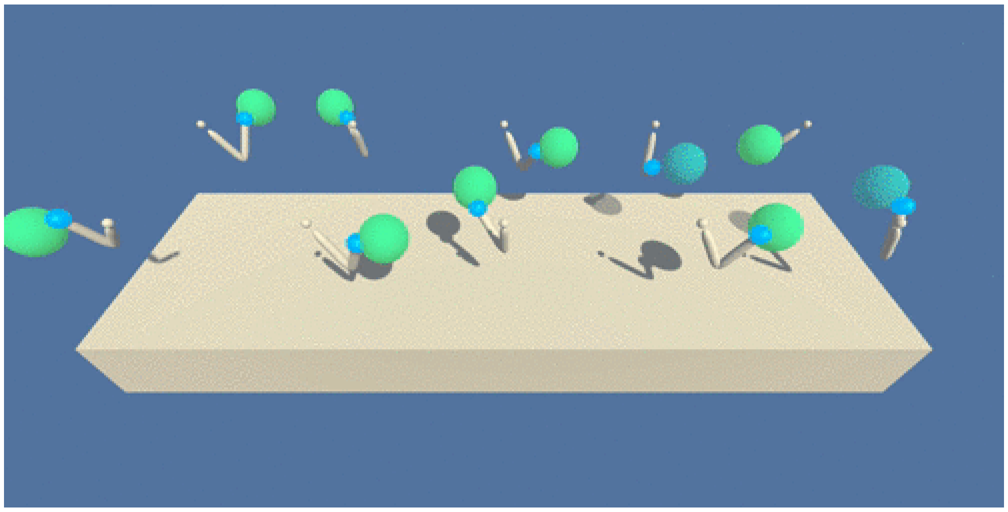
Project Report
DRLND Project 2: **Continuous Control**

Problem Description

For this project, we work with the Reacher environment..



The problem is framed as a rich game environment on the Unity platform. The environment has been designed by the Udacity Deep RL Nanodegree (DRLND) program team, and is provided as part of the starting materials. Interacting with the environment is through the use of a Python API exposed by the Unity Machine Learning Agents (ML-Agents) Toolkit.

In this environment, a double-jointed arm can move to target locations. A reward is provided for each step and is related to the proximity of the agent's hand to the goal location. The reward is in the range 0.00 - 0.04. The goal of the agent is to maintain its position at the target location for as many time steps as possible. Each game (episode) ends at 1,000 steps.

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.

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

The barrier for solving the multi-agent version of the environment takes into account the presence of many agents. In particular, the 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).

The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

Learning Algorithm

To train an agent to navigate and achieve high rewards in the Reacher environment, we resort to Reinforcement Learning using an Actor-Critic Method. The Deep Deterministic Policy Gradient (DDPG) solution used by Lillicrap et al. [1] is well-suited for this task, given the Continuous nature of the action space used by the environment.

The observation space provides us with direct measurements of variables corresponding to position, rotation, velocity and angular-velocities of the arm. Effectively, this does the work of the convolutional layers used for pixel-based input in the experiments used in the DDPG paper, and we can treat the 33 dimensions of observations to directly feed the dense layers in our Neural Network.

We can, therefore, use a low-dimensional Neural Network with just fully-connected layers.

Model (Actor and Critic)

- The neural network uses the rectified non-linearity for all layers.
- The final output layer of the actor is a *tanh* layer, to bound the actions.
- The networks have 2 hidden layers with 256 and 128 units respectively.
- Actions are not included until the 2nd hidden layer of *Q*.
- The final layer weights and biases of both the actor and critic are initialized from a uniform distribution [-0.003, 0.003]. This is to ensure the initial outputs for the policy and value estimates are near zero.
- We train with minibatch sizes of 128.
- We use a replay buffer size of 10^6.

The agent selects and executes actions provided by the policy gradient based actor. To aid the task of converging, and to reduce training time, we resort to two key features:

- **Experience Replay**: a rolling history of past data via a re-play pool. Using the replay pool the behavior distribution is averaged out over many of its previous states smoothing out learning and avoiding oscillations. This has the advantage that each step update is potentially used in many weight updates.
- **Ornstein-Uhlenbeck Noise**: For the exploration noise process we use temporally correlated noise in order to explore well in the environment.

The full algorithm for training the DDPG-network is as follows:

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    end for
end for
```

Training Results

The implementation for solving this project used Python 3 and PyTorch. The code is shared on GitHub: https://github.com/aquanta-aagyaa/drlnd-p2-continuous-control

The important files are:

1. Continuous_Control.ipynb - Jupyter notebook to walk through the training process
2. ddpg_agent.py - The DDPG agent, replay buffer and OUNoise implementation
3. model.py - implementation of the Actor and Critic (DNN) classes using PyTorch

Hyperparameters

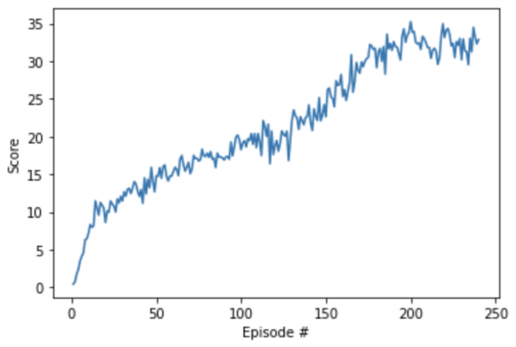
The following hyperparameters allow the model to achieve the desired goal, of a 20-agent average score of +30.0 over the last 100 episodes.

● 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-4	# learning rate of the critic
● WEIGHT_DECAY: 0	# L2 weight decay
● UPDATE_EVERY: 20	# how often to trigger updates to the network
● UPDATE_TIMES: 10	# how often to update the network, when triggered

Successful training session:

```
Episode 10      Average Score (over agents): 7.18       Average Score (over last 100 episodes): 3.73
Episode 20      Average Score (over agents): 8.60       Average Score (over last 100 episodes): 6.733
Episode 30      Average Score (over agents): 11.44      Average Score (over last 100 episodes): 8.14
Episode 40      Average Score (over agents): 12.05      Average Score (over last 100 episodes): 9.33
Episode 50      Average Score (over agents): 14.78      Average Score (over last 100 episodes): 10.18
Episode 60      Average Score (over agents): 15.36      Average Score (over last 100 episodes): 11.00
Episode 70      Average Score (over agents): 15.07      Average Score (over last 100 episodes): 11.72
Episode 80      Average Score (over agents): 17.78      Average Score (over last 100 episodes): 12.40
Episode 90      Average Score (over agents): 16.88      Average Score (over last 100 episodes): 12.93
Episode 100     Average Score (over agents): 18.25      Average Score (over last 100 episodes): 13.49
Episode 110     Average Score (over agents): 20.38      Average Score (over last 100 episodes): 15.06
Episode 120     Average Score (over agents): 18.78      Average Score (over last 100 episodes): 16.04
Episode 130     Average Score (over agents): 22.15      Average Score (over last 100 episodes): 16.92
Episode 140     Average Score (over agents): 24.18      Average Score (over last 100 episodes): 17.88
Episode 150     Average Score (over agents): 22.62      Average Score (over last 100 episodes): 18.80
Episode 160     Average Score (over agents): 25.30      Average Score (over last 100 episodes): 19.90
Episode 170     Average Score (over agents): 28.40      Average Score (over last 100 episodes): 21.05
Episode 180     Average Score (over agents): 29.13      Average Score (over last 100 episodes): 22.40
Episode 190     Average Score (over agents): 32.58      Average Score (over last 100 episodes): 23.83
Episode 200     Average Score (over agents): 35.23      Average Score (over last 100 episodes): 25.26
Episode 210     Average Score (over agents): 31.81      Average Score (over last 100 episodes): 26.58
Episode 220     Average Score (over agents): 33.14      Average Score (over last 100 episodes): 27.80
Episode 230     Average Score (over agents): 30.18      Average Score (over last 100 episodes): 29.08
Episode 240     Average Score (over agents): 32.85      Average Score (over last 100 episodes): 30.05

Environment solved in 140 episodes!     Average Score: 30.05
```



The model weights of the successful agent are saved to the folder:
train_0a6cfeee_solved_2021-04-29_2251

Ideas for Future Work

Some ideas we can pursue to improve the performance of our agent:

Prioritized Experience Replay
The current implementation samples uniformly from the replay buffer. We might want to sample more frequently those transitions from which there is much to learn.

Parameter Noise
The current implementation adds Ornstein-Uhlenbeck noise to the action space. We might want to add noise to the parameters of the neural network policy to improve performance.

Parallel Processing
We might want to try solving the learning problem using algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

References

1. Lillicrap, Timothy, et al. “Continuous control with deep reinforcement learning”. https://arxiv.org/abs/1509.02971