Deep Reinforcement Learning Nanodegree Project 3 - Collaboration and Competition

Jean Salvi DUKUZWENIMANA

June 26 ,20201

Abstract

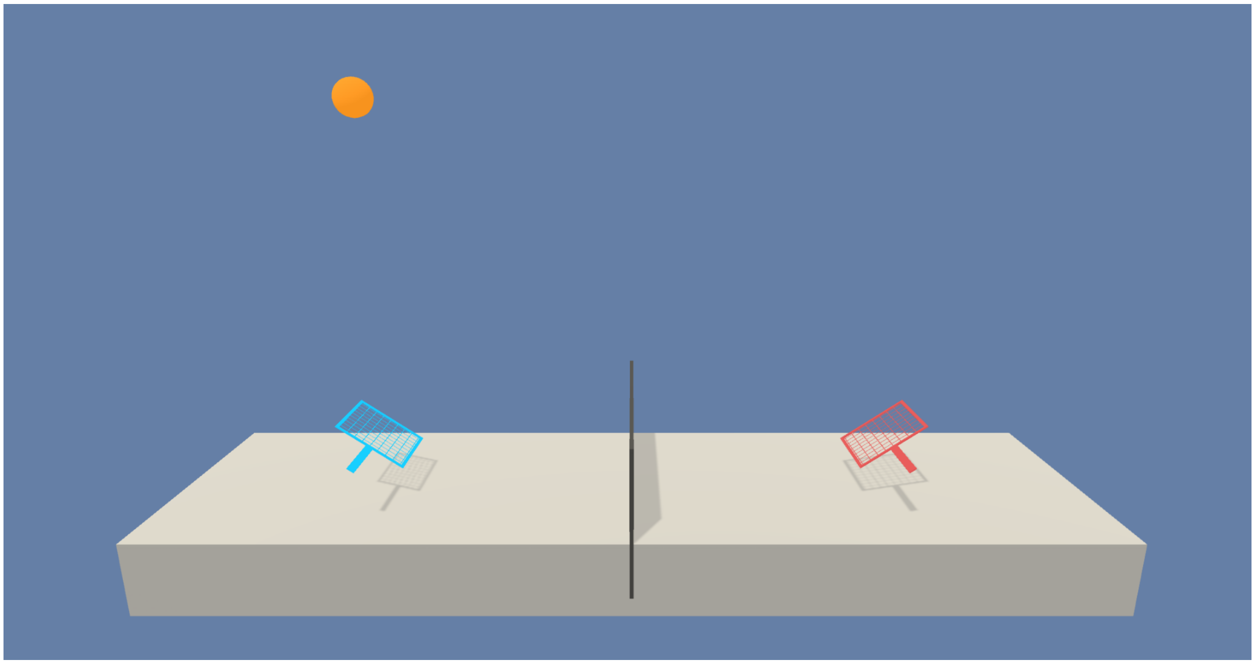
In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in 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, our 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.



Learning algorithm

Agent implementation: Deep Deterministic Policy Gradient (DDPG)

For this project I implemented an off-policy method called Deep Deterministic Policy Gradient, you can read more about it in this paper: Continuous control with deep reinforcement learning.

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. It combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network). It uses

Experience Replay and slow-learning target networks from DQN, and it is based on DPG, which can operate over continuous action spaces.

By combining the actor-critic approach with the Deep Q Network method, the algorithm uses two networks:

- the Actor network, which proposes an action given a state
- the Critic network, which predicts if the action is good (positive value) or bad (negative value) given a state and an action.

These two networks are deployed alongside 2 more techniques:

- 2 Target networks, which add stability to training by learning from estimated targets. The Target networks are updated slowly, hence keeping the estimated targets stable.
- Experience Replay, by storing a list of tuples (state, action, reward, next_state), and instead of learning only from recent experience, the agent learns from sampling all of the experience accumulated so far.

In addition to this implementation of the DDPG algorithm, I experimented with and without increasing/decreasing two learning parameters, gamma (the discount factor for expected rewards) and tau (the multiplicative factor for the soft update of target weights).

Increasing gamma after each episode:

```
gamma(n + 1) = gamma_final + (1 - gamma_rate) * (gamma_final -
gamma(n))
```

Decreasing tau after each episode:

```
tau(n + 1) = tau_final + (1 - tau_rate) * (tau_final - tau(n))
```

Learning stability has been way better with this use of dynamic parameters, allowing to get higher and higher average scores even after many episodes of training.

Code implementation

The code is organized in three files:

model.py:This file contains the Actor and the Critic class and each of them are then used to implement a "Target" and a "Local" Neural Network for training/learning.

Here are the Actor and Critic network architectures:

The actor model is a simple feedforward network:

- **First layer** : input size =24 and output size = `256`
- **Second layer** : input size = `256` and output size = `100`
- **Third layer** : input size = `100` and output size = 2

The critic model:

- **First layer** : input size = 24 and output size = `256`
- **Second layer** : input size = `258` and output size = `100`
- **Third layer** : input size = `100` and output size = 1

Ddpg_agent.py: Here you can find three classes, the (DDPG) Agent, the Noise and the Replay Buffer class, and a function called Load_and_test.

The (DDPG) class Agent contains 5 methods:

- constructor, which initializes the memory buffer and two instances of both Actor's and Critic's NN (target and local).
- **step()**, which allows to store a step taken by the RL agent (state, action, reward, next_state, done) in the Replay Buffer/Memory. Every four steps, it updates the target NN weights with the current weight values from the local NN (Fixed Q Targets technique)
- **act()**, which returns actions for given state as per current policy through an Epsilon-greedy selection in order to balance exploration and exploitation in the Q Learning process
- **learn()**, which updates value parameters using given batch of experience tuples in the form of (state, action, reward, next_state, done)
- **soft_update()**, which is used by the learn() method to softly updates the target NN weights from the local NN weights for both Actor and Critic networks.

The ReplayBuffer class consists of Fixed-size buffer to store experience tuples (state, action, reward, next_state, done) and contains these methods:

- **add()**, which adds a new experience tuple to memory
- **sample()**, which randomly sample a batch of experiences from memory for the learning process of the agent
- **len()**, which returns the current size of internal memory

The **OUNoise** class implements an Ornstein-Uhlenbeck Process to add noise to the action output.

The function **Load_and_test** is designed to run pre trained agents by loading the actor and critic weights obtained through the training process shown in the jupyter notebook.

Tennis.ipynb: This is the Jupyter notebook where I trained the agent. These are the steps taken in it:

- Importing the necessary packages
- Examining the State and Action Spaces
- Testing random actions in the Environment
- Training the two DDPG agents
- Plotting the training scores

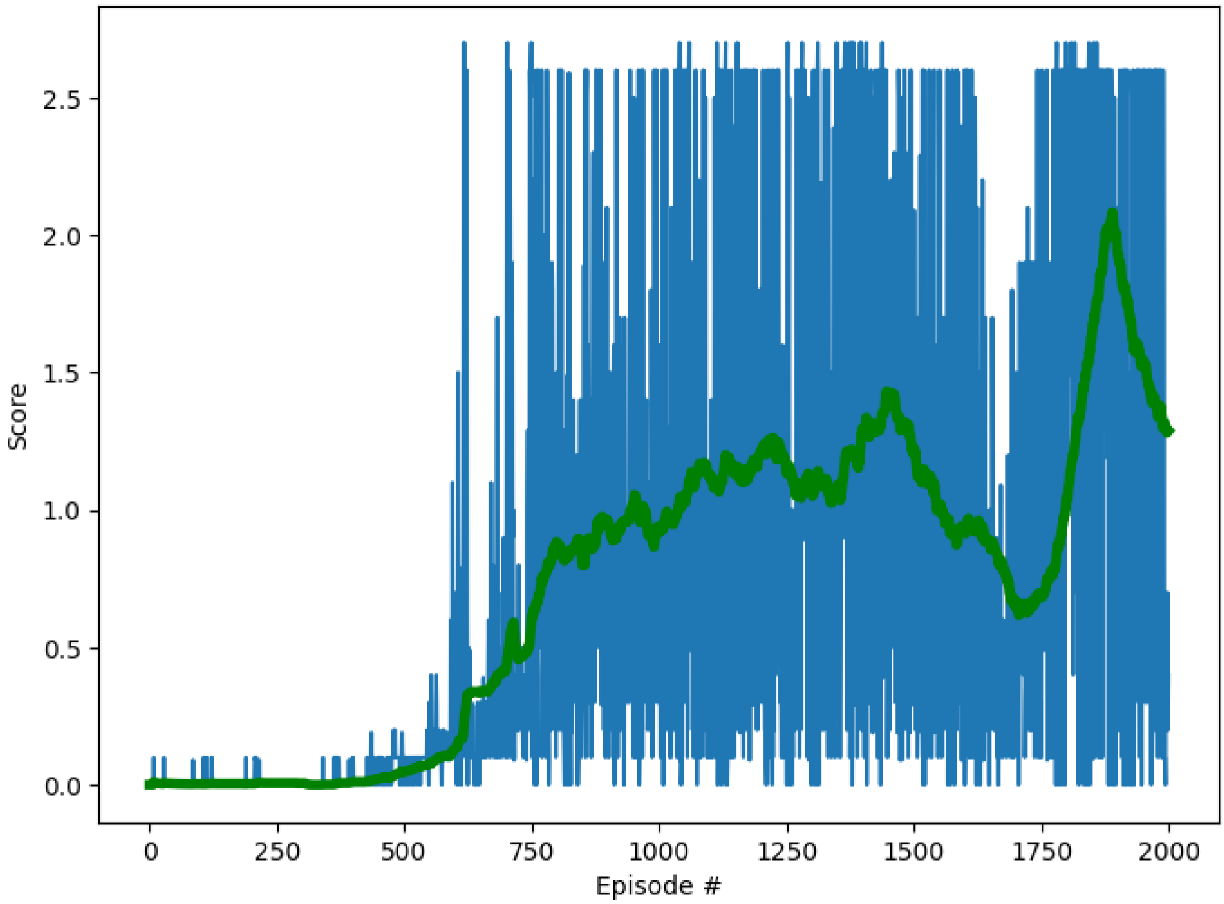
Hyperparameters

Here are the hyperparameters that I used to train the two DDPG agents in the tennis environment:

```
    n_episodes=2000,
    max_t=1500,
    print_every=100,
    gamma = 0.96,
    gamma_final = 0.99,
    gamma_rate = 0.02,
    tau = 0.008,
    tau_final = 0.001,
    tau_rate = 0.001,
    noise_scale = 1.0,
    buffer_size = int(1e6),
    batch_size = 512,
    lr_actor = 1e-3,
    lr_critic = 2e-3,
    noise_theta = 0.12,
    noise_sigma = 0.07,
    update_every = 15,
    num_updates = 12
```

Results

Here is the evolution of the score per episodes:



After 707 episodes the 100 period moving average reached a score of 0.5154, getting above the challenge's goal of at least +0.5.

The highest value of the average score has been achieved after 1890 episodes with a score of 2.0827.

Possible Improvements

Here are some ideas on further developments of the algorithm, beyond simply playing around with the presented architecture and hyperparameters tuning.

- Distributed Distributional Deterministic Policy Gradients

- Sample Efficient Actor-Critic with Experience Replay
- A2C - Asynchronous Methods for Deep Reinforcement Learning
- A3C - Asynchronous Methods for Deep Reinforcement Learning
- Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments