University of Liège

Assignment 2

Optimal decision making for complex problems

Maxime Meurisse (s161278)
François Rozet (s161024)

Academic year 2020-2021

1 Implementation of the Domain

We implement the different components of the *car-on-the-hill* problem in the `domain.py` file.

In this problem, the time is discretized, meaning that between two successive time steps t and $t+1$ a constant time $\Delta = 0.1\,\text{s}$ passes. To implement the environment dynamics, we exploit the *Euler integration method*. The integration step being $\delta = 0.001\,\text{s}$, at each transition, we update the state by applying $\frac{\Delta}{\delta} = 100$ times

$$x \leftarrow x + \delta \dot{x} \tag{1}$$

where $\dot{x}$ is equivalent to $(\dot{p}, \dot{s})$, defined in the statement. Our implementation handles the terminal states case by checking if either $|p| > 1$ or $|s| > 3$.

1.1 Rule-based policy

We called our rule-based policy the "step back policy" as the agent initially moves backward before accelerating to gain momentum in the downhill. Formally,

$$\mu(p, s) = 4 \begin{cases} -1 & \text{if } -0.5 < p < 0 \text{ and } -1.5 < s < 0 \\ +1 & \text{else} \end{cases} \tag{2}$$

This policy is used to simulate a *terminating* trajectory, starting at an initial state sampled from the distribution presented in the statement ($p_0 \sim \mathcal{U}(-0.1, 0.1)$ and $s_0 = 0$). This trajectory is illustrated in Listing 1.

```
((0.014, 0), 4, 0, (0.001, -0.290))
((0.001, -0.29), 4, 0, (-0.043, -0.592))
((-0.043, -0.592), -4, 0, (-0.141, -1.409))
((-0.141, -1.409), -4, 0, (-0.331, -2.415))
((-0.331, -2.415), 4, 0, (-0.572, -2.224))
((-0.572, -2.224), 4, 0, (-0.754, -1.389))
((-0.754, -1.389), 4, 0, (-0.852, -0.575))
((-0.852, -0.575), 4, 0, (-0.873, 0.16))
((-0.873, 0.16), 4, 0, (-0.82, 0.92))
((-0.82, 0.92), 4, 0, (-0.686, 1.778))
((-0.686, 1.778), 4, 0, (-0.47, 2.448))
((-0.47, 2.448), 4, 0, (-0.228, 2.247))
((-0.228, 2.247), 4, 0, (-0.031, 1.675))
((-0.031, 1.675), 4, 0, (0.118, 1.409))
((0.118, 1.409), 4, 0, (0.255, 1.348))
((0.255, 1.348), 4, 0, (0.391, 1.382))
((0.391, 1.382), 4, 0, (0.535, 1.514))
((0.535, 1.514), 4, 0, (0.697, 1.737))
((0.697, 1.737), 4, 0, (0.884, 2.028))
((0.884, 2.028), 4, 1, (1.104, 2.363))
```

Listing 1 – A terminating trajectory of the policy (2).

2 Expected Return of a Policy in Continuous Domain

In the car-on-the-hill problem, the dynamics f and reward signal r of the environment are deterministic, *i.e.* a state-action pair (x, u) will always lead to the same one-step transition (x, u, r, x'). However, the choice of the initial state x_0 is stochastic. Therefore, the expected return J^μ of a policy μ can be expressed as

$$J^\mu = E_{x_0 \sim p(x_0)} \left\{ \lim_{N \to \infty} \sum_{i=0}^{N-1} \gamma^i r(x_i, \mu(x_i)) \right\} \tag{3}$$

where $x_{i+1} = f(x_i, \mu(x_i))$ and $\gamma \in [0; 1)$ is the discount factor. Interestingly, because r is non-null only once in a trajectory, we have

$$\|J^\mu\|_\infty \leq \|r\|_\infty = B_r = 1. \tag{4}$$

In order to estimate the expected return, we have to fix a time horizon N such that

$$J^\mu_N = E_{x_0 \sim p(x_0)} \left\{ \sum_{i=0}^{N-1} \gamma^i r(x_i, \mu(x_i)) \right\} \tag{5}$$

is a good approximation of J^μ. In the car-on-the-hill environment, it can be shown [1, 2] that

$$\|J^\mu - J^\mu_N\|_\infty \leq \gamma^N \|J^\mu\| \leq \gamma^N B_r \tag{6}$$

which can be used to enforce a certain precision. With a threshold $\epsilon = 10^{-2}$, we have

$$\begin{aligned} \gamma^N B_r &\leq \epsilon \\ N &\geq \log_\gamma \frac{\epsilon}{B_r} \end{aligned}$$

which leads the smallest acceptable value

$$N = \left\lceil \log_\gamma \frac{\epsilon}{B_r} \right\rceil = 90. \tag{7}$$

Still, it is not possible to compute J^μ_N exactly as there is an infinite number of possible initial states x_0. Therefore, the expectation has to be approximated as well. This is done using the *Monte Carlo* method, *i.e.* approximating the expected return by the average cumulative reward over several trajectories. Let $h = (x_0, u_0, r_0, x_1, \ldots, u_{t-1}, r_{t-1}, x_t)$ denote a trajectory of length t. Since x_t can be the only terminal state of h, the cumulative reward C_h is computed as

$$C_h = \sum_{i=0}^{t-1} \gamma^i r_i = \gamma^{t-1} r_{t-1}. \tag{8}$$

Then, to estimate J^μ_N, we simulate n trajectories h_i of length N or less and compute

$$J^\mu_N \simeq \frac{1}{n} \sum_{i=0}^{n-1} C_{h_i}. \tag{9}$$

We implement this as a routine and estimate the expected return of our policy (2) over $n = 50$ trajectories (truncated at N). The obtained values, from 1 to N[1], are shown in Figure 1.

[1] With definition (5), J^μ_0 doesn't have a meaning.

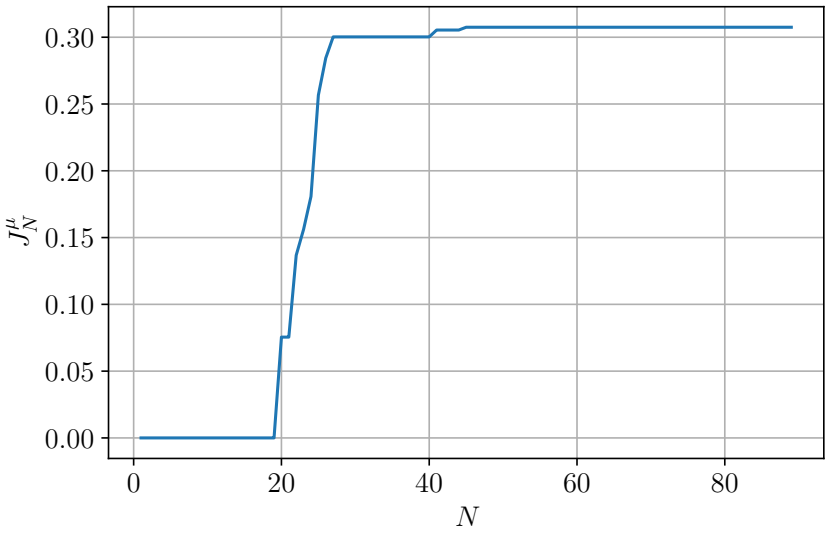


Figure 1 – J_N^μ of the policy (2) w.r.t. N.

One can see that the (approximated) expected return converged to a fixed value close to 0.31 (with a precision ϵ).

3 Visualization

The visual rendering tools are implemented in the `display.py` file. Our routine is inspired from the Python script provided with the statement.

We simulate the policy (2) starting at initial state $(p_0, s_0) = (0, 0)$ and produce a GIF file by assembling all output images produced. The latter, provided in the archive of our project, can also be visualized in Figure 2.

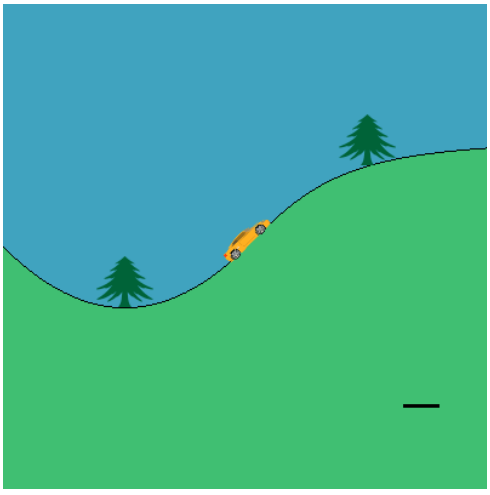


Figure 2 – Visualization of the policy (2), starting at initial state $(p_0, s_0) = (0, 0)$. *This is an animated figure that is best viewed using Adobe Reader.*

4 Fitted-Q-Iteration

The goal of the *Fitted-Q-Iteration* algorithm is to train a sequence of supervised regressors $\hat{Q}_1$, $\hat{Q}_2$, ..., $\hat{Q}_N$ estimating respectively the functions Q_1, Q_2, ..., Q_N. To do so, N training sets are built. Let h be an available set of t one-step transitions $\{(x_k, u_k, r_k, x'_k) \mid k = 1, \ldots, t\}$. Then, the training sets are

$$\begin{aligned} \mathrm{TS}_1 &= \{((x,u), r) \mid (x,u,r,x') \in h\} \\ \mathrm{TS}_N &= \left\{ \left((x,u), r + \gamma \max_{u' \in U} \hat{Q}_{N-1}(x', u') \right) \mid (x,u,r,x') \in h \right\} \quad \forall N \geq 2 \end{aligned}$$

and $\hat{Q}_N$ is trained on TS_N. However, in the car-on-the-hill problem, one cannot take an action if the current state is terminal. Therefore, we tweak the formula as

$$\mathrm{TS}_N = \left\{ \left((x,u), r + I_{\{r=0\}} \gamma \max_{u' \in U} \hat{Q}_{N-1}(x', u') \right) \mid (x,u,r,x') \in h \right\} \quad \forall N \geq 2 \tag{10}$$

where I is the identity function.

4.1 Generation of h

We implement two generation strategies code-named "Exhaustive" and "Monte Carlo", respectively.

Exhaustive The first generation strategy is inspired by the "exhaustive grid search" hyperparameter optimization technique. We cut the state space into 200×200 uniformly spaced grid cells $x_{ij} = (p_i, s_j)$ and apply all possible actions u (4 or -4) to them.

$$h = \{(x_{ij}, u, r(x_{ij}, u), f(x_{ij}, u)) \mid 1 \leq i, j \leq 200, u \in U\} \tag{11}$$

The main advantage of this strategy is that the state space is very well covered. Especially, edge cases are represented. However, a drawback is that the distribution of state-action pairs is not similar to those of trajectories. Therefore, a lot of pairs could be useless (unreachable) and confuse the supervised algorithm.

Monte Carlo The second strategy is simply to simulate n trajectories h_i with a uniform random policy starting from a random ($p_0 \sim \mathcal{U}(-0.1, 0.1)$ and $s_0 = 0$) initial state x_0 and to aggregate them as a single set.

$$h = \bigcup_{i=0}^{n-1} h_i \tag{12}$$

In practice, we don't choose n; we simulate trajectories of at most 1000 transitions until the total number of transitions is 80 000, *i.e.* the same number of transitions than the exhaustive strategy.

The advantages/drawbacks of this strategy are the opposite of the previous one: the state space is unlikely to be well covered but the distribution should correspond better to those of actual trajectories.

4.2 Stopping rules

We thought of two stopping rules.

1. As in Section 2, for the car-on-the-hill problem, it can be shown [1, 2] that

$$\left\| J^{\mu^*} - J^{\mu^*_N} \right\|_\infty \leq \frac{2\gamma^N}{1-\gamma} \left\| J^\mu \right\| \leq \frac{2\gamma^N}{1-\gamma} B_r. \tag{13}$$

   Our first rule is to choose N such that the suboptimality of μ^*_N is bounded by $\epsilon = 10^{-2}$, which is achieved with

$$N = \left\lceil \log_\gamma \left(\frac{\epsilon}{2B_r}(1-\gamma) \right) \right\rceil = 162. \tag{14}$$

2. The second idea is to stop the FQI algorithm when the estimators' predictions have or seemed to have "converged", *i.e.* we want

$$\left\| \hat{Q}_N - \hat{Q}_{N-1} \right\|_\infty \leq \epsilon \tag{15}$$

   with ϵ arbitrarily set to 0.02. However, in the continuous domain it is not possible to compute exactly this infinite norm. Instead, we replace it by a mean absolute difference

$$\frac{1}{2t} \sum_{(x,u,r,x') \in h, u' \in U} \left| \hat{Q}_N(x', u') - \hat{Q}_{N-1}(x', u') \right| \tag{16}$$

   which allows to "reuse" quantities computed during the creation of TS_N and TS_{N+1}.

4.3 Supervised algorithms

As requested by the statement, we implement 3 supervised algorithms:

1. Linear Regression, borrowed as-is from `scikit-learn`;
2. Extremely Randomized Trees, also borrowed from `scikit-learn` but with "only" 20 estimators;
3. Neural Network, implemented using `tensorflow.keras`. As architecture, we chose a simple *Multi-Layer Perceptron* (MLP) with 3 hidden layers of 8 neurons each and ReLU activation functions. The inputs are (p, s, u) triples and the outputs are $\hat{Q}_N((p, s), u)$ scalars. This architecture is represented in Figure 3.

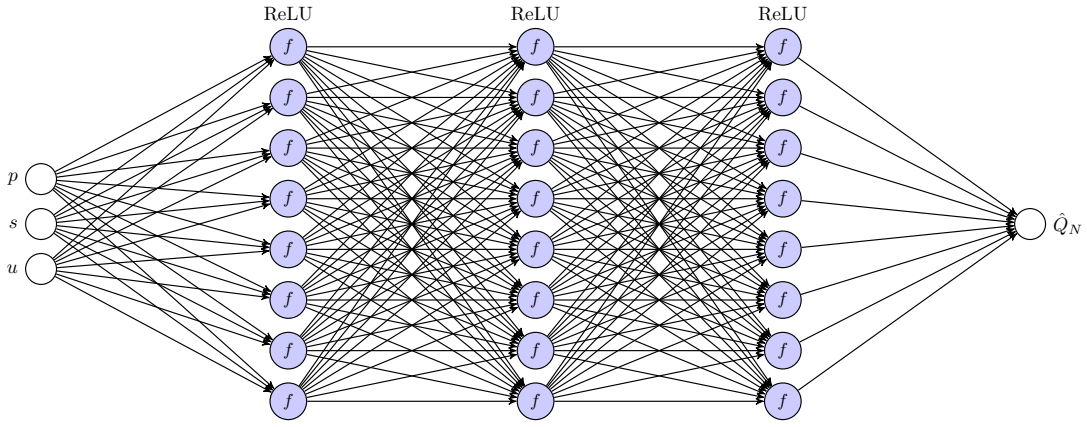


Figure 3 – Architecture of the MLP used in the Fitted-Q-Iteration algorithm.

The network is trained using the Adam [3] optimizer with the *Mean Squared Error* (MSE) loss. This architecture is relatively narrow and shallow. The reason behind this choice is that the size(s) of our training set(s) is relatively moderate and the number of inputs is very small. Consequently, if we were to increase the width and/or depth of our network, the network could end-up overfitting the training data. In fact, restricting the width/depth of a neural network is a kind of *regularization* as it reduces the model's expressivity, which usually helps to *generalize*. The same goes for the number of estimators and maximal depth of Randomized Trees.

We train these supervised algorithms, for all (4) combinations of generation strategy and stopping rule, using the FQI algorithm. We then use them to estimate the state-action values $Q_N(x, u)$ on a uniformly spaced (resolution of 0.01) grid of the state space for all possible actions.

In the following, the state-action values are displayed on separate figures for each action. A third figure is used to display the policy $\hat{\mu}_N^*$ inferred from $\hat{Q}_N$ as

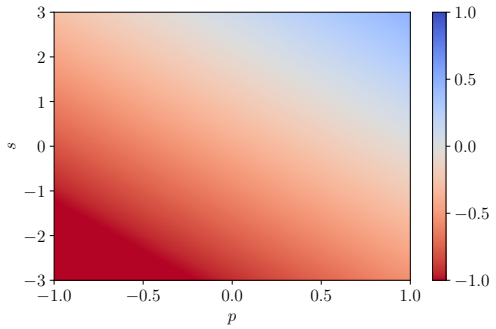
$$\hat{\mu}_N^*(x) \in \arg\max_{u \in U} Q(x, u). \tag{17}$$

The action -4 is represented in red and the action $+4$ in blue.

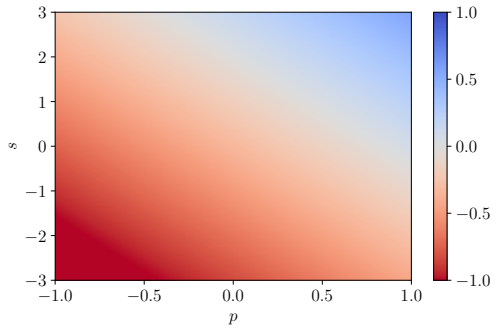
We then estimate the expected return $J^{\hat{\mu}_N^*}$ of the policy, using the routines implemented for Section 2. It should be noted that we truncate the sampled trajectories to $N' = 90$ transitions, according to relation (7).

For the sake of conciseness, we chose to display the figures in 2×2 grids, which necessarily reduces the size of the figures and labels. However, the plots are *vectorized*, which implies that, in the digital document, the font will stay crisp at any level of zoom.

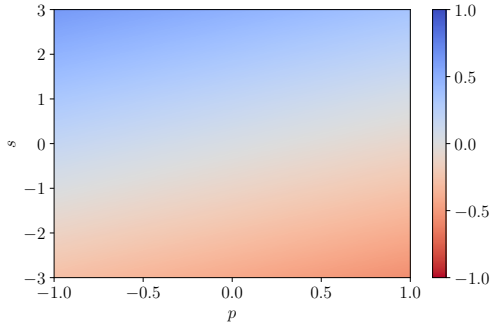
4.3.1 Linear Regression



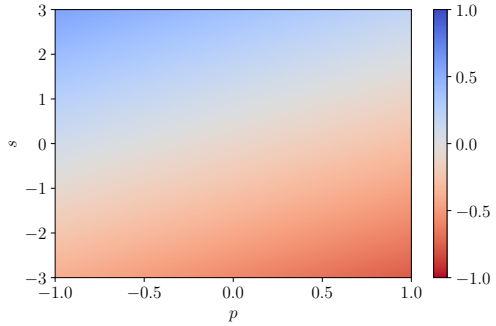
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 7$.

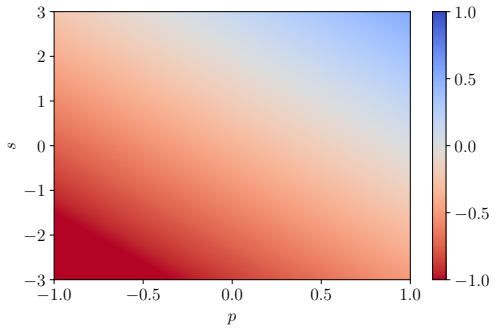


(c) Monte Carlo, stopping rule 1; $N = 162$.

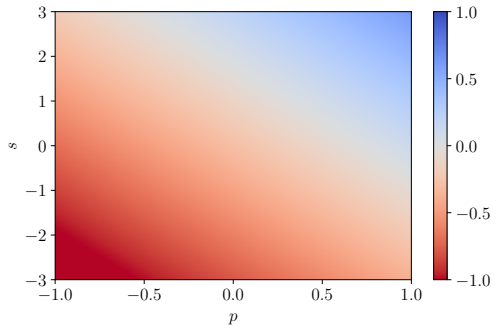


(d) Monte Carlo, stopping rule 2; $N = 10$.

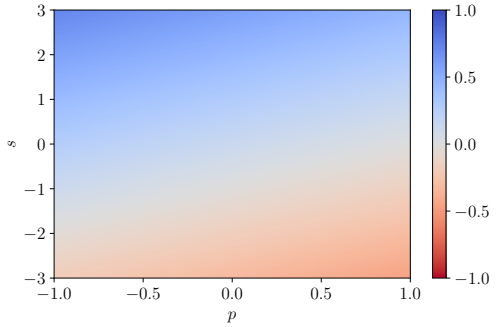
Figure 4 – $\hat{Q}_N((p, s), -4)$ values of the Linear Regression algorithm.



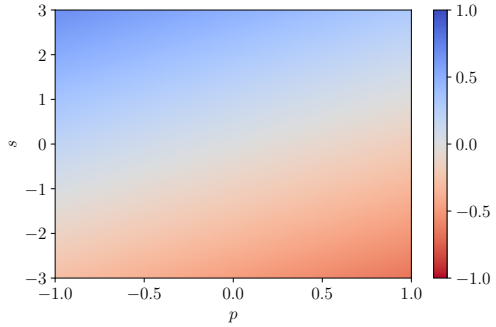
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 7$.

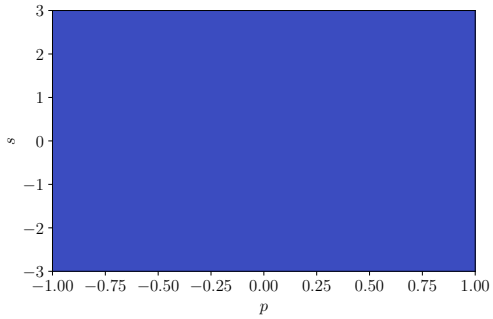


(c) Monte Carlo, stopping rule 1; $N = 162$.

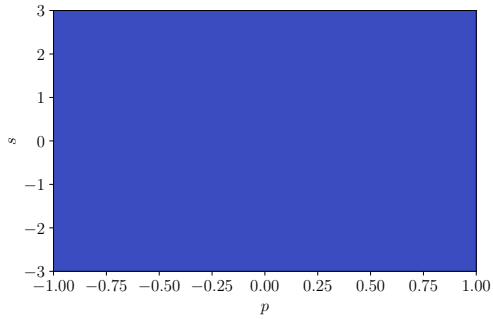


(d) Monte Carlo, stopping rule 2; $N = 10$.

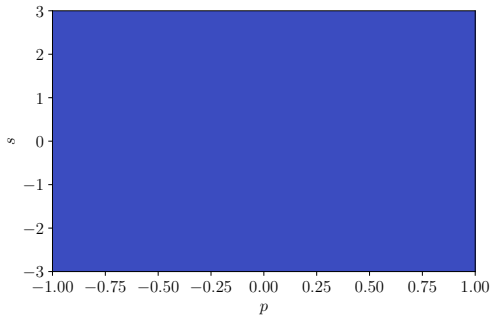
Figure 5 – $\hat{Q}_N((p, s), +4)$ values for the Linear Regression algorithm.



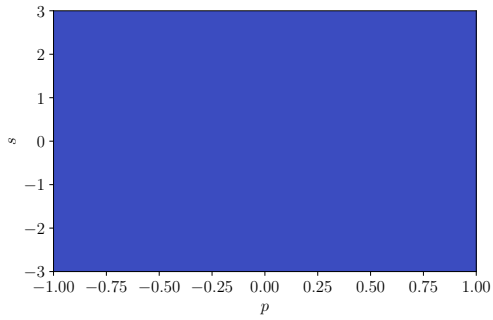
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 7$.



(c) Monte Carlo, stopping rule 1; $N = 162$.



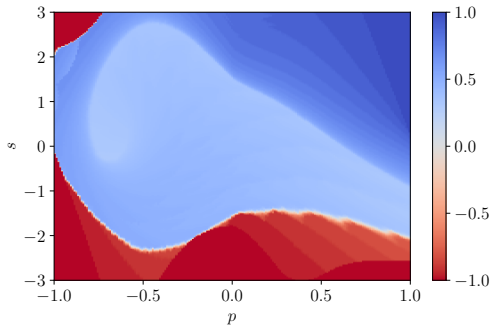
(d) Monte Carlo, stopping rule 2; $N = 10$.

Figure 6 – $\hat{\mu}_N^*$ policies inferred from the Linear Regression algorithm.

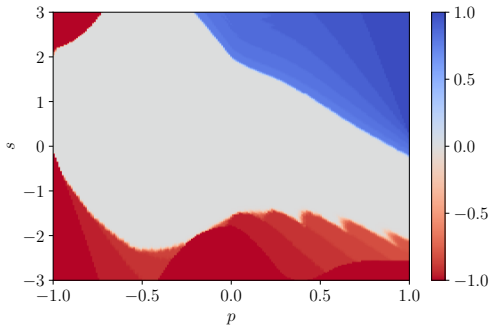
	Stopping rule 1	Stopping rule 2
Exhaustive	0.0	0.0
Monte Carlo	0.0	0.0

Table 1 – Estimations of the expected returns of $\hat{\mu}_N^*$.

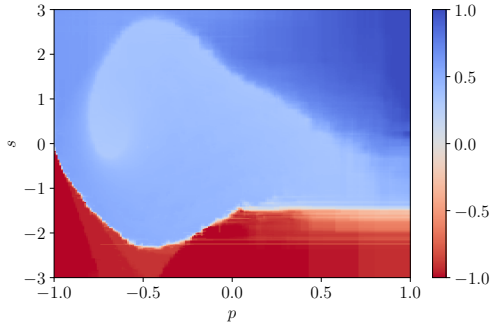
4.3.2 Extremely Randomized Trees



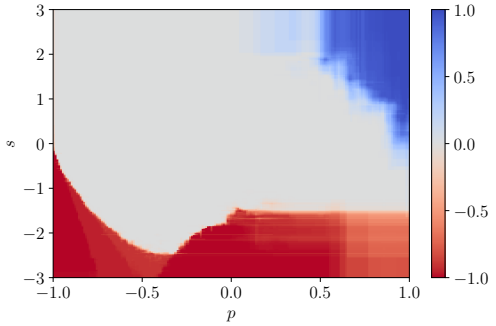
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 6$.

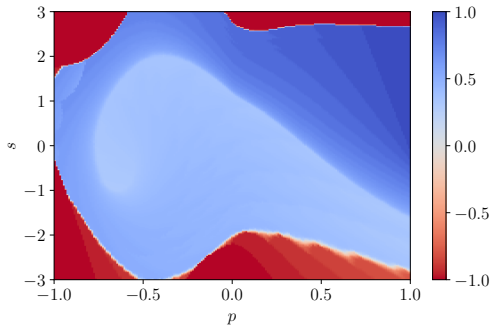


(c) Monte Carlo, stopping rule 1; $N = 162$.

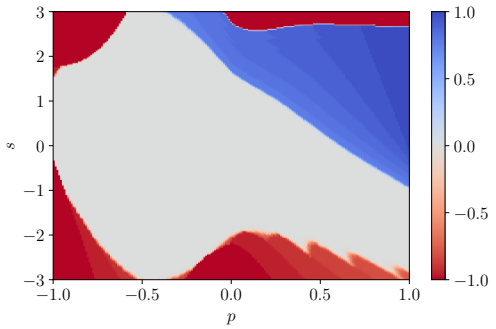


(d) Monte Carlo, stopping rule 2; $N = 3$.

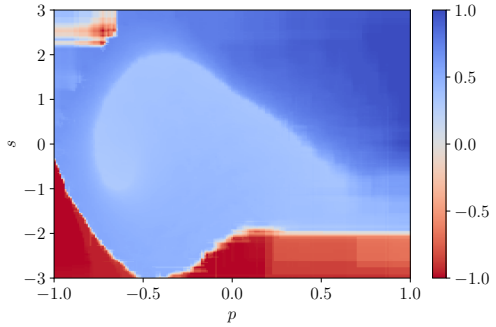
Figure 7 – $\hat{Q}_N((p, s), -4)$ values for the Extremely Randomized Trees algorithm.



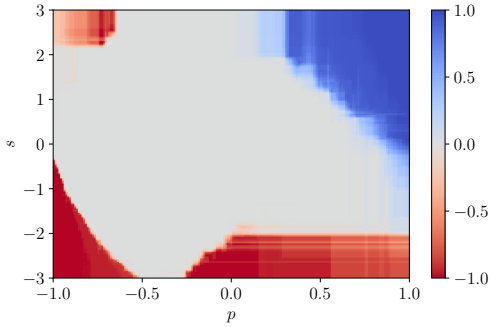
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 6$.

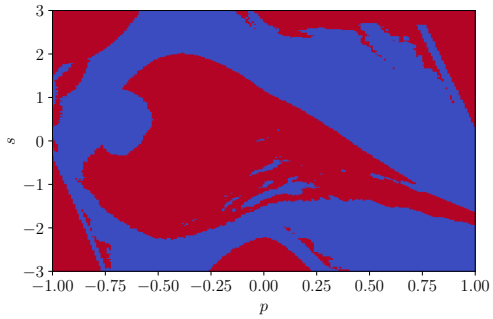


(c) Monte Carlo, stopping rule 1; $N = 162$.

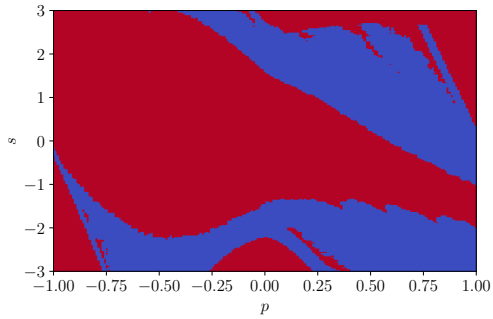


(d) Monte Carlo, stopping rule 2; $N = 3$.

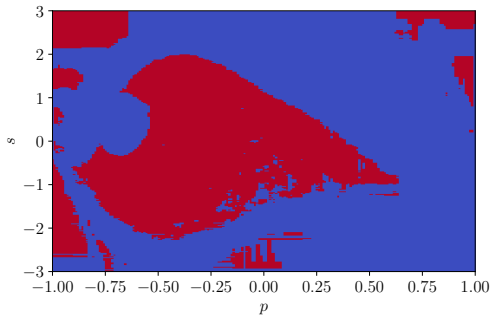
Figure 8 – $\hat{Q}_N((p, s), +4)$ values for the Extremely Randomized Trees algorithm.



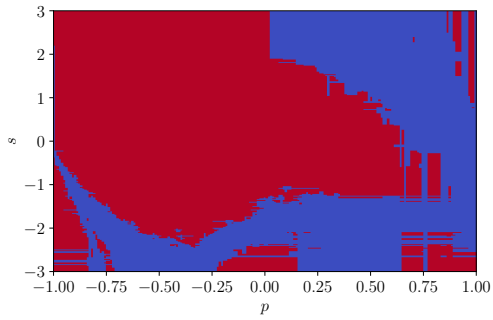
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 6$.



(c) Monte Carlo, stopping rule 1; $N = 162$.



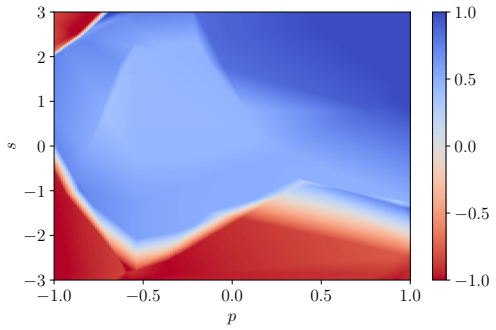
(d) Monte Carlo, stopping rule 2; $N = 3$.

Figure 9 – $\hat{\mu}_N^*$ policies inferred from the Extremely Randomized Trees algorithm.

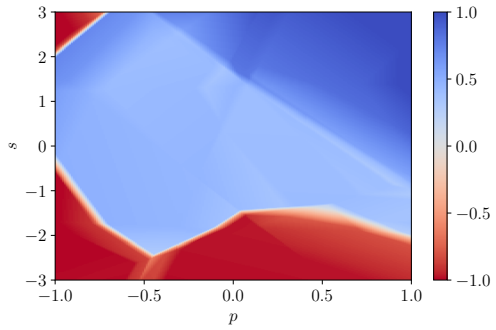
	Stopping rule 1	Stopping rule 2
Exhaustive	0.42	0.0
Monte Carlo	0.42	0.0

Table 2 – Estimations of the expected returns of $\hat{\mu}_N^*$.

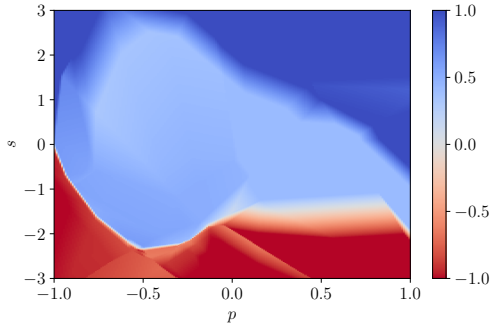
4.3.3 Neural Network



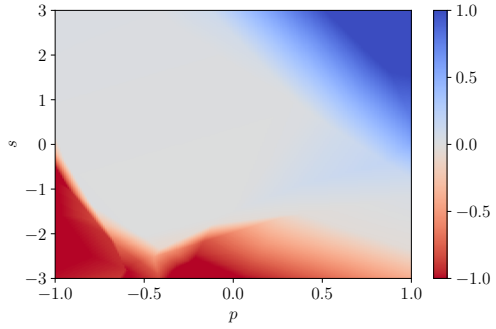
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 196$.

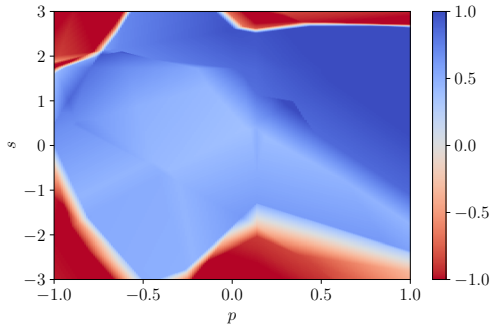


(c) Monte Carlo, stopping rule 1; $N = 162$.

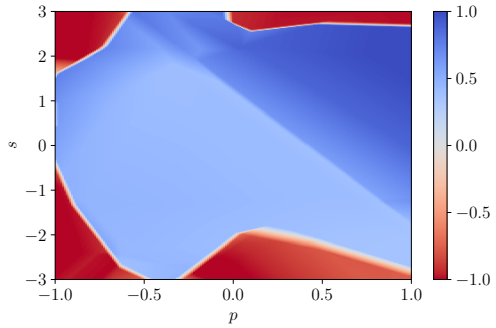


(d) Monte Carlo, stopping rule 2; $N = 5$.

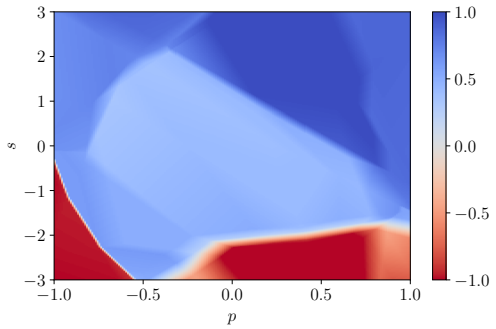
Figure 10 – $\hat{Q}_N((p, s), -4)$ values for the MLP algorithm.



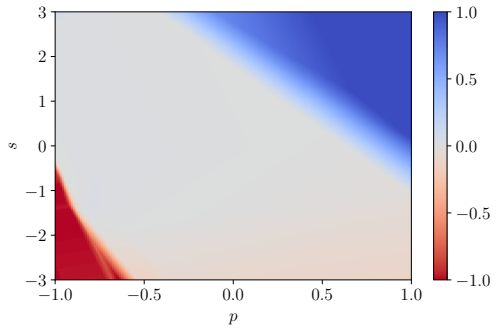
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 196$.

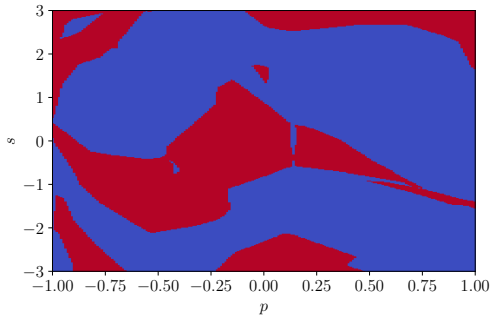


(c) Monte Carlo, stopping rule 1; $N = 162$.

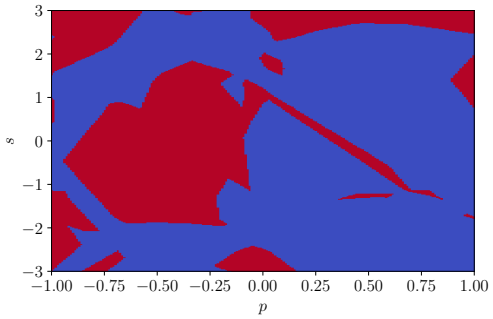


(d) Monte Carlo, stopping rule 2; $N = 5$.

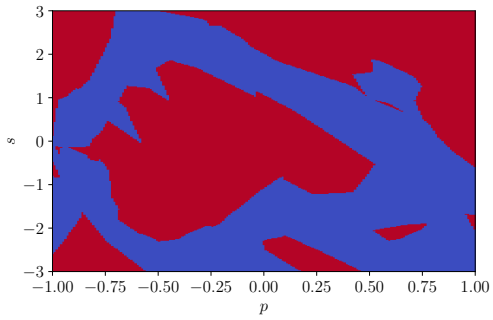
Figure 11 – $\hat{Q}_N((p, s), +4)$ values for the MLP algorithm.



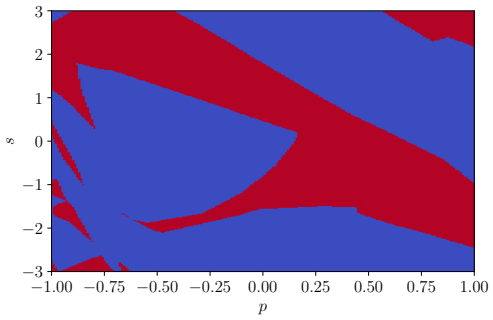
(a) Exhaustive, stopping rule 1; $N = 162$.



(b) Exhaustive, stopping rule 2; $N = 196$.



(c) Monte Carlo, stopping rule 1; $N = 162$.



(d) Monte Carlo, stopping rule 2; $N = 5$.

Figure 12 – $\hat{\mu}_N^*$ policies inferred from the MLP algorithm.

	Stopping rule 1	Stopping rule 2
Exhaustive	0.38	0.35
Monte Carlo	0.24	0.0

Table 3 – Estimations of the expected returns of $\hat{\mu}_N^*$.

4.4 Discussion

In general, we observe that the results of the Linear Regression are terrible, those of the Extremely Randomized Trees are great and those with the MLP are in between.

Concerning the Linear Regression, all combinations of generation strategy and stopping rule give the same policy: the agent systematically performs the action $u = +4$ (*cf.* Figure 6). This might reflect that the model is only able to learn that, in general, the agent must go to the right. Indeed, by accelerating only to the right, the agent has a chance of getting a positive reward, whereas by accelerating only to the left, the agent is sure to never be able to win. Unfortunately this policy does not allow the agent to obtain rewards in a reasonable time, hence the null expected rewards.

For Extremely Randomized Trees, the estimated $\hat{Q}_N$ values, for the stopping rule 1, seem very accurate. Indeed, we can see (*cf.* Figures 7 and 8) that the estimator learned quite well were accelerating forward or backward makes you lose immediately, mainly because of the speed limit. For example, if you have a high positive speed at the top of the right hill, it is dangerous to accelerate forward. Interestingly, we observe that these "danger zones" are not as well defined for the Monte Carlo generation strategy. This is explained

by the fact that it is not possible to be in that position with that much speed starting from the initial state. Therefore, even if the policies inferred with each generation strategy are slightly different, their are equivalent in practice and have the same expected return (*cf.* Table 2), which is higher than the one of our step back policy (2).

For the stopping rule 2, the estimator is agnostic for a big part of the state-action space. This is explained by the fact that the FQI algorithm was stopped very early and before convergence. In fact, the threshold imposed on the mean absolute difference (16) does not ensure convergence afterwards. A more valid criterion would have been to wait for a *plateau* of the mean absolute difference, *i.e.* a long period during which the mean absolute difference doesn't change significantly.

Finally, concerning the MLP, we notice that the $\hat{Q}_N$ values (*cf.* Figures 10 and 11) are similar in sign to those of the Extremely Randomized Trees. However, their amplitudes are a little off, especially for the Monte Carlo generation strategy, which causes the inferred policy to be less effective and have a lower expected return (*cf.* Table 3).

For the exhaustive generation strategy and stopping rule 2, the FQI algorithm was stopped after $N = 196$ iterations. This is probably due to luck as it is very different with the Monte Carlo generation strategy ($N = 5$). Nevertheless, we observe that the $\hat{Q}_N$ values are actually closer to those of the Extremely Randomized Trees algorithm, which indicates that our MLP could profit from more FQI iterations.

5 Parametric Q-Learning

As shown in the previous assignment, the *Q-learning algorithm* allows us to directly infer $\hat{Q}$ from a set (not necessarily finite) of one-step transitions $h = \{(x_k, u_k, r_k, x'_k) \mid k = 1, \dots, t\}$.

We can [1] extend this algorithm to the case where a parametric Q-function estimator of the form $\hat{Q}(x, u, a)$ is used (*cf.* Algorithm 1).

Algorithm 1. Parametric Q-learning (PQL)

1. Initialize $\hat{Q}(x, u, a)$ to 0 everywhere.
2. Until some stopping criterion, update the parameters following

$$a \leftarrow a + \alpha\delta(x, u, r, x')\frac{\partial\hat{Q}(x, u, a)}{\partial a} \tag{18}$$

where (x, u, r, x') tuples are sampled from h and

$$\delta(x, u, r, x') = r + I_{\{r=0\}}\gamma \max_{u' \in U} \hat{Q}(x', u', a) - \hat{Q}(x, u, a) \tag{19}$$

is the *temporal difference*.

In fact, this algorithm is very close to a regular *stochastic gradient descent* (SGD) and is therefore suitable for training neural networks, which are general-purpose parametric function estimators. However, in gradient descent algorithms, the whole training set is usually used several times, called epochs, which is not necessarily the case in Q-learning algorithms. In fact, it corresponds to a special case of *offline* Q-learning algorithm where a finite and predetermined set h is iterated over several times.

5.1 Routine

In order to make a fair comparison between the Fitted-Q-Iteration (FQI) and the Parametric Q-Learning (PQL) algorithms, we use the same MLP architecture (*cf.* Figure 3) in both cases. We also use the same generation strategy, *i.e.* the Monte Carlo strategy defined in Section 4.1, as it is closer to what is usually done in a Q-learning algorithm. We choose different numbers[2] of one-step transitions n: 1000, 5000, 10 000, 50 000 and 100 000. Using these training sets, we apply the FQI algorithm until $N = 162$, *i.e.* using the first stopping rule (*cf.* Section 4.2). Since the network is trained during 5 epochs for each FQI iteration, in our implementation of PQL (realised with PyTorch), we train the network for $N \times 5 = 810$ epochs on the full training set.

Additionally, in (18), the parameters are updated one transition at a time, which corresponds to "pure" SGD. This strategy is usually avoided because it is very hazardous (high stochasticity) and quite slow as you have to update all the parameters at each step. Instead, we use the *mini-batch* SGD, for which the updating gradient is averaged over a few (32 in our case) transitions at each step. The learning rate α is arbitrarily set to 10^{-3}.

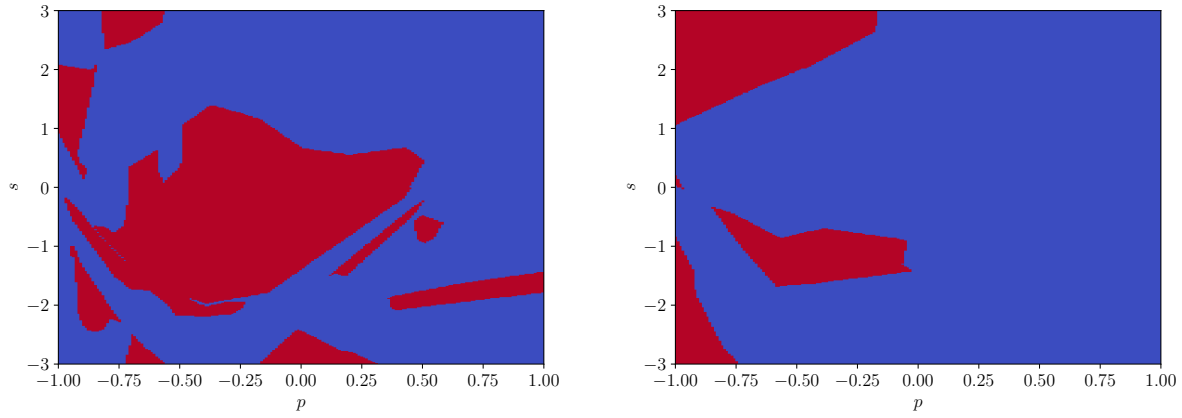
[2] We couldn't consider more than 5 training set sizes because each training took up to 2 hours on our computers and our time was limited.

Finally, for each algorithm, we infer the policy(ies) $\hat{\mu}^*$ from $\hat{Q}$ using (17) and display it in a colored 2D grid, as required by the statement. We then estimate the expected return(s) $J^{\hat{\mu}^*}$ of the policy(ies) using the routines implemented for Section 2. Once again, the expected return is approximated by $J^{\mu}_{N'}$ with $N' = 90$ from relation (7). We compare the expected returns in a curve plot where the x-axis is the number of one-step transitions (n) and the y-axis is the expected return.

5.2 Results

5.2.1 SGD

The results obtained with the FQI and PQL (using SGD) algorithms are shown in Figures 13 and 14.



(a) FQI

(b) PQL (using SGD)

Figure 13 – $\hat{\mu}^*$ policies inferred from $\hat{Q}$, for the FQI and PQL algorithms.

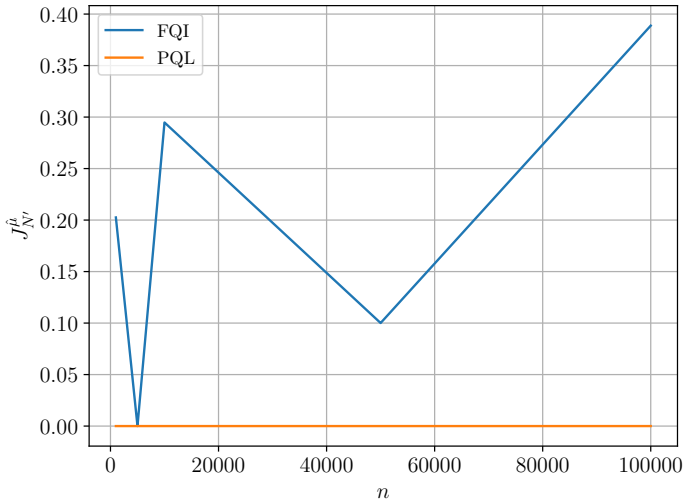


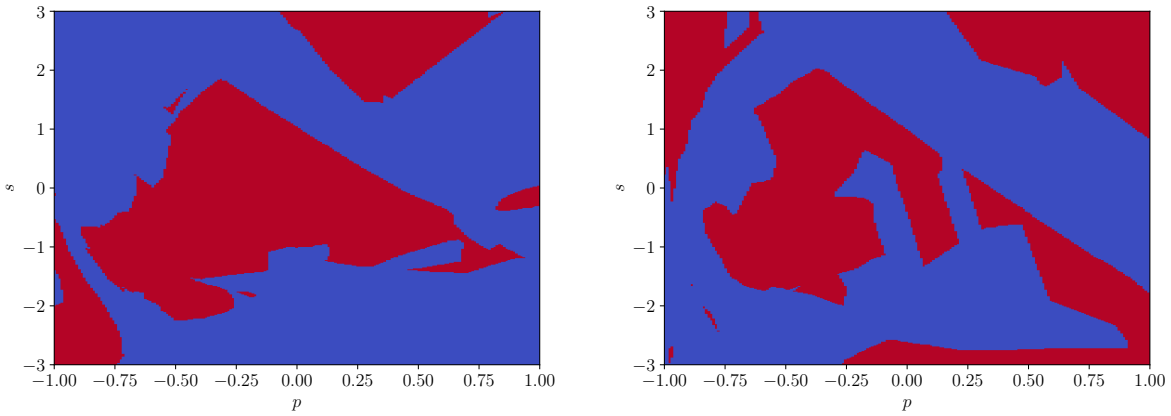
Figure 14 – Expected return comparison between FQI and PQL algorithms.

We can observe that the results, with the SGD optimizer, are terrible for the PQL algorithm, regardless of the size of the training set. Indeed, it didn't manage to find a good policy, while the FQI algorithm did, as expected from Section 4. This is actually

not surprising as "vanilla" PQL is known to perform poorly [4]. Also, we note that FQI doesn't seem to improve consistently with the number of transitions in the training set, although the number of data points (5) is not sufficient to draw any conclusion.

5.2.2 Adam

Hoping to improve the quality of PQL's results, we replace the SGD optimizer by Adam [3]. We also implement the *Double Q-learning* (DQL) algorithm [4], which is a variant of the PQL algorithm where a target network is introduced. The results are presented in Figures 15 and 16.



(a) PQL (b) DQL

Figure 15 – $\hat{\mu}^*$ policies inferred from $\hat{Q}$ for the PQL and DQL algorithms, using the Adam optimizer.

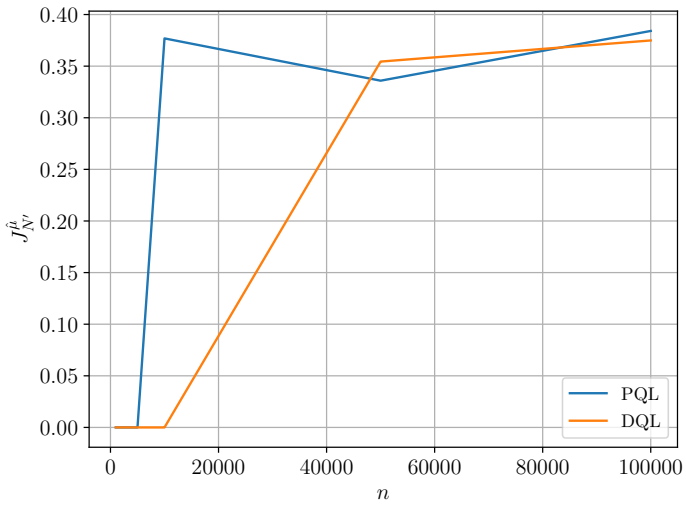


Figure 16 – Expected return comparison between PQL and DQL algorithms, using the Adam optimizer.

This time, the inferred policies are much closer to what they should be (*cf.* Figure 9), for both PQL and DQL. As a logical consequence, the expected returns are now almost matching those of FQI with the same MLP, although they still lag behind the Extremely Randomized Trees that reach 0.42 of expected return.

Concerning convergence, it seems like PQL and DQL (with Adam) are more stable than FQI, but we should still keep in mind that we only have 5 data points and the experience was only conducted once. An improvement would have be to compute the mean and variance of the expected return while changing the random seeds, but it would take a lot of time.

Normalised Parametric Q-Learning

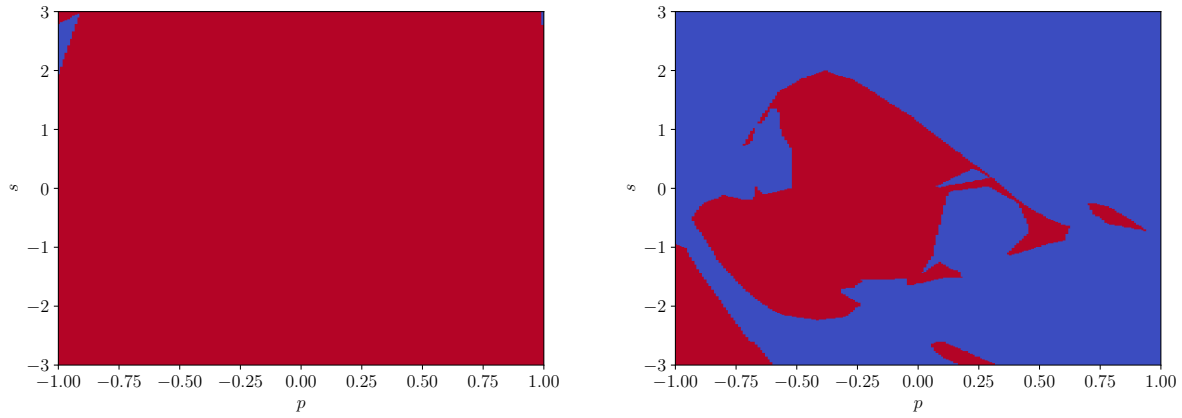
To study the impact of normalisation, we divide the update term of the PQL algorithm by its 2-norm, *i.e.* (18) becomes

$$a \leftarrow a + \alpha \frac{g}{\|g\|_2} \tag{20}$$

with

$$g = \delta(x, u, r, x') \frac{\partial \hat{Q}(x, u, a)}{\partial a},$$

everything else remaining the same. The results are shown in Figures 17 and 18.



(a) PQL (b) DQL

Figure 17 – $\hat{\mu}^*$ policies inferred from $\hat{Q}$ for the PQL and DQL algorithms, using the Adam optimizer and a normalised update term.

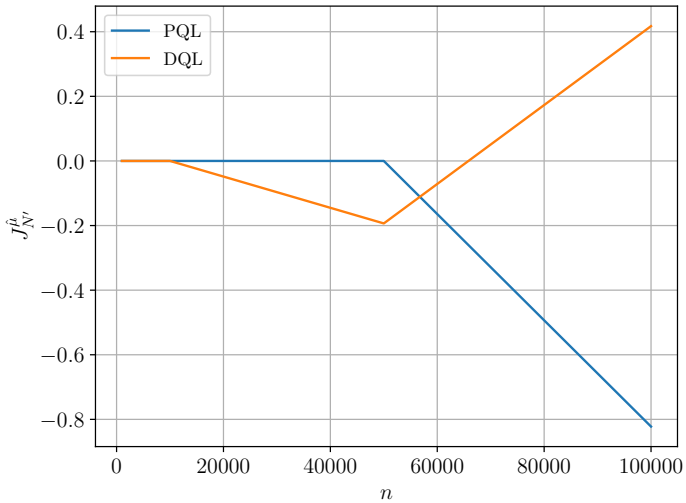


Figure 18 – Expected return comparison between PQL and DQL algorithms, using the Adam optimizer and a normalised update term.

In this setting, the results are very different according to the algorithm. Indeed, normalised PQL provides a terrible (even worse than Linear Regression) policy, which is

highlighted by its negative expected return, while DQL leads to a good (close 0.42) expected return.

However, for both algorithms, the policies change a lot depending on the number of one-step transitions in the training set n (*cf.* Figure 18). Especially, PQL's expected return dropped inexplicably to -0.8 for the largest set and DQL's expected return went from -0.2 at $n = 50\,000$ to 0.42 at $n = 100\,000$.

Because of this apparent instability, it is hard to draw any valuable conclusions. However, we investigated a bit and realized that, in our implementation, the update term's norm $\|g\|_2$ is actually quite small (to the order of 10^{-2}). Therefore, normalizing g actually increases it by a factor up to 10^2, since we didn't change the learning rate α. This could explain the observed behaviours, as it is well known that having very large parameter updates lead to training instabilities.

References

[1] Damien Ernst. "Optimal sequential decision making for complex problems". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2020/02/RL-1.pdf (pages 2, 5, 14).

[2] François Rozet. "Suboptimality of stationary policies". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2021/02/suboptimality_bound_proof-1.pdf (pages 2, 5).

[3] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (pages 6, 16).

[4] Sergey Levine and Damien Ernst. "Advanced algorithms for learning Q-functions." URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2018/02/More-on-Q-Learning.pdf (page 16).