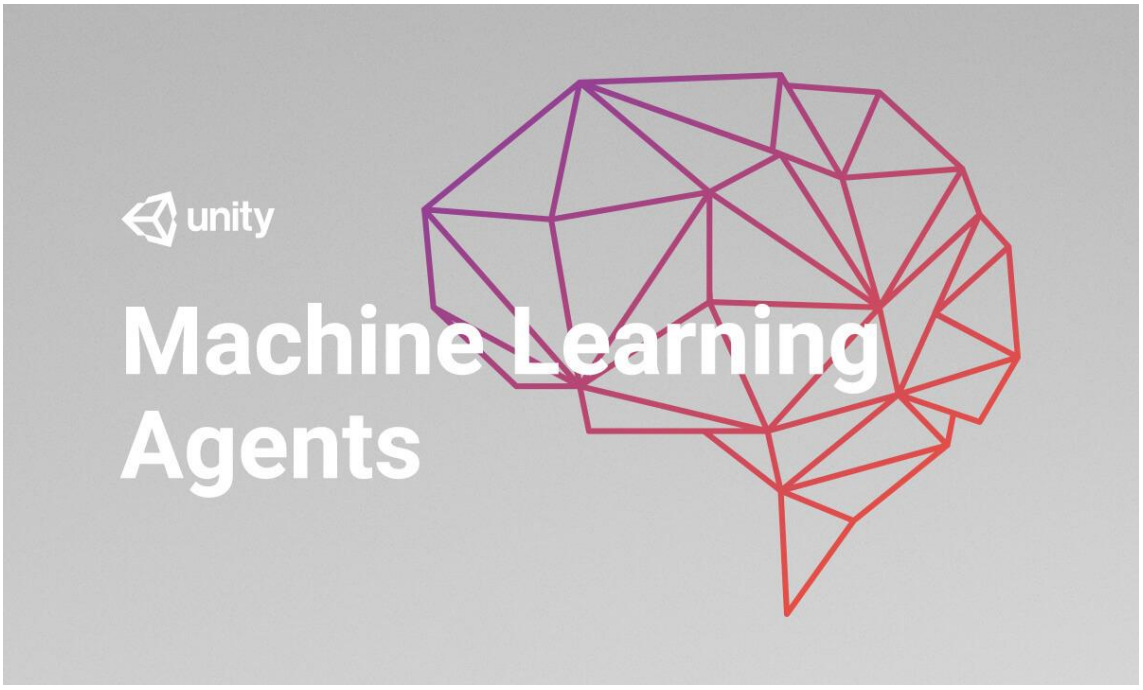
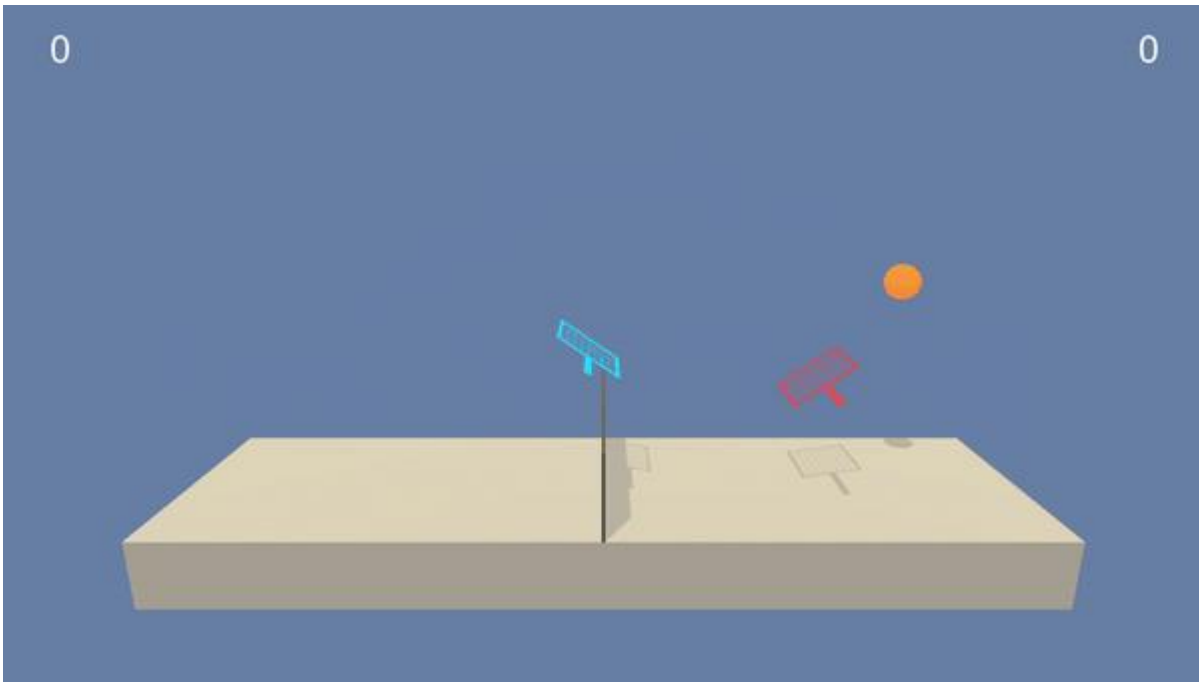
DEEP REINFORCEMENT LEARNING NANODEGREE



Tennis in Unity environment with Udacity



Introduction

For this project, I have worked with the Tennis environment.

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**. Hence it is a **collaborative** task.

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.

Task

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,

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.

Decomposition of the learning process

We call the reset method of environment class for the first time, it returns a state

We call the act method from the agent class which infer an action randomly from the local actor-network

We call the step method of the environment passing it the action previously inferred, which returns a new observation (next state, reward, done)

Focus on the step method, what happens there:

We are training the model from previous experience. We are picking this experience from a fixed size memory that we need to first fill with observations. We need to push at least 256 batches simultaneously into the network, so we need to wait for the replay memory to be filled with more than 256 observations (of 20 agents) before beginning to learn.

The solution we are using here with our DDPG agent is the actor-critic method. It signifies we are training two different architecture; one is a value-based method and the second one in a *policy-based* method. The value-based method is the critic agent and is used to return a value taking as input a state action tuple. Then this information is used to calculate the actor loss and make it learn on it.

From a tuple of experience (states, actions, rewards, next_states, dones), the next action is inferred from the actor target (policy-based method) using the next states, this next action is fed into the

critic target (value-based method) along with the next state and return the Qtarget next. Q target next is then used to calculate the Q target at the current state

We are then backpropagating the critic loss between Q_expected and Q_target to make the critic local network converge to the solution.

In the update actor, we are only inferencing on the local actor (policy-based method) passing it the current states. Then we use the previously trained critic local network to infer the Q local value. We are then defining the invert of this value as being the actor_loss that we are backpropagating into the actor local network.

Then through the soft update method, we are slightly updating both the actor target and the critic target to make them a composition of 99% their respective local network parameters and 0.01% identic to their current target network parameters.

Hyperparameters

Buffer size

The buffer size is listing the tuple experience (states, actions, rewards, next_states, dones) but has a fixed size. During all the training process (140 episodes), with 20 agents, nearly 2 700 000 experience tuples are generated. If we choose a fixed buffer size of 10e5, the buffer will be filled out until it reaches its limit. This limit passed, the buffer will only keep the 100 000 most recent tuple which means toward the end of the training the network will no longer train on the first tuple. With a buffer size of 1e6, the buffer keeps a significant amount of the total experience and can sometimes pick ancient ones but will never keep them all and eventually will get rid of around half of the whole experience. With a buffer size of 1e7, all the experiences are kept in the buffer. The problem with this last one is that it will be drawn among all those experiences and will eventually have a hard time to converge. This situation is where prioritized experience replay is becoming unavoidable. As long as we are not using prioritized exploration, a buffer size of 1e6 seems to be a good trade-off between exploration and short-term memory.

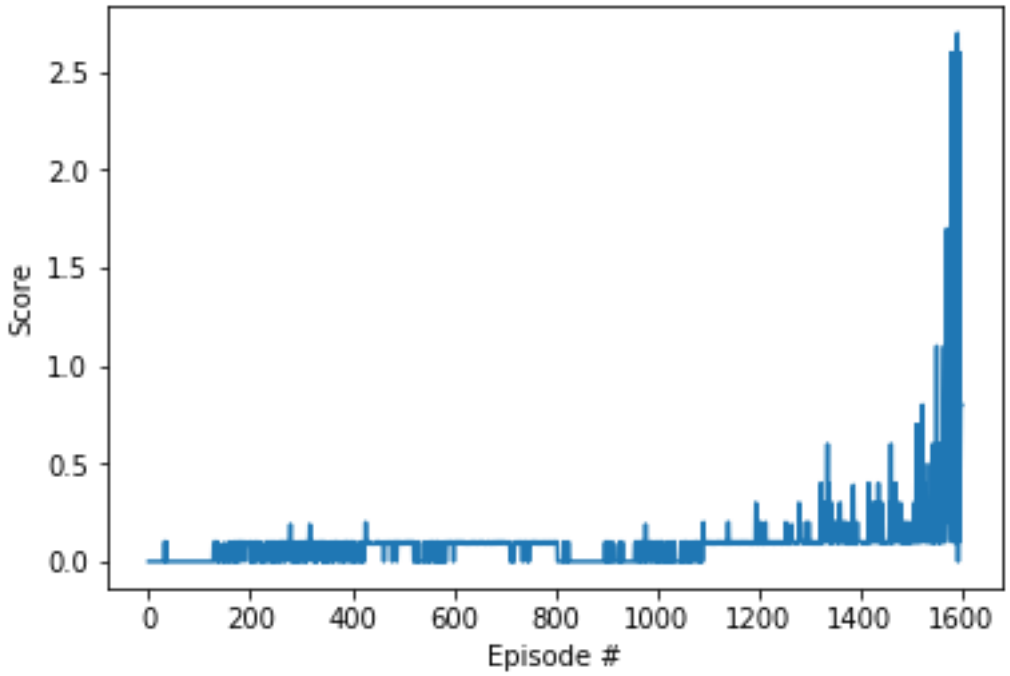
Batch size

The batch size will characterize the amount data input being flowed through the network simultaneously. Higher batch size will make the agent converge faster at each time step, but episodes themselves will take longer to train. With small batch size, the elapsed time between each episode will be short, but it will take more episodes to reach maximal score. Batch size of 128 and 256 seemed to be a good trade-off regarding efficiency.

Gamma

Every 20 step the learning method of the agent is called. This method first makes converge the local network which is then used to soft update the target ones. Every 20 steps, the target networks are defined to be 99% of the local network's parameters and 1% of there previous parameters states. Gamma characterizes these melting parameters. Here we want to make converge the target as smoothly as possible to get as much stability as we can. Here again, the difficulty is to find the perfect trade-off to make the agent learn

Plot of Rewards



Reward as a function of episodes for a 2 agents' environment

Update every 20 steps, 10 times in a row

```
Environment solved in 1501 episodes! Average Score: 0.50
```

Ideas for future works

Prioritized experience replay

The first idea for future work would be to add prioritized experience replay. It should drastically accelerate the learning process by pseudo-randomly selecting experience based on the loss they caused.

Ornstein - Uhlenbeck Process generalized to weights

It has been shown that the noise we add to the actions inferred in the act method to help the model explore and generalize was even more efficient if diluted directly into the network weights themselves.

MADDPG

Here the agent is trained using DDPG. MADDPG is an **extension of DDPG** with an actor-critic architecture where the **critic** is **augmented with other agents' actions**, while the actor only has **local information**.

In other words, this structure turns the method into a **centralized training with decentralized execution**. However this method was difficult to generalize and to make converge.