Deep Reinforcement Learning Expert Nanodegree

João Pedro Megid Carrilho

Report
Project 2: Continuous Control

Introduction

In this project an Actor-Critic Method of Reinforcement Learning is utilized to train an agent in a 3D simulated environment. The agent is represented as a robotic arm and It's goal is to learn how to maintain Its position at a target location, represented by an orbiting sphere.

The project is composed by five files: *Continuos_Control.ipynb* a Jupyter Notebook file containing the main code to initialize dependencies, environment and Agent; *ddpg_agent.py*, a code that contains the characteristics of the agent and how it behaves through this task; *model.py,* containing the deep neural networks (*DNN*) architectures used by the agent; *checkpoint_actor.pth* and *checkpoint_critic.pth* files with saved *DNN's* weights, that solved the environment.

The problem trying to be solved is modeled as a Markov Decision Process, involving *mappings* from states to actions, called *policy,* in such way that these actions will maximize the total cumulative reward signal of the agent. States are any information that the environment provides the agent, excluding the reward signal. Actions are ways that an agent can interact with the environment, and rewards are signals that the agent receive after interacting with the environment, shaping Its learning.

The solution to the problem, on this project, is obtained by utilizing a Policy-Based Method called Deep Deterministic Policy Gradient (*DDPG)*. Using *DNN's* as non-linear function approximators, the algorithm cana approximate an optimal *policy*. The model take as input a given *state* and outputs the best action to be taken, in that state.

Implementation

Preparation

The goal is to train an agent able to get an average score of +30 over 100 consecutive episodes. The scores are distributed like follows: +0.1 each step the agent's hand is in goal location. As the agent interacts with the environment, the reward signal guides it towards maintaining contact with the target location.

At first, in the notebook file, the dependencies are installed, libraries are imported, the simulation environment is initialized.

The next step is to explore the State and Action Spaces. The State 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.

To learn how the Python API controls the agent and receives the feedbacks from the environment, a code cell is provided with a random action agent.

Learning Algorithm

The *ddpg* algorithm is an approximate Actor-Critic Method, but also resembles the DQN approach of Reinforcement Learning. The agent is composed of two Neural Networks (*NNs*) the Actor and the Critic, both with target and local networks totalizing 4 *NNs*.

The learning pipeline takes first a state as input in the Actor network, outputting the best possible action in that state, this procedure makes possible for ddpg to tackle continuous action spaces, in contrast to the regular DQN approach. This action is used in the Critic network, alongside with the state, where it outputs an action value function (*q*), this *q* is used as a baseline for updating both Actor and Critic networks, reducing the variance and instability of classic RL algorithms. The optimization is done with a gradient ascent between both Actor's and Critic's target and local networks parameters.

The behaviour of the agent can be explored in the *ddpg_agent.py* file. Important libraries and components are imported and local parameters are initialized: *BUFFER_SIZE,* defines the replay buffer size, this is an object that contains tuples called experiences composed by state,actions,rewards,next states and dones, these are necessary informations for learning; *BATCH_SIZE*, when the number of experiences in the replay buffer exceeds the batch size, the learning method is called; *TAU*, this hyperparameter controls the model *soft updates*, a method used for slowly changing the target networks parameters slowly, improving stability; *LR_ACTOR* and

LR_CRITIC, the optimizer learning rates, these control the gradient ascent step; *WEIGHT_DECAY*, the l2 regularization parameter of the optimizer.

The main implementation begins on fourth step: additional libraries and components are imported, an *agent* is created and initialized with proper parameters: *state_size* and *action_size.* The *ddpg* function is created, taking as parameters the number of episodes (*n_episodes)* and the maximum length of each episode (*max_t*).

In each episode the environment is reseted and the agent receives an initial state. While the number of timesteps is less than *max_t,* the following procedures are done:

The agent use it's *act* method with the current state as input, the method takes the input and passes it through the actor network, returning an action for the state. A environment *step* is taken, using the previous obtained action, and it returns: next state, rewards and dones (if the episode is terminated or not). These are stored in the *env_info,*variable, that passes them individually for each of these information's new variables. The agent uses it *step* method, the method first adds the experience tuple for the replay buffer and, depending on the size, calls the *learn* method. The rewards are added to the scores variable and the state receives the next state, to give continuation to the environment, if any of the components of the done variable indicates that the episode is over, the loop of *max_t* breaks, and a new episode is initialized.

If the average score of the last 100 episodes is bigger than 30, the networks weights are save and the loop of *n_episodes* breaks and the *ddpg* function returns a list with each episode's score. This list is plotted with the episodes number, showing the Agent's learning during the algorithm's execution.

Neural Network Architecture

Actor

Composed of three hidden layers with 600, 400, 200 nodes respectively. Each hidden layer is followed by a ReLU activation function. The output layer is followed by a Tanh function, making possible tackling continuous action spaces. The network take as input the state and outputs the best calculated action for this.

Critic

Composed of two hidden layers with 400+(action space size), 200 nodes respectively. Each hidden layer is followed by a ReLU activation function. The network take as input the state and outputs the action value function for the best action outputted by the Actor network.

Rewards per Episode

Next, an image of the Agent's rewards obtained in each episode until solving the environment.

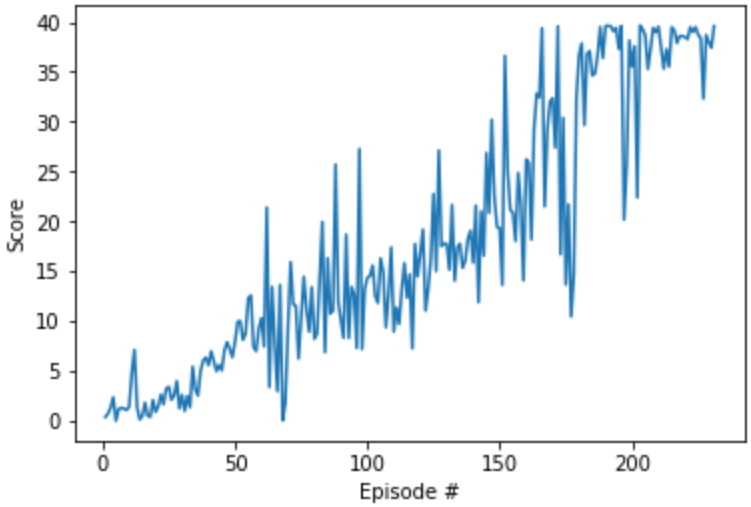

Figure 1: Rewards per episode plot of the solving agent.

Results

Simply applying the ddpg algorithm to the environment didn't resulted in learning. Then, by changing hyperparameters the agent started to learn and solved the environment. Initially learning only how to maintain contact during half of the target's orbit and, with some more changes, successfully solving it.

Episode	Average Score
50	3.19
100	7.15
150	13.82
200	23.03
231	30.08

Table 1: Average scores in their following episode.

A table with used hyperparameters is included below:

Hyperparameter	Value
BUFFER_SIZE	int(1e5)
BATCH_SIZE	128
GAMMA	0.99
TAU	$1e^{-3}$
LR_ACTOR	1.5e-4
LR_CRITIC	1.5e-4
WEIGHT_DECAY	0.0001

Table 1: Hyperparameters used by the best solving agent.

Ideas for Future Work

The advantages of policy-based method allows Reinforcement Learning to tackle continuous problems, approximating the real world. This motivates the intention to create devices such as the physical double jointed arm mentioned in the *The Environment - Real World* , it would be interesting to create mechanisms that can, initially, operate in environments like the Project 1: Navigation. Also, studying more about reward function designs and how to create Unity ML-Agents simulations.

Regarding this project, besides solving the 20 arms and the Crawler environment, there are many improvements that can and should be implemented, prioritized experience replay is one of these. Also the D4PG algorithm, that showed state of art results should be used and compared.