Project Collaboration and Competition

Luís Melo dos Santos

August 14, 2019

This is the third and last project in a series of projects linked to the *Deep Reinforcement Learning for Enterprise Nanodegree Program* at *Udacity*.

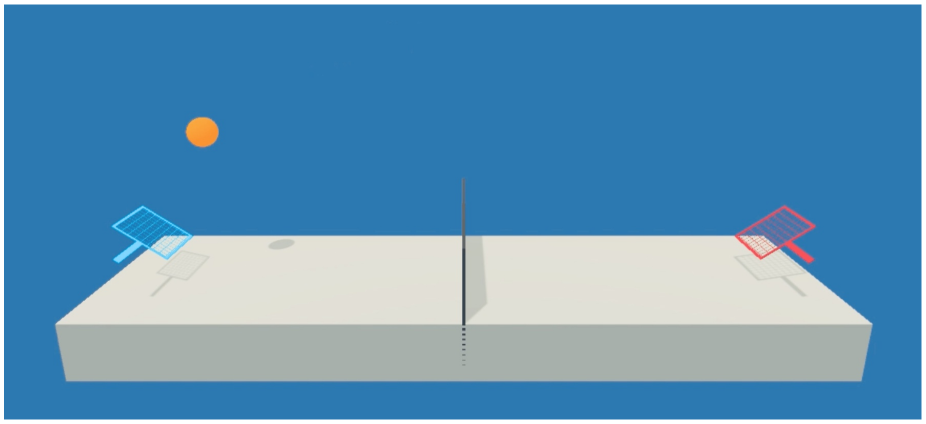
In this project, we aim to train agents in the `Tennis` *Unity* environment.[1]

In this environment two agents control rackets in order 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. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

[1]This environment was tailored by Udacity specifically for this nanodegree, similar to the original `Tennis` environment publicly available in the *Unity* github repository.



- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single *score* for each episode.

The environment is considered solved, when the average (over 100 episodes) of those *scores* is at least +0.5.

1 Learning Algorithms

We started the project by adapting the *Distributed Deep Deterministic Policy Gradient (D3PG)* model developed in the previous Project to this environment. We then enhanced the model by introducing a few improvements in the algorithm – *Noisy Nets* and *Prioritisation Experience Replay (PER).*

We then proceeded by applying the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) model introduced in class to this environment. We ended this Project testing out a few more potential enhancements to the MADDPG model – *Explicit Collaboration Level Setting* and *Energy Expenditure Constraint.*

Next we go through each of these components in more detail.

1.1 D3PG model

We adapted the D3PG model based on Barth-Maron *et al.* article [1] from 2018 to the `Tennis` environment. In our implementation we distributed the same policy μ and value function Q in synchronous between the two agents in the environment. All experiences were shared between the agents as agents updated and learnt from the same memory pool.

The concept of memory pool comes from the DQN model [2]. We continue to use initially a randomly accessed storage of previous experiences - *ReplayBuffer()* - for the learning process, in an attempt to economically maximise information absorption from past experiences and stabilising learning in the process.

For stability, we also keep the concept of local and target neural networks. In the learning process the local neural networks with weights θ learn from experiences but the target neural networks with weights θ' are kept constant until a separate step - *soft_update()* - when θ' gets updated according to the equation,

$$\theta' = \theta' + \tau(\theta - \theta'), \tag{1}$$

where τ is the *soft_update()* learning rate parameter controlling the rate at which new information propagates from the local network through to the target neural network.

The D3PG model just like the DDPG model [3] departs from the DQN model by taking some inspiration from actor-critic models. It tries to both approximate a policy μ and a value function Q with neural networks.

The policy neural network μ,

$$a = \mu(s|\theta^{\mu}) \tag{2}$$

maps from each state s in the environment a single continuous action a using the neural network weights θ^{μ} as the control parameters of the approximation - variables that the learning algorithm will vary in order to improve the approximation.

The Q neural network will have a slightly altered structure from the one in the DQN model in order to accommodate continuous actions. This time around Q,

$$v = Q(s, a|\theta^{Q}) \tag{3}$$

will have both the state of the environment s and action a as inputs and will only provide a real number v as output - the value of the respective state s and action a in the environment. Again, θ^{Q} correspond to the weights of the neural network that we will use to control the quality of the approximation.

The Q-learning in D3PG is similar to the Q-learning in DQN and based on a classical 1-step temporal-difference method – TD(0) – introduced in [4] where a $Q(S, A)$ state-action value function gets updated by ΔQ at each step with new information $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ acquired by the agent as it interacts with the environment,

$$\Delta Q(S_t, A_t) = \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big] \tag{4}$$

S_t is the current state that the agent occupies in the environment, A_t is the action that the agent takes, R_{t+1} is the reward that the agent receives from the environment, S_{t+1} is the state where the agent ends up after taking the action and A_{t+1} is the next action the agent will take at state S_{t+1}. The remaining two parameters in the *Q-update* equation (4) are,

- γ – the discount rate of future rewards;
- α – the learning rate.

γ dictates how eager our agent is to defer rewards and α controls the speed at which the new information transforms the Q function.

In the context of the D3PG model equation (4) takes the form,

$$\Delta Q(S_t, A_t|\theta^{Q}) = \alpha\big[R_{t+1} + \gamma Q'(S_{t+1}, \mu'(S_t|\theta^{\mu'})|\theta^{Q'}) - Q(S_t, A_t|\theta^{Q})\big] \tag{5}$$

where prime refers to the target versions of both policy μ and value Q neural networks.

A_t will be determined not only by the policy network μ but also by a noise factor $\mathcal{N}$ during the learning process,

$$A_t = \mu(S_t|\theta^{\mu}) + \mathcal{N} \tag{6}$$

so the agent can keep exploring, making D3PG an off-policy learning algorithm.

In the original paper [1] the noise factor $\mathcal{N}$ was chosen to be a random variable from a standard normal distribution,

$$\mathcal{N} = \epsilon_0 \cdot (1/2)^{n/\epsilon_{HL}} \cdot N(0,1) \tag{7}$$

We have two noise hyperparmeters: ϵ_0 determining the initial level of noise and ϵ_{HL} determining the half-life of the noise in relation to the number of episodes n experienced in the learning session.

The update of the value function Q in equation (5) can be codified as a gradient descent step on minimising the loss function L^Q with the following structure,

$$L^Q(\theta^Q) = \frac{1}{N}\sum_i \left[R_{t+1} + \gamma Q'(S_{t+1}, \mu'(S_t|\theta^{\mu'})|\theta^{Q'}) - Q(S_t, A_t|\theta^Q)\right]^2 \tag{8}$$

where N is the number of experiences in each learning batch.

The policy μ update can also be codified as a gradient descent of the loss function L^μ,

$$L^\mu(\theta^\mu) = -\frac{1}{N}\sum_i Q(S_t, \mu(S_t, \theta^\mu)|\theta^Q) \tag{9}$$

making the μ policy the greedy policy solution of the environment.

The neural network structure we chose for both actor μ and critic Q approximations was again based in article [3].

For the policy μ we have used a fully connected neural network with two hidden layers. The input layer has the size of the state space and the output layer has the size of the action space. We used *tanh* as the last layer transformation in order to keep action values inside the interval $[-1,1]$.

For the value function Q we have also used a fully connected neural network with two hidden layers. The input layer has again the size of the state space. It is only in the second hidden layer that we introduce the action values concatenated to the values coming from the first hidden layer. The output layer has only one node with no non-linear transformation performed. We use ReLU functions for any other node transformations and Adam was kept as the optimiser of choice.

1.2 MADDPG model

In this project we have also explored a multi-agent approach to solve the environment. This particular environment can be seen as a *Markov Game* as well as a *Markov Decision Process (MDP)*. A *Markov Game* is a N-agent extension where each agent i has a private observation $O^{(i)}$ of the total state S of the environment and receives a private reward $R^{(i)}$ related to their individual action $A^{(i)}$ on the state S of the environment. Each agent i aims to maximise its own total expected discounted return $\sum_{t=0}^{T}\gamma^t R_t^{(i)}$.

We developed a MADDPG model based on the Lowe *et al.* article [5] from 2017, which is designed to solve Markov Games with continuous actions. MADDPG in a lot of aspects is very similar to a D3PG model. Each agent i has its own local critic $Q^{(i)}$ and policy $\mu^{(i)}$ networks and their target copies. The agents still learn from a Replay Buffer.

But MADDPG also diverge in some facets. The data structure stored in memory contains all the partial observations $O_t^{(i)}$, actions $A_t^{(i)}$ and their individual rewards $R_{t+1}^{(i)}$, as well as, the next partial observations $O_{t+1}^{(i)}$ and if the episode has terminated or not, $done^{(i)}$.

The policy network for each agent i still only depends on its private observation $\pi^{(i)}(O^{(i)})$ but its critic network is omniscient of the full state $S \equiv (O^{(1)}, \ldots, O^{(n)})$, where n is the number of agents – in this case 2 – and all the individual agents' actions – $Q^{(i)}(S, A^{(1)}, \ldots, A^{(n)})$ – a centralised action-value function.

The loss function L^Q for the critic network still follows the same structure as equation (8),

$$\begin{aligned} L^Q = \frac{1}{n} \sum_i \big[R_{t+1}^{(i)} + \gamma Q'^{(i)}(S_{t+1}, \mu'^{(1)}(O_t^{(1)}), \ldots, \mu'^{(n)}(O^{(n)})) \\ - Q^{(i)}(S_t, A_t^{(1)}, \ldots, A_t^{(n)} | \theta_{(i)}^Q) \big]^2 \end{aligned} \tag{10}$$

and the loss function $L_{(i)}^{\mu}$ for each agent's policy also follows the same structure as D3PG policy equation (9),

$$L_{(i)}^{\mu} = -Q^{(i)}(S_{t+1}, \mu^{(1)}(O_t^{(1)}), \ldots, \mu^{(i)}(O^{(i)} | \theta_{(i)}^{\mu}), \ldots, \mu'^{(n)}(O^{(n)})) \tag{11}$$

where only the network parameters $\theta_{(i)}^{\mu}$ get updated in the gradient descent in each agent i learning step.

For the policy μ we kept the same structure that we have used in the D3PG model. A fully connected neural network with two hidden layers. The input layer has the size of the state space and the output layer has the size of the action space. We used *tanh* as the last layer transformation in order to keep action values inside the interval $[-1, 1]$.

For the value function Q we have also used a fully connected neural network with two hidden layers. The input layer has this time the size of the state space multiplied by the number of agents, in this case 2. It is only in the second hidden layer again that we introduce the action values, this time for both agents, concatenated to the values coming from the first hidden layer. The output layer has only one node with no non-linear transformation performed. We use ReLU functions for any other node transformations and Adam was kept as the optimiser of choice.

There were a few enhancements that we thought could help these two models – D3PG and MADDPG – performance. As we will see in more detail in the results section, learning starts quite slowly as the positive reward initially is difficult to come by. The probability of the racquet to hit a ball is quite low and the probability that the hit causes the ball to cross the net is a few orders of magnitude even lower.

In order to intensify the signal, especially initially, of the positive reward we explored the efficacy of Noisy Nets as source of exploratory noise and later the efficiency of Prioritisation Experience Replay (PER) in supporting the recall of these positive reward signals.

1.3 Noisy Nets

We based our development approach of the Noisy Nets in the article of Fortunato *et al.* from 2018 [6], where we introduced a similar noise term from equation (7) but this time not directly in the action of each agent i as in equation (6), but on each element of the policy network parameters, $\theta^{\mu}_{(i)}$. The explorative action is then determined by this new noisy policy function and equation (6) for Noisy Nets changes to the following,

$$A^{(i)} = \mu^{(i)}(O^{(i)}|\theta^{\mu}_{(i)} + \mathcal{N}). \tag{12}$$

1.4 PER

Randomly accessing memory always seems very inefficient, especially in environments where positive reward signals are very sparse in the state-action space. Based on the article from Schaul *et al.* from 2016 [7], we adapted an efficient implementation already developed by C. Joo in [8] to our model.

In PER we assign a priority to each experience that goes into memory. The priority p is based on the error E in the TD(0) update of the critic network Q for each experience,

$$\begin{aligned} E = \max_i \mid R^{(i)}_{t+1} + \gamma Q'^{(i)}(S_{t+1}, \mu'^{(1)}(O^{(1)}_t), \ldots, \mu'^{(n)}(O^{(n)})) \\ - Q^{(i)}(S_t, A^{(1)}_t, \ldots, A^{(n)}_t|\theta^{Q}_{(i)}) \mid \end{aligned} \tag{13}$$

In C. Joo's implementation the priorities have the following structure,

$$p = (E + e)^a, \tag{14}$$

where $a = 0.6$ and $e = 0.01$. This structure ensures no experience is left with null probability of being picked and with a below 1 we avoid large errors to overtake completely the learning process.

As we are changing the probability of a particular experience to be recalled it is advisable to change the weight w of this experience in the learning update process (10). In some sense resizing the learning rate for each individual experience depending on its priority. In C. Joo's implementation the weights w are given by the following equation,

$$w \propto (p + 10^{-10})^{-\beta}, \tag{15}$$

where β starts at 0.4 and increases by 0.001 per learning update until it reaches 1.

At each learning step, the error E values for each experience sampled gets updated by equation (13) for the most recent levels of the critic network Q. The C. Joo's implementation uses a *Sum Tree* structure that makes such experience priority update in memory quite efficient with the least amount of calculations – $\mathcal{O}(\log(n))$ – where n is the number of leaf nodes of the tree.

One comment about C. Joo's algorithm – it starts its sampling in the cumulative space with an equidistant grid. Effectively sampling with no replacement and with a relatively sparse density sample of the experience space at all times. The experience priority value gets further away from the effective probability of recall of the experience in this set-up.

1.5 Explicit Collaboration Level Setting

We further experimented with the learning performance of the model by explicitly changing the reward system in the environment. We altered the reward for each agent i in equation (10), using instead $R^{(i)}_{CL}$ defined by,

$$R^{(i)}_{CL} = (1-c)R^{(i)} + c\sum_j R^{(j)}, \tag{16}$$

where c defines explicitly the level of collaboration of the agents. In the limit $c = 1$ the agents share all the rewards. Or, when $c < 0$ we start to define explicitly the level of competition between the agents.

1.6 Energy Expenditure Constraint

When solving the environment we have noticed that although the agents had perfect action on the ball maintaining the game going until the end of the episode when trained, they would also demonstrate a quite erratic and inefficient movement without the ball.

Initially we tried to curtail the issue with some L2 regularisation of the policy neural networks. But later we introduced an extra term in the loss function L^{μ} for each agent,

$$\begin{aligned} L^{\mu}_{(i)} = -Q^{(i)}(S_{t+1}, \mu^{(1)}(O^{(1)}_t), \dots, \mu^{(i)}(O^{(i)}|\theta^{\mu}_{(i)}), \dots, \mu'^{(n)}(O^{(n)})) \\ + e \mid \mu^{(i)}(O^{(i)}|\theta^{\mu}_{(i)}) \mid, \end{aligned} \tag{17}$$

where e is an Energy Expenditure parameter constraining the actions of the agent.

In the next section, we will detail how we implemented these learning algorithms in practice.

2 Implementation

Our implementation of the D3PG model follows closely the code architecture introduced in class. The definition of the D3PG agents is split between two files,

- `d3pg_agent_final.py` and `d3pg_agent_nn_per.py`
- `model_d3pg_final.py` and `model_d3pg_nn_per.py`

In `d3pg_agent_final.py` and `d3pg_agent_nn_per.py`, we define the *Agent()* class. One of the alterations to the class definition we introduce was to expand the inputs and include a dictionary with all the hyperparameters to improve flexibility in analysing and tuning the model. In this class the methods *step*, *act*, *learn* and *soft_update* get defined. In *step()* we sequence how the agent stores each experience in memory and how often it starts the learning process. In *act()*, we define the *behaviour policy* of the agent – based on equation (6) if it is learning or the greedy policy μ otherwise. It is in the *learn()* method that the loss functions L^Q and L^μ, as defined in (8, 9), get calculated and optimised. In this implementation we use an Adam optimization.[2] It is also worth mentioning that we perform batch learning – for each training step we sample the agent's memory for a set of experiences and use their information in the optimisation step. At each learning step we also perform a soft update of the target neural network parameters $(\theta^{Q'}, \theta^{\mu'})$, according to equation (1), using the *soft_update()* method.

A second class is defined in `d3pg_agent_final.py` and `d3pg_agent_nn_per.py` – the *ReplayBuffer()* class – where the agent's memory structure gets defined. In this case a *queue* of fixed maximum size is initialised and *add()* and *sample()* methods get defined. In `d3pg_agent_final.py` memory sampling is done randomly where each experience has an equal probability[3] to be picked for the learning process whilst in `d3pg_agent_nn_per.py` there is the option to turn on PER. There is also an option in `d3pg_agent_nn_per.py` to turn on Noisy Nets which are defined in the *act()* method.

In `model_d3pg_final.py` and `model_d3pg_nn_per.py`, the neural networks get defined. We use *PyTorch* to define a fully connected deep neural network with two hidden layers.[4] Both input layer units and output layer units are fixed in the D3PG model but we kept the number of units for both hidden layers variable in order to be able to test and tune them.

We created a third set of files in order to systematise our agents' training process – `training_d3pg_agent_final.py` and `training_d3pg_agent_nn_per.py`. We loop the agents through episodes keeping track of their performance in order to establish when they have successfully solved the environment. We also define a maximum amount of episodes `n_episodes` after which if our agents are still not successful we stop the training process.

Our implementation of the MADDPG model follows closely the code architecture of D3PG. In fact there is only one more file than in D3PG.The definition of the MADDPG agents is split between two files just like the D3PG model,

- `maddpg_agent_cl_ee.py`

[2]We kept Adam optimisation from the original implementation and it was something we have not tested in terms of performance against other optimisation schemes.

[3]with replacement.

[4]We have not tested the performance of our neural network architecture against other types

- `model_maddpg.py`

In the MADDPG model each individual agent cannot learn only act so the learning process and the coordination of each agent is provided in the file,

- `maddpg_cl_ee.py`

where the class *Team()* gets defined. It is in *Team()* that the method *learn()*, *soft_update()*, *step()* and even a *Team()* version of *act()* gets defined.

The last two files part of this implementation is

- `SumTree.py`
- `prioritized_memory.py`

In these files we define the PER algorithm. In `prioritized_memory.py` we define the class *Memory()* and the methods to *add()*, *sample()* and *update()* the priority of experiences. In `SumTree.py` we define the necessary tree structure underlying the efficiency of the algorithm.

3 Results

In this section we summarise the results of our analysis that expands four notebooks.

3.1 D3PG analysis

We initially tried to solved the environment using the same D3PG model developed for the previous project using Barth-Maron's hyperparameters [1], as we can see in Table 1 column *Article*, with increased initial noise and higher number of episodes per training session to intensify exploration. We were able to solve the environment 80% of the times in 1650 episodes on average.

We then moved on to use the Tuned Hyperparameters from previous project also in Table 1 in column *Base*. After testing for 50 learning sessions, we were able to solve the environment 94% of the times in 2000 episodes on average as we can see in Figure 1.

After specifically tuning for this environment we arrived at the hyperparameters in column *Long* also in Table 1 and we ran a long training session with no stop at the target score to see how far we could take the model. We were able to make the agent learn to almost perfect score of 2.6, as we can see in Figure 2.

We have also tried to work on the movement of the agent without the ball using higher *L2* values of regularisation on further training of the agent but with very limited success.

Name	Article	Base	Long	Description
`n_episodes`	5000	5000	1500	max no of episodes per training session
`max_t`	1000	1000	1000	max no of steps per episode
`actor_fc1_units`	400	400	400	no of units for first hidden layer actor net
`actor_fc2_units`	300	300	300	no of units for second hidden layer actor net
`critic_fc1_units`	400	400	400	no of units for first hidden layer critic net
`critic_fc2_units`	300	300	300	no of units for second hidden layer critic net
`buffer_size`	10^6	10^5	10^6	memory size
`batch_size`	256	128	512	no of examples fed at each learning step
`gamma`	0.99	0.99	0.99	rewards discounting rate
`tau`	10^{-3}	10^{-3}	10^{-3}	soft update learning rate for target nets
`lr_actor`	10^{-4}	10^{-4}	10^{-4}	local actor net learning rate
`lr_critic`	10^{-4}	10^{-4}	10^{-4}	local critic net learning rate
`weight_decay`	0	0	10^{-6}	L2 weight decay
`update_every`	1	1	1	no of action steps per learning step
`epsilon`	0.9	0.9	0.5	uniform noise process initial scale parameter

Table 1: Hyperparameters values used in the D3PG analysis.

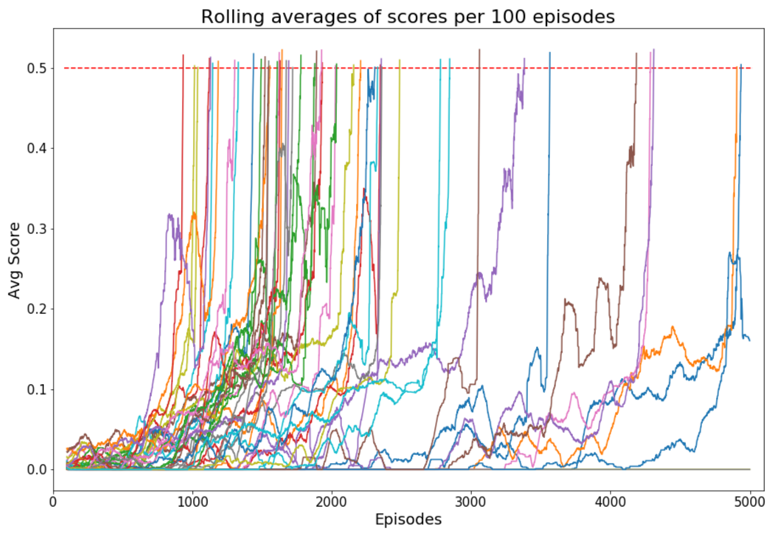


Figure 1: Rolling average scores of 50 different training sessions using previously tuned hyperparameters on a D3PG model.

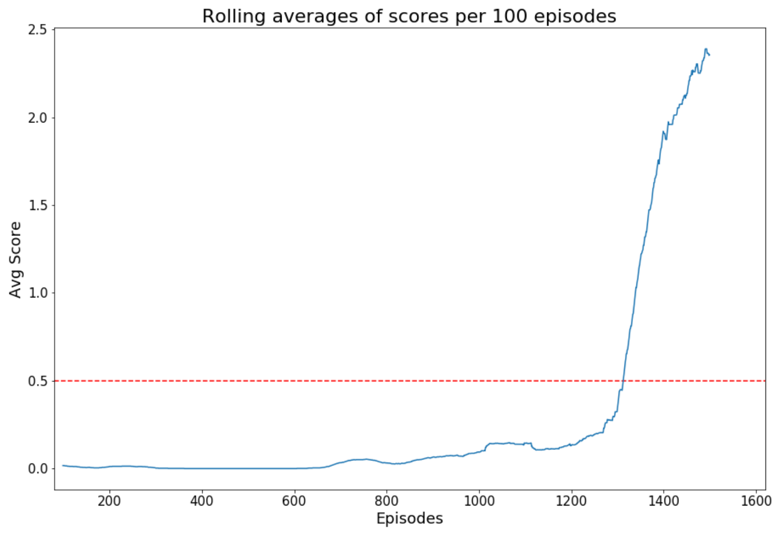


Figure 2: Rolling average scores of a training session using the final tuned hyperparameters on a D3PG model.

3.2 D3PG with Noisy Nets and PER analysis

The introduction of Noisy Nets to the D3PG model did not seem advantageous. We used $\epsilon_0 = 0.005$ and $\epsilon_{HL} = 200$. And results were poor at 80% success rate and an average of 2100 episodes taken to solve the environment. But PER, on the other hand, was quite successful. Using the hyperparameters in the *Long* column of Table 1, we achieved 100% success rate in 10 training sessions taking on average of 2000 episodes to solve the environment, as we can see in Figure 3.

We tried to switch both Noisy Nets and PER at the same time but again this did not bring any improvement to the model performance achieving only 70% of success rate.

3.3 MADDPG analysis

In our implementation of MADDPG we have used the hyperparameters in the column *Long* of Table 1 and for Noisy Nets the same parameters as before $\epsilon_0 = 0.005$ and $\epsilon_{HL} = 200$.

We only ran one training session for MADDPG with each extra feature switched on. This time the best performance came from MADDPG with Noisy Nets solving the environment in 2646 episodes as we can see in Figure 4.

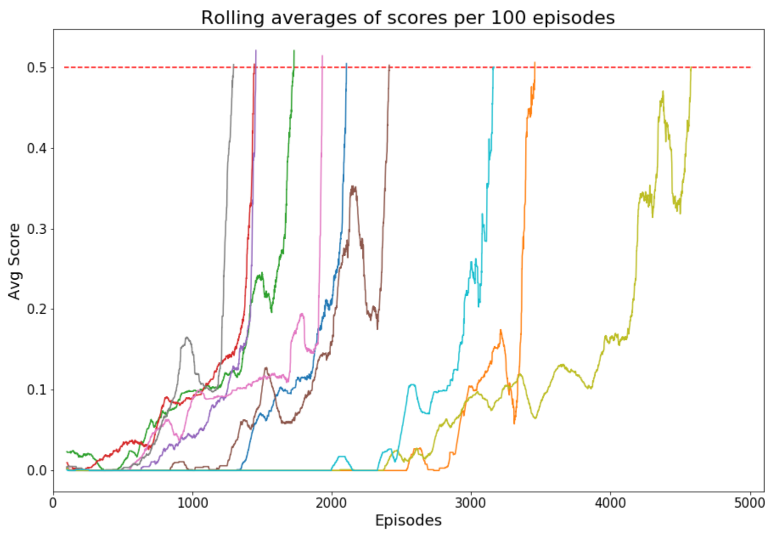


Figure 3: Rolling average scores of 10 training sessions using a D3PG model with PER.

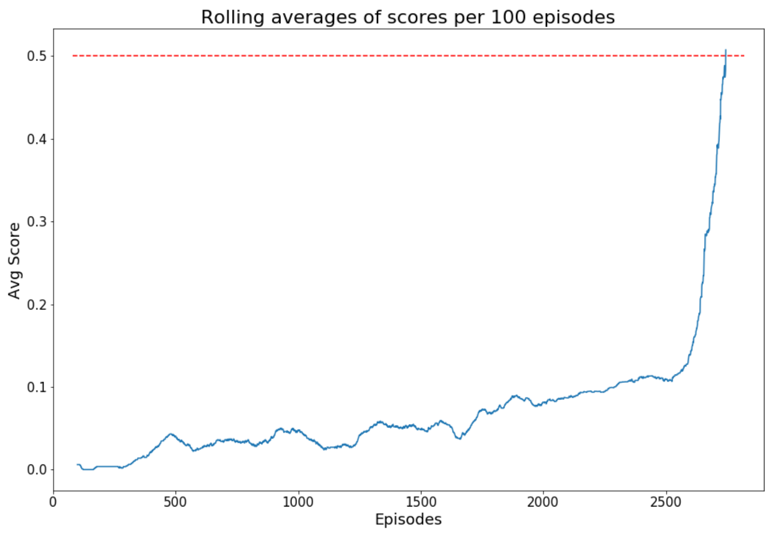


Figure 4: Rolling average scores of a training session using a MADDPG model with Noisy Nets.

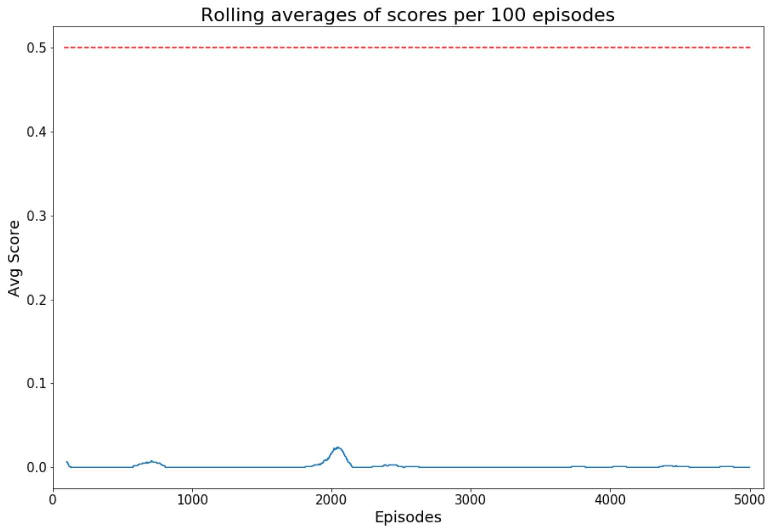


Figure 5: Rolling average scores of a training session using a MADDPG model with Unit Collaboration Level.

3.4 MADDPG with Collaboration Level and Energy Expenditure analysis

In this notebook we tried to explore further the MADDPG model and its interaction with the environment.

Setting a Collaboration Level $c = 1$ was unsuccessful as the model failed to learn at all as we can see in Figure 5.

But surprisingly, with $c = 1$ and Energy Expenditure $e = 10^{-4}$ or even $e = 10^{-3}$ the model learns quite successfully, as we can see in Figure 6.

In the final long run we execute in this notebook with more than 3100 episodes and score averaging 1.66, we can see that the Energy Expenditure term is helping the agent movement to settle as agents tend to have a less erratic and smoother movement without the ball.

4 Ideas for Future Work

Although a lot has been developed and analysed in this Project. There is plenty more to investigate further and understand better about the learning algorithms implemented with all the different enhancements and their interaction with this specific environment.

Here is a list of different aspects that we collected whilst doing the research that would be interesting to examine further in this Project:

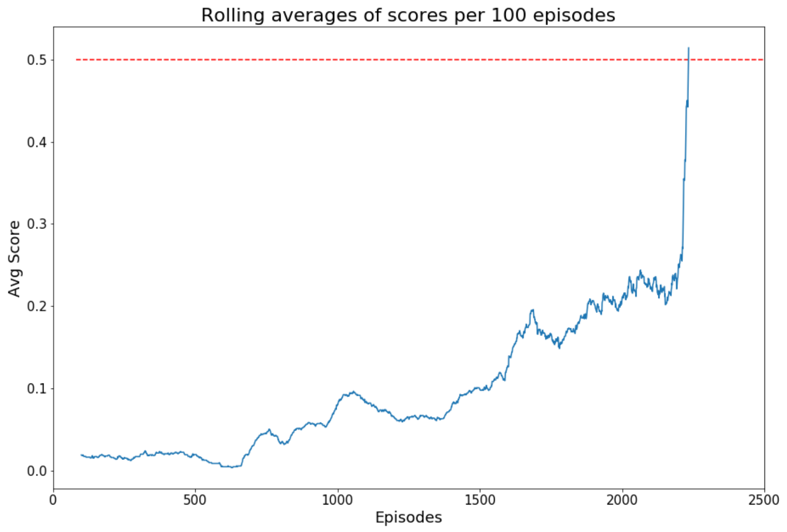


Figure 6: Rolling average scores of a training session using a MADDPG model with Unit Collaboration Level and Energy Expenditure 10^{-4}.

- Specifically tune the hyperparameters of MADDPG and understand if D3PG is really better at solving this environment,
- Test changing and tune the PER parameters kept fixed in this Project,
- Change the error formula in PER from an L1 to a squared or higher order term and analyse the impact in the learning process,
- Change the Energy Expenditure term from an L1 to a squared term of the actions or even higher order and analyse the change in the movement without the ball of the agent,
- Further understand Collaboration Level impact in the learning process. Analyse it for different positive and negative numbers.
- Understand the interaction between Collaboration Level and Energy Expenditure. Analyse why the model learns with both switched on but not separately.

The results of some of these investigations might come handy as we intend to tackle next the `Soccer` environment - a more challenging *Unity*'s environment suggested in the lecture notes.

Our idea is to keep this project going and continue posting more notebooks with further improvements and more results of our exploration as we go along.

Watch this space... ☺

References

[1] G. Barth-Maron *et al.*, "Distributed distributional deterministic policy gradients," *ICLR conference*, 2018.

[2] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.

[3] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arxiv.org*, February 2016.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning - An Introduction.* The MIT Press, second ed., 2018.

[5] R. Lowe *et al.*, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arxiv.org*, p. 1706.02275v3, January 2018.

[6] M. Fortunato *et al.*, "Noisy networks for exploration," *arxiv.org*, no. 1706.10295, 2017.

[7] T. Schaul *et al.*, "Prioritized experience replay," *arxiv.org*, no. 1511.05952, 2016.

[8] C. Joo, "Prioritized experience replay (per) implementation in pytorch," *https://github.com/rlcode/per*, January 2018.

[9] M. G. Bellemare *et al.*, "A distributional perspective on reinforcement learning," *arxiv.org*, no. 1707.06887, 2017.

[10] Y. Duan *et al.*, "Benchmarking deep reinforcement learning for continuous control," *arxiv.org*, no. 1604.06778, 2016.

[11] J. Schulman *et al.*, "Proximal policy optimization algorithms," *arxiv.org*, no. 1707.06347, 2017.

[12] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," *arxiv.org*, no. 1602.01783, 2016.