Continuous control project (Reacher)

as part of UDACITY: Deep Reinforcement Learning Nanodegree Program

Ivan Masmitja

***Abstract*—This report describes the basic ideas behind the project number 2 (aka Reacher) conducted as part of the UDACITY nanodegree called: Deep Reinforcement Learning Nanodegree Program, where a Deep Deterministic Policy Gradients (DDPG) is used to train an actor-critic agent in order to move a double-jointed arm to target locations.**

Keywords — Reacher, deep deterministic policy gradients, DDPG, reinforcement learning, deep neural network, artificial intelligence.

I. Introduction

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

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 torque applicable to 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, your agent must get an average score of +30 over 100 consecutive episodes.

II. Implementation

The implementation is based on the work conducted in [1], where the DDPG algorithm was firstly introduced. They introduced this algorithm as an "Actor-Critic" method. Though, some researchers think DDPG is best classified as a Deep Q-Network method for continuous action spaces, along with Normalized Advantage Functions (NAF) [2]. Regardless, DDPG is a very successful method and it's good for you to gain some intuition.

This model-free approach can learn competitive policies for different tasks using low-dimensional observations (e.g. cartesian coordinates or joint angles) using the same hyper-parameters and network structure. In many cases, it is also able to learn good policies directly from pixels, again keeping hyperparameters and network structure constant. A key feature of the approach is its simplicity: it requires only a straightforward actor-critic architecture and learning algorithm with very few “moving parts”, making it easy to implement and scale to more difficult problems and larger networks.

The DDPG algorithm is presented below (which has obtained from [1]):

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

For this project, we have used the second version which contains 20 identical agents, each with its own copy of the environment, as it is presented in Fig. 2.

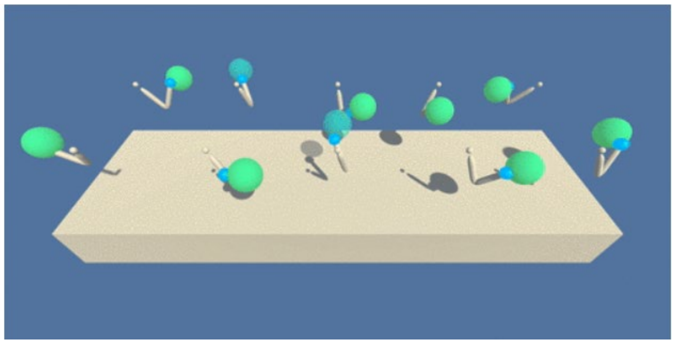
Fig. 1 Unity ML-Agents Reacher Environment with 20 identical agents. Each one is a double-jointed arm that can move to target locations.

To solve the environment, the only change we made was to use gradient clipping when training the critic network. In addition, we decided to get less aggressive with the number of updates per time step. In particular, instead of updating the actor and critic networks 20 times at every timestep, we amended the code to update the networks 10 times after every 20 timesteps. This was enough to solve the environment.

A. Hyperparameters

The hyperparameters used during the training were:

- BUFFER_SIZE = 1e5 (replay buffer size)
- BATCH_SIZE = 128 (minibatch size)
- # BATCH_SIZE = 64 (minibatch size)
- GAMMA = 0.95 (discount factor)
- TAU = 1e-3 (for soft update of target parameters)
- LR_ACTOR = 1e-4 (learning rate of the actor)
- LR_CRITIC = 1e-3 (learning rate of the critic)
- WEIGHT_DECAY = 0 (L2 weight decay)
- TRAIN_EVERY = 20 (How many iterations to wait before updating target networks)
- UPDATE_TIMES = 10 (Number of times we update the networks)
- fc1_units=400 (First hidden layer units)
- fc2_units=300 (Second hidden layer units)

B. Model Architecutre

The model architecture used for the DDPG agent was following a traditional Actor-Critic structure [3] and shown in Fig. 2. Two Neural Networks (NN) have been created, one for the Actor and one for the Critic.

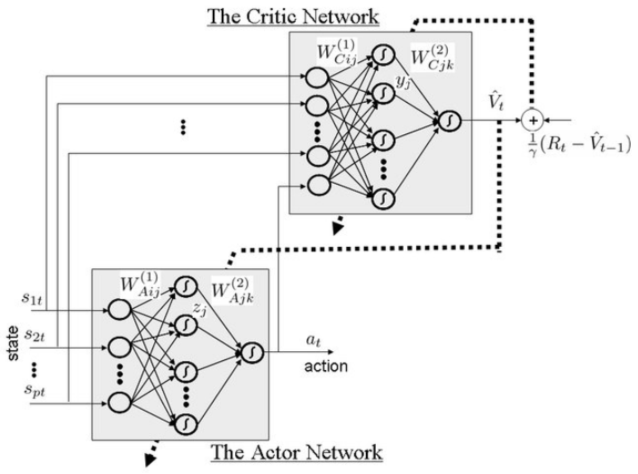

Fig. 2. the actor-critic learning architecture.

The Actor NN has 3 full connected linear layers. The first 2 layers have a ReLU activation function, whereas the last one has a tanh activation function to give an output control between -1 and 1. The first and second layers have 400 and 300 units respectively.

The Critic NN has also 3 full connected linear layers with a ReLU activation function. In this case, the output does not have the tanh activation. In addition, the second layer has been concatenated with the action provided by the Actor, as shown in Fig 2. The first and second layers have also 400 and 300 units respectively.

III. RESULTS

The above model proved to be very good at solving this problem. It achieved an average score above the score threshold of +30 over 100 consecutive episodes as can be observed in Fig 3, and summarized below.

Episode 10	Average Score (averaged over agents): 0.87
Episode 20	Average Score (averaged over agents): 0.97
Episode 30	Average Score (averaged over agents): 1.27
Episode 40	Average Score (averaged over agents): 1.83
Episode 50	Average Score (averaged over agents): 2.68
Episode 60	Average Score (averaged over agents): 3.62
Episode 70	Average Score (averaged over agents): 4.80
Episode 80	Average Score (averaged over agents): 6.08
Episode 90	Average Score (averaged over agents): 7.53
Episode 100	Average Score (averaged over agents): 8.96
Episode 110	Average Score (averaged over agents): 11.43
Episode 120	Average Score (averaged over agents): 14.43
Episode 130	Average Score (averaged over agents): 17.76
Episode 140	Average Score (averaged over agents): 21.12
Episode 150	Average Score (averaged over agents): 24.19
Episode 160	Average Score (averaged over agents): 26.96
Episode 170	Average Score (averaged over agents): 29.40
Episode 180	Average Score (averaged over agents): 31.60
Episode 190	Average Score (averaged over agents): 33.24
Episode 200	Average Score (averaged over agents): 34.76

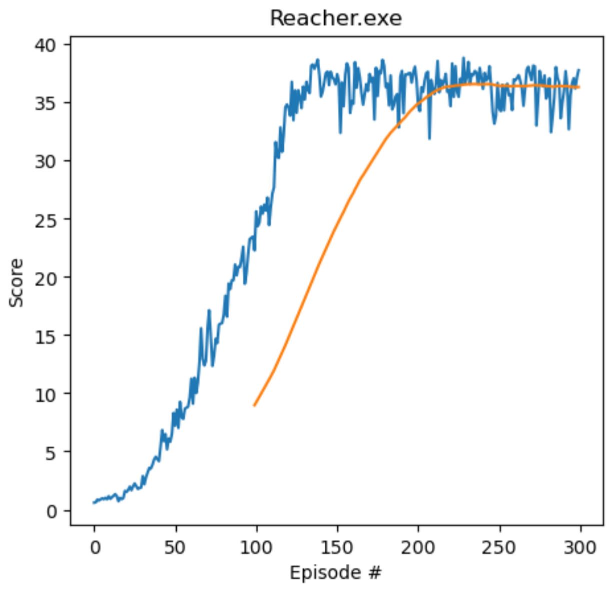

Fig. 3. Scores evolution over episode for continuous control project. Red line indicates the average values over 100 iterations. Here the second version using 20 identical agents has been used.

IV. Future work

As a future work, I will implement other algorithms that have shown great performance in continuous control scenarios. For example:

a) Proximal Policy Optimization (PPO) [4]

b) Asynchronous Advantage Actor-Critic (A3C) [5]

c) Distributed Distributional Deterministic Policy Gradients (D4PG) [6]

d) Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG) [7].

Finally, this algorithms will be implemented to obtain the optimal trajectory for an underwater autonomous vehicle in order to localize and track an underwater target [8]–[10].

References

[1] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, 2016.

[2] S. Gu, T. Lillicrap, U. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 6, pp. 4135–4148, 2016.

[3] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, "Boosting the performance of computing systems through adaptive configuration tuning," *Proc. ACM Symp. Appl. Comput.*, no. January 2014, pp. 1045–1049, 2009, doi: 10.1145/1529282.1529511.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv*, pp. 1–12, 2017.

[5] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 4, pp. 2850–2869, 2016.

[6] P. O. G. Radients *et al.*, "D Istributed D Istributional D Eterministic," *Int. Conf. Learn. Represent.*, pp. 1–16, 2018.

[7] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 3, pp. 2001–2014, 2016.

[8] I. Masmitja *et al.*, "Optimal path shape for range-only underwater target localization using a Wave Glider," *Int. J. Rob. Res.*, vol. 37, no. 12, pp. 1447–1462, 2018, doi: 10.1177/0278364918802351.

[9] I. Masmitja *et al.*, "Range-Only Single-Beacon Tracking of Underwater Targets from an Autonomous Vehicle: From Theory to Practice," *IEEE Access*, vol. 7, pp. 86946–86963, 2019, doi: 10.1109/ACCESS.2019.2924722.

[10] I. Masmitja *et al.*, "Mobile robotic platforms for the acoustic tracking of deep-sea demersal fishery resources," *Sci. Robot.*, vol. 5, no. eabc3701, 2020.