Deep Reinforcement Learning– Project 1: Navigation

Abhranil Chandra

This document presents a technical description of the Navigation project in the context of the Deep Reinforcement Learning Nanodegree from Udacity.

1. Summary:Banana Hungry Agent

In this project, the goal is to train an agent to navigate a virtual world and collect as many yellow bananas as possible while avoiding blue bananas. The agent learns by itself, that is by **trial and error**, to collect special types of bananas, the yellow ones while avoiding the blue ones in a restricted environment. The goal is to maximize the overall collected yellow bananas in a given episode so there are elements of **delayed reward**.These two characteristics – trial-and-error search and delayed reward – suggests that our problem is well framed to be categorized as a **Reinforcement Learning (RL)** problem.

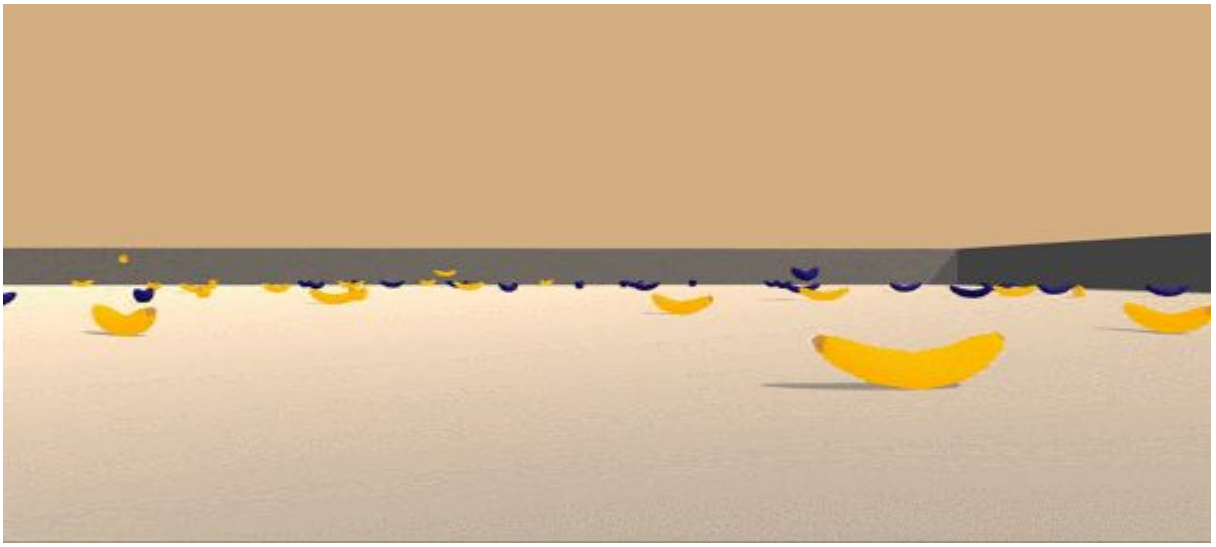

Figure 1: Overview of the environment

2. Environment Details

The environment is based on **Unity ML-agents**
Note: The project environment provided by Udacity is similar to, but not identical to the Banana Collector environment on the Unity ML-Agents GitHub page.

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API.

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 the 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, the agent must get an average score of +13 over 100 consecutive episodes**.

3. From RL to Deep RL

In a Reinforcement Learning (RL) setting, there are **Monte Carlo** and **Temporal-Difference** (TD) methods we can choose from.
While **Monte-Carlo** approaches requires we run the agent for the whole episode before making any decisions, this solution is no longer viable with **continuous tasks** that does not have any terminal state, as well as **episodic tasks for cases when we do not want to wait for the terminal state** before making any decisions in the environment's episode.

This is where **Temporal-Difference** (TD) Control Methods steps in, they update estimates based in part on other learned estimates, without waiting for the final outcome. As such, TD methods will update the **Q-table** after every time step.
The Q-table is used to **approximate the action-value function q_π** for the policy π.
You can find an example below of a Q-table that considers which action to take depending on the environment.

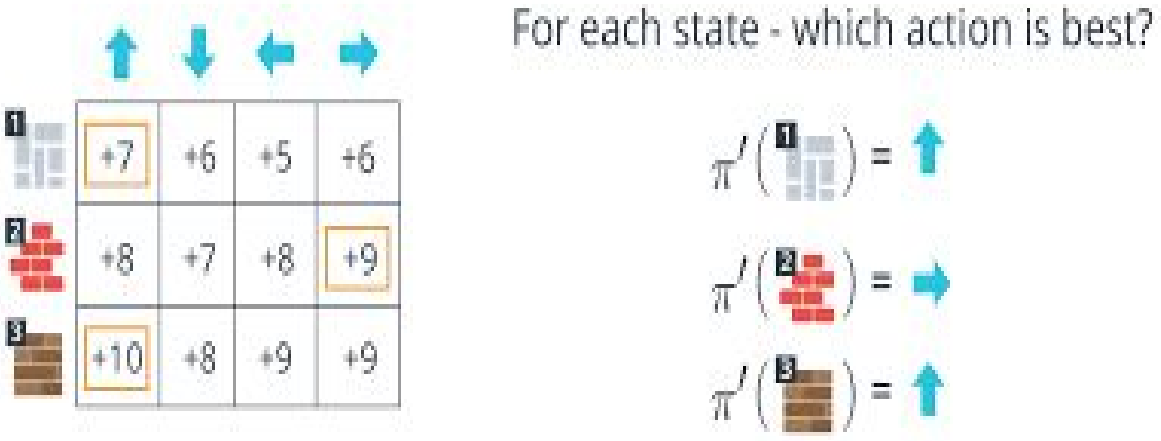

Figure 2: Q-table Example

The Q-Learning is one effective TD method that uses an update rule that attempts to approximate the optimal value function at every time step :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)$$

One caveat with Q-Learning is that the use of the Q-table assumes that we are using only **discrete** states and actions. This is a major assumption since in general, most actions that need to take place in a physical environment are continuous in nature. Moreover, the states can also be continuous, for instance sensors' information as well as visual by taking images as the environment.

To address this issue, the use of **function approximation** created a major breakthrough, since this is exactly the purpose of **neural networks** to create such approximations.

In the standard problem of Navigation, we are given 37, continuous states are 4, discrete actions so a **fully-connected neural network** is appropriate for the optimal-value function q_*.

4. Agents Implementation(Algorithm)

This project implements a ***Value Based*** method called **Deep Q-Networks**.

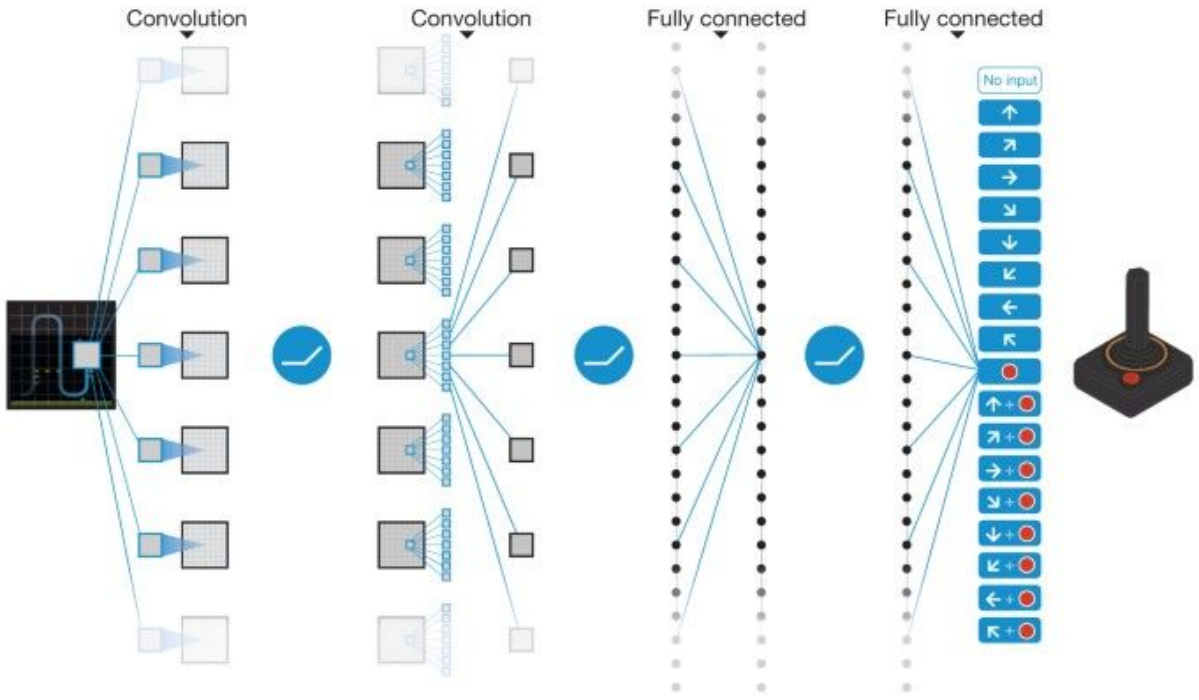

Figure 3: Deep Q-Network

Deep Q Learning combines 2 approaches :

- A Reinforcement Learning method called **Q-Learning (aka SARSA max)**
- A Deep Neural Network to learn a Q-table approximation (action-values)

Especially, this implementation includes the 2 major training improvements by Deepmind and described in their Nature publication : "Human-level control through deep reinforcement learning (2015)"

- **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.

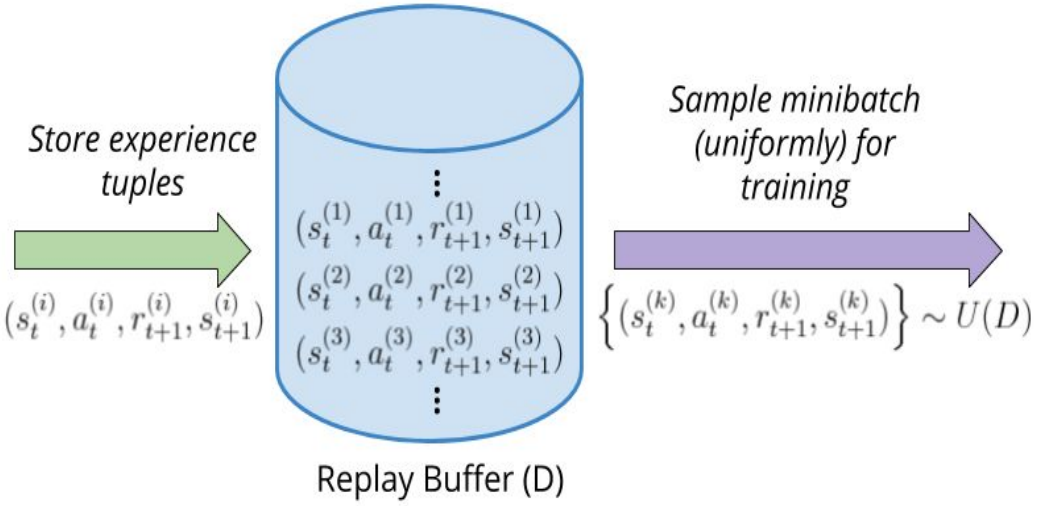

Figure 4: Replay Buffer

- **Fixed Q Targets**

  In Q-Learning, when we want to compute the TD error, we compute the the difference between the TD target and the current predicted Q-value (estimation of Q) .

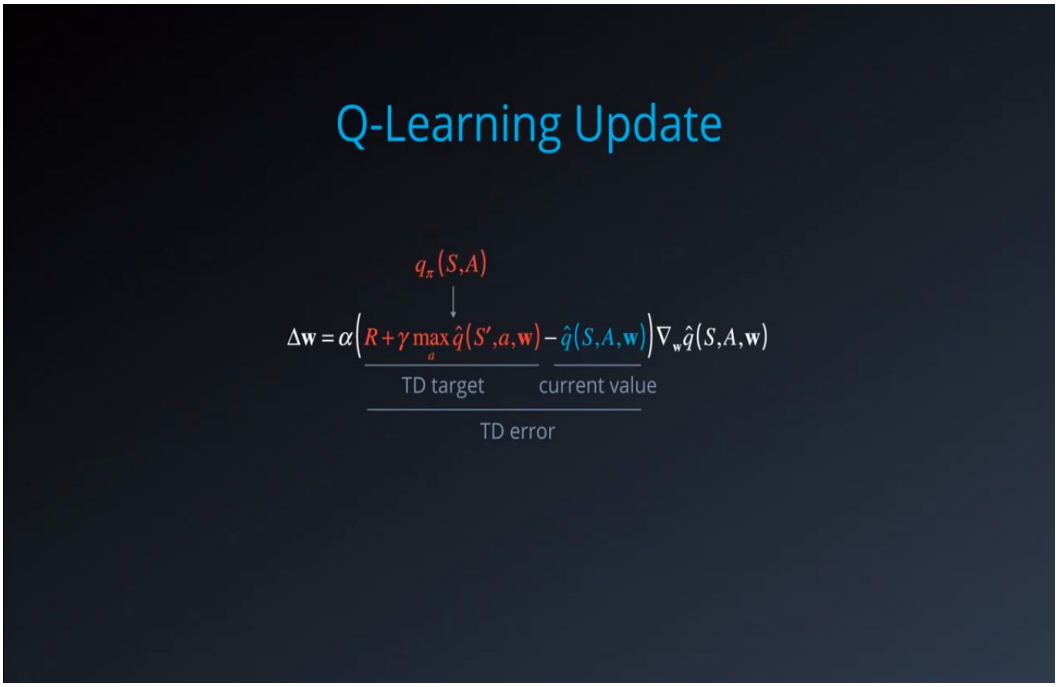

Figure 5: Update rule of Q-Learning

But we do not have any idea of the real TD target and this is the reason why we estimate it.However, the problem is that we are using the same parameters (weights) for estimating the target and the Q-value. As a consequence, there is a **big correlation between the TD target and the parameters we are changing.**

Therefore, it means that at every step of training, **our Q-values shift but also the target value shifts**. So we are getting closer to our target but the target is also moving. It is like chasing a moving target and thus results in divergence or oscillations in the training phase.

To remove these correlations with the **target**, we use a separate target network that has the same architecture as the Deep Q-Network. The change lies in the fact that we are **updating the target Q-Network's weights less often than the primary Q-Network.**

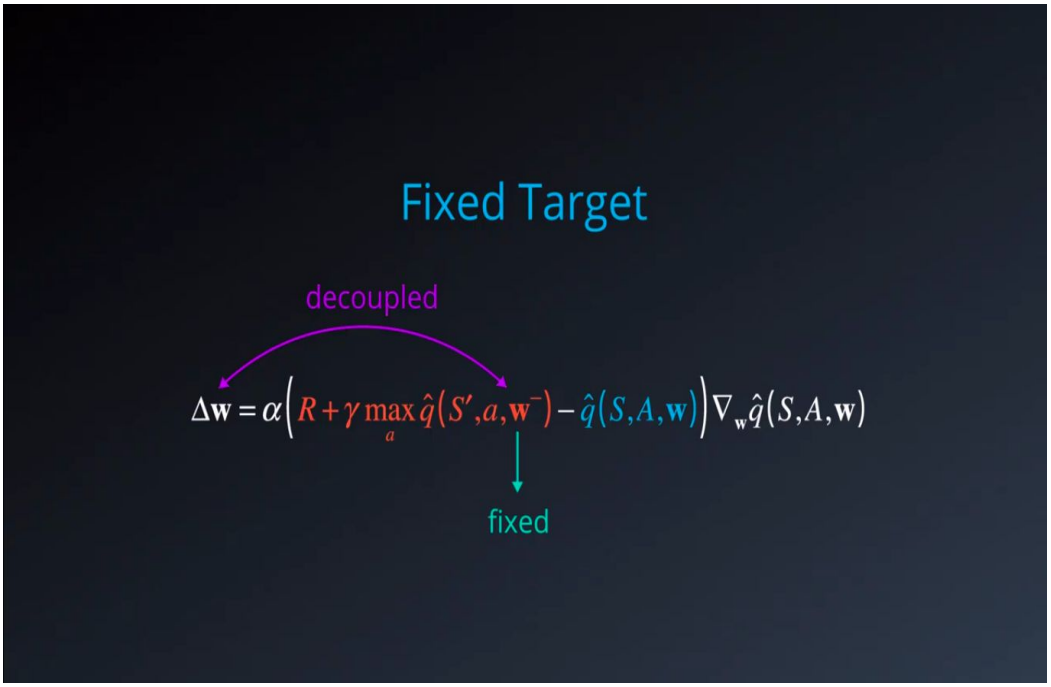

Figure 6: Update rule of Deep Q-Learning

Here $w^- \leftarrow w$ at every C steps. In the implementation, we update the target Q-Network's weights every **4 time steps**.As for the other parameters involved in the update, α or the learning rate is initialized at 5e − 4 and is learned thanks to the usage of **Adam** optimizer. The discount rate γ is set to 0.99 which is very close to 1, meaning that the return objective takes future rewards into account more strongly and the agent becomes more farsighted.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function20. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values and the target values . We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Second, we used an iterative update that adjusts the action-values towards target values that are only periodically updated, thereby reducing correlations with the target.

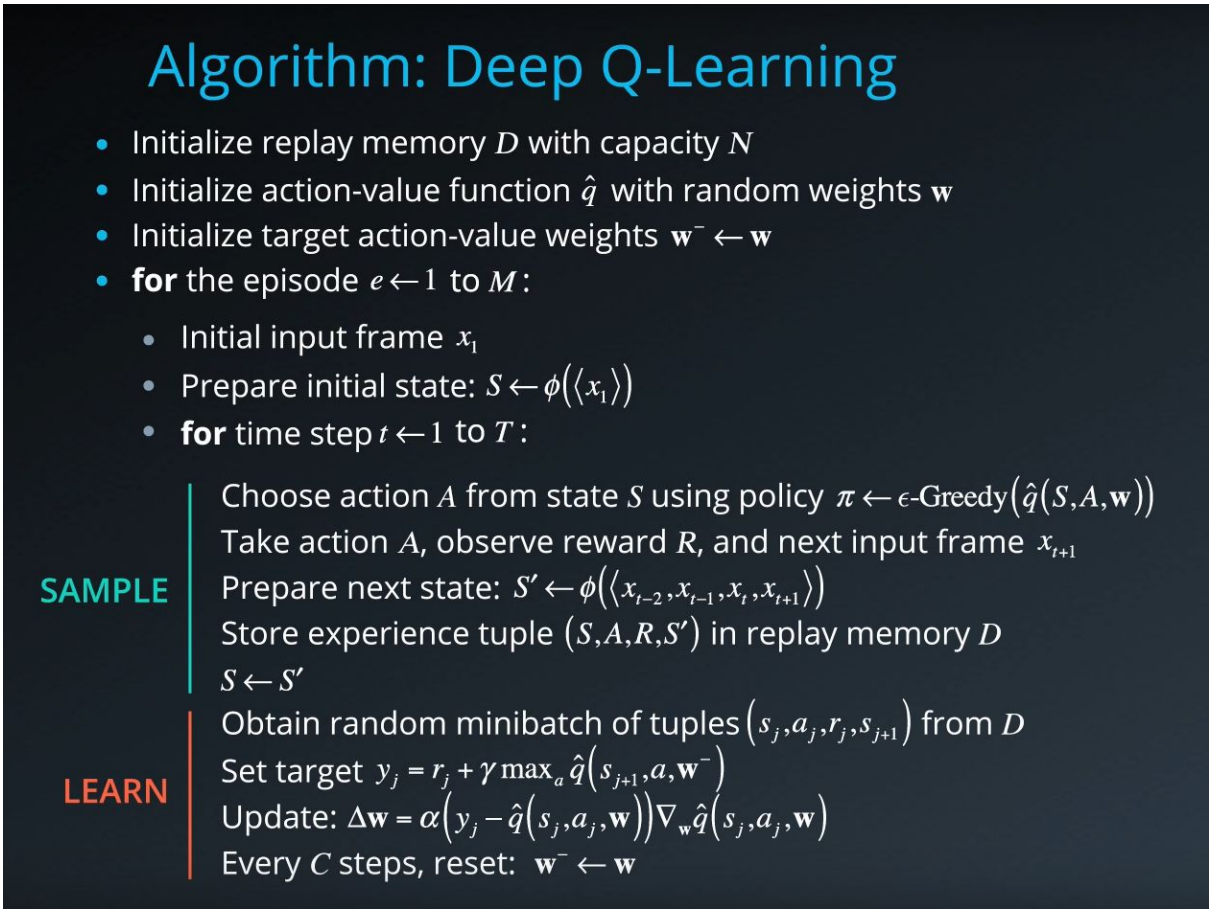

Figure 7: Deep Q-Learning Algorithm

5. Agents Implementation(Code)

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 there 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 a 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
- **Navigation.ipynb** : This Jupyter notebooks allows the agent to train. 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

6. DQN Parameters and Results

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) -> Output nodes (4)
```

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

Given the chosen architecture and parameters, our **results** are :


```
Episode 100     Average Score: 0.25
Episode 200     Average Score: 2.64
Episode 300     Average Score: 5.89
Episode 400     Average Score: 7.67
Episode 500     Average Score: 11.79
Episode 600     Average Score: 12.04
Episode 700     Average Score: 10.95
Episode 800     Average Score: 12.11
Episode 900     Average Score: 12.55
Episode 1000    Average Score: 12.76
Episode 1015    Average Score: 13.08
Environment solved in 915 episodes!     Average Score: 13.08

Total Training time = 23.3 min
```

Figure 8: Training-Logs

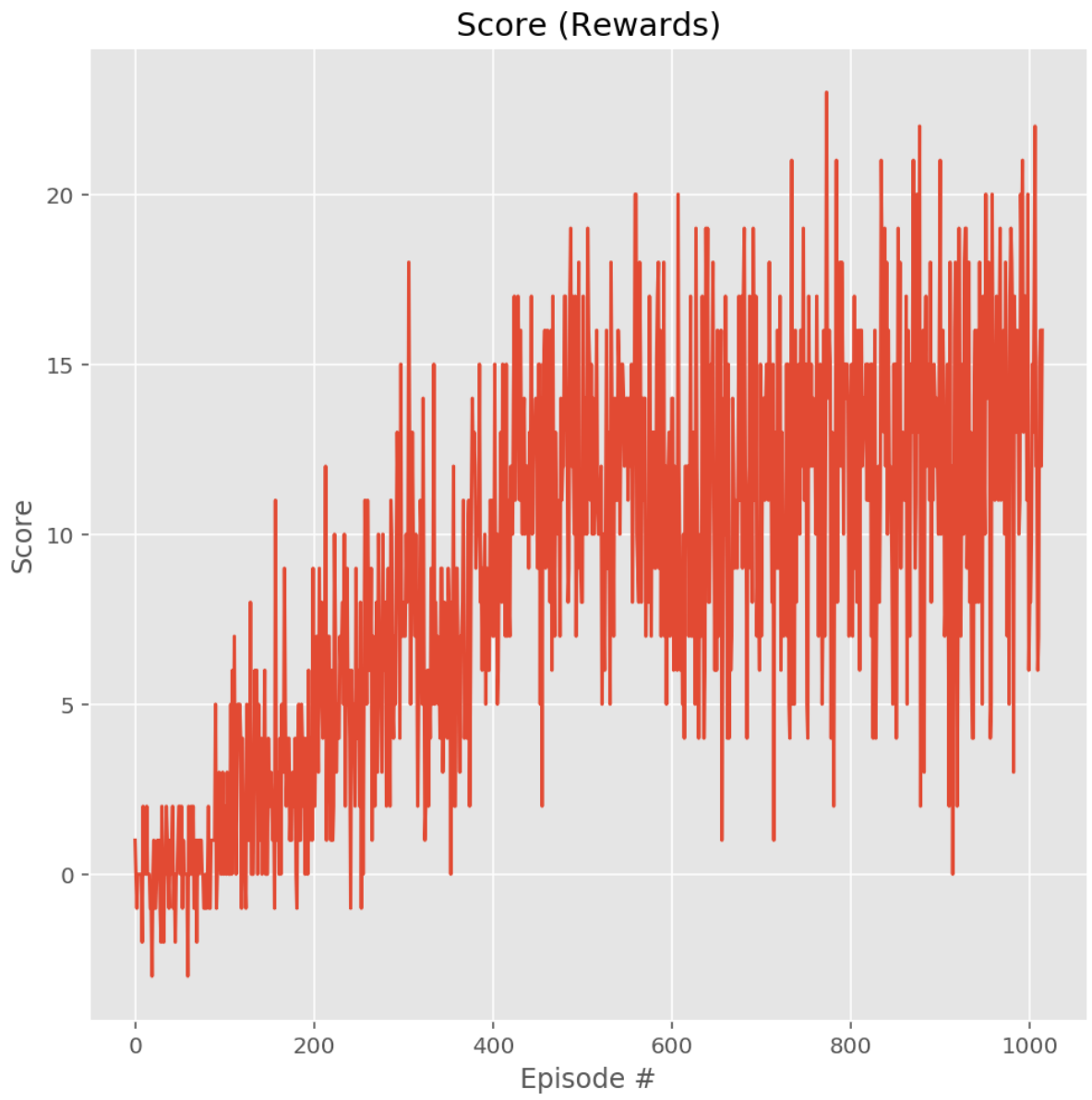

Figure 9: Score-Plot

These results meets the project's expectation as the agent is able to receive an average reward, over 100 episodes of at least +13(+13.08 to be precise), and in 915 episodes only (In comparison, according to Udacity's solution code for the project, their agent was benchmarked to be able to solve the project in fewer than 1800 episodes)

7. Ideas for future work

As discussed in the Udacity Course, a further evolution to this project would be to train the agent directly from the environment's observed raw pixels instead of using the environment's internal states (37 dimensions)

To do so a Convolutional Neural Network would be added at the input of the network in order to process the raw pixels values (after some little preprocessing like rescaling the image size, converting RGB to grayscale, ...). A very good resource to explore this idea is in one of Andrej Kaparthy's blogs- **Deep Reinforcement Learning: Pong from Pixels**

Other enhancements might also be implemented to increase the performance of 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 overestimations are common, whether they harm performance, and whether they can generally be prevented. In this paper, we answer all these questions affirmatively. In particular, we first show that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. We then show that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. We propose a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized, but that this also leads to much better performance on several games.

- **Dueling DQN**

In recent years there have been many successes of using deep representations in reinforcement learning. Still, many of these applications use conventional architectures, such as convolutional networks, LSTMs, or auto-encoders. In this paper, we present a new neural network architecture for model-free reinforcement learning. Our 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. Our results show that this architecture leads to better policy evaluation in the presence of many similar-valued actions. Moreover, the dueling architecture enables our RL agent to outperform the state-of-the-art on the Atari 2600 domain.

- **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. In this paper we develop a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently. We use prioritized experience replay in Deep Q-Networks (DQN), a reinforcement learning algorithm that achieved human-level performance across many Atari games. DQN with prioritized experience replay achieves a new state-of-the-art, outperforming DQN with uniform replay on 41 out of 49 games.