[Udacity] Deep Reinforcement Learning Nanodegree Program Project 2 Report: Continuous Control

Hao-En (Hogan) Sung

November 22, 2021

1 Problem Definition

In this project, the goal is to make sure the double-jointed arm(s) can move toward the target locations. Depending on the problem setting, there might be one arm or twenty arms moving at the same time in the environment.

1.1 Number of states

There are overall 33 states, which involve the positions, rotations, velocities, and angular velocities of the arm(s).

1.2 Number of actions

There is a vector 4 actions, which are corresponding to torque applicable to the two joints. Notice that each value in the vector should be a number between -1 and $+1$.

2 Methodology

The algorithm that is implemented is called **Deep Deterministic Policy Gradient (DDPG)**, which 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; whereas the latter network takes both the state and action 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-A* tuples.

2.1 Hyper-parameters

For most the parameters, I referred them from examples **ddpg-bipedal** and **ddpg-pendulum** and hortovanyi's implementation on GitHub.

- Number of maximum episodes: 2000
- Maximum number of timestamps in each episode: 1000
- Batch size: 256

- Replay buffer size: 10^6
- 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^{-4}
- Number of learns when *update* happens: 10
- Number of timestamps before an *update* happens: 20

3 Performance

In the training phase, I was able to reach to an average scores of 30 with 34 episodes, as one can see in Table 1 and Figure 1. It is noticeable that the model's average score sort of plateau after episode 11 with scores between 38 and 40.

# episodes	average score
1	0.23
2	0.39
3	0.60
4	0.79
5	1.01
6	1.58
7	2.34
8	4.11
9	6.01
10	8.36
11	10.74
12	13.10
13	15.11
14	16.80
15	18.27
16	19.59
17	20.73
18	21.77
19	22.68
20	23.52
21	24.28
22	24.97
23	25.59
24	26.15
25	26.68
26	27.16
27	27.61
28	28.03
29	28.42
30	28.79
31	29.13
32	29.46
33	29.76
34	30.05

# episodes	average score
35	30.31
36	30.57
37	30.81
38	31.04
39	31.25
40	31.46
41	31.65
42	31.84
43	32.02
44	32.19
45	32.35
46	32.50
47	32.65
48	32.79
49	32.92
50	33.05
51	33.18
52	33.30
53	33.41
54	33.52
55	33.63
56	33.73
57	33.83
58	33.92
59	34.01
60	34.10
61	34.17
62	34.25
63	34.32
64	34.35
65	34.40
66	34.45
67	34.51
68	34.58

# episodes	average score
69	34.62
70	34.68
71	34.74
72	34.79
73	34.84
74	34.89
75	34.94
76	34.99
77	35.04
78	35.09
79	35.14
80	35.19
81	35.23
82	35.27
83	35.32
84	35.36
85	35.40
86	35.44
87	35.48
88	35.52
89	35.56
90	35.59
91	35.64
92	35.67
93	35.71
94	35.75
95	35.79
96	35.82
97	35.85
98	35.88
99	35.91
100	35.94

Table 1: Moving average scores across 20 agents for 100 episodes

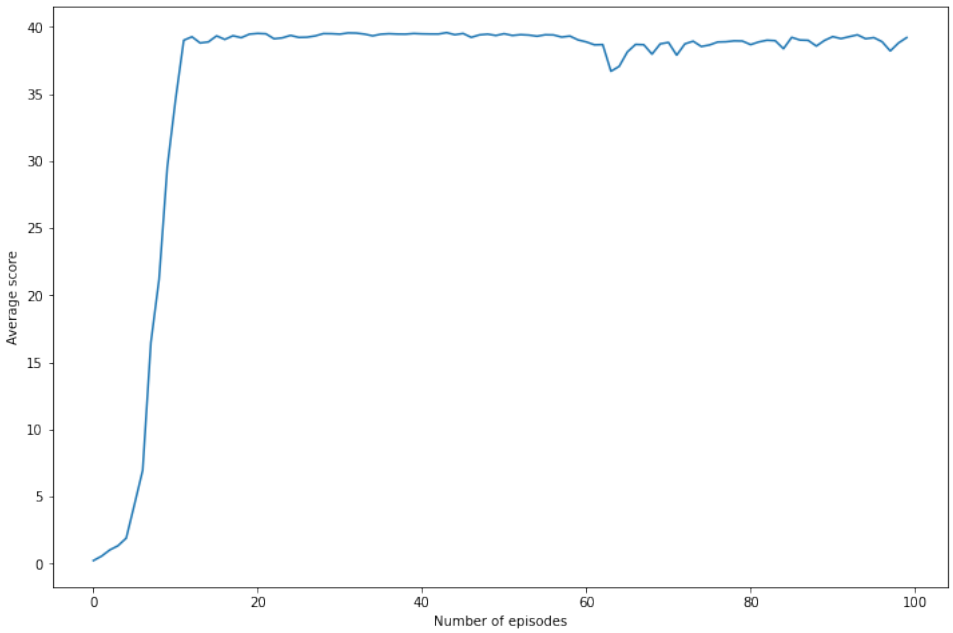

Figure 1: Average score by episodes across 20 agents for 100 episodes

4 Future work

4.1 Extend the DDPG algorithm to run with multi-processors

In my current implementation, even if multiple agents are observing and reacting to different environments, they are still reading and writing the same networks *sequentially*. This is just not efficient. A way to resolve this sequential execution problem is to spawn multi-processors and let each agent interacts with the environment and figures out the gradient individually. Later, one just needs to collect the gradient across processors and update the networks once.

Even though there is a great potential in this direction, I am not sure how easy this could be done. There are mainly two concerns: a) the interface to intersect with the environment requires a collection of actions, so it might require some synchronization here, which could add up lots of overheads and b) even though the inference process in torch networks can be shared across processors, I am unclear if the gradient calculation part is also shareable or not.

4.2 Implementation of other algorithms

I am personally very curious about the implementation of other fancy algorithms, 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.