A DDPG Implementation

Nicolas Monti

January 2024

1 Introduction

This report describes a Deep Deterministic Policy Gradient (DDPG) implementation to solve a continuous control environment which was carried out as part of the Deep Reinforcement Learning Nanodegree program from *Udacity*. The environment is provided by *Unity Machine Learning Agents (ML-Agents)* - an open-source plug-in that enables games and simulations to serve as environments for training intelligent agents.

The goal is to train a double-jointed arm to maintain its position at a target location for as many time steps as possible. In this particular implementation, we decided to train 20 agents (double-jointed arms) in parallel to move to the target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to the torque applicable on the two joints. Every entry in the action vector should be a number between -1 and 1.

The task is episodic, and in order to solve the environment, the agents must get an average reward of +30 over 100 consecutive episodes (see section 5).

2 Training Algorithm

The algorithm implemented to train the agents is based from the article titled *Continuous Control With Deep Reinforcement Learning* [1], where the authors combine the Actor-Critic approach with insights from the Deep Q-Network (DQN) algorithm presented on the article titled *Human-level Control Through Deep Reinforcement Learning* [2].

There are two main processes running on the DDPG algorithm implemented: sampling and learning.

The first process takes care of sampling the environment at each time-step t by taking an action a_t and storing the observed experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ into a replay memory $D_t = (e_1, ..., e_t)$ - where s_t and r_t represent the environment's state and the reward collected at time-step t, respectively, and s_{t+1} is the environment's next state after taking action a_t.

The second process randomly selects a batch of experiences e from the replay memory D and learns from that batch using a gradient descent step. It is important to highlight that since we are using an Actor-Critic approach then the learning step is done based on two loss functions: one for the Actor and one for the Critic.

Finally, it is worth to mention that these two processes are not dependent on each other, and we can perform multiple sample steps and then one learning step – or even multiple learning steps from different batches. On this implementation, it was decided to perform U learning steps every C steps. A brief summary of the overall training procedure is depicted on Algorithm 1.

3 Actor-Critic Architectures

This implementation uses two neural networks: one for the Actor and one for the Critic. The Actor-NN maps states into actions while the Critic-NN maps states and actions into action values. The chosen neural network architecture for the Actor and Critic is very similar and they consist of three fully connected linear layers using a $ReLU$ activation function after the first and second layers. In both networks, the first and second linear layers output 256 features each. The only difference between the networks is that since the Actor-NN is mapping states into actions then we added a $tanh$ activation function after the third layer.

4 Training Set-up

The hyper-parameters chosen for this implementation were selected based on experience and results from a similar environment solved within the Nanodegree program. The training set-up consist of the following:

- Total number of episodes: $M = 300$
- Maximum time-steps per episode: 2,000
- Learn every time steps: $C = 20$
- Number of learning steps: $U = 10$
- Batch size: $B = 128$
- Replay memory D with capacity $N = 1,000,000$
- $\gamma = 0.99$
- $\tau = 0.001$
- Actor learning rate: $LR_{actor} = 0.0001$
- Critic learning rate: $LR_{critic} = 0.0003$

As suggested on the Udacity benchmark solution, we have introduced a gradient clipping when training the Critic network and we have added some normally distributed noise to the best actions output from the Actor network to improve the exploration of different actions.

Algorithm 1 A DDPG Training Algorithm

```
Initialize:
    * Replay memory $D$ to capacity $N$
    * Local Actor-NN $A_L$ and local Critic-NN $C_L$ with random
    weights $\theta_A$ and $\theta_C$, respectively
    * Target Actor-NN $A_T$ and target Critic-NN $C_T$ with
    weights $\theta_A^- = \theta_A$ and $\theta_C^- = \theta_C$, respectively
    * $U$ learning steps
for $episode = 1$ to $M$ do
    Observe current state $S = s_0$

    for time-step $t = 1$ to $T$ do
        Choose best action $a_t$ from current state $S$ using the local Actor-NN
        Take action $a_t$ and observe $r_t$, $s_{t+1}$
        Save experience tuple $e_t = (S, a_t, r_t, s_{t+1})$ into replay memory $D$

        Every $C$ time-steps repeat U-times
          Randomly sample a batch of experiences $e$ from replay memory $D$:
            $e_j = (s_j, a_j, r_j, s_{j+1})$

          Compute the target action values:
            $a_{j+1} = A_T(s_{j+1}, \theta_A^-)$
            $Q_{target} = r_j + \gamma * C_T(s_{j+1}, a_{j+1}, \theta_C^-)$

          Compute the expected action values:
            $Q_{expected} = C_L(s_j, a_j, \theta_C)$

          Compute the Critic loss:
            $L_{critic} = \frac{1}{B} * \sum_{k=0}^{B} (Q_{expected,k} - Q_{target,k})^2$ where $B$ is batch size

          Calculate $C_L$ gradients with backward pass
          Update $C_L$ weights $\theta_C$ with learning rate $LR_{critic}$

          Compute the Actor loss:
            $a_{predicted} = A_L(s_j, \theta_A)$
            $L_{actor} = -\frac{1}{B} * \sum_{k=0}^{B} C_L(s_j, a_{predicted}, \theta_C)$

          Calculate $A_L$ gradients with backward pass
          Update $A_L$ weights $\theta_A$ with learning rate $LR_{actor}$

          Update $A_T$ and $C_T$ network weights $\theta_A^-$ and $\theta_C^-$:
            $\theta^- = \tau * \theta + (1 - \tau) * \theta^-$

        Update current state: $S \leftarrow s_{t+1}$
    end for
end for
```

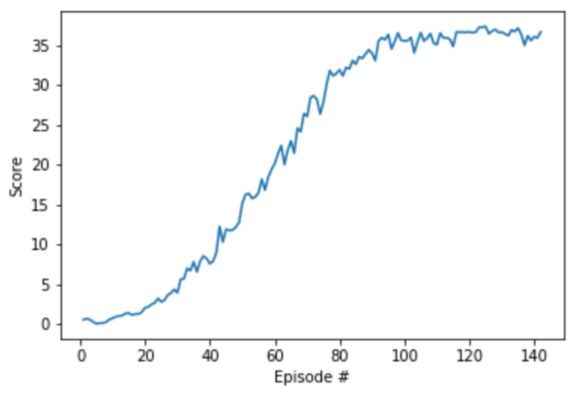

Figure 1: Agents' average rewards per episode

5 Results

On figure 1, we show the average rewards across all agents per episode. At episode 42 we consider the environment as solved which is when the agents achieved the goal of an average reward of +30 over 100 consecutive episodes. This result seems reasonable considering the benchmark implementation provided by Udacity as part of the Nanodegree program where using a similar DDPG implementation it took 63 episodes to solve this environment.

6 Future Work

Although this implementation managed to successfully solve the environment in a reasonable amount of episodes, there are many others algorithms proposed in the literature that could exhibit a better performance on solving this continuous control challenge. In particular, these algorithms stand out as good candidates:

- **PPO**. As described on the article named *Proximal Policy Optimization Algorithms* [3], the authors propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. These new methods, which they call Proximal Policy Optimization (PPO), have according to the authors some of the benefits of Trust Region Policy Optimization (TRPO) while being easier to implement and they have better sample complexity (empirically).

- **A3C**. As described on the article named *Asynchronous Methods for Deep Reinforcement Learning* [4], the authors propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers where parallel actor-learners have a stabilizing effect on training. According to the authors, the best of the proposed methods, called Asynchronous Advantage Actor-Critic (A3C), mastered a variety of continuous motor control tasks -like the environment described on this report- as well as learned general strategies for exploring 3D mazes purely from visual inputs.

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous Control With Deep Reinforcement Learning. *Arxiv*, Sep 2015. doi:https://doi.org/10.48550/arXiv.1509.02971.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518:529–523, 2015. doi:https://doi.org/10.1038/nature14236.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *Arxiv*, Jul 2017. doi:https://doi.org/10.48550/arXiv.1707.06347.

[4] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *Arxiv*, Feb 2016. doi:https://doi.org/10.48550/arXiv.1602.01783.