Collaboration and Competition

Ke Wang

Dec. 2018

1 Introduction

In this project, a multi-agent DDPG (deep deterministic policy gradient) algorithm was developed to solve a tennis in Unity environment.

1.1 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.

1.2 Solving criterion

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). 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.

2 Algorithm

I adopted a Multi-Agent Actor-Critic algorithm described in Algorithm 1.

Algorithm 1 Multi-Agent Actor-Critic

```
1: Randomly initialize the weights for local critic network and local actor network
2: Randomly initialize the weights for local critic network and local actor network
3: Initialize replay buffer $R$
4: for $episodei = 1$ to $M$ do
5:     Reset Environment
6:     Initialize a random process N for action exploration
7:     Receive initial observation state $s_1$
8:     for $t = 1$ to $T$ do
9:         Select action at $= \mu + N_t$ according to the current policy and exploration noise
10:        Execute action at and observe reward $r_t$ and observe new state $s_{t+1}$
11:        Store transition $(o[0]_t, o[1]_t, a[0]_t, a[1]_t, r[0]_t, r[1]_t, o[0]_{t+1}, o[1]_{t+1})$ in $R$
12:        Sample a random minibatch of M transitions from $R$
13:        Update the critic networks with transitions $(o[0]_i, o[1]_i, a[0]_i, a[1]_i, r[0]_i, r[1]_i, o[0]_{i+1}, o[1]_{i+1})$
14:        Update the actor[0] policy with transitions $(o[0]_i, a[0]_i, r[0]_i, o[0]_{i+1})$
15:        Update the actor[1] policy with transitions $(o[1]_i, a[1]_i, r[1]_i, o[1]_{i+1})$
16:        Soft update the target networks
```

3 Neural network

3.1 Actor network

I proposed a three-layer fully connected linear neural network for actor network. The first layer is input layer. The hidden and output layers have sizes of 128 and 64. The last layer is the output layer with size 2. The first two layers adopt leaky_relu as their activation function. The last layer adopt tanh as its activation function to bound the output.

3.2 Critic network

I proposed a three-layer fully connected linear neural network for actor network. The first layer is input layer. The hidden and output layers have sizes of 256 and 128. The last layer is the output layer with size 1. The first two layers adopt leaky_relu as their activation function. Note: critic network double the size of actor network, because a critic networks need to process information as double as it for an actor network.

4 Results

The of parameters of the algorithm are:

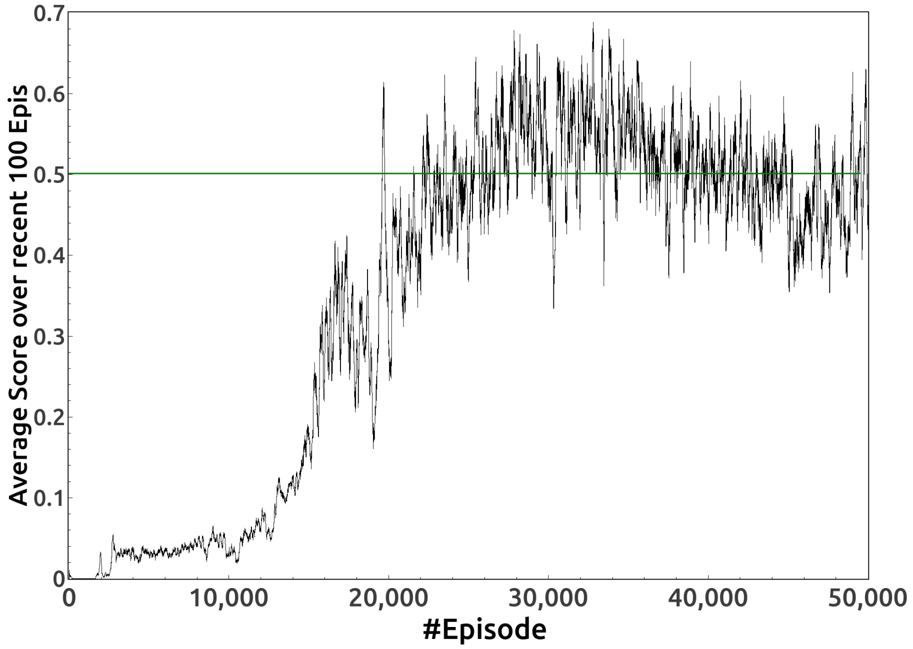

Figure 1: Rewards per episode

- replay buffer size: 100000
- minibatch size: 256
- discount factor: 0.99
- soft update of target parameters: 0.02
- learning rate of the actor: 2e-4
- learning rate of the critic: 1e-4

I notice that the transactions in the replay-buffer strongly affect the convergence of the algorithm. If the learning start before a successful action of a player has been seen, the learning of this player will be dramatically delayed. To avoid this situation, I delay the learning until the buffer has 4×batchsize experience.

The learning curve (a plot of average rewards vs. #episode) is shown in Figure 1. Without fixing a random seed, the problem is solved in the range of 15,000 to 20,000.

The pure testing of trained models (without noise for exploration) shows an average score of 1.94 over 100 episodes.

5 Conclusion

In this project, I developed a multi-agent DDPG (deep deterministic policy gradient) algorithm to solve a tennis problem in Unity environment. The problem is solved after 20,000 episodes.

Next step, I want to add GPU supports. The two players in tennis problem are symmetric. To utilize the symmetry, it is possible to mirror the a transaction $T = (o[0]_t, o[1]_t, a[0]_t, a[1]_t, r[0]_t, r[1]_t, o[0]_{t+1}, o[1]_{t+1})$ and switch players to get a new transaction T'. After getting each transaction T, we can generate a mirror transaction T' and put both T and T' to replay buffer. I think this idea will largely improve the learning speed. I will try this idea later.