Master in applied sciences
Computer science and engineering
Year 2018-2019

Introduction to Complex problem solving

Assignement 2 (part 2)

INFO8003-1
Damien Ernst

Olivier Moitroux
s152122

Liège

Février 2019

5 Fitted-Q Iteration

Organisation of the code

Several files are included inside the submission. `main.py` is the main script that defines the method associated to each question of the assignment. `toolbox.py` is where the logic is coded. It defines a class `Domain`, a class `Simulator` to test policies, build random episodes, ... as well as some useful methods (extracting positions from histories h, ...) by interacting with the *domain* instance. All the unit tests are available in a dedicated file, namely `unit_test.py`. Finally, `plot.py` contains the code used to build plots.

The main script calls the method `Q5_train()` to train the 3 models asked for the assignment on the same random history. Most of the data or variables that require a lot of computing have been saved via the *pickle* package or directly using *Keras*. The evaluated time to run the complete script (training, videos and plots included) for question 5, is estimated to be a bit more than 12 minutes.

5.1 Generating the dataset

All the trajectories generated starts in $[\mathrm{p}_0, \mathrm{s}_0]$ and applies a random policy. The dataset has been built from 1000 random histories which yield a total of 63929 tuples. Among them, only 18 trajectories leaded to a win. 18 success is enough for the fitted Q algorithm to converge as proved in section 3. Another alternative would have been to use a random policy as well but, this time, starting in a random valid state (random position and speed). It can be easily done by not providing an `initState` to the method `buildEpisodes` so that they are chosen randomly. The benefits of such technique have already been explained and demonstrated in the last report.

5.2 The estimators

For this assignement, 3 estimators have been considered:

- Linear regression from *Scikit-learn* with default parameters
- Extremely randomized trees from *Scikit-learn* with default parameters and $n = 100$ trees.
- Neural network by using *Keras*
  - 2 hidden layers of 100 nodes with *Relu* activation
  - 1 final layers with *Linear* activation
  - Loss function is *mean square error*
  - Optimizer is *Adam*

The structure of the neural network is chosen simple as longer running time are expected when increasing the number of layers or nodes in the hidden layers. Increasing the number of nodes or number of layers would allow the neural network to better fit more complex (non-linear) relationship in the data. By looking at the result given by the extra trees or the linear regression, one can indeed notice that an non-linear estimator would give much better results. Making the neural network more complex is quite easy in *Keras*. Two hidden layers with 4 nodes weren't satifying. Increasing the number of nodes to 100 in each of the two layers gave good results.

Table 1 summarises the relevant pieces of information for each training.

Estimator	**Number of fitted-Q iterations**	**Epochs**
Linear regression	150	/
Extremely randomized trees	100	/
Neural network	70	10

Table 1 – Training parameters for fitted q learning

The number of iterations for the fitted-Q algorithm is different for each estimators due to historical reasons. In order to stop the fitted Q algorithm, it was planned to use the square error between two computation (see dedicated subsection) to determine an acceptable threshold. However, it turned out not to be working. For this reason, the estimators have been trained with a given number of iterations. 50 iterations turned out to be enough for each of them.

The extremely randomized tree takes quite some time to fit the dataset due to the 100 asked iterations. It can be noted that random forest gave the same results with no significant performance loss.

5.3 Estimated cumulative rewards

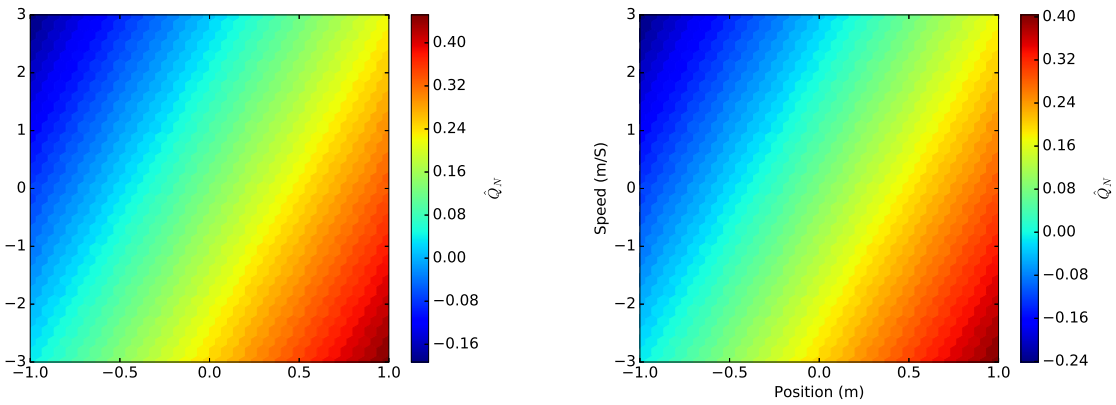
Figure [1-3] display the expected cumulative rewards for each (discretized) state. The linear regression perfoms not well at all as the relation between the states and the cumulative rewards turns out not to be linear at all. The highest cumulative reward is at leat expected near the position 1.0. By generating the video, we observe that the car is always accelerating without succeeding to climb the hill. This indeed makes sense with figure 1. Quite naturally, it will never reach a winning state as its target is wrong: the highest expected reward is the one found for a negative speed.

Extremely randomized trees perform clearly the best with an expected cumulative reward that seems quite logical. Indeed, the highest cumulative rewards are located where the speed is high and the position close to 1. The concave area of the hill is where the cumulative reward is the lowest, even if the speed is moderate.

The neural networks with 2 hidden layers of 4 nodes seems to start a little bit to fit correctly the data in the bottom left (at least the shape in the heat map, so the relative difference between rewards of neighbour states). By increase the number of nodes to 100, we obtain figure 3 which turns out pretty good. A kind of spiral shape can ge guessed like with extra trees as well.

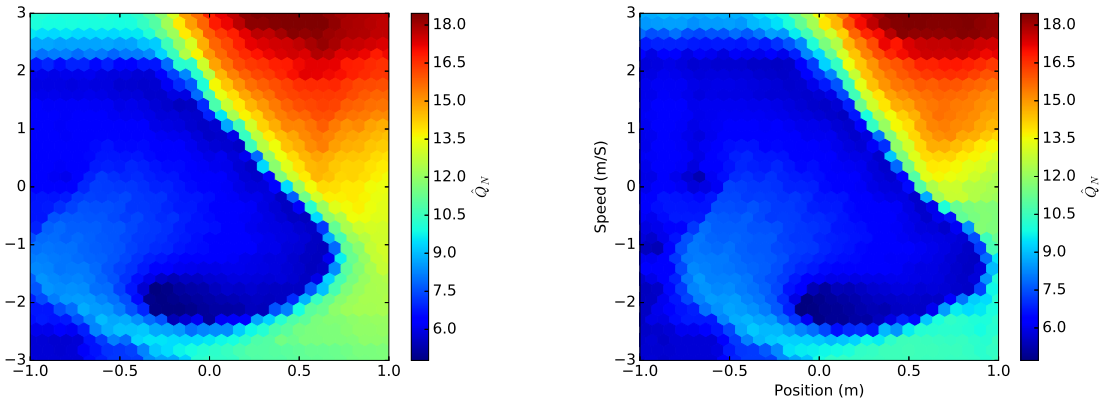
The heat maps that represent the value of $\hat{Q}_N$ for $N = [1, 5, 10, 20, 50]$ have been generated for each of the two actions. However, they won't be displayed in the report due to the number of high number plots. 20 iterations is enough for both the estimation of the linear regressor and the extremely randomized trees to stabilize. However, the heatmaps keeps changing in the neural network (with 2 hidden layers of 4 nodes) even after 20 iterations which highlights a slower convergence (confirmed on figure 4(d)). This is however not the case when the number of nodes in the hidden layers are set to 100. The visual difference between $\hat{Q}_{50}$ and $\hat{Q}_{70}$ is hard to tell though.

Finally it can be noted the expected cumulative reward of the correspondent states computed for different estimators doesn't have the same magnitude.



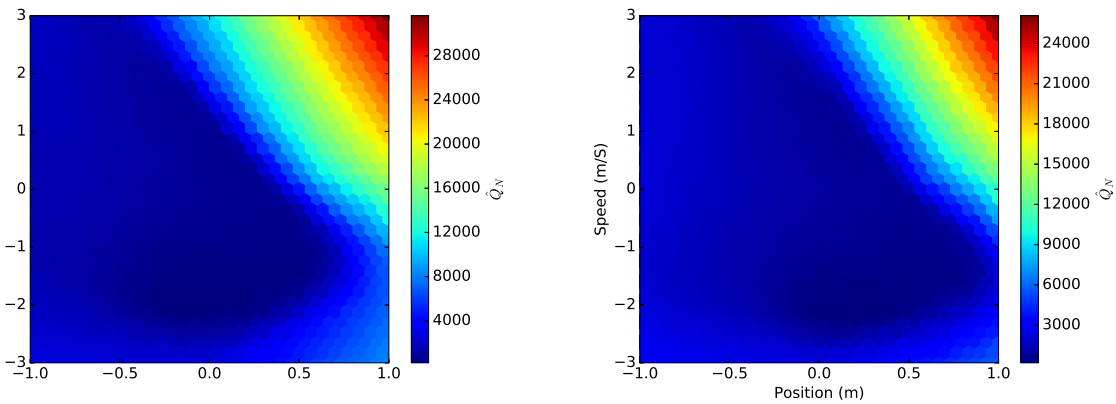
(a) $\hat{Q}_{50}$, u = acc (b) $\hat{Q}_{50}$, u = dec

Figure 1 – Linear regression



(a) $\hat{Q}_{50}$, u = acc (b) $\hat{Q}_{50}$, u = dec

Figure 2 – Extremely randomized trees



(a) $\hat{Q}_{50}$, u = acc (b) $\hat{Q}_{50}$, u = dec

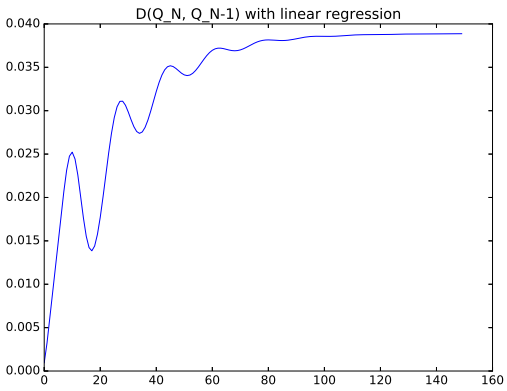
Figure 3 – Neural network

5.4 Conversnce of the fitted Q algorithm

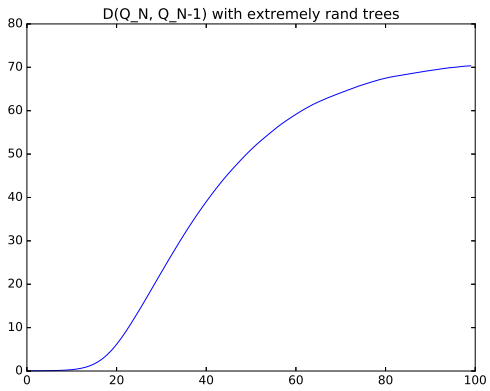
Initially, it was planned to use the distance between two consecutive estimation to stop the fitted Q algorithm:

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

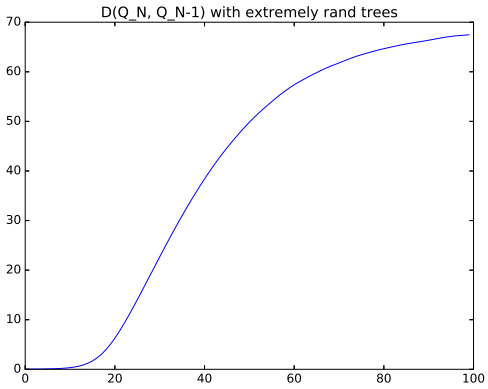
This distance was expected to shrink close to zero after enough iterations. However, as the number of iterations increases, the range of the values of the expected cumulative reward keeps getting larger. In other words the minimum expected cumulative reward decreases and the maximum increases still a bit from iteration 50 to 70 for example, even if no real visual difference on the heatmaps can be observed. This way, the distance keeps growing even if the relative difference between each states stays approximately the same (thus a heatmap with same shapes and colours). To resolve this problem, the code should perhaps normalise the predictions of the algorithm before computing the distance.



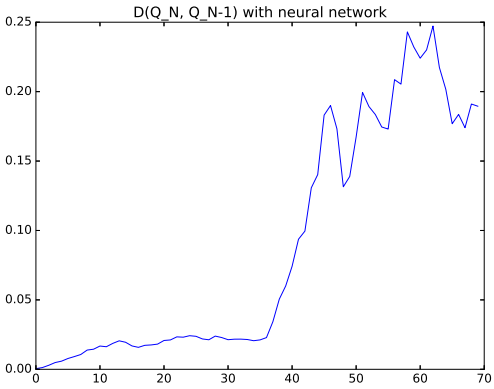
(a) Linear regression



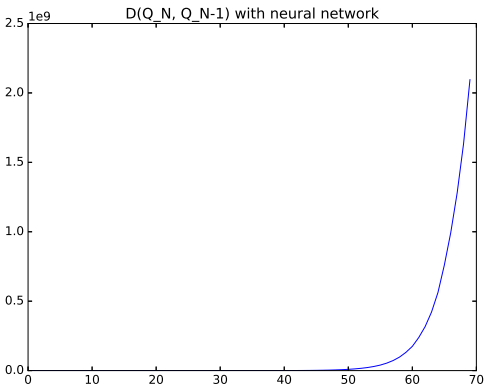
(b) Extremely randomized trees



(c) Random forest [1]



(d) Neural network (2x4)



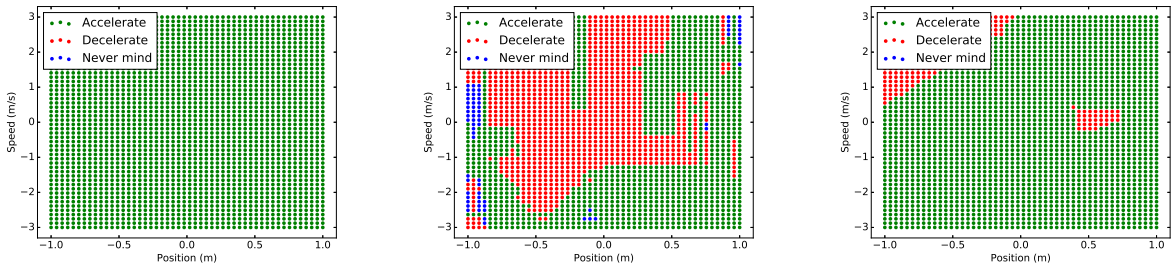
(e) Neural network (2x100)

Figure 4 – Expected cumulative rewards

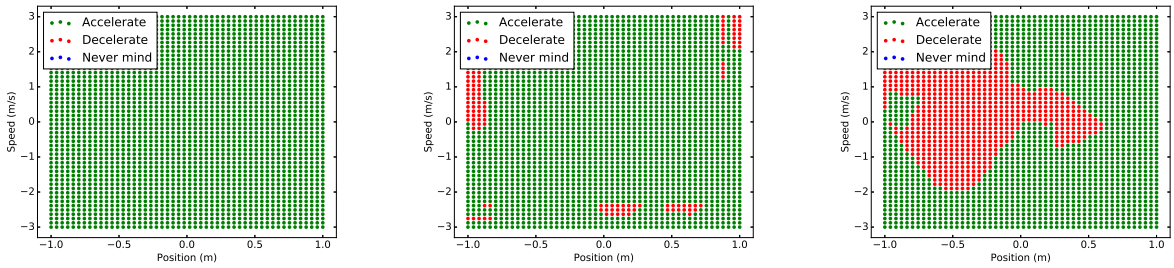
The distance between two consecutive predictions with the linear regression seems to drop each 20 iterations an stabilise around a 100. On the other hands, trees require a bit more iterations but the converge is more predictable (smooth and regular evolution). Finally, the simple neural network is slow to converge up to 40 iterations but then, the distance increases suddenly. The new plateau seems to be reached after 70 iterations. By increasing the complexity of the neural network, one obtains figure 4(e) which does not seem to converge to a fixed value at all. Despite this weird behaviour, the optimal trajectory is still observed by using this estimator starting in the same initial condition as is proved by figure 6(b).

5.5 Policies

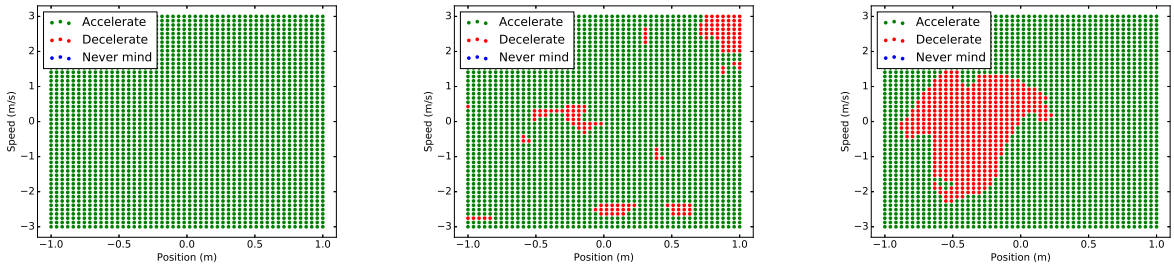
Figure 5 highlights the evolutions of the estimated optimal policies accross iterations in the fitted Q algorithm for each of 3 estimators. Like stated above, the linear regression leads to a policy that always accelerate. The extra trees give the same results as in the paper and the video confirms the correct behaviour of the car. Some shapes in the plot of the policy for the extremely randomized trees and the neural network (2x100) after 20 iterations look a bit the same. While the heatmaps that represent the expected cumulative reward by using the NN estimator seemed to stay almost the same after 20 iterations, it is not the case for the estimation of the optimal policy. Indeed, even letting the neural network run for 20 more fitted Q iterations revealed another plot a bit different of Figure 5(o). This can be put in parallel of figure 4(e) where no plateau is distinguishable in the distance between two Q's.



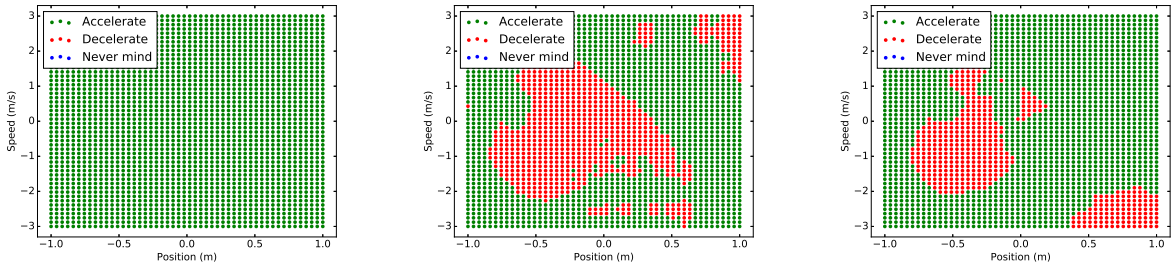
(a) Linear regression : $\hat{\mu}_1$ (b) Extr. rand. trees : $\hat{\mu}_1$ (c) Neural network : $\hat{\mu}_1$



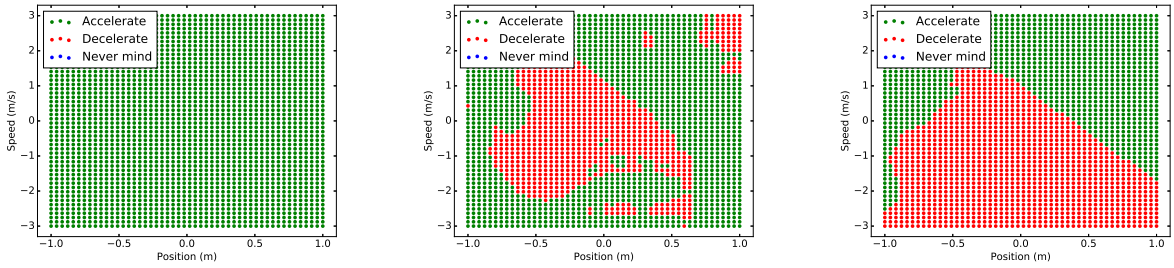
(d) Linear regression : $\hat{\mu}_5$ (e) Extr. rand. trees : $\hat{\mu}_5$ (f) Neural network : $\hat{\mu}_5$



(g) Linear regression : $\hat{\mu}_{10}$ (h) Extr. rand. trees : $\hat{\mu}_{10}$ (i) Neural network : $\hat{\mu}_{10}$



(j) Linear regression : $\hat{\mu}_{20}$ (k) Extr. rand. trees : $\hat{\mu}_{20}$ (l) Neural network : $\hat{\mu}_{20}$

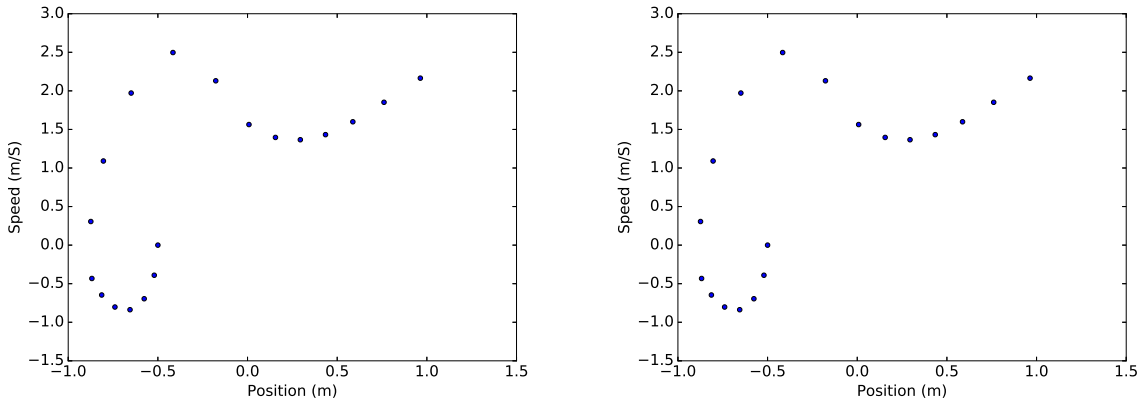


(m) Linear regression : $\hat{\mu}_{50}$ (n) Extr. rand. trees : $\hat{\mu}_{50}$ (o) Neural network : $\hat{\mu}_{50}$

Figure 5 – Estimation of the optimal policy depending on the number N of iterations of the fitted Q algorithm for different estimators.

5.6 Optimal trajectory with fitted Q algorithm

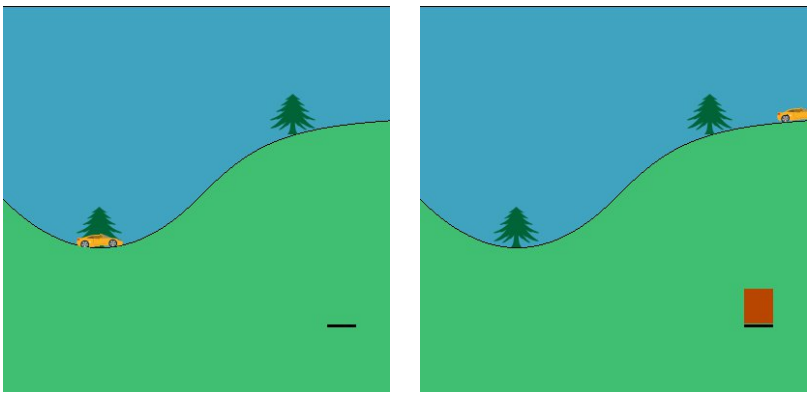
Figure 6 is generated by applying the policy computed with the fitted Q algorithm and the extremely randomized trees after 100 iterations. 18 Steps are necessary to reach the top of the hill. This leads to a cumulative reward of 0.3972.



(a) Extremely randomized trees (b) Neural network

Figure 6 – Estimated optimal trajectories by estimators.

5.7 Video with extremely randomized trees



(a) $t_{step} = 0$ (b) $t_{step} = 18$

Figure 7 – The video, composed of 18 images, that confirms the car has the right expected behaviour.

6 Parametric Q-Learning

- Save target network parameters $\phi' \leftarrow \phi$
  - Collect M datapoints $\{s_i, a_I, s'_i, r_i\}$ by applying a random policy. We add them to the dataset
    - Sample a batch of tuples from the dataset
    - $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma max_{a'} Q_{\phi'}(s'_i, a'_i)])$
    - Do that K times

- Do that N times

This way, the idea is to generate 1000 random trajectories. Then, we select a group of tuples to use them directly to update our model by training it with these tuples. The model considered is a neural network that uses backpropagation with gradient descent to compute the derivative (with respect to the parameter ϕ) automatically. The learning rate, α, should be quite small.

The tricky part is to find the right loss function that will make the training of the neural network equivalent to $(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma max_{a'} Q_{\phi'}(s'_i, a'_i)])$. An absolute error does not seem to be enough. Rather, one can build its own objective function in keras so that it implements $\delta(x_t, a_i) = Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma max_{a'} Q_{\phi'}(s'_i, a'_i)])$, in other words, the difference between the old estimation and the sum of the direct reward and the largest next estimation w.r.t. the action. Once a certain number of epochs have been performed, one can then resample a new batch to train on. The code below shows an example of custom objective function. To use it, one juste need to call `model.compile(loss=custom_objective, optimizer='adadelta')` for example.

Due to time reason, I want have the time to experiment with this idea though. Extracting and displaying the policy is done the same way as in the previous section. Comparing FQI and parametric Q-learning in the completely same settings is not possible due to the randomness. However, by making enough iterations, we can expect to find similar but less acurate results with parametric Q-learning. The comparison can be done by plotting the square error on the prediction of the expected cumulative reward after the same number of steps or by comparing manually the plots displayed in the previous section.

Example of code to build custom loss function in Keras

```
import theano
import theano.tensor as T

epsilon = 1.0e-9
def custom_objective(y_true, y_pred):
    '''Just another crossentropy'''
    y_pred = T.clip(y_pred, epsilon, 1.0 - epsilon)
    y_pred /= y_pred.sum(axis=-1, keepdims=True)
    cce = T.nnet.categorical_crossentropy(y_pred, y_true)
    return cce
```