Navigation

April 19, 2020

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 [2]: !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 [3]: from unityagents import UnityEnvironment
        import numpy as np
```

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 [4]: # 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 [5]:
```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

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

In [6]:
```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.

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 [7]: 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 st
            if done:                                       # exit loop if episode finished
                break

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

```
Score: 0.0
```

When finished, you can close the environment.

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!

1.1 Comment:

- Q_agent.py is modified based on dqn.py from the exercise project.
- Two training mode are allowed:

1) mode = 'dqn' --> Deep Q Network;
2) mode = 'double' --> double DQN.

- In 'double DQN mode', the local network is used for greedy policy to pick up an action, while the target network is used for determining the action value.

- When implementing 'double DQN mode', the problem can be solved in around 1100 episodes with an average reward 17.58. The saved model weights is: 'DoubleQlearning64.pth'
- future improvement: Prioritized Experience Replay could be used to improve the current model performance

```
In [8]: from Q_agent import Agent
        from collections import deque
        import torch
        import matplotlib.pyplot as plt
```

```
In [9]: def training(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
            score_hold = -999
            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)

                    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]

                    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)>=score_hold:
                    print('\nSave model after {:d} episodes!\tAverage Score: {:.2f}'.format(

                    torch.save(agent.qnetwork_local.state_dict(), save_name)
                    score_hold = np.mean(scores_window)
        return scores
```

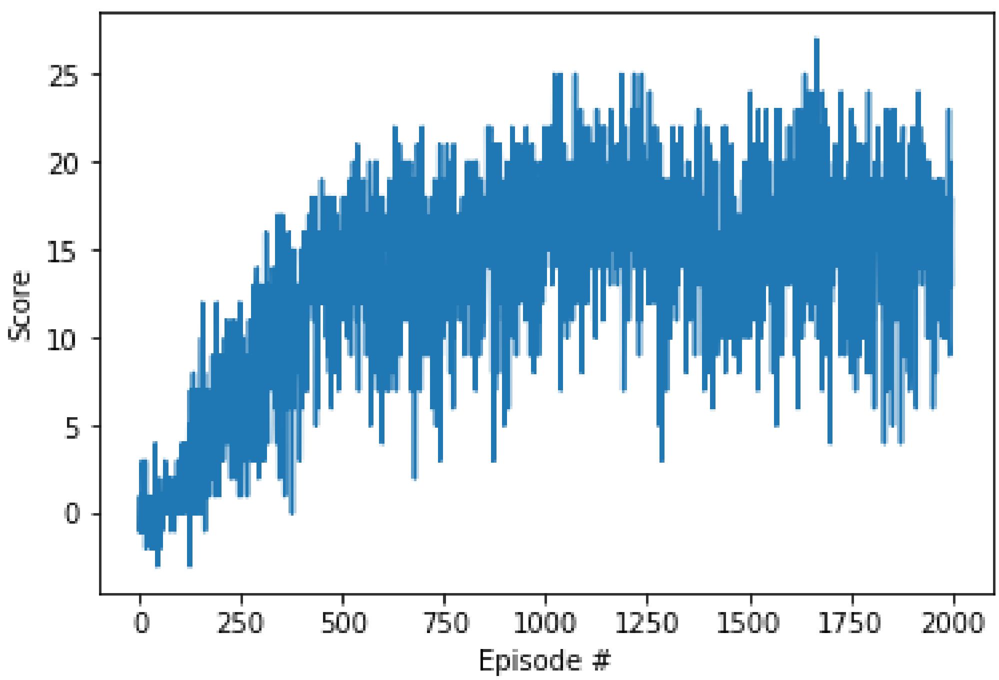
In [10]:
```
agent = Agent(state_size=37, action_size=4, hidden_size = 64, seed=0, mode = 'double')
```

In [11]:
```
scores = training(save_name = 'DoubleQlearning64.pth')
```

```
Episode 100	Average Score: 0.37

Save model after 0 episodes!	Average Score: 0.37
Episode 200	Average Score: 3.57

Save model after 100 episodes!	Average Score: 3.57
Episode 300	Average Score: 6.93

Save model after 200 episodes!	Average Score: 6.93
Episode 400	Average Score: 10.17

Save model after 300 episodes!	Average Score: 10.17
Episode 500	Average Score: 13.21

Save model after 400 episodes!	Average Score: 13.21
Episode 600	Average Score: 14.29

Save model after 500 episodes!	Average Score: 14.29
Episode 700	Average Score: 14.54

Save model after 600 episodes!	Average Score: 14.54
Episode 800	Average Score: 14.40
Episode 900	Average Score: 15.53

Save model after 800 episodes!	Average Score: 15.53
Episode 1000	Average Score: 15.75

Save model after 900 episodes!	Average Score: 15.75
Episode 1100	Average Score: 16.54

Save model after 1000 episodes!	Average Score: 16.54
Episode 1200	Average Score: 17.58

Save model after 1100 episodes!	Average Score: 17.58
Episode 1300	Average Score: 16.18
```

```
Episode 1400	Average Score: 16.07
Episode 1500	Average Score: 14.61
Episode 1600	Average Score: 16.03
Episode 1700	Average Score: 16.20
Episode 1800	Average Score: 15.76
Episode 1900	Average Score: 15.40
Episode 2000	Average Score: 15.65
```

In [12]:
```python
# 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()
```



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