Project

This is the third project of the Deep Reinforcement Learning Nanodegree. I trained a Multi DDPG Agent to solve the Tennis environment. This project is influenced by the previous one: https://github.com/escribano89/reacher-ddpg and the DDPG implementations from the Udacity's repository https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum.

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. 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. 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.

Implementation

We have implemented a Multi Deep Deterministic Policy Gradient (*MADDPG*). This algorithm is composed by *n* DDPG agents *(in this concrete environment is composed by 2 ddpg agents).*

DDPG Algorithm (revisited from the previous project)

In order to apply this algorithm I used the resources detailed below:

- DDPG original paper: https://arxiv.org/pdf/1509.02971.pdf
- MADDPG paper: https://arxiv.org/abs/1706.02275
- Udacity - DDPG Pendulum implementation: https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum
- Open AI DDPG: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

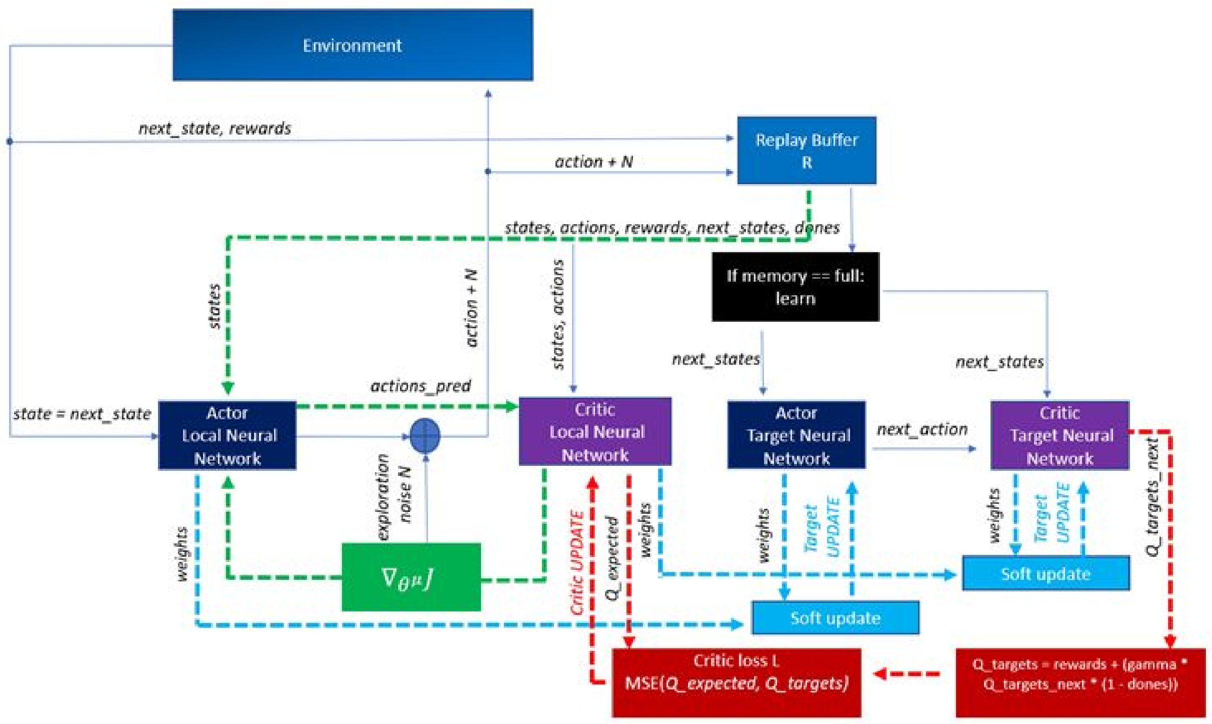
DDPG Algorithm

Algorithm 1 Deep Deterministic Policy Gradient

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:     Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:     Execute $a$ in the environment
6:     Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:     If $s'$ is terminal, reset environment state.
9:     if it's time to update then
10:        for however many updates do
11:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:            Compute targets
                   $y(r, s', d) = r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:            Update Q-function by one step of gradient descent using
                   $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s,a) - y(r,s',d)\right)^2$
14:            Update policy by one step of gradient ascent using
                   $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:            Update target networks with
                   $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$
                   $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$
16:        end for
17:    end if
18: until convergence
```

Deep Q-Learning models are not straightforwardly applied to continuous tasks due to the high number of actions (inf) in the continuous domain. We can't apply the argmax method to that amount of possibilities. Here is where DDPG comes in. DDPG is like an "extension" of the DQN algorithm in order to work with continuous environments.

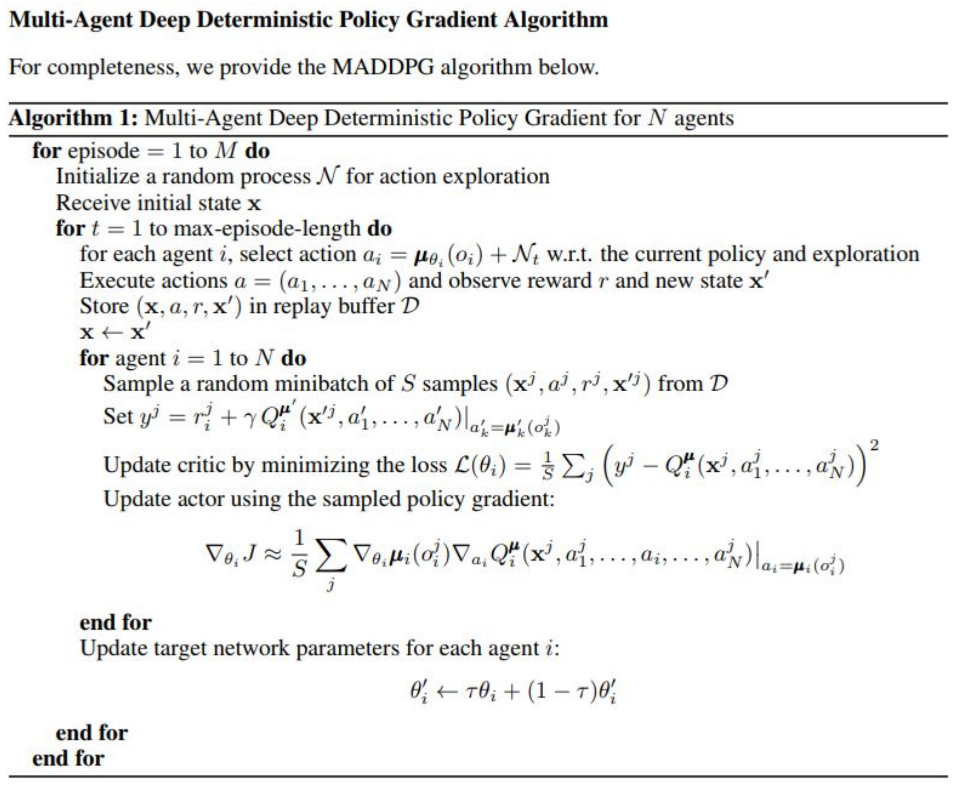
Structure (taken from https://miro.medium.com/max/1508/1*x6HIECcvr60kzSHdDU0zZw.png)



Although we are using an Actor-Critic approach, the roles of both are different than the explained ones in the A2C classroom section. The actor implements a deterministic policy to map states to the best action available. The critic is trained as the Q-Learning method, but the next action in the equation is given from the Actor's target output. The actor is trained using the gradient from maximizing the estimated q-value from the critic. The actor's predicted action (*the best one in that moment*) is used as the input to the critic.

Two important things to highlight is that DDPG uses a Replay buffer to gather experiences from the agent and the target networks are updated using soft updates, mixing both neural networks slowly. This improves the convergence of our model.

MADDPG Algorithm



Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

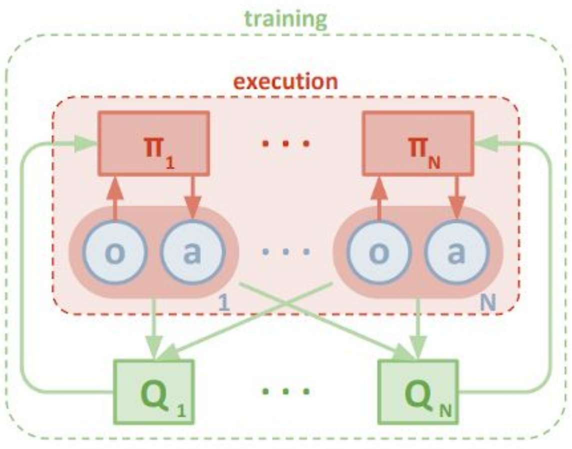
```
for episode = 1 to $M$ do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length do
    for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \dots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to $N$ do
      Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \dots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \dots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

        $$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \dots, a_i, \dots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent $i$:

        $$\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$$

  end for
end for
```

The MADDPG algorithm is the multi-agent version of DDPG. Extracted from the paper:

"We explore deep reinforcement learning methods for multi-agent domains. We begin by analyzing the difficulty of traditional algorithms in the multi-agent case: Q-learning is challenged by an inherent non-stationarity of the environment, while policy gradient suffers from a variance that increases as the number of agents grows. We then present an adaptation of actor-critic methods that considers action policies of other agents and is able to successfully learn policies that require complex multi-agent coordination. Additionally, we introduce a training regimen utilizing an ensemble of policies for each agent that leads to more robust multi-agent policies. We show the strength of our approach compared to existing methods in cooperative as well as competitive scenarios, where agent populations are able to discover various physical and informational coordination strategies."

MADDPG adapts the actor-critic method to consider action policies of other agents by adopting a **centralized training** with **decentralized execution**.

The execution is a responsibility of each Actor network but the training is centralized by the critic using a shared experience replay buffer. Each critic observes the tuples of all the agents in order to train more effectively. It’s important to remember that each agent still takes actions based on its own unique observations of the environment.

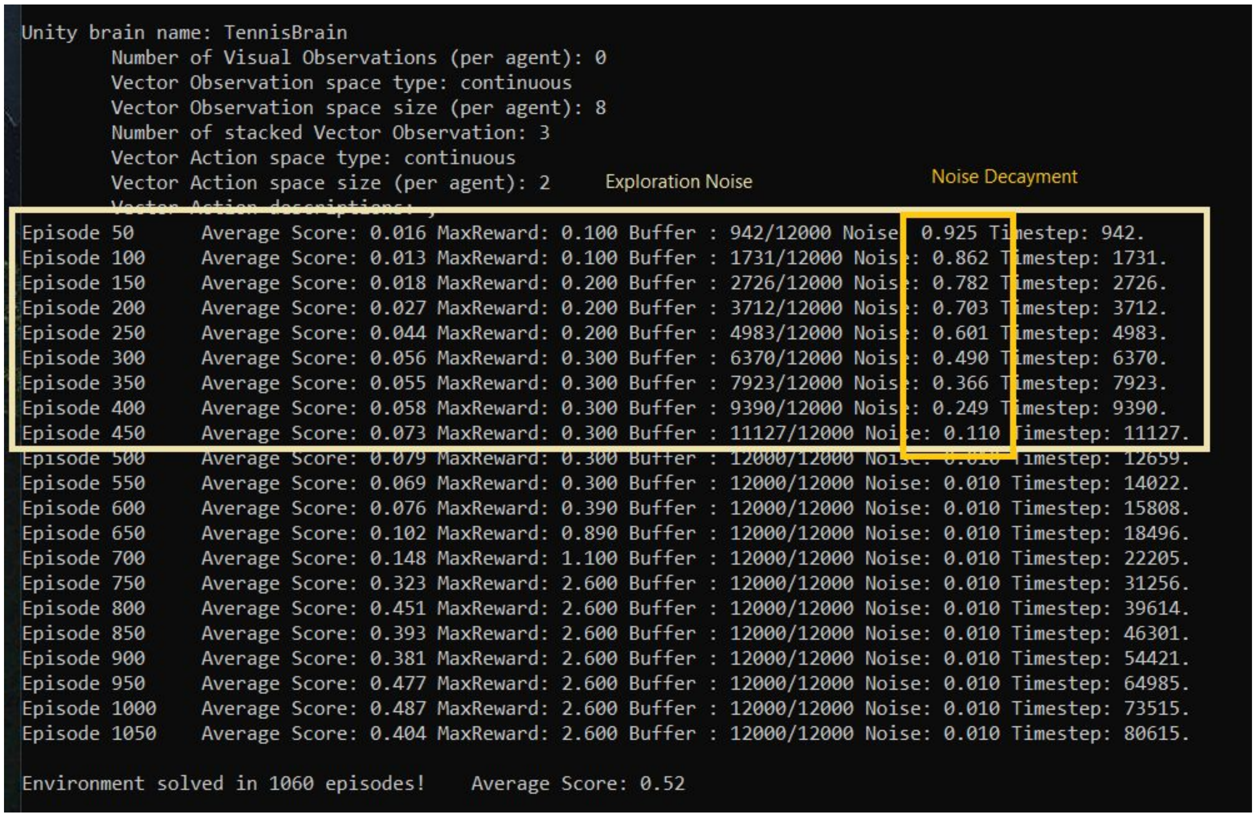


Strategy

After some experiments and different configurations, I reached some conclusions that allowed me to reach the goal of the Tennis environment.

I noticed that I had to reduce the experience buffer in order to constantly remove the early experiences. This time I had to introduce Exploration Noise (I didn’t include it in the previous project) because it’s very necessary to explore the environment and get positive signals (hit the ball properly). For that purpose I used the provided Ornstein-Uhlenbeck process by the Udacity Repo:
(https://github.com/udacity/deep-reinforcement-learning/blob/master/ddpg-pendulum/ddpg_agent.py)

That’s why I set up a strong initial noise with some decayment until reaching the buffer capacity (more or less). *(please see the next image).*



```
Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
Episode 50      Average Score: 0.016 MaxReward: 0.100 Buffer : 942/12000 Noise: 0.925 Timestep: 942.
Episode 100     Average Score: 0.013 MaxReward: 0.100 Buffer : 1731/12000 Noise: 0.862 Timestep: 1731.
Episode 150     Average Score: 0.018 MaxReward: 0.200 Buffer : 2726/12000 Noise: 0.782 Timestep: 2726.
Episode 200     Average Score: 0.027 MaxReward: 0.200 Buffer : 3712/12000 Noise: 0.703 Timestep: 3712.
Episode 250     Average Score: 0.044 MaxReward: 0.200 Buffer : 4983/12000 Noise: 0.601 Timestep: 4983.
Episode 300     Average Score: 0.056 MaxReward: 0.300 Buffer : 6370/12000 Noise: 0.490 Timestep: 6370.
Episode 350     Average Score: 0.055 MaxReward: 0.300 Buffer : 7923/12000 Noise: 0.366 Timestep: 7923.
Episode 400     Average Score: 0.058 MaxReward: 0.300 Buffer : 9390/12000 Noise: 0.249 Timestep: 9390.
Episode 450     Average Score: 0.073 MaxReward: 0.300 Buffer : 11127/12000 Noise: 0.110 Timestep: 11127.
Episode 500     Average Score: 0.079 MaxReward: 0.300 Buffer : 12000/12000 Noise: 0.010 Timestep: 12659.
Episode 550     Average Score: 0.069 MaxReward: 0.300 Buffer : 12000/12000 Noise: 0.010 Timestep: 14022.
Episode 600     Average Score: 0.076 MaxReward: 0.390 Buffer : 12000/12000 Noise: 0.010 Timestep: 15808.
Episode 650     Average Score: 0.102 MaxReward: 0.890 Buffer : 12000/12000 Noise: 0.010 Timestep: 18496.
Episode 700     Average Score: 0.148 MaxReward: 1.100 Buffer : 12000/12000 Noise: 0.010 Timestep: 22205.
Episode 750     Average Score: 0.323 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 31256.
Episode 800     Average Score: 0.451 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 39614.
Episode 850     Average Score: 0.393 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 46301.
Episode 900     Average Score: 0.381 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 54421.
Episode 950     Average Score: 0.477 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 64985.
Episode 1000    Average Score: 0.487 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 73515.
Episode 1050    Average Score: 0.404 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 80615.

Environment solved in 1060 episodes!    Average Score: 0.52
```

So balancing initial strong exploration with later knowledge from a reduced buffer it allows us to converge properly. I have included Batch Normalization in both neural networks (Actor and critic) in order to stabilize the learning process. I clipped the critic gradient too in order to prevent instabilities due to the gradient exploding.

Another important thing (*it's a difference from the previous project)* is that I got better results updating the neural network more frequently and more slowly (soft-updates). I increased the batch size a bit because the buffer is reduced.

Neural Network structure *(per each agent)*

Actor (*2 equal Neural Networks - Regular and Target*)

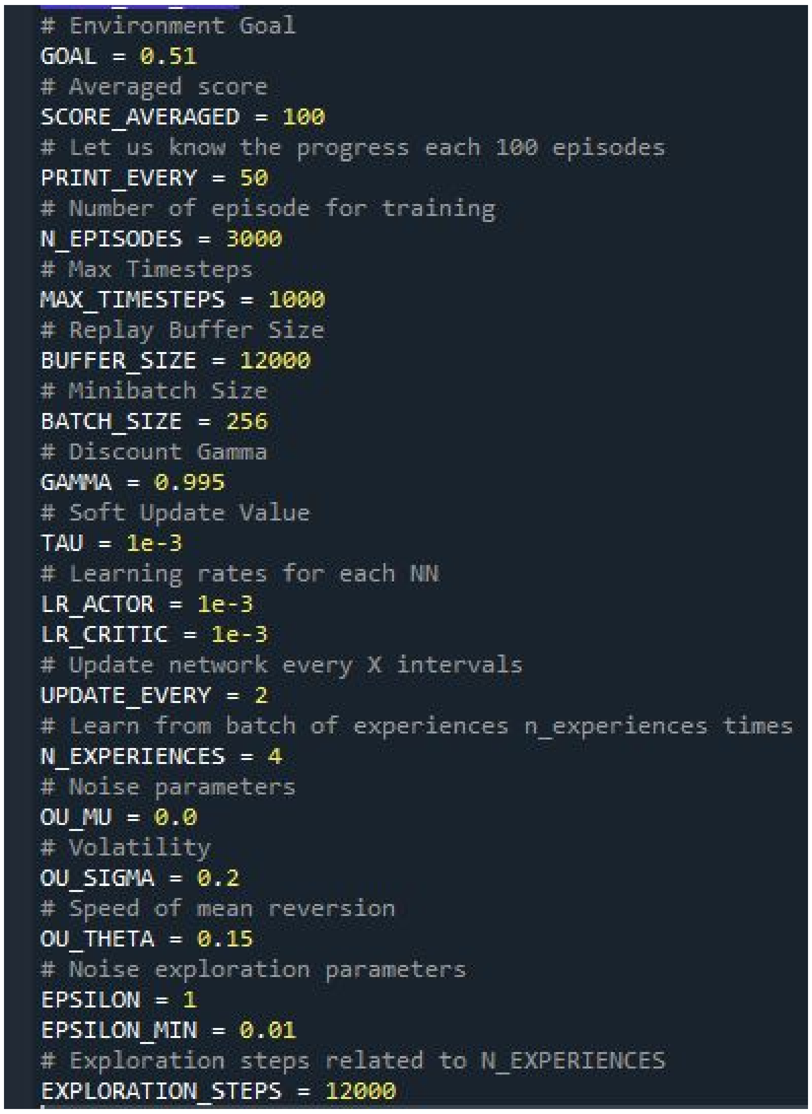
Input -> Vector Observation size
Hidden Layer 1 - 128 Units with Batch Normalization
Hidden Layer 2 - 64 Units
Hidden Layer 3 - 32 Units
Hidden Layer 4 - 16 Units
Output layer -> Action size -> Used tanh as output activation function

<u>Critic</u> (*2 equal Neural Networks - Regular and Target*)

Input -> state + action size
Hidden Layer 1 - 256 Units with Batch Normalization
Hidden Layer 2 - 128 units
Hidden Layer 3 - 64 units
Hidden Layer 4 - 32 units
Output layer -> 1

We use RELU as an activation function. The optimizer used is Adam.

The hyperparameters chosen to reach the goal after some trials and experiments are configured in the *params.py* file:

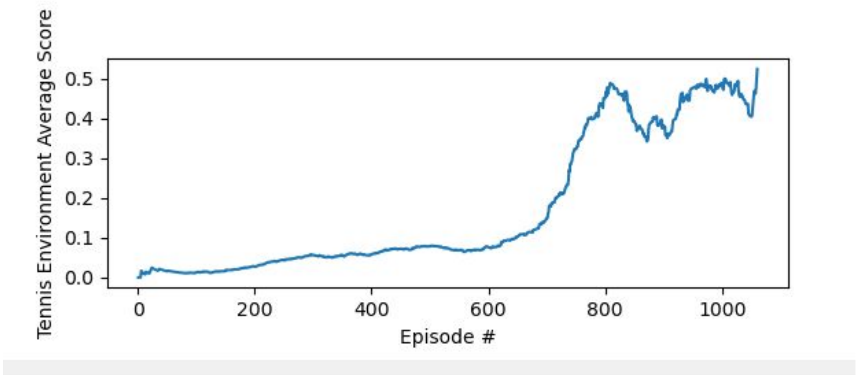


```
# Environment Goal
GOAL = 0.51
# Averaged score
SCORE_AVERAGED = 100
# Let us know the progress each 100 episodes
PRINT_EVERY = 50
# Number of episode for training
N_EPISODES = 3000
# Max Timesteps
MAX_TIMESTEPS = 1000
# Replay Buffer Size
BUFFER_SIZE = 12000
# Minibatch Size
BATCH_SIZE = 256
# Discount Gamma
GAMMA = 0.995
# Soft Update Value
TAU = 1e-3
# Learning rates for each NN
LR_ACTOR = 1e-3
LR_CRITIC = 1e-3
# Update network every X intervals
UPDATE_EVERY = 2
# Learn from batch of experiences n_experiences times
N_EXPERIENCES = 4
# Noise parameters
OU_MU = 0.0
# Volatility
OU_SIGMA = 0.2
# Speed of mean reversion
OU_THETA = 0.15
# Noise exploration parameters
EPSILON = 1
EPSILON_MIN = 0.01
# Exploration steps related to N_EXPERIENCES
EXPLORATION_STEPS = 12000
```

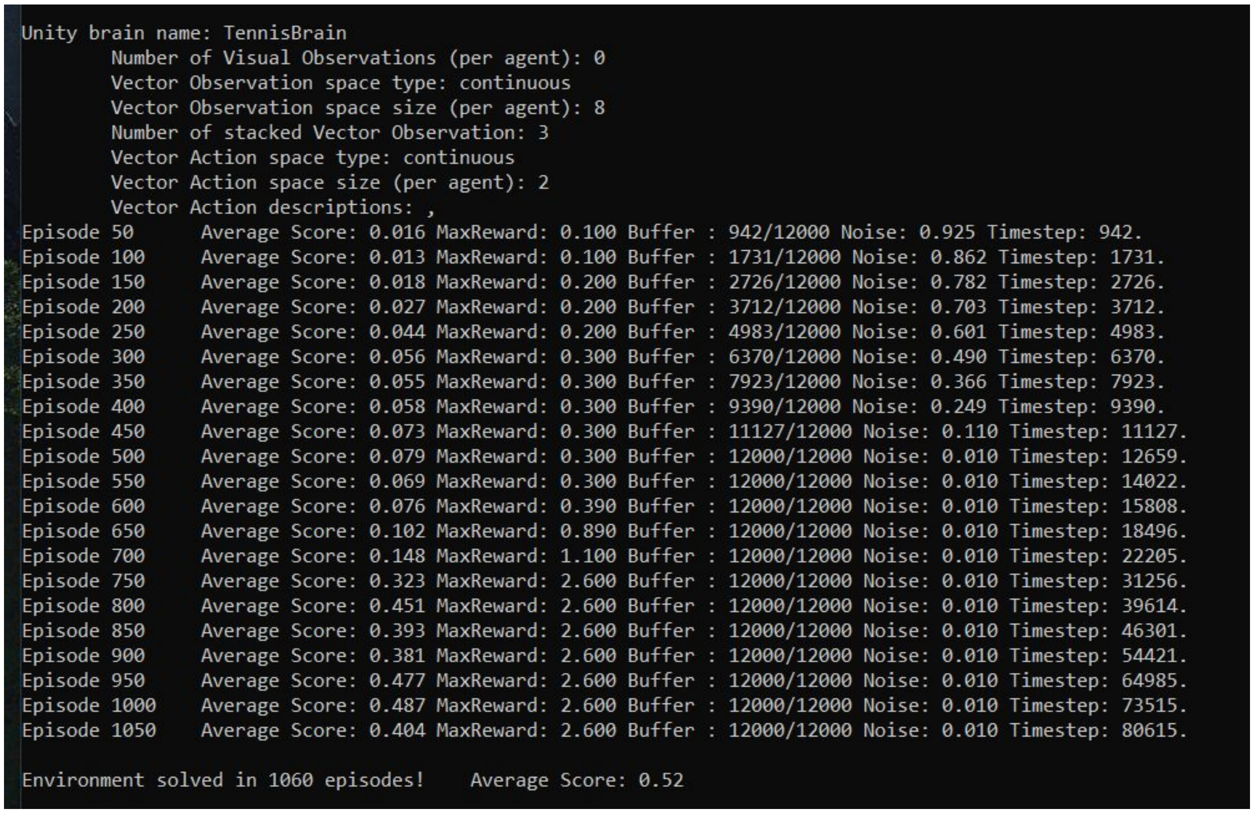
Plot of rewards *(averaged over 100 episodes)*

We have followed the below instructions in order to average the scores:

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.



We reached our goal in less than 1100 episodes.



```
Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
Episode 50      Average Score: 0.016 MaxReward: 0.100 Buffer : 942/12000 Noise: 0.925 Timestep: 942.
Episode 100     Average Score: 0.013 MaxReward: 0.100 Buffer : 1731/12000 Noise: 0.862 Timestep: 1731.
Episode 150     Average Score: 0.018 MaxReward: 0.200 Buffer : 2726/12000 Noise: 0.782 Timestep: 2726.
Episode 200     Average Score: 0.027 MaxReward: 0.200 Buffer : 3712/12000 Noise: 0.703 Timestep: 3712.
Episode 250     Average Score: 0.044 MaxReward: 0.200 Buffer : 4983/12000 Noise: 0.601 Timestep: 4983.
Episode 300     Average Score: 0.056 MaxReward: 0.300 Buffer : 6370/12000 Noise: 0.490 Timestep: 6370.
Episode 350     Average Score: 0.055 MaxReward: 0.300 Buffer : 7923/12000 Noise: 0.366 Timestep: 7923.
Episode 400     Average Score: 0.058 MaxReward: 0.300 Buffer : 9390/12000 Noise: 0.249 Timestep: 9390.
Episode 450     Average Score: 0.073 MaxReward: 0.300 Buffer : 11127/12000 Noise: 0.110 Timestep: 11127.
Episode 500     Average Score: 0.079 MaxReward: 0.300 Buffer : 12000/12000 Noise: 0.010 Timestep: 12659.
Episode 550     Average Score: 0.069 MaxReward: 0.300 Buffer : 12000/12000 Noise: 0.010 Timestep: 14022.
Episode 600     Average Score: 0.076 MaxReward: 0.390 Buffer : 12000/12000 Noise: 0.010 Timestep: 15808.
Episode 650     Average Score: 0.102 MaxReward: 0.890 Buffer : 12000/12000 Noise: 0.010 Timestep: 18496.
Episode 700     Average Score: 0.148 MaxReward: 1.100 Buffer : 12000/12000 Noise: 0.010 Timestep: 22205.
Episode 750     Average Score: 0.323 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 31256.
Episode 800     Average Score: 0.451 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 39614.
Episode 850     Average Score: 0.393 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 46301.
Episode 900     Average Score: 0.381 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 54421.
Episode 950     Average Score: 0.477 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 64985.
Episode 1000    Average Score: 0.487 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 73515.
Episode 1050    Average Score: 0.404 MaxReward: 2.600 Buffer : 12000/12000 Noise: 0.010 Timestep: 80615.

Environment solved in 1060 episodes!    Average Score: 0.52
```

Future ideas

In order to improve our model, it would be nice to perform a grid search to find the best hyperparameters. It would be nice to optimize the critic and the actor separately (*applying value-based and policy-based techniques respectively*). I would like to explore more in detail the relation between the buffer size, the experience sampling and the OU Noise.

It would be nice to implement other algorithms like Twin Delayed DDPG (*TD3*). Indeed, they can be included in the same project parametrizing the options in order to execute one or another using the command line. A benchmark comparing the result would be very useful.

MADDPG is using a shared Replay Buffer, but this can be improved by prioritizing it. This could be another improvement that can have a great impact in our algorithm. I already tried using this resource *https://pylessons.com/CartPole-PER/* but I wasn't able to integrate it properly within the available time to deliver the project.

Another idea would be to replace the information provided by Unity (observation space) and learn from the raw pixel instead. That would imply the use of a Convolutional Neural Network to apply filters and find patterns directly from the images.

Finally, I would like to adapt this project to solve the Soccer environment. Definitely is the next thing I'm gonna do.