Université de Liège
Master 1 - Data Science and Engineering
Année 2020-2021

Optimal decision making for complex problems

Assignment 2 - Part 2

INFO8003-1 Optimal decision making for complex problems
ERNST Damien
AITTAHAR Samy
BARDHYL Miftari

PIRLET Matthias
s172330
CHRISTIAENS Nicolas
s171216

Liège
Tuesday, 3 March 2021

1 Parametric Q-Learning

Parametric Q-learning is a algorithm which updates its parameters θ after each sample of type:

$$(x_t, u_t, r_t, x_{t+1})$$

If the parametric Q-function is of the form $\hat{Q}_\theta(x, u)$ and q_j^* is a observation of the optimal state-action value function Q^*, the algorithm will aim to minimize the empirical cost function:

$$\sum_j (q_j^* - \hat{Q}_\theta(x_j, u_j))^2$$

Using stochastic gradient descent, we obtain :

$$\theta_i = \theta_{i-1} - \frac{\alpha}{2} * \nabla_{\theta_{i-1}} (q_i^* - \hat{Q}_\theta(x_i, u_i))^2 = \theta_{i-1} + \alpha * (\nabla_{\theta_{i-1}} \hat{Q}_\theta(x_i, u_i)) * (q_i^* - \hat{Q}_{\theta_{i-1}}(x_i, u_i))$$

Which can be transformed into :

$$\theta_i = \theta_{i-1} + \alpha * (\nabla_{\theta_{i-1}} \hat{Q}_\theta(x_i, u_i)) * (r_i + \gamma * max_{u \in U} \hat{Q}_{\theta_{i-1}}(x_{i+1}, u) - \hat{Q}_{\theta_{i-1}}(x_i, u_i))$$

where

$$\delta_i = r_i + \gamma * max_{u \in U} \hat{Q}_{\theta_{i-1}}(x_{i+1}, u) - \hat{Q}_{\theta_{i-1}}(x_i, u_i)$$

is called the temporal difference.

We use a neural network as the approximation architecture.
We implemented the algorithm in batch mode (because the statement does not specify something and the bonus section is the online one) with samples done in the same way as the previous section (no greedy policy or other, just random actions). And we have not implemented the replay experiences. We made these choices because the main purpose of this section is to compare FQI and DQN. So we weren't going to add a technique (like replays experiments) that could have increased the efficiency of the algorithm and therefore biased our comparison.

The architecture of our neural network must first have two important properties :

- Input layer : the dimension of this layer must correspond to the state, i.e. p and s. We must therefore have a dimension of 2 for input.
- Output layer : the dimension must correspond to the number of possible actions, i.e. accelerated or decelerated (4 or -4). So we must have a dimension of 2 for output.

This means that for a input state, we will have at output the value of Q-function for each possible action.
For the hidden layers we did some research and most of the examples we found only took 2 layers of a hundred neurons with "relu" activation because the experiments were not very complex. Here too we have a not too complex experience and therefore we have chosen to keep this type of architecture.
To be in agreement with the equations written above, we have taken as optimizer "sgd" (stochastic gradient descent) and as loss function "mse" (mean squared error).
We use "Keras" and by going to see the documentation and some articles, we noticed that the back-propagation was implemented by default and therefore that we did not have to extract the gradient to update the parameters θ. Simply using "fit" at each iteration is sufficient.

The policy $\hat{\mu}^*$ can be easily derived from $\hat{Q}_\theta$:

$$\hat{\mu}^*(x) \in argmax_{u \in U} \hat{Q}_\theta(x, u)$$

Which can be represented with these grids :

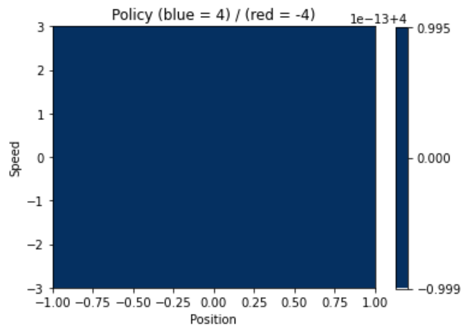

Figure 1: Policy $\hat{\mu}^*$ with $\alpha = 0.05$ out of 10 random simulations

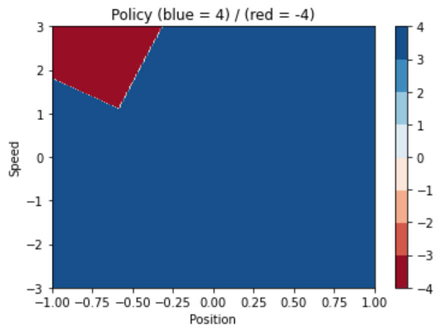

Figure 2: Policy $\hat{\mu}^*$ with $\alpha = 0.05$ out of 100 random simulations

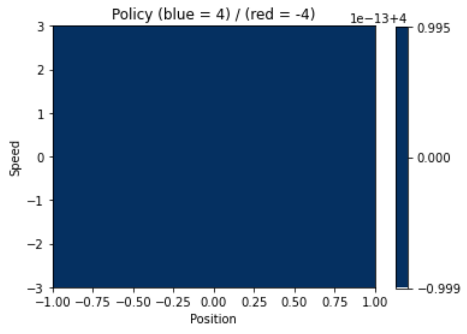

Figure 3: Policy $\hat{\mu}^*$ with $\alpha = 0.2$ out of 10 random simulations

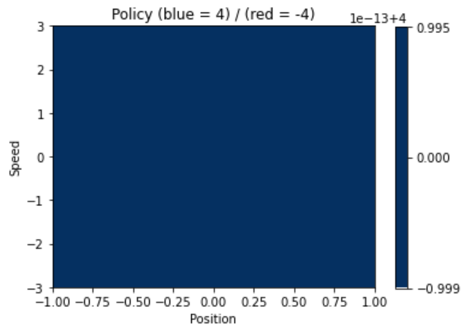

Figure 4: Policy $\hat{\mu}^*$ with $\alpha = 0.2$ out of 100 random simulations

As we can see on these graphs, it seems that the algorithm does not manage to pass the hill and is therefore satisfied (most of the time) to accelerate to the right so as not to have a negative reward (that of left). We will therefore have an expected return tending to 0 because we remain stuck indefinitely with this kind of policy (without a strategy to go up the hill).
As expected, when we calculate the expected return of these policies we get a value that tends to 0.
N.B: The policies shown above are not stable in the sense that they often change for each execution (this is obviously due to the fact that we are working with random simulations) but it seems clear that they tend to describe an acceleration at right on the whole domain (We have this policy more and more frequently the more we increase the simulations).

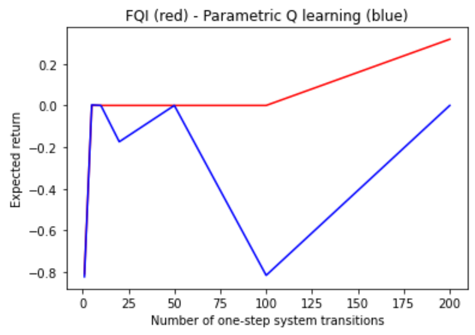

Figure 5: Comparison between FQI and parametric Q-learning with $\alpha = 0.05$

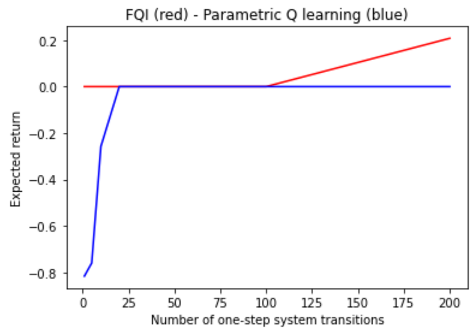

Figure 6: Comparison between FQI and parametric Q-learning with $\alpha = 0.2$

We can observe that unlike FQI, the parametric Q algorithm does not manage to pass the column (for all alpha values) and so it can't have a expected return greater than 0. Therefore we can say without too much doubt that FQI is much better . The FQI with linear regression arrives at the same result.
At this stage, there are 2 possible explanations for these results. Either the "improved" version of the parametric Q learning algorithm (the one built for the bonus) will obtain better results because it will solve the correlation and target problems. Either this type of algorithm is simply not adapted to this type of problem and even with the most optimal parameters, no strategy will allow to pass this hill and therefore, no improvement is possible.

2 Normalised parametric Q-learning

We have to build 2 algorithms : the online Q-iteration algorithm and the online Q-iteration algorithm with normalised update term of the Q-function.
These 2 algorithms have a very similar structure (as seen in lecture):

- Replay buffer to avoid correlated samples
- Target network (with parameters ϕ') to prevent the target value from changing at each step
- Personal policy to create one step transitions (we chose the easiest : ϵ-greedy policy)

The only difference is in the calculation of the parameters. With the basic algorithm we will have :

$$\phi \leftarrow \phi - \alpha * \sum_i \frac{\partial Q_\phi}{\partial \phi}(x_i, u_i) * (Q_\phi(x_i, u_i) - [r(x_i, u_i) + \gamma max_{u' \in U} Q_{\phi'}(x_{i+1}, u')])$$

while for the normalized algorithm we will have :

$$\phi \leftarrow \phi - \alpha * \sum_i \frac{\partial Q_\phi}{\partial \phi}(x_i, u_i) * \frac{1}{||\frac{\partial Q_\phi}{\partial \phi}(x_i, u_i)||} * (Q_\phi(x_i, u_i) - [r(x_i, u_i) + \gamma max_{u' \in U} Q_{\phi'}(x_{i+1}, u')])$$

So the norm we use is just the norm of the gradient which is a vector ($||g||$ is thus a scalar) : $\frac{g}{||g||}$

We implemented the full version but not the normalized one so we didn't make any practical comparison.

3 References

[1] Matthieu Geist,Olivier Pietquin : "A Brief Survey of Parametric Value Function Approximation", 2010