Navigation

April 22, 2021

1 Navigation

You are welcome to use this coding environment to train your agent for the project. Follow the instructions below to get started!

1.0.1 1. Start the Environment

Run the next code cell to install a few packages. This line will take a few minutes to run!

```
In [1]: !pip -q install ./python
```

```
tensorflow 1.7.1 has requirement numpy>=1.13.3, but you'll have numpy 1.12.1 which is incompatib
ipython 6.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.15, but you'll have prompt-toolkit 3.0.
```

```
In [4]: import sys
        !{sys.executable} -m pip install numpy==1.13.3
```

```
Collecting numpy==1.13.3
  Downloading https://files.pythonhosted.org/packages/57/a7/e3e6bd9d595125e1abbe162e323fd2d06f6f
    100% || 17.0MB 2.2MB/s eta 0:00:01   23% |                          | 3.9MB 23.3MB/s eta 0:00:
Installing collected packages: numpy
  Found existing installation: numpy 1.12.1
    Uninstalling numpy-1.12.1:
      Successfully uninstalled numpy-1.12.1
Successfully installed numpy-1.13.3
```

```
In [ ]: #import sys
        #!{sys.executable} -m pip install prompt_toolkit==1.0.15
```

The environment is already saved in the Workspace and can be accessed at the file path provided below. Please run the next code cell without making any changes.

```
In [5]: from unityagents import UnityEnvironment
        import numpy as np

        # please do not modify the line below
        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 [6]: # 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

Run the code cell below to print some information about the environment.

```
In [7]: # 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.

Note that **in this coding environment, you will not be able to watch the agent while it is training**, and you should set `train_mode=True` to restart the environment.

```
In [15]: env_info = env.reset(train_mode=True)[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 s
             if done:                                       # exit loop if episode finished
                 break

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

```
Score: 0.0
```

When finished, you can close the environment.

```
In [16]: env.close()
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! A few **important notes**: - 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]
```

- To structure your work, you're welcome to work directly in this Jupyter notebook, or you might like to start over with a new file! You can see the list of files in the workspace by clicking on ***Jupyter*** in the top left corner of the notebook.
- In this coding environment, you will not be able to watch the agent while it is training. However, ***after training the agent***, you can download the saved model weights to watch the agent on your own machine!

In [8]:
```
import random
import torch
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline
```

In [12]:
```
# Instantiate an Agent object
from dqn_agent import Agent

agent = Agent(state_size=state_size, action_size=action_size, seed=0)

print(agent.qnetwork_local)
```

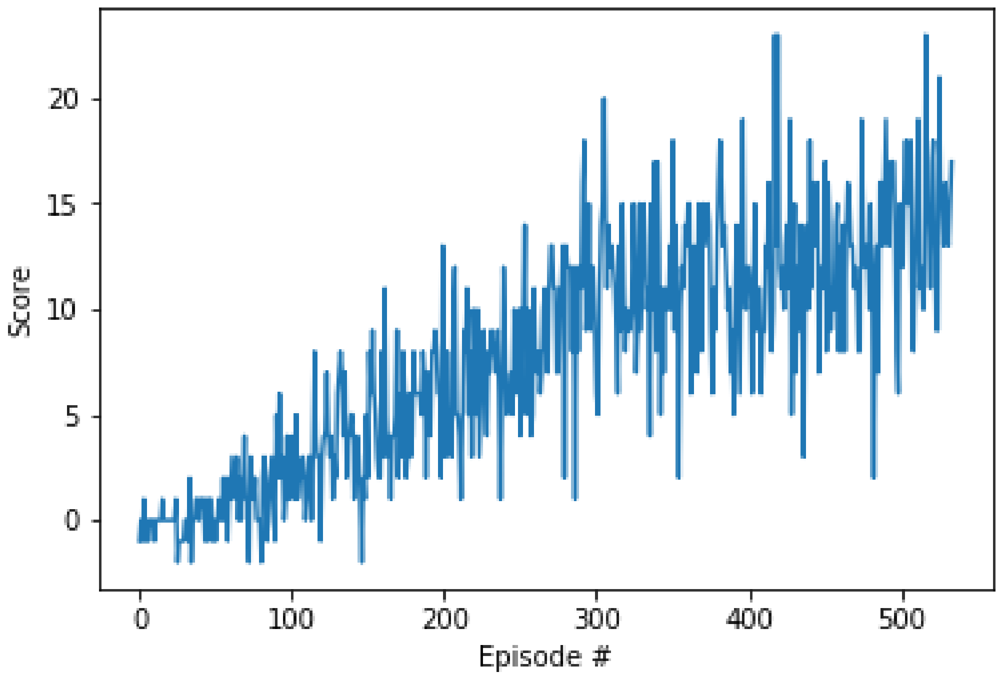
```
QNetwork(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

In [15]:
```
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)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    WINNING_AVG_SCORE = 13             # how to know we won (specified in problem state
    for i_episode in range(1, n_episodes+1):                  # each new episode...
        env_info = env.reset(train_mode=True)[brain_name] # > reset environment
        state = env_info.vector_observations[0]           # > initial state
        score = 0                                         # > reset score
        for t in range(max_t):
```

```
                action = agent.act(state, eps)                     # choose action with GL
                env_info = env.step(action)[brain_name]            # send the action to th
                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 fi
                agent.step(state, action, reward, next_state, done) # learning update
                score += reward                                    # update the score
                state = next_state                                 # roll over the state t
                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)>=WINNING_AVG_SCORE:
                print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
                torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
                break
        return scores
```

In [16]:
```
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: 0.61
Episode 200	Average Score: 4.35
Episode 300	Average Score: 8.22
Episode 400	Average Score: 11.37
Episode 500	Average Score: 12.22
Episode 534	Average Score: 13.06
Environment solved in 434 episodes!	Average Score: 13.06
```




```
In [ ]:
```