Code implementation

The code used here is derived from the "Lunar Lander" tutorial from the Deep Reinforcement Learning Nanodegree, and has been slightly adjusted for being used with the banana environment.

The code consist of :

- model.py : In this python file, a PyTorch QNetwork class is implemented. This is a regular fully connected Deep Neural Network using the PyTorch Framework. This network will be trained to predict the action to perform depending on the environment observed states. This Neural Network is used by the DQN agent and is composed of :
    - the input layer which size depends of the state_size parameter passed in the constructor
    - 2 hidden fully connected layers of 64 node each
    - the output layer which size depends of the action_size parameter passed in the constructor
- dqn_agent.py : In this python file, a DQN agent and a Replay Buffer memory used by the DQN agent) are defined.
    - The DQN agent class is implemented, as described in the Deep Q-Learning algorithm. It provides several methods :
        - constructor :
            - Initialize the memory buffer (*Replay Buffer*)
            - Initialize 2 instance of the Neural Network : the *target* network and the *local* network
        - step() :
            - Allows to store a step taken by the agent (state, action, reward, next_state, done) in the Replay Buffer/Memory
            - Every 4 steps (and if their are enough samples available in the Replay Buffer), update the *target* network weights with the current weight values from the *local* network (That's part of the Fixed Q Targets technique)
        - act() which returns actions for the given state as per current policy (Note : The action selection use an Epsilon-greedy selection so that to balance between *exploration* and *exploitation* for the Q Learning)
        - learn() which update the Neural Network value parameters using given batch of experiences from the Replay Buffer.
        - soft_update() is called by learn() to softly updates the value from the *target* Neural Network from the *local*network weights (That's part of the Fixed Q Targets technique)
    - The ReplayBuffer class implements a fixed-size buffer to store experience tuples (state, action, reward, next_state, done)
        - add() allows to add an experience step to the memory
        - sample() allows to randomly sample a batch of experience steps for the learning
- Navigation.ipynb : This Jupyter notebooks allows to train the agent. More in details it allows to :
    - Import the Necessary Packages
    - Examine the State and Action Spaces
    - Take Random Actions in the Environment (No display)
    - Train an agent using DQN
    - Plot the scores

Algorithm: Deep Q-Learning

- Initialize replay memory D with capacity N
- Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
- Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
- **for** the episode $e \leftarrow 1$ to M:
  - Initial input frame x_1
  - Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
  - **for** time step $t \leftarrow 1$ to T:

SAMPLE

Choose action A from state S using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S,A,\mathbf{w}))$
Take action A, observe reward R, and next input frame x_{t+1}
Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
Store experience tuple (S,A,R,S') in replay memory D
$S \leftarrow S'$

LEARN

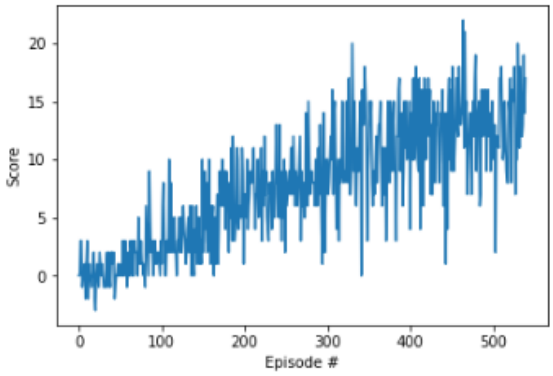
Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D
Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
Update: $\Delta\mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$
Every C steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$

The Neural Networks use Adam optimizer with a learning rate LR=5e-4 and are trained using a BATCH_SIZE=64.

Results:

These results meets the project's expectation as the agent is able to receive an average reward (over 100 episodes) of at least 13, and in 440 episodes only (In comparison, according to Udacity's solution code for the project, their agent was benchmarked to be able to solve the project in fewer than 1800 episodes)

```
Episode 100     Average Score: 1.01
Episode 200     Average Score: 4.62
Episode 300     Average Score: 7.74
Episode 400     Average Score: 10.72
Episode 500     Average Score: 12.64
Episode 540     Average Score: 13.00
Environment solved in 440 episodes!     Average Score: 13.00
```



Future updates:

The more challenging task would be to implement the learning directly from pixels using CNN (convolution neural network) to process raw pixel values.

Other enhancement to improve the model performance:

Double DQN: Applied algorithm Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.

Prioritized Experience Re-play: Another challenge with Deep Q-Learning is the sample experience transitions uniformly from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.

Dueling DQN: At present, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values for each action. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, the value of each state can be assessed without having to learn the effect of each action.