[Udacity] Deep Reinforcement Learning Nanodegree Program Project 3 Report: Tennis (Collaboration and Competition)

Hao-En (Hogan) Sung

November 29, 2021

1 Problem Definition

In this project, two agents control rackets to bounce a ball over a net, and the goal is to have them play against each other as many rounds as possible. In other words, both agents should make sure the other one can catch the ball, instead of letting a ball hit the ground or out of bounds.

1.1 Number of states

There are overall 24 states for each agent, which are formed of a stack of 3 continuous frames. In each frame, 8 variables are corresponding to the *position* and *velocity* of the racket in a 2D space.

1.2 Number of actions

There is a vector of 2 actions for each agent, which are corresponding to *moving toward (or away from) the net* as well as *jumping*. Notice that each value in the vector should be a number between -1 and $+1$.

2 Methodology

The algorithm that I implemented is a variant version of **Multi-Agent Deep Deterministic Policy Gradient (MADDPG)**. Just like **Deep Deterministic Policy Gradient (DDPG)**, it maintains two deep learning networks inside it – **Actor** network and **Critic** network. The former network takes in a state in order to figure out the *best* action. As to the latter network, instead of taking the states and actions for a single agent as input, it now takes the shared states and actions (across multiple agents) into consideration and outputs the proper Q-value of that combination. During the training phase, two networks sort of compete against each other and help each other get better and better over time.

In order to make sure the networks are *stable* and *efficient in learning* during the training phase, a few techniques are used, including the **differentiation of the local and target networks**, as well as the use of **replay buffer**. Target networks are the ones fixed when local networks are updated by the gradients. Later, target networks will be slightly modified by a process called **soft-update**. The replay buffer is the same as the one used in **Deep Q-Learning Network (DQN)**, which reduces the duplicate efforts in sampling and allows the models to learn from the past *S*-*A*-*R*-*S* tuples.

2.1 Hyper-parameters

The final set of hyper-parameters were partially adapted the parameters from the previous projects and partially referred from sliao-mi-luku's implementation on GitHub.

- Number of maximum episodes: 10^4

- Maximum number of timestamps in each episode: None
- Batch size: 512
- Replay buffer size: 10^5
- Reward discount factor: $\gamma = 0.99$
- Soft update factor: $\tau = 10^{-3}$
- Learning rate for **Actor** network: 10^{-4}
- Learning rate for **Critic** network: 10^{-3}
- Number of learns when *update* happens: 1
- Number of timestamps before an *update* happens: 1

3 Performance

In the training phase, I was able to reach 0.5081 in 100-episode moving-average score at Episode 1786, as one can see in Table 1 and Figure 1.

# episodes	moving average score
100	0.005757
200	0.000000
300	0.000000
400	0.004040
500	0.000909
600	0.023030
700	0.034040
800	0.079494
900	0.091313

# episodes	moving average score
1000	0.095050
1100	0.098888
1200	0.119292
1300	0.127777
1400	0.150101
1500	0.175050
1600	0.162424
1700	0.251919
1786	0.5081

Table 1: Moving average of the maximum score among two agents per 100 episodes

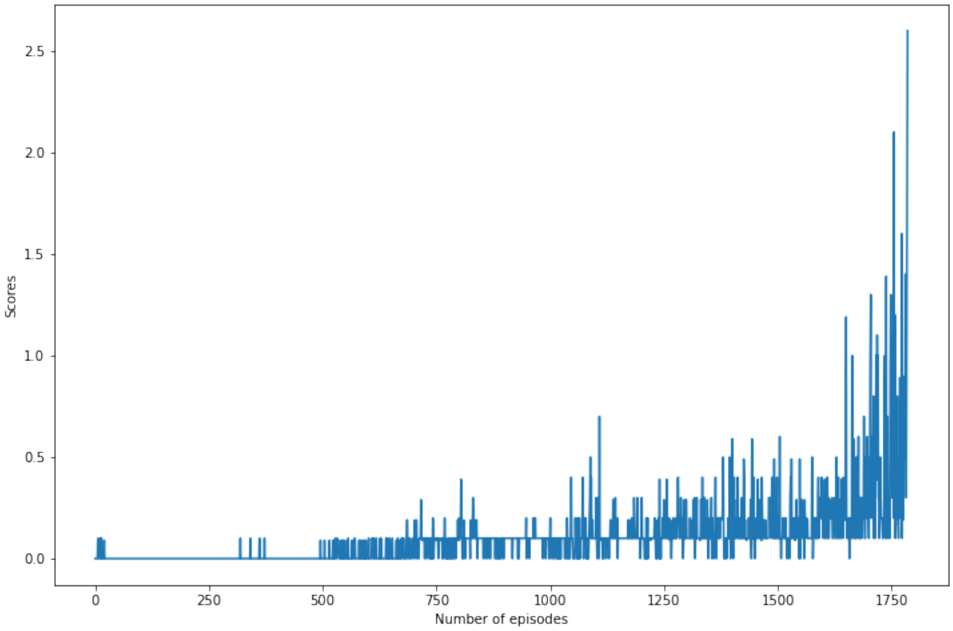

Figure 1: Maximum score among two agents per episode

4 Retrospective: Debug process and Bugs

Even if I have high confidence of my understanding to the algorithm and I had checked my implementations again and again for hours, there were still undiscovered bugs – some were obvious, but some were more hidden.

4.1 Debug process

After I tried out all possible ways of reviewing my implementation myself, I decided to refer to other's work – sliao-mi-luku's implementation. Even though the gist of his implementation is similar to mine, there are still quite significant differences that I need to overcome.

The first step I took was to make minimal changes to his work, and at the same time, I had to make sure his solution still *worked*. Secondly, I fixed every possible random seed, including the one from *random*, *numpy*, and *torch* libraries, in every single file. Later, I was finally able to execute the code step-by-step and cross-compare the values of *states*, *actions*, *model weights*, etc. between two Jupyter notebooks.

The whole debug process approximately took me 8 hours overall. It took longer than expected because some bugs were not very straightforward, so I had to get deeper into the rabbit role.

4.2 Bugs

1. When deciding the next action for agents using the local actor network, I forgot to add an additional *batch* dimension, so the forwarded result was very off.

2. Initially, I decayed the scale of the *Ornstein–Uhlenbeck* noise by 0.9995 per timestamp. However, that should be decayed per episode instead.

3. I made an one-byte mistake in calculating the "predicted q value" – instead of having $\underline{Q = R + \gamma \cdot Q' \cdot (1 - \text{done})}$, I wrote $\underline{Q = R + \gamma \cdot Q' + (1 - \text{done})}$.

4. When updating local actor network, one needs to calculate the predicted agent actions. In the calculation, exactly one of the agents should have gradient enabled. However, I disabled the gradient calculation for both local actor networks. This is the most hidden bug, and itself took me 3+ hours in debugging.

5 Future work

5.1 Increase the stability of the *MADDPG*

Not sure if it is because I didn't implement the exact *MADDPG* algorithm or the algorithm itself is just less stable, the success of my execution relies heavily on the hyper-parameters, especially the learning rate of the critic network. For example, in my experience, using critic models with 10^{-4} learning rate doesn't even converge after 10^4 episodes.

5.2 Implementation of other algorithms

Similar to the future work for the second project, I am now very curious about extending and implementing other algorithms to solve the multi-agent problem, including **Proximal Policy Optimization (PPO)**, **Advantage Actor-Critic (A2C)**, **Asynchronous Advantage Actor-Critic (A3C)**, and even **Soft Actor-Critic (SAC)**. Hopefully, I could finish the rest of the courses and come back to this part as soon as possible.