Report

September 17, 2022

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.

[ ]:
```
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")
```

[ ]:
```
env = UnityEnvironment(file_name='Reacher.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.

[ ]:
```
# 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.

[ ]:
```
# 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: 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  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
  5.55726624e+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!

[ ]:
```python
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)))
```

When finished, you can close the environment.

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

1.0.4 4. It's Your Turn!

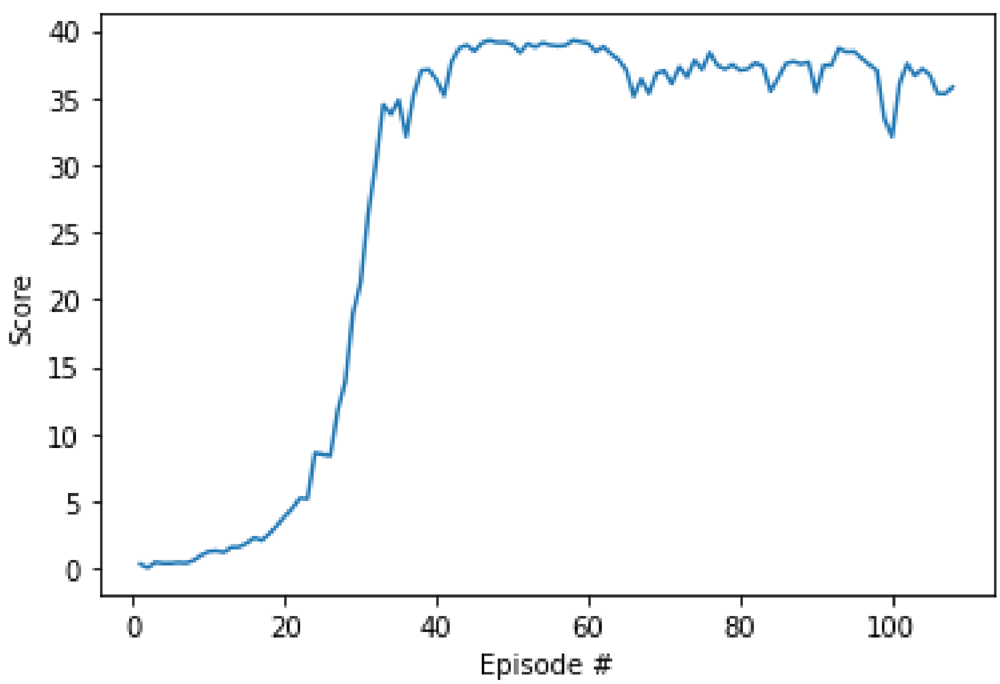
Now it's your turn to train your own agent to solve the environment! When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

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

```
[ ]: from collections import deque
from ddpg_agent import Agent
import torch
import matplotlib.pyplot as plt
%matplotlib inline

agent = Agent(state_size=state_size, action_size=action_size, agent_size =␣
 ↪num_agents, random_seed=0)

def ddpg(n_episodes=200, print_every=100):
    scores_deque = deque(maxlen=print_every)
    scores = []
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        states = env_info.vector_observations
        agent.reset()
        score = np.zeros(num_agents)
        # for t in range(2):
        while True:
            actions = agent.act(states)
            # print(actions)
            env_info = env.step(actions)[brain_name]
            next_states = env_info.vector_observations
            rewards = env_info.rewards
            dones = env_info.local_done
            agent.step(states, actions, rewards, next_states, dones)
            states = next_states
            score += rewards
            if np.any(dones):
                break

        scores_deque.append(np.mean(score))
        scores.append(np.mean(score))
```

```
        print('\rEpisode {}\tAverage Score: {:.2f}\tCurrent Score: {:.2f}'.
↪format(i_episode, np.mean(scores_deque), np.mean(score)), end="")
        if i_episode % print_every == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_deque)))
        if np.mean(scores_deque)>=30.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode, np.mean(scores_deque)))
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
            break
    return scores

scores = ddpg()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(1, len(scores)+1), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

```
Episode 100     Average Score: 27.27    Current Score: 32.13
Episode 108     Average Score: 30.14    Current Score: 35.81
Environment solved in 108 episodes!     Average Score: 30.14
```



[ ]:

```python
env_info = env.reset(train_mode=False)[brain_name]     # reset the environment ␣
↪

agent = Agent(state_size=state_size, action_size=action_size, agent_size =␣
↪num_agents, random_seed=0)
agent.actor_local.load_state_dict(torch.load('checkpoint_actor.pth'))
agent.critic_local.load_state_dict(torch.load('checkpoint_critic.pth'))

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 = agent.act(states)
    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: 33.071999260783194
```