Collaboration and Competition with Deep Reinforcement Learning

Ismail Geles

May 2020

Abstract

The third project of the Udacity Deep Reinforcement Learning Nanodegree is about playing tennis. The Unity environment has two tennis rackets playing against each other, which both are controlled by a DDPG agent and trained via self-play.

1 The Environment

The objective of both agents controlling the rackets is to keep the ball in the play, which means avoiding the ball touching the ground or hitting it out of bounds. If an agent hits the ball over the net it receives a reward of +0.1, however if the ball hits the ground or hits out of bounds it receives a reward of -0.01.

1.1 States and Actions

The observation space consists of 24 dimensions and contains the position and velocity of the ball and racket (for each agent).
Given this information, the agent has to learn how to best select actions.
The action an agent can take is described by a vector with two real numbers. Every entry in the action vector should be a number between -1 and 1.
There are infinitely many actions available, therefore the action-space is considered continuous.
The task is episodic and is solved when the maximum score of both agents reaches a score of +0.5 over 100 consecutive episodes.

2 Implementation of the DDPG Agent

The codes basic structure was taken from previous tasks of the Nanodegree. A DDPG agent, as described in the initial paper "Continuous control with deep reinforcement learning" (`arXiv:1509.02971`), was used. Both players are controlled by the same actor and critic networks, each having a regular and a target network. The critic is used to approximate the maximizer of Q-Values of the following states. That is why DDPG is a renowned Deep Q-Learning approach for continuous actions spaces. The experiences are stored in a shared replay buffer and updates to the networks are soft.
The expected reward was discounted and with the Ornstein-Uhlenbeck process noise was added to the actions in order to bring in some randomisation.

2.1 Network Architecture

The model, used to solve the environment, has two identical network architectures for the actor and critic (critic has only one output as usual). After trying out some architectures the best results were achieved with two hidden layers consisting of 64 units each.

So to summarize the layer structure of the actor network (in `model.py`):

24 units (input) - 64 units (1st hidden) - 64 units (2nd hidden) - 2 units (output)

The activation function for the hidden layers used, is a Rectified Linear Unit (ReLU) function for the actor network, for the critic it is a leaky-ReLU. The actor networks output is mapped to a hyperbolic tangent function in order to assure values between $[-1, 1]$.

2.2 Network Hyper-parameters

Following hyper-parameters were used for training:

Parameter	Value
buffer size	1e5
batch size	128
gamma (discount factor)	0.99
tau (soft update interpolation)	1e-3
learning rate actor	5e-4
learning rate critic	5e-4
update rate	4
optimization steps	20
weight decay	0.0

Note: after trying out different update rates and weight decay it was better to update the networks at every forth timestep and use no weight decay at all. Every forth timestep 20 optimization steps were performed. An episode is considered finished if the ball hits the ground or it hits out of bounds. Therefore the episodes were shorter at the beginning of training.

3 Training Results

The rewards over the number of episodes can be found below. Notice that the environment was solved at episode 770, following an average over +0.5 for the next 100 episodes.

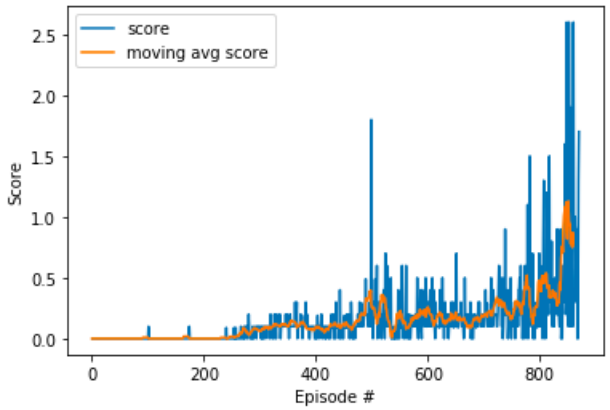

Figure 1: Scores over episodes during training.

4 Ideas for Future Work

There are still many possibilities of network architectures and hyper-parameters to try out for better performance. Another approach to tackle this problem would be solving the environment with other actor-critic methods, like A2C, A3C, D4PG, PPO,..., especially since this environment offers a simple method to control both agents in parallel.