Udacity Deep Reinforcement Learning Nanodegree

Project 1: Navigation (Train an RL Agent to collect yellow bananas)

Goal: In this project, I build a reinforcement learning (RL) agent that navigates an environment that is like Unity’s banana collector environment.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The goal of our agent is to collect as many yellow bananas as possible while avoiding blue bananas. To solve the environment, our agent must achieve an average score of +13 over 100 consecutive episodes.

Environment Description: The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent must learn how to best select actions. Four discrete actions are available, corresponding to:

- 0 - move forward.
- 1 - move backward.
- 2 - turn left.
- 3 - turn right.

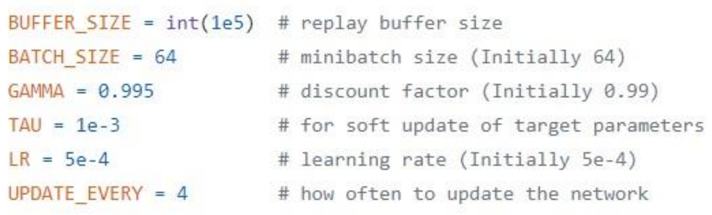
The task is episodic.

Learning Algorithm: I used deep Q-learning algorithm to train the agent. Agents use a policy to decide which actions to take within an environment. The primary objective of the learning algorithm is to find an optimal policy—i.e., a policy that maximizes the reward for the agent. Since the effects of possible actions aren't known in advance, the optimal policy must be discovered by interacting with the environment and recording observations. Therefore, the agent "learns" the policy through a process of trial-and-error that iteratively maps various environment states to the actions that yield the highest reward. This type of algorithm is called Q-Learning.

Code Implementation: The code I used here is derived from the Deep Reinforcement Learning Nanodegree. The code consists of:

1. model.py : In this python file, a PyTorch QNetwork class is implemented. This is regular fully connected Deep Neural Network using 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 on the state_size parameter passed in the constructor, two hidden fully connected layers of 1024 cells each and the output layer which size depends on the action_size parameter passed in the constructor.
2. dqn_agent.py : In this python file, a DQN agent and a Replay Buffer memory used by the DQN agents are defined. The DQN agent class is implemented, as described in the Deep Q-Learning algorithm.

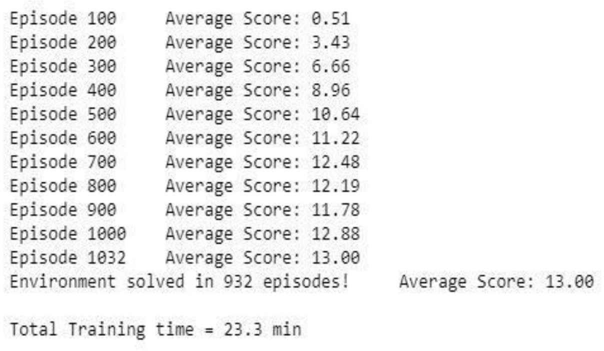
DQN parameters: The DQN agent uses the following parameters values (defined in dqn_agent.py)



```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size (Initially 64)
GAMMA = 0.995           # discount factor (Initially 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate (Initially 5e-4)
UPDATE_EVERY = 4        # how often to update the network
```

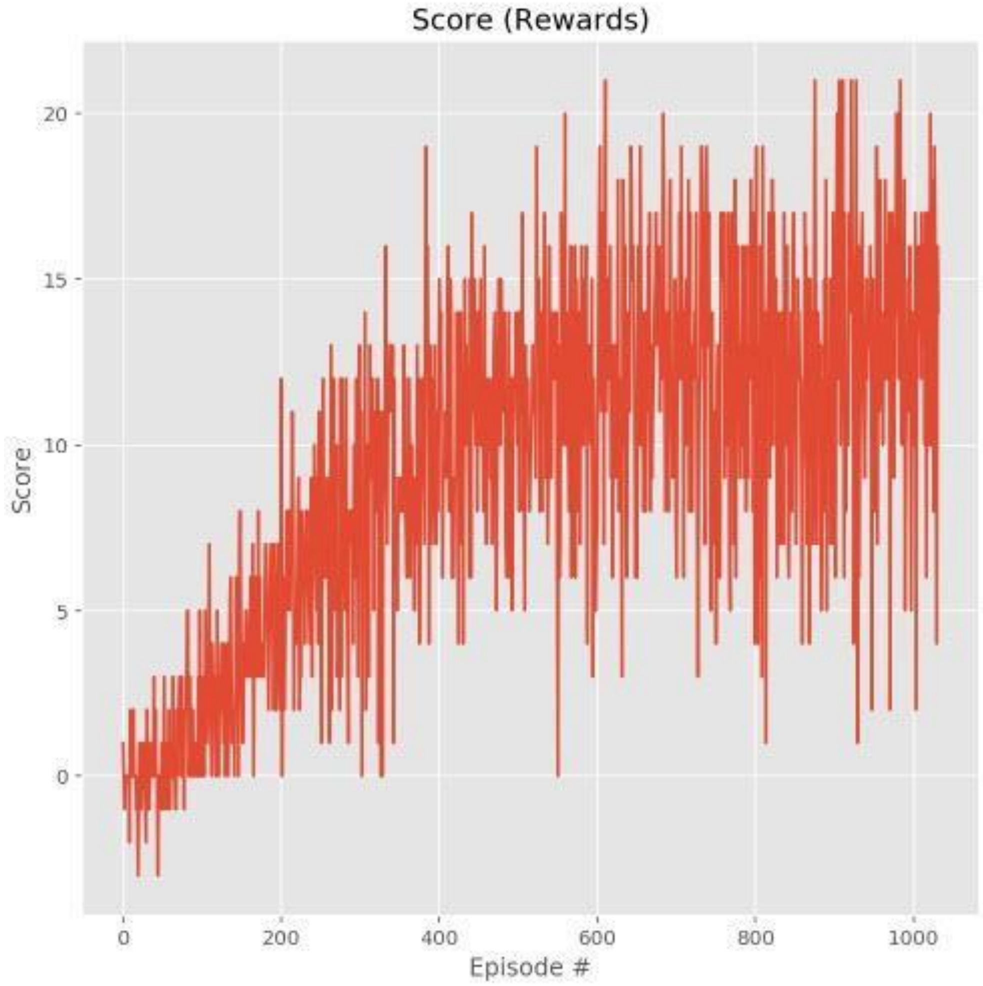
The Neural Networks use the Adam optimizer with a learning rate LR=5e-4 and are trained using a BATCH_SIZE=64

Given the chosen architecture and parameters, our results are:



```
Episode 100     Average Score: 0.51
Episode 200     Average Score: 3.43
Episode 300     Average Score: 6.66
Episode 400     Average Score: 8.96
Episode 500     Average Score: 10.64
Episode 600     Average Score: 11.22
Episode 700     Average Score: 12.48
Episode 800     Average Score: 12.19
Episode 900     Average Score: 11.78
Episode 1000    Average Score: 12.88
Episode 1032    Average Score: 13.00
Environment solved in 932 episodes!     Average Score: 13.00

Total Training time = 23.3 min
```



These results meet the project's expectation as the agent is able to receive an average reward (over 100 episodes) of at least +13, and in 932 episodes only.

Improvement: As discussed in the Udacity Course, a further evolution to this project would be to train the agent directly from the environment's observed raw pixels instead of using the environment's internal states (37 dimensions).

To do so a CNN would be added at the input of the network in order to process the raw pixels values (after some little pre-processing like rescaling the image size, converting RGB to grey scale, etc.)