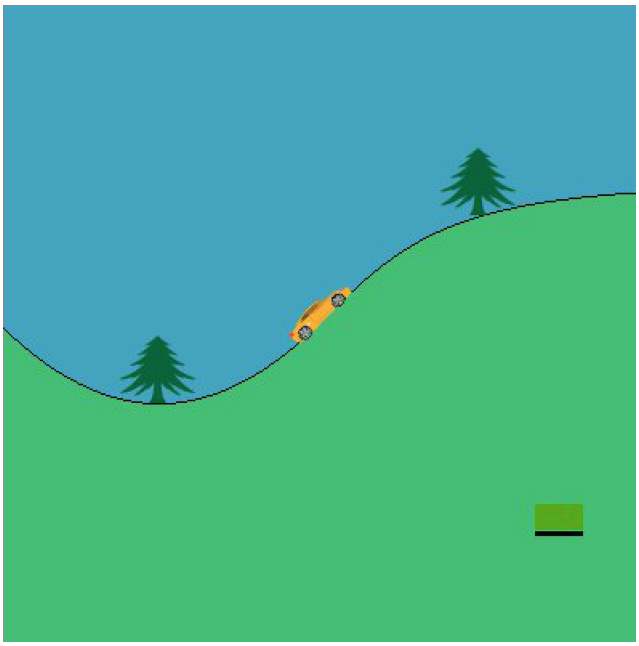
1st year Ms Engineering in Computer Science

Optimal Decision Making

Report: Assignment 2
Reinforcement Learning in a Continuous Domain

Géraldine Brieven



Academic Year 2017-2018

1 Introduction

In this second assignment, the main purpose is applying the reinforcement learning concept in a continuous domain. One of the challenges here is that we have to learn in an environment which is only partially observable, meaning that we have no direct access to the dynamic of the system and the reward function to establish an optimal policy. To deal with this, several strategies are implemented[1] and studied in this report.

2 Implementation of the domain

First, the class `Domain` is defined to represent the *car on the hill* problem. The parameters related to this problem, its dynamic and its reward function are encoded, in addition to auxiliary functions helping in the management of this domain.

Next, the class `Policy` is specified. Its main variable is the policy itself. To manage the continuity property of the state, we had to discretize its domain. Then, the policy is represented by a 2D array like exposed through Table 1:

	[-3;-2.9[	[-2.9;-2.8[	...	[2.9;3]
[-1;-0.9[	4	-4	...	4
[-0.9;-0.8[	-4	-4	...	-4
...	...	...	...	...
[0.9;1]	4	4	...	4

Table 1: Representation of a policy

Finally, to test the results by simulating some random policies, a very basic method has been implemented to visualize the evolution of the position and the speed over time and to check the coherence. One of the result is exposed through Figure 1. The position is printed, followed by an arrow illustrating the speed from this position. A point means that the speed is 0 and the longer the arrow, the higher the speed. It can be observed that the evolution of both the position and the speed are continuous and logical with respect to the actions which are token.[2]

```
==================== START ====================
Position 0      :   .   NEXT ACTION: 4.0
Position -0.0397:   <   NEXT ACTION: 4.0
Position -0.0793:   <   NEXT ACTION: 4.0
Position -0.14  :   <-   NEXT ACTION: -4.0
Position -0.2436:   <--   NEXT ACTION: -4.0
Position -0.348 :   <---   NEXT ACTION: -4.0
Position -0.4377:   <----   NEXT ACTION: -4.0
Position -0.5073:   <-----   NEXT ACTION: 4.0
Position -0.4948:   <----   NEXT ACTION: -4.0
Position -0.5068:   <-----   NEXT ACTION: 4.0
Position -0.4944:   <----   NEXT ACTION: -4.0
Position -0.5064:   <-----   NEXT ACTION: 4.0
Position -0.4941:   <----   NEXT ACTION: -4.0
Position -0.5063:   <-----   NEXT ACTION: 4.0
Position -0.494 :   <----   NEXT ACTION: -4.0
Position -0.5062:   <-----   NEXT ACTION: 4.0
Position -0.494 :   <----   NEXT ACTION: -4.0
Position -0.5062:   <-----   NEXT ACTION: 4.0
Position -0.4939:   <----   NEXT ACTION: -4.0
Position -0.5061:   <-----   NEXT ACTION: 4.0
============= END with reward 0 ==============
```

Figure 1: Simulation of a policy

3 Expected Return of a policy

Contrary to the first assignment, as mentioned before, here we don't have access to all the information about the domain, which makes more difficult the evaluation of policy! Then, the Monte Carlo principle is used.

[1] The instructions to use the code to reproduce results are exposed through the last point of this report.
[2] A smarter visualization tool will be implemented in point 4.

3.1 Explanations about the implementations

Basically, it consists of using randomness. In terms of implementation here is what we do:

1. **Generation of 1000 episodes of size 1000**
   Each episode is made up of one-step transitions like: (x, u, r, x_{next}), where r has been computed from the reward function and x_{next} directly from the dynamic of the system.[3]

   Those episodes represent our knowledge about the system (which is of course incomplete).

2. **Evaluation of the policy for each episode**
   At this stage, during a given period of time and beginning at the initial state, we simulate our policy.

   For each current state x, we compute the corresponding action u given by the policy. Then, we search among the one-step transitions (x', u', r', x'_{next}) of the current episode the one which is the closest to the pair (x, u).

   The distance metric which is used to determine this one-step transition is the following: $|p\text{-}p'| + |s\text{-}s'|$, where $x = (p, s)$ and $x' = (p', s')$. In addition to this, because the action space is only made up of only 2 elements, we impose that the actions are the same: $u = u'$ once we assimilate one-step transition (x', u', r', x'_{next}) to a given pair state-action $((p, s), u)$.

   Let's remind that the "distance approach" can be used thanks to the continuity property of the state space. If it was chaotic, this approach would be completely irrelevant.

3. **Average of the rewards derived from each episode to get the final expected return**
   The formula which is used is exactly the one which was exposed in the presentation of RaphaÃ«l Fonteneau.

3.2 Results

As expected, once some tests are performed from very basic (or random) policy, the expected return is 0, meaning that the car didn't manage to reach the top of the hill. To be able to reach this point, a more clever policy must be established. It's the purpose of the following points of this assignment.

4 Visualization

To get a better idea about the behaviour of our system once a policy is evaluated, a video is created to visualize the resulting sequence of states. To produce this video, the package *moviepy.editor* is used.

5 Fitted-Q-Iteration

5.1 Generation of one-step transitions

There are many possible ways to generate the set of one-step transitions representing the base of our learning phase. Here are the different strategies which has been studied:

- **Random generation of one-step transitions all over the state space**
  It's the most basic approach which was used in the previous part for evaluating the expected return. For this scenario, we expect a quite uniform set of information over the continuous domain. The weakness of this approach is that we have no link between the one-step transitions, meaning that, during the evaluation of Q, we have to take the closest tuple belonging to the episode, like in the Monte-Carlo approach.

- **Local generation of one-step transitions over the state space**
  This kind of training set is expected to be quite poor because the dispersion of the information about the domain is low, meaning that, even if we have a quite strong knowledge about part of the domain, some other parts are completely unknown. Then, the error on approximated Q-function are likely to be non-negligible. Moreover, here, the idea is using this approximated Q function to evaluate the next-one. Then, the fact of using a bad approximation of the Q-function would lead to an amplification of the error or, in other words, to divergence.

[3] Each of those transitions are completely independent from each others, meaning that they don't represent a continuous trajectory once their are put together.

- **Generation of one-step transitions forming a continuous trajectory**
  In this case, the next state is the current state of the next one-step transition and represents a realistic situation which could be met in reality. The weakness of this is that, contrary to the random generation, we are not sure to get information about all the state space. We may quickly fall in a final state before having explored all the state space. However, the advantage is that, here, we can directly collect each of the one-step transition having led to this reward and then, deriving a the corresponding Q value (without using the Monte-Carlo approach).

5.2 Q-functions

Let's now think about the way to represent the approximated Q-functions. In the first assignment, the idea was representing the Q-values through the following form:

	State 1	State 2	...	State m
$\hat{Q}_1$	$\hat{Q}_{11}$; best_action$_{11}$	$\hat{Q}_{12}$; best_action$_{12}$	...	$\hat{Q}_{1m}$; best_action$_{1m}$
$\hat{Q}_2$	$\hat{Q}_{21}$; best_action$_{21}$	$\hat{Q}_{22}$; best_action$_{22}$	...	$\hat{Q}_{2m}$; best_action$_{2m}$
...	...	...	...	...
$\hat{Q}_N$	$\hat{Q}_{N1}$; best_action$_{N1}$	$\hat{Q}_{N2}$; best_action$_{N2}$	...	$\hat{Q}_{Nm}$; best_action$_{Nm}$

Table 2: Representation of the $\hat{Q}_N$

For each particular cell, we stored the evaluated value of the Q-function and the best action to perform for a give state and a given number of steps.

In this second assignment, the particular feature is that the evaluated value of the Q-function is always 0, except if we completely success or completely fail. In other words, the reward function can be seen as "quite poor" because it reveals nothing about the quality of the intermediate state we go through. So the quality evaluation of those intermediate states only relies on the following term: $\gamma.\max_{u' \in U}[Q_{N-1}(x_{k+1}, u')]$. It reflects the "quality value" we can get from the next state with (N-1) steps.

Then, in the context of our problem, for each of the 2 actions, we build a table like:

	State 1	State 2	...	State m-1	State m
$\hat{Q}_1$	1	0	...	0	0
$\hat{Q}_2$	1	0	...	0	γ
...	...	...	...	...	...
$\hat{Q}_{N-1}$	1	γ^n	...	$(\text{-}\gamma)^{N-1}$	γ
$\hat{Q}_N$	1	γ^n	...	$(\text{-}\gamma)^{N-1}$	γ

Table 3: Representation of the $\hat{Q}_N$ for the car-hill problem

An element of the first line is different from 0 if it's present among the tuples of the episode which is considered and if its associated reward is 1 or -1. If no such tuple exists, it's impossible to learn anything from the episode because it means we only have reward of 0 on our disposal. Then no policy is better than another.

Moreover, once a Q-value different from 0 is assigned to a state, it doesn't change later. In this problem, the Q-value of a given state and a given action corresponds to:

$$\hat{Q}_k(x,u) = \begin{cases} \gamma^n & \text{if origin\_reward} = 1 \\ (-\gamma)^n & \text{if origin\_reward} = \text{-1} \end{cases}$$

where origin reward corresponds to the "closest" final state from x (either a failure or a success) and n is the number of remaining steps to finally reach this final state.

Then, for instance, considering this sequence of state-action leading to a failure:

$$((x_0,u_0),(x_1,u_0),(x_2,u_2),(x_3,u_3)...(x_n,u_n)) \tag{1}$$

the corresponding Q-values are:

$$Q_1(x_n,u_n) = -1 \quad ; \quad Q_2(x_{n-1},u_{n-1}) = -\gamma \quad ; ... \quad ; \quad Q_2(x_0,u_0) = (-\gamma)^n \tag{2}$$

And in the specific case of Table 3, it can be seen that it's possible to reach the top of the hill in one step from state 1 by taking the action u corresponding to this table. If we have 2 steps on our disposal, the success state can be reach from state m (by performing action u). Finally, if we can use n steps ($n < N$), the success state can be reached from state 2 (by performing action u).

But as it can be guessed, it's impossible to store all the possible states. Here, we restrict on the m states which were picked to build our episode. So we have a lack of information about our domain.

Then, the series of the Q-function is estimated by using some learning algorithms. Because we are in the particular case where the action space contains only 2 elements, we can explicitly create of model for $\hat{Q}_N(x, -4)$ and $\hat{Q}_N(x, 4)$. Once they are evaluated, the corresponding policy can be established, in such a way that, for each state, we select the action associated to the largest value of the Q-function: $u* = \underset{u' \in U}{\text{argmax}}[\hat{Q}_N(x, u')]$.

For a fixed N, the dataset which has to be provided to the algorithm is the following: $\{((p_l, s_l), Q_l)\}_{l=1}^{n}$, where (p_l, s_l) represents the state x_l.

In addition to this, following the approach of the Fitted-Q-Iteration, this dataset is updated by using the model which was built to predict the preceding Q-function. It has been decided to use it to add information about the "poor zones" of the input domain (where few samples take their input values) in such a way that we increase the dispersion factor α.

5.3 Comparison of $\hat{Q}$ from the supervised learning algorithms

Here, the predicted Q-values derived from different algorithms are compared. Of course, because the number of states is infinite, we only focus on some of them, covering all the state space. This comparison is illustrated through Figure 2 and the dataset which is provided to fit each estimator is built from one-step transitions (independent from each others).

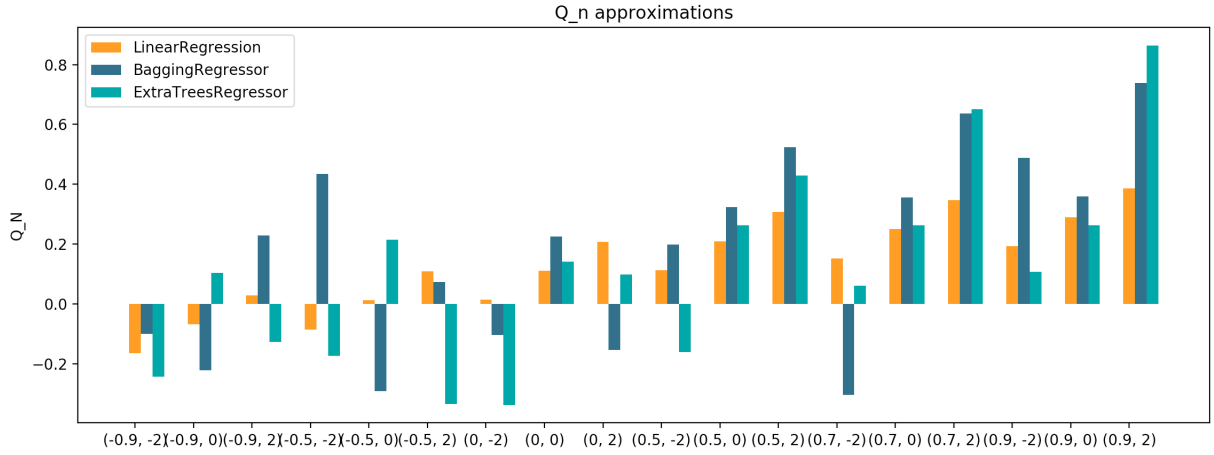


Figure 2: Comparison of different supervised learning algorithm through their evaluation of Q, for different states

From Figure 2, it can be seen that coherent results are obtained. For a state corresponding to a position closed to 1 and a speed directed to the right, the predicted Q-values are large. On the contrary, for states which are closed to the extreme left position, the estimated Q-values are globally the lowest the lowest. Finally, if, for a given state, we compare the Q-values which are predicted by the different algorithm, it can be noticed, that for some particular states, those values differ from each others.

It's important to remind those estimators are based on datasets which may be "poor" in some zones in the state space, that's why, sometimes, strange values appear.

Moreover, the Q-values may vary quite a lot from one dataset to another. Maybe it's due to the fact that N isn't large enough or simply, the one-step transitions which are picked sometimes provide less information (because they are related to a same zone of the state space for instance).

5.4 Behaviour resulting from the evaluated policy

If we have a look at the expected return (evaluated thanks to the Monte-Carlo method), it's not very relevant because it only signals the case where the car fully succeeds. Otherwise it's 0. Then we have no idea about how close we are to the optimal behaviour (or policy). That's why, to get an view on this, the visualization tool which was implemented in the preceding point is used.

From the video, it can be seen that the car acquired some knowledge about its environment.

First, it always tries to go to the right top of the hill leading to a reward of 1. And, on the contrary, it tries to avoid both reaching the left extreme position or having a speed which is too large.

In addition to this, once it's on the hill and if it doesn't manage to get enough speed to go further, it "knows" that it has to go on the bottom (and even on the other side) to gain some boost to begin trying climbing the hill again. Then, if we have an overview of the video, the car oscillates, reaching a higher point at each oscillation.

However, it can be noticed that in some zones, it tries to accelerate in vain during a few second before "understanding" it needs more boost. Maybe it's due to a "bad learning" in those zones.

6 Parametric Q-Learning

In this last part, the main idea is combining both the Bellman and the "trial-error" approaches.

6.1 Implementation Explanation

Basically, this algorithm is iterative. At each iteration, the estimated $\hat{Q}$ are updated following this equation:

$$\hat{Q}_N(x_k, u_k) \leftarrow (1-\alpha).\hat{Q}_N(x_k, u_k) + \alpha.(r + \gamma. \underset{u' \in \{-4,4\}}{\text{argmax}} [\hat{Q}_{N-1}(x, u')]) \tag{3}$$

More specifically, here are the steps which are followed:

1. Generation of 50 trajectories, each of them leading to a final states
2. Initialization of the Q values related to each state to 0
3. Iteration over the trajectories.
   - For each of them, computation of the associate Q-values according to Expressions 1 and 2.
   - Use of those new estimation to update the evaluated Q by applying Equation 3.
4. For each state, derivation of the policy by selecting the action associated the the highest Q-value (as for the Q-fitted algorithm).

6.2 Comparison with the Q-fitted algorithm

When the policy is derived from the Q-learning approach, once the visualization tool is launched to get the behaviour of the car, its knowledge about the environment seems better than once the Q-fitted algorithm.

The 2 approaches are completely different: the Q-fitted algorithm builds a model to evaluated the Q-function and then deriving the policy while the Q-learning just uses the experience which has been acquired over the environment. So the information it's based on are more reliable than the ones used by the Q-fitted algorithms which are derived from a model and not from real observations and feedback's of the environment.

Finally, in terms of computation time, the Q-learning approach is more efficient.

7 Conclusion

The general aim of the assignment has been met: several strategies have been experimented to evaluate an optimal policy in a continuous environment and machine learning algorithms have been experimented.

In this project, the machine learning approach (used by the fitted-Q algorithm) doesn't appear as being better than the Q-learning approach. Maybe it's simply due to the too low quality of the dataset which is provided to the learning algorithm. Then the resulting model isn't reliable enough.

A difficulty during this project was the fact of dealing with a continuous domain. We have a trade-off between the accuracy and the quantity of different states we have to consider. Indeed, the larger the intervals where we consider the values as being the same, the less states we have to work with and then the quicker are the computation. But for too large intervals, the resulting domain becomes irrelevant. because it contains states whose corresponding optimal action is completely different.

8 Improvement

First, A deeper study on the optimal values for the parameters has not been done. For instance, the convergence of the evaluated Q should have been studied to find the minimum N such that the evaluated Q doesn't evolve much from. Another interesting parameter whose value was intuitively fixed is the number of additional samples which are provided by using the current evaluated Q in the fitted-Q algorithm. Finally, the number of episodes and the size of each of them was also chosen in such a way that they are large enough but those parameters weren't specifically tuned.

Besides this, the Neural Network technique was handled but no significant results were obtained. It's probably due to the fact that the structure of the network wasn't appropriate. Then, no deep study was led around this supervised learning technique.

Finally, no parameter was used for the parametric Q-learning part.

9 Code Management

Here are the explanations about the way to manage the code. At the beginning of the main function, it's possible to select one specific question through its number. If it's *None*, all the questions are sequentially launched.

For *Question 2*, we just check the coherence of the successive states once a simple policy is applied.
For *Question 3*, the expected returned computed from the Monte-Carlo approach is printed.
For *Question 4*, the visualization tool is tested with a default sequence of states.
For *Question 5*, first a graph comparing the different evaluation of Q by using different algorithms is displayed. Then the expected return is computed. [4] Finally, the visualization tool is used to illustrate the behaviour of the car.
For *Question 6*, like in the preceding question, the expected return is computed and the visualization tool is used .
Finally, the values of the parameters can be easily changed. They are defined as global variables at the beginning of the file.

[4] It's often 0 because the duration of the simulation isn't long enough. It shows that the estimated policy could be better to limit the number of needed steps for reaching the top of the hill.