UNIVERSITÀ DEGLI STUDI DI TRIESTE

A model for a self-flying drone

Reinforcement Learning and Cyber Physical Systems (CPS)

Motahhareh Nadimi

January 7, 2021

Abstract: In this work we develop a Q-learning algorithm with an epsilon-greedy method for a self-flying drone. It uses an exploration and exploitation strategy in a three-dimensional space. We analyse its results using both a deterministic and a non-deterministic implementation.

CONTENTS

1 Introduction **3**

2 The Algorithm **3**

2.0.1 The Model 3
2.0.2 Epsilon-greedy method 3
2.0.3 Non-deterministic approach 4
2.0.4 Q-Learning 4

3 Conclusions **4**

1 Introduction

A Q-learning algorithm with an epsilon-greedy method can be used to guide the flight of a drone in space. In this work we model such a system, which is a nice example of an application of Reinforcement Learning. The *agent* (a drone) moves (deterministically or non-deterministically) in space trying to reach to the *goal* state (e.g. the airport) avoiding the *loss* state (e.g. a river). It is a generalisation of a grid world game to three dimensions.

We start by defining a three dimensional array, as shown in Fig. 1.1. We assume that the drone is the only vehicle in this space. We can break up the space into a $(N_x \times N_y \times N_z)$ grid. In the following we set $N_x = 4$, $N_y = 5$, $N_z = 3$.[1]

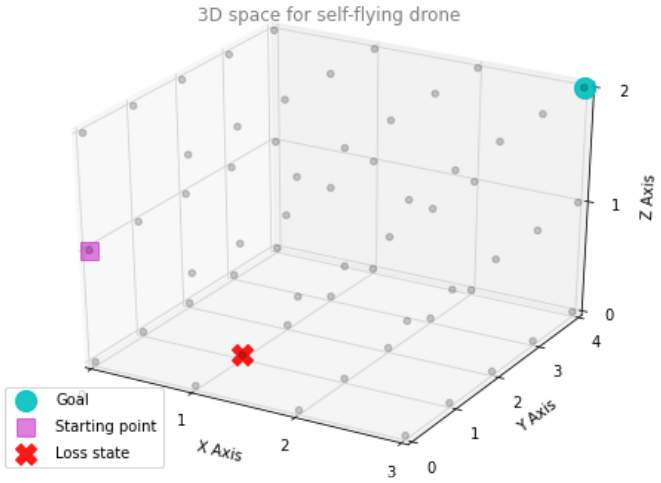

Figure 1.1: The 3 dimensional space of the problem. The *goal* is shown as green circle, the *loss state* as a cross, and the *starting point* as a purple square.

2 The Algorithm

2.0.1 The Model

The possible actions are movements in different (x, y, z) axes, as defined below:
Actions = ["left", "right", "up", "down", "top", "bottom"]

We define the limit of coordinates for the drone not to go out of the defined space.

In the following we use as a benchmark point the states, which are also illustrated in Fig. 1.1:

START = (0, 0, 1)
LOSS STATE = (1, 1, 0)
GOAL STATE= (3, 4, 2)

We give +1 points for a successful reach of the *goal* state, and substract 1 point when it goes to the *loss* state.

2.0.2 Epsilon-greedy method

The Epsilon-Greedy Algorithm makes use of the exploration-exploitation tradeoff. In the exploration case the agent with probability $\epsilon = 0.8$ choses any action at random. In the exploitation case, which occurs with probability $1 - \epsilon = 0.2$, the program iterates over all actions and picks the one which has the highest Q-value. This allows to balance exploration and exploitation, so that we avoid the agent to get stuck in some local maximum.

[1] Notice that we use a different convention for the axes with respect to the usual one.

2.0.3 *Non-deterministic approach*

In the deterministic case, the agent choses the action resulting from the exploration-exploitation probability distribution, call it a_0, with probability $p = 1$. We also implement a non-deterministic selection of action, which could be use to model errors in the execution of the action (in the case of a Drone), or say an emotional behaviour, in the case of humans. That is, it allows to answer the question: which is the best policy if the agent does not take the strategic action a_0 some percentage of the time?

This is done as follows: with probability $p = 0.7$, the agent choses the action a_0 (that is, the output of the epsilon-greedy algorithm) and with probability $1 - p = 0.3$, the agent choses randomly among all other actions, with a uniform probability distribution of 0.1. We compute the next state with this action and its correspondent Q-value, and associate the latter to the Q-value of the state and the action a_0. The results are shown in Fig. 3.3.

2.0.4 *Q-Learning*

We create a 4D numpy array with size $(6 \times N_x \times N_y \times N_z)$ to hold the current Q-values for each state and action. The value of this array is initialised to zero, e.g., the agent has no information about the environment. We set the discount value $\gamma = 0.9$ and the learning rate $\alpha = 0.1$. The algorithm runs n times, and at each run we update the Q-values as follows:

$$Q(s_t, a) = Q(s_t, a) + \alpha \left(R_{t+1} + \gamma \max_a [Q(s_{t+1}, a)] - Q(s_t, a) \right) \tag{2.1}$$

The Q-value is the expected discounted reward the agent will receive if it takes an action (e.g. left) in a given state (e.g. (1, 1, 0)). To calculate the Q-values, we add the reward of our chosen action to the discounted maximum Q-value for that state, and subtract from this our estimate of the Q-value of the current state. This is then multiplied by the learning rate. Say the agent is in position (0,0,1) and moves right. In this case, our new Q-value, Q ("right", 0, 0, 1) will remain 0 because we get no reward by moving to $(1, 0, 1)$. If we move from the current state to the goal, we get a reward, so the table of Q-values gets updated. We run through the same procedure again and update the Q-values accordingly. After enough iterations, it should converge to the correct answer, and we can obtain instructions of what the agent should do at each state, i.e., a mapping from state to action, or policy.

3 CONCLUSIONS

The deterministic results are displayed for different number of iterations: $n = 10^4$ in Fig. 3.1 and $n = 10^5$ in Fig. 3.2. The Q-values are shown in the left panel, while an example of policy is shown in the right panel. We find that after approximately $n = 10^5$ iterations the results do not change any further and they make physical sense for this simple model (the "loss" state is avoided, and the path taken to the goal is the one that minimizes the distance, e.g., the number of steps). We have also checked that the algorithm works well for different values for the *starting*, *goal* and *loss* states.

Notice that in this simple model there is some obvious space-symmetry in our scenario: for instance, from "start" the agent could go to "goal" "all the way up", then "all the way right" and finally "all the way top", or via different permutations of these. It could also go in different paths with different actions at each steps. This is also clear from the Q-value table (left panel) in Fig. 3.2, which shows a perfect diagonal degeneracy $(x - y)$, and also a $(x - y - z)$. This is broken by the code at the action level, giving just one example of policy (right panel).

Non-deterministic results are shown in Fig. 3.3, for $n = 10^5$. We observed that in these case the degeneracies as broken, as some random errors are made. These translates into some obvious "mistaken actions", such as going "down" in state $(1, 0, 0)$ or "top" in state $(3, 1, 2)$. As we argued, these could model some kind of random behaviour.

We conclude by saying that this algorithm could also be extended by adding another fourth coordinate. This would apply for instance to a situation in which the *goal* state changes with time. For example, it could correspond to the case in which the drone *follows* another object, or flies inside a tunnel.

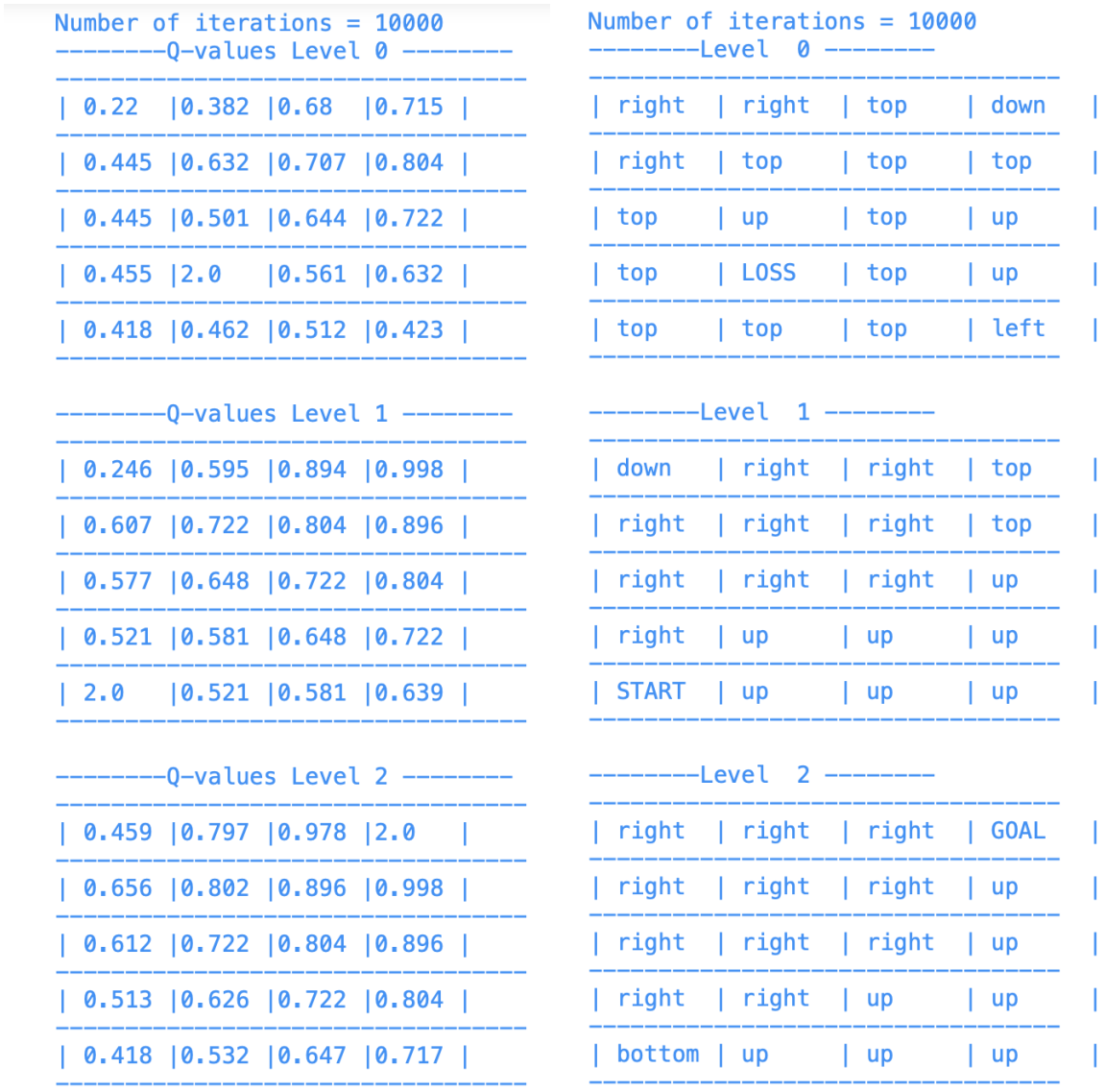

Figure 3.1: Deterministic results after $n = 10^4$ iterations. The left panel shows the Q-values, while the right panel shows the policy.

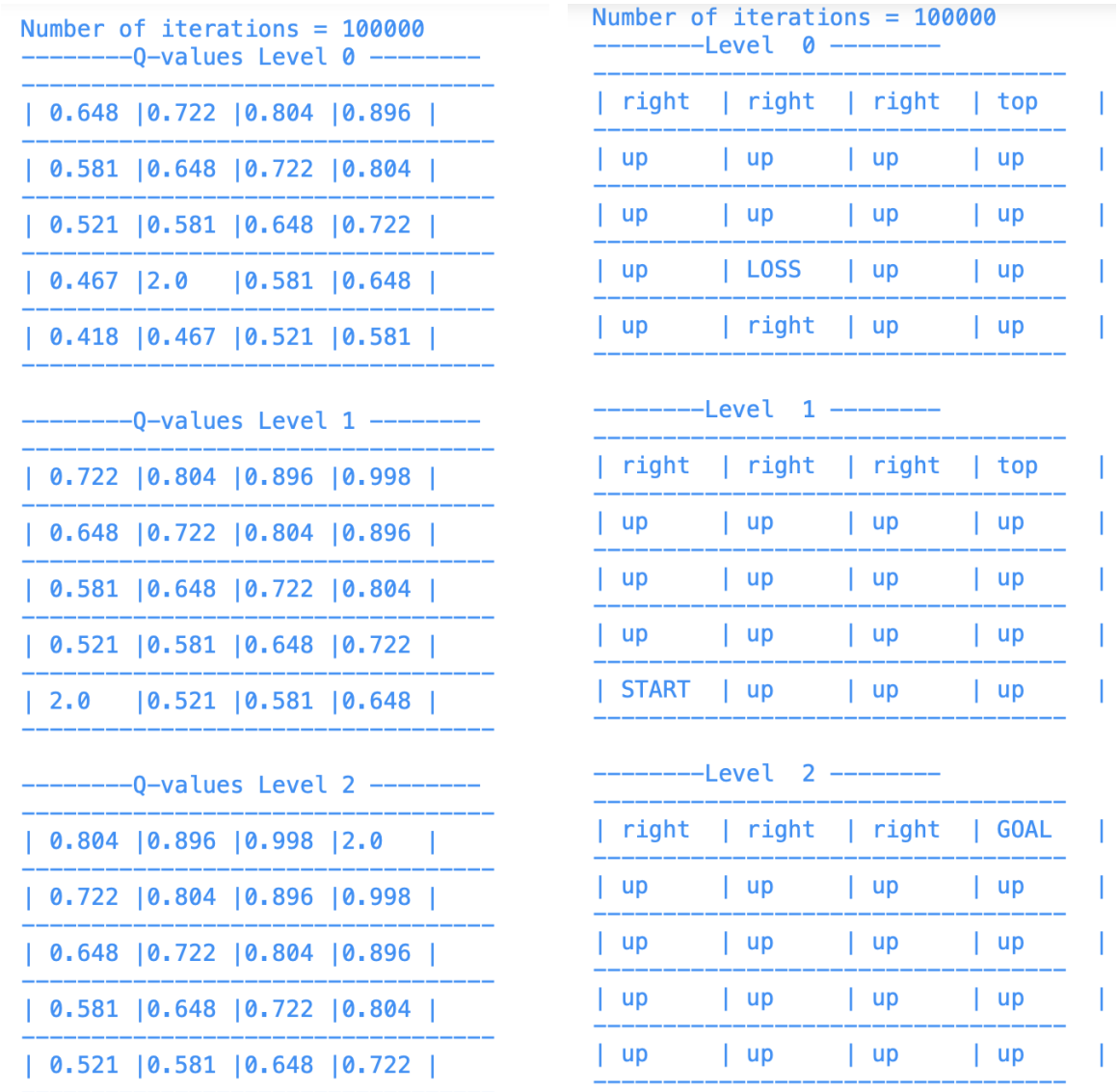

Figure 3.2: Same as Fig. 3.1 for $n = 10^5$ iterations.

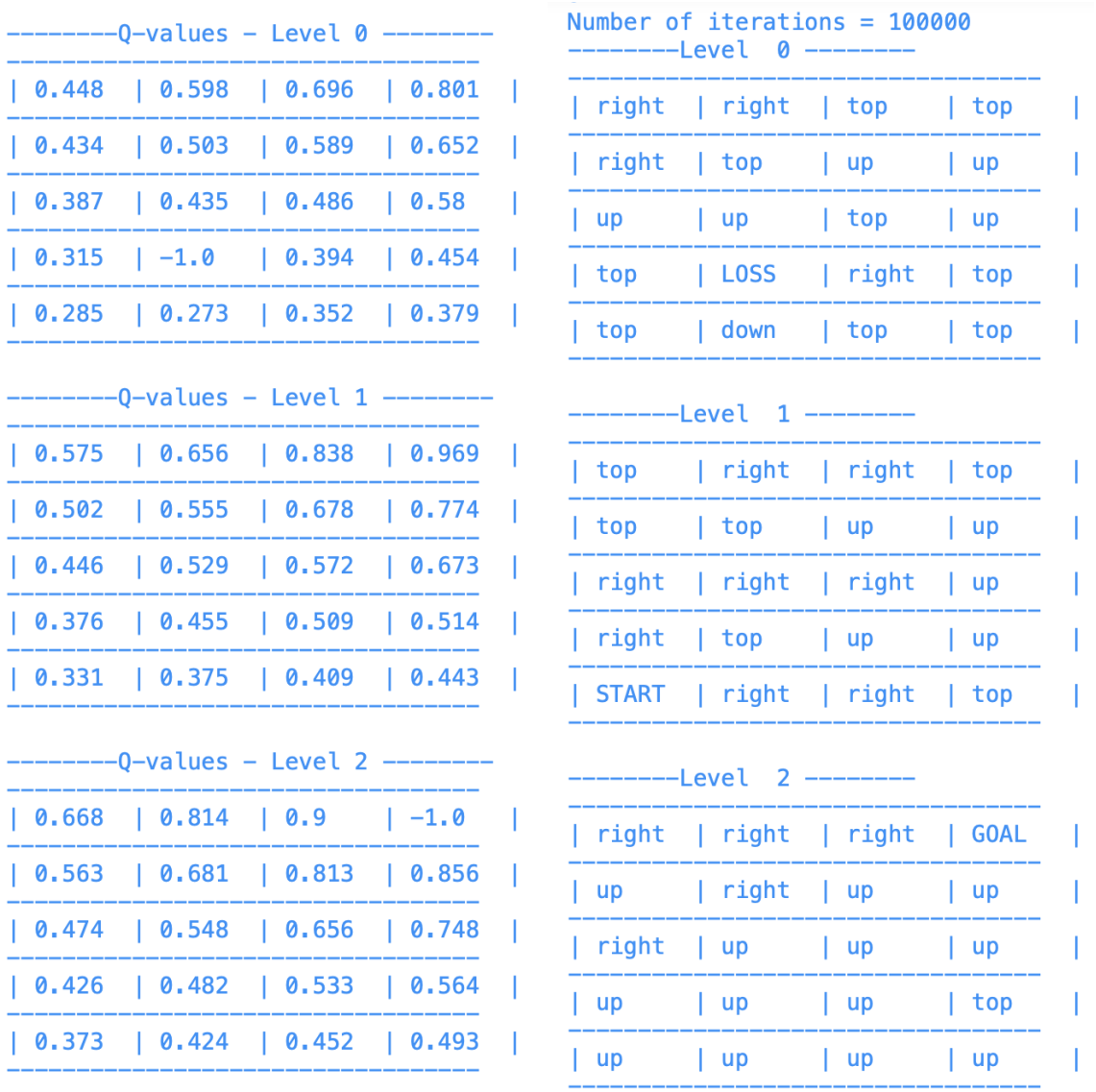

Figure 3.3: Non-deterministic results after $n = 10^5$ iterations.