Continuous Control

This document provides description of the implementation for the Continuous Control project.

The 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 the 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.

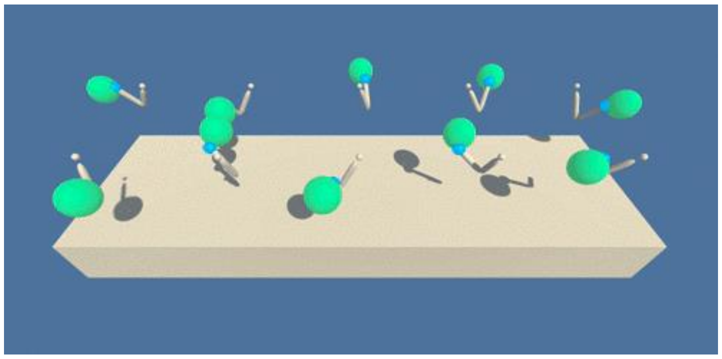


Figure 1. Unity ML 'Reacher' Environment

The version used in this project contains 20 identical agents, each with its own copy of the environment.

Solving the Environment

The barrier for solving this version of the environment is to take 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) are added up, to get a score for each agent. This yields 20 (potentially different) scores. Then the average of these 20 scores is taken.
- 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.

The Implementation

The implementation uses **DDPG (Deep Deterministic Policy Gradients)** to solve the task. A DDPG agent interacts with and learns from the Unity ML environment 'Reacher' for the task.

The agent implementation is written in PyTorch and the Python API was used to control the agent.

The solution has following files:

- ***Continuous_Control.ipynb***: Jupyter notebook, code for training and testing the agent.

- ***ddpg_agent.py***: The Agent, OUNoise, and ReplayBuffer classes. The Agent class implements DDPG agent. The sampled batch of experience tuples is provided thru ReplayBuffer class. ReplayBuffer class implements a fixed-size buffer to store experience tuples. OUNoise class implements Ornstein-Uhlenbeck noise process.

- ***model.py***: The Actor and Critic classes. Defines neural network architectures for actor (policy) and critic (value) networks.

- ***checkpoint_actor.pth***: Saved model weights for the Actor network of the trained agent.

- ***checkpoint_critic.pth***: Saved model weights for the Critic network of the trained agent.

The **Actor** class has following methods:
- *__init__*: Initializes parameters and build model
- *reset parameters*: resets parameters
- *forward*: Builds an actor (policy) network that maps states -> actions

The **Critic** class has following methods:
- *__init__*: Initializes parameters and build model
- *reset parameters*: resets parameters
- *forward*: Builds a critic (value) network that maps states -> actions

The **Agent** class has following methods:
- *__init__*: Initializes an Agent object.
- *step*: Saves experience in replay memory.
- *act*: Returns actions for given state as per current policy.
- *learn*: Updates value parameters using given batch of experience tuples.
- *soft_update*: Soft updates model parameters.

The **OUNoise** class has following methods:

- *__init__*: Initializes parameters and noise process.
- *reset*: Reset the internal state (=noise) to mean (=mu).
- *sample*: Update internal state and return it as a noise sample.

The **ReplayBuffer** class has following methods:

- *__init__*: Initializes a ReplayBuffer object.
- *add*: Adds a new experience to memory.
- *sample*: Randomly samples a batch of experiences from memory.
- *__len__*: Returns the current size of internal memory.

DDPG Agent's Actor and Critic network architectures consists of fully connected layers. Input layers have 33 inputs (corresponding to 33 dimensions of the environment state). Actor network has two hidden layers (256 nodes each), Critic network has three hidden layers (256, 256 and 128 nodes). Actor network has 4 outputs (corresponding to 4 actions of the agent) and Critic network has one output.

The architecture of the DDPG Agent's Actor (local and target) network model looks like the following:

```
Actor(
  (fc1): Linear(in_features=33, out_features=256, bias=True)
  (bn): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=256, out_features=4, bias=True)
)
```

The architecture of the DDPG Agent's Critic (local and target) network model looks like the following:

```
Critic(
  (fcs1): Linear(in_features=33, out_features=256, bias=True)
  (fc2): Linear(in_features=260, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=1, bias=True)
)
```

Adam optimizer is used as the optimizer.

Ornstein-Uhlenbeck noise process was used to add noise to the action space.

In order to make the code work with 20 agents, the agent code was implemented in such a way that after each step:

- each agent adds its experience to a replay buffer that is shared by all agents, and
- the (local) actor and critic networks are updated, using a sample from the replay buffer.

To get less aggressive with the number of updates per time step, instead of updating the actor and critic networks **20 times** at **every timestep**, the code was changed to update the networks **10 times** after every **20 timesteps** as suggested in the benchmark implementation.

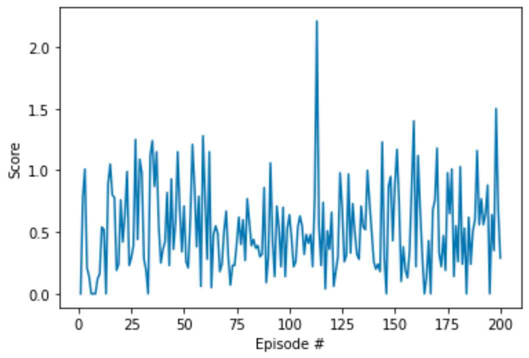
Replay buffer size was set to 1,000,000 to be sufficiently large for 20 agents.

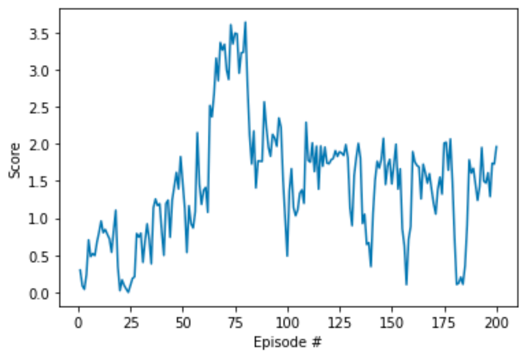
Gradient clipping was used when training the critic network as suggested in the benchmark implementation. The corresponding snippet of code was as follows:

```
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

Batch normalization was added to the Actor network after the first layer.

However, the first runs of training were quite unsuccessful, and the agent did not converge at all.





The main change that appeared to make the biggest difference was decaying the amount of noise being introduced to the action space during training over time.

It seems that in this project the balance between exploration and exploitation is critical.

By decaying the amount of noise, exploration has been encouraged at the beginning, but, after a while, exploitation has been encouraged.

The agent was trained in a Jupyter notebook.

Environment was solved in **294** episodes with an average score of **30.01**. Figure 2 and Figure 3 show how agent's score evolves throughout the training. It can be observed that the agent is able to receive an average score (over 100 episodes) of at least +30.

The training was continued after the environment was solved for about 100 more episodes (up to 400 episodes) to be able to see the stability of the training. It is observed that the average score (over 100 episodes) reached to **36.47**. The maximum average score over all episodes has been **38.70**.

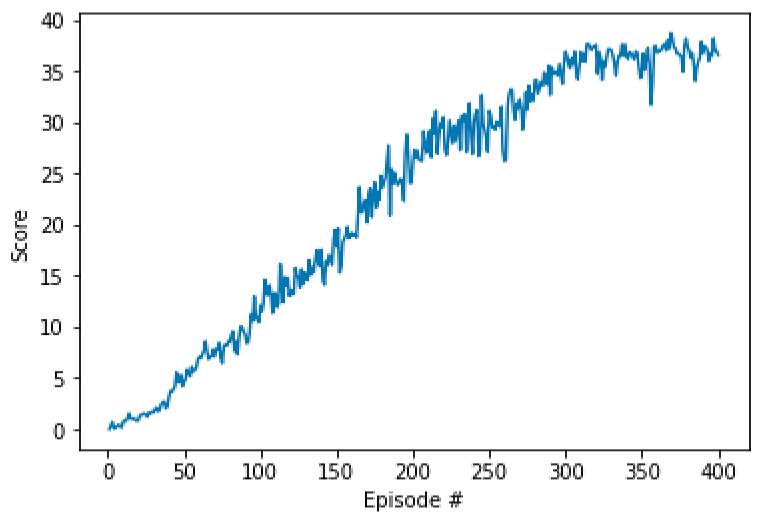
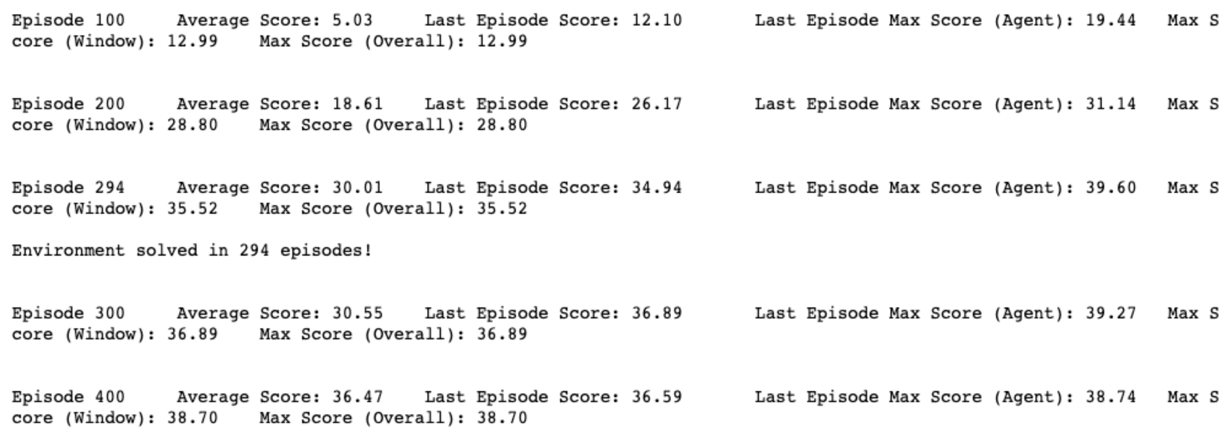


Figure 2. Plotted Scores



```
Episode 100     Average Score: 5.03     Last Episode Score: 12.10       Last Episode Max Score (Agent): 19.44    Max S
core (Window): 12.99     Max Score (Overall): 12.99


Episode 200     Average Score: 18.61    Last Episode Score: 26.17       Last Episode Max Score (Agent): 31.14    Max S
core (Window): 28.80     Max Score (Overall): 28.80


Episode 294     Average Score: 30.01    Last Episode Score: 34.94       Last Episode Max Score (Agent): 39.60    Max S
core (Window): 35.52     Max Score (Overall): 35.52

Environment solved in 294 episodes!


Episode 300     Average Score: 30.55    Last Episode Score: 36.89       Last Episode Max Score (Agent): 39.27    Max S
core (Window): 36.89     Max Score (Overall): 36.89


Episode 400     Average Score: 36.47    Last Episode Score: 36.59       Last Episode Max Score (Agent): 38.74    Max S
core (Window): 38.70     Max Score (Overall): 38.70
```

Figure 4. Training Execution

Hyperparameters and Settings

The various hyperparameters and settings used throughout the implementation are given below:

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-4	# learning rate of the actor
LR_CRITIC = 1e-3	# learning rate of the critic
WEIGHT_DECAY = 0	# L2 weight decay
UPDATE_INTERVAL = 20	# network update interval in timesteps
UPDATE_TIMES = 10	# number of network update passes at each interval

```
EPSILON = 1.0                # starting value of epsilon for the noise added to the actions
EPSILON_DECAY = 1e-5         # decay for epsilon to be subtracted for each timestep of episodes

n_episodes =  400            # maximum number of training episodes
max_t = 1000                 # maximum number of timesteps per episode

state_size = 33              # dimension of each state
action_size = 4              # dimension of each action
seed = 2                     # random seed

fc_units = 256               # number of nodes in hidden layers of Actor network

fc1_units = 256              # number of nodes in first hidden layer of Critic network
fc2_units = 256              # number of nodes in second hidden layer of Critic network
fc3_units = 128              # number of nodes in third hidden layer of Critic network

mu = 0                       # mu parameter of Ornstein-Uhlenbeck noise process
theta = 0.15                 # theta parameter of Ornstein-Uhlenbeck noise process
sigma = 0.2                  # sigma parameter of Ornstein-Uhlenbeck noise process
```

Future Improvements

Although they are not tried in the implementation, if it is required to improve the agent's performance with a method that will be more stable, Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG) could achieve better performance. An implementation of Proximal Policy Optimization (PPO) can also be done, which has also demonstrated good performance with continuous control tasks.

Recent Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm can be tried as another method for adapting DDPG for continuous control.