Deep Reinforcement Learning Nanodegree

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:

Continuous_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 modelled 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 can approximate an optimal policy. The model take as input a given state and outputs the best action to be taken, in that state.

Implementation

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.

Hyperparameters :

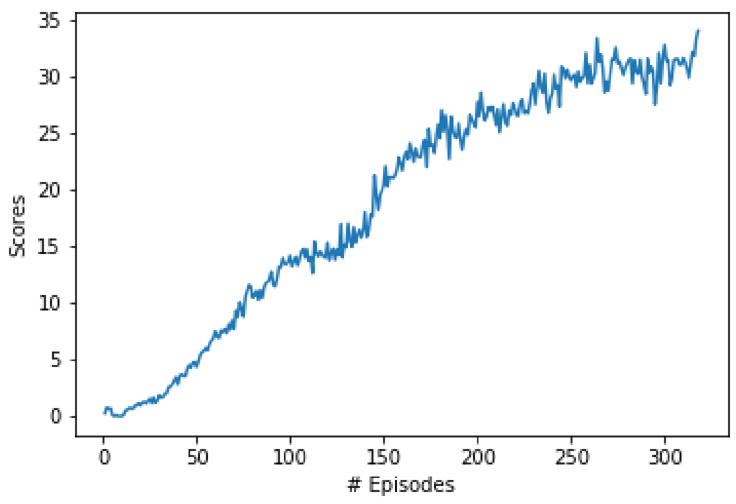
The hyperparameters used in this experiment are given below :

Hyperparameter	Value
Replay Buffer Size	1e6
Batch Size	1024
Gamma (discount factor)	0.99
tau	1e-3
Actor learning rate	1e-4
Critic learning rate	3e-4
Update interval	20
Update times per interval	10
Leak for Leaky ReLU	0.01

Plot of Rewards :

```
Episode: 10     Average Score: 0.29     Current Score: 0.04
Episode: 20     Average Score: 0.52     Current Score: 0.98
Episode: 30     Average Score: 0.81     Current Score: 1.89
Episode: 40     Average Score: 1.23     Current Score: 2.84
Episode: 50     Average Score: 1.81     Current Score: 4.37
Episode: 60     Average Score: 2.53     Current Score: 7.54
Episode: 70     Average Score: 3.26     Current Score: 7.73
Episode: 80     Average Score: 4.13     Current Score: 10.50
Episode: 90     Average Score: 4.93     Current Score: 12.78
Episode: 100    Average Score: 5.74     Current Score: 14.21
Episode: 110    Average Score: 7.12     Current Score: 13.75
Episode: 120    Average Score: 8.47     Current Score: 15.36
Episode: 130    Average Score: 9.81     Current Score: 14.94
Episode: 140    Average Score: 11.18    Current Score: 18.04
Episode: 150    Average Score: 12.64    Current Score: 20.53
Episode: 160    Average Score: 14.18    Current Score: 21.73
Episode: 170    Average Score: 15.74    Current Score: 22.94
Episode: 180    Average Score: 17.14    Current Score: 24.58
Episode: 190    Average Score: 18.54    Current Score: 25.91
Episode: 200    Average Score: 19.79    Current Score: 27.81
Episode: 210    Average Score: 21.07    Current Score: 25.75
Episode: 220    Average Score: 22.31    Current Score: 27.28
Episode: 230    Average Score: 23.59    Current Score: 29.44
```

```
Episode: 240     Average Score: 24.83     Current Score: 28.54
Episode: 250     Average Score: 25.94     Current Score: 29.71
Episode: 260     Average Score: 26.80     Current Score: 31.08
Episode: 270     Average Score: 27.53     Current Score: 28.77
Episode: 280     Average Score: 28.22     Current Score: 31.18
Episode: 290     Average Score: 28.72     Current Score: 28.48
Episode: 300     Average Score: 29.23     Current Score: 32.83
Episode: 310     Average Score: 29.64     Current Score: 31.67
Episode: 318     Average Score: 30.07     Current Score: 34.09
Environment solved in 218 episodes!       Average Score: 30.07
```



Ideas for future work :

- Implementation of other algorithms like TRPO, PPO, A3C, A2C would result in better performance.
- General optimization techniques like cyclical learning rates and warm restarts could be useful as well.