Reinforcement Learning

* Reinforcement learning focuses on teaching agents through trial and error methods.
* Reinforcement learning is the fourth major learning method in [machine learning](https://databasecamp.de/en/machine-learning), along with [supervised](https://databasecamp.de/en/ml/supervised-learning-models), unsupervised, and semi-supervised learning. The main difference is that the model does not need any [data](https://databasecamp.de/en/data) to train.
* It learns structures by being rewarded for desired behaviors and punished for bad ones.
* Also known to be scienc and framework of learning to make decisions from interaction.

## **What problems are RL used to solve?**

Rather than the typical ML problems such as Classification, Regression, Clustering and so on, RL is most commonly used to solve a different class of real-world problems, such as a Control task or Decision task, where you operate a system that interacts with the real world.

* eg. A robot or drone that has to learn the task of picking a device from one box and putting it in a container

It is useful for a variety of applications like:

* Operating a drone or autonomous vehicle
* Manipulating a robot to navigate the environment and perform various tasks
* Managing an investment portfolio and taking trading decisions
* Playing games such as Go, Chess, video games

## **Reinforcement Learning happens through trial and error**

With RL the learning happens from experience by trial and error, similar to a human eg. A baby can touch fire or milk and then learns from negative or positive reinforcement.

* The baby takes some action
* Receives feedback from the environment about the result of that action
* Repeats this process till it learns which actions produce favorable results and which actions produce unfavorable results.



A baby learns from positive and negative reinforcement

# **To use RL, structure your problem as a Markov Decision Process**

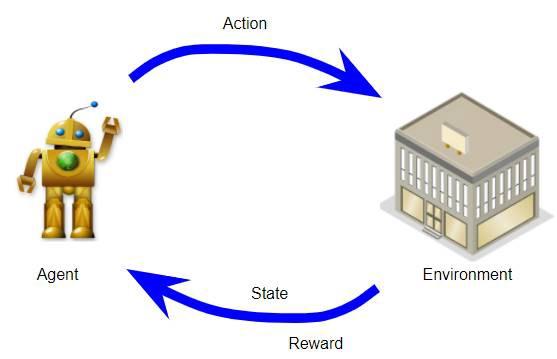
Let’s say you want to train a robot. How would you use RL to solve a problem like this?

To apply RL, the first step is to structure the problem as something called a Markov Decision Process (MDP).

So let’s try to understand what an MDP is. An MDP has five components that work together in a well-defined way.

**Agent:** it’s the system that you operate eg. the robot. This is the model that you want to build and train using RL.

**Environment:** the real-world environment with which the agent interacts as part of its operation. eg. The terrain that the robot has to navigate, its surroundings, factors such as wind, friction, lighting, temperature and so on.



An MDP has an Agent, Environment, States, Actions and Rewards

**State:** this represents the current ‘state of the world’ at any point. eg. it could capture the position of the robot relative to its terrain, the position of objects around it, and perhaps the direction and speed of the wind.

There could be a finite or infinite set of states.

**Action:** these are the actions that the agent takes to interact with the environment. eg. The robot can turn right, left, move forward, go backward, bend, raise its hand and so on.

There could be a finite or infinite set of possible actions.

**Reward:** is the positive or negative reinforcement that the agent receives from the environment as a result of its actions. It is a way to evaluate the ‘good-ness’ or ‘bad-ness’ of a particular action.

eg. If moving in a particular direction causes the robot to run into a wall, that would have a negative reward. On the other hand, if turning left causes the robot to locate the object it needs to pick up, it would get a positive reward.

## **What should you keep in mind when defining your MDP?**

**Agent and Environment**: Obviously, the first step is to decide the role and scope of your agent and the environment for the problem you are trying to solve.

**State**: Next, you have to define what data points the state contains and how they are represented.

The important thing is that it captures everything that is needed for your problem to represent the current world situation so that the agent can reason about the future without requiring information about the past or any additional knowledge.

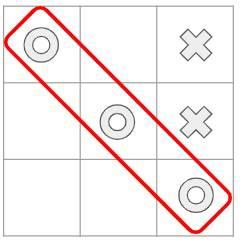
In other words, the definition of the state should be self-contained. So, for instance, if you need to know something about how you arrived at this state, that history should be encapsulated within your state definition itself.

**Actions**: What is the set of the potential actions your agent can take?

**Reward**: This is how the agent learns from experience. So this is something that you need to give a fair amount of thought to because it is critical that you define the rewards in a way that truly reflects the behavior that you want your agent to learn.

# **How does an MDP work?**

Now that we’ve seen what an MDP is, we’ll go into how it works. Let’s use the game of Tic-Tac-Toe (aka Noughts and Crosses) as a simple example. Two players play this game by placing their tokens on a 3x3 grid. One player places Noughts (the donut-shape) while the other places Crosses. The objective is to win the game by placing three of your tokens in a line.



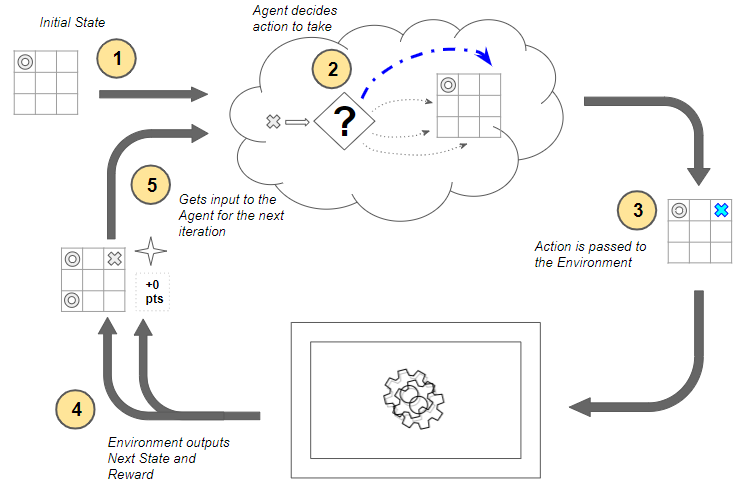
## **You can define your MDP as follows:**

* The agent plays against the environment, so the environment acts as its opponent.
* The state at any point, is the current position of all the tokens, both the agent’s and the environment’s, on the board.
* There are 9 possible actions where the agent can place its token at each of the 9 available squares in the grid.
* If the agent wins it gets a positive reward of +10 points and if it loses it gets a negative reward of -10 points. Each intermediate move gives a neutral reward of 0 points.

## **Now let’s go through the operation of the MDP as it plays the game.**

The agent interacts with its environment over a sequence of time-steps. A set flow of operations occurs in each time-step and that flow then repeats in each time-step.

The sequence starts with an initial state, which becomes the current state. For instance, your opponent, the environment has placed their token in a particular position, and that is the starting state for the game.



How the MDP works

Now, starting with the first time-step, the following steps occur at each time-step:

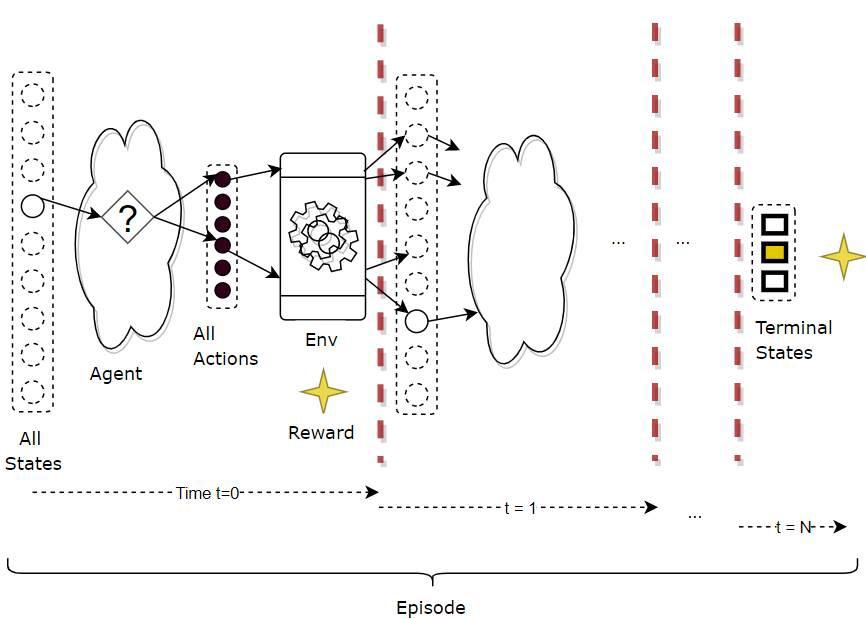
1. The environment’s current state is input to the agent.
2. The agent uses that current state to decide what action it should take. It does not need a memory of the full history of states and actions that came before it. The agent decides to place its token in some position. There are many possible actions to choose from, so how does it decide what action to take? It’s a very important question but we’ll come to that later.
3. That action is passed as input to the environment.
4. The environment uses the current state and the selected action and outputs two things — it transitions the world to the next state, and it provides some reward. For instance, it takes the next move by placing its token in some position and provides us a reward. In this case, since no one has won the game yet, it provides a neutral reward of 0 points. How the environment does this is opaque to the agent, and not in our control.
5. This reward from the environment is then provided as feedback to the agent as a consequence of the previous action.

This completes one time-step and moves us to the next time-step. This next state now becomes the current state which is then provided to the agent as input, and the cycle repeats.

Throughout this process, it is the agent’s goal to maximize the total amount of rewards that it receives from taking actions in given states. It wants to maximize not just the immediate reward, but the cumulative rewards it receives over time. We will return to this topic shortly.

# **An MDP iterates over a sequence of time-steps**

Here is another view of the MDP’s operation which shows the progression of time-steps.



An MDP iterates over a sequence of time steps

In each time-step, three things occur — state, action and reward, which fully describe what happened in that time-step.

## **A Trajectory describes the execution over multiple time-steps**

So the execution of the MDP can be described as a trajectory of occurrences (in terms of state, action, reward) over a sequence of time-steps, as below.

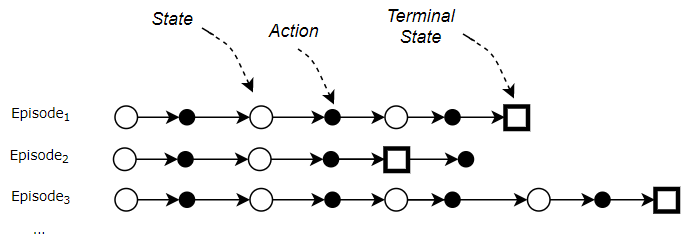
(s3,a3,r4, s4,a4,r5, s5,a5,r6, s6)

## **Episodic Tasks end in a Terminal State**

For RL tasks that have a well-defined end or Terminal state, a complete sequence from the starting state to the end state is called an episode. eg. Each round of a game is an episode.

* So at the end of an episode, you can reset to a starting state (or randomly pick one from a set of starting states) and play another complete episode, and repeat.
* Each episode is independent of the next one.

Therefore an RL system’s operation repeats over multiple episodes. Within each episode, it repeats over multiple time-steps.



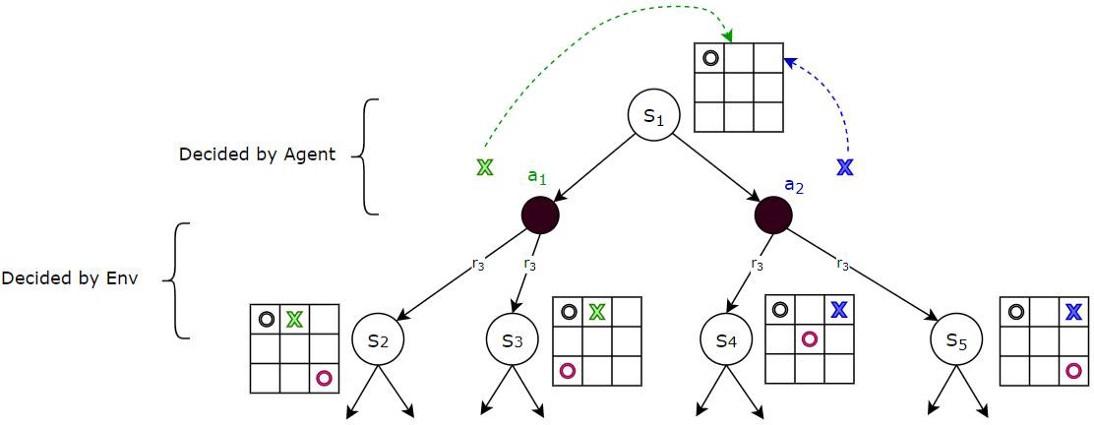
Each Episode ends in a Terminal State

## **Continuing Tasks go on forever**

On the other hand, RL tasks which have no end are known as Continuing Tasks, and can go on forever (or till you stop the system). Eg. A robot that continuously manages manufacturing or warehouse automation.

# **Agent and Environment control the state-action transitions**

As we just saw, the MDP operates by alternating between the agent doing something and then the environment doing something, in each time-step:



Given a state, the Agent decides the action. Given an action (and state), the Environment decides the next state.

* Given the current state, the next action is decided by the agent. In fact, that is the only job of the agent. For instance, from the current state, the agent can choose, say, action *a₁* or *a₂* to place its token.
* Given the current state, and the next action chosen by the agent, the transition to the next state and the reward is controlled by the environment. For instance, if the agent had chosen action *a₁*, the environment could transition to state *S₂* or *S₃* by playing different moves. Another video game example could be that starting from a given state (eg. character is standing on a roof), the same agent action (character jumps) could with some probability, end up in more than one next state (eg. land on a neighboring roof, or fall to the ground), as controlled by the environment.

## **How does the environment transition to the next state?**

Given a current state, and the action picked by the agent, how does the environment figure out the outcome ie. the next state and reward?

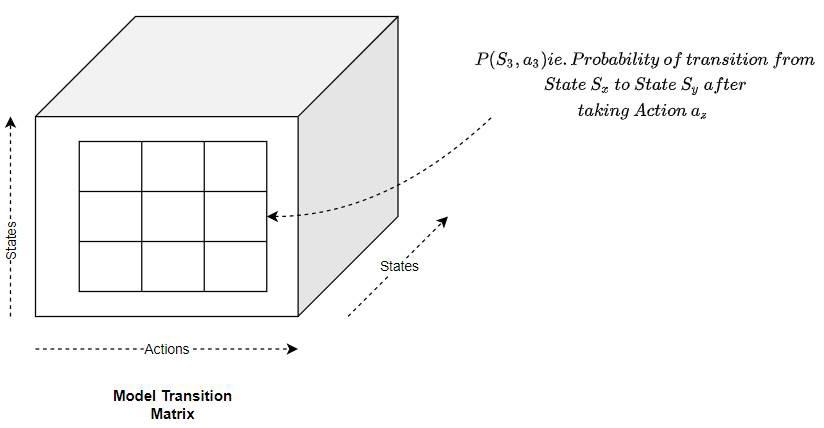
For most realistic RL problems that we will deal with, the answer will usually be that 'it just does’. Most environments have complex internal dynamics that control how they behave when an action is taken from a particular state.

For instance, in a stock-trading RL application, the stock market environment has a range of unseen factors that determine how stock prices move. Or the environment in a drone navigation RL application depends on the laws of physics that control air flows, motion, thermodynamics, visibility and so on in a variety of terrains and micro-weather conditions.

Our focus is on training the agent and we can usually treat the environment as an external black box.

Note that this external black box might be a simulator for the environment. In many cases, it might not be practical to build a simulator and we would interact with the real environment directly.

However, just for completeness, let me briefly mention that if we did build such an environment model, an MDP would represent it as a large transition probability matrix or function.



This matrix maps a given state and action pair to:

* The next state, with some probability, since we could end up in different states each with some probability. This is known as the Transition Probability.
* The reward.

## **How does the Agent pick the action?**

On the other hand, we are very interested in how the agent decides what action to pick in a given state. That is, in fact, precisely the RL problem that we want to solve.

For that it uses three concepts, which we will explore next:

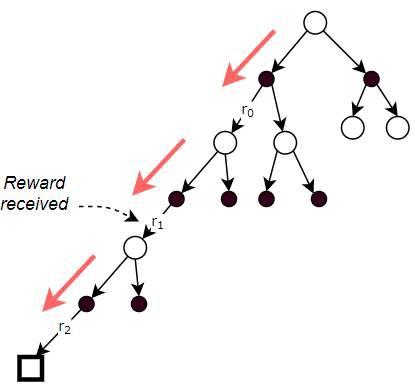
* Return
* Policy
* Value

# **The Return is the total reward over all time-steps**

As the agent executes time-steps, it accumulates reward at each time-step.

However, rather than any individual reward, what we really care about is the cumulative rewards.

We call this the Return. It is the total reward that the agent accumulates over the duration of a task.



The Return is the total of the rewards received at each time-step

## **Return is computed using Discounted Rewards**

When we calculate Return, rather than simply adding up all the rewards, we apply a discount factor γ to weight later rewards over time. These are known as Discounted Rewards.

***Return = r₀ +γ r₁ + γ² r₂***

and, more generally:

***Return = r₀ +γ r₁ + γ² r₂ + ….+ γ*ⁿ *r*ₙ**

This way, cumulative rewards do not grow infinitely as the number of time-steps becomes very large (like for continuing tasks, or for very long episodes).

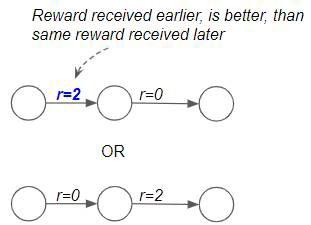
It also encourages the agent to care more about the immediate reward compared to later rewards since later rewards will be more heavily discounted.

The ultimate goal of the agent is to get the maximum Return, not just over one episode, but over many, many episodes.

Based on this discount, we can see that there are two factors the agent considers when evaluating rewards.

## **Immediate Reward is more valuable than Later Reward**

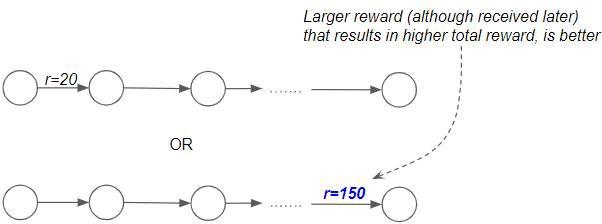
The first point is that if the agent had to choose between getting some amount of reward now versus later, the immediate reward is more valuable. Since the discount factor, γ, is less than 1, we will discount later rewards more than immediate rewards.



Immediate Reward is more valuable than Later Reward

## **Rewards that give us the highest Total Returns are better**

The second point is that if the agent had to choose between getting some reward now versus getting a much bigger reward later, the bigger reward is most likely preferable. This is because we want the agent to look at total Returns rather than individual rewards. eg. In a game of chess, the agent has to pick the better of two paths. In the first, it can kill off a few pieces early on by playing aggressively. That gives it some immediate reward. However in the long run that puts it in a disadvantaged position, and it loses the game. Hence it gets a large negative reward at the end. Alternately it can play a different set of moves which yields lower rewards at first but where it ultimately wins the game. And thus gets a large positive reward. Clearly, the second approach is better since it gives a higher total Return as opposed to a bigger immediate reward.



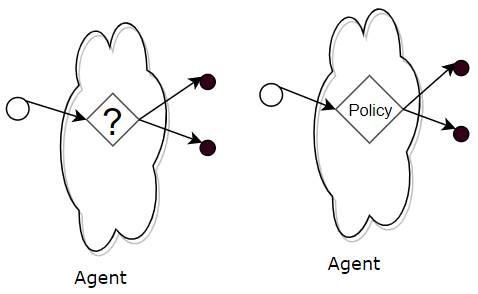
We want to get a higher Total Reward

# **Policy is the strategy followed to pick an action**

The second concept for us to cover is Policy. Earlier we had deferred one very important question, which was, how the agent decides which action to pick in a given state. There can be many different strategies that an agent might use:

* Eg. Always pick the next action at random
* Eg. Always pick the next state that gives the highest known reward
* Eg. Take chances and explore new states in the hope of finding a better path.
* Eg. Always play it safe and avoid the chance of a negative reward.

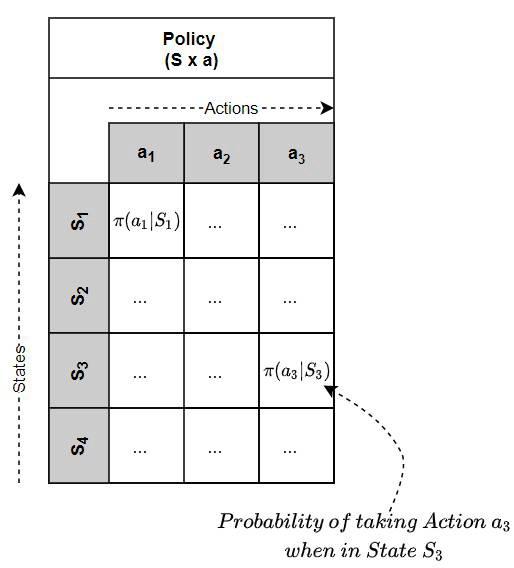
Any strategy that the agent follows to decide which action to pick in a given state, is called a Policy. Although that sounds abstract, a Policy is simply something that maps a given state to an action to be taken.



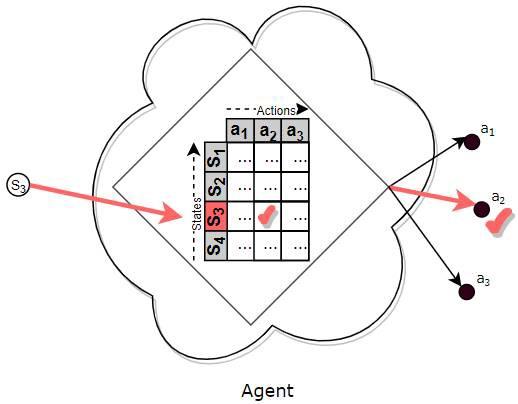
The Policy tells the Agent which action to pick from any state

## **Policy is like a (huge) Lookup Table**

You could think of a Policy as a (huge) Lookup Table which maps a state to an action.



So given the current state, the agent looks up that state in the table to find the action that it should pick.



The Policy is like a (huge) Lookup Table

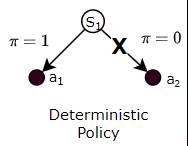
In practice, for real-world problems, there are so many states and so many actions, that a function is used, not a Lookup Table, that maps a state to an action.

However, the intuition is the same — think of a function as a ‘huge lookup table’.

## **Deterministic and Stochastic Policies**

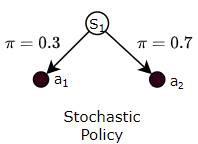
Policies can be either Deterministic or Stochastic.

A Deterministic Policy is a Policy where the agent always chooses the same fixed action when it reaches a particular state.



Alternately, a Stochastic Policy is a Policy where the agent varies the actions it chooses for a state, based on some probability for each action.

It might do this while playing a game, for instance, so that it doesn’t become completely predictable. Eg while playing Rock Paper Scissors, if it always played the same move, opponents can figure this out and easily defeat it.



## **How does the agent get a Policy?**

We’ve been talking about the Policy as though the agent already had one readily available for it to use. But that is not really the case.

Much like a human baby, the agent doesn’t really have a useful policy when it starts out and has no idea what action it should take from any given state. Then, by using the Reinforcement Learning algorithm, it slowly learns a helpful policy that it can use.

## **There are so many possible Policies, which one should the Agent use?**

The action that the agent takes from a given state determines the reward it obtains, and therefore over time, the eventual total Return. Hence the goal of the agent is to pick the action that maximizes its Return.

Put another way, the agent’s goal is to follow a Policy (which is how it picks its actions) that maximizes its Return.

So, out of all the Policies the agent could follow, it wants to pick the best one ie. the one which gives it the highest Return.

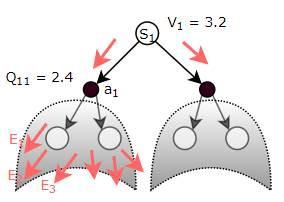
In order to do that, the agent needs to compare two Policies to decide which is better. For that, we need to understand the notion of Value.

# **The Value tells you the expected Return by following some Policy**

Let’s say the agent is in a particular state. Also, let’s say that the agent has somehow been given a policy, π. Now, if it starts from that state, and always picks actions based on that policy, what is the Return it could expect to get?

This is the same as saying, if the agent starts from that state, and always picks actions based on that policy, what would its average Return be over many, many episodes?

This average long-term Return, or expected Return, is known as the Value of that particular state, under policy π.



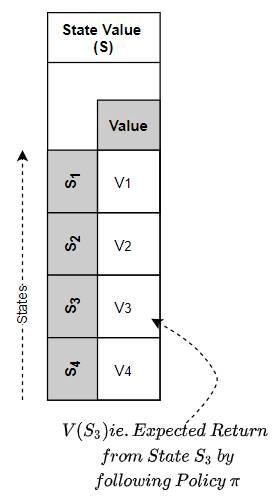
The State Value (V) or the State-Action Value (Q) is the expected Return obtained from a particular state or state-action respectively, by following the given Policy over many episodes

Alternately, the agent could start from a state-action pair ie. it has already taken a particular action from a particular state. If going forward from that state-action, it always picks actions based on the given policy π, what is the Return it could expect to get?

As discussed earlier for the Policy Table, we can think of Value as a (huge) Lookup Table which maps a State, or a State-Action pair, to a Value.

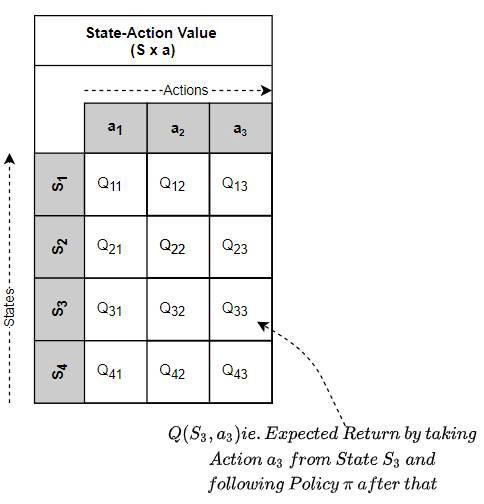
Thus we have two types of Value:

* State Value — the expected Return from a given state, by executing actions based on a given policy π from that state onward. In other words, the State Value function maps a State to its Value.



The State Value Function maps a State to its Value

* State-Action Value (aka Q-Value) — the expected Return by taking a given action from a given state, and then, by executing actions based on a given policy π after that. In other words, the State-Action Value function maps a State-Action pair to its Value.



The State-Action Value Function maps a State-Action pair to its Value

## **Relationship between Reward, Return and Value**

* Reward is the immediate reward obtained for a single action.
* Return is the total of all the discounted rewards obtained till the end of that episode.
* Value is the mean Return (aka expected Return) over many episodes.

Think of Reward as immediate pleasure and Value as long-lasting happiness.

Intuitively one can think of Value as follows. Like a human, the agent learns from experience. As it interacts with the environment and completes episodes, it obtains the Returns for each episode.

As it accumulates more experience (ie. obtains Returns for more and more episodes), it gets a sense of which states, and which actions in those states yield the most Return.

It stores this ‘experience’ as ‘Value’.

## **Why does the Value depend on the policy we’re following?**

Clearly, the rewards we get (and hence the Return and therefore the Value) depends on the action we take from a given state. And since the action depends on the chosen Policy, it follows that the Value depends on the Policy.

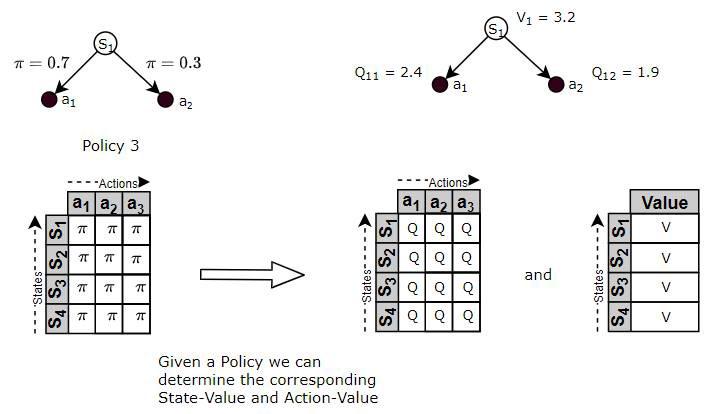
Eg. if our policy was to choose entirely random actions (i.e., sample actions from a uniform distribution), the Value (expected reward) of a state would probably be pretty low, since we’re definitely not choosing the best possible actions.

Eg. Instead, if our policy was to choose actions from a probability distribution that produces the maximum rewards when sampled, the Value (expected reward) of a state would be much higher.

# **Use the Value Function to compare Policies**

Now that we understand Value, let’s go back to our earlier discussion about comparing two policies to see which is better. How do we evaluate what ’better’ means?

Given two policies, we can determine the corresponding State-Value or State-Action Value functions for each of those policies, by following the policy and evaluating the Returns.

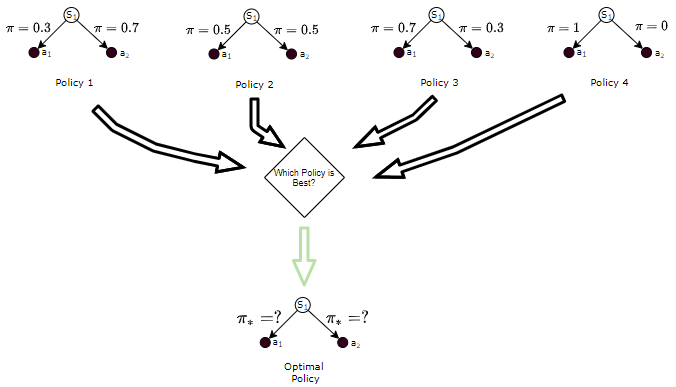


Once we have the respective Value Functions, we can use those Value Functions to compare the policies. The policy, whose Value Function is higher, is better because that means it will yield higher Returns.

# **The ‘best’ Policy is called the Optimal Policy**

Since we can now compare policies to figure out which ones are 'good' and which ones are 'bad’, we can also use that to find the 'best' policy. This is known as the Optimal Policy.

The Optimal Policy is the policy that will yield more Returns to the agent than all other policies.



The Optimal Policy is the one that is better than all other policies

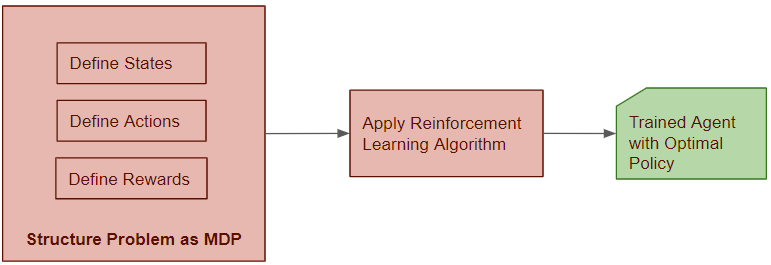
# **Solve the RL Problem by Finding the Optimal Policy**

So now we have the approach to solve an RL problem.

We structure our problem as an MDP and we can then solve this problem by building an agent viz. the brains of the MDP, in such a way that it can make decisions about which action to take. It should do this in a way that maximizes Returns.

In other words, we need to find the Optimal Policy for the agent. Once it has the Optimal Policy it simply uses that policy to pick actions from any state.

We will apply a Reinforcement Learning algorithm to build an agent model and train it to find the Optimal Policy. Finding the Optimal Policy essentially solves the RL problem.



# **RL Solution Categories**

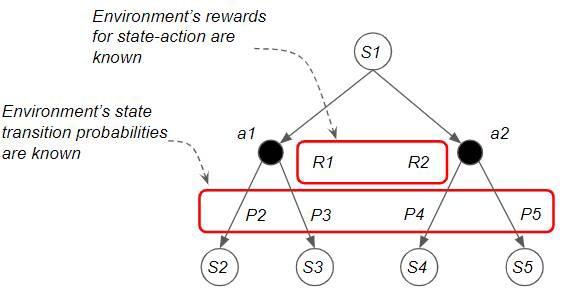
‘Solving’ a Reinforcement Learning problem basically amounts to finding the Optimal Policy (or Optimal Value). There are many algorithms, which we can group into different categories.

## **Model-based vs Model-free**

Very broadly, solutions are either:

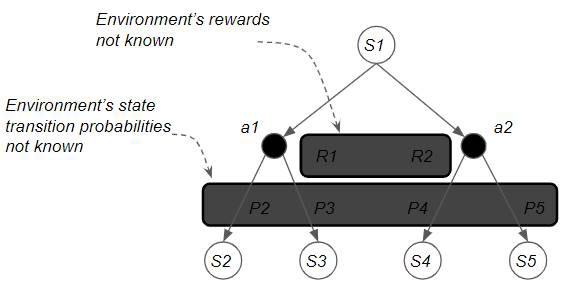
1. Model-based (aka Planning)
2. Model-free(aka Reinforcement Learning)

Model-based approaches are used when the internal operation of the environment is known. In other words, we can reliably say what Next State and Reward will be output by the environment when some Action is performed from some Current State.



Model-based: Internal operation of the environment is known

Model-free approaches are used when the environment is very complex and its internal dynamics are not known. They treat the environment as a black-box.

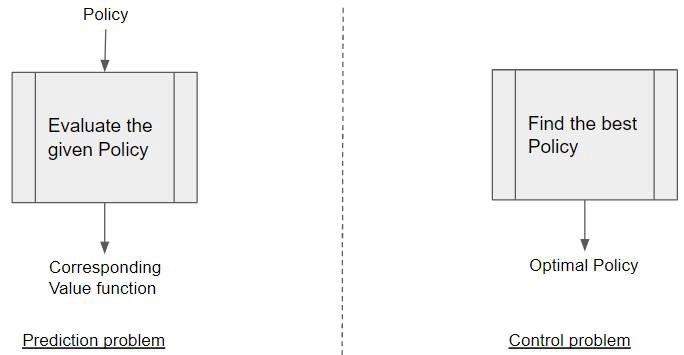


Model-free: Environment is a black box

## **Prediction vs Control**

Another high-level distinction is between Prediction and Control.

With a **Prediction problem**, we are given a Policy as input, and the goal is to output the corresponding Value function. This could be any Policy, not necessarily an Optimal Policy.



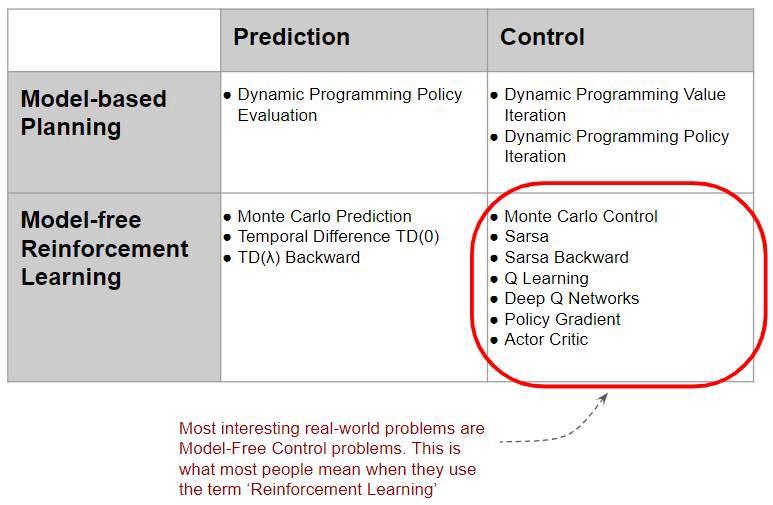
Prediction vs Control problems

With a **Control problem**, no input is provided, and the goal is to explore the policy space and find the **Optimal Policy**.

Most practical problems are Control problems, as our goal is to find the Optimal Policy.

## **Classifying Popular RL Algorithms**

The most common RL Algorithms can be categorized as below:



Taxonomy of well-known RL Solutions

Most interesting real-world RL problems are model-free control problems. So we will not explore model-based solutions. Everything we discuss from here on pertains only to model-free control solutions.

# **Model-based Approaches**

Because they can produce the exact outcome of every state and action interaction, model-based approaches can find a solution analytically without actually interacting with the environment.

As an example, with a model-based approach to play chess, you would program in all the rules and strategies of the game of chess. On the other hand, a model-free algorithm would know nothing about the game of chess itself. Its only knowledge would be generic information such as how states are represented and what actions are possible. It learns about chess only in an abstract sense by observing what reward it obtains when it tries some action.

Most real-world problems are model-free because the environment is usually too complex to build a model.

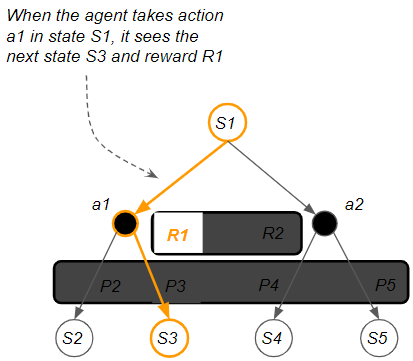
# **Model-free Approaches**

Model-free solutions, by contrast, are able to observe the environment’s behavior only by actually interacting with it.

## **Interact with the environment**

Since the internal operation of the environment is invisible to us, how does the model-free algorithm observe the environment’s behavior?

We learn how it behaves by interacting with it, one action at a time. The algorithm acts as the agent, takes an action, observes the next state and reward, and repeats.

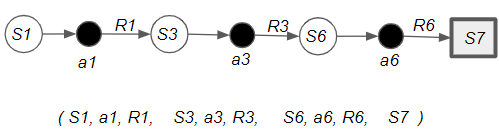


Model-free algorithms learn about the environment by interacting with it, one action at a time.

The agent acquired experience through trial and error. It tries steps and receives positive or negative feedback. This is much the same as a human would learn.

## **Trajectory of interactions**

As the agent takes each step, it follows a path (ie. trajectory).



A trajectory of interactions

The agent’s trajectory becomes the algorithm’s ‘training data’.

# **The Bellman Equation is the foundation for all RL algorithms**

Before we get into the algorithms used to solve RL problems, we need a little bit of math to make these concepts more precise.

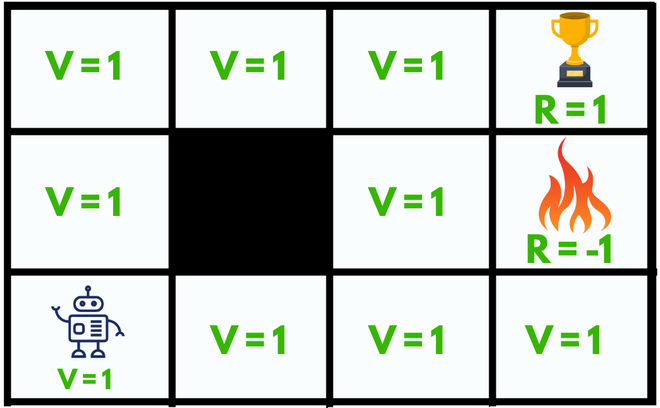
The math is actually quite intuitive — it is all based on one simple relationship known as the Bellman Equation.

This relationship is the foundation for all the RL algorithms. This equation has several forms, but they are all based on the same basic idea. Let’s go through this step-by-step to build up the intuition for it.

According to the **Bellman Equation**, long-term- reward in a **given action is equal to the reward from the current action combined with the expected reward from the future actions** taken at the following time. Let’s try to understand first.

**Let’s take an example:**

Here we have a **maze** which is our environment and the sole **goal** of our agent is to reach the **trophy state (R = 1)** or to get **Good reward** and to **avoid the fire state** because it will be a **failure (R = -1)** or will get **Bad reward**.



**What happens without Bellman Equation?**

Initially, we will give our agent some time to explore the environment and let it figure out a path to the goal. As soon as it reaches its goal, it will **back trace its steps** back to its starting position and **mark values of all the states** which eventually leads towards the goal as **V = 1**.

The agent will face **no problem until** we **change its starting position**, as it will **not be able** to find a path towards the trophy state since the value of all the states is **equal to 1**. So, to solve this problem we should use **Bellman Equation:**

***V(s)=maxa(R(s,a)+ γV(s’))***

**State(s):** current state where the agent is in the environment

**Next State(s’):** After taking action(a) at state(s) the agent reaches s’

**Value(V):** Numeric representation of a state which helps the agent to find its path. **V(s)** here means the value of the state s.

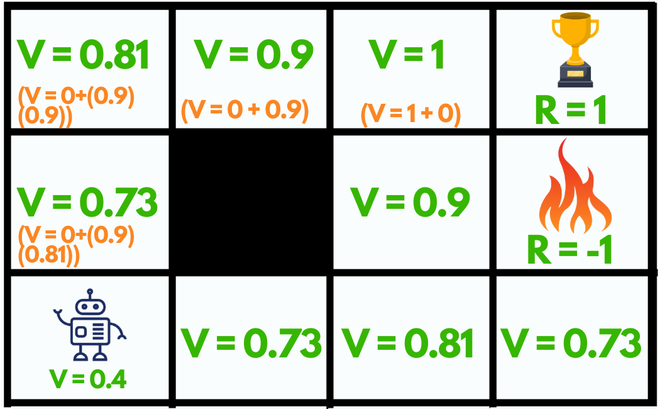
**Reward(R):** treat which the agent gets after performing an action(a).

* **R(s):** reward for being in the state s
* **R(s,a):** reward for being in the state and performing an action a
* **R(s,a,s’):** reward for being in a state s, taking an action a and ending up in s’

e.g. **Good reward** can be **+1**, **Bad reward** can be **-1**, **No reward** can be **0**.

**Action(a):** set of possible actions that can be taken by the agent in the state(s). e.g. (**LEFT**, **RIGHT**, **UP**, **DOWN**)

**Discount factor(γ):** determines how much the agent cares about rewards in the distant future relative to those in the immediate future. It has a value **between 0 and 1**. **Lower value** encourages **short–term** rewards while **higher value** promises **long-term reward**



The **max** denotes the most **optimum** action among all the actions that the agent can take in a particular state which can lead to the reward after **repeating this process every consecutive step.**

**For example:**

* The state left to the fire state (V = 0.9) can go **UP**, **DOWN**, **RIGHT** but **NOT LEFT** because it’s a wall(not accessible). Among all these actions available the **maximum value** for that state is the **UP** action.
* The current starting state of our agent can choose any **random** action **UP** or **RIGHT** since both lead towards the reward with the **same number of steps.**

By using the Bellman equation our agent will calculate the value of every step **except for the trophy and the fire state (V = 0)**, they cannot have values since they are the **end of the maze**.

So, after making such a plan our agent can easily accomplish its goal by just following the **increasing values.**

**Subproblems of the RL problem**

**Prediction and control**

* Prediction : evaluate the future (for a given policy)
* Control: Optimis the future (find the best policy)
* These can be strongly related to the
  + 𝝅(s) = argmax V(s)

**Learning and planning**

* Learning
  + When the environment is initially known
  + The agent interacts with the environment
* Planning
  + A model of the environment is given ( or learnt)
  + The agent plans in this model (without external interaction)
  + A.k.a reasoning,pondering,thought,search,...

**Conclusion**

* **For further learning and planning the RL model** we can use neural networks, and use **deep learning** technique to learn and predict.
* Eg:
  + Atari game
  + Grid world prediction

# Q learning

* Q-learning is a model-free reinforcement learning algorithm.
* Q-learning is a values-based learning algorithm. Value based algorithms updates the value function based on an equation(particularly Bellman equation). Whereas the other type, policy-based estimates the value function with a greedy policy obtained from the last policy improvement.
* Q-learning is an off-policy learner. Means it learns the value of the optimal policy independently of the agent’s actions. On the other hand, an on-policy learner learns the value of the policy being carried out by the agent, including the exploration steps and it will find a policy that is optimal, taking into account the exploration inherent in the policy.

**What’s this ‘Q’?**

The ‘Q’ in Q-learning stands for quality. Quality here represents how useful a given action is in gaining some future reward.

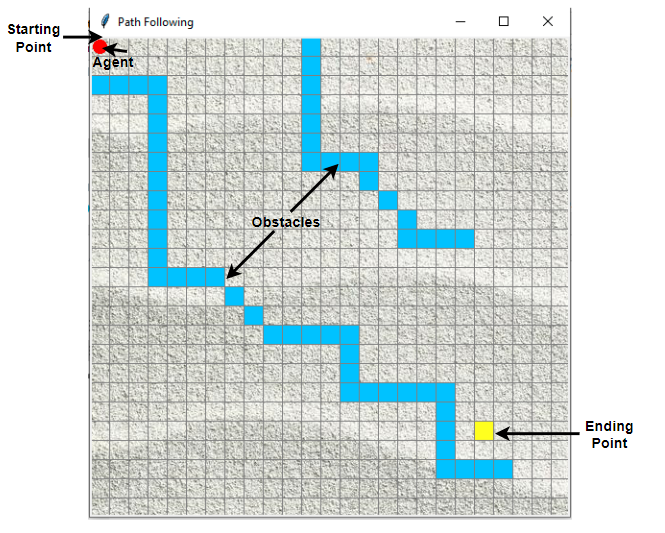
**Q-learning Definition**

* **Q\*(s,a)** is the expected value (cumulative discounted reward) of doing a in state s and then following the optimal policy.
* Q-learning uses **Temporal Differences(TD)** to estimate the value of Q\*(s,a). Temporal difference is an agent learning from an environment through episodes with no prior knowledge of the environment.
* The agent maintains a table of **Q[S, A]**, where **S** is the set of **states** and **A** is the set of **actions**.
* Q[s, a] represents its current estimate of Q\*(s,a).

Q-learning Simple Example

In this section Q-learning has been explained along with a demo.

Let’s say an agent has to move from a starting point to an ending point along a path that has obstacles. Agent needs to reach the target in the shortest path possible without hitting the obstacles and he needs to follow the boundary covered by the obstacles. For our convenience, I have introduced this in a customized grid environment as follows.



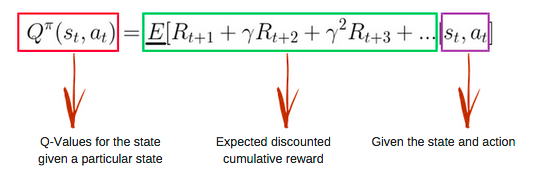
Agent and its Environment

Introducing the Q-Table

Q-Table is the data structure used to calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state. To learn each value of the Q-table, Q-Learning algorithm is used.

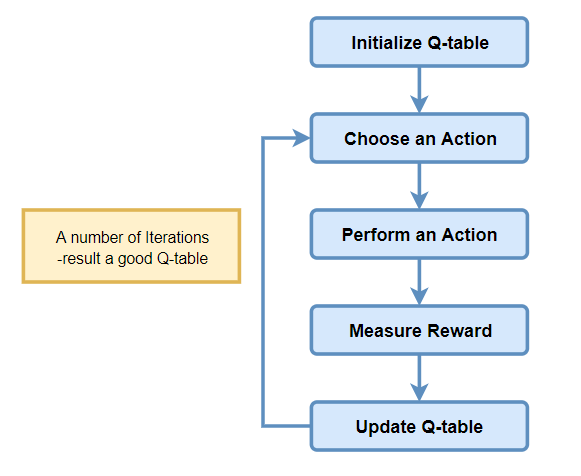
Q-function

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



Bellman Equation. Source

Q-learning Algorithm Process

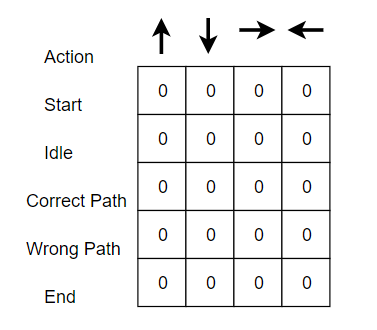
****

Q-learning Algorithm

Step 1: Initialize the Q-Table

First the Q-table has to be built. There are n columns, where n= number of actions. There are m rows, where m= number of states.

In our example n=Go Left, Go Right, Go Up and Go Down and m= Start, Idle, Correct Path, Wrong Path and End. First, let’s initialize the values at 0.



Initial Q-table

Step 2 : Choose an Action

Step 3 : Perform an Action

The combination of steps 2 and 3 is performed for an undefined amount of time. These steps run until the time training is stopped, or when the training loop stops as defined in the code.

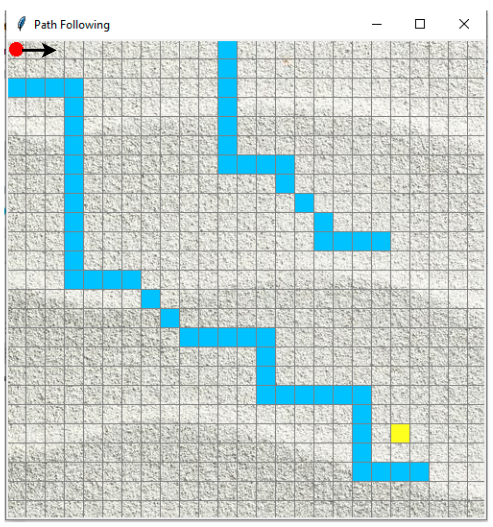
First, an action (a) in the state (s) is chosen based on the Q-Table. Note that, as mentioned earlier, when the episode initially starts, every Q-value should be 0.

Then, update the Q-values for being at the start and moving right using the Bellman equation which is stated above.

***Epsilon greedy strategy*** concept comes into play here. In the beginning, the epsilon rates will be higher. The agent will explore the environment and randomly choose actions. This occurs like this logically,since the agent does not know anything about the environment. As the agent explores the environment, the epsilon rate decreases and the agent starts to exploit the environment.

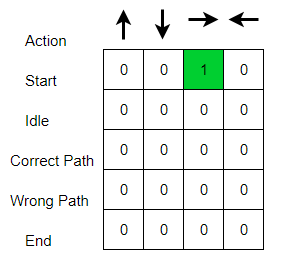
During the process of exploration, the agent progressively becomes more confident in estimating the Q-values.

In our Agent example, when the training of an agent starts, the agent is completely unaware about the environment. So let’s say it takes a random action to its ‘right’ direction.



Action : Agent follows ‘right’

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



Updated Q-table

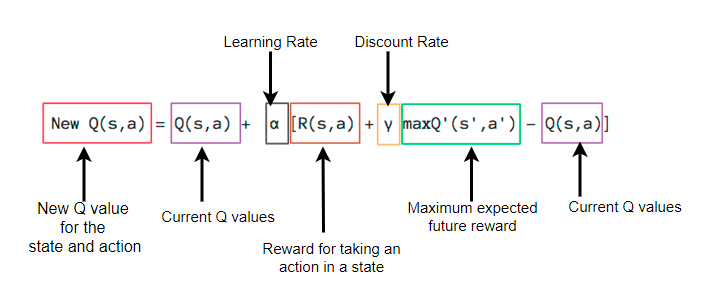
**Steps 4 : Measure Reward**

Now we have taken an action and observed an outcome and reward.

**Steps 5 : Evaluate**

We need to update the function Q(s,a).

This process is repeated again and again until the learning is stopped. In this way the Q-Table is been updated and the value function Q is maximized. Here the Q(state, action) returns the **expected future reward** of that action at that state.



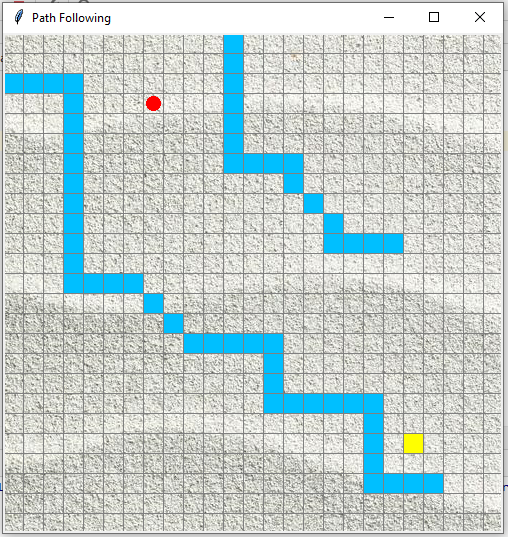
Bellman Equation Explanation for the episodes

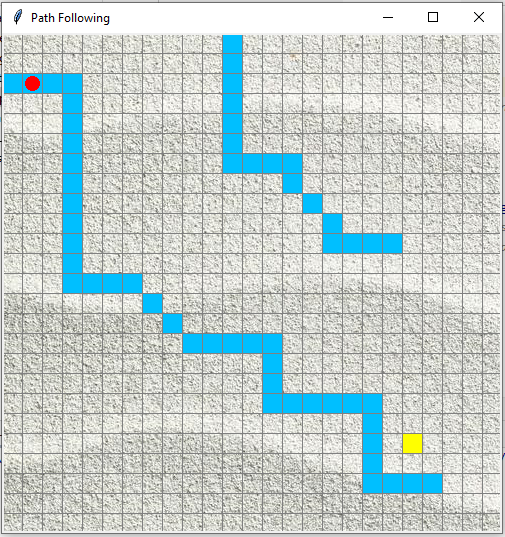
In the example, I have entered the rewarding scheme as follows.

*Reward when reach step closer to goal= +1*

*Reward when hit obstacle =-1*

*Reward when idle=0*

**

**

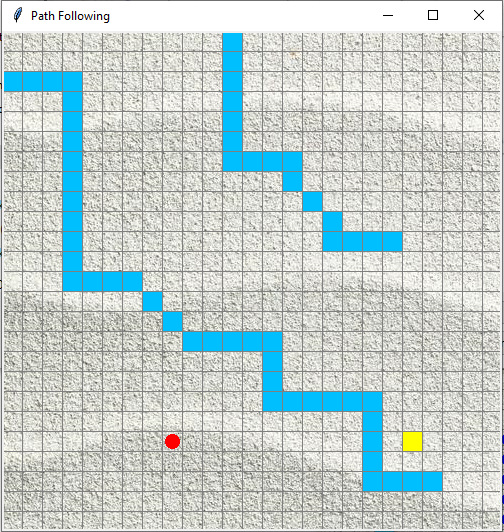
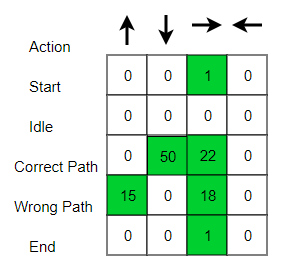
**

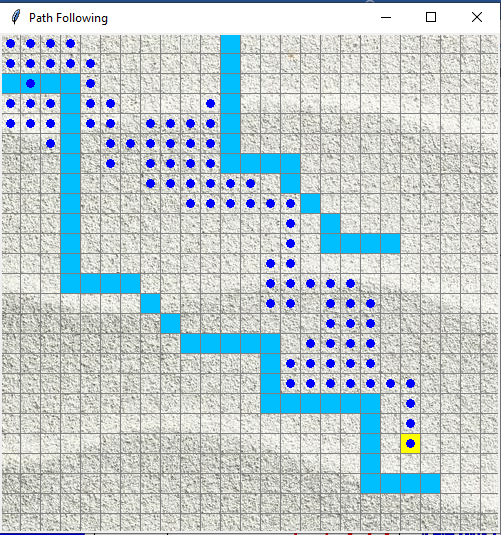
Figure (a) Positive Reward, (b) & (c) Negative Rewards

Initially, we explore the agent’s environment and update the Q-Table. When the Q-Table is ready, the agent start to exploit the environment and start taking better actions. Final Q-table can be like following (for example).

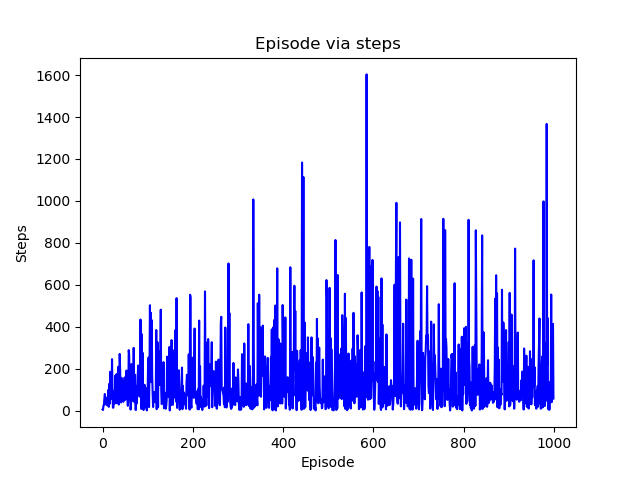


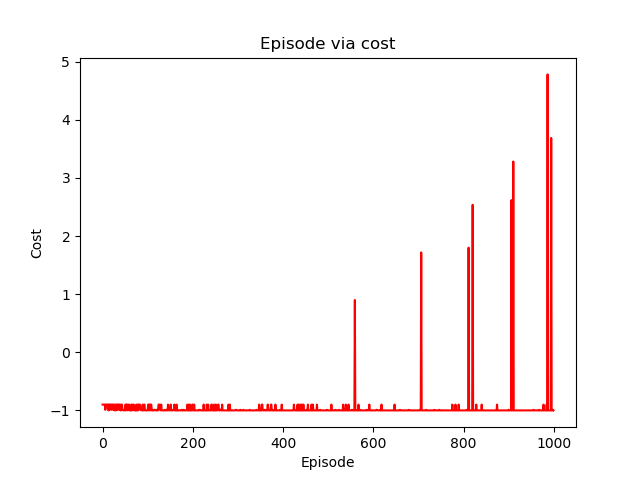
Example final Q-table

Following are the outcomes that results the agent’s shortest path towards goal after training.



Agent’s navigation towards the goal





Plotting the results for the number of steps (a) Episode via steps, (b) Episode via cost