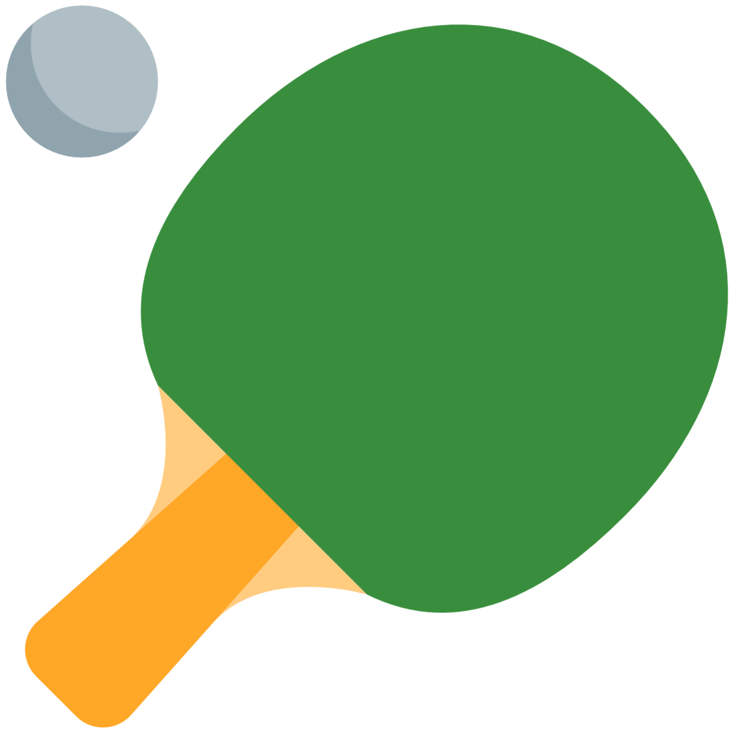
COLLABORATION - COMPETITION PROJECT REPORT

Two Deep Reinforcement Learning agents that collaborate so as to learn to play a game of tennis



Dimitris Poulopoulos

11.08.2018

Udacity Deep Reinforcement Learning Nanodegree

INTRODUCTION

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents.

For game developers, these trained agents can be used for multiple purposes, including controlling NPC behaviour (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

In this project, we develop two reinforcement learning agents to collaborate in a table tennis game, so as to keep the ball in the game as long as possible. 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. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. To solve the environment, the agent must get an average score of +0.5 over 100 consecutive episodes.

LEARNING ALGORITHM

This project creates a DDPG agent, which implements the corresponding algorithm, pioneered by Lillicrap, Timothy P., et al.[1]. We use a wrapper for this agent, that allows us to train two different agents using the same algorithm, leveraging the fact that each agent has its own observations. This network takes as input a state space of 24 variables corresponding to position, and velocity of both the agent (racket) and the ball. Each action is a vector of two numbers, corresponding to movements towards the net of not, as well as movements in the y axis. Then, it passes this information through a relatively simple network architecture, consisting of a few fully connected, hidden layers, with ReLU activation functions, called the Actor. The output and final layer of the network is a *tanh* activation layer, that emits the best believed action for a specific, given state. It consists of a two dimensional action vector, where every number is between -1 and 1.

[1] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971(2015).

The output of the actor, is fed into a second network, called the Critic, which learns to evaluate the output of the Actor. That is to evaluate the optimal action value function using the Actor's best believed action.

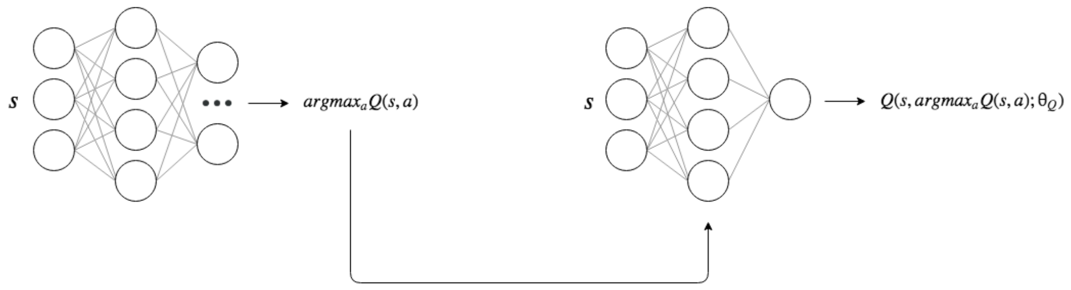


Figure 1 - DDPG Network Architecture

EXPERIENCE REPLAY

Like in the Q-Learning algorithm, to avoid getting swayed by the correlation between a sequence of tuples, we use a technique called *Experience Replay*. Thus, instead of training the agents while they observe the environment and take action, we are keeping track of a *replay buffer* and later, during training, we sample from it at random, breaking those correlations and preventing action values from oscillating or diverging catastrophically. This is possible since each agent has its own observation, thus we can use a “shared” experience replay buffer.

The *replay buffer* is a collection of *experience* tuples - S, A, R, S'. The act of sampling a small batch of tuples from the *replay buffe*r in order to learn is known as *Experience Replay*. Moreover, this technique allows us to reuse individual tuples to learn more or recall rare occurrences.

SOFT UPDATES

In DDPG the target networks are updated using a *soft update* procedure. A soft update strategy consists of slowly blending the values of regular network weights with the target network parameters, like the formula below:

$$w^{-} = \tau w + (1 - t) w^{-}$$

where τ is another hyperparameter, that determines how much of your local networks weights will be mixed with the target network weights.

In this project we use one common Actor network for both agents and two different Critics, corresponding to the number of agents.

RESULTS

The results of the experiment are summarised in table 1 and depicted in figure 2.

EPISODE	AVERAGE SCORE	MAX SCORE
1000	0.03	0.3
1100	0.07	0.5
1200	0.15	2.6
1260	0.51	2.7

Table 1 - Results

To consider the environment solved, the agent must get an average score of +0.5 over 100 consecutive episodes. For the specific network architecture this happens in 1160 episodes according to the results. To achieve this, we built the network with the following hyperparameters:

- Learning rate for Actor: 0.0001
- Learning rate for Critic: 0.001
- Batch size: 64
- Discount rate (γ): 0.99
- Soft update (τ): 0.001
- Actor $ Critic Networks:
    - Hidden layers: 2
        - First hidden layer units: 512
        - Second hidden layer units: 256
- Buffer size: $1 \cdot 10^5$

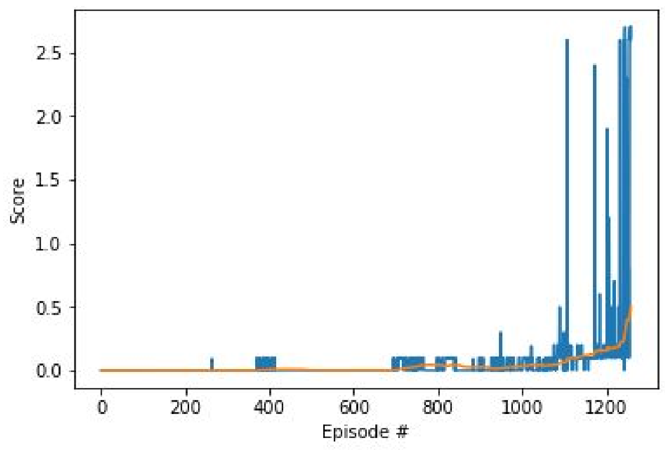


Figure 2 - Results

FUTURE IMPROVEMENTS

For future work, we could leverage another continuous space method, such as NAF [2], which provides a variant of the intuitive Q learning algorithm, for continuous action spaces. Also, ideas described in papers like AlphaGo Zero[3] can also be used in this project, as the agents learn using self-play. We could finally examine a GAE implementation[4], to leverage the n-step bootstrapping notion, that sometimes yields superior results.

[2] Gu, Shixiang, et al. "Continuous deep q-learning with model-based acceleration." International Conference on Machine Learning. 2016.

[3] Silver, David, et al. "Mastering the game of Go without human knowledge." Nature 550.7676 (2017): 354.

[4] Schulman, John, et al. "High-dimensional continuous control using generalized advantage estimation." arXiv preprint arXiv:1506.02438 (2015).