Q-Learning Applied to a Simple Board Game

João Brito, M9984
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
joao.pedro.brito{at}ubi.pt

***Abstract*—In this project, a reinforcement learning technique (Q-Learning) will be employed to help an agent navigate through a coloured board. Based on this environment, several experiments will be conducted to determine the impact of specific hyper-parameters on the agent's behaviour.**

I. Introduction

Unlike Supervised Learning, where we have access to labelled instances and try to predict new ones, or Unsupervised Learning, where we try to highlight patterns in the data, Reinforcement Learning is very much based on giving an agent a positive or negative reward, based on how it performs. That type of learning is done by repeating tasks to build experience and, hopefully, avoid the same mistakes.

The present document will give an overview of a problem that can be solved using the aforementioned techniques and, in specific, how certain parameters affect the decision-making process of our agent.

II. Game Overview

Reinforcement Learning algorithms usually take place within a game environment. To that end, this work consisted in guiding an agent with only four allowed actions (go left, up, right or down) from a starting position to a finishing one in a NxN board. As figure 1 shows, there are three types of cells:

- **blue**: the agent will want to traverse the board with these cells, which give a penalty of 1 lost point;
- **red**: these are the cells to be avoided and, depending on the experiment, they will either penalise the agent with 100 or 5 lost points;
- **yellow**: these represent the starting and finishing positions, with the latter rewarding the agent with 50 points.

Based on the description above, the agent's goal is to minimise the penalties received during the traversal of the board (i.e. take the shortest path between the top yellow cell and the bottom counterpart, while avoiding the red cells as much as possible).

To avoid backtracking to the original position, the top yellow cell equates to a penalty of 50 lost points, encouraging the agent to never choose that cell, after starting the traversal.

Finally, section IV will have examples with either 30% or 50% of the total cells in a board marked as red.

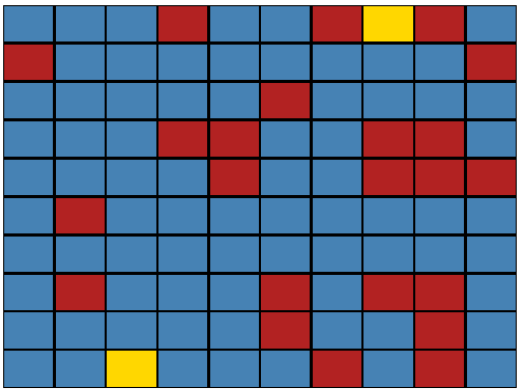

Fig. 1: Example of a 10x10 board with 30% red cells.

III. Overview Of Q-Learning

To guide the agent, one of the better known algorithms is Q-Learning. Its simplicity and intuitive reasoning are key factors, especially in situations where we can store every state the agent can be in. Thus, there are 2 main data structures to keep in mind:

- **Q-Table**: this NxA matrix (with N being the number of possible states and A the number of allowed actions in each state) represents the knowledge gathered by the agent at any time instance t. The purpose of Q-Learning is to build a Q-Table such that the agent can confidently choose actions that maximise the score at the end of the game.
- **Rewards Policy**: this structure is responsible for giving the agent the notion of immediate rewards by storing the reward/penalty of every possible state transition. It can be represented as a N^2xN^2 matrix or, as chosen for this project, a directed graph with N nodes and bidirectional edges. The graph has the advantage of being incredibly more storage efficient when compared to a sparse matrix like the one mentioned (we would have N^2 columns for each line but only 2, 3 or 4 cells of those columns would have meaningful values). The sparse nature of the matrix comes from the fact that each state only has direct access to, at most, 4 other states (figure 1).

As an example, a 10x10 board (i.e. 100 states) equates to a 100x100 matrix (i.e. 10000 cells). Using Python, this would take 80112 bytes of memory, whereas a directed graph (from the NetworkX [1] package) with 100 nodes and 2 edges linking each node pair would only amount to approximately 192 bytes.

This algorithm is mainly focused on the balance between immediate rewards (from nearby states) and future rewards (estimates of what is to come). In addition, the agent has to consider the binomial "exploration-exploitation" (i.e. roam around the environment or stick to the present knowledge).

Below is the pseudocode for Q-Learning, as described above [2]:

Algorithm 1: Q-Learning

input : B (NxN matrix with A actions allowed), R (directed graph with N nodes)
output: Q (NxA matrix with the knowledge obtained)
$Q(s, a) \leftarrow$ initialise_Q_table()
while $episode\_count < max\_episodes$ **do**
  $s \leftarrow start\_state$
  while $s! = end\_state$ **do**
    $a \leftarrow$ using ϵ-greedy, choose an action from $Q(s, a)$
    $s' \leftarrow$ perform action a from state s
    $Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, s') + \gamma * \max_a Q(s', a) - Q(s, a)]$
    $s \leftarrow s'$
  end
end

Some notes regarding the pseudocode above:

- **Q-Table initialisation**: if an action is impossible in a given state, initialise the correspondent position with *None* or some kind of flag to indicate just that.
- **Hyper-parameters**: α is a small number that determines how much the Q-Table values are updated in each time iteration t (i.e. the commonly called learning rate); γ, usually selected from the range [0,1], balances the immediate ($R(s, s')$) and future ($\max_a Q(s', a)$) rewards; ϵ tells the agent whether to explore or exploit.
- **ϵ-greedy policy**: given that ϵ is constrained in the range [0,1] and represents a probability, we should choose and action at random with probability ϵ (exploration). Otherwise, we exploit the current knowledge of the Q-Table.

IV. Experiments

In this section, a few experimental exercises will take place, with respect to the way changes in hyper-parameters affect the agent's abilities and the overall complexity of the game (for example, board size and percentage of red cells).

Note that the path taken by the agent will be highlighted in white.

A. Base Configuration

Both 10x10 boards in this subsection were solved with parameters that, during the tests made, seem to allow the agent to reach its goal with relative ease:

- α = 0.5 with linear decay to 0.15.
- γ = 0.1 with linear growth to 0.4.
- ϵ = 1.0 with linear decay to 0.1.

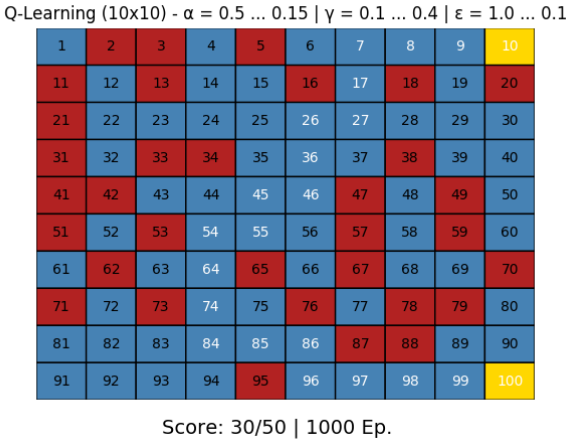

Fig. 2: First example of a successful run.

Figure 2 shows a successful case where the agent had to move somewhat far to the left in order to uncover the best path.

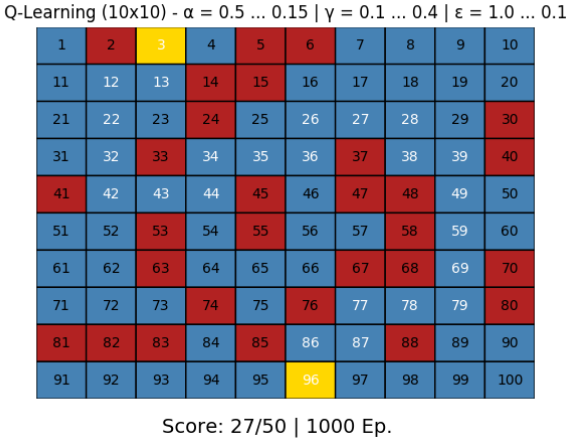

Fig. 3: Second example of a successful run.

In figure 3 we can also observe a successful case were the agent's path has 2 distinct curves that allowed it to avoid the red walls.

From the two examples shown so far we can derive the following conclusion: on its way to the goal cell, the agent moves in straight lines, to minimise any penalties, and only makes turns or curves when needed.

B. Changes in Hyper-parameters

1) α - learning rate:

This hyper-parameter controls how quickly we make strides towards convergence. In other words, it determines how much the Q-Table is updated with respect to t. As a test, the following 2 figures show the same board and final score but with a big difference: in figure 4, the value of α was set to a really low value and that alteration lead to the need of more episodes until convergence (about 4800 episodes); on the other hand, if we linearly grow α as training progresses (an increase to just 0.1 and everything else remaining the same) the number of episodes needed is reduced to just 1000 (figure 5).

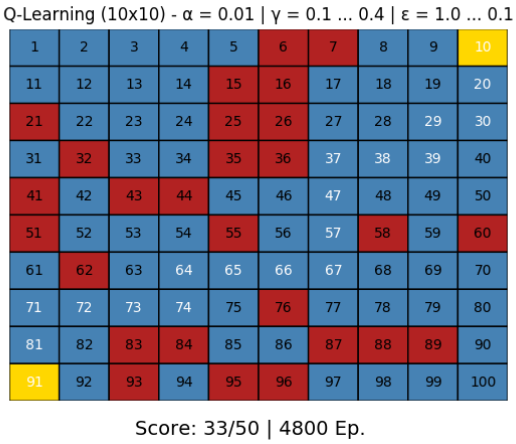

Fig. 4: Example with a low value of α.

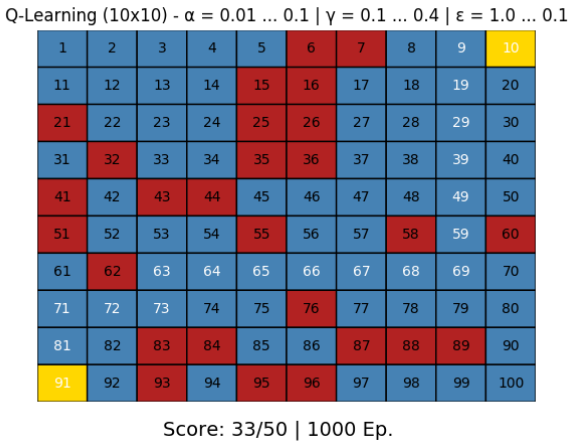

Fig. 5: Example with a growing value of α.

This extreme example shows how small values for α delay convergence and how a simple growth policy can help the agent during training. By taking a look at the Q-Table update formula in the Q-Learning pseudocode, we can see that α is affecting the sum of immediate and future rewards. A smaller α will cause the values in the Q-Table to be more cluttered together and the agent can suffer from a fuzzier sense of direction (in figure 4 the final path has unnecessary turns). On the other hand, a bigger value will give clearer directions to the agent. Obviously, the value shouldn't be that much bigger, otherwise the agent will not learn properly.

2) γ - balance between immediate and future rewards:

The hyper-parameter γ is really important because it can give the agent a short sight or a sense of long term direction. In contrast to every other example, the following figures have red cells with a penalty of just 5 lost points (instead of the traditional 100):

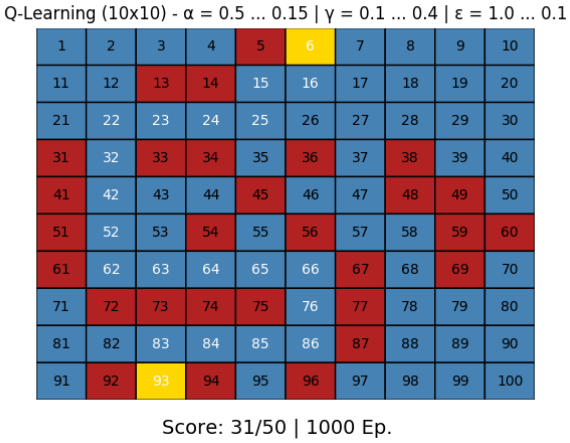

Fig. 6: Case where the agent opted to follow the blue cells.

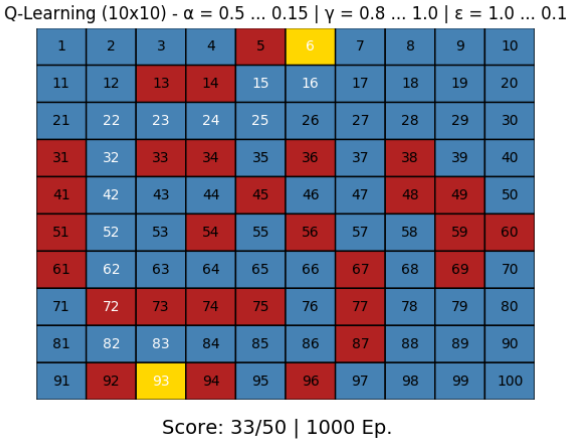

Fig. 7: Case where the agent took a shortcut (a red cell) and ended up getting a better score.

The behaviour shown in figures 6 and 7 is really interesting: with a bigger value of γ, the agent realised that by breaking a wall and taking a penalty of 5 lost points, it would end up benefiting it in the long run, instead of just sticking to the familiar blue cells.

C. Board Size

With this test, the main purpose was to add more complexity to the game: the board got bigger (15x15), more red cells were distributed at random (50%) and the 2 yellow cells were placed in opposite corners, maximising the length of the ideal path:

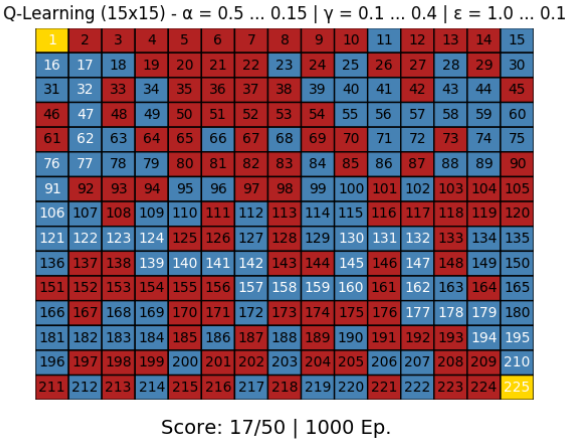

Fig. 8: Example of a successful path on a bigger board.

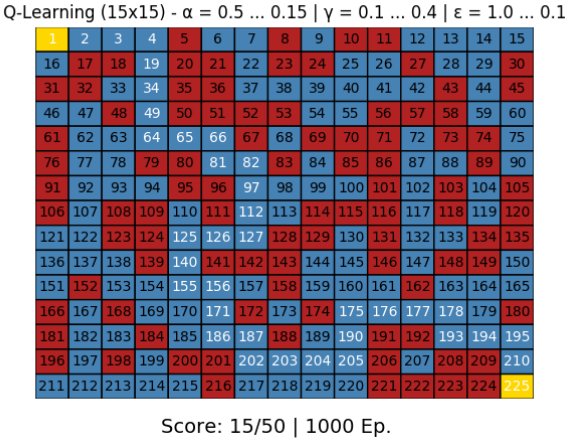

Fig. 9: Another example of a great run on a 15x15 board.

Once again, the agent was able to go from one side to the other, while taking the shortest route. In spite of being larger and more complex, the board was generated and solved within the time frame of the previous tests.

V. Conclusion

The project described in this document showed how a problem can be solved with Reinforcement Learning strategies, like Q-Learning. Moreover, it gave examples of how hyper-parameter tweaking can alter the quality of the generated solutions and/or the number of episodes needed to converge to the optimal path. Despite being a toy exercise, it brought forward transcendent concepts in this sub-area of Machine Learning.

References

[1] NetworkX: Network Analysis in Python. [Online] https://networkx.github.io/

[2] Q-Table update formula. [Online] https://subscription.packtpub.com/book/data/9781789345803/4/ch04lvl1sec33/the-learning-parameters-alpha-gamma-and-epsilon