Multi Agent DDPG

Description

As described in the README.md, the current project solves a continuous control task, the Unity Tennis environment. In order to solve this problem, I have used the Multi Agent Deep Deterministic Policy Gradient (MADDPG) algorithm [1].

The project is structured in 5 scripts:

- **main.py.** It is the main script from which we would select the hyperparameters and we will run the code.
- **models.py**. Implements the Actor-Critic neural network architectures.
- **maddpg_agent.py**. Provides all the necessary code to execute an agent following the MADDPG approach.
- **utils.py**. Implements utilities that may be used by the agent, such as replay buffers and normal noise generator to get actions.
- **plot_utils.py**. Implements helpful functions to plot the obtained scores.

In order to run the project, we would only have to select the desired hyperparameters and set maximum number of episodes (the training will finish either all the maximum number of episodes have been reached or the goal has been achieved).

Once the train process has finished, the weights of the actors are exported in order to be able to use them at execution/evaluation time (for this part the critic is not required, but it's weights are also saved if needed). This could be done in the main by selecting *evaluation* in the mode attribute.

***By default, the weights of the models and the scores are exported (into a folder that is automatically generated) inside the results folder. Inside each test_X folder could be found:*

- *Both actors and critic weights at the moment the environment has been solved*
- *Both actors and critic weights after achieving a higher performance*
- *A hyperparamets.md where it could be seen the used hyperparams*
- *A pickle file with all the necessary information about the scores:*
    - *Scores(list): with the agent that has more score in each episode*
    - *Score1(list): agent 1 obtained scores*
    - *Score2(list): agent 2 obtained scores*
    - *Checkpoint(int): when has been achieved the goal*
    - *Checkpoint_best(int): when has been achieved the best performance*

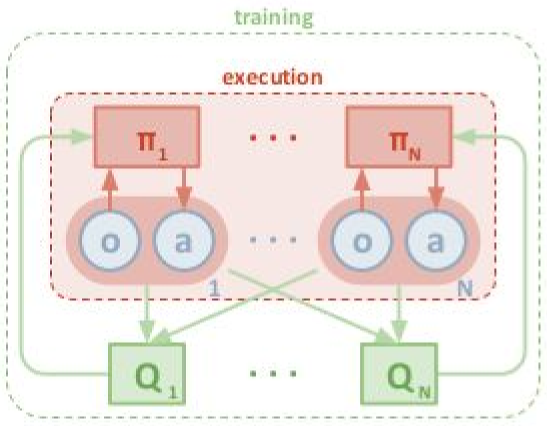
In the repository you would find a plot_utils file, which will help with the visualization of the scores if needed (I encourage you to use the analyzeSingleRun function).

Learning Algorithm

The algorithm we are implementing is known as MADDPG [1], and in this section I am going to describe it briefly.

Traditional reinforcement learning algorithms are not designed to work in a multi-agent environment. These approaches assume that the environment is stationary. However, in a multi agent environment each agent's policy changes as the training progresses, which yields to have a non-stationary view from the perspective of each agent.
Because of that, the stability during learning is a big challenge for value-based approaches such as DQN (where experience replay could not be used) and policy-gradient methods (high variance is its characteristic, and these increases when multiple agents appear into scene).

In order to overcome this problems, [1] proposes a framework which consists of centralized training and a decentralized execution using an actor-critic approach, meaning that the policies would be able to use extra information during the training to ease its process, but they would not have this information during their execution time.



The gradient of the expected return for each agent (i) could be written:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^{\boldsymbol{\mu}}, a_i \sim \boldsymbol{\pi}_i}[\nabla_{\theta_i} \log \boldsymbol{\pi}_i(a_i|o_i) Q_i^{\boldsymbol{\pi}}(\mathbf{x}, a_1, ..., a_N)].$$

And the centralized action-value in the following way:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{\mathbf{x},a,r,\mathbf{x}'}[(Q_i^{\boldsymbol{\mu}}(\mathbf{x}, a_1, \ldots, a_N) - y)^2], \quad y = r_i + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}', a_1', \ldots, a_N')\big|_{a_j' = \boldsymbol{\mu}_j'(o_j)}$$

As you may expect, the action-value function (Q) is part of the critic and it takes as inputs:

- X: state related information (usually consist of all the observations of the agents).
- Action values: the actions of all the agents.

The typical approach (the one adopted in the paper) is to have one critic per actor so that each action-value is used independently by each actor, making it possible to have different reward structures between agents (including configurations at competitive environments). However, for our current project we are going to use a single critic for our two actors, as they share the same type of information and reward structures.

Furthermore, the expected gradient for our problem has to be updated respect to the one exposed before, as we are not working with discrete action spaces; we work with continuous actions.

$$\nabla_{\theta_i} J(\boldsymbol{\mu}_i) = \mathbb{E}_{\mathbf{x},a\sim\mathcal{D}}[\nabla_{\theta_i}\boldsymbol{\mu}_i(a_i|o_i)\nabla_{a_i}Q_i^{\boldsymbol{\mu}}(\mathbf{x},a_1,...,a_N)|_{a_i=\boldsymbol{\mu}_i(o_i)}]$$

The main difference between the first and the latter actor update formulation is that the first samples its state and actions from the actor in an on-policy way; while in the latter one (the one we are going to use) we sample our state and actions from a replay buffer (D) as in off-policy approaches (see DPG original paper for better understanding).

Hyperparameters

parameter	value
GAMMA	0.99
TAU	{**0.01**, 0.001}
BUFFER_SIZE	100,000
BATCH_SIZE	{**128**,256}
LR_ACTOR	0.001
LR_CRITIC	0.001
POLICY_UPDATE	{**1**,2}
EPISODES	3000

The hyperparameters that have multiple values are the ones I have played most with and I got the best results (with different parameter combinations). In bold the ones that I would use.

I want to stand out the importance of selecting a small buffer_size (1e5 instead of 1e6 as I have used in other projects). It seems that if we get a big replay buffer, when sampling from it the probability of selecting good samples is smaller. Thus, the learning process, once achieved a level of learning for any of the agents, it requires new samples according to its new skill level.

Moreover, a smaller tau seems to get better results, as the target networks are updated too slow for the current environment behaviour.

Neural Network

Following the approach stated in DDPG and adapting it based on the explanations of MADDPG, we have built an actor-critic architecture composed of 2 actors and 1 critic.

Critic

As stated in the [1], the main extension from DDPG to MADDPG is that the critic augment its capacity by collecting extra information about the policies of the other agents.

The main approach says to use a unique centralized action-value function per each actor. However, this is done because agents could have different reward structures, but in the case of the current project both actors have the same reward architecture. Thus, we are going to use a common critic for both actors.

Independently of using 1 or many critics, the network architecture is gonna be the same and it is going to be completed by:

- X → state information of all the agents (o1,o2,...on)
- A → action values of all the agents (i.e: for the case of 2 actions per agent: a11,a12,a21,a22....a1n,a2n)

For this project we would have 2 identical agents, which could take 2 continuous actions and with a state space equal to 24. Hence, the critic would have to process:

- X: (o1,o2)
    - Each of those observations with a size of 24.
- A: (a11, a12, a21, a22)
    - Each action could take values between {-1,1}

These yields in a total dimension of 52 parameters (2*24 + 2*2). In our case, instead of feeding the first input layer directly with all the parameters, we would first input the state parameters (48) and then add the output (128 neurons) with the action parameters (4).

```
DDPG_Critic(
  (input_layer): Linear(in_features=48, out_features=128, bias=True)
  (hidden_layers): ModuleList(
        (0): Linear(in_features=132, out_features=128, bias=True)
  )
  (output_layer): Linear(in_features=128, out_features=1, bias=True)
)
```

All the layers use ReLu as activation function, except the output layer that does not use anything.

Actor

The actor receives the state and outputs the 2 possible action outcomes. As the actions are continuous and have to be between {-1,1}, we use a Hyperbolic Tangent function for the output layer.

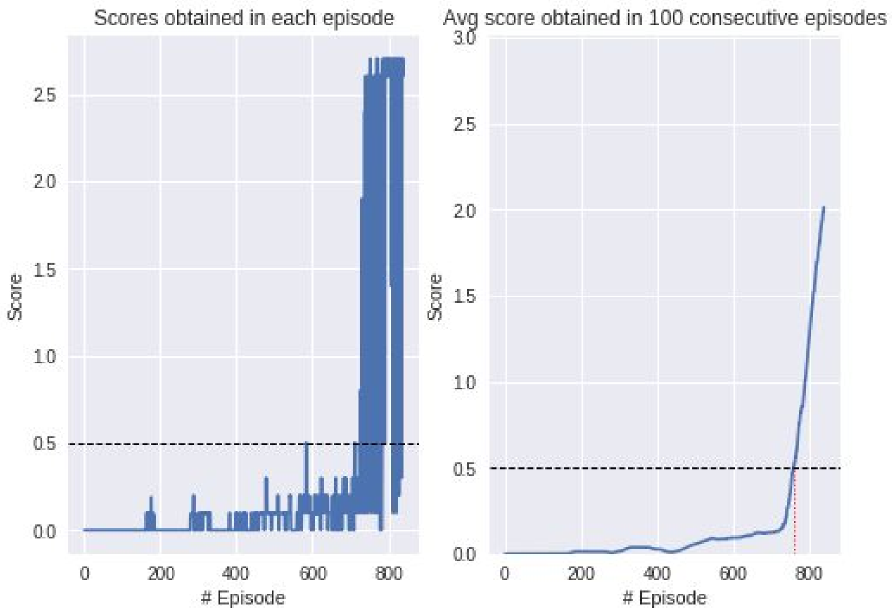
```
DDPG_Actor(
  (input_layer): Linear(in_features=24, out_features=128, bias=True)
  (hidden_layers): ModuleList(
        (0): Linear(in_features=128, out_features=128, bias=True)
  )
  (output_layer): Linear(in_features=128, out_features=2, bias=True)
 )
```

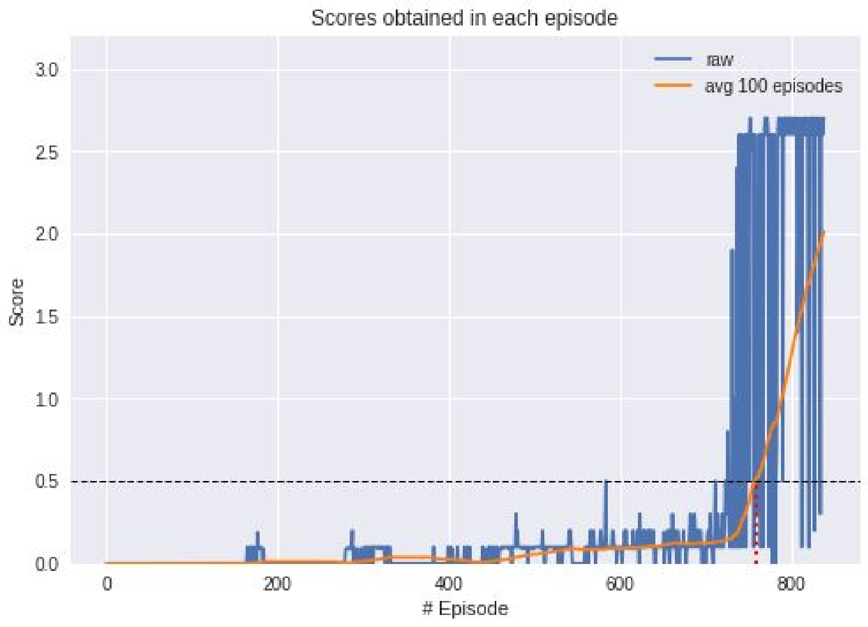
As stated before, we would use two identical networks. The unique reason for creating two different networks instead of a unique one, is their initialization (they would have two different seeds) and that both networks would be independent. Taking those away, we could use a single actor for both agents, as their behaviour should be the same.

Plot of Rewards

The problem has been solved many times at different runtime instances and with different hyperparameters. All the obtained results could be found alongside its weights in the **/results** folder.

A lot of tests get good results and achieve the expected goal. Just to demonstrate the obtained results, we are going to attach here the plot of rewards of the test_1.

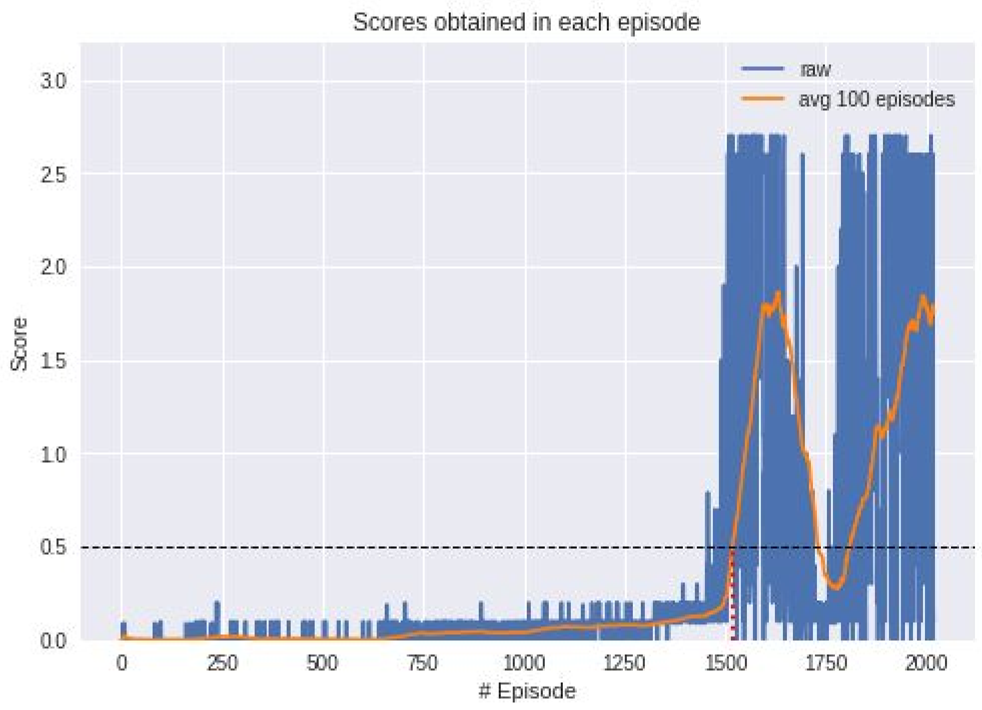




For this case, the problem seems to be solved after 758 episodes of training, this is, after 658 episodes!

Moreover, if we keep the learning process a little bit further, our agents achieve a better average score after 800 episodes, that is nearly 2.0.

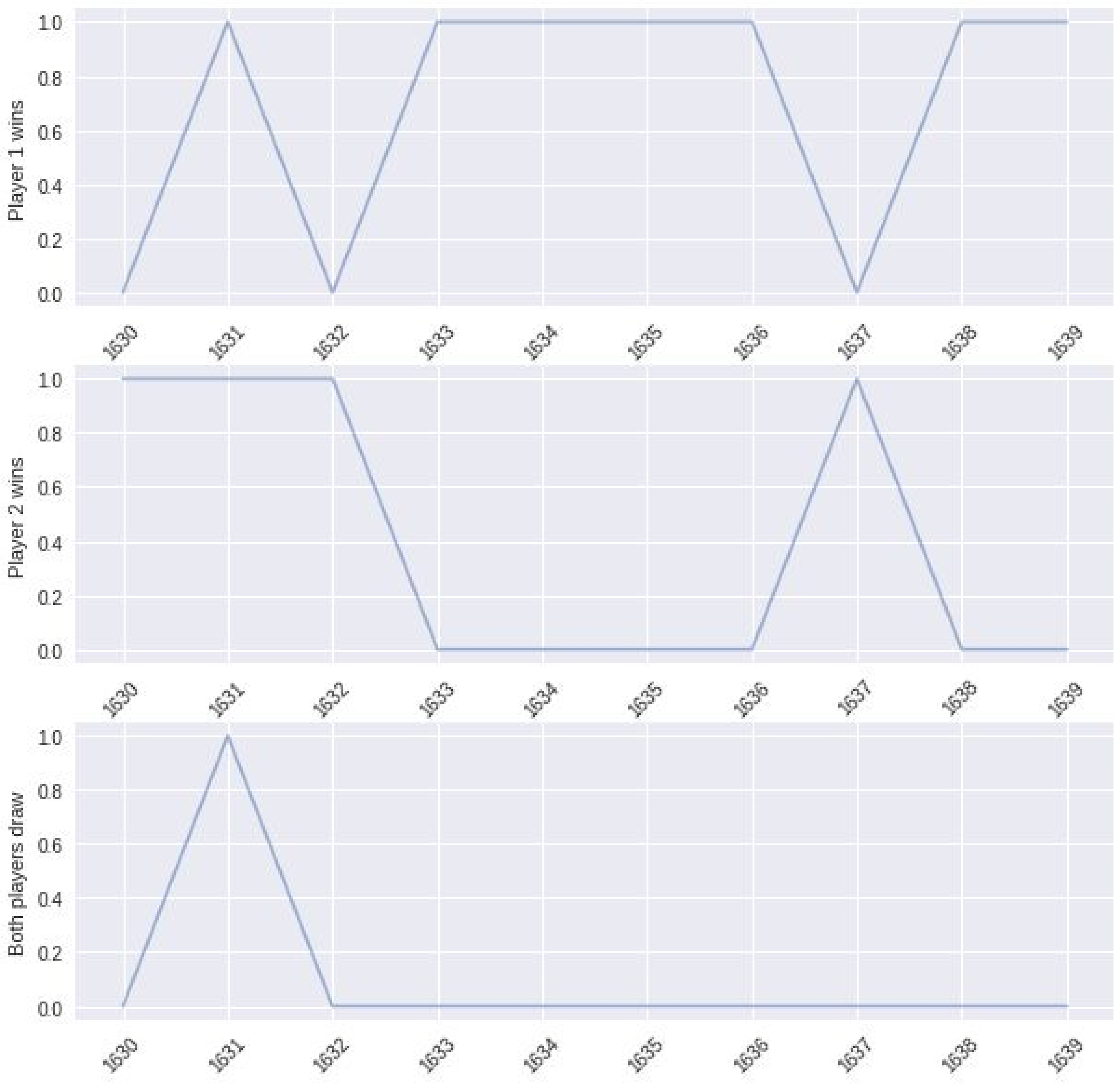
However, just looking at this test we could think that our agent will keep increasing its success rate with more episodes, but this is not true. This could be seen in the results obtained in test_2, where after solving the environment in 1518 episodes of training and achieve an average score of 1.8, its performance decrease:



This might be because the episodes are limited to a certain amount of visits (1000) and the goal of the agents is to accumulate as much reward as possible. Once the goal is achieved and the correct weights are found out, the agent could not get much better as it has reached its top. If we continue training our agents, the weight changes have to be minimal as a big step/change in those values could lead into big differences (remember that the ACTOR is a policy-based network with very high variance). Moreover, if one of the actors begins to perform worse, it would affect our global score although the other agent keeps performing as its best.

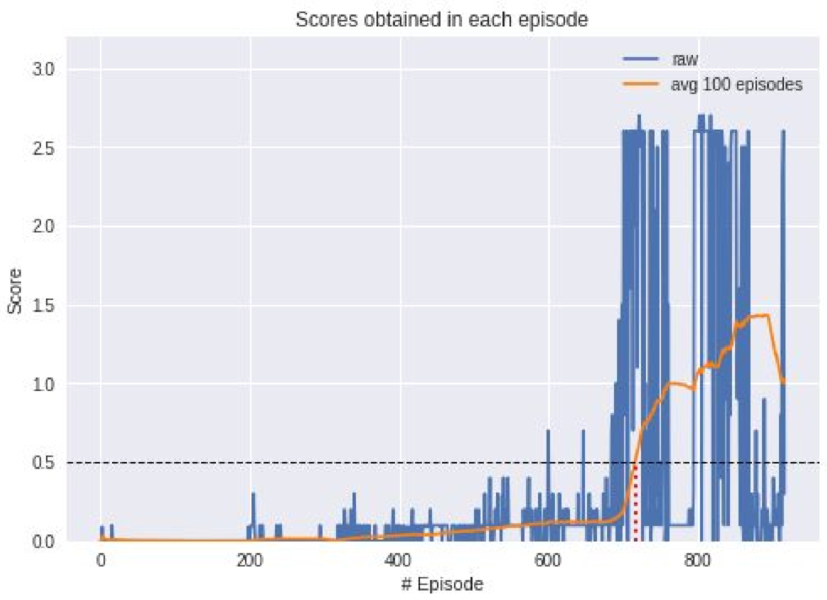
Analyzing why performance decrease

Following the results obtained in test_2, we could see that the issue has something to do with the scores obtained by each agent independently:

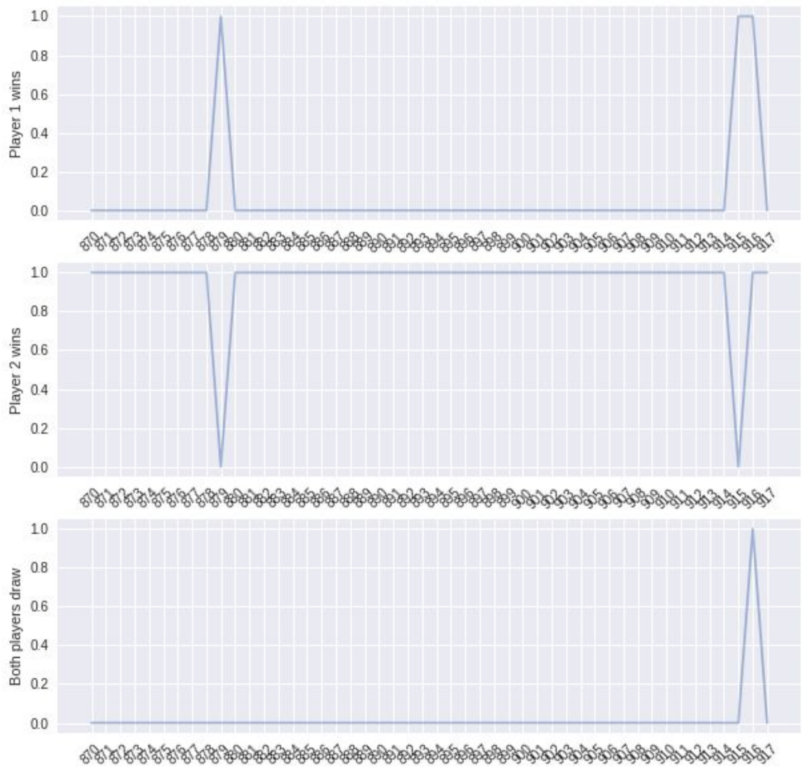


In the above graph, we plot the scores obtained in the interval in which the global score begins to decrease drastically (1630-1640), exactly after the 1632 episode. It could be seen that this happens when the player 2 fails some consecutive episodes.

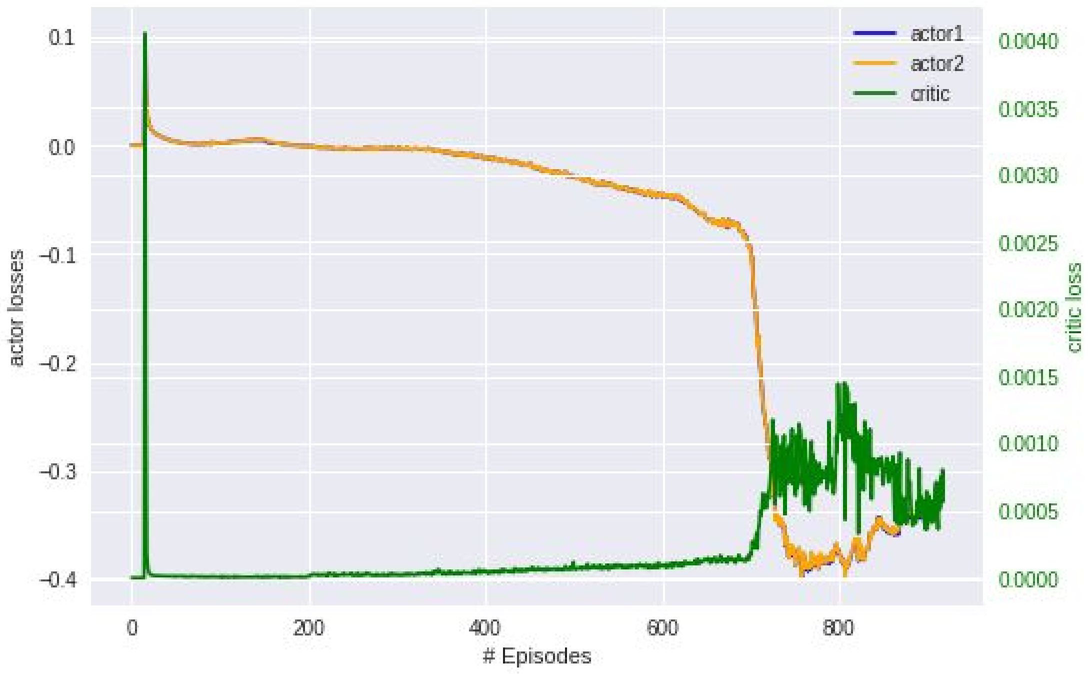
This could be seen also in the test_4 (checkpoint: episode 717), where after obtaining a score close to 1.5 in avg, it decreases its performance after episode 895:



If we plot the scores obtained by each agent from episode 870 in advance, we could see that the player1 begins to lose a lot of games, and the global score performance begins to decrease drastically after episode 895.



Moreover, we could draw the losses obtained in each network in every episode in order to see what has happened.



The actor losses are very similar in both cases. Their values are around 0 and -0.1 in the initial 700 episodes, but after that they decrease until reaching -0.4, which happens when our agents begin to perform as their best (scoring +2.5).

This also could be seen in the critic, where its losses increased around this interval too (from 0.0000 to around 0.001). As the actor losses are subject to the critic output, we could say that the agents begin to learn when the critic improves its performance. Moreover, as the error at the critic holds at that level and does not return to its baseline, it keeps noise in the actor losses, which finally yields to instabilities in the learning process.

Ideas for future work

Try to overcome the performance decrease of the actors, trying to keep them always at the top.

Implement other Multi Agent algorithms such as MA-PPO to compare with the obtained results.

Adapt this code to other environments would be challenging and would spread the knowledge. Thus, trying to solve any problem with discrete action spaces would be a good idea, as a lot of current problems work with them and it would require changing this approach based on DDPG, which only is usable with continuous action spaces.

Finally, adapt the code in order to use Prioritized Experience Replay in order to use the collected experience samples in a correct way.

Bibliography

[1] Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments
[2] Continuous control with deep reinforcement learning