University of Liège
School of Engineering

Inverted Double Pendulum : Searching High-Quality Policies to Control an Unstable Physical System.

INFO8003: Optimal decision making for complex problems

Julien Gustin, Joachim Houyon

April 26, 2022

1 Domain

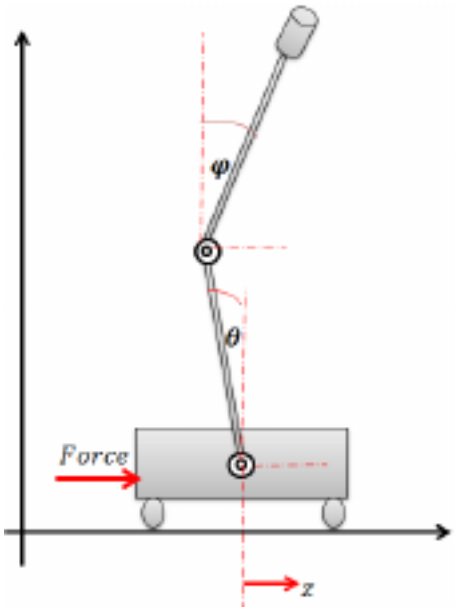


Figure 1. Double Inverted Pendulum environment.

1.1 State space

The Double Inverted Pendulum consists of two joint pendulums connected to a cart that is moving on a track. The state space X is composed of 6 continuous states variables[1] $\mathbf{s} \in \mathbb{R}^6$, for simplicity we will represent the six-tuple by the variable $\boldsymbol{s}$ as

$$\boldsymbol{s} = \begin{bmatrix} x & \dot{x} & \theta & \dot{\theta} & \varphi & \dot{\varphi} \end{bmatrix}^T \tag{1}$$

However in the implementation that we have[2] what the agent observes is not necessarily a state of the environment. We will then use as notation $\boldsymbol{o(s)} \in \mathbb{R}^9$ to represent an observation of state $\boldsymbol{s}$.

$$\boldsymbol{o(s)} = \begin{bmatrix} x & \dot{x} & p_x & \cos\theta & \sin\theta & \dot{\theta} & \cos\varphi & \sin\varphi & \dot{\varphi} \end{bmatrix}^T \tag{2}$$

As it is quite tedious to interpret all the variables directly by looking at the source code, we inspired our-self by looking at [1] and [2].

- x: correspond to the cart position in the x axis, the domain of this variable is $x \in [-1, 1]$
- θ and γ represent respectively the angle of the first and second pendulums as we can see in Figure 1.
- p_x represent the x position of the second pole
- $\dot{a}$, $a \in \{x, \theta, \varphi\}$ represent the velocity of variable a.

[1] In the source they use the notation γ instead of φ, however in our formulation γ represent the discount factor. We have therefore rename this variable with respect to 1

[2] https://github.com/benelot/pybullet-gym/blob/master/pybulletgym/envs/roboschool/envs/pendulum/inverted_double_pendulum_env.py

1.2 Action space

The action space U is continuous where $U = [-1, 1]$, it correspond to the force applied to the cart in one or the other direction.

1.3 Reward function

The implementation define the reward function $r(x, u)$ as follow:

$$r(x_t, u_t) = Ab - Dp_t - Vp \mid x_t, u_t \in X \times U$$

Where Ab and Vp are constants values of respectively 0 and 10.

$$Dp_t = 0.01p_{x_{t+1}}^2 + (p_{y_{t+1}} + 0.3 - 2)^2$$

Where p_x and p_y are the components of the position of the second pole $p = (p_x, p_y)$

1.4 Discount factor

We have set the discount factor γ to 0.99 as the one used in [3], indeed this high discount factor allow the agent to try not to be in a terminal state as much as possible.

1.5 Characterizations of the environment

- Deterministic: There is no noise on the state when applied a force on it
- Partially observable: What the agent observes is not a state of the environment
- Time-invariant
- Continuous
- Single agent

2 Policy search technique

2.1 Continuous action space

2.1.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an algorithm which learns two parameterised functions: a Q-function $Q(s, a|\theta^Q)$ and a policy $\mu(s, \theta^\mu)$, where θ^Q and θ^μ are respectively the parameters of the Q-function and the policy. For the sake of the terminology, the Q-function is called the critic, while the policy is called the actor.

DDPG is an off-policy learning algorithm, meaning that the improvement of the learned policy relies on another policy for the action selection. Furthermore, DDPG is designed for a continuous action space setting. However, since DDPG is an algorithm that makes the use of non-linear function approximators, we have no guarantee of convergence.

2.1.2 Learning strategy

Given a set of transitions $\mathcal{D}$, the learning of the Q-function is done by minimizing the mean-squared Bellman error (MBSE) function:

$$L(\theta^Q, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}[(Q(s, a|\theta^Q) - (r + \lambda(1 - d)(Q(s', \mu(s'|\theta^\mu)|\theta^Q))))^2]$$

where $Q(s', \mu(s'|\theta^\mu)|\theta^Q) \approx \max_{a'} Q(s', a'|\theta^Q)$, and d states whether or not we are in a terminal state. In this setting, a continuous action space is considered. Therefore, applying Q-learning in this setting is intractable as Q-learning implies the computation of the action a which maximises the current Q-value for a state s at time t. Instead, the actor is trained such that it should sample actions that maximise $Q(s, \mu(s|\theta^\mu)|\theta^Q)$. Hence, we can perform gradient ascent such that

$$\theta^\mu = \mathop{argmax}_{\theta} \mathop{\mathbb{E}}_{s\sim\mathcal{D}}[Q(s, \mu(s|\theta)|\theta^Q)].$$

2.1.3 Experience replay

When using supervised methods such as neural networks, it assumes that each sample is independently and identically distributed. However, this is not the case when the samples are generated by exploring the environment. In DDPG, the training is done online and it explores its environment. To address this dependence between samples, the algorithm makes the use of a replay buffer such that if a batch is created by sampling in the replay buffer, it is less likely that samples are highly correlated.

In our case, the replay buffer is of finite size $\mathcal{R}$ and the set of transitions $\mathcal{D}$ is randomly sampled from it. Each transition (s, a, r, s', d) sampled from the environment by using the exploration policy is stored in the replay buffer. The FIFO rule is used for discarding transitions when storing new ones in the case when the buffer is full.

2.1.4 Target networks

One could think that implementing directly the Q-learning as presented in the subsection 2.2 is a good idea, experiments have shown that the learning is very unstable. Indeed, since the target $(r+\lambda(1-d)(Q(s', \mu(s'|\theta^{\mu})|\theta^{Q})$ is moving from one update step to another, the learning is more likely to diverge if the target suffers from high changes. To address this issue, the algorithm instantiates two target networks $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ and the weights of these target networks are updated using exponential averaging:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

With $\tau \in [0, 1]$. Using this kind of update rule has shown to make the learning more stable, because the target networks cannot change too drastically (controlled by the hyper parameter τ). Hence, Q-learning in DDPG is computed by minimizing:

$$L(\theta^{Q}, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}[(Q(s, a|\theta^{Q}) - (r + \lambda(1-d)(Q'(s', \mu'(s'|\theta^{\mu'})|\theta^{Q'})))^2]$$

where $Q'(s', \mu'(s'|\theta^{\mu'}|\theta^{Q'})) \approx \max_{a'} Q'(s', a'|\theta^{Q'})$.

2.1.5 Exploration policy

In a continuous action space setting, the policy $\mu(s|\theta^{\mu})$ is used by adding noise to the sampled action. In our configuration, temporally correlated noise has shown to be very efficient on physical control problems. This being said, an Ornstein-Uhlenbeck process has been used for the exploration.

2.1.6 Algorithm

The figure below shows the algorithm of DDPG. This is the algorithm we have followed for computing a policy for controlling the double inverted pendulum.

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 2. DDPG Algorithm [3]

2.2 Discrete action space

2.2.1 Deep Q-Network

Deep Q-Network (DQN) is an adaptation of DDPG for a discrete action space. In fact, DQN doesn't parameterise the policy, instead, the only thing it learns is a Q-function. In other words, DQN is an algorithm which learns one parameterised function: a Q-function $Q(s, a|\theta)$, where θ is the set of parameters of the function.

The algorithm still uses a target network $Q'(s, a|\theta')$ which is updated in the same way as stated in 2.1.4 with the parameters θ, the algorithm make also use of a replay buffer. The exploration policy is different because, in DDPG, the way we explore is only suited for continuous action spaces.

2.2.2 Learning the Q-function

In a discrete action space setting, the computation of an action a which maximises the current Q-value for a state s at a time t is tractable. Therefore, given a set of transitions $\mathbb{D}$, the learning of the Q-function is done by minimizing the MBSE function:

$$L(\theta, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}[(Q(s, a|\theta) - (r + \lambda(1 - d)\mathop{max}_{a'} Q'(s', a'|\theta')))^2]$$

2.2.3 Exploration policy

The exploration policy is built such that at the beginning it should explore a lot, and decrease the exploration rate as time goes on.

This behavior can be done using an epsilon greedy algorithm where ϵ annealed linearly starting from $\epsilon_0 = 1$, decrease the exploration rate down to 0.1 such that $\epsilon_t \leq \epsilon_{t-1}$ and $\epsilon_t \geq 0.1\ \forall t \in [1, T]$ for the first 50% steps of the total number of steps used for training. Then ϵ stay at 0.1.

2.2.4 Algorithm

The figure below shows the algorithm of DQN. This is the algorithm we have followed for computing a policy for controlling the double inverted pendulum in a discrete action space setting.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3. DQN Algorithm [4]

2.3 Fitted-Q iteration with Extremely Randomized Trees

For the fitted-Q iteration (FQI) we have adapted our model from assignment 2 to the double pendulum problem. The idea is that at each iteration, one build a training set using h_t and the approximation of the precedent Q iteration. Where h_t is a set of one-step transitions $\{(x_k, u_k, r_k, x'_k)\}_{k=0}^{t-1}$ More formally

- $N = 1$ the algorithm determine $\hat{Q}_1$ of $Q_1(x,u) = E_{w\sim P_w(.|x,u)}[r(x,u,x)]$ on the training set
$$TS_1 = \{((x_k, u_k), r_k)\}_{k=0}^{t-1}$$
- $N > 1$ the algorithm outputs a model $\hat{Q}_N$ of $Q_N(x,u) = E_{w\sim P_w(|x,u)}[r(x,u,x) + \gamma max_{u'\in U}(x_{k+1}, u')]$ by using
$$TS_N = \{((x_k, u_k), r_k + max_{u'\in U}\hat{Q}_{N-1}(x_{k+1}, u'))\}_{k=0}^{t-1}$$

Where $\hat{Q}$ is estimated using extremely randomized tree with 50 extra-trees.

3 Experiments

In this section, we will explain how the algorithms have been implemented and evaluated. The DDPG, the DQN for various discretisation of the action space and the Fitted-Q algorithms will be evaluated and compared with respect to expected cumulative rewards of the derived policies.

We estimated the expected cumulative reward of a policy J^μ using Monte-Carlo principle. We used a set of K[3] i.i.d initial states which are same set for every episodes and algorithms[4]. Then we computed the expected return such that

$$J_\infty^\mu = \frac{\sum_{x\in X^i} J_\infty^\mu(x)}{K} \tag{3}$$

Where $J_\infty^\mu(x)\ \forall x \in X^i, 0 \leq i < K$ has been estimated by simulating the policy starting from an initial state x for N steps. We will name this approximation as J_N^μ.

$$J_N^\mu(x) = \sum_{t=0}^{N-1} \lambda^t r(x_t, \mu(x_t))|x_0 = x$$

Where γ is the discount factor. In the experiments, γ has been set to 0.99.

3.1 DDPG

3.1.1 Network architectures

The critic network Q is a MLP that has 2 hidden layers of 400 and 300 neurons respectively. The action is only included at the first hidden layer. Adam optimizer has been used with a learning rate of 10^{-3}. The weights of the last layers are initialized from a uniform distribution $[3 \times -10^{-4}, 3 \times -10^4]$.

[3] K is set to 50 in our experiments

[4] using the same seed

The actor network μ is a MLP that has 2 hidden layers of 400 and 300 neurons respectively. The last layer of the network is a tanh layer so that the action is bounded in $[-1, 1]$. Adam optimizer has been used with a learning rate of 10^{-2}. The weights are initialized from a uniform distribution $[3 \times -10^{-3}, 3 \times -10^{3}]$.

For both networks, all the hidden layers use ReLU as activation function. The target networks are updated with $\tau = 0.001$. The architecture of these networks have been inspired from [3]

3.1.2 Performances

The performance of DDPG has been evaluated on 500 episodes, where each episode represents 1000 steps in the environment. The replay buffer has a capacity of 10^6 samples and batches of size 64 are sampled from it.

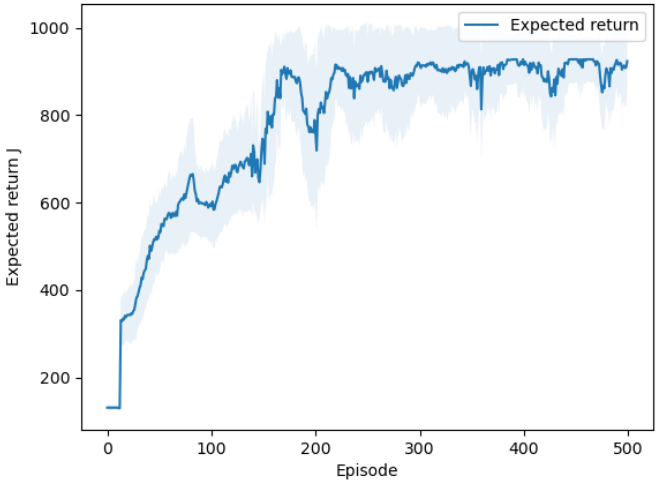


Figure 4. Expected DDPG training using $\gamma = 0.99$

The algorithm seems to converge after $\sim$ 200 episodes with an expected return of $\sim$ 930. When simulating the policy on the double pendulum, it almost never fails and seems to have a relatively rational policy. Indeed, it keeps oscillating by going left and right to keep the double pendulum stable. A video is available at **gif/optimal_policy.gif** in the project folder

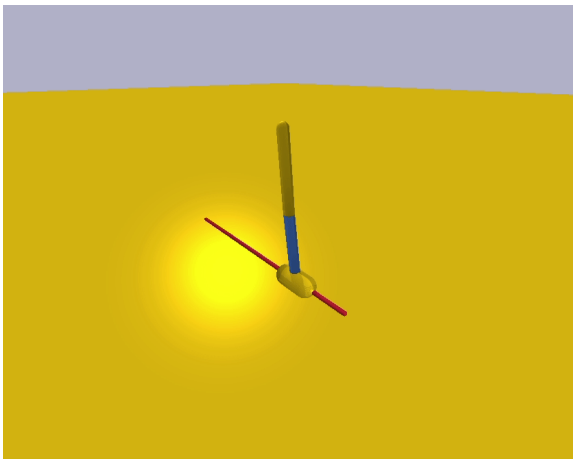


Figure 5. Animation of the double pendulum using a policy trained by DDPG

3.2 DQN

3.2.1 Network architecture

The critic network has the same architecture, optimizer and initialization as the critic in the DDPG algorithm except that it doesn't take as input an action. Instead, the output layer of the critic contains a number of neurons equal to the action space: The $i-th$ neuron of the output layer represents the estimated expected output of the $i-th$ action when being at a state s.

3.2.2 Performances

The evaluation of DQN has been done using the same setting than the evaluation of DDPG. Regarding the discretisation of the action space, it has been evaluated on different sizes by evenly spaced numbers over the interval $[-1, 1]$.

By looking at the plot 7 we can see that using DQN with diverse discretisation need almost 200 more episodes than DDPG to converges, moreover DQN reach a plateau at the value ~ 810 which is $\sim 10\%$ worst than DDPG. This result can be due by the fact that discretize the action space does not give a high enough degree of freedom for the agent.

Also DQN look like being less stable than DDPG by looking at all these spikes. Surprisingly increasing the number of actions does not change that much the expected cumulative reward from 3 to 13 actions, and it is quite interesting to look at a simulation of these different model because according to the action space the model adapt its strategies, for example with 11 actions the network learned a strategy that consist at staying at the left side of the rope and applying small force on the left and right in order to keep the double pendulum stable. It looks like cheating but it works! At the opposite when really few actions are available its strategy is to do big left and right oscillation.

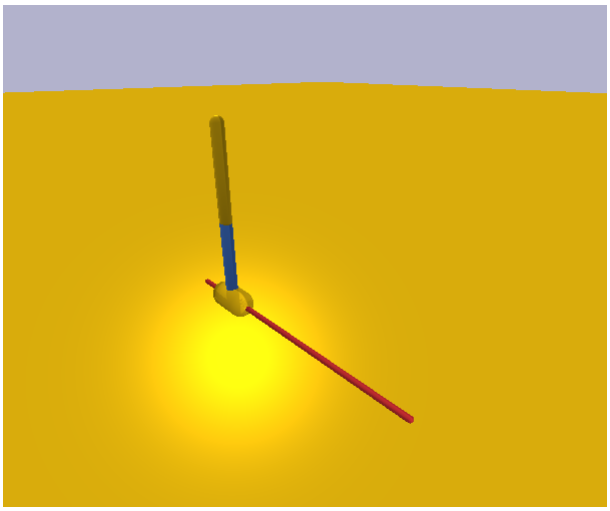
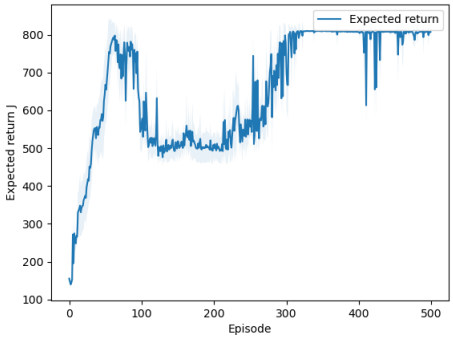
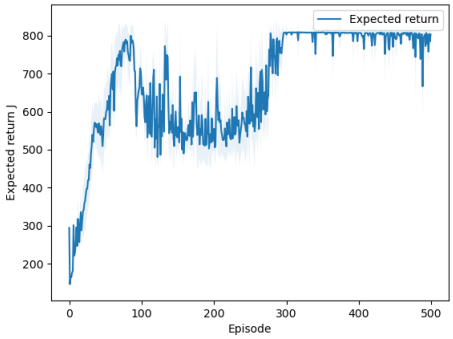


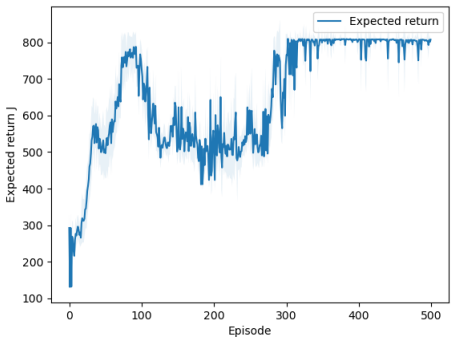
Figure 6. Strategy of DQN with 11 actions



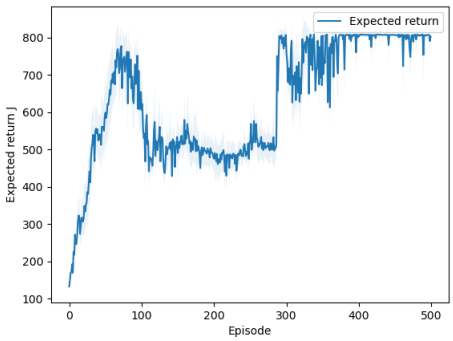
(a) DQN Discrete action: 3



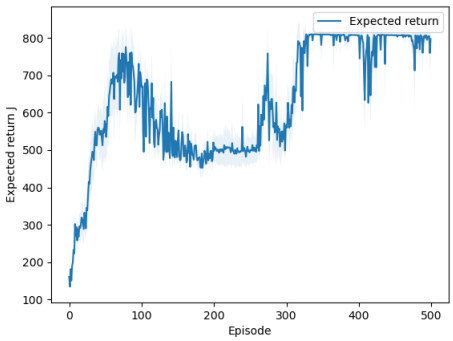
(b) DQN Discrete action: 5



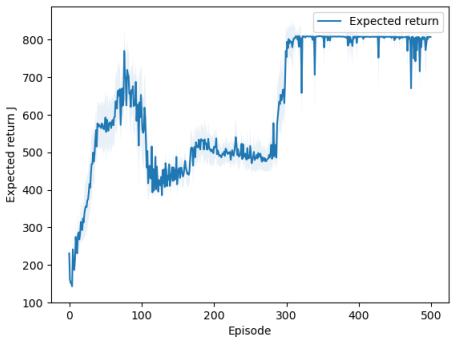
(c) DQN Discrete action: 7



(d) DQN Discrete action: 9



(e) DQN Discrete action: 11



(f) DQN Discrete action: 13

Figure 7. DQN algorithm on diverse action space with $\gamma = 0.99$

3.3 FQI

1. We generate 500000 samples of (state, action, reward, state') where we restarted the environment every time a terminal state is reached.
2. Compute a bound N such that $N = \left\lceil \log_\gamma(\frac{\epsilon}{2B_r}(1-\gamma)^2) \right\rceil$, with $\epsilon = 0.1, B_r = 10$ and $\gamma = 0.99$. This gives $N = 1444$. However, by looking at the plot it is clear that it has converged to after $N = 100$, moreover performing 1444 iterations comes with high computational time. That is why we decided to fix N to 200.

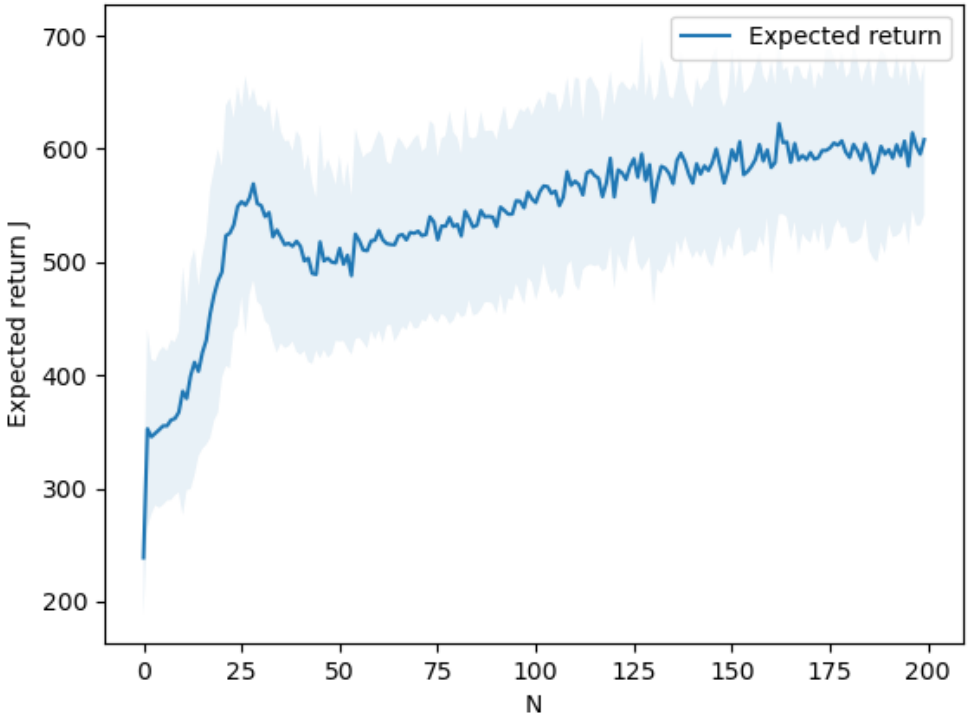


Figure 8. Expected cumulative reward using fitted-Q iteration fed with 500k samples with respect to N.

The expected return seems to converge towards ~ 600, this value is way much lower than DQN and DDPG, it may be due by the way we sampled trajectories, indeed, trajectories had been sampled using a random policy beforehand. The chaoticness of the double pendulum does not allow moving too far from the centre without failing by only applying random actions.

This will result by having a model that as more we move away from the middle the less it will knows what to do. When looking at a simulation we can clearly see that when getting closer to an edge the model perform some weird action leading to failure while when it is at the center its move does make sense.

4 Implementation

All the code is available in the folder with DDPG and DQN pretrained model in */saved_models*. Please look at the **README.md** in order to use our code.

4.1 Use

Make sure to have installed pybullet-gym[5] before using the program.

```
python main.py [-h] [-ddpg] [-fqi] [-dql] [-batchnorm] [-render RENDER] [-gamma
GAMMA] [-samples SAMPLES] [-actions ACTIONS] [-seed SEED]
```

`RENDER` should be a file toward a saved model for either **dql** or **ddpg**

-> *This will render the double pendulum with the given pretrained model.*

`GAMMA` is the discount factor $\gamma \in [0, 1]$

-> *0.99 give the best results*

`SAMPLES` are the number of samples used when training **fqi**

-> *Higher is the better but 200k give reasonable good result (computation expensive)*

`ACTIONS` number of discrete action when using either **ddpg** or **fqi**

-> *Should be an odd number*

`SEED` the seed to use

4.2 Examples

4.2.1 Load a saved model using ddpg

```
python main.py -ddpg -gamma 0.99 -render saved_models/DDPG
```

4.2.2 Train a dql using 11 discrete actions

```
python main.py -dql -gamma 0.99 -actions 11
```

[5] <https://github.com/benelot/pybullet-gym>

5 Conclusion

5.1 Policy search techniques

During this project, several policy search techniques have been implemented for controlling an unstable physical system. Overall, all of the techniques managed to find a decent but not necessarily optimal policy.

Fitted-Q Iteration has found the policy with the lowest expected cumulative reward. Even though the experiment did not iterate enough in order to reach the bound N, we can expect FQI with XRT to be less efficient as it requires high computational time to compute the policy, and XRT are not the best function approximators, even though non-parametric approximators have the advantage of ensuring convergence.

Considering DDPG and DQN, DDPG has found a policy with a higher expected cumulative reward than the policy found by DQN. One could say that DDPG is better at finding good policies than DQN, however, the hyper-parameter tuning is more tedious; DDPG trains an additional network, which is a bunch of parameters to tune. Therefore, DQN could offer more stable learning than DDPG.

5.2 To go further

In order to find better policies, several techniques could be used, such as prioritized experience replay. Prioritized experience replay is a form of experience replay where we replay more frequently transitions that have a high expected learning process; this is characterized by the magnitude of the temporal-difference error. Therefore, transitions are not uniform anymore.

Another improvement could be to find a policy based on images; the state space would become a snapshot of the environment. This problem would be tackled by either DDPG or DQN, where the critic and the actor architectures would be convolutional neural networks.

References

[1] Fredrik Gustafsson. "Control of Inverted Double Pendulum using Reinforcement Learning". URL: http://cs229.stanford.edu/proj2016/report/Gustafsson-ControlOfInvertedDoublePendulumUsingReinforcementLearning-report.pdf (page 1).

[2] openocl. "Modeling for Reinforcement Learning and Optimal Control: Double pendulum on a cart". URL: https://openocl.github.io/tutorials/tutorial-01-modeling-double-cartpole/ (page 1).

[3] Google Deepmind. "Continuous control with deep reinforcement learning". URL: https://arxiv.org/pdf/1509.02971.pdf (pages 2, 4, 8).

[4] DeepMind Technologies. "Playing Atari with Deep Reinforcement Learning". URL: https://arxiv.org/pdf/1312.5602.pdf (page 6).