Continuous control: Unity Reacher environment with deep reinforcement learning

Ali Hummos

1 Overview:

This the second project of the Udacity deep RL nanodegree and it aims to solve the Unity Reacher environment, by achieving a score of ¿30 on 100 consecutive episodes over all 20 agents. This project uses the 20 -agent version of the environment.

The environment simulates an arm with two joints. The goal of the task is to keep the tip of the arm inside a spherical target region, rotating around the arm. The environment provides a reward of +0.1 for each time step the agent maintains its tip within the spherical target region. The state of the environment is represented in a vector of 33 values representing the position and velocities of the arm segments.

The environment is continuous rather than episodic but can easily be sampled to a desired episode length without information loss as it's a generally stationary environment. The action space is continuous, which constrain our choice of algorithm to solve it.

This project went through a few iterations of trying to code the PPO algorithm [5] to solve the environment. The code was very difficult to implement, despite the model being "simple" compared to previous iterations (eg. [4]). The computations involve discounting of future rewards and discounting of advantage functions which can complicate the math. In addition, its most successful implementations typically use a value estimator network as well as the policy network. Which introduces the difficulties of jointly optimizing two networks.

I ended up abandoning the efforts of trying to stabilize my implementation of it. Despite seeing multiple examples of successfully implemented PPO for this task. I guess it speaks to the brittleness of these models in general, and how debugging implementations is extremely hard, because the performance of the model depends on a variety of choices in implementation and hyperparameters. So suboptimal code pieces/parameters are very hard to identify.

I then turned to using an implementation of the DDPG model as a starting point. Even with access to well-tested code, it proved quite a journey to tune the code to work for multiple agents and adjusting hyperparameters to stabilize learning.

2 Implementation:

The code utilizes the DDPG algorithm [2], and builds on the code provided by the Udacity deep reinforcement learning library [1]. The algorithm can be considered an extension of the Deep Q-Leaning network [3] which enables it to deal with continuous action spaces by adding a policy network (an actor) to output action values, while the Q-network continues to learn an estimation of the value function.

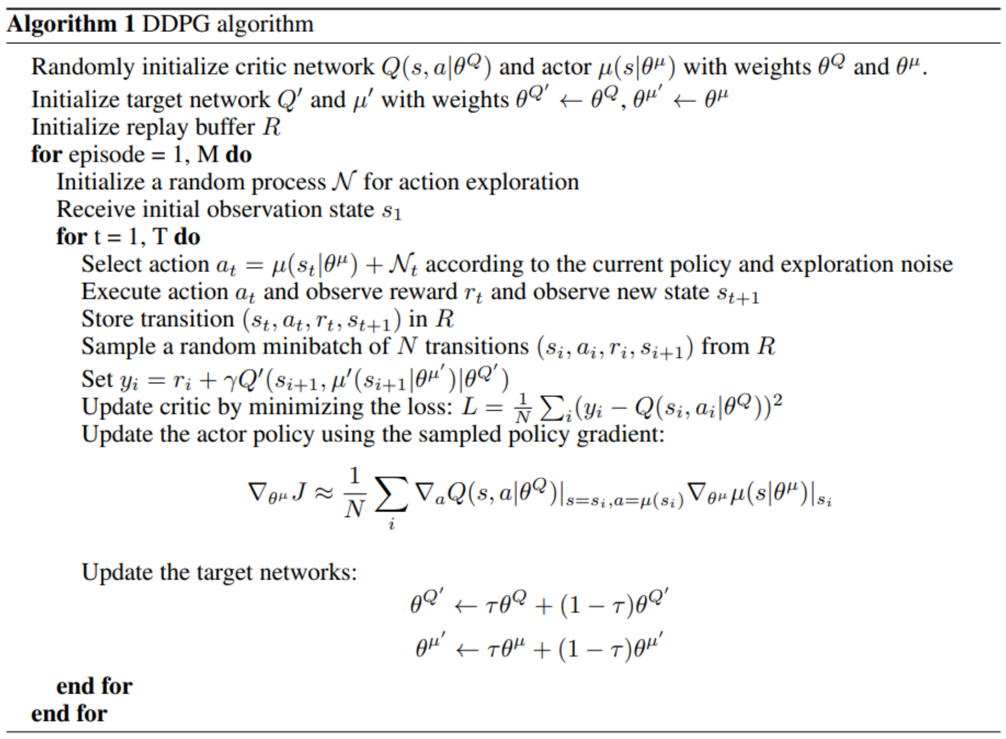
As in DQN paper, DDPG uses a target and a local version of both policy and value networks, the local version is updated through SGD, while the target version gets soft updates throughout training.

DDPG uses a deterministic policy network, which means that it estimates the actions that would maximize the return or value as estimated by the value network. It directly outputs the action values it

[1] https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-bipedal

predicts would greedily exploit the current states. Which means that exploration has to be factored. In this implementation, we use an Ornstein-Uhlenbeck process to generate white noise (sigma=0.15, theta=0.2) and add it to the predict action vector to encourage exploration.

The DDPG algorithm is summarized in this pseudocode taken from the original paper.



```
**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
```

Figure 1: DDPG - Pseudocode from original paper [2]

The policy network $\mu(s|\theta^\mu)$ is updated with the gradient of the expected return J with respect to the parameters θ^μ:

$$\nabla_\theta J = \frac{1}{N}\sum_i \nabla_\alpha Q(s, a;\ \theta^Q)\nabla_\theta \mu(s|\theta^\mu)$$

(1)

Accordingly, the policy network (architecture given in table2) attempts to output actions that would maximize the value estimated by the value network. *i.e.* it learns to maximize the output of the value network.

Where as the value network $Q(s, a;\ \theta^Q)$ is updated to minimize the loss L:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^\mu))^2 \tag{2}$$

Where y_i is the actual reward obtained at episode i and Q is the estimated value of taking action a_i in state s_i, parameterized by a neural network with parameters θ^Q (see table 1). *i.e.* the mean squared error of the value network as it tries to predict the return from the current episode.

Layer	Input	Output	Activation
Layer 1	33	256+4	ReLU
Layer 2	256+4	128	ReLU
Layer 3	128	1	None

Table 1: Value network architecture

Layer	Input	Output	Activation
Layer 1	33	256	ReLU
Layer 2	256	128	ReLU
Layer 3	128	4	tanh

Table 2: Policy network architecture

The second model implemented and tested was the Proximal Policy Optimization (PPO) method, that was first introduced by [5]. My first implementation of this method highly relayed on Udacity implementation of the PPO for Pong with few changes to use continuous actions. This agent failed to solve the environment after 4000 steps. My second implementation was based on Shangtong Zhang GitHub repository[2]. While the code did solve the task, any edits to the specific components chosen in the model, such as a critic, or the implementation of the GAE function, or any of the hyperparameters lead to the model failing to learn the task.

3 Hyperparameters:

Here is a direct snippet from the code listing the hyperparameters used. I did little exploration or deviation from most commonly used values in online implementations. The hyperparameters are named in sufficiently expressive terms to indicate their nature in the algorithm.

- state size= env info.vector observations.shape[1]
- action size= brain.vector action space size
- no agents= len(env info.agents)
- device= device
- gradient clip = 1
- rollout length = 1001

[2]Source:`https://github.com/ShangtongZhang/DeepRL`

- episode count = 251
- buffer size = int(1e5)
- batch size = 128
- lr actor = 1e-4
- lr critic = 1e-4
- discount rate = 0.99
- tau = 1e-3
- weight decay = 0

4 Training:

The agent was trained locally on a NVIDIA TITAN RTX graphics card. Training the model takes about 1.5 hours of CPU/GPU time. The model learned quickly and stabilized as shown in the average reward plot 2.

5 Future work:

Future work can involve devising ways to stabilize the solution. It remains brittle and sensitive to hyperparameters and to specific algorithmic choices. This can involve implementation of more recent extnsions and enhancements to the DDPG algorithm such as the distributed distributional version of it [1]. In addition, considering the model's sensitivity to hyperparameters, a space search technique such as Bayesian optimization, might yield some efficient and possibly more robust regions in the hyperparameters space.

References

[1] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed Distributional Deterministic Policy Gradients. *arXiv e-prints*, page arXiv:1804.08617, Apr 2018.

[2] 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.org*, September 2015.

[3] 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(7540):529–533, February 2015.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv.org*, February 2015.

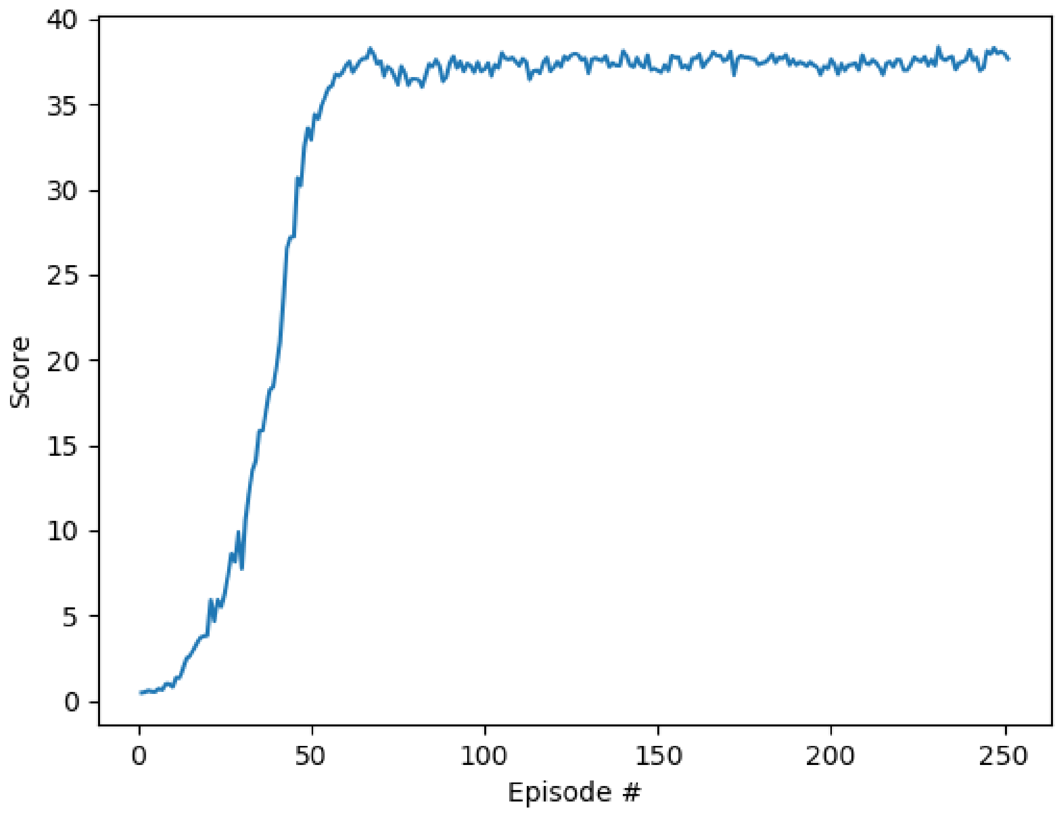


Figure 2: Average rewards over 20 agents per episode

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv.org*, July 2017.