Navigation

June 1, 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 [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.
```

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 [2]: 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 [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

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.

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 [5]: 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: 1.0
```

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!

1.0.5 Step 5: Import the dependencies

```
In [13]: from collections import deque
         import matplotlib.pyplot as plt
         import random
         import torch


         %matplotlib inline
```

1.0.6 Step 6: Start Implementing

In [14]:
```
from agent import Agent
```

1.0.7 Step 7: Deep Q Learning

In [15]:
```
# Deep Q-Learning Function

def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995, trai
        ckpt_path='saved_weights/weights.pth'):
    """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
        train_mode (bool): if 'True' set environment to training mode

    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    moving_avgs = []                   # list of moving averages
    eps = eps_start                    # initialize epsilon
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=train_mode)[brain_name] # reset environment
        state = env_info.vector_observations[0]                 # get current state
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)                      # select an action
            env_info = env.step(action)[brain_name]             # send action to enviro
            next_state = env_info.vector_observations[0]        # get next state
            reward = env_info.rewards[0]                        # get reward
            done = env_info.local_done[0]                       # see if episode has fi
            agent.step(state, action, reward, next_state, done) # learning step
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)          # save most recent score to window
        scores.append(score)                 # save most recent score to total
        moving_avg = np.mean(scores_window)  # calculate moving average
        moving_avgs.append(moving_avg)       # save most recent moving average
        eps = max(eps_end, eps_decay*eps)    # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, moving_avg), end=
        if i_episode % 100 == 0:
```

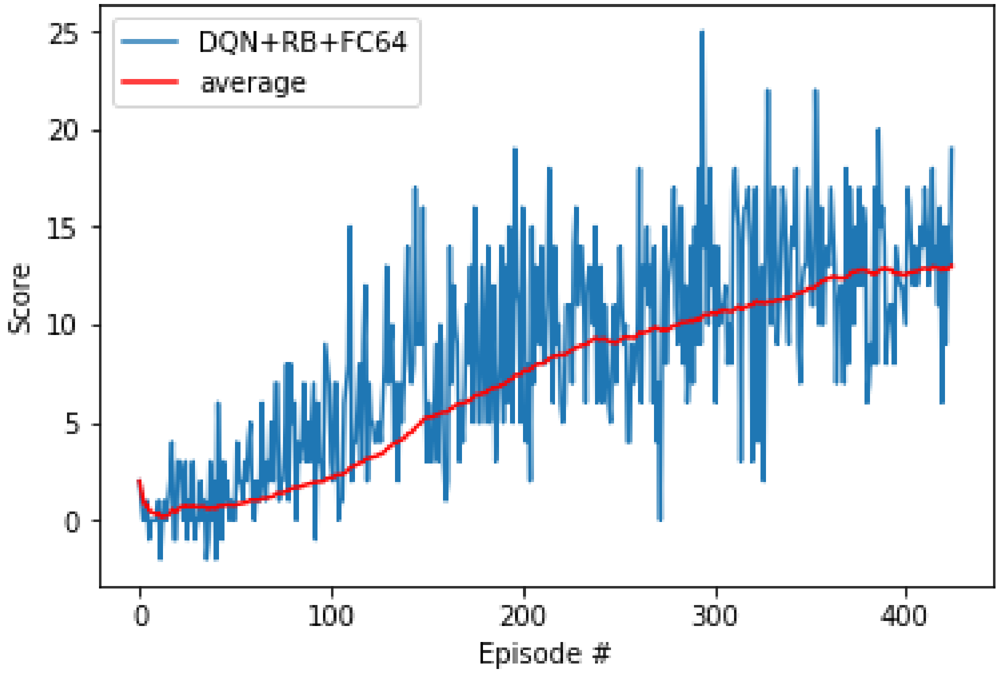
```
                    print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, moving_avg))
                if moving_avg >= 13.0:
                    print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
                    if train_mode:
                        torch.save(agent.qnetwork_local.state_dict(), ckpt_path)
                    break
            return scores, moving_avgs
```

1.0.8 Best Performing Agent

- Standard DQN + replay buffer (no double, no dueling)

```
In [18]: # run the training loop
         agent = Agent(state_size=state_size, action_size=action_size, seed=0, use_double=False,
         scores, avgs = dqn(n_episodes=600, eps_decay=0.98, eps_end=0.02, ckpt_path='saved_weigh

         # plot the scores
         fig = plt.figure()
         ax = fig.add_subplot(111)
         plt.plot(np.arange(len(scores)), scores, label='DQN+RB+FC64')
         plt.plot(np.arange(len(scores)), avgs, c='r', label='average')
         plt.ylabel('Score')
         plt.xlabel('Episode #')
         plt.legend(loc='upper left');
         plt.show()
```

```
Episode 100        Average Score: 2.16
Episode 200        Average Score: 7.44
Episode 300        Average Score: 10.62
Episode 400        Average Score: 12.56
Episode 425        Average Score: 13.01
Environment solved in 325 episodes!        Average Score: 13.01
```

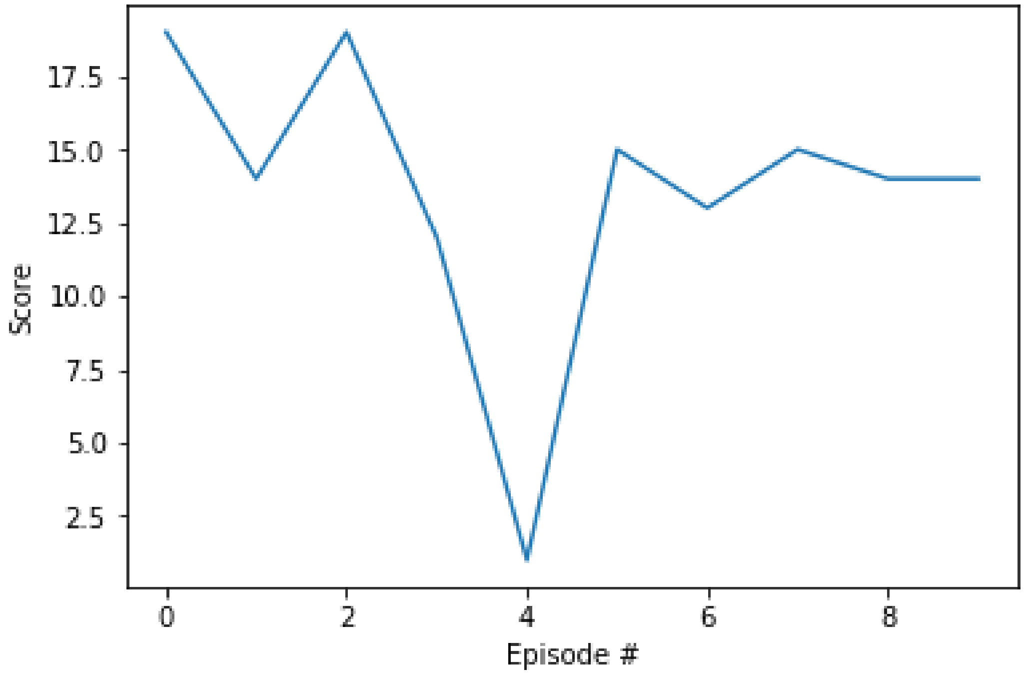


1.0.9 Testing of Best Performing Agent

In [20]:
```
## Test the saved agent

# initialize the agent
agent = Agent(state_size=state_size, action_size=action_size, seed=0)

# load the weights from file
checkpoint = 'saved_weights/final_weights.pth'
agent.qnetwork_local.load_state_dict(torch.load(checkpoint))

num_episodes = 10
scores = []
for i_episode in range(1,num_episodes+1):
    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, eps=0)               # select an action
        env_info = env.step(action)[brain_name]        # send the action to the environ
        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) # do the learning
```

```
            score += reward                                  # update the score
            state = next_state                               # roll over the state to next ti
            if done:                                         # exit loop if episode finished
                scores.append(score)
                print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(score
                break

    # 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 1        Average Score: 19.00
Episode 2        Average Score: 16.50
Episode 3        Average Score: 17.33
Episode 4        Average Score: 16.00
Episode 5        Average Score: 13.00
Episode 6        Average Score: 13.33
Episode 7        Average Score: 13.29
Episode 8        Average Score: 13.50
Episode 9        Average Score: 13.56
Episode 10        Average Score: 13.60
```



1.0.10 Summary

In [23]:
```
from IPython.display import Image

Image(filename='images/Results.png', width=800)
```

Out[23]:

Iteration	Agent	Model (FC1 units)	Eps Decay	Eps End	Result (# episodes)
28	DQN+RB	64	0.98	0.02	200
8	DQN+RB	64	0.95	0.03	216
29	DDQN+RB	64	0.98	0.02	231
9	DDQN+RB+Dueling	64	0.98	0.02	232
10	DQN+RB	64	0.98	0.02	245
26	DQN+RB+Dueling	128	0.98	0.02	246
19	DQN+RB+Dueling	128	0.985	0.015	248
12	DQN+RB	64	0.99	0.01	263
25	DDQN+RB	128	0.98	0.02	263
20	DQN+RB	128	0.985	0.015	266
31	DDQN+RB+Dueling	64	0.98	0.02	274
22	DQN+RB	128	0.985	0.05	285
21	DQN+RB	128	0.95	0.03	294
18	DDQN+RB	128	0.985	0.015	297
23	DQN+RB	128	0.985	0.01	312
27	DDQN+RB+Dueling	128	0.98	0.02	312
17	DDQN+RB+Dueling	128	0.985	0.015	317
30	DQN+RB+Dueling	64	0.98	0.02	328
11	DDQN+RB+Dueling	64	0.99	0.01	333
24	DQN+RB	128	0.98	0.02	362
2	DQN+RB	64	0.995	0.01	380

Agent Legend	
DQN	Deep Q-Network
DDQN	Double DQN
Dueling	Dueling DQN
RB	Replay Buffer

In [ ]: