Project

In this environment, a double-jointed arm can move to target locations. A reward 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.

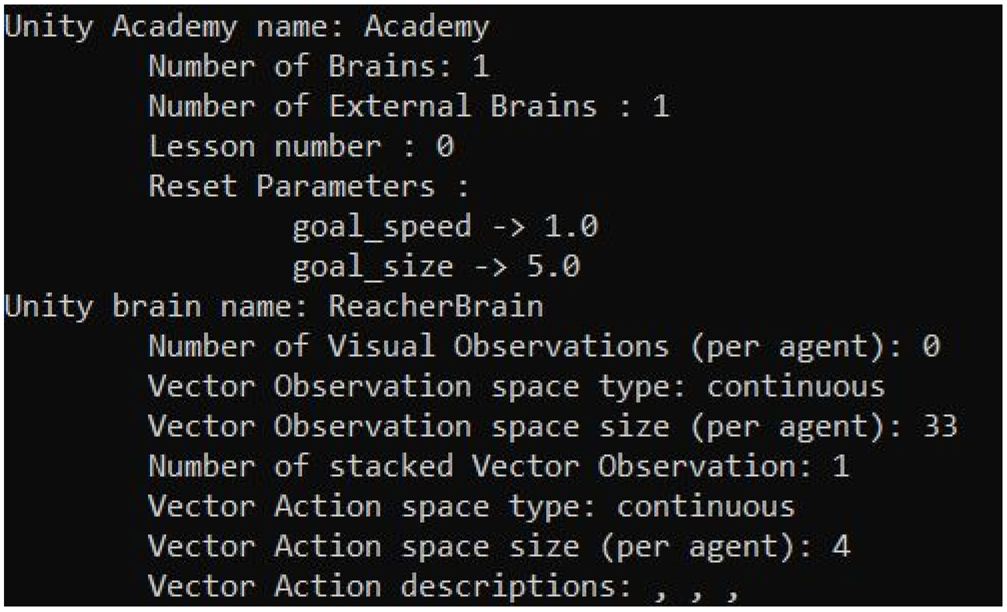
The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

I opted for solving the second version of the problem. The agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically,

- After each episode, I add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. I then take the average of these 20 scores.

- This yields an **average score** for each episode (where the average is over all 20 agents).

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md



```
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
                goal_speed -> 1.0
                goal_size -> 5.0
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

Implementation

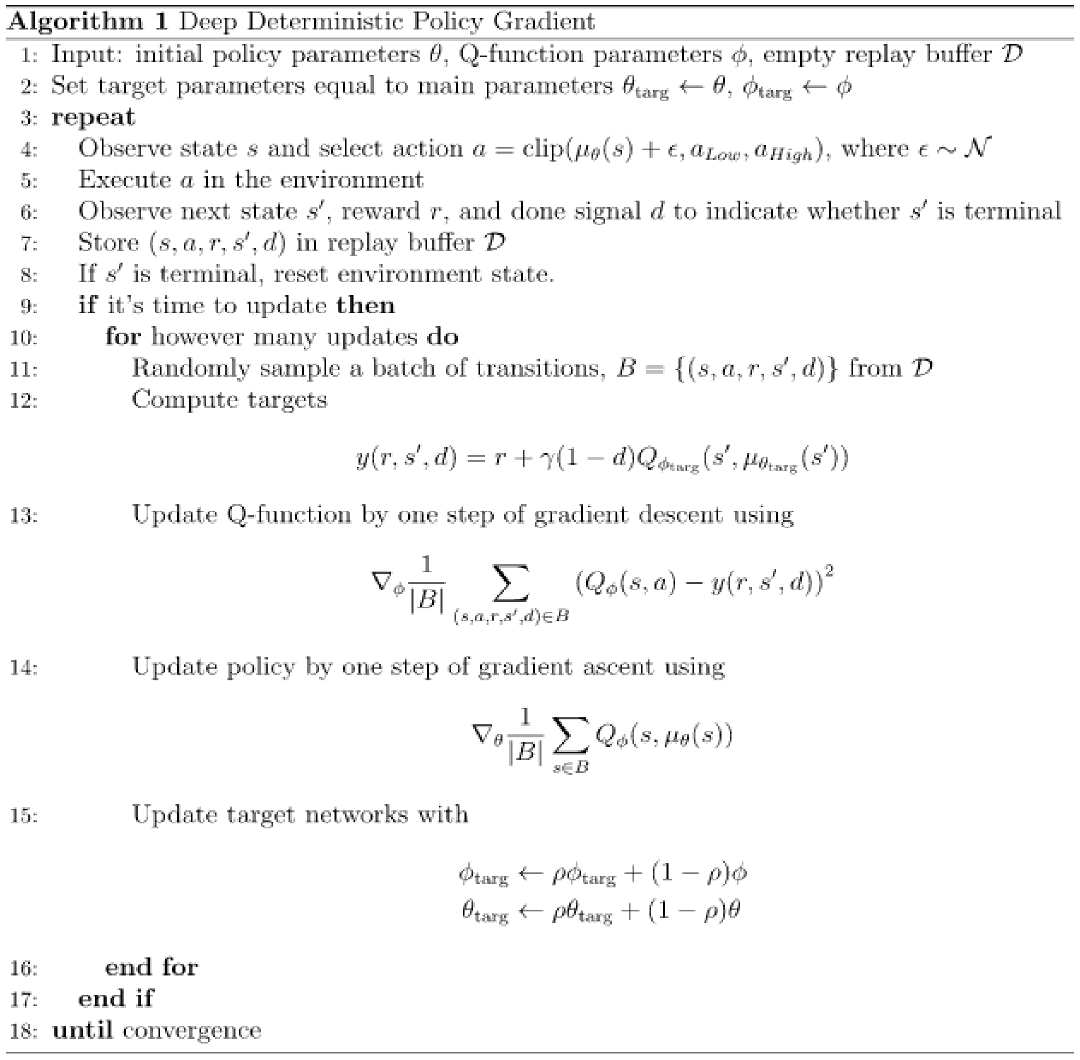
We have implemented a Deep Deterministic Policy Gradient (DDPG) agent without exploration noise, just to mimic the presented theory in the classroom.

DDPG Algorithm

In order to apply this algorithm I used two main resources (*in addition to the Udacity content itself*):

DDPG original paper: https://arxiv.org/pdf/1509.02971.pdf
Udacity - DDPG Pendulum implementation:
https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum
Open AI DDPG: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Algorithm

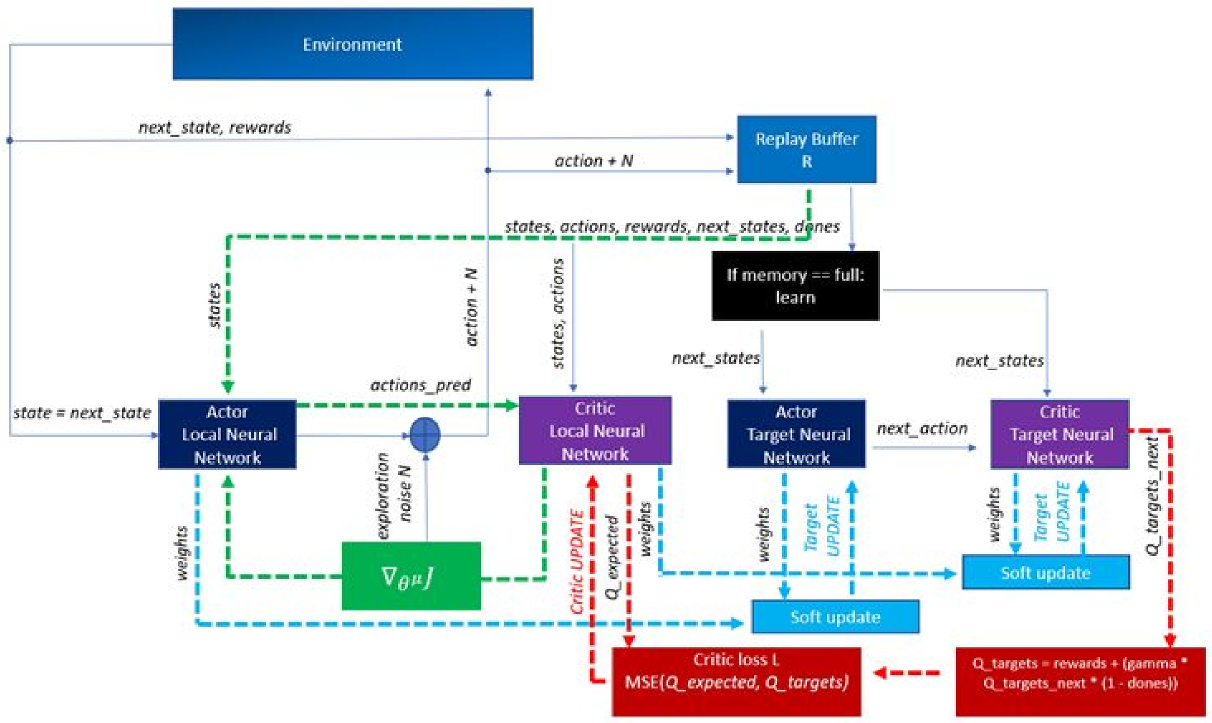


Algorithm 1 Deep Deterministic Policy Gradient

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:     Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:     Execute $a$ in the environment
6:     Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:     If $s'$ is terminal, reset environment state.
9:     if it's time to update then
10:        for however many updates do
11:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:            Compute targets
                   $y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:            Update Q-function by one step of gradient descent using
                   $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:            Update policy by one step of gradient ascent using
                   $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:            Update target networks with
                   $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$
                   $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$
16:        end for
17:    end if
18: until convergence
```

Deep Q-Learning models are not straightforwardly applied to continuous tasks due to the high number of actions (inf) in the continuous domain. We can't apply the argmax method to that amount of possibilities. Here is where DDPG comes in. DDPG is like an "extension" of the DQN algorithm in order to work with continuous environments.

Structure (taken from https://miro.medium.com/max/1508/1*x6HIECcvr60kzSHdDU0zZw.png)



Although we are using an Actor-Critic approach, the roles of both are different than the explained ones in the A2C classroom section. The actor implements a deterministic policy to map states to the best action available. The critic is trained as the Q-Learning method, but the next action in the equation is given from the Actor's target output. The actor is trained using the gradient from maximizing the estimated q-value from the critic. The actor's predicted action (*the best one in that moment*) is used as the input to the critic.

Two important things to highlight is that DDPG uses a Replay buffer to gather experiences from the agent and the target networks are updated using soft updates, mixing both neural networks slowly. This improves the convergence of our model.

Neural Network structure

Actor (*2 equal Neural Networks - Regular and Target*)

Input -> Vector Observation size
Hidden Layer 1 - 128 Units with Batch Normalization
Hidden Layer 2 - 64 Units with Batch Normalization
Hidden Layer 3 - 32 Units with Batch Normalization
Hidden Layer 4 - 128 Units with Batch Normalization
Output layer -> Action size -> Used tanh as output activation function

Critic (*2 equal Neural Networks - Regular and Target*)

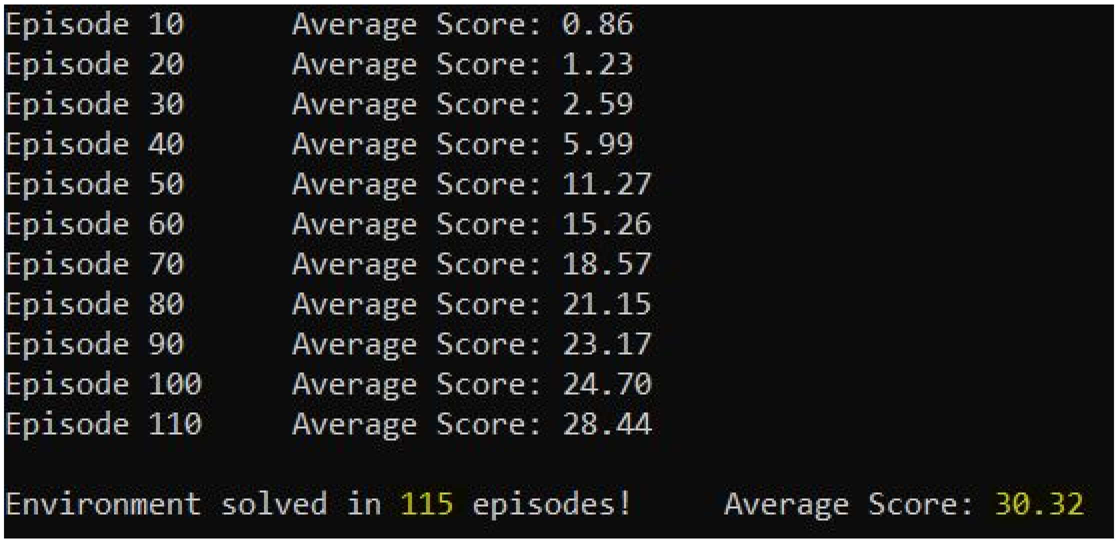
Input -> Vector Observation size
Hidden Layer 1 - 128 Units with Batch Normalization
Hidden Layer 2 - 128 + action_size
Hidden Layer 3 - 64 units
Hidden Layer 4 - 32 units
Hidden Layer 5 - 16 units
Output layer -> 1

We use SELU as an activation function. The optimizer used is Adam.

The hyperparameters chosen to reach the goal after some trials are:

```
# Replay Buffer Size
BUFFER_SIZE = int(1e6)
# Minibatch Size
BATCH_SIZE = 256
# Discount Gamma
GAMMA = 0.995
# Soft Update Value
TAU = 1e-2
# Learning rates for each NN
LR_ACTOR = 1e-3
LR_CRITIC = 1e-3
# Update network every X timesteps
UPDATE_EVERY = 32
# Learn from batch of experiences n_experiences times
N_EXPERIENCES = 16
```

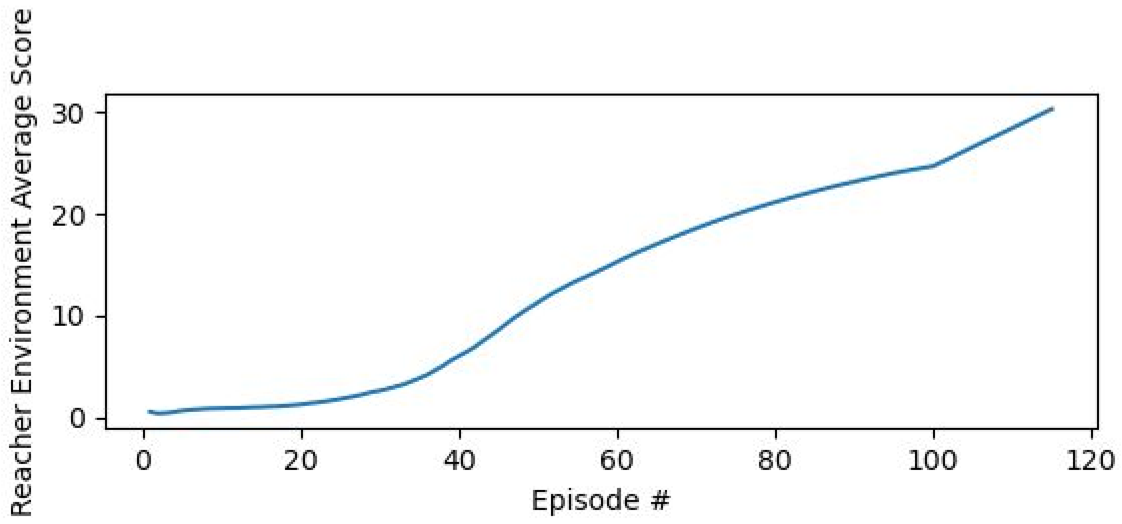
Plot of rewards *(averaged over 100 episodes)*



```
Episode 10      Average Score: 0.86
Episode 20      Average Score: 1.23
Episode 30      Average Score: 2.59
Episode 40      Average Score: 5.99
Episode 50      Average Score: 11.27
Episode 60      Average Score: 15.26
Episode 70      Average Score: 18.57
Episode 80      Average Score: 21.15
Episode 90      Average Score: 23.17
Episode 100     Average Score: 24.70
Episode 110     Average Score: 28.44

Environment solved in 115 episodes!     Average Score: 30.32
```

We reached our goal in about 120 episodes.



Future ideas

In order to improve our model, it would be nice to perform a grid search to find the best hyperparameters. It would be nice to optimize the critic and the actor separately (applying value-based and policy-based techniques respectively).

It would be nice to implement other algorithms reviewed in the classroom like A2C, A3C or PPO. Indeed, they can be included in the same project parametrizing the options in order to execute one or another using the command line. A benchmark comparing the result would be very useful.

I decided to remove the exploration noise in order to mimic the contents reviewed in the classroom but I will add it to check the performance improvements.

DDPG is using a Replay Buffer, but this can be improved by prioritizing it. This could be another improvement that can have a great impact in our algorithm.

Another idea would be to replace the information provided by Unity (observation space) and learn from the raw pixel instead. That would imply the use of a Convolutional Neural Network to apply filters and find patterns directly from the images.

Finally, I would like to adapt this project to solve the Crawler environment.