Continuous_Control

February 25, 2021

1 Continuous Control

You are welcome to use this coding environment to train your agent for the project. Follow the instructions below to get started!

1.0.1 1. Start the Environment

Run the next code cell to install a few packages. This line will take a few minutes to run!

```
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.
```

The environments corresponding to both versions of the environment are already saved in the Workspace and can be accessed at the file paths provided below.

Please select one of the two options below for loading the environment.

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

        # select this option to load version 1 (with a single agent) of the environment
        #env = UnityEnvironment(file_name='/data/Reacher_One_Linux_NoVis/Reacher_One_Linux_NoVis

        # select this option to load version 2 (with 20 agents) of the environment
        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: , , ,
```

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 [3]: # 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

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

```
In [4]: # 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]
```

1.0.3 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.

Note that **in this coding environment, you will not be able to watch the agents while they are training**, and you should set `train_mode=True` to restart the environment.

```
In [5]: env_info = env.reset(train_mode=True)[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)))
```

```
Total score (averaged over agents) this episode: 0.23249999480322003
```

```
In [6]: #hack for forcing to reload the used .py files
        %load_ext autoreload
        %autoreload 2
```

```
In [7]: import random
        import torch
        import numpy as np
        from collections import deque
        import matplotlib.pyplot as plt
        %matplotlib inline

        %load_ext autoreload
        %autoreload 1

        from ddpg_agent import Agent
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

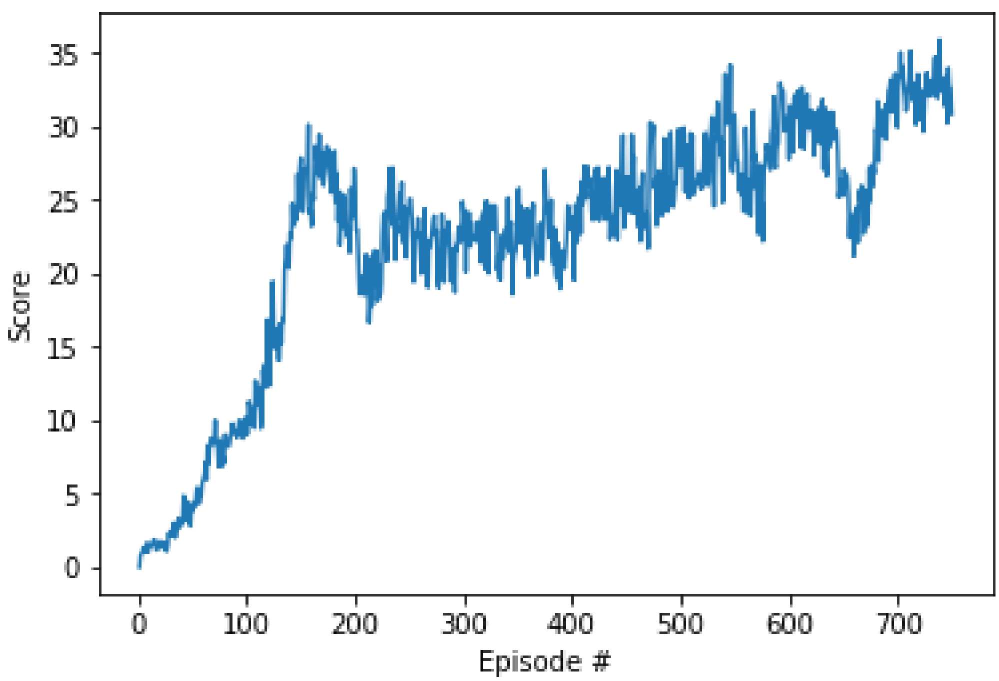
```
In [8]: agent = Agent(state_size=states.shape[1], action_size=brain.vector_action_space_size, ra
```

When finished, you can close the environment.

```
In [9]: from workspace_utils import active_session
        with active_session():
            #using the provided ddpq model from earlier, minimal changes only
            def ddpg(n_episodes=10000, max_t=1000, 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(max_t):
                        actions = agent.act(states)

                        env_info = env.step(actions)[brain_name]
                        next_states = env_info.vector_observations         # get next state (for
                        rewards = env_info.rewards                         # get reward (for eac
                        dones = env_info.local_done                        # see if episode fini

                        agent.step(states, actions, rewards, next_states, dones, t)
                        states = next_states
                        score += rewards
                        if any(dones):
                            break
                    #storing here the mean, and at the print the mean of the means :)
                    scores_deque.append(np.mean(score))
                    scores.append(np.mean(score))

                    print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores

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

```
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: 5.00
Episode 200        Average Score: 21.19
Episode 300        Average Score: 21.98
Episode 400        Average Score: 22.51
Episode 500        Average Score: 25.49
Episode 600        Average Score: 28.08
Episode 700        Average Score: 28.54
Episode 750        Average Score: 30.05
Environment solved in 650 episodes!        Average Score: 30.05
```



1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! A few **important notes**: - 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]
```

- To structure your work, you're welcome to work directly in this Jupyter notebook, or you might like to start over with a new file! You can see the list of files in the workspace by clicking on ***Jupyter*** in the top left corner of the notebook.
- In this coding environment, you will not be able to watch the agents while they are training. However, ***after training the agents***, you can download the saved model weights to watch the agents on your own machine!

In [10]:
```
#env.close()
```

In [ ]: