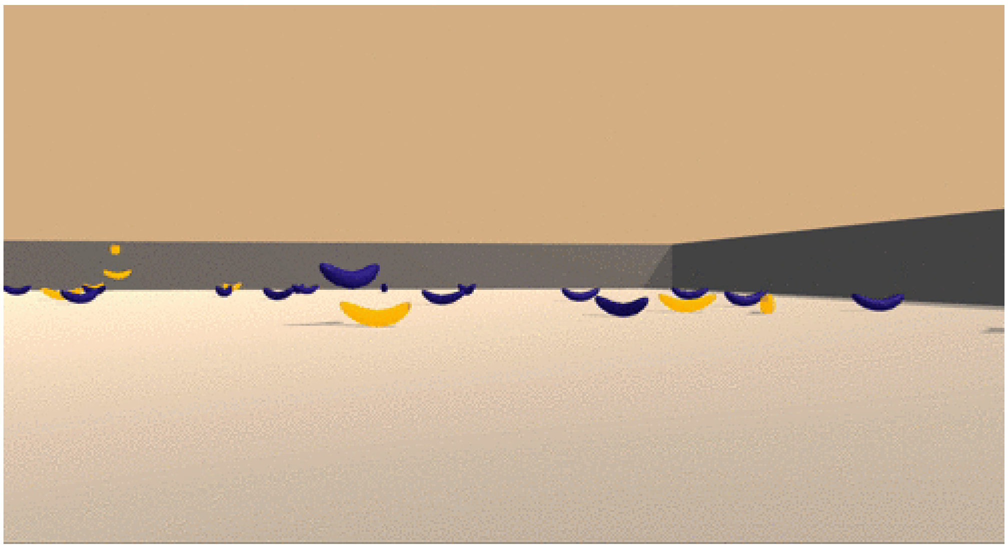
Banana Hungry Agent

Goal

The main idea of this project is to make an agent navigate a virtual world, with the goal of collecting as many yellow bananas as possible, while avoiding blue bananas.



Environment

The virtual environment is provided by Udacity and is based on Unity ML agents. This is an episodic task.

The agent gets a reward of +1 from the environment for collecting a yellow banana, while for collecting a blue banana, it gets a reward of -1.

The state space is continuous and has 37 dimensions. Agent's velocity, along with ray-based perception of objects around it's forward direction.

The action space is discrete. Namely, there are 4 of them:

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

In the benchmark implementation, the agent gets an average score (cumulative reward) of +13 over a span of 100 consecutive episodes.

Learning Algorithm

In order to solve this task, a Deep Reinforcement Learning approach has been implemented. To be precise, this project implements the **Double Deep Q-Network** (DQN) solution.

Double DQN is an improvement over the vanilla DQN algorithm. Specifically, it mitigates the problem of over-estimation of Q values.

Pseudo Code

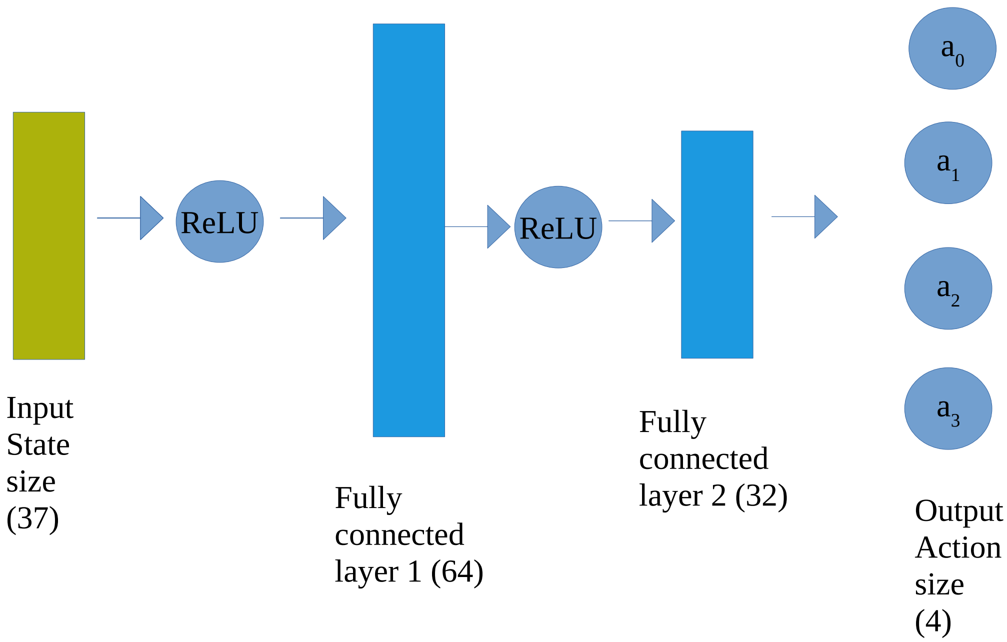
Following is the pseudo code for the Double DQN learning algorithm.

- *Init Replay Memory D with capacity N*
- *Init local action value function* ***Q*** *with random weights* ***w***
- *Init target action value function* ***Q'*** *with weights* ***w'***
- *for episode e ←1 to M:*
- *get starting state* ***s***
- *for time step t ← 1 to T:*
- *choose action* ***a*** *from* ***s*** *using π ← ε-greedy(Q(****s****,* ***a****, w))*
- *take* ***a****, observe* ***r*** *and get next state* ***s'***
- *store experience tuple (****s****,* ***a****,* ***r****,* ***s'****) in memory D*
- $\boldsymbol{s} \leftarrow \boldsymbol{s'}$
- *obtain random mini-batch of tuples from D:* $(\boldsymbol{s_j}, \boldsymbol{a_j}, \boldsymbol{r_j}, \boldsymbol{s_{j+1}})$
- *set target* $\boldsymbol{y_j} \leftarrow \boldsymbol{r_j} + \gamma * \boldsymbol{Q'}(\boldsymbol{s_j}, argmax(\boldsymbol{Q}(\boldsymbol{s_{j+1}}, \boldsymbol{a_j}, \boldsymbol{w})), \boldsymbol{w'})$
- *get expected action value* $\boldsymbol{y_j'} \leftarrow \boldsymbol{Q}(\boldsymbol{s_j}, \boldsymbol{a_j}, \boldsymbol{w})$
- *Compute loss*$(\boldsymbol{y_j}, \boldsymbol{y_j'})$*, and perform one step of gradient descent*
- *Every C steps update:* $\boldsymbol{w'} \leftarrow \boldsymbol{w}$

This algorithm has been implemented in *dqn_agent.py.*

Neural Network Architecture

For this algorithm, two networks with identical structures were required. The architecture that I used for this project is as follows:



Two fully connected layers have been used of size 64 and 32, as shown in the above figure. Note that there is no activation function leading to the final output layer. This is because the action values can be negative as well, but having a **ReLu** activation will only output positive values.

This architecture is implemented in the *model.py* file.

Hyperparameters

The DQN uses the following hyperparameters (set in `*dqn_agent.py*`):

```
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
```

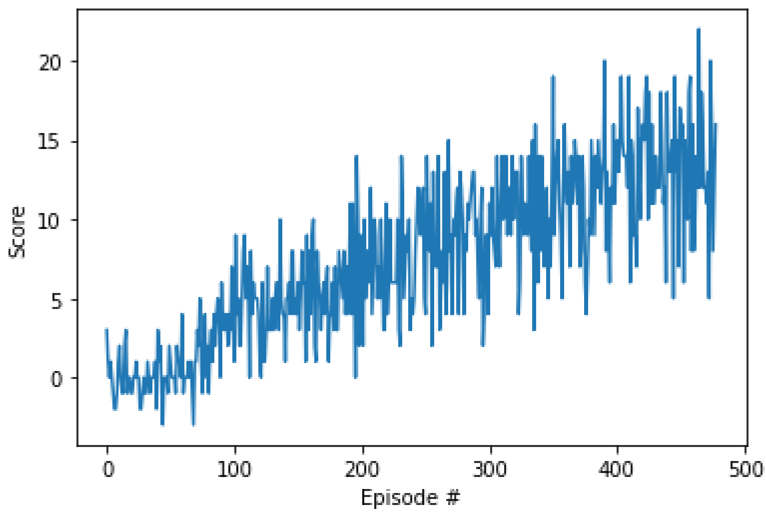
And the epsilon decay rate is taken as 0.995

Plot of Rewards

The training plot and output after running the algorithm is shown below.

```
Episode 100     Next Eps:0.6057704364907278     Average Score: 0.866
Episode 200     Next Eps:0.3669578217261671     Average Score: 5.1337
Episode 300     Next Eps:0.22229219984074702    Average Score: 7.944
Episode 400     Next Eps:0.1346580429260134     Average Score: 10.9775
Episode 478     Next Eps: 0.09108200798387568   Average Score: 13.08

Environment solved in 378 episodes!     Average Score: 13.08
Training time taken: 7.90057510137558 min
```



Ideas for Future Work

There are many areas where the solution to this problem can be improved.

1. A bigger problem to solve would be to **train the agent directly from the pixels data** (as mentioned in the Udacity course as well). A convolutional neural network needs to be trained for this purpose.
2. **Prioritized Experience Replay**: This is based on the idea that the agent can learn more effectively from some transitions than from others. Thus, the more important transitions should be sampled with higher probability.
3. **Dueling DQN**: Intuition is that the value of states don't vary a lot across actions, so it makes sense to directly estimate the value function. Hence, the final Q values is calculated as a combination of the state values and the Advantage values (calculated using a branch of the dueling network).