Udacity Deep Reinforcement Learning Nanodegree

Project 2: Continuous Control

Introduction

This project repository contains Nishi Sood's work for the Udacity's Deep Reinforcement Learning Nanodegree **Project 2: Continuous Control.** For this project, I have trained an agent that could control a double-jointed arm towards specific locations.

Project's goal

- 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.
- The goal of the agent is to maintain its position at the target location for as many time steps as possible.
- The environment solved in this project is a variant of the Reacher Environment agent by Unity.
- 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.

Project's Environment

- The environment is based on Unity ML-agents. The project environment provided by Udacity is similar to the Reacher environment on the Unity ML-Agents GitHub page.
- **Set-up:** Double-jointed arm which can move to target locations.
- **Goal:** The agents must move its hand to the goal location, and keep it there.
- **Agents:** The environment contains 20 agents linked to a single Brain.
    - The provided Udacity agent versions are Single Agent or 20-Agents
- Agent Reward Function (independent):
    - +0.1 Each step agent's hand is in goal location.
- **Brains**: One Brain with the following observation/action space.
    - Vector Observation space: 26 variables corresponding to position, rotation, velocity, and angular velocities of the two arm Rigid bodies.

    - Vector Action space: (Continuous) Size of 4, corresponding to torque applicable to two joints.
    - Visual Observations: None.
- Reset Parameters: Two, corresponding to goal size, and goal movement speed.
- The task is episodic, with 1000 timesteps per episode. In order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.
- **Benchmark Mean Reward: 30**

Solving the Environment

Depending on the chosen environment for the implementation, there are 2 possibilities:

Option 1: Solve the First Version

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

Option 2: Solve the Second Version

The barrier for solving the second version of the environment is slightly different, to consider 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, the rewards that each agent received (without discounting) are added up , to get a score for each agent. This yields 20 (potentially different) scores. The average of these 20 scores is then used.

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.

In the implementation I have chosen to solve the Second version of the environment (Multi Agent) using the off-policy DDPG algorithm.

Project Environment Setup/ Configuring Dependencies:

Step 1: Clone the DRLND Repository and installing dependencies

1. Set Up for the python environment to run the code in this repository

Environment

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

2. Install of OpenAI gym packages

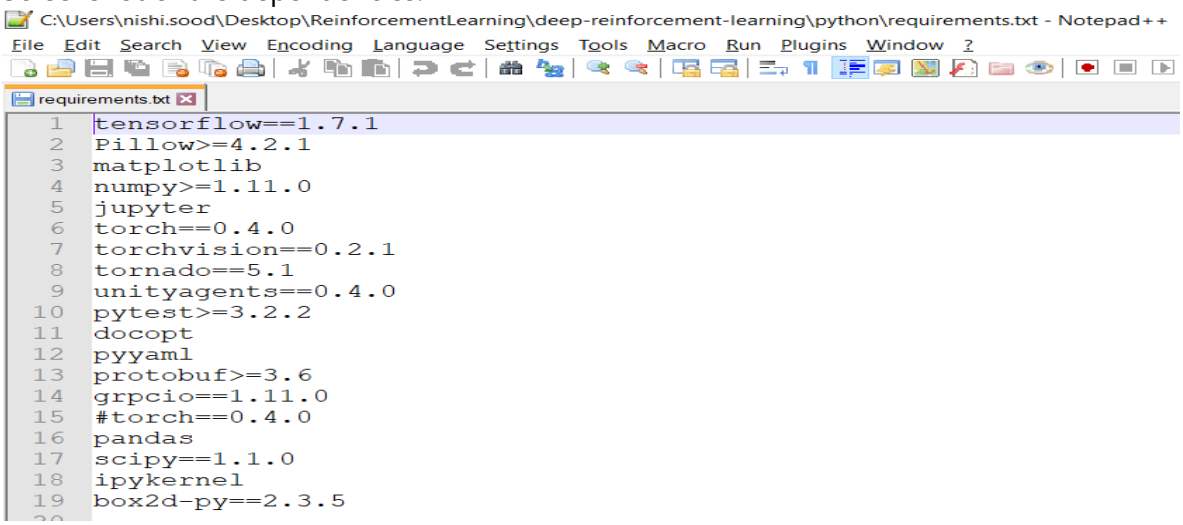
OpenAI gym

git clone https://github.com/openai/gym.git
cd gym
pip install -e .
pip install -e .[classic_control]
conda install swig
pip install -e .[box2d]

3. Other Dependencies

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

Screenshot of the dependencies:



NOTE: List of the installed dependencies can be found in “requirements.txt” attached with the project.

Trouble shooting:

If you get the error for installing torch library:

On Windows:

1. Open deep-reinforcement-learning\python\requirements.txt
2. Remove the line torch==0.4.0

On Anaconda:

3. conda install pytorch=0.4.0 -c pytorch
4. cd deep-reinforcement-learning/python

pip install .
conda install pytorch=0.4.0 -c pytorch

4. Create an IPython kernel for the drlnd environment

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

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

Step 2: Download the Unity Environment

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

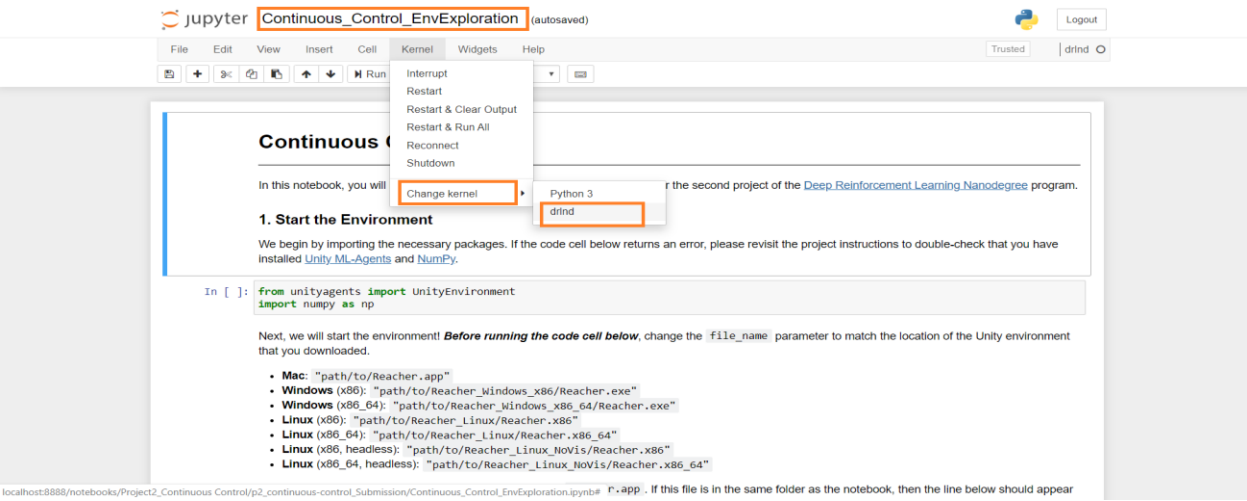
Version 2: Twenty (20) Agents

- Linux: click here
- Mac OSX: click here
- Windows (32-bit): click here
- Windows (64-bit): click here

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

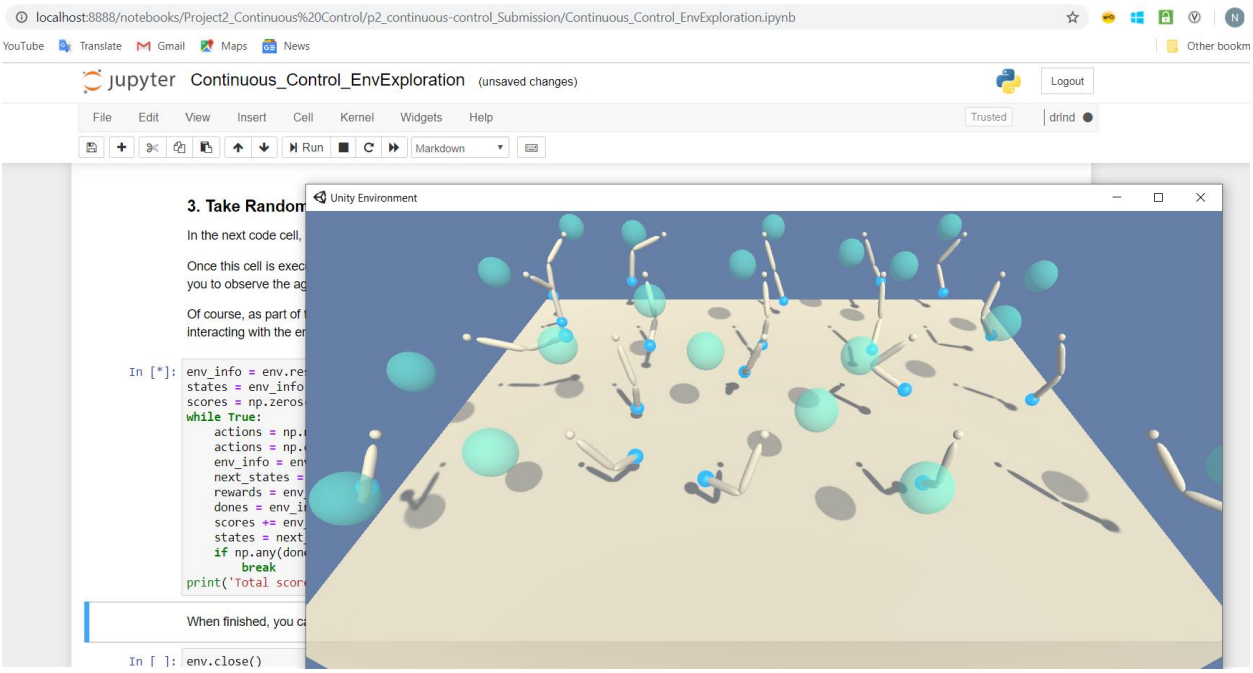
Step 3: Explore the Environment

After you have followed the instructions above, open **Continuous_Control.ipynb** (located in the **p2_continuous-control/** folder in the DRLND GitHub repository) and follow the instructions to learn how to use the Python API to control the agent.



Note: Don't forget to change the kernel to the Virtual Env "dlrnd"

Output of the Unity agent after running "**Continuous_Control_EnvExploration.ipynb**"



Description

- **Continuous_Control_EnvExploration.ipynb:** Notebook used to verify the Environment Setup and initial working of Unity Agent
- **Continuous_Control_Solution.ipynb:** Notebook used to control and train the final version of the agent
- **ddpg_agent.py:** Create an Agent class that interacts with and learns from the environment
- **model.py:** Actor and Critic classes
- **checkpoint_actor.pth** : saved model for the actor
- **checkpoint_critic.pth** : saved model for the critic
- **learning_curves.jpg :** Learning Curve Displaying the trained agent
- **report.pdf:** The submission includes a file in the root of the GitHub repository that provides a technical description of the implementation.
- **README.md :** The README describes the project environment details (i.e., the state and action spaces, and when the environment is considered solved). It has instructions for installing dependencies or downloading needed files and process to run the code in the repository, to train the agent.

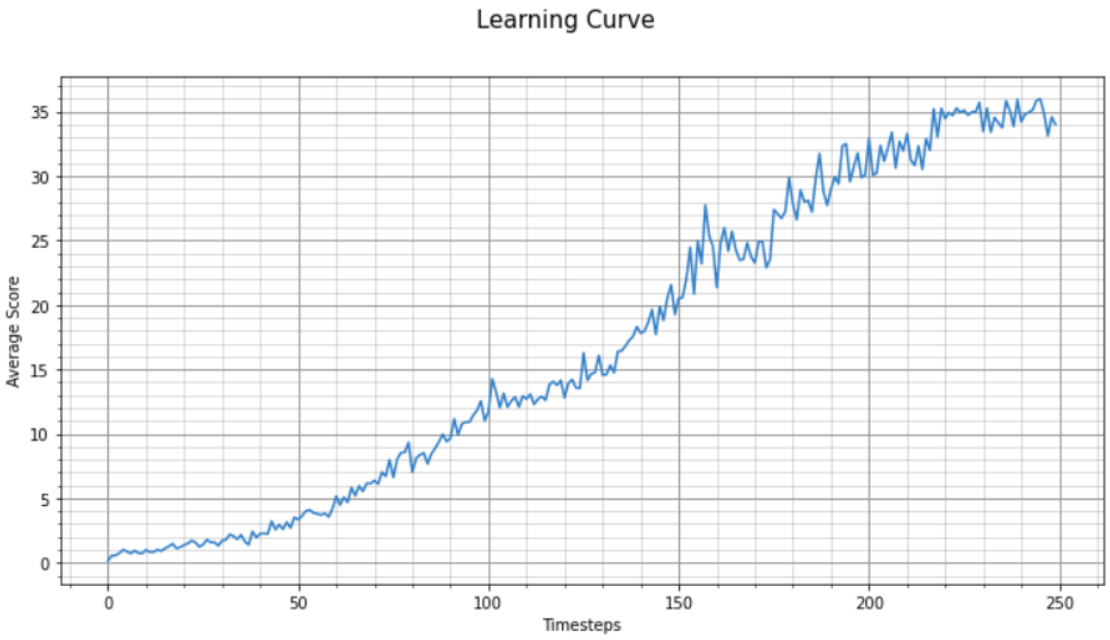
Steps to Run the Project

1. Download the environment and unzip it into the directory of the project. Rename the folder to Reacher_Windows_x86_64_20Agents and ensure that it contains Reacher.exe
2. Use jupyter to run the Continuous_Control_Solution.ipynb notebook: **`jupyter notebook Continuous_Control_Solution.ipynb`**
3. Run the cells in order in the notebook **`Continuous_Control_Solution.ipynb`** to train an agent that solves our required task of moving the double-jointed arm.
4. They will initialize the environment and train until it reaches the goal condition of +30
5. A graph of the scores during training will be displayed after training.

Results

Plot showing the score per episode over all the episodes.

Environment solved in 150 episodes with Average Score: 30.11



Ideas for future work

- We could have used different Optimization algorithm to check the difference in the Agent trained and its performance.

- There could be better results by making use of **prioritized experience replay** with the existing learning algorithm.

- **Distributed Distributional Deterministic Policy Gradients (D4PG)** has achieved state of the art results on continuous control problems. Also, PPO, A3C can be used in multi agent training environment.

- It would be interesting to see how the agent performs on this environment in future implementations.

- In future implementations, I can try testing the agent with difference hyperparameter values, like different Sigma values, faster and smaller learning rates, tweaking the neural network, to choose the final model

REPORT.md

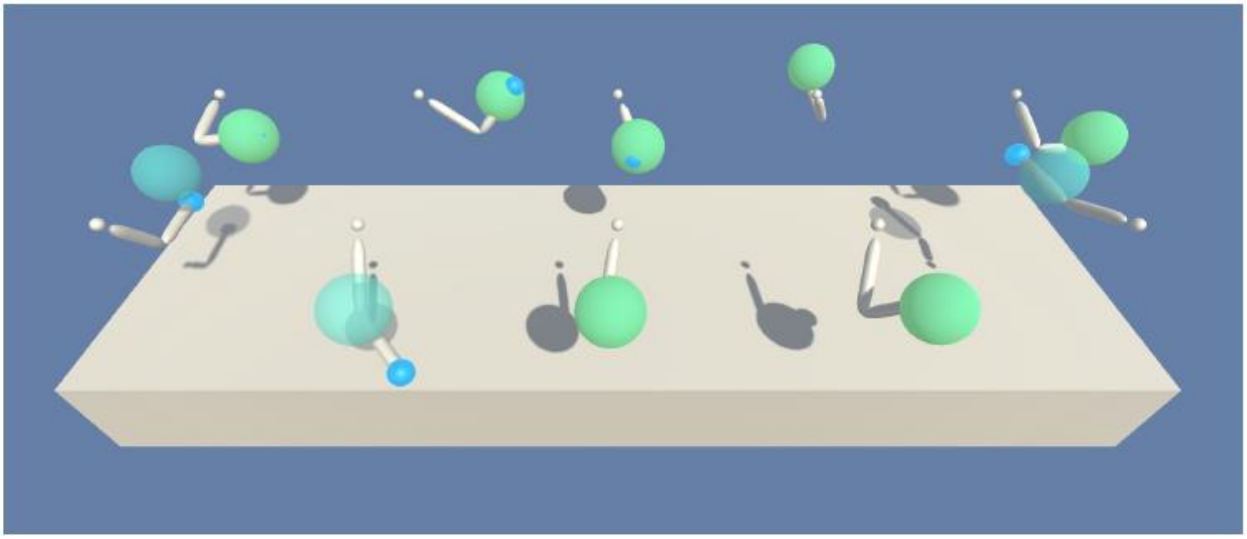
Deep Reinforcement Learning : Project 2: Continuous Control Report

Author : Nishi Sood

This document presents a technical description of the Continuous Control project in the context of the Deep Reinforcement Learning Nanodegree from Udacity.

Summary of environment and task

This project trains a Reinforcement Learning agent to solve a variant of the Reacher Environment agent by Unity.



Environment details

- The environment is based on Unity ML-agents. The project environment provided by Udacity is similar to the Reacher environment on the Unity ML-Agents GitHub page.
- **Set-up:** Double-jointed arm which can move to target locations.
- **Goal:** The agents must move its hand to the goal location, and keep it there.
- **Agents:** The environment contains 20 agents linked to a single Brain.
    - The provided Udacity agent versions are Single Agent or 20-Agents
- Agent Reward Function (independent):
    - +0.1 Each step agent's hand is in goal location.

- **Brains**: One Brain with the following observation/action space.
    - Vector Observation space: 26 variables corresponding to position, rotation, velocity, and angular velocities of the two arm Rigid bodies.
    - Vector Action space: (Continuous) Size of 4, corresponding to torque applicable to two joints.
    - Visual Observations: None.
- Reset Parameters: Two, corresponding to goal size, and goal movement speed.
- The task is episodic, with 1000 timesteps per episode. In order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.
- **Benchmark Mean Reward: 30**

About Deep Reinforcement Learning Terminologies

- Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a dimension over many steps; for example, maximize the points won in a game over many moves.
- They can start from a blank slate, and under the right conditions they achieve superhuman performance.
- Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.
- Methods Used:

Policy-based & value-based methods

- With value-based methods, the agent uses its experience with the environment to maintain an estimate of the optimal action-value function. The optimal policy is then obtained from the optimal action-value function estimate:

$$\text{Interaction} \rightarrow \text{Optimal Value Function } q_* \rightarrow \text{Optimal Policy } \pi_*$$

Value-based methods

- **Policy-based methods** directly learn the optimal policy, without having to maintain a separate value function estimate:

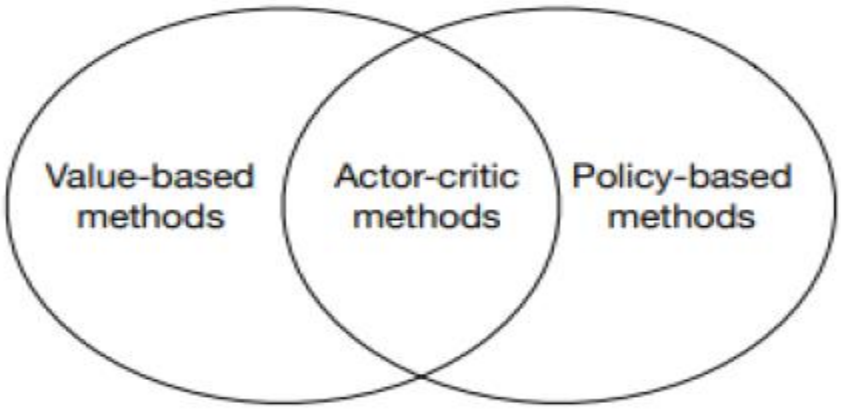
Interaction $\rightarrow$ Optimal Policy π_*

Policy-based methods

- One of the limitations of value-based methods is that they tend to a deterministic or near deterministic policies.
- On the other hand, policy-based methods can learn either stochastic or deterministic policies, so that they can be used to solve environments with either finite or continuous action spaces.

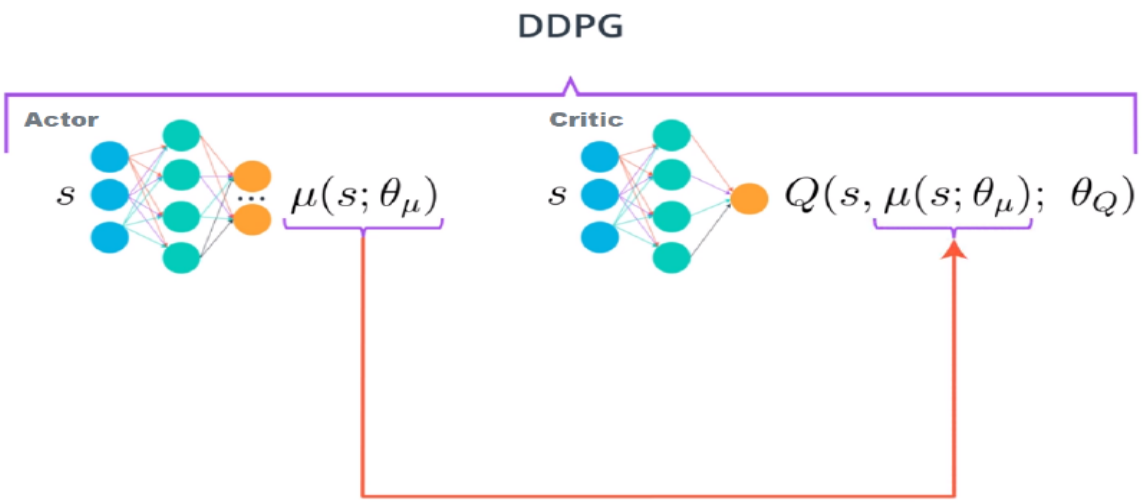
Actor-critic methods

- In actor-critic methods, we are using value-based techniques to further reduce the variance of policy-based methods.
- Basically actor-critic are a hybrid version of the policy- and value- based methods, in which the actor estimates the policy and the critic estimates the value function.



Deep Deterministic Policy Gradient (DDPG)

- Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy.
- It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.
- The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.
- In DDPG, we use 2 deep neural networks : one is the actor and the other is the critic:



More details available on the Open AI's Spinning Up website.

Algorithm

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

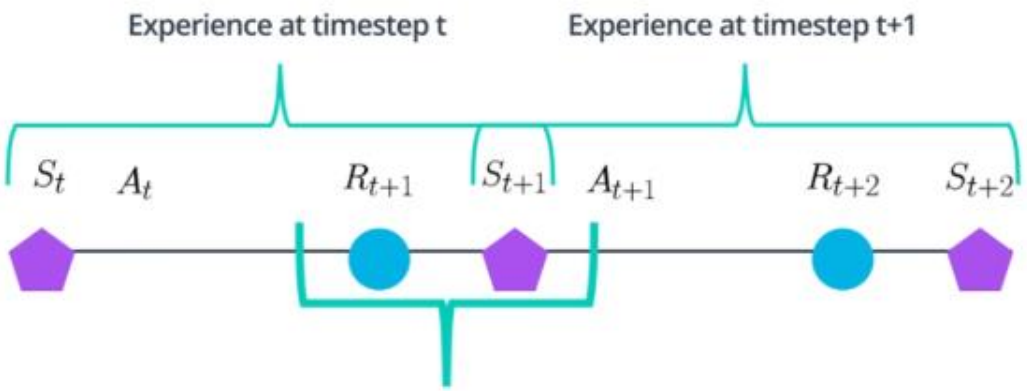
$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

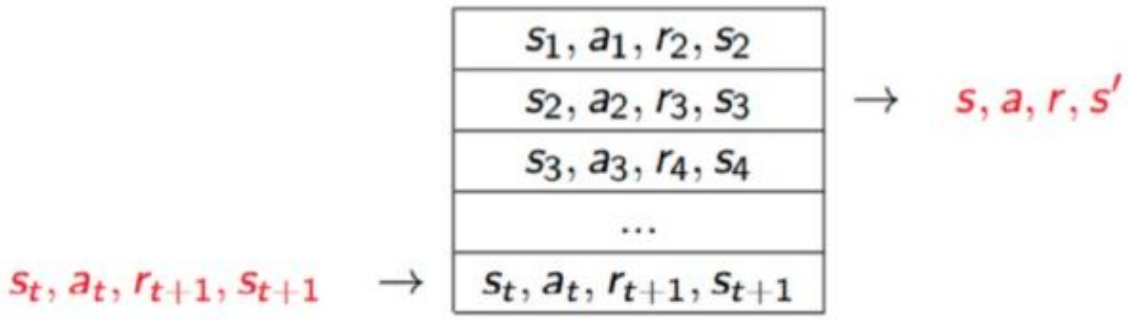
This algorithm screenshot is taken from the DDPG algorithm from the Spinning Up website

Replay Buffer

- In Reinforcement Learning, samples are generated from exploring sequentially in an environment, resulting in the previous assumption that no longer holds.
- Action A_t is partially responsible for the reward and state at time (t + 1) :



- DDPG is an off-policy algorithm, that is it employs a separate behavior policy independent of the policy being improved



Replay Buffer – fixed size

Code implementation Details

The code used here is derived from the "DDPG bipedal" tutorial from the Deep Reinforcement Learning Nanodegree, and has been slightly modified to handle multiple simultaneous agents in the same environment.

The code is written in Python 3.6 and is using PyTorch 0.4.0 framework.

The code consists of :

- **Continuous_Control_EnvExploration.ipynb:** Notebook used to verify the Environment Setup and initial working of Unity Agent
- **Continuous_Control_Solution.ipynb:** Notebook used to control and train the final version of the agent
- **ddpg_agent.py:** Create an Agent class that interacts with and learns from the environment
- **model.py:** Actor and Critic classes
- **checkpoint_actor.pth** : saved model for the actor
- **checkpoint_critic.pth** : saved model for the critic
- **learning_curves.jpg :** Learning Curve Displaying the trained agent
- **report.pdf:** The submission includes a file in the root of the GitHub repository that provides a description of the implementation.
- **README.md :** The README describes the project environment details (i.e., the state and action spaces, and when the environment is considered solved). It has instructions for installing dependencies or downloading needed files and process to run the code in the repository, to train the agent.

The **Actor Neural Networks** use the following architecture :

```
# ACTOR NETWORK
Input nodes (33)
  -> Fully Connected Layer (256 nodes, Relu activation)
    -> Batch Normlization
      -> Fully Connected Layer (128 nodes, Relu activation)
         -> Ouput nodes (4 nodes, tanh activation)
```

The **Critic Neural Networks** use the following architecture :

```
# CRITIC NETWORK
Input nodes (33)
  -> Fully Connected Layer (128 nodes, Relu activation)
    -> Batch Normlization
      -> Include Actions at the second fully connected layer
        -> Fully Connected Layer (256+33 nodes, Relu activation)
          -> Fully Connected Layer (2128 nodes, Relu activation)
            -> Ouput node (1 node, no activation)
```

Training of the agent was performed on the `train.ipynb` notebook.

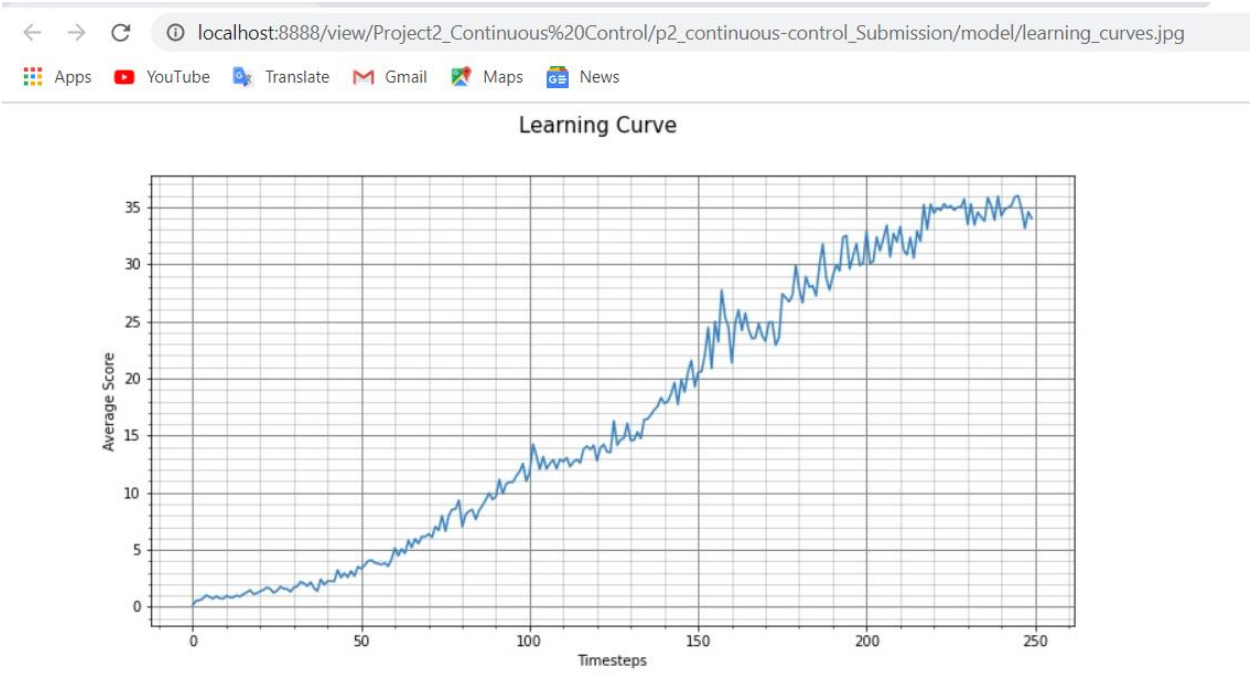
Training

Training of the agent was performed on
the **`Continuous_Control_Solution.ipynb`** notebook.

1. Run the cells in order in the notebook **`Continuous_Control_Solution.ipynb`** to train an agent that solves our required task of moving the double-jointed arm.
2. They will initialize the environment and train until it reaches the goal condition of +30
3. The agent is trained until it solves the environment, that is to say an average reward of at least +30 for the last 100 episodes.
4. A graph of the scores during training will be displayed after training.

Results

The agent was able to solve the environment in 150 episodes with an average Score: 30.11. Below is the learning curve.




```
Environment solved in 150 episodes!  Average Score: 30.11
```

Ideas for future work

- We could have used different Optimization algorithm to check the difference in the Agent trained and its performance.

- There could be better results by making use of prioritized experience replay with the existing learning algorithm.

- **Distributed Distributional Deterministic Policy Gradients** - D4PG looks very interesting has achieved state of the art results on continuous control problems

Algorithm 1 D4PG

Input: batch size M, trajectory length N, number of actors K, replay size R, exploration constant ϵ, initial learning rates α_0 and β_0

Initialize network weights (θ, w) at random
Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
Launch K actors and replicate network weights (θ, w) to each actor
for $t = 1, \ldots, T$ **do**
  Sample M transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length N from replay with priority p_i
  Construct the target distributions $Y_i = \sum_{n=0}^{N-1} \gamma^n r_{i+n} + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
  Compute the actor and critic updates

$$\delta_w = \frac{1}{M} \sum_i \nabla_w (Rp_i)^{-1} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$$

$$\delta_\theta = \frac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i)\ \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}_i, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$$

  Update network parameters $\theta \leftarrow \theta + \alpha_t\, \delta_\theta$, $w \leftarrow w + \beta_t\, \delta_w$
  If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
  If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
end for
return policy parameters θ

Actor

repeat
  Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon\mathcal{N}(0, 1)$
  Execute action $\mathbf{a}$, observe reward r and state $\mathbf{x}'$
  Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
until learner finishes

- Also, PPO, A3C can be used in multi agent training environment.

- It would be interesting to see how the agent performs on this environment in future implementations.

- In future implementations, I can try testing the agent with difference hyperparameter values, like different Sigma values, faster and smaller learning rates, tweaking the neural network, to choose the final model

- Open AI's blog post Better Exploration with Parameter Noise