Faculty of Applied Sciences
University of Liege
March 27, 2019

-

Optimal decision making for complex problems:

Assignment 2 - Section 1 to 6

s141770
Derroitte Natan

2 Implementing the domain

The first part of the statement asked to implement the domain in which the agent, the car, will evolve during this project.
In order to test this implementation, a simple policy was created. The program therefore allows to easily generate, according to the given argument, 3 very simple policies: always move to the right, always move or take a random action.
Formally, this can be summarized by

$$\mu(x) = \begin{cases} 4 & \forall x \quad OR \\ -4 & \forall x \quad OR \\ \{-4, 4\} & \text{chosen randomly} \quad \forall x \end{cases}$$

Since no initial state or policy is recommended in the statement, it was chosen to take the initial state $x = (-0.99, 0.0)$ and the policy of always moving to the right. The corresponding result is that the car arrives at the top of the hill. These initial values have been chosen arbitrarily as examples and can therefore be modified.

In order to visualise the result, the implementation developed for question 4 was already used. The corresponding video has been jointed in the archive under the name `Q2.avi`.

3 Expected return of a policy in continuous domain

In order to estimate the expected return of a policy in continuous domain, two methods were considered. The first is to take a large number of trajectories from random initial states, following the policy. For each trajectories, it is then necessary to calculate the expected cumulative reward. When the number of

trajectories is high, the average of the expected cumulative reward will tend to average $J(x) \forall x \in X$.
The second consists in not relying on trajectories with a random initial position. Instead, one goes through the states that can take $x \in X$ per small step and creates a trajectory, still following to the policy, for each of these states. From these trajectories, estimating the expected return of the policy is the same as with the first method : calculated the expected cumulative reward for each trajectory and averaged over all trajectories.
The problem with this method is that if the step is too small, many initial states will be considered and the computation time will be too long. On the contrary, if the step is too large, the precision of the technique is too low...
The main advantage of this second method however is that if the step is chosen properly, it will need a lot less trajectories to converge towards the expected return.

In both case, the method used is based on the expected cumulative reward in deterministic case which is :

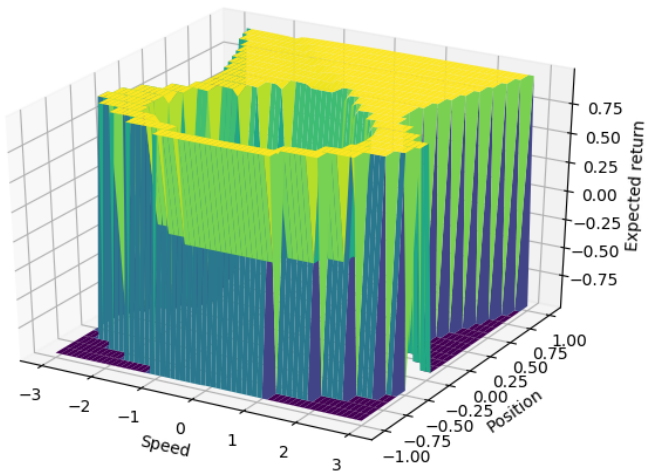
$$J^{\mu}(x) = \lim_{T->\infty} \left[\sum_{t=0}^{\infty} \gamma^t r(x_t, u_t \sim \mu(h_t, ..)) | x_0 = x \right] \tag{1}$$

This formula will be use to compute the expected cumulative reward of the trajectories in our implementation.

The second method was chosen with a step of 0.1 for position and speed for its shorter computation time and good accuracy. The results can be seen in the figures 1 and 2. These are computed for trajectories of sizes 10000. In order to recover the t of formula (1), it is mandatory to consider the integration time step: 0.001.
From then on, it appears that:

$$t = experience\_number * 0.001 \tag{2}$$

For the figures 1, the policy of always moving to the right is considered. The expected return of the policy in this case is 0.2739. This represents the average of $J(x) \forall x$ and can be seen as the average final reward when the policy of always moving to the right is adopted. The figures 1 represent $J(x)$. It can be seen that depending on the speed and initial position, different states can be reached. It appears that the agent has a good chance of reaching the winning state, where the reward is 1, especially when the initial position is high.



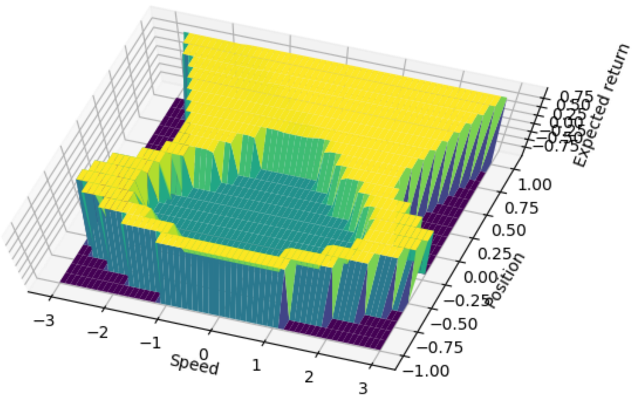
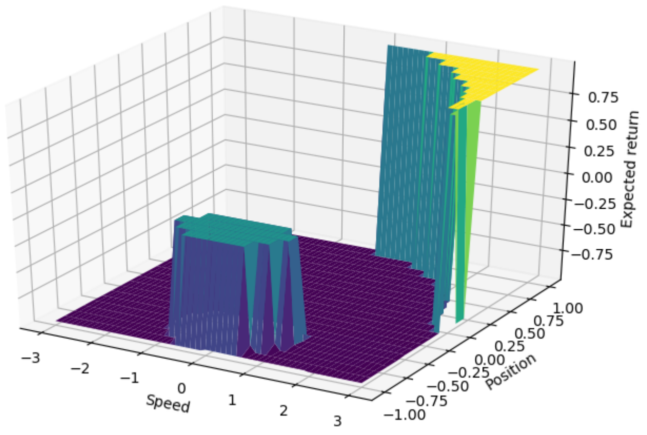


Figure 1: $J^{\mu}(x) \forall x \in X$ when $\mu(x) = 4 \forall x \in X$.

As shown in the figures 2, when the policy is to always move to the right, the chance of reaching a winning final state ($J(x) = 1$) is much lower but the chance of being in a losing final state is higher. Extremely favourable initial conditions are required to reach a winning state.
The expected return of this policy is -0.7360.



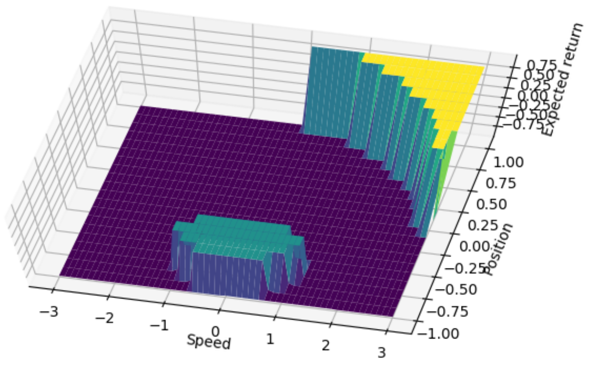


Figure 2: $J^{\mu}(x) \forall x \in X$ when $\mu(x) = -4 \forall x \in X$.

The problem being entirely discrete, the use of the Monte Carlo principle results to making an arithmetical sum of a single element. However, the code has been designed in such a way that it is scalable and can therefore be applied to a stochastic case.

4 Visualisation

In order to visualise the calculated results, a method allowing to generate a video from a series of states has been implemented. To do so, the `opencv` package was used.
As mentioned above, an example video has been placed in the archive under the name `Q2.avi`.
Some frames of this video are shown in the figure 4.

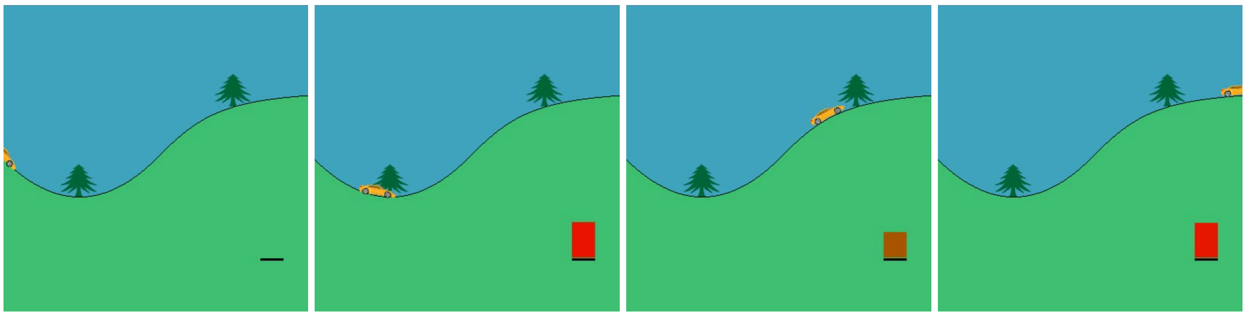


Figure 3: Some frames from the video generated with `opencv`.

The simultaneous use of `opencv` and `pygame` on the *macOS* operating system can cause problems. Beyond 928 frames recorded, the video cannot be recorded and the program will shut down.
The origin of this error is purely due to the fact that these packages were not originally developed for *macOS* and that the versions developed for this system still contain some bugs. Under no circumstances does the segmentation fault originate in our code.
The simulation time should therefore not exceed 36 seconds for *macOS* but is not limited for other operating systems. No problems were detected on *Linux*, no matter how many frames were recorded.

5 Fitted Q Iteration

5.1 Creation of the learning sample

The first thing that needs to be explained is how the learning sample can be created. Different approaches have been tested and the best one has been preserved. This first part of this question presents these approaches.

5.1.1 Uniform experience drawing

The first method considered is based on what was also used in question 3. This consists in creating a mesh of our state domain and taking as input all these states.
This method has several advantages: the first is that the accuracy of the mesh can be seen as a parameter. The smaller it is, the more states will be studied. This makes it possible to very quickly obtain results with a rough mesh, in order to test algorithms for example, and then consider a finer mesh to get the final results.
Another advantage is that this technique will consider the whole field uniformly. There will be no area where knowledge will be lower, which would lead to holes in the algorithm's learning curve.
The latter advantage can also be seen as a disadvantage: some parts of the field may require more study and knowledge than others. A uniform distribution would not be appropriate in this case, but it does not concern our problem.

In practice, this technique allows very good results to be obtained in a short calculation time.

5.1.2 Totally random experience

The second method simply consists of considering a number t of state. Once this number t is set, it is sufficient to randomly draw t states from the number of state distributions.
For this method to be effective, t must absolutely be high. Otherwise, there is a very good chance that all the states considered do not understand certain parts of the field. From then on, this will be marked by learning holes in the machine learning algorithm.

Although effective when t is very large (~ 10000), the computation time is therefore greater for results similar to the first method.

5.1.3 Using a trajectory

The last technique consists in not taking states independent of each other but considering states that are in a single trajectory. After each iteration, the algorithm creates a new trajectory following the policy derived by the last Q computed.
Although very simple, both from an understanding and an implementation point of view, this technique has a major flaw: all the states will be linked and therefore dependent. The learning sample will therefore be composed of non-independent data, which will have a major impact on the algorithm's results.

5.1.4 Discussion

The first method, for its best results, was considered for this question. The figure 4 is used to justify this choice. The 3 previous approaches were tested using the same algorithm (Extremly randomized trees) and the same number of iterations (100).

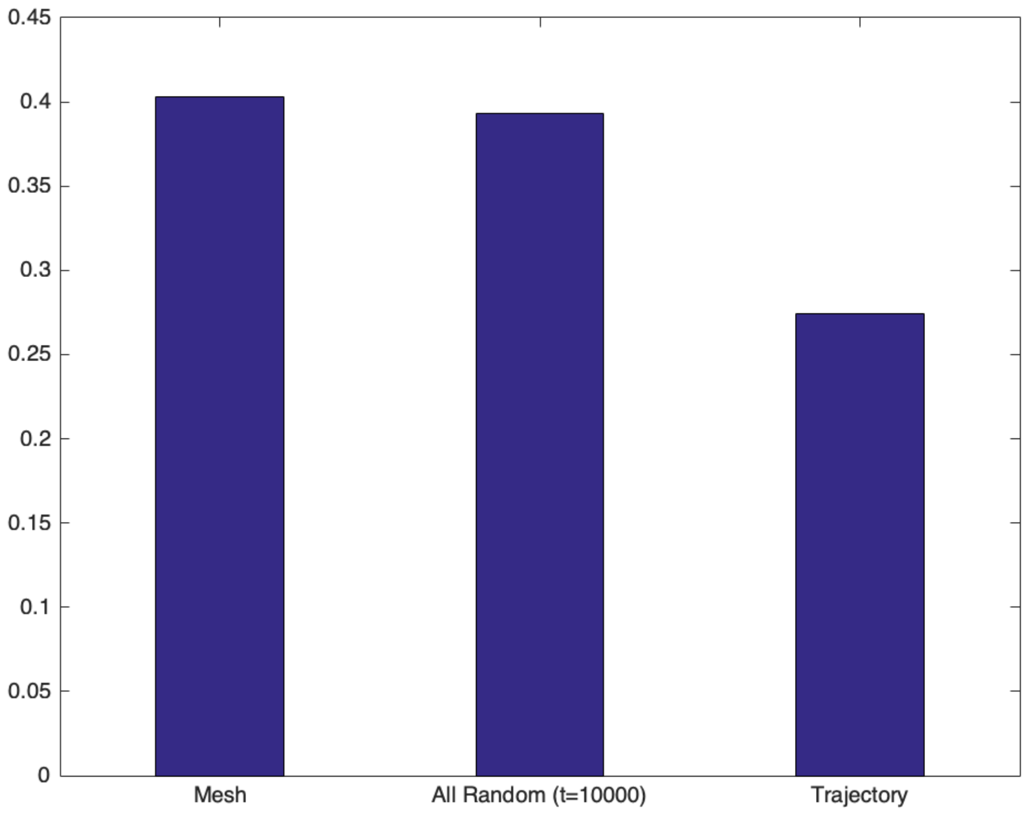


Figure 4: Comparison of the different learning sample creation technique. On the y axis, one can observer the expected return.

From this image, the result clearly appears to be better when the first best result is presented.

5.2 Discussion about the form of $Q(x, u)$

Before explaining how the different machine learning algorithms were chosen and implemented, this section details the shape of the $\hat{Q}(x, u)$ that the Q fitted Iteration tries to approximate.

The problem presented in this question is different from the previous assignment because it has no reward for intermediate states. He only has an ultimate reward of -1 or 1 if the car "loses" or "wins".
From this point on, it can be deduced that if x_m is the state just before winning and u_0 the action that allows to reach the top of the mountain from x_m (the final state), $Q(x_{m+1}, u_0)$ must be equal to 1.

At the iteration N, the predictions on which machine learning algorithms learn are in the form $y_k = r_k + \gamma \max_{u^* \in U} Q_{N-1}(x_{m+1}, u^*)$. In the case where $k = m$, we will have $y_k = \gamma$.
For the same reason, it is possible to deduce the exact value of y_0 corresponding to x_0. This one will be equal to γ^m. This is consistent since all rewards up to m, i. e. the intermediate state rewards, are nil.
We can therefore conclude the following reasoning:

$$y = \begin{cases} \gamma^m & \text{if last state is the winning one} \\ -\gamma^m & \text{if last state is the losing one} \\ 0 & \text{if no final state is reached} \end{cases} \tag{3}$$

This reasoning only holds water if the machine learning algorithms perfectly approximate Q, which is not in practice the case. In the following section, a less theoretical and more practical approach will be discussed.

5.3 Supervised Machine Learning Algorithms

Three algorithms were tested to approximate $Q(x, u)$. This section will detail their implementation when necessary and compare the results obtained.

The first two algorithms did not require much parameterization. Linear Regression (`LR`) and Extremely Randomized Trees (`ERT`) are both implemented in the *skitlearn* package that was used to implement this question.
Skitlearn was also used to implement neural networks (`NN`). Working on 3 different features, it was decided to add 3 layers to our network.

The comparison of the results can be seen in Figure 5. This also includes the evaluation of policies in the event that the agent moves uniformly in one direction.

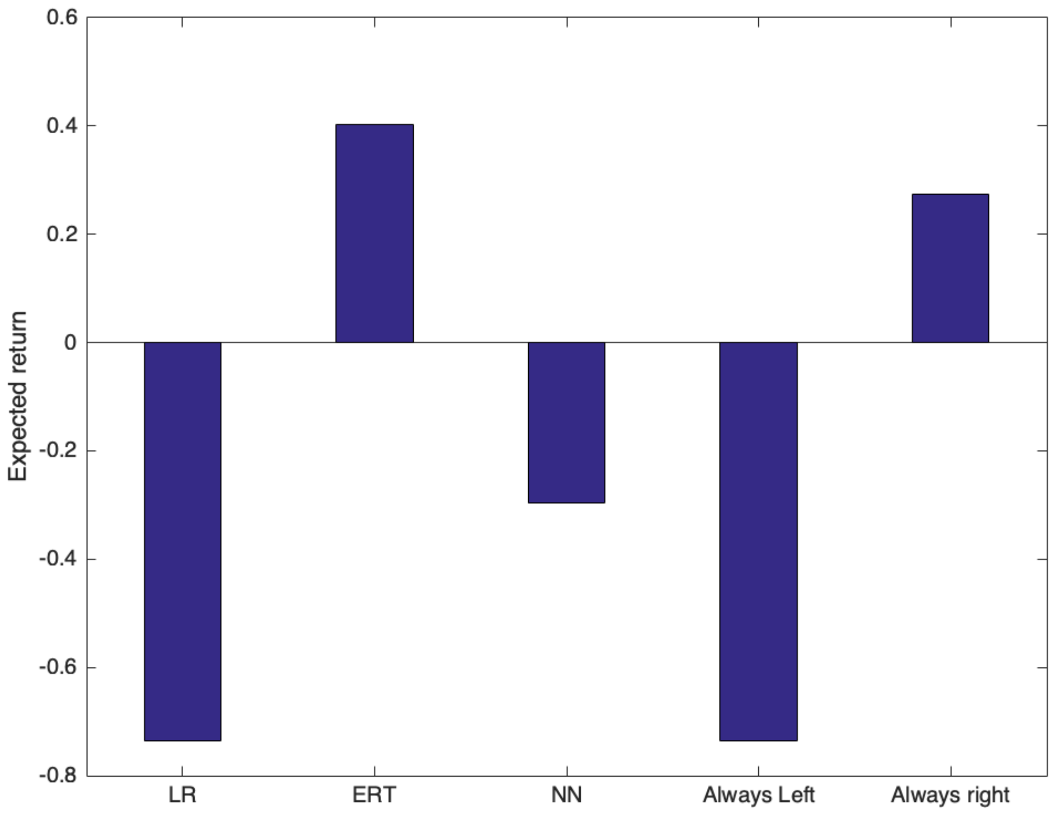
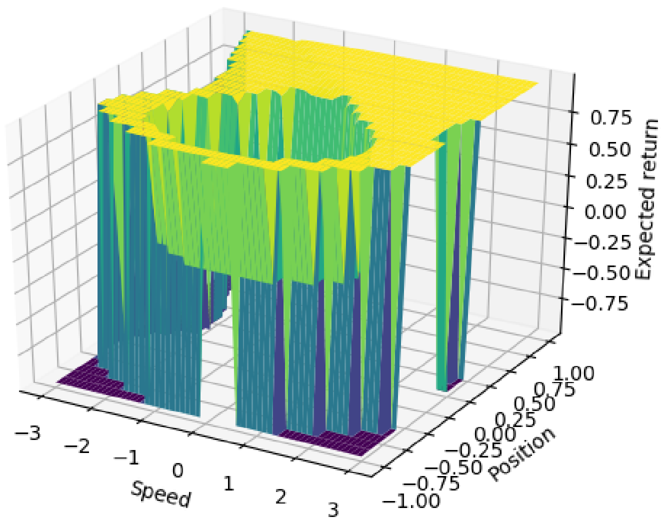


Figure 5: Comparison of the expected return for the Q Fitted Iteration Algorithm using 3 different Machine Learning 3 algorithms implemented for 100 iterations.

It may be observed that the Linear Regression algorithm did not converge to a plausible Q and then proposes a policy of always moving to the left.

The implemented Neural network also offers poor results. This is probably due to the lack of network configuration. This approach was not favoured as it seemed less appropriate to the problem.

Finally, the ERT algorithm offers a better result, a clear improvement over the simplistic policy. A 3D graph, in figure 6, of these results allows to be related to the image 1.



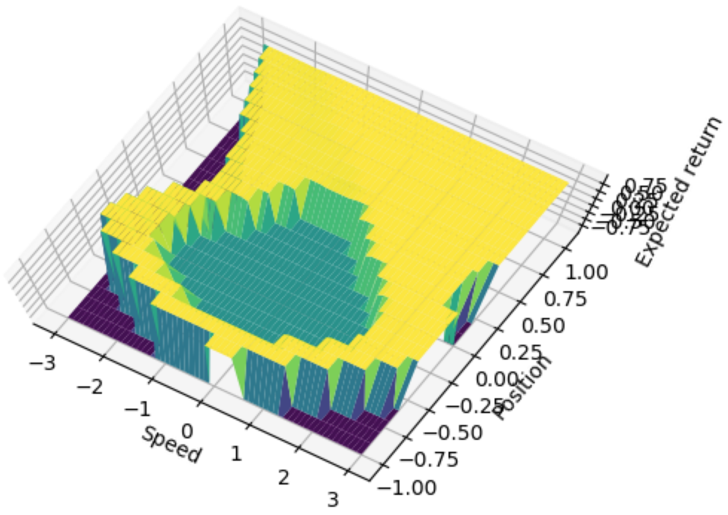


Figure 6: $J^{\mu}(x) \forall x \in X$ when $\mu(x)$ is determined using Q fitted iteration algorithm with ERT.

This comparison shows that many states where speed was high were not won, probably because the speed exceeded the maximum value. The agent has therefore learned to avoid this "engine overheating".

It appears that there is still a zone where the car doesn't reach a terminal state. This can be solved by considering a higher number of iterations.

The video of the car for a specific starting position can be obtain by uncommenting the visualisation line in the `run.py` file. The starting position can be set at this same place.

6 Parametric Q Learning

The problem having an entirely discrete action space, the use of a parameter in the Q Learning algorithm is not useful. In this case, it would consist in training a neural network to predict the best value between 2. A simple max can solve the problem.

In terms of implementation, the equation implemented can be described as :

$$\hat{Q}_N(x_k, u_k) = (1-\alpha)\hat{Q}_N + \alpha\left(r + \gamma \underset{u^* \in \{-4,4\}}{\arg\max}\left[\hat{Q}_{N-1}(x_{k+1}, u^*)\right]\right) \tag{4}$$

$$= (1-\alpha)\hat{Q}_N + \alpha y_k \tag{5}$$

where y_k was defined in the section 5.2, in the equation 3. Therefore, the equation (5) can be simplified. This simplification will be used to gain computation time in our implementation.

6.1 Results and comparison

It is possible to compare the results with the previous results. This is done in figure 7.

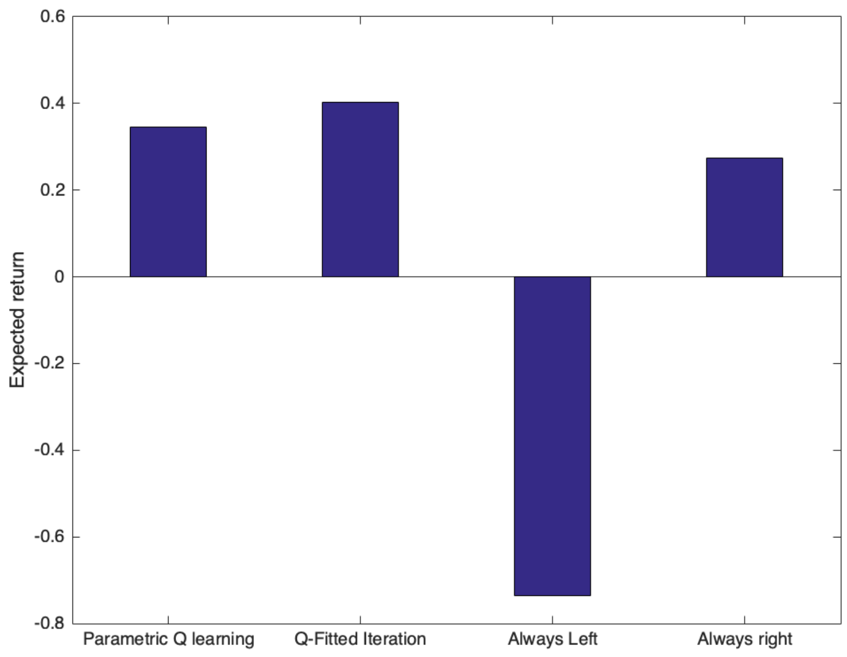
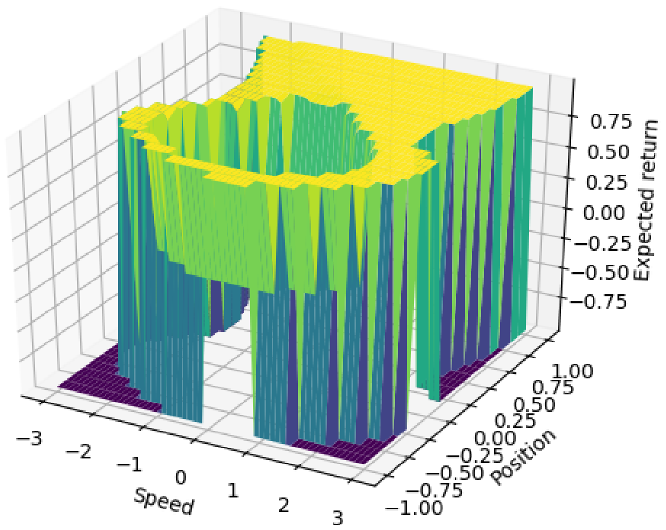


Figure 7: Comparison of the best Q-fitted Iteration and the parametric Q-Learning (100 iterations).

It appears that this algorithm has slightly poorer results than the best Q fitted Algorithm but remains above all others. It is also better than simplistic algorithms of uniform direction. A different approach to compare the results can be simply to use the 3D visual that has been used so far. This is done in figure 8.



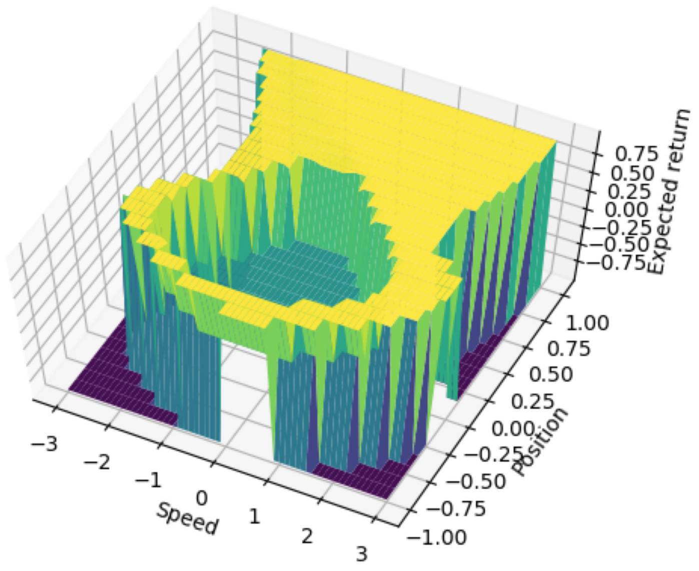


Figure 8: $J^{\mu}(x) \forall x \in X$ when $\mu(x)$ is determined using Parametric Q Learning algorithm.

In terms of computation time, the Parametric Q Learning algorithm is faster then the ones implemented during the 5th question.

Once again, the video of the car for a specific starting position can be obtain by uncommenting the visualisation line in the `run.py` file. The starting position can be set at this same place.