LIÈGE université
Sciences Appliquées

Université de Liège
Faculté des Sciences Appliquées

Project 3: RL in physical systems

INFO8003-1 : Optimal decision making for complex problems

Jérôme Pierre (s190979)
Arthur Vogels (s191381)

May 26, 2023

Single inverted pendulum

Environment

The environment of the inverted pendulum problem consists of a single inverted pendulum on a cart on which a force can be applied.

It is mathematically described as follows:

1. Action set:

$$action = F \in \mathcal{A} = \mathcal{R} \cap [-3, 3]$$

where F stands for the force applied on the cart.

2. State space:

$$state = (x, \theta, v, \omega) \in \mathcal{S} = \mathcal{R}^4$$

where:
x is the position of the cart
θ is the vertical angle of the pole on the cart
v is the velocity of the cart
ω is the angular velocity of the pole on the cart

All the different elements defining the tuple of the initial state are initialized according to a uniform distribution $\mathcal{U}(-0.01, 0.01)$

A terminal state is reached when $|\theta| > 0.2$ (rad). We also consider that when all the elements of the state are infinite, a terminal state is reached. The generation of a trajectory is also stopped when the time limit (1000 timesteps) is reached.

3. Reward:

$+1$ is awarded for each timestep that the pole is upright.

In other terms: $+1$ for each timestep when a terminal state is not reached.

4. Dynamics:

For further information on the dynamics, please refer to the gym documentation.

5. Discount factor

The larger the discount factor γ, the more attention is put on long-term reward. Since, in this case, we want to keep the pendulum up as long as possible, it made sens for us to restrict the γ to values that are close to 1.

Fitted-Q iteration

The first algorithm we tried in order to solve the task is Fitted-Q iterations. This algorithm consists in iteratively creating learning sets LS_i and test sets TS_i in order to use supervised learning technique to predict iteratively the Q functions $Q_1, Q_2, ..., Q_N$. In this project, we chose to work with extremely randomized tree.

Let $\mathcal{S}$ be a set of one-step transitions.

$$\mathcal{S} = \{\Big((x_t, \theta_t, v_t, \omega_t), u_t, r_t, (x_t^{'}, \theta_t^{'}, v_t^{'}, \omega_t^{'})\Big)\}$$

Sampling this set allows to construct the following learning set:

$$\text{Inputs} = \mathcal{I} = \{\mathrm{i_k}\}_{\mathrm{k=1}}^{\mathrm{K}} = \{(\mathrm{x_k}, \theta_\mathrm{k}, \mathrm{v_k}, \omega_\mathrm{k}, \mathrm{u_k})\}_{\mathrm{k=1}}^{\mathrm{K}}$$

In order to get $\hat{Q}_1$, we can construct the following first outputs set:

$$\text{Outputs}_1 = \mathcal{O}_1 = \{\mathrm{o_k}\}_{\mathrm{k=1}}^{\mathrm{K}} = \{(\mathrm{r_k})\}_{\mathrm{k=1}}^{\mathrm{K}}$$

Then, we have to update the output set by using the previous predictor to predict the next Q functions $\hat{Q}_2, \hat{Q}_3, ...$

$$\text{Outputs}_\mathrm{N} = \mathcal{O}_\mathrm{N} = \{\mathrm{o_k}\}_{\mathrm{k=1}}^{\mathrm{K}} = \{(\mathrm{r_k} + \gamma \max_{\mathrm{u}^{'}} \hat{\mathrm{Q}}_{\mathrm{N-1}}(\mathrm{x_k^{'}}, \theta_\mathrm{k}^{'}, \mathrm{v_k^{'}}, \omega_\mathrm{k}^{'}, \mathrm{u}^{'}))\}_{\mathrm{k=1}}^{\mathrm{K}}$$

Generating $\mathcal{S}$

It is primordial to realize the importance of the quality of the data for this algorithm. Indeed, if only show trajectories with poor results, the algorithm will learn what it shouldn't do and not what it should do. Actually, this method would work but would require us to generate an extremely large On the other hand the contrary is also possible. It is why we put a particular emphasis on the way our data was generated.

We decided to generate $\mathcal{S}$ by using an epsilon-greedy policy. More precisely, we will generate multiple generation of trajectories. At first, 500 unbounded trajectories are generated using an agent that uniformly chooses his action in $[-3, 3]$. For the following iterations, we will use the estiamtor that has been obtained right before and iteratively generate 500 trajectories by using an epsilon-greedy policy. In other words, we sample a random number from a $\mathcal{U}(0, 1)$ and if this number is smaller than a threshold which is 0.9, the action is the same as the one predicted by the model. In the other case, we add some Gaussian noise to the predicted action $(0.5\,\mathcal{N}(0, 1))$.

The main idea of our implementation is to obtain a new dataset $(Inputs, Outputs_N)$ from scratch and concatenate it with the ones obtained at previous iterations. Then, we retrain another model on the concatenated datasets in order to generate the following trajectories. It allows to generate longer trajectories right after the first iteration. This technique should allow to get a great dataset on which the final model will be trained.

Figure 1 shows the evolution of the reward through the simulations. 500 trajectories have been generated to obtain those values. Let's point out that the results that are

obtained have a high variance and depend a lot on the state transitions present in the dataset. When we ran our algorithm multiple times, the cumulative return would change drastically from one run to another. It is probably the reason why running our code another won't give results as good as the ones from the figure **??**

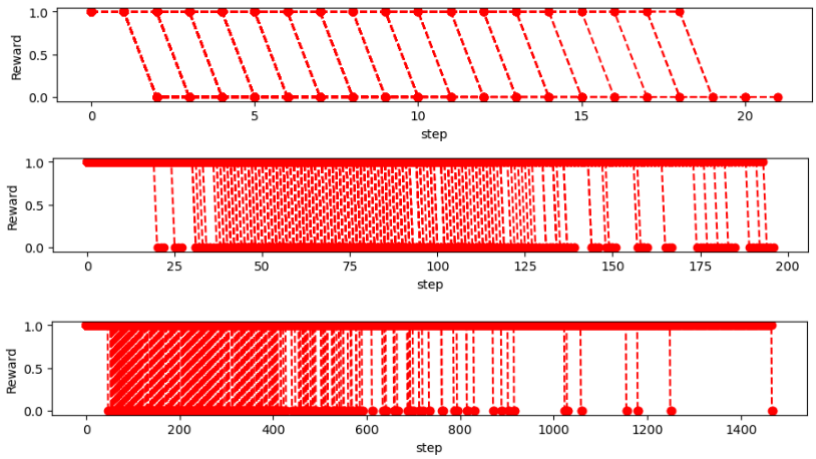

Figure 1: Evolution of the reward in the simulations. We can observe that doing multiple generation steps increases the length of the trajectories that are obtained.

An estimate of the cumulative expected return of a typical estimator that can be obtained with this method was computed with Monte-Carlo estimation. For FQI, 100 trajectories were used.

Similar measures are performed in the other cases. In the case of FQI with extremely randomized tree for the inverted pendulum, the mean J is of around 211.36 ± 149.4. Such a large variance shows that our estimator can sometimes lead to very long simulation and sometimes to very short ones. The best estimator we obtained beat largely beat this model and lead most of the time to trajectories with length bigger than 400 but we deem to have been excessively lucky during its generation.

Let's note that at some point, the size of the concatenated dataset should be too large for efficient train the estimator or even to be in memory. In our case, we do not have this issue since we only perform three generations. However, we think it is possible to use some measures in order to give each tuples a score representing how important it is in the dataset and then try having the best dataset possible. Another possibility would be to replace old tuples by new ones after a few generations.

Stopping rule

It is possible to bound the value at which N should be set by using the upper-bound on the suboptimality of a set of optimal policies μ_N^* given by :

$$\epsilon = ||J_\infty^{\mu_N^*} - J_\infty^{\mu^*}||_\infty \leq 2\frac{\gamma^N Br}{(1-\gamma)^2}$$
$$\iff N \geq log_\gamma \left((1-\gamma)^2 \frac{\epsilon}{2Br}\right)$$

Br represents the largest possible reward which is 1 in the case of the single inverted pendulum. We chose to have $\epsilon = 0.1$.

In this part of the project, we have chosen $\gamma = 0.99$ in order to still take into account long-term rewards. However, using such a value of γ leads to $N = 1215$ which is not feasible considering it would require hours. Instead, we have chosen to use $N = 180$ which corresponds to a γ of 0.95

Reinforce

From now on, we have to work with policy-gradient methods. The first method that will be explored is the REINFORCE algorithm.

For this method, we decided to define the policy trough a normal distribution where the mean and the standard deviation are found using a neural network. In order to respect the different domain restrictions, we have used the following representation:

$$\pi(u|x,\theta,v,\omega) \sim \mathcal{N}(\mu_\phi, \sigma_\phi); \qquad \mu_\phi = 3\ tanh(NN_1(x,\theta,v,\omega)); \qquad \sigma_\phi = exp(NN_2(x,\theta,v,\omega))$$

where NN_1 and NN_2 are respectively the first and the second output of the neural network.

Using three times the hyperbolic tangent allows to brings the mean in the interval [-3, 3].

Algorithm 1 REINFORCE algorithm:

Initialize θ randomly
Initialize the step size α
for $i \leftarrow 1, ..., \text{NbEpisodes}$ **do**
  Generate a trajectory $(s_0, a_0, r_1, ..., r_T)$ according to the current policy
  for each one-step transition $t \in \{0, 1, ..., T-1\}$ **do**
    $G = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\theta = \theta + \alpha\ \gamma^t\ G \nabla ln(\pi(a_t|s_t,\theta))$
  end for
end for
Return θ

In this project, we have used a unique neural network that has two outputs, the mean μ_θ of the action and the log standard deviation $log\,\sigma_\theta$. We choose to output the log of the standard deviation so that it can be negative or positive. The said neural network consists in a MLP with two hidden layer of size 64, the input dimensions being being given by the dimensions of the observation space (4 or 11).

The goal of the reinforce algorithm is to reinforce the good actions. In order to do that, we try to minimize the following loss using gradient descent with the Adam optimizer with a learning rate of $1e-3$.

$$U(t) = -\gamma^t \sum_{\tau, t'>t} P(\tau;\phi)\ G(\tau)$$
$$\phi \leftarrow \phi - \alpha \nabla\ U$$

where P denotes the probabilities of the different steps that are met during the trajectory and G is the cumulative return that has been experienced.

Results

We trained our algorithm during a few hours and this yielded the expected results that are displayed in Figure 2. As can be seen, the results are a bit disappointing. We can indeed see that the algorithm learns since we went from 0 to almost 200 after the 100000 iterations.

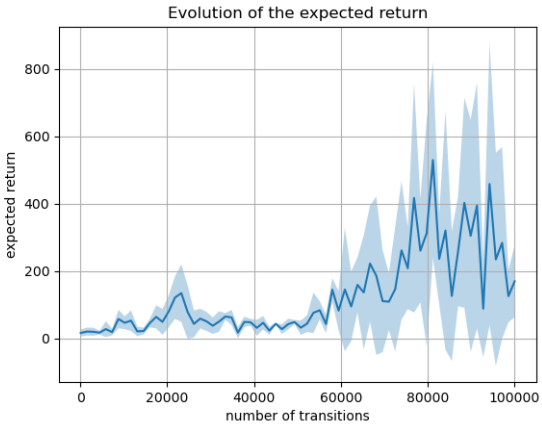

Figure 2: This graph shows the evolution of the expected result through the transitions steps.

We can see that the variance of the results is quite high, with some trajectories reaching the highest possible value and other not able to reach a hundred steps

Deep Determinisitic Policy Gradient (DDPG)

The last algorithm we implemented is DDPG. We chose to implement this algorithm because it uses a lot of interesting reinforcement learning concepts and gave some great results. It is also well adapted for environments that rely on a continuous action space such as the inverted pendulum environment.

The algorithm is detailed in Figure 3.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 3: DDPG algorithm [**DDPG**]

Neural network

DDPG relies on four different neural networks. Two of them are used for directly predicting the Q function or the policy while the two other are only used as target network. Those networks takes states as input and sometimes, they also require the action.

In our implementation, the critic network $Q(s, a|\theta^Q)$ is a multi-layer perceptron with a hidden layer with 64 neurons. The same architecture has been used for the network outputing the policy $\mu(s|\theta^\mu)$. The only difference is that we added an hyperbolic tangent and mutltiplied by 3 the output in order to force the domain of the output to be $[-3, 3]$.

The two other neural networks $\theta^{Q'}$ and $\theta^{\mu'}$ are used as target networks. It means that they are used in the evaluation of the loss of the algorithm in order to reduce the bad effects that may occur when we are aiming for a moving target. DDPG tries to optimize the Q network by minimizing the mean squared Bellman error.

$$L(\theta^Q, \mathcal{S}) = \underset{(s,a,r,d,s^{'})}{E}[\Big(Q(s,a|\theta^Q) - \Big(r + \lambda\ (1-d)\ Q(s^{'}, \mu(s^{'}|\theta^{\mu^{'}})\ |\theta^{Q^{'}})\Big)\Big)^2]$$

Where the $(s, a, r, d, s^{'})$ represents the one-step transitions. We have already explained that s was the current state, $s^{'}$ the next state, a the action and r the reward. On top of that, we now introduce d which is a boolean value encoding whether or not the generation is over because a terminal state has been reached.

DDPG also uses another loss on top of that. Indeed, once we perform an optimization step on the θ^Q, we try to optimize the θ^μ by trying to optimize the expected return J:

$$\theta^\mu = \underset{s}{argmax}[Q(s, \mu(s|\theta^\mu)|\theta^Q)]$$

Those two neural networks' parameters are optimized using the ADAM optimize with a learning rate of 0.001.

It is important to highlight the use of the target networks in the loss of θ^Q. Those are used because the target $r + \lambda\ (1-d)\ Q(s^{'}, \mu(s^{'}|\theta^{\mu^{'}})\ |\theta^{Q^{'}})$ would largely change thought the different learning steps. It has been shown that using target networks was helpful on the matter.

The weights of those networks are not optimized classically though. Indeed, those are used by combining their weights with the current ones using Polyak averaging. In our code, we used $\tau = 0.995$

$$\theta^{\mu^{'}} \leftarrow (1-\tau)\theta^\mu + \tau\theta^{\mu^{'}}$$
$$\theta^{Q^{'}} \leftarrow (1-\tau)\theta^Q + \tau\theta^{Q^{'}}$$

Data management

The data that is used for training is generated through simulations between each optimization step. It induces that the different obtained data points are very correlated with each other. In order to reduce this effect, DDPG relies on a replay buffer in order to stack the different one-step transitions and sample a batch of data randomly.

This trick allows to mitigate correlation issues.

Results

This technique yielded some great results. As can be seen in Figure 4, DDPG actually beat all of the other methods and allowed to reach a reward that wouldn't stop increasing. We even had to limit the number of steps otherwise, it would have kept going for a while.

Moreover, it is also the mehtod that requires the less computational time for training. The following graph has been obtained in only a couple of minutes.

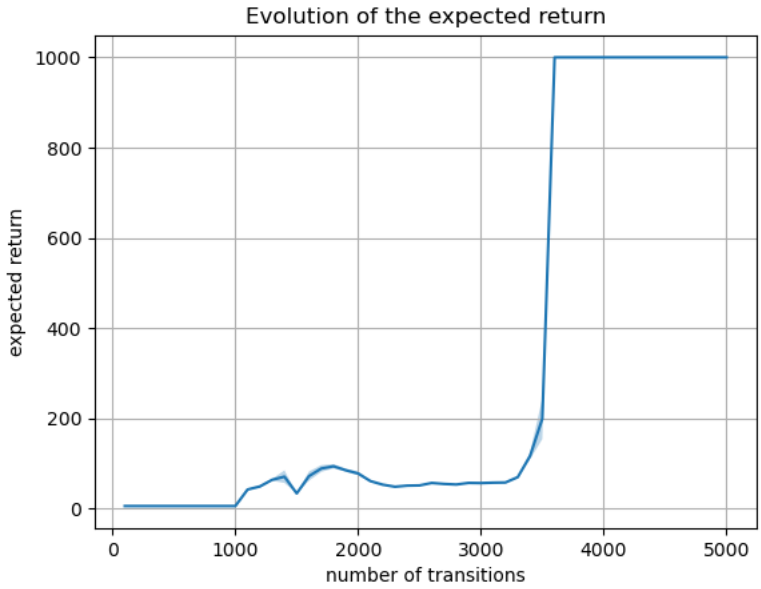

Figure 4: Lenght of the trajectories through the learining steps. We can observe that DDPG achieves a larger reward than other methods.

Double inverted pendulum

In the second part of this project, we used the same algorithms as in the first part but with the double inverted pendulum. Let's introduce the environment:

1. Action set:

$$action = F \in \mathcal{A} = \mathcal{R} \cap [-1, 1]$$

where F stands for the force applied on the cart.

2. State space:

$$state = (x, v, \theta, \dot{\theta}, \phi, \dot{\phi}) \in \mathcal{R}^6$$

where:
x is the position of the cart
v is the velocity of the cart
θ and ϕ are the angles for two pendulums
$\dot{\theta}$ and $\dot{\phi}$ are the angular velocities of the two pendulums

The gym implementation of the system do not directly observe those variables though. Instead, the following observations are used in order to define the environment.

$$state = (x, sin(\theta), sin(\phi), cos(\theta), cos(\phi), \dot{x}, \dot{\theta}, \dot{\phi}, C_1, C_2, C_3) \in \mathcal{R}^{11}$$

with C_1, C_2 and C_3 being constraint forces.

All the variables of the system are initialized with a uniform distribution $\mathcal{U}(-0.1, 0.1)$.

A terminal state is reached when y_coordinate of the tip of the second pole is less than or equal to 1.

3. Reward:

The reward is composed of three parts:

(a) **Alive bonus** b**:** A reward of $+10$ is attributed at each timesteps when the second pole is upright.

(b) **Distance penalty** d**:** This penalty represents how far the tip of the second pendulum moves. Let x be the x-coordinate of the tip and y be the y-coordinate of the tip of the second pole.

$$d = 0.01\ x^2 + (y - 2)^2$$

(c) **Velocity penalty** v**:** This term allows to penalize the agent when it moves too fast

$$v = 0.001\ v_1^2 + 0.005\ v_2^2$$

The total reward $\mathcal{D}$ is obtained through a combination of the previous terms:

$$\mathcal{D} = b - d - v$$

4. Dynamics:

   The dynamics is obtained through physical simulations. For information on the dynamics, please refer to the gym documentation.

5. Discount factor:

   The discount factor represents the importance that is associated with long-term rewards. The larger the γ, the more importance further rewards have. In this setting, we decided to work with $\gamma = 0.99$

FQI

As in the simple inverted pendulum, the first method that is used in the double pendulum is fitted-Q iterations with extremely randomized trees. We use the same formula as in the first part.

$$N \geq log_\gamma \left((1-\gamma)^2 \frac{\epsilon}{2Br} \right)$$

In this case, we have $\gamma = 0.99$, $\epsilon = 0.1$ and $Br = 10$. Even though the variables are unbounded and then the value for the maximal reward could be higher than that. In practice, we can see that it is a safe upper bound.

This leads to $N = 1443$ which would takes hours to train. That is why we have decided to keep the same $\gamma = 180$ as in the first part. Let's note that taking this N is better than considering that $\epsilon = 1$ instead of 0.1 and $\gamma = 0.95$ which seems still acceptable hypotheses.

Applying the same algorithm presented in the first part lead us to obtain an estimator that yields[1] an expected reward of 314.53 ± 102.5. Ounce again, the variance is pretty big and shows that our algorithm can achieve a large range of results.

[1] We still consider 100 trajectories in order to get this result

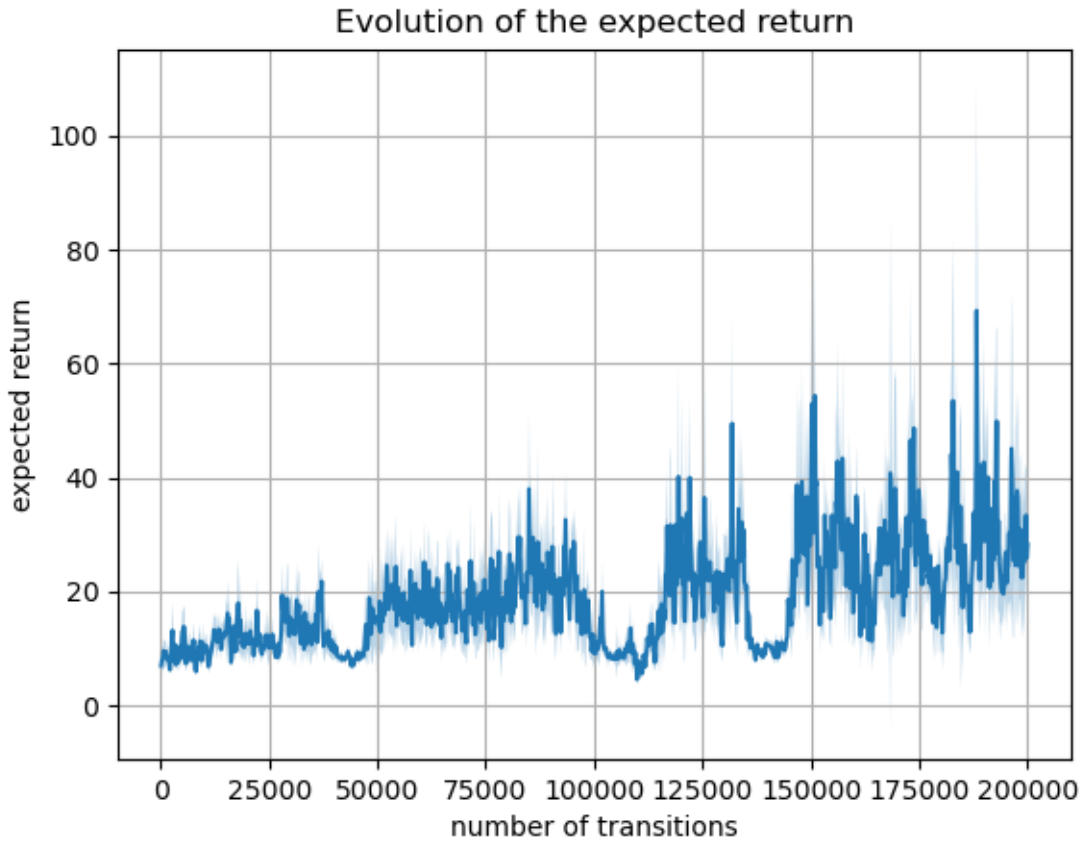

Figure 5: Evolution of the accumulated reward over 200000 state transitions

Reinforce

We took the same implementation as in the first part of the project and applied it to the double inverted pendulum. This yielded the following results:

As we can see, the vanilla REINFORCE algorithm has difficulties to learn a way to balance the double pendulum. After reaching 200 thousand state transitions, it is only able to reach around 30 steps per trajectory.

DDPG

We directly reused the DDPG algorihtm for this part. The neural network and hyper-parameters remain the same, with the only difference that the neural network takes 11 inputs, and that the action outputed by the network is now clipped between -1 and 1. For the double inverted pendulum, it does not seem to easily go forever. The evolution of the expected reward are available in Figure 6

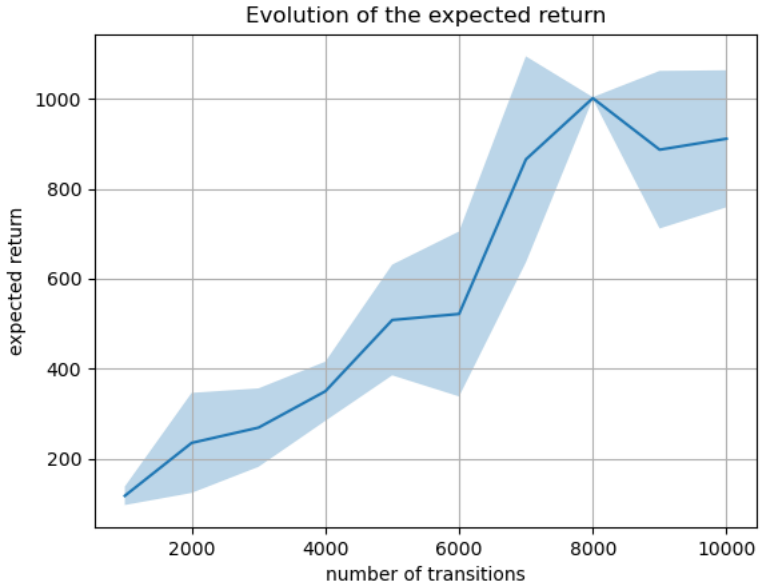

Figure 6: Expected reward as a function of the number of optimization steps

Please note that we also forced the maximal reward to 1000 in this setting in order to force our algorithm to stop. It is probably possible to further increase our performances.

Conclusion

To sum up, we have found that value-based method such as FQI are not as good as advanced policy-based methods such as DDPG. Even though, we haven't performed the iterations until the required N in order to satisfy the theoretical bound, we can still suppose that our FQI wouldn't have reached significantly better results. Our FQI implementation also requires a lot of time. This is partially due to the choice of estimator.

Regarding Reinforce, we observed that the results that were obtained were disappointing. Even though, some learning have been made, the algorithm has not been able to reach great perforamances.

Finally, DDPG has shown to be the best method explored in this project and although it is by far the most complex among them, it requires way less time for learning.

Furhter work

Regarding fitted-Q-iterations, it would be interesting to observe the impact of other estimators than extremely randomized trees.

Further methods could also be used in order to get the best dataset possible. We could for instance use measure of similarities between the trajectories in order to determine which one-steps transitions are really important. Decreasing the necessary size of our learning set could allow to importantly decrease the running time of our algorithm.

It would also be interesting to compare our results with other algorithms such as PPO in order to multiply the number of comparisons that are performed.

What's more, the architectures of the different neural networks used in this project are quite simple. Indeed, we have only used simple multi-layer perceptrons in this project. It would be interesting to try to further optimize the architectures. It could for instance be interesting to try using recurrent neural network or transformers in order to consider sequence of state instead of only focusing on the current one. This would add context to the current situation and may help reaching better results as measures of the acceleration would then be known.