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 (https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893)](https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893) program.

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 (https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation.md)](https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation.md) and [NumPy (http://www.numpy.org/)](http://www.numpy.org/).

In [1]:
```
from unityagents import UnityEnvironment
import numpy as np
```

In [2]:
```
import random
import torch
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline

from ddpg_agent import Agent
```

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")
```

In [3]:
```
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.

In [4]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

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.

In [5]:
```
# 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(st
ates.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.        ]
```

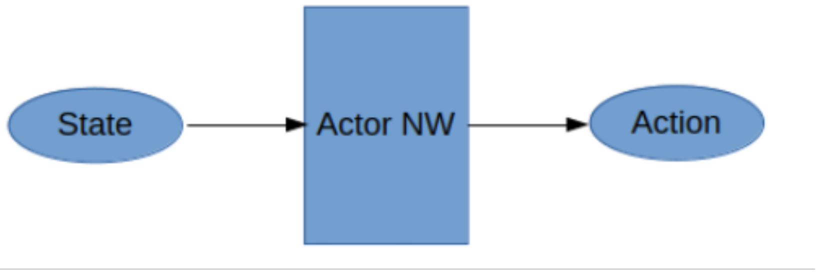
3. Implement DDPG

DDPG

Here in this report, Deep Deterministic Policy Gradients (DDPG) algorithm is used for the arms to continuously touch to the target. DDPG is one of the policy gradient method to learn deterministic function to decide the agent's behavior. Following Actor and Critic improves each, so in the end, Actor Network to output values that improves value out of Critic Network.

Actor Neural Network, each connected with ReLU, except final output, which is tanh.

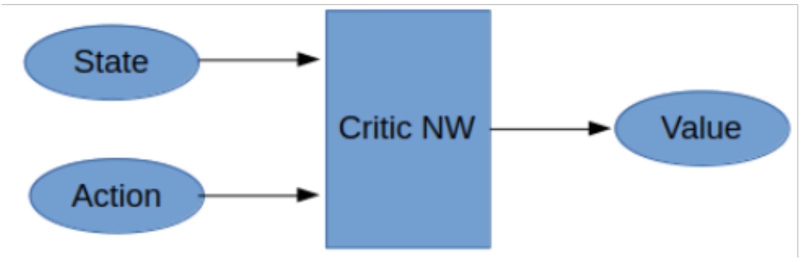
```
self.fc1 = nn.Linear(state_size, fc1_units)
self.fc2 = nn.Linear(fc1_units, fc2_units)
self.fc3 = nn.Linear(fc2_units, action_size)
```



Critic Neural Network, each connected with ReLU

Critic network is uses Q-Network, and introduced the following network for optimization.

```
self.fcs1 = nn.Linear(state_size, fcs1_units)
self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
self.fc3 = nn.Linear(fc2_units, 1)
```



cf.

- [https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html (https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html)](https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html)
- [https://arxiv.org/abs/1509.02971 (https://arxiv.org/abs/1509.02971)](https://arxiv.org/abs/1509.02971)

In [7]:
```
def moving_average(a, n=3) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n
```

In [8]:
```
def ddpg(agent, n_episodes=100, max_t=10000, print_every=100):
    scores_deque = deque(maxlen=print_every)
    average_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()
        scores = np.zeros(num_agents)
        moving_avgs = []                              # list of moving aver
ages

        for t in range(max_t):
            actions = agent.act(states)
            env_info = env.step(actions)[brain_name]        # send the actio
n to the environment
            next_states = env_info.vector_observations   # get the next stat
e
            rewards = env_info.rewards                   # get the reward
            dones = env_info.local_done

            for state, action, reward, next_state, done in zip(states, actio
ns, rewards, next_states, dones):
                agent.step(state, action, reward, next_state, done)
            states = next_states
            scores += rewards
            if np.any(dones):                                   # exit loop
when episode ends
                break

        average_scores.append(np.mean(scores))

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, averag
e_scores[-1]), end="")
        torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
        torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
        if i_episode % print_every == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, av
erage_scores[-1]))
            print('Moving Average Score: {}'.format(moving_average(average_s
cores, n=print_every)[-1]))

    return average_scores
```

In [9]:
```
def plotscores(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()
    ma = moving_average(scores, n=100)
    plt.plot(np.arange(1, len(ma)+1), ma)
    plt.ylabel('Moving Average Score')
    plt.xlabel('Episode #')
    plt.show()
```

4. 'Nice' hyper parameters value

Following is selected hyperparameter values to be adjuested in this report.

Hyper Parameter	Description
_BUFFER_SIZE	replay buffer size
_BATCH_SIZE	minibatch size
_GAMMA	discount factor
_TAU	for soft update of target parameters
_LR_ACTOR	learning rate of the actor
_LR_CRITIC	learning rate of the critic
_WEIGHT_DECAY	L2 weight decay
_mu	Ornstein-Uhlenbeck process
_theta	Ornstein-Uhlenbeck process
_sigma	Ornstein-Uhlenbeck process
_actor_fc1_units	Actor Layer 1 units
_actor_fc2_units	Actor Layer 2 units
_critic_fc1_units	Critic Layer 1 units
_critic_fc2_units	Critic Layer 2 units

After a several tries, following values are selected. Not that GAMMA need to be larger. When the value 0.90 is selected, the performance was not good. This can be estimated that this model needs longer history to get a proper value for a better action/policy.

Hyperparameters

Following is the values set to the hyperparameters in the end.

In [10]:
```
_BUFFER_SIZE = int(1e6)  # replay buffer size
_BATCH_SIZE = 256        # minibatch size
_GAMMA = 0.995            # discount factor
_TAU = 1e-3              # for soft update of target parameters
_LR_ACTOR = 1e-4         # learning rate of the actor
_LR_CRITIC = 1e-4        # learning rate of the critic
_WEIGHT_DECAY = 0        # L2 weight decay
_mu=0.                   # Ornstein-Uhlenbeck noise parameters
_theta=0.15              # Ornstein-Uhlenbeck noise parameters
_sigma=0.1               # Ornstein-Uhlenbeck noise parameters
_actor_fc1_units=64
_actor_fc2_units=32
_critic_fc1_units=64
_critic_fc2_units=32
```

5. Run the DDPG agent

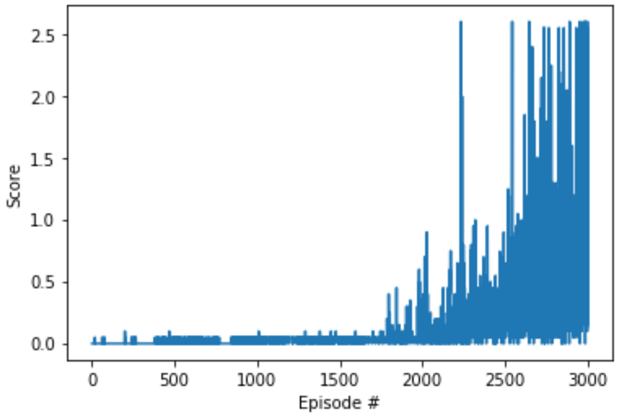
After running the DDPG agent with the Prioritized Experience Replay, the environment is considered solved. The moving average of consequtive 100 episodes achived more than +0.5 scores, as can be seen in the graph.

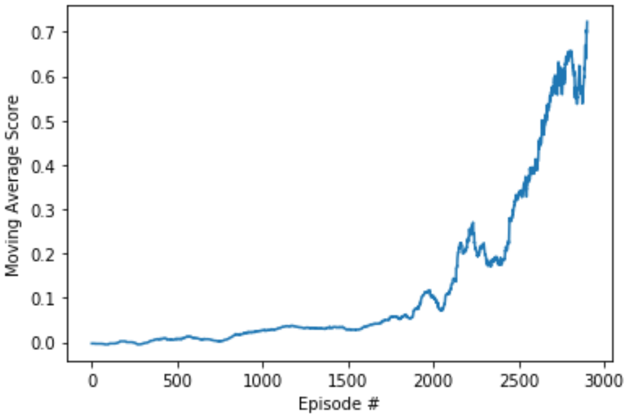
In [11]:
```
agent = Agent(state_size=state_size, action_size=action_size, random_seed=2,
    BUFFER_SIZE = _BUFFER_SIZE,  # replay buffer size
    BATCH_SIZE = _BATCH_SIZE,        # minibatch size
    GAMMA = _GAMMA,            # discount factor
    TAU = _TAU,              # for soft update of target parameters
    LR_ACTOR = _LR_ACTOR,         # learning rate of the actor
    LR_CRITIC = _LR_CRITIC,        # learning rate of the critic
    WEIGHT_DECAY = _WEIGHT_DECAY,       # L2 weight decay
    mu = _mu,
    theta= _theta,
    sigma= _sigma)
```

In [11]:
```
scores = ddpg(agent, 3000)
```

```
Episode 100     Average Score: -0.00
Moving Average Score: -0.0029999998584389685
Episode 200     Average Score: 0.100
Moving Average Score: -0.0034999998658895495
Episode 300     Average Score: -0.00
Moving Average Score: 0.0005000001937150955
Episode 400     Average Score: -0.00
Moving Average Score: -0.0029999998584389685
Episode 500     Average Score: -0.00
Moving Average Score: 0.006500000283122063
Episode 600     Average Score: 0.050
Moving Average Score: 0.007500000298023224
Episode 700     Average Score: -0.00
Moving Average Score: 0.010000000335276127
Episode 800     Average Score: -0.00
Moving Average Score: 0.005000000260770321
Episode 900     Average Score: 0.050
Moving Average Score: 0.008000000305473804
Episode 1000    Average Score: 0.05
Moving Average Score: 0.02050000049173832
Episode 1100    Average Score: 0.050
Moving Average Score: 0.027000000588595866
Episode 1200    Average Score: 0.050
Moving Average Score: 0.03150000065565109
Episode 1300    Average Score: 0.050
Moving Average Score: 0.034500000700354576
Episode 1400    Average Score: 0.050
Moving Average Score: 0.03200000066310167
Episode 1500    Average Score: 0.050
Moving Average Score: 0.033000000678002836
Episode 1600    Average Score: 0.050
Moving Average Score: 0.02900000061839819
Episode 1700    Average Score: 0.050
Moving Average Score: 0.034000000692903994
Episode 1800    Average Score: 0.050
Moving Average Score: 0.04695000088773668
Episode 1900    Average Score: 0.050
Moving Average Score: 0.052500000968575475
Episode 2000    Average Score: 0.050
Moving Average Score: 0.07550000131130219
Episode 2100    Average Score: -0.00
Moving Average Score: 0.10245000171475112
Episode 2200    Average Score: 0.350
Moving Average Score: 0.12100000198930502
Episode 2300    Average Score: 0.300
Moving Average Score: 0.22655000356957317
Episode 2400    Average Score: 0.050
Moving Average Score: 0.205000003259629
Episode 2500    Average Score: 0.100
Moving Average Score: 0.17645000284537674
Episode 2600    Average Score: 1.000
Moving Average Score: 0.3370000052172691
Episode 2700    Average Score: 0.300
Moving Average Score: 0.40605000625364485
Episode 2800    Average Score: 0.350
Moving Average Score: 0.558600008552894
Episode 2900    Average Score: 0.100
Moving Average Score: 0.6471500098425895
Episode 3000    Average Score: 0.150
Moving Average Score: 0.7224500109627843
```

In [12]:
```
plotscores(scores)
```





Upper graphs shows the average score of each eisodes. Lower graph shows the moving average of the recent 100 episodes shown in the upper graph.

With models saved,

- For the Actor Network, **'checkpoint_actor.pth'**
- For the Critic Network, **'checkpoint_critic.pth'**

We can (, but not always though,) get more than +0.5 scores as below.

In [16]:
```
agent.actor_local.load_state_dict(torch.load('checkpoint_actor.pth'))
agent.critic_local.load_state_dict(torch.load('checkpoint_critic.pth'))

env_info = env.reset(train_mode=False)[brain_name]     # reset the environme
nt
states = env_info.vector_observations                  # get the current sta
te (for each agent)
scores = np.zeros(num_agents)                          # initialize the scor
e (for each agent)
while True:
    actions = agent.act(states)                        # select an action (f
or each agent)
    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 eac
h agent)
    dones = env_info.local_done                        # see if episode fini
shed
    scores += env_info.rewards                         # update the score (f
or each agent)
    states = next_states                               # roll over states to
next time step
    if np.any(dones):                                  # exit loop if episod
e finished
        break
print('Total score (averaged over agents) this episode: {}'.format(np.mean(s
cores)))
```

```
Total score (averaged over agents) this episode: 1.045000015757978
```

When finished, you can close the environment.

In [17]:
```
env.close()
```

6. Future improvements

Prioritized Replay Buffer

Prioritized Replay Buffer to accelerate the leaarning. https://github.com/rlcode/per (https://github.com/rlcode/per)

Reward and Policy Gradient method

When played with the saved model, the rackets motion was not stable, but vibrating always. This can be solved by other rewarding factors to the Unity-ML model configuration. The better reward setting and resolution algorithm will need to be investigated.