Project: Collaboration and Competition

1 Overview

In this report, the deep reinforcement learning algorithm I implemented for the collaboration and competition project is briefly explained. The performance of the trained model and possible future improvement are summarized.

2 Environment and Task

In this project, two agents playing tennis are trained with a deep reinforcement learning algorithm.

For each player, a state is 24-dimensional. This state encodes the positions and velocities of the ball and racket of the player. An action is 2-dimensional and each entry takes a value in $[-1, 1]$. This 2-dimensional action describes the movement toward/away from the net and jump at the location of the player. At each time step, an agent gets a reward $+0.1$ if the agent hits a ball over the net. If an agent lets a ball hit on the ground or hit the ball out of bounds, then the agent receives a reward -0.01.

To see the performance of the trained agents at each episode, the maximum value of the scores (= the sum of rewards in a given episode) among the two players is used. In the following, I will simply call this maximum value by the score for the episode. The environment is regarded as being solved when the average score over the last 100 episodes reaches $+0.5$ or higher. In the rest of this report, I will denote state, action and reward at a given time step for an agent $i(= 1, 2)$ at a time step t by $s_t^{(i)}$, $a_t^{(i)}$ and $r_t^{(i)}$.

3 Learning Algorithm

To train multiple agents at the same time in a collaborative manner, I use the multi-agent deep deterministic policy gradient (MADDPG) algorithm [1]. This is an Actor-Critic-based algorithm for training multiple cooperative or competitive agents. In this part, I provide a brief review of this algorithm and summarize the hyper-parameters used for the training.

3.1 Multi-Agent Deep Deterministic Policy Gradients (MADDPG)

In the standard deep deterministic policy gradient (DDPG) algorithm [2], a single agent is considered, and two neural networks corresponding to the actor and critic models are trained. It is possible to use this standard DDPG for each agent, but, to train multiple agents which either cooperate or compete with each other, it is important for a given agent to take into account the other agents' states and action. In MADDPG, as in the case of the standard DDPG, for each agent, the two neural networks corresponding to the actor and critic models are built. For the actor model, in the same way as the standard DDPG, a state of the agent is feed as an input and an action for the same agent is returned, i.e. $a_t^{(i)} = \pi^{(i)}(s_t^{(i)}; w^{(i)})$ where $\pi^{(i)}(...)$ denotes the policy determined by the actor model of the agent i and $w^{(i)}$ is the weights of the corresponding neural network. For the critic model, on the other hand, states and actions of *all* the agents are feed as an input and a Q-value is returned, i.e. $Q^{(i)}(s_t^{(1)}, s_t^{(2)}, a_t^{(1)}, a_t^{(2)}; \theta^{(i)})$ where $Q^{(i)}(...)$ denotes the Q-value determined by the critic model of the agent i and $\theta^{(i)}$ is the weights of the neural network of the critic model. Thus, through the critic model, each agent takes into account the behaviors of the other agents.

3.2 Update of Networks with MADDPG

The learning process is done as follows for the actor and critic models of each agent. In the same as the Q-learning or DDPG, the target networks are introduced for the actor and critic models (here I add a bar ($\bar{\theta}^{(i)}$ etc.) to denote the weights for the target models). For the agent 1, with a sample of experiences $\{(s_t^{(i)}, a_t^{(i)}, r_t^{(i)}, s_{t+1}^{(i)})\}_{i=1}^2$, the target $y_t^{(1)}$ is computed as

$$y_t^{(1)} = \begin{cases} r_t^{(1)} & \text{if episode ends at time step } t+1\,, \\ r_t^{(1)} + \gamma Q^{(1)}(s_{t+1}^{(1)}, s_{t+1}^{(2)}, \bar{a}_{t+1}^{(1)}, \bar{a}_{t+1}^{(2)}; \bar{\theta}^{(1)}) & \text{otherwise}\,, \end{cases}$$

where $\gamma(= 0.998)$ is the discount factor and $\bar{a}_{t+1}^{(i)} = \pi^{(i)}(s_{t+1}^{(i)}; \bar{w}^{(i)})$. Then, the weights of the critic model for the agent 1 are updated such that the mean of $[y_t^{(1)} - Q^{(1)}(s_t^{(1)}, s_t^{(2)}, a_t^{(1)}, a_t^{(2)}; \theta^{(1)})]^2$ over given samples is minimized. The update of the actor model for the agent i is then carried out such that the mean of $Q(s_t^{(1)}, s_t^{(2)}, a'^{(1)}_t, a_t^{(2)}; \theta^{(1)})$ over the samples is maximized. Here we note that $a'^{(1)}_t = \pi^{(1)}(s_t^{(1)}; w^{(1)})$. The same steps with the agent 1 and the agent 2 exchanged are implemented for training the actor and critic models for the agent 2. For the optimization, I have used the Adam optimizers for the actor and critic models for all the agents (the learning rates are 0.0001 for the actor models and 0.001 for the critic models). In the end of each training for all the agents, the target networks are softly updated as $\bar{w}^{(i)} \leftarrow \tau w^{(i)} + (1-\tau)\bar{w}^{(i)}$ and $\bar{\theta}^{(i)} \leftarrow \tau\theta^{(i)} + (1-\tau)\bar{\theta}^{(i)}$ where $\tau(= 0.1)$ is an hyper-parameter for the soft-update.

To train the model, we introduce the standard replay buffer (buffer size: 100000) which stores the experience tuples (a tuple of state, action, reward, next state and if the episode is done or not) and from which 256 samples are randomly selected in each learning process. To train efficiently at the early stage, it is important to explore the state space broadly. For this purpose, I selected actions randomly (i.e. without using the policy determined by the actor models) for the first 200 episodes without any update of the agents. After this, actions are selected by using the actor models (with noises added. The noises are generated by using the Ornstein-Uhlenbeck process with hyper-parameters $\theta = 0.15, \mu = 0.0, \sigma = 0.2$ in the notation of [3]). The overall scale of the noise decreases by the multiplication factor 0.999 at each time step until it reaches 30% of the initial scale. The learning process is carried out every 1 time step.

3.3 Architecture of Neural Networks

The architectures of the actor and critic model for a single agent are pictorially described in the Fig. 1 below. I stress that each agent has the actor and the critic models with these architectures.

4 Performance

To see the performance of the trained model, I plotted the time evolution of the score as well as that of the average score (over the last 100 episodes) in Fig.2 and Fig.3. The environment is solved at the episode 555 and the average score has reached to more than 0.8 at the episode 596 where I terminated the training.

5 Summary and Comments

In this project, I implemented MADDPG algorithm to train the two agents playing tennis. The agents are trained quickly and the environment has been solved with 555 episodes.

One crucial point for the training is to broadly move around the state space at the early stage. In my implementation, I used the random sampling before starting the updating the weights of the neural networks and the samples are stored in the standard replay buffer for the learning process. It will be useful to implement the prioritized replay buffer with which the agents are expected to be trained more efficiently.

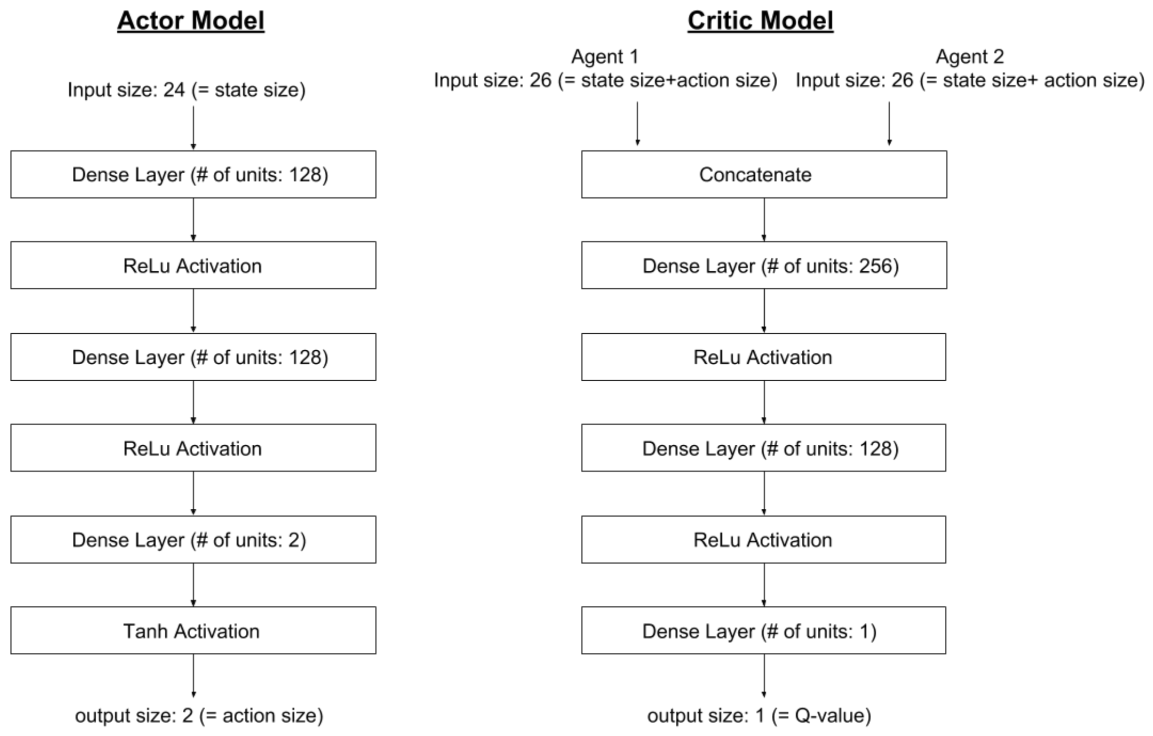
Actor Model
Input size: 24 (= state size)
Dense Layer (# of units: 128)
ReLu Activation
Dense Layer (# of units: 128)
ReLu Activation
Dense Layer (# of units: 2)
Tanh Activation
output size: 2 (= action size)
Critic Model
Agent 1
Input size: 26 (= state size+action size)
Agent 2
Input size: 26 (= state size+ action size)
Concatenate
Dense Layer (# of units: 256)
ReLu Activation
Dense Layer (# of units: 128)
ReLu Activation
Dense Layer (# of units: 1)
output size: 1 (= Q-value)

Figure 1: Architectures of neural networks for the actor and critic models (for each agent)

Another important point is that the training process highly depends on the choice of the random seed(s). It will be valuable to systematically analyze how the seeds can affect the efficiency of the training process.

As can be seen in Fig.2, the score oscillates a lot in the course of the training. In my implementation, the decay of the scale factor of the noise helped the reduction of this oscillation somehow (without this decay, even the agents got a high score at some point, it was hard to keep getting high scores in the next time steps). It will be meaningful to fine tune this decay of the scale factor further in detail.

Finally, as reported by some students in the Slack, because of the symmetric and cooperative nature of the environment, two independent DDPG agents can work well for this project. It will be worthwhile to compare this with the MADDPG implementation quantitatively in detail.

References

1. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments," R. Lowe et. al., arXiv:1706.02275[cs.LG]

2. "Continuous control with deep reinforcement learning," T. P. Lillicrap et. al., arXiv:1509.02971[cs.LG]

3. https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process

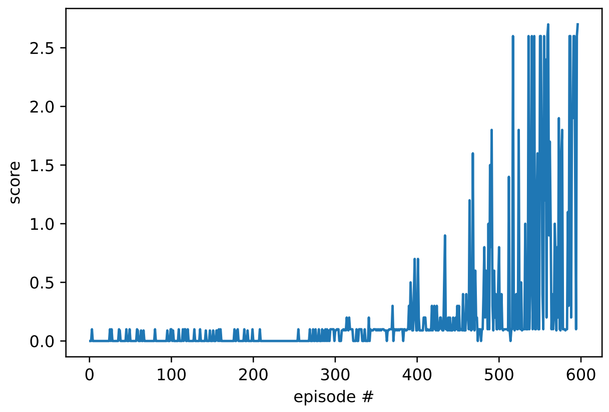

Figure 2: (Maximum) score at each episode.

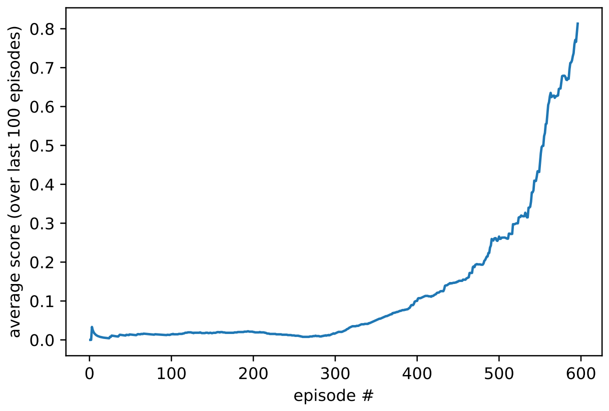

Figure 3: Average score (over the last 100 episodes) at each episode.