Tennis

February 18, 2021

1 Collaboration and Competition

In this notebook, you will learn how to use the Unity ML-Agents environment for the third 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]:
```
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/Tennis.app"`
- **Windows** (x86): `"path/to/Tennis_Windows_x86/Tennis.exe"`
- **Windows** (x86_64): `"path/to/Tennis_Windows_x86_64/Tennis.exe"`
- **Linux** (x86): `"path/to/Tennis_Linux/Tennis.x86"`
- **Linux** (x86_64): `"path/to/Tennis_Linux/Tennis.x86_64"`
- **Linux** (x86, headless): `"path/to/Tennis_Linux_NoVis/Tennis.x86"`
- **Linux** (x86_64, headless): `"path/to/Tennis_Linux_NoVis/Tennis.x86_64"`

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

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

[2]:
```
env = UnityEnvironment(file_name="Tennis_Linux/Tennis.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 :
```

```
Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        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, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

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: 2
Size of each action: 2
```

```
There are 2 agents. Each observes a state with length: 24
The state for the first agent looks like: [ 0.          0.          0.
0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.         -6.65278625 -1.5
 -0.          0.          6.83172083  6.         -0.          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 agents and receive feedback from the environment.

Once this cell is executed, you will watch the agents' performance, if they select actions at random with each time step. A window should pop up that allows you to observe the agents.

Of course, as part of the project, you'll have to change the code so that the agents are able to use their experiences to gradually choose better actions when interacting with the environment!

[ ]:
```
for i in range(1, 6):                                      # play game for 5␣
 ↪episodes
    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
        print(actions.shape)
        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('Score (max over agents) from episode {}: {}'.format(i, np.
↪max(scores)))
```

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="Tennis_Linux/Tennis.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 :

Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
```

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

[3]:
```
# 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(states.shape)
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: 2
Size of each action: 2
(2, 24)
There are 2 agents. Each observes a state with length: 24
The state for the first agent looks like: [ 0.          0.          0.
0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.         -6.65278625 -1.5
 -0.          0.          6.83172083  6.         -0.          0.        ]
```

[4]:

```
from utils.utilities import transpose_list, transpose_to_tensor
from utils.buffer import ReplayBuffer

from collections import deque
import os

import torch

BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 256         # minibatch size
LEARNING_STEPS = 15
LEARN_EACH = 20

model_dir= os.getcwd()+"/checkpoints"

def run_experiment(agent, n_episodes=10000, agent_ckp_prefix='agent',␣
 ↪critic_ckp_prefix='critic'):
    scores_agent = [list() for i in range(num_agents)]
    rolling_average = [list() for i in range(num_agents)]
    scores_deque = [deque(maxlen = 100) for i in range(num_agents)]

    max_scores_deque = deque(maxlen = 100)
    max_scores_list = []
    prv_avg_score = 0
    average_score = 0
    max_average_score = 0
```

```
    # amplitude of OU noise
    # this slowly decreases to 0
    noise = 1.0
    noise_reduction = 0.9995
    noise_reduction_end = 0.1
    episodes_before_training=10
    steps_before_noise_reduction = 1000
    step = 0

    buffer = ReplayBuffer(BUFFER_SIZE)

    for i_episode in range(1, episodes_before_training+n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]     # reset the␣
↪environment
        obs = env_info.vector_observations                # get the current␣
↪state (2,24)
        states = [agent_state for agent_state in obs]     # make a list
        score = np.zeros(num_agents)
        agent.reset()

        while True:
            if i_episode <= episodes_before_training:
                actions_array = np.random.uniform(-1, 1, 4).reshape(num_agents,␣
↪action_size)
            else:
                actions = agent.act(transpose_to_tensor([states]), noise=noise)␣
↪       # each agent choose the actions
                actions_array = torch.stack(actions).detach().numpy().squeeze()␣
↪                 # numpy array from ations

            env_info = env.step(actions_array)[brain_name]              # The␣
↪environment accepts full actions

            next_obs = env_info.vector_observations                     # get␣
↪next observations
            next_states = [agent_state for agent_state in next_obs]     #␣
↪reshape them into a list

            rewards = np.array(env_info.rewards)
            dones = np.array(env_info.local_done)

            # add data to buffer
            transition = ([states, actions_array, rewards, next_states, dones])
            buffer.push(transition)
```

```
            score += rewards
            states = next_states # after each timestep update the obs to the␣
↪new obs before restarting the loop

            if np.any(dones):                                   # exit loop if␣
↪episode finished
                break

            if (len(buffer) >= BATCH_SIZE) and i_episode >␣
↪episodes_before_training and step % LEARN_EACH == 0: # Learn
                for _ in range(LEARNING_STEPS):
                    for agent_num in range(num_agents):
                        samples = buffer.sample(BATCH_SIZE) # sample the buffer
                        agent.update(samples, agent_num)   # update the agent
                    agent.update_targets() # soft update the target network␣
↪towards the actual network

            if i_episode > episodes_before_training:
                step += 1

        for agent_num in range(num_agents):
            scores_agent[agent_num].append(score[agent_num])
            scores_deque[agent_num].append(score[agent_num])
            rolling_average[agent_num].append(np.mean(scores_deque[agent_num]))

        max_scores = np.max(score)
        max_scores_deque.append(max_scores)
        max_scores_list.append(max_scores)
        average_score = np.mean(max_scores_deque)

        if i_episode > episodes_before_training and step >␣
↪steps_before_noise_reduction:
            noise = max(noise * noise_reduction, noise_reduction_end)

        if i_episode > episodes_before_training and i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.4f}\tSteps: {}'.
↪format(i_episode, average_score,step))
        if i_episode <= episodes_before_training:
            print("\rSave exprieriences from random play", end='')
            if i_episode == episodes_before_training:
                print("\t....Done")


        # We can break if the max between the average scores is >= 0.5
        if np.max(rolling_average) >= 0.5:
            agent.save(agent_ckp_prefix, critic_ckp_prefix)
```

```
            print('\rEnvironment solved Episode {}\tAverage Score: {:.4f}'.
↪format(i_episode, np.mean(scores_deque)))
            break

    return scores_agent, rolling_average
```

[5]:
```
from multi_agents.maddpg import MADDPG


agent = MADDPG(state_size, action_size, random_seed = 33, num_agents=num_agents)
print("Agent created.")
```

```
Agent created.
```

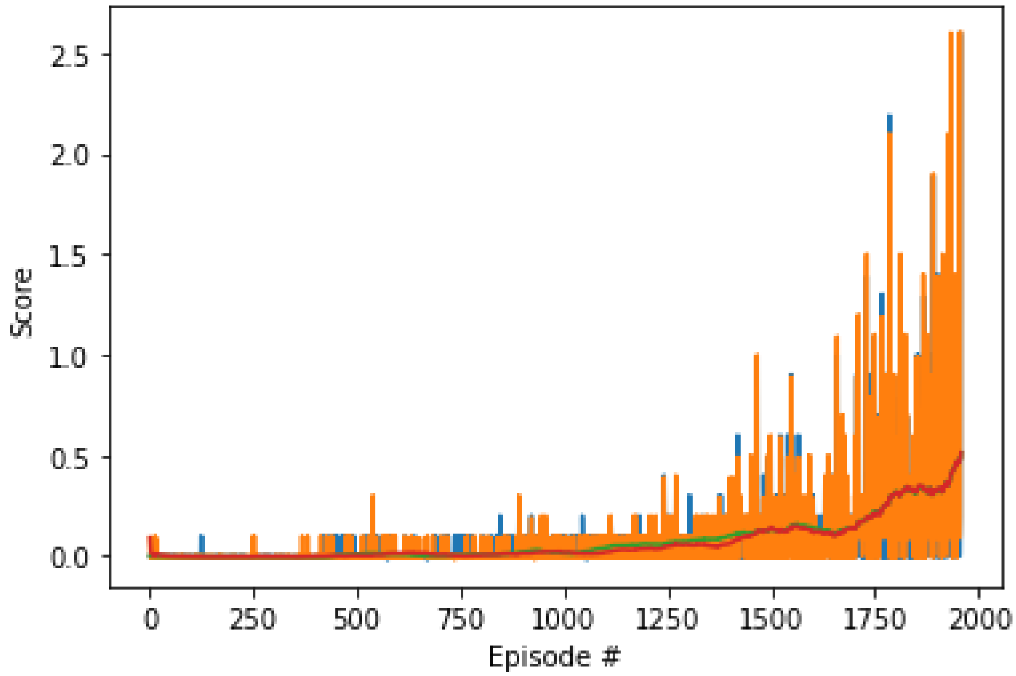
[6]:
```
agent_scores, agent_scores_avg = run_experiment(agent)
```

```
Save expreriences from random play      …Done
Episode 100     Average Score: 0.0028   Steps: 1225
Episode 200     Average Score: 0.0010   Steps: 2559
Episode 300     Average Score: 0.0020   Steps: 3919
Episode 400     Average Score: 0.0020   Steps: 5276
Episode 500     Average Score: 0.0189   Steps: 7088
Episode 600     Average Score: 0.0239   Steps: 9079
Episode 700     Average Score: 0.0125   Steps: 10717
Episode 800     Average Score: 0.0195   Steps: 12500
Episode 900     Average Score: 0.0315   Steps: 14646
Episode 1000    Average Score: 0.0315   Steps: 16800
Episode 1100    Average Score: 0.0627   Steps: 19302
Episode 1200    Average Score: 0.0808   Steps: 22612
Episode 1300    Average Score: 0.1018   Steps: 26562
Episode 1400    Average Score: 0.1198   Steps: 31268
Episode 1500    Average Score: 0.1648   Steps: 38094
Episode 1600    Average Score: 0.1634   Steps: 44548
Episode 1700    Average Score: 0.1751   Steps: 51296
Episode 1800    Average Score: 0.3453   Steps: 64982
Episode 1900    Average Score: 0.3454   Steps: 78690
Environment solved Episode 1958 Average Score: 0.5126
```

[7]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(1, len(agent_scores[0])+1), agent_scores[0])
plt.plot(np.arange(1, len(agent_scores[1])+1), agent_scores[1])
plt.plot(np.arange(1, len(agent_scores_avg[0])+1), agent_scores_avg[0])
plt.plot(np.arange(1, len(agent_scores_avg[1])+1), agent_scores_avg[1])
```

```
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



[8]:
```
env.close()
```

1.1 Play with the agent

[ ]:
```
agent = MADDPG(state_size, action_size, random_seed = 33, num_agents=num_agents)
agent.load(agent_ckp_prefix, critic_ckp_prefix)
```