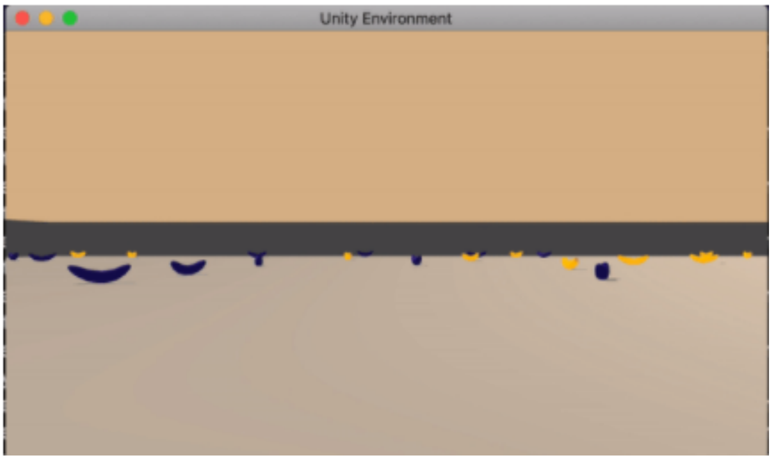
Project 1: Navigation

In this report, I am going to present about the environment and the algorithms that used to solve the navigation problem where the agent is to collect bananas. The project is to solved continuous state space of 37 dimensions, with the goal to collect yellow banana (maximizing the reward) and avoiding the blue banana (minimizing the reward). And there are 4 actions that can be choosed. Picture below is environment representation:



Learning Algorithm

This project use Deep Q-Learning as the learning algorithm. This method is based on Q-learning (temporal difference learning)

The Big idea of temporal difference learning is learn from every experience!

- Update V(s) each time we experience a transition (s, a, s', r)
- Likely outcomes s' will contribute updates more often

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

Not like Monte-Carlo methods, the Q-learning from TD learning can learn from each step, we use current Q-Value of the states to estimate future rewards.

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

Model Architecture

However, the algorithm described above in its raw form is highly unstable. Two techniques contributed significantly towards stabilizing the training:

- **Fixed Q-targets**: As can be seen from the equation above, the target during training itself is dependent on `w`, the parameter being updated. This leads to constantly moving targets and hurts training. The idea behind fixed q-targets is to fix the parameter `w` used in the calculation of the target, $\hat{Q}(s,a;w)$. This is achieved by having two separate networks, one is the online network being learned and the other being the target network. The weights of the target network are taken from the online network itself by freezing the model parameters for a few iterations and updating it periodically after a few steps. By freezing the parameters this way, it ensures that the target network parameters are significantly different from the online network parameters.

- **Experience Replay**: This is the other important technique used for stabilizing training. If we keep learning from experiences as they come, then we are basically observed a sequence of observations each of which are linked to each other. This destroys the assumption of the samples being independent. In ER, we maintain a Replay Buffer of fixed size (say N). We run a few episodes and store each of the experiences in the buffer. After a fixed number of iterations, we sample a few experiences from this replay buffer and use that to calculate the loss and eventually update the parameters. Sampling randomly this way breaks the sequential nature of experiences and stabilizes learning. It also helps us use an experience more than once.

DQN Architecture

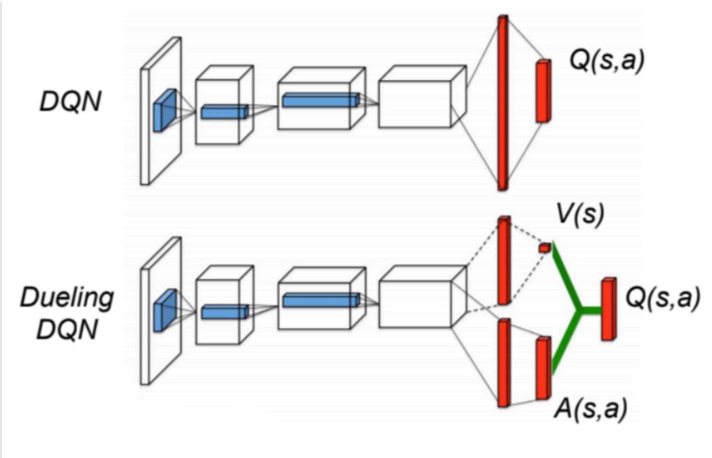
```
State --> 128 --> ReLU --> 128 --> ReLU --> action
```

Pytorch Implementation

```
    self.seed = torch.manual_seed(seed)
    self.fc1 = nn.Linear(state_size, fc1_units)
    self.fc2 = nn.Linear(fc1_units, fc2_units)
    self.fc3 = nn.Linear(fc2_units, action_size)

def forward(self, state):
    """Build a network that maps state -> action values."""
    x = F.relu(self.fc1(state))
    x = F.relu(self.fc2(x))
    return self.fc3(x)
```

Both of the above mentioned techniques were incorporated. The entire implementation was done in PyTorch. Also, various other improvements have been proposed upon the original DQN algorithm, and this repository contains the implementations of two of those:

- **Double DQN***: DQNs are known to overestimate the value function `max` because of the operator. The idea of Double DQN is to disentangle the calculation of the Q-targets into finding the best action and then calculating the Q-value for that action in the given state. The trick then is to use one network to choose the best action and the other to evaluate that action. The intuition here is that if one network chose an action as the best one by mistake, chances are that the other network wouldn't have a large Q-value for the sub- optimal action. The network used for choosing the action is the online network whose parameters we want to learn and the network to evaluate that action is the target network described earlier. More details can be found in the paper.



Hyperparameters

Hyperparameter	Value
Replay bufer size	1e5
Batch size	64
γ(discount factor)	0.99
τ	1e-3
Learning rate	5e-4
update interval	4
Number of episodes	2000
Max number of timesteps per episode	2000
Epsilon start	1.0
Epsilon minimum	0.1
Epsilon decay	0.995

Result

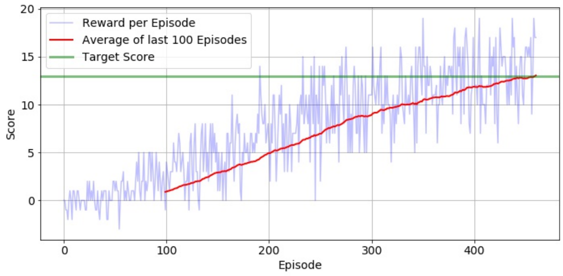
For training process the DQN need 361 episodes to finish

```
In [10]: # train the agent
         scores = dqn(n_episodes, max_t, eps_start, eps_end, eps_decay)

         Episode 100     Average score:  0.89
         Episode 200     Average score:  4.92
         Episode 300     Average score:  8.88
         Episode 400     Average score:  11.86
         Episode 461     Average score:  13.04
         Environment solved in  361 episodes!    Average Score:  13.04
```

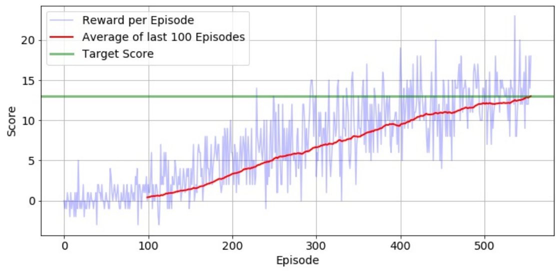
Instead for Double DQN need 456 episodes

```
In [20]: # train the agent
         scores = ddqn(n_episodes, max_t, eps_start, eps_end, eps_decay)

Episode 100     Average score:  0.41
Episode 200     Average score:  3.29
Episode 300     Average score:  6.67
Episode 400     Average score:  9.30
Episode 500     Average score:  12.14
Episode 556     Average score:  13.01
Environment solved in  456 episodes!     Average Score:  13.01
```

The best performance was achieved by double dqn with score 16 compare to original dqn with score 6



DQN



Double DQN

Performance

Below is performance for DQN and Double DQN

```
In [15]: env_info = env.reset(train_mode=False)[brain_name] # reset the environment
         state = env_info.vector_observations[0]            # get the current state
         score = 0                                          # initialize the score
         while True:
             action = agent.act(state)                      # select an action
             env_info = env.step(action)[brain_name]        # send the action to the environment
             next_state = env_info.vector_observations[0]   # get the next state
             reward = env_info.rewards[0]                   # get the reward
             done = env_info.local_done[0]                  # see if episode has finished
             score += reward                                # update the score
             state = next_state                             # roll over the state to next time step
             if done:                                       # exit loop if episode finished
                 break

         print("Score: {}".format(score))

         Score: 6.0
```

```
In [24]: env_info = env.reset(train_mode=False)[brain_name] # reset the environment
         state = env_info.vector_observations[0]            # get the current state
         score = 0                                          # initialize the score
         while True:
             action = agent.act(state)                      # select an action
             env_info = env.step(action)[brain_name]        # send the action to the environment
             next_state = env_info.vector_observations[0]   # get the next state
             reward = env_info.rewards[0]                   # get the reward
             done = env_info.local_done[0]                  # see if episode has finished
             score += reward                                # update the score
             state = next_state                             # roll over the state to next time step
             if done:                                       # exit loop if episode finished
                 break

         print("Score: {}".format(score))

         Score: 14.0
```

Ideas for future works

- To improve the performance, I am planning to Dueling DQN, Prioritized Experienced Replay.
- Train network from image data directly