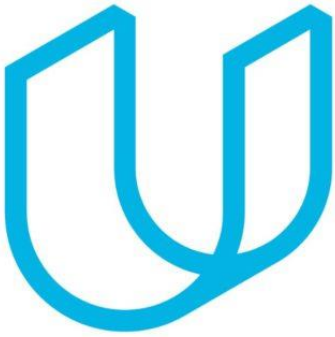

UDACITY

DEEP REINFORCEMENT LEARNING

VALUE BASED METHODS

PROJECT: *NAVIGATION*

AUTHOR: ALBERTO GARCÍA GARCÍA

Introduction

The problem to be solved in this project consists in training an agent through reinforcement learning algorithms. Such agent must pick yellow bananas (positive reward of +1) and avoid blue ones (negative reward of -1) in a square world. An image of such environment is presented in Fig. 1. The problem is considered to be solved when the average score of the last 100 episodes is equal or greater than 13 points.

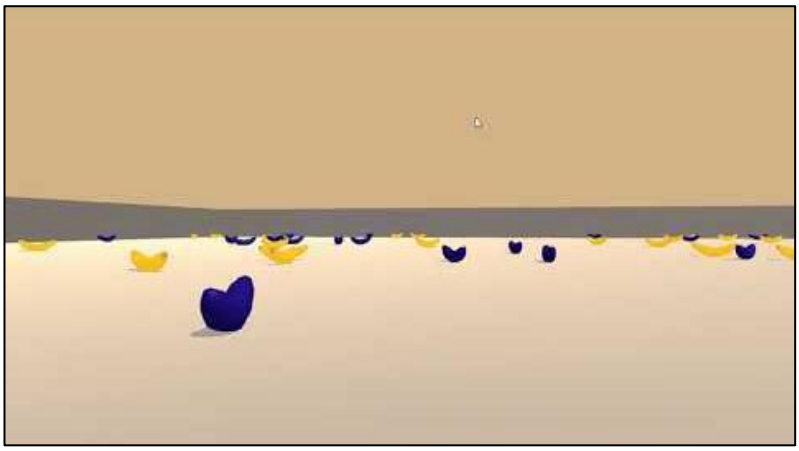

Fig. 1. Environment

This document includes the details of the algorithm developed to train the agent, the conclusions obtained after the completion of the project and some future ideas to improve the current implementation.

Implementation

In order to train the agent, the Deep Q-Learning algorithm has been chosen. This algorithm uses a deep Q-network to learn a successful policy, applying the advantages of deep neural networks to the classic reinforcement learning.

The deep Q-network has been implemented using the PyTorch framework and presents the following architecture:

- An input layer with 37 neurons (37 is the number of observations in any given state of the problem).
- Two hidden layers of 64 neurons each.
- An output layer with 4 neurons (4 is the number of possible actions that the agent can choose).

Regarding the network hyperparameters, Adam has been chosen as the optimizer, ReLU as the activation function in the hidden layers, 0.0001 as the learning rate and L2 as the loss function. It must be said that other configurations were tested, like a more complex architecture

with more layers and neurons, different learning rate values, other optimizers (i.e.: SGD) or even weight initialization techniques (i.e.: He initialization). Nevertheless, the simple network described above has shown the best results.

Along with the deep Q-network, other key features that have improved the agent's performance are the following:

- An epsilon-greedy policy to choose between random actions and the actions selected by the neural network. The epsilon value starts in 1.0 and has a 0.99 decay per episode ($epsilon = epsilon * 0.99$), being 0.001 the minimum value this parameter can take. This way the agent tends to explore in the early episodes and to exploit the best actions in the latest ones.
- A replay memory from which random batches of past experiences can be sampled to train the deep Q-network. The batch size chosen for this problem was 128 experiences, and the training of the network is performed in every step of every episode (once the replay memory contains at least 128 experiences).
- A second neural network identical to the main deep Q-network (sharing the same architecture), which is used to evaluate the actions chosen by the main deep Q-network. This second network is not trained periodically as the main one. Instead, every certain number of steps (in this case, 150 steps), the weights of the main deep Q-network are copied into the weights of this second network. This technique, called double deep Q-network, improves the learning process avoiding the overestimation of the rewards that a single neural network would do otherwise.

Conclusion

The implementation described in the previous section is able to solve the problem in just 395 episodes, showing a reasonably high performance (the execution took around 9 minutes in a non-GPU environment). Fig. 2 shows the evolution of the agent's performance.

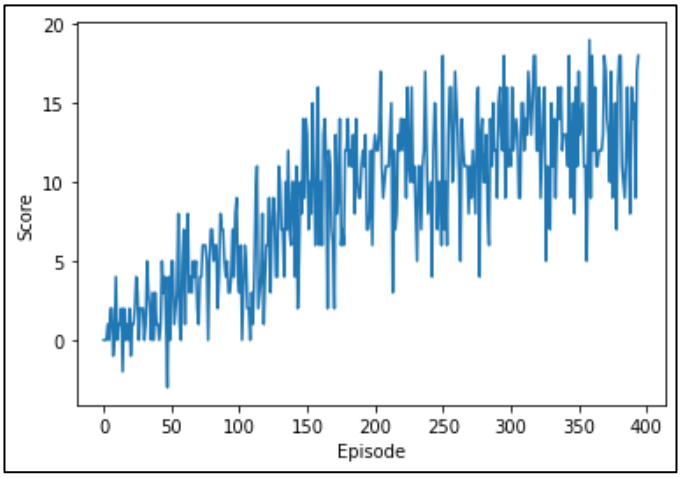

Fig. 2. Agent's score history

This project has been a great opportunity to explore the deep Q-learning algorithm, some of its extensions and its outstanding capabilities.

Future Work

Even though the agent has successfully learnt to solve the problem, there are some ideas that could have been explored to improve even more its current performance. The most well-known extensions are dueling deep Q-networks and prioritized experienced replay.