An introduction to Q-Learning: reinforcement learning

This article is the second part of my “Deep reinforcement learning” series. The complete series shall be available both on Medium and in videos on my YouTube channel.

In the first part of the series we learnt the **basics of reinforcement learning**.

Q-learning is a values-based learning algorithm in reinforcement learning. In this article, we learn about Q-Learning and its details:

- What is Q-Learning ?
- Mathematics behind Q-Learning
- Implementation using python

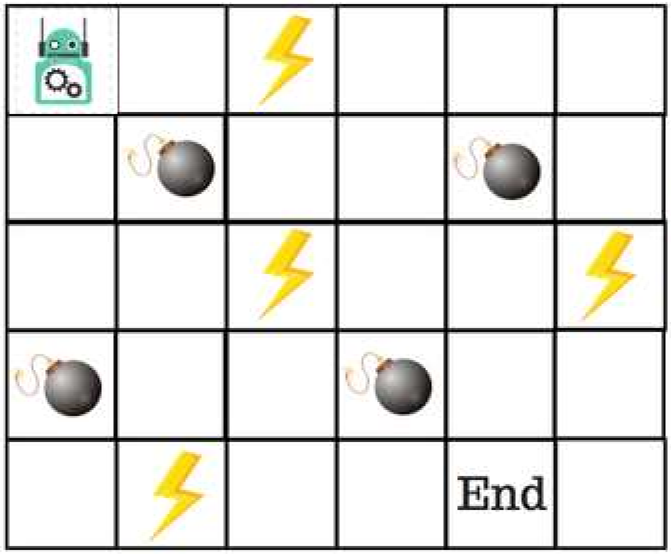
Q-Learning — a simplistic overview

Let’s say that a **robot** has to cross a **maze** and reach the end point. There are **mines**, and the robot can only move one tile at a time. If the robot steps onto a mine, the robot is dead. The robot has to reach the end point in the shortest time possible.

The scoring/reward system is as below:

1. The robot loses 1 point at each step. This is done so that the robot takes the shortest path and reaches the goal as fast as possible.
2. If the robot steps on a mine, the point loss is 100 and the game ends.
3. If the robot gets power ⚡, it gains 1 point.
4. If the robot reaches the end goal, the robot gets 100 points.

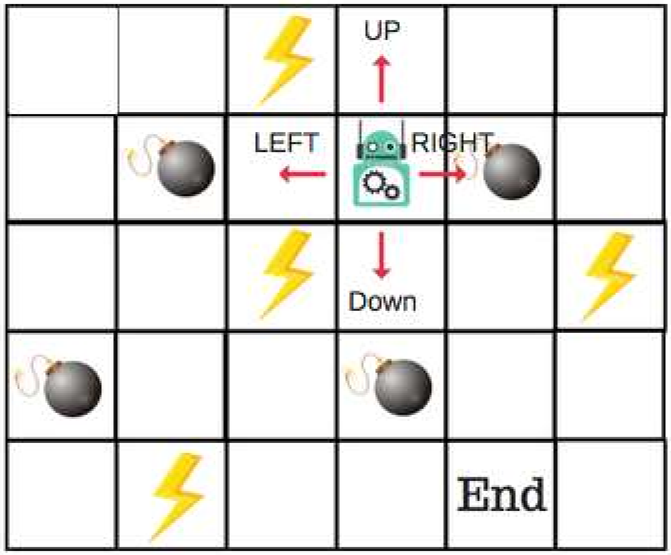
Now, the obvious question is: **How do we train a robot to reach the end goal with the shortest path without stepping on a mine?**



So, how do we solve this?

Introducing the Q-Table

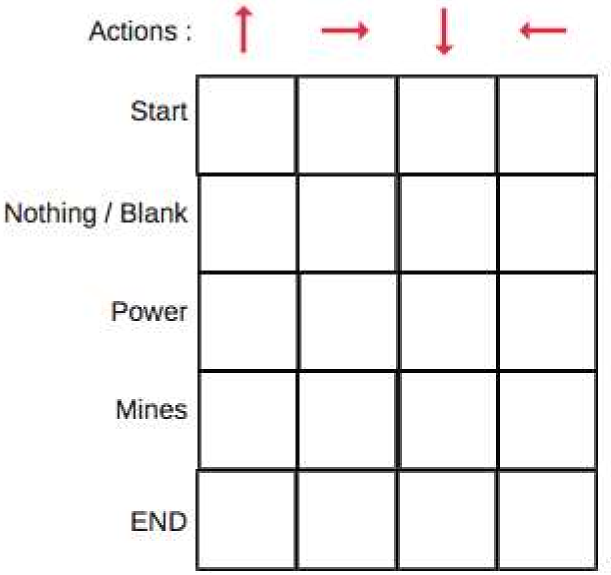
Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.



There will be four numbers of actions at each non-edge tile. When a robot is at a state it can either move up or down or right or left.

So, let’s model this environment in our Q-Table.

In the Q-Table, the columns are the actions and the rows are the states.



Actions :	↑	→	↓	←
Start				
Nothing / Blank				
Power				
Mines				
END				

Each Q-table score will be the maximum expected future reward that the robot will get if it takes that action at that state. This is an iterative process, as we need to improve the Q-Table at each iteration.

But the questions are:

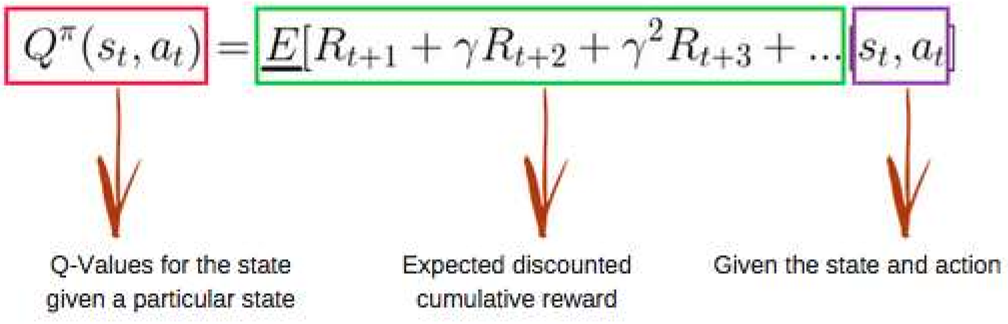
- How do we calculate the values of the Q-table?
- Are the values available or predefined?

To learn each value of the Q-table, we use the **Q-Learning algorithm.**

Mathematics: the Q-Learning algorithm

Q-function

The **Q-function** uses the Bellman equation and takes two inputs: state (**s**) and action (**a**).



$$Q^{\pi}(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$

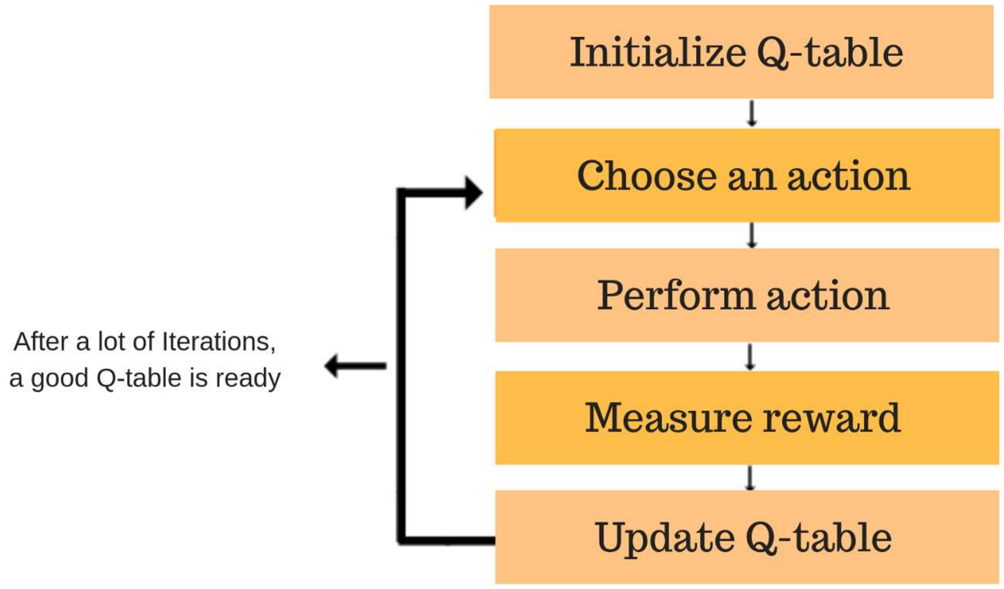
Using the above function, we get the values of **Q** for the cells in the table.

When we start, all the values in the Q-table are zeros.

There is an iterative process of updating the values. As we start to explore the environment**,** the Q-function gives us better and better approximations by continuously updating the Q-values in the table.

Now, let’s understand how the updating takes place.

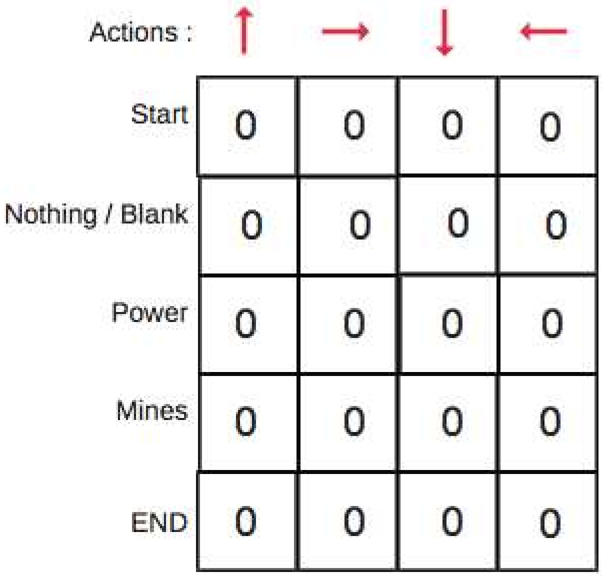
Introducing the Q-learning algorithm process



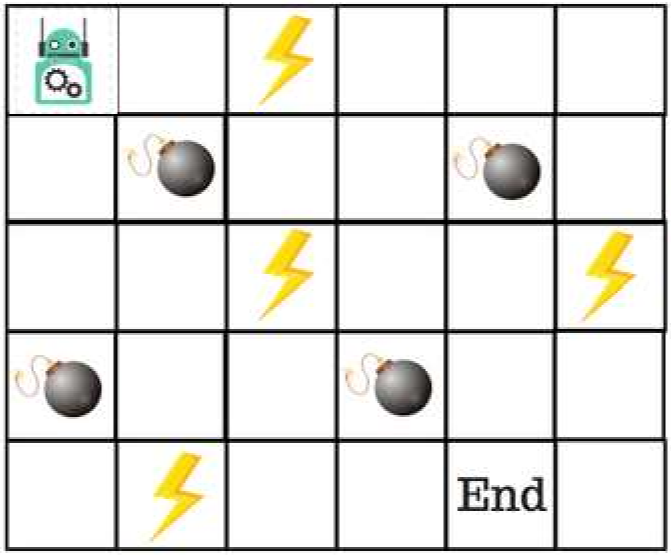
Each of the colored boxes is one step. Let’s understand each of these steps in detail.

Step 1: initialize the Q-Table

We will first build a Q-table. There are n columns, where n= number of actions. There are m rows, where m= number of states. We will initialise the values at 0.



Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0



In our robot example, we have four actions (a=4) and five states (s=5). So we will build a table with four columns and five rows.

Steps 2 and 3: choose and perform an action

This combination of steps is done for an undefined amount of time. This means that this step runs until the time we stop the training, or the training loop stops as defined in the code.

We will choose an action (a) in the state (s) based on the Q-Table. But, as mentioned earlier, when the episode initially starts, every Q-value is 0.

So now the concept of exploration and exploitation trade-off comes into play. This article has more details.

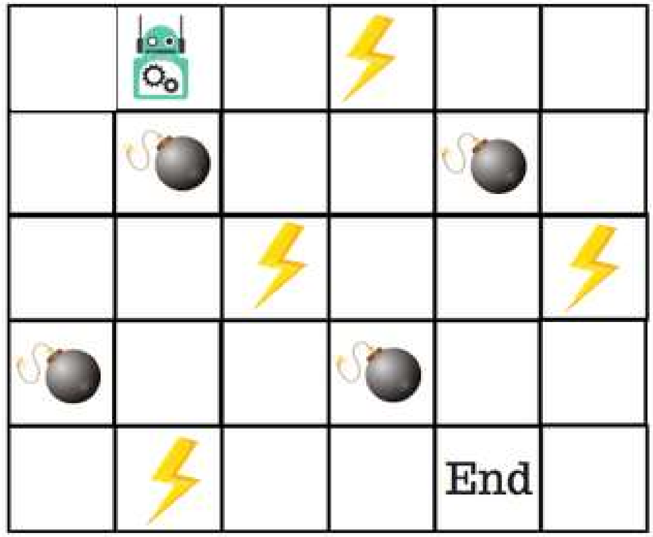
We'll use something called the **epsilon greedy strategy**.

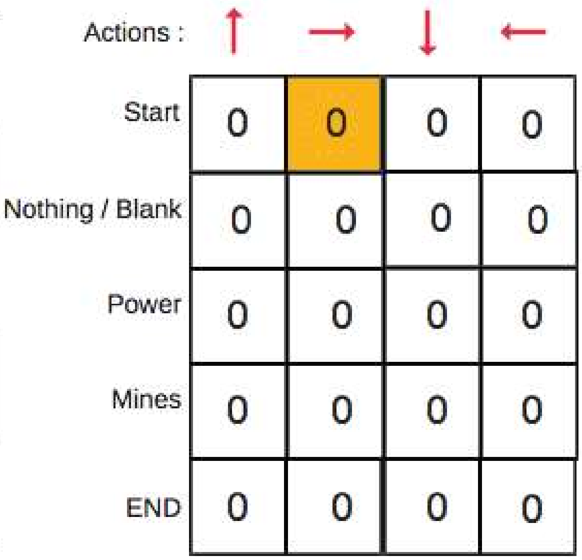
In the beginning, the epsilon rates will be higher. The robot will explore the environment and randomly choose actions. The logic behind this is that the robot does not know anything about the environment.

As the robot explores the environment, the epsilon rate decreases and the robot starts to exploit the environment.

During the process of exploration, the robot progressively becomes more confident in estimating the Q-values.

For the robot example, there are four actions to choose from: up, down, left, and right. We are starting the training now — our robot knows nothing about the environment. So the robot chooses a random action, say right.



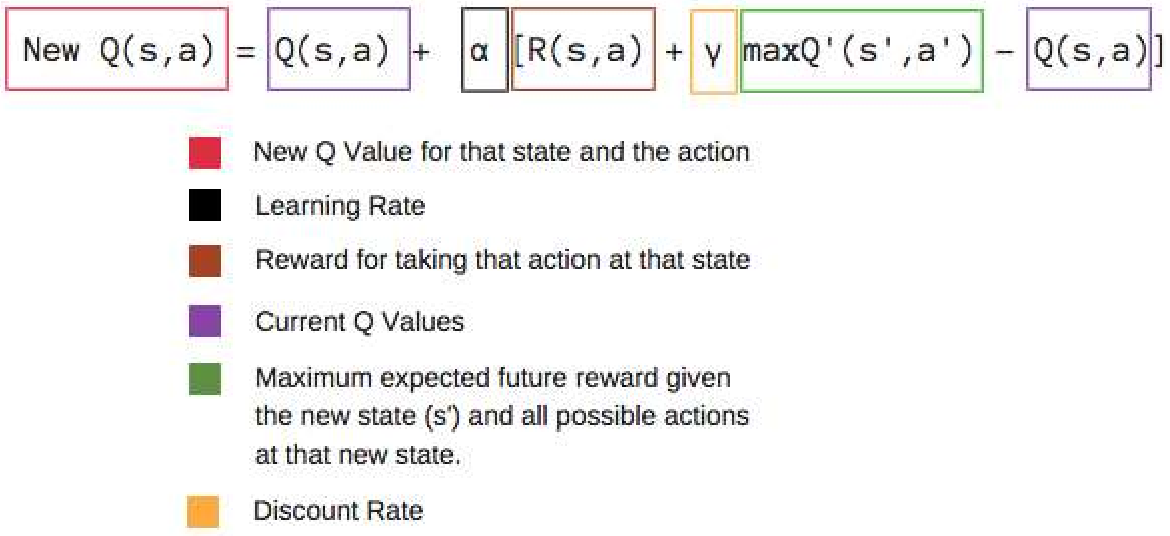


Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

We can now update the Q-values for being at the start and moving right using the Bellman equation.

Steps 4 and 5: evaluate

Now we have taken an action and observed an outcome and reward.We need to update the function Q(s,a).



$$\boxed{\text{New } Q(s,a)} = \boxed{Q(s,a)} + \boxed{\alpha}\,\boxed{[R(s,a)} + \boxed{\gamma}\,\boxed{\max Q'(s',a')} - \boxed{Q(s,a)}]$$

🟥 New Q Value for that state and the action

⬛ Learning Rate

🟫 Reward for taking that action at that state

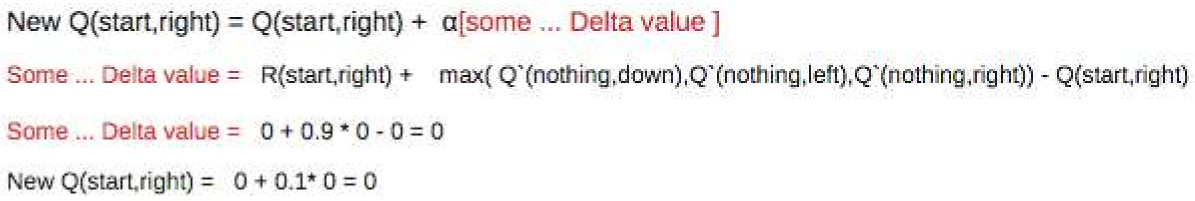
🟪 Current Q Values

🟩 Maximum expected future reward given the new state (s') and all possible actions at that new state.

🟧 Discount Rate

In the case of the robot game, to reiterate the scoring/reward structure is:

- **power** = +1
- **mine** = -100
- **end** = +100

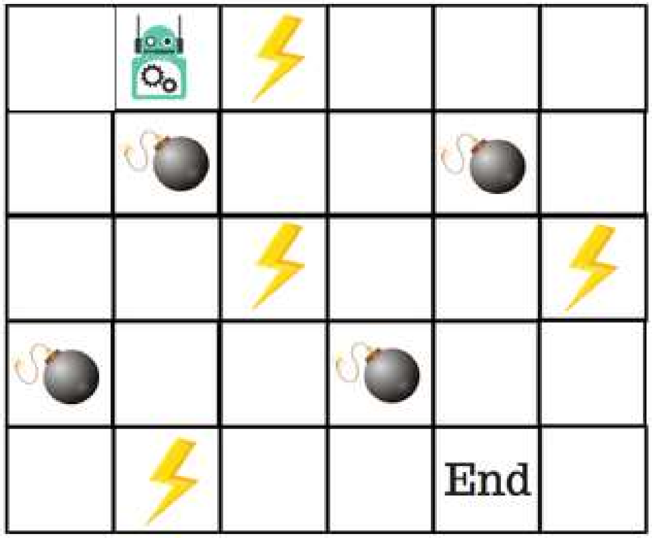


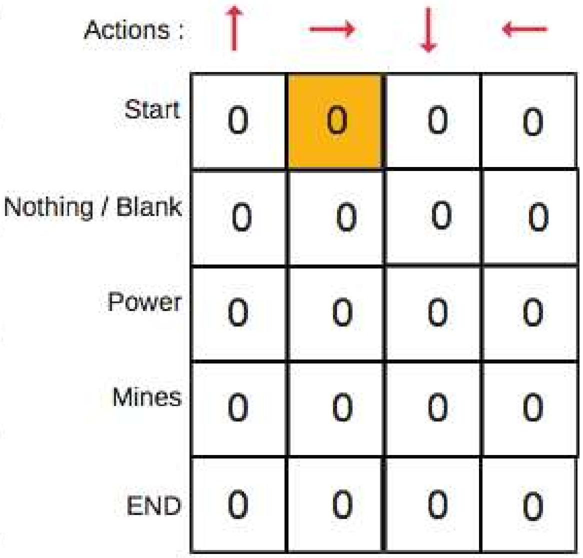
New Q(start,right) = Q(start,right) + α[some ... Delta value]

Some ... Delta value = R(start,right) + max(Q`(nothing,down),Q`(nothing,left),Q`(nothing,right)) - Q(start,right)

Some ... Delta value = 0 + 0.9 * 0 - 0 = 0

New Q(start,right) = 0 + 0.1* 0 = 0





Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

We will repeat this again and again until the learning is stopped. In this way the Q-Table will be updated.

At last…let us recap

- Q-Learning is a value-based reinforcement learning algorithm which is used to find the optimal action-selection policy using a Q function.
- Our goal is to maximize the value function Q.
- The Q table helps us to find the best action for each state.
- It helps to maximize the expected reward by selecting the best of all possible actions.
- Q(state, action) returns the expected future reward of that action at that state.
- This function can be estimated using Q-Learning, which iteratively updates Q(s,a) using the **Bellman equation.**
- Initially we explore the environment and update the Q-Table. When the Q-Table is ready, the agent will start to exploit the environment and start taking better actions.