Navigation

July 3, 2018

1 Navigation

In this notebook we train an agent in the Unity ML-Agents environment for the first project of the Deep Reinforcement Learning Nanodegree. The agent code is mostly independent from the environment - we used a DQN agent developed to work with OpenAI Gym's LunarLander-v2 environment for the Deep Q-Network lesson. The agent and the corresponding networks are defined in a separate files.

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
```

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="/data/Banana_Linux_NoVis/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]
```

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. Start the Agent

```
In [5]: from agent import Agent
        agent = Agent(state_size=state_size, action_size=action_size, seed=0)
```

1.0.4 4. Train the Agent

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]
```

Importing packages Most of these are needed only for debugging and experimenting purposes. The agent is standalone.

```
In [6]: import torch
        from collections import namedtuple, deque

        #import numpy as np
        #import random
        #from model import QNetwork
        #import torch.nn.functional as F
        #import torch.optim as optim
```

Training parameters

```
In [7]: 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
```

Training algorithm and Agent training

```
In [8]: 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 selectio
                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] # reset the environment
                state = env_info.vector_observations[0]            # get the current state
                #state = env.reset()
                score = 0
                for t in range(max_t):
                    action = agent.act(state, eps)
                    env_info = env.step(action)[brain_name]        # send the action to the envi
                    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
                    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_win
                if i_episode % 100 == 0:
                    print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores
                if np.mean(scores_window)>=14.0:
                    print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format
                    torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
                    break
            return scores

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

```
Episode 100	Average Score: 1.00
Episode 200	Average Score: 4.52
```

```
Episode 300	Average Score: 8.26
Episode 400	Average Score: 9.80
Episode 500	Average Score: 11.98
Episode 600	Average Score: 13.07
Episode 627	Average Score: 14.01
Environment solved in 527 episodes!	Average Score: 14.01
```

```
In [9]: # save the trained weights to file
        torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
```

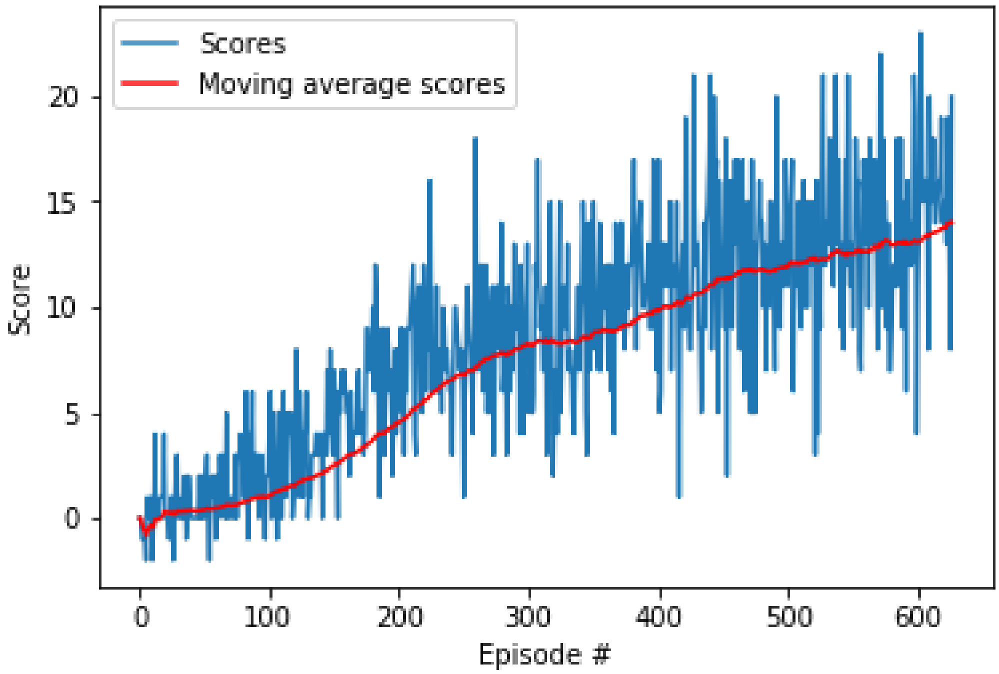
Scores plot

```
In [10]: import matplotlib.pyplot as plt
         %matplotlib inline
```

```
In [11]: # plot the scores
         fig = plt.figure()
         #ax = fig.add_subplot(111)
         plt.plot(np.arange(len(scores)), scores, label='Scores')
         plt.ylabel('Score')
         plt.xlabel('Episode #')

         cumsum_vec = np.cumsum(scores)
         window_width = 100
         ma_vec1 = np.cumsum(scores[:window_width])/(np.arange(window_width)+1)
         ma_vec2 = (cumsum_vec[window_width:] - cumsum_vec[:-window_width]) / window_width
         ma_vec = np.concatenate((ma_vec1, ma_vec2))
         plt.plot(np.arange(len(scores)), ma_vec, label='Moving average scores', color='r')
         plt.legend()
         #https://stackoverflow.com/a/34387987

         plt.show()
```



When finished, you can close the environment.

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

1.0.5 4. Watch a Smart Agent!

You can load the trained weights from file to watch a smart agent in the Navigation_Demo.ipynb !

1.0.6 References and Acknowledgments

This file based on the corresponding file in the first project of the Deep Reinforcement Learning Nanodegree, https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

The agent and the training algorithms are from the *Deep Q-Networks* with slight modifications to account for the different environment (instead of OpenAI gym interface we use Unity environment).