# Project 2: Continuous Control

Soroosh Rezazadeh

This report presents a technical description of the method used to solve the continuous control problem where a number of agents, double-jointed arms, are to reach specified goals.

## Environment:

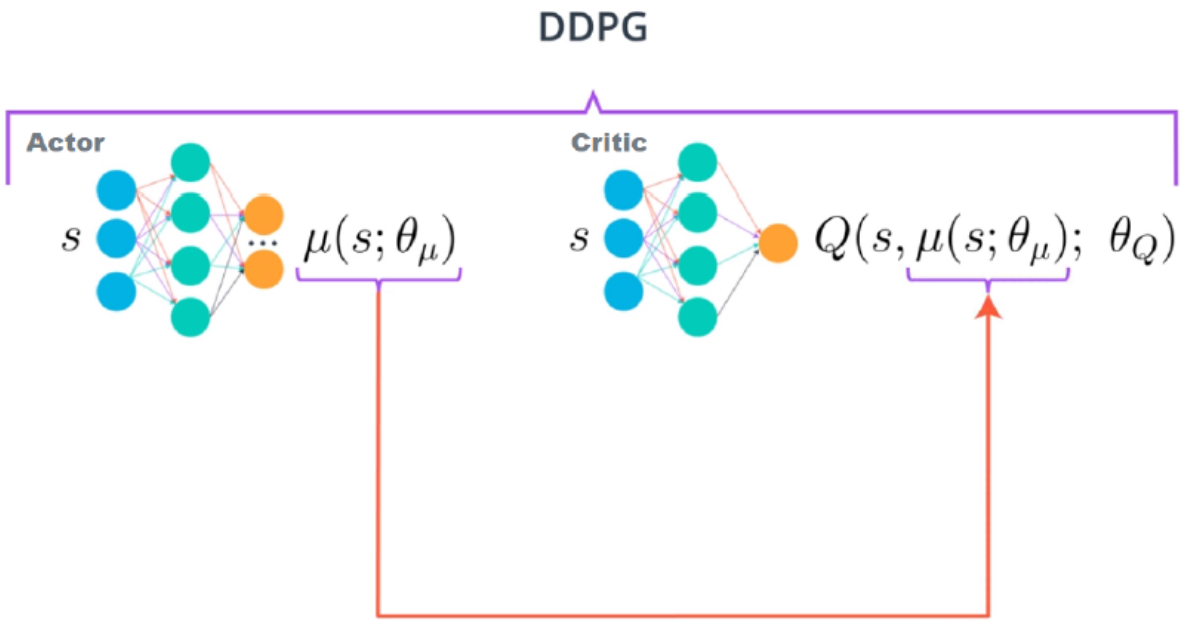
In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.
The observation space consists 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.
This task is episodic where each episode has max 2000 time steps.

```
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
Number of agents: 20
Size of each action: 4
There are 20 agents. Each observes a state with length: 33
```

## Learning Algorithm:

We use DDPG (Deep Deterministic Policy Gradient) algorithms to solve this project. DDPG is an actor-critic algorithm that extends DQN to work in continuous spaces. DDPG uses two deep neural networks, one as actor and the other as critic. The actor is used to approximate the optimal policy deterministically. That is, we want to output the best believed action for any given state.

The critic learns to evaluate the optimal action-value function by using the actor's best believed action.




The DDPG algorithm uses 4 neural networks: *actor_target*, *actor_local*, *critic_target* and *critic_local*. Classes *Actor* and *Critic* are provided by **model.py**

```
actor_target(state) -> action
critic_target(state, action) -> Q-value
actor_local(states) -> actions_pred
-critic_local(states, actions_pred) -> actor_loss
```

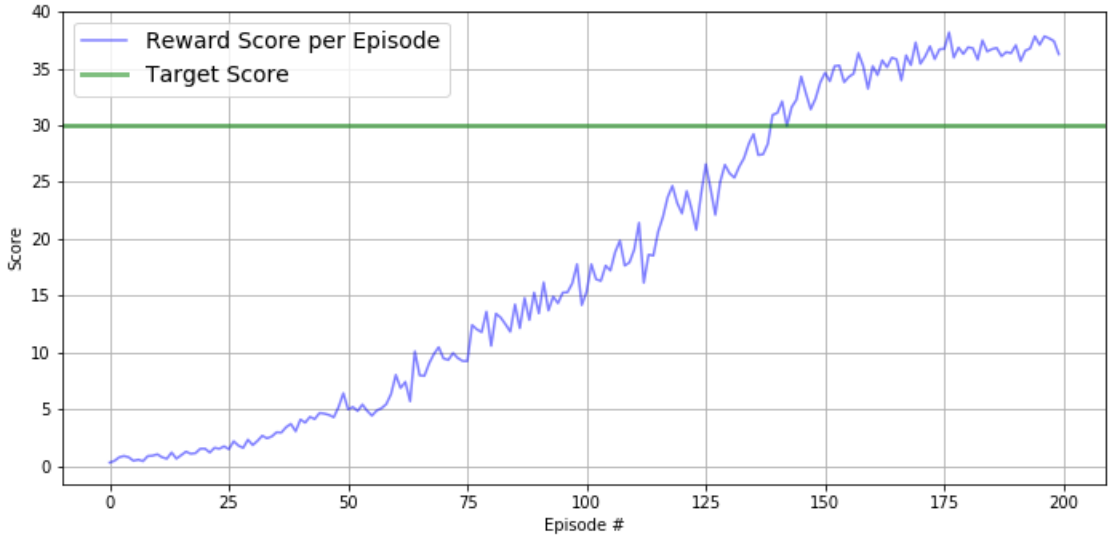
## Hyperparameters:

We used the following hyperparameters in our network:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 256        # 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
EPSILON = 1.0           # epsilon noise parameter
EPSILON_DECAY = 1e-6    # decay parameter of epsilon
LEARNING_PERIOD = 20    # learning frequency
UPDATE_FACTOR   = 10    # how much to learn
```

## Rewards Plot:

The desired average reward was achieved in 200 episodes with training the agent on the provided Udacity GPU. The plot below shows the rewards per episode and the target.




## Future Works:

- To improve the performance, we can explore algorithms such as PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.
- Investigate different hyperparameter values such as *BATCH_SIZE*, *LR_ACTOR*, *LR_CRITIC*, *LEARNING_PERIOD*, and *UPDATE_FACTOR*.
- Applying batch normalization to accelerate network training.