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

We begin by importing some 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) and NumPy (http://www.numpy.org/).

In [1]:

```
# conda activate drlnd
from unityagents import UnityEnvironment
import numpy as np
np.version.version
```

Out[1]:

```
'1.18.1'
```

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_Linux/Banana.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: 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 double_dqn_agentyagents:', 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 double_dqn_agentyagents: 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]:

```
# Original Not-trained-Brain Code, removed (the almost) endless loop and replaced by a
 200-action-loop
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
for j in range(10):
    action = np.random.randint(action_size)        # select an action
    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 finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next time s
tep
    if done:                                       # exit loop if episode finished
        break

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

```
Score: 0.0
```

4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! 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]
```

In [6]:

```
env_info = env.reset(train_mode=True)[brain_name]
```

In [7]:

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

In [8]:

```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

Out[8]:

```
device(type='cpu')
```

In [9]:

```
def dqn(n_episodes=1000, max_t=10000, eps_start=0.5, eps_end=0.01, eps_decay=0.98, ddqn
=False):
    """Runs 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 selecti
on
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
    scores = []                        # scores from each episode will be stored here
    scores_window = deque(maxlen=100)  # newest scores
    eps = eps_start                    # epsilon init
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=False)[brain_name]
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            #Environment interaction block:
            action = agent.act(state, eps)
            env_info = env.step(action)[brain_name]
            next_state = env_info.vector_observations[0]
            reward = env_info.rewards[0]
            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)
        scores.append(score)
        eps = max(eps_end, eps_decay*eps)
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_wi
ndow)), end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(score
s_window)))
        if np.mean(scores_window)>=13.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
t(i_episode-100, np.mean(scores_window)))
            if ddqn:
                torch.save(agent.qnetwork_local.state_dict(), 'checkpointddqn.pth')
            else:
                torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

In [10]:

```
# Import my own agent
from dqn_agent import Agent
#Create Agent
agent = Agent(state_size=37, action_size=4, seed=0, ddqn=False)
```

In [11]:

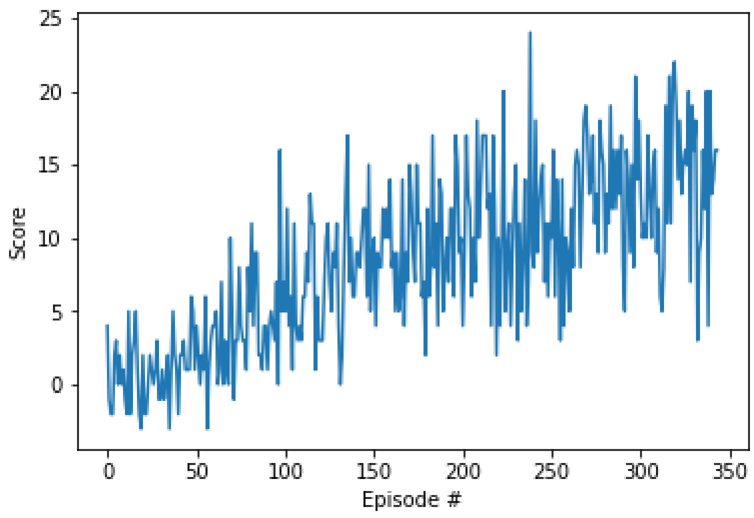
```
#Train my agent
scores = dqn()
```

```
Episode 100     Average Score: 2.35
Episode 200     Average Score: 8.41
Episode 300     Average Score: 11.71
Episode 344     Average Score: 13.01
Environment solved in 244 episodes!     Average Score: 13.01
```

In [12]:

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

#illustrate that the agent is able to receive an average reward (over 100 episodes) of
 at least +13.
```



In [13]:

```
# load the weights from file
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))
agent.qnetwork_local
```

Out[13]:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

In [14]:

```
# Uncomment following two lines to validate that the networks has loaded the parameter
s:
#for param in agent.qnetwork_local.parameters():
#    print(param)
```

In [15]:

```
env_info = env.reset(train_mode=False)[brain_name]
state = env_info.vector_observations[0]
state
```

Out[15]:

```
array([0.        , 1.        , 0.        , 0.        , 0.14546908,
       0.        , 0.        , 1.        , 0.        , 0.03315355,
       0.        , 1.        , 0.        , 0.        , 0.48084822,
       0.        , 0.        , 1.        , 0.        , 0.05033452,
       0.        , 1.        , 0.        , 0.        , 0.39260173,
       0.        , 0.        , 1.        , 0.        , 0.0361835 ,
       1.        , 0.        , 0.        , 0.        , 0.18227261,
       0.        , 0.        ])
```

In [16]:

```
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
for j in range(500):
    action = agent.act(state)        # select an action
    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 finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next time s
tep
    print('\rIntermediate Score: {}'.format(score), end="")
    if done:
        print('\rFinal Score: {}            '.format(score))
        break
```

```
Final Score: 6.0
```

When finished, you can close the environment.

```
#Create a mp4-video with kazam screencapture
#Convert mp4 to gif
#Then execute following python code:

import ffmpy
ff = ffmpy.FFmpeg(inputs = {"DQN2020-03-12.mp4" : None}, outputs = {"DQN2020-03-
12.gif" : None})
ff.run
```

5. How to improve the DQN

In [17]:

```
# Human-level control through deep reinforcement learning
# https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

# There many different algorithms to improve the standard DQN:
# ===============================================================
# Deep Reinforcement Learning with Double Q-learning
# https://arxiv.org/abs/1509.06461

# Dueling Network Architectures for Deep Reinforcement Learning
# https://arxiv.org/pdf/1710.02298.pdf

# Addressing Function Approximation Error in Actor-Critic Methods
# https://arxiv.org/abs/1802.09477

# Comparing different techniques
# Rainbow: Combining Improvements in Deep Reinforcement Learning
# https://arxiv.org/pdf/1710.02298.pdf
```

Implementation of the Double-DQN

In [18]:

```
# Next Step, improve the DQN by implementing a Double DQN
# The DQN has a tendency to overestimate the Q values and this may be bad for the train
g performance and could lead
# bad policies
# The Double-DQN is basically the same as the DQN, but the loss function is different:
# action selection is done by using the greedy policy with the online network
# action evaluation is done by using the target network
# the update of the target network with the values of the online network is done the sa
me as with the DQN


# Create the Double-DQN-Agent
agent = Agent(state_size=37, action_size=4, seed=0, ddqn=True)
```

In [19]:

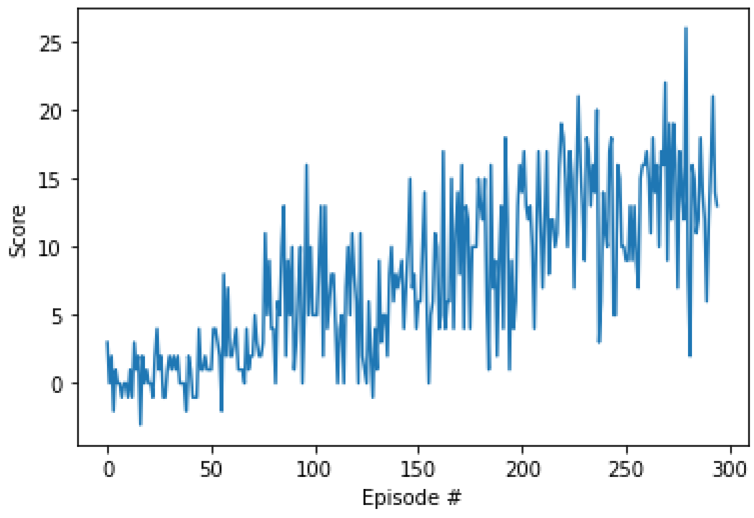
```
#Train my DDQN-agent
scores = dqn(ddqn=True)
```

```
Episode 100	Average Score: 2.53
Episode 200	Average Score: 7.42
Episode 295	Average Score: 13.02
Environment solved in 195 episodes!	Average Score: 13.02
```

In [20]:

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

#illustrate that the agent is able to receive an average reward (over 100 episodes) of
 at least +13.
```



In [21]:

```
# load the weights from file
agent.qnetwork_local.load_state_dict(torch.load('checkpointddqn.pth'))
agent.qnetwork_local
```

Out[21]:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

In [22]:

```
env_info = env.reset(train_mode=False)[brain_name]
state = env_info.vector_observations[0]
state
```

Out[22]:

```
array([1.        , 0.        , 0.        , 0.        , 0.15390952,
       0.        , 1.        , 0.        , 0.        , 0.30432957,
       0.        , 1.        , 0.        , 0.        , 0.14255197,
       0.        , 0.        , 1.        , 0.        , 0.18746784,
       0.        , 1.        , 0.        , 0.        , 0.25839198,
       0.        , 1.        , 0.        , 0.        , 0.37331545,
       0.        , 1.        , 0.        , 0.        , 0.28597626,
       0.        , 0.        ])
```

In [23]:

```
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
for j in range(500):
    action = agent.act(state)         # select an action
    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 finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next time s
tep
    print('\rIntermediate Score: {}'.format(score), end="")
    if done:
        print('\rFinal Score: {}            '.format(score))
        break
```

```
Final Score: 5.0
```

In [24]:

```
env.close()
```

Discussion of the results

The Double-DQN and the DQN seem to have the same performance in training speed and reward scoring. During this execution run, the Double-DQN showed a better performance, but previous runs showed that Double-DQN can also show an inferior performance. This is due to the highly stochastic scenario generation and learning process. A systematic many-times-simulation and calculation of the reward distribution as it is was done in the original Double-DQN-paper would provide a more accurate insight.

In [ ]:

```

```