Udacity

Project Report

P2 - Continuous Control

Author:
Thomas Di Martino

Supervisor:
Udacity

A report submitted in fulfilment of the requirements
for the Student in the Deep Reinforcment Learning Nanodegree Program

of the

Artificial Intelligence department

May 2020

Declaration of Authorship

I, Thomas Di Martino, confirm that this work, 'P2 - Continuous Control', submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 22/05/2020

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem diagnosis

In the context of the *Reacher* environment, our agent is an arm with two joints that needs to move to different target locations, modelized by a round 3D Green ball rotating around the arm's extremity. The Reacher environment of the agent is highly similar to the one developed by Unity in the following sets of learning environment: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher

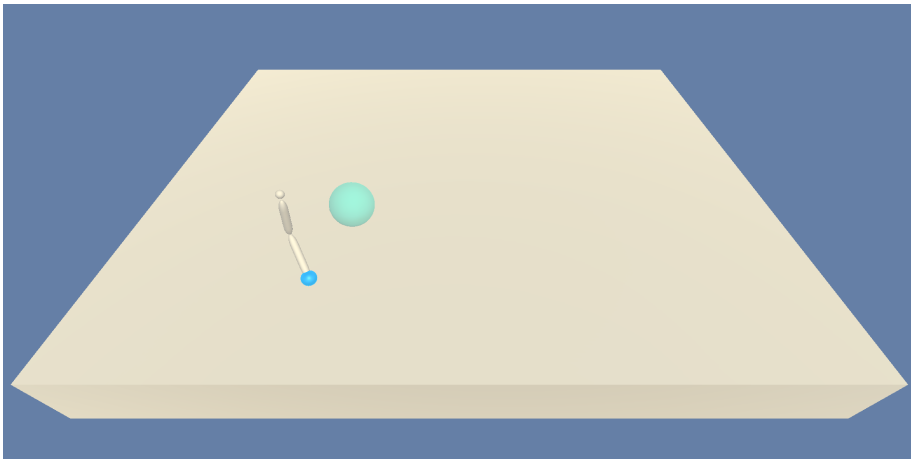

FIGURE 1.1: Screenshot of the agent's environment

1.2 Environment Introduction, a RL perspective

1.2.1 States

To capture is knowledge of the environment, our agent has an observation space of size 33. Amongst all the variables making up this vector, we can count the position of the arm, its velocity (of the arm as a whole and of its angles), its rotation index and so on.

```
[ 0.00000000e+00 -4.00000000e+00  0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
  5.55726671e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

FIGURE 1.2: Agent state example

1.2.2 Rewards

The reward of the agent is pretty straight forward:
it is given +0.1 score for every time-step where the agent's hand is inside the ball. As no negative reward is given to the agent, its behaviour may not be physically-speaking ideal:
any torque will not be penalised. However, such bad behaviour should always be taken into account when developing a RL agent for a concrete robot implementation. A more elaborate reward function could then be helpful to train a better agent, even if training would then be made harder.

1.2.3 Actions

To collect these rewards, our agent can move itself. These movement are represented as a vector of 4 values, with each value being between -1 and 1.

1.2.4 Success Criteria

When training the agent, it is considered successful to the eyes of the assignment when it gathers a reward of **+30** over 100 consecutive episodes.

1.3 Report Structure

In this report, we will present our implementation as well as expose their resulting learning performance. We will then explore eventual amelioration of the current work, exposing the weaknesses of both the retained model as well as for the environment (a parallel may even be drawn with real life situations).

1.4 Code location

Located in a Github repository at the following address https://github.com/dimartinot/P2-Continuous_Control

Chapter 2

Models & Results

2.1 Retained models

One agent was trained during this work:

- A DDPG [1] Agent that implements **prioritised experience replay** using the following formula $\frac{p^\alpha}{\sum_i p_i^\alpha}$ where p is the lastly measured td error of the experience tuple, before its insertion in the buffer. Alpha is an hyper-parameter: the closer alpha is to zero, the more uniform the sampling distribution will be. The closer to one, the less uniform it will be.

2.1.1 DDPG Agent

2.1.1.1 Hyper-Parameters and formula presentation

Member of the family of Actor-Critic methods, the DDPG Agent is ideal for continuous task. Actor-Critic methods are a family of RL algorithms mixing policy-based and value-based learning:

- In **value-based** methods, the agent is learning to estimate the value of actions and situations;
- In **policy-based** methods, the agent is learning to act.

Hence, Actor-Critic methods try to learn both of these concepts, through the use of two separate neural networks:

[1] Deep Deterministic Policy Gradient

1. The **Actor**, outputting the probability of taking any action given the state and the policy;
2. The **Critic**, giving the value of an input state.

Actor-Critic methods follow this algorithm:

- Step 1: input current state S into the actor, get the action a to perform (stochastically);
- Step 2: observe next state S' and reward r -¿ collect a tuple (S, a, r, S');
- Step 3: Train the critic with r + gamma * V(S');
- Step 4: Use the critic to calculate the advantage $A(S, a) = r + \gamma * V(S'; \theta_v) - V(S; \theta_v)$
- Step 5: Train the actor with the calculated advantage $A(S, a)$.

From this algorithm, the DDPG implementation varies in the following points:

- the actor **always** output the best possible action (no stochastic choice of action);
- the critic evaluates the actor's best-believed action;
- the DDPG algorithm uses **soft updates** and mix in 0.1% of the regular weights in the target weights at **every time step**.

Hence, given that the DDPG uses 2 networks (actor and critic networks) and that each of them as their mutual *regular* and *target* version, a DDPG algorithm manipulates a total of 4 distinct neural networks.

The Critic network The critic network was trained with the following hyper-parameters:

- Adam optimiser with a Learning Rate of 10^{-3};
- A batch size of 128;
- Soft update rate $\tau = 10^{-3}$;
- Delayed update: the networks are updated 20 times every 10 time step;

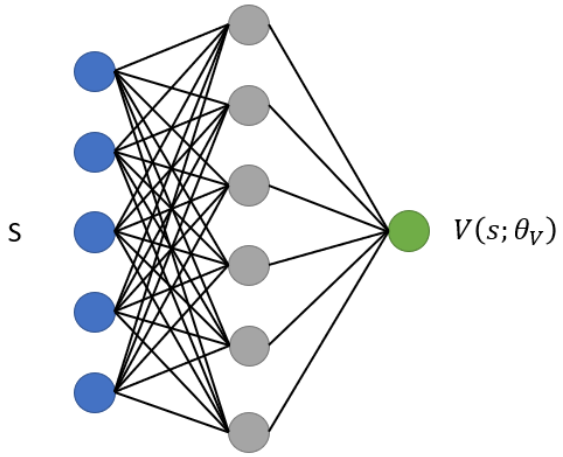

FIGURE 2.1: Schematic representation of a critic network

- Epsilon value for noise addition is set at $\epsilon = 1.0$ at the start of training and is slowly decreasing by the subtraction of 10^{-6} at every timestep;
- MSE Loss between the TD target and the local weights evaluation of the (state, action) tuple. Formally, it can be written as: $L = \frac{1}{N}\sum_{n=1}^{N}(R_{t+1}+\gamma*max_a(\hat{q}(S_{n+1},a,W_t))-\hat{q}(S_n,A,W_l))^2$

where:

- $R_{t+1} + \gamma * max_a(\hat{q}(S_{n+1},a,W_t))$ is the target value (which is evolving through computation), that we would write as $\hat{y}$ in a common Machine Learning problem;
- $\hat{q}(S_n,A,W_l)$ is the predicted value, that we would write as y;
- N is the batch size (in our implementation, we selected 128 as a batch size).

The Actor network The actor network was trained with the following hyperparameters:

- Adam optimiser with a Learning Rate of 10^{-3};
- A batch size of 128;
- Soft update rate $\tau = 10^{-3}$;
- Delayed update: the networks are updated 20 times every 10 time step;
- Epsilon value for noise addition is set at $\epsilon = 1.0$ at the start of training and is slowly decreasing by the subtraction of 10^{-6} at every timestep;

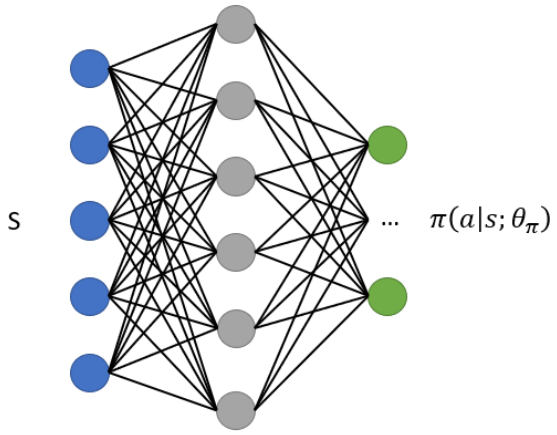

FIGURE 2.2: Schematic representation of an actor network

- The evaluation by the critic of the action chosen by the actor acts as a loss: set negative, we try to minimise it (transformed as a negative function, minimising it is the same as maximising the positive function).

To train the local network, we use a *Mean Squared Error* loss between the TD Target and the local weights evaluation of the (state, action) tuple.

2.1.1.2 Deep Learning Model Architecture

The following figures (cf. 2.3 & 2.4) presents the retained Deep Learning architectures. I took massive inspiration from the model used in the lab of ddpg. Experimentation was done by adding/removing layers to have a good balance between depth and shallowness.

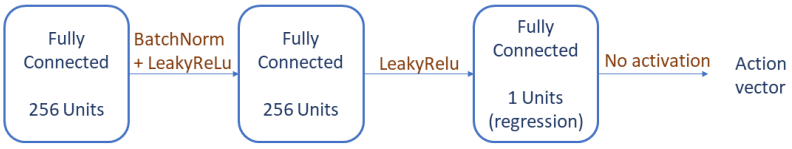

FIGURE 2.3: Critic Model Architecture

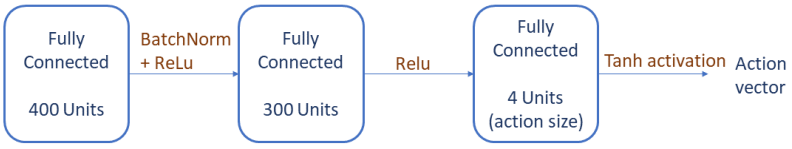

FIGURE 2.4: Actor Model Architecture

2.1.1.3 File architecture

Relevant files for this part of the implementation are:

- *ddpg_agent.py*: Contains the class definition of the basic DPPG learning algorithm that uses soft update (for weight transfer between the local and target networks) as well as a uniformly distributed replay buffer and OUNoise to model the exploration/exploitation dilemma;
- *model.py*: Contains the PyTorch class definition of the Actor and the critic neural networks, used by their mutual target and local network's version;
- *prioritized_ddpg_agent.py*: Contains the class definition of the DDPG learning algorithm with prioritized replay buffer;
- *DDPG.ipynb*: Training of the DDPG agent;

- $Prioritised_D DPG.ipynb : Training of the prioritised DDPG agent.$

2.2 Results

We now present the results of training these two architecture with the curve of the *reward per episode* function. In the following figure (cf. 2.5), two graphs are plot:

- The blue graph represents the score of the agent at the end of each episode.

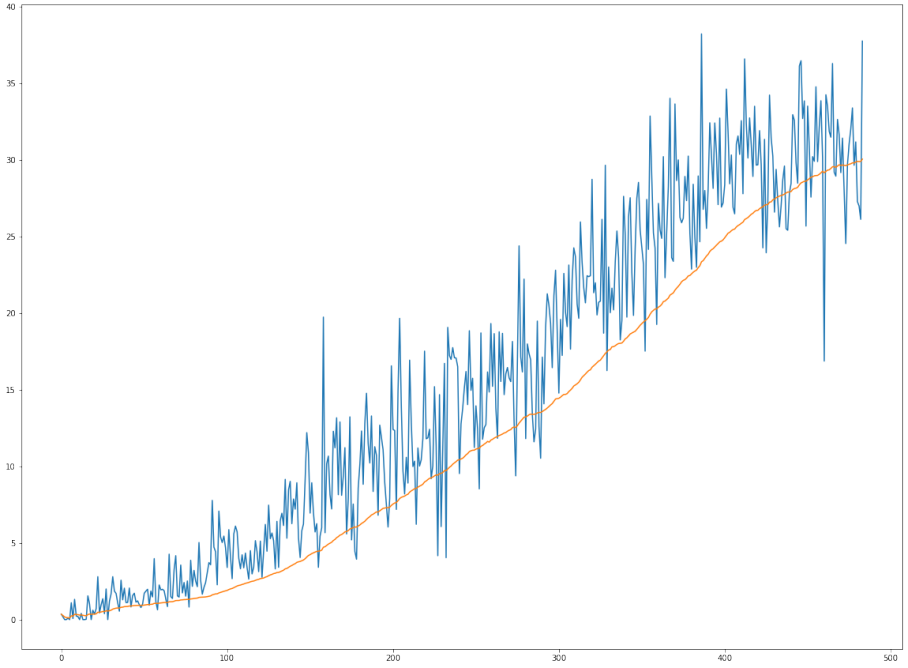

FIGURE 2.5: DDPG architecture training performance with uniform replay buffer

- The orange graph represents the moving average score of the agent. It better represents the overall learning behaviour of the agent.

Our learning was pretty stable throughout the whole process, with occasional drop from time to time. However, the moving average perfectly describes the learning tendency followed by our agent. The task was solved in 384 episodes.

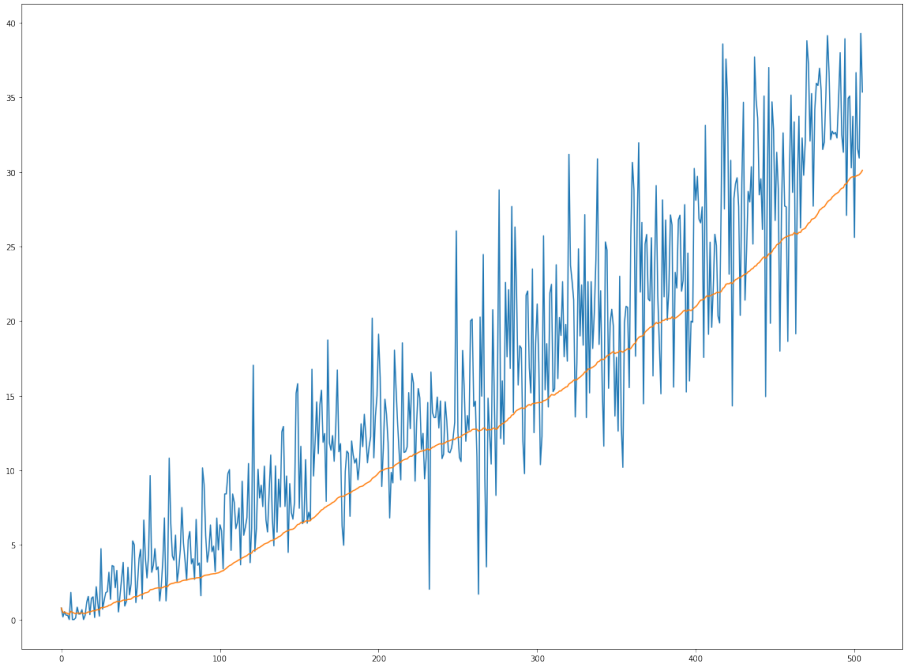

FIGURE 2.6: Prioritised DDPG architecture training performance

We see (in 2.6) that my implementation of the prioritised replay buffer gives a more steady moving average over every run ! However, the task was solved in 406 episodes, almost the same amount than with an usual replay buffer but still a bit more.

Chapter 3

Conclusion

3.1 Summary

As a summary, two methods were studied and used in this project: while sensibly similar performances were obtained, no thorough conclusion can be drawn with regards to their relative performance. Indeed, as not a lot of hyper parameter exploration has been done, we can only base ourselves on these limited results.
However, we see a steadier progression for the prioritised replay buffer: we could then hypothesise that if we were to keep on training our model, it would keep on improving. However, the usual DDPG architecture seems to have started to hit some kind of a learning plateau. We could then hypothesise a better performing algorithm, at term, for the prioritised DPPG architecture.

3.2 Overture

We solved the option one of the environment, with a single agent. However, with a total of 20 agents gathering information, one may think that the performance of training would significantly improve as we train a single model with the experience of every agent. Sharing this parallel experience could show itself highly useful for such algorithms.

However, this idea is pure assumptions and is yet to have been studied by myself.

Bibliography