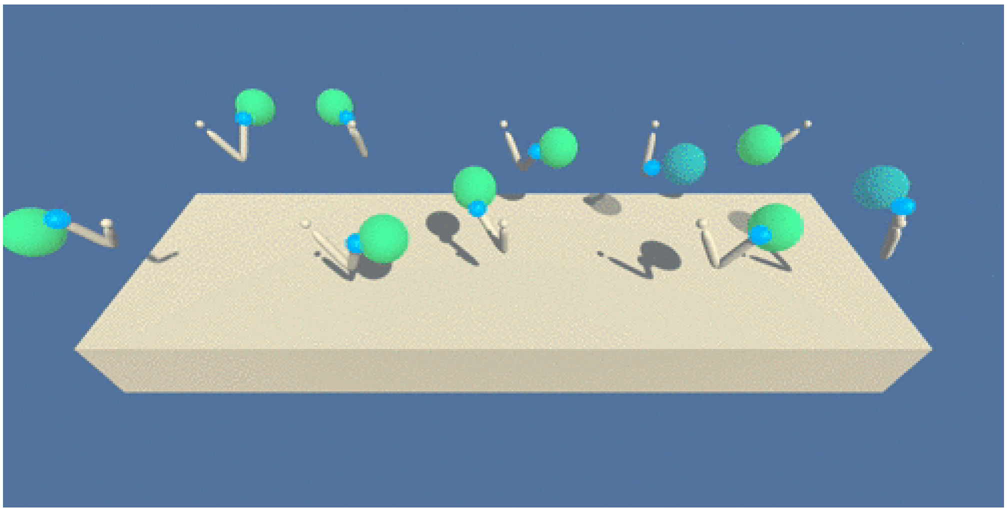
DDPG (Actor/Critic) Reinforcement Learning using PyTorch and Unity ML - Reacher Environment

Overview

This project was developed as part of Udacity Deep Reinforcement Learning Nanodegree course. This project solves Reacher environment by training the agent to control robotic arms by positioning them in moving target locations using Deep Deterministic Policy Gradient (DDPG) algorithm. The environment is based on Unity ML agents.

Introduction

For this project, the unity ML Reacher environment is used . Agent is trained to learn to control l robotic arms to position them in moving target locations



The environment contains 20 identical agents, each with its own copy of the environment. The environment takes into account the presence of many agents. In particular, agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically,

- After each episode, the rewards that each agent received (without discounting), is used 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.

Rewards:

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.

Environment:

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.

Getting Started

Installation and Dependencies

To set up your python environment to run the code in this repository, follow the instructions below.

1. Create (and activate) a new environment with Python 3.6.
    - **Linux** or **Mac**:

    ```
    conda create --name drlnd python=3.6
    source activate drlnd
    ```

    - **Windows**:

    ```
    conda create --name drlnd python=3.6
    activate drlnd
    ```

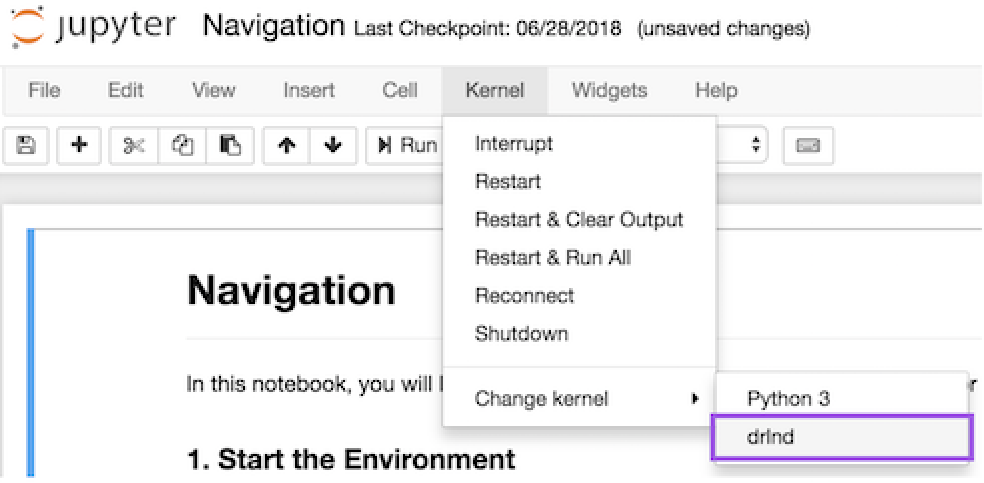
2. Follow the instructions in this repository to perform a minimal install of OpenAI gym.
    - Next, install the **classic control** environment group by following the instructions here.
    - Then, install the **box2d** environment group by following the instructions here.
3. Clone the repository (if you haven't already!), and navigate to the `python/` folder. Then, install several dependencies.

```
git clone https://github.com/udacity/deep-reinforcement-learning.git
cd deep-reinforcement-learning/python
pip install .
```

1. Create an IPython kernel for the `drlnd` environment.

```
python -m ipykernel install --user --name drlnd --display-name "drlnd"
```

1. Before running code in a notebook, change the kernel to match the `drlnd` environment by using the drop-down `Kernel` menu.



Unity Environment Setup:

Unity Environment is already built and made available as part of Deep Reinforcement Learning course at Udacity.

1. Download the environment from one of the links below. You need only select the environment that matches your operating system:
   - ***Version 1: One (1) Agent***
     - Linux: click here
     - Mac OSX: click here
     - Windows (32-bit): click here
     - Windows (64-bit): click here
   - ***Version 2: Twenty (20) Agents***
     - Linux: click here
     - Mac OSX: click here
     - Windows (32-bit): click here
     - Windows (64-bit): click here

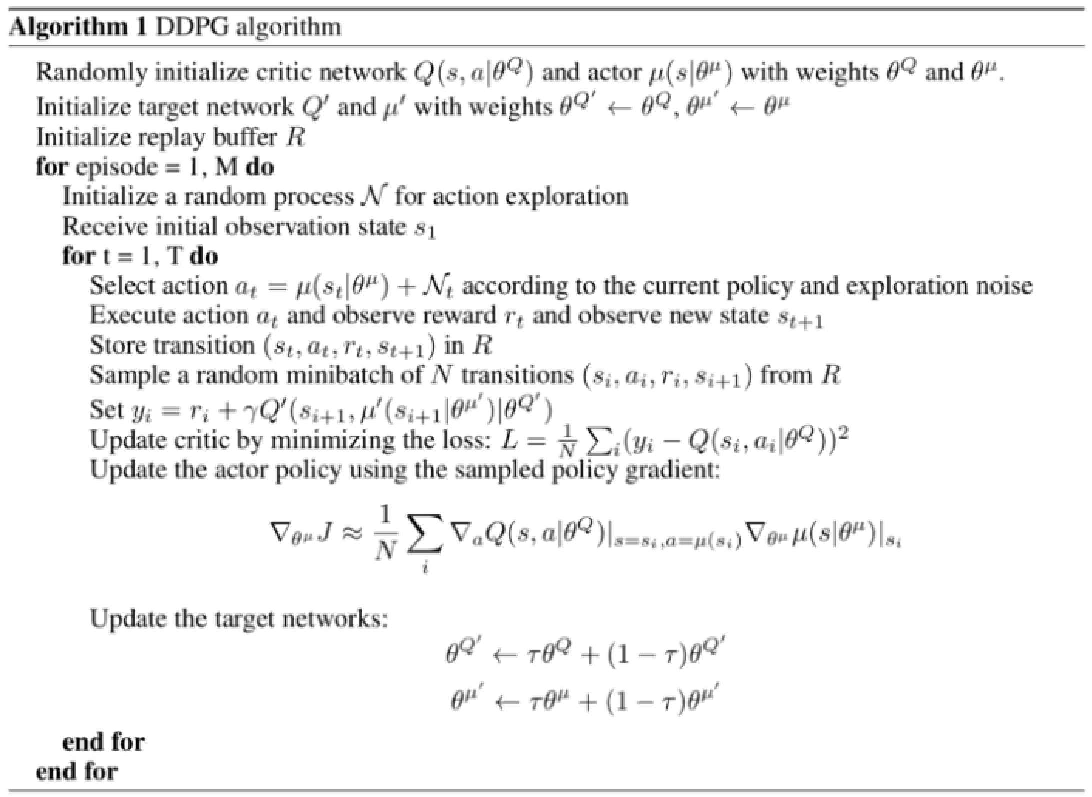
   (*For Windows users*) Check out this link if you need help with determining if your computer is running a 32-bit version or 64-bit version of the Windows operating system.

   (*For AWS*) If you'd like to train the agent on AWS (and have not enabled a virtual screen), then please use this link (version 1) or this link (version 2) to obtain the "headless" version of the environment. You will **not** be able to watch the agent without enabling a virtual screen, but you will be able to train the agent. (*To watch the agent, you should follow the instructions to enable a virtual screen, and then download the environment for the **Linux** operating system above.*)

2. Place the file in the DRLND GitHub repository, in the `p2_continuous-control/` folder, and unzip (or decompress) the file.

Deep Deterministic Policy Gradient Algorithm

DDPG is a model-free off-policy actor-critic algorithm that learns directly from observation spaces. DDPG employs Actor-Critic model, where Actor learns the policy and Critic learns the value function to evaluate the quality of the action chosen by the policy. while Deep Q-Network learns the Q-function using experience replay and works well in discrete space, DDPG algorithm extends it to continuous action spaces using Actor-Critic framework while learning policy.



```
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
```

Fig 3. DDPG Algorithm. (Image source: Lillicrap, et al., 2015)

Repository

The repository contains the below files:

- ddpg_continuous_agent.ipynb : Model for Actor and Critic along with agent learns using Experience Replay and OUNoise. Training the agent and testing the agent is implemented here.
- ddpg_continuous_agent_prioritized_replay.ipynb : Model for Actor and Critic along with agent learns using Prioritized Experience Replay and OUNoise. Training the agent and testing the agent is implemented here. The agent trained using DDPG with prioritized experience replay buffer learned faster.
- checkpoint_actor.pth : Learned model weights for Actor
- checkpoint_critic.pth : Learned model weights for Critic
- checkpoint_actor_preplay.pth : Learned model weights for Actor with Prioritized replay
- checkpoint_critic_preplay.pth : Learned model weights for Critic with Prioritized replay
- images directory: contains images used in documentation
- multiple_agents: Please copy Reacher environment click here to this location or modify the filepath in ddpg_continuous_agent.ipynb to point to the correct location.

Model Architecture:

Pendulum-v0 environment with Deep Deterministic Policy Gradients (DDPG) is used as reference to build the model. The model architecture that is used is:

Actor:
Input(state size of 32) → Dense Layer(256) → RELU → Dense Layer(128) → RELU → Dense Layer(action size of 4) → TANH

Critic:

Input(state size of 32) → Dense Layer(256) → RELU → Dense Layer(128) → RELU → Dense Layer(action size of 4)

Agent:

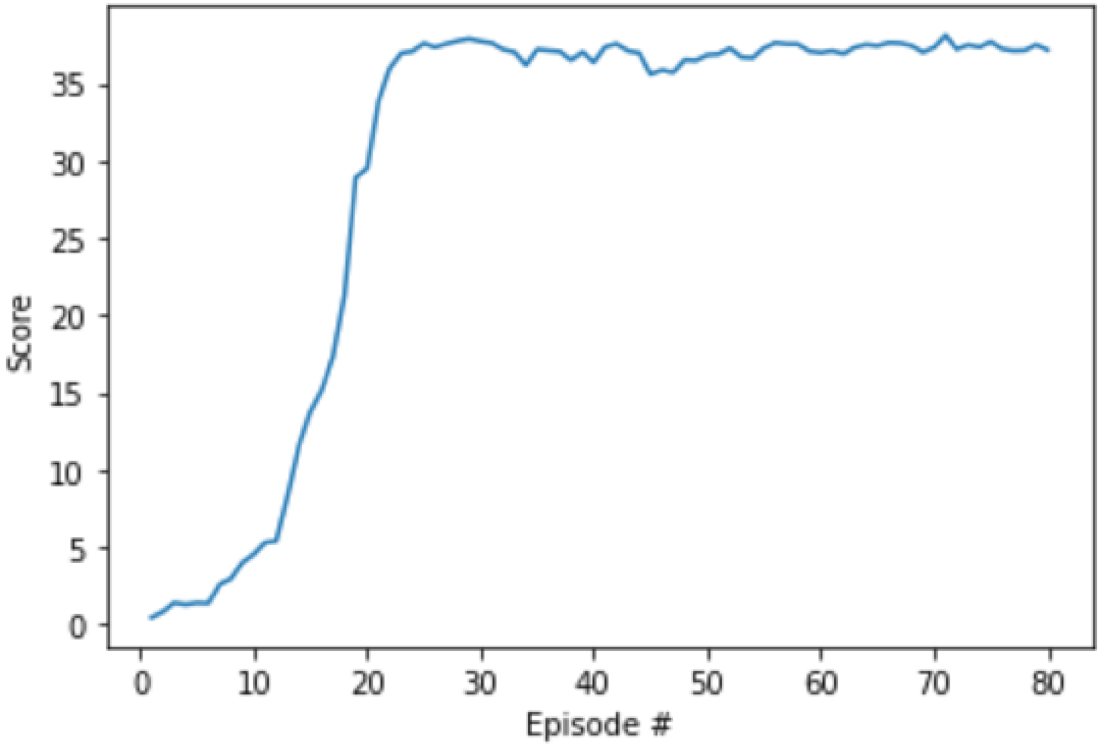
Actor Local and Critic Local networks are trained and updates the Actor Target and Critic Target networks using weighting factor Tau.

Please refer to Report.md for more details on the model and parameters used for tuning

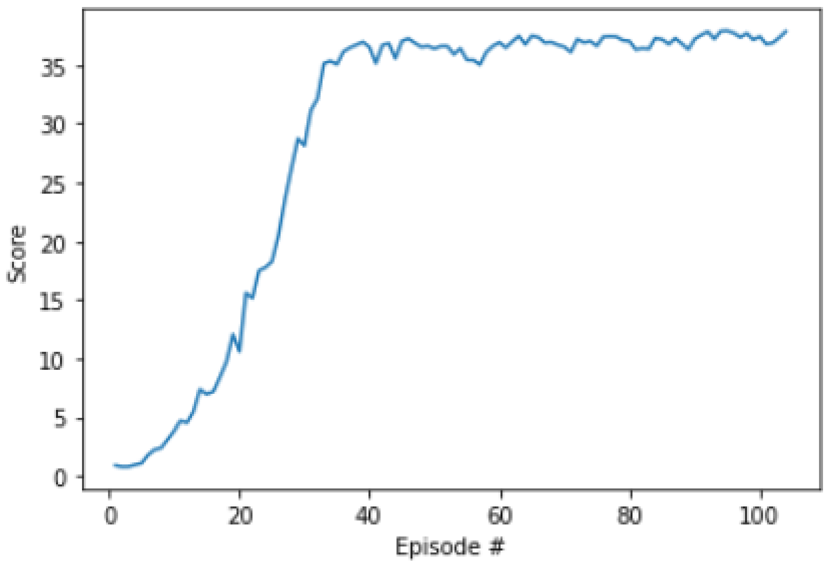
Results:

Results from DDPG training with prioritized experience replay and experience replay buffer are shared below. The agent trained using prioritized experience replay showed faster and efficient learning than experience replay. The agent with DDPG with prioritized experience replay learned the environment in 24 less episodes than the agent with DDPG and experience replay.

DDPG with Prioritized Experience Replay



DDPG with Experience Replay


Score
0
5
10
15
20
25
30
35
Episode #
0
20
40
60
80
100