Project: Continuous-Control

1 Overview

This report explains the deep reinforcement learning algorithm I implemented for the continuous control project. The performance of my trained model and comments for future improvement are summarized. Note that I used the option 2 environment (20 agents) for this project and Deep Deterministic Policy Gradient (DDPG) algorithm for training the agent.

2 Environment and Task

In this project, 20 doubly-jointed robot arms are considered. The goal of this project is to build a model based on deep reinforcement learning algorithm such that the end point of each arm is guided to a target location (which moves inside a 3-dimensional space as time goes).

For each robot arm, a state is 33-dimensional (which encodes position, rotation, velocity and angular velocities of the arm) and an action is 4-dimensional (with each element in the range $[-1, 1]$). At each time, step each arm gets reward $+0.1$ if the arm is located in the target location correctly. The environment is regarded as being solved once the average score over all the agents and over the last 100 episodes becomes $+30$ or larger. In the rest of this report, I denote state, action and reward at a given time step t by s_t, a_t and r_t.

3 Learning Algorithm

In this project, I used the deep deterministic policy gradients (DDPG) algorithm [1] to solve the environment. Here I provide a brief summary of this DDPG algorithm. The hyper-parameters provided for this project are also given here.

3.1 Deep Deterministic Policy Gradients (DDPG)

DDPG is an Actor-Critic-based algorithm in which two neural networks are used for training the agent. The first neural network (actor model) takes a state s as an input and then returns a action, $a = \pi(s; w)$ where w denotes the weights of the actor model. The agent decides which action to take based the output of the actor model with noise added. We note that in this project, the noise was generated by using the Ornstein-Uhlenbeck process (with hyper-parameters $\theta = 0.15, \mu = 0.0, \sigma = 0.1$ in the notation of [2]). Another neural network (critic model) takes both state s and action a as inputs and then returns a Q-value, $Q(s, a; \theta)$ where θ denotes the weight of the critic model.

In the training process, as in the usual Q-learning, the target neural networks for both actor and critic models are introduced (we denote their weights by w^{target} and θ^{target}, respectively). Then the target y_t is computed as

$$y_t = \begin{cases} r_t & \text{if episode ends at time step } t+1\,, \\ r_t + \gamma Q(s_{t+1}, a_{t+1} = \pi(s_{t+1}, w^{target}); \theta^{target}) & \text{otherwise}\,, \end{cases}$$

where $\gamma (= 0.99)$ is the discount factor. The weights of the critic model are updated such that the mean square loss $(y_t - Q(s_t, a_t; \theta))^2$ over samples is minimized. For the actor model, the update of the weight is done such that the mean of $Q(s_t, \pi(s_t; w))$ over samples is maximized, i.e. such that the action selected by the actor model gives a larger Q-value.

As for the optimizer we have used the Adam optimizers for both the actor and critic models (the learning rate is 0.002 (resp. 0.001) for the actor (resp. critic) model). The training is carried out with the standard replay buffer with buffer size 100000 from which 256 experience tuples are sampled randomly at each learning step. We note that a single replay buffer stores experiences from all the robot arms (a experience tuple is a tuple containing state, action, reward, next state and if an episode is done or note). The learning process is implemented 10 times after every 20 time steps. In the learning process, the gradient clipping is applied to the gradient of the critic model such that the norm of the gradient does not exceed 1. At the end of the learning process. the soft-update is applied for the update of the target networks, $w^{target} \leftarrow \tau w + (1-\tau)w^{target}$ and $\theta^{target} \leftarrow \tau\theta + (1-\tau)\theta^{target}$ where $\tau(= 0.001)$.

3.2 Architecture of Neural Network

The architectures of the actor and critic model are given in the Figure 1 below.

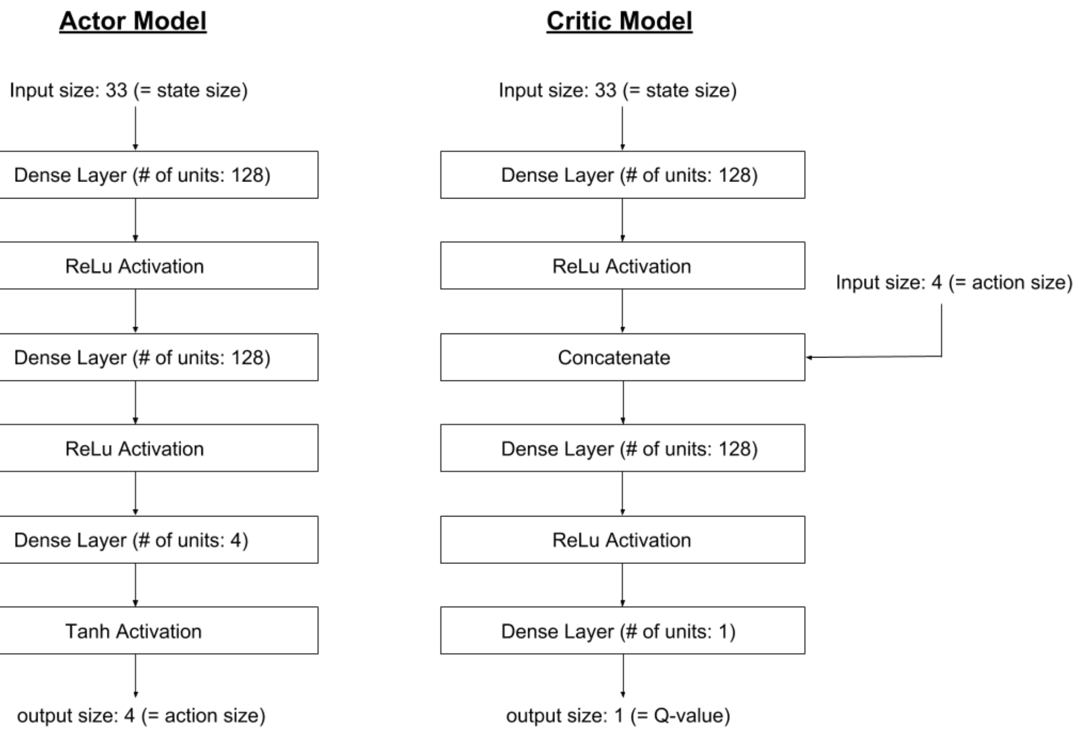
Actor Model
Input size: 33 (= state size)
Dense Layer (# of units: 128)
ReLu Activation
Dense Layer (# of units: 128)
ReLu Activation
Dense Layer (# of units: 4)
Tanh Activation
output size: 4 (= action size)
Critic Model
Input size: 33 (= state size)
Dense Layer (# of units: 128)
ReLu Activation
Input size: 4 (= action size)
Concatenate
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

4 Performance

To see the performance of the trained model, here I provide the plots for the history of the scores. Figure. 2 displays the history of the average scores over all the 20 agents for each episode. One can see that this average score reaches over 30 in less than 40 episodes. Figure. 3 plots the average score over all the 20 agents and over the last 100 episodes. This model solved the environment at the episode 106. The averaged total score (over all the agents, over the last 100 episodes) has reached +36.05 at the episode 125.

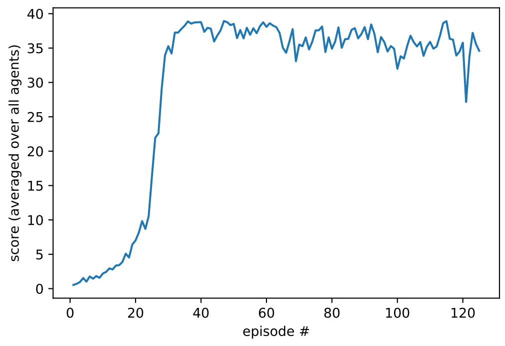

Figure 2: Score averaged over all the agents for each episode

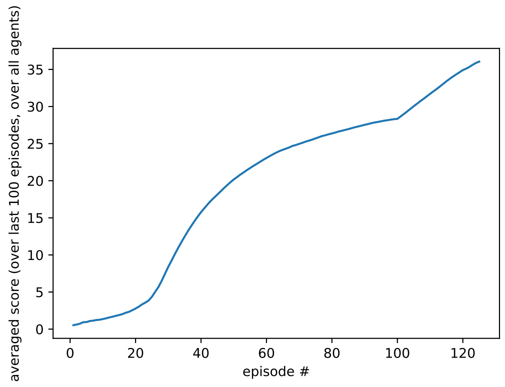

Figure 3: Average score over all the agents and over the last 100 episodes

5 Summary and Comments

In this project, I have built a model for the continuous control task by using DDPG. The model has been trained quickly and the environment has been solved at 106 episodes. For further improvements, it is worthwhile to analyze further in detail the dependence of the training process on the initialization of the weights and choice of the random seed. So far as I tried some examples, whether the training proceeds quickly or not as well as the final performance of the agent highly depends on these quantities. It is important to checked the dependency systematically. Another important direction is to use more improved algorithms such as PPO [3], A3C [4] and D4PG [5]. It is also worthwhile to implement the prioritized replay buffer for more efficient sampling in the learning process.

References

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

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

3. "Proximal Policy Optimization Algorithms," J. Schulman et. al., arXiv:1707.06347[cs.LG]

4. "Asynchronous Methods for Deep Reinforcement Learning", V. Mnih et. al, arXiv:1602.01783[cs.LG]

5. “Distributed Distributional Deterministic Policy Gradients”, G. Barth-Maron et. al., arXiv:1804.08617[cs.LG]