Navigation

August 7, 2019

1 Navigation

In this notebook, you will learn how to use the Unity ML-Agents environment for the first project of the Deep Reinforcement Learning Nanodegree.

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.

[1]:
```
from unityagents import UnityEnvironment
import numpy as np
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline
```

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")
```

[2]:
```
import os
folder = r"c:
 ↪\Users\jayagup\Projects\deep-reinforcement-learning\p1_navigation\Banana_Windows_x86_64"
env = UnityEnvironment(file_name=os.path.join(folder, "Banana.exe"))
```

```
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.

[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.

[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.

Once this cell is executed, you will watch the agent's performance, if it selects an action (uniformly) at random with each time step. A window should pop up that allows you to observe the agent, as it moves through the environment.

Of course, as part of the project, you'll have to change the code so that the agent is able to use its experience to gradually choose better actions when interacting with the environment!

[5]:
```
# NOTE: During training, set train_mode to True.
# Otherwise, its slow has hell!
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 = 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 step
    if done:                                       # exit loop if episode␣
 ↪finished
        break

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

```
Score: 0.0
```

When finished, you can close the environment.

[6]:
```
# env.close()
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! 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]
```

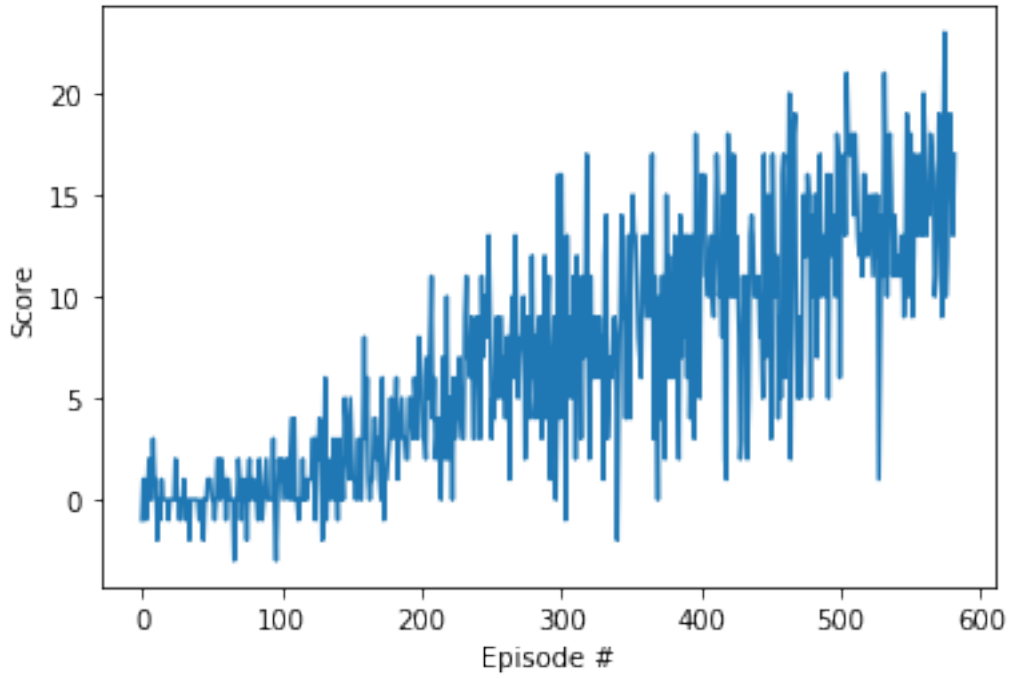
[7]:
```
from dqn_agent import Agent
env_info = env.reset(train_mode=True)[brain_name]
agent = Agent(state_size=37, action_size=4, seed=0)
```

Training Loop

[8]:
```
from agent_env_play import dqn
scores = dqn(agent=agent, env=env, n_episodes=1000)

# 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: 0.19
Episode 200     Average Score: 2.20
Episode 300     Average Score: 6.32
Episode 400     Average Score: 8.55
Episode 500     Average Score: 11.11
Episode 582     Average Score: 14.01
Environment solved in 482 episodes!     Average Score: 14.01
```



1.0.5 5. Architecture of the Q-Network & Hyperparameters

We use a simple multi-layered feed forward network to solve the problem.

Considering the input has a dimensionality of 37 and the output has a dimensionality of 4, the neural architecture was designed using the guidance provided here

- Per the guidance above, 2 layers were selected. The number of neurons for each layer were set by following the heuristic of *0.75 x input_nodes + output_nodes*. This results in the number of neurons per hidden layer to be ~32.
- The performance improved with adding a batch normalization layer.
- The learning rate of 1.0e-4 seemed to perform well.
- The default settings outlined in this exercise worked well.
- I experimented with the number of episodes. I started with 500 to understand the rate of improvement for the scores and finally settled on an upper bound of 1000.
- The model weights corresponding to the solution are saved in `checkpoint.pth`

1.0.6 6. Future Work

In order to make the solution more robust, the following directions could prove fruitful: 1. Prioritized experience replay: The current replay buffer extracts a random sample. Since it is more helpful to learn from samples that have a higher td-error, it could help in convergence. 2. Noisy Network might also prove a useful direction. The paper shows that parametric noise added to the network weights can be used to aid efficient exploration.