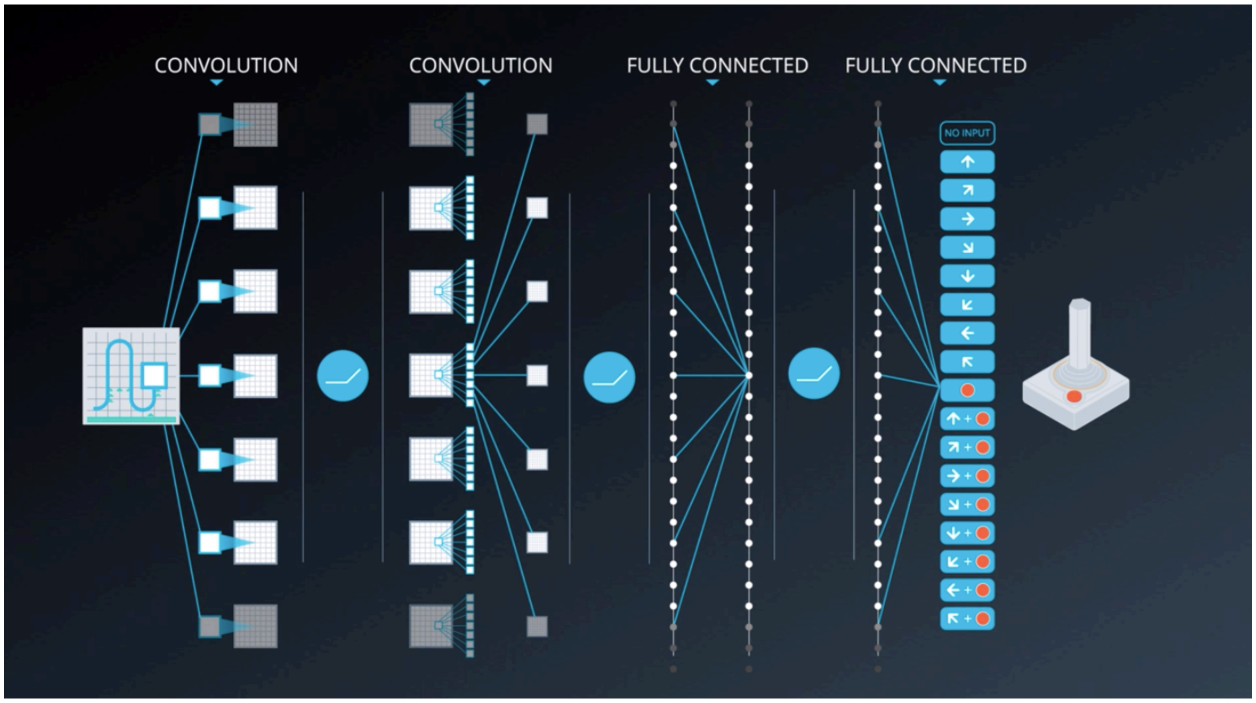
Deep Q-Network (DQN) Report

In 2015, Deep Mind design an algorithm that could learn to play video games better than any humans. They called this algorithm Deep Q-Network. This algorithm can play bunch of Atari games with just looking at their raw pixel data and nothing else. As its name suggests, there is a deep neural network at the heart of this algorithm which acts as a function approximator. The steps are as follows:

1. The agent receives image of video game at each time
2. Then it produces a vector of action values which the maximum number indicates the action to take.
3. Then it is fed back the change in game score at each time step

In this way, at the very beginning, we are going to have actions that are totally random. But over time, the agent learns to chooses better action that will result in maximising the reward.



Training such a network requires a lot of data, but even then, it is not guaranteed to converge on the optimal value function. In fact, there are situations where the network weights can oscillate or diverge, due to the high correlation between actions and states. This can result in a very unstable and ineffective policy. In order to overcome these challenges, we should slightly modify the base Q-learning algorithm. We will take a look at the following two techniques which are the most important ones:

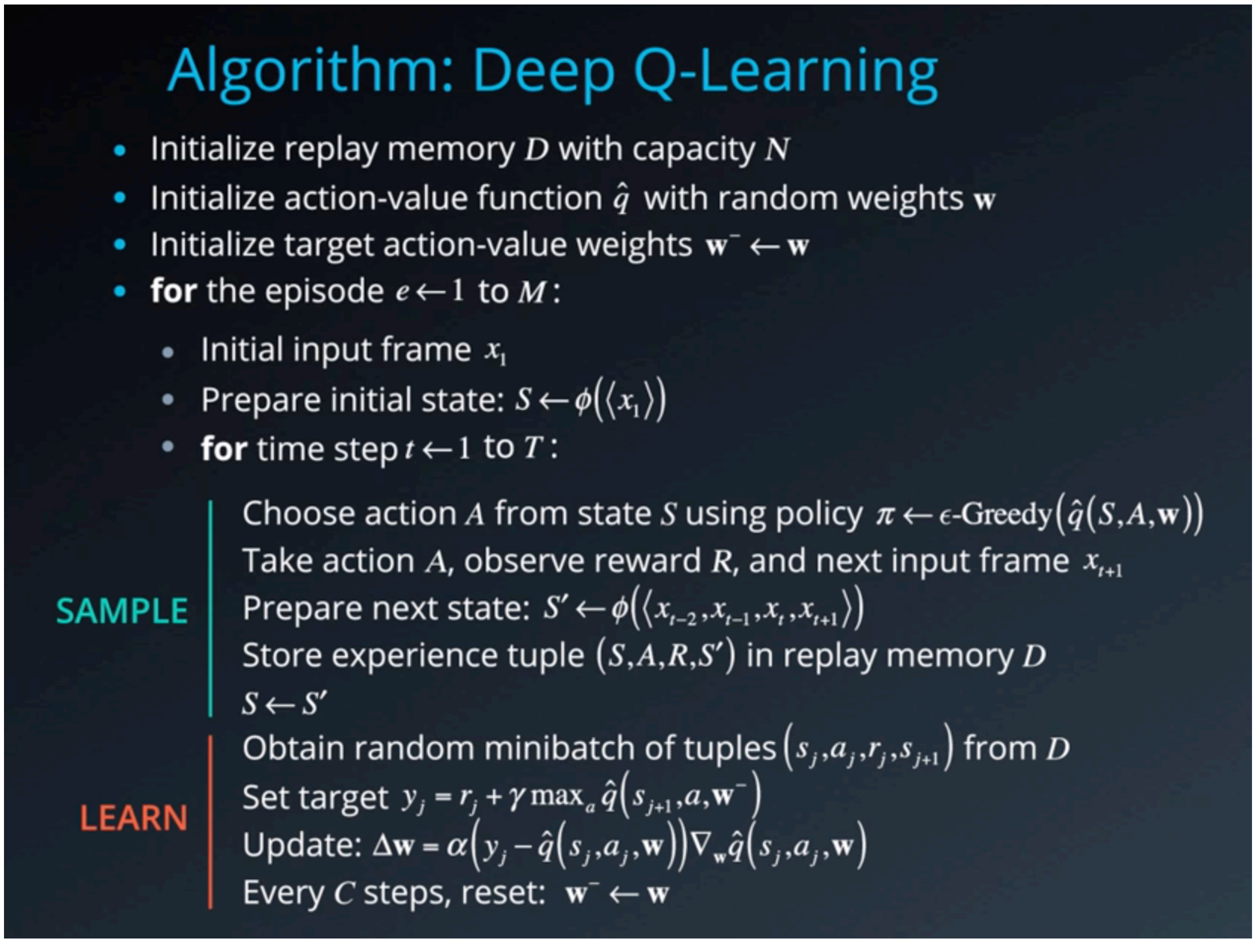
- Experience replay
- Fixed Q-Target

There are two main processes that are interleaved in this Deep Q-Learning Algorithm:

1. One, is where we **sample** the environment by performing actions and store away the observed experienced tuples in a replay memory.
2. The other is where we select the small batch of tuples from this memory, randomly, and **learn** from that batch using a gradient descent update step.

These two processes are not directly dependent on each other. So, you could perform multiple sampling steps then one learning step, or even multiple learning steps with different random batches. The rest of the algorithm is designed to support these steps.

In the beginning you need to initialize an empty replay memory D. Note that memory is finite, so you may want to use something like a circular Q that retains the N most recent experience tuples. Then, you also need to initialize the parameters or weights of your neural network. There are certain best practices that you can use, for instance, sample the weights randomly from a normal distribution with variance equal to two by the number of inputs to each neuron. These initialization methods are typically available in modern deep learning libraries like Keras and TensorFlow, so you won't need to implement them yourself. To use the fixed Q targets technique, you need a second set of parameters w^- which you can initialize to w. Now, remember that this specific algorithm was designed to work with video games. So, for each episode and each time step t within that episode you observe a raw screen image or input frame x_t which you need to convert to grayscale crop to a square size, etc. Also, in order to capture temporal relationships you can stack a few input frames to build each state vector. Let's denote this pre-processing and stacking operation by the function phi, which takes a sequence of frames and produces some combined representation. Note that if we want to stack say four frames will have to do something special for the first, three time steps. For instance, we can treat those missing frames as blank, or just used copies of the first frame, or we can just skip storing the experience tuples till we get a complete sequence. In practice, you won't be able to run the learning step immediately. You will need to wait till you have sufficient number of tuples in memory. Note that we do not clear out the memory after each episode, this enables us to recall and build batches of experiences from across episodes. There are many other techniques and optimizations that are used in the DQN paper, such as reward clipping, error clipping, storing past actions as part of the state vector, dealing with terminal states, digging epsilon over time, et cetera. I encourage you to read the paper, especially the methods section before trying to implement the algorithm yourself. Note that you may need to choose which techniques you apply and adapt them for different types of environments.



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: $\mathbf{w}^- \leftarrow \mathbf{w}$
```

The code used here is derived from the "Lunar Lander" tutorial from the Deep Reinforcement Learning Nanodegree, and has been slightly adjusted for being used with the banana environment. The code consist of :

- model.py : In this python file, a PyTorch QNetwork class is implemented. This is a regular fully connected Deep Neural Network using the PyTorch Framework. This network will be trained to predict the action to perform depending on the environment observed states. This Neural Network is used by the DQN agent and is composed of :
    - the input layer which size depends of the state_size parameter passed in the constructor
    - 2 hidden fully connected layers of 1024 cells each
    - the output layer which size depends of the action_size parameter passed in the constructor
- dqn_agent.py : In this python file, a DQN agent and a Replay Buffer memory used by the DQN agent) are defined.
    - The DQN agent class is implemented, as described in the Deep Q-Learning algorithm. It provides several methods :
        - constructor :
            - Initialize the memory buffer (*Replay Buffer*)
            - Initialize 2 instance of the Neural Network : the *target* network and the *local* network

        - step() :
            - Allows to store a step taken by the agent (state, action, reward, next_state, done) in the Replay Buffer/Memory
            - Every 4 steps (and if their are enough samples available in the Replay Buffer), update the *target* network weights with the current weight values from the *local* network (That's part of the Fixed Q Targets technique)
        - act() which returns actions for the given state as per current policy (Note : The action selection use an Epsilon-greedy selection so that to balance between *exploration* and *exploitation* for the Q Learning)
        - learn() which update the Neural Network value parameters using given batch of experiences from the Replay Buffer.
        - soft_update() is called by learn() to softly updates the value from the *target* Neural Network from the *local* network weights (That's part of the Fixed Q Targets technique)
    - The ReplayBuffer class implements a fixed-size buffer to store experience tuples (state, action, reward, next_state, done)
        - add() allows to add an experience step to the memory
        - sample() allows to randomly sample a batch of experience steps for the learning
- DQN_Banana_Navigation.ipynb : This Jupyter notebooks allows to train the agent. More in details it allows to :
    - Import the Necessary Packages
    - Examine the State and Action Spaces
    - Take Random Actions in the Environment (No display)
    - Train an agent using DQN
    - Plot the scores

The DQN agent uses the following parameters values (defined in dqn_agent.py)

```
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
```

The Neural Networks use the following architecture :
Input nodes (37) -> Fully Connected Layer (1024 nodes, Relu activation) -> Fully Connected Layer (1024 nodes, Relu activation) -> Ouput nodes (4)

The Neural Networks use the Adam optimizer with a learning rate LR=5e-4 and are trained using a BATCH_SIZE=64

There are few improvements that we can do for having a better result using DQN. These improvements are as follows:

- **Double DQN:** Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.

- **Prioritized experience replay:** Deep Q-Learning samples experience transitions uniformly from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.

- **Dueling DQN:** Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values for each action. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action.