UDACITY

# (DEEP REINFORCEMENT LEARNING)

PROJECT 01: NAVIGATION

By
Imran Yasin

**“This report presents a specialized portrayal of the Navigation venture with regards to the Deep Reinforcement Learning Nanodegree from Udacity”**

## 1. SUMMARY

The project contains training an agent that learns by himself; i: e. by **trials and errors**, to collect special type of bananas (yellow ones) while avoiding blue ones in a restricted environment. The goal is to make the most of the overall poised yellow bananas in a given episodes so there are elements of **delayed reward**.

These two features – trials-and-error search and delayed reward – suggests that our problem is well framed to be considered as a **Reinforcement Learning** (RL) problem.

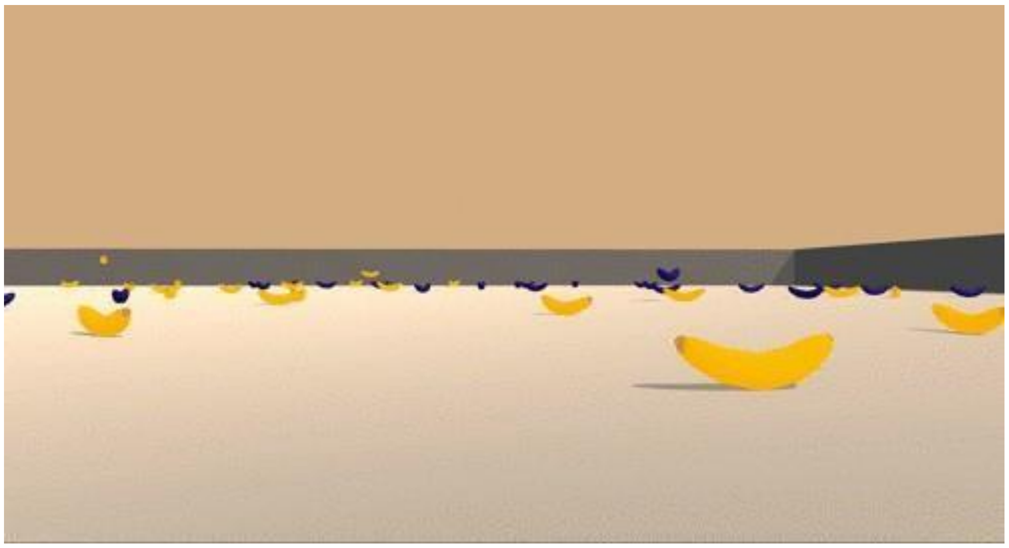


Figure 1: Overview of the environment

# 2. FROM ‘RL’ TO DEEP ‘RL’

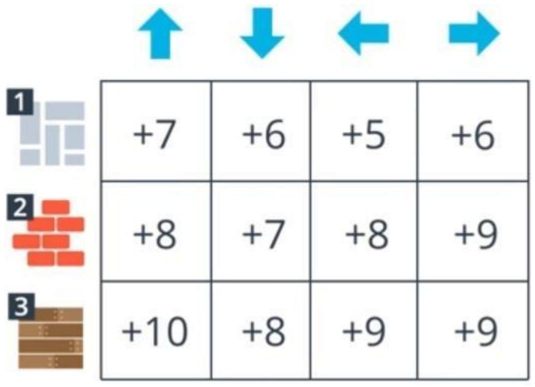
## 2.1 Reinforcement Learning

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 step 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 steps.

The Q-table is used to approximate the **action-value function** $q_\pi$ for the policy $\pi$. You can find an example below of a Q-table that considers which action to take depending on the environment:



|   | ↑ | ↓ | ← | → |
|---|---|---|---|---|
| 1 | +7 | +6 | +5 | +6 |
| 2 | +8 | +7 | +8 | +9 |
| 3 | +10 | +8 | +9 | +9 |

Figure 2: Q-table example

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

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

## 3. STABILIZING DEEP REINFORCEMENT LEARNING

Deep Q-Learning alone has been found to be notoriously **unstable** when neural networks are used to represent the action values.

In particular, two key features address these instabilities:

- **Experience Replay**
- **Fixed Q-Targets**

### 3.1 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*$^{\prime}$). 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 reaching 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.

I used a **buffer size of 50000**, meaning that we store up to 50000 experience tuples at a time. This value should be high enough to sample from a variety of samples and not too high at the same time due to memory limitations of the workstation.

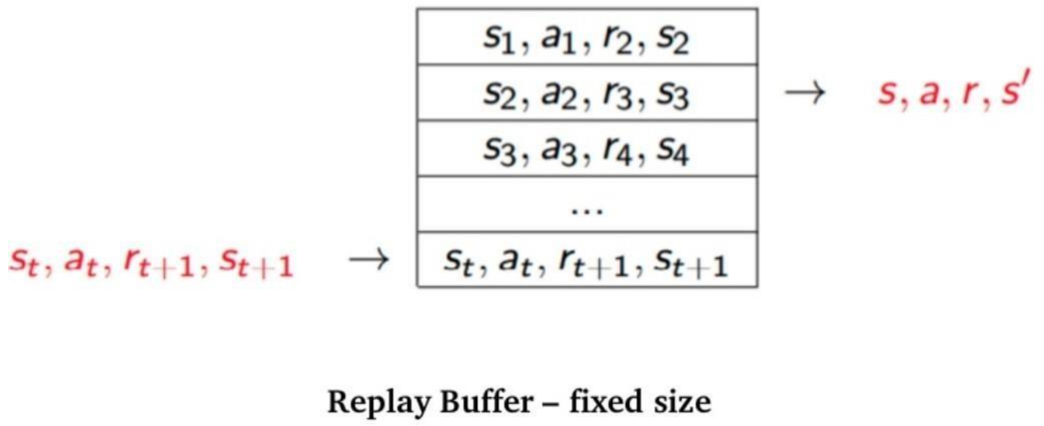

$s_1, a_1, r_2, s_2$
$s_2, a_2, r_3, s_3$
$s_3, a_3, r_4, s_4$
…
$s_t, a_t, r_{t+1}, s_{t+1}$
$s, a, r, s'$
$s_t, a_t, r_{t+1}, s_{t+1}$
Replay Buffer – fixed size

Figure 4: Replay Buffer overview

## 3.2 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) .

$$\Delta \mathbf{w} = \alpha\Big(\underbrace{\underbrace{R+\gamma \max_{a} \hat{q}(S',a,\mathbf{w})}_{\text{TD target}} - \underbrace{\hat{q}(S,A,\mathbf{w})}_{\text{current value}}}_{\text{TD error}}\Big)\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

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 to 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**.

$$\Delta \mathbf{w} = \alpha\Big(R+\gamma \max_{a} \hat{q}(S',a,\underset{\substack{\downarrow \\ \text{fixed}}}{\mathbf{w}^-}) - \hat{q}(S,A,\mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

Figure 6: Update rule of Deep Q-Learning

with :

$$w^- \leftarrow w \text{ at every C steps}$$

In the implementation, we update the target Q-Network's weights every **4** time steps.

As for the other parameters involved in the update, $\boldsymbol{\alpha}$ or the learning rate is initialized at $5e-4$ and is learned thanks to the usage of **Adam** optimizer.

The discount rate $\boldsymbol{\gamma}$ 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.

## 4. Policy

The learning policy is **Greedy in the Limit with Infinite Exploration** (GLIE). This strategy initially favors exploration versus exploitation, and then gradually prefers exploitation over ex-proration.

As such, we train using an $s$-greedy policy with $s$ annealed exponentially from 1.0 to 0.005 with a decay of 0.999 every episode. After 5296 episodes, the parameter $s$ is fixed at 0.005, but in this project, we do not train the agent this long.

## 5. Result

The agent is trained during 1000 episodes, and we compute the cumulative reward per episode. We consider the environment to be solved when the average reward over the last 100 episodes is at least +13, for that we use **queue** (FIFO : First In, First Out) with length 100 using a **deque** in the collections library of Python to progressively compute the average over the last 100 episodes. When the training is done, we can inspect plot of rewards per episode:

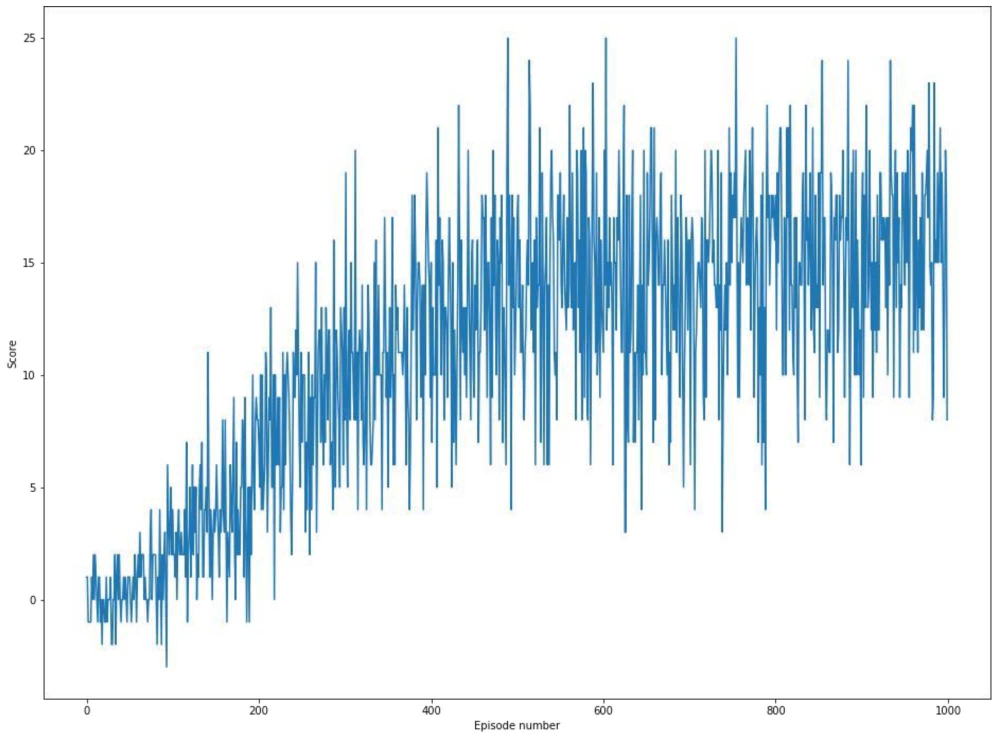


Figure 7: Plot of scores per episode

**The environment is solved in 392 episodes**!

## 6. What’s next

**Prioritized experience replay** is an improvement to the experience replay procedure, and the main idea is to accord more interest on experiences that seems to be more important for the learning procedure.

There is a **visual version of this environment** where the states are only the visual information from the agent. Based on Andrej Karpathy’s famous blogpost[1] ”Deep Reinforcement Learning: Pong from Pixels ”, this challenging task appears to be interesting