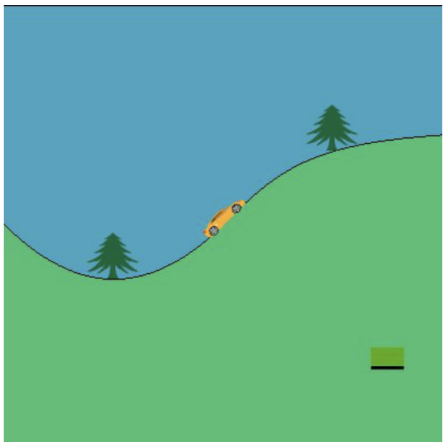
LIÈGE université
School of Engineering

INFO8003-1: Optimal Decision Making for Complex Problems



Assignment 1: Reinforcement Learning in a Continuous Domain

Staff :
ERNST Damien, *Teacher*
LOUETTE Arthur, *Teaching Assistant*
MIFTARI Bardhyl, *Teaching Assistant*

Group :
LAMBERMONT Romain
LOUIS Arthur

April 1, 2024

Contents

List of Figures

List of Tables

1 Implementation of the Domain

For this project, we decided to make two classes that we would use throughout the project. Those two classes are `Domain` and `Agent`.

1.1 Domain

In this first class we implemented the domain with the functions representing the hill, its derivatives, the transition dynamics and reward signal. The different functions implemented to represent the hill (height, first and second derivatives) with respect to `p` are as follows :

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

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

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

We also need to compute a reward signal associated with each action taken in every possible state (joined position and speed tuple). The formulation of the car on the hill tells us that the rewards returned depending on the action should be of -1 when reaching the left of the hill and $+1$ when reaching the right (top) of the hill:

```
if self.terminate:
    return 0

p_next, s_next = self.dynamics(p, s, u)

if p_next < -1 or np.abs(s_next) > 3:
    self.terminate = True
    return -1
elif p_next > 1 and np.abs(s_next) <= 3:
    self.terminate = True
    return 1
else:
    return 0
```

At each action taken, we thus need to check if a terminal state as been reached (either $|\texttt{p}| > 1$ or $|\texttt{s}| > 3$) so that we can set an attribute of the class `Domain` (`self.terminate`) to `True` so that the dynamics always return the same `p` and `s` to ensure nothing evolves in the simulation and the reward signal keeps returning 0.

As for the dynamics of the domain we formulated:

$\texttt{u} = \{-4, 4\}$ the action taken

$\texttt{m} = 1$ the mass of the car

$\texttt{g} = 9.81$ the gravitational force

$$\texttt{p\_prime} = s$$

$$\texttt{s\_prime} = \frac{u}{m \times \left(1 + \texttt{hill\_prime(p)}^2\right)} - \frac{g \times \texttt{hill\_prime(p)}}{1 + \texttt{hill\_prime(p)}^2} - \frac{s^2 \times \texttt{hill\_prime(p)} \times \texttt{hill\_prime\_prime(p)}}{1 + \texttt{hill\_prime(p)}^2}$$

These domain dynamics are computed using an Euler method integration using an integration time step of 0.001 second and a discretization time step of 0.1 second. The derivatives of the position and the speed are thus computed using a loop of $\frac{0.1}{0.001} = 100$ steps that looks like this:

```
if np.abs(p) > 1 or np.abs(s) > 3:
    return p, s

p_next = p
s_next = s

for _ in range(100):
    p_prime = s_next
    s_prime = ((u) / (M * (1 + hill_prime(p_next)**2))) \
    - ((G * hill_prime(p_next)) / (1 + hill_prime(p_next)**2)) \
    - ((s_next**2 * hill_prime(p_next) * hill_prime_prime(p_next)) / (1 + hill_prime(p_next)**2))

    p_next += 0.001 * p_prime
    s_next += 0.001 * s_prime
```

1.2 Agent

In this second class we implemented the agent that can take actions in the domain for the first three sections. The agent takes an argument, `randomized`, that sets the behavior of the agent between following a policy of choosing random actions or of always accelerating, when it is initialized. The only attributes of the agent or thus `init_p` that is drawn from a random uniform distribution, `init_s` that is set to 0 and `randomized` that sets the behavior of the agent as mentioned before. The initialization of the agent thus looks like this:

```
def __init__(self, randomized=False):
    self.init_p = np.random.uniform(-0.1, 0.1)
    self.init_s = 0
    self.randomized = randomized
```

As for the policy of the agent, the implementation is straightforward as the choice between -4 and $+4$ acceleration stored in an array `U` only depends on the attribute `randomized`:

```
def policy(self):
    if self.randomized:
        return np.random.choice(U)
    else:
        return U[1]
```

1.3 Trajectories

Once the domain and the agent were correctly implemented, we tried to run multiple tests of the generation of trajectories using our function `generate_trajectory` that takes as argument an agent, the domain and a length `N`.

We found these two basic behaviors for the `randomized=True` agent with `N = 10`:

- The agents does not reach a terminal state and keeps choosing actions at random for the whole duration of the trajectory:

```
(x_0 = (0.0098, 0.0000), u_0 = 4, r_0 = 0, x_1 = (-0.0046, -0.2914))
(x_1 = (-0.0046, -0.2914), u_1 = -4, r_1 = 0, x_2 = (-0.0698, -1.0403))
(x_2 = (-0.0698, -1.0403), u_2 = 4, r_2 = 0, x_3 = (-0.1916, -1.3923))
(x_3 = (-0.1916, -1.3923), u_3 = 4, r_3 = 0, x_4 = (-0.3438, -1.6118))
(x_4 = (-0.3438, -1.6118), u_4 = 4, r_4 = 0, x_5 = (-0.5006, -1.4434))
(x_5 = (-0.5006, -1.4434), u_5 = 4, r_5 = 0, x_6 = (-0.6199, -0.8981))
(x_6 = (-0.6199, -0.8981), u_6 = 4, r_6 = 0, x_7 = (-0.6773, -0.2386))
(x_7 = (-0.6773, -0.2386), u_7 = 4, r_7 = 0, x_8 = (-0.6681, 0.4295))
```

```
(x_8 = (-0.6681, 0.4295), u_8 = 4, r_8 = 0, x_9 = (-0.5926, 1.0774))
(x_9 = (-0.5926, 1.0774), u_9 = -4, r_9 = 0, x_10 = (-0.4975, 0.7818))
(x_10 = (-0.4975, 0.7818), u_10 = -4, r_10 = 0, x_11 = (-0.4416, 0.3142))
```

- The agent reaches a terminal state (here at step `N = 3`, the speed exceeds 3 in absolute value), creating a reward (here -1), then stays in the same `p` and `s` whatever the action taken and collects no rewards:

```
(x_0 = (0.0393, 0.0000), u_0 = -4, r_0 = 0, x_1 = (0.0051, -0.6896))
(x_1 = (0.0051, -0.6896), u_1 = -4, r_1 = 0, x_2 = (-0.1026, -1.5111))
(x_2 = (-0.1026, -1.5111), u_2 = -4, r_2 = 0, x_3 = (-0.3051, -2.5872))
(x_3 = (-0.3051, -2.5872), u_3 = -4, r_3 = -1, x_4 = (-0.6061, -3.1986))
(x_4 = (-0.6061, -3.1986), u_4 = -4, r_4 = 0, x_5 = (-0.6061, -3.1986))
(x_5 = (-0.6061, -3.1986), u_5 = 4, r_5 = 0, x_6 = (-0.6061, -3.1986))
(x_6 = (-0.6061, -3.1986), u_6 = 4, r_6 = 0, x_7 = (-0.6061, -3.1986))
(x_7 = (-0.6061, -3.1986), u_7 = -4, r_7 = 0, x_8 = (-0.6061, -3.1986))
(x_8 = (-0.6061, -3.1986), u_8 = 4, r_8 = 0, x_9 = (-0.6061, -3.1986))
(x_9 = (-0.6061, -3.1986), u_9 = 4, r_9 = 0, x_10 = (-0.6061, -3.1986))
(x_10 = (-0.6061, -3.1986), u_10 = -4, r_10 = 0, x_11 = (-0.6061, -3.1986))
```

With the deterministic agent, we computed the first 20 steps of the trajectory, showing that indeed, as the agent is initially on a slope, it descends it and then starts climbing it but without sufficient momentum does not reach the top and starts going back once again creating a loop:

```
(x_0 = (0.0098, 0.0000), u_0 = 4, r_0 = 0, x_1 = (-0.0046, -0.2914))
(x_1 = (-0.0046, -0.2914), u_1 = 4, r_1 = 0, x_2 = (-0.0486, -0.5913))
(x_2 = (-0.0486, -0.5913), u_2 = 4, r_2 = 0, x_3 = (-0.1226, -0.8918))
(x_3 = (-0.1226, -0.8918), u_3 = 4, r_3 = 0, x_4 = (-0.2256, -1.1575))
(x_4 = (-0.2256, -1.1575), u_4 = 4, r_4 = 0, x_5 = (-0.3493, -1.2806))
(x_5 = (-0.3493, -1.2806), u_5 = 4, r_5 = 0, x_6 = (-0.4718, -1.1133))
(x_6 = (-0.4718, -1.1133), u_6 = 4, r_6 = 0, x_7 = (-0.5629, -0.6706))
(x_7 = (-0.5629, -0.6706), u_7 = 4, r_7 = 0, x_8 = (-0.6027, -0.1111))
(x_8 = (-0.6027, -0.1111), u_8 = 4, r_8 = 0, x_9 = (-0.5852, 0.4624))
(x_9 = (-0.5852, 0.4624), u_9 = 4, r_9 = 0, x_10 = (-0.5128, 0.9705))
(x_10 = (-0.5128, 0.9705), u_10 = 4, r_10 = 0, x_11 = (-0.3987, 1.2633))
(x_11 = (-0.3987, 1.2633), u_11 = 4, r_11 = 0, x_12 = (-0.2711, 1.2400))
(x_12 = (-0.2711, 1.2400), u_12 = 4, r_12 = 0, x_13 = (-0.1575, 1.0096))
(x_13 = (-0.1575, 1.0096), u_13 = 4, r_13 = 0, x_14 = (-0.0710, 0.7129))
(x_14 = (-0.0710, 0.7129), u_14 = 4, r_14 = 0, x_15 = (-0.0148, 0.4093))
(x_15 = (-0.0148, 0.4093), u_15 = 4, r_15 = 0, x_16 = (0.0114, 0.1150))
(x_16 = (0.0114, 0.1150), u_16 = 4, r_16 = 0, x_17 = (0.0086, -0.1752))
(x_17 = (0.0086, -0.1752), u_17 = 4, r_17 = 0, x_18 = (-0.0235, -0.4718))
(x_18 = (-0.0235, -0.4718), u_18 = 4, r_18 = 0, x_19 = (-0.0857, -0.7763))
(x_19 = (-0.0857, -0.7763), u_19 = 4, r_19 = 0, x_20 = (-0.1781, -1.0680))
(x_20 = (-0.1781, -1.0680), u_20 = 4, r_20 = 0, x_21 = (-0.2963, -1.2730))
```

2 Expected Return of a Policy in Continuous Domain

In Reinforcement Learning, we would like to know the expected return of a policy with an infinite time horizon, but since that it is not possible, we found a way to have an upper bound of the infinite norm of the difference between our expected return over a finite time of N steps J_N^μ and the infinite horizon J_∞^μ. The formula for this upper bound was seen during lectures, uses the discount factor γ, the time horizon N and the upper bounds of the rewards $B_r = 1$ and is written as follows:

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

We computed a number of possible values for N, as seen in the Table 1 and the one we chose is $N = 500$ as it provides a good enough upper bound with a manageable computational complexity.

N	$\|J_N^\mu - J_\infty^\mu\|_\infty$
10	11.9747
50	1.5388
100	0.1184
500	1.4549e-10
1000	1.0584e-21
5000	8.2996e-111

Table 1: $\|J_N^\mu - J_\infty^\mu\|_\infty$ with respect to N

Once this upper bound was chosen, we simply had to recreate the same functions as in the first assignment:

- `expected_return_continuous` takes as an argument a domain, an agent and a time horizon $N = 500$ in our case. It simply computes the cumulative return with a discount factor $\gamma = 0.95$ by following the policy of the agent and using the domain's dynamics and reward signal.
- `monte_carlo_simulations_continuous` takes as an argument an agent behavior, a domain class, a time horizon $N = 500$ and a number of initial states of 50 as asked in the assignment. This functions takes advantage of the Monte Carlo principle by averaging the results of each trajectory into a single array.

With this newly computed average over our 50 initial states we can plot the resulting array over the N time steps for both our agents. The results can be seen in Figures 1 and 2.

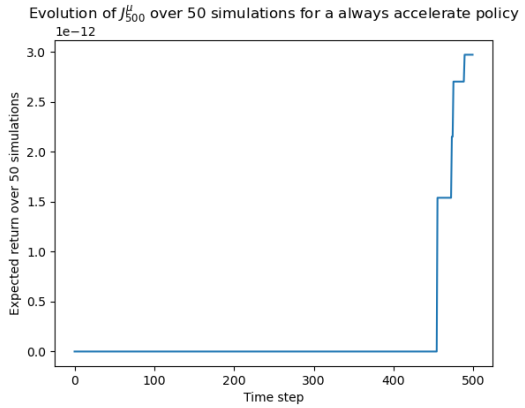


Figure 1: J_{500}^μ for the "always accelerate" agent averaged over 50 initial states

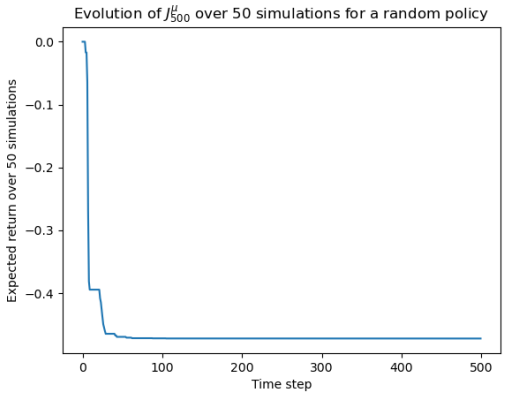


Figure 2: J_{500}^μ for the random action agent averaged over 50 initial states

As seen in the Figure 1, showing the "always accelerate" trajectories, the expected return stays at 0 for more than 400 steps and then as 4 of the trajectories reach a terminal state (not normally possible), it increases but very slightly (e^{-12}) which when compared to the previously discussed bound over the infinite norm $\|J_N^\mu - J_\infty^\mu\|_\infty$ shows us indeed that the expected return is really 0. Indeed, an always accelerate would not reach any of the terminal states thus not collecting any reward.

For the Figure 2, showing the random trajectories, we can see that after around 50 steps, every trajectory has reached the -1 rewarded terminal state thus converging to the value -0.4721. With this behaviour we can confidently say that after 100 random actions, it is very unlikely to be in a idling state.

3 Visualization

For the last two sections we used in addition to standard libraries, `numpy` and `matplotlib`, we used the libraries `imageio` and `tqdm` respectively for image management for the GIFs, and to see the advancements of for loops. We also took advantage of the available `display_caronthehill.py` script.

As for the creation of GIFs, we implemented it in a straightforward manner:

1. Initialize the Agent and Domain, making sure that the initial values of the position and speed are exactly at 0 and not using the uniform as in the previous sections. Create an empty `images` array.
2. Using the Agent, Domain and $N = 500$ maximum steps to launch our "GIF creating" routine.
3. Creating an image with the given script saving the current state of the problem, saving it and copying it's path into the `images` array.
4. Checking if we are in a terminal state, breaking the loop if so exiting the loop and saving the GIF using the `images` array and the `imageio.mimsave()` function.
5. Choosing the action to take using the Agent's policy.
6. Computing the next state of the problem using the Domain's dynamics. Jump to step 3.

The generated GIFs for the randomized and always accelerate agents are respectively available in the files `car_500_randomized_True.gif` and `car_500_randomized_False.gif`. As expected for both the GIFs, the random policy quickly finds itself in a -1 rewarded terminal state as the "always accelerate" policy loops forever on the hill never reaching a terminal state.

4 Fitted Q-Iteration

In this last section of the assignment, we will use the Fitted Q-Iteration with our one-step system transition to try to learn the Q_N functions relative to each action (-4 or 4). For that we will try multiple models to learn those functions based on two different dataset generation techniques and two stopping criteria discussed in the next sections.

The main principle of the Fitted Q-Iteration algorithm is that the Q_N functions are learned in an iterative manner starting from the one-step system transitions and using the Q_N function formula to learn the next iteration of our model. Indeed the learning sets of the models at each iteration will look like:

$$\begin{aligned}
\mathrm{LS}_N &= \{i_N, o_N\} && \text{(respectively for inputs and outputs)} \\
\mathrm{LS}_1 &= \left\{\left((p_k, s_k), u_k\right), r_k\right\}_{k=0}^{k=\text{number of samples per epoch}} \\
\mathrm{LS}_N &= \left\{\left((p_k, s_k), u_k\right), r_k + \gamma \max_{u \in \{-4,4\}} \hat{Q}_{N-1}\left(\left((p_k, s_k), u\right)\right)\right\}_{k=0}^{k=\text{number of samples per epoch}}
\end{aligned}$$

Once the two functions have been learned, we will extract and display the optimal policy by taking the action maximizing the Q_N value in each state of our base problem, discretized with a step of 0.01, creating a representation with a $\frac{1+|-1|}{0.01} = 200$ by $\frac{3+|-3|}{0.01} = 300$.

4.1 One-step System Transitions Generation Methods

In this section we will dive in the two one-step system transitions methods we used to create the initial dataset of our models. These are fairly similar and only differ in a simple aspect. The `osst` array is generated using an `Agent` taking random actions in the `Domain` (reusing the implementation of the previous sections). The `mode` of generation is set by choosing a parameter of the `generate_osst` between `full` and `reduced`.

The `reduced` generation mode generates the trajectories using the same distribution for the initial position of the `Agent` as in the previous sections while for the `full` mode initializes the position of the `Agent`

uniformly between -1 and 1 (the entire possible space). At first glance, the `full` method would generate a dataset more useful for the models as it would explore more of the possible joined position-speed states compared to the `reduced` who would be stuck in an infinite loop exploring the same states over and over again. This will be discussed in the Section 5 by comparing the results obtained by the different methods.

In our implementation, we used combined 100 trajectories of maximum 1000 steps to train our models (maximum because the trajectories are stopped when reaching a terminal state).

4.2 Stopping Criteria

In combination of the two methods to generate the one-step system transitions, we explored two stopping criteria to decide if we needed to stop the training iterations of the models. The two strategies we used are:

- Convergence checking: After each training epoch of our models we compared the predictions made by new and the previous models and computed the infinite-norm between the two. Our first stopping criterion simply looks like this:
$$\|\hat{Q}_N - \hat{Q}_{N-1}\|_\infty < \epsilon$$
The value of ϵ used for this criterion was 0.005, ensuring that the models did not differ in the worst case by more than this bound. If the model did not converge we simply let it run until the step 1000, hard-stopping the training.

- Fixed number of steps: By taking advantage of a formula seen in class on the infinite-norm between expected returns of two consecutive models. The formula is the following:
$$\|J^{\hat{\mu}_N} - J^{\hat{\mu}_{N-1}}\|_\infty \leq \frac{2\gamma^N B_r}{(1-\gamma)^2}$$
For this criterion, we used a bound of 0.05 with the same discount factor $\gamma = 0.95$ and reward upper bound $B_r = 1$ as previously. For this certainty on the expected return, we needed 189 training iterations for our models.

4.3 Models

In this section, we will explore three different models to try predict the Q_N functions by our estimators $\hat{Q}_N$. The Python package used throughout this part to create our regression models is `scikit-learn`[2], taking advantage of the already implemented methods for creating and fitting the models as well as using them to predict new data.

4.3.1 Linear Regression

The first model we used to learn the Q_N function was the `sklearn.linear_model.LinearRegression` regressor. This model does not take any hyperparameter as input. The $\hat{Q}_N$ functions learned for each action (either -4 or 4) are displayed in:

- Figure 3 for the model learned with the reduced OSST and convergence checking.
- Figure 4 for the one learned with the full OSST and convergence checking.
- Figure 5 for the one learned with the reduced OSST and expected return certainty.
- Figure 6 for the one learned with full OSST and expected return certainty.

From those we can see that the stopping criteria does not influence the model much but the learning set influences it much more as the four figures trained with the same learning set look alike. In every case, the -4 value looks like a "up shifted" version of the $+4$ value, this would mean that the model should always choose the same action as the shape of a linear model regression does allow much complexity.

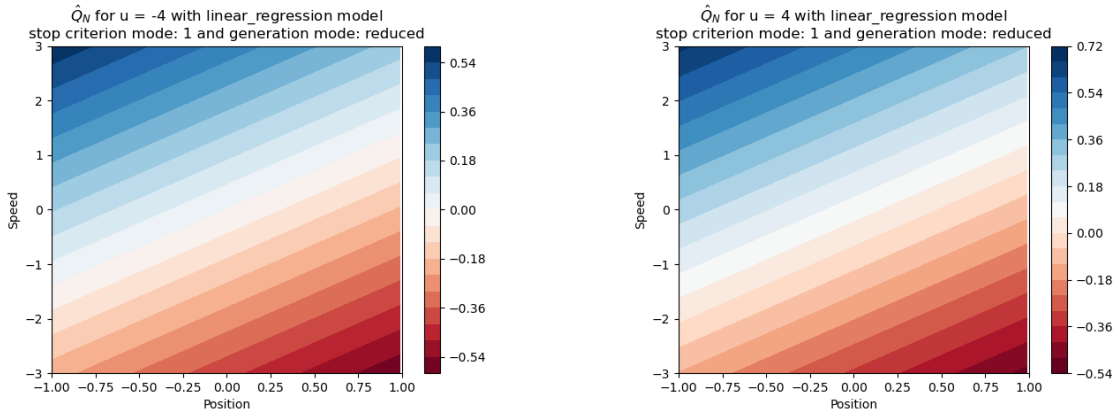


Figure 3: $\hat{Q}_N$ learned from the linear regression model trained with reduced OSST and convergence checking

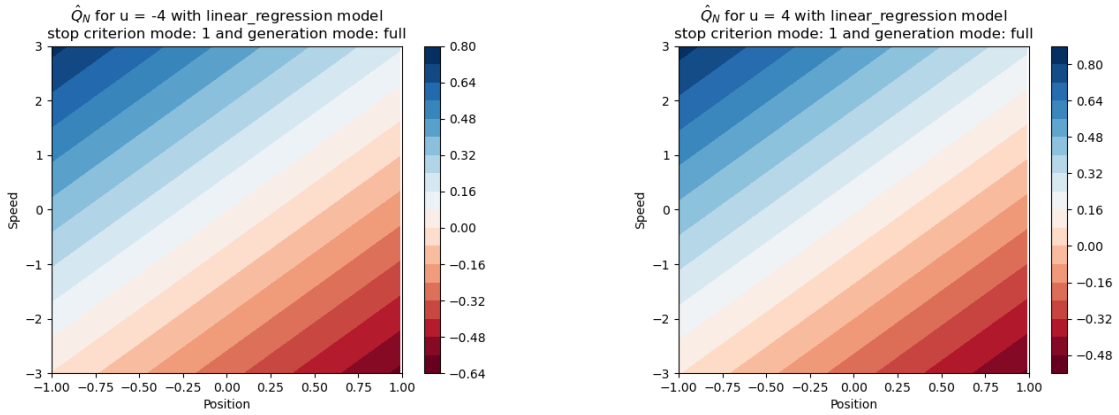


Figure 4: $\hat{Q}_N$ learned from the linear regression model trained with full OSST and convergence checking

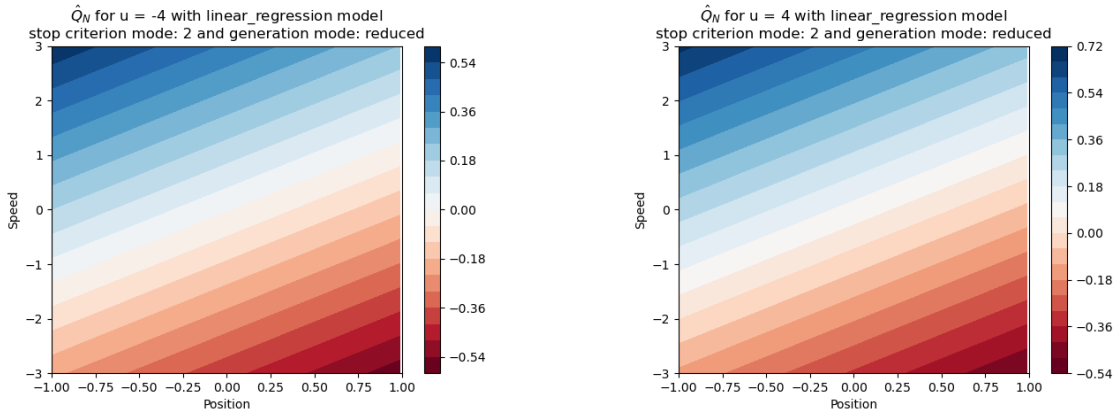


Figure 5: $\hat{Q}_N$ learned from the linear regression model trained with reduced OSST and return certainty

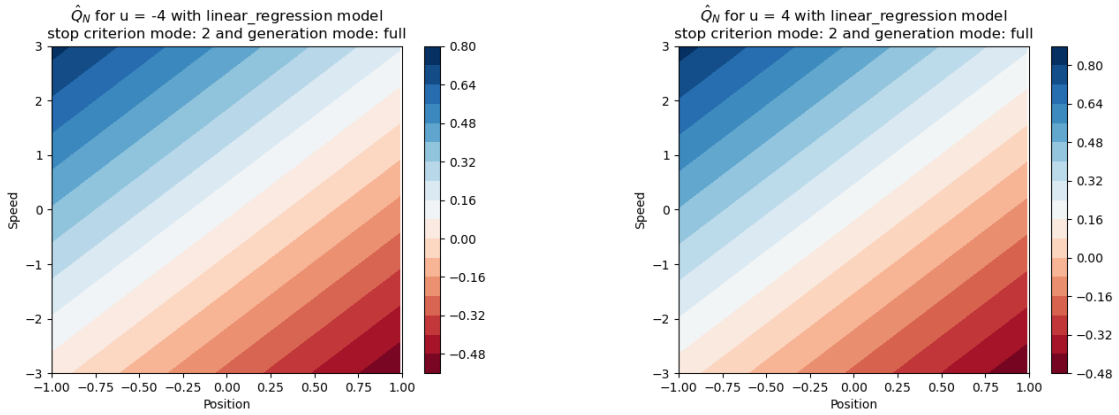


Figure 6: $\hat{Q}_N$ learned from the linear regression model trained with full OSST and return certainty

From those $\hat{Q}_N$, we extract the policies by choosing the action that maximizes the value of $\hat{Q}_N$ and display it as mentioned in the beginning of the section with representing a -4 action and blue a $+4$ action:

- Figure 7 for the model learned with the reduced OSST and convergence checking.
- Figure 8 for the model learned with the full OSST and convergence checking.
- Figure 9 for the model learned with the reduced OSST and expected return certainty.
- Figure 10 for the model learned with the full OSST and expected return certainty.

As expected from the $\hat{Q}_N$ functions, only a single action could have been taken by the linear regression model and a -4 action would have produced a negative reward, lower than a one equal to 0. But as mentioned in the first sections, this agent will never reach the top of the hill.

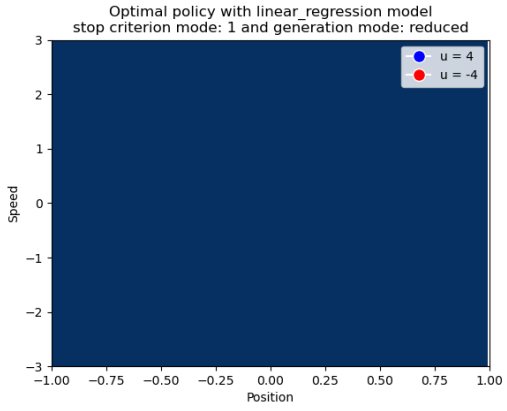


Figure 7: Policy derived from the $\hat{Q}_N$ functions learned by the linear regression with the reduced OSST and convergence checking

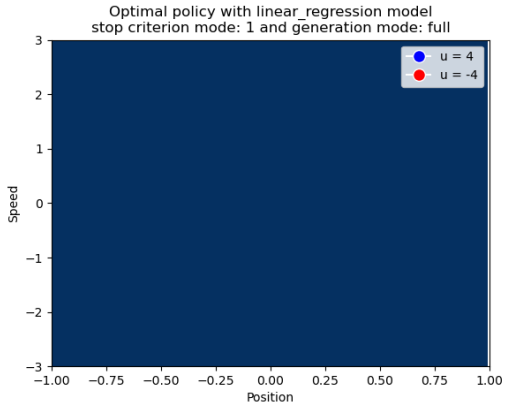


Figure 8: Policy derived from the $\hat{Q}_N$ functions learned by the linear regression with the full OSST and convergence checking

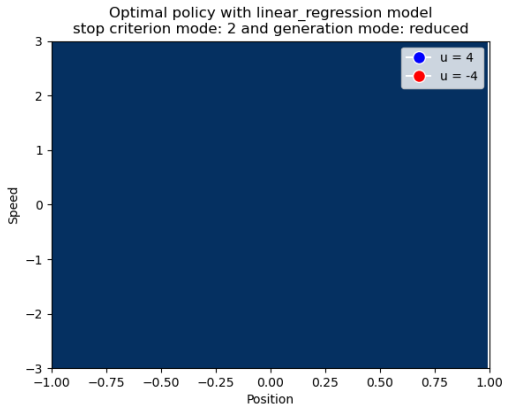


Figure 9: Policy derived from the $\hat{Q}_N$ functions learned by the linear regression with the reduced OSST and return certainty

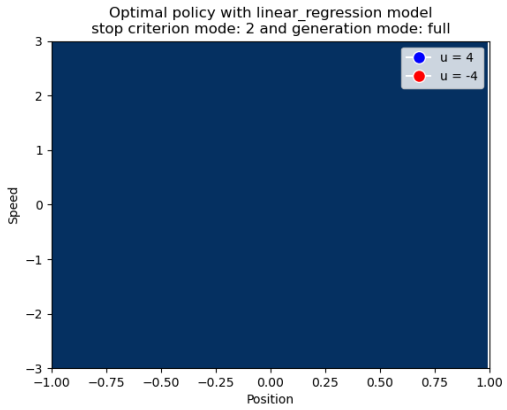


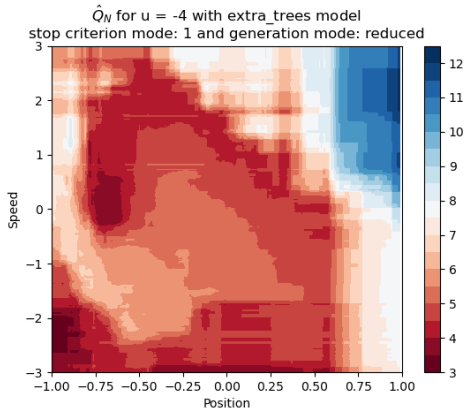
Figure 10: Policy derived from the $\hat{Q}_N$ functions learned by the linear regression with the full OSST and return certainty

4.3.2 Extremely Randomized Trees

For the second model, we used the `sklearn.ensemble.ExtraTreesRegressor` regressor. As seen while reading the research from the scientific paper [1], we set the number of trees to 10 as it is said that "We observe that the score grows rapidly with M, especially with Extra-Trees and Tree Bagging in which cases a value of M = 10 would have been sufficient to obtain a good solution.". The $\hat{Q}_N$ functions learned for each action (either -4 or 4) are displayed in:

- Figure 11 for the model learned with the reduced OSST and convergence checking.
- Figure 12 for the one learned with the full OSST and convergence checking.
- Figure 13 for the one learned with the reduced OSST and expected return certainty.
- Figure 14 for the one learned with full OSST and expected return certainty.

With this model, we can clearly see that the model boundaries seem to be much more adapted to the problem at hand. Indeed, for the car to be able to climb the hill, it needs to go back and then accelerate, thus needing two "behaviours" based on the position of the car and its speed.



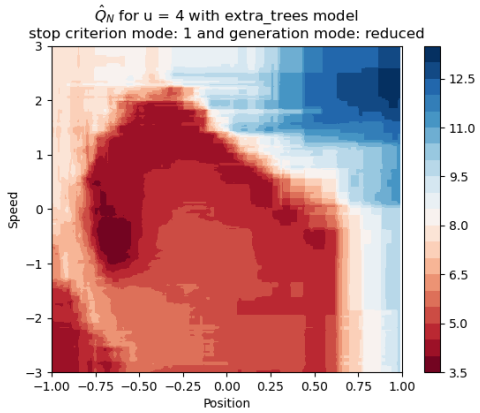


Figure 11: $\hat{Q}_N$ learned from the extremely randomized trees model trained with reduced OSST and convergence checking

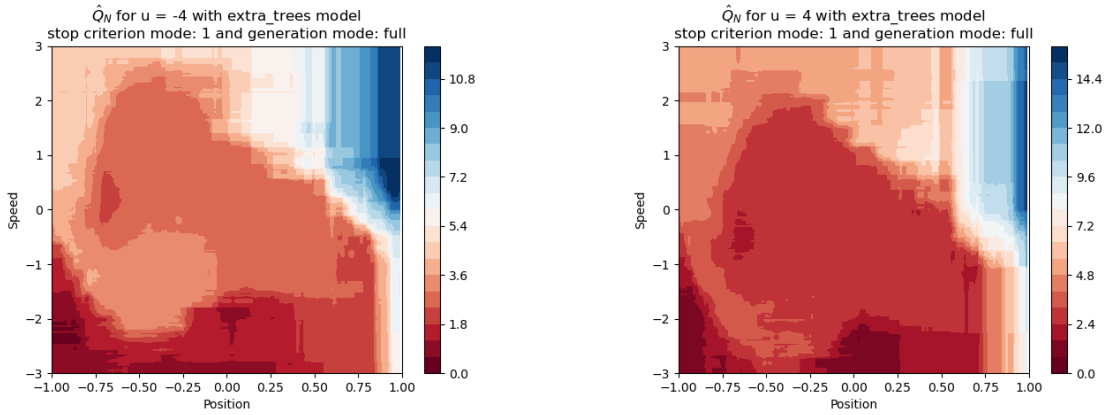


Figure 12: $\hat{Q}_N$ learned from the extremely randomized trees model trained with full OSST and convergence checking

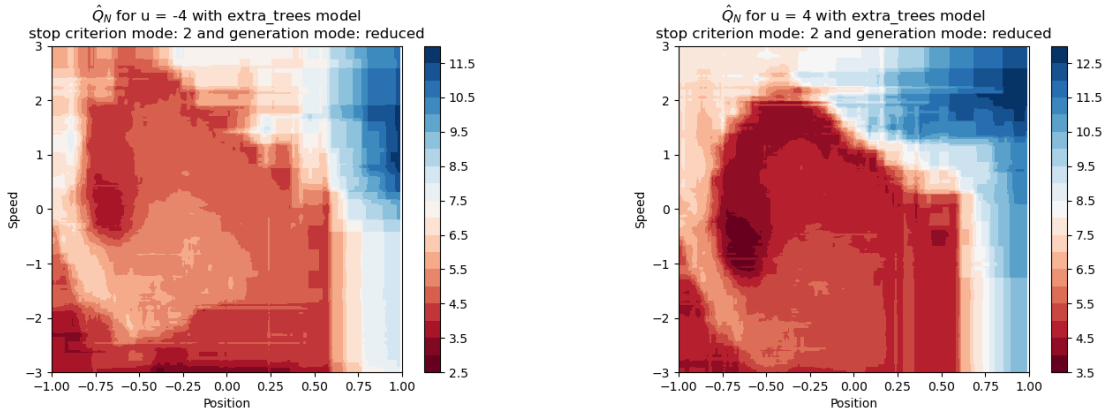


Figure 13: $\hat{Q}_N$ learned from the extremely randomized trees model trained with reduced OSST and return certainty

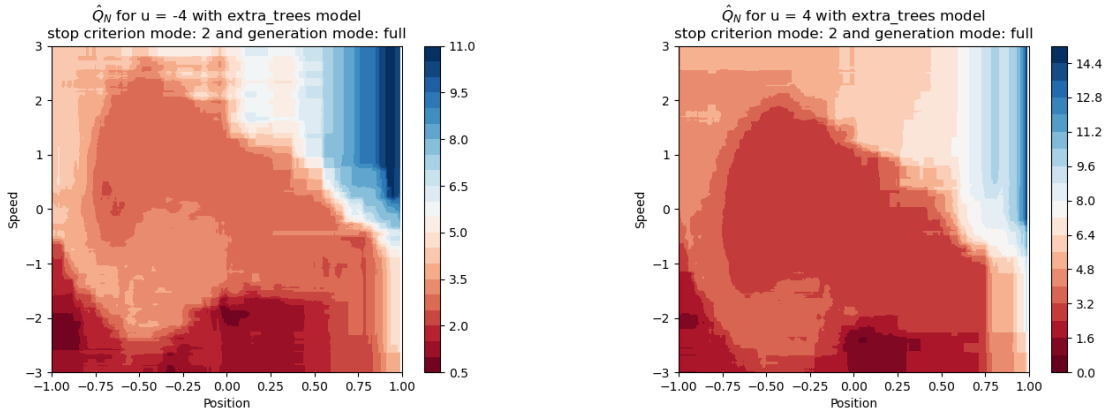


Figure 14: $\hat{Q}_N$ learned from the extremely randomized trees model trained with full OSST and return certainty

From those $\hat{Q}_N$, we extract the policies by choosing the action that maximizes the value of $\hat{Q}_N$ and display

it as mentioned in the beginning of the section with representing a -4 action and blue a $+4$ action:

- Figure 15 for the model learned with the reduced OSST and convergence checking.
- Figure 16 for the model learned with the full OSST and convergence checking.
- Figure 17 for the model learned with the reduced OSST and expected return certainty.
- Figure 18 for the model learned with the full OSST and expected return certainty.

When looking at these policies, it is hard to say which of the four will produce the best results at first glance but they all seem to converge towards the same "2-mode" pattern, first starting by going back but avoiding the -1 reward and then full gas with the gained momentum to go above the crest of the hill picking up the $+1$ reward.

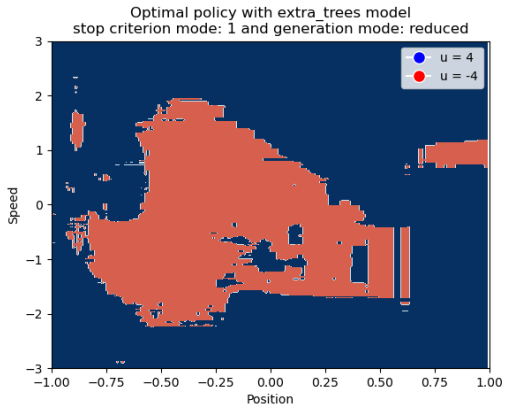


Figure 15: Policy derived from the $\hat{Q}_N$ functions learned by the extremely randomized trees with the reduced OSST and convergence checking

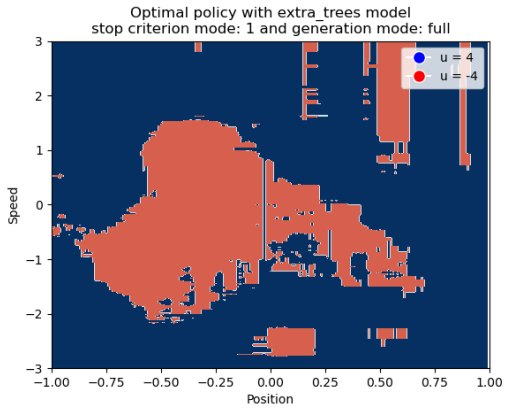


Figure 16: Policy derived from the $\hat{Q}_N$ functions learned by the extremely randomized trees with the full OSST and convergence checking

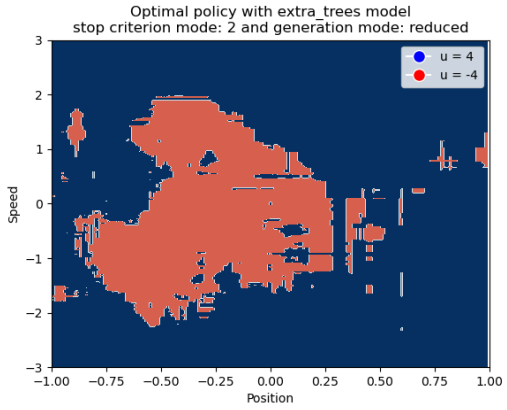


Figure 17: Policy derived from the $\hat{Q}_N$ functions learned by the extremely randomized trees with the reduced OSST and return certainty

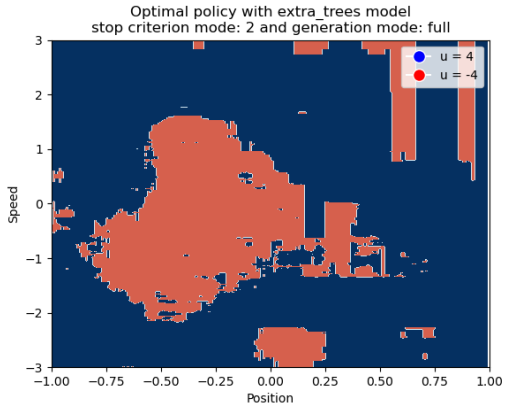


Figure 18: Policy derived from the $\hat{Q}_N$ functions learned by the extremely randomized trees with the full OSST and return certainty

4.3.3 Neural Network

For the third and last model, we used the `sklearn.neural_network.MLPRegressor` regressor. For the design of the neural network, we had 3 values as input (position, speed and action) and 1 as output, the value of

$\hat{Q}_N$. At first, with our small deep learning background, we decided to make a neural network with a rectified learning unit activation function (ReLU) and hidden layers following the shape $\{16, 32, 64, 32, 16, 8\}$ as it seemed to be a good idea but it did not generate any good results.

We then had the idea to change the activation function from ReLU to tanh and everything started to work better. It seemed weird at first as hyperbolic neural network are basically better for classification than regression problems but in our case, we think that it acts as a classification task between going frontwards or backwards capturing the non-linear aspect of the probem (hill equation, its derivatives, the discrete possible actions, ...). The $\hat{Q}_N$ functions learned for each action (either -4 or 4) are displayed in:

- Figure 19 for the model learned with the reduced OSST and convergence checking.
- Figure 20 for the one learned with the full OSST and convergence checking.
- Figure 21 for the one learned with the reduced OSST and expected return certainty.
- Figure 22 for the one learned with full OSST and expected return certainty.

At first glance, the neural networks seem to produce much smoother results than the extremely randomized trees. On a second look, only the networks trained with the `full` OSTT generation seem to produce the same results as the extremely randomized trees but we will dive in the analysis of the results in the Section 5.

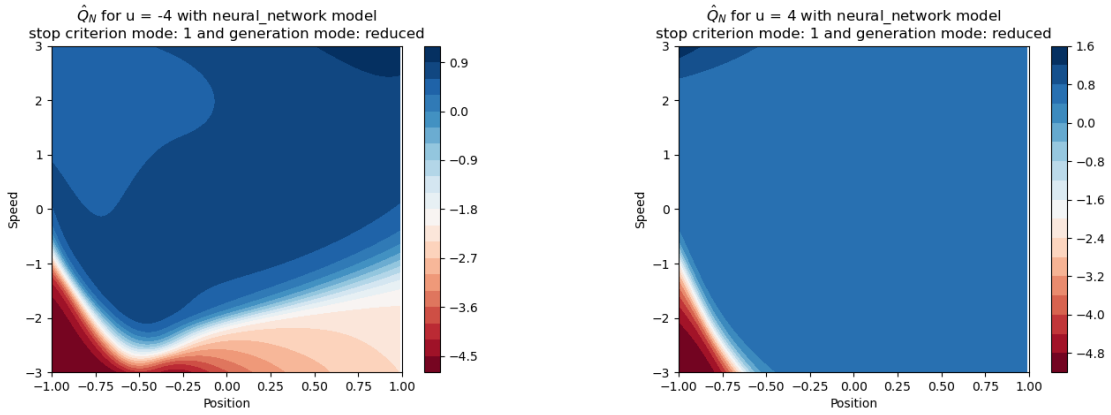


Figure 19: $\hat{Q}_N$ learned from the neural network model trained with reduced OSST and convergence checking

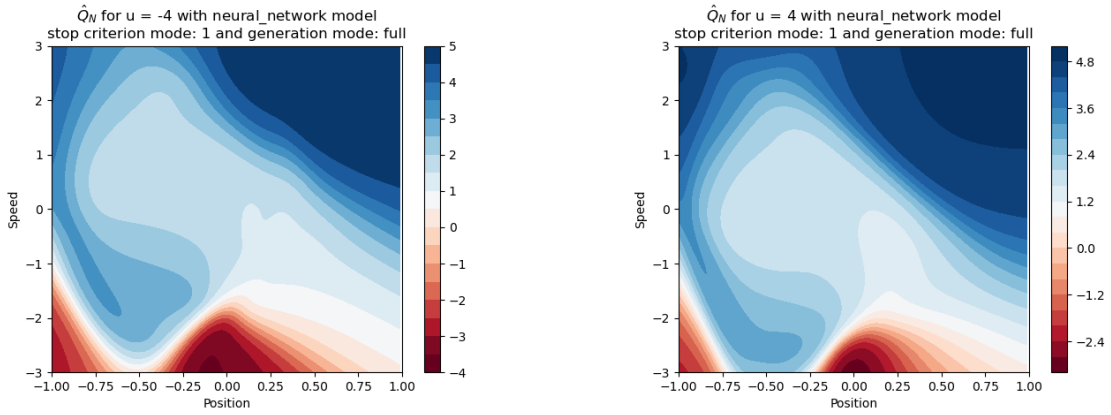


Figure 20: $\hat{Q}_N$ learned from the neural network model trained with full OSST and convergence checking

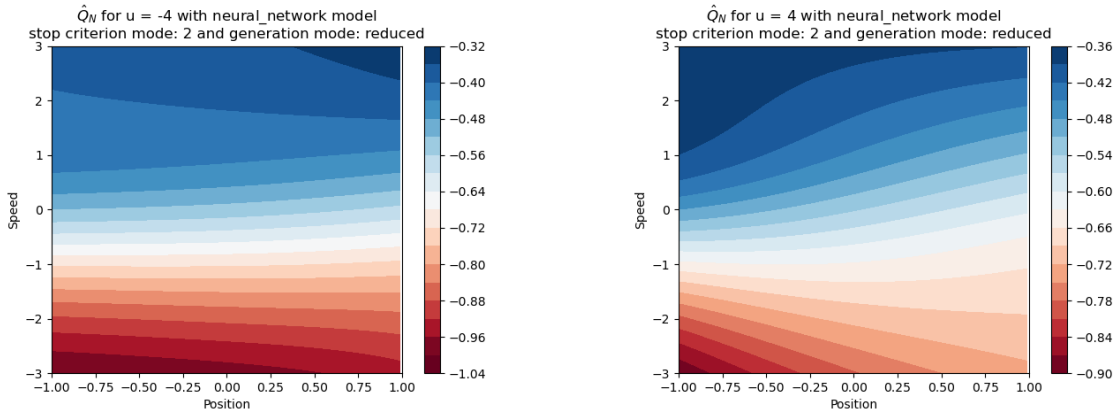


Figure 21: $\hat{Q}_N$ learned from the neural network model trained with reduced OSST and return certainty

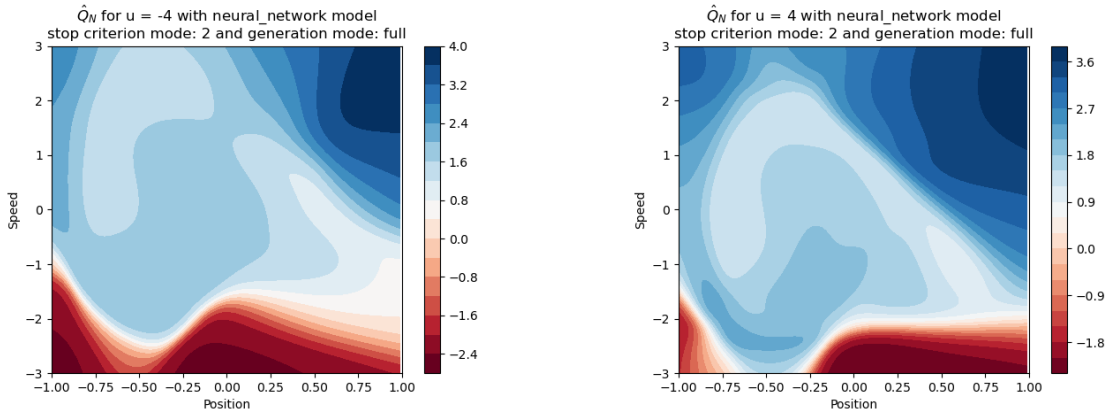


Figure 22: $\hat{Q}_N$ learned from the neural network model trained with full OSST and return certainty

From those $\hat{Q}_N$, we extract the policies by choosing the action that maximizes the value of $\hat{Q}_N$ and display it as mentioned in the beginning of the section with representing a -4 action and blue a $+4$ action:

- Figure 23 for the model learned with the reduced OSST and convergence checking.
- Figure 24 for the model learned with the full OSST and convergence checking.
- Figure 25 for the model learned with the reduced OSST and expected return certainty.
- Figure 26 for the model learned with the full OSST and expected return certainty.

Once again, by looking at the policies derived from the $\hat{Q}_N$ functions we can see this "2-mode" behaviour on the `full` OSST trained models. Indeed they have a very similar look than the extremely randomized trees ones but looking more smooth on the borders. The ones trained with the `reduced`, on the other hand, look nothing alike what we saw earlier. We will see how these models perform in Section 5.

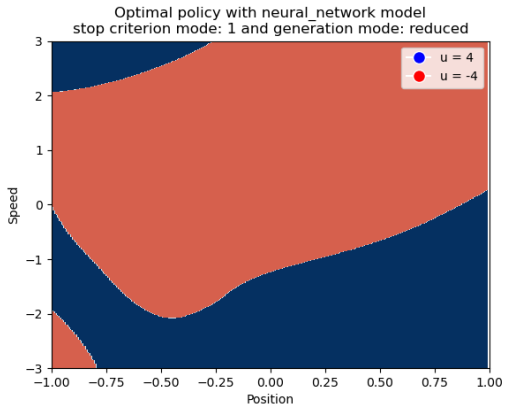


Figure 23: Policy derived from the $\hat{Q}_N$ functions learned by the neural network with the reduced OSST and convergence checking

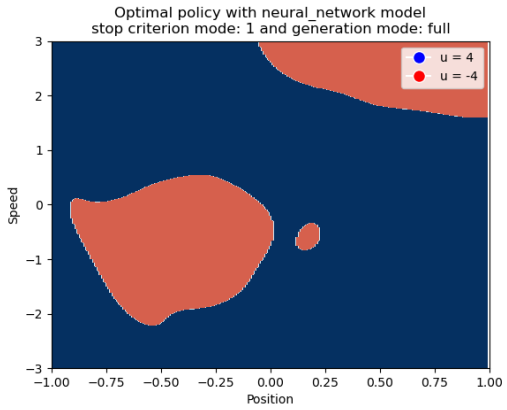


Figure 24: Policy derived from the $\hat{Q}_N$ functions learned by the neural network with the full OSST and convergence checking

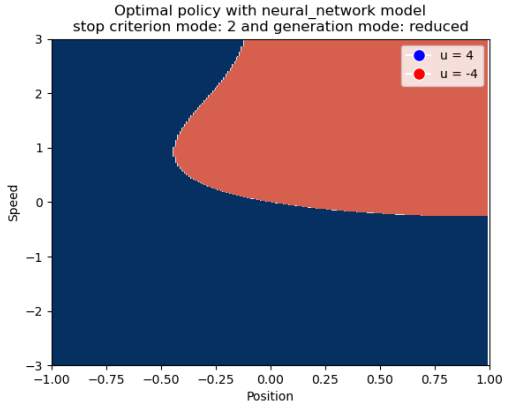


Figure 25: Policy derived from the $\hat{Q}_N$ functions learned by the neural network with the reduced OSST and return certainty

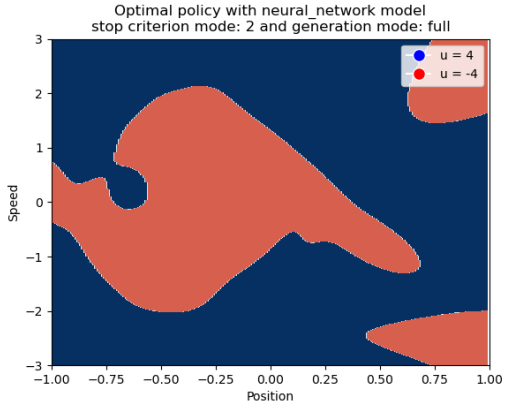


Figure 26: Policy derived from the $\hat{Q}_N$ functions learned by the neural network with the full OSST and return certainty

4.4 Discussion

With all our models trained on all possible combinations of data generation techniques and stopping criteria, we can now assess their performance of the main problem at hand, do they reach the top of the hill in a good way ? We will discuss of the performance achieved based on training complexity and their actual results gathered using our `main` script that dumps the results into a `.csv` file.

4.4.1 Training Complexity

The only model that was fast and easy to train was the `LinearRegression` as it did not need any hyperparameter to tune and that training converged fast, leading into less iterations than the two other implementations. But when looking at the results produced, going fast is not interesting as the models are too simple to solve the task, never being able to reach the top of hill, whatever the training data and stopping criterion.

The two other types of models, `ExtraTreesRegressor` and `MLPRegressor` took around the same time to train but differ in terms of choosing the parameters. Indeed, finding the good number of trees was straightforward by reading the paper [1] but the neural network took a more trial and error approach.

4.4.2 Actual Results

After we trained our models and created the different figures, we computed the expected returns using a Monte Carlo approach with, once again, 50 different initial states. We then created one GIF file per model-OSST strategy-stopping criterion combination. These GIF file are part of the submission under the name `car_500_{model_type}_{stopping_criterion}_{OSST_generation}.gif`, where everything between brackets needs to be replaced by the combination you want to look at. After the GIFs were generated, we stored the different statistics of each combination in a `.csv` file per `model_type` category. The results gathered are displayed in the Table 2.

Model Type	Linear Regression				Extra Trees Regressor				MLP Regressor			
OSST mode	Reduced		Full		Reduced		Full		Reduced		Full	
Stopping criterion	1	2	1	2	1	2	1	2	1	2	1	2
Expected Return	0	0	0	0	0.404	0.406	0.388	0.386	0	0	0.313	0.348
Training Iterations	65	189	52	189	1000	189	1000	189	1000	189	1000	189
Steps to reach the top	/	/	/	/	19	18	19	19	/	/	22	20

Table 2: Results obtained by each model type - OSST generation technique - stopping criterion combination

Once again, as expected from previous observations, the `ExtraTreesRegressor` is the best model we have, always being able to reach the top in a fast way whatever the dataset it has been trained on and whatever the stopping condition. The `LinearRegression`, as expected does not produce any satisfying results, never able to reach the top. For the neural networks however, the models produced in the `full` OSST generation technique are able to reach the top of the hill quickly but the `reduced` ones cannot.

5 Conclusion

With all the knowledge gathered during this project, we can safely say that the `ExtraTreesRegressor` is the best model we tried for solving the car on the hill problem. It is much easier to train than the neural network and always produces satisfying results compared to the neural networks trained on the `reduced` OSST datasets. We are convinced that the neural networks could have produced the same results as the `ExtraTreesRegressor` if we have tried more implementations, but once again the `ExtraTreesRegressor` provide good enough results compared to the work needed to optimize the neural networks.

As for the different strategies used for OSST generation and stopping criteria, we can say, just by looking at the neural network, that the `full` generation technique seems to provide better results for the models, since starting from diverse positions will create more versatile models. After discussing the results, we could have chosen a smaller number of maximum time steps for the OSST arrays to avoid some long infinite loops as we said in the first sections that around 50 steps were enough to reach a terminal state when using a random strategy. This could have made our models a little better, especially the `reduced` models of `MLPRegressor`. For the different stopping criteria, it is hard to discuss the results obtained as the iterations of the random forests and neural networks are hard to distinguish after long training times but we are sure that the first method is more complex as we have to predict some data with the models at each training iteration compared to just computing the bound on the expected return for the second method.

References

[1] D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(18):503–556, 2005.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.