Continuous_Control

January 22, 2021

1 Continuous Control

In this notebook, you will learn how to use the Unity ML-Agents environment for the second project of the Deep Reinforcement Learning Nanodegree program.

1.0.1 1. Start the Environment

We begin by importing the necessary packages. If the code cell below returns an error, please revisit the project instructions to double-check that you have installed Unity ML-Agents and NumPy.

[1]:
```python
from unityagents import UnityEnvironment
import numpy as np
```

Next, we will start the environment! ***Before running the code cell below***, change the `file_name` parameter to match the location of the Unity environment that you downloaded.

- **Mac**: `"path/to/Reacher.app"`
- **Windows** (x86): `"path/to/Reacher_Windows_x86/Reacher.exe"`
- **Windows** (x86_64): `"path/to/Reacher_Windows_x86_64/Reacher.exe"`
- **Linux** (x86): `"path/to/Reacher_Linux/Reacher.x86"`
- **Linux** (x86_64): `"path/to/Reacher_Linux/Reacher.x86_64"`
- **Linux** (x86, headless): `"path/to/Reacher_Linux_NoVis/Reacher.x86"`
- **Linux** (x86_64, headless): `"path/to/Reacher_Linux_NoVis/Reacher.x86_64"`

For instance, if you are using a Mac, then you downloaded `Reacher.app`. If this file is in the same folder as the notebook, then the line below should appear as follows:

```
env = UnityEnvironment(file_name="Reacher.app")
```

[2]:
```python
env = UnityEnvironment(file_name='Reacher_20.app')
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
                goal_speed -> 1.0
```

```
                goal_size -> 5.0
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

[3]:
```python
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

1.0.2 2. Examine the State and Action Spaces

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 must be a number between `-1` and `1`.

Run the code cell below to print some information about the environment.

[4]:
```python
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)

# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.
 ↪format(states.shape[0], state_size))
print('The state for the first agent looks like:', states[0])
```

```
Number of agents: 1
Size of each action: 4
```

```
There are 1 agents. Each observes a state with length: 33
The state for the first agent looks like: [ 0.00000000e+00 -4.00000000e+00
0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
  5.55726671e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

1.0.3 3. Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agent and receive feedback from the environment.

Once this cell is executed, you will watch the agent's performance, if it selects an action at random with each time step. A window should pop up that allows you to observe the agent, as it moves through the environment.

Of course, as part of the project, you'll have to change the code so that the agent is able to use its experience to gradually choose better actions when interacting with the environment!

[5]:
```
env_info = env.reset(train_mode=False)[brain_name]     # reset the environment ␣
 ↪
states = env_info.vector_observations                  # get the current state␣
 ↪(for each agent)
scores = np.zeros(num_agents)                          # initialize the score␣
 ↪(for each agent)
while True:
    actions = np.random.randn(num_agents, action_size) # select an action (for␣
 ↪each agent)
    actions = np.clip(actions, -1, 1)                  # all actions between -1␣
 ↪and 1
    env_info = env.step(actions)[brain_name]           # send all actions to␣
 ↪tne environment
    next_states = env_info.vector_observations         # get next state (for␣
 ↪each agent)
    rewards = env_info.rewards                         # get reward (for each␣
 ↪agent)
    dones = env_info.local_done                        # see if episode finished
    scores += env_info.rewards                         # update the score (for␣
 ↪each agent)
    states = next_states                               # roll over states to␣
 ↪next time step
    if np.any(dones):                                  # exit loop if episode␣
 ↪finished
```

```
        break
print('Total score (averaged over agents) this episode: {}'.format(np.
 ↪mean(scores)))
```

```
Total score (averaged over agents) this episode: 0.0
```

When finished, you can close the environment.

[6]:
```
env.close()
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

```
env_info = env.reset(train_mode=True)[brain_name]
```

[1]:
```
from unityagents import UnityEnvironment
import numpy as np

env = UnityEnvironment(file_name='Reacher_single.app')
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
                goal_size -> 5.0
                goal_speed -> 1.0
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

[2]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

[3]:
```
import os.path

def restore_agent(actor_name, filepath_local_actor, filepath_local_critic,␣
 ↪filepath_target_actor, filepath_target_critic):
    # function to read and load saved weights into agent networks
```

```
    checkpoint_local_actor = torch.load(filepath_local_actor)
    checkpoint_local_critic = torch.load(filepath_local_critic)
    checkpoint_target_actor = torch.load(filepath_target_actor)
    checkpoint_target_critic = torch.load(filepath_target_critic)

    if actor_name == 'ddpg':
        loaded_agent = Agent(state_size, action_size, random_seed = 33,␣
↪max_score= 40)
    elif actor_name == 'td3':
        loaded_agent = Agent(state_size, action_size, random_seed = 33,␣
↪policy_noise=0.2)

    loaded_agent.actor_local.load_state_dict(checkpoint_local_actor)
    loaded_agent.actor_target.load_state_dict(checkpoint_target_actor)
    loaded_agent.critic_local.load_state_dict(checkpoint_local_critic)
    loaded_agent.critic_target.load_state_dict(checkpoint_target_critic)

    return loaded_agent
```

[4]:
```
from collections import deque
import torch

def run_experiment(agent, n_episodes=2000, max_t=10000,␣
↪agent_ckp_prefix='agent', critic_ckp_prefix='critic'):

    scores_deque = deque(maxlen=100)
    rolling_average_score = []
    scores = []

    current_score = 0

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]      # reset the␣
↪environment
        states = env_info.vector_observations                  # get the current␣
↪state
        score = np.zeros(num_agents)
        agent.reset()                                          # reset the agent
        for t in range(max_t):
            actions = agent.act(states, current_score, add_noise=False)

            env_info = env.step(actions)[brain_name]           # send all␣
↪actions to tne environment
            next_states = env_info.vector_observations         # get next␣
↪state (for each agent)
```

```
            rewards = env_info.rewards                            # get reward␣
↪(for each agent)
            dones = env_info.local_done                           # see if␣
↪episode finished

            for state, action, reward, next_state, done in zip(states, actions,␣
↪rewards, next_states, dones):
                agent.step(state, action, reward, next_state, done, t)

            states = next_states
            score += rewards

            if np.any(dones):
                break

        score = np.mean(score)
        scores_deque.append(score)
        rolling_average_score.append(np.mean(scores_deque))
        scores.append(score)

        current_score = np.mean(scores_deque)

        print('\rEpisode {}\tAverage Score: {:.2f}\tScore: {:.2f}'.
↪format(i_episode,
                                                                                   np.
↪mean(scores_deque),
                                                                                  ␣
↪score), end='')

        if i_episode % 10 == 0:
            print('\rSave_agent\r')
            torch.save(agent.actor_local.state_dict(),␣
↪agent_ckp_prefix+'_ckpt_local.pth')                      # save local actor
            torch.save(agent.actor_target.state_dict(),␣
↪agent_ckp_prefix+'_ckpt_target.pth')                     # save target actor
            torch.save(agent.critic_local.state_dict(),␣
↪critic_ckp_prefix+'_ckpt_local.pth')                     # save local critic
            torch.save(agent.critic_target.state_dict(),␣
↪critic_ckp_prefix+'_ckpt_target.pth')                  # target critic
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_deque)))
        if np.mean(scores_deque)>= 30.0:
            torch.save(agent.actor_local.state_dict(),␣
↪agent_ckp_prefix+'_ckpt_local.pth')                      # save local actor
```

```
            torch.save(agent.actor_target.state_dict(),␣
↪agent_ckp_prefix+'_ckpt_target.pth')               # save target actor
            torch.save(agent.critic_local.state_dict(),␣
↪critic_ckp_prefix+'_ckpt_local.pth')               # save local critic
            torch.save(agent.critic_target.state_dict(),␣
↪critic_ckp_prefix+'_ckpt_target.pth')
            print('\rEnvironment solved Episode {}\tAverage Score: {:.2f}'.
↪format(i_episode, np.mean(scores_deque)))
            break
    return scores, rolling_average_score
```

[5]:

```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)

# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.
↪format(states.shape[0], state_size))
print('The state for the first agent looks like:', states[0])
```

```
Number of agents: 1
Size of each action: 4
There are 1 agents. Each observes a state with length: 33
The state for the first agent looks like: [ 0.00000000e+00 -4.00000000e+00
0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
  5.55726671e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

1.1 DDPG

The first algorithm will be a standard DDPG as found in the examples from the Udacity DeepLearning NanoDegree: https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpgbipedal

It will be adapted to solve the Reacher environment

[6]:
```
from actors.ddpg_actor import Agent

agent_name = 'checkpoints/agent_ddpg_noise_reduction_single'
critic_name = 'checkpoints/critic_ddpg_noise_reduction_single'

local_actor_path = agent_name+'_ckpt_local.pth'
target_actor_path = agent_name+'_ckpt_target.pth'
local_critic_path = critic_name+'_ckpt_local.pth'
target_critic_path = critic_name+'_ckpt_target.pth'


# if checkpoint exists we load the agent
if os.path.isfile(local_actor_path):
    agent = restore_agent('ddpg', local_actor_path, local_critic_path,␣
 ↪target_actor_path, target_critic_path)
    print("Agent loaded.")
else:
    agent = Agent(state_size, action_size, random_seed = 33, max_score= 40)
    print("Agent created.")
```

```
Agent created.
```

[ ]:
```
scores, rolling_average = run_experiment(agent, agent_ckp_prefix=agent_name,␣
 ↪critic_ckp_prefix=critic_name)
```

[ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# plot scores across episodes
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='DDPG')
plt.plot(np.arange(len(scores)), rolling_average, c='r', label='Rolling AVG')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.show()
```

[ ]:
```
env.close()
```

1.2 TD3

The first improvement tried is the TD3 algorithm which essentially make 3 improvements to the DDPG. 1. Twin network for the critic 2. Add noise to actions used to compute targets 3. Delayed updates of the policy

Please restart the invirnonment before running the cells below

[6]:
```
from actors.td3_actor import Agent

agent_name = 'checkpoints/agent_td3_single'
critic_name = 'checkpoints/critic_td3_single'

local_actor_path = agent_name+'_ckpt_local.pth'
target_actor_path = agent_name+'_ckpt_target.pth'
local_critic_path = critic_name+'_ckpt_local.pth'
target_critic_path = critic_name+'_ckpt_target.pth'


# if checkpoint exists we load the agent
if os.path.isfile(local_actor_path):
    agent = restore_agent('td3', local_actor_path, local_critic_path,␣
 ↪target_actor_path, target_critic_path)
    print("Agent loaded.")
else:
    agent = Agent(state_size, action_size, random_seed = 33)
    print("Agent created.")
```

```
Agent created.
```

[7]:
```
# run
scores, rolling_average = run_experiment(agent, agent_ckp_prefix=agent_name,␣
 ↪critic_ckp_prefix=critic_name)
```

```
Save_agent      Average Score: 0.56     Score: 1.85
Save_agent      Average Score: 0.82     Score: 1.03
Save_agent      Average Score: 0.76     Score: 0.34
Save_agent      Average Score: 1.12     Score: 5.80
Save_agent      Average Score: 1.20     Score: 4.25
Save_agent      Average Score: 1.25     Score: 1.18
Save_agent      Average Score: 1.38     Score: 2.81
Save_agent      Average Score: 1.64     Score: 2.18
Save_agent      Average Score: 1.82     Score: 3.27
Save_agent0     Average Score: 1.99     Score: 4.64
Episode 100     Average Score: 1.99
Save_agent0     Average Score: 2.29     Score: 6.09
Save_agent0     Average Score: 2.58     Score: 2.31
Save_agent0     Average Score: 3.03     Score: 4.36
Save_agent0     Average Score: 3.38     Score: 6.66
Save_agent0     Average Score: 4.00     Score: 8.418
Save_agent0     Average Score: 4.56     Score: 4.266
Save_agent0     Average Score: 5.21     Score: 9.598
```

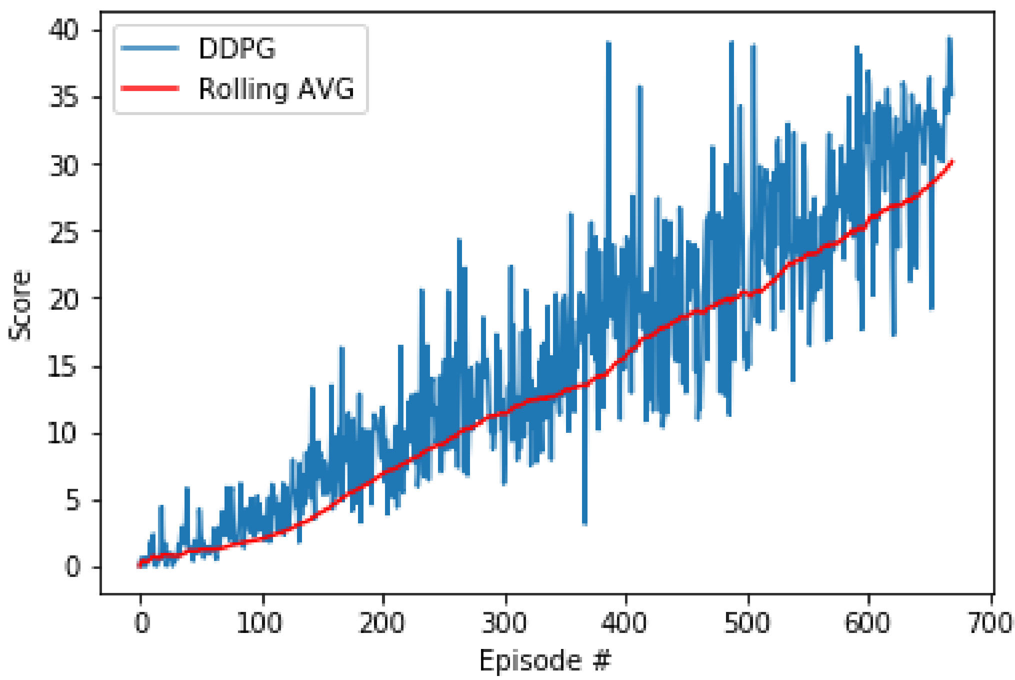
```
Save_agent0     Average Score: 5.69     Score: 4.686
Save_agent0     Average Score: 6.22     Score: 10.10
Save_agent0     Average Score: 6.79     Score: 7.972
Episode 200     Average Score: 6.79
Save_agent0     Average Score: 7.21     Score: 5.085
Save_agent0     Average Score: 7.69     Score: 7.199
Save_agent0     Average Score: 8.16     Score: 5.938
Save_agent0     Average Score: 8.74     Score: 10.78
Save_agent0     Average Score: 9.05     Score: 7.132
Save_agent0     Average Score: 9.67     Score: 12.67
Save_agent0     Average Score: 10.25    Score: 11.56
Save_agent0     Average Score: 10.62    Score: 13.43
Save_agent0     Average Score: 11.14    Score: 10.01
Save_agent0     Average Score: 11.43    Score: 12.38
Episode 300     Average Score: 11.43
Save_agent0     Average Score: 11.91    Score: 13.63
Save_agent0     Average Score: 12.27    Score: 10.87
Save_agent0     Average Score: 12.49    Score: 15.29
Save_agent0     Average Score: 12.62    Score: 7.945
Save_agent0     Average Score: 13.02    Score: 12.70
Save_agent0     Average Score: 13.30    Score: 15.93
Save_agent0     Average Score: 13.47    Score: 17.79
Save_agent0     Average Score: 14.16    Score: 23.47
Save_agent0     Average Score: 14.76    Score: 18.51
Save_agent0     Average Score: 15.44    Score: 14.69
Episode 400     Average Score: 15.44
Save_agent0     Average Score: 16.34    Score: 21.39
Save_agent0     Average Score: 17.05    Score: 12.32
Save_agent0     Average Score: 17.63    Score: 11.50
Save_agent0     Average Score: 18.20    Score: 21.41
Save_agent0     Average Score: 18.61    Score: 17.50
Save_agent0     Average Score: 19.05    Score: 23.61
Save_agent0     Average Score: 19.26    Score: 26.28
Save_agent0     Average Score: 19.70    Score: 25.89
Save_agent0     Average Score: 20.02    Score: 28.48
Save_agent0     Average Score: 20.41    Score: 15.80
Episode 500     Average Score: 20.41
Save_agent0     Average Score: 20.50    Score: 23.09
Save_agent0     Average Score: 21.06    Score: 21.78
Save_agent0     Average Score: 21.92    Score: 19.11
Save_agent0     Average Score: 22.62    Score: 32.34
Save_agent0     Average Score: 23.16    Score: 23.13
Save_agent0     Average Score: 23.37    Score: 23.51
Save_agent0     Average Score: 23.92    Score: 17.01
Save_agent0     Average Score: 24.38    Score: 28.32
Save_agent0     Average Score: 24.88    Score: 26.06
Save_agent0     Average Score: 25.48    Score: 31.59
Episode 600     Average Score: 25.48
```

```
Save_agent0      Average Score: 26.24    Score: 34.58
Save_agent0      Average Score: 26.87    Score: 30.10
Save_agent0      Average Score: 27.08    Score: 36.05
Save_agent0      Average Score: 27.50    Score: 23.18
Save_agent0      Average Score: 28.25    Score: 33.02
Save_agent0      Average Score: 29.08    Score: 30.25
Save_agent0      Average Score: 30.15    Score: 35.12
Environment solved Episode 670  Average Score: 30.15
```

[8]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# plot scores across episodes
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='TD3')
plt.plot(np.arange(len(scores)), rolling_average, c='r', label='Rolling AVG')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.show()
```



[9]:

```
env.close()
```

[ ]: