**Q-Learning Algorithm: How to Successfully Teach an Intelligent Agent to Play a Game**

Q-Learning falls under the family of Reinforcement Learning algorithms and, more specifically, under the Value-based methods branch.

# Reinforcement Learning terminology

Before we look at Q-Learning, here’s a quick recap of the fundamental elements of Reinforcement Learning:

* **Agent** — an “intelligent actor” that can interact with its environment, e.g. a player in a game.
* **Environment** — the “world” where that agent “lives” or operates.
* **Action Space**— a list or range of actions the agent can perform.
* **State/Observation Space** — a list or range of possible environment configurations. A state/observation provides information to the agent about its environment (e.g. its location).
* **Reward**— incentive (or disincentive) that we give to the agent when it performs desired (undesired) actions at various states. We can reduce future rewards relative to present rewards by using the **discount factor {gamma(𝛾)}**.
* **Exploration/Exploitation {epsilon(𝜖)}**— enables us to set how much time the agent should spend exploring the environment vs exploiting its existing knowledge about the environment.
* **Episode** — one complete cycle from the start position to the end position. E.g., in the context of a game, an episode would last from the moment your agent starts a new level until it dies or completes the level.
* **Alpha(𝛼)**— learning rate, which influences the learning speed and convergence towards the optimal policy.
* **Policy(𝜋)** —an agent’s strategy to pursue a goal.

# How does Q-Learning work?

## **Difference between Policy-based and Value-based methods**

Q-Learning falls into the category of **Value-based methods**, so let’s start by understanding the difference between Value-based and Policy-based methods.

* **Policy-based methods** — we train the agent **directly** on what action to take in which state. It is described by a **policy function** that can be either **deterministic** (gives the precise action for each state) or **stochastic** (provides a probability distribution over actions).
* **Value-based methods**— we train the agent **indirectly** by teaching it to identify which states (or state-action pairs) are more valuable so that it can be guided by value maximization. It is described by a **value function** where the value of a state is the expected discounted return the agent can get if it starts in that state.

Regardless of which method we use to train our agent, finding the optimal policy function or the optimal value function equates to discovering the **optimal policy(𝜋).**

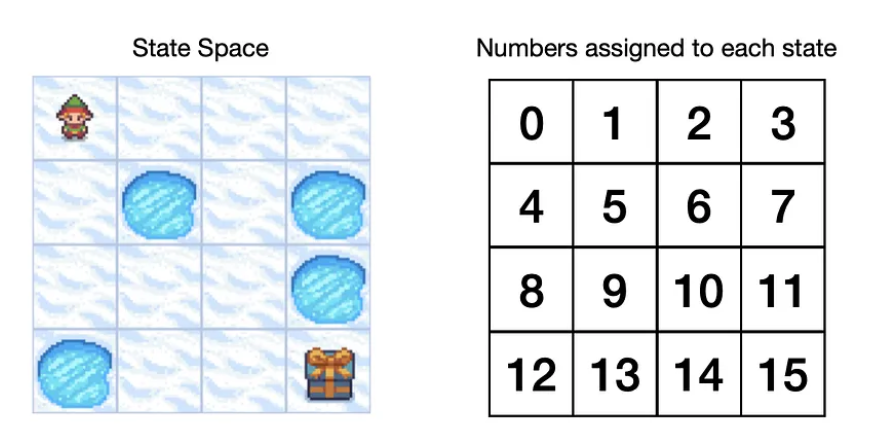
## **Q-function and Q-table**

Since Q-Learning is a Value-based method, we must have a **value function**, which we will call a **Q-function**. Inside, it will have a **Q-table**containing every **state-action pair**.

Training a Q-function is simply finding the values associated with each state-action pair stored in a Q-table. Knowing these values enables the agent to choose the best action at each state.

In the later Python section, we will teach the agent to play a Frozen-Lake game, so let’s use the same game to demonstrate what a Q-table looks like.

Our Frozen-Lake environment will be a 4x4 grid consisting of frozen squares and squares with holes, a total of 16 squares. Each square represents a possible state, which we can label by assigning numbers to them.



The **action space** will consist of 4 distinct actions that our agent can take: **Left(0), Down(1), Right(2), Up(3)**.

Now we can form a Q-table with states as rows and actions as columns.

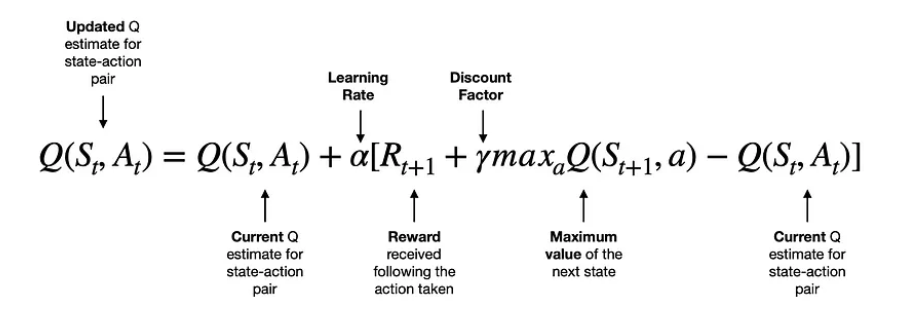
## **Q-Learning algorithm**

Before we look at the actual Q-Learning algorithm, here are a couple more things to note:

* **On-policy vs Off-policy:** Q-Learning is an **off-policy** algorithm, which means that during training, we use different policies for the agent to act (acting policy) and to update the Q-function (updating policy). Meanwhile, On-policy means that the same policy is used for acting and updating. More on this in the Python section.
* **Temporal Difference (TD) vs Monte Carlo:** Q-Learning uses a **TD approach**, which means that during training, it updates the Q-function after each step. Meanwhile, the Monte Carlo approach is to wait until the end of the episode before making the update to the value function.

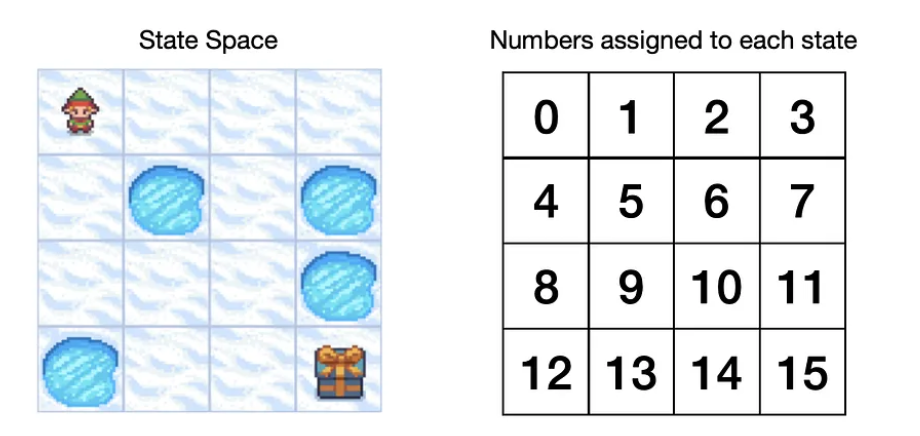
Now let’s take a look at the Q-Learning algorithm to see how it trains the Q-function (updates the Q-table):

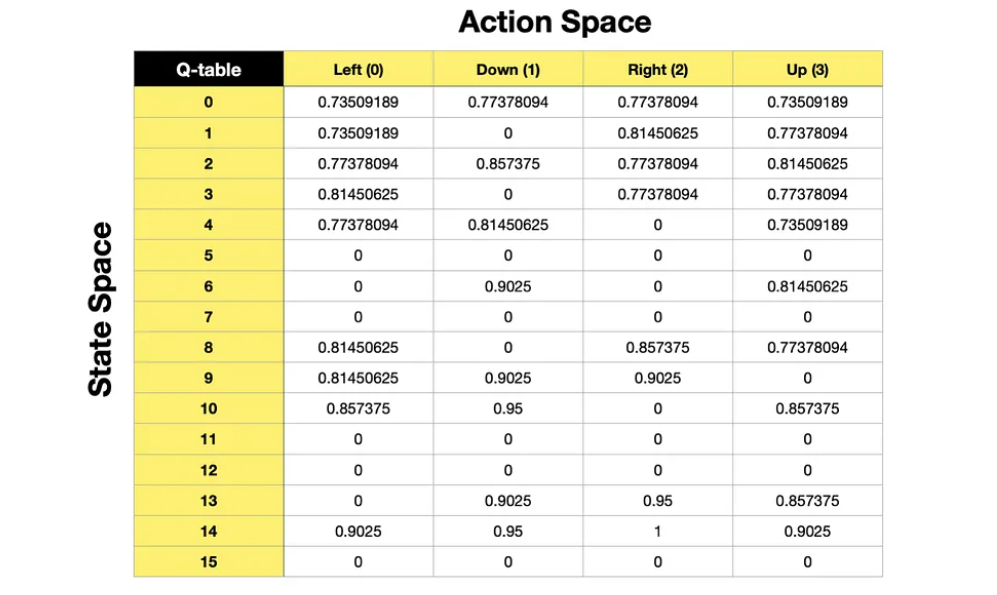
* Q — Q-function
* 𝑆𝑡 — current state(observation)
* 𝐴𝑡 — current action
* 𝑅𝑡+1 — reward received following current action
* 𝑆𝑡+1 — next state(observation)
* 𝛼 (alpha) — learning rate parameter
* 𝛾 (gamma) — discount factor parameter
* 𝑚𝑎𝑥𝑎𝑄(𝑆𝑡+1,𝑎) — maximum value for the next state(observation) across the possible action space



* To modify the current value, we take the **reward** following the action taken by the agent 𝑅𝑡+1 add the **maximum value we can get from the next state** 𝛾𝑚𝑎𝑥𝑎𝑄(𝑆𝑡+1,𝑎) **discounted by gamma**, and subtract the **current value** Q(𝑆𝑡,𝐴𝑡).
* So, the terms in the square brackets can lead to either a positive, zero or negative value. Hence, the new Q value for that state-action pair will either increase, stay the same or reduce. Note that we also apply a learning rate to control the “size” of each update.

Since Q-Learning uses **Temporal Difference (TD) approach**, the algorithm will keep updating Q-table after each step until we reach a position where no more updates can be made, i.e., it has converged to an optimal solution.





**Other important aspects:**

## **Reward (and Discount Factor)**

Perhaps the most crucial piece of the puzzle is the **reward**. We train the agent to take the “best” actions by giving a positive, neutral or negative reward based on whether the action has taken the agent closer to achieving a specific goal.

Assume we are playing a game where the objective is to catch a squirrel. The agent would start by randomly exploring its environment, and we would reward the agent (+1 point) if it got closer to a squirrel and “punish” (-1 point) if it moved further away from the squirrel. Finally, we would give a significant reward (+1000) when the agent achieved its goal, i.e., caught the squirrel.

Just by taking random actions and receiving corresponding rewards, the agent can learn what it needs to do to **maximize the cumulative reward** and **achieve the goal**.

Note that the reward values in this example are just for illustration purposes. We often choose to **create our own reward function** when we do Reinforcement Learning. Usually, the “quality” of your reward function will be a significant driver of the success of your model.

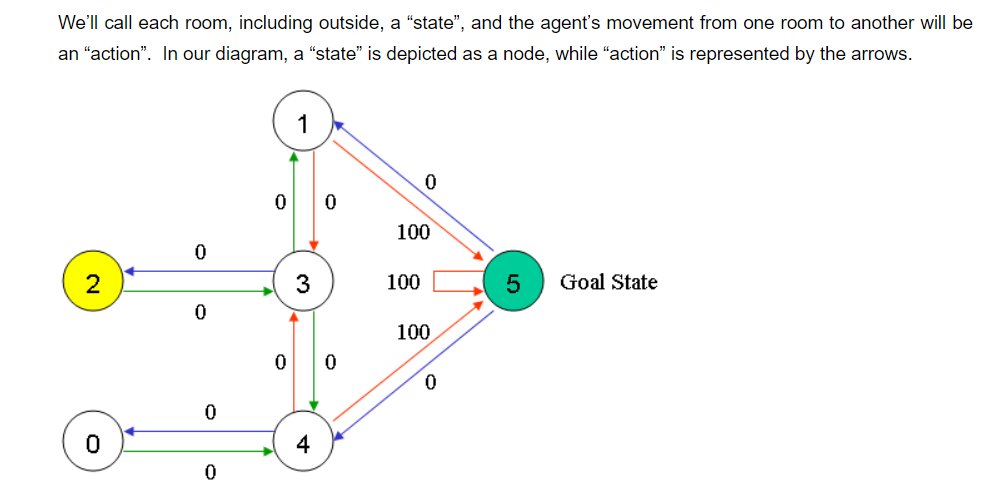
Another critical aspect of rewards is the **discount factor (gamma)**. It can range between 0 and 1, but we would typically choose a value between 0.95 and 0.99. The purpose of a discount factor is to give us control over the preference for short-term vs long-term rewards. High Gamma values will prioritize long term rewards and low gamma values will prioritize short term rewards.

For example, a move that captures the opponent’s piece would be rewarded in the chess game. However, we wouldn’t want the agent to prioritize that move if it put us in a losing situation over the longer term. Hence, it is essential to balance short-term and longer-term rewards. Because at the end we want to **maximize the cumulative reward, not individual rewards,** and **achieve the goal**.

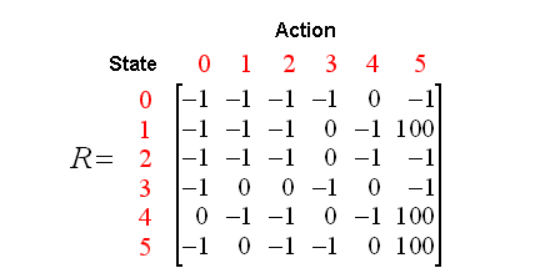
In the context of Reinforcement Learning, we want to encourage the agent to spend part of its time exploring. Otherwise, the agent may end up constantly repeating the same move, not realising that there are much better moves to make. We do this via an **additional parameter (epsilon)**. We specify in what percentage of situations the agent should take a **random action (i.e. explore)**.

* **Value-based methods**— we train the agent **indirectly** by teaching it to identify which states (or state-action pairs) are more valuable so that it can be guided by value maximization. It is described by a **value function** where the value of a state is the expected discounted return the agent can get if it starts in that state.

**Regardless of which method we use to train our agent, finding the optimal policy function or the optimal value function equates to discovering the optimal policy(𝜋).**



We can put the state diagram and the instant reward values into the following reward table, “matrix R”:



Each episode consists of the agent moving from the initial state to the goal state.  Each time the agent arrives at the goal state, the program goes to the next episode.

The algorithm above is used by the agent to learn from experience.  Each episode is equivalent to one training session.  In each training session, the agent explores the environment (represented by matrix R ) and receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the ‘brain’ of our agent, represented by matrix Q.  More training results in a more optimized matrix Q.  In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

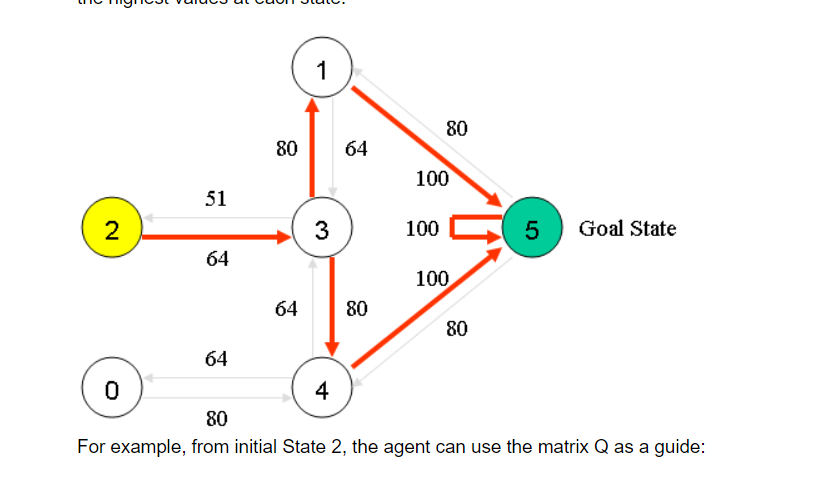
The Gamma parameter has a range of 0 to 1 (0 <= Gamma > 1).  If Gamma is closer to zero, the agent will tend to consider only immediate rewards.  If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to the goal state.  The algorithm finds the actions with the highest reward values recorded in matrix Q for the current state:

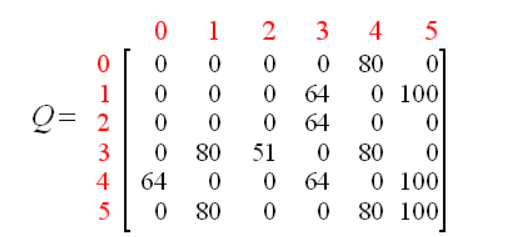
Algorithm to utilize the Q matrix:

1. Set current state = initial state.
2. From the current state, find the action with the highest Q value.
3. Set current state = next state.
4. Repeat Steps 2 and 3 until current state = goal state.

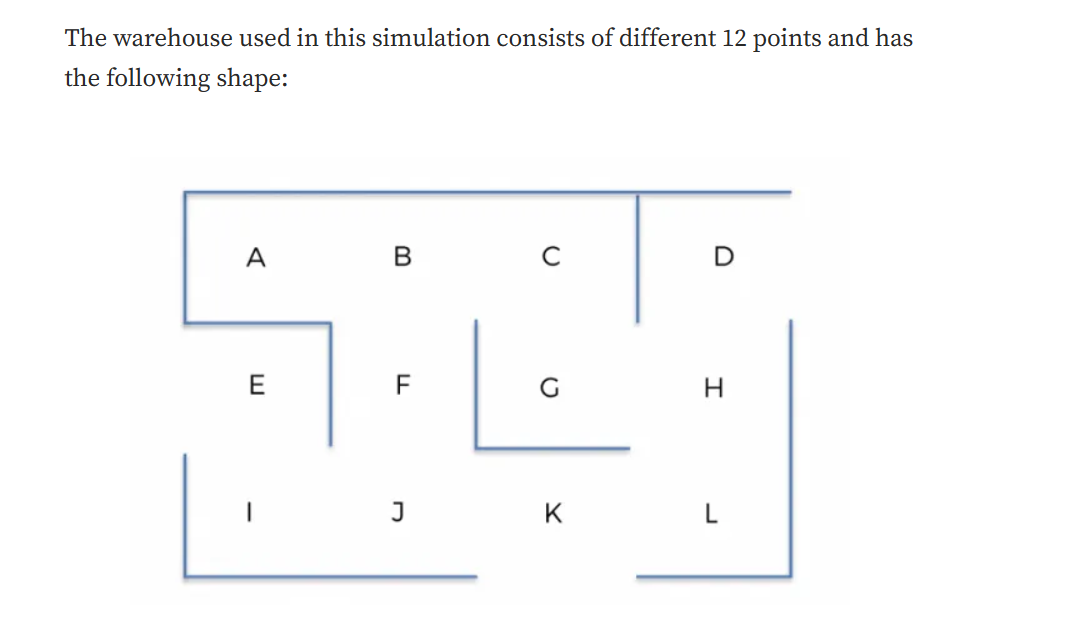
The algorithm above will return the sequence of states from the initial state to the goal state.



Above are the optimal paths with the Q-values.



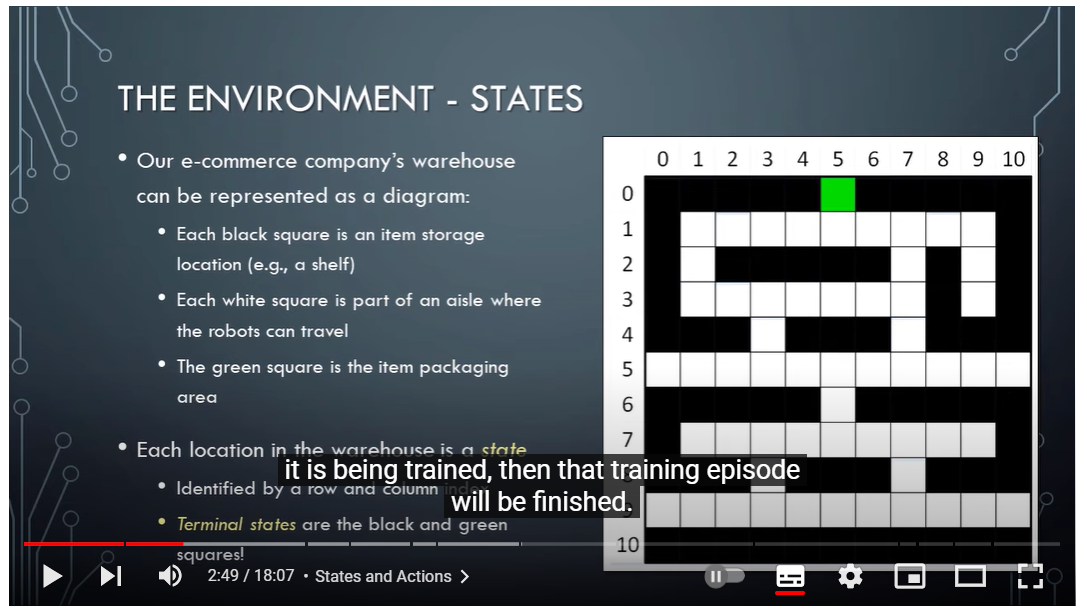
Another example:



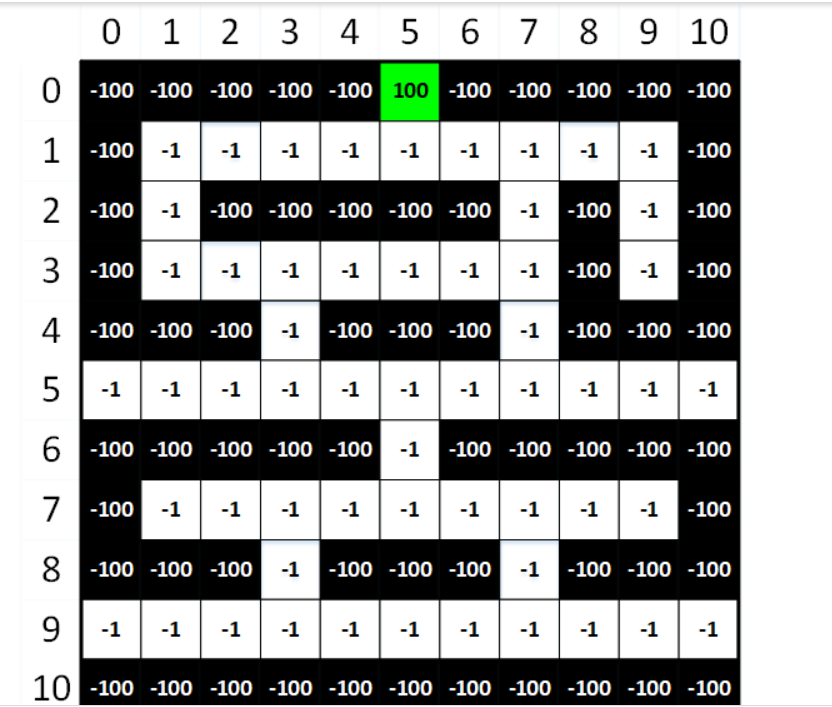
Starting from location According t the warehouse map, the robot can only move to location B. while being in location B gives the possibility to move to location A, C or F.



Another example:



Rewards table:



**To summarize:**

**Q learning is a** **Value-based methods — we train the agent indirectly by teaching it to identify which states (or state-action pairs) are more valuable so that it can be guided by value maximization.** Once we train the agent the Q-Table values is being updated through each step, until it reaches the maximum number of episodes, an episode as a fix number of steps, or the Q-table reaches it most optimal policy values, rewards that guide the best actions to take by the agent in each state, **the algorithm converges.**

Important parameters to tune are epsilon, gamma, learning rate.

**Epsilon**: enables us to set how much time the agent should spend exploring the environment vs exploiting its existing knowledge about the environment.

**Gamma (Discount factor):** Typically choose a value between 0.95 and 0.99. The purpose of a discount factor is to give us control over the preference for short-term vs long-term rewards. High Gamma values will prioritize long term rewards and low gamma values will prioritize short term rewards.

**Alpha(𝛼)**: Learning rate, which influences the learning speed and convergence towards the optimal policy.