Andrea Pena

Udacity DRLND

Project 2: Navigation

Description of the Project

Note: To resolve this project (P1: Navigation), I used a similar architecture that I used in the Deep Deterministic Policy Gradients (DDPG) coding exercise in this lesson.

Goal of the Project

For this project, you will work with the Reacher environment provided by Udacity.

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.

Note: This project provides two different versions, one agent or 20 agents. I decided to use the second version that contains 20 identical agents with their own copy environment.

Project requirements: See 'requirements.txt'

Environment Detail

- Set-up: Double-jointed arm which can move to target locations.
- Goal: Each agent must move its hand to the goal location and keep it there.
- Agents: The environment contains 20 agents linked to a single Brain.

- Agent Reward Function (independent): +0.1 for each timestep agent's hand is in goal location.
- Brains: One Brain with the following observation/action space.
- Vector Observation space: 33 variables corresponding to position, rotation, velocity, and angular velocities of the two arm Rigidbodies.
- Vector Action space: (Continuous) 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.
- Visual Observations: None.
- Reset Parameters: Two, corresponding to goal size, and goal movement speed.
- Benchmark Mean Reward: 30

Learning algorithm.

The algorithm used is Deep Deterministic Policy Gradients (DDPG). To be more specific, I used the algorithm mentioned in the "*Continuous Control with Deep Reinforcement Learning,* by researchers at Google Deepmind. In this paper, the authors present "a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces." They emphasize that DDPG can be viewed as an extension of Deep Q-learning for continuous tasks.

DDPG Hyper Parameters

- n_episodes=500
- max_t=1000
- solved_score=30.0
- consec_episodes=100
- print_every=1

- train_mode=True
- actor_path='actor_ckpt.pth'
- critic_path='critic_ckpt.pth'

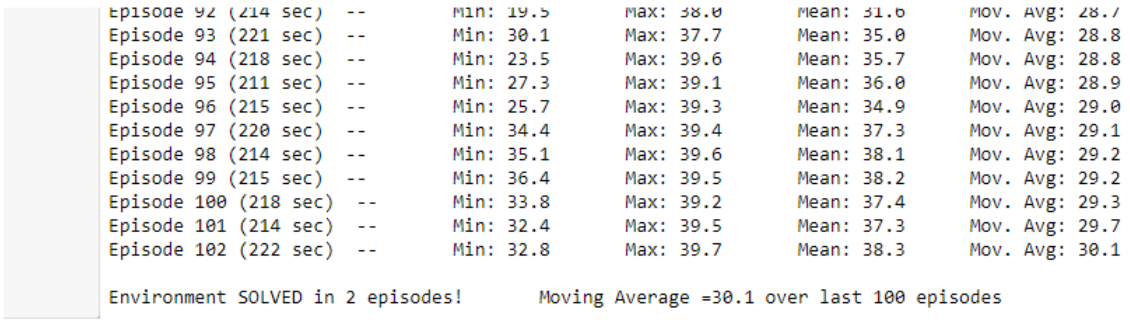
DQN Agent Hyper Parameters

- 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-3   # learning rate of the actor
- LR_CRITIC = 1e-3   # learning rate of the critic
- WEIGHT_DECAY = 0   # L2 weight decay
- LEARN_EVERY = 20   # learning timestep interval
- LEARN_NUM   = 10   # number of learning passes
- GRAD_CLIPPING = 1.0   # Gradient Clipping
- OU_SIGMA = 0.2
- OU_THETA = 0.15
- EPSILON   = 1.0   # for epsilon in the noise process (act step)
- EPSILON_DECAY = 1e-6

Neural Network

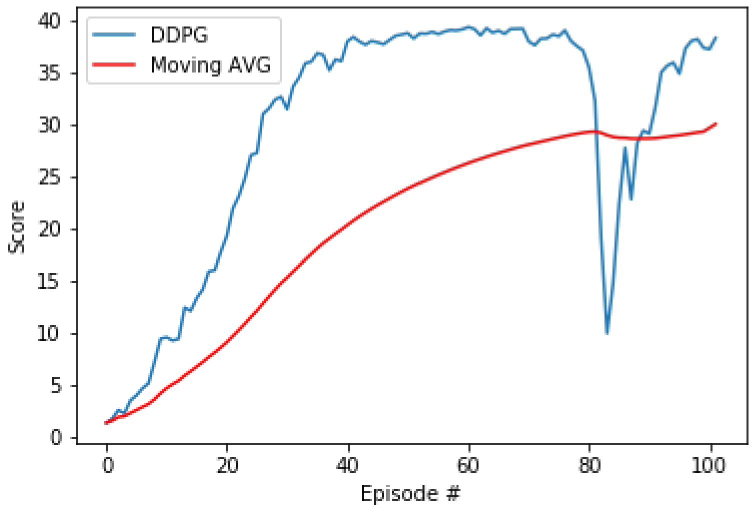
The neural network is saved in the file model.py. In this case, both Actor and Critic both models have 33 nodes for input layer, two hidden layers and an output layer. For the first hidden layer, actor model has 400 nodes whereas critic model has 400 state nodes plus 4 action nodes. For the second hidden layer, both have 300 nodes. Finally, the output layer for actor model has 4 nodes ranging between -1 to 1 therefore hyperbolic tangent is used for the activation function whereas critic model has only 1 node without activation function. Both models use batch normalization as well as Xavier uniform weight initialization.

Plot of Rewards



```
Episode 92 (214 sec)  --          Min: 19.5        Max: 38.0        Mean: 31.6        Mov. Avg: 28.7
Episode 93 (221 sec)  --          Min: 30.1        Max: 37.7        Mean: 35.0        Mov. Avg: 28.8
Episode 94 (218 sec)  --          Min: 23.5        Max: 39.6        Mean: 35.7        Mov. Avg: 28.8
Episode 95 (211 sec)  --          Min: 27.3        Max: 39.1        Mean: 36.0        Mov. Avg: 28.9
Episode 96 (215 sec)  --          Min: 25.7        Max: 39.3        Mean: 34.9        Mov. Avg: 29.0
Episode 97 (220 sec)  --          Min: 34.4        Max: 39.4        Mean: 37.3        Mov. Avg: 29.1
Episode 98 (214 sec)  --          Min: 35.1        Max: 39.6        Mean: 38.1        Mov. Avg: 29.2
Episode 99 (215 sec)  --          Min: 36.4        Max: 39.5        Mean: 38.2        Mov. Avg: 29.2
Episode 100 (218 sec)  --         Min: 33.8        Max: 39.2        Mean: 37.4        Mov. Avg: 29.3
Episode 101 (214 sec)  --         Min: 32.4        Max: 39.5        Mean: 37.3        Mov. Avg: 29.7
Episode 102 (222 sec)  --         Min: 32.8        Max: 39.7        Mean: 38.3        Mov. Avg: 30.1

Environment SOLVED in 2 episodes!       Moving Average =30.1 over last 100 episodes
```



Future Work

I would recommend for future improvement a better hyperparameter implementation because I have a little trouble keeping up the score in the last episodes. Besides, it also will be a good idea try different algorithms to solve this environment, such as TRPO (Trust Region Policy Optimization or D4PG (Distributed Distributional Deterministic Policy Gradients). Also, add prioritized experience replay instead of the random selection of tuples. In that way, the prioritized will select experiences based on a priority value which is correlated with the magnitude error. In that manner, it will improve the learning by increasing the probability of the experience vectors that are sampled.