Deep Reinforcement Learning – Project 2 : Continuous Control

Melan Vijayaratnam

This document presents a technical description of the Continuous Control project in the context of the Deep Reinforcement Learning Nanodegree from Udacity.

1 Summary

For this project, we will work with the Reacher environment.

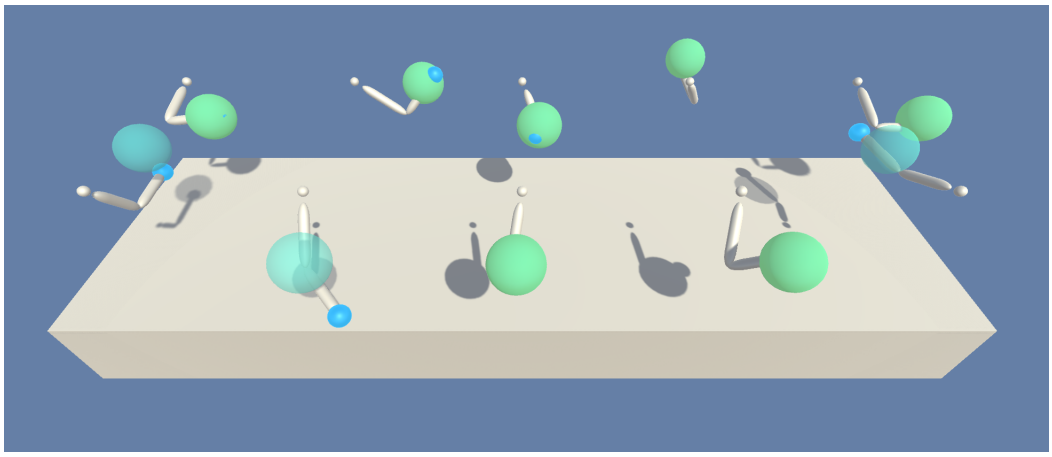

Figure 1: Unity ML-Agents Reacher environment

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 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.

This report will focus on one of the two versions of the environment, that is the one that contains a single agent. The task is **episodic** where each episode has 1000 timesteps. In order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes

2 Methods

2.1 Policy-based & value-based methods

With **value-based methods**, the agent uses its experience with the environment to maintain an estimate of the optimal action-value function. The optimal policy is then obtained from the optimal action-value function estimate:

$$\text{Interaction} \rightarrow \text{Optimal Value Function } q_* \rightarrow \text{Optimal Policy } \pi_*$$
Value-based methods

Policy-based methods directly learn the optimal policy, without having to maintain a separate value function estimate:

$$\text{Interaction} \rightarrow \text{Optimal Policy } \pi_*$$
Policy-based methods

One of the limitations of value-based methods is that they tend to a **deterministic** or **near-deterministic** policies.

On the other hand, policy-based methods can learn either **stochastic** or **deterministic** policies, so that they can be used to solve environments with either finite or continuous action spaces.

2.2 Actor-critic methods

In **actor-critic methods**, we are using value-based techniques to further reduce the variance of policy-based methods.
Basically actor-critic are a hybrid version of the policy- and value- based methods, in which the actor estimates the policy and the critic estimates the value function.

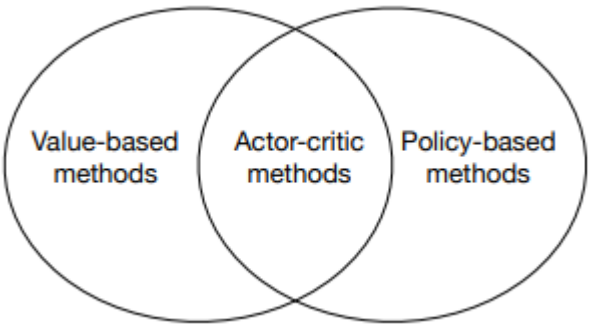

Figure 2: Actor-critic and relation to other Reinforcement Learning methods

3 Deep Deterministic Policy Gradient (DDPG)

3.1 Overview

DDPG (Lilicrap et al., 2015), short for **Deep Deterministic Policy Gradient**, is a model-free off-policy actor-critic algorithm, combining DPG with DQN. Recall that DQN (Deep Q-Network) stabilizes the learning of the Q-function by using experience replay and a frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.

In DDPG, we use 2 deep neural networks : one is the actor and the other is the critic:

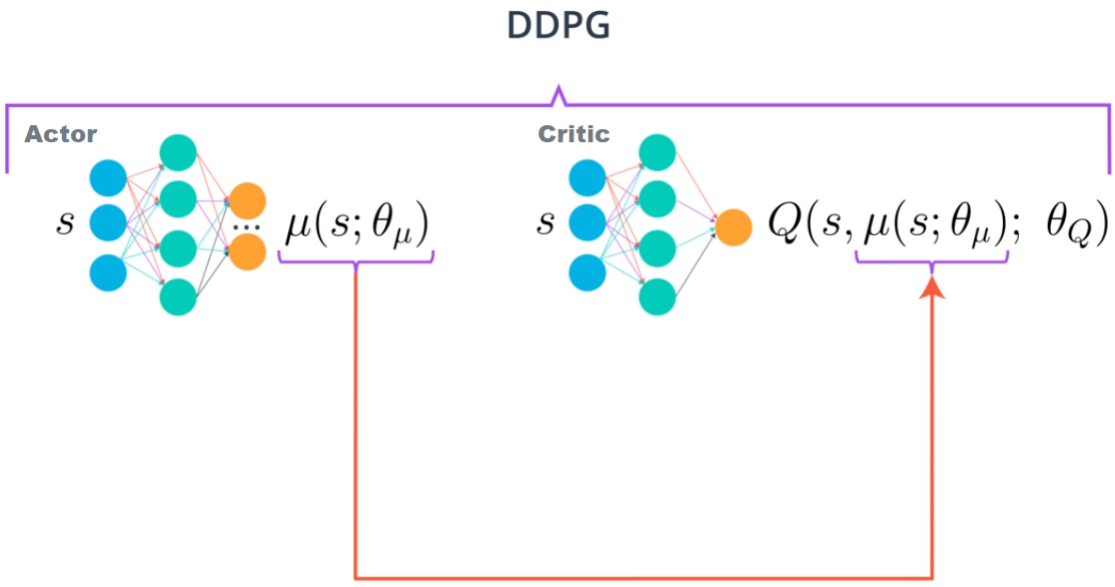

Figure 3: DDPG structure

The actor is used to approximate the optimal policy deterministically. That means we always want to output the best believed action for any given state. This is unlike a stochastic policy which learn a probability distribution over the actions.
In DDPG, we want the believed best action every single time we query the actor network, that is a deterministic policy. The actor is basically learning $argmax_a\, Q(s, a)$ which is the best action.

The critic learns to evaluate the optimal action-value function by using the actor's best believed action.

3.2 Replay Buffer

When using neural networks, it is usually assumed that samples are independently and identically distributed (i.i.d). Alas in Reinforcement Learning, samples are generated from exploring sequentially in an environment, resulting in the previous assumption that no longer holds. Action A_t is partially responsible for the reward and state at time (t + 1) :

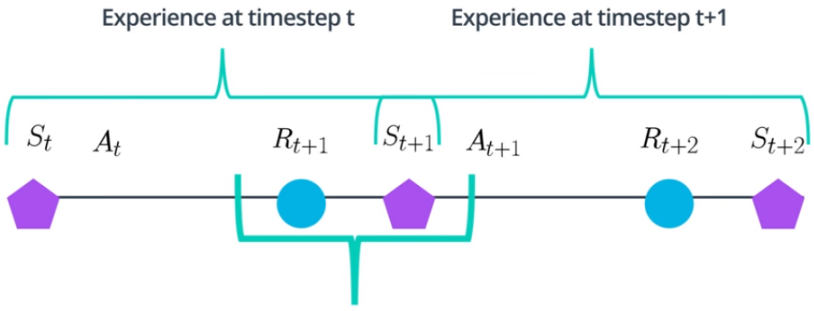

Figure 4: Highlight of correlation of sequential samples in Reinforcement Learning

As in DQN, we use a replay buffer to address this issue. Because DDPG is an off-policy algorithm, that is it employs a separate behavior policy independent of the policy being improved upon, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

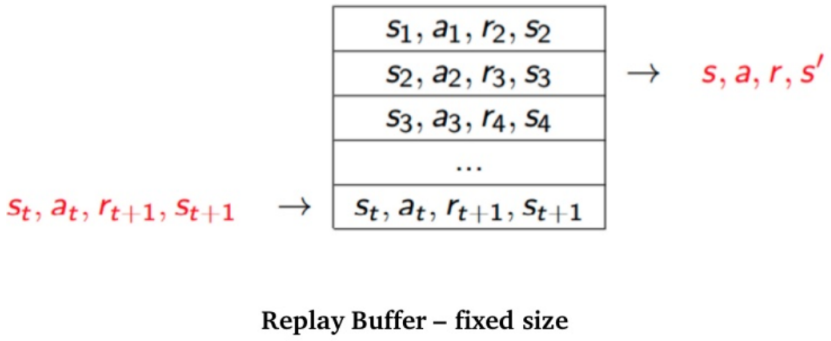

Figure 5: Replay buffer overview

3.3 Soft target updates

In DDPG, not only we have a regular network for the actor as well for the critic, we do also have a copy of these regular networks. We have a target actor and target critic, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively that are used for calculating the target values.

We update those target networks using a **soft-update strategy**. A soft-update strategy consists of slowly blending your regular network weights with your target network weights:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

,with $\tau \ll 1$, usually 1% or 0.01.

This means that the target values are constrained to change slowly,
greatly improving the stability of learning.

3.4 Noise for exploration

A major challenge of learning in continuous action spaces is exploration. An advantage of off-policies algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm.

As such, an exploration policy μ' is constructed by adding noise $\mathcal{N}$:

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

$\mathcal{N}$ can be chosen to suit the environment. We use the **Ornstein-Uhlenbeck process** to model the velocity of a Brownian particle with friction, which results in temporally correlated values centered around 0. This allows exploration efficiency in physical control problems with inertia.

Start with Brownian motion (also known as Wiener process): $dW_t = \mathcal{N}(0, dt)$
Then add friction: $dx_t = \theta(\mu - x_t)\, d_t + \sigma\, dW_t$

Where θ controls the amount of friction to pull the particle towards the global mean μ. The parameter σ controls the scale of the noise. By discretizing the previous formula, we get:

$$X_{n+1} = X_n + \theta(\mu - X_n)\Delta t + \sigma\Delta W_n$$

Because ΔW_n are i.i.d, then $\Delta W_n \sim \mathcal{N}(0, \Delta t) = \sqrt{\Delta\, t}\,\mathcal{N}(0, 1) = \mathcal{N}(0, 1)$ because $\Delta t = 1$ between two timesteps.

4 DDPG Algorithm

The DDPG algorithm use the techniques explained in the previous section :

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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 6: DDPG Algorithm

5 Implementation details

Because we have two models; the actor and critic that must be trained, it means that we have two set of weights that must be optimized separately. Adam was used for the neural networks with a learning rate of 10^{-3} and 10^{-3} respectively for the actor and critic. For $\mathcal{Q}$, I used a discount factor of $\gamma = 0.99$. For the soft target updates, the hyperparameter τ was set to 0.001.

The neural networks have 2 hidden layers with 250 and 100 units respectively. For the critic $\mathcal{Q}$, the actions were not included the 1st hidden layer of $\mathcal{Q}$. The final layer weights and biases of both the actor and critic were initialized from a uniform distribution $[-3 \times 10^{-3}, 3 \times 10^{-3}]$ and $[3 \times 10^{-4}; 3 \times 10^{-4}]$ to ensure that the initial outputs for the policy and value estimates were near zero. As for the layers, they were initialized from uniform distribution $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer.

The training used minibatches of 128 and the replay buffer was of size 10^6.

For the exploration noise process, the parameters used for the Ornstein-Uhlenbeck process were $\theta = 0.15$ and $\sigma = 0.2$.

6 Results

The agent is trained until it solves the environment, that is to say an average reward of at least $+30$ for the last 100 episodes. Because I have trained the agent for 1000 episodes and found it was not enough, I restarted from these weights.

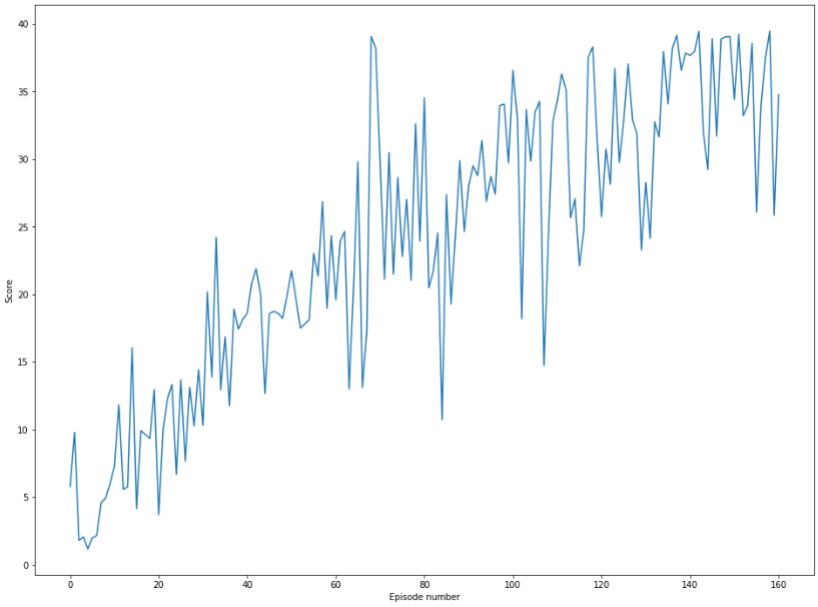

Figure 7: Plot of scores per episode (from episode 1000)

The environment is solved in 1060 episodes !

7 What's next

The first version of this environment covered in this report, takes into account only one agent. There is another version which now take over 20 agents in the same environment.

This seems to be a great challenge to tackle, leaving place 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.