Navigation

April 10, 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]: %%capture
        !pip install numpy --upgrade;
        !pip install --upgrade ipython;
        !pip -q install ./python;
```

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: 0.0
```

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 5. Train the Agent with DQN

5.1. Set up

```
In [6]: import gym
        import random
        import torch
        #import numpy as np
        from collections import deque
        from dqn_agent import Agent
        import matplotlib.pyplot as plt
        %matplotlib inline
```

5.2. Initialize the agent

```
In [7]: agent = Agent(state_size, action_size, seed=0)
```

5.3. Define the DQN

Main definitions for DQN algorithm:

- **Deep Q-Network(DQN):**

Combined reinforcement learning with deep learning, by stabilizing the q learning in a neural network using Experience Replay and Fixed-Q Targets.

- **Architecture:**

Is composed of three fully connected layers with 64 neurons in the first two and an equal number of neurons as actions in the last layer. The activation function for the output transformer of the two first layers is a ReLU function.

- **Hyperparameters:**

```
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
```

```
In [8]: def dqn(n_episodes, max_t, eps_start, eps_end, eps_decay):
            """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]
                state = env_info.vector_observations[0]
                score = 0
                for t in range(max_t):
                    action = agent.act(state, eps)                 # Next 4 lines are the on
                    env_info = env.step(action)[brain_name]        # get environment info
                    next_state = env_info.vector_observations[0]   # get state observations
                    reward = env_info.rewards[0]                   # get reward
```

```
                done = env_info.local_done[0]                        # get episode status
                agent.step(state, action, reward, next_state, done)# update agent
                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)>=13:
                print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format
                torch.save(agent.qnetwork_local.state_dict(), 'model.pth')
                break
        return scores
```

5.4. Set the hiperparameters

```
In [9]: n_episodes=2000
        max_t=1000
        eps_start=1.0
        eps_end=0.01
        eps_decay=0.995
```

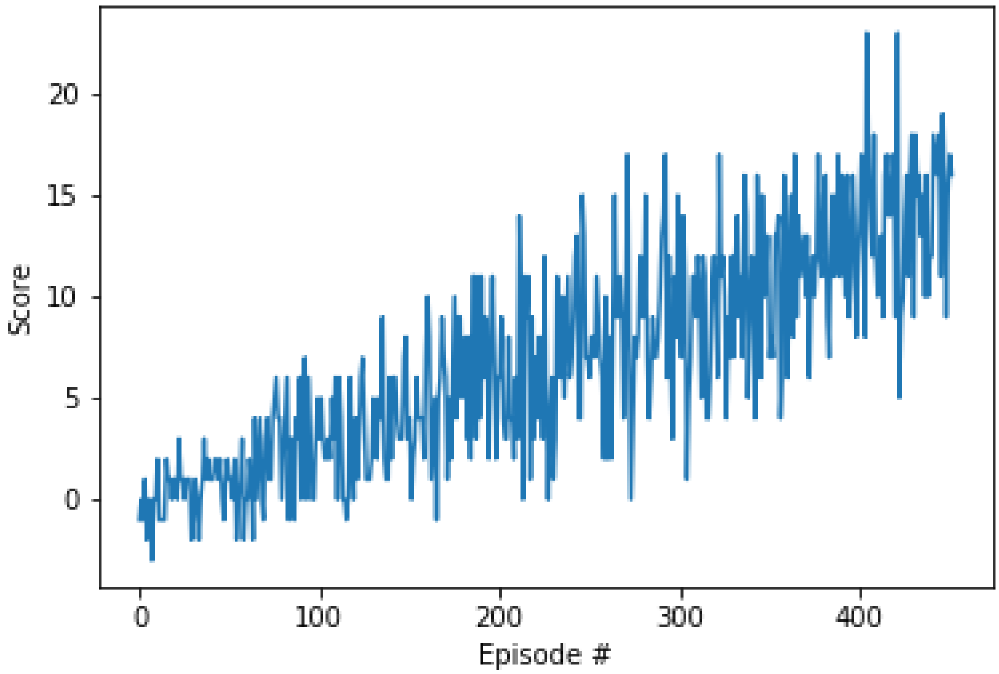
5.5. Train

```
In [10]: scores = dqn(n_episodes, max_t, eps_start, eps_end, eps_decay)
```

```
Episode 100        Average Score: 1.26
Episode 200        Average Score: 4.46
Episode 300        Average Score: 7.58
Episode 400        Average Score: 10.65
Episode 452        Average Score: 13.02
Environment solved in 452 episodes!        Average Score: 13.02
```

5.6. Show results

```
In [11]: # 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()
```



1.0.6 6. Run Trained agent

```
In [12]: #Load a trained agent
         agent.qnetwork_local.load_state_dict(torch.load("model.pth"))

         for i in range(3):
             env_info = env.reset(train_mode=False)[brain_name]
             state = env_info.vector_observations[0]
             score = 0
             for j in range(1000):
                 action = agent.act(state)
                 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

             print("\rEpisode {}\t Score: {:.2f}".format(i+1, score))
```

```
Episode 1          Score: 20.00
Episode 2          Score: 18.00
```

```
Episode 3          Score: 23.00
```

```
In [13]: env.close()
```

1.0.7 7. Ideas for future work

There are other two possible aproximations to improve the results:

1. Double DQNs
2. Dueling DQNs