:

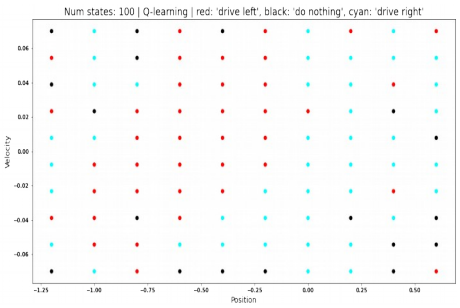


■ For these experiments, initializing our Q matrix with uniform values between -1 and +1 worked way better than an initialization with zeros, probably because the randomness helps in the exploration (it makes the exploitation also some kind of exploration in the beginning).

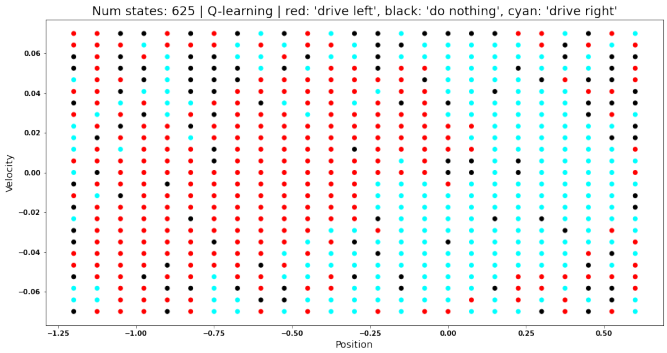
■ The training takes a lot of time compared to the Frozen Lake, or the Mountain Car using the first two algorithms: each iteration takes *0.02 seconds*. Similarly to the Frozen Lake, we used a geometric decay for the learning rate (*0.9994* for small | *0.9999* for large), and for epsilon (0.9992 for small, and *0.99995* for large). We also needed to decrease the clipping value of the learning rate to 10^{-7} for the algorithm of the large to converge.

■ The algorithm converged after *10000* iterations for the small problem. The noise in the curve of rewards is due to the relatively high learning rate (high clipping value, 10^{-4}). Decreasing it can make the training continue and get a better result, in addition to smoothing the curve.

Here are the final policies and average rewards, given by the Q-learning algorithm:



Average rewards on 100 episodes	
Small	-168
Large	-106



■ The policy representation is very different from the one found using VI and PI. By looking closely, the suggested actions are the same for the frequently visited states, which are located between (starting position, 0) and (0.5, 0.04-0.05). But for the others ones, they are different because they are uncommon to the agent, since they are rarely visited in actual episodes.

■ The average rewards for the Small instance is lower than the one for VI and PI (*-168* versus *-156*), but the one for the Large Instance is way better (*-106* versus *-138)*, this makes Q-learning the best algorithm for the Mountain Car! It is certainly thanks to the continuous nature of the problem: the Q-learning, not knowing the environment, tends to find the rules which decide of the best rewards, on its own way which is close to the reality, comparing to the model (states, rewards) on which Value Iteration and Policy Iteration are based.

6. Other experiments

- Frozen Lake - Adding a slippery factor:

If we consider the ice slippery, the agent does not always move in the intended direction. In that case, the best policy makes the agent do detours to avoid as much as possible moving closely to a hole (since there is a chance the slippy ice drives the agent to the hole). Value and Policy Iteration work just fine, however it gets more difficult for Q-learning (more exploration/iterations needed) since the randomness makes the understanding of the environment harder.

- Frozen Lake – Adding more holes:

If we choose a map with a huge number of holes, the Value Iteration and Policy Iteration work well if there is at least a possible path from the starting point S to the goal G. For Q-learning, it works too but it needs way more exploration and iterations to find rewarding episodes, from which the agent can learn and the model gets improved.

- Mountain Car – Considering 100 discrete values per dimension (10 000 states):

Increasing the size of the problem makes it difficult for the three algorithms to converge. Value Iteration converges in the end but it takes a huge number of iterations. Policy Iteration could not converge, probably because the policy is infinitely getting changed, it goes on a infinite loop. As for Q-learning, it does not improve at all, it remains stuck at an average reward of -200, because it's very unlikely to play a scenario where the car reaches the flag, even with a lot of exploration. And therefore, the agent never learns and the algorithm does not progress.