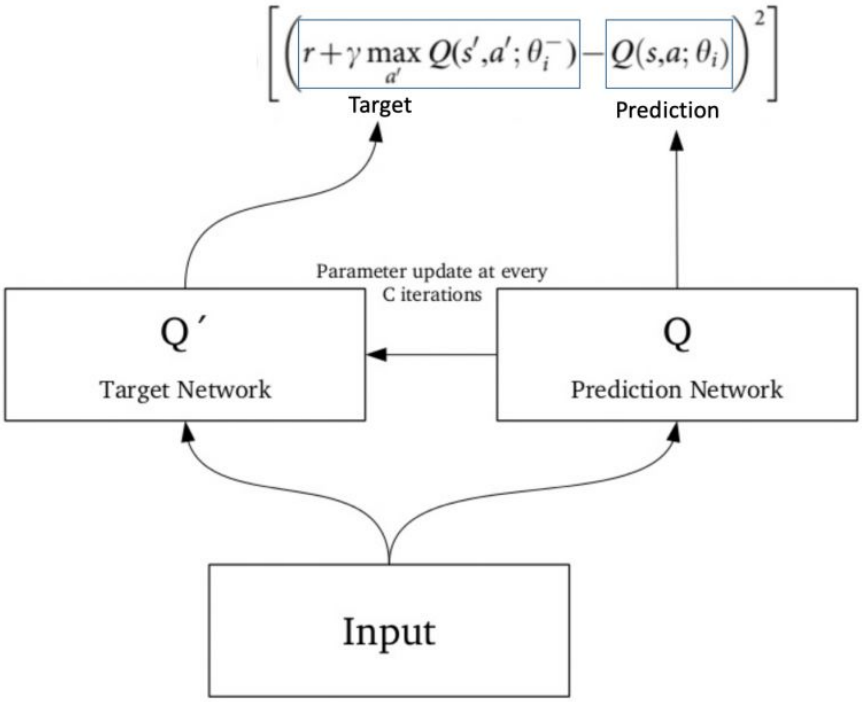
Udacity Deep RL Project 1: Navigation

In this project, we train an agent in the Unity ml-agents environment to pick up bananas. The goal of the agent is to pick up as many yellow bananas as possible avoiding the blue ones. The agent gets a +1 reward for each yellow banana it picks up and a reward of -1 for each blue banana.

Algorithm:
The agent was trained on a Deep Q network. We use two different models with the same architecture. We use one model for training and another for doing prediction. We update the target network from the prediction network after some time. We use the MSE error function to calculate the error in our network.

We store the agent's experiences at each time step in a data set called the replay memory. All of the agent's experiences at each time step over all episodes played by the agent are stored in the replay memory. A key reason for using replay memory is to break the correlation between consecutive samples.



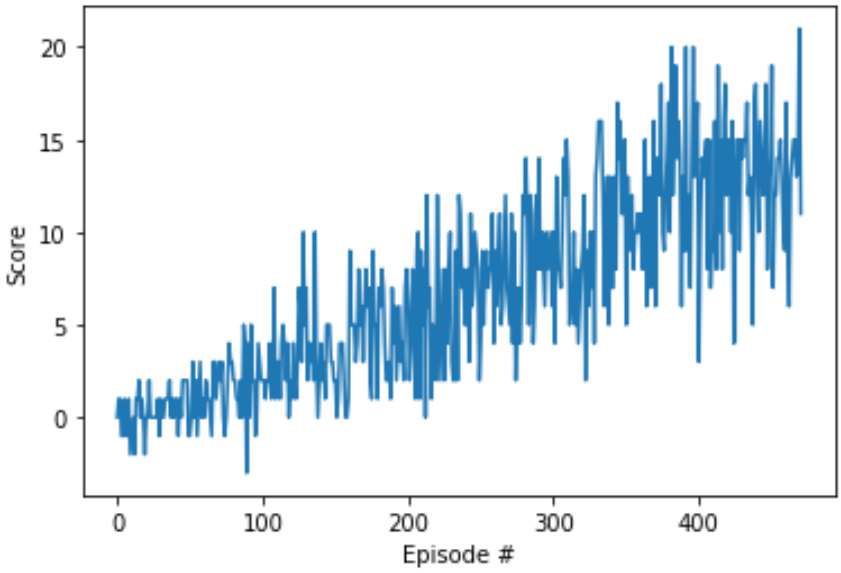
Model Architecture:
The model consists of 4 Fully-connected layers with relu activation functions.

```
self.fc1 = nn.Linear(state_size, 128)
self.fc2 = nn.Linear(128, 64)
self.fc3 = nn.Linear(64, 32)
self.fc4 = nn.Linear(32, action_size)
```

Here, state_size = 37 and action_size = 4

Hyperparameters:
Learning Rate (LR): = 5e-4
Discount Factor (GAMMA): = 0.99
Soft update of target parameters (TAU) = 1e-3
Batch size (BATCH_SIZE) = 64
Replay memory buffer (BUFFER_SIZE) = int(1e5)

Training:
The model trains for about 400 episodes and achieves an average score of +13.
The average score is calculated over 100 consecutive episodes.
The following is the plot of score for each episode. As we can see, the score increases over time as the model starts learning to pick the yellow bananas, while avoiding the blue ones.



Future improvements:

- **Prioritized Experience Replay (aka PER)**: It is an improvement over the default DQN. Prioritized sampling, as the name implies, will weigh the samples so that “important” ones are drawn more frequently for training. The magnitude of the TD error (squared) is what we want to minimize in the Bellman equation. Hence, pick the samples with the largest error so that our neural network can minimize it.

- **Double DQNs**: we use two networks to decouple the action selection from the target Q value generation. We use our DQN network to select what is the best action to take for the next state and use our target network to calculate the target Q value of taking that action at the next state. Double DQN helps us reduce the overestimation of q values and, as a consequence, helps us train faster and have more stable learning.

- **Pixel inputs**: Currently, the model takes input state of size 37. We can modify the algorithm to take inputs in the form of raw pixels and use Convolutional Neural Networks to train the agent from the pixels.

- **Dueling DQNs**: We decompose Q(s, a) into:
    - V(s): the value of being at that state
    - A(s, a): the advantage of taking that action at that state

  separate the estimator of these two elements, using two new streams:
  One to estimate V(s) and another to estimate advantage for each action A(s,a)
  We combine them using an aggregation layer to get an estimate of Q(s, a).