P3: Collaboration and Competition

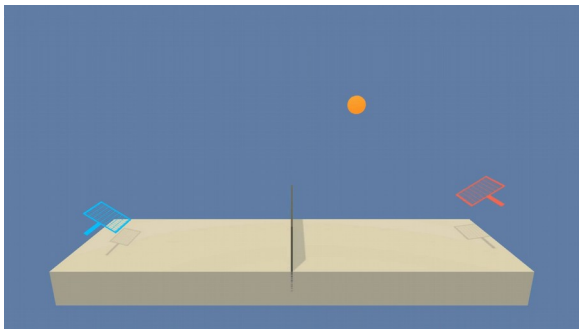
Environment

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.

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.

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.



DDPG Algorithm

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. The algorithm uses most of the techniques of Deep Q-Network (DQN) such as a **Replay Buffer** to store experiences and the use of local and target networks to calculate losses. In this context, the Q-function is known as a **Critic** and the policy function is known as an **Actor**. In DDPG however, the Actor is used mostly to handle actions in a continuous domain, in contrast with other Actor-Critic methods which use the Critic as a baseline to reduce variance for the policy. The policy Actor is a **deterministic** function of states, so it outputs and action directly instead of a probability for each action. Because actions are obtained deterministically, exploration is introduced adding noise to the Actor's output. For the noise, an **Ornstein–Uhlenbeck** process was chosen, as it seems to be standard practice. The DDP algorithm was proposed and is described in detail in reference [1].

MADDPG Algorithm

Multi Agent Deterministic Policy Gradient (MADDPG) is an extension of DDPG for a multi-agent domain. The main difference is that MADDPG is able to incorporate collaboration and competition by allowing each agent to learn a model of the other agents. In this algorithm, each agent has its own independent Actor-Critic pair. For each agent, the **Actor** only has access to its local observations but the **Critic** is a function of the observations and actions of all agents. This allows for complex communication between agents to be taken into account in the policy. The MADDP algorithm was proposed and is described in reference [2].

Implementation

For this project, both DDPG and MADDPG were implemented to solve the environment. A quick comparison is presented in this report.

Model
In both approaches the same model was used. The Agent is a **3-layered** network with **Relu** activation for the hidden dimensions and a **Tanh** activation for the output. The Agent is deterministic, so it receives a state as input and outputs an action. The critic has a similar architecture but actions are being concatenated to the state after the first layer. It also receives a state but outputs a single number (with no activation), which represents the predicted Q-value for that action-state. For the DDPG implementation, BatchNorm1d was added between layers.

First Implementation: DDPG

This implementation used **exactly** the same code as the one used for P2-Continuous Control [3]. Even the hyper-parameters are the same. The code is in turn based on Udacity's ddpg-pendulum example found in [4].

Hyperparameters

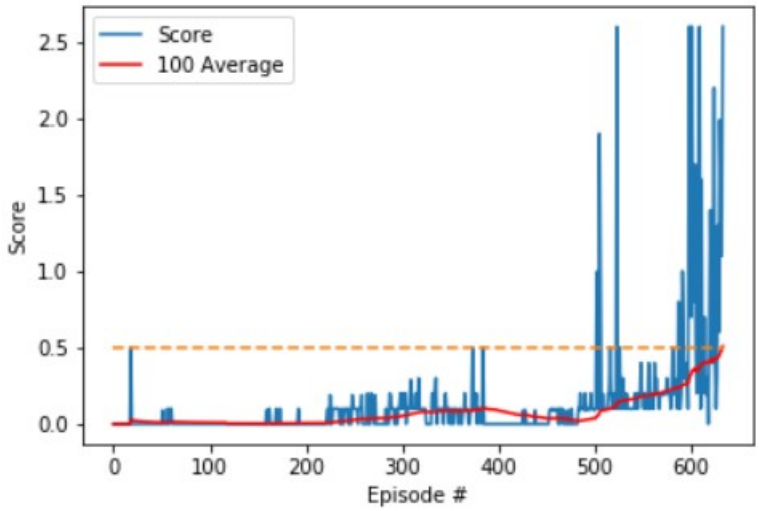
BUFFER_SIZE = int(1e6)	*Replay buffer size*
BATCH_SIZE = 256	*Minibatch size*
GAMMA = 0.99	*Discount factor*
TAU = 1e-3	*For soft update of target parameters*
LR_ACTOR = 1e-3	*Learning rate of the actor*
LR_CRITIC = 1e-3	*Learning rate of the critic*
WEIGHT_DECAY = 0	*L2 weight decay*
EPSILON = 1.0	*Noise decay start value*
EPSILON_DECAY = 1e-6	*Noise decay factor*
GRAD_CLIPPING = 1.0	*Gradient Clipping*
FC1 = 400	*Size of first FC layer*
FC2 = 300	*Size of second FC layer*

Result

Solved in: **634** episodes

Running time (Udacity's workspace): **13.2 minutes**.

Average Score (last 100): **0.51**



Second Implementation: MADDPG

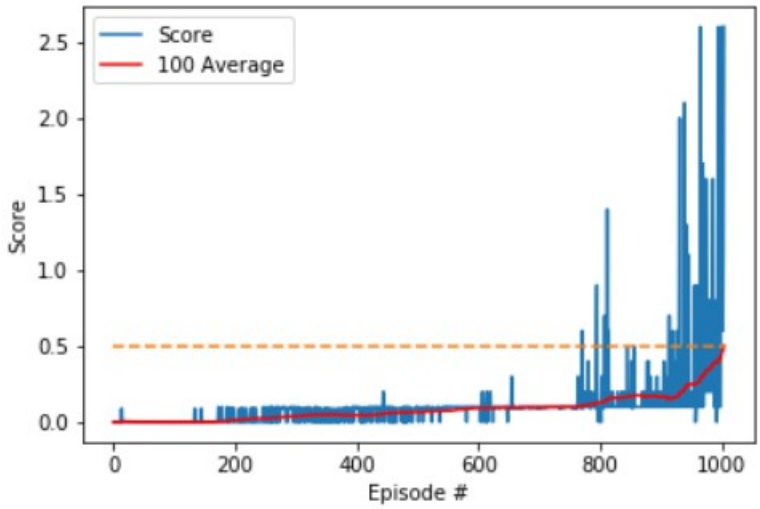
The changes in hyper-parameters for this implementation was crucial. I specially noticed that training took much longer with a higher GAMMA. Also the size of the network was reduced drastically, from FC1=400 to FC1=64 and FC2=300 to FC2=64. A higher value of TAU also sped thing up.

Hyperparameters

BUFFER_SIZE = int(1e6)	*Replay buffer size*
*BATCH_SIZE = **512***	*Minibatch size*
*GAMMA = **0.95***	*Discount factor*
*TAU = **1e-2***	*For soft update of target parameters*
LR_ACTOR = 1e-3	*Learning rate of the actor*
LR_CRITIC = 1e-3	*Learning rate of the critic*
WEIGHT_DECAY = 0	*L2 weight decay*
EPSILON = 1.0	*Noise decay start value*
EPSILON_DECAY = 1e-6	*Noise decay factor*
GRAD_CLIPPING = 1.0	*Gradient Clipping*
FC1 = 64	*Size of first fully connected layer*
FC2 = 64	*Size of second connected layer*

Result

Solved in: **1004** episodes
Running time (Udacity's workspace): **27.6 minutes**.
Average Score (last 100): **0.50**



Comparison

The first thing to notice is that DDPG solved the environment much quicker. Roughly in half the episodes and half the time. This could be because my MADDPG implementation is not great, but I think it makes sense for this environment given that is highly symmetrical. However, a direct comparison is difficult because the hyper-parameters used were different. The ones I used for DDPG didn't worked in MADDPG.

Ideas for Future Work

It should be very interesting to try different algorithms to solve this environment. Particularly I want to try Proximal Policy Optimization (**PPO**), since it's supposed to work well for this case.
Apart from that, playing a bit more with the hyper parameters might make a difference. For example, I didn't touch the noise parameters, which might result in something interesting. I also chose to learn once every time step, because it worked ok, but definitely other behaviors might make an important difference.

References

[1] https://arxiv.org/abs/1509.02971
[2] https://arxiv.org/abs/1706.02275
[3] https://github.com/ccquiel/drlnd-p2-continuous_control
[4] https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum