Faculty of Applied Sciences

Optimal Decision Making for complex problems

INFO8003-1

Assignment 2 : Reinforcement Learning in a Continuous Domain

Authors :
Delcour Florian - s181063
Makedonsky Aliocha - s171197

Instructors :
Ernst Damien
Teaching assistants :
Aittahar Samy
Miftari Bardhyl

March 2022

1 Implementation of the domain

To implement the dynamics of the system, we need to define the derivative and the second derivative of the function Hill(p) w.r.t. p. They are defined as follows :

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

$$\mathbf{Hill'}(\mathbf{p}) = \begin{cases} 1 + 2p & \text{if p} < 0 \\ \dfrac{1}{(1+5p^2)^{3/2}} & \text{otherwise} \end{cases}$$

$$\mathbf{Hill''}(\mathbf{p}) = \begin{cases} 2 & \text{if p} < 0 \\ \dfrac{-15p}{(1+5p^2)^{5/2}} & \text{otherwise} \end{cases}$$

Then we can compute the new value of the position p and the speed s of the car after each action taken with the Euler integration method with an integration time step Δt_i of 0.001.
We obtain those new values by computing $\dfrac{\Delta t}{\Delta t_i}$ with Δt the time step used to discretize the domain, 0.1 here, so 100 times the following functions :

$$p = p + \Delta t_i * \dot{p}$$

$$s = s + \Delta t_i * \dot{s}$$

with $\dot{p} = s$ and $\dot{s} = \dfrac{u}{m(1+Hill'(p))} - \dfrac{gHill'(p)}{1+Hill'(p)^2} - \dfrac{s^2 Hill'(p)Hill''(p)}{1+Hill'(p)^2}$, m = 1, g = 9.81

Figure 1 represents the trajectory we obtained with a random policy, so the action taken by the car is either 4 or -4 with equal probabilities at each time step :

```
Step 0 : (x_0 = (0.028,0.000), u_0 = -4, r_0 = 0, x_1 = (-0.006,-0.694))
Step 1 : (x_1 = (-0.006,-0.694), u_1 = 4, r_1 = 0, x_2 = (-0.093,-1.037))
Step 2 : (x_2 = (-0.093,-1.037), u_2 = -4, r_2 = 0, x_3 = (-0.241,-1.968))
Step 3 : (x_3 = (-0.241,-1.968), u_3 = -4, r_3 = 0, x_4 = (-0.486,-2.867))
Step 4 : (x_4 = (-0.486,-2.867), u_4 = -4, r_4 = 0, x_5 = (-0.772,-2.649))
Step 5 : (x_5 = (-0.772,-2.649), u_5 = -4, r_5 = -1, x_6 = (-1.001,-1.930))
Step 6 : (x_6 = (-1.001,-1.930), u_6 = -4, r_6 = 0, x_7 = (-1.001,-1.930))
Step 7 : (x_7 = (-1.001,-1.930), u_7 = 4, r_7 = 0, x_8 = (-1.001,-1.930))
Step 8 : (x_8 = (-1.001,-1.930), u_8 = -4, r_8 = 0, x_9 = (-1.001,-1.930))
Step 9 : (x_9 = (-1.001,-1.930), u_9 = -4, r_9 = 0, x_10 = (-1.001,-1.930))
Step 10 : (x_10 = (-1.001,-1.930), u_10 = -4, r_10 = 0, x_11 = (-1.001,-1.930))
```

Figure 1: Trajectory generated with random policy

At step 5 the car reaches a terminal state (indeed, $|p_6| > 1$) so the system is then stuck (position and speed of the car don't change) and the rewards are always 0 after that step.

2 Expected return of a policy in continuous domain

Since we can't have truly an infinite time horizon, we must approximate J^{μ}_{∞} by computing it over N steps. The error we obtain with this approximation is bounded, we have

$$||J^{\mu}_{N} - J^{\mu}_{\infty}||_{\infty} \leq \frac{\gamma^{N}}{1-\gamma} B_r$$

For different values of N we get the following bounds

N	Bound
100	0.11
200	$7 * 10^{-4}$
300	$4.15 * 10^{-6}$
400	$2.46 * 10^{-8}$
500	$1.45 * 10^{-10}$

Table 1: Bound value for different values of N

We decided to take 400 as value for N, because we have a good bound on the error ($2.46 * 10^{-8}$) which is really accurate, and 400 steps will be done really quickly so it's a good choice.
On the Figure 2 we can observe the average cumulative expected return over 400 steps and over 50 episodes (with random initial position). It decreases under 0 after few steps which means that the case where p < -1 or |s| > 3 happens quickly for some episodes. There

is no increase of the expected cumulative return, which means that in our 50 episodes, there is no case where the car successes going up the hill. Indeed it is the harder case to reach because the car should back up and then accelerate non-stop to counter the slope of the hill. With a random policy, the car has much more chance to go in the case where p < -1 or |s| > 3, and so the case where it gets a negative reward.
We see that after approximately 60 time steps the expected cumulative return becomes constant, which means that the car of each episode has reached a terminal state (or one has still not reached a terminal state in 400 steps, which is nearly impossible with a random policy).

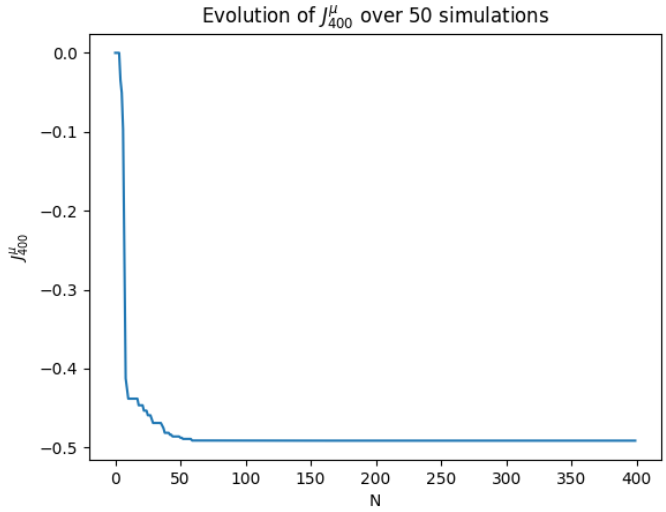

Figure 2: Average J^{μ}_{400} over 50 simulations

3 Visualization

For the visualization part, we use the script given in the statement 300 times (this number is chosen based on the observations of the section 2, since after 60 time steps there is no change, so 300 is largely enough to see all the car dynamics) with each time an updated position, for a random policy. It thus produces 300 images, and then we make a GIF from those 300 images, the GIF is in our deliverable archive. It took 8.13s to generate those 300 images.
With a lot of chance, we see in the GIF that the car manage to climb the hill with a random policy.

4 Fitted Q-iteration

To estimate the Q_N-functions, one can use the fitted Q-iteration algorithm where we use samples of one-step system transition to learn the $\hat{Q}_N$-functions. $\hat{Q}_1$ is learned using the

following dataset :

$$LS_{\hat{Q}_1} = \left\{ \left((p^i, s^i), u^i \right), r^i \right\}_{i=1}^{N_b}$$

where $\left((p^i, s^i), u^i \right)$ are the inputs and r^i is the output. The next datasets used to learn $\hat{Q}_N$ have the following form :

$$LS_{\hat{Q}_N} = \left\{ \left((p^i, s^i), u^i \right), r^i + \gamma \cdot \max_{u' \in U} \hat{Q}_{N-1}(y^i, u') \right\}_{i=1}^{N_b}$$

Two strategies have been implemented in order to generate sets of one-step system transitions :

- ❋ The first idea was to generate trajectories starting from the given initial state $p_0 \sim \mathcal{U}([-0.1, 0.1])$, $s_0 = 0$ and using a random policy that stops when we reach a terminal state.
- ❋ The second idea, which is an improvement of the first one, is the following : instead of starting from the given initial position, we select as initial position a non-terminal random position, i.e between -1 and 1. The main goal of this choice is to avoid grey-areas, so one-step transitions on which we would not have information.

For each strategy, we simulate 60 episodes and we collect the associated transitions. A trajectory (i.e an episode) stops when either we have reached a terminal state or when the length of the trajectory is equal to 100, in order to avoid repeating patterns.

We also have implemented two stopping rules for the computation of the $\hat{Q}_N$-functions :

- ❋ As seen during the theoretical course, one can stop the iteration algorithm when the infinite norm between Q_N and Q_{N-1} drops under a certain threshold ρ (set to 0.08) such that

  $$||Q_N - Q_{N-1}||_\infty \leq \rho$$

  As we have seen in the course, taking the infinite norm is more reliable than L2-norm because the algorithm could stop very quickly if the trajectory is mainly composed of states in the region of the initial state. Indeed, the algorithm needs a certain number of iterations before obtaining values of $\hat{Q}_N$ around this region different from 0.
- ❋ Another theoretical stopping rule is to select N such that :

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

  where $\hat{\mu}_N^*$ is the optimal policy after N iterations obtained from $\hat{\mu}_N^* = \underset{u \in U}{\text{argmax}} \hat{Q}_N(x, u)$, $\hat{\mu}^*$ is the optimal policy after an infinity of iterations, $\gamma = 0.95$ and $Br = 1$. Indeed, the right-hand-side of the previous equation is an upper bound on the error made

on the expected reward following the estimated optimal policy and it's a strictly decreasing function depending on N. Thus we just need to fix a threshold limit (0.01 in our case leading to $N = 221$).

There is several supervised learning methods to learn the $\hat{Q}_N$-functions and we will present the results obtained with three different methods, using the second strategy to build our learning sets and the second stopping rule.

4.1 Linear regression

The first supervised machine learning algorithm used to learn the $\hat{Q}_N$-functions is the linear regression. There are no hyper-parameters to tune. Figure 3 and 4 show $\hat{Q}_N$ for each action $u = -4$, $u = 4$ and with a resolution of 0.01 on the domain. We observe that the values obtained are always bigger when taking action $u = 4$. This observation is confirmed by Figure 5 which shows the estimated optimal policy $\hat{\mu}_N^* = \underset{u \in U}{\text{argmax}} \hat{Q}_N(x, u)$.

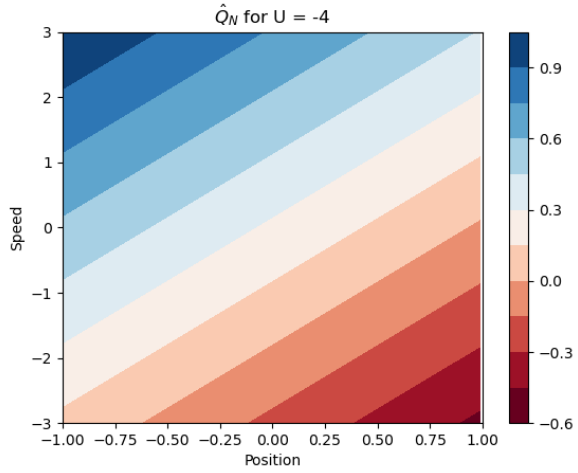

Figure 3: $\hat{Q}_N$ for linear regression, $u = -4$

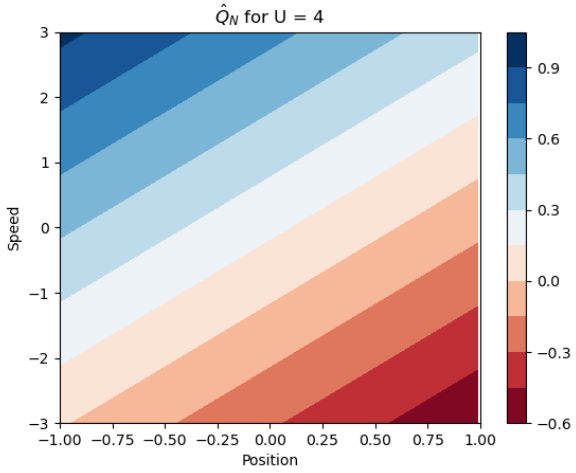

Figure 4: $\hat{Q}_N$ for linear regression, $u = 4$

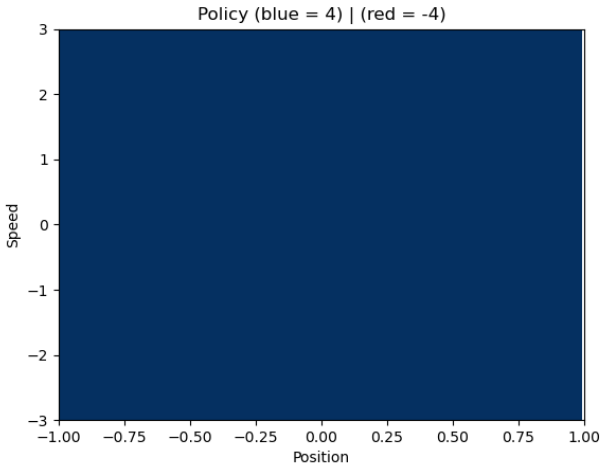

Figure 5: Policy derived from $\hat{Q}_N$, linear regression

The policy obtained with this linear regression is not good at all, leading to an expected return equal to 0. Indeed, with such a policy, the car cannot climb the hill. In conclusion this algorithm does not perform better results than a simple policy (e.g. always play 4).

4.2 Extremely Randomized Trees

The second supervised machine learning algorithm used is the extremely randomized trees. This algorithm is well suited for reinforcement learning problems as we can see in the related paper read in class. The number of estimator is set to 10. Indeed more estimators do not provide better result for the Car-on-the-Hill problem as shown in the research paper. Moreover if the number of estimator is bigger, the computation time will be longer.
Figure 6 and 7 show $\hat{Q}_N$ for each action $u = -4$, $u = 4$ and with a resolution of 0.01 on the domain. As can be seen in the Figure 8 the policy extracted using Extremely Randomized Trees is much more complex than the one extracted from a linear regression. The red area located in $p = [-0.75, 0.2]$ and $s = [-1.5, 1]$ indicates that the car first backs up and then accelerate in order to have a bigger acceleration to climb the hill.
The expected return of the policy is equal to 0.395 and the number of time steps required to climb the hill and reach a positive terminal state is about 21-22. We can conclude that extremely randomized trees is a very powerful algorithm to learn Q-functions.

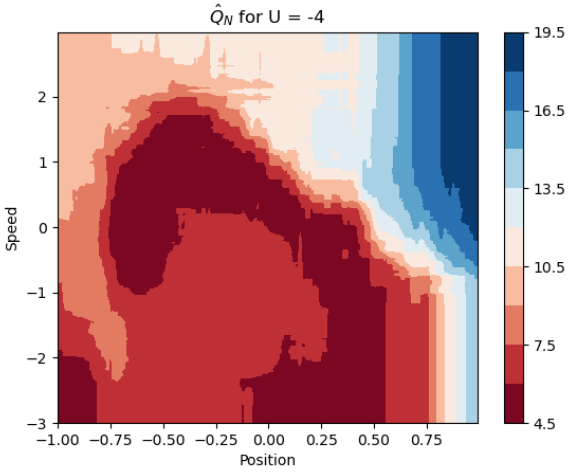

Figure 6: $\hat{Q}_N$ for Extremely Randomized Trees, $u = -4$

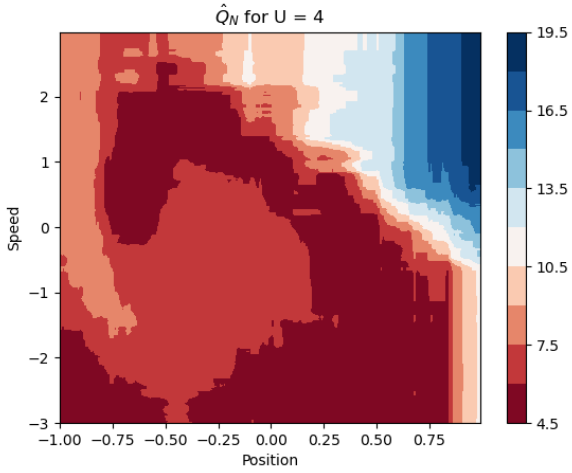

Figure 7: $\hat{Q}_N$ for Extremely Randomized Trees, $u = 4$

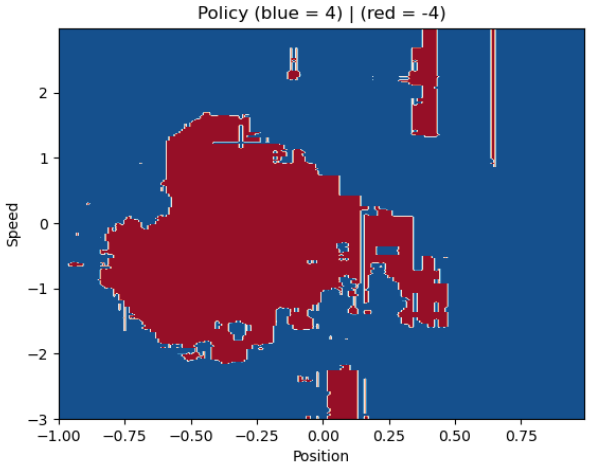

Figure 8: Policy derived from $\hat{Q}_N$, Extremely Randomized Trees

4.3 Neural network

The last supervised learning algorithm experimented is the neural network. We do not need deep neural networks because the function to approximate is not strongly complex and there are only 3 features as inputs to the network. The hard task is to design it since we do not have any particular information on which structure would be the best. We tried several possibilities essentially depending on the number of hidden-layers and the number of neurons in each of these hidden-layer. The chosen architecture is the following one :

- ❋ Input layer : 3 features (position, speed, action)
- ❋ Hidden-layers : 4 layers composed of 20/20/20/10 neurons

❉ Output layer : The value of $\hat{Q}_N$
❉ Activation function : tanh

Figure 9 and 10 show that the levels of $\hat{Q}_N$ are much more smooth than the ones for Extra trees. In a similar way, Figure 11 shows a smoothness contour for neural network policy whereas Extra trees policy is more disturbed. The expected return of the neural network policy is 0.369 which is a little lower than Extra trees. It corresponds to a rise of the hill in 20 time steps.
We observe that the red area of the policy has the same pattern as Extra trees policy except in the bottom left. About the $\hat{Q}_N$ value, we see that the top right corner is associated to highest value which is expected because this area of states (position, speed) is closed to the positive terminal state. For action $u = -4$, we also see that the top left corner is associated to a high value. Indeed, the car must first backs up and then accelerate to climb the hill.

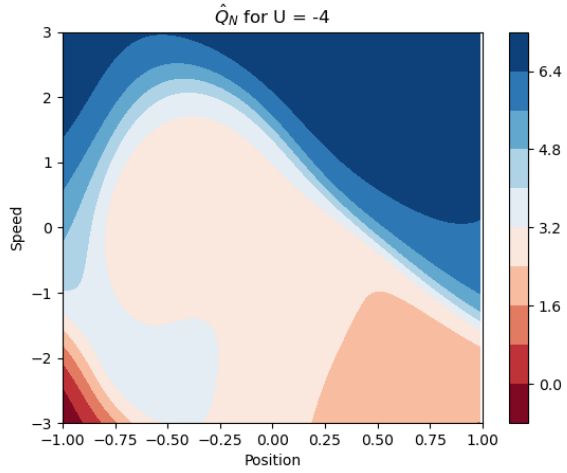

Figure 9: $\hat{Q}_N$ for Neural network, $u = -4$

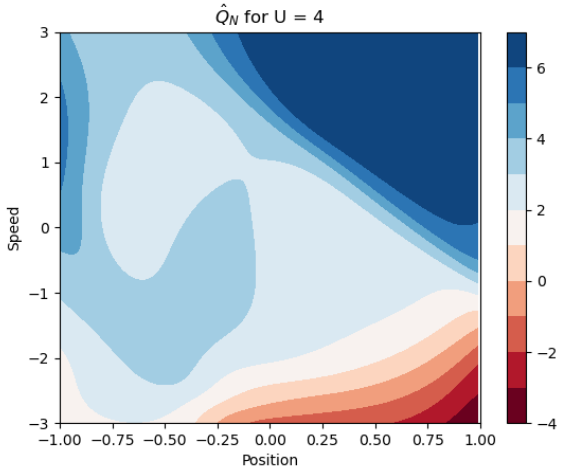

Figure 10: $\hat{Q}_N$ for Neural network, $u = 4$

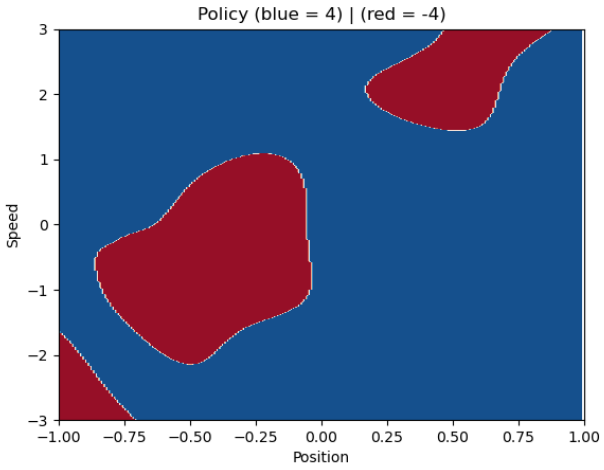

Figure 11: Policy derived from $\hat{Q}_N$, Neural network

4.4 Results and discussion

In terms of computation time, the linear regression is by far the fastest. The computation time of neural networks strongly depend on its complexity (number of layers and number of neurons in layers) while the one of Extremely Randomized Trees depends on the number of estimators.

In conclusion, Extremely Randomized Tree is the best technique for fitted-Q iteration because it allows to obtain very good results while being easy to use and having not too long computation times. We could probably obtain equivalent results with a well design neural network but it takes much more time to tune it.

Table 2 and 3 shows the expected return of the estimated optimal policy considering all supervised learning algorithms, the different stopping conditions and one-step transitions generation strategies.

	Stop condition 1	Stop condition 2
$J^{\hat{\mu}^*}_{lin\_reg}$	0.0	0.0
$J^{\hat{\mu}^*}_{trees}$	0.0	0.0
$J^{\hat{\mu}^*}_{nn}$	0.0	0.0

Table 2: Expected return of $\hat{\mu}^*_N$, 1^{st} generation strategy (60 episodes)

We observe that all the expected return in Table 2 are equal to 0 and can be explained by the small number of episodes used with this generation strategy. Indeed, the special terminal states are hard to reach starting from $p \in [-0.1, 0.1]$ and following a random policy.

	Stop condition 1	Stop condition 2
$J^{\hat{\mu}^*}_{lin\_reg}$	0.0	0.0
$J^{\hat{\mu}^*}_{trees}$	0.0	0.395
$J^{\hat{\mu}^*}_{nn}$	0.0	0.369

Table 3: Expected return of $\hat{\mu}^*_N$, 2^{nd} generation strategy, (60 episodes)

To discuss the impact of the one-step transitions generation strategies, the first technique will lead to more identical samples because the different episodes start in a smaller space which is in the bottom of the hill. Thus it will be hard for one episode to explore some hardly-accessible states (e.g the top of the hill) which requires some ordered actions to reach them. For these reasons, the second technique is better because on a small number of episodes, it will cover a larger space of the domain and thus collect more information.
However, for a high number of episodes, the first technique will also be able to cover the whole domain such that both techniques would lead to equivalent results. But it would require a longer computation time as we increase the number of episodes so the second technique is a better choice.

About the stopping rules, the first stopping condition is more time-consuming than the second one because you need to make predictions with the current $\hat{Q}$ model and the previous one whereas the second stopping condition only depends on the fixed upper bound. In addition, there is no guarantee that the supervised algorithm converges and thus we could wait in an infinite loop. The second stopping condition is not practical but theoretical and we are sure that the algorithm stops at some point depending on the fixed upper bound. However, this rule does not take into account the fact that a previous iteration could be a better estimate of Q_N than the current one.

5 Parametric Q-learning

5.1 Algorithm

From the previous sections, we have seen that that we can infer $\hat{Q}$ from a set of one-step transitions using the fitted-Q iteration algorithm. Another algorithm called parametric Q-learning is an other estimator of the Q-function and has the form $\hat{Q}(x, u, \theta)$.

5.1: Algorithm 1 - Parametric Q-learning (PQL)

1. Initialize $\hat{Q}(x, u, \theta)$ everywhere.
2. While the stopping criterion is not met, update the parameters θ as :

$$\theta \leftarrow \delta + \alpha\delta\left(x, u, r, x'\right)\frac{\partial\hat{Q}(x, u, \theta)}{\partial\theta}$$

where (x, u, r, x') are one-step transitions of a trajectory h and

$$\delta\left(x, u, r, x'\right) = r + \gamma\max_{u'\in U}\hat{Q}\left(x', u', \theta\right) - \hat{Q}(x, u, \theta)$$

which is the temporal difference.

This algorithm updates the parameter θ in a way similar to stochastic gradient descent. The difference is that, with PQL, we may not use multiple times the whole training set, in contrary with gradient descent.
But we do use it several times, using the Monte Carlo principle as explained in the section 2 of this project, since the initial state is generated randomly it allows us to counter this randomness.
θ is updated at each transition which is quite slow as the number of transitions increases. We set the learning rate α arbitrarily to 10^{-2}

To compare properly the fitted Q-iteration and the Parametric Q-learning we must use the same architecture for the neural network as in the section 4, so :

- ❋ Input layer : 3 features (position, speed, action)
- ❋ Hidden-layers : 4 layers composed of 20/20/20/10 neurons
- ❋ Output layer : The value of $\hat{Q}_N$
- ❋ Activation function : tanh

The FQI is performed with the second stopping criterion, the one based on the theoretical bound on the infinite norm between $J^{\hat{\mu}^*_N}$ and $J^{\hat{\mu}^*}$ (see section 4).
The optimal policy is then inferred for both algorithm in the same way than in section 4, namely $\hat{\mu}^*_N = \underset{u\in U}{\text{argmax}}\hat{Q}_N(x, u)$

5.2 Results

As we can see in the Figure 12, the results are pretty bad with a trajectory of length 1000. Indeed, looking to the colored 2D grid plotting the optimal action to take with respect to the actual state of the game, the agent must frequently decelerate which doesn't help it to go on the top of the hill.

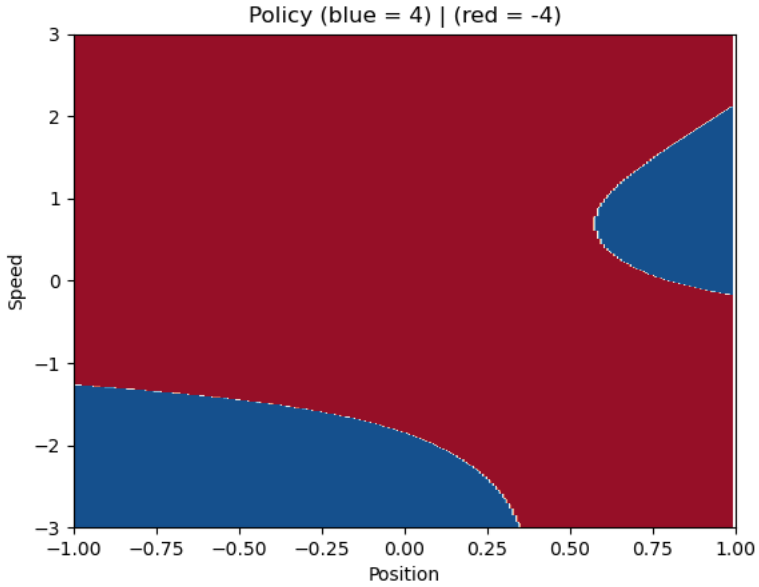

Figure 12: Estimated policy obtained with Parametric Q-learning

The expected reward with the estimated policy is -0.438, which means that the car generally goes too fast, or goes in the position at the extreme left.
On the Figure 13 and the Figure 14 are the values of the $\hat{Q}_N$ for respectively u = -4 and u = 4.

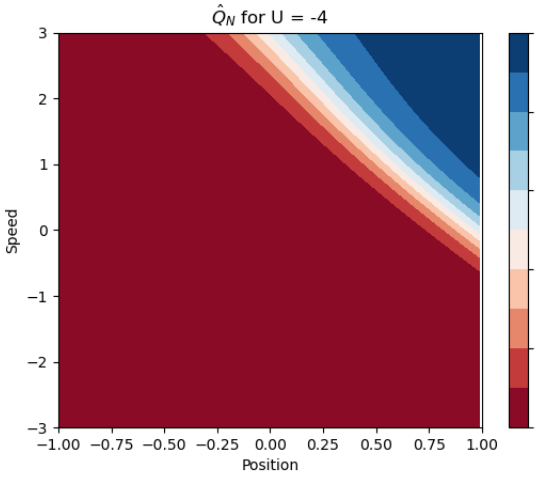

Figure 13: $\hat{Q}_N$ obtained with PQL, $u = -4$

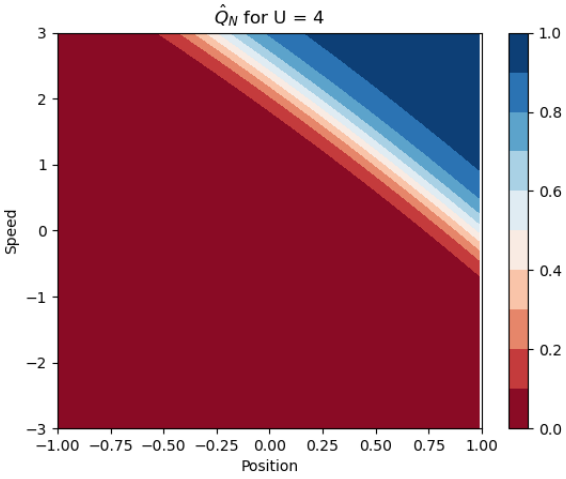

Figure 14: $\hat{Q}_N$ obtained with PQL, $u = 4$

Of course, closer the car is to the top of the hill with a correct speed, higher the $\hat{Q}_N$ is since the car is going to reach the top and thus have the positive reward.

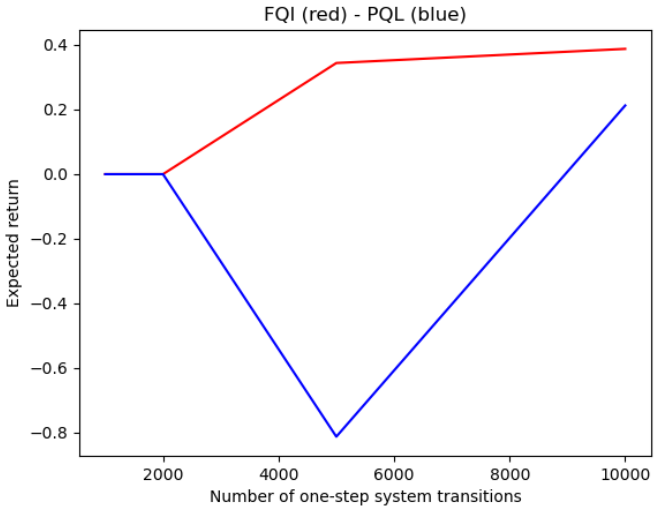

Figure 15: Expected return over different number of transitions, for FQI vs PQL

We also plotted in the Figure 15 the expected return of the policy, for different number of one-step transitions. The one obtained with PQL is always smaller than with FQI, which isn't surprising seeing the results we had with PQL just above.
The PQL algorithm just doesn't succeed to find a good optimal policy. It could be expected as PQL is known to perform poorly when used like that. However, when the number of transitions increases, we clearly see that the expected return increases as-well. To be sure that it would converge to the optimal return, we should also increase the number of transitions but we ran out of time as the neural network is time-consuming to train.

6 Bonus : Normalized parametric Q-learning

The difference with parametric Q-learning is in the way we update the parameters. Instead of having

$$\theta \leftarrow \delta + \alpha\delta\left(x, u, r, x'\right)\frac{\partial\hat{Q}(x, u, \theta)}{\partial\theta}$$

as before, we now have

$$\theta \leftarrow \delta + \alpha\frac{k}{||k||_2}$$

where $k = \delta\left(x, u, r, x'\right)\frac{\partial\hat{Q}(x,u,\theta)}{\partial\theta}$, so that the update term is divided by its 2 norm.
We thus build an online Q-iteration algorithm with the normalized update term. We use a replay buffer to prevent our algorithm from correlated samples. Figure 16 shows that the normalized parametric Q-learning does not improve our results. Moreover, we could not train the neural network with higher number of transitions, again caused of time.

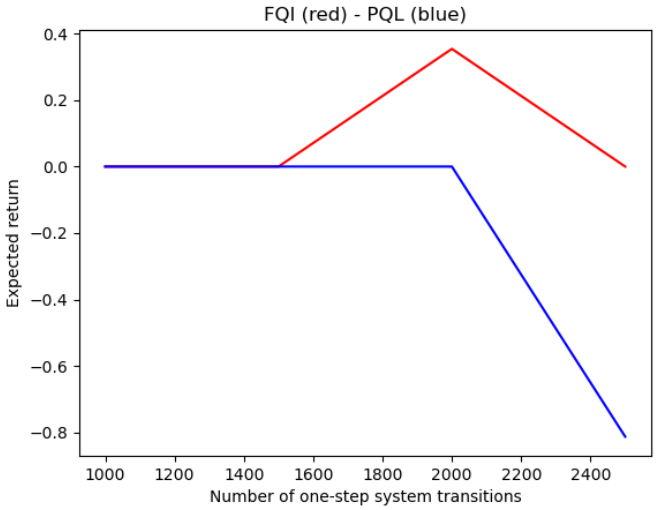

Figure 16: Expected return over different number of transitions with normalized gradient, for FQI vs PQL