Navigation With Deep Reinforcement Learning

Romain Fournier

August 23, 2018

1 Introduction

In this brief report, we present our implementation of a deep reinforcement learning algorithm that solves an environment similar to the Unity's Banana Collector one. An agent moves within an environment containing blue and yellow bananas. The goal is to collect the yellow ones while avoiding the blue ones. This environment provides states to the agent in the form of a 37-dimensional continuous space. It contains the agent's velocity, along with a ray-based perception of objects around the agent's forward direction. In response to this state, the agent can take one of these four actions: move forward, move backward, turn left or turn right. A reward of +1 is obtained for each yellow bananas encountered and -1 for the blue ones. The environment is considered solved when the average score over 100 episodes overreaches 13.

2 Agent

2.1 General Reinforcement Learning Setup

We consider an agent present that can take actions $a \in A$ in a certain environment, where A represents the set of all possible actions. In response to this action, the environment provides a new state $s \in S$ to the agent, as well as a reward $r \in \mathbb{R}$. The goal of the agent is to maximize the number of rewards it gets during his lifetime.

To achieve this, we define a policy as a (stochastic) function $\pi : S \to A$ that maps a state to the next action of the agent. Finding the best policy is the goal of the reinforcement learning problem. To help us with this task, we define the Q function $Q : S \times A \to \mathbb{R}$ that gives the expected discounting reward of the agent for leaving the state s with action a, and then follow the best policy. The discounting reward is defined as :

$$R = \sum_{i=t}^{\inf} \gamma^{(i-t)} r_i, \tag{1}$$

where $\gamma \in (0, 1)$, t is the current state and the next rewards are the ones obtained following the procedure described above. With this definition and knowledge

about the state s_{t+1}, we can define Q by the self-consistent equation that follows:

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) \tag{2}$$

One procedure for finding this Q function is the topic of the next subsection.

2.2 Deep Reinforcement Learning

The idea is to approximate the Q function with a deep neural network F with weights w :

$$F(s, a; w) \approx Q(s, a). \tag{3}$$

For a given architecture, the task of finding the best policy reduces to finding the best parameters w. The basic algorithm can be summarized with the following steps:

0. Initialize the weights w and w^- randomly to generate to functions F.

1. For a decaying towards 0 sequence of ϵ, sample sequences of $(s_t, a_t, r_{t+1}, s_{t+1})$ by choosing a random action with probability ϵ or the one that maximizes $F(s_t, a_t; w)$ otherwise.

2. After n steps, choose some sequences randomly and update the weights with the following update rule :

$$\Delta w = \alpha(r_j + \gamma \max_a F(s_{j+1}, a; w^-) - F(s_j, a_j))\nabla_w F(s_j, a_j; w), \tag{4}$$

where α is the learning rate.

3. Each m steps, update $w^- \leftarrow w$

4. Repeat

Note: The trick of using two different networks w and w^- is used to avoid chasing a moving target.

2.3 Two Improvements

Previous works have shown that this implementation tends to overestimate the Q function. A quick fix, known as Double Deep Q Network, it to modify the previous equation with :

$$\Delta w = \alpha(r_j + \gamma F(s_{j+1}, arg\max_a(F(s_{j+1}, a; w)); w^-) - F(s_j, a_j))\nabla_w F(s_j, a_j; w). \tag{5}$$

The second improvement is done by modifying the structure of the neural network F. Instead of simply stacking together fully connected layers, this method, known as dueling, adds a fork as the penultimate layer. This fork contains two layers, one having an output of dimension one and the other a number of dimensions corresponding to the dimension of the action space. The idea is that the first layer corresponds to the value of the state and the second one to the "advantage" of taking a particular action. The output of the network is the sum of these two layers.

3 Results

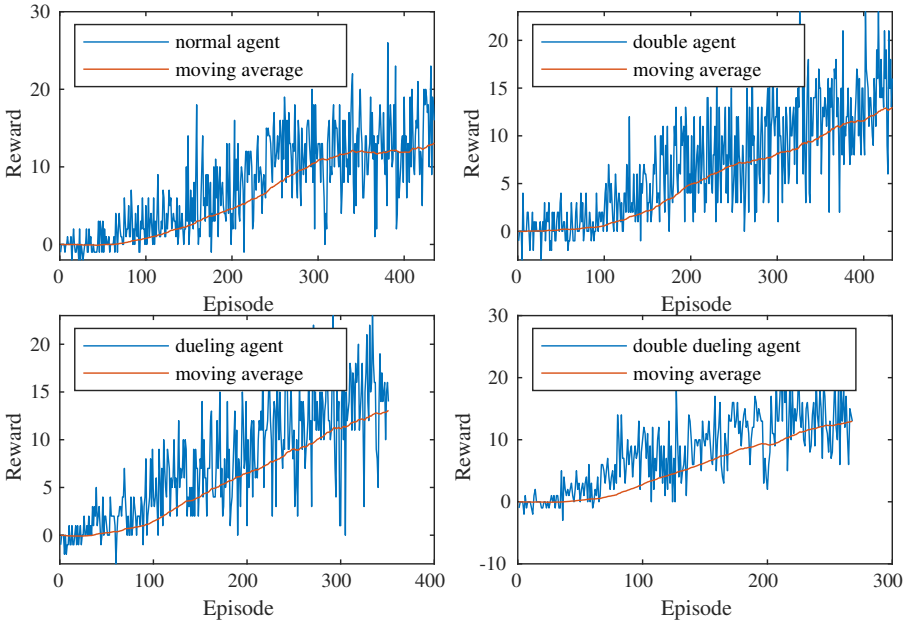
3.1 Practical details

For the following tests, we used a fully connected feed forward neural network containing a first layer outputting a 128dimensional vector, followed by a ReLU, then by a layer outputting a 64dimensional vector, followed by a ReLU. The last layer outputting a 4dimensional vector is either a standard one or a fork as described previously for dueling.

The list of parameters used for obtaining the following results are the default ones present in the repository. We noticed that ϵ_decay plays a significant role in the number of episodes needed to reach the target. The value of 0.985 chosen is the one that minimized this number for the double dueling agent.

3.2 Scores

The following figure shows the results of the training for the four combinations of our agents (implementing or not double DQN and dueling DQN). The agent implementing both reaches the target in 168 episodes.



4 Further Ideas

Following steps would consist of tuning the parameters and testing other network architectures. We could also implement a prioritized memory replay algorithm, which makes the agent learn the challenging cases more often.