Project: Collaboration and Competition

Introduction

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, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents).

Learning Algorithm

Environment was solved by implementing Deep Deterministic Policy Gradient (DDPG), an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces.

Deep DPG (DDPG) can learn competitive policies using low-dimensional observations (e.g. Cartesian coordinates or joint angles) using the same hyper-parameters and network structure. In many cases, agent can learn good policies directly from pixels, again keeping hyperparameters and network structure constant [ref].

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

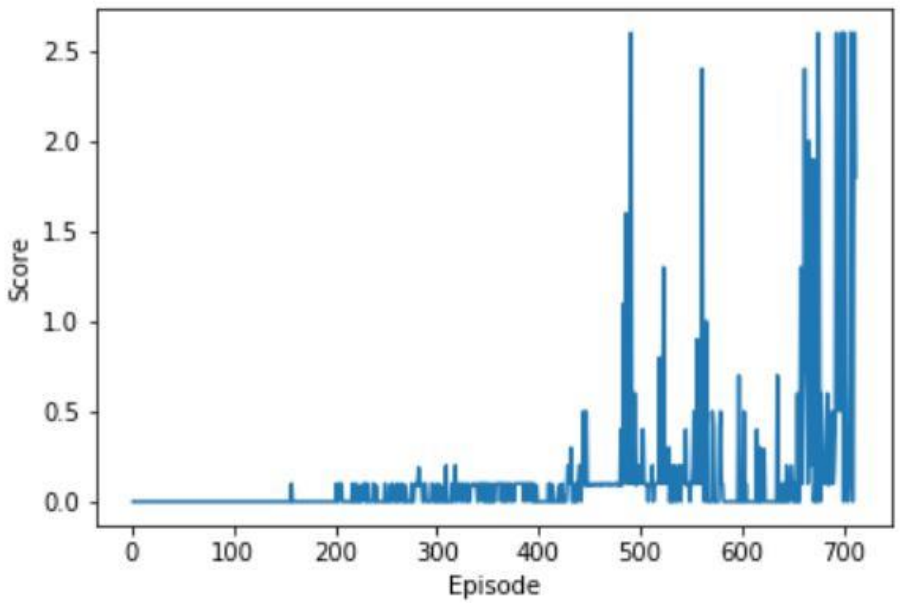
    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

For this project, I used Udacity Reinforcement Learning second project [ref] as starting point and adapted it for multi-agent problem. For efficient fine-tuning, I decided keep both networks simple and work with only two hidden layers, limiting size up to 512 nodes. Experiments show that even slight change in parameters or network architecture could stop training. Many times, training score reached around 0.3 and stopped converging, but adjusting Ornstein-Uhlenbeck parameters and increasing *max_t* (max number of timesteps per episode) helped to reach 0.5 score.

	Agent
Actor	fc1_units=512, fc2_units=256
	super(Actor, self).__init__() self.seed = torch.manual_seed(seed) self.fc1 = nn.Linear(state_size, fc1_units) self.bn1 = nn.BatchNorm1d(fc1_units) self.fc2 = nn.Linear(fc1_units, fc2_units) self.bn2 = nn.BatchNorm1d(fc2_units) self.fc3 = nn.Linear(fc2_units, action_size)
Critic	fc1_units=512, fc2_units=256
	super(Critic, self). __init__() self.seed = torch.manual_seed(seed) self.fc1 = nn.Linear(state_size, fc1_units) self.bn1 = nn.BatchNorm1d(fc1_units) self.fc2 = nn.Linear(fc1_units+action_size, fc2_units) self.fc3 = nn.Linear(fc2_units, 1) self.reset_parameters()
HP	BUFFER_SIZE = int(1e7) BATCH_SIZE = 512 GAMMA = 0.99 TAU = 1e-3 LR_ACTOR = 1e-3 LR_CRITIC = 1e-3 WEIGHT_DECAY = 0 OU_THETA = 0.35 OU_SIGMA = 0.25
Solved in	712
Time	37 mins



Plot 1: Agent 1 plot of rewards

Ideas for Future Work

- Implement other algorithms like PPO, A3C, and D4PG;
- Try more bigger and complex model architectures;
- Increase number of agents.