Navigation

January 29, 2020

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.

```
In [1]: from unityagents import UnityEnvironment
        import numpy as np
        from collections import deque
        import random
        import torch
        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.app")
```

```
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 [8]: # get the default brain
        brain_name = env.brain_names[0]
        print(brain_name)
        brain = env.brains[brain_name]
        print(brain)
```

```
BananaBrain
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: , , ,
```

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.

```
In [9]: # 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.          0.43657523 1.
 0.          0.          0.          0.19398789 1.          0.
 0.          0.          0.47860974 0.          0.          1.
 0.          0.52109712 0.          0.          1.          0.
 0.38285938 1.          0.          0.          0.          0.10405888
 1.          0.          0.          0.          0.37148568 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!

```
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 environmen
            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
            if done:                                       # exit loop if episode finished
                break

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

```
Score: 0.0
```

When finished, you can close the environment.

```
In [6]: #env.close()
```

2 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]
```

2.0.1 My solution:

```
In [4]: import sys
        import pandas as pd
        import numpy as np
        from numpy import linalg as LA
```

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

        agent = Agent(state_size=37, action_size=4, seed=0)
```

```
version 3.0
version 3.0
```

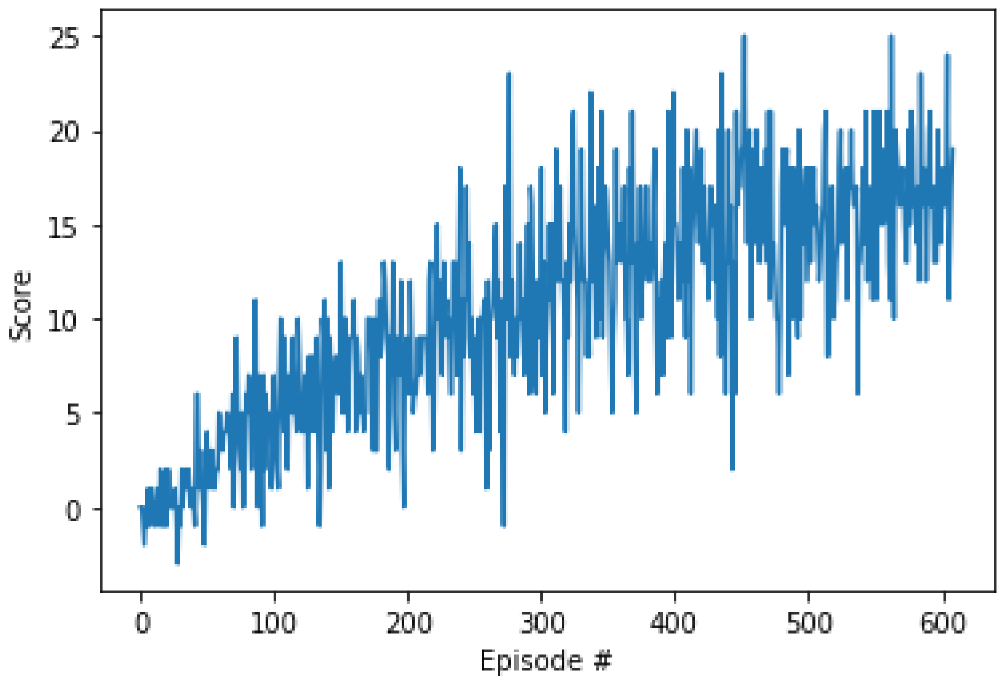
My configurations:
I used: eps_decay = 0.991
CNN with two hidden layers of 50 neurons each (dqn_agent.py)
I disable the dropout. The results were better without it.

```
In [7]: def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.991):
            """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 select
                eps_end (float): minimum value of epsilon
```

```
            eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
        """
        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)        # calculate an action
                env_info = env.step(action)[brain_name]        # send the action to the en
                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 finish
                agent.step(state, action, reward, next_state, done)
                score += reward                  # update the score
                state = next_state                             # roll over the state to nex

                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_wi
            if i_episode % 100 == 0:
                print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(score
            if (np.mean(scores_window)>=16.0) or (i_episode == n_episodes):
                print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
                torch.save(agent.qnetwork_local.state_dict(), 'model.pt')
                break
        return scores

    scores = dqn()

    # 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: 2.07
Episode 200	Average Score: 6.76
Episode 300	Average Score: 9.55
Episode 400	Average Score: 12.88
```

```
Episode 500	Average Score: 14.67
Episode 600	Average Score: 15.86
Episode 609	Average Score: 16.02
Environment solved in 509 episodes!	Average Score: 16.02
```




```
In [10]: # load the weights from file
         agent.qnetwork_local.load_state_dict(torch.load('model.pt'))

         for i in range(1):
             env_info = env.reset(train_mode=False)[brain_name]
             state = env_info.vector_observations[0]
             for j in range(10000):
                 action = agent.act(state)
                 #env.render()


                 env_info = env.step(action)[brain_name]        # send the action to the envir
                 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

                 if done:
                     break
```

```
In [11]: env.close()
```

2.0.2 Future Works

To improve the results I will:
Implement a double DQN, a dueling DQN and/or prioritized experience replay.

```
In [ ]:
```