# Q Learning: All you need to know about Reinforcement Learning

- What is Reinforcement Learning?
- Q-Learning Process
- Bellman Equation
- Markov Decision Process
- Q-Learning Demo: NumPy

## What is Reinforcement Learning?

Let’s have a look at our day to day life. We perform numerous tasks in the environment and some of those tasks bring us rewards while some do not. We keep looking for different paths and try to find out which path will lead to rewards and based on our action we improve our strategies on achieving goals. This my friends are one of the simplest analogy of Reinforcement Learning.

Key areas of Interest :

- Environment
- Action
- Reward
- State

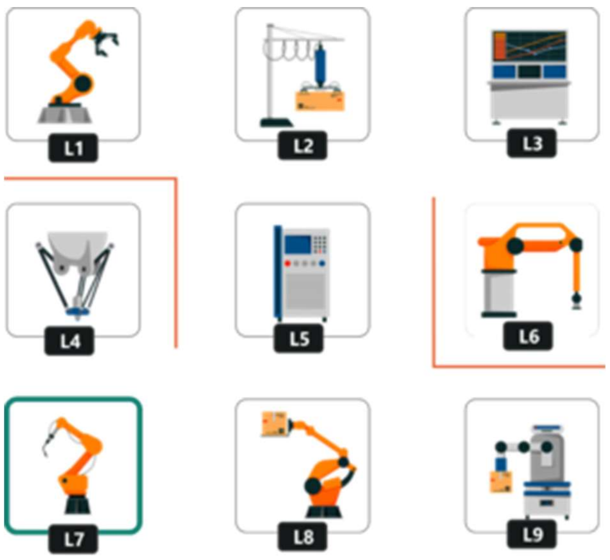
Reinforcement Learning is the branch of machine learning that permits systems to learn from the outcomes of their own decisions. It solves a particular kind of problem where decision making is sequential, and the goal is long-term.

## Q-Learning Process

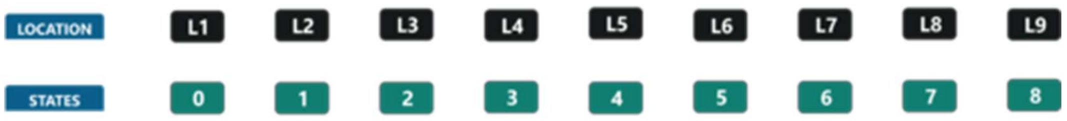
Let's understand what is Q learning with our problem statement here. It will help us to define the main components of a reinforcement learning solution i.e. agents, environment, actions, rewards, and states.

**Automobile Factory Analogy:**

We are at an Automobile Factory filled with robots. These robots help the Factory workers by conveying the necessary parts required to assemble a car. These different parts are located at different locations within the factory in 9 stations. The parts include Chassis, Wheels, Dashboard, Engine and so on. Factory Master has prioritized the location where chassis is being installed as the highest priority. Let's have a look at the setup here:



**States:**

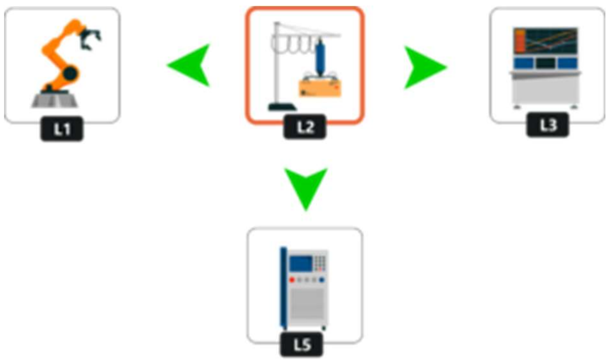


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| LOCATION | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 |
| STATES | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

The Location at which a robot is present at a particular instance is called its state. Since it is easy to code it rather than remembering it by names. Let's map the location to numbers.

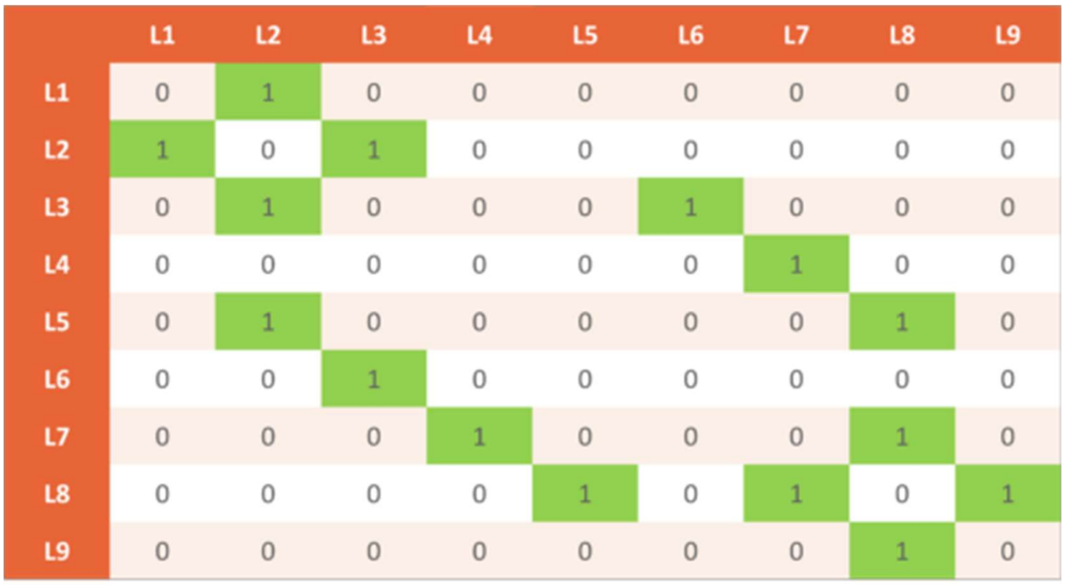
**Actions:**

Actions are nothing but the moves made by the robots to any location. Consider, a robot is at the L2 location and the direct locations to which it can move are L5, L1 and L3. Let's understand this better if we visualize this:



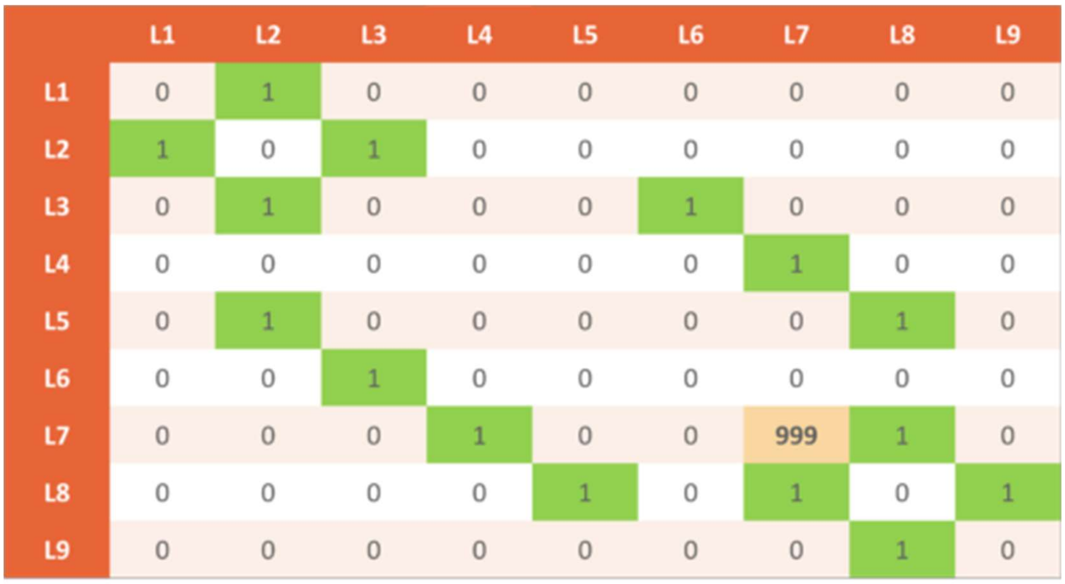
**Rewards:**

A Reward will be given to the robot for going directly from one state to another. For example, you can reach L5 directly from L2 and vice versa. So, a reward of 1 will be provided in either case. Let's have a look at the reward table:



|  | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 |
|---|---|---|---|---|---|---|---|---|---|
| L1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| L2 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| L3 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| L4 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| L5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| L6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| L7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| L8 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| L9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

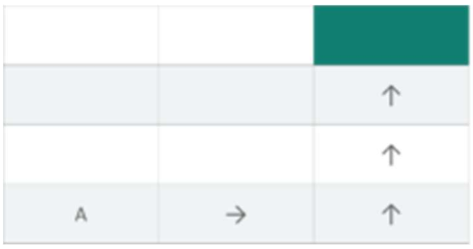
Remember when the Factory Master Prioritized the chassis location. It was L7, so we are going to incorporate this fact into our rewards table. So, we'll assign a very large number(999 in our case) in the (L7, L7) location.



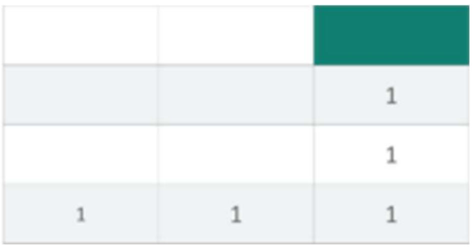
|  | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 |
|---|---|---|---|---|---|---|---|---|---|
| L1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| L2 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| L3 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| L4 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| L5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| L6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| L7 | 0 | 0 | 0 | 1 | 0 | 0 | 999 | 1 | 0 |
| L8 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| L9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

## Bellman Equation

Now suppose a robot needs to go from point A to B. It will choose the path which will yield a positive reward. For that suppose we provide a reward in terms of footprint for it to follow.



| | | |
|---|---|---|
| | | 🟩 |
| | | ↑ |
| | | ↑ |
| A | → | ↑ |



| | | |
|---|---|---|
| | | 🟩 |
| | | 1 |
| | | 1 |
| 1 | 1 | 1 |

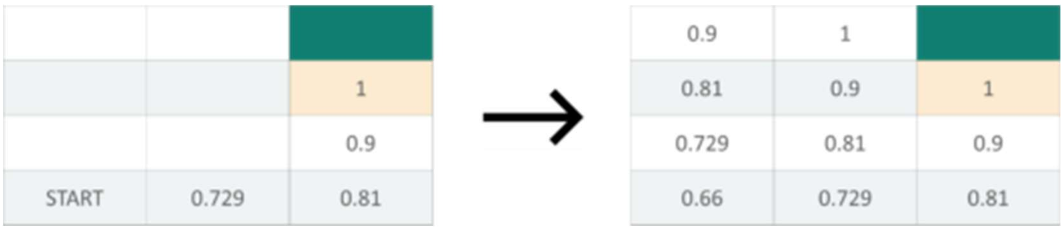
But what if the robot starts from somewhere in between where it can see two or more paths. The robot can thus not take a decision and this primarily happens because it doesn’t possess a **memory**. This is where the Bellman Equation comes into the picture.

$$V(s) = \max(R(s,a) + \gamma V(s'))$$

Where:

- s = a particular state
- a = action
- s' = state to which the robot goes from s
- $\gamma$ = discount factor
- R(s, a) = a reward function which takes a state (s) and action (a) and outputs a reward value
- V(s) = value of being in a particular state

Now the block below the destination will have a reward of 1, which is the highest reward, But what about the other block? Well, this is where the discount factor comes in. Let’s assume a discount factor of 0.9 and fill all the blocks one by one.

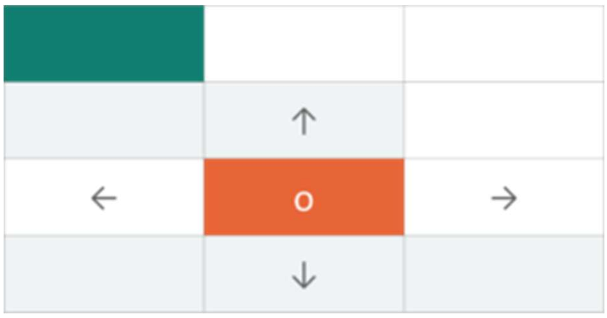


| | | |
|---|---|---|
| | | 🟩 |
| | | 1 |
| | | 0.9 |
| START | 0.729 | 0.81 |

| | | |
|---|---|---|
| 0.9 | 1 | 🟩 |
| 0.81 | 0.9 | 1 |
| 0.729 | 0.81 | 0.9 |
| 0.66 | 0.729 | 0.81 |

## Markov Decision Process

Imagine a robot is on the orange block and needs to reach the destination. But even if there is a slight dysfunction the robot will be confused about which path to take rather than going up.



So we need to modify the decision-making process. It has to **Partly Random** and **Partly under the robot's control**. Partly Random because we don't know when the robot will dysfunction and Partly under control because it's still the robot's decision. And this forms the base for the Markov Decision Process.

***A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.***

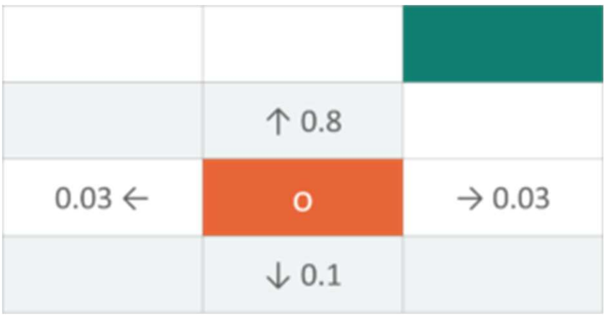
So we are going to use our original Bellman equation and make changes in it. What we don't know is the next state ie. ***s'.*** What we know is all the possibilities of a turn and let's change the Equation.

$$V(s) = \max(R(s, a) + \gamma V(s'))$$

$$V(s) = \max(R(s, a) + \gamma \Sigma_{s'} P(s, a, s') V(s'))$$

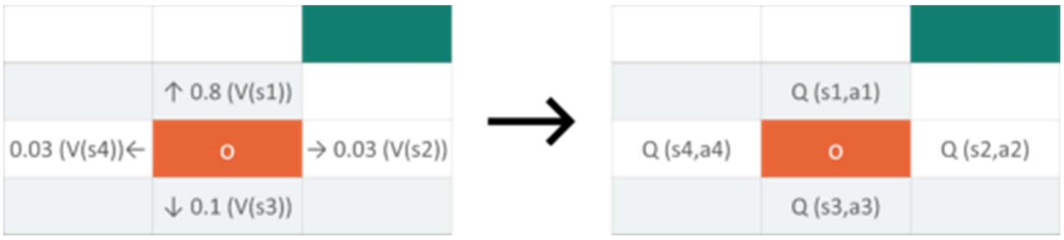
***P(s,a,s') :*** Probability of moving from state ***s*** to ***s'*** with action ***a***

$\Sigma_{s'} P(s,a,s') V(s')$ **:** Randomness Expectations of Robot



$$V(s) = max(R(s, a) + \gamma((0.8V(room_{up})) + (0.1V(room_{down}) + ....))$$

Now, Let’s transition into Q Learning. Q-Learning poses an idea of assessing the quality of an action that is taken to move to a state rather than determining the possible value of the state it is being moved to.



This is what we get if we incorporate the idea of assessing the quality of actions for moving to a certain state s'. From the updated Bellman Equation if we remove them ***max*** component, we are assuming only one footprint for possible action which is nothing but the **Quality** of the action.

$$Q(s, a) = (R(s, a) + \gamma \Sigma_{s'} P(s, a, s') V(s'))$$

In this equation that Quantifies the Quality of action, we can assume that V(s) is the maximum of all possible values of Q(s, a). So let's replace v(s') with a function of Q().

$$Q(s, a) = (R(s, a) + \gamma \Sigma_{s'} P(s, a, s') \max Q(s', a'))$$

We are just one step close to our final Equation of Q Learning. We are going to introduce a **Temporal Difference** to calculate the Q-values with respect to the changes in the environment over time. But how do we observe the change in Q?

$$TD(s, a) = (R(s, a) + \gamma \Sigma_{s'} P(s, a, s') \max Q(s', a')) - Q(s, a)$$

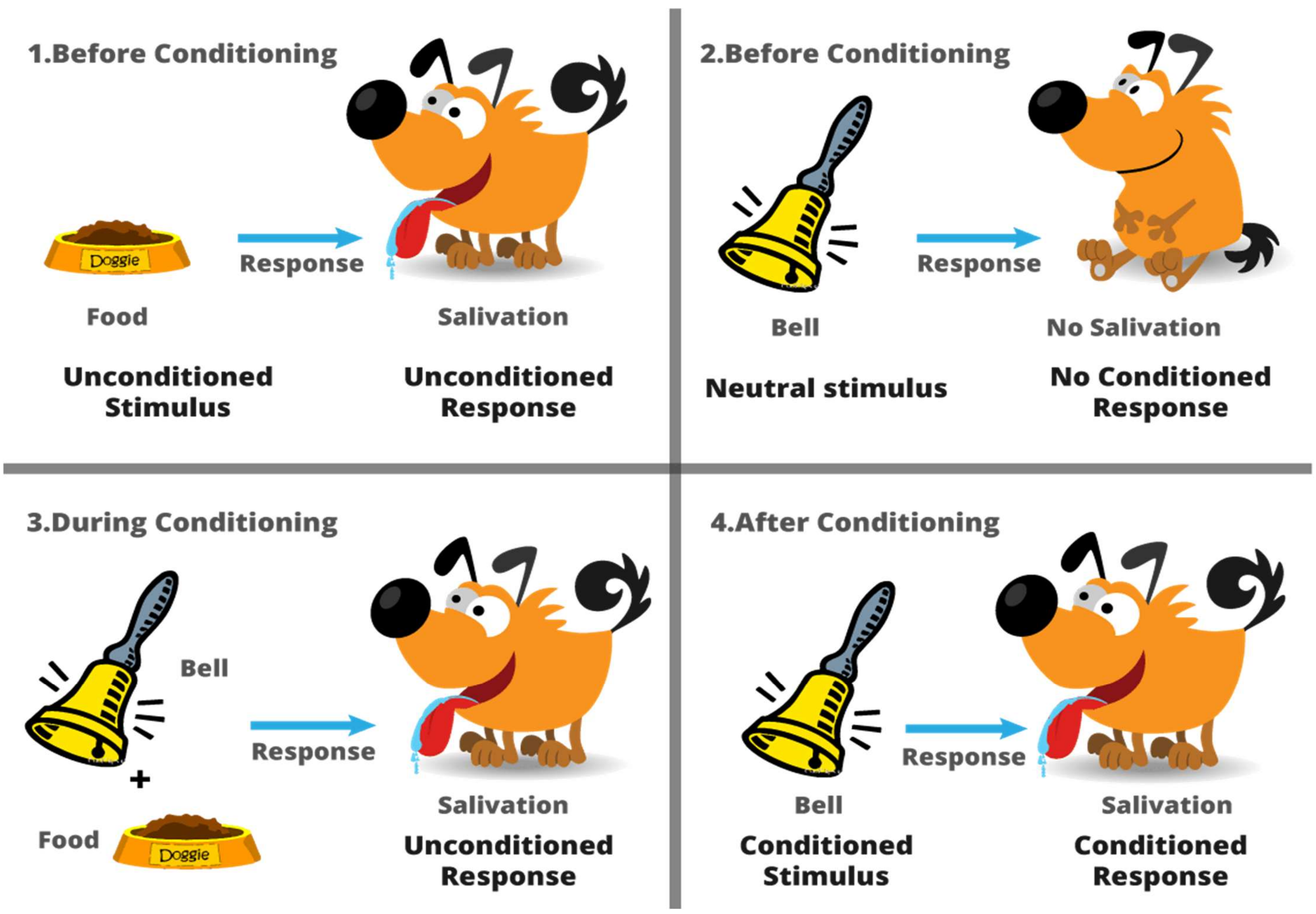
We recalculate the new Q(s, a) with the same formula and subtract the previously known Q(s, a) from it. So, the above equation becomes:

$$Q_t(s, a) = Q_{t-1}(s, a) + \alpha\, TD_t(s, a)$$

$Q_t(s, a)$ = *Current Q-value*

$Q_{t-1}(s, a)$ = *Previous Q-value*

$$Q_t(s, a) = Q_{t-1}(s, a) + \alpha\, (R(s, a) + \gamma \max Q(s', a') - Q_{t-1}(s, a))$$


1.Before Conditioning
Food
Response
Salivation
Unconditioned Stimulus
Unconditioned Response
2.Before Conditioning
Bell
Response
No Salivation
Neutral stimulus
No Conditioned Response
3.During Conditioning
Bell
+
Food
Response
Salivation
Unconditioned Response
4.After Conditioning
Bell
Response
Salivation
Conditioned Stimulus
Conditioned Response

# Reinforcement Learning, Part 6: TD(λ) & Q-learning

- What is **TD(λ)** and how is it used?
- How does **Q-learning**’s classic off-policy method work?
- What does a **Python** realization of Q-learning look like?

## Temporal-Difference Learning: TD(λ)

In my last post, we mentioned that if we replace $Gt$ in the MC-updated formula with an estimated return $Rt+1+V(St+1)$, we can get $TD(0)$:

$$\boxed{V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]}$$

Where:

- $Rt+1+V(St+1)$ is called **TD target value**
- $Rt+1+V(St+1)$- $V(St$） is called **TD error**

Now, we replace the **TD target value** with **Gt(),** we can have TD(λ). **Gt()**is generated as below:

$$
\begin{aligned}
G_t^{(1)} &= R_{t+1} + \gamma V(S_{t+1}) \\
G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\
G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n V(S_{t+n}) \\
G_t^{(\lambda)} &= (1-\lambda) G_t^{(1)} + (1-\lambda)\lambda\, G_t^{(2)} + \dots + (1-\lambda)\lambda^{n-1} G_t^{(n)} \\
&\approx [(1-\lambda) + (1-\lambda)\lambda + \dots + (1-\lambda)\lambda^{n-1}] V(S_t) = V(S_t)
\end{aligned}
$$

So the *TD(λ)* formula is:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(\lambda)} - V(S_t)]$$

In which:

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

As discussed, Q-learning is a combination of Monte Carlo (MC) and Temporal Difference (TD) learning. With MC and *TD(0)* covered in Part 5 and *TD(λ)* now under our belts, we're finally ready to pull out the big guns!

## Q-Learning

Q-Value formula:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, a)]$$

From the above, we can see that Q-learning is directly derived from *TD(0)*. For each updated step, Q-learning adopts a greedy method: maxaQ (St+1, a).

This is the main difference between Q-learning and another TD-based method called Sarsa, which I won’t explain in this series. But as an RL learner, you should know that Q-learning is not the only method based on TD.

## An Example of How Q-Learning Works

Let’s try to understand this better with an example:

You’re having dinner with friends at an Italian restaurant and, because you’ve been here once or twice before, they want you to order. From experience, you know that the Margherita pizza and pasta Bolognese are delicious. So if you have to order ten dishes, experience might tell you to order five of each. But what about everything else on the menu?

In this scenario, you are like our “agent”, tasked with finding the best combination of ten dishes. Imagine this becomes a weekly dinner; you’d probably start bringing a notebook to record information about each dish. In Q-learning, the agent collects **Q-values** in a **Q-table**. For the restaurant menu, you could think of these values as a score for each dish.

Now let’s say your party is back at the restaurant for the third time. You’ve got a bit of information in your notebook now but you certainly haven’t explored the whole menu yet. How do you decide how many dishes to order from your notes — which you know are good, and how many new ones to try?

This is where **ε-greedy** comes into play.

## The ε-greedy Exploration Policy

In the above example, what happened in the restaurant is like our MDP (Markov Decision Process) and you, as our "agent" can only succeed in finding the best combination of dishes for your party if you explore it thoroughly enough.

So it is with Q-Learning: it can work only if the agent explores the MDP thoroughly enough. Of course, this would take an extremely long time. Can you imagine how many times you'd have to go back to the restaurant to try every dish on the menu in every combination?

This is why Q-learning uses the **ε-greedy policy**, which is ε degree "greedy" for the highest Q values and *1* — ε degree "greedy" for random exploration.

In the initial stages of training an agent, a random exploration environment (i.e. trying new things on the menu) is often better than a fixed behavior mode (i.e. ordering what you already know is good) because this is when the agent accumulates experience and fills up the Q-table.

Thus, it's common to start with a high value for ε, such as 1.0. This means the agent will spend 100% of its time exploring (e.g. using a random policy) instead of referring to the Q-table.

From there, the value of ε can be gradually decreased, making the agent more greedy for Q-values. For example, if we drop ε to 0.9, it means the agent will spend 90% of its time choosing the best strategy based on Q-table, and 10% of its time exploring the unknown.

The advantage of the *ε*-greedy policy, compared to a completely greedy one, is that it always keeps testing unknown regions of the MDP. Even when the target policy seems optimal, the algorithm never stops exploring: it just keeps getting better and better.

There are various functions for exploration, and many defined exploration policies can be found online. Do note that not all exploration policies are expected to work for both discrete and continuous action spaces.