Report

October 30, 2019

1. Report

The aim of this report is to describe the learning algorithm (DDPG) method used in the Continuous Control Project, provided by Udacity, along with its chosen hyperparameters.

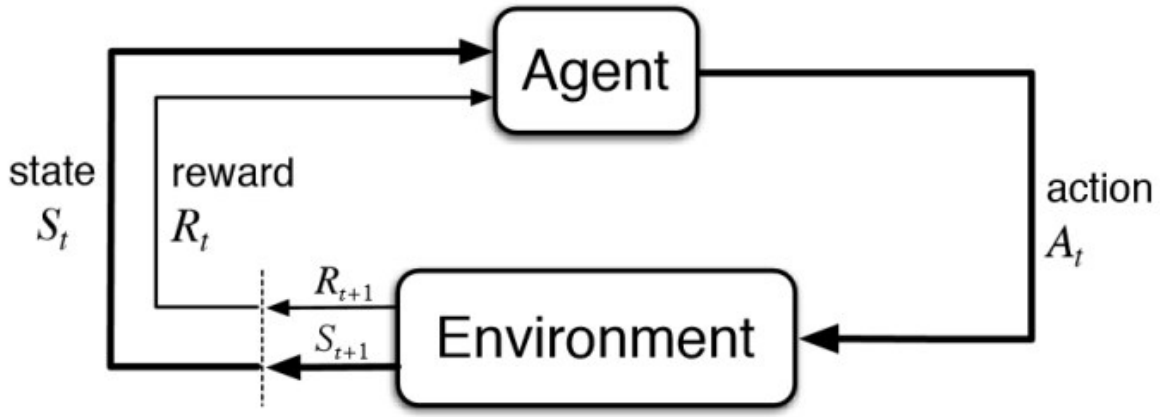
2. Background

2.1 – Markov Decision Process (MDP)

The Markov Property means that each *state* is dependent solely on its preceding state, the selected *action* taken from that state and the *reward* received immediately after that action was executed.
Mathematically, it means: $s' = s'(s,a,r)$, where s' is the future state, s is its preceding state and a and r are the action and reward. No prior knowledge of what happened before s is needed — the Markov Property assumes that s holds all the relevant information within it. A Markov Decision Process is decision process based on these assumptions.

2.1 – Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a machine learning approach to artificial intelligence concerned with creating computer programs that can solve problems requiring intelligence. The peculiar property of DRL algorithms is the learning through trial and error from feedback, that is simultaneously sequential, evaluative and sampled by leveraging powerful non-linear function approximations. DRL algorithms learns what to do, how to map situations to actions, so as to maximize a numerical reward signal.



Reinforcement Learning Illustration (https://i.stack.imgur.com/eoeSq.png)

2.1.1 – Policy (π)

The policy, denoted as **π**, is a mapping from some state *s* to the probabilities of selecting each possible action given that state. An optimal policy **π*** is the solution to the Markov Decision Process and it is the best strategy to accomplish its goal.

2.1.3 – State action value function (Q function)

A state-action function is also called the Q function. It specifies how good it is for an agent to perform a particular action in a state with a policy π.

We can define Q function as follows:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right]$$

This specifies the expected return starting in state *s* with the action *a* according to policy π.

2.1.4 – Q Learning Algorithm

Q-Learning is an *off-policy Reinforcement Learning* algorithm. In its most simplified form, it uses a table, also known as Q-table, to store all Q-values of all possible *state-action* pairs. It updates its table using the *Bellman equation,* while action selection is usually made with an ***ε***-greedy policy. The optimal Q-value, denoted as Q^* can be expressed as:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \lambda \max_{a'} Q^*(s',a') | s,a\right]$$

Optimal Q-value (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

	Action	
State	Left	Right
0	0	0
1	-100	65.61
2	59.049	72.9
3	65.61	81
4	72.9	90
5	81	100
6	0	0
7	100	81
8	90	72.9
9	81	0

Q-table Example

(https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8)

2.1.5 – Bellman Equation

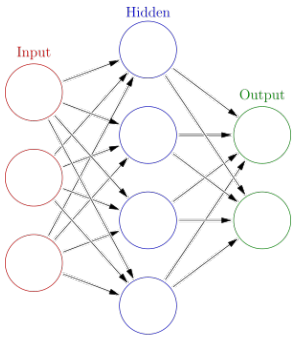
Defines the relationships between a given state (or state-action pair) to its successors. While many forms exist, the most common one usually encountered in *Reinforcement Learning* tasks is the Bellman equation for the optimal Q-value, which is given by:

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Bellman Equation

2.2 – Artificial Neural Networks

Artificial neural networks are computing systems that are inspired by biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules.



ANN architecture

2.3 – Deep Deterministic Policy Gradients Algorithm (DDPG)

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

In previous work, we have implemented Deep Q Learning algorithm. Using deep neural network function approximators, this algorithm could estimate the action-value function using unprocessed pixels for input. However, this project had continuous actions and DQN cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

Deep Deterministic Policy Gradients is an *off-policy actor-critic* algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. DDPG uses four neural networks: Q-network, a deterministic policy network, a target Q Network and a target policy network.

The policy network is called actor and directly maps states to actions. The Q Network is called critic Q (s, a) is learned using the Bellman equation as in Q-Learning. The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.

Parameters:

θ^Q : Q network

θ^μ : Deterministic policy function

$\theta^{Q'}$: target Q network

$\theta^{\mu'}$: target policy network

The standard Actor & Critic architecture for the deterministic policy network and the Q network is implemented in the "model.py" file.

2.3.1 – Replay Buffer

As used in DQN, we store each experienced tuple of the agent into a **replay buffer** as we are interacting with the environment and then sample a small batch of tuples from it in order to learn. As a result, we're able to learn from individual tuples multiple times, recall rare occurrences and in general make better use of experience.

2.3.2 – Actor (Policy) & Critic (Value) Network Updates

The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the update Q value and the original Q value:

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

For the policy function, our objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}[Q(s,a)|_{s=s_t, a_t=\mu(s_t)}]$$

In order to calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter.

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

However, since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batches:

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N}\sum_i [\nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}]$$

2.3.3 – Target Network Updates

We make a copy of the target network parameters and have them slowly track those of the learned networks via "soft updates", as the equations in the pseudocode:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$
$$\text{where} \quad \tau \ll 1$$

2.3.4 – Exploration

In DQN, exploration is done via probabilistically selecting a random action (such as epsilon-greedy exploration). For continuous action spaces, exploration is done via adding noise to the action itself. In the DDPG, the authors used *Ornstein-Uhlenbeck Process* to add noise to the action output:

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

2.4 – Environment

The goal of this project is to train an agent to maintain its position at the target location for as many as time steps as possible.

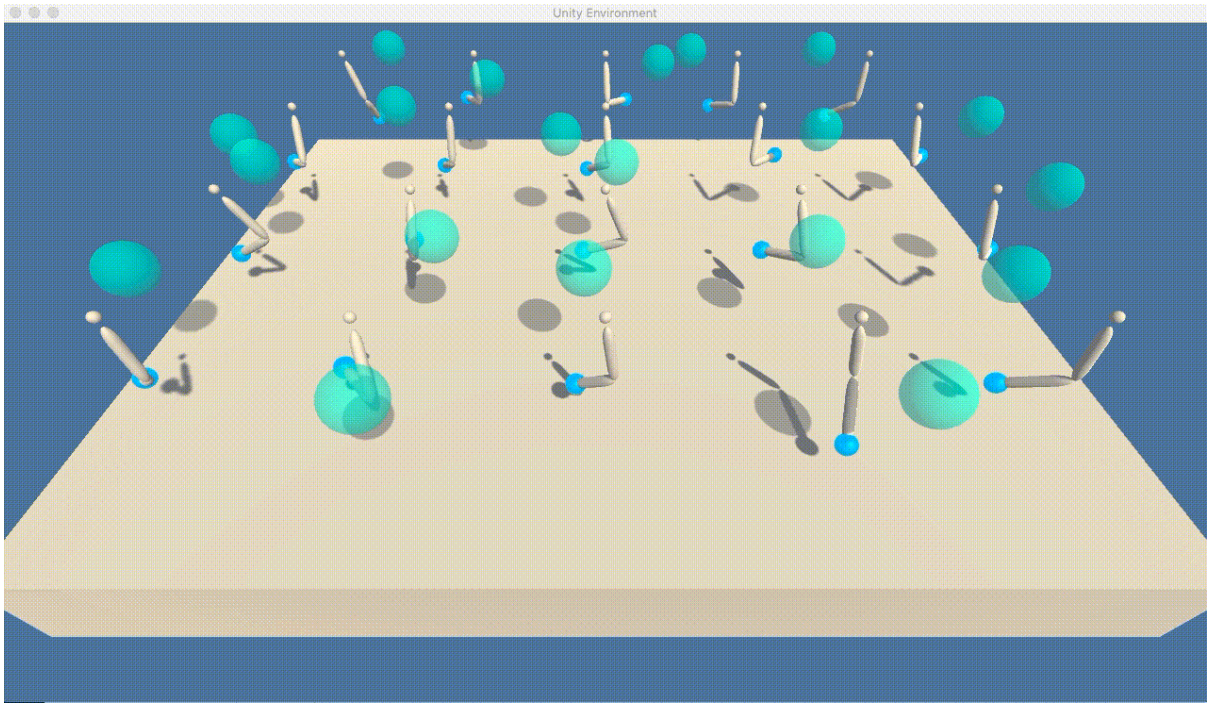
Details

This environment was developed on Unity Real-Time Development Platform and it has some peculiarities:

- The second version of the environment, which will be implemented, contains 20 identical agents, each with its own copy of the environment.
- 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 4 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 agents **must** get an average score of +30 (over 100 consecutive episodes and overall 20 agents).



Udacity's Environment

3.1 – Solution

The solution was implemented using Python 3 language programming. 3 scripts *(model.py, agent.py, Continuous_Control.py)* were created in order to maintain the code organized. The first script (model.py) contains the neural network architecture used for the solution. The Second (agent.py) contains the replay buffer as well as agent's class. Finally, the last script contains the function that will train the agent and plot scores in order to evaluate its performance.

3.2.2 – Hyperparameters

In order to gain optimal performance, hyperparameters tuning is very important. The following hyperparameters were set in the solution:

```
BUFFER_SIZE = int(1e6) # replay buffer size
BATCH_SIZE = 128       # minibatch size
GAMMA = 0.99           # discount factor
TAU = 1e-3             # for soft update of target parameters
LR_ACTOR = 1e-4        # learning rate of the actor
LR_CRITIC = 1e-4       # learning rate of the critic
WEIGHT_DECAY = 0.0     # L2 weight decay

N_LEARN_UPDATES = 10   # number of learning updates
N_TIME_STEPS = 20      # every n time step, update
```

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=1.5, sigma=0.4):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()
```

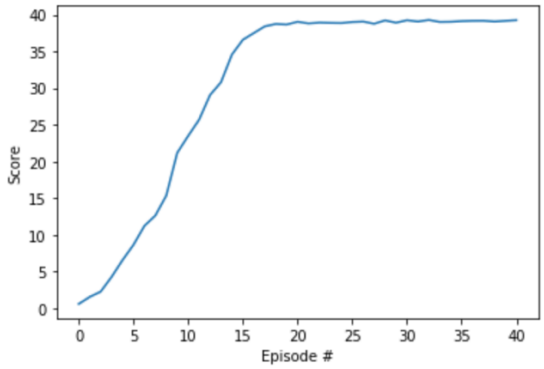
We used Adam for learning the neural network parameters with a learning rate of 1e-4 for both actor and critic. A discount factor was set to 0.99. For the soft target updates, we used TAU = 1e-3. The neural networks used the rectified non-linearity for all hidden layers. The final output layer of actor was a tanh layer, to bound the actions. The low-dimensional networks had 2 hidden layers with 256 and 128 nodes respectively. Actions were not included until the second hidden layer of Q. The other layers were initialized from the uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer.

4 – Plot of Rewards

The environment considered as being solved if an agent would get an average of +30 reward over 100 consecutive episodes. The result is an average +30.18 reward in 41 episodes, implemented by DDPG.

At first attempt, the hyperparameters of the Uhlenbeck & Ornstein noise were set to $\theta = 0.15$ and $\sigma = 0.2$. However, the agent achieved a slow performance. After 5 hours of training, the average score started to decay and it was not possible to solve the environment. However, after the parameters were set to $\theta = 1.5$ and $\sigma = 0.4$, the agent had a higher exploration rate

and solved the environment after +/- 30 minutes. The following image is the agent's learning scores:



5 – Ideas for Future Work

For further performance improvement, we could implement Proximal Policy Optimization (PPO) algorithm which demonstrated good performances for complex environments as well as Distributed Distributional Deterministic Policy Gradients, which combines the very successful distributional perspective on reinforcement learning, with a distributed framework for off-policy learning. The following image contains a comparison between both methods:

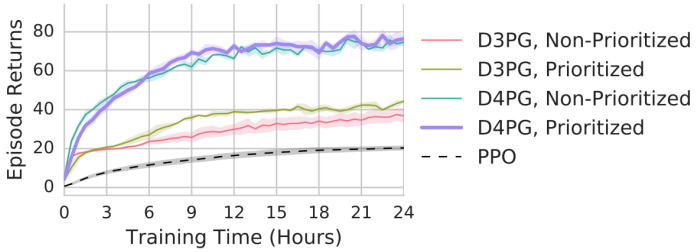


Figure 1: Experimental results for the three-dimensional (humanoid) parkour domain.

6 – References

Zychlinski, Shaked. "The Complete Reinforcement Learning Dictionary"
https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e

Google DeepMind, "Distributed Distributional Deterministic Policy Gradients"
https://arxiv.org/pdf/1804.08617.pdf

Yoon, Chris. "Deep Deterministic Policy Gradients Explained"
https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

Google DeepMind, “Continuous Control with Deep Reinforcement Learning”
https://arxiv.org/pdf/1509.02971.pdf

Google DeepMind, “Proximal Policy Optimization Algorithms”
https://arxiv.org/pdf/1707.06347.pdf