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

Friday 5th March, 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

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

and whose action space is defined by

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

is given by

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

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

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

In order to implement this dynamics, one exploits the Euler integration method to handle the differential equations. The integration time step if 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:

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

and

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

As a reminder, the reward signal is defined as

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

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 $\pi_{always\ accelerate}$ can be described as

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

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

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

```
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,(-0.059780500017068 7, -0.8922561629263243))
(x_3, u_3, r_3, x_4) : ((-0.059780500017068 7, -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,(-0.4781883170049 4535, 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.

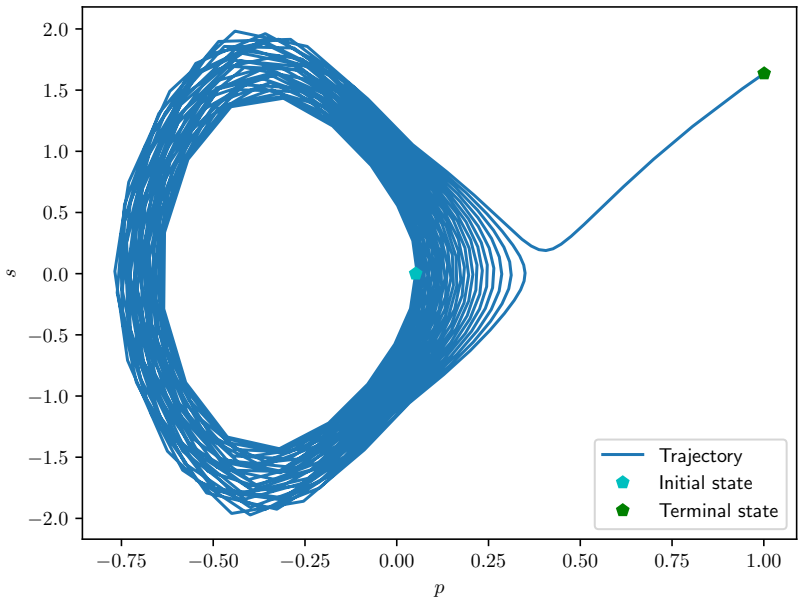


Figure 2: Simulated trajectory for policy *always accelerate*, for 5000 steps

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 $i = 1, ..., M$, the approximated expected return $\hat{J}^{\pi}_{trajectory_i}$ 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.

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

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.

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

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.

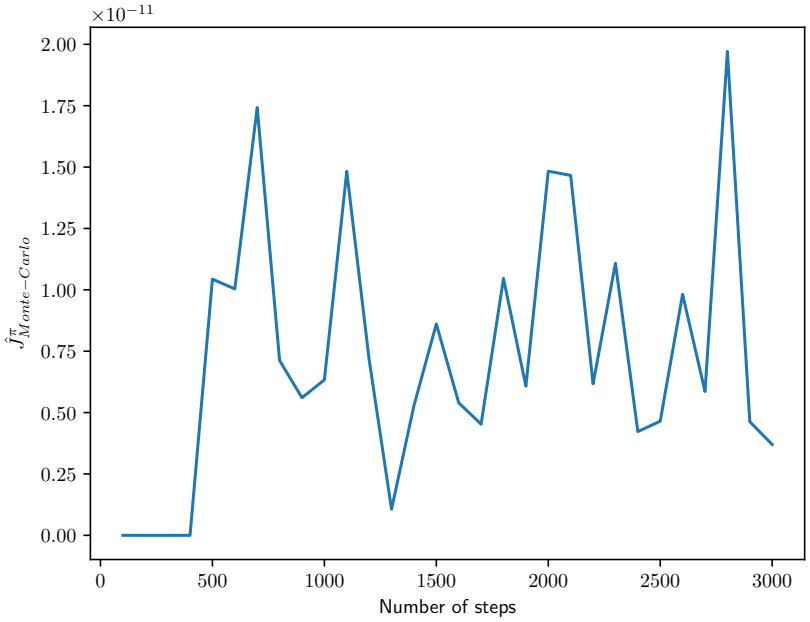


Figure 3: Estimator $\hat{J}^{\pi}_{Monte-Carlo}$, for a growing number of steps

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` script[1], 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.

[1] provided in the statement (`https:\\github.com\epochstamp\INFO8003-1\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).

$$\left\| J^{\mu^*_N} - J^{\mu^*} \right\|_\infty \leq \frac{2\gamma^N B_r}{(1-\gamma)^2} \tag{3}$$

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

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

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

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

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.

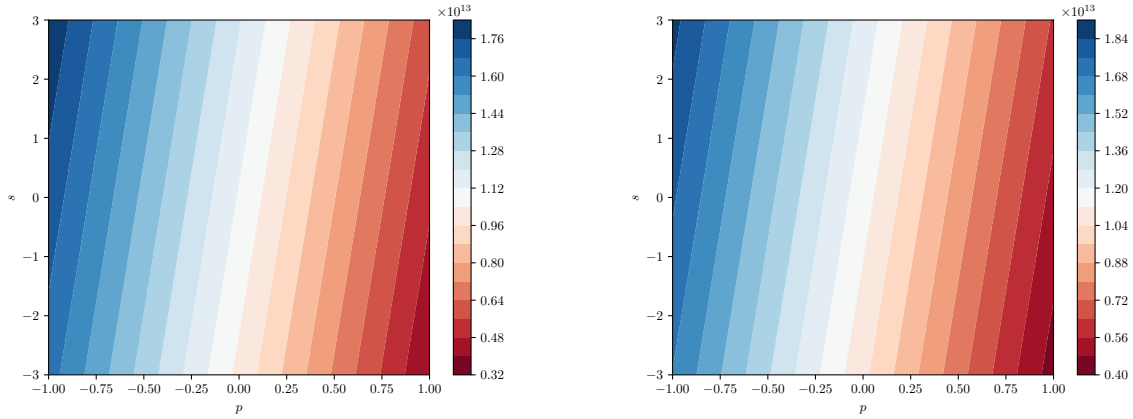
Results

N.B.: Due to an implementation error, the following results are not accurate and incomplete. Therefore, they have been generated again using the corrected code and can be seen in section 6.

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

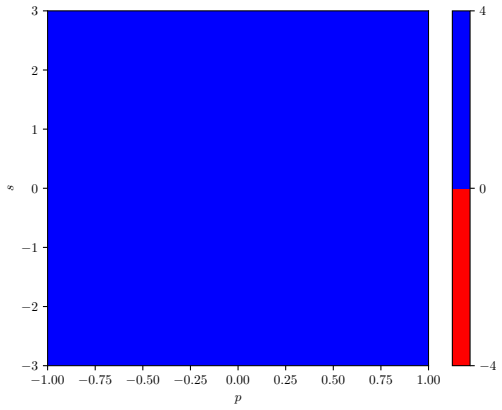


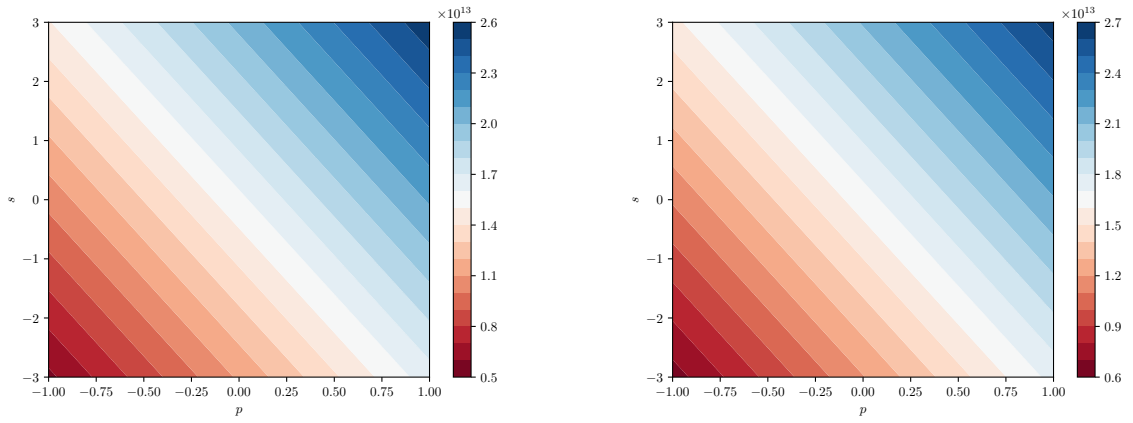
Figure 5: Optimal policy derived from Q_{50}, fixed N stopping rule and linear regression

From fig. 53b, 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



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

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

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

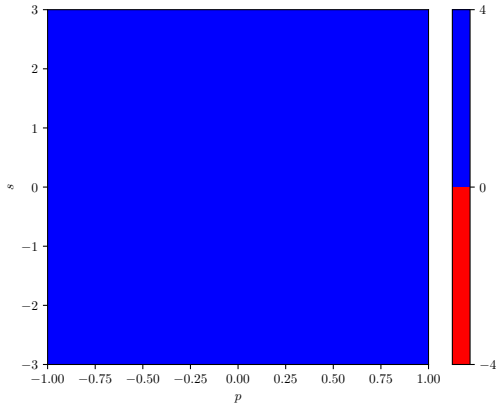


Figure 7: Optimal policy derived from Q_{50}, fixed N stopping rule and linear regression

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.

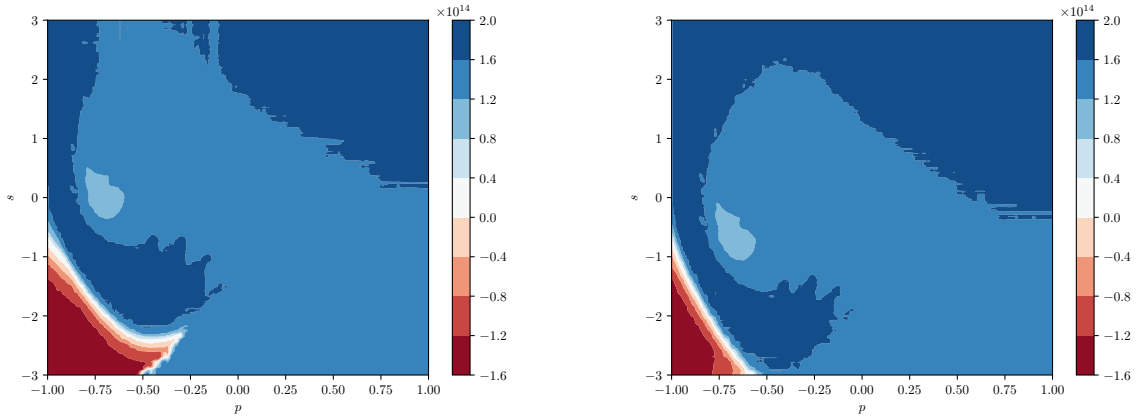
3. Observations

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



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

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

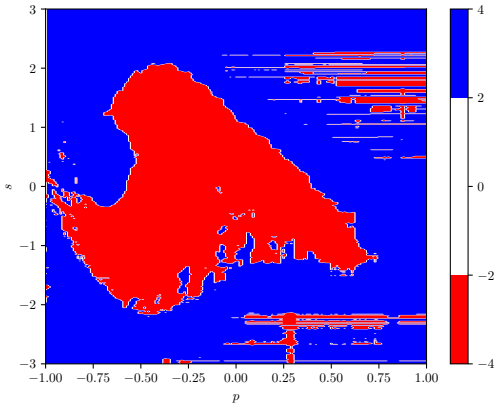


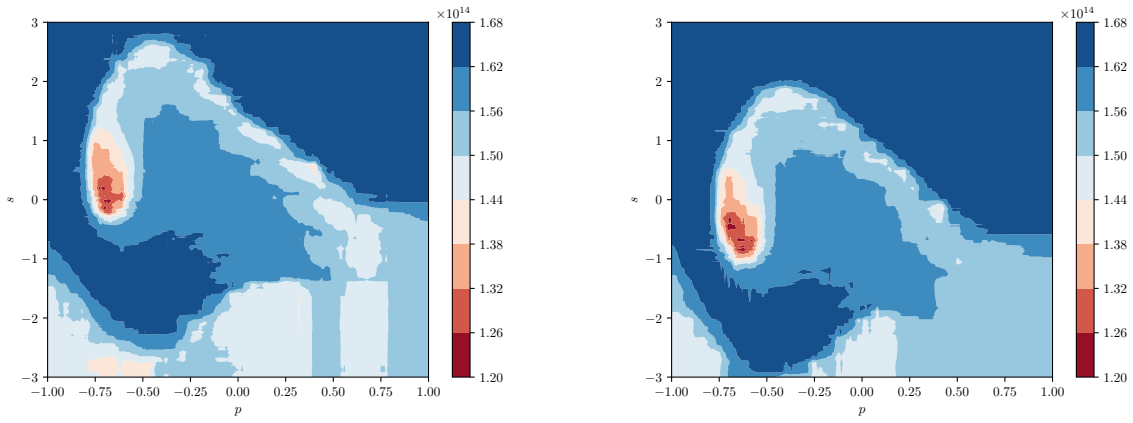
Figure 9: Optimal policy derived from Q_{50}, fixed N stopping rule and extremely randomized trees

   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



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

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

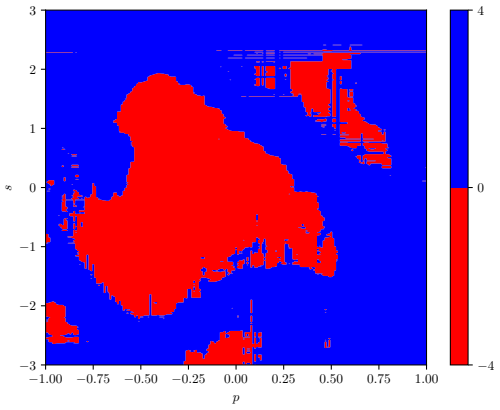


Figure 11: Optimal policy derived from Q_{50}, fixed N stopping rule and extremely randomized trees

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

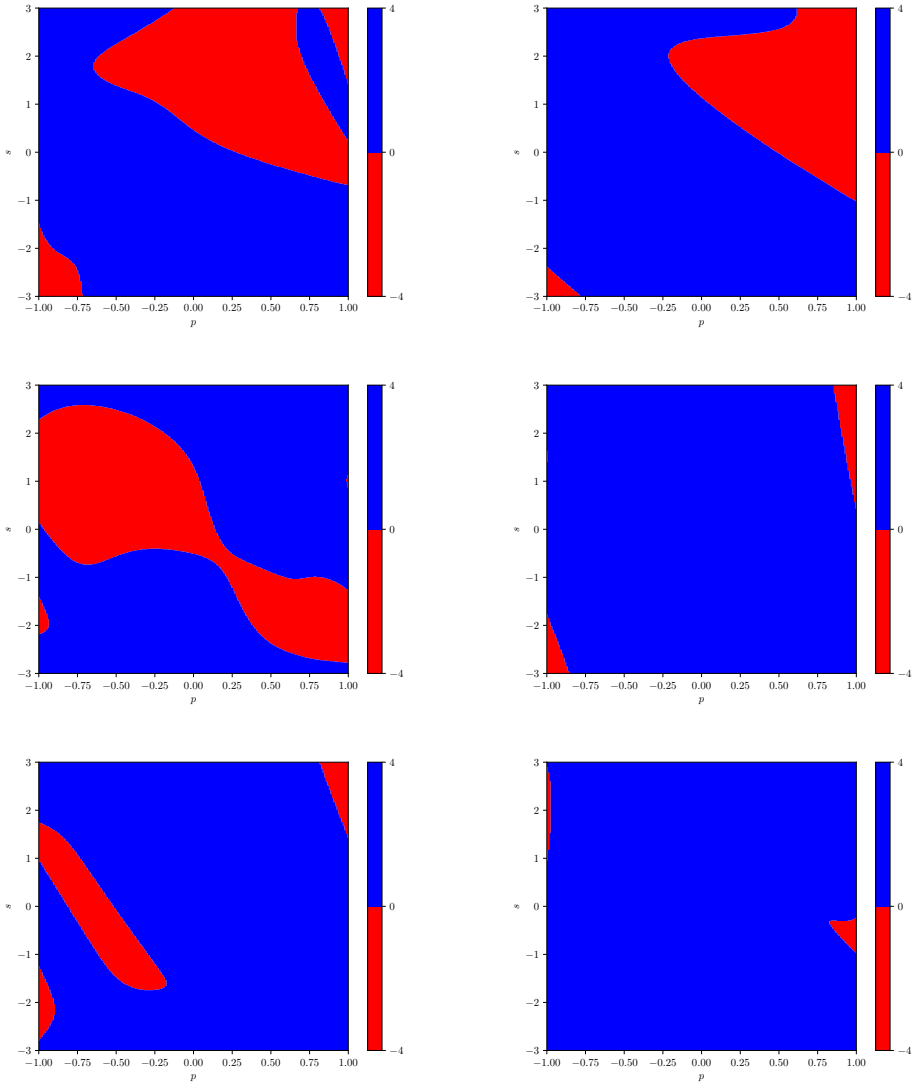


Figure 12: From left to right, top to bottom: Q_2, Q_4, Q_6, Q_8, Q_{10} and Q_{12}, with the hyperbolic tangent as activation function

   (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 $\widehat{Q}_N$ could not be set to 0 at a terminal state. Therefore, one was not able to obtain a decreasing distance between $\widehat{Q}_N$ and $\widehat{Q}_{N-1}$. 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.

5 Parametric Q-Learning

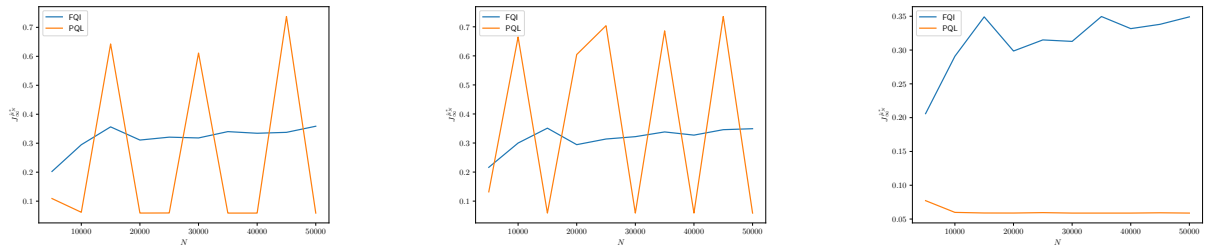
The Python implementation of the *Parametric Q-Learning* can be found in the file `section5.py` of the submitted archive.

Results for a neural network used as the approximation architecture

The neural network used in this section has been implemented using *PyTorch*, through its class *nn*. The parametric Q-Learning algorithm has then be performed exploiting the different facilities provided by this library. This algorithm uses as data set the whole set of tuples (x_t, u_t, r_t, x_{t+1}) randomly generated as previously depicted.

The parametrized approximation of the Q-function has been obtained by training a network with 3 inputs, one hidden layer composed of 5 neurons and one output, with the parametric Q-Learning algorithm. It has been decided by comparing the results obtained in FIGURE 13. These results are obtained using the experimental protocol explained in the next section. As none of the three cases were satisfying, the simplest network has been kept. The approximation $\hat{Q}((p,s),u,a)$ can be seen in FIGURE 14 and the corresponding optimal policy estimate $\hat{\mu}_*$ can be seen in FIGURE 15. The corresponding expected return is equal to 0.7641[2] which is a poor result, knowing that the one expected is 0.360 [1]. Similarly, the policy computed by the algorithm is far from optimal, as it is basically an "always decelerate" policy.

[2]The expected return is computed as explained in section 6

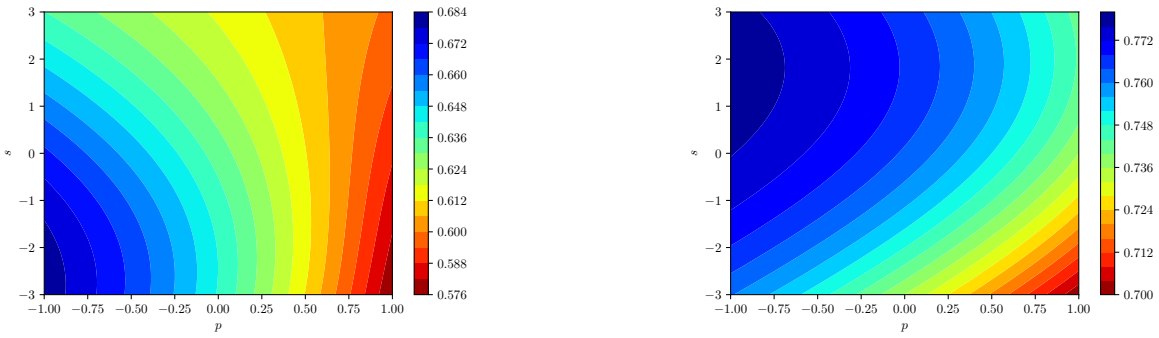


(a) 3 inputs, one hidden layer with 5 neurons, one output

(b) 3 inputs, one hidden layer with 25 neurons, one output

(c) 3 inputs, two hidden layers with 5 neurons, one output

Figure 13: Comparison of the expected returns depending on the parameters of the neural net



(a) $\hat{Q}((p,s),+4,a)$

(b) $\hat{Q}((p,s),-4,a)$

Figure 14: $\hat{Q}((p,s),u,a)$ derived from the PQL algorithm

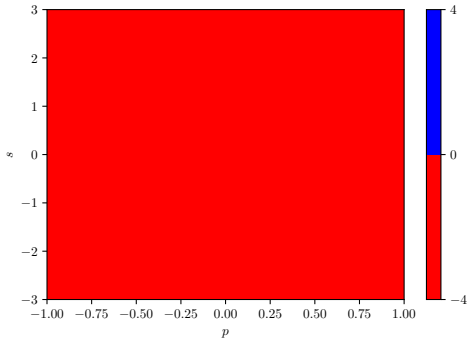


Figure 15: Optimal policy derived from the PQL algorithm

Comparison between *FQI* and *PQL*

The goal of this section is to design an experimental protocol in order to compare FQI and PQL through a curve plot, where the x-axis is the number of one-step system transitions and the y-axis is the expected return.

Experimental protocol

To reproduce the experiment, please run the following command line: `python section5.py -comp`

1. Initialize two empty vectors, aimed at storing the values of the expected return for the different numbers of steps.
2. Generate a random set of transitions by simulating M trajectories. The total amount of transitions stored in this data set must be at least equal to N_{max}, the maximum value of the number of one-step system transitions considered in the protocol. In the considered case, this data set is the same as the one previously used in section 4.
3. For N_i in $\boldsymbol{N}$, the considered range of values for the number of steps, from N_{min} up to N_{max} with step N_{step}:
   (a) Compute two estimates of the Q-function by performing FQI (using extremely randomized trees as SL technique and the fixed N stopping rule) and PQL, both using the same auxiliary data set of N_i one-step transitions *randomly drawn*[3] from the initial data set.
   (b) Derive the corresponding expected returns and store them in the two *ad hoc* vectors.
4. Display the plot curves representing the content of the two expected return vectors with respect to the number of one-step system transitions used to perform the two considered algorithms.

Results

The values used to perform the above-depicted protocol are the following: $M = 1000$, $N_{max} = 50 \cdot 10^3$, $N_{min} = 5000$ and $N_{step} = 5000$. The results are presented in fig. 16, for the one hidden layer neural network described before.

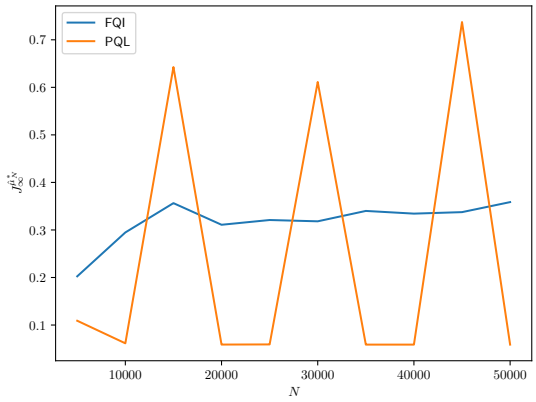


Figure 16: Comparison between FQI and PQL depending on the number of one-step system transitions

Several conclusions can be drawn from these results. First, let us motivate that the approximation $\hat{Q}_N$ of Q obtained with extremely randomized trees in FQI is the best one among the three SL techniques used in the experiments. According to [1], the sequence of approximations $\hat{Q}_N$ obtained with this technique cannot indeed diverge towards infinity and a bound on the norm of $\hat{Q}_N$ can be evaluated thanks to eq. (5).

$$\left\|\hat{Q}_N(x,u)\right\|_\infty \leq \frac{B_r}{1-\gamma}, \quad \forall N \in \mathbb{N} \tag{5}$$

[3]Drawing the tuples at random from the initial set is an arbitrary decision, though it modifies a bit the initial PQL algorithm, whose results have been presented earlier considering the tuples fed to the neural network in the same order as they had been generated. However, since PQL does not provide good results anyway and since FQI with extra trees does provide good results anyway (indeed, FQI's structure is independent of the order of the tuples in the data set), it has been decided to keep this protocol's structure.

In the considered problem, $B_r = 1$ and $\gamma = 0.95$, which leads $\left\|\hat{Q}_N(x,u)\right\|_\infty \leq 20$.

For linear regression and neural networks, one does not have this theoretical bound. Therefore, divergence can in those cases plague the FQI algorithm and lead to unwanted behaviours and results. This is the case for neural networks, while for linear regression, even if convergence happens, this algorithm does not allow to obtain non linear patterns, which, according to the best results obtained with extra trees, are the best approximation of Q. This reasoning justifies the choice of extremely randomized trees as SL technique to be used for FQI in this protocol.

Second, it is clear from fig. 16 that *Parametric Q-Learning* does not perform as good as FQI with extremely randomized trees. Indeed, PQL does not reach any convergence and does not seem to get close to the best estimate of the expected return (obtained with FQI and extremely randomized trees). This bad behaviour can be explained using several arguments, but it has been decided to highlight the two of them that have been discussed during the theoretical lectures.

The first one is related to the fact that, in *Parametric Q-Learning*, one applies a sort of gradient descent towards a changing target. Indeed, in the following classical PQL algorithm, with a the parameters of the neural network,

1. Extract the tuple $(p_t, s_t, u_t, r_t, p_{t+1}, s_{t+1})$ from the set of one-step system transitions
2. Compute $y_t = r_t + \gamma \max_{u \in U} \hat{Q}\left(p_{t+1}, s_{t+1}, u, a\right)$
3. Update $a \leftarrow a - \alpha \frac{\partial \widetilde{Q}}{\partial a}\left(p_t, s_t, u_t, a\right)\left(\hat{Q}\left(p_t, s_t, u_t, a\right) - y_t\right)$,

the second step constantly updates the target y_t used in the classical gradient descent update at the third step. Therefore, the third step corresponds to a gradient descent towards a target y_t which is not constant, and this is part of the explanation about the convergence issue that PQL is facing according to the presented results. Indeed, by definition, it does not make sense to converge towards a moving convergence target.

The second argument is related to the fact that the tuples extracted at the first step are strongly correlated. In the classical PQL algorithm, one indeed extracts these tuples in the same order as they have been generated, which eventually has a bad influence on learning. This could be explained at the light of information theory, as this strong correlation causes the mutual information to increase and the joint entropy of all one-step transitions to decrease, which has a detrimental effect in the scope of learning.

The first argument could be addressed by using a target network, as it has been suggested during the theoretical lectures, while the second one could benefit from a replay buffer. However, it has been decided to stick to the basic *Parametric Q-Learning* algorithm in this work. Though please note that, in the described protocol, the second argument does not strictly apply as the transitions are drawn at random to create each temporary data set (one for each value of the number of transitions). However, for the results presented at the beginning of this section, it applies.

Bonus : Normalized parametric Q-Learning

In this section, the comparison protocol is performed again but with a modified version of the *Parametric Q-Learning* presented above. Indeed, the term multiplied by the learning rate α in the update step of the algorithm (the third one in the above-described three-steps algorithm) is normalized. As previously mentioned, PQL is performed using the *PyTorch* library, which provides the embedded norm function `torch.norm` [4]. This function has been used with all its parameters set to their default value, which means that the norm effectively used is the Frobenius norm for matrices and the ℓ_2 norm for vectors, as it is the case in the considered case. It has been decided to let this by-default norm in order to capture all the "diversity" of the normalized term. Another norm that could have been used in the so-called infinite norm, in order to scale

[4] Note that this function is deprecated and may be removed in a future PyTorch release.

the normalized term's values between 0 and 1 in absolute value. Please though note that in the previous version of the PQL, a normalization term was already used in the implementation but on the gradient value only, rather than on the whole term multiplied by α. This normalization term, using `torch.norm` too, aimed at preventing the algorithm from numerical divergence in the case of very large or very small values of the gradient's elements.

The results of the protocol are presented below in fig. 17 and can be retrieved using the same *Python* code as before, but taking care to uncomment the dedicated lines in the `PQL` function.

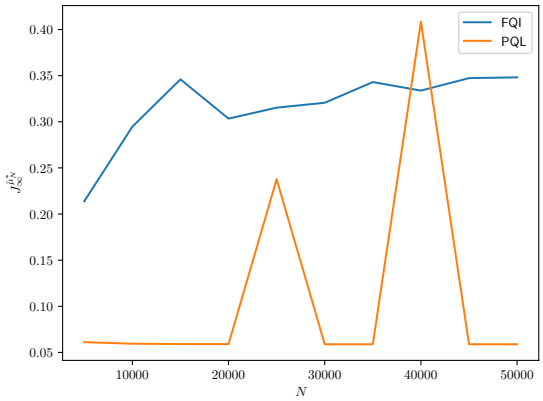


Figure 17: Comparison between FQI and the normalized PQL

One could have expected the normalization term introduced in this section to bring more stability to the algorithm, as it reduces the magnitude of the update term and thus leads to a "softer" learning. However, given the poor results obtained before with the un-normalized version, it is not surprising to observe again unsatisfactory performance with respect to FQI with extremely randomized trees. One can see that the normalized PQL does not converge to a value close to 0.35 and provide approximations that under estimate the expected return.

6 Fitted-Q-Iteration - Erratum

The results previously presented in section 4 were not accurate. Indeed, a mistake in the code has been identified as the source of the unexpected orders of magnitude present in the different figures. The issue lied in the way terminal states were handled during FQI iterations. Moreover, fixing this mistake allows to perform the second stopping-rule, which had no sense with the previously obtained results, and to obtain results using a neural network without *Scikit Learn* warnings about convergence. Therefore, it has been decided to generate the different plots again and to present them in this erratum section. The corresponding *Python* implementation can be found in the function `FQI` of the file `section5.py` of the submitted archive. *N.B.*: For the distance-related stopping rule, the critical distance has been arbitrarily chosen equal to 10^{-5}.

Computation of the expected return

The method used to approximate the expected return of the policies derived from the `FQI` is the one found in the paper of Ernst et al. [1]. It consists in the following steps: first, the state space is divided into a gird of 17×17 boxes such that the state space now contains 289 discrete states

$$X^i : \{(p, s) : i \in \{-8, -7, ..., 7, 8\}, j \in \{-8, -7, ..., 7, 8\}, p = 0.125 \times i, s = 0.375 \times j\}$$

Then, at each iteration of the `FQI` algorithm, the expected return of the optimal policy derived from the approximated Q function is computed for every discrete state and averaged by the number of states. Math-

ematically, one has

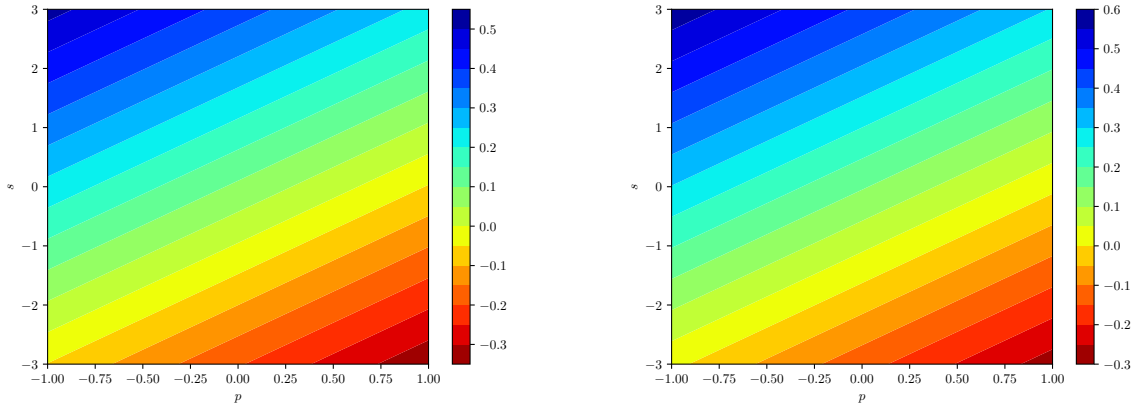
$$J_{\infty}^{\hat{\mu}_N^*} = \frac{\sum_{X^i} J_{\infty}^{\hat{\mu}_N^*}(x)}{\#X^i}$$

Finally, before discussing the results, it worth mentioning that the optimal expected return is 0.360 [1].

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

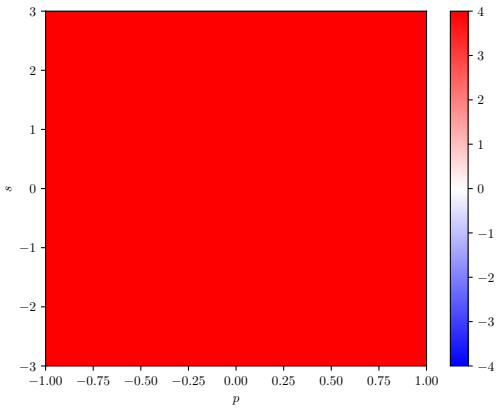


Figure 19: Optimal policy derived from Q_{50}, fixed N stopping rule and linear regression

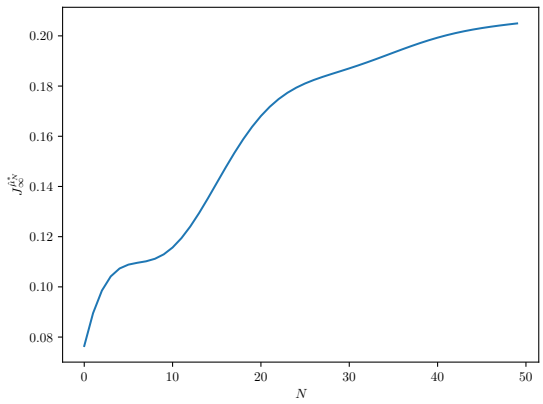
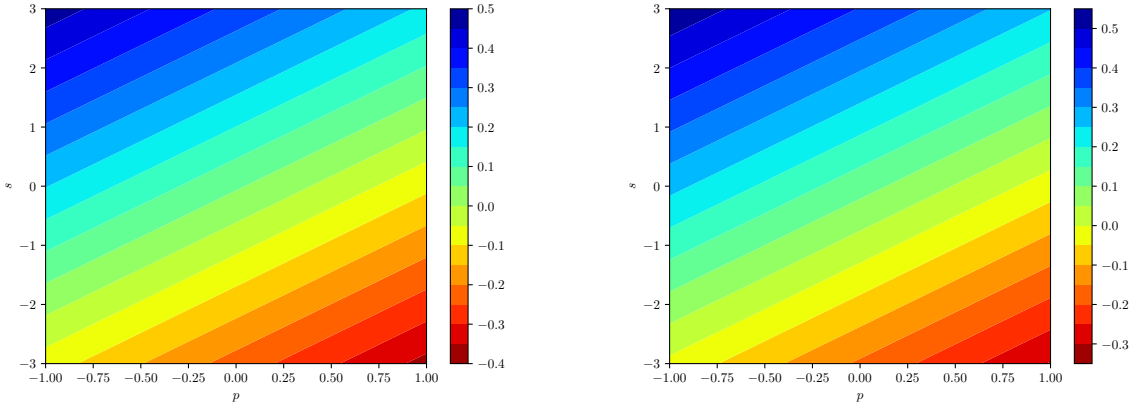


Figure 20: Expected return of the optimal policy

(b) Distance-related stopping rule



(a) $Q_{29}((p, s), -4)$ (b) $Q_{29}((p, s), 4)$

Figure 21: $Q_{29}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `LinearRegression`, distance-related stopping rule

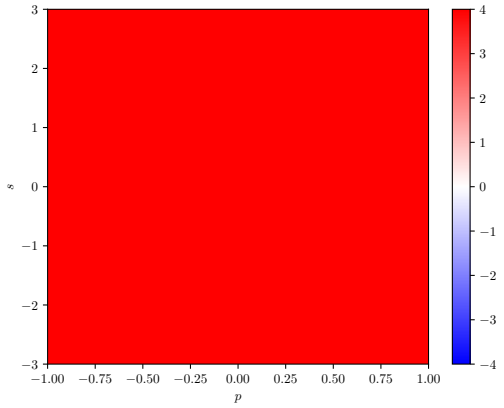


Figure 22: Optimal policy derived from Q_{29}, distance-related stopping rule and linear regression

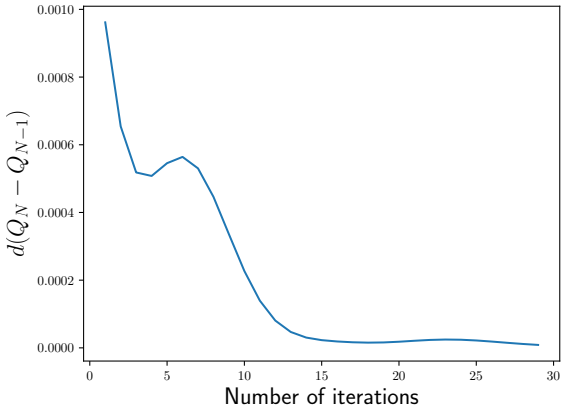


Figure 23: Distance-related convergence

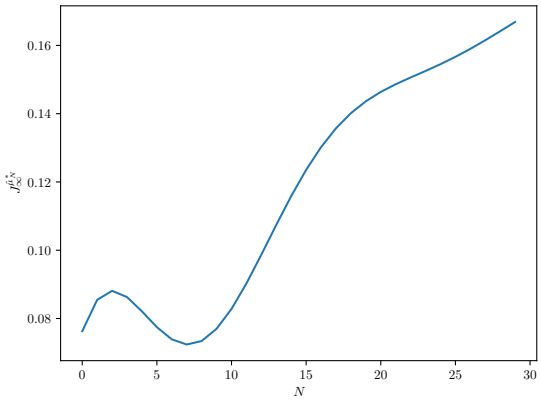
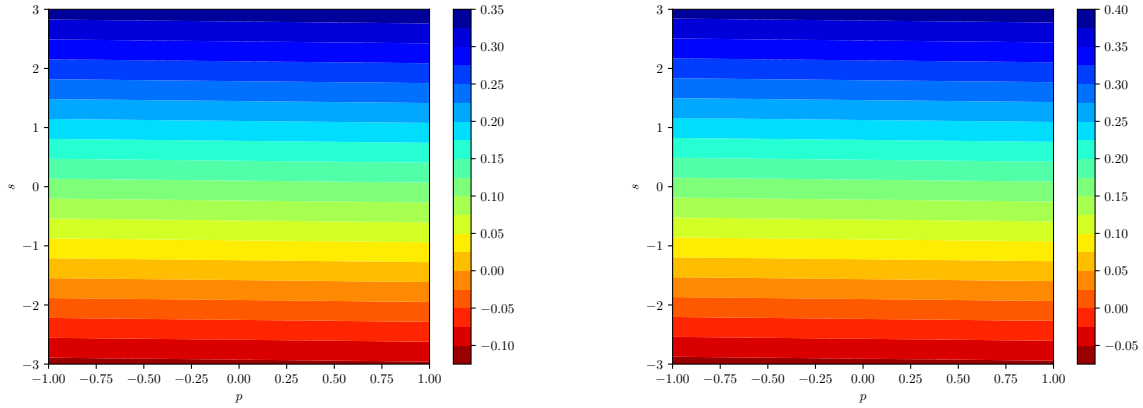


Figure 24: Expected return of the optimal policy, distance-related

2. ϵ-greedy generation strategy

   (a) Fixed N stopping rule



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

Figure 25: $Q_{50}((p, s),\ u)$ obtained with *FQI* and *Scikit learn*'s `LinearRegression`, fixed N stopping rule

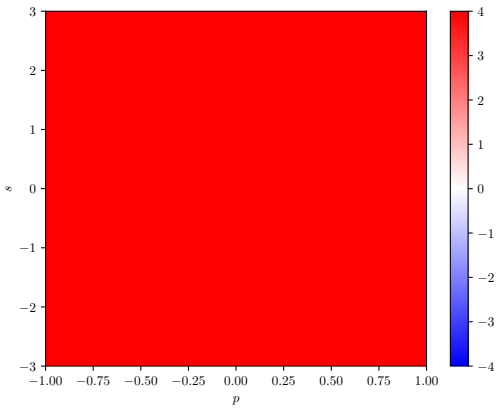


Figure 26: Optimal policy derived from Q_{50}, fixed N stopping rule and linear regression

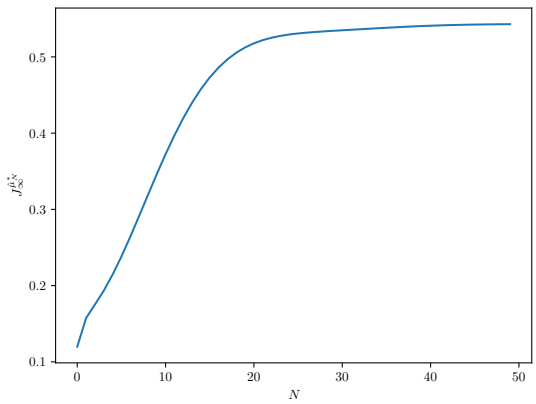
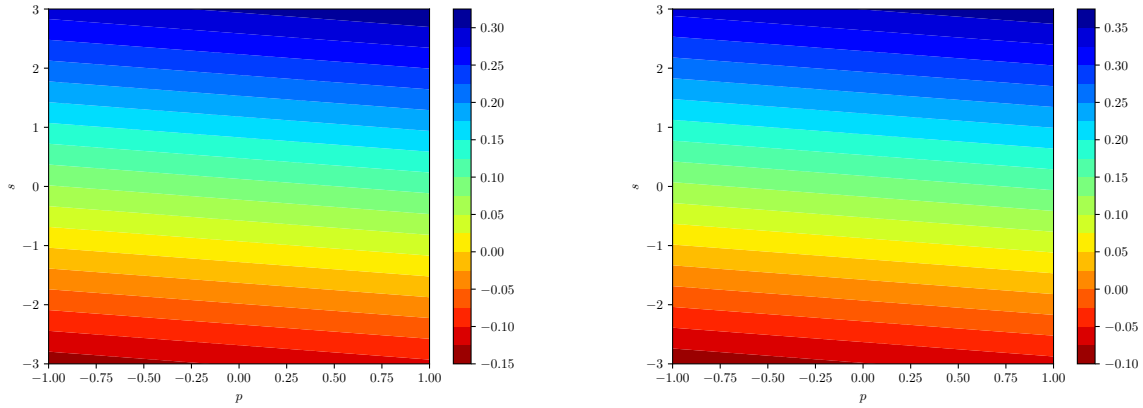


Figure 27: Expected return of the optimal policy, ϵ-greedy exploraration

(b) Distance-related stopping rule



(a) $Q_{23}((p, s), -4)$ (b) $Q_{23}((p, s), 4)$

Figure 28: $Q_{23}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `LinearRegression`, distance-related stopping rule

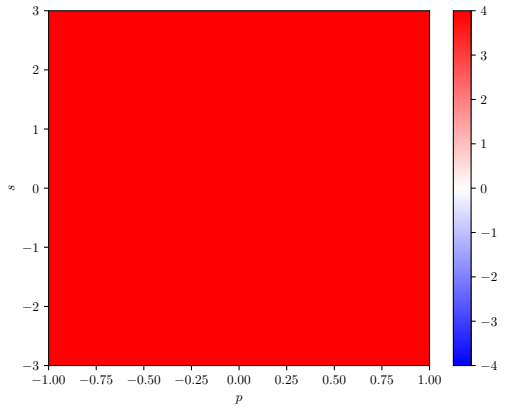


Figure 29: Optimal policy derived from Q_{23}, distance-related stopping rule and linear regression

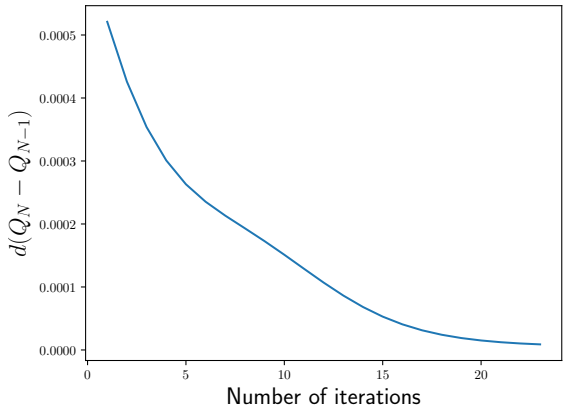


Figure 30: Distance-related convergence

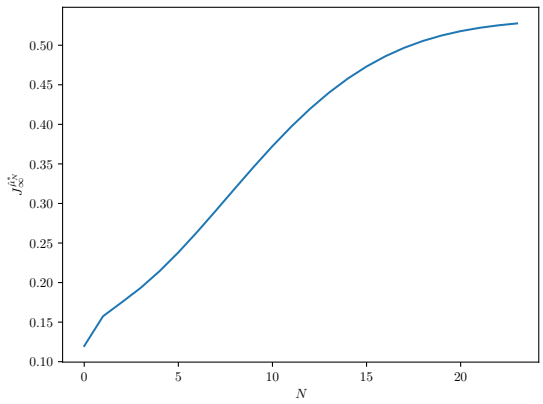
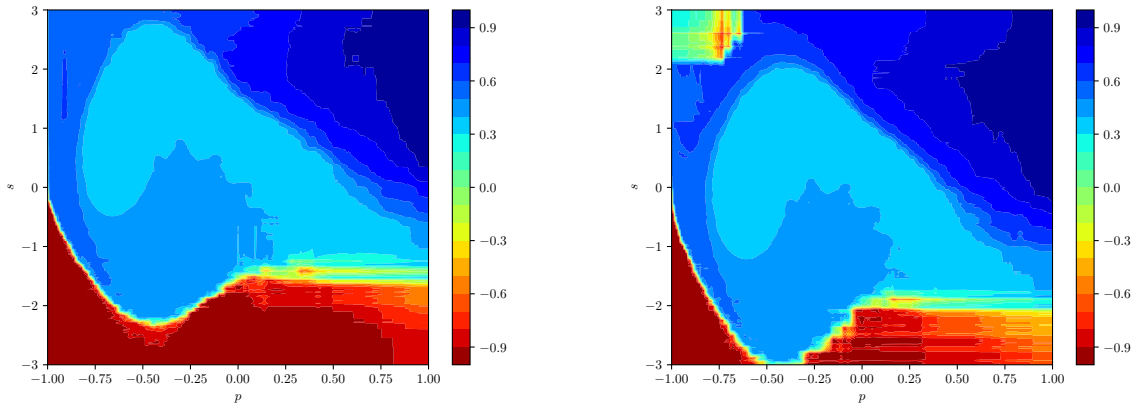


Figure 31: Expected return of the optimal policy, ϵ-greedy exploration, distance-related

Extremely randomized trees

1. Random generation strategy

   (a) Fixed N stopping rule



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

Figure 32: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `ExtraTreesRegressor`, fixed N stopping rule

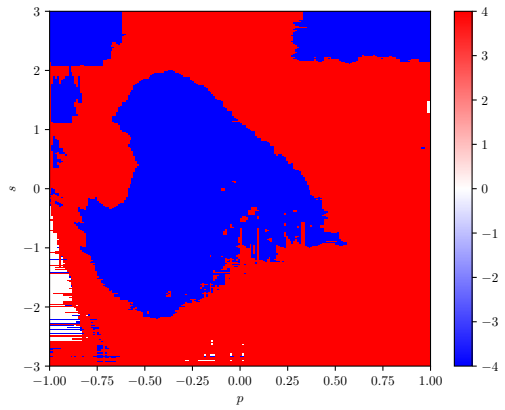


Figure 33: Optimal policy derived from Q_{50}, fixed N stopping rule and extremely randomized trees

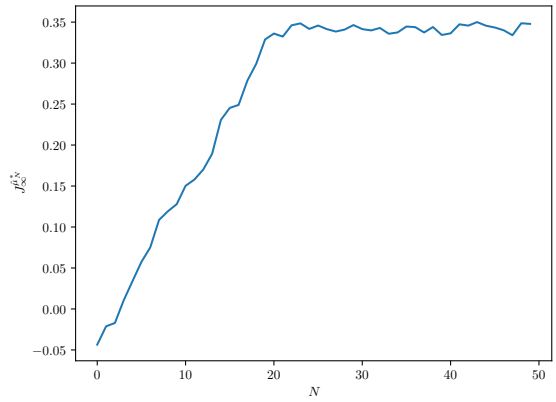
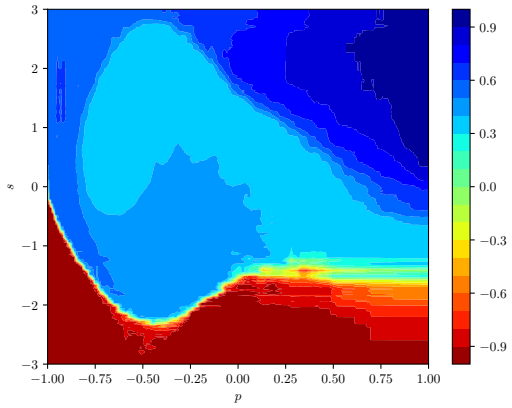
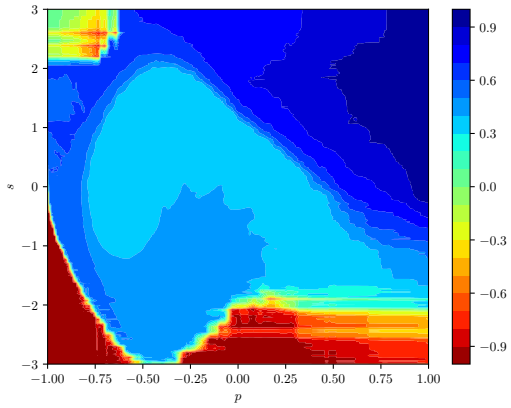


Figure 34: Expected return of the optimal policy

(b) Distance-related stopping rule



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



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

Figure 35: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `ExtraTreesRegressor`, distance-related stopping rule

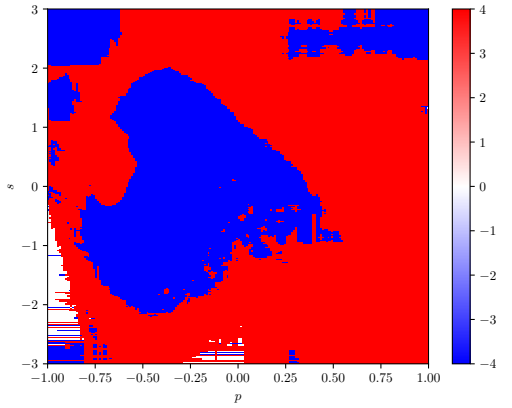


Figure 36: Optimal policy derived from Q_{50}, distance-related stopping rule and extremely randomized trees

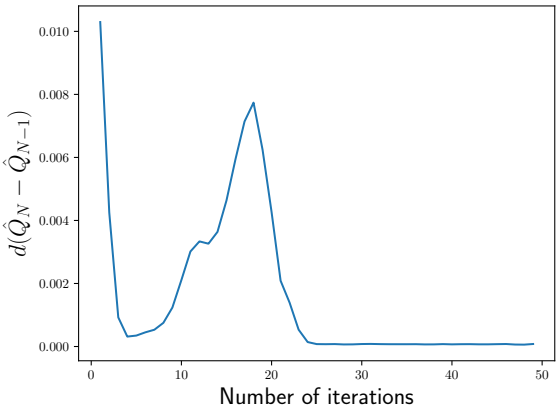


Figure 37: Distance-related convergence

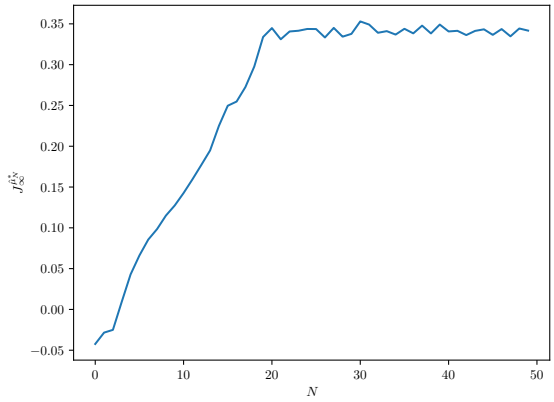
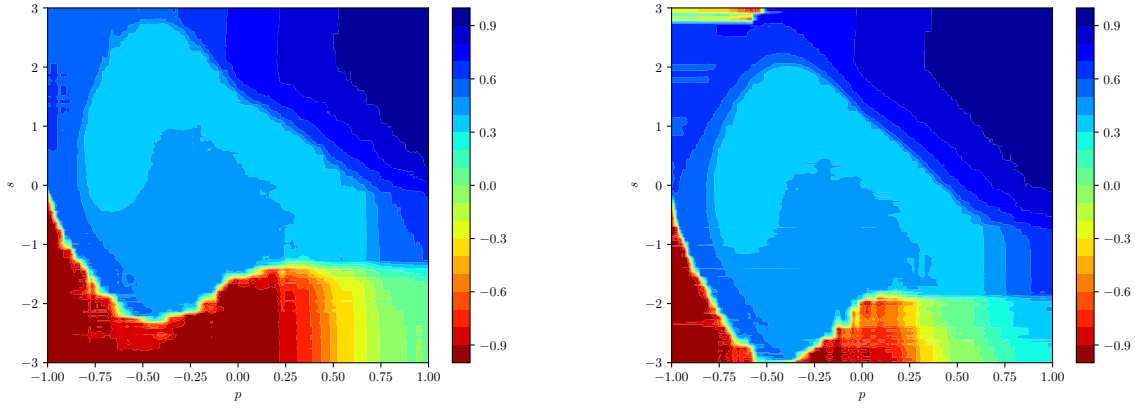


Figure 38: Expected return of the optimal policy, distance-related

2. ϵ-greedy generation strategy

   (a) Fixed N stopping rule



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

Figure 39: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `ExtraTreesRegressor`, fixed N stopping rule

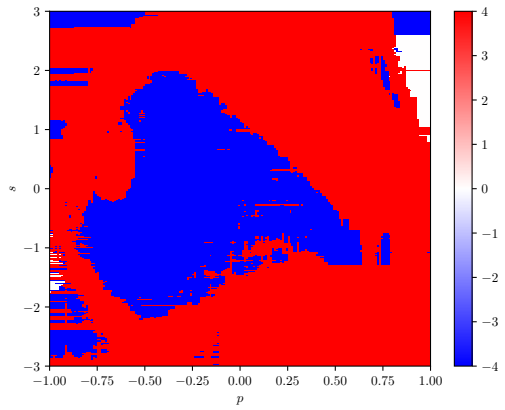


Figure 40: Optimal policy derived from Q_{50}, fixed N stopping rule and extremely randomized trees

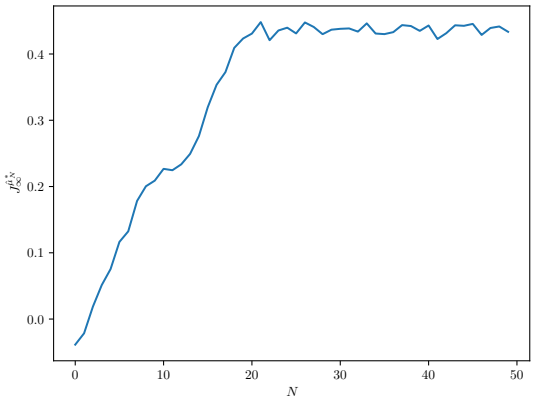
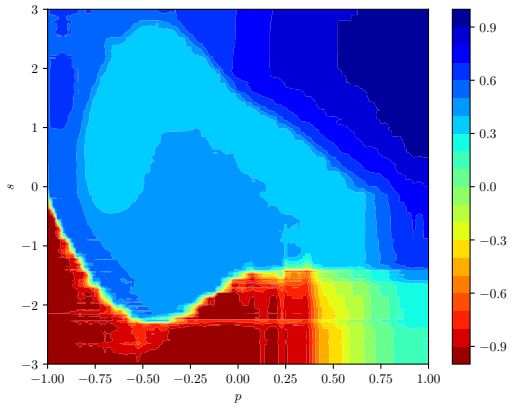
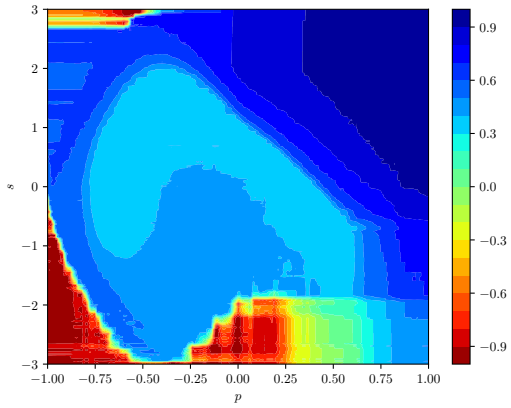


Figure 41: Expected return of the optimal policy, ϵ-greedy exploration

(b) Distance-related stopping rule



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



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

Figure 42: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `ExtraTreesRegressor`, distance-related stopping rule

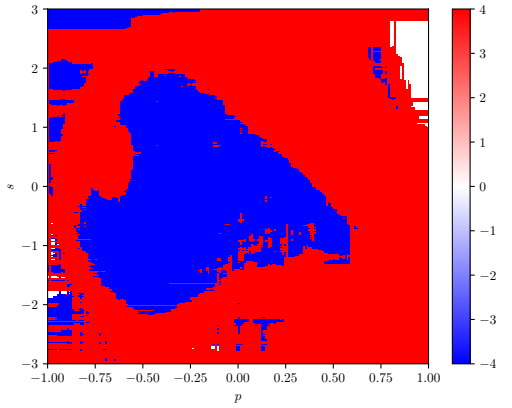


Figure 43: Optimal policy derived from Q_{50}, distance-related stopping rule and extremely randomized trees

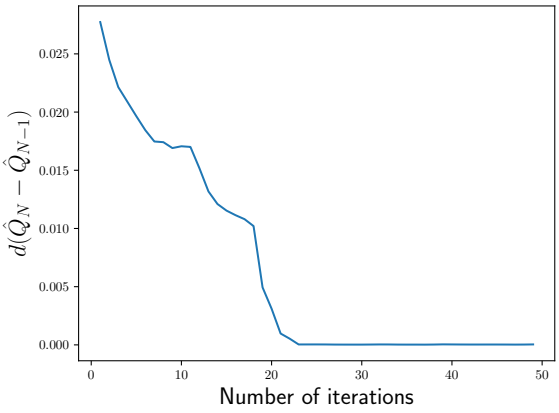


Figure 44: Distance-related convergence

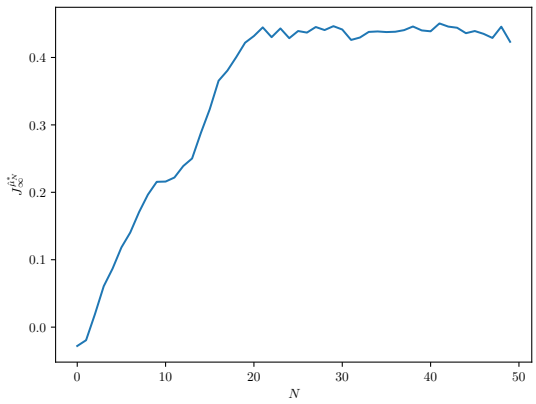


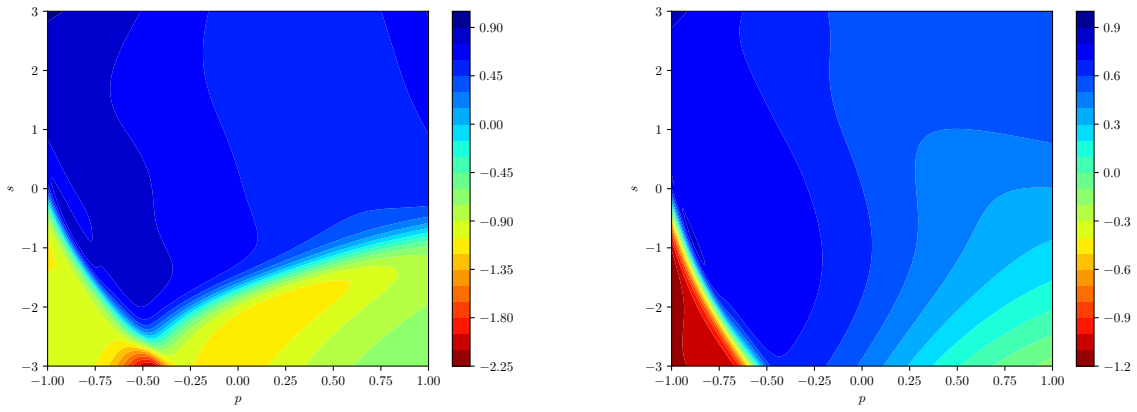
Figure 45: Expected return of the optimal policy, distance-related, ϵ-greedy exploration

Neural networks

The neural network used in this section is composed of two layers of five neurons each, with the default activation function of the `MLPRegressor` from *Scikit Learn*, *i.e.* the ReLU function.

1. Random generation strategy

   (a) Fixed N stopping rule



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

Figure 46: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `MLPRegressor`, fixed N stopping rule

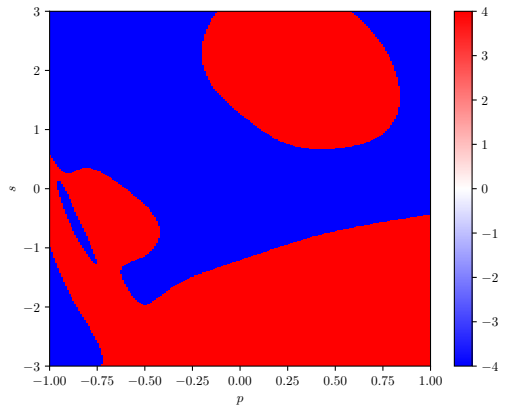


Figure 47: Optimal policy derived from Q_{50}, fixed N stopping rule and neural networks

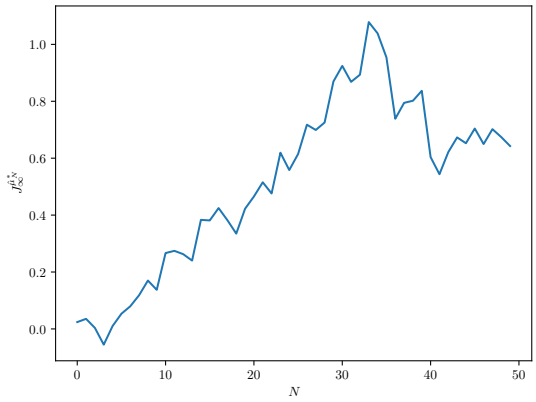
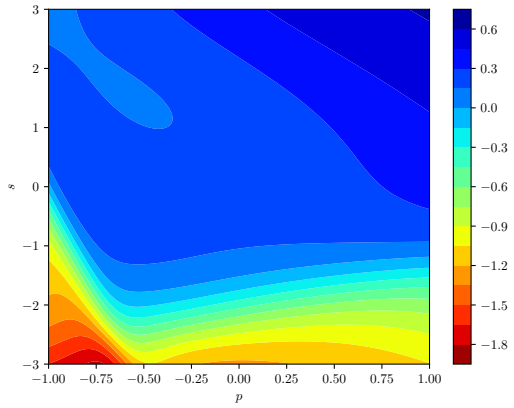
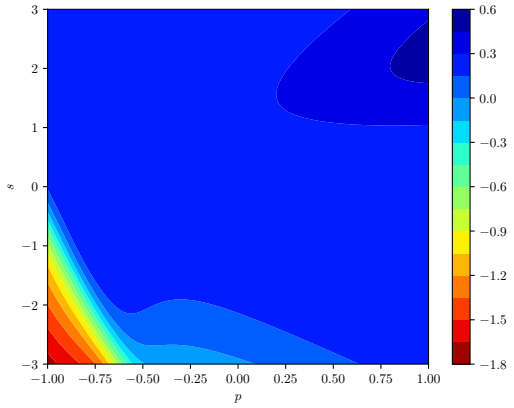


Figure 48: Expected return of the optimal policy

(b) Distance-related stopping rule



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



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

Figure 49: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `MLPRegressor`, distance-related stopping rule

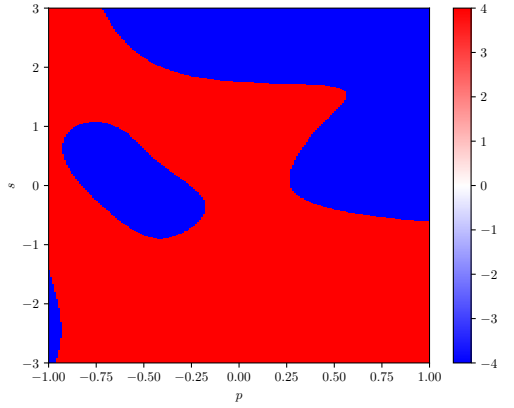


Figure 50: Optimal policy derived from Q_{50}, distance-related stopping rule and neural networks

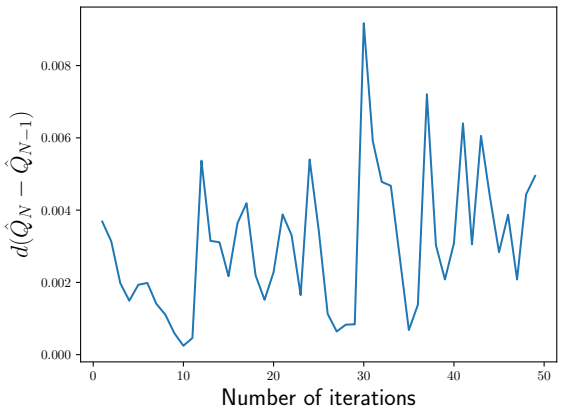


Figure 51: Distance-related convergence

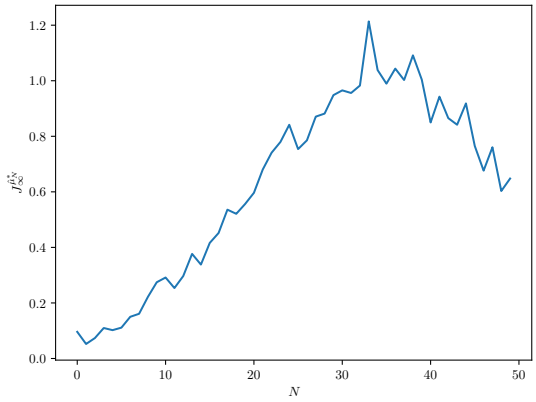
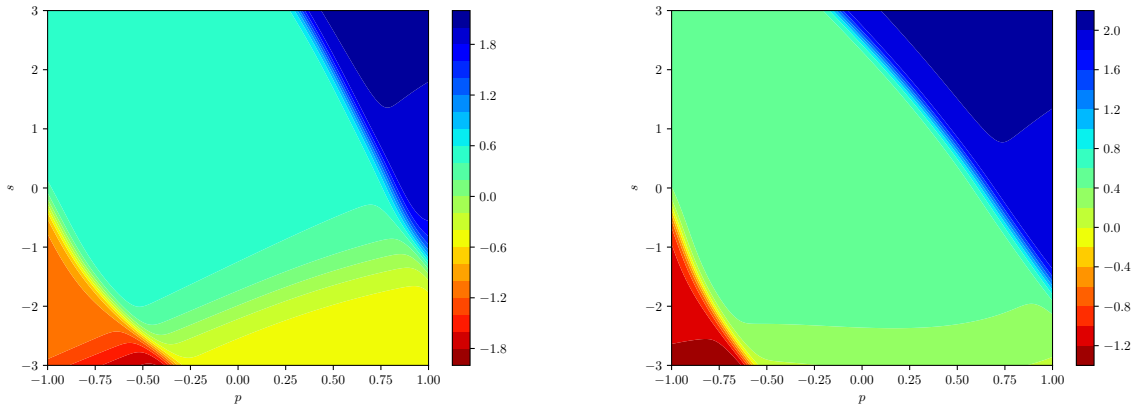


Figure 52: Expected return of the optimal policy, distance related

2. ϵ-greedy generation strategy

   To speed-up the computations, the value of ϵ has been changed from $\epsilon = 0.25$ to $\epsilon = 0.5$ in this case.

   (a) Fixed N stopping rule



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

Figure 53: $Q_{50}((p, s), u)$ obtained with *FQI* and *Scikit learn*'s `MLPRegressor`, fixed N stopping rule

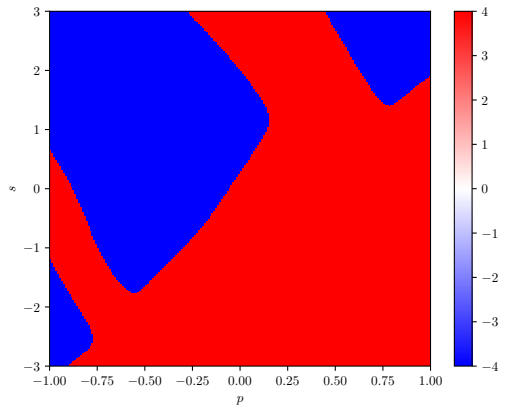


Figure 54: Optimal policy derived from Q_{50}, fixed N stopping rule and neural networks

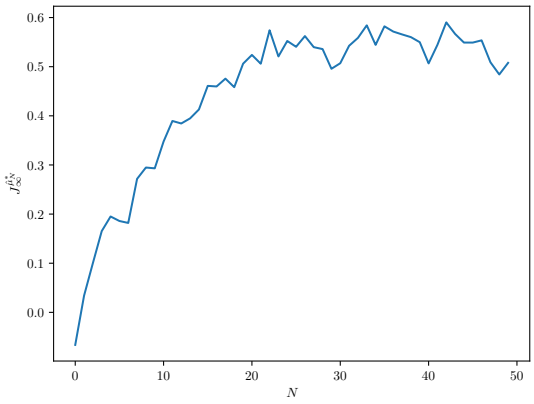


Figure 55: Expected return of the optimal policy, ϵ-greedy exploration

(b) Distance-related stopping rule

Regarding the poor results already obtained with the designed neural network and taking into account the quite large computing time required for neural networks, it has been judged pointless to perform FQI with a ϵ-greedy policy and the distance-related stopping rule.

Brief review of the influence of the generation strategy and of the stopping rule

Using an ϵ-greedy exploration strategy tends to overestimate the optimal expected return. One can see that in the linear regression case, while the expected return was about 0.2 when the generation strategy was a random exploration of the state space, the ϵ-greedy case quickly overcome this threshold to converge at about 0.55. The same phenomenon occurs with extra randomized trees, where the optimal expected return converges to 0.45 rather than 0.35. The only exception appears considering the neural net. In that case the impact of the generation strategy is that in the case of an ϵ-greedy exploration, the expected return increases monotonically while it reached a peak in the random case.

The impact of the stopping rule will depend on the threshold decided for the minimal distance between two successive models. In that case, 10^{-5} was arbitrarily chosen as the minimal distance. This minimal

distance is only reached in the linear regression case. Indeed, this supervised learning model is the less complex between the three models that have been considered. This means that at each iteration, the model will slightly change as the approximation is linear. However, one can observe that, in general, except for the MLP regressor case, the distance between two successive models tend to decrease and converge towards 0 after some number of iterations. As a result, one could have considered a threshold a bit greater, such that the algorithm would have stopped a bit earlier at the price of a bit less accurate approximation of Q. Indeed, the algorithm would have stopped at the beginning of the plateau of the expected return. Nonetheless, 50 iterations is a reasonable number of steps to ensure the convergence of the expected return while avoiding unnecessary computation time. Using the fixed-N with $N = 50$ could thus be seen as a safer stopping rule to use because the distance between successive models does not always decrease depending on the supervised learning algorithm.

References

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