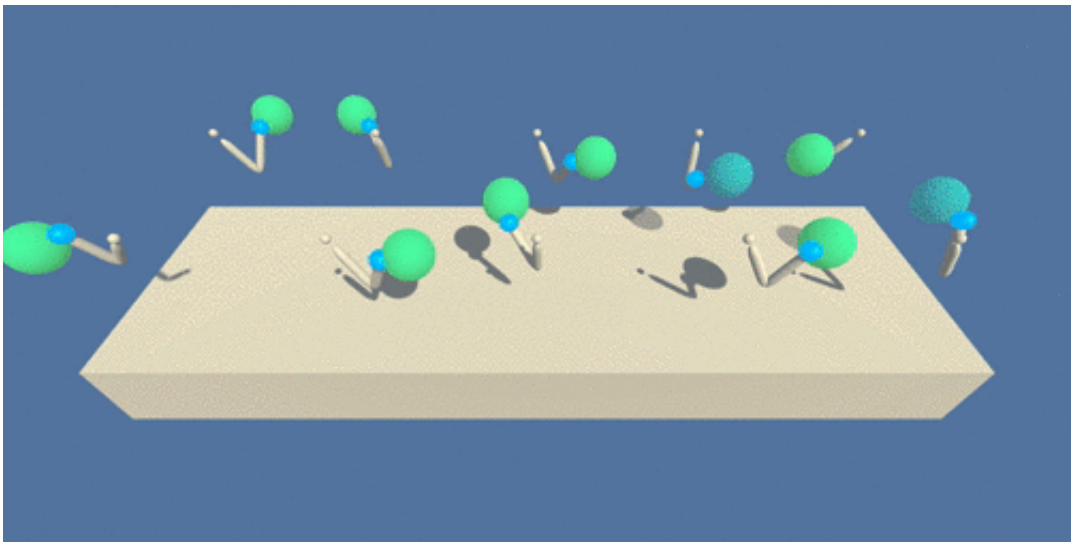
Project 2 Report: Continuous Control

TAN H. CAO*

August 14, 2020

1 Project Goal

In this project, we will work with the 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 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.

Our goal here is to solve the second version of the Unity environment that contains 20 identical agents, each with its own copy of the environment. This version is 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.

*Department of Applied Mathematics and Statistics, SUNY (State University of New York) Korea, Yeonsu-Gu, Incheon, Korea (tan.cao@stonybrook.edu).

2 Learning Algorithm

To solve this challenging environment, we have explored and implemented the Deep Deterministic Policy Gradient algorithm (DDPG) as described in the paper [1].

2.1 Introduction

This paper presents an actor-critic, model-free algorithm based on the *deterministic policy gradient* that can operate over continuous action spaces. The algorithm proposed in the paper is able to find policies whose performance is competitive with those found by a *planning algorithm* with full access to the dynamics of the domain and its derivatives.

One of the primary goal of the field of AI is to solve complex tasks from unprocessed, high-dimensional, sensory input. There has been some significant progress:

- combining advances in deep learning for sensory processing with reinforcement learning,
- deep Q Network (DQN) algorithm.

However, DQN can only handle discrete and low-dimensional action spaces. A possible approach to adapting DRL methods such as DQN to continuous domains is to simply discretize the action space, but this approach has many limitations with *dimensionality*.

2.2 The Deep DPG algorithm

To overcome such challenging issues, the authors proposed a model-free, off-policy actor-critic algorithm using deep function approximator that can learn policies in high-dimensional, continuous action spaces. Their work is based on the **DPG** algorithm. It is worth mentioning that a naive application of this actor-critic method with neural function approximators is unstable for challenging problems.

The idea is we need to combine the actor-critic approach with DQN that learns value functions using function approximators in a stable and robust way, along with *batch normalization* – a recent advance in deep learning.

The model-free approach proposed in the paper, called **Deep DPG** (DDPG), can learn competitive policies for all of the tasks discussed there. This approach is easy to implement as it requires only a straightforward actor-critic architecture and learning algorithm with very few "moving parts".

2.3 Background

To proceed, we consider a standard RL framework consisting of the interactions between an agent and an environment E in discrete time steps. At each time step t, the agent

- receives an observation x_t,
- takes an action a_t,
- receives a scalar reward r_t.

In general, the environment may be partially observed so that the entire history of the observation, action pairs $s_t = (x_1, a_1, \ldots, a_{t-1}, x_t)$ may be required to describe the state. Here, we assumed the environment is fully-observed so $s_t = x_t$.

An agent's behavior is defined by a policy π, which maps states to a probability distribution over the actions $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$. The environment E may also be *stochastic*. We model it as a MDP with

- a state space $\mathcal{S}$,
- action space $\mathcal{A} = \mathbb{R}^N$,
- an initial state distribution $p(s_1)$,
- transition dynamics $p(s_{t+1}|s_t, a_t)$,
- reward function $r(s_t, a_t)$

The agent's goal is to learn a policy that maximizes the expected return

$$J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1].$$

The discounted state visitation distribution for a policy π is denoted as ρ^π. The action-value function describes the expected return after taking an action a_t in state s_t and thereafter following policy π :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}[R_t|s_t, a_t],$$

where $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$ with a discounting factor $\gamma \in [0, 1]$.

Recall the *Bellman* equation:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim E}[Q^\pi(s_{t+1}, a_{t+1})]]$$

which reduces to

$$Q^\mu(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))]$$

if the target policy is deterministic. This means that it is possible to learn Q^μ off-policy, using transitions that are generated from a different stochastic behavior policy β.

Q-learning (off-policy algorithm) uses the greedy policy $\mu(s) = \arg\max_a Q(s, a)$. We will optimize function approximators parametrized by θ^Q by minimizing the loss:

Loss (2.1)
$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right]$$

where

target (2.2)
$$y_t := r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q).$$

2.4 Actor-Critic Method

We will not apply Q-learning directly to continuous action spaces because doing so would require solving an optimization of a_t at every time step which will take a lot of time. Instead, we used an actor-critic approach based on the DPG algorithm.

In the DPG algorithm,

- a parameterized **actor function** $\mu(s|\theta^\mu)$ specifies the current policy by deterministically mapping states to a specific action,
- the **critic** $Q(s, a)$ is learned using the *Bellman* equation as in Q-learning.
- the **actor** is updated by following the applying the chain rule to the expected return from the start distribution J w.r.t the actor parameters:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

The name of the algorithm is Deep DPG (DDPG). DDPG is a modification of DPG, inspired by the success of DQN, which allow it to use neural network function approximators to learn in large state and action spaces online.

The actor-critic approach would leverage the strengths of both *policy-based* and *valued-based* methods. In fact, the actor-agent would learn how to act by directly estimating the optimal policy through **gradient ascent** using a policy-based approach. Meanwhile, the critic-agent will learn how to estimate the state-action valued functions using a value-based approach.

2.5 The Replay Buffer

An issue when using neural networks for RL is that most optimization algorithms assume that the samples are independently and identically distributed, which may fail when the samples are generated from exploring sequentially in an environment. In addition, to use hardware optimizations efficiently, it would be better to learn in minibatches rather than online.

DQN algorithm uses a replay buffer to address these issues. The replay buffer is a finite sized cache $\mathcal{R}$. Transitions were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. When the replay buffer was full, the oldest samples were discarded. At each time step, the actor and critic are updated by sampling a minibatch uniformly from the buffer. DDPG is an off-policy algorithm, so the replay buffer can be large. This will benefit the algorithm from learning across a set of uncorrelated transitions.

2.6 The Soft Update

In order to avoid unstable learnings, we would not directly implement Q learning in (2.1) with neural networks. Instead, we will do the following:

- Creating a copy of the **actor** and **critic** networks $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively that are used for calculating the target values.
- Updating the weights of these target networks by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This will change the target slowly which would greatly improve the stability of learning.

In order to train the critic consistently without divergence, both targets μ' and Q' must have stable targets y_i. This may slow learning since the target network delays the propagation of value estimations.

2.7 Batch Normalization

Another issue arising when learning from low dimensional feature vector observation is that the different components of the observation may have different physical units and the ranges may vary across environments. This can make it difficult for the network to learn effectively and may make it difficult to find hyper-parameters to generalize across environments with different scales of state values. We address this issue by using *batch normalization* to normalize each dimension across the samples in a minibatch to have unit mean and variance, resulting in effective learning across many different tasks with differing types of units without needing to manually ensure the units were within a set range.

2.8 Exploration

It is well-known that exploration is a major challenge of learning in continuous action spaces. In DDPG (and off-policy algorithms in general), the exploration problem can be treated independently from the

learning algorithm. An exploration policy μ' can be constructed by adding noise sampled from a noise process $\mathcal{N}$ to our actor policy as follows

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

$\mathcal{N}$ can be chosen to suit the environment.

It seems that DDPG combined insights from recent advances in deep learning and reinforcement learning, resulting in an algorithm that robustly solves challenging problems across a variety of domains with continuous action spaces, even when using raw pixels for observations. However, DDPQ (as with most model-free reinforcement approaches) has a limitation that requires a large number of training episodes to find solutions.

2.9 Code Implementation of DDPG

Our code here is a modification of the code of "pendulum" from the course "Deep Reinforcement Learning Nanodegree". The code consists of the following files:

(i) **model.py**: In this file, we created **actor** and **critic** networks respectively.

- The actor network receives one of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm, as an input and returns each action as a vector with four numbers, corresponding to torque applicable to two joints. Each entry of the action vector varies from -1 to 1. This network has two hidden layers with 400 and 300 units respectively.
- The critic network also receives one of 33 variables representing the observation space. However, the output of its first hidden layer and the action learned from the actor network should be stacked in order to be passed in as an input for the critic's second hidden layer. Finally the critic network will return the approximation of the target value based on the given state and the estimated best action.
- The numbers of units of the hidden layers were taken from the paper [1].
- Using batch normalization we address the issue of running computations on large input values and model parameters by scaling the features to be within the same range throughout the model. Therefore, we use batch normalization to the output of the first fully-connected layers of both the actor and critic models.

(ii) **ddpg_agent.py**: this file contains the DDPG-agent, the replay buffer implementation, and the exploration implementation (QUNoise)

- DDPG hyperparameters:

  - `batch_size = 128`
  - `buffer_size` = 10^5
  - `gamma = 0.99`
  - `lr_actor = 0.001`
  - `lr_critic = 0.001`
  - `tau = 0.001`
  - `noise_decay = 0.999`

- The DDPG agent class has the following methods:
  - `__init__()`: would initialize an agent object.

    + Randomly initialize critic network $Q_{loc}(s, a|\theta^Q)$ and actor network $\mu_{loc}(s|\theta^\mu)$ with weights θ^Q and θ^μ.
    + Initialize the target network Q_{tar} and μ_{tar} with weights

$$\theta^{Q_{tar}} \leftarrow \theta^{Q_{loc}}, \quad \theta^{\mu_{tar}} \leftarrow \theta^{\mu_{loc}}$$

by making two copies of the actor and critic networks from the model.

```
# Actor Network
self.actor_local = Actor(state_size, action_size, random_seed).to(_device)
self.actor_target = Actor(state_size, action_size, random_seed).to(_device)

# Critic Network
self.critic_local = Critic(state_size, action_size, random_seed).to(_device)
self.critic_target = Critic(state_size, action_size, random_seed).to(_device)
```

+ Implement Adam optimization algorithm using `optim.Adam`.
+ Initialize the replay buffer: `self.replay_buffer = ReplayBuffer(random_seed)`
+ Initialize the noise process by setting: `self.noise = QUNoise(action_size, random_seed)`

- `step()` would save experience in replay memory and extract a sample from the buffer with the batch size 128 to learn.
- `act()` would return an action for a given state per current policy.

  + We first pass the state to the actor-local network.
  + Get the action from this network.
  + Add noise to this action in order to explore the environment:
  `action = action + self.noise_decay*self.noise.sample()`
  Here we decay the noise in order to exploit later on.
  `self.noise_decay *= self.noise_decay`
  Note that the noise here is created from the class `OUNoise` with the method `sample()`. This kind of noise is made by perturbing the state by `state = state + dx`, where `dx` is specified in the class `OUNoise`.
  Finally we need to clip the action to ensure it takes the values in the interval $[-1, 1]$.
- `learn()` would update the policy and value parameters using the given batch of experience tuples.
- `soft_update()` would update the model parameters softly with parameter `tau = 0.01`.

(iii) **Continuous_Control.ipynb**: this Jupyter notebook has the reference project starter code and the code that we have implemented to train the agent.

3 Plot of Rewards

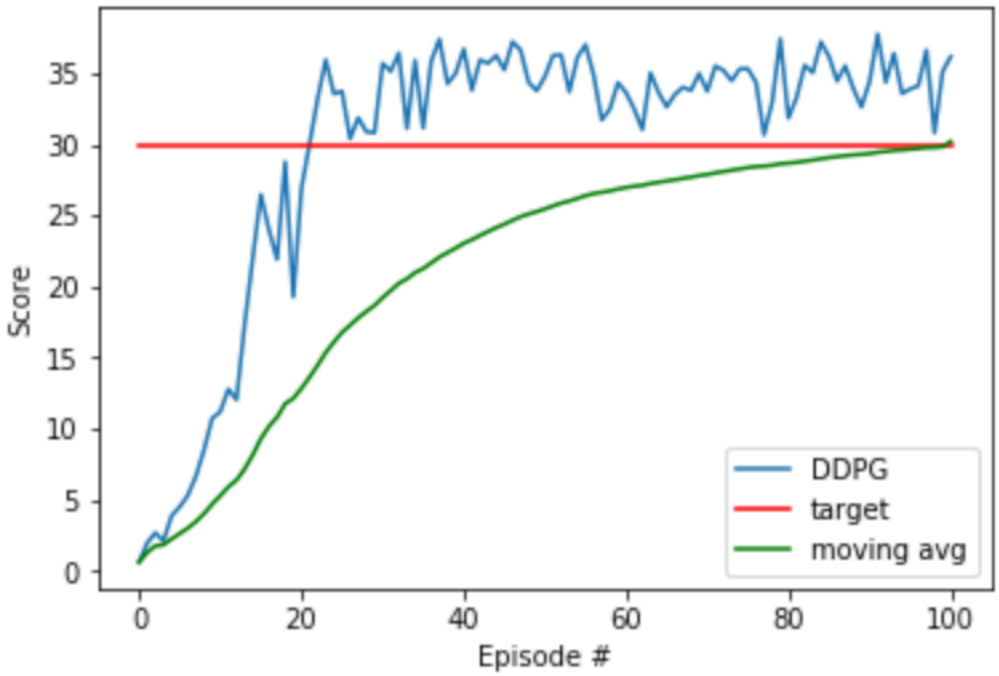
After running our code with the following data:

- `n_episodes = 500`
- `max_t = 1000`
- `solved_score = 30.0`
- `consec_episodes = 100`

we are able to solve the environment in the first episode. The graph below shows our final results with the top mean score of 39.66 in the 98th episode.

```
Episode  85 (03m23s)    Score: 37.20 (H: 39.63 / L: 28.94)      Moving Average: 28.98
Episode  86 (03m23s)    Score: 36.18 (H: 39.56 / L: 26.69)      Moving Average: 29.07
Episode  87 (03m23s)    Score: 34.45 (H: 38.67 / L: 15.18)      Moving Average: 29.13
Episode  88 (03m23s)    Score: 35.52 (H: 39.21 / L: 32.37)      Moving Average: 29.20
Episode  89 (03m23s)    Score: 33.93 (H: 39.51 / L:  3.38)      Moving Average: 29.25
Episode  90 (03m22s)    Score: 32.63 (H: 39.37 / L: 18.05)      Moving Average: 29.29
Episode  91 (03m23s)    Score: 34.42 (H: 38.82 / L: 25.68)      Moving Average: 29.35
Episode  92 (03m23s)    Score: 37.76 (H: 39.51 / L: 35.46)      Moving Average: 29.44
Episode  93 (03m22s)    Score: 34.37 (H: 39.49 / L: 27.96)      Moving Average: 29.49
Episode  94 (03m23s)    Score: 36.40 (H: 39.45 / L: 23.81)      Moving Average: 29.57
Episode  95 (03m23s)    Score: 33.57 (H: 38.84 / L: 27.72)      Moving Average: 29.61
Episode  96 (03m29s)    Score: 33.89 (H: 37.96 / L: 29.01)      Moving Average: 29.65
Episode  97 (03m24s)    Score: 34.11 (H: 38.78 / L: 30.32)      Moving Average: 29.70
Episode  98 (03m24s)    Score: 36.63 (H: 39.66 / L: 33.78)      Moving Average: 29.77
Episode  99 (03m24s)    Score: 30.81 (H: 36.58 / L: 14.30)      Moving Average: 29.78
Episode 100 (03m23s)    Score: 35.12 (H: 39.01 / L: 26.62)      Moving Average: 29.83
Episode 101 (03m24s)    Score: 36.17 (H: 39.45 / L: 28.30)      Moving Average: 30.19

Environment SOLVED in 1 episodes!       With the Moving Average of =30.2.
```



4 Ideas for Future Work

When we first run our code, the moving average was increasing very slowly at the first few episodes but was somehow stable later. This was probably because of outsized gradients. To address this issue of exploding gradients, we would implement the gradient clipping using the `torch.nn.utils.clip_grad_norm_` function in the "update critic" section of the `Agent.` method. We hope implementing this will improve our agent performance.

In the future we would like to implement following other actor-critic algorithms to solve our environment:

- TRPO: Trust Region Policy Optimization (see [2])
- GAE: Generalized Advantage Estimation (see [3])
- A3C: Asynchronous Advantage Actor-Critic (see [4])

- A2C: Advantage Actor-Critic
- ACER: Actor Critic with Experience Replay (see [5])
- PPO: Proximal Policy Optimization (see [6])
- D4PG: Distributed Distributional Deterministic Policy Gradients (see [7])

5 Acknowledgments

We are gratefully indebted to Nilson Chapagain for his useful remarks and discussions on how to select good hyperparameters so that the agents could achieve their highest performance.

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra, Continuous control with deep reinforcement learning.

[2] John Schulman, Sergey Levine, Philipp Moritz, Michael l. Jordan, & Pieter Abbeel, Trust Region Policy Optimization.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, & Pieter Abbeel, High-Dimensional Continuous Control Using Generalized Advantage Estimation.

[4] Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, & Koray Kavukcuoglu, Asynchronous Methods for Deep Reinforcement Learning.

[5] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, & Nando de Freitas, Sample Efficient Actor-Critic with Experience Replay.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, & Oleg Klimov, Proximal Policy Optimization Algorithms.

[7] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, & Timothy Lillicrap, Distributed Distributional Deterministic Policy Gradients.