pdf

November 6, 2019

1 Collaboration and Competition

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 environment is already saved in the Workspace and can be accessed at the file path provided below.

```
In [2]: from unityagents import UnityEnvironment
        import numpy as np
        from maddpg_agent import Agent
        from collections import deque
        import random
        import time
        import torch
        import matplotlib.pyplot as plt



        env = UnityEnvironment(file_name="/data/Tennis_Linux_NoVis/Tennis")
```

```
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.

```
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: 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.         -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 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]: for i in range(5):                                         # 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
            scores = np.zeros(num_agents)                          # initialize the score (for e
            while True:
                actions = np.random.randn(num_agents, action_size) # select an action (for each
                actions = np.clip(actions, -1, 1)                  # all actions between -1 and
                env_info = env.step(actions)[brain_name]           # send all actions to tne env
                next_states = env_info.vector_observations         # get next state (for each ag
                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
                states = next_states                               # roll over states to next ti
                if np.any(dones):                                  # exit loop if episode finish
                    break
            print('Total score (averaged over agents) this episode: {}'.format(np.mean(scores)))
```

```
Total score (averaged over agents) this episode: 0.04500000085681677
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
```

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 [11]: # CONSTANTS

         GOAL_AVG_SCORE = 0.5
```

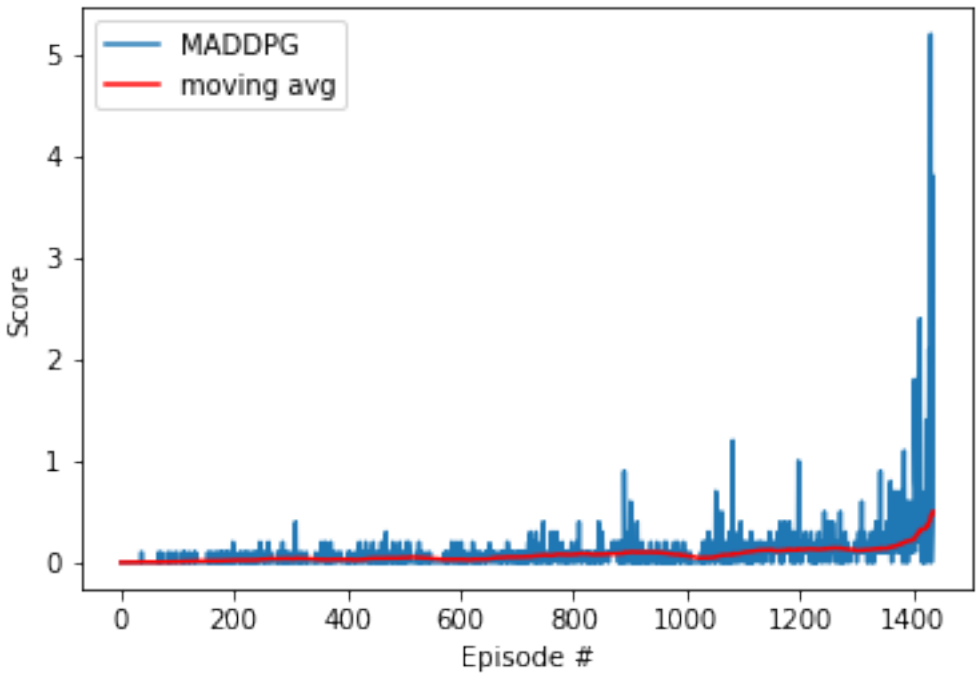
```
CONSEC_EPISODES = 100
PRINT_EVERY = 100
ADD_NOISE = True
STOP_FLAG = 300
N_EPISODES = 4000
MAX_T = 2000
TRAIN_MODE = True


# MADDPG function

def maddpg(n_episodes= N_EPISODES, max_t= MAX_T, train_mode= TRAIN_MODE):
    """Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

    Params
    ======
        n_episodes (int)      : maximum number of training episodes
        max_t (int)           : maximum number of timesteps per episode
        train_mode (bool)     : if 'True' set environment to training mode

    """
    scores_window = deque(maxlen=CONSEC_EPISODES)
    scores_all = []
    moving_average = []
    best_score = -np.inf
    best_episode = 0
    already_solved = False

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=train_mode)[brain_name]         # reset the env
        states = np.reshape(env_info.vector_observations, (1,48)) # get states and comb
        agent_0.reset()
        agent_1.reset()
        scores = np.zeros(num_agents)
        while True:
            actions = get_actions(states, ADD_NOISE)           # choose agent actions a
            env_info = env.step(actions)[brain_name]           # send both agents' acti
            next_states = np.reshape(env_info.vector_observations, (1, 48)) # combine t
            rewards = env_info.rewards                         # get reward
            done = env_info.local_done                         # see if episode finishe
            agent_0.step(states, actions, rewards[0], next_states, done, 0) # agent 1 l
            agent_1.step(states, actions, rewards[1], next_states, done, 1) # agent 2 l
            scores += np.max(rewards)                          # update the score for e
            states = next_states                               # roll over states to ne
            if np.any(done):                                   # exit loop if episode f
                break

        ep_best_score = np.max(scores)
```

```
                scores_window.append(ep_best_score)
                scores_all.append(ep_best_score)
                moving_average.append(np.mean(scores_window))

                # save best score
                if ep_best_score > best_score:
                    best_score = ep_best_score
                    best_episode = i_episode

                # print the results
                if i_episode % PRINT_EVERY == 0:
                    print('Episodes {:0>4d}-{:0>4d}\tMax Reward: {:.3f}\tAverage: {:.3f}'.forma
                        i_episode-PRINT_EVERY, i_episode, np.max(scores_all[-PRINT_EVERY:]), mo

                # determine if the env meets the avg score goal
                if moving_average[-1] >= GOAL_AVG_SCORE:
                        print('<-- Environment solved in {:d} episodes! \
                        \n<-- Average: {:.3f} over past {:d} episodes'.format(
                            i_episode-CONSEC_EPISODES, moving_average[-1], CONSEC_EPISODES))
                        # save the weights model
                        torch.save(agent_0.actor_local.state_dict(), 'checkpoint_actor_0.pth')
                        torch.save(agent_0.critic_local.state_dict(), 'checkpoint_critic_0.pth'
                        torch.save(agent_1.actor_local.state_dict(), 'checkpoint_actor_1.pth')
                        torch.save(agent_1.critic_local.state_dict(), 'checkpoint_critic_1.pth'
                        break
                else:
                        continue

            return scores_all, moving_average

In [12]: def get_actions(states, add_noise):
            '''gets actions for each agent and then combines them into one array'''
            action_0 = agent_0.act(states, add_noise)    # agent 0 chooses an action
            action_1 = agent_1.act(states, add_noise)    # agent 1 chooses an action
            return np.concatenate((action_0, action_1), axis=0).flatten()

        # initialize agents
        agent_0 = Agent(state_size, action_size, num_agents=1, random_seed=0)
        agent_1 = Agent(state_size, action_size, num_agents=1, random_seed=0)

In [13]: # run the training loop
        from workspace_utils import active_session

        with active_session():
            scores, avgs = maddpg()
```

```
Episodes 0000-0100        Max Reward: 0.100        Average: 0.008
Episodes 0100-0200        Max Reward: 0.200        Average: 0.020
```

```
Episodes 0200-0300        Max Reward: 0.200         Average: 0.037
Episodes 0300-0400        Max Reward: 0.400         Average: 0.029
Episodes 0400-0500        Max Reward: 0.300         Average: 0.047
Episodes 0500-0600        Max Reward: 0.200         Average: 0.030
Episodes 0600-0700        Max Reward: 0.200         Average: 0.053
Episodes 0700-0800        Max Reward: 0.400         Average: 0.079
Episodes 0800-0900        Max Reward: 0.900         Average: 0.099
Episodes 0900-1000        Max Reward: 0.600         Average: 0.068
Episodes 1000-1100        Max Reward: 1.200         Average: 0.094
Episodes 1100-1200        Max Reward: 1.000         Average: 0.129
Episodes 1200-1300        Max Reward: 0.500         Average: 0.119
Episodes 1300-1400        Max Reward: 1.100         Average: 0.220
<-- Environment solved in 1337 episodes!
<-- Average: 0.501 over past 100 episodes
```

In [14]:
```
# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='MADDPG')
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()
```



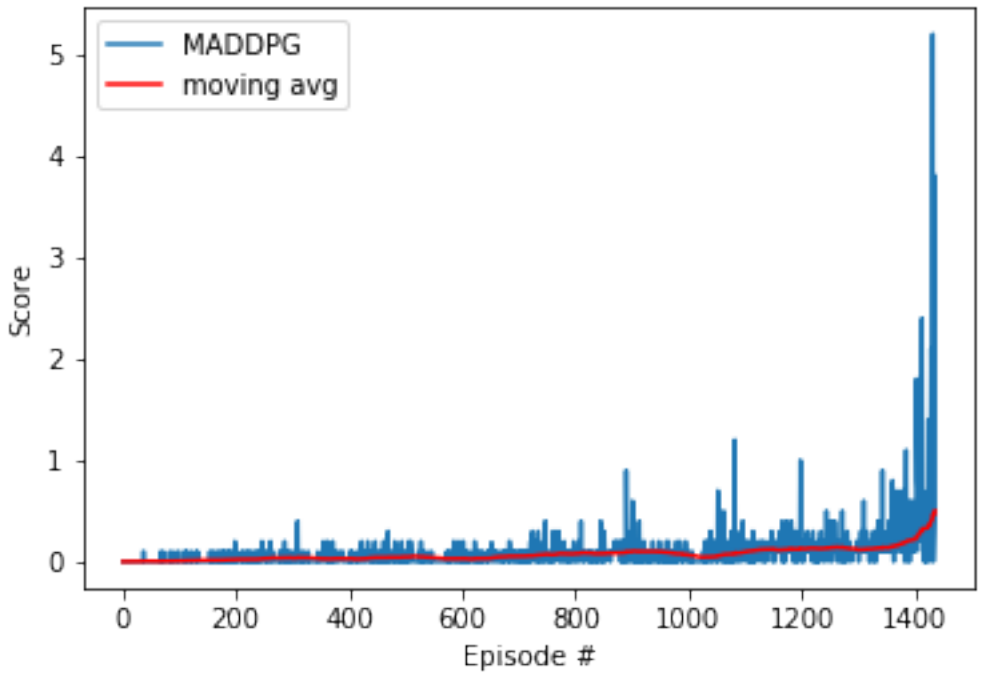
2 4. Testing a trained agent

In [15]:
```
EPISODES = 100
MAX_T = 2000
PRINT_EVERY = 10
ADD_NOISE = False
TRAIN_MODE = False

## itialize the agents
agent_0 = Agent(state_size, action_size, num_agents=1, random_seed=0)
agent_1 = Agent(state_size, action_size, num_agents=1, random_seed=0)

# load the weights from the saved checkpoints
agent_0_weights = 'checkpoint_actor_0.pth'
agent_1_weights = 'checkpoint_actor_1.pth'
agent_0.actor_local.load_state_dict(torch.load(agent_0_weights))
agent_1.actor_local.load_state_dict(torch.load(agent_1_weights))
```

In [16]:
```
def test(n_episodes= EPISODES , max_t= MAX_T, train_mode= TRAIN_MODE):

    scores_window = deque(maxlen=EPISODES)
    scores_all = []
    moving_average = []

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=train_mode)[brain_name]          # reset the env
        states = np.reshape(env_info.vector_observations, (1,48)) # get states and comb
        scores = np.zeros(num_agents)
        while True:
            actions = get_actions(states, ADD_NOISE)           # choose agent actions a
            env_info = env.step(actions)[brain_name]           # send both agents' acti
            next_states = np.reshape(env_info.vector_observations, (1, 48)) # combine t
            rewards = env_info.rewards                         # get reward
            done = env_info.local_done                         # see if episode finishe
            scores += np.max(rewards)                          # update the score for e
            states = next_states                               # roll over states to ne
            if np.any(done):                                   # exit loop if episode f
                break

        ep_best_score = np.max(scores)
        scores_window.append(ep_best_score)
        scores_all.append(ep_best_score)
        moving_average.append(np.mean(scores_window))

        # print results
        if i_episode % PRINT_EVERY == 0:
            print('Episodes {:0>4d}-{:0>4d}\tMax Reward: {:.3f}\tAverage: {:.3f}'.forma
                i_episode-PRINT_EVERY, i_episode, np.max(scores_all[-PRINT_EVERY:]), mo
```

```
            return scores_all, moving_average
```

In [17]:
```
# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='MADDPG')
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 [18]:
```
env.close()
```

In [ ]: