Udacity Deep Reinforcement Learning
Project Continuous Control

Rohil Pal (rohilpal9763@gmail.com)

Abstract

In this project, we have to work with the Reacher enviroment provided by Unity ML Agents[1]. In this environment, we have to control a double-jointed arm to track a target location. We can use any of the policy-based methods taught in the module.

Environment

The **state space** has 33 dimensions each of which is a continuous variable. It includes position, rotation, velocity, and angular velocities o the agent.

The **action space** is a subset of $\mathbb{R}^4$. That is, each action is a vector of four real numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number in the interval $[-1, 1]$.

A reward of $+0.1$ is provided for each step that agent's hand is in the goal location. The goal of the agent is to maintain contact with the target location for as many time steps as possible.

Distributed training

For this project, 2 environments are provided:

- The first version contains a single agent
- The second version contains 20 identical agents, each with its own copy of the environment. This version is particularly useful for algorithms like *PPO*, *A3C*, and *D4PG* that use multiple (non-interacting parallel) copies of the same agent to distribute the task of gathering experience.

Solving the environment

Solve the First Version

The task is episodic, and in order to solve the environment, the agent must get an average score of $+30$ over 100 consecutive episodes.

Solve the Second Version

Since there are more than 1 agents, we must achieve an average score of $+30$ (over 100 consecutive episodes, and over all agents).

Algorithm

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

DDPG interleaves learning an approximator to $Q^*(s, a)$ **(Critic)** with learning an approximator to $a^*(s)$**(Actor)**, and it does so in way which is specifically adapted for environments with continuous action spaces.

Actor and Critic Network Architecture

Both the *Actor* and the *Critic* networks have 2 hidden layers each. In both networks, the 1^{st} layer has 400 neurons while the 2^{nd} layer has 300 neurons. Also, both networks have a batch normalization layer prior to the 1^{st} hidden layers. The final layer in Actor Network is a linear layer with neurons $(= \#actions)$ and *tanh* activation. Critic network has only one neuron in its final layer with no activation. Since, the critic network estimates Q-values for all state-action pairs, we have to somehow insert the corresponding actions at some layer. We do this in the 2^{nd} hidden layer of the Critic network.

Weight Initialization

All layers but the final layer in both the networks are initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The final layer weights in both networks were initialized from a uniform distribution $[-3\times10^{-3}, 3\times10^{3}]$.

Solution

Since, the enviroment contains 20 agents working in parallel, I had to amend the standard single-agent implementation of *DDPG*. As suggested in the *Benchmark implementation* ***(Attempt #4)***, the agents learnt from the experience tuples every 20 timesteps and at very update step, the agents learnt 10 times. Also, gradient clipping as suggested in ***Attempt #3*** helped improved the training. Also, to add a bit of exploration while choosing actions, as suggested in the *DDPG* paper, *Ornstein-Uhlenbeck process* was used to add noise to the chosen actions.

Hyperparameters

- Learning rate (Actor) : $1e^{-4}$
- Learning rate (Critic) : $1e^{-3}$
- Batch size : 1024
- Replay Buffer size : $1e^{6}$
- Discount factor (γ) : 0.99
- Soft Update parameter (τ) : 0.001

Results

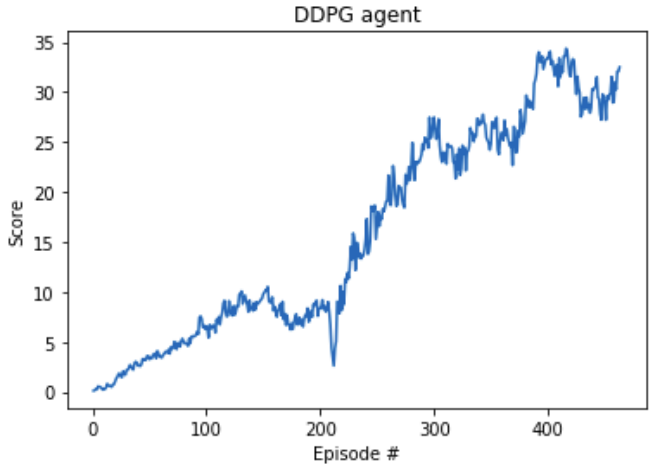

Figure 1: DDPG agent (scores (averaged over all 20 agents))

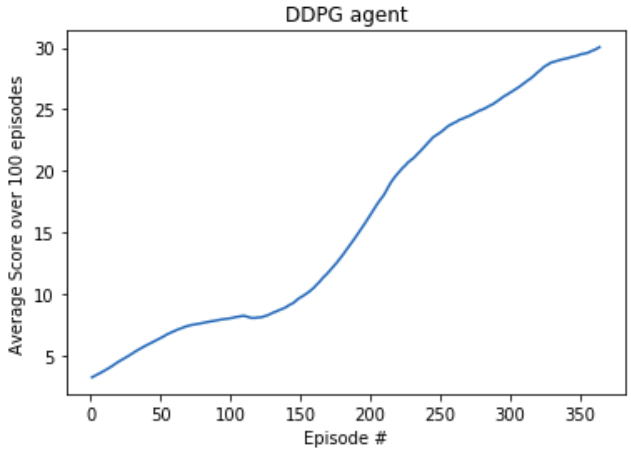

Figure 2: DDPG agent (scores (averaged over all 20 agents and 100 episodes))

The algorithm was able to solve the environment in approximately 360 episodes.

Future work

- Use of **Prioritized Experience Replay Buffer [2]** which is better in sampling more important but rarer experiences.
- As suggested in the *Benchmark Implementation*, I would like to implement a more stable approach like **Trust Region Policy Optimization**

(TRPO)[**3**] and **Truncated Natural Policy Gradient (TNPG)**[4].

- I would also like to implement my own version of **Proximal Policy Optimization (PPO)**[**5**], and the very recent paper **Distributed Distributional Deterministic Policy Gradients (D4PG)** [**6**]

References

[1] `https://github.com/Unity-Technologies/ml-agents`

[2] `https://arxiv.org/abs/1511.05952`

[3] `https://arxiv.org/abs/1502.05477`

[4] `https://papers.nips.cc/paper/2073-a-natural-policy-gradient.pdf`

[5] `https://arxiv.org/abs/1707.06347`

[6] `https://arxiv.org/pdf/1804.08617.pdf`