Navigation

July 31, 2022

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]:
```
import random
from collections import deque

import matplotlib.pyplot as plt
import numpy as np
import torch
from unityagents import UnityEnvironment

from dqn_agent import Agent
from model import DuelingQNetwork, QNetwork

%matplotlib inline

plt.ion()
```

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]:
```
env = UnityEnvironment(file_name="Banana_Linux/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.

[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. Train the agent

Here, we will use the Deep Q-Network (DQN) algorithm from Mnih et al. (2015). The model (`model.py`) consists of: * input size is 37 (state size) * 3 fully-connected layers * 64x64 nodes for the first two layers * ReLU activation is used for the output of FC1 and FC2. * ouptut size is 4 (number of possible actions)

I have modified the `Agent` class to accept both DQN and Dueling-DQN algorithm. In addition, the agent class can use Double-DQN to predict its next action state.

[5]:
```
def dqn(
    n_episodes=2000,
    max_t=1000,
    eps_start=1.0,
    eps_end=0.01,
    eps_decay=0.995,
    score_cutoff=14.0,
    checkpoint_name="dqn.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␣
 ↪selection
        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)
            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

        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_window)
            ),
            end="",
        )
        if i_episode % 100 == 0:
            print(
                "\rEpisode {}\tAverage Score: {:.2f}".format(
                    i_episode, np.mean(scores_window)
                )
            )

        if np.mean(scores_window) >= score_cutoff:
            print(
                "\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}".
↪format(
                    i_episode - 100, np.mean(scores_window)
                )
```

```
            )
            torch.save(agent.qnetwork_local.state_dict(), checkpoint_name)
            break

    return scores
```

1.0.4 Initialize Agent

[6]:
```
# parameters
n_episodes = 5000
max_t = 2000
eps_start = 1.0
eps_end = 0.1
eps_decay = 0.995
```

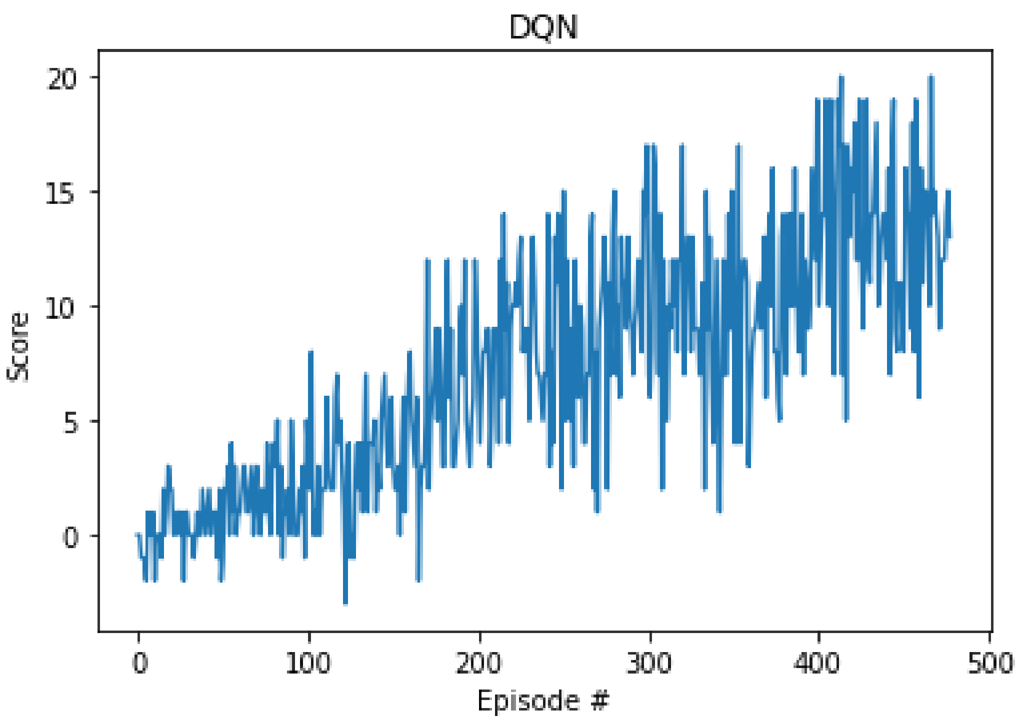
2 Vanilla DQN

[7]:
```
# Initialize Agent
agent = Agent(
    qnetwork=QNetwork,
    update_type="dqn",
    state_size=state_size,
    action_size=action_size,
    seed=0,
)
```

[8]:
```
# train the agent
scores = dqn(
    n_episodes,
    max_t,
    eps_start,
    eps_end,
    eps_decay,
    score_cutoff=13.0,
    checkpoint_name="dqn.pth",
)
```

```
Episode 100     Average Score: 1.00
Episode 200     Average Score: 4.23
Episode 300     Average Score: 8.88
Episode 400     Average Score: 9.98
Episode 478     Average Score: 13.01
Environment solved in 378 episodes!     Average Score: 13.01
```

[9]:
```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.title("DQN")
plt.savefig("dqn_scores.png", bbox_inches="tight")
```



[10]:
```python
agent.qnetwork_local.load_state_dict(torch.load("dqn.pth"))
```

[11]:
```python
# Watch trained Agent
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)  # 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: 16.0
```

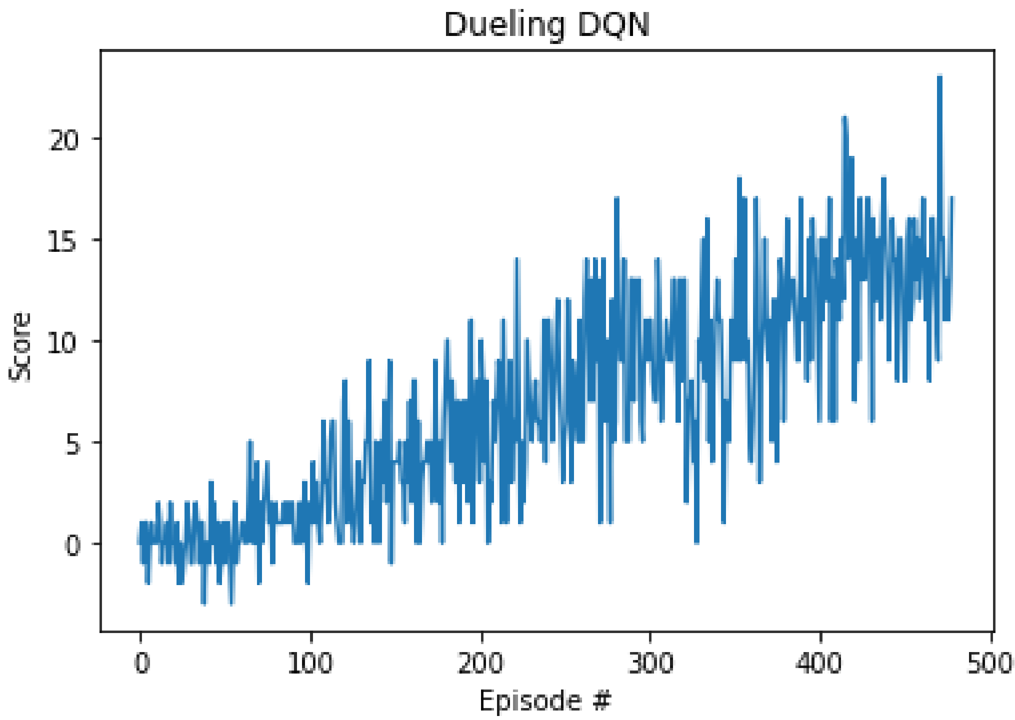
3 Dueling DQN

[12]:
```
# Initialize Agent
agent = Agent(
    qnetwork=DuelingQNetwork,
    update_type="dqn",
    state_size=state_size,
    action_size=action_size,
    seed=0,
)
```

[13]:
```
# train the agent
scores = dqn(
    n_episodes,
    max_t,
    eps_start,
    eps_end,
    eps_decay,
    score_cutoff=13.0,
    checkpoint_name="dueling-dqn.pth",
)
```

```
Episode 100	Average Score: 0.51
Episode 200	Average Score: 3.88
Episode 300	Average Score: 7.61
Episode 400	Average Score: 9.97
Episode 479	Average Score: 13.02
Environment solved in 379 episodes!	Average Score: 13.02
```

[14]:
```
# 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.title("Dueling DQN")
plt.savefig("dueling_dqn_scores.png", bbox_inches="tight")
```



[15]:
```
agent.qnetwork_local.load_state_dict(torch.load("dueling-dqn.pth"))
```

[16]:
```
# Watch trained Agent
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)  # 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: 19.0
```

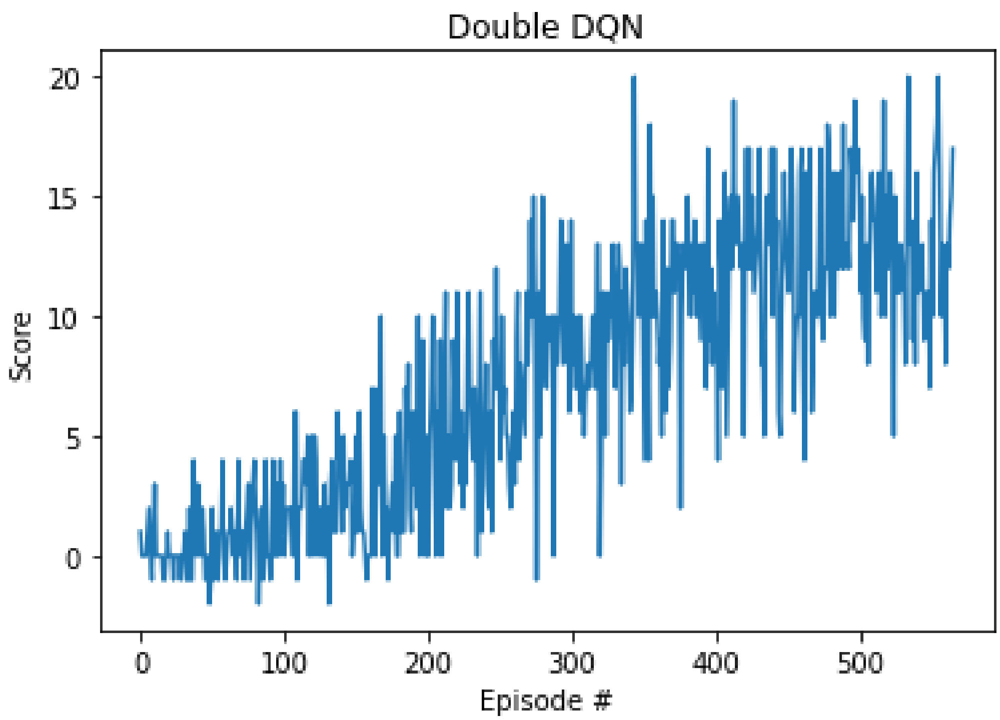
4 Double DQN

[17]:
```
# Initialize Agent
agent = Agent(
    qnetwork=QNetwork,
    update_type="double-dqn",
    state_size=state_size,
    action_size=action_size,
    seed=0,
)
```

[18]:
```
# train the agent
scores = dqn(
    n_episodes,
    max_t,
    eps_start,
    eps_end,
    eps_decay,
    score_cutoff=13.0,
    checkpoint_name="double-dqn.pth",
)
```

```
Episode 100     Average Score: 0.62
Episode 200     Average Score: 2.65
Episode 300     Average Score: 6.83
Episode 400     Average Score: 10.04
Episode 500     Average Score: 12.54
Episode 565     Average Score: 13.00
Environment solved in 465 episodes!     Average Score: 13.00
```

[19]:
```
# 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.title("Double DQN")
plt.savefig("double-dqn_scores.png", bbox_inches="tight")
```




```
[20]: agent.qnetwork_local.load_state_dict(torch.load("double-dqn.pth"))
```

```
[21]: # Watch trained Agent
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)  # 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: 11.0
```

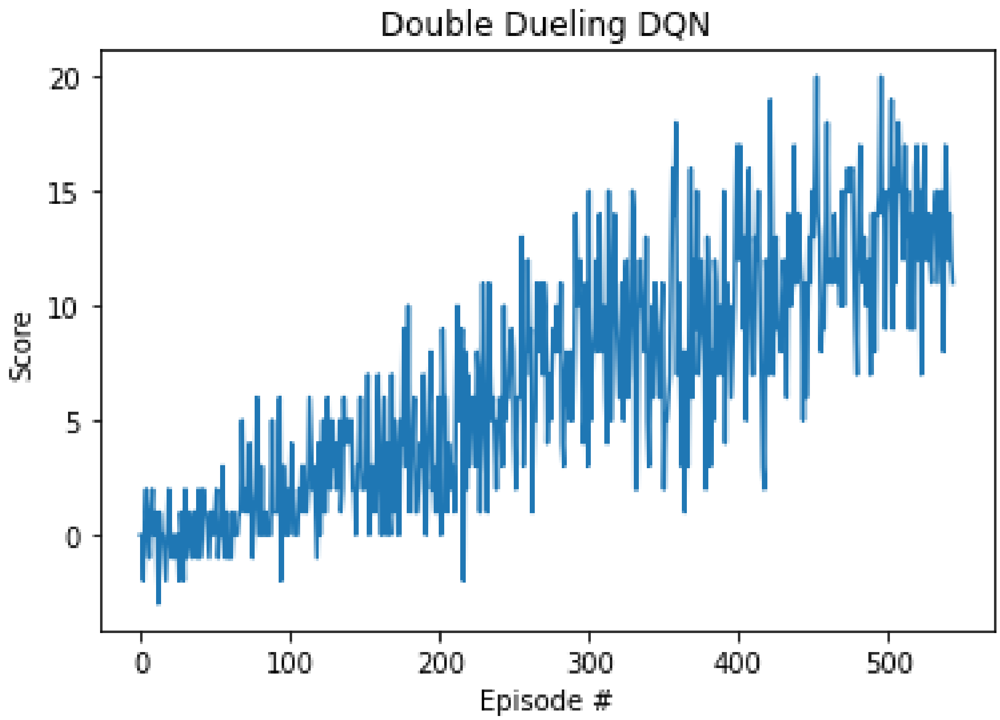
5 Double Dueling DQN

[22]:
```
# Initialize Agent
agent = Agent(
    qnetwork=DuelingQNetwork,
    update_type="double-dqn",
    state_size=state_size,
    action_size=action_size,
    seed=0,
)
```

[23]:
```
# train the agent
scores = dqn(
    n_episodes,
    max_t,
    eps_start,
    eps_end,
    eps_decay,
    score_cutoff=13.0,
    checkpoint_name="double-dueling-dqn.pth",
)
```

```
Episode 100     Average Score: 0.58
Episode 200     Average Score: 2.97
Episode 300     Average Score: 6.32
Episode 400     Average Score: 8.87
Episode 500     Average Score: 11.81
Episode 544     Average Score: 13.02
Environment solved in 444 episodes!     Average Score: 13.02
```

[24]:
```
# 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.title("Double Dueling DQN")
plt.savefig("double_dueling_dqn_scores.png", bbox_inches="tight")
```



[25]:
```
agent.qnetwork_local.load_state_dict(torch.load("double-dueling-dqn.pth"))
```

[26]:
```
# Watch trained Agent
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)  # 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: 15.0
```

When finished, you can close the environment.

[27]:
```
env.close()
```

5.0.1 4. Future directions

Possible ideas to experiment to improve learning is to apply other DQN-based algorithms: * Learning from multi-step bootstrap targets * Distributional DQN * Noisy DQN * Rainbow - combines 6 different DQN algorithms

[ ]:
```

```