Navigator Project Report

Objective:

For this project, you will train an agent to navigate (and collect bananas!) in a large, square world.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

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

The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

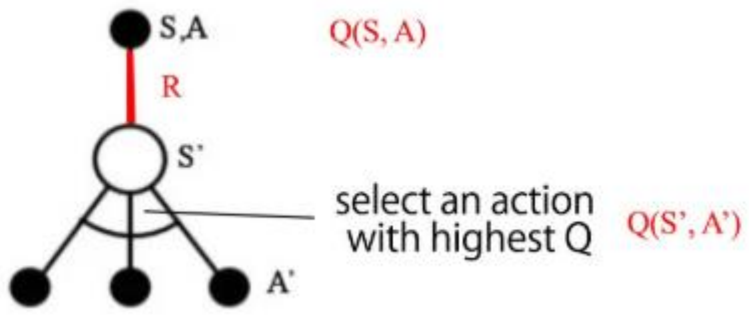
Algorithm:

Deep Q Network (DQN):

First let’s talk about Q learning which is the foundation of DQN.

Q-learning learns the action-value function Q(s, a): how good to take an action at a particular state. For example, for the board position below, how good to move the pawn two steps forward. Literally, we assign a scalar value over the benefit of making such a move.

Q learning is about updating Q table and take action with highest Q value.



However, there is some challenge with Q learning such as Target unstable. For example, we build a deep network to learn the values of Q but its target values are changing, as we know things better. The target values for Q depends on Q itself, we are chasing a non-stationary target. To solve this problem, we introduced deep neural network to the Q learning.

Deep Q-Learning with experience replay:

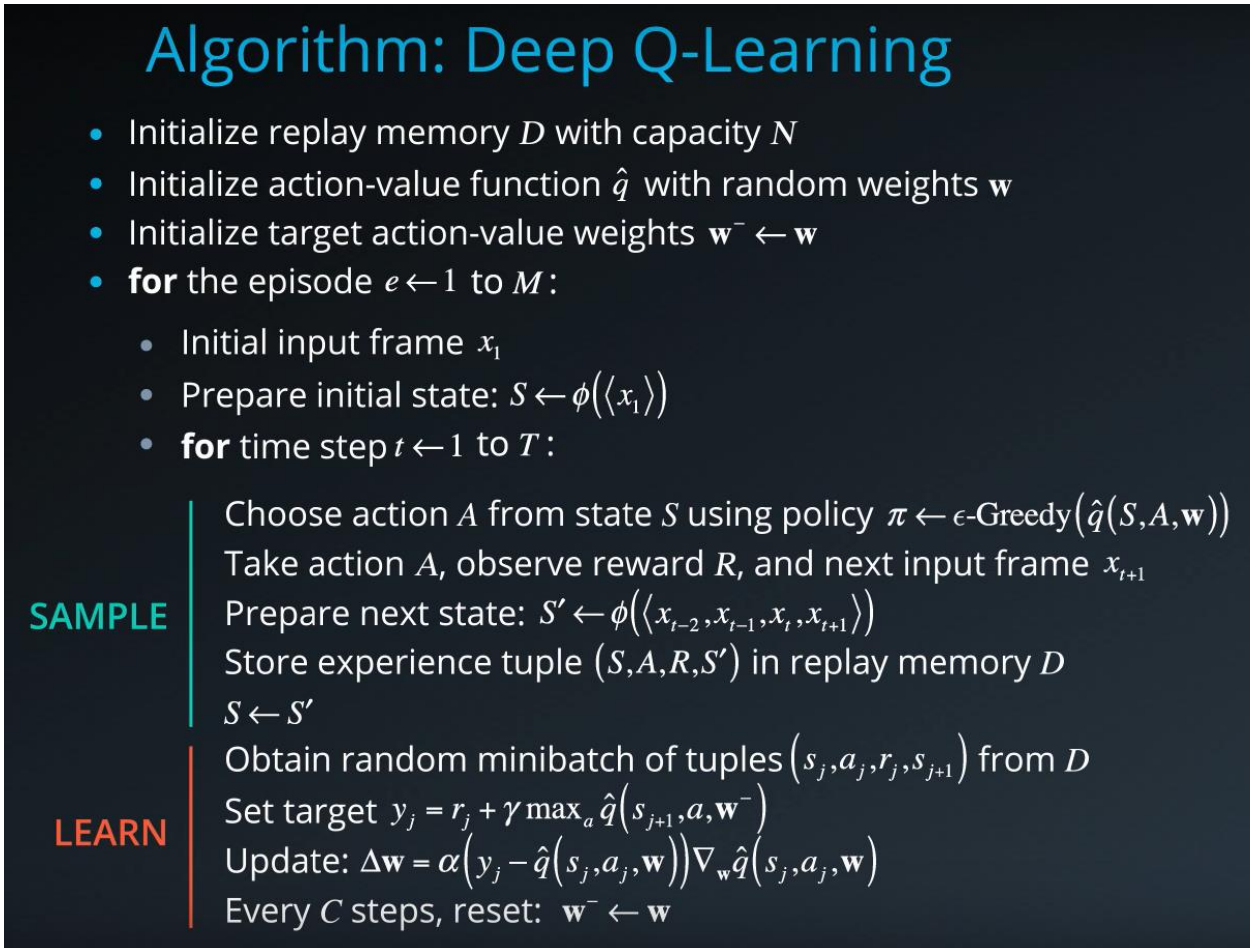
In this project, we put past states in a buffer and sampled a mini-batch of size 32 to train the deep network. This will generate a good dataset with data more independent to each other.

Target network:

We created two deep network, target network and labeled network. We use the first one to retrieve Q values while the second one to remember all the updates. After certain steps, we will partially update the first network with the second one. The purpose is to fix the Q-value targets temporarily so we don't have a moving target to chase. The update of weights are shown below:

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

The whole idea of the Deep Q-network is as below:



Algorithm: Deep Q-Learning

```
• Initialize replay memory $D$ with capacity $N$
• Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
• Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
• for the episode $e \leftarrow 1$ to $M$:
    • Initial input frame $x_1$
    • Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
    • for time step $t \leftarrow 1$ to $T$:
SAMPLE  | Choose action $A$ from state $S$ using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S,A,\mathbf{w}))$
        | Take action $A$, observe reward $R$, and next input frame $x_{t+1}$
        | Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
        | Store experience tuple $(S,A,R,S')$ in replay memory $D$
        | $S \leftarrow S'$
LEARN   | Obtain random minibatch of tuples $(s_j, a_j, r_j, s_{j+1})$ from $D$
        | Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
        | Update: $\Delta\mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$
        | Every $C$ steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$
```

Double Q learning:

In this project, I also used double Q network to improve the model.

In Q learning, we used the following way to update target value for Q:

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)\,.$$

However, the max operation creates a positive bias towards the Q estimations. By theory and experiments, DQN performance improves if we use the network θ to greedy select the action and the target network θ- to estimate the Q value.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\ D}\left(r + \gamma Q(s', \arg\max_{a'} Q(s',a';\theta); \theta_i^-) - Q(s,a;\theta_i)\right)^2$$

Therefore, we use prioritized experience replay in the double DQN.

We can select transition from the buffer:

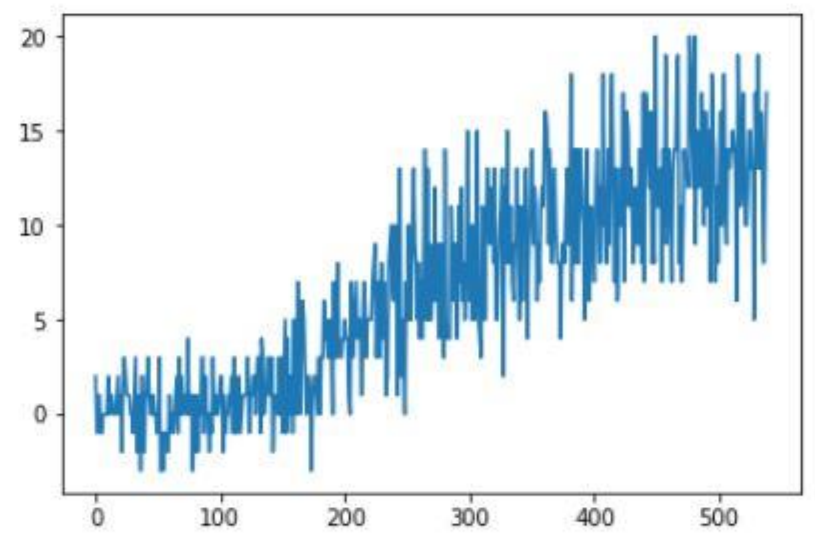
- pick transitions with higher error more frequently, or
- rank them according to the error value and select them by rank (pick the one with higher rank more often).

Code implementation:

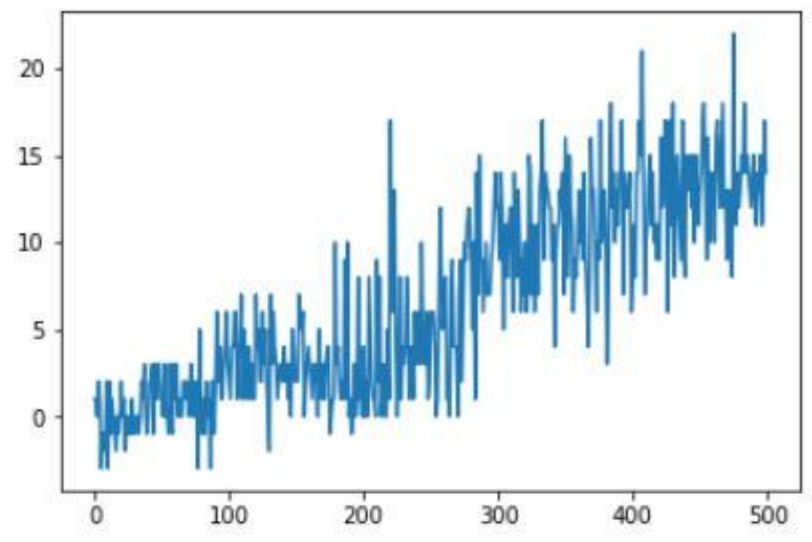
The code consist of:

- **model.py**: define the neural network of model in PyTorch. The DQN is composed of:
    - the input layer which size depends of the state_size parameter passed in the constructor
    - 2 hidden layer with relu activation function
    - The output layer represent the action to be taken under input state
- **dqn_agent.py**
    - In this file ,the DQN agent is defined and the following functions are written.
        - Step: save experience in replay buffer and "learn" in every certain steps
        - Act: Return state for given state per current policy
        - Learn: Update value parameters.
        - Soft_update: copy part of parameters from local network to target network
    - A fixed size replayed buffer to save experiences
        - Add: add experience to the buffer
        - Sample: randomly sample batch size of experience from buffer
        - Len: return length of buffer

- **Navigation.ipynb**
    - In this file the DQN and double DQN models are trained and the targeted score of 13 are achieved.
    - The following plots shows the change of score with episodes.



Simple DQN (x axis is episodes, y axis is score)



Double DQN

Future work:

1. Implement DQN algorithm using pixels.

Implement DQN from pixel is amazing. We can mock the game playing as human being from scratch, which is amazing.

2. Implement Dueling network for Deep Reinforcement Learning.

"Dueling DQN presents a change in the network structure comparing to DQN.

Dueling DQN uses a specialized Dueling Q Head in order to separate Q to an A (advantage) stream and a V stream. Adding this type of structure to the network head allows the network to better differentiate actions from one another, and significantly improves the learning.

In many states, the values of the different actions are very similar, and it is less important which action to take. This is especially important in environments where there are many actions to choose from. In DQN, on each training iteration, for each of the states in the batch, we update the :ath:`Q` values only for the specific actions taken in those states. This results in slower learning as we do not learn the Q values for actions that were not taken yet. On dueling architecture, on the other hand, learning is faster - as we start learning the state-value even if only a single action has been taken at this state."