Navigation

January 18, 2022

1 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.

1.0.1 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 and NumPy.

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

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")
```

[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.

[3]:
```python
# 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

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.

[4]:
```python
# 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
```

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.

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!

[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␣
↪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 step
    if done:                                       # exit loop if episode␣
↪finished
        break

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

```
Score: 1.0
```

1.0.4 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]
```

[6]:

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

[7]:

```
def train(agent, model_file_path, n_episodes=1800, 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 episodes
        eps_start (float): starting value of epsilon, for epsilon-greedy action␣
↪selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing␣
↪epsilon
    """
    scores = []
    scores_window = deque(maxlen=100) # last 100 scores
    eps = eps_start
    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]  # get the current state
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)[brain_name]
            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
            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)
        eps = max(eps_end, eps_decay*eps)
        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(scores_window)))
```

```
        if np.mean(scores_window)>=13.0:
            print("\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}".format(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), model_file_path)
            break
    return scores
```

4.1. Train the Agent with a DQN Train the Agent with a basic Deep Q-Network (two hidden layers with 64 units each) and the following settings:

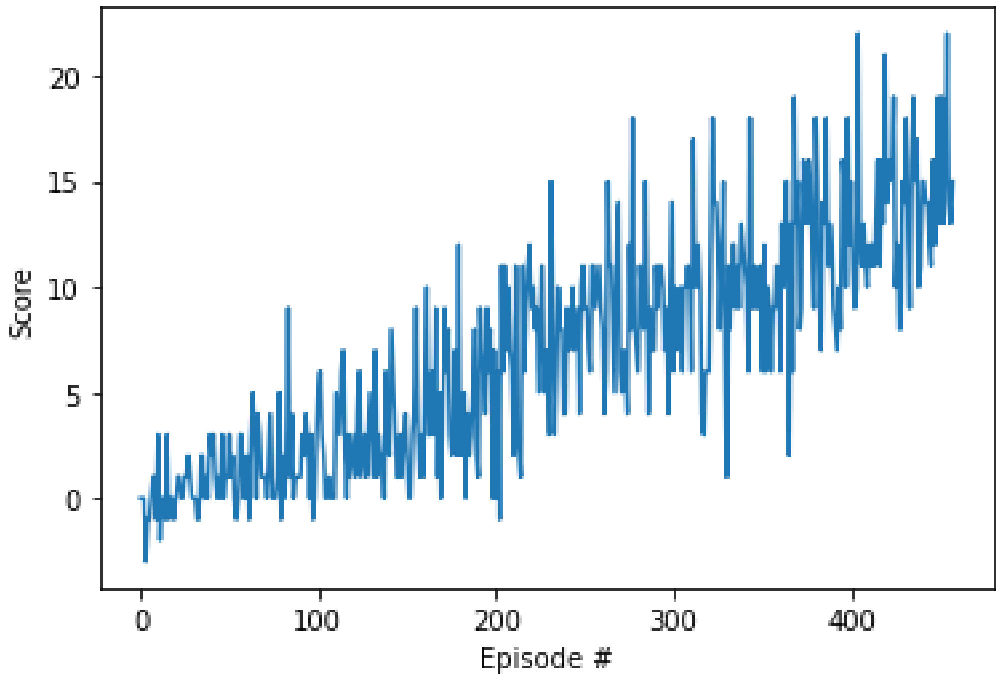
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

[8]:
```
from dqn_agent import DqnAgent
dqnagent = DqnAgent(state_size=state_size, action_size=action_size, seed=0)
```

[9]:
```
scores = train(dqnagent, 'dqn-checkpoint.pth')

#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()
```

```
Episode 100     Average Score: 1.05
Episode 200     Average Score: 3.59
Episode 300     Average Score: 8.11
Episode 400     Average Score: 10.60
Episode 458     Average Score: 13.05
Environment solved in 358 episodes!     Average Score: 13.05
```



4.2. Train the Agent with a Double DQN Train the Agent with a Double Deep Q-Network (two hidden layers with 32 units each) and the following settings:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-2              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

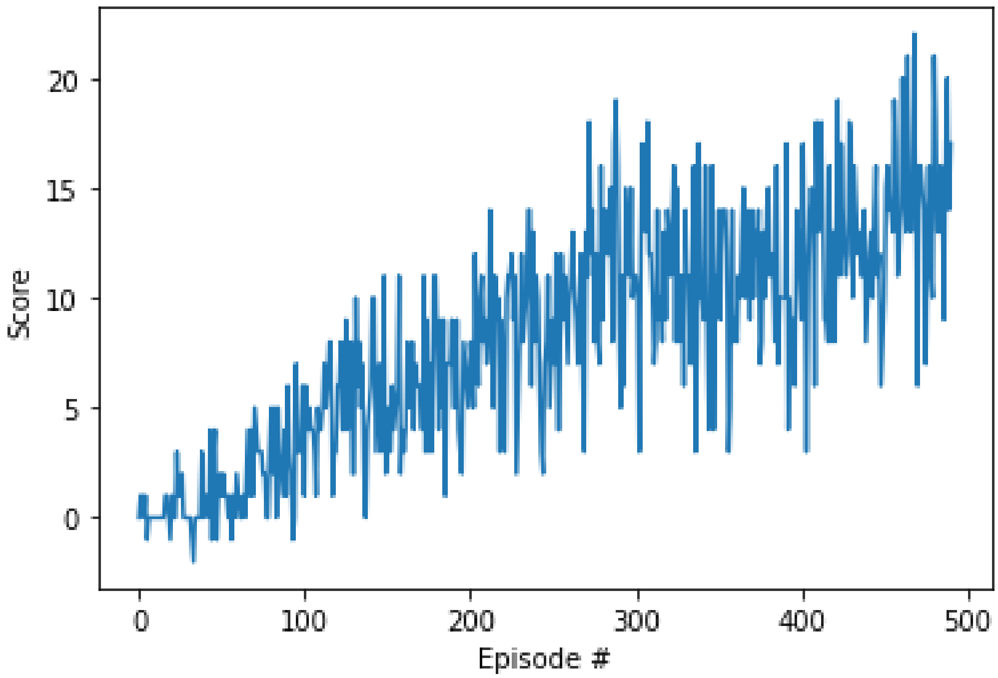
You can read more about Double DQN by perusing this research paper.

[10]:
```
from double_dqn_agent import DoubleDqnAgent
doubledqnagent = DoubleDqnAgent(state_size=state_size, action_size=action_size,␣
↪seed=0)
```

[11]:
```
scores = train(doubledqnagent, 'double-dqn-checkpoint.pth')

#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()
```

```
Episode 100	Average Score: 1.32
Episode 200	Average Score: 5.48
Episode 300	Average Score: 9.58
Episode 400	Average Score: 10.83
Episode 491	Average Score: 13.05
Environment solved in 391 episodes!	Average Score: 13.05
```



See the Learn Agent in Action

[12]:
```
# load the weights from file
doubledqnagent.qnetwork_local.load_state_dict(torch.load('double-dqn-checkpoint.
 ↪pth'))

for i in range(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 = doubledqnagent.act(state)
        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 step
        if done:                                      # exit loop if episode␣
↪finished
            break

    print("Score for trial {} : {}".format(i, score))
```

```
Score for trial 0 : 19.0
Score for trial 1 : 9.0
Score for trial 2 : 16.0
Score for trial 3 : 16.0
Score for trial 4 : 16.0
```

When finished, you can close the environment.

[13]:
```
env.close()
```

[ ]:
```

```