# Project 1: Navigation

Author: Ashraf Ibrahim

Date: 12.10.2020

## Introduction

For the first Submission of Udacity's Nanodegree in *Deep Reinforcement Learning*, the students have to train an agent, which is able to manoeuvre through a virtual environment (specifically a *Unity Environment* ), to collect yellow bananas (which gives a reward of +1) and avoiding blue bananas (which gives a reward of -1). The Environment is defined as space state, consisting of 37 dimension and allowing 4 actions: *forward movement*, *backward movement*,*left turn* and *right turn*.

Using a DQN and a Double-DQN, i am going to train an intelligent agent, which is able to navigate through the environment, while fulfilling the criteria of at least 13 score-points in 100 consecutive episodes.

## Theoretical Explanations

In the following Chapter, i am going to briefly introduce, some theoretical explanations behind *Deep Q-Networks* and *Double Deep Q-Networks*.

### Basics of Deep Q-Learning

One basic idea behind Q-Learning, is the concept of the *action-value-function* $Q(s, a)$, which describes the Quality $Q$ (measured by its score) of an action $a$ in state $s$. In that manner, *Q-Learning* is a function, which calculates the future return of an action *a* in state *s*:

$$Q(S_0, A_0) = Q(S_0, A_0) + \alpha(r + \gamma * \underset{a \in A}{m} axQ(S_1, a) - Q(S_0, A_0))$$

whereas $\alpha$ stands for the learning rate, $r$ for the reward, $\gamma$ for the discount rate and $maxQ(S_1, a)$ for the future value estimate. Under this assumption, the Q-Value is calculated based on greedy premise, even if it is an off-policy Method! In a non-finite action-space, the future value has to be approximated. This is where Deep Neural Networks come into play!

The basic idea behind the usage of *DNN* is to estimate the Q-Values and compare the Q-Value with its estimate $Q(s, a|\theta)$ and thus redefining the situation to a supervised Learning Problem, where the aim is to minimize the difference between the approximation and the following Q-Value (Buduma 2017: 263).

### Double DQN

Double DQN are a slightly modified version of a DQN. As DQN's are prone to overestimating Q-Values, which is based in their greedy behaviour. Especially in the begining, the estimation is based on more random occasions. To assess this problem, the procedures uses two networks. One to estimate the best actions for the next State and another one to estimate the Q-Values for the selected action. The first one is the online - the learning- network and the second one is the target - the evaluating - network (Geron 2019: 640).

Comparing both approaches, the difference is minimal but has a meaningful impact. Let us look at Calculation of the weights, first for a vanilla DQN-Approach:

$$\Delta w = \alpha(r + \gamma\hat{q}(S', argmax\hat{q}(S', a, w), \boldsymbol{w})\nabla w\hat{q}(S, A, w)$$

Here it is clear,that the weight $w$ is simultaneously a result, as well as part of the calculation. To decouple this, the evaluation is done by a separate model, which leads to the following equation:

$$\Delta w = \alpha(r + \gamma\hat{q}(S', argmax\hat{q}(S', a, w), \boldsymbol{w'})\nabla w\hat{q}(S, A, w)$$

Thus $w'$ resembles the weights of the online model.

## Implementations

In this section i am going to introduce the used hyperparameters at first. Then i am going to present the results.

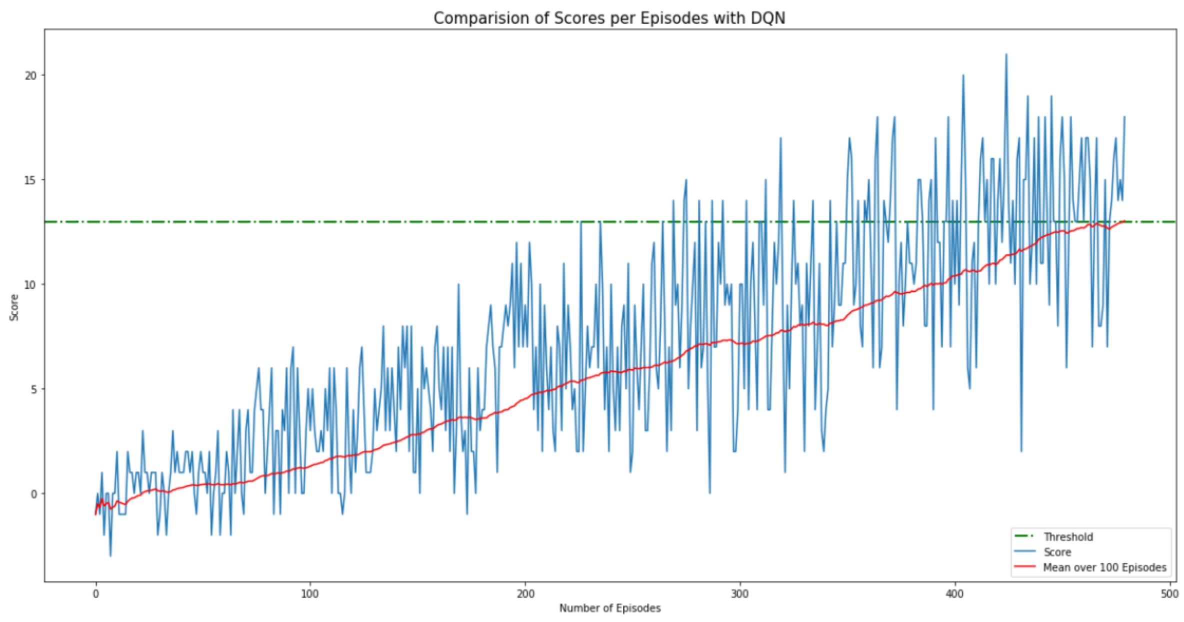
### Hyperparameters

| **Hyperparameter** | **Value** |
|---|---|
| Buffer size | $1e^5$ |
| Batch Size | 64 |
| $\gamma$ | 0.99 |
| $\tau$ | $1e^{-3}$ |
| $\alpha$ | $5e^{-4}$ |
| Update after _ Episodes | 4 |

With a Buffer size of $1e^5$, i used a large replay buffer to gather a wide range of experiences. The Batch size is configured to take 64 tuples of {state, action,reward,state + done}, the discounted rate $\gamma$ tends almost completely for a long-term benefit, the soft update for target parameters ($\tau$) is small, which results in a soft update of $1 - 0.001 = 0.999$. The learning rate $\alpha$ is about 0.0005. The model is updated every 4 episodes.

### DQN Results

Using the vanilla DQN-approach the model achieved the goal of a minimum of at least 13 score-points in 100 consecutive episodes, after 480 episodes. The following Plot shows, that the scores fluctuates heavily but is steady increasing towards the threshold of 13 points (the green line).

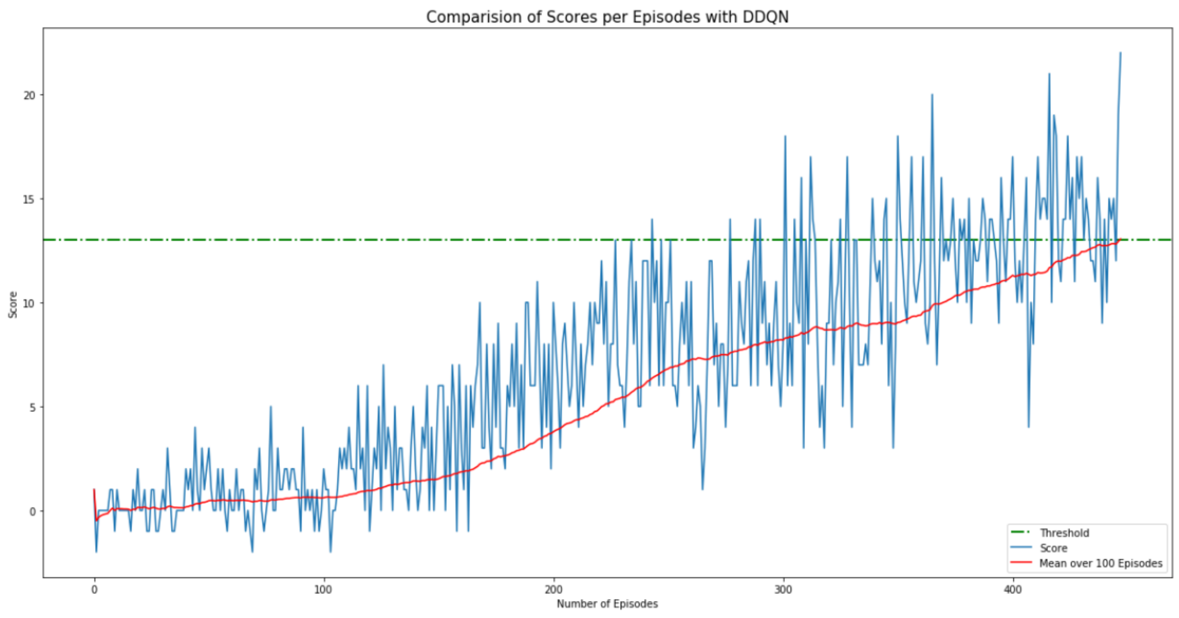


## DDQN Results

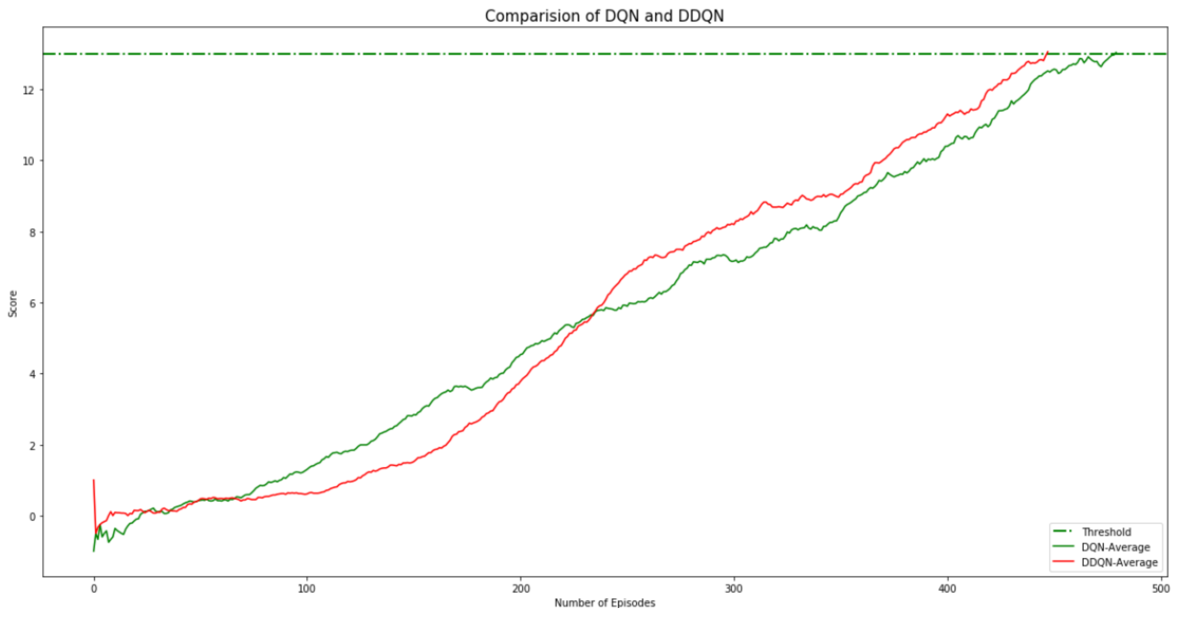
To create a DDQN Agent, i copied the Agent-class and modified it slightly. This was done by adding a few lines of code to the *learn function* of the agent class:

```
next_actions = self.qnetwork_local(next_states).detach().argmax(1).unsqueeze(1)
Q_targets_next = self.qnetwork_target(next_states).gather(1, next_actions)
```

The double DQN did perform a bit better, it didn't fluctuated as much in the beginning and seems to have fewer negative outliers concerning the scores. It took fewer episodes to achieve the minimum amount of score points (448 episodes). Especially in the begining the mean-rate resembles more a learning curve, which makes it a more steady start.



Comparing the evolution of the mean over 100 episodes, it is obvious that both have are similar in their learning behaviour. The only memorable difference is the more steady learning curve from the DDQN approach.



## Loading and Playing

Loading the agent and playing the environment, shows that both in fact react very similar. Over 5 Episodes, the mean score for the dqn agent was 15 and the mean score for the ddqn agent was 16.

## Further Improvements

It would be interesting to do some Hyperparameter tuning, especially to define a grid over certain values as $\gamma$, $\alpha$ and $\tau$ as well as calibrating the replay buffer and thus comparing both approaches. Other ideas like implementing a duel-DQN or a noisy DQN could be as well interesting. Looking at the heavy fluctuation of scores, a prioritized experience replay could be very usefull, as it would make the positive outlier more important in the learning procedures.