Report: Project2

Deep Reinforced Learning Nanodegree:
Continuous Control with DDPG

Introduction

This project solves for actuation of a 2 degree of freedom robotic arm to have the end actuator reach the goal position. Implemented using DDPG (Deep Deterministic Policy Gradient) algorithm. This is a model free learning method, is pretty cool and easily portable across different applications.

Environment

Reacher environment from Unity Machine Learning Agent Toolkit is used for this project. 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 is a number between -1 and 1.

State/ Observation size: 33

Action space size : 4

Goal: The environment is considered solved, when the average (over 100 episodes) of those average scores (all 20 agents) is at least +30.

Learning Algorithm

DDPG - Deep Deterministic Policy Gradient

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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

DDPG is the an Actor-Critic algorithm that simultaneously learns a policy and an action-value function Q(s,a). The actor takes a deterministic action following a policy and critic critics the actors action based on its action-value function.

Noise is added to the action intentionally to match up for a continuous action space use case through the output is deterministic.

Hyperparameters

GAMMA	0.99	Discount factor
TAU	1e-3	Soft update of target parameters
LR_ACTOR	1e-3	Learning rate of the actor
LR_CRITIC	1e-3	Learning rate of the actor
WEIGHT_DECAY	0.0000	L2 weight decay
BATCH_SIZE	512	Batch size
BUFFER_SIZE	int(1e6)	Replay buffer size
learn_every	20	How often to update target network
num_learn	10	Number of updates at each step
OUNoise.theta	0.15	Ornstein-Uhlenbeck Noise parameter
OUNoise.mu	0.2	Ornstein-Uhlenbeck Noise parameter

Network Structure

Actor : NN

a_FC1 : state_size - > 200

a_FC1 : ReLU (Batch Normalization (a_FC1))

a_FC2 : ReLU (200 (a_FC1) -> 100)

a_FC3 : tanh (100 (a_FC2) -> action_size)

Critic: NN

c_FC1 : state_size - > 200

c_FC1 : ReLU (Batch Normalization (c_FC1))

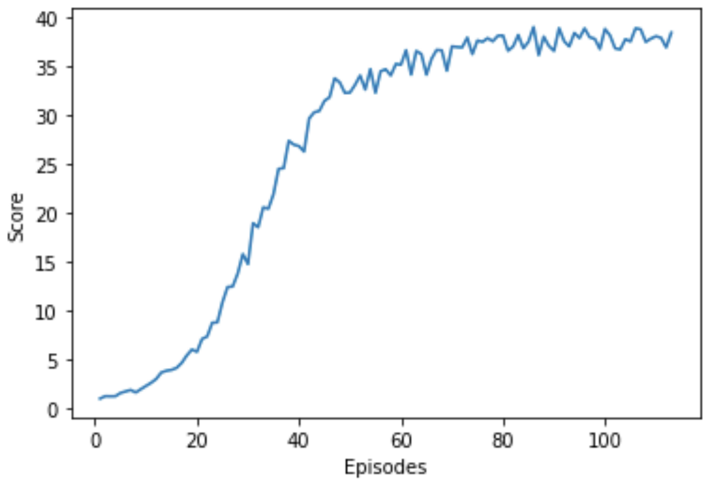
c_FC2 : ReLU (200 ([c_FC1 <concat> a_FC3]) -> 100)

c_FC3 : 100 (c_FC2) -> 1

Result :

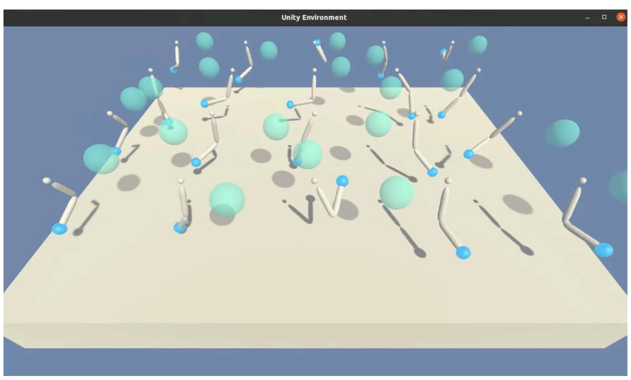
Goal was achieved after 113 episodes with total average score of 30.05

Score Vs Episode Plot

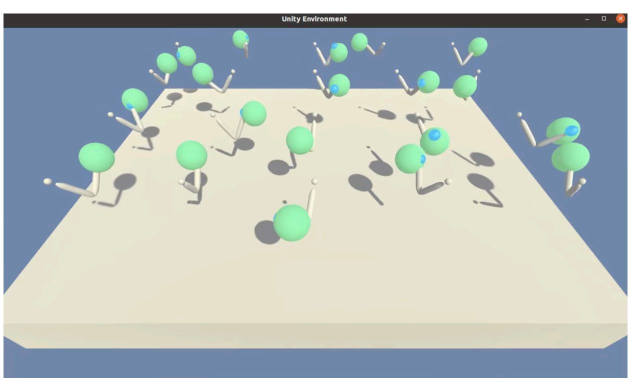


Result GIF:

Untrained: https://gph.is/g/Z866Pbb



Trained: https://gph.is/g/aRVV5qw



Future Work :

- Try out the crawler environment
- Use prioritized experience replay
- Use tensorboard to compare results form multiple hyper parameter choices