# 1 Environment details

## 1.1 State and Action spaces

The state space space has 33 dimensions corresponding to the position, rotation, velocity, and angular velocities of the robotic arm. There are two sections of the arm: analogous to those connecting the shoulder and elbow (i.e., the humerus), and the elbow to the wrist (i.e., the forearm) on a human body.

Each action is a vector with four numbers, corresponding to the torque applied to the two joints (shoulder and elbow). Every element in the action vector must be a number between -1 and 1, making the action space continuous.

For this project, I chose to use the environment with multiple agents (20) and use a single model to control them.

# 2 Learning algorithm

Since the action space is continuous and we have multiple agents collecting experiences concurrently, I decided to use a policy-based learning algorithm in this project.

## 2.1 Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm introduced by Lillicrap et al. (`https://arxiv.org/pdf/1509.02971.pdf`) is a model-free, off-policy actor-critic algorithm that uses deep function approximators that can learn policies in high-dimensional, continuous action spaces. They highlight that DDPG can be viewed as an extension of Deep Q-learning to continuous tasks.

I started with the based DDPG implementation given in the Udacity repository and improved upon that.

## 2.2 Actor-Critic Methods

Actor-critic methods leverage the strengths of both policy-based and value-based methods. Using a policy-based approach, the **Actor** learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Meanwhile, employing a value-based approach, the **Critic** learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs. Actor-critic methods combine these two approaches in order to accelerate the learning process. Actor-critic agents are also more stable than value-based agents, while requiring fewer training samples than policy-based agents. We use two networks, for the Actor and Critic (local and target networks) to improve stability.

## 2.3 Exploration vs Exploitation

In DQNs we use a $\epsilon$ greedy approach to address the exploration vs exploitation dilemma. However, this approach will not work for controlling a robotic arm as the actions are no longer a discrete set of simple directions (e.g., up, down, left, right). The actions driving the movement of the arm are forces with different magnitudes and directions. If we base our exploration mechanism on random uniform sampling, the direction actions would have a mean of zero, thereby canceling each other out. This can cause the system to oscillate without making much progress.

Therefore, we use the Ornstein-Uhlenbeck process, introduced by Lillicrap et al. The Ornstein-Uhlenbeck process adds a certain amount of noise to the action values at each time step. This noise is correlated to previous noise, and therefore tends to stay in the same direction for a longer duration without canceling itself out. This allows the arm to maintain velocity and explore the action space with more continuity.

The Ornstein-Uhlenbeck process has three hyperparameters that determine the noise characteristics and magnitude:

- mu: the long-running mean
- theta: the speed of mean reversion
- sigma: the volatility parameter

Of these, I only tuned sigma. After running a few experiments, I reduced sigma from 0.3 to 0.2. The reduced noise volatility seemed to help the model converge faster. There is also an epsilon ($\epsilon$) parameter used to decay the noise level over time. This decay mechanism ensures that more noise is introduced earlier in the training process (i.e., higher exploration), and the noise decreases over time as the agent gains more experience (i.e., higher exploitation). The starting value for epsilon and its decay rate are two hyperparameters that were tuned during experimentation.

The final noise parameters were set as follows:

```
OU_SIGMA = 0.2  # Ornstein-Uhlenbeck noise parameter
OU_THETA = 0.15  # Ornstein-Uhlenbeck noise parameter
EPSILON = 1.0  # explore->exploit noise process added to act step
EPSILON_DECAY = 1e-6  # decay rate for noise process
```

### 2.4 Learning Interval

Performing the learning step at each time step is computationally expensive. Therefore, the learning interval was set to 20 time steps. At every 20th time step, the algorithm randomly samples experiences from the replay buffer and updates the weights 10 times.

### 2.5 Gradient Clipping

We implement gradient clipping using the torch *clip_grad_norm* function. The function was set to "clip" the norm of the gradients at 1, therefore placing an upper limit on the size of the parameter updates, and preventing them from growing exponentially. This implementation, along with batch normalization, made the learning process more stable.

### 2.6 Batch Normalization

Running computations on large input values and model parameters can inhibit learning. Batch normalization addresses this problem by scaling the features to be within the same range throughout the model and across different environments and units. In additional to normalizing each dimension to have unit mean and variance, the range of values is often much smaller, typically between 0 and 1. Batch normalization was used on the outputs of the first fully-connected layers of both the actor and critic models.

### 2.7 Experience Replay

The experience gained by each agent while acting in the environment is saved in a memory buffer, and a small batch of observations from this list is randomly selected and used as the input to train the weights of the DNN (i.e., Experience Replay). In this project, we use one central replay buffer utilized by all 20 agents, therefore allowing agents to learn from each others' experiences.

The replay buffer contains a collection of experience tuples with the state, action, reward, and next state "(s, a, r, s')". Each agent samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive algorithm could otherwise become biased by correlations between sequential experience tuples.

## 3 Results

Figure 1 shows the average score per episode, as well as the rolling average reward for over the last 100 steps for the best DDPG model. The agent can solve the environment (i.e., get an average score of +30 over 100 consecutive episodes) in 112 episodes.

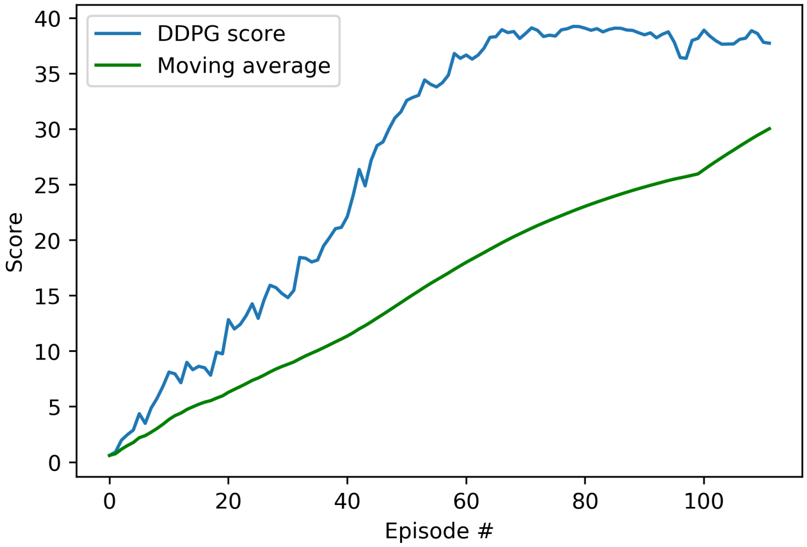

Figure 1: Average score per episode and the moving average.

## 4 Future Work

A more exhaustive search for hyper-parameters could result in faster convergence of the DDPG model. It would also be interesting to see how different network architectures affect the agent's behavior. The use of other algorithms other than DDPG may increase the accuracy of the agent. Although there is room for improvement, the current implementation solved the environment well above the required threshold.