1. Neural Network Architecture

Deep Q Learning uses a Deep Neural Network at its core. The model is configured to select approximate action based on a given state.

Our neural network will be used for classification task. In layman's terms we will be classifying which actions to take based on given state. This means that the input size will be our state size and the output size will be the action size.

Here the state size is 37 and the action set is 4.

Three fully connected layers with increasing size helped train faster and converged efficiently to find the best action for any state input.

2. The Agent

The system utilizes a Deep Q-Network which takes advantage of `Experience Replay` and `Fixed Q-Targets` optimization techniques.

2.1 The Hyperparameters

Below are the set of hyperparameters used in the reinforcement learning process. The values have been selected after several iteration by means of trial and error. The following configuration helps the model train faster and efficiently by taking lesser number of episodes to converge.

```
Agent Hyperparameters

    BUFFER_SIZE = int(1e5) # replay buffer size
    BATCH_SIZE = 128        # minibatch size
    GAMMA = 0.99            # discount factor
    TAU = 1e-3              # for soft update of target parameters
    LR = 5e-4               # learning rate
    UPDATE_EVERY = 5        # how often to update the network

Model Hyperparameters

    n_episodes = 2000       # maximum number of training episodes
    max_t = 1000            # maximum number of time steps per episode
    eps_start = 1.0         # starting value of epsilon, for epsilon-greedy action sele
    ction
    eps_end = 0.01          # minimum value of epsilon
    eps_decay = 0.995       # multiplicative factor (per episode) fro decreasing epsilo
    n
```

2.2 Replay Buffer

To take advantage of the `Experience Replay` functionality we need to create a buffer that will store the SARS experience tuples in a memory which is more of a `double-ended queue`

a. `__init__ constructor` It initializes the object of the class which creates

- a `double-ended queue` with a specified buffer size which is provided to the constructor to hold the experience tuples
- a `named tuple` to index the stored experiences by their titles

b. `add` method The add method appends the SARS tuples to the memory. c. `sample` method Returns a set of random tuples from the memory buffer based on the batch_size specified. d. `__len__ method` Returns the size of the internal memory.

2.3 The Agent Class

The agent will trian the Deep Q-Networks to explore or exploit accoding to a given probability which will be used to find the optimum policy that solves the RL problem.

It will consist of the following methods.

a. `__init__` constructor This constructor will take three parameters which are *state*size_, *action size*, and *seed*.

- We initialize the two Q Network instances with the same random seed and also the same weights. The first Q Network will be the online netwoek and the second one will be the target network which will compare the changes that take place in the online network and synchronizes them with the online network based on the `UPDATE_EVERY` frequency from the parameters above.
- The *Optimizer* is also defined here. We use Adam optimizer with a specified LR.
- Lastly the ReplayBuffer is also initialized to store the experiences.

b. `step` method This method adds experiences to the buffer. If our current step matches the number of steps based on which we wish to perform learning (UPDATE_EVERY), then the agent learns based on random SARS tuple samples as long as the memory contains at least as many items as our batch size (128).

c. `act` method The act method will fetch the existing state, then we make a forward pass the state to the neural network in evaluation mode and get the output with the scores for the available actions. Now according to the probabilities given by the *epsilon-greedy-policy* we will either select the highest scored action or a random action.

d. `learn` method Q-Learning (SarsaMax) dictates that the next action we take into account is the one that always maximizes the reward at the next state based on our policy. Now applying Q-Learning in a DeepQN.

So the *Q_targets_next* obtains the highest action for next state, populated by our batch_size.

We use this to calculate our *Q_targets* based on our existing rewards, so we have a complete SARSA now.

So far we obtained our SARSA and its now time to see how this change compares to our existing network weights configuration so we make any alterations necessary!

Now with a forward call to our DQN we can gather the state values, which we call *Q_expected*.

Now its time to measure the error of our Neural Network. To do so, we will use a regression error function, the *MSE (Mean of Squared Errors)*, which is used to quantify the distance between *Q_expected* and and *Q_targets*. This distance is our error.

Now all that is left is utilize this error to actually train the neural network, expecting to get a new smaller-error.

- zero the gradients

- quantify the loss to all weights using backpropagation by using *backward()* call
- and finally apply the gradient change to existing weights using the optimizer's step() to come up with new weights

Finally we've finished our learning step.

The only thing is that our target network should sync to our local (online) network until the next learning step (whenever it will reoccur based on the UPDATE_EVERY variable).

So we will *soft_update* our target network to match our local network.

e. `soft_update` method Finally we reach the point where the target network will sync to our online local network. That means that learning has concluded, and we sync the networks to prepare for our next learning. The soft update takes a constant τ (tau), which defines how much the largest network will match the weifht configuration of the local network.Passing value 1 will result in clones (fully sync), while anything lower than that will reflect slight variations between the old weights to the new ones. Setting a tau of 0 will never update the target network and it will always keep its original initial configuration.

2.4 Initializing the Agent object

Now we have all the tools to initialize the agent which will explore/exploit the banana world! We initialize the agent setting the `state_size` and `action_size`

```
agent = Agent(state_size  = brain.vector_observation_space_size,
              action_size = brain.vector_action_space_size,
              seed = 0)
```

3 Interacting with the Environment

At this stage, we will put everything together so we can interact with the environment for several episodes, until we come up with a good policy estimation via our DQN network, and eventually solve the RL task.

We will define a function *DQN* which will put everything together and drive the learning through interaction. Our DQN method will run for a maximum of 2000 episodes, after which if we have not reached our desired per 100 episodes score, the method will terminate. In a nutshell, at every episode we will:

- Reset the environment
- For each state, loop by getting the current state
  - Follow the epsilon-greedy-policy so the agent acts on that state according policy and epsilon
  - Get observation and reward
  - Use the agent's `step` function as described above to populate the experience buffer and possibly trigger the `learn` function We keep doing this until our average score over 100 episodes is equal or higher than 13 (which relates to gathering 13 yellow bananas on time).

After every 100 episodes we save a `checkpoint` if the average score has increased, but we also overwrite this checkpoint once we reach our goal. Now one might ask, what is the use to save intermediate scores. Well, sometimes you might interrupt this method earlier, and just want to check how your model performs with this "early" training (so it is there just for curiosity reasons).

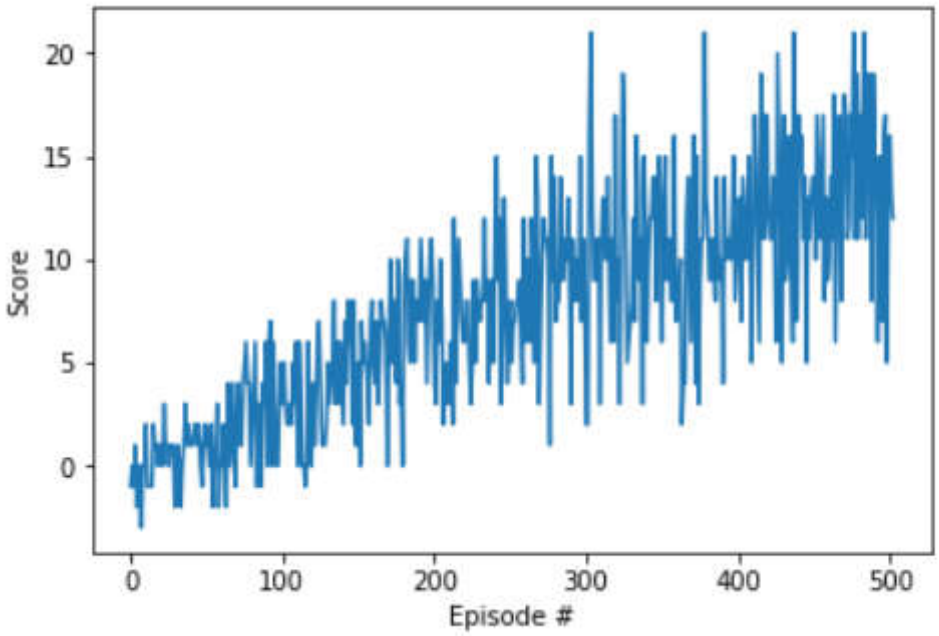
4.Performane

Given our existing approach, we reached the desired 13+ cumulative result (over 100 episodes) at episode 503.

More specifically, here comes the outcome of the DQN function which took 503 episodes to conclude:

```
Episode 100        Average Score: 1.26
Episode 200        Average Score: 4.94
Episode 300        Average Score: 8.07
Episode 400        Average Score: 10.31
Episode 500        Average Score: 12.91
Episode 503        Average Score: 13.00
Environment solved in 503 episodes!    Average Score: 13.00
```

5. Visualization



6. Further Improvements

There are several ways by which we can improve the performance of our DWN agent.

Based on our exiting setup, we can play with our hyperparameters to come up with an efficient setup which could enable the agent to train better.

There are certain structural refinements that could be done to obtain better results.

- We could use **prioritized learning** to revisit the specific actions that gave better results rather than randomly selecting SARS tuples from our experience buffer.
- A **Double DQN** could be used to minimize overextimation of action values
- **Dueling DQN** could be used to generate learning across actions without imposing any change to the underlying reinforcement learning algorithm.