Université de Liège
Master 1 - Data Science and Engineering
Année 2020-2021

Optimal decision making for complex problems

Assignment 2 - Part 1

INFO8003-1 Optimal decision making for complex problems
ERNST Damien
AITTAHAR Samy
BARDHYL Miftari

PIRLET Matthias
s172330
CHRISTIAENS Nicolas
s171216

Liège
Tuesday, 23 February 2021

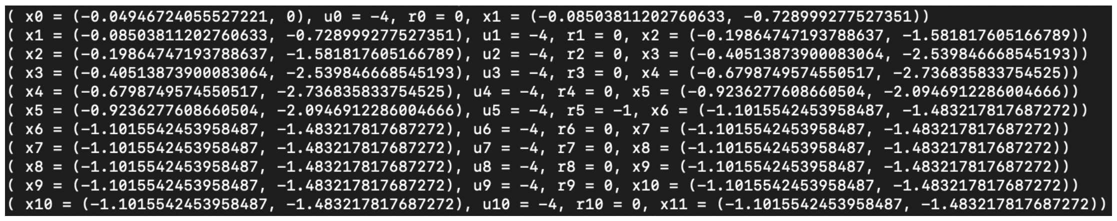
1 Implementation of the domain

We implemented the domain as a class with several attributes as the mass of the car m, the gravitational constant g, the discrete time between t and t+1, the integration time step, the discount factor and finally the different possible actions. We also implemented multiple methods as the Hill function and its 2 derivatives needed to compute the dynamics of the system. A method to get the next state given an action and the actual state, another to get the reward of the next state which is always 0 except when we are in a terminal state but all rewards that we get after the one of the terminal state must be 0 too. And so a final method to know if we are in a terminal state which is the case when the position of the car in absolute value is greater than 1 or if the speed of the car again in absolute value is greater than 3.

The policy we choose to implement consists in always accelerate to the left (i.e always select the action "-4"). We have chosen this policy because it is easy to see if our results are coherent compared to a policy in which you select an action at random where it is more difficult to interpret.
We see that with our policy, we will always have a reward of -1 and another interesting result here is that :

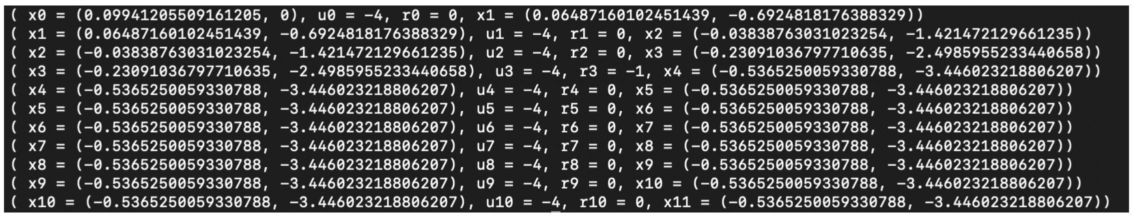
- if the initial position is negative : the reward is -1, always at the 6^{th} transition and because of the position of the car (p < -1)
- if the initial position is positive : the reward is -1, always at the 4^{th} transition and because of the speed of the car (s < -3)

We can maybe interpret that as if we start at a positive position, the car is upper on the hill and then as we accelerate to the left, its potential energy (i.e its upper position) is transformed into kinetic energy (i.e speed). So it is normal that we reach a terminal state because of the speed while when we start at a negative position, the car is lower on the hill and then we reach after some iterations the terminal state because of its position because it has not enough potential energy to exceed the maximum speed. You can fin in the 2 figures below the 2 scenarios :



```
( x0 = (-0.04946724055527221, 0), u0 = -4, r0 = 0, x1 = (-0.08503811202760633, -0.728999277527351))
( x1 = (-0.08503811202760633, -0.728999277527351), u1 = -4, r1 = 0, x2 = (-0.19864747193788637, -1.581817605166789))
( x2 = (-0.19864747193788637, -1.581817605166789), u2 = -4, r2 = 0, x3 = (-0.40513873900083064, -2.539846668545193))
( x3 = (-0.40513873900083064, -2.539846668545193), u3 = -4, r3 = 0, x4 = (-0.6798749574550517, -2.736835833754525))
( x4 = (-0.6798749574550517, -2.736835833754525), u4 = -4, r4 = 0, x5 = (-0.9236277608660504, -2.0946912286004666))
( x5 = (-0.9236277608660504, -2.0946912286004666), u5 = -4, r5 = -1, x6 = (-1.1015542453958487, -1.483217817687272))
( x6 = (-1.1015542453958487, -1.483217817687272), u6 = -4, r6 = 0, x7 = (-1.1015542453958487, -1.483217817687272))
( x7 = (-1.1015542453958487, -1.483217817687272), u7 = -4, r7 = 0, x8 = (-1.1015542453958487, -1.483217817687272))
( x8 = (-1.1015542453958487, -1.483217817687272), u8 = -4, r8 = 0, x9 = (-1.1015542453958487, -1.483217817687272))
( x9 = (-1.1015542453958487, -1.483217817687272), u9 = -4, r9 = 0, x10 = (-1.1015542453958487, -1.483217817687272))
( x10 = (-1.1015542453958487, -1.483217817687272), u10 = -4, r10 = 0, x11 = (-1.1015542453958487, -1.483217817687272))
```

Figure 1: Trajectory with a negative initial position



```
( x0 = (0.09941205509161205, 0), u0 = -4, r0 = 0, x1 = (0.06487160102451439, -0.6924818176388329))
( x1 = (0.06487160102451439, -0.6924818176388329), u1 = -4, r1 = 0, x2 = (-0.03838763031023254, -1.421472129661235))
( x2 = (-0.03838763031023254, -1.421472129661235), u2 = -4, r2 = 0, x3 = (-0.23091036797710635, -2.4985955233440658))
( x3 = (-0.23091036797710635, -2.4985955233440658), u3 = -4, r3 = -1, x4 = (-0.5365250059330788, -3.446023218806207))
( x4 = (-0.5365250059330788, -3.446023218806207), u4 = -4, r4 = 0, x5 = (-0.5365250059330788, -3.446023218806207))
( x5 = (-0.5365250059330788, -3.446023218806207), u5 = -4, r5 = 0, x6 = (-0.5365250059330788, -3.446023218806207))
( x6 = (-0.5365250059330788, -3.446023218806207), u6 = -4, r6 = 0, x7 = (-0.5365250059330788, -3.446023218806207))
( x7 = (-0.5365250059330788, -3.446023218806207), u7 = -4, r7 = 0, x8 = (-0.5365250059330788, -3.446023218806207))
( x8 = (-0.5365250059330788, -3.446023218806207), u8 = -4, r8 = 0, x9 = (-0.5365250059330788, -3.446023218806207))
( x9 = (-0.5365250059330788, -3.446023218806207), u9 = -4, r9 = 0, x10 = (-0.5365250059330788, -3.446023218806207))
( x10 = (-0.5365250059330788, -3.446023218806207), u10 = -4, r10 = 0, x11 = (-0.5365250059330788, -3.446023218806207))
```

Figure 2: Trajectory with a positive initial position

2 Expected return of a policy in continuous domain

As we know the reward function and the dynamics of the environment here, we can use dynamic programming. We have :

$$J_N^\mu(x) = r(x, \mu(x)) + \gamma J_{N-1}^\mu(f(x, \mu(x))), \qquad \forall N \geq 1$$

The reward signal $r(x, \mu(x))$ here is always equal to 0 except when we arrive in a terminal state. This means that we have **only** one reward signal different from 0.
We also have to choose N carefully to approximate well the infinite time horizon of the policy. Not too big otherwise it will take too much time to compute and if it is too small then the estimation will be far from the true value. The value has to be taken in order to be negligible in comparison with the order of magnitude of the values found in our simulations (order of 10^-2). This error can be bounded by :

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

where γ is a constant representing the decay factor (which is equal to 0.95) and B_r is another constant representing the largest reward value (which is equal to 1 in our case). The values of this bound can be found in the table below :

N	Bound value
10	11.9747
100	0.1184
500	$14.5489 * 10^{-11}$

Table 1: Values of the bound depending on N

We implemented 2 different functions that plot the expected return of our "always accelerate on the right" policy. The first one is implemented having only 50 different initial positions, one for each of the 50 simulations and the second is implemented having 50 different initial positions for **each** N and so we have in total 50*N different initial positions. The plots can be seen in Figure 3 and Figure 4 below :

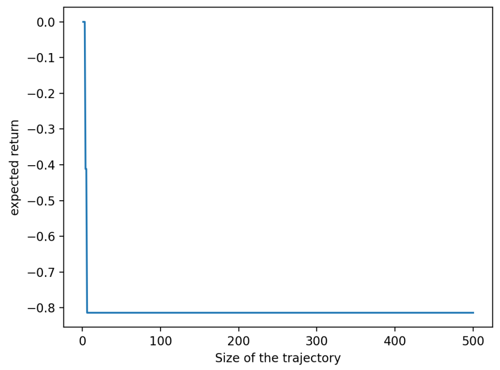


Figure 3: Expected return over 50 simulations using the first method

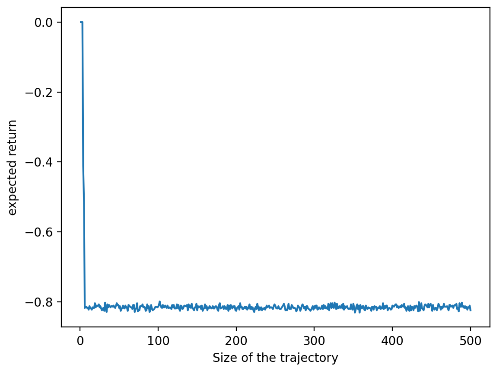


Figure 4: Expected return over 50 simulations using the second method

This result is the one expected because if, in a simulation, we use the same initial position, we will only have one expected return value which will be different from 0 for one N and all the others will be

0 and so at the end we will have a perfect line. While if we take a different initial position for all N, we could have 2 different values for the expected return of 2 different N and so it is no more a straight line.

3 Visualization

To implement this section we created one image per step in our trajectory. We only generated 10 images because after 6 images the image stays the same and do not change anymore because the car is in a terminal state.

4 Fitted-Q-Iteration

We have implemented 2 strategies for generating sets of one-step system transitions :

- A naive reasoning for collecting tuples would be to generate from the given starting position trajectories with a random policy that stops once an final state is reached.
- A possible improvement of this technique would be to no longer take the starting point as the starting point provided (uniform between -0.1 and 0.1) but to start at a non-terminal random position (uniform between -1 and 1). This in order to perhaps prevent gray areas on certain pairs (p, s) on which we have no information.

For each of these strategies we run N simulations and we collect the transitions of these simulations. The simulation stops when a final state is turned off or when 100 transitions have been made (rarely or never happens but it is a protection). The number of one-step system transitions generated by these methods are very often between 2000 and 3000.

We have implemented 2 stopping rules for the computation of the $\hat{Q}_N$-functions sequence :

- The most intuitive way would be to see if at each step we are improving our predictions. We will therefore calculate the difference between Q_N and Q_{N-1} on the training set. We would then stop the algorithm when the improvement between each iteration is lower than a fixed threshold. The problem with this method is that there is no guarantee that there is an improvement between each iteration for all the supervised learning techniques. And so we could end up with endless iterations.
- To overcome the potential problem of the first method, we will move towards a more theoretical stopping criterion. We know that the following equation bound on the suboptimality of $\hat{\mu}^*_N$ with respect to μ^* :
$$||J^{\hat{\mu}^*_N} - J^{\hat{\mu}^*}||_\infty \leq \frac{2 * \gamma^N * Br}{(1-\gamma)^2}$$
with $Br = 1$, $\gamma = 0.95$ in our case and with $\hat{\mu}^*_N$ which is the optimal policy after N iterations of the algorithm and μ^* after an infinity of iterations. By fixing a tolerance threshold we can limit the number of iterations N. Indeed, the right hand side is a strictly decreasing function and so it will pass at some point under the fixed tolerance threshold.

For the neural network the hardest task was to design it since we had no knowledge on which structure would be the best (deep learning is given this semester). So we tested several possibilities (one or more hidden layers with a few or a lot of neurons in it) but the computation time being extremely high, we were not able to test many possibilities.
Our final neural network structure is composed of :

- First layer : 3 inputs are needed for p , s and u.
- Hidden layers : We found that 5 hidden layers of 9 neurons each gives us the best result (with tanh for activation function).
- Last layer : only 1 output is needed for $\hat{Q}_N$.

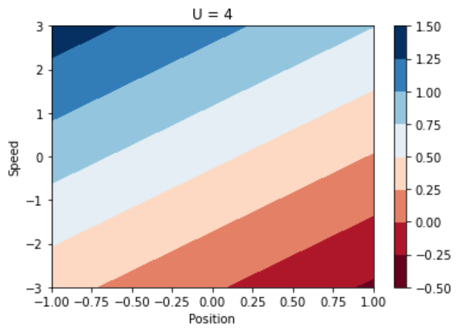


Figure 5: $\hat{Q}_N$ linear regression for action = 4

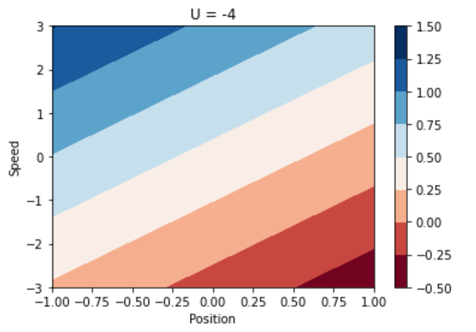


Figure 6: $\hat{Q}_N$ linear regression for action = -4

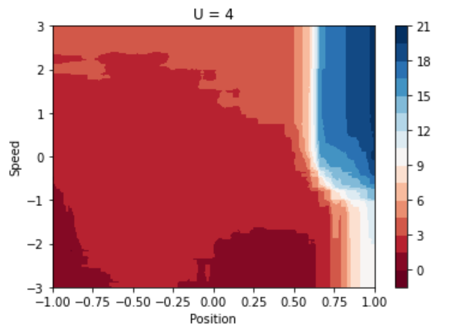


Figure 7: $\hat{Q}_N$ Extremely Randomized Trees for action = 4

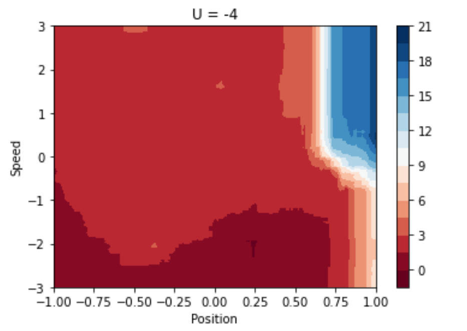


Figure 8: $\hat{Q}_N$ Extremely Randomized Trees for action = -4

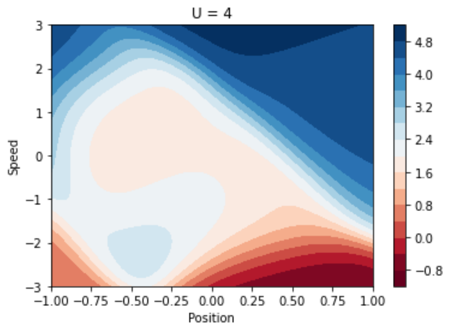


Figure 9: $\hat{Q}_N$ Neural Network for action = 4

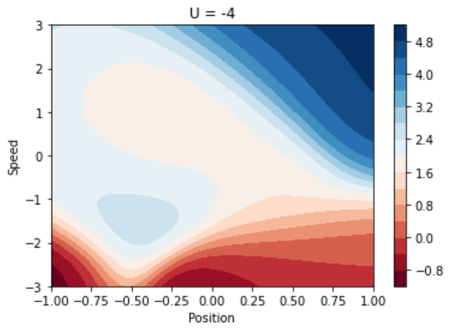


Figure 10: $\hat{Q}_N$ Neural Network for action = -4

The policy $\hat{\mu}_N^*$ can be easily derived from $\hat{Q}_N$ N for each supervised learning algorithm :

$$\hat{\mu}_N^*(x) \in argmax_{u \in U} \hat{Q}_N(x, u)$$

Which can be represented with these grids :

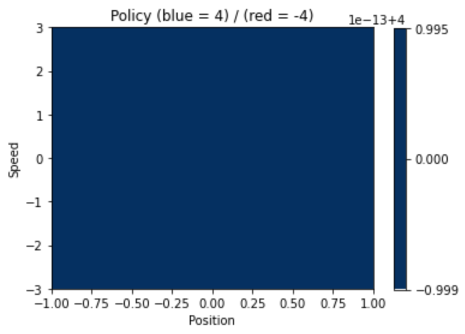


Figure 11: Policy derived from $\hat{Q}_N$ for linear regression

with a expected return equals to $1.5 * 10^{-11}$.

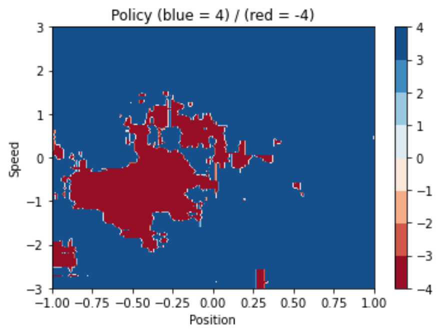


Figure 12: Policy derived from $\hat{Q}_N$ for Extremely Randomized Trees

with a expected return equals to 0.382.

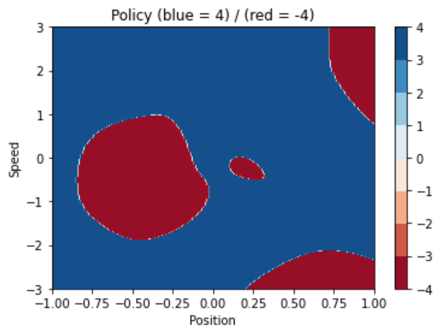


Figure 13: Policy derived from $\hat{Q}_N$ for Neural Network

with a expected return equals to 0.375.

On these results we can clearly see in these figures that linear regression is not at all adapted to

a problem like this one and therefore the expected value does not exceed 0 because the policy cannot overcome the obstacle (the rise of right). On the other hand for the Extremely Randomized Trees we have an expected value greater than 0.382 which corresponds to a rise in about 18-19 time steps. For Neural Network, we have 0.375 which is a little less good but we manage to pass the climb.
In terms of computation time, the liner regression is by far the fastest. The computation time of neural networks strongly depend on the complexity (number of layers and number of neurons in layers) of it while that 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. Maybe with a more suitable neural network we could maybe have had a result equivalent to that obtained using Extremely Randomized Tree but we are not yet familiar enough with neural networks to find it and that would be much more complicated than using Extremely Randomized Tree.

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

- For the stopping rules : To calculate the improvement between each iteration, it is therefore necessary to make predictions with the previous model and the new model obtained. Except that to do so requires predictions from 2 models which makes us consume a lot of additional resources. Which is not too disturbing for linear regression but which takes a long time for the neural network. Moreover the convergence of Q is not proven and therefore one could easily find oneself in an infinite loop if the tolerance is badly chosen. Concerning the other rule it is no longer practical but theoretical. So it does not consume additional resources and whatever the tolerance set, the algorithm will terminate at some point because fixing the tolerance amounts to fixing the number of iterations. However this rule does not take into account that a previous iteration could be a better estimate than a future iteration and therefore we could miss a better approximation.
- For the one-step system transitions generation strategies : The first technique will give a greater number of identical samples (because the simulations start in a smaller space) and will cover less well the less accessible parts (such as the rise) which requires a precise order of action to reach it. On a small number of simulations, the 2nd technique will be better because it will have less identical sample and will cover a larger area and therefore will provide more information. On the other hand, on a high number of simulations, even the 1st technique will be able to cover the whole area because there will be all the imaginable sequences (and therefore the 2 techniques will be equivalent). However, the more samples there are, the longer the computation time will be (as seen theoretically) and therefore the 2nd technique allows to obtain better results (or at the worst equivalent) in a reduced time.