



LIÈGE université
Sciences Appliquées

INFO8003 - OPTIMAL DECISION MAKING FOR COMPLEX
PROBLEMS

ASSIGNMENT 2 - REPORT

Reinforcement learning in a continuous domain

Authors:

Pierre NAVEZ (s154062)
Antoine DEBOR (s173215)

Lecturer:

Pr. D. ERNST

T.A. :

S. AITTAHAR

B. MIFTARI

Tuesday 23rd February, 2021

1 Implementation of the domain

The Python implementation of the domain along with the policy simulation can be found in the file `section1.py` of the submitted archive.

Car on the hill problem specifications

As a reminder, the dynamics of the domain, whose state space is defined by

$$X = \{(p, s) \in \mathbb{R}^2 \mid |p| \leq 1, |s| \leq 3\}$$

and whose action space is defined by

$$U = \{4, -4\},$$

is given by

$$\dot{p} = s, \quad \dot{s} = \frac{u}{m(1 + Hill'(p)^2)} - \frac{gHill'(p)}{1 + Hill'(p)^2} - \frac{s^2Hill'(p)Hill''(p)}{1 + Hill'(p)^2}$$

with $m = 1$, $g = 9.81$ and with

$$Hill(p) = \begin{cases} p^2 + p & \text{if } p < 0 \\ \frac{p}{\sqrt{1+5p^2}} & \text{otherwise.} \end{cases}$$

In order to implement this dynamics, one exploits the Euler integration method to handle the differential equations. The integration time step is fixed to 0.001 s and the discrete-time dynamics is obtained by discretizing the time with the time between t and $t + 1$ chosen equal to 0.100 s. This leads to a number of iterations of the Euler method equal to

$$N_{iterations} = \frac{0.100}{0.001} = 100.$$

The first and second order derivatives of the *Hill* function defined here above, which are involved in the dynamics equations and must thus be implemented as well, can be analytically derived by hand as follow:

$$\begin{aligned} \text{If } p \geq 0 : Hill'(p) &= \left(\frac{p}{\sqrt{1+5p^2}} \right)' \\ &= \frac{\sqrt{1+5p^2} - p \cdot \frac{1}{2}(1+5p^2)^{-1/2} \cdot 10p}{1+5p^2} \\ &= (1+5p^2)^{-1/2} - 5p^2 \cdot (1+5p^2)^{-3/2} \\ &= (1+5p^2)^{-3/2} \\ \text{If } p < 0 : Hill'(p) &= (p^2 + p)' = 2p + 1 \end{aligned}$$

and

$$\begin{aligned} \text{If } p \geq 0 : Hill''(p) &= \left((1+5p^2)^{-3/2} \right)' \\ &= \frac{-3}{2} \cdot (1+5p^2)^{-5/2} \cdot 10p \\ &= -15p \cdot (1+5p^2)^{-5/2} \\ \text{If } p < 0 : Hill''(p) &= (2p + 1)' = 2. \end{aligned}$$

As a reminder, the reward signal is defined as

$$r(p_t, s_t, u_t) = \begin{cases} -1 & \text{if } p_{t+1} < -1 \text{ or } |s_{t+1}| > 3 \\ 1 & \text{if } p_{t+1} > 1 \text{ and } |s_{t+1}| \leq 3 \\ 0 & \text{otherwise,} \end{cases}$$

but one needs to take special care about terminal states, which are reached if $|p_{t+1}| > 1$ or $|s_{t+1}| > 3$. Indeed, for such a terminal state, all the future rewards obtained in the aftermath are zero.

Policy simulation

Two rule-based policies have been implemented: the *always accelerate* policy, which corresponds to always taking action $u = 4$, and a *random* policy, which corresponds to always drawing action u at random in U . For the sake of clarity, only the first one will be used in this part of the report. However, the second one can be easily tested as mentioned in the corresponding part of the source code.

Formally, the *always accelerate* policy $\pi_{\text{always accelerate}}$ can be described as

$$\pi_{\text{always accelerate}} : X \rightarrow U : u_t = \pi_{\text{always accelerate}}(x_t) = 4, \quad \forall t \geq 0.$$

This policy can be simulated in the domain by generating a trajectory starting at an initial state, defined by

$$p_0 \sim \mathcal{U}([-0.1, 0.1]), \quad s_0 = 0.$$

An example of trajectory is displayed in fig. 1 under the form of a sequence of tuples

$$(x_0, u_0, r_0, x_1), \dots, (x_{10}, u_{10}, r_{10}, x_{11}).$$

```
Trajectory for policy "always accelerate" :
(x_0, u_0, r_0, x_1) : ((0.06888437030500963, 0), 4, 0, (0.054873443333765634, -0.2829605383180524))
(x_1, u_1, r_1, x_2) : ((0.054873443333765634, -0.2829605383180524), 4, 0, (0.012556948990931883, -0.5668925176014514))
(x_2, u_2, r_2, x_3) : ((0.012556948990931883, -0.5668925176014514), 4, 0, (-0.0597805000170687, -0.8922561629263243))
(x_3, u_3, r_3, x_4) : ((-0.0597805000170687, -0.8922561629263243), 4, 0, (-0.16575421470543156, -1.2269004000074772))
(x_4, u_4, r_4, x_5) : ((-0.16575421470543156, -1.2269004000074772), 4, 0, (-0.30214889869377337, -1.4745297991046384))
(x_5, u_5, r_5, x_6) : ((-0.30214889869377337, -1.4745297991046384), 4, 0, (-0.4505865389033085, -1.4261480526470611))
(x_6, u_6, r_6, x_7) : ((-0.4505865389033085, -1.4261480526470611), 4, 0, (-0.5743371542527728, -0.9936041437841515))
(x_7, u_7, r_7, x_8) : ((-0.5743371542527728, -0.9936041437841515), 4, 0, (-0.6440117412221893, -0.3816824757441462))
(x_8, u_8, r_8, x_9) : ((-0.6440117412221893, -0.3816824757441462), 4, 0, (-0.6504347351457217, 0.2601388693801346))
(x_9, u_9, r_9, x_10) : ((-0.6504347351457217, 0.2601388693801346), 4, 0, (-0.5930822731721664, 0.8854348632535726))
(x_10, u_10, r_10, x_11) : ((-0.5930822731721664, 0.8854348632535726), 4, 0, (-0.47818831700494535, 1.3758218077339661))
```

Figure 1: Screenshot of a terminal showing an example of simulated trajectory for policy *always accelerate*

From this screenshot, one can see that the agent (the car) started from the initial state $(p_0, s_0) = (0.06888437030500963, 0)$ and that the action taken at every step is always $u = 4$, as defined by the policy. One can also observe that the reward perceived is always equal to zero, as the agent never meets the criteria leading to a non-zero reward as defined before. Indeed, one can see that the agent does never reach a terminal state during the generated trajectory.

However, one can simulate a trajectory for a larger number of steps, which leads to the trajectory displayed in fig. 2.

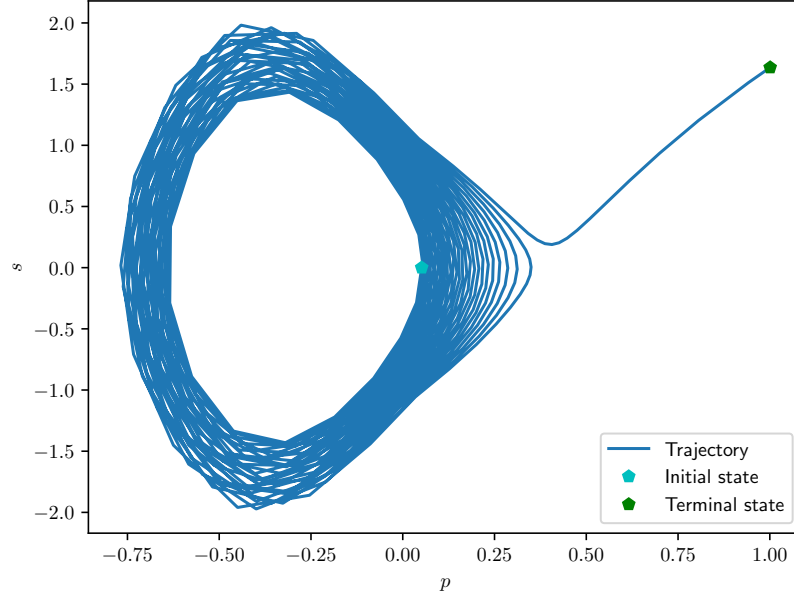


Figure 2: Simulated trajectory for policy *always accelerate*, for 5000 steps

From this figure, it is clear that, in this case, the agent reached a terminal state, denoted by a green pentagon on the plot. This corresponds to the car finally reaching the top of the hill after having moved back and forth (or from left to right on the p axis) in the cuvette in a "pendulum" fashion, always getting further to the right until eventually managing to climb to the top. The same kind of behaviour can be observed in the GIF file provided in the archive, as explained in section 3.

2 Expected return of a policy in continuous domain

The Python implementation of the routine which estimates the expected return of a policy for the *car on the hill* problem can be found in the file `section2.py` of the submitted archive.

The Monte Carlo principle

The implemented routine exploits the Monte Carlo principle, which roughly consists in repeatedly sampling a quantity of interest at random, before averaging this quantity over the drawn samples. In the considered case, one generates $M = 50$ trajectories starting from M initial states randomly drawn from the distribution specified in section 1, and following a given policy. For each trajectory $i = 1, \dots, M$, the approximated expected return $\hat{J}_{trajectory_i}^\pi$ is computed using eq. (1), where $\gamma = 0.95$ is the discount factor, N the number of transition steps of the considered trajectory and $r_{i,n}$ the reward associated with the n^{th} step of the considered trajectory.

$$\hat{J}_{trajectory_i}^\pi = \sum_{n=1}^N \gamma^n \cdot r_{i,n} \quad (1)$$

The approximated expected return of the considered policy considering all M trajectories is then computed using eq. (2), which consists in averaging the expected returns obtained at the previous step over the M trajectories.

$$\hat{J}_{Monte-Carlo}^{\pi} = \frac{1}{M} \sum_{i=1}^M \hat{J}_{trajectory_i}^{\pi} \quad (2)$$

Expected return of the policy

Again, in this part of the report, one considers the policy $\pi_{always\ accelerate}$ defined in section 1 only. For growing values of the number of steps of the drawn trajectories, one can perform the Monte carlo approach explained above with $M = 50$ and display the corresponding derived expected return for each value of this number of steps in a curve plot. For the number of steps spanning from 100 to 3000 (with a fixed step of 100), one thus obtains the curve plot of fig. 3.

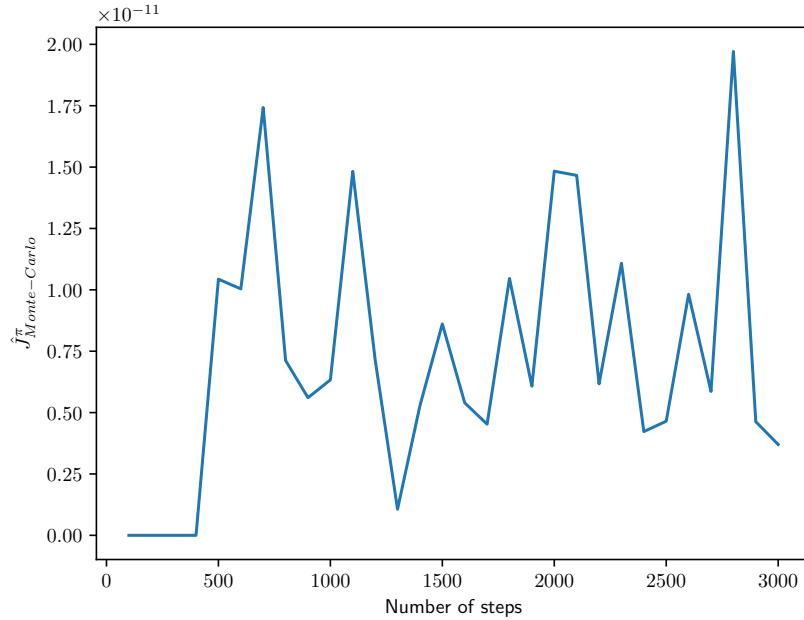


Figure 3: Estimator $\hat{J}_{Monte-Carlo}^{\pi}$, for a growing number of steps

From this curve plot, one can observe that a large enough value of the number of steps to approximate well the infinite time horizon of the policy is, for instance, $N = 500$. Indeed, for a number of steps greater or equal to this value, one can notice that the estimated expected return is different from zero, which means that the agent always reaches at least once the terminal state associated to the positive +1 reward during the Monte-Carlo process. Still, some oscillations remains, which are related to the inherent randomness of the Monte Carlo process. However, it is worth noting that these oscillations stay in the same order of magnitude, which is quite small since, when the agent reaches the terminal state, the multiplicative term γ^n is very small. Indeed, provided that this state is reached for about $N = 500$ steps, $\gamma^{500} = 7.274 \cdot 10^{-12}$.

3 Visualization

The Python routine producing a video from any *car on the hill* trajectory can be found in the file `section3.py` of the submitted archive.

Implementation

The routine is quite simple: it takes as input the considered trajectory (under the form depicted in section 1) and produces a GIF file corresponding to this very trajectory. For each state of the trajectory, the routine calls the `save_caronthehill_image` function of the `display_caronthehill.py` script¹, producing an image stored as `images/car{i}.jpeg` with `i` the index of the considered state. The *Imageio* library of Python is then used to append this image to the video, stored as a list. Finally, the same library is again used to save the video as a GIF file under the name `visualization.gif`, also provided in the submitted archive.

Observed behaviour

This video corresponds to a trajectory starting from the initial state $(p_0, s_0) = (0, 0)$ and following the policy $\pi_{always\ accelerate}$ for 800 transition steps. Similarly to what has been said regarding fig. 2, one can see in the GIF file that the car goes back and forth in the valley, while getting closer to the top of the hill as time goes by, until eventually reaching this top. The latter corresponds to a terminal state since $p = 1$ and since the car always accelerates on a now flat ground, obviously leading to $|p| > 1$ for the next steps.

4 Fitted-Q-Iteration

The Python implementation of the *Fitted-Q-Iteration* can be found in the file `section4.py` of the submitted archive.

Implementation

In this work, three supervised learning techniques are used in the *Fitted-Q-Iteration* algorithm: linear regression, extremely randomized trees and neural networks. These three techniques are used *via* their *scikit learn* library implementation, that is the `LinearRegression`, `ExtraTreesRegressor` and `MLPRegressor` functions. For the first one, all the default parameters are kept, while for the second one two parameters are changed: `n_estimators` is set to 50, such as in the *FQI* literature [1], and `n_jobs` is set to `-1` in order to speed up the computations.

The neural network structure has to be selected and motivated regarding the considered problem. As the *Car on the hill* problem depicted in the statement has a quite non-complex domain (bounded state-space and discrete binary action-space), it has been decided to use a simple architecture, made of two hidden layers of five neurons each. At first, the activation function is let by default, *i.e.* the rectifier linear unit (*ReLU*) function.

Strategies for generating sets of one-step system transitions

The *Fitted-Q-Iteration* algorithm makes use of sets of one-step system transitions to fit *SL* models. These sets can be generated following several strategies but, in the present work, only two of them are considered and used in the experiments.

The **first one** is a *random* strategy, which consists in generating a certain number M of trajectories (or episodes) following a random policy, in order to store the corresponding one-step transitions as the learning set. For each trajectory, this strategy follows the same process as in section 1, except that it uses a random policy instead of $\pi_{always\ accelerate}$. Each episode starts from the state $(p, s) = (-0.5, 0)$, *i.e.* the agent being at the bottom of the hill, and stops when a terminal state is reached. For this strategy, one has fixed $M = 1000$ episodes.

This strategy has been selected because it does not need any prior knowledge about an optimal policy, it is simple and it can be used prior to another strategy (see the second one presented here). However, it is prone to over-exploration.

¹provided in the statement (https://github.com/epochstamp/INF08003-1/blob/master/continuous_domain/display_caronthehill.py)

The **second one** is an ϵ -greedy strategy, which exploits an estimate \hat{Q}_N computed using the *random* strategy. Indeed, for a given *SL* technique, this second strategy first needs to perform the *Fitted-Q-Iteration* algorithm following that first strategy and using the same *SL* technique, in order to obtain a first estimate \hat{Q}_N . Then, a new set of one-step system transitions is generated following exactly the same methodology as in the *random* strategy, except that an ϵ -greedy policy according to \hat{Q}_N is followed by the agent at each step, with $\epsilon = 0.25$.

This strategy aims at tackling the trade-off between exploitation and exploration in reinforcement learning.

Stopping rules for the computation of the \hat{Q}_N -functions sequence

These conditions are needed to determine at which iteration N the process should be stopped. Indeed, one can define some rules to be followed in order to reach good accuracy while keeping reasonable computation time.

The **first rule** consists in fixing a maximum number of iterations N *a priori*. A way to set this value is to consider the theoretical error bound on the sub-optimality of μ_N^* with respect to μ^* in terms of number of iterations, which is given by eq. (3).

$$\|J^{\mu_N^*} - J^{\mu^*}\|_{\infty} \leq \frac{2\gamma^N B_r}{(1-\gamma)^2} \quad (3)$$

Indeed, knowing B_r , γ and given a wanted accuracy level (*i.e.* a given maximum error), one can easily derive the corresponding value of N .

Regarding the considered *Car on the hill* problem, one has $B_r = +1$ and $\gamma = 0.95$. For a maximum infinite-norm error of, *e.g.* 0.1, eq. (3) yields

$$N \geq \log_{0.95}\left(\frac{0.1 \cdot (1 - 0.95)^2}{2 \cdot 1}\right) = 175.2.$$

Therefore, the number of iterations satisfying this fixed accuracy level while leading to the shortest computation time is equal to $N = 176$. However, it has been chosen to fix the number of iterations to $N = 50$ for the following experiments, as it has been done in the paper [1].

The **second rule** could be qualified of *convergence-related*. Indeed, it consists in stopping the process when the distance between \hat{Q}_N and \hat{Q}_{N-1} becomes smaller than a certain threshold. Therefore, if the process converges, this rule provides a simple-to-check criterion to not waste computation time. However, this criterion does not make sense in all cases since, for some *SL* algorithms, the sequence of \hat{Q}_N -functions does not converge with certainty. To face this problem, the number of iteration of the algorithm has been limited to 50. The distance measure is the following

$$d(\hat{Q}_N - \hat{Q}_{N-1}) = \sum_{l=1}^{\#\mathcal{F}} \frac{(\hat{Q}_N(x_t^l, u_t^l) - \hat{Q}_{N-1}(x_t^l, u_t^l))^2}{\#\mathcal{F}}. \quad (4)$$

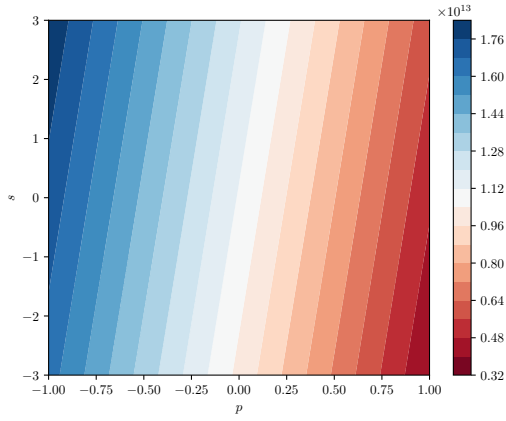
This can be referred as the mean square error. $\#\mathcal{F}$ denotes the number of samples of the data set containing the one-step transition of the system and (x_t^l, u_t^l) denotes the state-action pair l of this data set.

Results

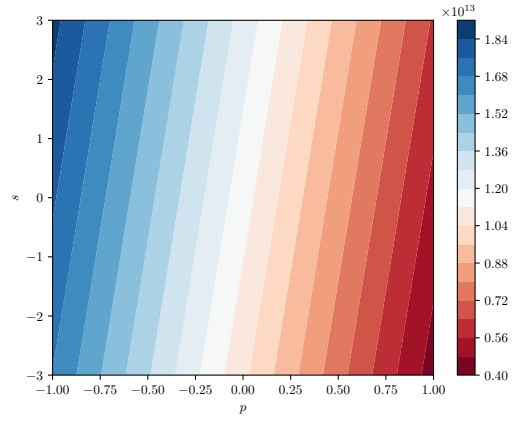
Linear regression

1. Random generation strategy

(a) Fixed N stopping rule



(a) $Q_{50}((p, s), -4)$



(b) $Q_{50}((p, s), 4)$

Figure 4: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `LinearRegression`, fixed N stopping rule

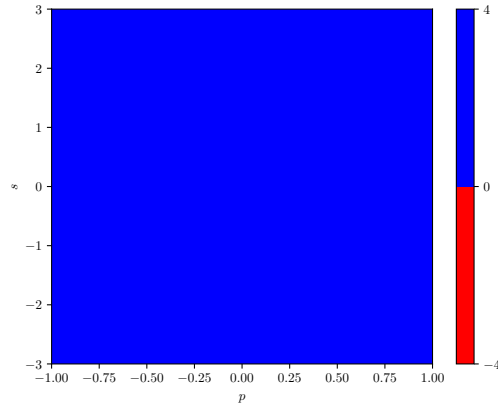


Figure 5: Optimal policy derived from Q_{50} , fixed N stopping rule and linear regression

From fig. 4b, one can evaluate the expected return for the initial state $(0, 0)$ as being between $1.12 \cdot 10^{13}$ and $1.20 \cdot 10^{13}$. As explained later in the observations, this large value does not make much sense, but it can be compared to values obtained with other strategies.

(b) Distance-related stopping rule

See the discussion about stopping rules in the following.

2. ϵ -greedy generation strategy

(a) Fixed N stopping rule

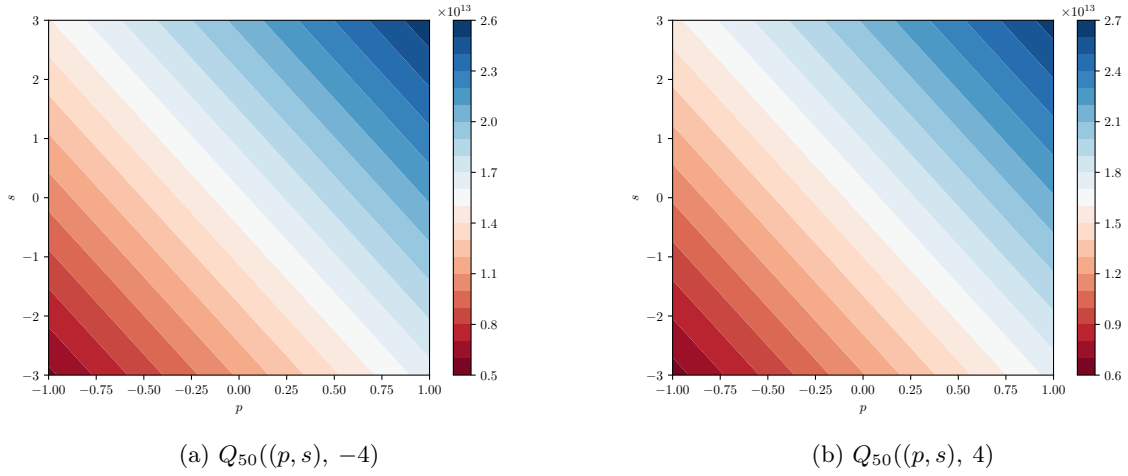


Figure 6: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `LinearRegression`, fixed N stopping rule

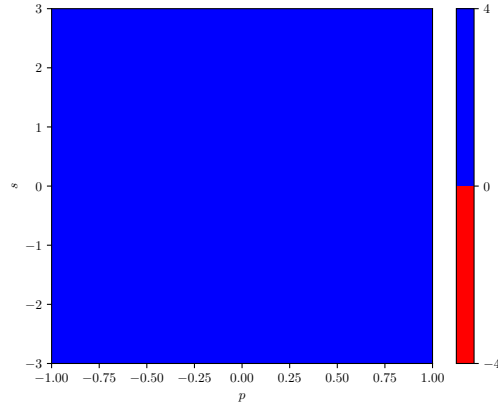


Figure 7: Optimal policy derived from Q_{50} , fixed N stopping rule and linear regression

This time, the expected return for $(0, 0)$ can be evaluated as being between $1.6 \cdot 10^{13}$ and $1.7 \cdot 10^{13}$, which is greater than the one obtained with the random generation strategy.

(b) Distance-related stopping rule

See the discussion about stopping rules in the following.

3. Observations

One can see that, for the linear regression, the different estimates Q_{50} show a linear aspect, *i.e.* their levels are linearly separated from each other. This is not surprising since the *SL* technique *is* linear, which can not lead to non linear pattern. One can also observe that $Q_{50}((p, s), 4)$ is at all points greater than $Q_{50}((p, s), -4)$ at the same points, which leads to an optimal policy equal everywhere: always accelerate. Finally, it is worth noting that the order of magnitude presented on the Q_{50} plots is quite large. This is due to the fact that, at each iteration of the *FQI* algorithm, the samples of the data set which correspond to a terminal state should correspond to a value of $Q_N = 0$. Since a *Scikit learn* model is updated in the code, there is no way to "directly" force it to output a zero for this kind of input, and this has not been handled. By lack of time, it has been decided to present these results anyway. This remark applies to the other *SL* techniques as well.

Extremely randomized trees

1. Random generation strategy

(a) Fixed N stopping rule

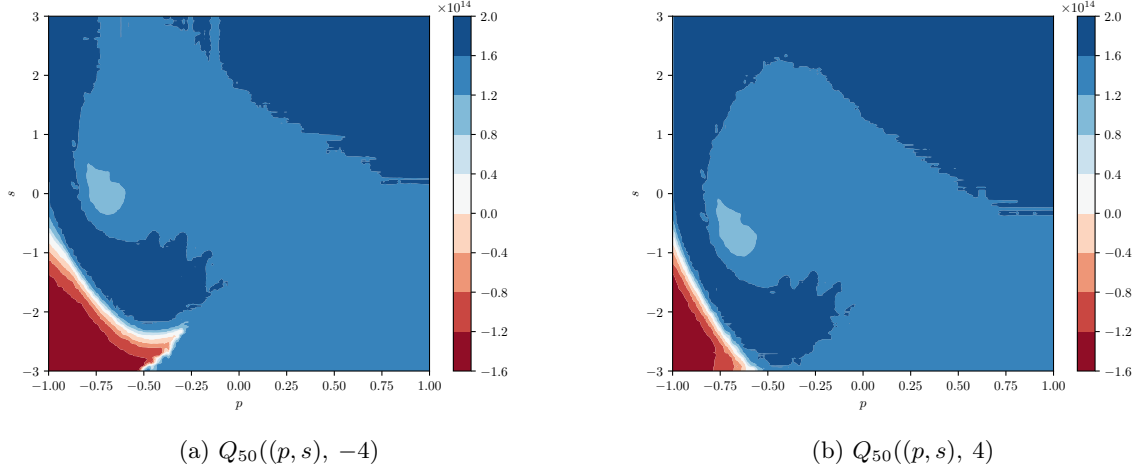


Figure 8: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s *ExtraTreesRegressor*, fixed N stopping rule

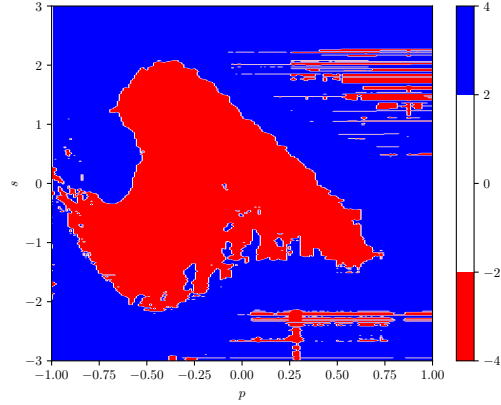


Figure 9: Optimal policy derived from Q_{50} , fixed N stopping rule and extremely randomized trees

As for the linear regression, the expected return of the optimal policy can be estimated using the plot of Q_{50} . In this case, it is evaluated between $1.2 \cdot 10^{14}$ and $1.6 \cdot 10^{14}$.

(b) Distance-related stopping rule

See the discussion about stopping rules in the following.

2. ϵ -greedy generation strategy

(a) Fixed N stopping rule

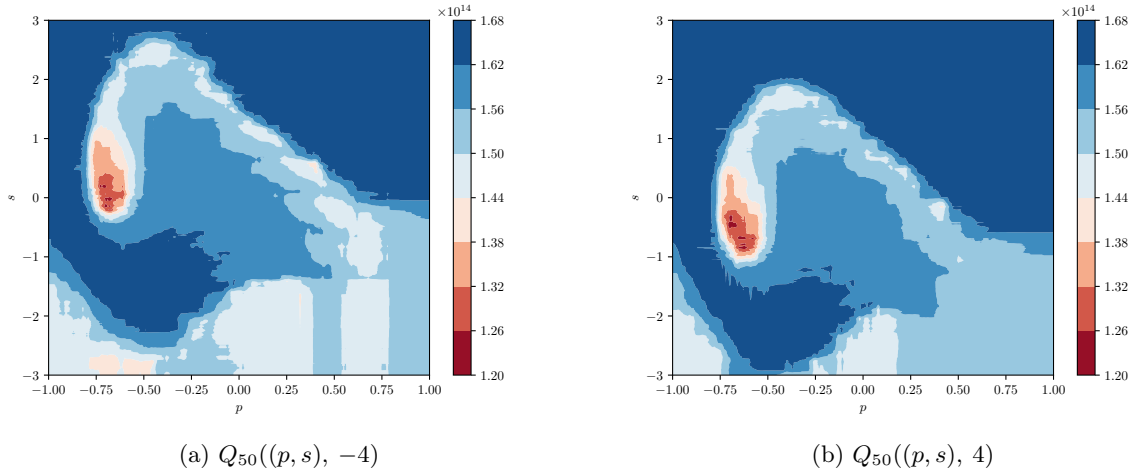


Figure 10: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s *ExtraTreesRegressor*, fixed N stopping rule

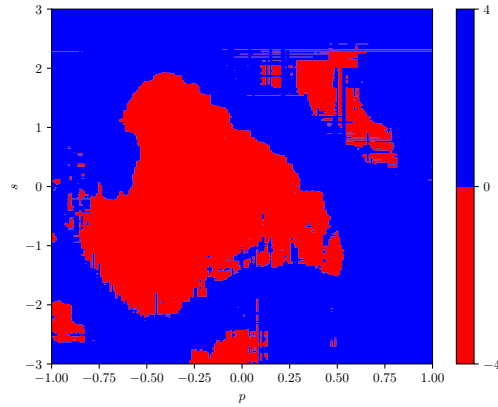


Figure 11: Optimal policy derived from Q_{50} , fixed N stopping rule and extremely randomized trees

This time, the expected return for $(0, 0)$ is evaluated between $1.56 \cdot 10^{14}$ and $1.62 \cdot 10^{14}$.

(b) Distance-related stopping rule

See the discussion about stopping rules in the following.

3. Observations

Contrary to the linear regression ones, the extra randomized trees results show non linear patterns. Indeed, one can see that the Q_{50} plots are "wave-shaped". This time though, the $Q_{50}((p, s), 4)$ is not at all points greater than $Q_{50}((p, s), -4)$. Therefore, the optimal policies show this time non constant values: for some states, the optimal action is $u = 4$ (accelerate, in blue), for some other states, it is $u = -4$ (decelerate, in red), and for some few other states, it is either the first one, either the second one (in white).

Neural networks

Experiments using neural networks in *FQI* did not lead to fine results. Indeed, while running the algorithm for the random generation strategy, *Scikit Learn* throws several warnings occurring at different iterations,

telling that the neural network optimization process did not reach convergence after `max_iter` iterations. To avoid that, several changes have been made, unfortunately without success. First, the maximal number of iterations has been increased. Then, the structure of the neural network has been modified in order to increase its complexity. Finally, the activation function has been changed from *ReLU* to the hyperbolic tangent. For this last case, the plot of the optimal policy derived at each 2 iterations has been generated, but the process has been stopped after the first warnings occurred. These results can be seen below, in fig. 12. One can see that the policy varies a lot from an iteration to another. Since the results are this bad for the random strategy, it is obviously not worth it to try the ϵ -greedy one upon these results. It has thus been abandoned.

1. Random generation strategy

(a) Fixed N stopping rule

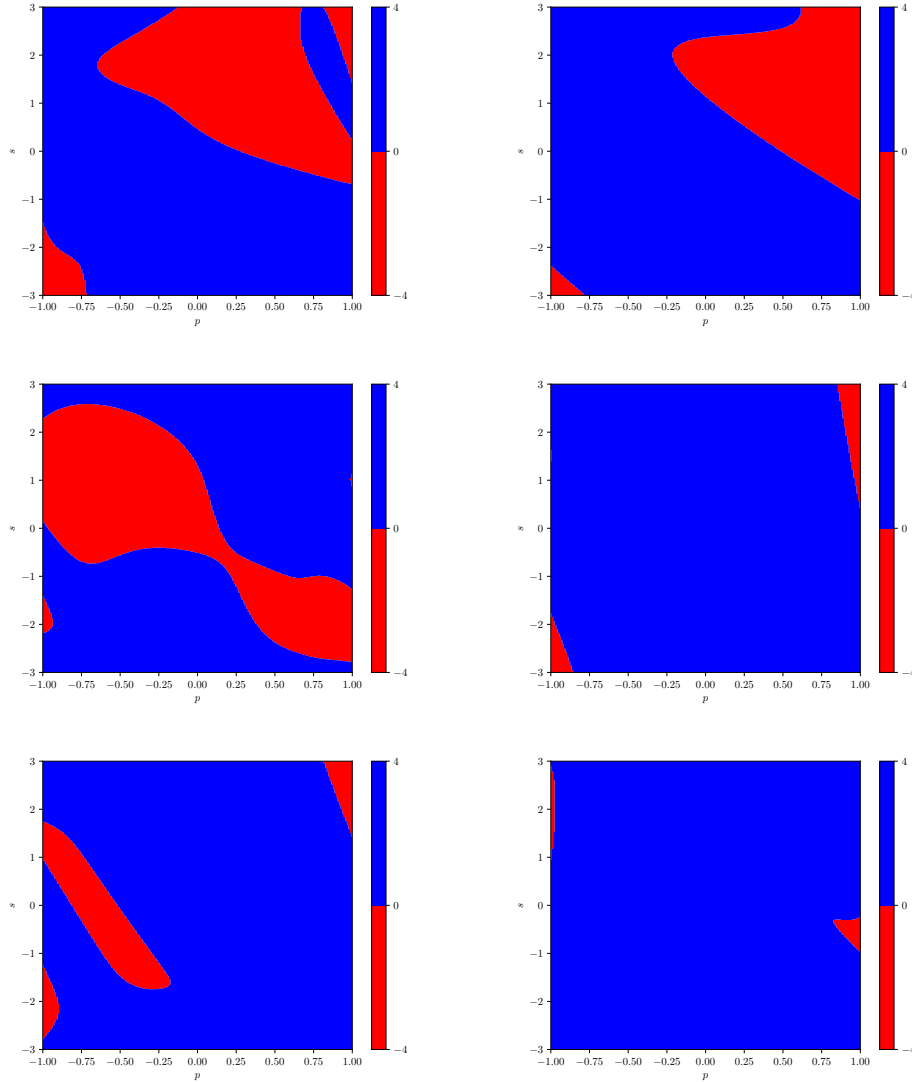


Figure 12: From left to right, top to bottom: Q_2 , Q_4 , Q_6 , Q_8 , Q_{10} and Q_{12} , with the hyperbolic tangent as activation function

(b) Distance-related stopping rule

See the discussion about stopping rules in the following.

2. ϵ -greedy generation strategy
Not performed.

Impact of the stopping rules and the one-step system transitions generation strategies

Stopping rules

Due to the fact that, by lack of time, the implementation of the algorithm has not been changed, the value of \hat{Q}_N could not be set to 0 at a terminal state. Therefore, one was not able to obtain a decreasing distance between \hat{Q}_N and \hat{Q}_{N-1} . Indeed, when the output of the model is updated at each iteration, the value of this output monotonically increases such that the new model never converges to the previous one whatever the regression algorithm used, that is why this stopping condition could not be used to obtain some results. Concerning the other stopping rule, which consists to stop the algorithm after a fixed number of iteration, it has been observed that depending on the regression model used, it is unnecessary to perform as much iteration to find the final policy. For example, the policy corresponding to the linear regression is obtained after 2 iterations and does not evolve anymore when this number increases.

Generation strategies

For *FQI* using linear regression, one can not see a great impact of the generation strategy. Indeed, even if Q_{50} is different, the derived optimal policy is still the same. This could be explained by the fact that, as the ϵ -greedy strategy aimed at exploiting the results of the random one, it does not gain any really interesting information since the random results were linear and not that expressive.

For *FQI* using extremely randomized trees, however, one can see that the *epsilon*-greedy generated model seems to refine its predictions by using information from the random-generated one. Indeed, although the "wave" shape is still present, it can be seen that more variations are showed. However, this might actually come from the color scale, which is automatically defined. Still, some reddish negative parts originally present have been replaced by positive blueish levels, and one observes that Q_{50} is now positive everywhere. For the optimal policy, one can see that the white regions have disappeared, meaning that the policy is more confident, and does not show any white "uncertainty" boundary between the blue and red parts.

References

- [1] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.