Introduction

In this project, a smart agent is trained to navigate a virtual world from the input of sensory data.

- The **size of the state space is 37,** which includes the (agent's velocity, along with ray-based perception of objects around the agent's forward direction). Complete definitions of states or how the states are being implemented are unknown. for instance, like velocity only have the linear direction? or is there any angular velocity? or is there any coordinate of pre-defined yellow and blue bananas in the 2D / 3D map being provided? But the honest answer is, we don't know.

  It just looks like a black box with which we pass through these "nodes" to be integrated with ours to be trained weight, in a hope that to find out the best combination of weight for each node in each layer of the neural network to have a good approximation to the ideal policy.

- **Reward:** {Blue banana collected by agent: -1; No banana collected: 0; Yellow banana collected: +1;}

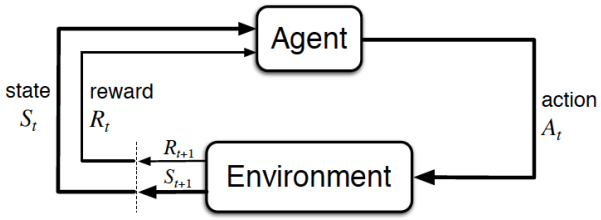

Image 1: Overview Diagram of Reinforcement Learning [1]

Learning Algorithm

Reinforcement learning is a branch of machine learning where an agent outputs an action and the environment returns an observation or, the state of the system and a reward. The goal of an agent is to best determine the best action to take. Trying to maximize the overall or total reward.

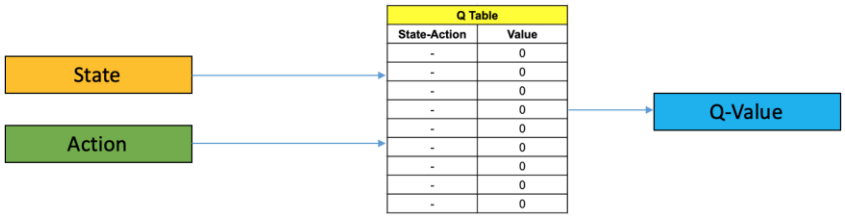

Image 2: Reinforcement Learning by the Value Based Method – Q Learning [2]

Source: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

For the deep reinforcement learning, it is using nonlinear function approximations to calculate the value action based directly on observation from the environment. It is represented as a Deep Neural Networks and through the learning to find the optimal parameters for these function approximators. Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values. To address these instabilities, the **Deep Q-Learning** algorithm is used which has two key features:

- Experience Replay
- Fixed Q-Targets

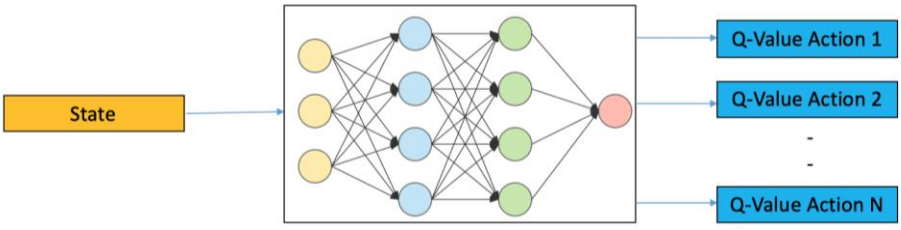

Image 3: Deep Reinforcement Learning by the Value Based Method – DQN [2]

Source: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Experience Replay

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a **replay buffer** and using **experience replay** to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The **replay buffer** contains a collection of experience tuples (*S*, *A*, *R*, *S*′). The tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as **experience replay**. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

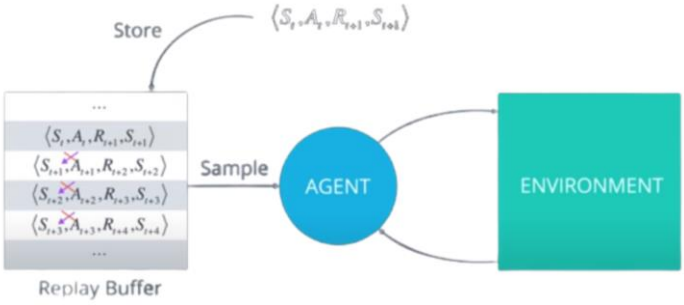

Image 4: Illustration of Experience Replay [2]

Fixed Q-Targets

In Q-Learning, we update a guess with a guess, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters *w* in the network $\hat{q}$ to better approximate the action value corresponding to state *S* and action *A* with the following update rule:

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

where w^- are the weights of a separate target network that are not changed during the learning step. And (*S*, *A*, *R*, *S*′) is an experience tuple

Model architectures and Hyper-parameters

The designed Deep-Q-Networks **Model** for this project is basically made of three fully connected (hidden) layers with the RELU activation functions after the 1st and 2nd hidden layers.

```
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        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)
```

The hyper-parameters of the **agent** are set as follows:

```
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
```

- **BUFFER_SIZE = 10,000**, this is the maximum size of buffer to store experience tuples. It is set to high value so that there is more set of previous experience the agent can sample and learn from.
- **Minibatch = 64**, this is where we can take the advantage of neural network computational power / strategy to let the agent learn several experiences set in parallel while reducing the chance of harmful correlation.
- **GAMMA = 0.99**, this is the discount factor where it should be set higher so that the agent is more focused on the immediate reward rather than the distant one.

```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
```

- **eps_start = 1.0**, set to 1 initially so that the agent is biased toward the exploration (where the probability of taking each of the actions is equally distributed) since the agent does not know the policy yet.
- **eps_decay = 0.995**, a factor to constantly decay the epsilon as the agent learn the better policies so that the agent gradually favors the epsilon-greedy actions selection (exploitation) over the random actions (exploration).

Plot of Rewards

The result of the training shows that the agent is able to receive an average reward (over 100 episodes) of at least +13 after the 600th episode to solve the environment.

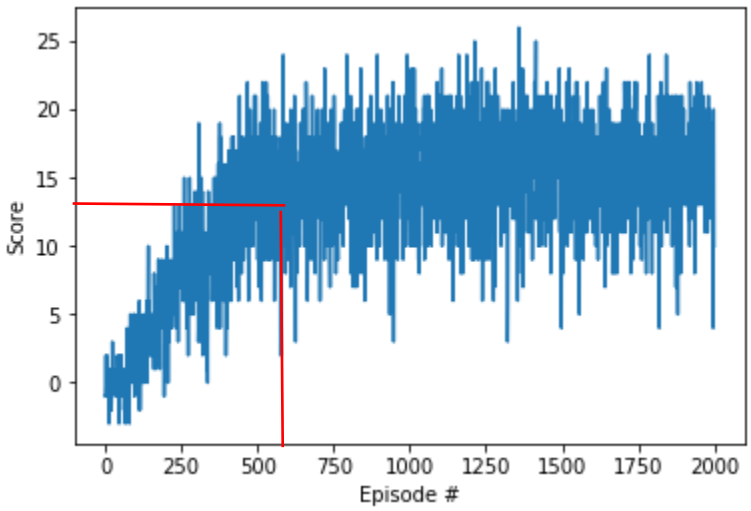

Image 5: Plot of Training Results

Idea for Future Work

1. Deploy **Double Q-learning** as it can make estimation more robust by selecting the best action using one set of parameters w , but evaluating it using a different set of parameters w'.
2. Some of the previous experiences may be more important for learning than others, and they might occur infrequently. Hence, this is where **Prioritized Experience Replay** should come into place.

Reference

1. https://learn.udacity.com/nanodegrees/nd893
2. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/