Master M2 MVA 2015/2016 - Reinforcement Learning - TP 4

Mountain Car

By: Lilian Besson (`lilian.besson at ens-cachan.fr`). Attached programs: The programs for this TP are included in the `zip` archive I sent, and are regular **MATLAB/Octave** programs[1].

---

1 Position of the problem

Question 0 : *What do you think the optimal value function looks like?*

The car has to go to the left at full speed, climb up, and then to the right full speed until it reaches the right side.

So the optimal value function $V^\star$ (as a 2D heat map) should be hot for negative x, regardless of the speed, and hot for positive speed and positive x (and cold for the rest).

The robot is happy if it is going into the good direction to the objective ($x \to 0.6$, $v > 0$), and has to learn to be happy if it is going to the left with high speed ($x \to -1.2, \text{abs}(v) >> 0$).

---

2 Modeling

Question 1 : *How are the characteristics of a feature function Φ stored in its corresponding vector `theta`?*

For the action[2] index $i \in \{1, 2, 3\}$, the parameters of the i-th Gaussian Φ_i are its mean μ_i and its *diagonal* covariance Σ_i, which are stored in the big vector `theta` as `mui = theta[4*(i-1)+1:2]` and `sigmai = theta[4*(i-1)+3:4]`.

For a concrete implementation of this "parameter extraction", see the attached file `phiQ.m`. It does a `switch... case... case... end` on action `a` ($\in \{-1, 0, 1\}$) and not on an index $i \in \{1, 2, 3\}$.

2.1 Some additionnal functions

- `createQ.m`: was given by the TP, uses the coefficients a_i (stored in `alpha`) and features θ_i (stored in `thetas`) to create the function Q,
- `argmax.m`: is just a manual patch on a missing function in Octave/Matlab: `[~, i] = max(v)` $\Leftrightarrow$ `i = argmax(v)`,
- `valueFunction.m`: uses the idea given in the TP:

  ```
  V = @(s) max(arrayfun(@(a)Q(s,a), [-1,0,1]))
  ```

  in order to return the associated Value function V for a Q function (in a general setting the `[-1,0,1]` should be replaced by $\mathcal{A}$ the action space, finite or not).
- `policy.m`: does the same but with an `argmax` and not a `max`, in order to get the policy: from state s to the action $\pi(s) = a$ with the highest value (ie. π is the **greedy** policy according the V).

[1] Note: I only tested my programs with GNU Octave, but both version 3 and 4 should work, on Windows or on Linux.
[2] For the action set $\mathcal{A} = \{-1, 0, 1\}$ in **this** order, see `actionPossib` in `fitted_q.m` and `LSTD.m`, we simply have $a = i - 2$.

Therefore, we are able to create an initial guess of the Q-function, with this snippet (head of `mainTP4.m`):

```
d = 20;  % Number of features, arbitrary choice!
thetasQ = rand_featureQ(d);    % Random features
alpha = ones(1, d) / d;        % Arbitrary initial weights, sum to 1
Q = createQ(alpha, thetasQ);  % Random Q function
```

We can display (with the given function `plotf.m`) the value function V^Q (given by Q), and the greedy policy π^V (given by V), see Figures 1, 2 below:

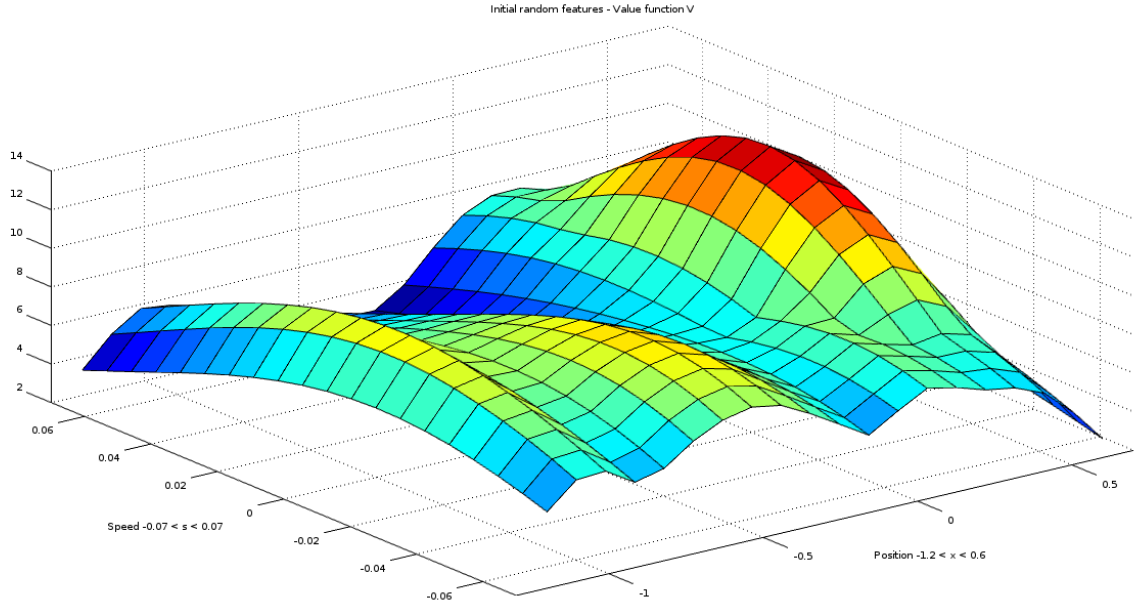

Figure 1: Initial value function V_{init} created with arbitrary (uniform) initial weights, $\alpha_i = 1/d$.

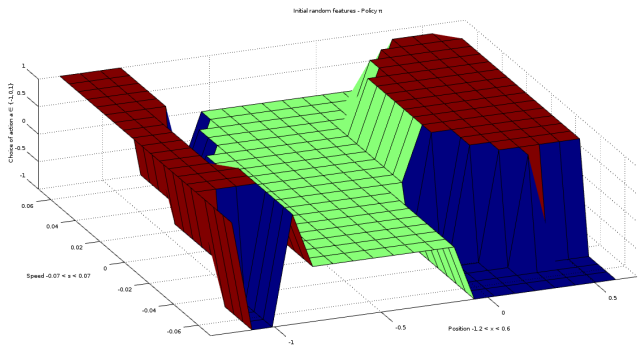

Figure 2: Initial greedy policy $\pi^{V_{\text{init}}}$ created with arbitrary (uniform) initial weights, $\alpha_i = 1/d$.

3 Approximate Value Iteration: *Fitted-Q*

Below is displayed in Figure 3 the final Value function $V^{\star}_{\text{Fitted-Q}}$ obtained by the Fitted-Q algorithm, and then in Figure 4 its greedy policy.

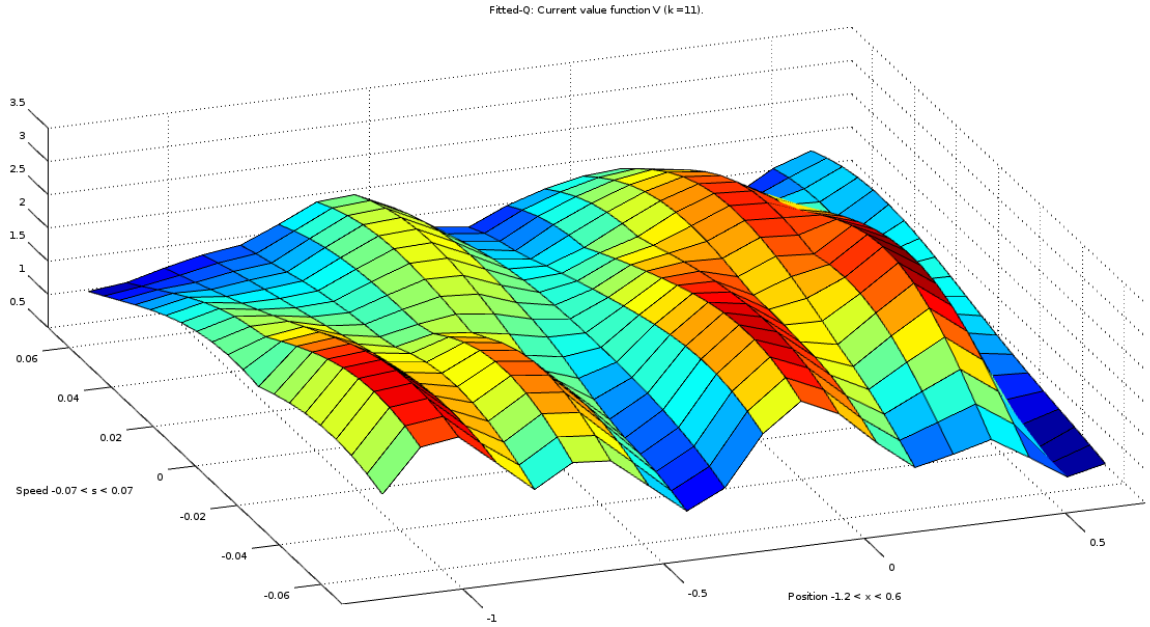

Figure 3: Value function $V^{\star}_{\text{Fitted-Q}}$ returned by the Fitted-Q algorithm.

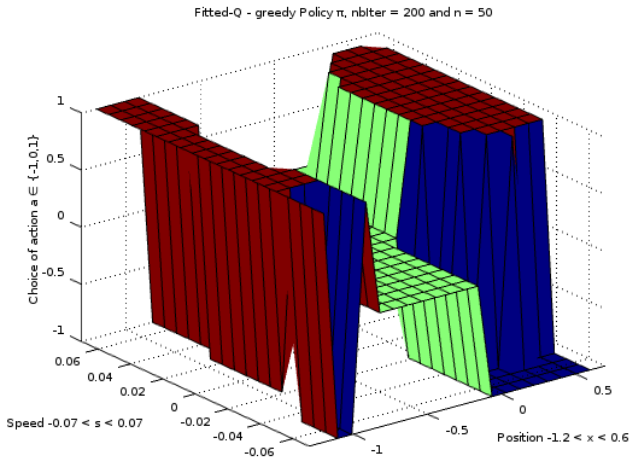

Figure 4: Greedy policy $\pi^{V^{\star}}_{\text{Fitted-Q}}$ function returned by the Fitted-Q algorithm.

We used the following parameters:

- `d = 20`: number of features (ie. dimension of the function space $\mathcal{E}_Q$). It has to be relatively small to have a quick enough program, but big enough to have a set of functions expressive enough (as always, a *trade-off* between complexity and expressiveness).

- `thetasQ = rand_featureQ(d)`: generate random features $\phi_i : (s, a) \mapsto \phi_i(s, a)$ (so we should try several times the same program, in order to be confident in our result).
- `nbIter = 200`: is the time horizon of the simulation (number of iterations in the algorithm).
- `n = 50`: is the number of trajectories[3] to try (in "parallel" – even if our implementation is sequential) for the Fitted-Q algorithm. As usual, we have a *trade-off* between time complexity (a big n implies a longer execution time) and confidence/efficiency (a small n gives less trustworthy/efficient results).
- `gamma = 0.8`: is the **discount parameter**, used as before in the RL course (the future rewards are geometrically decreasing by a γ^t factor). I have **not** tried other values.

For Fitted-Q, the algorithm was not so hard to implement, runs quickly (for small values of `nbIter` and `n`) but does not give extraordinary results.

---

4 Approximate Policy Iteration: *LSTD*

Below is displayed in Figure 5 the final Value function $V^\star_{\text{LSTD}}$ obtained by the LSTD algorithm, and then in Figure 6 its greedy policy.

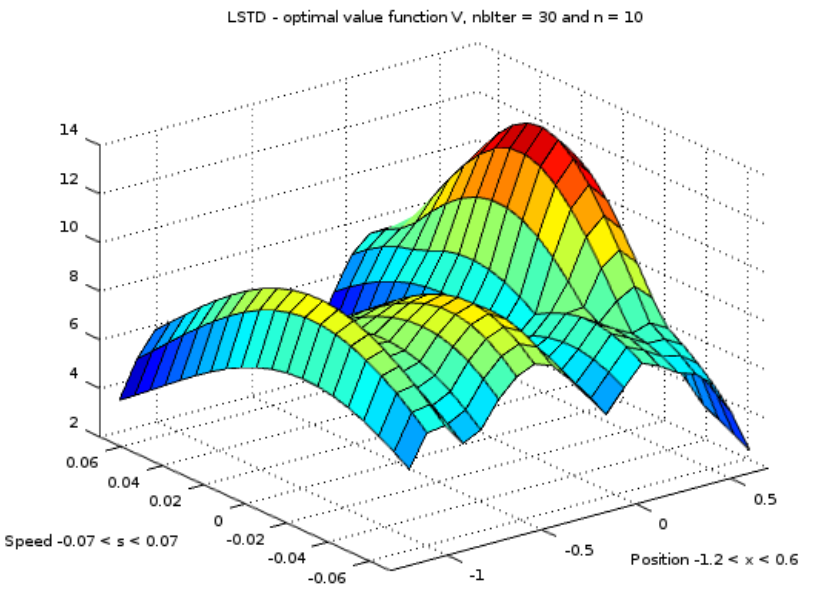

Figure 5: Value function $V^\star_{\text{LSTD}}$ returned by the LSTD algorithm.

[3] See the slides of the course (lecture 05) for the details.

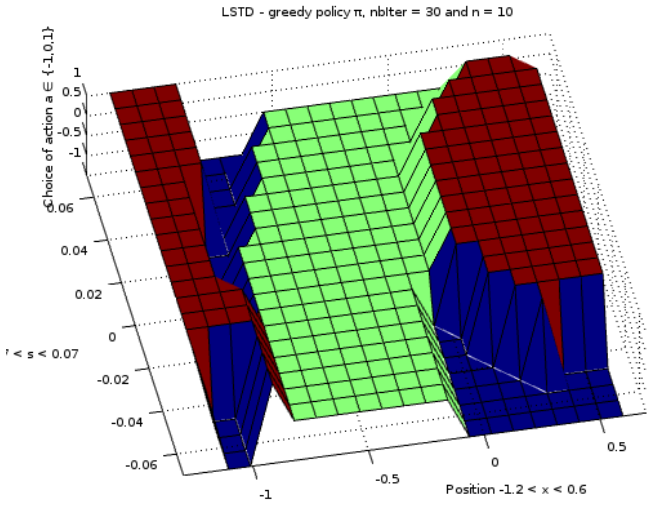

Figure 6: Greedy policy $\pi^{V^\star}_{\text{LSTD}}$ function returned by the LSTD algorithm.

We used the following parameters:

- `nbIter = 30` is the time horizon of the simulation (and it has to be smaller than the one for Fitted-Q, as each time step is computationally heavier).

For LSTD, I felt that it was harder to implement, and it runs way less quickly than Fitted-Q (for small values of `nbIter` and `n`), but apparently it gives better results (but please mind that it is hard to evaluate the quality of a given V or π).

Conclusion

In fact, the final $V^\star$ given by both algorithm (Fitted-Q and LSTD) did not seem really good...

LSTD was running very slowly on my machine (several minutes for 10 steps), and both algorithm are highly dependent on our lucky drawing of the (random) features, and are also influenced by the numerical values chosen for the simulation parameters:

- Number of features: `d = 20`,
- Number of iterations: `nbIter = 200` (or `10` for LSTD),
- Number of simulations: `n = 50`,
- And the discount factor: `gamma = 0.8`.

In order to get better results[4], we should improve the numerical efficiency of LSTD (I checked twice, but maybe one line is wrongly coded and could be improved), or try to use a better machine (or MATLAB, which seemed quicker than Octave on some friends' machine during the TP).

But another importance direction to look at is the simulation parameters, and tweaking them correctly could drastically improve the general quality of the approximated optimal value function $V^\star$.

> This last TP was interesting and useful, as the whole course, so thanks! It was fun to try to solve a problem "less artificial" than last TPs.
> Even if nothing was required, I wanted to finish it quickly and send the two $V^\star$ obtained with *Fitted-Q* and *LSTD*.

[4] At least a policy which is able to start going left, then right, and which is not initially stuck in $x = 0, v = 0$ and decide to chose action $\pi(0, 0) = 0$ ie. not doing anything...