Navigation

In this notebook, you will learn how to use the Unity ML-Agents environment for the first project of the Deep Reinforcement Learning Nanodegree.

0. Install the dependencies

In [ ]:
```
!pip -q install ./python
```

1. Start the Environment

We begin by importing some necessary packages.

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

import random
import torch
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline
```

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/Banana.app"`
- **Windows** (x86): `"path/to/Banana_Windows_x86/Banana.exe"`
- **Windows** (x86_64): `"path/to/Banana_Windows_x86_64/Banana.exe"`
- **Linux** (x86): `"path/to/Banana_Linux/Banana.x86"`
- **Linux** (x86_64): `"path/to/Banana_Linux/Banana.x86_64"`
- **Linux** (x86, headless): `"path/to/Banana_Linux_NoVis/Banana.x86"`
- **Linux** (x86_64, headless): `"path/to/Banana_Linux_NoVis/Banana.x86_64"`

For instance, if you are using a Mac, then you downloaded `Banana.app`. If this file is in the same folder as the notebook, then the line below should appear as follows:

```
env = UnityEnvironment(file_name="Banana.app")
```

In [2]:
```
env = UnityEnvironment(file_name="Banana_Windows_x86_64/Banana.exe")
```

```
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: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
```

```
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        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]
```

2. Examine the State and Action Spaces

The simulation contains a single agent that navigates a large environment. At each time step, it has four actions at its disposal:

- `0` - walk forward
- `1` - walk backward
- `2` - turn left
- `3` - turn right

The state space has `37` dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. A reward of `+1` is provided for collecting a yellow banana, and a reward of `-1` is provided for collecting a blue banana.

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 in the environment
print('Number of agents:', len(env_info.agents))

# number of actions
action_size = brain.vector_action_space_size
print('Number of actions:', action_size)

# examine the state space
state = env_info.vector_observations[0]
print('States look like:', state)
state_size = len(state)
print('States have length:', state_size)
```

```
Number of agents: 1
Number of actions: 4
States look like: [1.         0.         0.         0.         0.84408134 0.
 0.         1.         0.         0.0748472  0.         1.
 0.         0.         0.25755    1.         0.         0.
 0.         0.74177343 0.         1.         0.         0.
 0.25854847 0.         0.         1.         0.         0.09355672
 0.         1.         0.         0.         0.31969345 0.
 0.        ]
States have length: 37
```

3. Train the Agent

In the next code cell, we will train the Agent to navigate around the environment.

In [5]:
```
from dqn_agent import Agent

agent = Agent(state_size=state_size, action_size=action_size, seed=0)
```

In [6]:
```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action sele
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilo
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    env_passed = False                 # check is average score >= 13
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name] # reset the environment
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps).astype(int)
            env_info = env.step(action)[brain_name]        # send the action to the
            next_state = env_info.vector_observations[0]
            reward = env_info.rewards[0]                   # get the reward
            done = env_info.local_done[0]
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(sc
        if np.mean(scores_window)>=13.0 and not env_passed:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.fo
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint_passing.pth')
            env_passed = True
        if np.mean(scores_window)>=14.0:
            print('\nEnvironment Cracked average score of {:.2f} in {:d} episodes!'.
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint_optimal.pth')
            break
    return scores

scores = dqn()
```

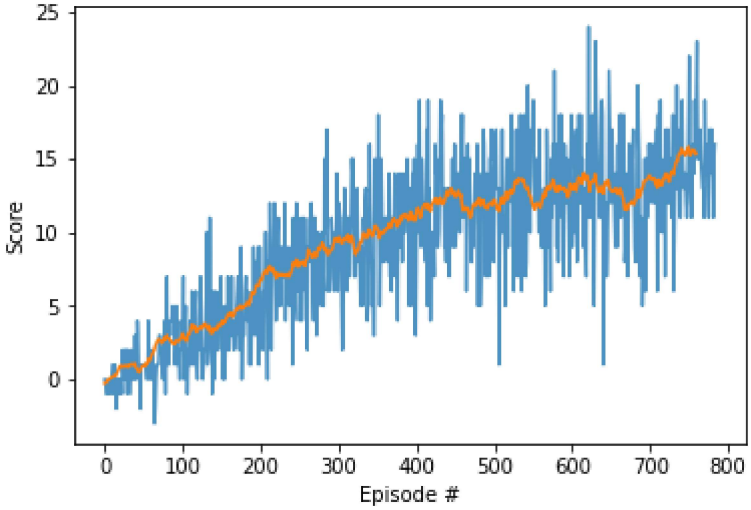
```
Episode 100     Average Score: 1.03
Episode 200     Average Score: 3.84
Episode 300     Average Score: 7.59
Episode 400     Average Score: 9.77
Episode 500     Average Score: 11.98
Episode 600     Average Score: 12.63
Episode 633     Average Score: 13.05
Environment solved in 533 episodes!     Average Score: 13.05
Episode 700     Average Score: 12.79
```

```
Episode 784     Average Score: 14.04
Environment Cracked average score of 14.04 in 684 episodes!
```

In [7]:

```
def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, alpha=0.8)
plt.plot(np.arange(len(moving_average(scores, 25))), moving_average(scores, 25))
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



3. Watch the Agent Perform

In [10]:

```
env_info = env.reset(train_mode=False)[brain_name] # reset the environment
state = env_info.vector_observations[0]            # get the current state
score = 0                                          # initialize the score
while True:
    action = agent.act(state, 0).astype(int)        # select an action
    env_info = env.step(action)[brain_name]        # send the action to the environm
    next_state = env_info.vector_observations[0]   # get the next state
    reward = env_info.rewards[0]                   # get the reward
    done = env_info.local_done[0]                  # see if episode has finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next tim
    if done:                                       # exit loop if episode finished
        break

print("Score: {}".format(score))
```

```
Score: 15.0
```

When finished, you can close the environment.

In [6]:

```
env.close()
```