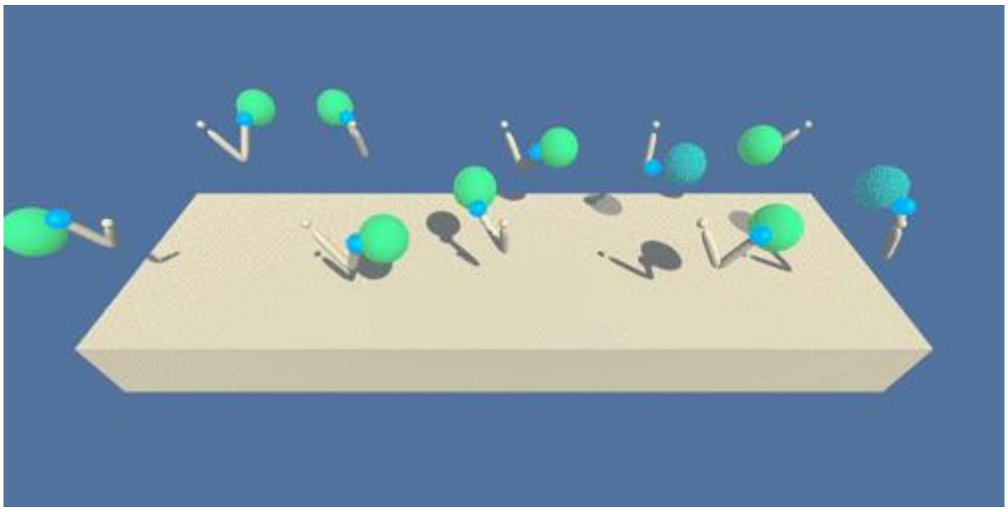
Deep Reinforcement Learning Nanodegree

Project 2: Continuous Control

Pradeep Dayanandam

Introduction:

In this project we will work with the Reacher environment.



Unity ML-Agents Reacher 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.

Distributed Training:

For this project, we are provided with two separate versions of the Unity environment:

- The first version contains a single agent.
- The second version contains 20 identical agents, each with its own copy of the environment.

The second version is useful for algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

Solving the Environment:

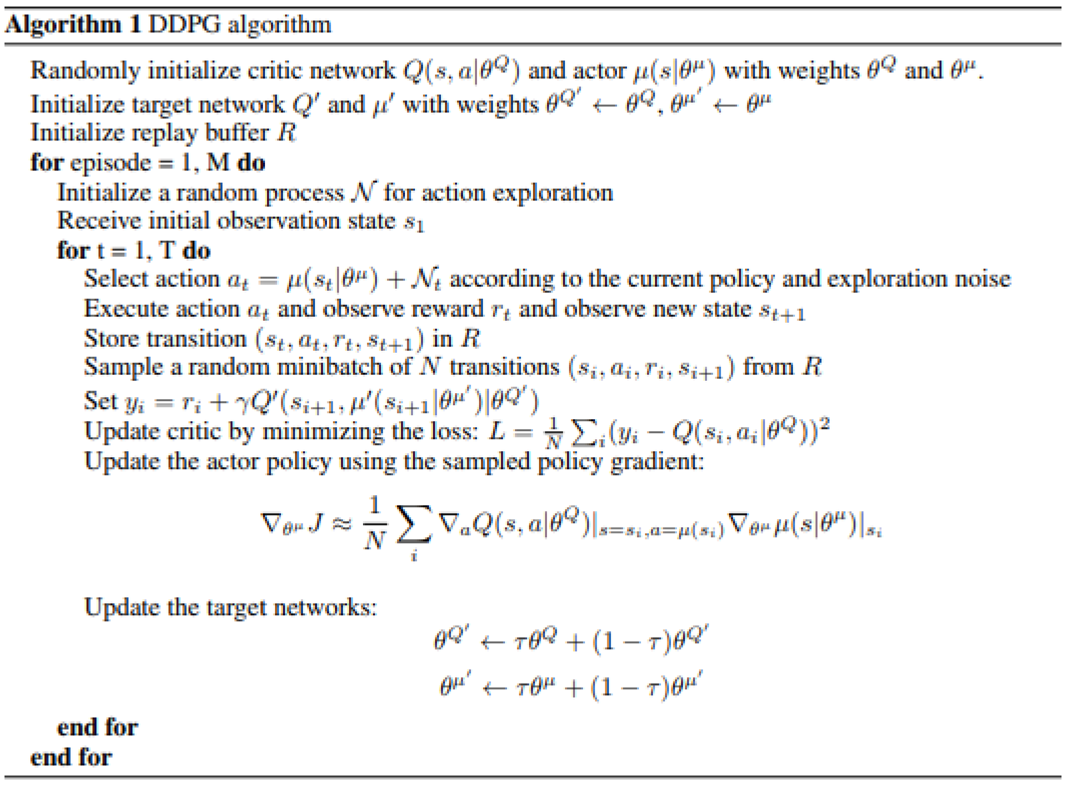
For solving this environment, we will use the first version where we use only one agent to solve the environment

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

Algorithm Used:

The Algorithm used to solve this problem is **Deep Deterministic Policy Gradient** (DDPG). DDPG was proposed in Continuous control with deep reinforcement learning (Lillicrap et al, 2015). DDPG is an off-policy algorithm and can only be used only for continuous action spaces. It simultaneously learns a function and a policy. It's an actor critic algorithm and uses two neural networks. The algorithm also applies experience replay and target networks to stabilize the training. To boost exploration, the authors of the DDPG paper have used Ornstein-Uhlenbeck Process to add noise to the action output.

Pseudocode:



```
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
```

Taken from Continuous control with deep reinforcement learning (Lillicrap et al, 2015)

Model Architecture:

The Actor network consist of three fully connected layer with batch normalization applied at the first layer. The network maps states to actions. It uses ReLU as activation function except the last layer where it uses tanh.

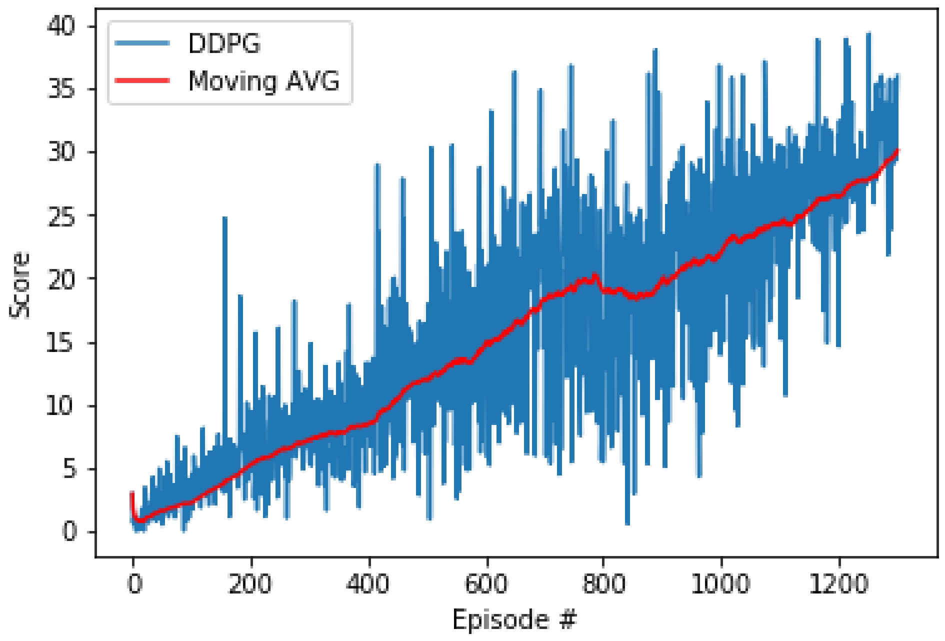
The critic network also consists of three fully connected layer with batch normalization applied at the first layer. The network maps (state, action) pairs to Q-values. It uses ReLU as activation function in the first two layers and no activation function for the last layer.

Hyperparameters

- fc1_units=400 # Number of nodes in the first hidden layer
- fc2_units=300 # Number of nodes in the second hidden layer
- 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-3 # learning rate of the critic
- WEIGHT_DECAY = 0 # L2 weight decay
- LEARN_EVERY = 20 # learning timestep interval
- LEARN_NUM = 10 # number of learning passes
- GRAD_CLIPPING = 1.0 # gradient clipping
- OU_SIGMA = 0.2 # OU noise parameter
- OU_THETA = 0.15 # OU noise parameter
- EPSILON = 1.0 # for epsilon in the noise process (act step)
- EPSILON_DECAY = 1e-6 3 epsilon decay rate
- num_episodes=2500 # maximum number of training episodes
- max_t=1000 # maximum number of timesteps per episode

Results:

The environment is considered to be solved when the average reward is at least 30 for the last 100 episodes. The below plot shows the moving average against number of episodes.



Environment solved in 1301 episodes!!!

Average score (when the env was first solved): 30.09

Future ideas to improve the agent's performance:

More experiments can be done to increase the performance of agent

- DDPG can be improved by using prioritized experience replay.
- Fine-tuning of hyperparameters can also lead to better results and faster training time
- The second version of using 20 agents can also be implement which can result in quicker convergence.