Continuous control report

For this project we were given 2 options, and option 2 was chosen:

Option 1: Solve the First Version

The task is episodic, and in order to solve the environment, your agent must get an average score of +30 over 100 consecutive episodes.

Option 2: Solve the Second Version

The barrier for solving the second version of the environment is slightly different, to take into account the presence of many agents. In particular, your agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically,

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores.
- This yields an **average score** for each episode (where the average is over all 20 agents).

Deep DPG

For this project we implemented a model-free algorithm called Deep Deterministic Policy Gradient. While requiring larger number of learning episodes to provide a solution to continuous action space problems, DDPG is a rather simple method that operates through an actor critic architecture and a learning algorithm.

Using the Bellman equation, the agents are able to learn the Q-function, and through the Q-function learns the optimal policy. However, the algorithm is able to learn through large, non-linear functions because it trains the network with samples from a replay buffer rather than the policy space, hence reducing correlation between states. Next, another network is trained with a target Q network.

In order to apply Q-learning to large continuous spaces and avoid having to optimize the action state at each timestep, the proposed algorithm utilizes and actor-critic approach.

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s,a) - y(r,s',d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

    end for
  end if
until convergence

Image obtained from https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Implementation description:

The submission consists on 3 files:

Model.py: Contains the Actor and Critic classes, both containing a Target and a Local Neural Network for training.
Ddpg_agent.py: Contains the DDPG agent, a Noise (Ornstein-Uhlenbeck process)
, and a Replay Buffer class.
Continuous_Controil.ipiynb: Imports the packages, Train 20 agents using DDPG, and plot Socres and rewards

Hyperpatameters:

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-3 # learning rate of the critic
WEIGHT_DECAY = 0 # L2 weight decay
LEARN_EVERY = 20 # Update the networks 10 times after every 20 timesteps
LEARN_NUMBER = 10 # Update the networks 10 times after every 20 timesteps
EPSILON = 1.0 # Noise factor
EPSILON_DECAY = 0.999999 # Noise factor decay

The architecture of the Networks are the following :

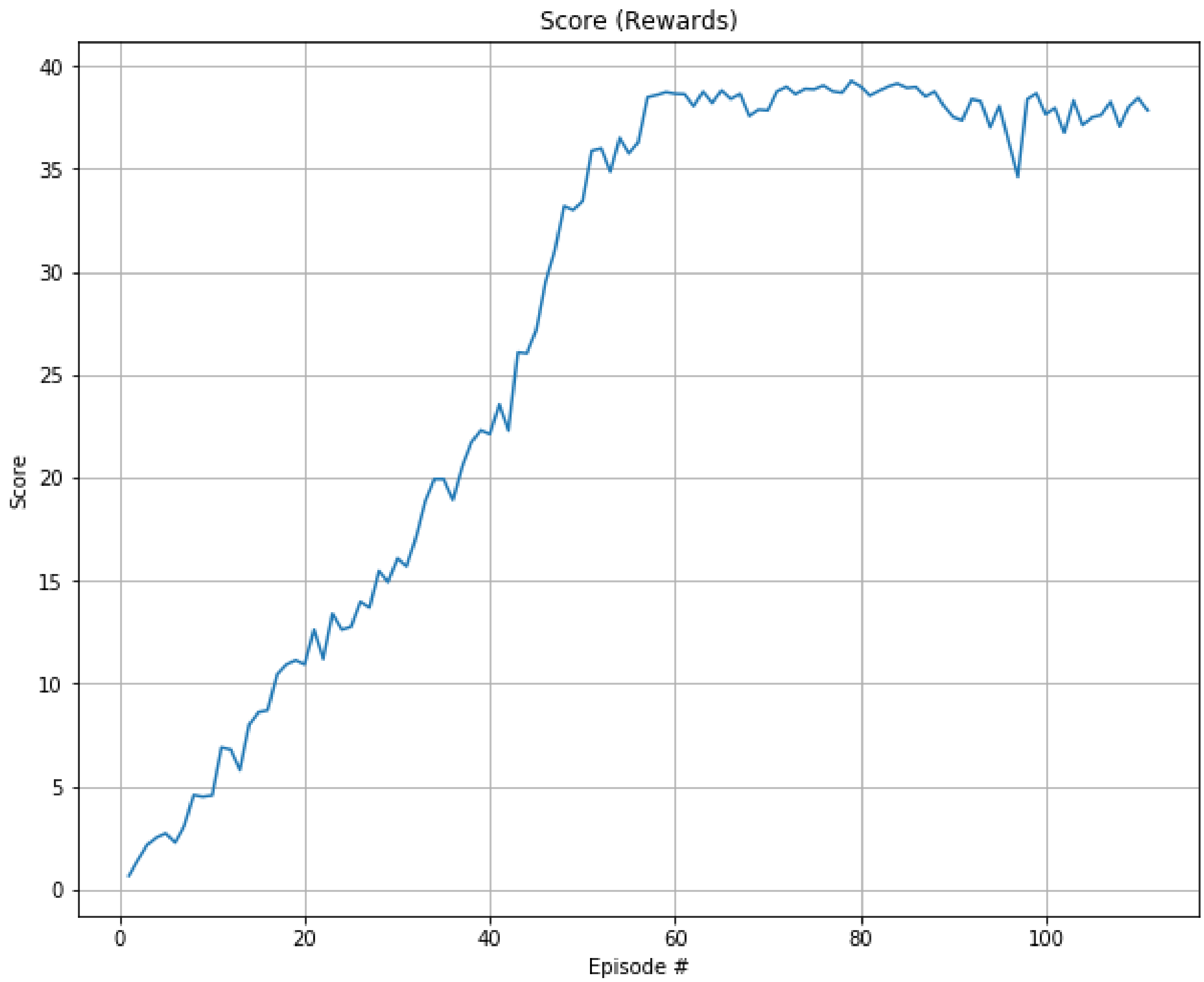
Actor:
First layer: input= 33, output= 400
Second layer: input= 400, output= 300
Third layer : input= 300, output= 4

Critic
First layer: input= 33, output= 400
Second layer: input= 400, output= 300
Third layer : input= 304 (plus actions), output= 1

Results:

```
Using:  cuda:0
Episode 10  Mean_reward: 4.57  Average100 Score: 2.85
Episode 20  Mean_reward: 10.93  Average100 Score: 5.84
Episode 30  Mean_reward: 16.08  Average100 Score: 8.45
Episode 40  Mean_reward: 22.12  Average100 Score: 11.26
Episode 50  Mean_reward: 33.45  Average100 Score: 14.72
Episode 60  Mean_reward: 38.65  Average100 Score: 18.43
Episode 70  Mean_reward: 37.85  Average100 Score: 21.26
Episode 80  Mean_reward: 39.03  Average100 Score: 23.47
Episode 90  Mean_reward: 37.53  Average100 Score: 25.15
Episode 100  Mean_reward: 37.66  Average100 Score: 26.39
Episode 110  Mean_reward: 38.45  Average100 Score: 29.87

Environment solved in 11 episodes!   Average100 Score: 30.18
```



Ideas for future work:

Implement other algorithm like PPO, A3C, or D4PG.