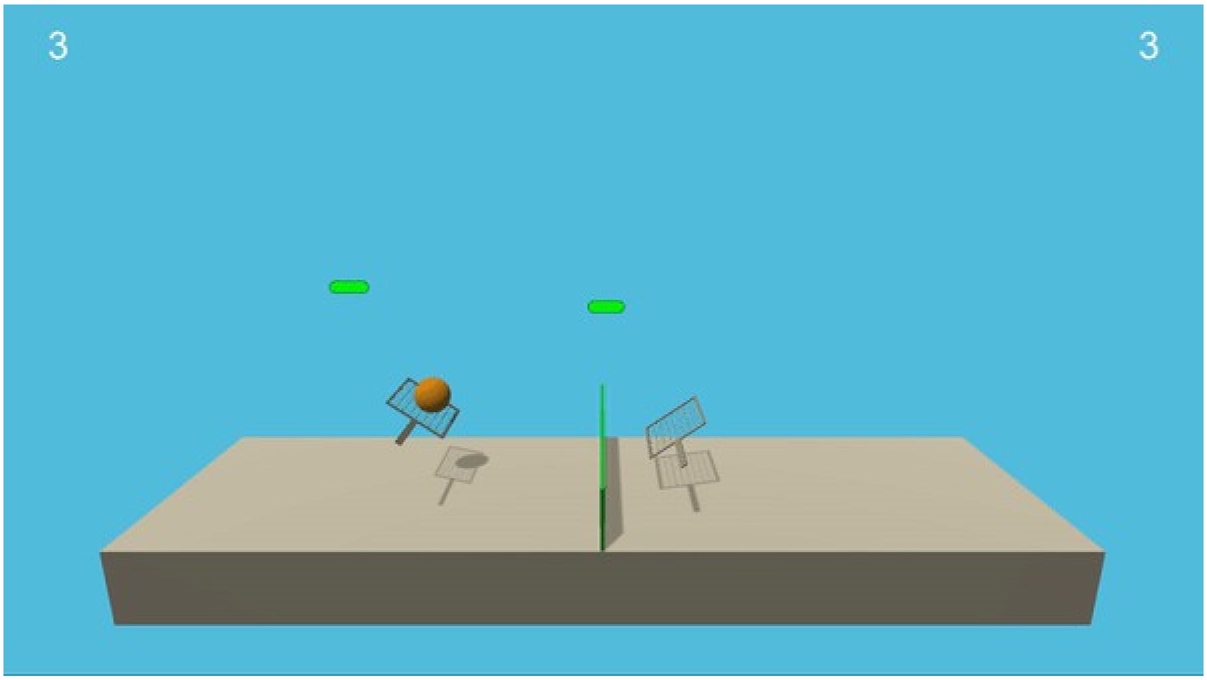
Deep Reinforcement Learning Nanodegree

Project 3- Collaboration-Competition.

REPORT



Introduction

This report provides a description of the implementation for the Deep Reinforcement Learning Nanodegree Project 3 to address the Multi-Agent Collaboration and Competition problem. 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.. Please refer to the README.md on this repository for more information regarding this environment and its installation.

Learning Algorithm

1. The Agent

After going through the Multi-agent lab to address the **Physical Deception** problem, I wanted to base this project of the code developed for this lab. Unfortunately, there were two main reasons I abandoned this idea:

- Udacity's Unity environment doesn't allow multiple environments (Threads). Unity crashes if you try to make a second instance of the environment...but on the Tennis environment on build by Unity ML agents they have solved this problem
- Also the way the **Physical Deception** Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm is structured assumes a sungle "brain" controlling 1 single agent. For that reason in the algorithm, the actions must be explicitly calculated for each agent separately. On the other side, on P2-Continous Control (previous project) using 20 agents, the environment was using 1 single brain controlling all 20 agents. When I worked on this environment I already did the work needed for understanding and implementing the "multi-agent decentralized actor, centralized critic approach". That meant that the code from P2 needed minor changes to be able to work on P3
- After going through the **Physical Deception** problem lab code in detail, I knew that in reality, and knowing that I could not use multiple thread to accelerate the learning, there were esentially no mayor differences with the DDPG code developed on P2

Therefore I adapted the code from P2 to solve this problem

The Model Architecture Problem

Having already addressed my concerns about the architecture on P2 I decided to, first, try to run it with the same architecture since the problems were similar in their Observation and Actions Spaces. My intial trials showed success and therefore I decided to simply stick to my Model 2 architecture:

1. Model 2: with **3 hidden** layers (Actor) with a linear activation for each and Tanh activation for the output and with 2 **hidden** layers (Critic) with a Relu activation for each.

Recall that in my previous project I perform a set of tests trying different number of layers, different number of neuruns per layer (filters on the Conv) and compared Relu and Leaky Relu and I noticed that the model with 3 hidden layers (originally found on Udacity's implementations) was learning faster and keeping a higher average of rewards

Summary of Additional Model Changes

ReLu Vs LeakyReLu: Leaky ReLU has a small slope for negative values, instead of altogether zero as ReLu does. The reason I wanted to try LeakyReLu was because it has two benefits over ReLu:

- It fixes the “dying ReLU” problem, as it doesn’t have zero-slope parts.
- It speeds up training. There is evidence that having the “mean activation” be close to 0 makes training faster. (It helps keep off-diagonal entries of the Fisher information matrix small, but you can safely ignore this.) Unlike ReLU, leaky ReLU is more “balanced,” and may therefore learn faster.

In this case, LeakyReLu did NOT show evidences of faster learning, although is possible to think that a different combination of my hyper-parameters could actually bring up LeakyReLu as a better choice.

Batch-normalization: The use of Batch normalization is a fascinating topic that started with me reading the paper that introduced it (https://arxiv.org/abs/1502.03167) Its benefits made clear sense after reading it but questions like: Should I apply Batch-Norm to just the input layer or to every input of every layer? Would it really make a difference? Researching on this topic was interesting and sometimes frustrating since I had no other choice but to perform a few tries…hoping that the differences observed were not going to be simply associated to the initial randomness of the weights. Batch-Normalization applied to every layer’s input showed the best performance. For details, please refer to model.py and model2.py files.

Gradient Clipping on the Critic Network: As it was clearly explained on the lectures, during ‘Training’ of a Deep Learning Model, we back-propagate our Gradients through the Network’s layers. Sometimes these gradients (tangent of the slopes) might be very large causing an overflow. Gradient clipping will ‘clip’ the gradients or cap them to a threshold value to prevent the gradients from getting too large. On the DDPG case and since the Critic Network uses the Actor’s best action approximation for training the Value/Critic Network, Clipping is only necessary on the Critic Net.

2. Experience Replay Vs Prioritized Experience Replay

The replay memory is a critical element on the DDPG (single and Multi-agent) implementation. In this case, using 2 agents, the replay memory was implemented “outside” the agent’s class allowing the agent’s experiences to be “memorized” together and later shared in the “Learning” process with the other agents. This learning model was described as “multi-agent decentralized actor, centralized critic approach” on “https://papers.nips.cc/paper/7217-multi-agent-actor-critic-for-mixed-cooperative-competitive-environments.pdf”

I wanted to test how Prioritized Experience Replay (PER) could improve my performance. It took me by surprise to see no improvement and even damaging the baseline performance. A quick look at this issue lead me to think that PER (https://arxiv.org/abs/1511.05952) uses the probability of a transition to occur, i.e. the number of times a transition occurs with respect the total number of transitions to calculate their “priority”. In a continuous action space, transitions can mathematically be infinite leading to priorities being totally normalized. At this point, when all memories (recorded transitions) have the same probability, sampling becomes exactly the same as in the basic Experience Replay, where we randomly select K samples. It seems only reasonable that prioritizing these memories using a different criteria of “importance” (more applicable to our case) might speed up the learning and even improve final performance. It make sense to think that using the “immediate” reward from every transaction might lead us to use and repeat the actions that lead us to a better reward. This is something I’d like to try in the future since time constraints to finish this assignment are not allowing me to do so now.

3. Other parameters of the MADDPG algorithm

Looking at “Class **args”** (defined on Tennis.ipynb) we can now focus and discuss how these arguments/parameters are used and how they affect the algorithm performance. Let me start by simply listing them below:

- seed = 777 Random seed
- disable_cuda = False Disable CUDA
- device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
- T_max = int(1e3) Number of training steps
- max_num_episodes = int(500) Max number of episodes
- hidden_1_size = 96 Network hidden layer 1 size
- hidden_2_size = 96 Network hidden layer 2 size
- hidden_3_size = 96 Network hidden layer 3 size
- noise_std = 0.05 Initial standard deviation of noise added to weights
- memory_capacity = int(1e6) Experience replay memory capacity
- batch_size = 256*1 Batch size: Number of memories will be sampled to learn
- learning_starts_ratio = 1/75 Number of steps before starting training => mem_capacity * ratio
- learning_frequency = 2 Steps before we sample from the replay Memory again
- priority_exponent = 0.5 Prioritized experience replay exponent (originally denoted α)

- priority_weight = 0.4 Initial prioritized experience replay importance sampling weight
- discount = 0.99 Discount factor
- target_update = int(30) # of steps after which to update target network
- tau = 1e-3 Soft Update interpolation parameter
- reward_clip = 1 Reward clipping (0 to disable)
- lr_Actor = 1e-3 Learning rate - Actor
- lr_Critic = 1e-3 Learning rate - Critic
- adam_eps = 1e-08 Adam epsilon (Used for both Networks)
- weight_decay = 0 Critic Optimizer Weight Decay

To keep the discussion short, and even though all parameters are relevant, I will only comment on those ones that I believe have a deeper impact on the final performance.

```
T_max = int(1e3)                # Number of training steps\n"
```

This parameter is definitely not that important, but I wanted to comment that not only is a good practice to limit the number of training steps taken on each episode to basically prevent it from getting into an infinite loop, but also it is interesting to see that in many “Toy Challenges” we need to use the “done” bit to be able to “reset” our environment in the case that “done” meant that we finished the task (failed or succeeded). So, for this reason I moved from: “while not np.any(dones):” to “while timestep<=T_Max:” in the Notebook code to make sure all episodes are consistently containing the same number of time steps/transitions.

```
hidden_1_size = 96              # Network hidden layer 1 size\n",
hidden_2_size = 96              # Network hidden layer 2 size\n",
hidden_3_size = 96              # Network hidden layer 3 size\n",
```

As mentioned before, these define the number of Neurons used in each of the 3 layers I finally used. 96 Neuron showed to be enough while 48 neurons were not enough to solve the task.

```
memory_capacity = int(1e6)      # Experience replay memory capacity\n",
batch_size = 256*1              # Batch size: Number of memories will be
sampled to learn"
learning_starts_ratio = 1/5     # Number of steps before starting training =
memory capacity * this ratio\n",
learning_frequency = 20         # Steps before we sample from the replay
Memory again    \n",
```

These are all Replay Memory/buffer related parameters. The “`learning_starts_ratio`” defines the number of transitions/steps that needed to be stored in memory before we start “learning”. The “`learning_frequency`“ defines the number of steps/transitions that we will have to go through (add to memory) after the condition to start sampling (learning) has been satisfied. In this way, we avoid sampling every timestep after this condition has been satisfied.

These parameters and their “proper” combination proved to be an important factor from the performance point of view. An art that took some time and testing to adjust. Please refer to P2 report for more details on how I tunned these parameters.

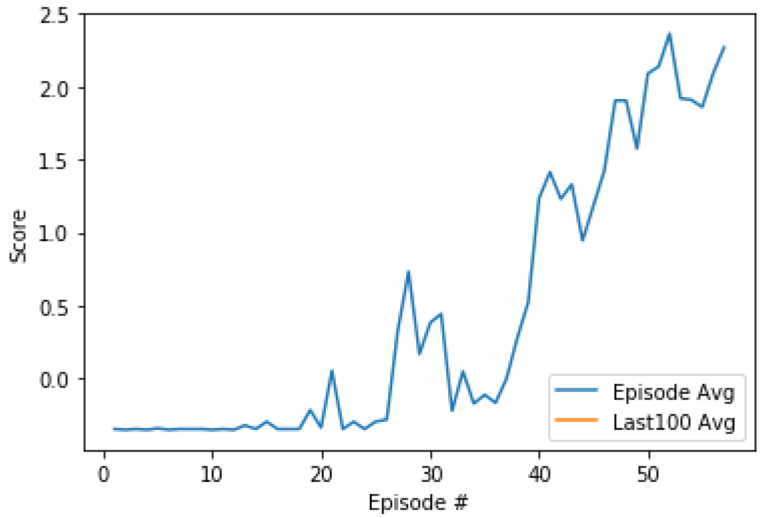
```
target_update = int(1)           # Number of steps after which to update target network (Soft update for Actor & Critic)\n",
tau = 1e-3                       # Soft Update interpolation parameter\n",
```

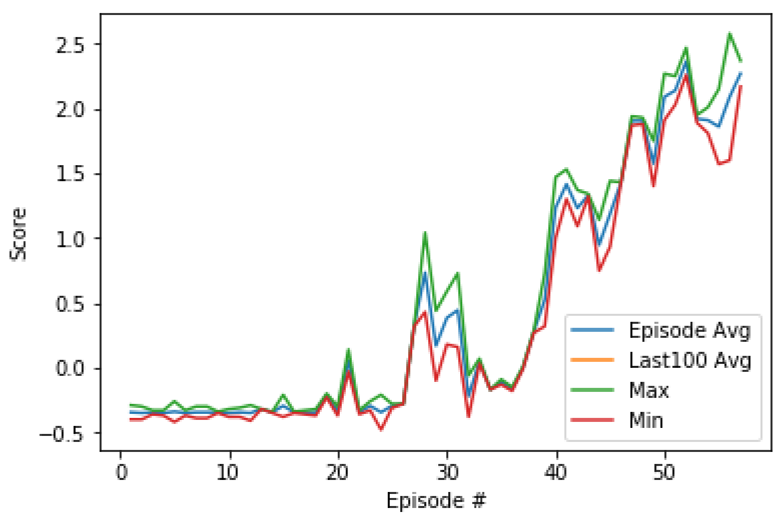
These 2 parameters are related to when (as in how often) and how do we update the 4 Networks: The current and target networks for each the Actor and the Critic. In my last modifications I decided to make a soft update every time we passed all the conditions to "learn"

Results

On my first run and after just making sure my previous DDPG algorithm was adapted for this problem I got very successful results:

```
Episode 57      last 100 avg: 0.52      avg score: 2.27
Solved in 57 episodes!  last100_best_scores_average: 0.52, time taken(min): 19.38560005823771
```





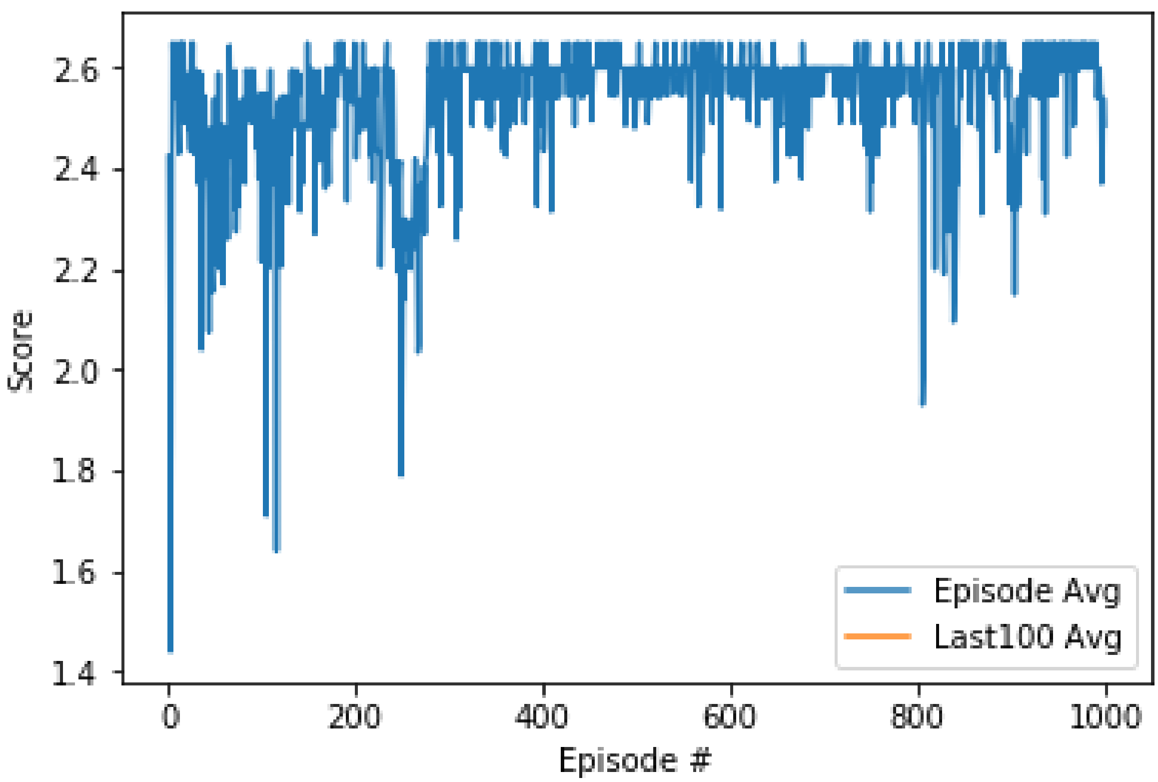
It took only 57 (constant length = 1000 steps) episodes to solve the challenge. That means that the algorithm needed 57,000 steps to solve the challenge

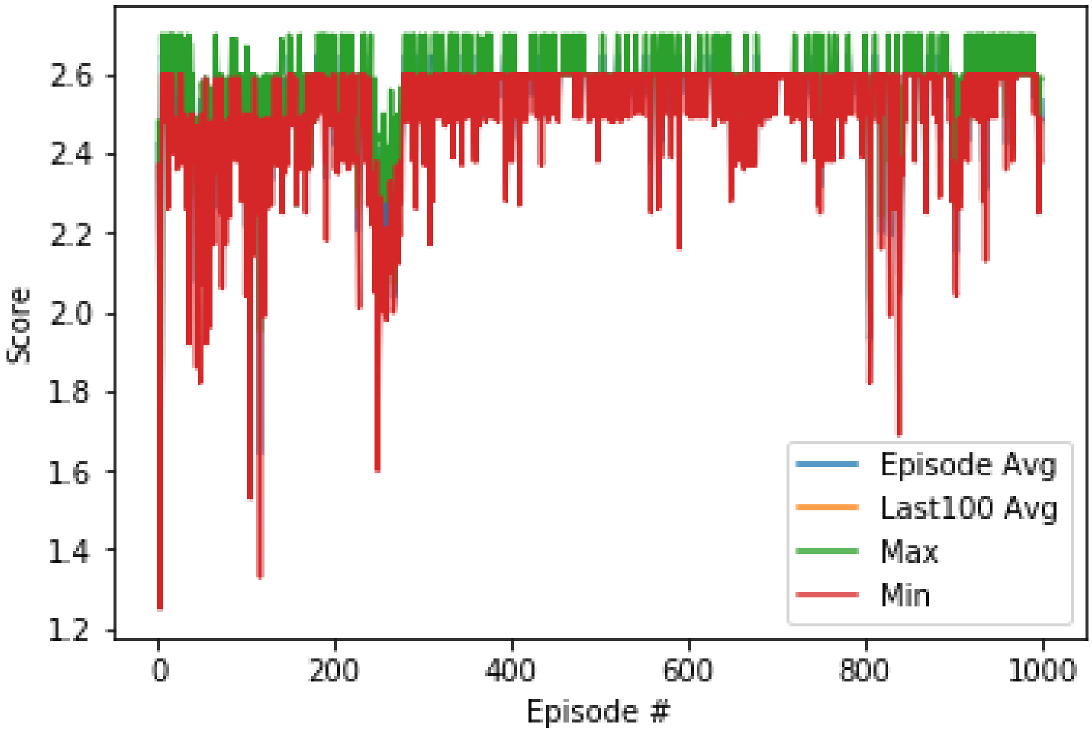
I decided not to stop the training when the challenge was solved (Average for the last 100 “best” agent episodes >= 0.5) and let the algorithm show me if it converges and has a plateau. Here are the results if let run for 1000 episodes at constant length of 1000 steps each episode, I.e 1,000,000 steps taken and used for learning: (Note: weights were pre-loaded from previous test that run 100 episodes)

```
Episode 10      last 100 avg: 2.40      avg score: 2.60
Episode 20      last 100 avg: 2.51      avg score: 2.54
Episode 30      last 100 avg: 2.53      avg score: 2.44
Episode 40      last 100 avg: 2.54      avg score: 2.54
Episode 50      last 100 avg: 2.52      avg score: 2.54
Episode 60      last 100 avg: 2.51      avg score: 2.49
Episode 70      last 100 avg: 2.51      avg score: 2.49
Episode 80      last 100 avg: 2.51      avg score: 2.39
Episode 90      last 100 avg: 2.52      avg score: 2.60
Episode 100     last 100 avg: 2.52      avg score: 2.22
Episode 110     last 100 avg: 2.52      avg score: 2.60
Episode 120     last 100 avg: 2.51      avg score: 2.37
Episode 130     last 100 avg: 2.50      avg score: 2.54
Episode 140     last 100 avg: 2.50      avg score: 2.59
Episode 150     last 100 avg: 2.51      avg score: 2.49
Episode 160     last 100 avg: 2.52      avg score: 2.54
Episode 170     last 100 avg: 2.52      avg score: 2.60
Episode 180     last 100 avg: 2.53      avg score: 2.55
Episode 190     last 100 avg: 2.53      avg score: 2.60
Episode 200     last 100 avg: 2.54      avg score: 2.60
Episode 210     last 100 avg: 2.55      avg score: 2.48
Episode 220     last 100 avg: 2.56      avg score: 2.60
Episode 230     last 100 avg: 2.57      avg score: 2.60
Episode 240     last 100 avg: 2.57      avg score: 2.59
Episode 250     last 100 avg: 2.55      avg score: 1.97
Episode 260     last 100 avg: 2.54      avg score: 2.30
Episode 270     last 100 avg: 2.51      avg score: 2.37
Episode 280     last 100 avg: 2.51      avg score: 2.65
Episode 290     last 100 avg: 2.51      avg score: 2.65
Episode 300     last 100 avg: 2.50      avg score: 2.54
Episode 310     last 100 avg: 2.50      avg score: 2.49
Episode 320     last 100 avg: 2.51      avg score: 2.60
Episode 330     last 100 avg: 2.52      avg score: 2.65
Episode 340     last 100 avg: 2.52      avg score: 2.55
Episode 350     last 100 avg: 2.54      avg score: 2.55
Episode 360     last 100 avg: 2.56      avg score: 2.49
Episode 370     last 100 avg: 2.59      avg score: 2.60
Episode 380     last 100 avg: 2.60      avg score: 2.60
Episode 390     last 100 avg: 2.60      avg score: 2.60
```

```
Episode 400     last 100 avg: 2.60      avg score: 2.60
Episode 410     last 100 avg: 2.60      avg score: 2.60
Episode 420     last 100 avg: 2.60      avg score: 2.60
Episode 430     last 100 avg: 2.60      avg score: 2.60
Episode 440     last 100 avg: 2.60      avg score: 2.60
Episode 450     last 100 avg: 2.60      avg score: 2.60
Episode 460     last 100 avg: 2.60      avg score: 2.60
Episode 470     last 100 avg: 2.60      avg score: 2.60
Episode 480     last 100 avg: 2.60      avg score: 2.60
Episode 490     last 100 avg: 2.60      avg score: 2.60
Episode 500     last 100 avg: 2.60      avg score: 2.55
Episode 510     last 100 avg: 2.61      avg score: 2.55
Episode 520     last 100 avg: 2.61      avg score: 2.65
Episode 530     last 100 avg: 2.61      avg score: 2.60
Episode 540     last 100 avg: 2.60      avg score: 2.55
Episode 550     last 100 avg: 2.60      avg score: 2.60
Episode 560     last 100 avg: 2.60      avg score: 2.60
Episode 570     last 100 avg: 2.60      avg score: 2.55
Episode 580     last 100 avg: 2.60      avg score: 2.60
Episode 590     last 100 avg: 2.60      avg score: 2.32
Episode 600     last 100 avg: 2.60      avg score: 2.60
Episode 610     last 100 avg: 2.60      avg score: 2.60
Episode 620     last 100 avg: 2.60      avg score: 2.50
Episode 630     last 100 avg: 2.60      avg score: 2.65
Episode 640     last 100 avg: 2.60      avg score: 2.60
Episode 650     last 100 avg: 2.60      avg score: 2.60
Episode 660     last 100 avg: 2.60      avg score: 2.60
Episode 670     last 100 avg: 2.60      avg score: 2.55
Episode 680     last 100 avg: 2.60      avg score: 2.49
Episode 690     last 100 avg: 2.59      avg score: 2.60
Episode 700     last 100 avg: 2.59      avg score: 2.60
Episode 710     last 100 avg: 2.59      avg score: 2.60
Episode 720     last 100 avg: 2.59      avg score: 2.60
Episode 730     last 100 avg: 2.59      avg score: 2.55
Episode 740     last 100 avg: 2.59      avg score: 2.60
Episode 750     last 100 avg: 2.59      avg score: 2.60
Episode 760     last 100 avg: 2.59      avg score: 2.60
Episode 770     last 100 avg: 2.59      avg score: 2.55
Episode 780     last 100 avg: 2.59      avg score: 2.55
Episode 790     last 100 avg: 2.60      avg score: 2.60
Episode 800     last 100 avg: 2.60      avg score: 2.65
Episode 810     last 100 avg: 2.58      avg score: 2.60
Episode 820     last 100 avg: 2.58      avg score: 2.55
Episode 830     last 100 avg: 2.58      avg score: 2.55
Episode 840     last 100 avg: 2.56      avg score: 2.19
Episode 850     last 100 avg: 2.56      avg score: 2.60
Episode 860     last 100 avg: 2.57      avg score: 2.65
Episode 870     last 100 avg: 2.57      avg score: 2.60
Episode 880     last 100 avg: 2.57      avg score: 2.60
```

```
Episode 890     last 100 avg: 2.57      avg score: 2.60
Episode 900     last 100 avg: 2.57      avg score: 2.43
Episode 910     last 100 avg: 2.57      avg score: 2.54
Episode 920     last 100 avg: 2.57      avg score: 2.55
Episode 930     last 100 avg: 2.58      avg score: 2.65
Episode 940     last 100 avg: 2.59      avg score: 2.60
Episode 950     last 100 avg: 2.60      avg score: 2.60
Episode 960     last 100 avg: 2.60      avg score: 2.65
Episode 970     last 100 avg: 2.60      avg score: 2.60
Episode 980     last 100 avg: 2.60      avg score: 2.65
Episode 990     last 100 avg: 2.60      avg score: 2.65
Episode 1000    last 100 avg: 2.60      avg score: 2.49
```





Even though the plots are not very clear due to the "clother" of the amount of data plotted, we can still clearly see that Agent converges to a max of 2.60 rewards points very early in about 200 episodes (recall that the agent was pre-loaded with weights already from 100 episodes training)

This plateau is simply due to the pre-fixed time length of the game. After certain time the game is over, otherwise, the agents would highly likely play "forever" without dropping the ball.

Please refer to the video included in this repository (/Video.Tennis.mp4) for more graphical behavior of the agents in action.

Ideas for Future Work

- I truly wanted to explore a multi-thread approach.
- I definitely need to Implement other algorithms to:
    - be able to compare performances.
    - Understand better the true nature of multi-agent problems with other more complex "reward" functions that might show all; competitive, collaborative (like in this case) and mixed behaviors.
- Implement "Priority Replay Memory" with a "priority" criteria that fits this continous actions and compare performances with normal (random sample) replay memory

- Investigate further the effects of other hyper parameters including: batch size, learning_starts_ratio and learning_frequency parameters described above.