Udacity Deep Reinforcement Learning Nanodegree

Project #2 Continuous Control

Report

1. Learning Algorithm

The algorithm implemented is the *DDPG (Deep Deterministic Policy Gradient) algorithm*, which is a particular type of Actor-Critic method. It can be seen as an approximate DQN algorithm for continuous action space.

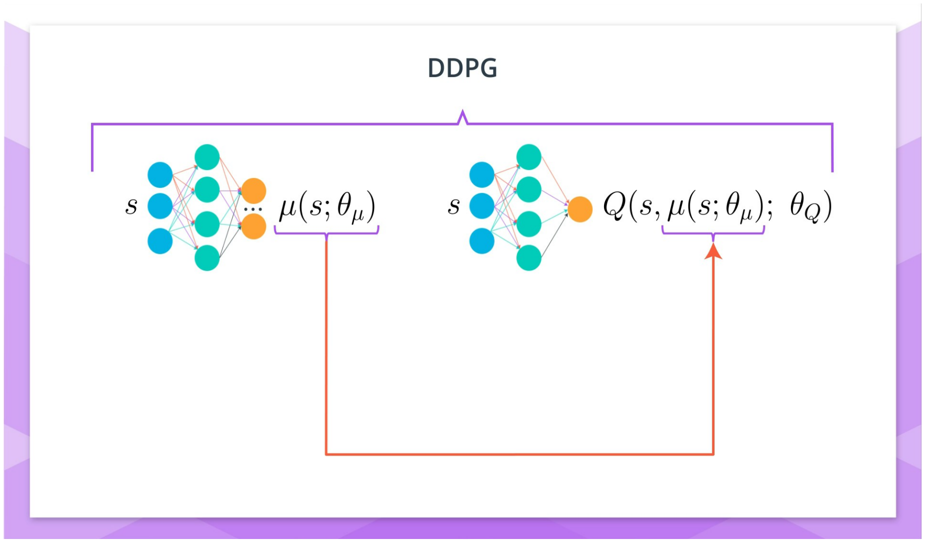
The Critic part, in fact, is used to approximate the maximizer over the Q values of the next states, which, in the case of continuous action spaces, would be too expensive to be calculated at each step of the agent, which would make the simple DQN algorithm not scalable with continous action spaces. The DDPG solves this problem well.

We have 2 neural networks, one for the Actor, one for the Critic.

The Actor approximates the optimal policy deterministically (i.e. it outputs the best believed action for any given state) and is it basically trying to learn the argmax over all actions for any state.

So the output action of the Actor, in conjunction with the state, is used to calculate a new target value for training the action-value function of the Critic.

The Critic, hence, is learning to evaluate the optimal value function by using the Actor best believed actions.



The Deep Deterministic Policy Gradient algorithm, as explained on Udacity Deep Reinforcement Learning Nanodegree.

2. Implementation

In this particular implementation, the DDPG agent trains 4 neural networks:

- Actor Local (4 layers):
  - 1 with size equal to the state space (33 variables for the test environment)
  - 1 fully connected layer with 256 units
  - 1 fully connected layer with 128 units
  - 1 final layer with size equal to the action space (4 in the test environment)
    - The output of the final layer is then transformed with the Tanh function in order to get the actual action
- Actor Target
  - Same as Actor Local
- Critic Local (4 layers):
  - 1 with size equals to the state space (33 in the test environment)
  - 1 fully connected layer with 256 units + the size of the action space (as the action is concatenated into and passed to this layer)
  - 1 fully connected layer with 128 units
  - 1 final layer with size equal to one (the believed Q-value)
- Critic Target
  - Same as Critic Local

The Target versions of the neural networks are necessary to implement the soft update strategy, where, instead of updating the neural network after a fixed number of timesteps, they are instead updated every time step by blending in the weights into the target network with a smoothing parameter governed by an hyperparameter, which has been shown to improve the stability of the learning process.

In order to guarantee the exploration of the environment, a noise factor, using the Ornstein-Uhlenbeck process, is applied to the actions returned by the Actor network.

As activation function between the layers, the ReLu function has been chosen.

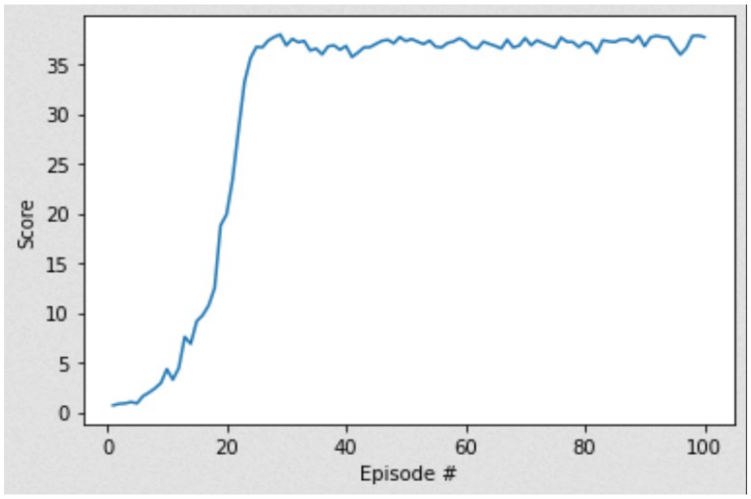
The use of a Replay Buffer is necessary to learn from samples that are independent and identically distributed. Also, the learning does not happen at every timestep but, instead, it happens after a fixed number of timesteps and more than one update pass is applied at once (all regulated by hyperparameters).

3. Hyperparameters

For solving the environment, the hyperparameters have been chosen as following:

```
BUFFER_SIZE = int(1e5)  # 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        # L2 weight decay
LEARN_EVERY = 5         # Learn every n steps
LEARN_HOWMANY = 10      # Learn how much every time
```

4. Plot of Rewards



Plot of rewards over episodes.

The agent was able to solve the environment (i.e. get *an average reward of all the agents of +30 over 100 consecutive episodes*) in **100** episodes, which is the minimum possible number of episodes.

5. Ideas for Future Work

There are some possible improvements for the DDQN algorithm:

1. Prioritized Experience Replay: DDQN samples experience transitions *uniformly* from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.

2. Other environments: It would be nice to experiment on other environments with the same DDPG implementation in order to be able to compare the results, such as the proposed Crawler environment challenge.
3. The use of pixels as input would be a nice to have feature, updating the Neural Network in order to accommodate some initial Convolutional Layers as needed.

6. Sources

- **https://github.com/udacity/deep-reinforcement-learning/tree/master/p2_continuous-control**
- **https://deepai.org/machine-learning-glossary-and-terms/relu**
- **https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893**