Competition and Collaboration Project - Tennis Players

1 Framework

1.1 Environment objective

There are 2 agents that control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

So under this framework the agents will try to keep the ball on the court as long as possible to maximize the possible rewards,id est, strictly speaking this task is a competitive task because the agents have to learn how to play along the other in harmonious manner so both can benefit, otherwise in a competitive environment they would have to learn how to beat the other agent or learn how to make the other fail. To tackle this task a multi-agent version of the Deep Deterministic Policy Gradient (DDPG) is used.

1.2 DDPG

In the previous project I added a briefly explanation and the pseudo code for DDPG [2], refer to section 1.2 in case you need a refresh.

2 DDPG implementation

The agents are trained using a common replay buffer that feeds a single pair of Actor-Critic networks, so a single policy is used by both agents. This can lead that the agents could have very similar styles of gaming. To avoid this situation a separated replay buffer could be used for each agent and use different networks to train them, to keep it simple in this exercise I went for the first approach.

At each step the environment returns the rewards and next states for each agent these are stored along with previous states and the taken actions into the replay buffer, the agent keep taking actions and repeating this process until after a certain number of steps, then a sample batch is taken from the buffer and the learning process starts. After compute the updates for each network(actor and critic) the weights are propagated to the agents.

2.1 Hyper-parameters

Hyper-parameters		
Parameter	Value	Description
BUFFER_SIZE	100,000	replay buffer size
BATCH_SIZE	128	minibatch size
GAMMA	0.99	discount factor used in TD targets
TAU	.001	for soft update of target parameters
LR_ACTOR	.003	learning rate used by the optimizer algorithm of the actor neural network
LR_CRITIC	.003	learning rate used by the optimizer algorithm of the critic neural network
UPDATE_EVERY	1	how step are needed to update the networks
NUMBER_UPDATES	1	number of time the networks are updated in each leaning phase
HIDDEN_LAYER_1	48	number of nodes for the first fully connected layer in critic and actor's network
HIDDEN_LAYER_2	24	number of nodes for the second fully connected layer in critic and actor's network
HIDDEN_LAYER_3	12	number of nodes for the second fully connected layer in critic and actor's network

2.1.1 Network's architecture

Actor Network		
Layers	Nodes	Activation function
Input layer	24	Leaky Rectifier Linear Unit
First hidden layer	48	Leaky Rectifier Linear Unit
Second hidden layer	24	Leaky Rectifier Linear Unit
Third hidden layer	12	Hyperbolic Tangent
Output layer	2	NA

To the output of the actor network the noise coming from an Ornstein–Uhlenbeck process is added and then clipped between -1 and +1. Another

Critic Network		
Layers	Nodes	Activation function
Input layer	24	Leaky Rectifier Linear Unit
First hidden layer	50	Leaky Rectifier Linear Unit
Second hidden layer	24	Leaky Rectifier Linear Unit
Third hidden layer	12	Identity
Output layer	1	NA

A thing to notice is that after feeding the input layer of the critic-network with states, the output of that layer and the actions coming from the output of the actor-network are concatenated to finally feed the second layer of the critic-network. Also although the observation space consists of 8 variables, the observation returned by the environment are 24 features vectors for each agent, this because is stacking the 3 observations frames that will comprise an experience-

2.1.2 Noise

To each action suggested by the actor-network the algorithm noise which follows the next equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

This stochastic process is known as Ornstein–Uhlenbeck and the W_t associated process(the Wiener process) can be seen as the limit of a random walk, so the sample taken from it should be modeled as $Normal(0, 1)$ to avoid add a positive bias, which it happens when sample from a uniform distribution given that actions must lay in $[-1, 1]$.

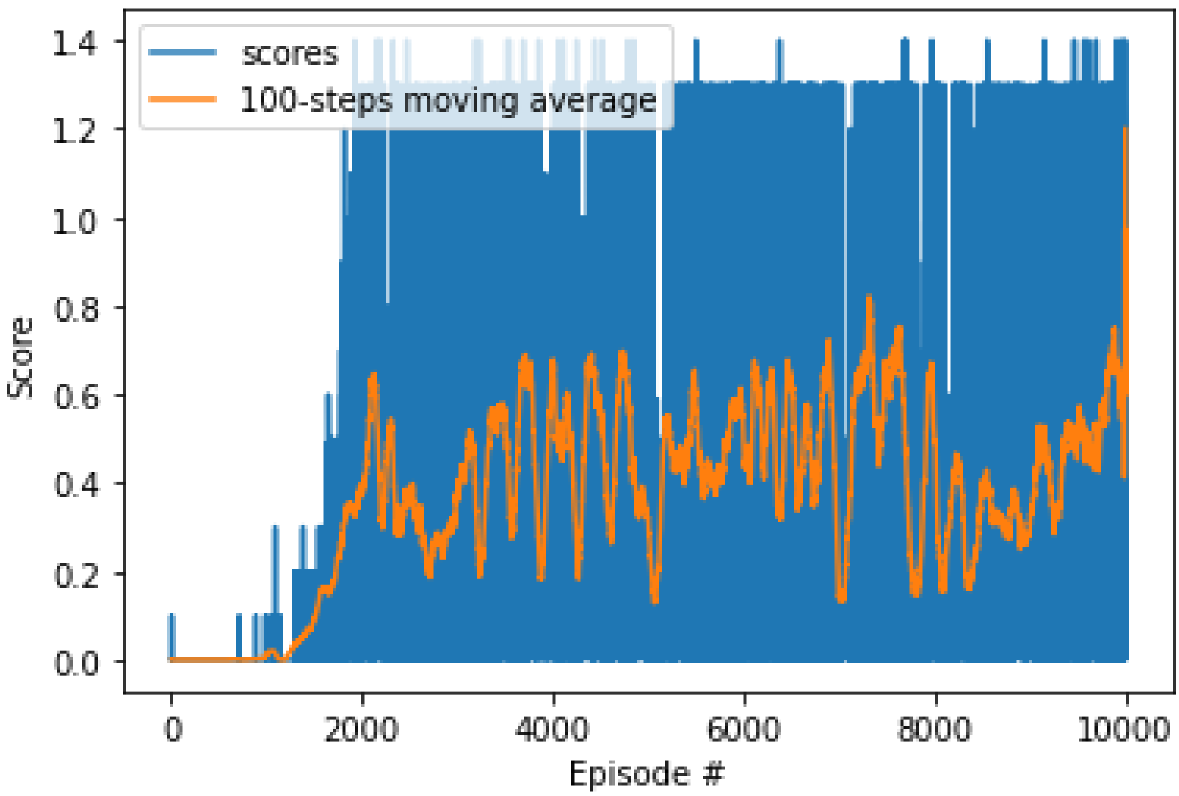
3 Results

To train the agents 10000 episodes were run (with a maximum of 500 steps per episode) and the weights for the first time the agent gets at least a .5 average over 100 episode are saved, and also the weights for the best performance achieved. Following it can be observed how the agents perform through the training phase, their average performance over a window of 100 hundreds episode every 100 episodes.

DDPG Agents	
Episode	Average Score
100	0.0010
200	0.0000
300	0.0000
400	0.0000
500	0.0000
600	0.0000
700	0.0000
800	0.0010
900	0.0010
1000	0.0020
1100	0.0126
1200	0.0090
1300	0.0030
1400	0.0370
1500	0.0593
1600	0.0929
1700	0.1632
1800	0.1746
1900	0.2832
2000	0.3427
2100	0.3968
2200	0.6144
2300	0.3170
2400	0.5343
2500	0.2909
2600	0.3849
2700	0.3305
2800	0.1928
2900	0.2736
3000	0.2825
3100	0.3615
3200	0.4617
3300	0.3247
3400	0.3966
3500	0.5244
3600	0.5384
3700	0.3546
3800	0.6707
3900	0.5064
4000	0.2175
4100	0.6790
4200	0.4580
4300	0.4811
4400	0.3181
4500	0.6747
4600	0.5527
4700	0.2850
4800	0.6305
4900	0.5677
5000	0.3194

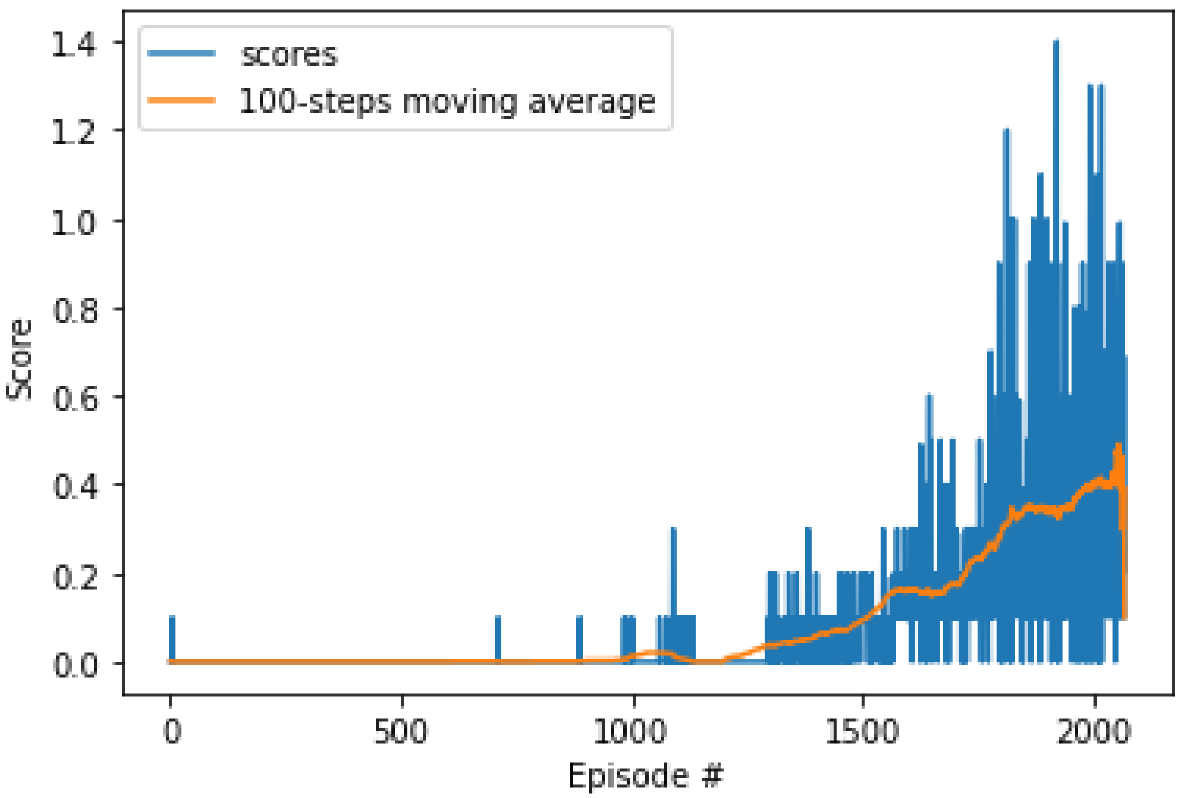
DDPG Agents	
Episode	Average Score
5100	0.3071
5200	0.2013
5300	0.5329
5400	0.4677
5500	0.4427
5600	0.5849
5700	0.4048
5800	0.3837
5900	0.4742
6000	0.5598
6100	0.4856
6200	0.6177
6300	0.4319
6400	0.6470
6500	0.3768
6600	0.6149
6700	0.3992
6800	0.4776
6900	0.4997
7000	0.6812
7100	0.1461
7200	0.3194
7300	0.6199
7400	0.7499
7500	0.5188
7600	0.6468
7700	0.6736
7800	0.4714
7900	0.1716
8000	0.4080
8100	0.5024
8200	0.1936
8300	0.4141
8400	0.3186
8500	0.2045
8600	0.4042
8700	0.3108
8800	0.3157
8900	0.3418
9000	0.2602
9100	0.3073
9200	0.4983
9300	0.4376
9400	0.3225
9500	0.5364
9600	0.4916
9700	0.4873
9800	0.5095
9900	0.5755
10000	0.6776

Environment solved in 2169 episodes! Average Score: 0.5080
The 10000 episodes were executed in 199 minutes.

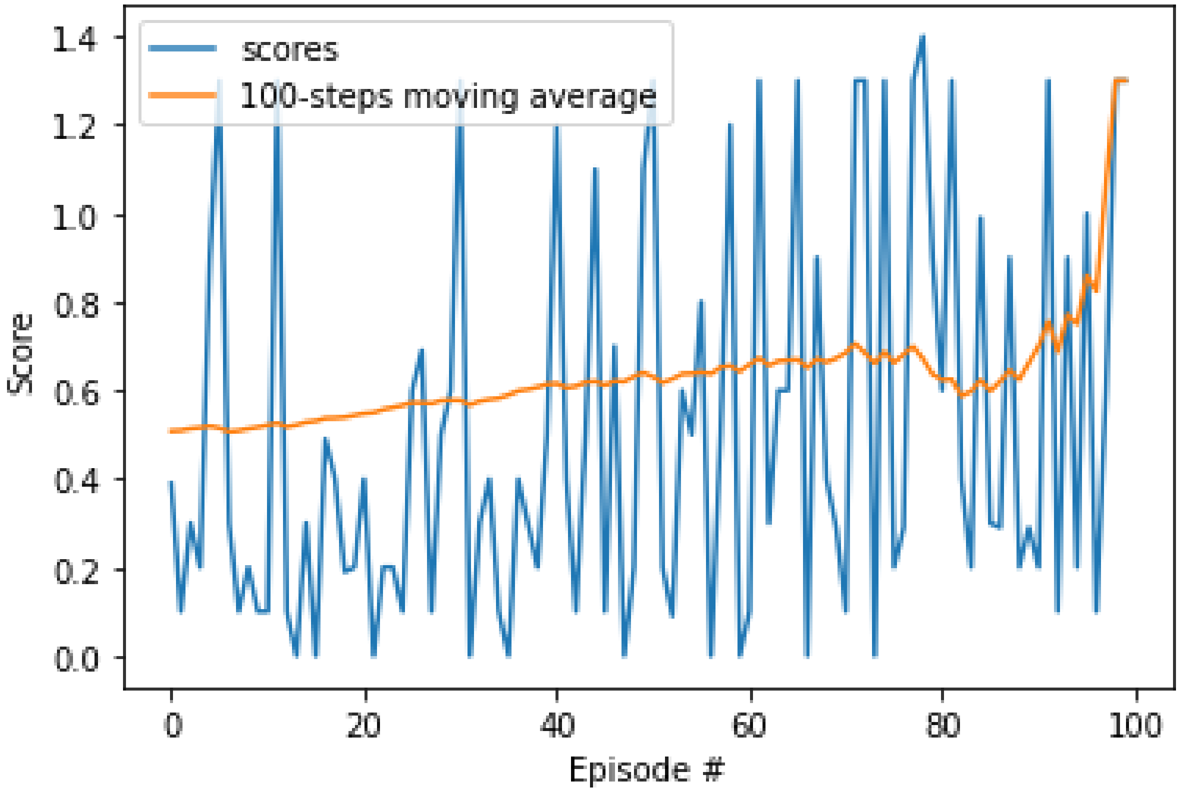


The training keeps improving pretty steady before the first 2000 episodes, at the beginning it needs to collect more data to start to learn something. Sometime moving average goes down but never reaches 0.

3.1 Results until the environment is solved

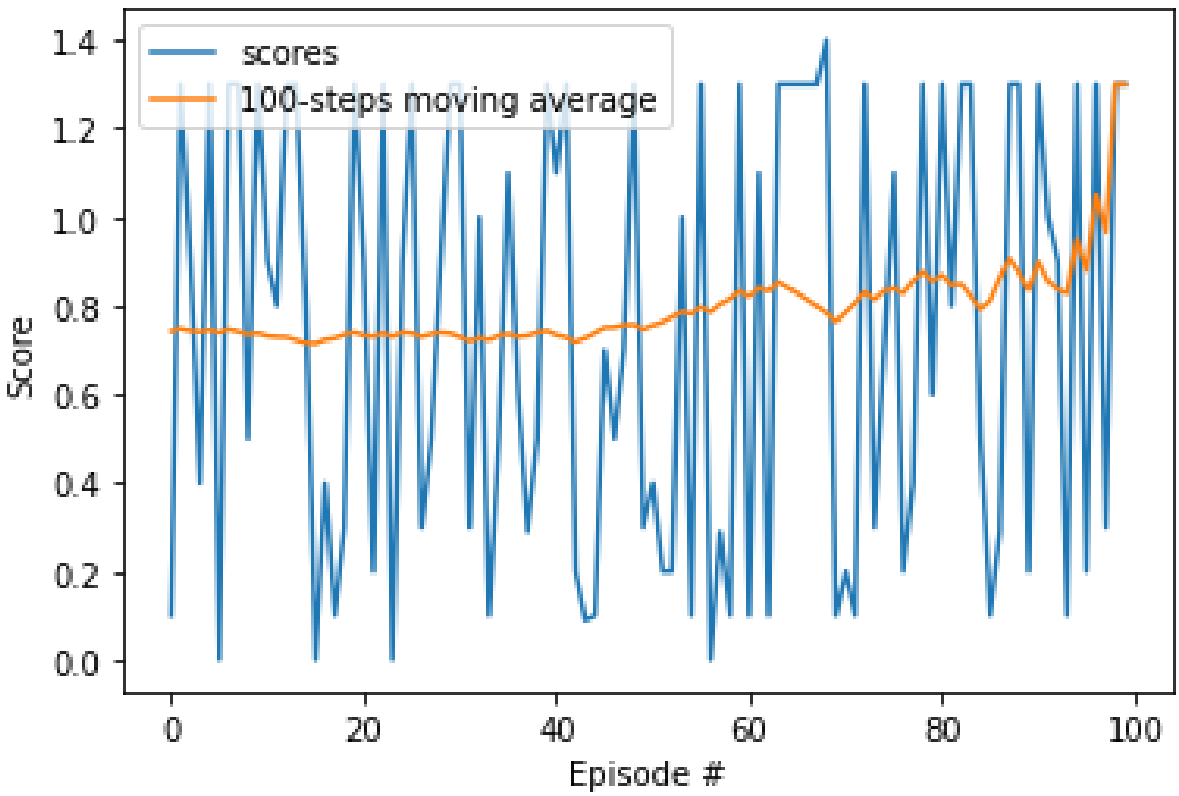


The agents need more data the first 1000 episodes to start to learn something and then training goes upwards pretty steady. The next is a zoom of the 100 episodes that solved the environment, beginning at episode 2070 to the episode 2169.



3.2 Results for the best solution achieved

The results during the training stage are:
The best episode scored: 1.4000000208616257
The best 100 episode average scores: 0.7426000110991299
The best average score over 100 episodes was reached at episode 8481
Graph from the episode 8382 to episode 8481 that got the best average score:



4 Future Work

The following steps would be:

Tackle the challenge for the soccer environment which needs a collaborative-competitive approach not just collaborative, given that in soccer game the teams need to score as much as possible avoiding the other team scores.

Three different ideas to improve or enhance the performance of the model would be:

- In some runs in the training stage, after the environment was solved, the learning went down to zero and it took a while to went back on rails again.So I think by applying a Prioritized Experience Replay Buffer could stabilize the learning process.
- As I mentioned before, use separated replay buffers to store the experiences of each agent. And also use different pairs of actor-critic networks, each pair fed by its respective replay buffer or PER. This approach could create more robust agents that follow different policies.
- I mentioned earlier to implement a PER could be a great idea, so why no go for the full implementation of DPG4[3]? So the next enhancements would have yet to be implemented: enable the use of n-steps bootstraping to compute the time difference targets(TD targets) and finally implement distributional critics network.

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, et al(2016). CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING. Google Deepmind. London, UK.

[2] Myself (2020). Continuous Control Reachers Report. https://github.com/chuquikun/Continuous$_{C}ontrol_{P}roject-Reacher/blob/main/report.pdf$

[3] Gabriel Barth-Maron , Matthew W. Hoffman, et al(2018). DISTRIBUTED DISTRIBUTIONAL DETERMINISTIC POLICY GRADIENTS. Google Deepmind. London, UK.