Continuous_Control_20

January 30, 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.

```
In [ ]: 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")
```

```
In [ ]: #env = UnityEnvironment(file_name='/data/Reacher_One_Linux_NoVis/Reacher_One_Linux_NoVis
        env = UnityEnvironment(file_name='/data/Reacher_Linux_NoVis/Reacher.x86_64')
```

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.

```
In [ ]: # 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.

```
In [ ]: # 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
        print('The state for the first agent looks like:', states[0])
```

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!

```
In [ ]: env_info = env.reset(train_mode=False)[brain_name]     # reset the environment
        states = env_info.vector_observations                  # get the current state (for each
        scores = np.zeros(num_agents)                          # initialize the score (for each
        while True:
            actions = np.random.randn(num_agents, action_size) # select an action (for each agen
            actions = np.clip(actions, -1, 1)                  # all actions between -1 and 1
            env_info = env.step(actions)[brain_name]           # send all actions to tne environ
            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 agen
```

```
            states = next_states                                # roll over states to next time s
            if np.any(dones):                                   # exit loop if episode finished
                break
        print('Total score (averaged over agents) this episode: {}'.format(np.mean(scores)))
```

When finished, you can close the environment.

```
In [ ]: 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]
```

```
In [1]: !pip -q install ./python
```

```
tensorflow 1.7.1 has requirement numpy>=1.13.3, but you'll have numpy 1.12.1 which is incompatib
ipython 6.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.15, but you'll have prompt-toolkit 3.0.
```

```
In [2]: from unityagents import UnityEnvironment
        import numpy as np

        #env = UnityEnvironment(file_name='/data/Reacher_One_Linux_NoVis/Reacher_One_Linux_NoVis
        env = UnityEnvironment(file_name='/data/Reacher_Linux_NoVis/Reacher.x86_64')
```

```
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: , , ,
```

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

In [4]:
```
import os.path

def restore_agent(actor_name, filepath_local_actor, filepath_local_critic, filepath_targ
    # 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)
    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
```

In [5]:
```
from collections import deque
import torch

def run_experiment(agent, n_episodes=2000, max_t=10000, agent_ckp_prefix='agent', critic

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

    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, add_noise=False)

            env_info = env.step(actions)[brain_name]               # send all actions to
            next_states = env_info.vector_observations             # get next state (for
            rewards = env_info.rewards                             # get reward (for each
            dones = env_info.local_done                            # see if episode finis

            for state, action, reward, next_state, done in zip(states, actions, rewards,
                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)

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

            if i_episode % 10 == 0:
                print('\rSave_agent\r')
                torch.save(agent.actor_local.state_dict(), agent_ckp_prefix+'_ckpt_local.pth
                torch.save(agent.actor_target.state_dict(), agent_ckp_prefix+'_ckpt_target.p
                torch.save(agent.critic_local.state_dict(), critic_ckp_prefix+'_ckpt_local.p
                torch.save(agent.critic_target.state_dict(), critic_ckp_prefix+'_ckpt_target
            if i_episode % 100 == 0:
                print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores
            if np.mean(scores_deque)>= 30.0 and i_episode >= 100:
                torch.save(agent.actor_local.state_dict(), agent_ckp_prefix+'_ckpt_local.pth
                torch.save(agent.actor_target.state_dict(), agent_ckp_prefix+'_ckpt_target.p
                torch.save(agent.critic_local.state_dict(), critic_ckp_prefix+'_ckpt_local.p
                torch.save(agent.critic_target.state_dict(), critic_ckp_prefix+'_ckpt_target
                print('\rEnvironment solved Episode {}\tAverage Score: {:.2f}'.format(i_epis
                break
        return scores, rolling_average_score
```

In [6]:
```
# 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
print('The state for the first agent looks like:', states[0])
```

```
Number of agents: 20
```

```
Size of each action: 4
There are 20 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
  -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
   5.55726624e+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/ddpg-bipedal

It will be adapted to solve the Reacher environment

```
In [7]: from actors.ddpg_actor import Agent

        agent_name = 'agent_ddpg_20_no_noise'
        critic_name = 'critic_ddpg_20_no_noise'

        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
            print("Agent loaded.")
        else:
            agent = Agent(state_size, action_size, random_seed = 33)
            print("Agent created.")
```

```
Agent created.
```

```
In [9]: from utils.workspace_utils import active_session

        with active_session():
            # run
            scores, rolling_average = run_experiment(agent, agent_ckp_prefix=agent_name, critic_
```

```
Save_agent          Average Score: 1.14         Score: 1.93
Save_agent          Average Score: 3.00         Score: 8.42
Save_agent          Average Score: 5.24         Score: 7.763
Save_agent          Average Score: 6.77         Score: 12.00
Save_agent          Average Score: 7.92         Score: 14.09
Save_agent          Average Score: 9.07         Score: 14.58
Save_agent          Average Score: 10.12         Score: 16.14
Save_agent          Average Score: 11.05         Score: 17.93
Save_agent          Average Score: 11.78         Score: 16.41
Save_agent0          Average Score: 12.52         Score: 18.96
Episode 100          Average Score: 12.52
Save_agent0          Average Score: 14.53         Score: 24.73
Save_agent0          Average Score: 16.80         Score: 31.33
Save_agent0          Average Score: 19.06         Score: 35.01
Save_agent0          Average Score: 21.66         Score: 38.13
Save_agent0          Average Score: 24.12         Score: 36.07
Save_agent0          Average Score: 26.32         Score: 37.14
Save_agent0          Average Score: 28.18         Score: 35.02
Save_agent0          Average Score: 29.92         Score: 36.01
Environment solved Episode 181        Average Score: 30.11
```

```
In [ ]: 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()
```

```
In [ ]: 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 invironnment before running the cells below

```
In [11]: from actors.td3_actor import Agent

          agent_name = 'agent_td3_20'
          critic_name = 'critic_td3_20'
```

```
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
    print("Agent loaded.")
else:
    agent = Agent(state_size, action_size, random_seed = 33)
    print("Agent created.")
```

```
Agent created.
```

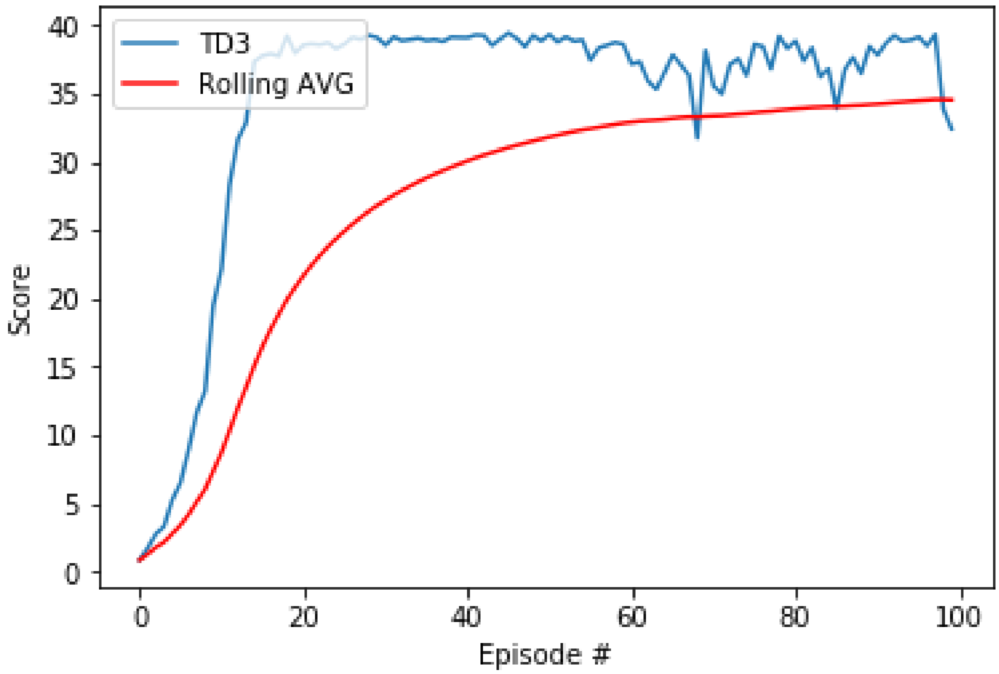
```
In [12]: from utils.workspace_utils import active_session

         with active_session():
             # run
             scores, rolling_average = run_experiment(agent, agent_ckp_prefix=agent_name, critic
```

```
Save_agent        Average Score: 7.37        Score: 19.46
Save_agent        Average Score: 20.83        Score: 37.94
Save_agent        Average Score: 26.81        Score: 39.08
Save_agent        Average Score: 29.84        Score: 39.09
Save_agent        Average Score: 31.67        Score: 38.84
Save_agent        Average Score: 32.83        Score: 38.59
Save_agent        Average Score: 33.33        Score: 38.13
Save_agent        Average Score: 33.82        Score: 38.29
Save_agent        Average Score: 34.18        Score: 38.44
Save_agent0        Average Score: 34.53        Score: 32.40
Episode 100        Average Score: 34.53
Environment solved Episode 100        Average Score: 34.53
```

```
In [13]: 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()
```




```
In [14]: env.close()
```

```
In [ ]:
```