RGPVNOTES.IN

Program : **B.E**

Subject Name: **Machine Learning**

Subject Code: **CS-8003**

Semester: **8th**

Reinforcement learning – geeks for geeks

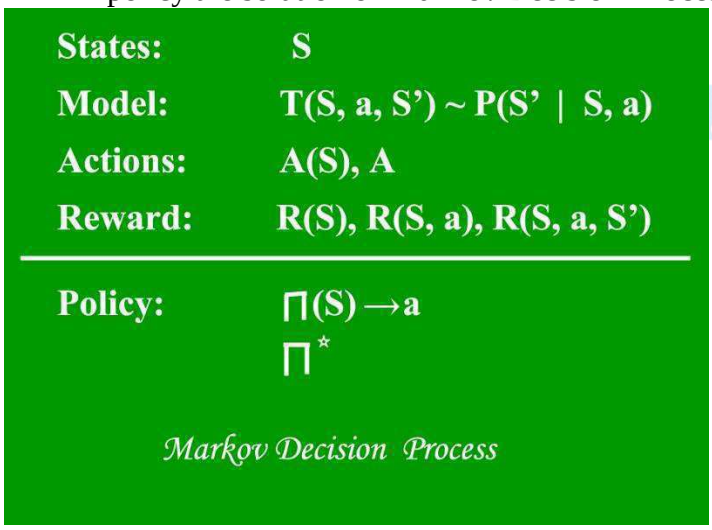
Markov Decision Process

Reinforcement Learning :

Reinforcement Learning is a type of Machine Learning. It allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behavior; this is known as the reinforcement signal.

There are many different algorithms that tackle this issue. As a matter of fact, Reinforcement Learning is defined by a specific type of problem, and all its solutions are classed as Reinforcement Learning algorithms. In the problem, an agent is supposed to decide the best action to select based on his current state. When this step is repeated, the problem is known as a **Markov Decision Process**.
A **Markov Decision Process (MDP)** model contains:

- A set of possible world states S.
- A set of Models.
- A set of possible actions A.
- A real valued reward function R(s,a).
- A policy the solution of **Markov Decision Process**.



What is a State?

A **State** is a set of tokens that represent every state that the agent can be in.

What is a Model?

A **Model** (sometimes called Transition Model) gives an action's effect in a state. In particular, T(S, a, S') defines a transition T where being in state S and taking an action 'a' takes us to state S' (S and S' may be same). For stochastic actions (noisy, non-deterministic)

we also define a probability P(S'|S,a) which represents the probability of reaching a state S' if action 'a' is taken in state S. Note Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

What is Actions?

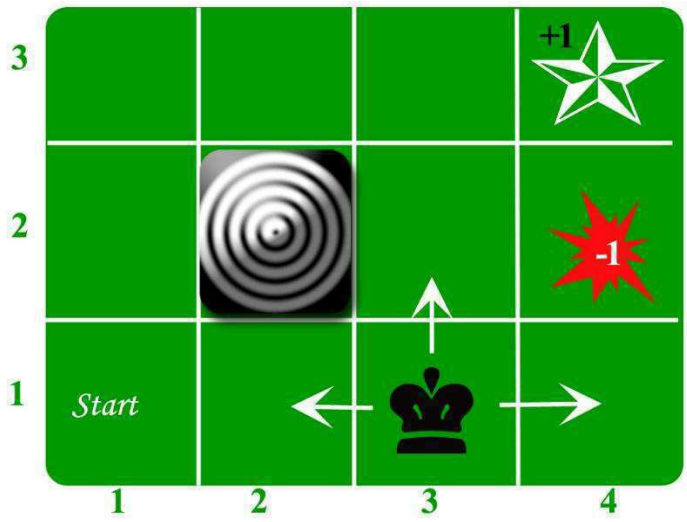
An **Action** A is set of all possible actions. A(s) defines the set of actions that can be taken being in state S.

What is a Reward?

A **Reward** is a real-valued reward function. R(s) indicates the reward for simply being in the state S. R(S,a) indicates the reward for being in a state S and taking an action 'a'. R(S,a,S') indicates the reward for being in a state S, taking an action 'a' and ending up in a state S'.

What is a Policy?

A **Policy** is a solution to the Markov Decision Process. A policy is a mapping from S to a. It indicates the action 'a' to be taken while in state S.
Let us take the example of a grid world:



An agent lives in the grid. The above example is a 3*4 grid. The grid has a START state(grid no 1,1). The purpose of the agent is to wander around the grid to finally reach the Blue Diamond (grid no 4,3). Under all circumstances, the agent should avoid the Fire grid (orange color, grid no 4,2). Also the grid no 2,2 is a blocked grid, it acts like a wall hence the agent cannot enter it.

The agent can take any one of these actions: **UP, DOWN, LEFT, RIGHT**
Walls block the agent path, i.e., if there is a wall in the direction the agent would have taken, the agent stays in the same place. So for example, if the agent says LEFT in the START grid he would stay put in the START grid.

First Aim: To find the shortest sequence getting from START to the Diamond. Two such sequences can be found:

- **RIGHT RIGHT UP UP RIGHT**
- **UP UP RIGHT RIGHT RIGHT**

Let us take the second one (UP UP RIGHT RIGHT RIGHT) for the subsequent discussion. The move is now noisy. 80% of the time the intended action works correctly. 20% of the time the action agent takes causes it to move at right angles. For example, if the agent says UP the probability of going UP is 0.8 whereas the probability of going LEFT is 0.1 and probability of going RIGHT is 0.1 (since LEFT and RIGHT is right angles to UP).

The agent receives rewards each time step:-

- Small reward each step (can be negative when can also be term as punishment, in the above example entering the Fire can have a reward of -1).
- Big rewards come at the end (good or bad).
- The goal is to Maximize sum of rewards.

Hidden Markov Models (HMM)

Introduction to Hidden Markov Models (HMM)

A *hidden Markov model* (HMM) is one in which you observe a sequence of emissions, but do not know the sequence of states the model went through to generate the emissions. Analyses of hidden Markov models seek to recover the sequence of states from the observed data.

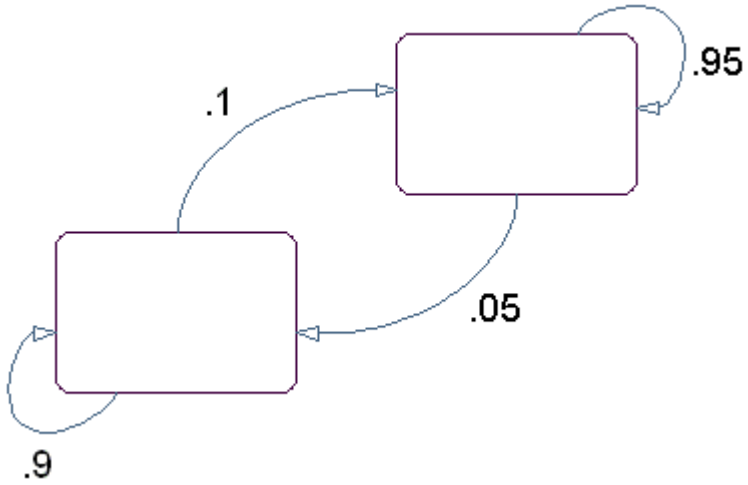
As an example, consider a Markov model with two states and six possible emissions. The model uses:

- A red die, having six sides, labeled 1 through 6.
- A green die, having twelve sides, five of which are labeled 2 through 6, while the remaining seven sides are labeled 1.
- A weighted red coin, for which the probability of heads is .9 and the probability of tails is .1.
- A weighted green coin, for which the probability of heads is .95 and the probability of tails is .05.

The model creates a sequence of numbers from the set {1, 2, 3, 4, 5, 6} with the following rules:

- Begin by rolling the red die and writing down the number that comes up, which is the emission.
- Toss the red coin and do one of the following:
  - If the result is heads, roll the red die and write down the result.
  - If the result is tails, roll the green die and write down the result.
- At each subsequent step, you flip the coin that has the same color as the die you rolled in the previous step. If the coin comes up heads, roll the same die as in the previous step. If the coin comes up tails, switch to the other die.

The state diagram for this model has two states, red and green, as shown in the following figure.



You determine the emission from a state by rolling the die with the same color as the state. You determine the transition to the next state by flipping the coin with the same color as the state.

The transition matrix is:

$$T = \begin{bmatrix} & \\ & \end{bmatrix} = \begin{matrix} 0.9 & 0.05 \\ 0.1 & 0.95 \end{matrix}$$

The emissions matrix is:

$$E = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/2 & 1/10 & 1/10 & 1/10 & 1/10 & 1/10 \end{matrix}$$

The model is not hidden because you know the sequence of states from the colors of the coins and dice. Suppose, however, that someone else is generating the emissions without showing you the dice or the coins. All you see is the sequence of emissions. If you start seeing more 1s than other numbers, you might suspect that the model is in the green state, but you cannot be sure because you cannot see the color of the die being rolled.

Hidden Markov models raise the following questions:

- Given a sequence of emissions, what is the most likely state path?
- Given a sequence of emissions, how can you estimate transition and emission probabilities of the model?
- What is the *forward probability* that the model generates a given sequence?
- What is the *posterior probability* that the model is in a particular state at any point in the sequence?

Bellman eq – nhi mila

value iteration and policy iteration – nhi mila

linear quad regulation –

The theory of optimal control is concerned with operating a dynamic system at minimum cost. The case where the system dynamics are described by a set of linear differential equations and the cost is described by a quadratic function is called the LQ problem. One of the main results in the theory is that the solution

is provided by the **linear–quadratic regulator (LQR)**, a feedback controller whose equations are given below. The LQR is an important part of the solution to the LQG (linear–quadratic–Gaussian) problem. Like the LQR problem itself, the LQG problem is one of the most fundamental problems in control theory.

The settings of a (regulating) controller governing either a machine or process (like an airplane or chemical reactor) are found by using a mathematical algorithm that minimizes a cost function with weighting factors supplied by a human (engineer). The cost function is often defined as a sum of the deviations of key measurements, like altitude or process temperature, from their desired values. The algorithm thus finds those controller settings that minimize undesired deviations. The magnitude of the control action itself may also be included in the cost function.

The LQR algorithm reduces the amount of work done by the control systems engineer to optimize the controller. However, the engineer still needs to specify the cost function parameters, and compare the results with the specified design goals. Often this means that controller construction will be an iterative process in which the engineer judges the "optimal" controllers produced through simulation and then adjusts the parameters to produce a controller more consistent with design goals.

The LQR algorithm is essentially an automated way of finding an appropriate state-feedback controller. As such, it is not uncommon for control engineers to prefer alternative methods, like full state feedback, also known as pole placement, in which there is a clearer relationship between controller parameters and controller behavior. Difficulty in finding the right weighting factors limits the application of the LQR based controller synthesis.

Linear–quadratic–Gaussian control

In control theory, the **linear–quadratic–Gaussian** (**LQG**) **control problem** is one of the most fundamental optimal control problems. It concerns linear systems driven by additive white Gaussian noise. The problem is to determine an output feedback law that is optimal in the sense of minimizing the expected value of a quadratic cost criterion. Output measurements are assumed to be corrupted by Gaussian noise and the initial state, likewise, is assumed to be a Gaussian random vector.

Under these assumptions an optimal control scheme within the class of linear control laws can be derived by a completion-of-squares argument.[1] This control law which is known as the **LQG** controller, is unique and it is simply a combination of a Kalman filter (a linear–quadratic state estimator (LQE)) together with a linear–quadratic regulator (LQR). The separation principle states that the state estimator and the state feedback can be designed independently. LQG control applies to both linear time-invariant systems as well as linear time-varying systems, and constitutes a linear dynamic feedback control law that is easily computed and implemented: the LQG controller itself is a dynamic system like the system it controls. Both systems have the same state dimension.

A deeper statement of the separation principle is that the LQG controller is still optimal in a wider class of possibly nonlinear controllers. That is, utilizing a nonlinear control scheme will not improve the expected value of the cost functional. This version of the separation principle is a special case of the separation principle of stochastic control which states that even when the process and output noise sources are possibly non-Gaussian martingales, as long as the system dynamics are linear, the optimal control separates into an optimal state estimator (which may no longer be a Kalman filter) and an LQR regulator.[2][3]

In the classical LQG setting, implementation of the LQG controller may be problematic when the dimension of the system state is large. The **reduced-order LQG problem** (fixed-order LQG problem) overcomes this by fixing *a priori* the number of states of the LQG controller. This problem is more difficult to solve because it is no longer separable. Also, the solution is no longer unique. Despite these facts numerical algorithms are available[4][5][6][7] to solve the associated optimal projection equations[8][9] which constitute necessary and sufficient conditions for a locally optimal reduced-order LQG controller.[4]

LQG optimality does not automatically ensure good robustness properties.[10] The robust stability of the closed loop system must be checked separately after the LQG controller has been designed. To promote robustness some of the system parameters may be assumed stochastic instead of deterministic. The associated more difficult control problem leads to a similar optimal controller of which only the controller parameters are different.[5]

value function approximation – nhi mila

Direct policy search[edit] – wiki

Partially observable Markov decision process

Diag frm quora

Let's say you want a 2 dimensional robot to move from left to right on a line. At a very high level of simplification the world in which this robot is evolving can be **discretized into 2 states**, state L: "I am on the left", and state R:"I am on the right".

Then for those two states you have 4 possible movements for the robot, called transitions: on the left there is action ll "stay left", action lr "go right", for state R there is action rr "stay right" and action rl "go left". A pretty simple world.

The system is evolving in time, with **S(t) the state of the robot at time t**. The evolution of the system is often not deterministic, and each transition has a certain probability contained in the 2*2 transition matrix T.

Now that you have this first part of the problem, the 'state-space' of states and transitions. You need to introduce a couple of other notions to define a POMDP.

First, it is a Markovian process, which means it has the **markov property** such as for all time t, the state S(t) only depends on the state S(t-1). This explains why your transition matrix is only 2*2, **your next state only depends on the state you are currently in**.

For example if you had a truly random system, with no preference for one transition over another, you would equal probability for each transition for a certain state, here with two states the transition probabilities would be 0.5 everywhere:

If we stopped here, this system would constitute a simple **Markov Process**. It would be a system that you only describe from the outside, and for which at each time t you are certain of the state of the system. **You are not certain of where it will be at t+1, but at t you know the position it is in.**

We have to transform this system into a **decision process.** The states are not changing on their own here, it is the robot that is taking actions, making decisions, to stay or move left or move right. Your model becomes a decision process when you change the level of your description to **consider the actions the robot is taking to change its state**. You need to define a list of actions that your robot can take to affect the evolution of the system, let's say he has one little motor to control its wheels, it can either do "motor off", or "motor on go left", or "motor on and go right".

Usually, we are interested in modeling such process because we want the agent to maximize some function, and behave optimally. Here let's imagine we want the robot to **maximize the number of time he has switched position.** To accomplish this its decision algorithm is pretty simple, **if Left go Right, if Right go Left.**

The Markov decision process allows for **actions with uncertain outcomes**, for example, **the robot would choose action "go left" but the motor would stop with probability 0.10** or go right with probability 0.15, the robot would stay at the same position with probability 0.25. That's one interesting property of Markov decision processes, it can **model faulty and uncertain behavior.**

Ok so now we have a Markov Decision Process, we have the states, we have actions, and we have transition probabilities from those actions.

Say the robot is picking its action on its own, without you intervening. In this markovian description, to decide its next action the robot has to know its current state, or at least try to guess it. **A classic MDP suppose the robot is always certain of its position** and is then able to make an informed decision . To remind you, here if it knows it is on the right it will turn its motor on and go left, if it is on the left, it will turn its motor on and go right.

To make things more interesting suppose now that to know its position the robot is using an old laser sensor that also has a 25% error rate. The robot would then **never be certain of the real state S(t) it is in**, because the sensor might give false informations. So to know its position he will only rely on the reading from its faulty sensor, but also the list of actions taken in the past, themselves with uncertain outcomes.

This system is described by a **partially observable Markov decision process (POMDP).** This is a computationally intensive problem to solve (P-SPACE-complete for a finite horizon). Partially observable because the agent only has access to limited amount of information about the current state of the system.

In order to make an informed decisions based on this partial data, the agent has to maintain **a list of belief-states** in which it could be at all time, **with their respective probability given the sequence of choice and observations.**

In terms of **learning,** the POMDP can use the same framework than in classic Reinforcement learning algorithms like Q-learning, where at each action or after a series of action you get a reward, but using belief-states instead of states. If you were learnin "on-line" as the robot evolves the reward would have to be propagated to all the beliefs states that could have led to the reward. With enough trials, the agent learns not only the transition probabilities for each action but also the noise in its instruments.

But wether you learn online or through simulation, since the number of sequences of belief states rapidly become intractable you will need efficient **sampling algorithms**(counting the samples to represent the probabilities over the belief states) to explore the belief space in an optimal way, such as point-based algorithms like SARSOP. There is also research on deep nets as function estimators.

As a side note, this is a very frequent configuration in nature and in our daily life. Think about the problem of determining what somebody else is thinking based on his speech. You do not have

access to the real state of mind of the other person, you are inferring it through the indirect use of language, you are limited by what the other is sharing, and what you know of the context. **There is a theory that explains how our brain might resolve this problem using a "mirror system" referred to as the Theory of Mind** (which confusingly refer here to the function, rather than the theory).

This system is a simulator, able to simulate another human being's brain processes, that can have a lot of its parameters tweaked, but **uses your own neural pathways to fill the gaps**. It is a sort of **model-based faulty sensor** that would continuously reproduce the set of states and actions of the other person inside your own mind, along with the environment you think he is evolving in, of his personality etc. Every bit of information you have about him can be fed into the simulation, because it would be using your own mental representations. The information you use is limited by your mental representations, you might have very wrong assumptions or not even be able to pick up on it. For example this system would have a harder time inferring mental states in a cultural context very different from yours. Some social cues and signals might be unknown to you, and your ability to represent a whole life in a different culture will often fall short (like what it could mean to live in a country where human rights are not respected). Once you have tweaked all those parameters the mirror system will produce an answer based on the simulation of how **your** brain "would do with all those constraints". It is a smart approach in the sense that modeling the entirety of the brain processes of the person in front of you would be intractable, so your brain has evolved to make the assumption that a LOT of stuff going on is actually common between two brains, and can be reused to "detect another's state of mind".

Children below 6 would have a great deal of problem turning this system down, they are animists and believe everything has a mind of its own that thinks and feels like ours. Which explain why they are able and enjoy playing with inanimate toys to give them life. It is a way of this system to learn through simulation. They also believe they are at the origin of much of the changes in the world, either because the system hasn't learn enough of the world yet and the cultural knowledge we have of the workings of the mind, or maybe because it is not being inhibited properly yet. This system is also supposedly impaired in autism, explaining part of the difficulties they might have in human interactions.

Q-Learning—a simplistic overview

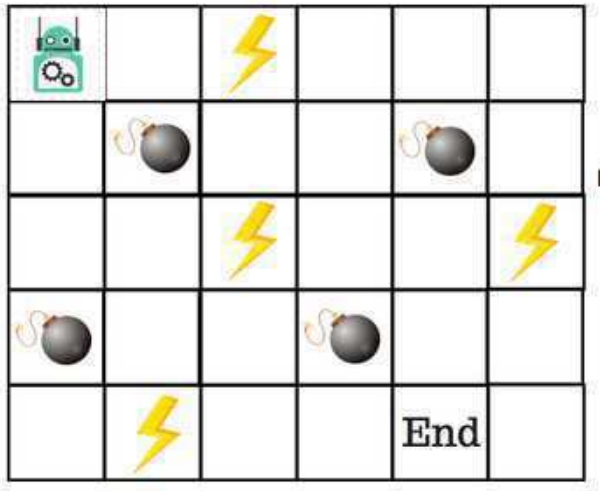
Let's say that a **robot** has to cross a **maze** and reach the end point. There are **mines**, and the robot can only move one tile at a time. If the robot steps onto a mine, the robot is dead. The robot has to reach the end point in the shortest time possible.

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.

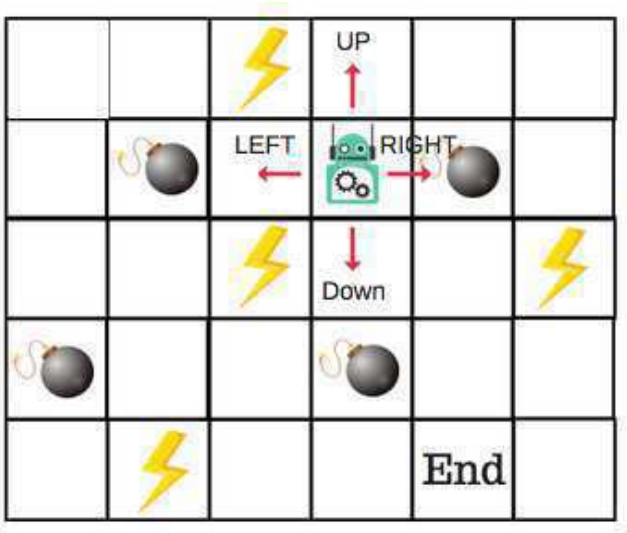
Now, the obvious question is: **How do we train a robot to reach the end goal with the shortest path without stepping on a mine?**



So, how do we solve this?

Introducing the Q-Table

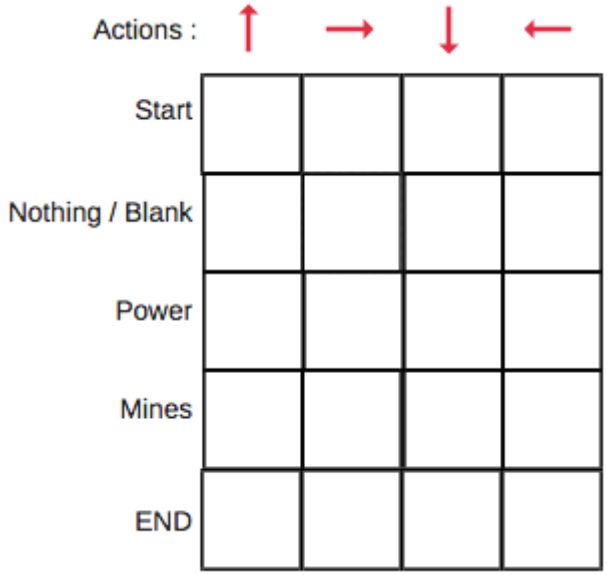
Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.



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.



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:

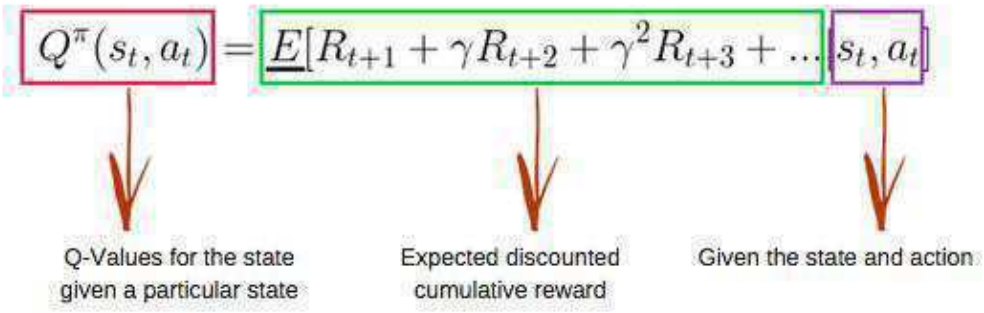
- How do we calculate the values of the Q-table?
- Are the values available or predefined?

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**).



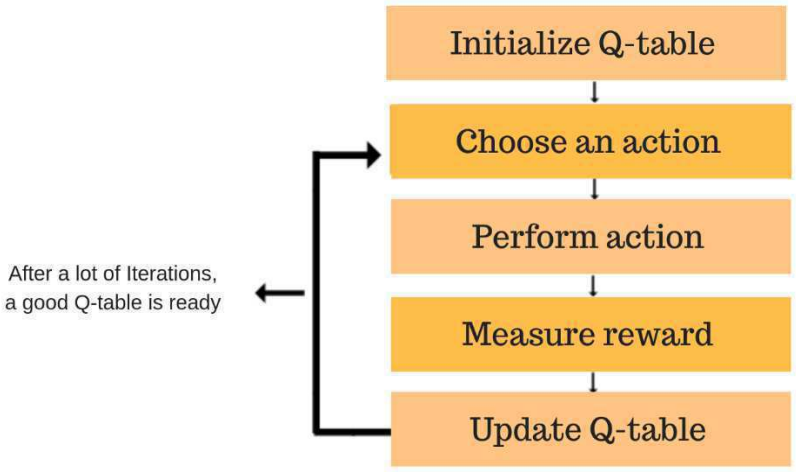
Using the above function, we get the values of **Q** for the cells in the table.

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.

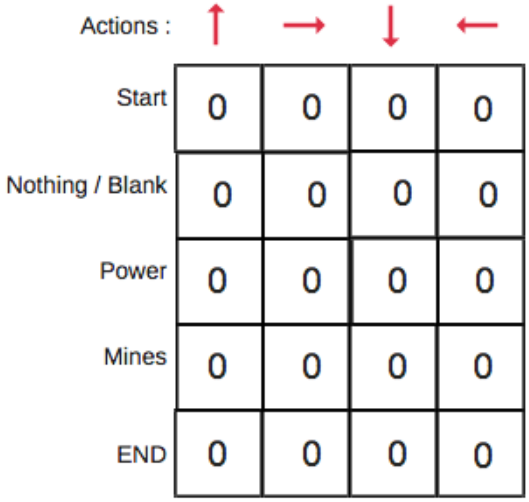
Introducing the Q-learning algorithm process

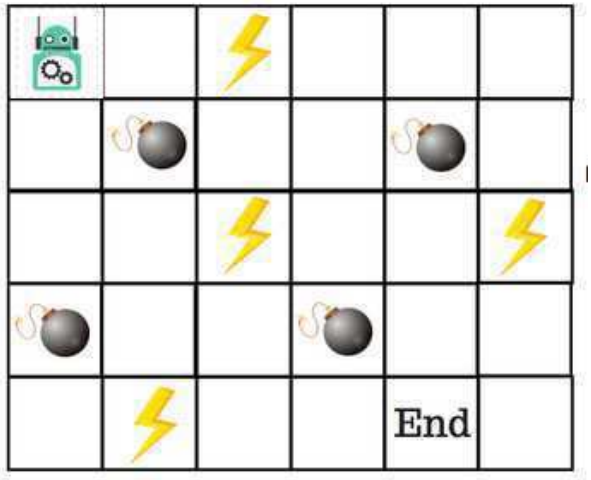


Each of the colored boxes is one step. Let's understand each of these steps in detail.

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.





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.

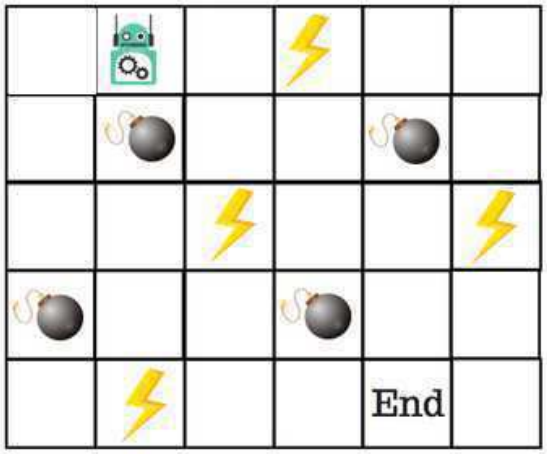
We'll use something called the **epsilon greedy strategy**.

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).

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

- 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:

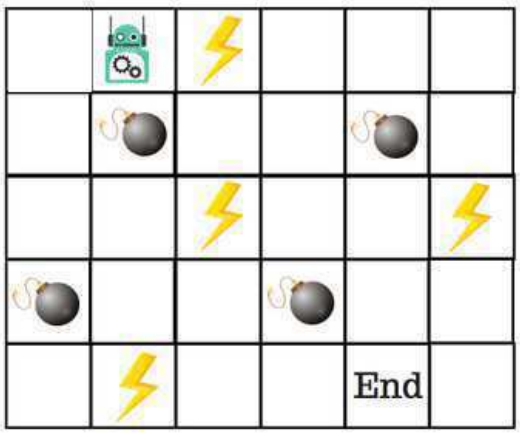
- **power** = +1
- **mine** = -100
- **end** = +100

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.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in