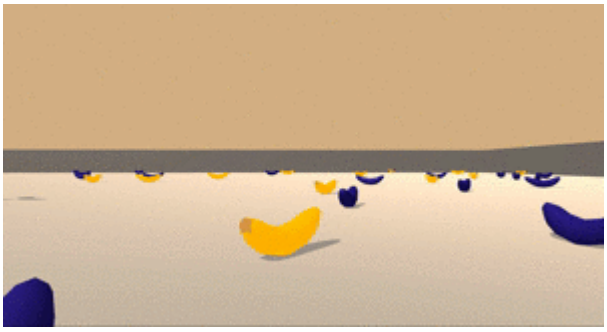
# Navigation

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



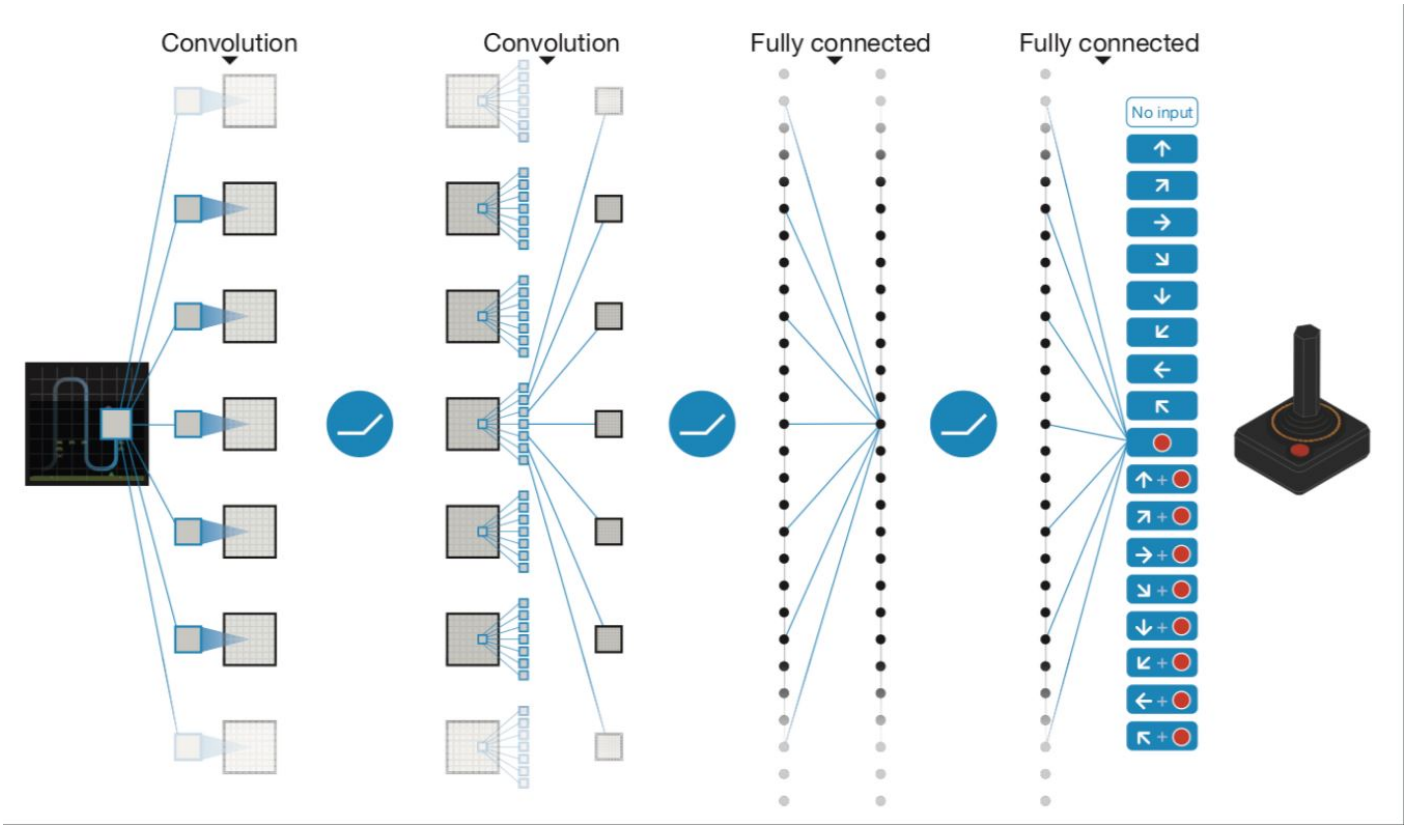
A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around 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.

## Agent implementation

### Deep Q-Networks

The DQN architecture is composed of a couple of convolutional layers and no fully-connected layers, which takes the state as input, and returns the corresponding predicted action values for each possible game action.




**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.

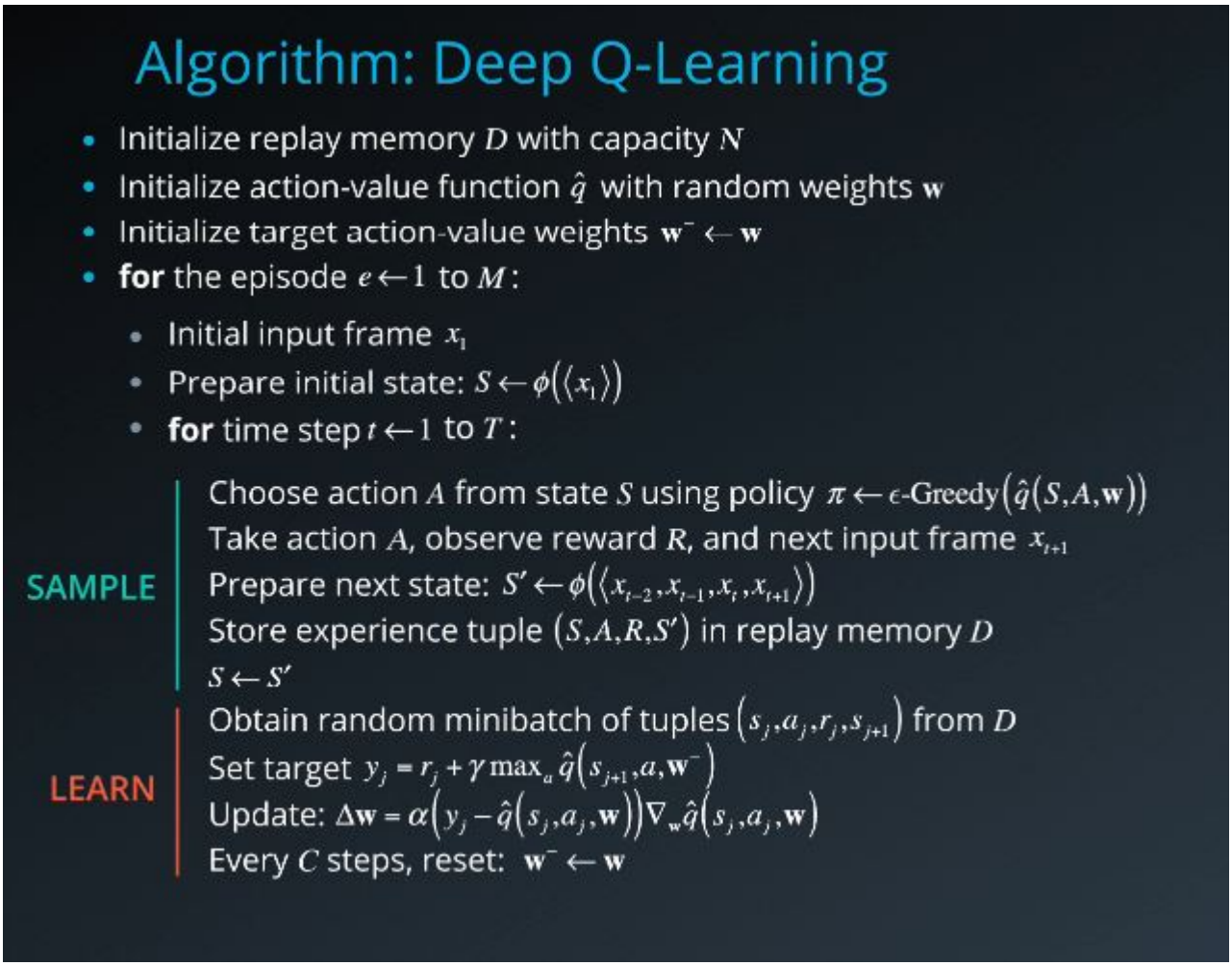
**Fixed Q-Targets**

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 ww in the network q to better approximate the action value corresponding to state S and action A with the following update rule:

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

Enjoy first-class Markdown support with easy access to Markdown syntax and convenient keyboard shortcuts.

**Algorithm**




## Code

Code consists of `Navigation.ipynb`

### QNetwork class

This is fully connected Deep Neural Network implemented using PyTorch. This network predicts the action to perform depending on the observed state.

- It consists of 2 hidden fully connected layers of 1024 cells each

### DQN Agent

- initialize:
  - Replay buffer
  - *Local* Q-Network and *target* Q-Network
- step:
  - Save experience tuple (S, A, R, S', done) in replay memory
  - Update the target network, every 4 steps, with the weights from local network
- act:
  - Returns actions for given state as per current policy (action selected using epsilon-Greedy policy)
- learn:
  - Update value parameters using given batch of experience tuples (S, A, R, S', done)
- soft update:
  - updates the value of target network with values from local network

### Replay Buffer

- add:
  - Add a new experience to replay memory
- sample:
  - Randomly sample a batch of experiences from memory

## DQN parameters

Neural network uses Adam optimizer with following parameters

```
BUFFER_SIZE = int(1e5) # replay buffer size
```

```
BATCH_SIZE = 64 # minibatch size
```

```
GAMMA = 0.995 # discount factor
```

```
TAU = 1e-3 # for soft update of target parameters
```

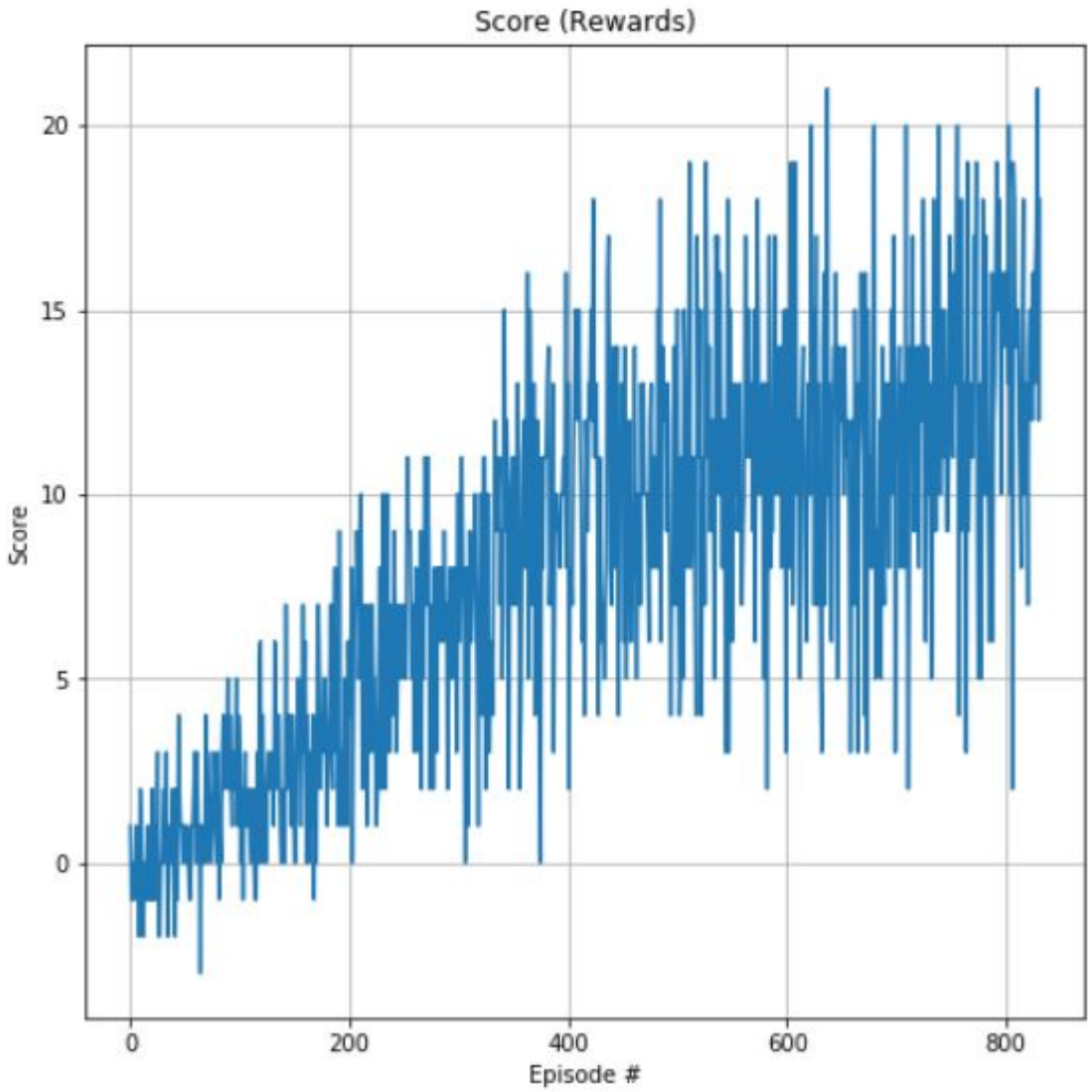
```
LR = 5e-4 # learning rate
```

```
UPDATE_EVERY = 4 # how often to update the network
```

## Results

```
Episode 100     Average Score: 0.88
Episode 200     Average Score: 2.69
Episode 300     Average Score: 5.90
Episode 400     Average Score: 8.38
Episode 500     Average Score: 10.17
Episode 600     Average Score: 11.00
Episode 700     Average Score: 10.92
Episode 800     Average Score: 12.30
Episode 832     Average Score: 13.04
Environment solved in 732 episodes!     Average Score: 13.04

Total Training time = 19.1 min
```




## Future Ideas

### Raw pixels

Instead of using internal state we can use raw pixels to trian the agent

## Double DQN

The popular Q-learning algorithm is known to overestimate action values under certain conditions. It was not previously known whether, in practice, such over-estimations are common, whether they harm performance, and whether they can generally be prevented.

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. We therefore propose to evaluate the greedy policy according to the online network, but using the target network to estimate its value.

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

## Prioritized Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. It is better to prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently.

The central component of prioritized replay is the criterion by which the importance of each transition is measured. One idealised criterion would be the amount the RL agent can learn from a transition in its current state (expected learning progress). While this measure is not directly accessible, a reasonable proxy is the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is: specifically, how far the value is from its next-step bootstrap estimate (Andre et al., 1998). This is particularly suitable for incremental, online RL algorithms, such as SARSA or Q-learning, that already compute the TD-error and update the parameters in proportion to δ.

## Dueling DQN

Dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm

The lower layers of the dueling network are convolutional as in the original DQNs (Mnih et al., 2015). However, instead of following the convolutional layers with a single sequence of fully connected layers, we instead use two sequences (or streams) of fully connected layers. The streams are constructed such that they have they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q function. As in (Mnih et al., 2015), the output of the network is a set of Q values, one for each action.