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

1. Start the Environment

Import of 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
import pandas as pd
%matplotlib inline
```

Start of the environment:

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:\n', 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. 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.

Once this cell is executed, you will watch the agent's performance, if it selects an action (uniformly) at random with each time step. A window should pop up that allows you to observe the agent, as it moves through the environment.

Of course, as part of the project, you'll have to change the code so that the agent is able to use its experience to gradually choose better actions when interacting with the environment!

In [5]:
```
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 = np.random.randint(action_size)        # select an action
    env_info = env.step(action)[brain_name]        # send the action to the environm
ent
    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
e step
    if done:                                       # exit loop if episode finished
        break

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

```
Score: 0.0
```

4. Train the agent

In [6]:
```
from dqn_agent import Agent

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

In [7]:
```
def train(n_episodes=1000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.99
5):
    """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
ction
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilo
n
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps).astype(int)                  # select an a
ction
            env_info = env.step(action)[brain_name]        # send the action to the
environment
            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 fini
shed
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward                                # update the score

            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
_window)), end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(sc
ores_window)))
        if np.mean(scores_window)>=15.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.fo
rmat(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

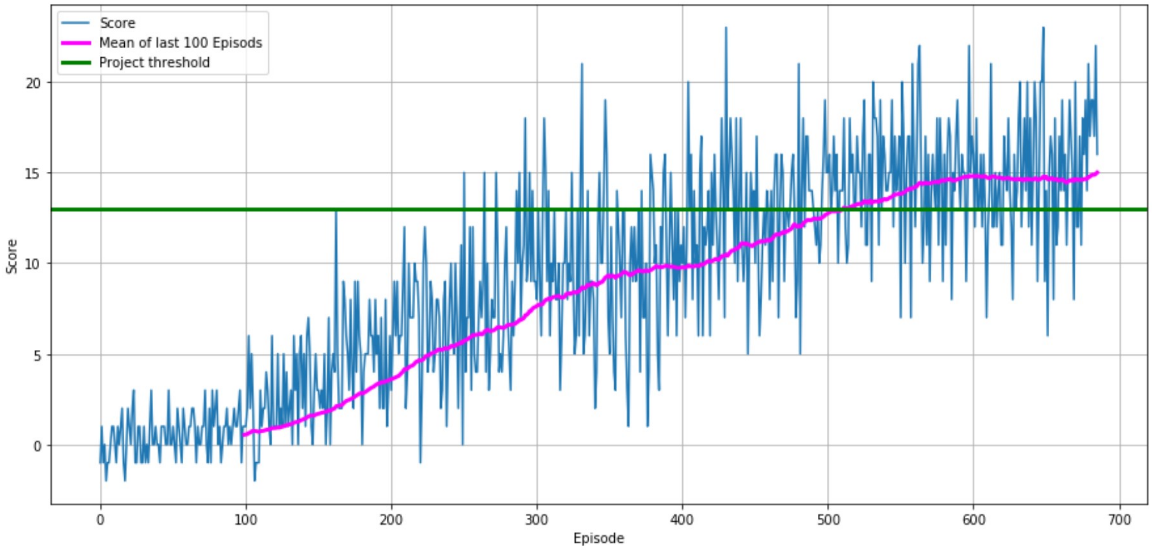
In [8]:
```
scores = train()
```

```
Episode 100	Average Score: 0.54
Episode 200	Average Score: 3.58
Episode 300	Average Score: 7.59
Episode 400	Average Score: 9.76
Episode 500	Average Score: 12.70
Episode 600	Average Score: 14.80
Episode 686	Average Score: 15.00
Environment solved in 586 episodes!	Average Score: 15.00
```

5. Training Plot

In [9]:
```
# plot the scores
fig = plt.figure(figsize=[15, 7])
ax = fig.add_subplot(111)

scores_mean = pd.Series(scores).rolling(100).mean()
plt.plot(np.arange(len(scores)), scores)
plt.plot(scores_mean, "-", c="magenta", linewidth=3)
plt.axhline(13, c="green", linewidth=3)
plt.ylabel('Score')
plt.xlabel('Episode')
plt.grid(which="major")
plt.legend(["Score", "Mean of last 100 Episods", "Project threshold"])
plt.show()
```



6. Watch trained Agent

run the first view cells in the notebook to load the required packages and start the environment. Then load the checkpoint:

In [10]:
```
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))
```

Out[10]: <All keys matched successfully>

Now you may watch the agent for 3 trails - each 100 movements. This should take less than 1 minute. If you'd like to see more or less, just adapte the values for "i" and "j".

In [11]:
```
for i in range (3):
    env_info = env.reset(train_mode=False)[brain_name] # reset the environment
    state = env_info.vector_observations[0]            # get the current state
    for j in range(100):
        action = agent.act(state).astype(int)          # select an action
        env_info = env.step(action)[brain_name]        # send the action to the envi
ronment
        state = env_info.vector_observations[0]        # get the next state
        done = env_info.local_done[0]                  # see if episode has finished
        if done:                                       # exit loop if episode finish
ed
            break
```

When finished you may close the environment.

In [12]:
```
env.close()
```