Continuous_Control

November 19, 2019

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 2.0.
```

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: 1
Size of each action: 4
There are 1 agents. Each observes a state with length: 33
```

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: 0.41999999061226845
```

When finished, you can close the environment.

```
In [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! 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!

1.1 4.1 Importing libraries

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

        import numpy as np
        import random
        import time
```

```python
import torch
# import utilities to keep workspaces alive during model training
from workspace_utils import active_session
from collections import deque
from ddpg_agent import Agent
from unityagents import UnityEnvironment
```

1.2 4.2 Setting parameters

BUFFER_SIZE = int(3e5) #replay buffer size
BATCH_SIZE = 128 #minibatch size GAMMA = 0.99 #discount factor
TAU = 1e-3 #for soft update of target parameters LR_ACTOR = 2e-4 #learning rate of the actor
LR_CRITIC = 2e-4 #learning rate of the critic WEIGHT_DECAY = 0 #L2 weight decay

Suggested on slack: LEARN_EVERY = 20 # learning timestep interval LEARN_NUM = 10 # number of learning passes GRAD_CLIPPING = 1.0 # Gradient Clipping

Ornstein-Uhlenbeck noise parameters OU_SIGMA = 0.02 OU_THETA = 0.1
EPSILON = 1.0 # for epsilon in the noise process (act step) EPSILON_DECAY = 1e-6

1.3 5.0 Training loop

class Actor(nn.Module): """Actor (Policy) Model."""

```
def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
    """Initialize parameters and build model.
    Params
    ======
        state_size (int): Dimension of each state
        action_size (int): Dimension of each action
        seed (int): Random seed
        fc1_units (int): Number of nodes in first hidden layer
        fc2_units (int): Number of nodes in second hidden layer
    """
    super(Actor, self).__init__()
    self.seed = torch.manual_seed(seed)
    self.fc1 = nn.Linear(state_size, fc1_units)
    self.bn1 = nn.BatchNorm1d(fc1_units)
    self.fc2 = nn.Linear(fc1_units, fc2_units)
    self.fc3 = nn.Linear(fc2_units, action_size)
    self.reset_parameters()

def reset_parameters(self):
    self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
    self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
    self.fc3.weight.data.uniform_(-3e-3, 3e-3)

def forward(self, state):
```

```
    """Build an actor (policy) network that maps states -> actions."""
    x = F.relu(self.bn1(self.fc1(state)))
    x = F.relu(self.fc2(x))
    return torch.tanh(self.fc3(x))
```

class Critic(nn.Module): """Critic (Value) Model."""

```
def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
    """Initialize parameters and build model.
    Params
    ======
        state_size (int): Dimension of each state
        action_size (int): Dimension of each action
        seed (int): Random seed
        fcs1_units (int): Number of nodes in the first hidden layer
        fc2_units (int): Number of nodes in the second hidden layer
    """
    super(Critic, self).__init__()
    self.seed = torch.manual_seed(seed)
    self.fc1 = nn.Linear(state_size, fc1_units)
    self.bn1 = nn.BatchNorm1d(fc1_units)
    self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)
    self.fc3 = nn.Linear(fc2_units, 1)
    self.reset_parameters()

def reset_parameters(self):
    self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
    self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
    self.fc3.weight.data.uniform_(-3e-3, 3e-3)

def forward(self, state, action):
    """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
    xs = F.relu(self.bn1(self.fc1(state)))
    x = torch.cat((xs, action), dim=1)
    x = F.relu(self.fc2(x))
    return self.fc3(x)
```

In [8]:
```
# DDPG function

def ddpg(n_episodes=2500, max_t=1000, print_every=10):
    """Deep Deterministic Policy Gradient (DDPG)

    Params
    ======
        n_episodes  (int)   : maximum number of training episodes
        max_t       (int)   : maximum number of timesteps per episode
        print_every (int)   : interval to display results
```

```
    """
    mean_scores = []                               # list of mean scores from each episo
    moving_avgs = []                               # list of moving averages
    best_score  = -np.inf
    scores_window = deque(maxlen=100)              # mean scores from most recent 100 ep

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]       # reset environment
        states = env_info.vector_observations                   # get current state for
        scores = np.zeros(num_agents)                           # initialize score for e
        agent.reset()
        start_time = time.time()
        for t in range(max_t):
            actions = agent.act(states, add_noise=True)         # select an action
            env_info = env.step(actions)[brain_name]            # send actions to enviro
            next_states = env_info.vector_observations          # get next state
            rewards = env_info.rewards                          # get reward
            dones = env_info.local_done                         # see if episode has fin
            # save experience to replay buffer, perform learning step at defined interva
            for state, action, reward, next_state, done in zip(states, actions, rewards,
                agent.step(state, action, reward, next_state, done, t)
            states = next_states
            scores += rewards
            if np.any(dones):                                   # exit loop when episode
                break

        duration = time.time() - start_time
        mean_scores.append(np.mean(scores))           # save mean score for the episode
        scores_window.append(mean_scores[-1])         # save mean score to window
        moving_avgs.append(np.mean(scores_window))    # save moving average

        if i_episode % print_every == 0:
            print('\rEpisode {} ({}s)\tMean: {:.1f}\tMoving Avg: {:.1f}'.format(\
                  i_episode, round(duration), mean_scores[-1], moving_avgs[-1]))

        if moving_avgs[-1] >= 30.00 and i_episode >= 100:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format
            break

    return mean_scores, moving_avgs
```

```
In [9]: start = time.time()
```

1.4 4.9 Instanstiating class Agent

random_seed=15 Episode 1 (15s) Mean: 1.5 Moving Avg: 1.5 Episode 2 (18s) Mean: 0.9 Moving Avg: 1.2 Episode 3 (17s) Mean: 0.8 Moving Avg: 1.0 Episode 4 (17s) Mean: 0.2 Moving Avg: 0.8 Episode 5 (17s) Mean: 0.0 Moving Avg: 0.7 Episode 6 (17s) Mean: 0.6 Moving Avg: 0.7 Episode 7

(17s) Mean: 0.6 Moving Avg: 0.7 Episode 8 (18s) Mean: 1.5 Moving Avg: 0.8 Episode 9 (18s) Mean: 0.4 Moving Avg: 0.7 Episode 10 (18s) Mean: 2.1 Moving Avg: 0.9 Episode 11 (18s) Mean: 0.5 Moving Avg: 0.8 Episode 12 (18s) Mean: 0.1 Moving Avg: 0.8 Episode 13 (18s) Mean: 1.0 Moving Avg: 0.8 Episode 14 (18s) Mean: 1.6 Moving Avg: 0.8 Episode 15 (18s) Mean: 1.1 Moving Avg: 0.9

```
In [10]: # run the training loop
         agent = Agent(state_size=state_size, action_size=action_size, random_seed=5)
         # this is a Workspaces-specific context manager to keep the connection
         # alive while training your model, not part of pytorch
         with active_session():
             scores, avgs = ddpg()
```

```
Episode 10 (20s)        Mean: 0.7        Moving Avg: 0.3
Episode 20 (20s)        Mean: 1.1        Moving Avg: 0.4
Episode 30 (20s)        Mean: 1.9        Moving Avg: 0.7
Episode 40 (20s)        Mean: 1.7        Moving Avg: 1.0
Episode 50 (20s)        Mean: 3.1        Moving Avg: 1.3
Episode 60 (20s)        Mean: 3.4        Moving Avg: 1.8
Episode 70 (20s)        Mean: 3.9        Moving Avg: 2.2
Episode 80 (20s)        Mean: 2.5        Moving Avg: 2.4
Episode 90 (20s)        Mean: 13.0        Moving Avg: 2.7
Episode 100 (20s)        Mean: 6.6        Moving Avg: 3.0
Episode 110 (20s)        Mean: 8.2        Moving Avg: 3.6
Episode 120 (21s)        Mean: 5.7        Moving Avg: 4.2
Episode 130 (21s)        Mean: 11.2        Moving Avg: 4.9
Episode 140 (21s)        Mean: 13.7        Moving Avg: 5.7
Episode 150 (21s)        Mean: 10.1        Moving Avg: 6.4
Episode 160 (21s)        Mean: 11.2        Moving Avg: 7.0
Episode 170 (21s)        Mean: 12.1        Moving Avg: 7.9
Episode 180 (21s)        Mean: 15.1        Moving Avg: 8.6
Episode 190 (21s)        Mean: 18.7        Moving Avg: 9.7
Episode 200 (21s)        Mean: 17.2        Moving Avg: 10.7
Episode 210 (21s)        Mean: 25.5        Moving Avg: 12.1
Episode 220 (21s)        Mean: 21.9        Moving Avg: 13.3
Episode 230 (21s)        Mean: 21.7        Moving Avg: 14.9
Episode 240 (21s)        Mean: 20.9        Moving Avg: 16.3
Episode 250 (21s)        Mean: 29.0        Moving Avg: 18.1
Episode 260 (21s)        Mean: 33.8        Moving Avg: 19.9
Episode 270 (21s)        Mean: 25.0        Moving Avg: 21.9
Episode 280 (22s)        Mean: 35.5        Moving Avg: 24.1
Episode 290 (22s)        Mean: 36.4        Moving Avg: 25.6
Episode 300 (22s)        Mean: 39.5        Moving Avg: 27.4
Episode 310 (22s)        Mean: 36.1        Moving Avg: 28.9

Environment solved in 218 episodes!        Average Score: 30.16
```

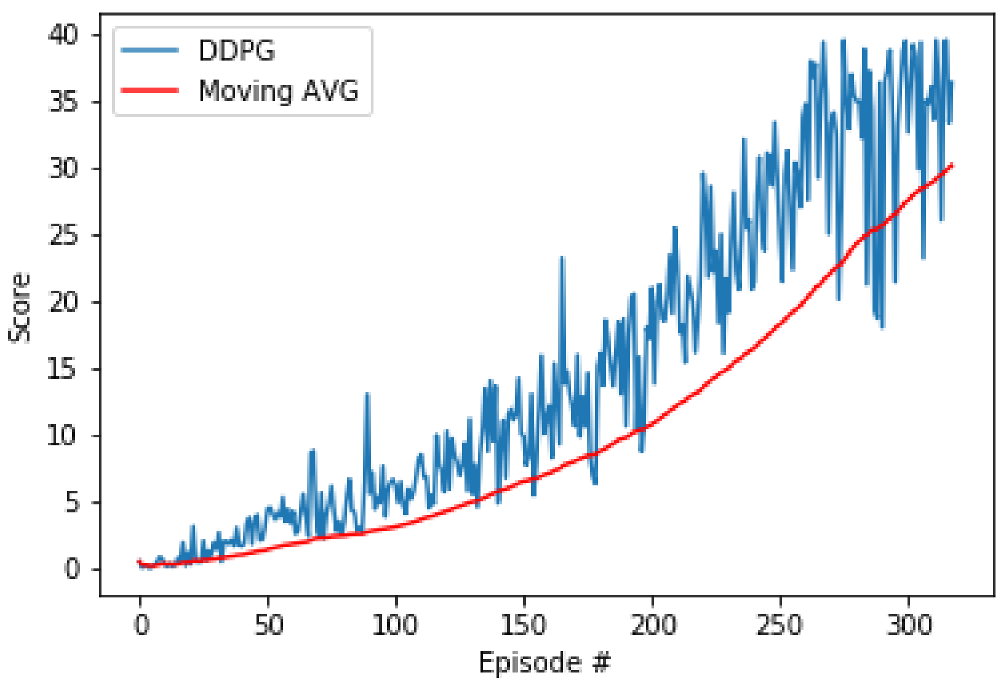
```
In [11]: end = time.time()
```

```
        elapsed = (end - start) / 60.0 # in minutes
        print("\nElapsed Time: {0:3.2f} mins.".format(elapsed))
```

```
Elapsed Time: 110.50 mins.
```

1.5 6. Visualizing rewards

```
In [12]: # plot the scores
        fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.plot(np.arange(len(scores)), scores, label='DDPG')
        plt.plot(np.arange(len(scores)), avgs, c='r', label='Moving AVG')
        plt.ylabel('Score')
        plt.xlabel('Episode #')
        plt.legend(loc='upper left');
        plt.show()
```




```
In [13]: env.close()
```

1.6 Ideas for improving the agent's performance

- Batch Normalization
- Neural network enhancement for a better performance.