Reinforcement Learning in a Continuous Domain - Mountain Car Problem

Antoine Louis (s152140)

1 Implementation of the domain

The considered domain refers to the well-known "Car on the hill problem". It consists of a state space $\mathbf{X}$ of 2-tuples (p, s), in which a single-agent can take actions from the action space $\mathbf{U}$. The problem is the following : a agent has to climb the hill from any initial position p and initial speed s, with as objective to maximize its cumulative reward through time. The reward that the agent gets in a state is always 0, except in two cases. First, when it achieves its goal of reaching the top of the hill, it then collects a discounted reward of 1. In contrast, when the steps back too much on the left, it then receives a discounted reward of -1. These two extreme cases are terminal states and all the rewards obtained after reaching them are zero.

The dynamics of the domain is continuous, but can be discretized using the Euler integration method, considering an integration time step being equal to $h = 0.001$. As the time between t and $t + 1$ is chosen to be equal to $ts = 0.1$s, the transition from a state at time t to the state at time $t + 1$ can be computed by the Algorithm 1, where the CONTINUOUS_DYNAMICS(p, s, u) function computes the dynamics of the continous problem such that $\dot{p} = s$ and

$$\dot{s} = \frac{u}{m(1 + Hill'(p)^2)} - \frac{gHill'(p)}{1 + Hill'(p)^2} - \frac{s^2 Hill'(p) Hill''(p)}{1 + Hill'(p)^2}$$

with

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

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

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

Algorithm 1 Euler integration method for the discrete-time dynamics

function DISCRETE_DYNAMICS(p, s, u)

  $p_{next} \leftarrow p, \quad s_{next} \leftarrow s$

  for $(i = 0; i < ts/h; i++)$ **do**

    $\dot{p}, \dot{s}$ = CONTINUOUS_DYNAMICS(p_{next}, s_{next}, u)

    $p_{next} \leftarrow p_{next} + h * \dot{p}$

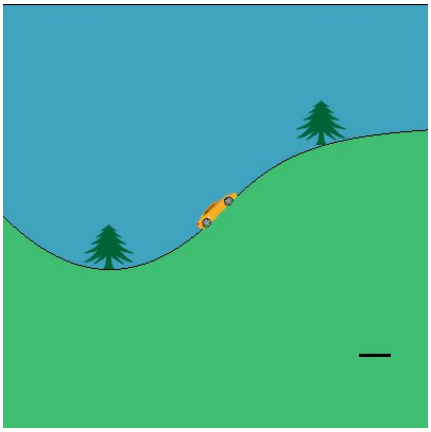
    $s_{next} \leftarrow s_{next} + h * \dot{s}$

  end for

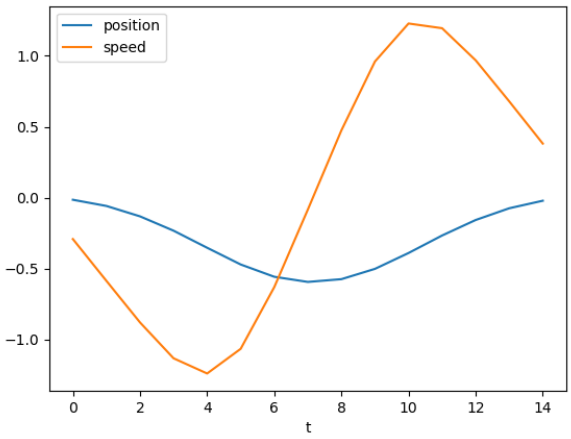
  return p_{next}, s_{next}

end function

To test our implementation, we ran an agent for 15 time steps, following the simple policy that always takes the action $u = 4$ and beginning at an initial state $(p, s) = (0, 0)$. Figure 1 shows the evolution of its position and speed through time. The results seem consistent : starting at position $p = 0$ with a zero speed $s = 0$, the car will first run down the hill until position $p = -0.6$ with its speed being negative, then its speed will become positive and it will go forwards trying to climb the hill and reaching a position near $p = 0$ after the 15 time steps.



(a) Display of the state (0,0) of the car



(b) Evolution of the position and the speed

Figure 1: Simple agent starting at (0,0) and following the policy that always takes action $u = 4$ for 15 time steps

2 Expected return of a policy

If we consider that the agent always start at state $(p, s) = (0, 0)$, the expected return of a given policy μ can be computed using the functions $J_N^\mu : X \to \mathbb{R}$ seen for discrete deterministic domain. As a reminder, they are defined by the recurrence equation :

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

with $J_N^\mu(0) \equiv 0$. We then know that the error between J_N^μ and J^μ is bounded by :

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

where B_r is here equal to 1 and γ to 0.95. By considering that the agent must manage to climb the hill in a reasonable amount of time steps (let's say maximum 200 time steps), the cumulative reward that it will get after this maximum number of time will be equal to $0.95^{200} * 1 = 3.5\text{e}{-5}$. The value of N for computing the J_N^μ must be chosen in order for the bound to be negligible in comparison with this latter order of magnitude. Choosing $N = 400$ leads to a bound equals to 2.45e−8, which can be considered as negligible.

3 Fitted-Q-Iteration

In this section, the Fitted-Q-Iteration algorithm (FQI) will be tested and compared using three supervised learning techniques : Linear Regression, Extremely Randomized Trees and Neural Networks. The majority of the figures are shown in Appendix to improve the readability.

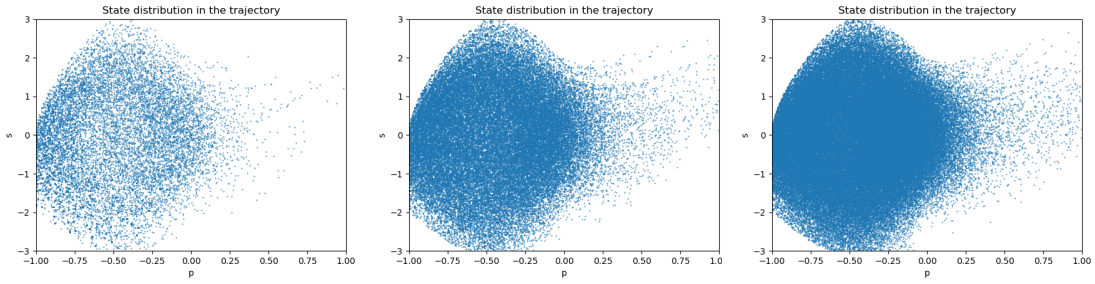
3.1 Four-tuples generation

As a reminder, FQI is an iterative algorithm that uses a set of four-tuples F together with a batch-mode supervised learning algorithm to compute a sequence of $\hat{Q}_N$-functions, approximations of the Q_N-functions. The set F results from the observation of a certain number of one-step system transitions (from t to $t+1$) where each transition provides the knowledge of a new four-tuple (x_t, u_t, r_t, x_{t+1}), such that :

$$F = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l)\}_{l=1}^{\#F}$$

This set can be generated in multiple ways. One way to generate it would be to gather the four-tuples corresponding to one single trajectory. Another way would be to consider several independently multi-steps trajectories. In this project, the latter will be used. As the goal is to find the optimal policy from a given initial state (p_{init}, s_{init}), one can choose to consider a set of trajectories that all start in that initial state. In each trajectory, the action u_t at each time step is chosen with equal probability among its two possible values $u = -4$ and $u = 4$, until a terminal state is reached (or a given maximal bound is achieved, preventing a potential infinite trajectory in the case of a car stuck in the dip of the hill).

To sum up, the set of four-tuples F is defined by four parameters : the number of trajectories considered n_{traj}, the maximal size of a trajectory $size_{max}$, the initial position p_{init} and the initial speed s_{init}. For the rest of the project, we will consider that this initial state is $(p_{init}, s_{init}) = (-0.5, 0)$, corresponding to the car stopped at the bottom of the hill. That means that the four-tuples will be generated considering trajectories starting all from that initial state. Figure 2 shows the distribution of the state space X in F generated from multiple trajectories when the car starts in $(-0.5, 0)$. In the following parts, 1000 trajectories will be considered, resulting in a set F of 57515 four-tuples.



(a) $n_{traj} = 1000$ (b) $n_{traj} = 5000$ (c) $n_{traj} = 10000$

Figure 2: State space distribution in F. All trajectories start in $(-0.5, 0)$.

3.2 Stopping conditions

The stopping conditions are required to decide at which iteration (i.e., for which value of N) the iterative process can be stopped. One simple way of doing this is to define a priori a maximum number of iterations by noting that an error bound on the sub-optimality in terms of number of iterations is given by the following equation :

$$||J^{\mu_N^*} - J^{\mu^*}||_\infty \leq \frac{2\ \gamma^N}{(1-\gamma)^2} B_r$$

where $B_r = 1$ and $\gamma = 0.95$. By making the hypothesis that the optimal trajectory has to be smaller than 50 time steps, the smallest cumulative reward that the car can get if it manages to climb the hill after these 50 time steps is equal to $0.95^{50} * 1 = 7.69\text{e}{-2}$. By fixing $N = 300$, the right hand side of the previous equation is equal to $1.66\text{e}{-4}$, which can be considered as a negligible bound.

Another possibility would be to stop the iterative process when the distance between $\hat{Q}_N$ and $\hat{Q}_{N-1}$ becomes negligible. However, this kind of convergence criterion doesn't make sense in practice because some supervised learning algorithms do not guarantee that the sequence of $\hat{Q}_N$-functions actually convergences, as will be seen later.

In practice, computing the sequence of $\hat{Q}_N$-functions with 300 iterations would take too much time for some of the supervised learning algorithms tested here, so the number of iterations was fixed to 50.

3.3 Metrics to assess performances of the algorithms

To rank performances of the various algorithms, some metrics need to be defined. In our analysis, four different metrics are considered, inspired from Paper [1].

Score of a policy First, to measure the quality of a solution given by an algorithm, the resulting stationary policy can be assessed. By computing the expected return of this stationary policy, we can say that the higher the expected return, the better the algorithm. Instead of computing the expected return for one single initial state, we define the *score* of a policy as the average expected return of that policy over a set of initial states X^i. If u is the policy, its score would be defined by :

$$score = \frac{\sum_{x \in X^i} J^{\mu}_{\infty}(x)}{\#X^i}$$

To compute these scores, the following set is considered :

$$X^i = \{(p, s) \in X | \exists\ i, j \in Z : (p, s) = (0.125 * i, 0.375 * j)\}$$

covering a total of 225 states (15 x 15) evenly distributed across the state space.

Bellman residual Second, to estimate the quality of a function $\hat{Q}$, we define the *Bellman residual of* $\hat{Q}$ as being the average over all samples contained in the four-tuples set of the squared distance between two consecutive approximates of Q :

$$d(\hat{Q}_N, \hat{Q}_{N-1}) = \frac{\sum_{l=1}^{\#F} (\hat{Q}_N(x_t^l, u_t^l) - \hat{Q}_{N-1}(x_t^l, u_t^l))^2}{\#F}$$

This distance will also help us study the speed of convergence of our algorithms.

Learning speed Then, a metrics could concern the learning speed of an algorithm, which will depend on the number of iterations N required to reach a policy close to the optimal one. The less iterations the algorithm needs to achieve that goal, the faster it learns.

Optimal trajectory Finally, one could study the optimal trajectory given by the algorithm after computing $\hat{Q}_N$.

3.4 Supervised learning techniques

Each supervised learning algorithm that is studied in the following parts will take as inputs a set of state-action pairs (p, s, u), sampled from the four-tuple set F, and will then output an approximation of Q. The problem resides in the parametrization of these algorithms, making them the best as possible in order to later compare them.

3.4.1 Linear Regression

Linear regression uses a linear approach to model the relationship between the scalar output ($\hat{Q}_N$) and the explanatory variables (p, s, u). The approximated value $\hat{Q}_N$ will be a linear combination of its features space, namely the position, speed and action.

The evolution of $\hat{\mu}_N^*$ is shown in Figure 3 in Appendix A.1. It can be seen that the algorithm is not able to reach a better policy than "always accelerate", even after 50 iterations. The consequence, as shown in Figure 5c, is that the car get stuck in the dip of the hill and fails to reach the top in a reasonable amount of time steps. This is probably due to the fact that the task is to complex to simply use the 3-tuple (p, s, u) to approximate the Q-value. A possible solution might be to increase the number of explanatory variables by considering, in addition to the previous ones, all their possible products. For example, if one decides to consider all the products of length 2 of the variables p, s and u, the final resulting explanatory variables would be : p, s, u, $p*p$, $s*s$, $u*u$, $p*u$, $s*u$ and $p*s$. This process might allow to better catch the complexity of the model and could be an option for further tests.

Notice, in Figure 5a, that the distances between $\hat{Q}_N$ and $\hat{Q}_{N-1}$ becomes almost negligible after 50 iterations. However, we have seen that these $\hat{Q}_N$ are far from the optimal ones. This is why one must be careful when considering this distance as a stopping criterion because some supervised learning algorithms do not guarantee that the sequence of $\hat{Q}_N$-functions actually converges.

Finally, with this policy of "always accelerating", Figure 5b shows that the resulting score is close to 0.182, which is the lowest score of our three algorithms.

3.4.2 Extremely Randomized Trees

Extremely Randomized Trees, often called "Extra-Trees", is a trees ensembles method where each tree is built from the complete original training set, contrary to Tree Bagging which uses the standard CART algorithm to derive the trees from a bootstrap sample. Three paramaters are associated to this algorithm : the number M of trees to build, the number K of candidate tests at each node and the minimal leaf size n_{min}. In our experiments, the Extra-Trees algorithm is used with fully developed trees (i.e., $n_{min} = 2$), considering a total of 50 trees.

Here, the estimated Q values, shown in Figure 7 in Appendix A.2, are far from reality. This is indeed a drawback of non kernel-based methods, they do not guarantee convergence. However, with the Extra-Trees algorithm, even if the sequence is not converging, it is still able to find an outstanding policy that allows the car to reach the top of the hill in only 18 time steps starting from the initial position $(p_{init}, s_{init}) = (-0.5, 0)$ and getting a final reward of 0.397. For comparison, starting from the same initial position, FQI with Linear Regression took 1247 time steps to reach the top with its simple policy "always accelerate". Furthermore, the score of that optimal policy stabilises quite quickly (approximately from the 15^{th} iteration) around 0.33, as shown in Figure 8b, which is a pretty good score and the best one among the three algorithms.

Notice that the sudden increase of the distance in Figure 8a is explained in Paper [1] by the fact that the distance is mostly determined by variations between $\hat{Q}_N$ and $\hat{Q}_{N-1}$ around the initial state $(p_{init}, s_{init}) = (-0.5, 0)$ (as can be seen in Figure 2) since all trajectories start from this state. Adding to that the fact that a certain number of iterations are needed for the algorithm to obtain nonzero values of $\hat{Q}_N$ around $(p_{init}, s_{init}) = (-0.5, 0)$ is what causes that sudden increase.

3.4.3 Neural networks

The main difficulty when building a neural network is to wisely choose the parameters. To choose the best neural net, we mainly played with the structure of the neural net, that is the number of hidden layers and the number of neurons in each layer, as well as with the activation function.

The starting point of our hyper-parameters tuning was the architecture used in Paper [2], where the neural network was composed of two hidden layers of 5 neurons each. From that topology, the different activation functions were tested, namely the logistic sigmoid (*sigmoid*), the hyperbolic tangent (*tanh*) and the rectified linear unit (*relu*). From these three, it turns out, after some tests, that the hyperbolic tangent was the most convenient for this problem. Unfortunately, the given results weren't good enough to keep the initial structure proposed in Paper [2].

Therefore, considering again two hidden layers, the number of neurons in each layer was increased and the neural net was tested for layers with 5, 10, 15 and 20 neurons. After testing all the possible combinations, it turns out that the best structure for our neural network is 15 neurons in the first hidden layer and 20 neurons in the second one, all neurons being activated with the hyperbolic tangent. This architecture gives pretty good results in general. From the initial state $(p_{init}, s_{init}) = (-0.5, 0)$, the car is able to climb the hill in 18 time steps, exactly as Extra-Trees. However, the resulting policy computed after 50 iterations is not the optimal one, as can be seen in Figure 9. Moreover, the scores computed at each iteration vary a lot. We can for example observe in Figure 11b that the score computed at iteration 20 is much better than the one from iteration 50. This is due to the fact that the sequence of $\hat{Q}_N$-functions is not converging at all, as can be seen in Figure 11a. Therefore, it is completely possible that some lower iteration gives a better policy than the one of iteration 50, as it is the case for iteration 20 (see Figure 9c).

3.4.4 Comparison of the algorithms

Some comparisons between the different supervised learning algorithms can be done by analysing the Figure 12 in Appendix A.4.

First, we can clearly see that Linear Regression gives the worst results in terms of scores of the policy, optimal trajectory and number of time steps needed to climb the hill. As previously explained, this model taking as inputs only the three-tuple (p, s, u) is too simple to estimate the corresponding $\hat{Q}$-values.

Then, it turns out that both Extra-Trees and Neural network manage to reach the top in only 20 time steps taking almost the same trajectory. However, the $\hat{Q}_N$-functions computed with Neural network vary a lot from one iteration to the other, leading therefore to very fluctuating scores and to policies that can get worse for bigger N.

A last point could concern the time needed for each algorithm to compute the sequence of $\hat{Q}_N$-functions. The quickest algorithm was without hesitation Linear Regression, taking less than a second to compute the all sequence. Then comes Extra-Trees that took a little more than 2 minutes and finally the Neural network that took more than half an hour to train. The exact times are given in Table 1. Note that the Neural net is trained using CPU.

To conclude, it clearly seems that Extra-trees is the best algorithm to use for Fitted Q-Iteration among the three.

Model	Linear Regression	Extra-Trees	Neural Network
Time	0.32s	122.68s	2174.42s

Table 1: Computational time of the different algorithms to compute the $\hat{Q}_N$-functions.

4 Parametric Q-Learning

This section describes the implementation of a routine which estimates the Q-function with Q-learning when a parametric approximation architecture of $Q(x, u, \theta^*)$ is used.

4.1 Implementation

Here, the algorithm performs online, meaning that rather than using a set of transitions together to deduce a policy, it observes one transition at a time and updates progressively its belief about the optimal policy. Note that these step-by-step transitions are observed using a personal policy (often ϵ-greedy).

Basically, the algorithm is the following : after observing a transition (x_t, u_t, r_t, x_{t+1}), it updates the parameter vector θ of the function approximator as follows :

$$\theta_{k+1} = \theta_k + \alpha_k \left[r_{k+1} + \gamma \max_{u' \in U} \hat{Q}_k(x_{k+1}, u') - \hat{Q}_k(x_k, u_k) \right] \frac{\partial}{\partial \theta_k} \hat{Q}_k(x_k, u_k) \tag{1}$$

With a linearly parameterized approximator, the update in Equation (1) becomes :

$$\theta_{k+1} = \theta_k + \alpha_k \left[r_{k+1} + \gamma \max_{u' \in U} \phi^T(x_{k+1}, u')\theta_k - \phi^T(x_k, u_k)\theta_k \right] \phi(x_k, u_k) \tag{2}$$

The full algorithm presenting gradient-based Q-learning with a linear parametrization and ϵ-greedy exploration, described in [3], can be consulted in Appendix B.

Here, it is asked to use a Radial Basis Function (RBF) as approximation architecture. A number of functions can be used as the RBF, namely the Gaussian, the multiquadratic, the inverse quadratic, the polyharmonic spline, the thin-plate spline, etc. Among these RBFs, the Gaussian is typically selected because it is compact, positive and is the only factorizable RBF (this property is desirable for hardware implementation of the RBF network) [5]. An RBF feature i can be defined as a Gauss error distribution curve that has a given standard deviation σ_i and a center c_i:

$$\phi_s(i) = exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2} \right)$$

To find the centers c_i, we can use K-means clustering on our input data. The only parameter to tune here is the number of clusters to consider. Some tests are discussed in the next section. Concerning σ_i, a single standard deviation for all clusters can be used where $\sigma = \frac{d_{\max}}{\sqrt{2n}}$, $d_{\max}$ being the maximum distance between any two cluster centers and n the number of cluster centers.

Once these parameters are known and after having initialised the parameter vector θ to 0, it can be updated during N iterations by the Equation (2).

4.2 Results

Four parameters need to be tuned in order to achieve the best results : the number n of clusters to consider when performing the K-means, the learning rate α, the exploration schedule ϵ and the number of episodes considered. After some tests, it turns out that the best results were given with $\alpha = 0.01$ and $\epsilon = 0.9$. Then, we vary the number of clusters n from 10 to 210, and compute a maximum of 2000 episodes for each test.

The results were pretty bad, none of them achieved to reach the top of the hill and neither outputs a good policy. It seems that the "best" results were achieved each time after 1000 episodes, and by considering a minimum number of clusters of 100. Some of the resulting policies for different values of n are shown in Figure 16 in Appendix B. As a consequence, the score of that algorithm was negative, oscillating around -0.3 as can be seen in Figure 16a. These results can probably be explained by the fact that the centers of the Gaussian function are defined with an unsupervised learning technique which might not be very precise in a sense that it is hard to tell which optimal number of clusters n must be considered to compute them.

4.3 Comparison with FQI

Here, it makes no doubt that the FQI algorithm performs much better than Parametric Q-learning on every front, even when FQI uses Linear Regression as supervised learning technique. It seems that FQI might be better suited for this problem than Parametric Q-learning.

In terms of computational time, while FQI with Extra-Trees took 122.68s to compute the $\hat{Q}_5$0-functions, Parametric Q-learning with 210 clusters took 377.12s considering 1000 episodes.

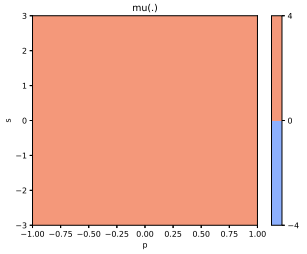
5 Conclusion

In order to solve the "Car on the Hill" problem, consisting in finding an optimal policy for the car to climb the hill, two RL algorithms were tested. The first one was the Fitted Q-iteration algorithm which basically approximate the Q-functions by using a supervised learning algorithm. Among the three supervised learning techniques that were tested here - Linear Regression, Extra-Trees and Neural Networks - Extra-Trees clearly outperforms the two others in terms of resulting optimal policy and expected returns. Indeed, FQI with Linear Regression outputs a very poor policy equivalent to "always accelerate" wherever you are, and Neural Network performed some pretty good policy and scores too but was much less stable than Extra-Trees in a sense that the approximated Q-functions are not converging towards a fixed value and so stopping the algorithm at a given iteration requires some knowledge on the problem.

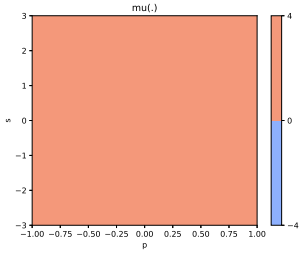
Then, the Parametric Q-learning algorithm was tested using clustering and Gaussian function as approximation architectures. This algorithm gives us the worst results among the others. Even by considering different number of clusters, it was not able to find a policy that led to the top of the hill. As a result, all the computed scores through the episodes were negative.

Appendix A Fitted Q-Iteration

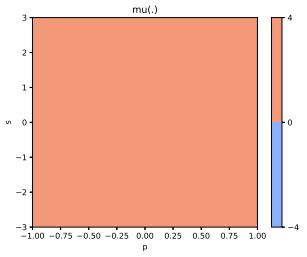
A.1 Linear Regression



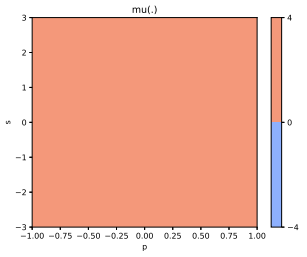
(a) $\arg\max_{u \in U} \hat{Q}_5(x, u)$



(b) $\arg\max_{u \in U} \hat{Q}_{10}(x, u)$

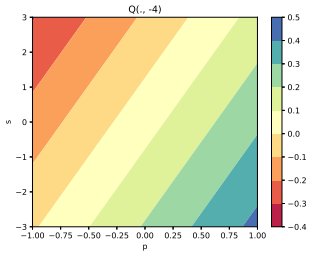


(c) $\arg\max_{u \in U} \hat{Q}_{20}(x, u)$

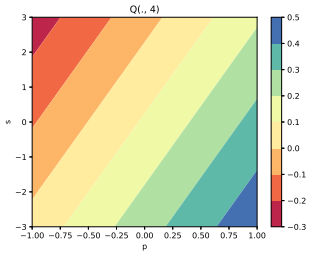


(d) $\arg\max_{u \in U} \hat{Q}_{50}(x, u)$

Figure 3: Representation of $\hat{\mu}_N^*$ for different values of N.

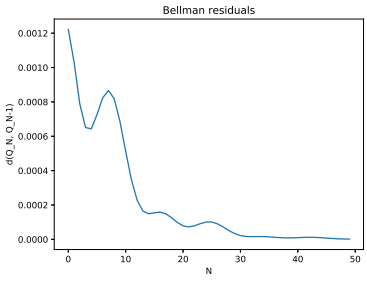


(a) $\hat{Q}_{50}(., -4)$

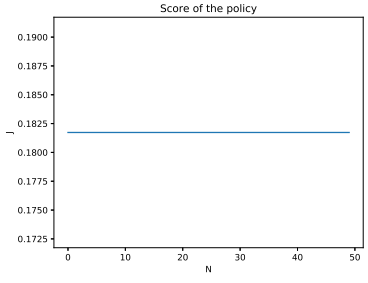


(b) $\hat{Q}_{50}(., 4)$

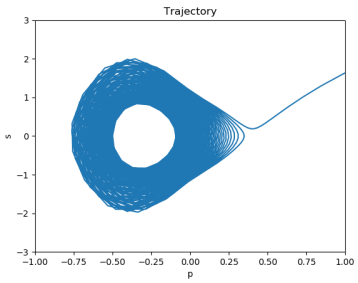
Figure 4: Representation of $\hat{Q}_{50}$.



(a) $d(\hat{Q}_N, \hat{Q}_{N-1})$



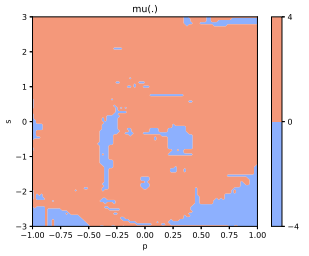
(b) $J_\infty^{\hat{\mu}_N^*}$



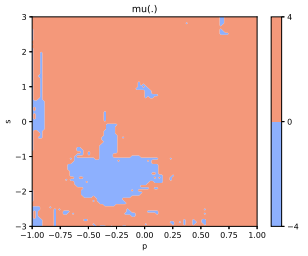
(c) Trajectory

Figure 5: Figure 5a represents the distance between $\hat{Q}_N$ and $\hat{Q}_{N-1}$. Figure 5b provides the average return obtained by the policy $\hat{\mu}_N^*$. Figure 5c represents the trajectory when $x_0 = (-0.5, 0)$ and when the policy $\hat{\mu}_{50}^*$ is used to control the system.

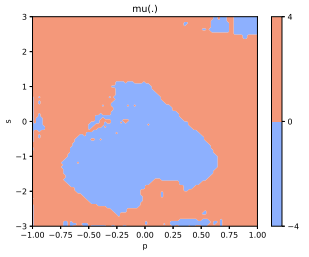
A.2 Extra-Trees



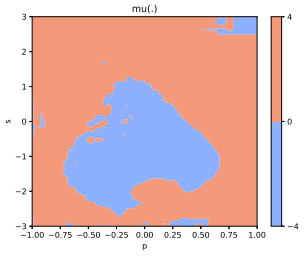
(a) $\arg\max_{u \in U} \hat{Q}_5(x,u)$



(b) $\arg\max_{u \in U} \hat{Q}_{10}(x,u)$

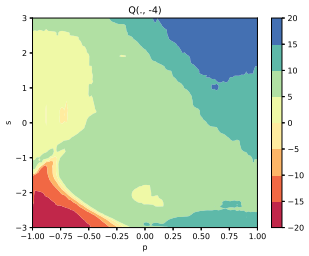


(c) $\arg\max_{u \in U} \hat{Q}_{20}(x,u)$

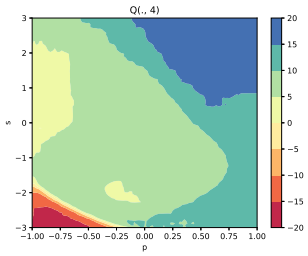


(d) $\arg\max_{u \in U} \hat{Q}_{50}(x,u)$

Figure 6: Representation of $\hat{\mu}_N^*$ for different values of N.

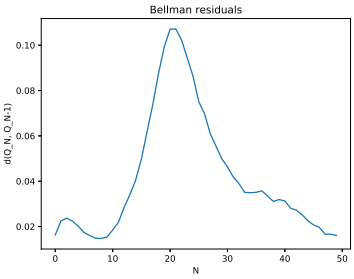


(a) $\hat{Q}_{50}(.,-4)$

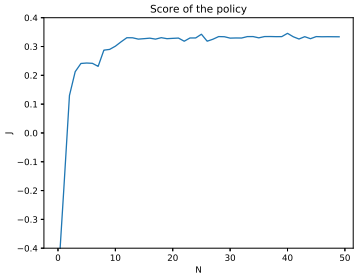


(b) $\hat{Q}_{50}(.,4)$

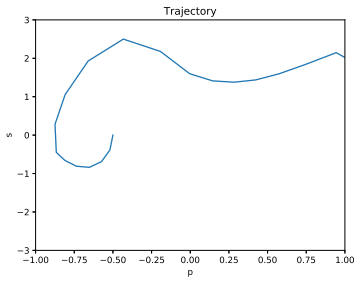
Figure 7: Representation of $\hat{Q}_{50}$.



(a) $d(\hat{Q}_N, \hat{Q}_{N-1})$



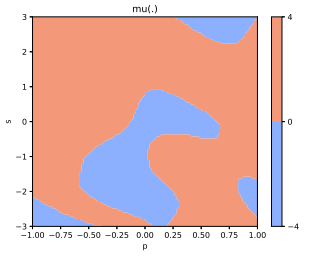
(b) $J_\infty^{\hat{\mu}_N^*}$



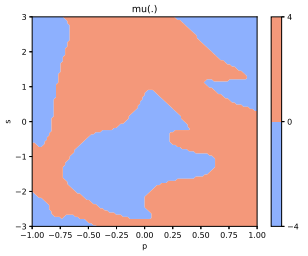
(c) Trajectory

Figure 8: Figure 8a represents the distance between $\hat{Q}_N$ and $\hat{Q}_{N-1}$. Figure 8b provides the average return obtained by the policy $\hat{\mu}_N^*$. Figure 8c represents the trajectory when $x_0 = (-0.5, 0)$ and when the policy $\hat{\mu}_{50}^*$ is used to control the system.

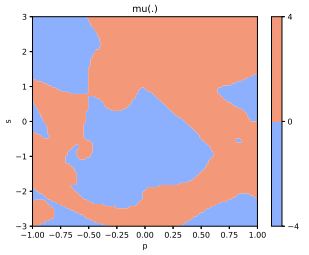
A.3 Neural Networks



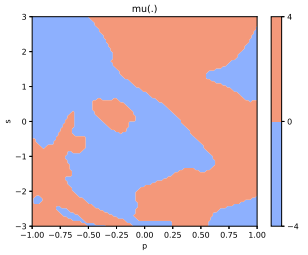
(a) $\arg\max_{u \in U} \hat{Q}_5(x,u)$



(b) $\arg\max_{u \in U} \hat{Q}_{10}(x,u)$

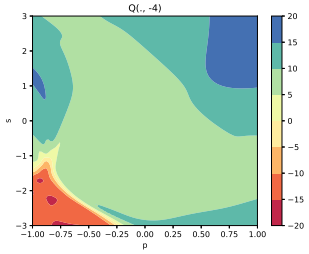


(c) $\arg\max_{u \in U} \hat{Q}_{20}(x,u)$

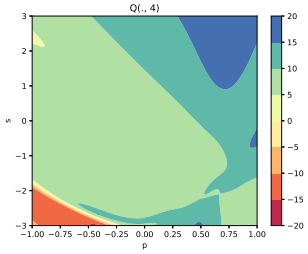


(d) $\arg\max_{u \in U} \hat{Q}_{50}(x,u)$

Figure 9: Representation of $\hat{\mu}_N^*$ for different values of N.

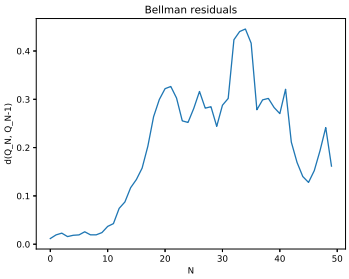


(a) $\hat{Q}_{50}(.,-4)$

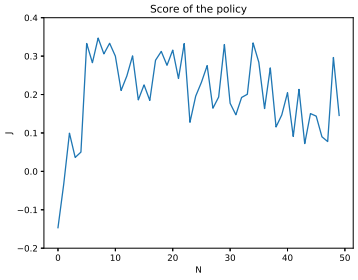


(b) $\hat{Q}_{50}(.,4)$

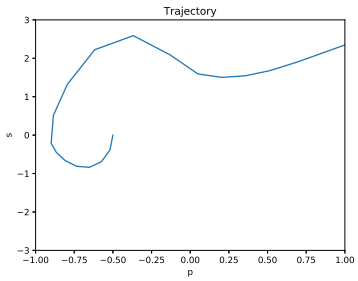
Figure 10: Representation of $\hat{Q}_{50}$.



(a) $d(\hat{Q}_N, \hat{Q}_{N-1})$



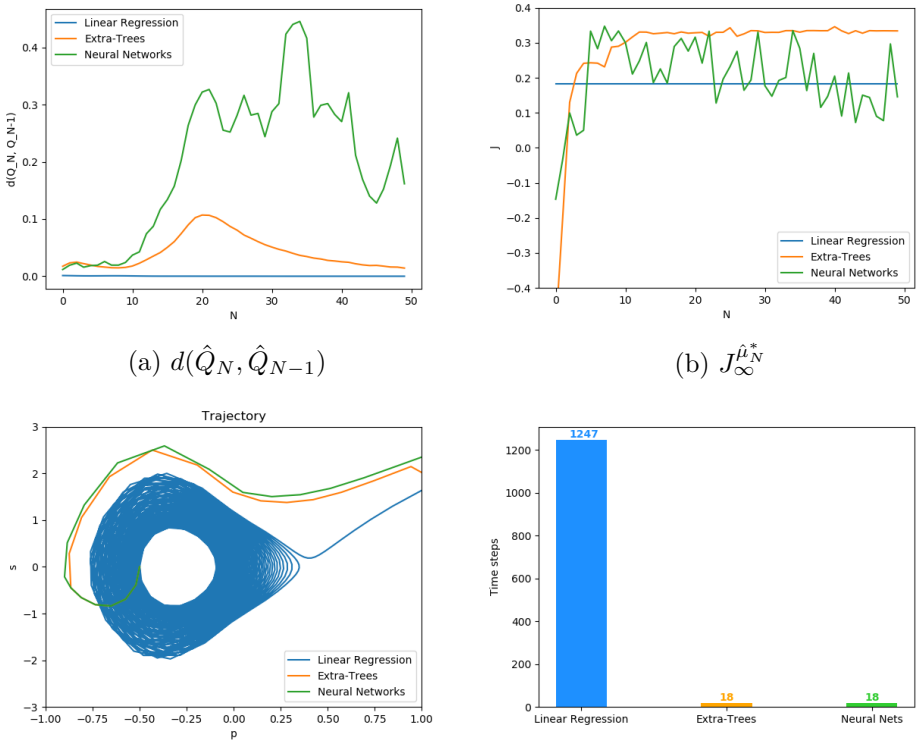
(b) $J_\infty^{\hat{\mu}_N^*}$



(c) Trajectory

Figure 11: Figure 11a represents the distance between $\hat{Q}_N$ and $\hat{Q}_{N-1}$. Figure 11b provides the average return obtained by the policy $\hat{\mu}_N^*$. Figure 11c represents the trajectory when $x_0 = (-0.5, 0)$ and when the policy $\hat{\mu}_{50}^*$ is used to control the system.

A.4 Comparison of the algorithms



(a) $d(\hat{Q}_N, \hat{Q}_{N-1})$ (b) $J_\infty^{\hat{\mu}_N^*}$

(c) Trajectories (d) Time steps

Figure 12: Figure 12a represents the distance between $\hat{Q}_N$ and $\hat{Q}_{N-1}$ for each algorithm. Figure 12b provides the average return obtained by the policy $\hat{\mu}_N^*$ for the different algorithms. Figure 12c represents the trajectories of the different algorithms when $x_0 = (-0.5, 0)$ and when the policy $\hat{\mu}_{50}^*$ is used to control the system. Figure 12d gives the number of time steps required for each algorithm to climb the hill.

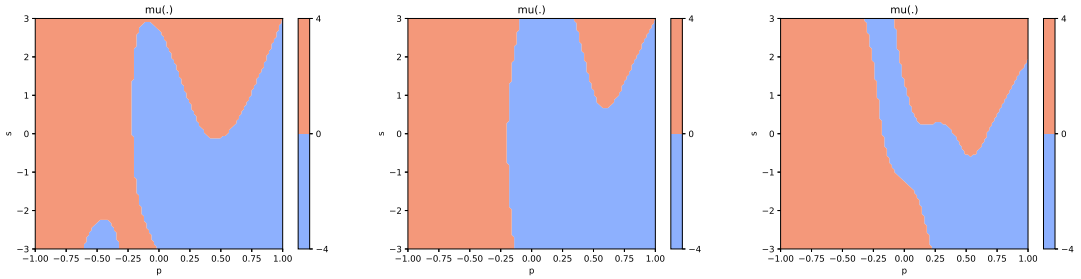
Appendix B Parametric Q-learning

ALGORITHM 3.3 Q-learning with a linear parametrization and ε-greedy exploration.

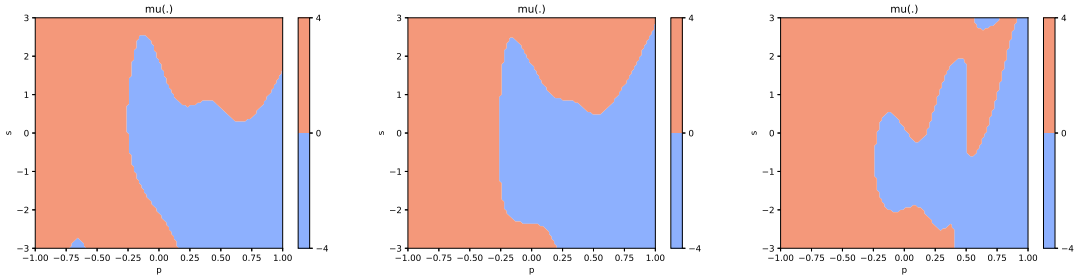
Input: discount factor γ,
BFs $\phi_1, \ldots, \phi_n : X \times U \to \mathbb{R}$,
exploration schedule $\{\varepsilon_k\}_{k=0}^{\infty}$, learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$

1: initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
2: measure initial state x_0
3: **for** every time step $k = 0, 1, 2, \ldots$ **do**
4: $u_k \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} \left(\phi^{\mathrm{T}}(x_k, \bar{u})\theta_k\right) & \text{with probability } 1 - \varepsilon_k \text{ (exploit)} \\ \text{a uniform random action in } U & \text{with probability } \varepsilon_k \text{ (explore)} \end{cases}$
5: apply u_k, measure next state x_{k+1} and reward r_{k+1}
6: $\theta_{k+1} \leftarrow \theta_k + \alpha_k \left[r_{k+1} + \gamma \max_{u'} \left(\phi^{\mathrm{T}}(x_{k+1}, u')\theta_k\right) - \phi^{\mathrm{T}}(x_k, u_k)\theta_k\right] \phi(x_k, u_k)$
7: **end for**

Figure 13: Q-learning with a linear parametrization and e -greedy exploration ([3]).

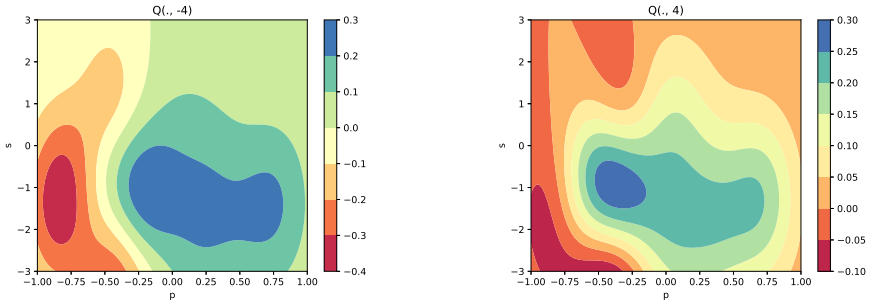


(a) $n = 110$ (b) $n = 130$ (c) $n = 150$



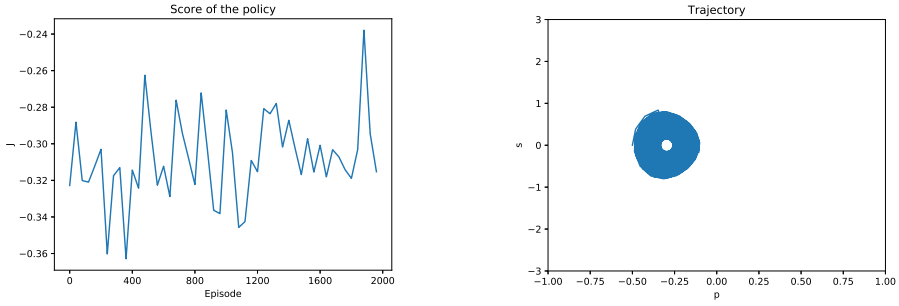
(d) $n = 170$ (e) $n = 190$ (f) $n = 210$

Figure 14: Representation of $\hat{\mu}^*$ computed over 1000 episodes when considering n clusters.



(a) $\hat{Q}(., -4)$ (b) $\hat{Q}(., 4)$

Figure 15: Representation of $\hat{Q}$ for $n = 210$ when considering 1000 episodes.



(a) $J_{\infty}^{\hat{\mu}^*}$ (b) Trajectory

Figure 16: Figure 16a provides the average return obtained by the policy $\hat{\mu}^*$ with $n = 210$. Figure 16b represents the trajectory when $x_0 = (-0.5, 0)$ and when the policy $\hat{\mu}^*$ is used to control the system when $n = 210$ and after 1000 episodes.

References

[1] Damien Ernst, Pierre Geurts and Louis Wehenkel, *Tree-based batch mode reinforcement learning*, Journal of Machine Learning Research, 503-506, 2005.

[2] Martin Riedmiller. *Neural Fitted Q iteration - First experiences with a data efficient neural reinforcement learning method*, Machine Learning: ECML 2005, 317-328, 2005.

[3] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*, CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[4] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*, ISBN 1-886529-10-8, 512 pages, 1996

[5] Wu, Yue Wang, Hui Zhang, Biaobiao Du, K.-L. (2012). *Using Radial Basis Function Networks for Function Approximation and Classification*, ISRN Applied Mathematics. 2012. 10.5402/2012/324194.