Navigation

May 6, 2019

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 [8]: from unityagents import UnityEnvironment
        import numpy as np
        from collections import deque
        import torch
        import matplotlib.pyplot as plt
```

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 [3]: # 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.

```
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
```

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 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 st
            if done:                                       # exit loop if episode finished
                break

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

```
Score: 0.0
```

When finished, you can close the environment.

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

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:

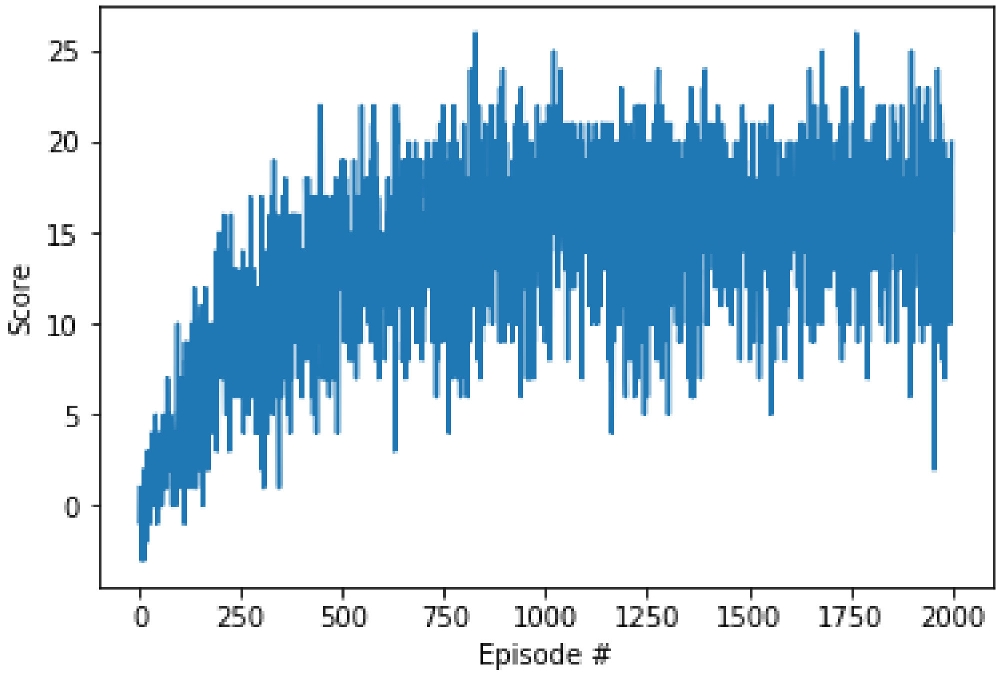
```
env_info = env.reset(train_mode=True)[brain_name]
```

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

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

```
In [10]: 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 selecti
                 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)
             eps = eps_start                    # initialize epsilon
             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):
                     action = agent.act(state, eps)
                     #next_state, reward, done, _ = env.step(action)[brain_name]
                     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)       # 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)>=20:
```

```
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
    1
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: 2.02
Episode 200         Average Score: 6.35
Episode 300         Average Score: 9.21
Episode 400         Average Score: 10.76
Episode 500         Average Score: 12.28
Episode 600         Average Score: 13.71
Episode 700         Average Score: 14.00
Episode 800         Average Score: 14.84
Episode 900         Average Score: 15.24
Episode 1000         Average Score: 15.38
Episode 1100         Average Score: 16.35
Episode 1200         Average Score: 15.55
Episode 1300         Average Score: 15.29
Episode 1400         Average Score: 15.69
Episode 1500         Average Score: 16.34
Episode 1600         Average Score: 15.21
Episode 1700         Average Score: 16.36
Episode 1800         Average Score: 15.86
Episode 1900         Average Score: 15.92
Episode 2000         Average Score: 15.98
```



1.1 Load weights and play games

```
In [ ]: # load the weights from file
        agent.qnetwork_local.load_state_dict(torch.load('checkpoint15.pth', map_location=lambda

        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)         # 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

            agent.step(state, action, reward, next_state, done)
            state = next_state
            if done:                                       # exit loop if episode finished
                break

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

1.2 Learning Algorithm

1.2.1 RL Algorithm

Here I use DDQN algorithm to train the agent. By looking at the Agent.learn() function we can see that there are two Q-networks, one is local Q-network and one is target Q-network, which implies an off-policy method.

We train the agent for at most 2000 episodes. The training process stops one the agent reaches 15 points on average. The agent would interact with the environment for 1000 times for each episode. The epsilon starts at 1 and would decay as the training process coontinues with decay rate 0.995. We set the minimum epsilon to be 0.01 to make sure the algorithm would continuously do exploration.

1.2.2 Neural Network Algorithm

Here I use Neural Network architecture described as follows: 37 * 64 * 64 * 4. The input layer has the same dimensions as the state vector, followed by 2 hidden layers with 64 neurons, and output layer of 4 neurons. Here the output layer returns probabilities vector instead of softmax result.

1.2.3 Future Work

It is worthwhile to try Prioritized experience replay. Also, more complicated neural network architectures have the potential to improve the agent's performance. Pixels-based model should be tried given we have GPU on our local machine.

```
In [ ]:
```