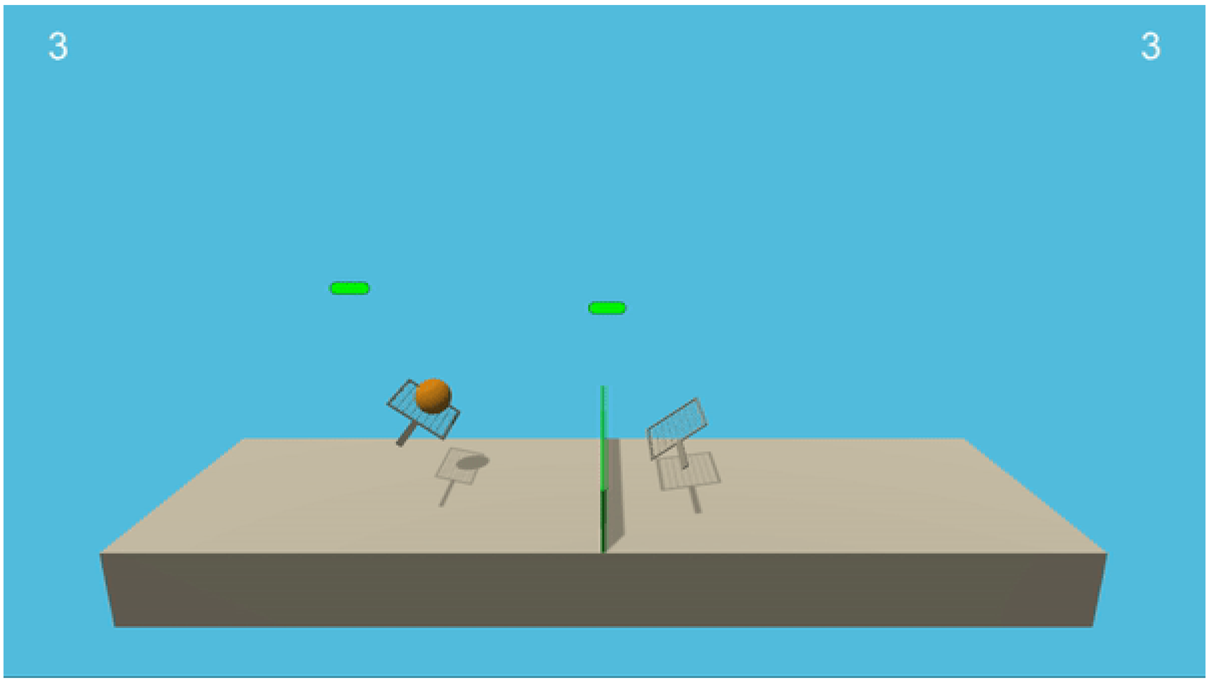
Tennis project report

Daniel Thell, November 2020

The Environment[1]

For this project, the work will be with the Tennis environment.



In this environment, two agents 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. Each agent receives its own, local observation. The observations received are 3 consecutive frames of 8 variables, so 24 entries, which allow the agent to determine the direction and speed of the ball. 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, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- 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.

[1] This first section "The Environment" is mainly taken from Udacity Collaboration and Competition Project from the Deep Reinforcement Learning Nanodegree

- This yields a single **score** for each episode.

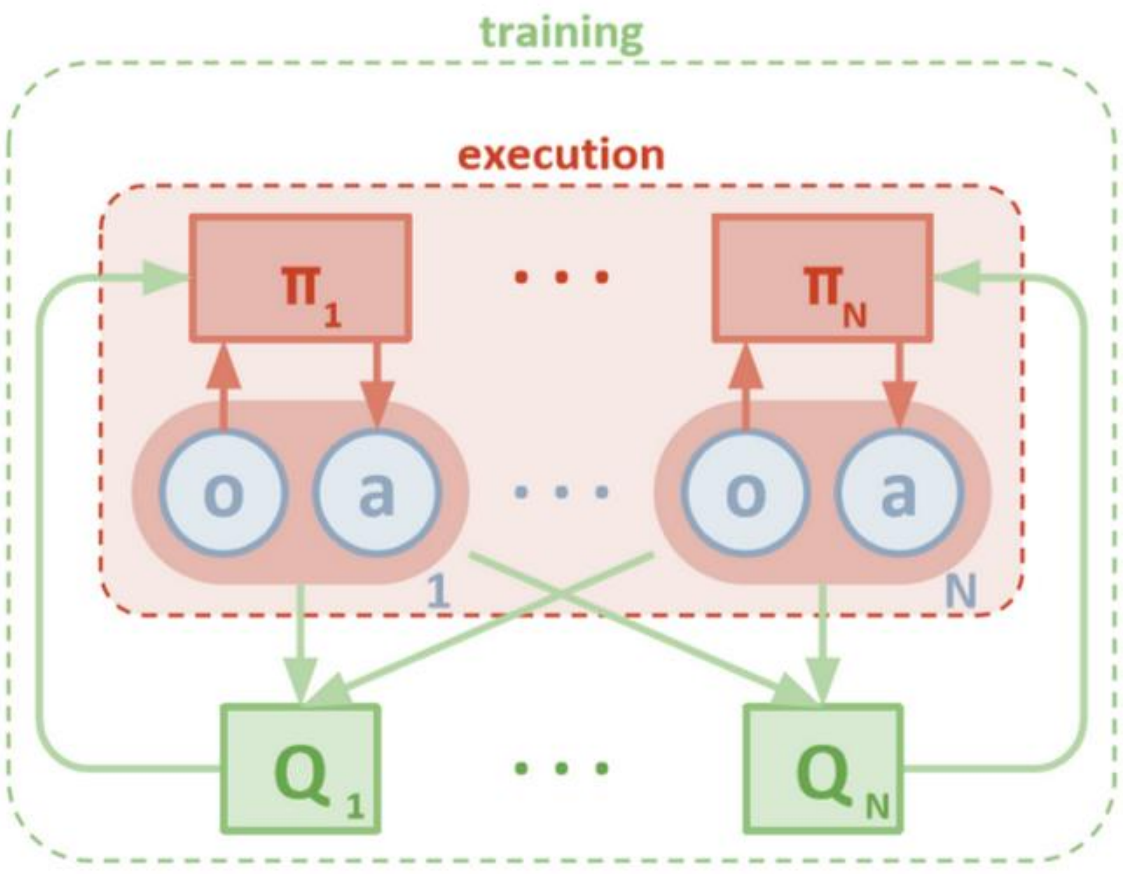
The environment is considered solved, when the average (over 100 episodes) of those **scores** is at least +0.5.

The learning algorithm

This environment is much harder to solve than the previous projects of the Nanodegree (Navigation or Continuous control) as the environment is dynamic (it depends on what the other agent is doing and this evolves over time) and continuous (so there is an infinite number of actions possible and DQN type algorithms cannot work).

The solution implemented is a multi-agent DDPG algorithm. It has 2 DDPG agent (one for each tennis player), each having 2 actor networks and 2 critic networks. The 2 actor networks and the 2 critic networks are identical in shape in my implementation.

The general framework for this algorithm is the following.



MADDPG Architecture (Lowe, 2018)

The agents learn from a number of random episodes. They try out some actions from a given observation of the environment (which may differ depending on the agent) using the actor network. This network estimates for each agent separately the best action to take and during the learning phase the critic network gives an estimate of the q-value given all of the actions chosen by all agents and the full observation of all agents. The actors' networks are only seeing the agent own observations while the critic network sees both agent observations and both agent actions. The networks are all updated through gradient descent using an Adam optimizer. The critic is only used during the training phase.

A first version of the algorithm tested is as follows. For each agent there are 2 actor networks (a local and a target) and 2 critic networks (a local and a target). The gradient descent is applied on the local networks. The critic network tries to minimize the TD error estimate and the actor network tries to minimize minus the q-value of the critic network (meaning to maximize the expected q-value but as the algorithm applies gradient descent a minus sign is used). The target networks are only soft updated using a Tau parameter to include Tau amount of the local network parameters and keep (1-Tau) of the old target network parameters. As Tau is small, of the order of 1e-2 or 1e-3, the target networks get updated slowly, ensuring some stability to the algorithm.

A second version of the algorithm tested, to reach a solution faster, is with a critic network common to both agents, as it sees all actions and observations. Sharing the same network means it is getting updated faster. Also I used the same actor network for both agents, but each time seeing only one agent observation. Having only one actor network is not anymore exactly applying an MADDPG algorithm, but I kept the main feature being the actor network sees only one agent observation at a time while the critic network sees all observations and all actions for all agents.

The learning is done from sampled episodes stored in a replay buffer. The learning cannot start for the very first time steps as the replay buffer is too small. We tested batch sizes of 512 or 1024 so learning car start only after having done at least 512 or 1024 timesteps.

The actor and critic networks used shared the same architecture, with 2 fully connected hidden layers every time, with 256 nodes and a ReLU activation function every time except on the out layer. The number of nodes in the hidden layers was increased gradually from 32 initially to 128 or 256 as a too small number of nodes was not allowing the agents to perform correctly. A 256 x 256 nodes hidden layers architecture for the 1st version of the algorithm and a 256 x 128 nodes hidden layers architecture for the second version of the algorithm were retained in the rest of the project. Both actors and critic networks have the same number of nodes in the hidden layers.

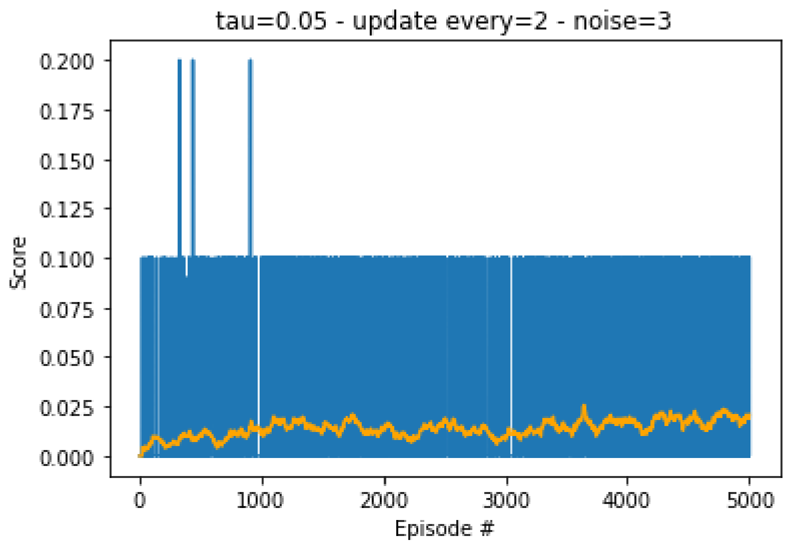
The only differences between actor and critic networks are:

- The input sizes: one agent observations of size 24 for the actors, 2 agents' observations and 2 agents actions with total size 52 for the critics.
- The output sizes: one action of size 2, with values between -1 and +1, for the actors, one q-value of size 1 for the critic.
- The activation function for the output layer: tanh for the actors to ensure action values between -1 and 1 while the plain linear output was kept for the critic output layer.

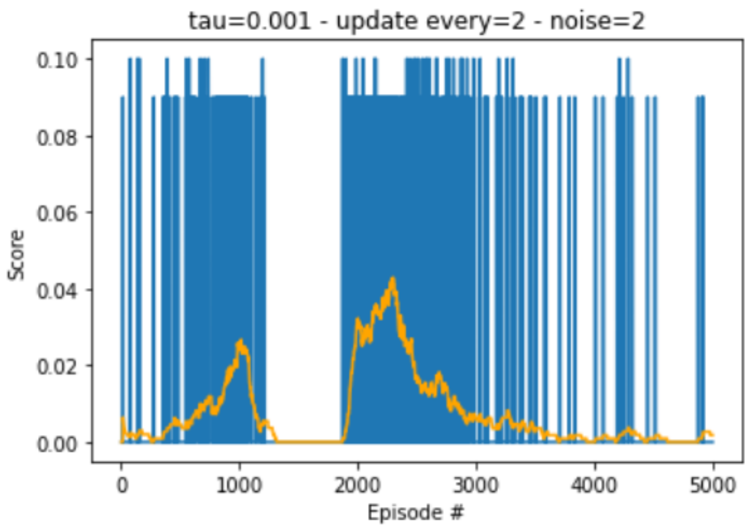
To prevent diverging given large gradients, gradient clipping was also implemented as in the DDPG algorithm used in the Continuous Control project, to cap the norm of the gradients applied.

The investigation

To test various set of hyperparameters with the first version of the algorithm, I first implemented a training loop with various levels of Tau (soft update weight, between 0.001 and 0.01), noise (to control the level of initial exploration between 0.5 and 3), and update frequency (between every 1 episodes and every 4 episodes). All over 5000 episodes. The results were very erratic but showed that the most promising sets would be with a fairly large amount of initial noise to explore a large number of cases although a large instability in the scores even after a large number of episodes (and a reduced level of noise) was seen (second graph below) even for small Tau values which were supposed to be more stable. The number of learning pass required to reach the +0.5 average mark is fairly large as the best average results were not even close to 0.1 after 5000 episodes.



Slow gradual improvement



Faster increase of the score while Tau is smaller but training can be erratic as here with pretty similar hyperparameters

To increase the number of updates, I switched to a learning pass at every timestep instead of at every episode. I also tested 5 learning passes every timestep, at the expense of the learning time.

Seeing the time taken by this learning, I implemented the second faster version of the algorithm described in the previous section and ran the training of the 2 versions in parallel.

The solution implemented

The solution implemented for the 1st version used the following parameters

Agents hyperparameters:

• BUFFER_SIZE = int(5e5)	# replay buffer size
• BATCH_SIZE = 1024	# minibatch size
• GAMMA = 0.95	# discount factor

- TAU = 5e-2 # for soft update of target parameters
- LR_ACTOR = 3e-5 # learning rate of the actor
- LR_CRITIC = 1e-5 # learning rate of the critic
- WEIGHT_DECAY = 1e-5 # L2 weight decay
- NUMBER_UPDATE_PASSES = 5 # number of learning passes every time the learning from the replay buffer is called
- NOISE_DECAY = 0.998 # speed decay of the OU noise
- learning_frequency = 1 # Number of timesteps between 2 learning calls

The critic network seemed to converge pretty quickly compared to the actor network so the learning rate was kept a bit higher for the actor networks.

A second solution used one shared actor network (but every time seeing only one single agent observation) and one shared critic network (which sees all observations and actions as in the previous solution). This goes faster as the networks gets updated twice faster and share knowledge faster. Sharing the same network for the agents works thanks to the fact that observations are exactly symmetrical for both agents (going toward or back from the net). The agents hyperparameters for this second solution are:

- BUFFER_SIZE = int(5e5) # replay buffer size
- BATCH_SIZE = 1024 # minibatch size
- GAMMA = 0.95 # discount factor
- TAU = 5e-2 # for soft update of target parameters
- LR_ACTOR = 1e-3 # learning rate of the actor
- LR_CRITIC = 1e-3 # learning rate of the critic
- WEIGHT_DECAY = 0 # L2 weight decay was not used here
- NUMBER_UPDATE_PASSES = 1 # number of learning passes every time the learning from the replay buffer is called
- NOISE_DECAY = 1.0 – 1.0/300 # speed decay of the OU noise, noise disappears linearly in 300 episodes
- learning_frequency = 1 # Number of timesteps between 2 learning calls

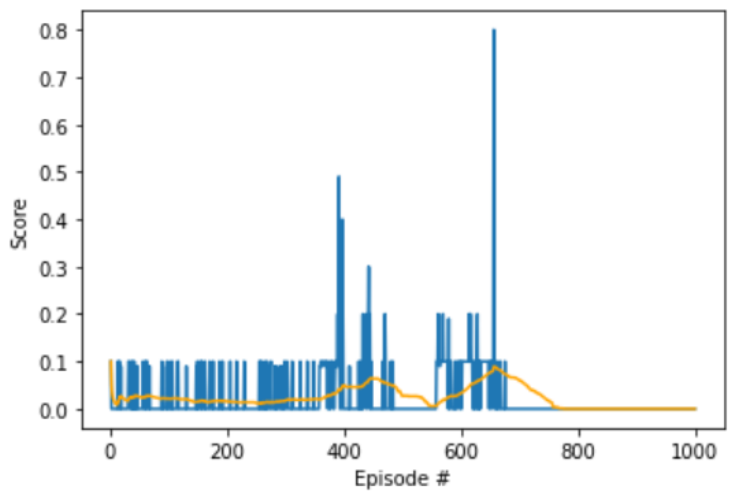
The learning rate here was kept higher at inception and decreased by a factor 0.999 after every episode beyond episode 100.

The results

The training can become very long as a large number of updates to the networks is needed. Also the length of one single episode is increasing as the agents learn. This slows down the progress in the number of episodes but it is a good sign as it means that the agents are learning to play together.

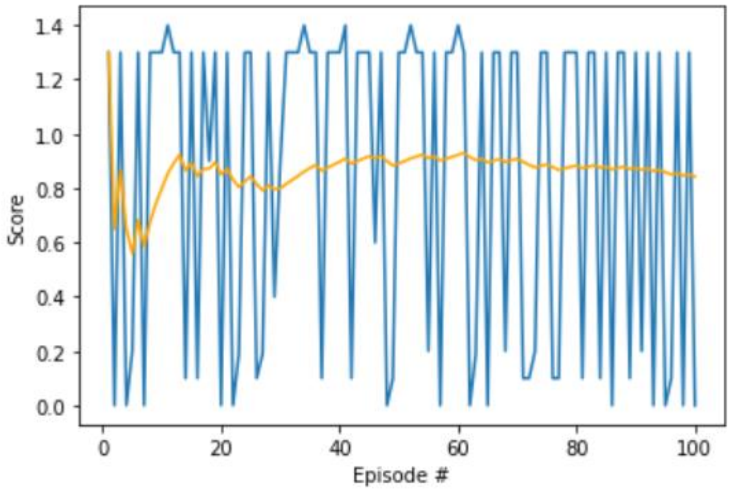
The first version of the algorithm (one set of actor and critic networks per agent, not shared) takes very long to train. It is still in progress at about episode 3600, after over 24 hours of training, with an average score over 100 episodes around or slightly below 0.1, meaning on average the agents hit the ball once at almost all episodes but cannot play together yet. The max score reached so far is +0.5, meaning 5 successful exchanges in one single game for one of the two agents. This is what is needed to solve the environment if it can be repeated over 100 consecutive episodes. Training is still on and I will update the report once it finishes.

The second version of the algorithm goes faster to train even if it still takes a large amount of time. The best score reached 0.8 after 700 episodes in the first loop before dropping.

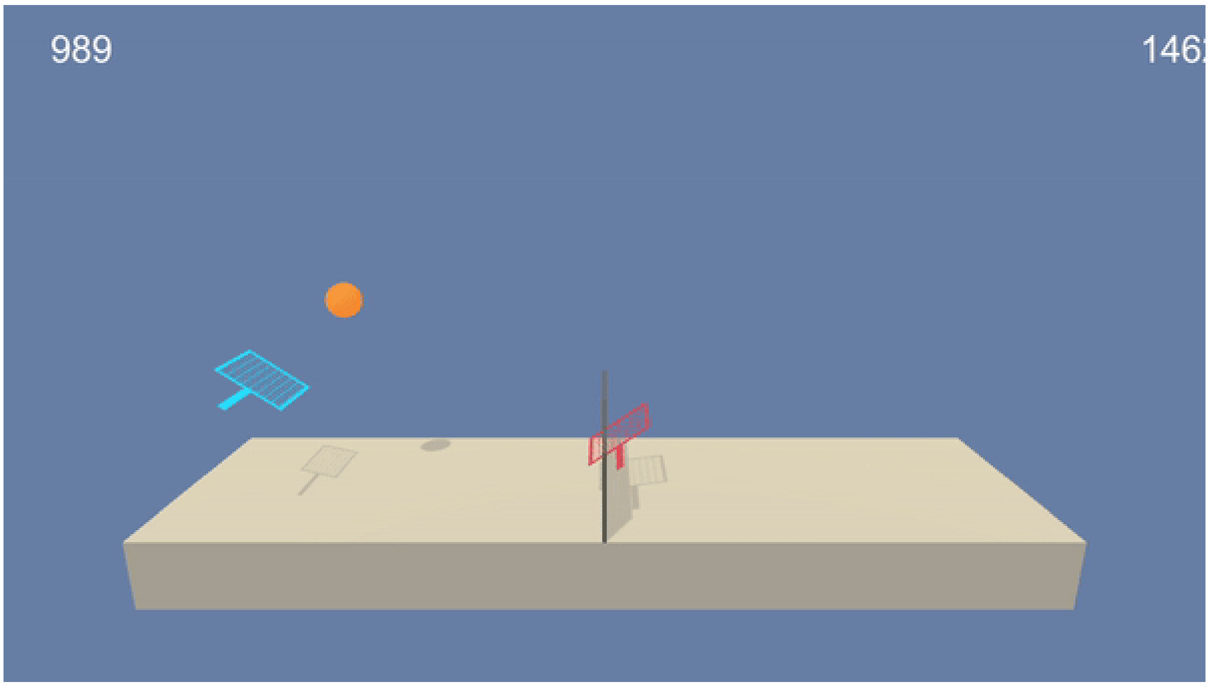


A second loop was launched starting from the model parameters achieved around episode 600 with a learning rate decreased by a factor 10 to fine tune and avoid losing the good learning reached so far. The process was iteratively followed, decreasing the learning rate every time the training was resumed from a better set of parameters.

After a few iterations and **in total 941 episodes, the training reached the goal with an average score of +0.50 over 100 consecutive episodes (from episode 941 to episode 1040)** and a max score of +1.4 over one single episode, which seems to be the maximum reachable score within the 500 timestep limit that we set for training. The learning graph below shows the score achieved over 100 episodes once the training reached this average level.



The final result gets a well-trained set of tennis players as seen below



The model weights are saved in the file “episode-140-update_every_t_tau-0.05-update_loop-1-noise-0_new-v3-solved.pt” and can be loaded in the 6th section of the Jupyter notebook.

Further improvements

It would be interesting to test a prioritized experience replay for the sampling into the replay buffer to see whether this could speed up the training as it takes very long to get the first meaningful results. Maybe batch normalization could also prove helpful to speed up learning as it proved useful in previous projects.

The learning is very unstable and maybe the network architecture is too complex and overfits the learning, reducing the number of nodes in the hidden layers could be tested, or otherwise a faster learning rate decay could be useful.

The learning spends a lot of time trying first to hit the ball and then to catch a first ball from the other agent. It is not a learning algorithm change idea here but a more refined reward scheme could help giving more insight to the agents on how good the action was instead of a mere good +0.1 or bad -0.01: Did they miss the ball by a lot or not? It is clearly better to miss it by an inch than being at the opposite side of the table. Did they hit the ball but it went in the net? It is better to hit the ball and put it in the net than not hit it at all, a small reward should be given. Is it the first hit or a hit after the other player successfully hit the ball? An increasing reward scheme (for example +0.1 for the first ball over the net, +0.15 for the second one, +0.2 for the 3rd one…) could push the agents quicker into playing together.

References

I used extensively the Knowledge forum of the Udacity nanodegree to get a sense of the good initial directions. I also read with a lot of interest and attention the following 2 articles on “toward data science”:

https://towardsdatascience.com/openais-multi-agent-deep-deterministic-policy-gradients-maddpg-9d2dad34c82

https://towardsdatascience.com/training-two-agents-to-play-tennis-8285ebfaec5f

Some ideas expressed inside such as the common critic network or the more frequent learning steps have been tested to progress towards the successful solution.

The python code overall architecture was adapted from the one of the Udacity MADDPG lab.