Environment

This project trains an agent to navigate (and collect bananas) in a large, square world.
A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.
The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. Given this information, the agent has to 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, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

Model

We will use a Deep Neural Network as the function approximator for the Q function. The chosen DNN architecture is a 3 layer fully connected neural network with the input layer dimension being 37, the first hidden layer being 64 dimensional, the second hidden layer also being 64 dimensional, and the final output layer being 4 dimensional (for each different action). The activation function for the first and second layer is a **RELU** function.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 64]           2,432
            Linear-2                   [-1, 64]           4,160
            Linear-3                    [-1, 4]             260
================================================================
```

Agent¶

The model is trained using Gradient Descent with the **Adam optimizer** to update the weights. Nonlinear function approximators like neural networks can run into instabilities. To help improve convergence, two modifications will be made:

- •**Experience Replay** - To avoid learning experiences in sequence (which will lead to correlations between them) we store all the experience tuples (state, action, reward, and next state) in a memory buffer. The agent will learn from randomly sampled experiences from this buffer. This also helps us learn from the same experience multiple times, which is especially useful when encountering rare experiences.
- •**Fixed Q-values** - The TD target is also dependent on the network parameter w that we are trying to learn/update, and this can lead to instabilities. To address it, a separate network with identical architecture but different weights is used. The target network gets updated slowly according to the hyperparameter TAU while the local network aggressively “learns” with each update (**soft update**). At a high level, this loss function is taking the squared error between a

conservative/stable estimate of the value of state s and action a (the actual reward received plus the discounted value from the more stable target model) and the more aggressive guess of that value prior to taking action a.

The learning algorithm uses an **Epsilon Greedy algorithm** to select actions while being trained. The epsilon represents the probability of choosing an action at random instead of following what is currently expected to be the "best" action in the given state (exploration vs. exploitation).
The Agent class defines how an agent acts when asked to provide an action, learn from a time step, update the target network and manages the memory buffer.
To use Experience Replay we define a memory buffer class. An object of ReplayBuffer initializes a deque to store the experience tuples. It has methods to store a new experience tuple and to return a sample of experiences of a given batch_size.

DQN Algorithm

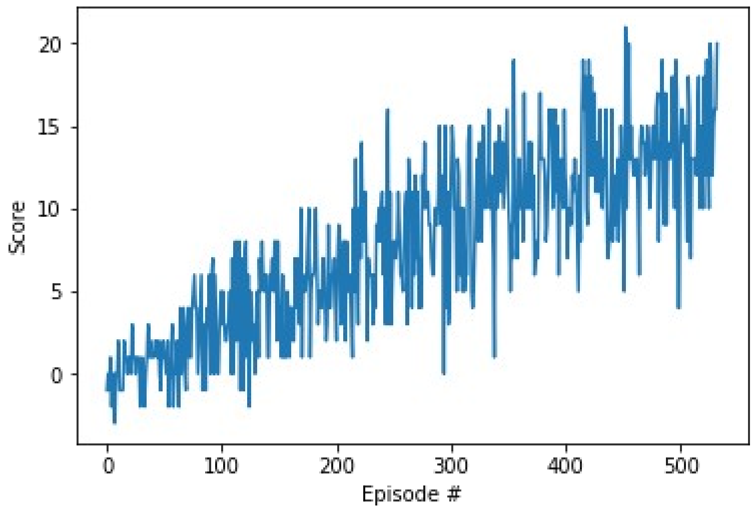
In the notebook a method called dqn contains the DQN algorithm. It returns the list of scores for all episodes and terminates when a value >= 13.0 of the average score over the last 100 episodes is achieved. Epsilon decays from a starting value eps_start=1.0 until it reaches a minimal value of eps_end=0.001 with eps_decay=0.995.
The relevant hyperparameters are:

```
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
```

The parameters were chosen as in the Lunar Lander example in Lesson 2. These proved to work very well for this project.

Result

An average score > 13.0 over the last 100 episodes was achieved at 433 episodes.



Since the benchmark mentioned that the environment was solved in less than 1800 episodes, solving it in 433 seems like a very good result.

Ideas for Future Work

The first thing that comes to mind as improvements for this project is the algorithms suggested in Lesson 2-8 (Deep Q-Learning Improvements), namely:

- Double DQN (DDQN)
- Prioritized Experience Replay
- Dueling DQN

Apart from that, playing a bit more with the hyper parameters might make a difference, specially in speed of training. I would start with **epsilon** and **TAU**. A different decay rate for epsilon might give better results. Also, adjustments in the way in which the target network is updated might give interesting results.

Another interesting thing will be to let the agent train for longer to get a maximum score achievable (surely higher than 13.0) with this hyper parameters and architecture.