Udacity

Project Report

P3 - Collaboration Competition

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, 'P3 - Collaboration Competition', 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: _____

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem diagnosis

In the context of the *Tennis* environment, two agents move rackets on a tennis field to bounce a ball over a net, delimiting their respective areas. The goal of the agents is to keep the ball up in the air (not to score a goal by making it fall in the adversary field). This environment 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#tennis

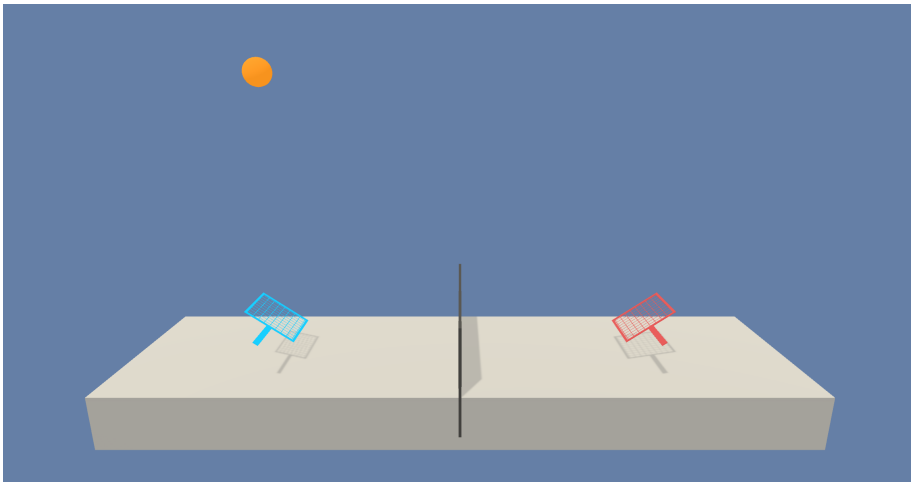

FIGURE 1.1: Screenshot of the agents' environment

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 24. Amongst all the variables making up this vector, we can count the position of the ball, the velocity of the racket.

```
        [ 0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.         -6.65278625 -1.5
 -0.          0.          6.83172083  6.         -0.          0.        ]
```

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 where the an agent hit the ball in the air, over the net. If the ball hits the ground or is thrown out of bound, it receives a -0.01 reward. The reward of the actions is common between the agents, to make them collaborate.

1.2.3 Actions

To collect these rewards, our agent can move itself. These movement are represented as 2 vector of 2 values, with each value being between -1 and 1:

```
        [
            [0.06428211, 1.        ],
            [0.06536206, 1.        ]
        ]
```

FIGURE 1.3: Agent action example

This two-sized array represent the agent actions, for each agent:

- First item is the speed of the racket;
- Second item is whether to jump or not.

The first vector is the action for the first agent, the second vector for the second agent.

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 **+0.5** over 100 consecutive episodes (i.e. that the ball has been, on average, thrown back 5 times between the agents).

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/P3-Collaborative`

Chapter 2

Models & Results

2.1 Retained models

Our multi-agents system was trained during this work:

- A DDPG [1] Agent that implements **experience replay** using the a uniformly sampled replay buffer.

2.1.1 Presentation of Vanilla DDPG

2.1.1.1 Hyper-Parameters and formula presentation

Adaption of the famous Actor-Critic methods, the Multi-Agent DDPG algorithm is ideal for our task where we need our agents to collaborate. 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. The **Actor**, outputting the probability of taking any action given the state and the policy;

[1] Multi-Agent Deep Deterministic Policy Gradient

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 Multi-Agent 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 MADDPG algorithm uses **soft updates** and mix in 0.3% of the regular weights in the target weights at **every time step**;
- each agent's critic is trained using the observations and actions from all the agents;
- each agent's actor is trained using just its own observations.

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

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

- Adam optimiser with a Learning Rate of 3*10^{-3};
- A batch size of 256;
- Soft update rate $\tau = 3 * 10^{-3}$;

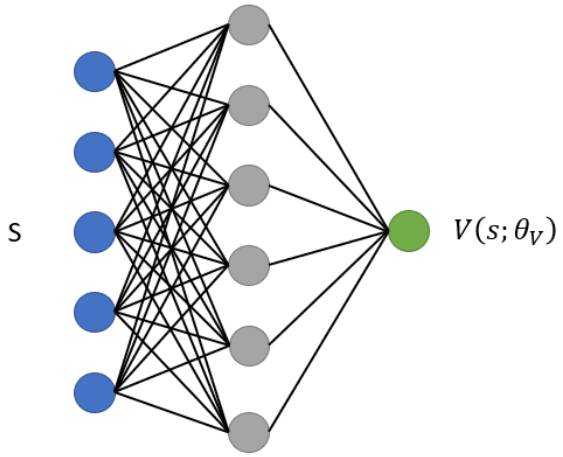

FIGURE 2.1: Schematic representation of a critic network

- Delayed update: the networks are updated only after 300 episodes are run. After this point, at every time-step, 3 updates are run;
- Epsilon value for noise addition is set at $\epsilon = 1.0$ at the start of training and get dropped after the 300 episodes are run using the following formula: $\epsilon = 0.999^{c-d}$ where c is the current episode count and d is the learning delay of 300 episodes;
- 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 256 as a batch size).

The Actor network The actor network was trained with the following hyper-parameters:

- Adam optimiser with a Learning Rate of 10^{-3};
- A batch size of 256;
- Soft update rate $\tau = 3 * 10^{-3}$;

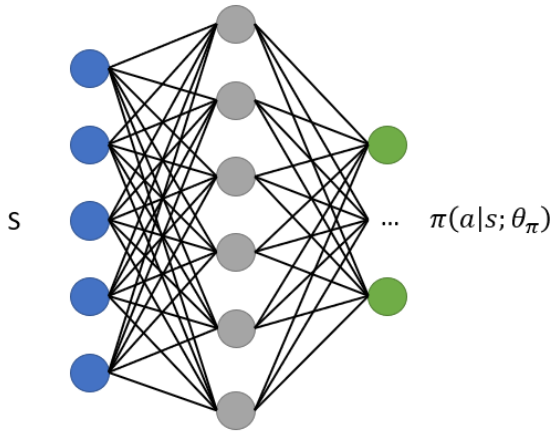

FIGURE 2.2: Schematic representation of an actor network

- Delayed update: the networks are updated only after 300 episodes are run. After this point, at every time-step, 3 updates are run;
- Epsilon value for noise addition is set at $\epsilon = 1.0$ at the start of training and get dropped after the 300 episodes are run using the following formula: $\epsilon = 0.999^{c-d}$ where c is the current episode count and d is the learning delay of 300 episodes;
- 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.

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 DDPG agent and the MADDPG wrapper. Uses soft update (for weight transfer between the local and

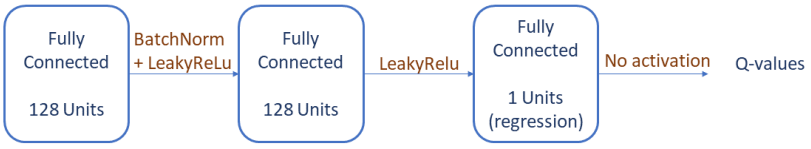

FIGURE 2.3: Critic Model Architecture

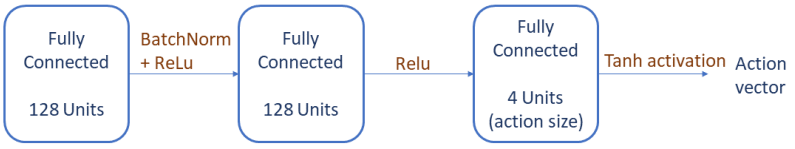

FIGURE 2.4: Actor Model Architecture

target networks) as well as a uniformly distributed replay buffer and a standardised Normal Random Variable 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;
- *DDPG.ipynb*: Training of the MADDPG agent;

2.1.2 Addition of Multi-Agent to the Vanilla DDPG algorithm

The addition of the Multi-Agent concept to the DDPG agent is materialised in the code behaviour by a couple of things:

- The shared Replay Buffer represents the common training of the critic networks using observations from **all the agents**;
- A wrapper class (here the `MADDPG` class) instantiates the DPPG Agent classes, calls the learning methods, sample the experiences and distributes learning accordingly;
- Reward is **common** between our agent: this concept is crucial to model collaboration work.

2.2 Results

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

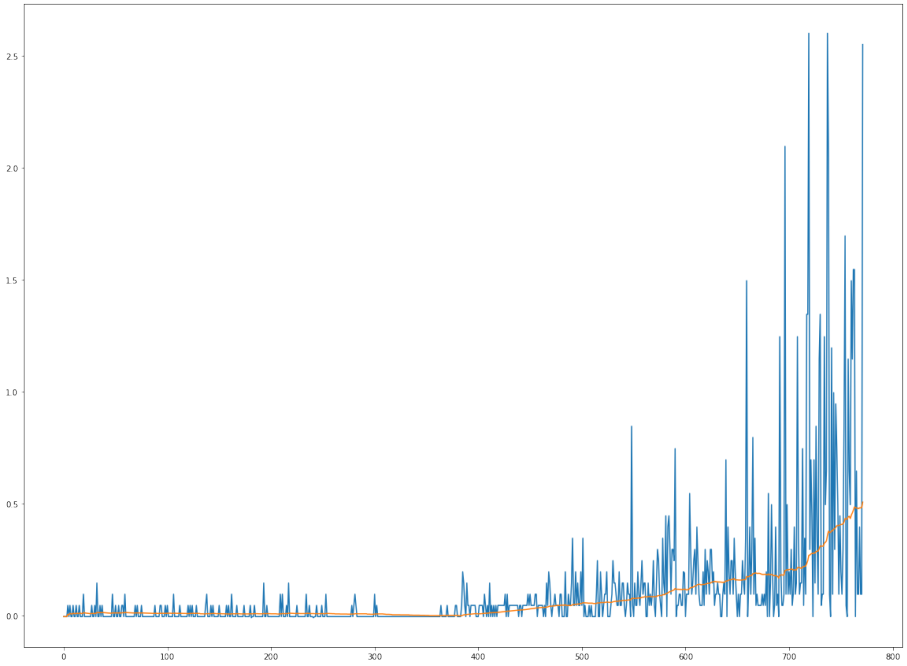

FIGURE 2.5: MADDPG architecture training performance with uniform replay buffer

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

Chapter 3

Conclusion

3.1 Summary

As a summary, the MADDPG method was studied and we saw how sharing the replay buffer between agents so that they learn from common experiences while keeping their decision making their own by having private and distinct actor models was efficient at training collaborative multi-agents. A very good analysis could be to try this same algorithm on other tasks where collaboration and competition would be in a form of balance.

3.2 Overture

We solved the option one of the environment, with an agent using uniformly sample replay buffer. However, I have noticed in the other projects how the use of a prioritised replay buffer may help either to train faster or to have more stable training. Hence, an area of improvement for my code would be to use a replay buffer. However, this idea is pure assumptions and is yet to have been studied by myself.

Bibliography