Udacity

Project Report

P1 - Navigation

Author:
Thomas Di Martino

Supervisor:
Udacity

A report submitted in fulfilment of the requirements
for the Student in the Deep Reinforcment Learning Nanodegree Program

of the

Artificial Intelligence department

May 2020

Declaration of Authorship

I, Thomas Di Martino, confirm that this work, 'P1 - Navigation', submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: _____

Date: _____

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem diagnosis

In the context of a *Banana Collection* task, our agent finds himself in a 3 dimensional world (although it can only move along x and y axis, no z-axis, meaning no jumps). In this environment, he encounters two types of bananas:

- Yellow Bananas, that are meant to be collected by the agent.
- Purple Bananas, that are meant to be avoided by the agent.

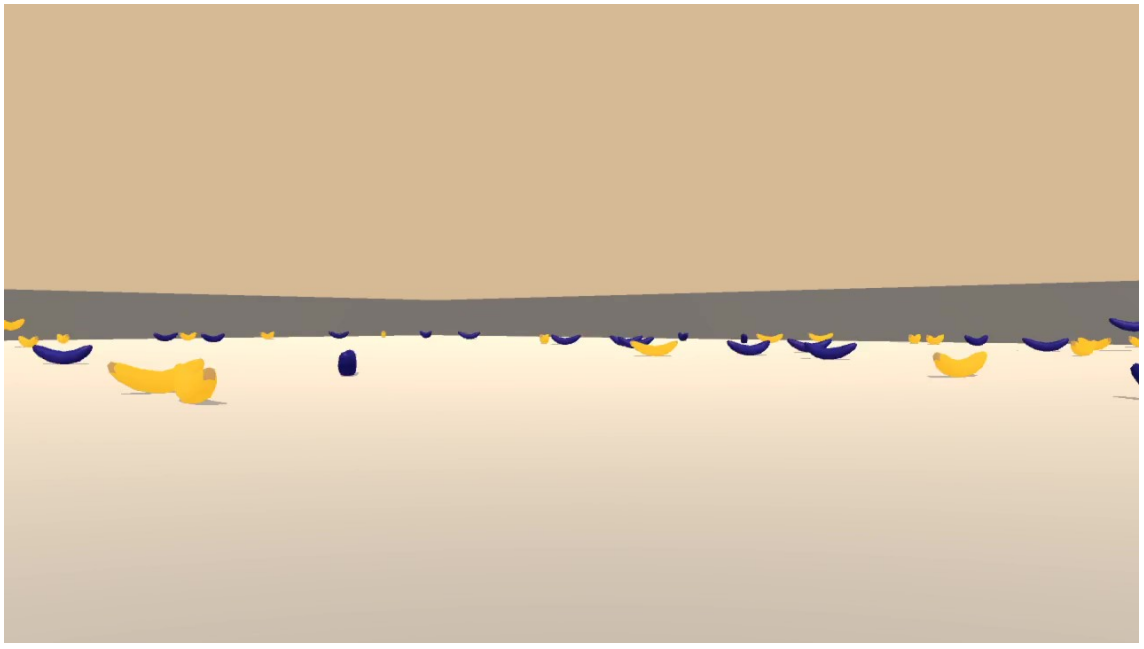

FIGURE 1.1: Screenshot of the agent's environment

1.2 Environment Introduction, a RL perspective

1.2.1 States

The state space of an agent is a 37 dimensions vector containing the agent's velocity as well as perception data of surrounding object: no visual feature is used by the agent to explore its environment.

```
[1.          0.          0.          0.          0.84408134 0.
 0.          1.          0.          0.0748472   0.         1.
 0.          0.          0.25755     1.          0.         0.
 0.          0.74177343  0.          1.          0.         0.
 0.25854847  0.          0.          1.          0.         0.09355672
 0.          1.          0.          0.          0.31969345 0.
 0.        ]
```

FIGURE 1.2: Agent state example

1.2.2 Rewards

Three types of reward exist for an banana collecting agent:

- When the agent does not encounter any banana, it receives a reward of **+0**;
- When the agent walks over yellow bananas, it receives a reward of **+1**;
- When the agent walks over purple bananas, it receives a reward of **-1**.

As no negative reward is given to an agent, it will not necessarily be motivated by taking the shortest path possible to collect yellow bananas and will (or at least should), thus, **always** avoid purple bananas **at all cost**. In other words, the agent should not develop a strategy under which sacrifices of privileging a shortest path, containing a purple banana, would be retained.

1.2.3 Actions

To collect these rewards, our agent is provided with 4 distinct actions:

- *0*: move forward
- *1*: move backward
- *2*: turn left
- *3*: turn right

1.2.4 Success Criteria

When training the agent, it is considered successful to the eyes of the assignment when it gathers a reward of **+13** over 100 consecutive episodes.

1.3 Report Structure

In this report, we will present our implementation (two slightly different ones) as well as expose their resulting learning performance. We will then explore eventual amelioration of the current work, exposing the weaknesses of both the retained model as well as for the environment (a parallel may even be drawn with real life situations).

1.4 Code location

Located in a Github repository at the following address https://github.com/dimartinot/P1-Navigation

Chapter 2

Models & Results

2.1 Retained models

Two agents were trained during this work:

- A Deep Q Learning Agent using a **Replay Buffer of uniformly distributed samples** as well as fixed Q Targets;
- A Double Q Learning Agent that implements **prioritised experience replay** using the following formula $\frac{p^\alpha}{\sum_i p_i^\alpha}$ where p is the lastly measured td error of the experience tuple, before its insertion in the buffer. Alpha is an hyper-parameter: the closer alpha is to zero, the more uniform the sampling distribution will be. The closer to one, the less uniform it will be.

2.1.1 Deep Q Learning with uniform Replay Buffer & Fixed Q-Targets

2.1.1.1 Hyper-Parameters and formula presentation

Every time we calculate the TD Target value ($td\_target = R_{t+1} + \gamma * \max_a(\hat{q}(S_{t+1}, a, W_t))$), we will use both a "*target*" model with weights different than the "*local*" model use to estimate the value of the current state and action. This helps to decouple target parameters from the agent actions.
The Deep Q Learning agent used for this task implements the update from the local to the target network using soft update:
at every learning time-step, the weight of the target network will be updated with the following rule.

$$W_t = \tau * W_l + (1 - \tau * W_t)$$

where:

- W_t are the target weights;
- W_l are the local weights;
- τ is an hyper-parameter, with value set at $1e-3$ for this project, that specifies the "*magnitude*" of the update.

To train the local network, we use a *Mean Squared Error* loss between the TD Target and the local weights evaluation of the (state, action) tuple. Formally, this is the loss trying to be minimised:

$$L = \frac{1}{N}\sum_{n=1}^{N}(R_{t+1} + \gamma * max_a(\hat{q}(S_{n+1}, a, W_t)) - \hat{q}(S_n, A, W_l))^2$$

where:

- $R_{t+1} + \gamma * max_a(\hat{q}(S_{n+1}, a, W_t))$ is the target value (which is evolving through computation), that we would write as $\hat{y}$ in a common Machine Learning problem;
- $\hat{q}(S_n, A, W_l)$ is the predicted value, that we would write as y;
- N is the batch size (in our implementation, we selected 64 as a batch size).

The learning rate used to update model weights is set as 5e-4 and used an Adam optimiser.

2.1.1.2 Deep Learning Model Architecture

The following figure (cf. 2.1) presents the retained Deep Learning architecture. We took massive inspiration from Google's DQN model, removing the feature extraction layer using convolutional layers, having our features "*already processed*" in the input.

2.1.1.3 File architecture

Relevant files for this part of the implementation are:

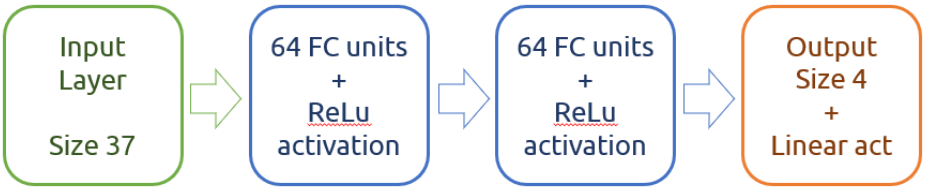

FIGURE 2.1: Model Architecture

- *agent.py*: Contains the class definition of the basic double q learning algorithm that uses soft update (for weight transfer between the local and target networks) as well as a uniformly distributed replay buffer;
- *model.py*: Contains the PyTorch class definition of a neural network, used by the target and local networks.

2.1.2 Double Q Learning with prioritised Replay Buffer & Fixed Q-Targets

To test the limitation of the Deep Q Learning model, a Double Q Learning network has been implemented, as well as a prioritised replay buffer.

2.1.2.1 Formula update

In this double Q Learning setup, we changed the td target formula to be the following:

$td\_target = R_{t+1} + \gamma * \hat{q}(S_{t+1}, \max_a(\hat{q}(S_{t+1}, a, W_t), W_l)$

Furthermore, another modification was made to the Replay Buffer that now uses the td error of an experience tuple as an importance measure in order to sample the most important tuples at all time.

2.1.2.2 File architecture

Relevant files for this part of the implementation are:

- *prioritizedAgent.py*: Contains a class (PrioritizedAgent) inheriting from the basic Agent class, adding a priority measure for sampling experience tuples from the replay buffer;

2.2 Results

We now present the results of training these two architecture with the curve of the *reward per episode* function.

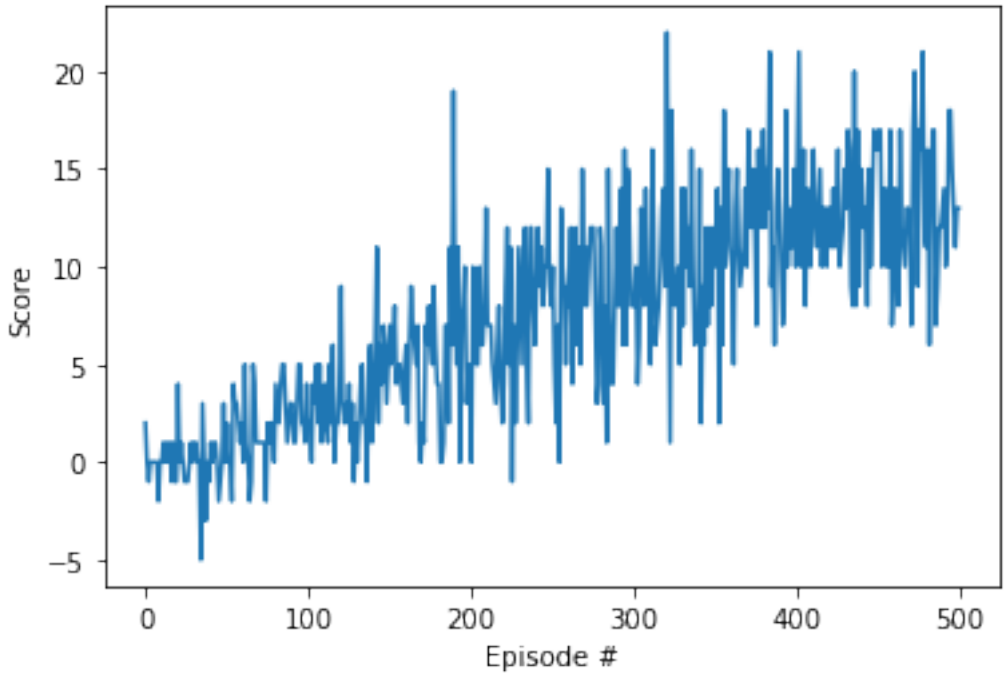

FIGURE 2.2: Deep Q Learning architecture training performance

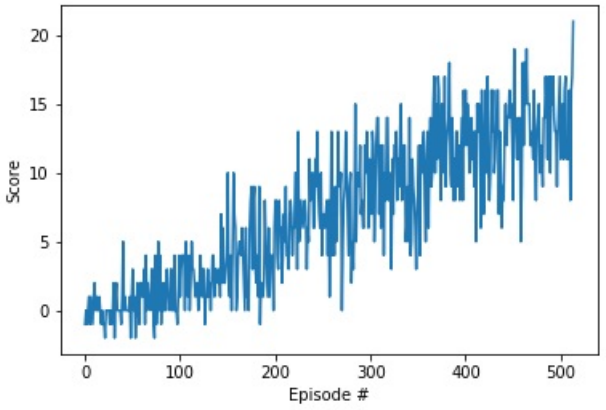

FIGURE 2.3: Prioritised Double Q Learning architecture training performance

Chapter 3

Conclusion

3.1 Summary

As a summary, two methods were studied and used in this project: while sensibly similar performances were obtained, no thorough conclusion can be drawn with regards to their relative performance. Indeed, as not a lot of hyper parameter exploration has been done, we can only base ourselves on these limited results. Furthermore, the threshold of *+13* score to be obtained limited our analysis of how far can both model go. However, this was **not** the goal of this project and every objectives were met.

3.2 Overture

An interesting concept to be explored is the feature extraction phase with the use of a state represented as a picture of the environment. Using convolutional layers, we would see an harder task unfolds before us, which would then be very interesting to compare to the other format of this project. The main guesses in the comparison are:

- The use of the 37-sized state vector would lead to faster and better converging algorithm (as well as easier to train);
- The use of the image input would lead to better overall performance, to the cost of an harder training pipeline as well as more computation involved.

However, both of these ideas are assumptions and are yet to have been studied by myself.

Bibliography