Introduction

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, I use the Unity environment as below:

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

This 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.

Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradient (DDPG) is a special class of Actor-Critic Methods. The actor produces a deterministic policy instead of the usual stochastic policy and the critic evaluates the deterministic policy. The critic is updated using the TD-error and the actor is updated to maximize the expected Q-value.

In total, four deep neural networks are used, including: local and target networks for both actor and the critic. Now the actor here is used to approximate the optimal policy deterministically. That means we want to always output the best-believed action for any given state. This is unlike a stochastic policy in which we want to learn a probability distribution over the action. In DDPG we want the best-believed action every time we query the actor-network. That is a deterministic policy. The actor is basically learning the argmax of Q(s,a) which is the best action. The critic learns to evaluate the optimal action-value function by using the actors' best-believed action.

In DDPG, target networks are updated using a soft update strategy. A soft update strategy consists of slowly blending in your regular network weights with your target network weights. So, every timestep you make your target network be 99.99 percent of your target network weights and only 0.01 percent of your regular network weights. You are slowly mixing your

regular network weights into your target network weights. Recall, the regular network is the most up to date network because it is the one we are training while the target network is the one we use for prediction to stabilize strain. In practice, you will get faster convergence by using the update strategy.

In Reinforcement learning for discrete action spaces, exploration is done via probabilistically selecting a random action such as epsilon-greedy or Boltzmann exploration. For continuous action spaces, exploration is done via adding noise to the action itself.

The agent interacts with the environment and learns from the sequence of state(S), actions(A) and rewards(R) and next state(S') which is stored in the form of an experienced tuple. These sequence of experience tuples can be highly correlated, and the DDQN algorithm like the Q-learning algorithm that learns from each of these experienced tuples in sequential order and can be swayed by the effects of this correlation. This can lead to oscillation or divergence in the action values, which can be prevented by having a replay buffer, from which experience replay tuples are used to randomly sample from the buffer. The replay buffer contains a collection of experience tuples (SS, AA, RR, S'S ′). The tuples are gradually added to the buffer as we are interacting with the environment. The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Architecture

The network architecture is comprised of two fully connected hidden layers of 300 and 200 units each with ReLU activations, respectively. In order to help speed up learning and avoid getting stuck in a local minimum, batch normalization was introduced to each hidden layer. The hyperbolic tan activation was used on the output layer for the actor-network as it ensures that every entry in the action vector is a number between -1 and 1. Adam was used as an optimizer for both actor and critic networks.

```
---Actor---
Actor(
  (fc1): Linear(in_features=33, out_features=300, bias=True)
  (bn1): BatchNorm1d(300, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (fc2): Linear(in_features=300, out_features=200, bias=True)
  (fc3): Linear(in_features=200, out_features=4, bias=True)
)
---Critic---
Critic(
  (fcs1): Linear(in_features=33, out_features=300, bias=True)
  (bn1): BatchNorm1d(300, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (fc2): Linear(in_features=304, out_features=200, bias=True)
  (fc3): Linear(in_features=200, out_features=1, bias=True)
```

)

Hyperparameters

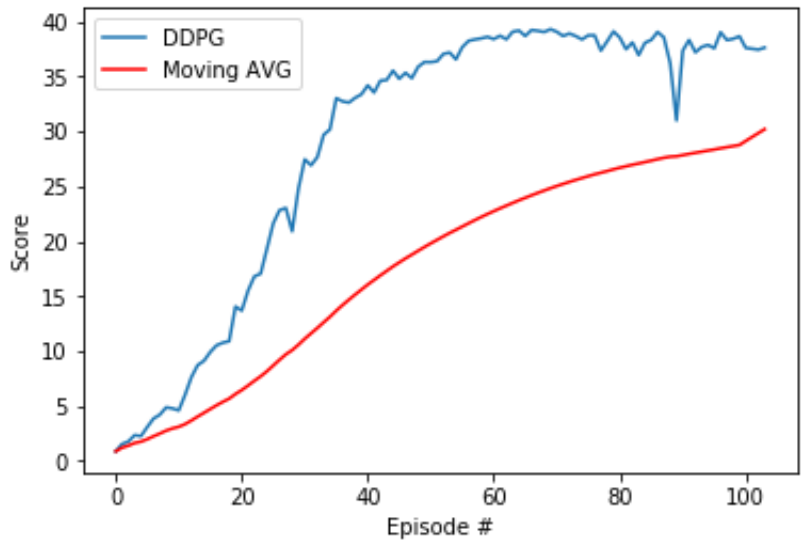
The various hyperparameters used are as follows:

```
# hyperparameters

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
MAX_T = 1000            # maximum number of timesteps per episode

LEARN_EVERY = 20        # learning timestep interval
LEARN_NUM   = 10        # number of learning passes
GRAD_CLIPPING = 1.0     # Gradient Clipping

# Ornstein-Uhlenbeck noise parameters
OU_SIGMA  = 0.2
OU_THETA  = 0.15
EPSILON = 1.0           # for epsilon in the noise process (act step)
EPSILON_DECAY = 1e-6
```



Plot of average scores (over all agents) and the average of the average scores each episode.

The environment is considered solved, when the average (over 100 episodes) of those **average scores** is at least +30. In the case of the plot above, the environment was solved at episode 35, since the average of the **average scores** after episodes 100 (inclusive) was greater than +30.

Future works

Some ideas to improve the agent's performance in the future:

- Tune up hyperparameters like size of networks, grad_clipping to improve computation time and scores
- Explore prioritized replay to improve performance
- Try other algorithms like TRPO, PPO, A3C, A2C which have been discussed in the course
- General optimization techniques like cyclical learning rates and warm restarts may be useful