Artificial Intelligence – Deep Deterministic Policy Gradient

Giulio Turrisi

Course: IA & Robotics

Reinforcement Learning is a technique based on a trial-and error paradigm, which aim is to develop an intelligent agent using only rewards as a feedback to maximize it’s performance.

A tone of different methods was developed to solve this kind of problems, and in some cases they worked amazingly. Just thinking about the Deep Mind Paper from 2012 for example, "Human-level control through deep reinforcement learning":
They solved and succeded in a lot of different games scenario, using just the image of the screen as input, gaining results outstanding and comparable/superior from the human's ones in the majority of the games.
(For this purpose, they used different tricks, as the "replay buffer" and a "target network", that will be present in this work too and described below.)

But one of the drawbacks of their techinique lies in the output space. They can only handle discrete and low dimensional actions! They used one neuron per actions as output, and so in many case this can become a serious problem. If you have to live with a continuos range of values(as in our case, a torque for a motor for example), we can't just use it. The output dimension will become huge and intractable.

A noticeable way to solve this problem can be found in the paper "Continuos Control with Deep Reinforcement Leaning", published in 2016, and that is the essence of this work.

It's based on the Determistic Policy Gradient alghorithm, with an addition of differents techiniques to working well with a neural network approach, knew to be very unstable in this kind of problems.

In the following part we will introduce some basic concept on the topic, the algorithm, the architecture of the neural net used and so on. And in the end, will be shown the result gained by applying it into the Gym(OpenIA) platform.

1. REINFORCEMENT LEARNING

As I said before, reinforcement learning is a technique that consists on the agent acting in the world, collecting reward from his actions and trying to optimize his behavior, autonomously.

In our case we represent all these concepts with this tuple, **<state,action,reward,new_state>**. The agent is in a particular state X of the world, performs an action A and goes in a new state X’, collecting a reward R from it.

So one of the crucial step is how to assign a reward, to allow an agent to learn to act optimally in the world. In our case, the platform used(Gym) is the responsible of their assignment. The way how it works will be described in a dedicated chapter.

The final outcome of the entire process is to find a good policy(behavior), which will permit to maximize the sum of all the rewards.

Policy gradient is an algorithm often used to solve continuous problem. The are two kind of version of it, the Stochastic and the Deterministic approach.

The first one consists in sample stochastically one action A in one state S, given some policy parameter, and then move this ones in the direction of greater maximum rewards. Instead in the last one we have just one association state→action.

The agent in the first time will begin to explore the world to collect the rewards and updating this values, acting randomly or in a specific way trying maximize them(exploitation step).

The first approach, the stochastic one, is more suitable for doing this. So in this paper they combine two approach, using a stochastic behavior policy for exploration, but learning estimating a deterministic target policy that is much easier to learn, using an off-policy algorithm.

For the exploration part, as suggest in the paper, I just added noise the the choice of the action. This noise little by little will become less present in our choice, tending to zero when the number of episode increase sufficently.

2. NEURAL NET ARCHITECTURE

The DDPG maintains 2 differents neural nets, to parametrice the policy function(called Actor Net), and the Q-Value(the Critic net). Actually the algorithm uses 4 nets, with the others two that can be seen as temporary “image” of the firsts described. But I’ll explain these concept later.

- In the **Actor Net,** I used three different layers, dimensions that are 400 neurons for the first layer, followed by 300 units and in the end just one output, that represent the single action that our game permits.

The input is the state of the game(parameters needed to describe the orientation of the pendulum in the world), and the output is strictly bounded in the range [-2.2], that represent the admissible torque.

N.B. If the game state was just the image of the screen, just adding a convolutional part before the fully connected layer will be good for make it works. Obviously the complexity in the training will be increased(but we can down sample the image), but the convergence will be feasible too.

- In the **Critic Net**, that represent the Q-Value, the inputs are again the state of the game, and the action choose in that particular moment. The output is the value associated with that tuple. Here too the layers is similar from the Actor Net, with the two inputs mixed up. (See the code for an easy understanding).

Before explaining the algorithm in detail, it’s easier to describe first of all the “trick” used to stabilize the use of NN in this case. This methods was introduced by DeepMind, that showed in their different papers how the training process was greatly helped by them. (In the last paragraphs of this work, I will provide some graphs to show the deterioration of the convergence if we drop them.)

These methods are called **“Experience Replay”**(called also replay buffer) and **“Target Net”**

Experience Replay:

One problem that causes instability of using this algorithm with a neural net, is the correlation between sequential states. One easy way to remove it, is to train our networks not with the actual state, but with a randomly sampled one obtained in the past.
So, every time we performs a step, we put the new state in a buffer(called in our case replay_buffer), and performing the backpropagation sampling randomly from it. This addition requires to populate it a little bit before starting training our network, and so in the beginning there are a fixed number of step where the backprop. step is not performed.(Actually, the tuple inserted is <state,action,reward,new_state>).

Target Networks:
An additional implementation is the creation of a second neural network for each principal Nets(so one for the Actor, one for the Critic - identical to them), so that at the beginning they have the same weight/bias values.
One of the target networks will be used to calculate the loss to perform the backprop. step(done on the Actor Net)!

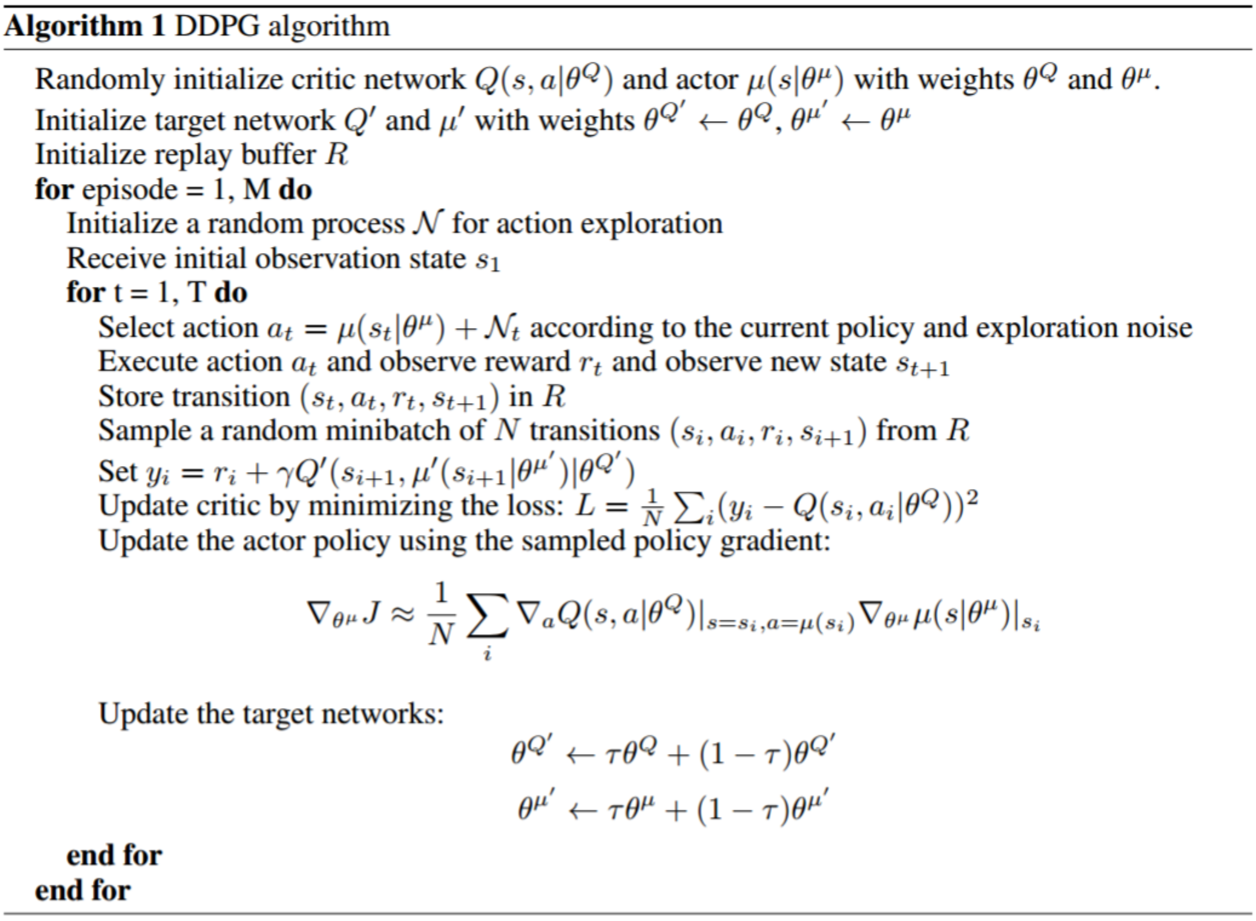
$$\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$$

Yj will be calculated using the target networks(sampling in the replay memory, taking the next-state and putting it as input in the Actor-Target to calculate the best action to perform. After that, the action taken and the states are used as inputs for the Critic-Target, to calculate the Q-Value on the left of the formula).
Instead the second element on the right, using directly the principal two nets, from the same sampling but using the old state and the action associated with it, salved in the replay buffer and taken from the same tuple of the next-state of before.

The reason why Deep Mind choose to implement it, is to increase further the stability of the algorithm. In fact during all the step, our Main networks values shift constantly, and if we use a constantly shifting values to adjust our network values, then it can give a large instability problem.
So they maintain the weight-bias of the Targets constant, and after some number of steps they update their values copying them from the main ones.

But instead to use this “hard copy approach”, it was shown that if we slightly copy little by little the weights between the nets, without make this copy brutally, the convergence will be easier. (For this I will provide in the end a graph to show the difference during the training too).

The next step is to explain the algorithm, and for doing this I will post the pseudo-code provided from the original paper, joining this with the comments inside the code provided.



Algorithm 1 DDPG algorithm

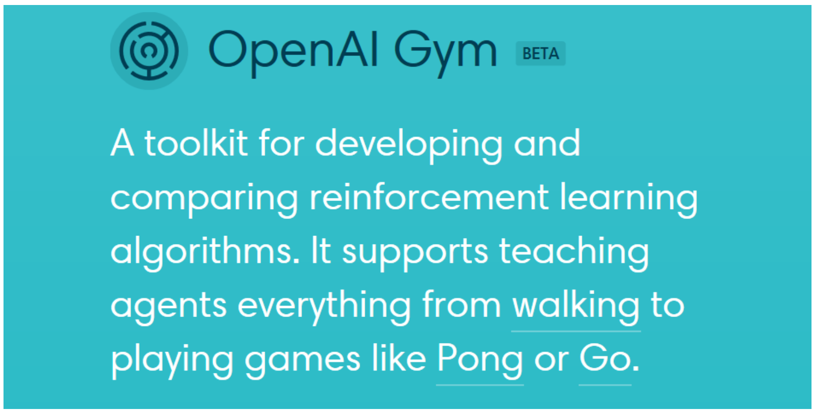
```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 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
```

The exploration was performed adding random noise to the chosen action(this noise will decrease each episode). For update the Actor-Net, as express in this pseudo-code I took the gradient from the Critic and multilied with the one of the Actor. The prove of this formula was provided by another paper, which link is in the last page. For the update of the targets I just taken an easy function found of the net, that performs the same operations.

As I said before, the explanation of the steps was provided before(the use of the replay buffer – target networks). In the code I added a few comment to make it more clear.

3. GYM – An OpenAI Platform

For testing our implementation, i used a fantastic tool provided by “OpenAI”, a non profit-organization with the aim to enhance the artificial intelligence research. They created Gym, a toolkit for fast prototyping algorithm, due to the easy access to a large number of possible environment where test it. There are present a lot of different games, with a large range of difficulties, where you can find everything from a simple game to Doom.



OpenAI Gym BETA

A toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

I used it to test the algorithm on the environment 'Pendulum-v0', one of the few choice present with a continuous range of action available. There are other different choice(game for gate generations for example), but the majority of them are just for a free limited trial.

In the case of discrete actions, there are instead a lot of different choice, with a large number of different Atari games.

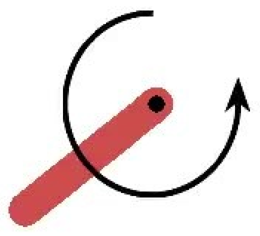
To interface with this tool, there are only few lines of code needed, easily explained in the table below.

```
#importing the libraries
import gym
#create the environment
env = gym.make('Pendulum-v0')
#resetting it
state = env.reset()
#after choosing an action, performing it and obtain reward and next state
obs, reward, done, _ = env.step(action)
```

As I said before, all the assignment of the reward is masked by the platform, permitting a rapid prototyping of the algorithm instead.

4. PENDULUM-v0

Pendulum-v0 is a typical example where this approach can be easily test. The state of the game is an array of values that describe the position(angle, orientation etc) of the link in the word, but this algorithm can be easily adapted in other cases where this is an image of the screen (just adding a convolutional neural net before the fully connected layer. The convergence is not a problem in this case neither.)

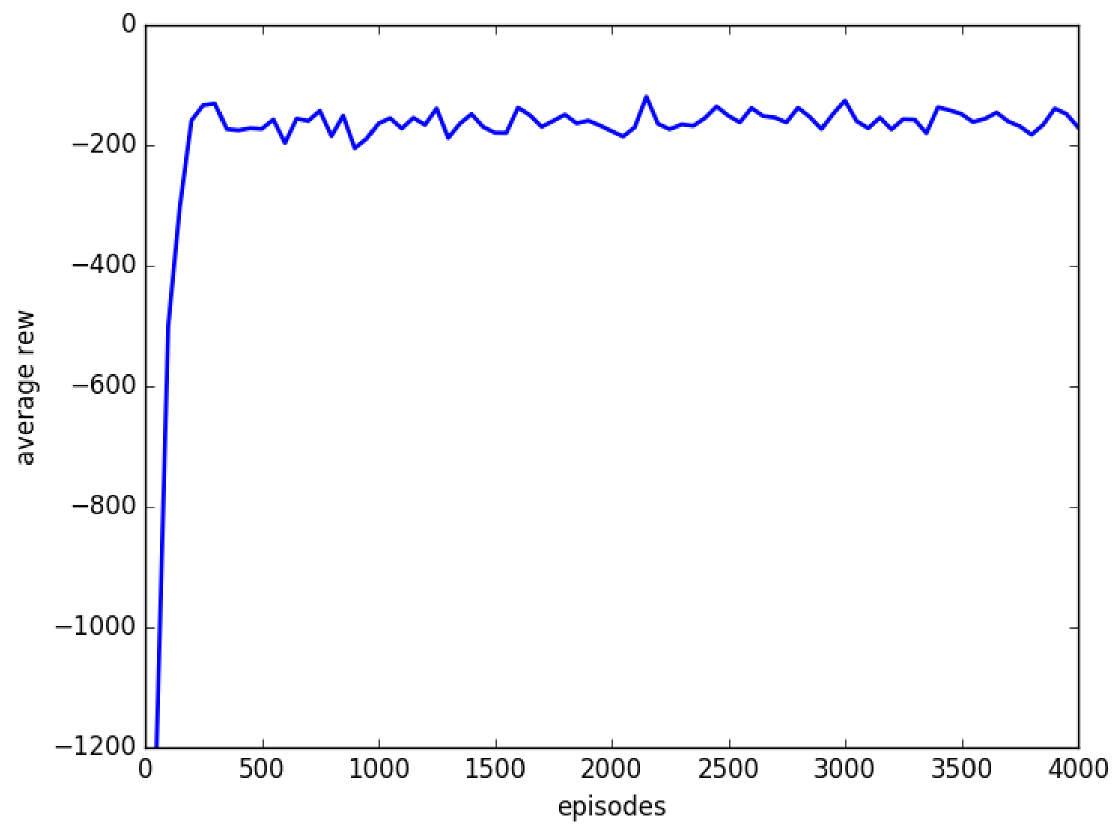


We have here just one continuous action, bounded in the interval [-2.2] that represent the torque admissible for the pendulum. The limited interval is quite little in this case, but as described in the paragraph before, the presence of a large one is not a threat for the final convergence.

From the Gym documentation:
"The inverted pendulum swing-up problem is a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright."

The learning part was performed over **4000 episodes**, each ones formed by **1000 max_epLenght**, which takes about 3 hours to end. The **replay_memory_size** was settled to **10000** elements, while the **tau variable**(that expresses the update rate between the target networks and the main ones) to **0.001.**

The graph resulting from the learning part is attached below:



How it can be seen, the algorithm converges very quickly just after 500 episodes. Instead, from my previous experience, in the most simple case(discrete action) the DQN from deepmind is rate slower.

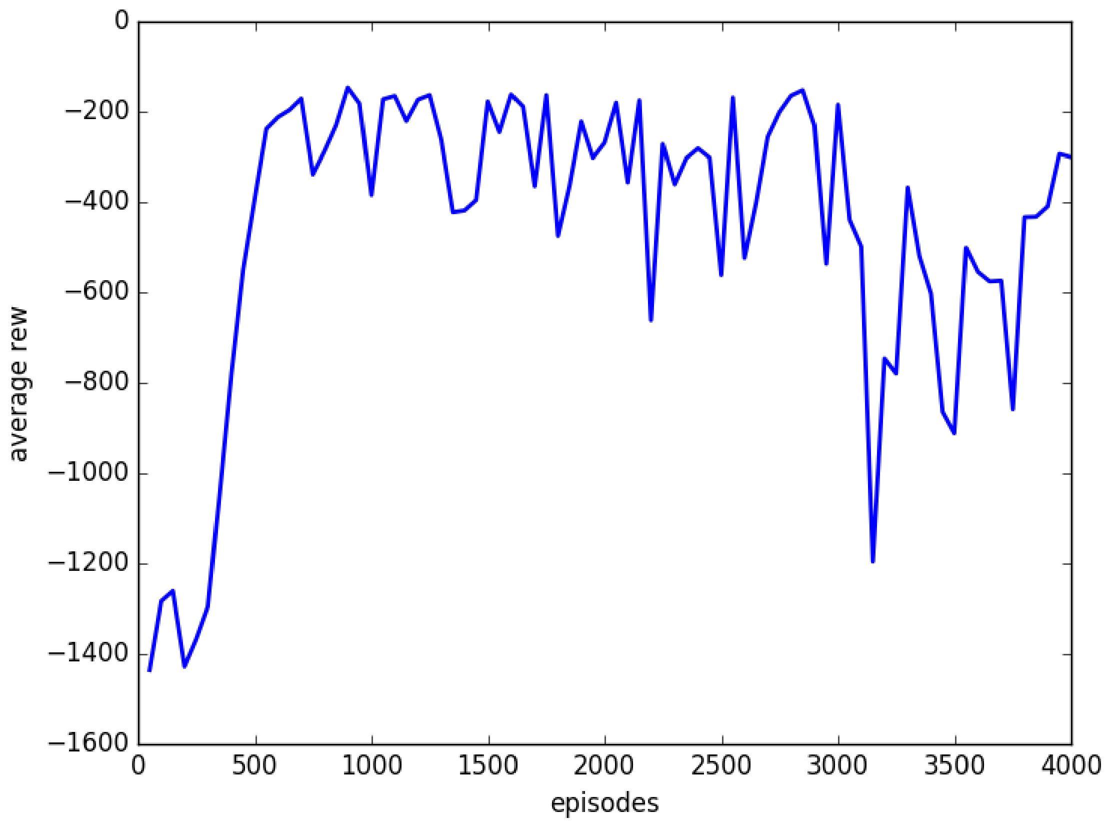
The result obtained is quite good, performing in the testing part **an average point of -144**. If we take a look from the official result in the Gym website for this environment, it is consistent with them.

The link learn to stand still vertically!

N.B The rewards are negatives, and the maximum obtainable is < 0.

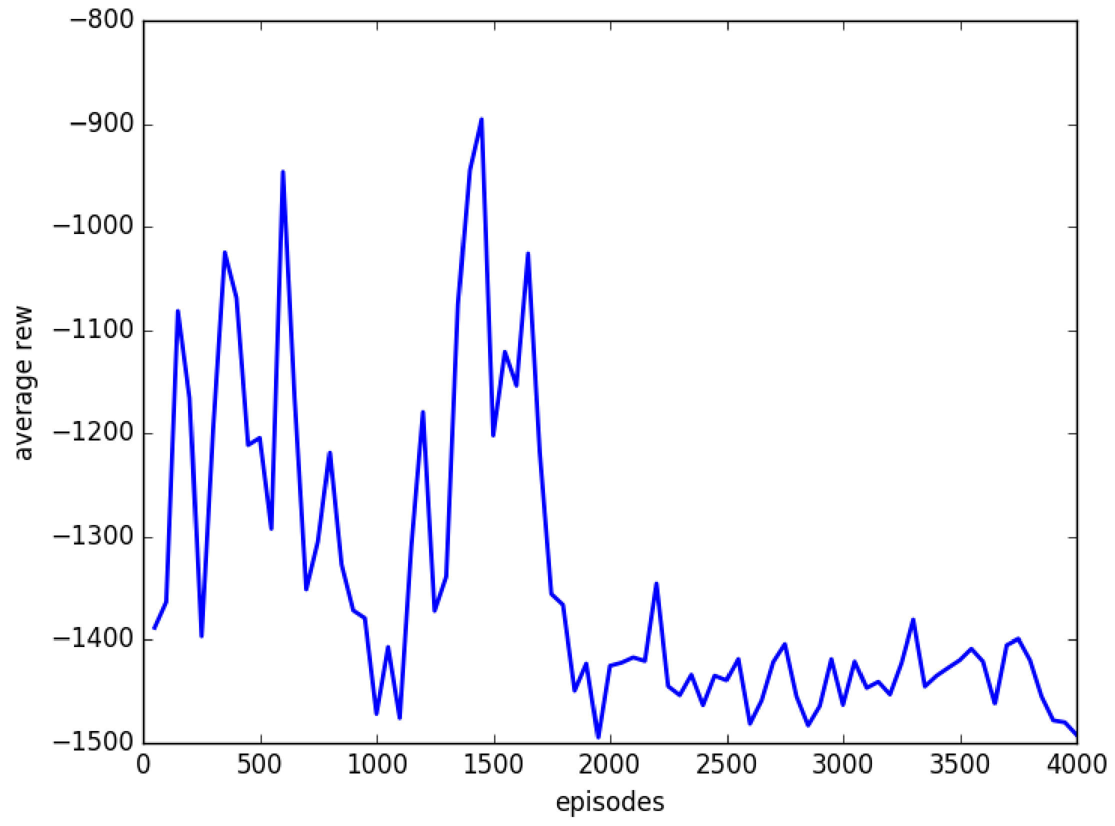
After obtaining this result, I just re-perform the training part with different initial condition. The biggest advantage is claimed to be obtained through the REPLAY BUFFER and the slowly update of the networks, so I just eliminate them from the code(singularly) and doing all again.

Without the slowly update, performing it copying directly all the weights/bias after each episode:



We can see how the training suddenly become unstable. If we modify the timing(performing the copy more often) it will better stabilize, but never reaching the good result of the adopted method.

Instead, without the replay buffer:



As we can see, the algorithm completely diverges! In this case, I just took the latest experience putting them directly in the process, without randomize any of it.

N.B. All the initialization of the different parameters was done without an explicit search of the best values. Just tried a few values and took the best. Obviously this is not a rigorous method, but the testing part took time to finish.

5. FINAL CONSIDERATION

As the graphs shown, the algorithm is very stable despite of the use of the neural net as approximator, known to be the origin of a lot of problems in this field.

The result of this implementation is quite good, reaching the best score obtained in this environment presents on the website.

But this is not the end. A lot of developments was done after this paper was published, for example from the paper "Asynchronous Methods for Deep Reinforcement Learning (A3C)", that get ride off of the replay buffer!

Instead of using it, this method use multiple agents on different threads to explore the state spaces and make decorrelated updates to the actor and the critic, making the training part a much way faster then the other methods(on GPU!), using only the processor of the machine.

- https://arxiv.org/pdf/1509.02971.pdf (Deep Deterministic Policy Gradient Paper)

- https://gym.openai.com/envs/Pendulum-v0 (Gym Pendulum Environment – with the list of scores gained from different implementation)

- https://arxiv.org/pdf/1602.01783.pdf (Asynchronous Methods for Deep Reinforcement Learning Paper)

- http://proceedings.mlr.press/v32/silver14.pdf (Deterministic Policy Gradient, prove of gradient)