Unit 4

Introduction to Semi-Supervised Learning

Semi-Supervised learning is a type of Machine Learning algorithm that represents the intermediate ground between Supervised and Unsupervised learning algorithms. It uses the combination of labeled and unlabeled datasets during the training period.

Before understanding the Semi-Supervised learning, you should know the main categories of Machine Learning algorithms. Machine Learning consists of three main categories: Supervised Learning**,** Unsupervised Learning**, and** Reinforcement Learning**.** Further, the basic difference between Supervised and unsupervised learning is that *supervised learning datasets consist of an output label training data associated with each tuple,* and *unsupervised datasets do not consist the same.* **Semi-supervised learning is an important category that lies between the Supervised and Unsupervised machine learning.** Although Semi-supervised learning is the middle ground between supervised and unsupervised learning and operates on the data that consists of a few labels, it mostly consists of unlabeled data. As labels are costly, but for the corporate purpose, it may have few labels.

The basic disadvantage of supervised learning is that it requires hand-labeling by ML specialists or data scientists, and it also requires a high cost to process. Further unsupervised learning also has a limited spectrum for its applications. **To overcome these drawbacks of supervised learning and unsupervised learning algorithms, the concept of Semi-supervised learning is introduced**. In this algorithm, training data is a combination of both labeled and unlabeled data. However, labeled data exists with a very small amount while it consists of a huge amount of unlabeled data. Initially, similar data is clustered along with an unsupervised learning algorithm, and further, it helps to label the unlabeled data into labeled data. It is why label data is a comparatively, more expensive acquisition than unlabeled data.

We can imagine these algorithms with an example. Supervised learning is where a student is under the supervision of an instructor at home and college. Further, if that student is self-analyzing the same concept without any help from the instructor, it comes under unsupervised learning. Under semi-supervised learning, the student has to revise itself after analyzing the same concept under the guidance of an instructor at college.

Assumptions followed by Semi-Supervised Learning

To work with the unlabeled dataset, there must be a relationship between the objects. To understand this, semi-supervised learning uses any of the following assumptions:

- **Continuity Assumption:**
  As per the continuity assumption, the objects near each other tend to share the same group or label. This assumption is also used in supervised learning, and the datasets are separated by the decision boundaries. But in semi-supervised, the decision boundaries are added with the smoothness assumption in low-density boundaries.

- **Cluster assumptions-** In this assumption, data are divided into different discrete clusters. Further, the points in the same cluster share the output label.
- **Manifold assumptions-** This assumption helps to use distances and densities, and this data lie on a manifold of fewer dimensions than input space.
- The dimensional data are created by a process that has less degree of freedom and may be hard to model directly. **(This assumption becomes practical if high).**

Working of Semi-Supervised Learning

Semi-supervised learning uses pseudo labeling to train the model with less labeled training data than supervised learning. The process can combine various neural network models and training ways. The whole working of semi-supervised learning is explained in the below points:

- Firstly, it trains the model with less amount of training data similar to the supervised learning models. The training continues until the model gives accurate results.
- The algorithms use the unlabeled dataset with pseudo labels in the next step, and now the result may not be accurate.
- Now, the labels from labeled training data and pseudo labels data are linked together.
- The input data in labeled training data and unlabeled training data are also linked.
- In the end, again train the model with the new combined input as did in the first step. It will reduce errors and improve the accuracy of the model.

Difference between Semi-supervised and Reinforcement Learning.

Reinforcement learning is different from semi-supervised learning, as it works with rewards and feedback. ***Reinforcement learning aims to maximize the rewards by their hit and trial actions, whereas in semi-supervised learning, we train the model with a less labeled dataset.***

Real-world applications of Semi-supervised Learning-

Semi-supervised learning models are becoming more popular in the industries. Some of the main applications are as follows.

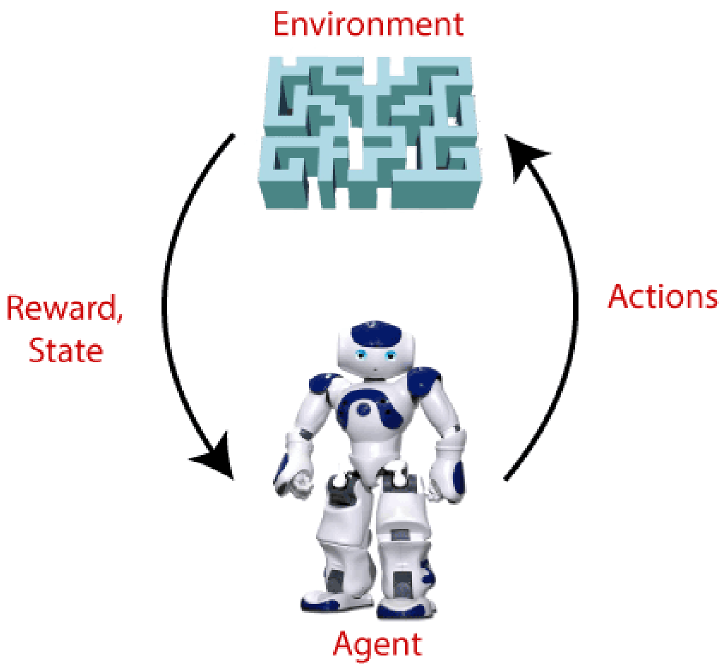
- **Speech Analysis-** It is the most classic example of semi-supervised learning applications. Since, labeling the audio data is the most impassable task that requires many human resources, this problem can be naturally overcome with the help of applying SSL in a Semi-supervised learning model.
- **Web content classification-** However, this is very critical and impossible to label each page on the internet because it needs mode human intervention. Still, this problem can be reduced through Semi-Supervised learning algorithms. Further, Google also uses semi-supervised learning algorithms to rank a webpage for a given query.
- **Protein sequence classification-** DNA strands are larger, they require active human intervention. So, the rise of the Semi-supervised model has been proximate in this field.

- **Text document classifier-** As we know, it would be very unfeasible to find a large amount of labeled text data, so semi-supervised learning is an ideal model to overcome this.

Reinforcement Learning

What is Reinforcement Learning?

- Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback, and for each bad action, the agent gets negative feedback or penalty.
- In Reinforcement Learning, the agent learns automatically using feedbacks without any labeled data, unlike supervised learning.
- Since there is no labeled data, so the agent is bound to learn by its experience only.
- RL solves a specific type of problem where decision making is sequential, and the goal is long-term, such as **game-playing, robotics**, etc.
- The agent interacts with the environment and explores it by itself. The primary goal of an agent in reinforcement learning is to improve the performance by getting the maximum positive rewards.
- The agent learns with the process of hit and trial, and based on the experience, it learns to perform the task in a better way. Hence, we can say that ***"Reinforcement learning is a type of machine learning method where an intelligent agent (computer program) interacts with the environment and learns to act within that."*** How a Robotic dog learns the movement of his arms is an example of Reinforcement learning.
- It is a core part of Artificial intelligence, and all AI agent works on the concept of reinforcement learning. Here we do not need to pre-program the agent, as it learns from its own experience without any human intervention.
- **Example:** Suppose there is an AI agent present within a maze environment, and his goal is to find the diamond. The agent interacts with the environment by performing some actions, and based on those actions, the state of the agent gets changed, and it also receives a reward or penalty as feedback.
- The agent continues doing these three things (**take action, change state/remain in the same state, and get feedback**), and by doing these actions, he learns and explores the environment.
- The agent learns that what actions lead to positive feedback or rewards and what actions lead to negative feedback penalty. As a positive reward, the agent gets a positive point, and as a penalty, it gets a negative point.



Terms used in Reinforcement Learning

- **Agent():** An entity that can perceive/explore the environment and act upon it.
- **Environment():** A situation in which an agent is present or surrounded by. In RL, we assume the stochastic environment, which means it is random in nature.
- **Action():** Actions are the moves taken by an agent within the environment.
- **State():** State is a situation returned by the environment after each action taken by the agent.
- **Reward():** A feedback returned to the agent from the environment to evaluate the action of the agent.
- **Policy():** Policy is a strategy applied by the agent for the next action based on the current state.
- **Value():** It is expected long-term retuned with the discount factor and opposite to the short-term reward.
- **Q-value():** It is mostly similar to the value, but it takes one additional parameter as a current action (a).

Key Features of Reinforcement Learning

- In RL, the agent is not instructed about the environment and what actions need to be taken.
- It is based on the hit and trial process.
- The agent takes the next action and changes states according to the feedback of the previous action.
- The agent may get a delayed reward.

- The environment is stochastic, and the agent needs to explore it to reach to get the maximum positive rewards.

Approaches to implement Reinforcement Learning

There are mainly three ways to implement reinforcement-learning in ML, which are:

1. **Value-based:**
   The value-based approach is about to find the optimal value function, which is the maximum value at a state under any policy. Therefore, the agent expects the long-term return at any state(s) under policy π.
2. **Policy-based:**
   Policy-based approach is to find the optimal policy for the maximum future rewards without using the value function. In this approach, the agent tries to apply such a policy that the action performed in each step helps to maximize the future reward. The policy-based approach has mainly two types of policy:
   - **Deterministic:** The same action is produced by the policy (π) at any state.
   - **Stochastic:** In this policy, probability determines the produced action.
3. **Model-based:** In the model-based approach, a virtual model is created for the environment, and the agent explores that environment to learn it. There is no particular solution or algorithm for this approach because the model representation is different for each environment.

Elements of Reinforcement Learning

There are four main elements of Reinforcement Learning, which are given below:

1. Policy
2. Reward Signal
3. Value Function
4. Model of the environment

1) Policy: A policy can be defined as a way how an agent behaves at a given time. It maps the perceived states of the environment to the actions taken on those states. A policy is the core element of the RL as it alone can define the behavior of the agent. In some cases, it may be a simple function or a lookup table, whereas, for other cases, it may involve general computation as a search process. It could be deterministic or a stochastic policy:

For deterministic policy: $a = \pi(s)$
For stochastic policy: $\pi(a \mid s) = P[A_t = a \mid S_t = s]$

2) Reward Signal: The goal of reinforcement learning is defined by the reward signal. At each state, the environment sends an immediate signal to the learning agent, and this signal is known as a **reward signal**. These rewards are given according to the good and bad actions taken by the agent. The agent's main objective is to maximize the total number of rewards for good actions. The reward signal can change the policy, such as if an action selected by the agent leads to low reward, then the policy may change to select other actions in the future.

3) Value Function: The value function gives information about how good the situation and action are and how much reward an agent can expect. A reward indicates the **immediate signal for each good and bad action**, whereas a value function specifies **the good state and action for the future**. The value function depends on the reward as, without reward, there could be no value. The goal of estimating values is to achieve more rewards.

4) Model: The last element of reinforcement learning is the model, which mimics the behavior of the environment. With the help of the model, one can make inferences about how the environment will behave. Such as, if a state and an action are given, then a model can predict the next state and reward.

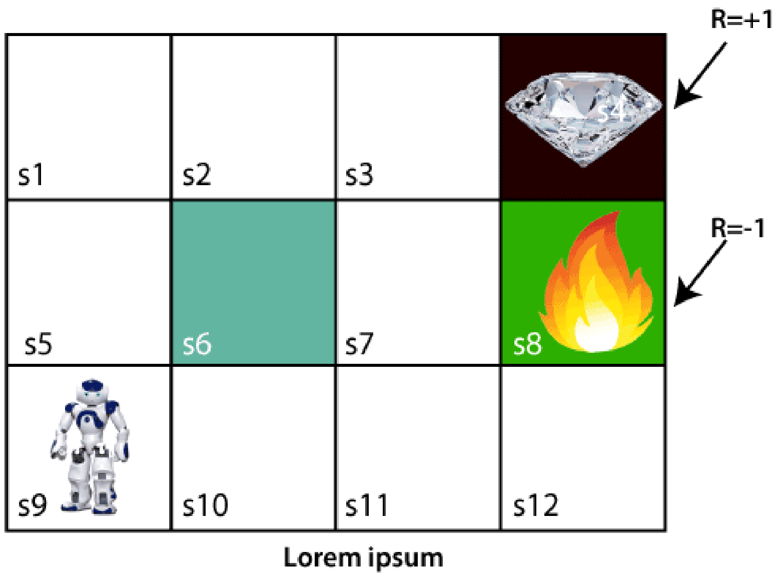
The model is used for planning, which means it provides a way to take a course of action by considering all future situations before actually experiencing those situations. The approaches for solving the RL problems **with the help of the model** are termed as the **model-based approach**. Comparatively, an approach **without using a model** is called a **model-free approach**.

How does Reinforcement Learning Work?

To understand the working process of the RL, we need to consider two main things:

- **Environment:** It can be anything such as a room, maze, football ground, etc.
- **Agent:** An intelligent agent such as AI robot.

Let's take an example of a maze environment that the agent needs to explore. Consider the below image:

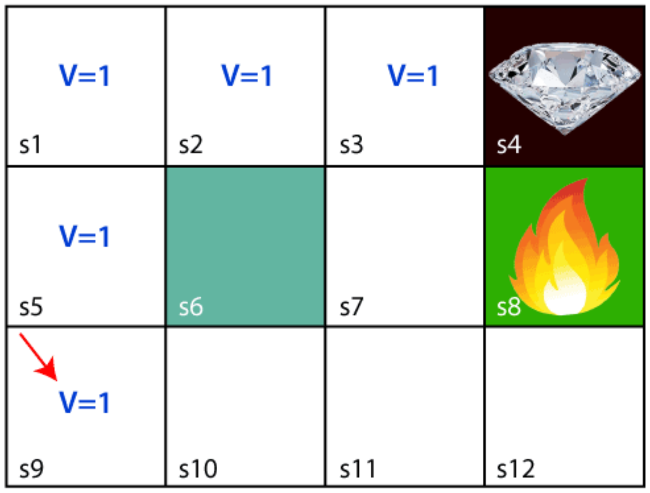


In the above image, the agent is at the very first block of the maze. The maze is consisting of an S_6 block, which is a **wall**, S_8 a **fire pit**, and S_4 a **diamond block.**

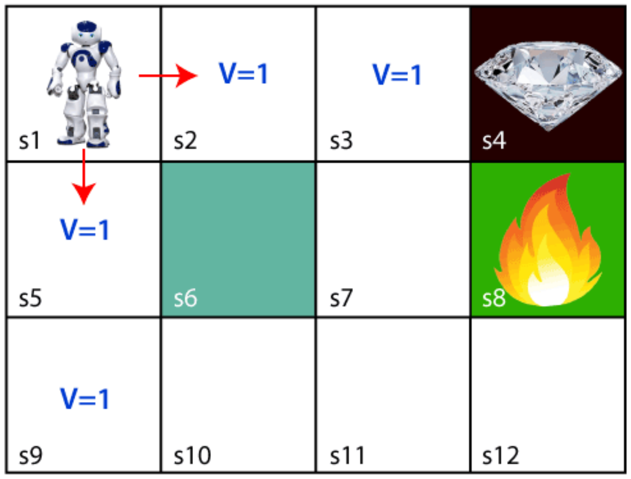
The agent cannot cross the S_6 block, as it is a solid wall. If the agent reaches the S_4 block, then get the **+1 reward;** if it reaches the fire pit, then gets **-1 reward point**. It can take four actions**: move up, move down, move left, and move right.**

The agent can take any path to reach to the final point, but he needs to make it in possible fewer steps. Suppose the agent considers the path **S9-S5-S1-S2-S3**, so he will get the +1-reward point.

The agent will try to remember the preceding steps that it has taken to reach the final step. To memorize the steps, it assigns 1 value to each previous step. Consider the below step:



Now, the agent has successfully stored the previous steps assigning the 1 value to each previous block. But what will the agent do if he starts moving from the block, which has 1 value block on both sides? Consider the below diagram:



It will be a difficult condition for the agent whether he should go up or down as each block has the same value. So, the above approach is not suitable for the agent to reach the destination. Hence to solve the problem, we will use the **Bellman equation**, which is the main concept behind reinforcement learning.

The Bellman Equation

The Bellman equation was introduced by the Mathematician **Richard Ernest Bellman in the year 1953**, and hence it is called as a Bellman equation. It is associated with dynamic programming and used to calculate the values of a decision problem at a certain point by including the values of previous states.

It is a way of calculating the value functions in dynamic programming or environment that leads to modern reinforcement learning.

The key-elements used in Bellman equations are:

- Action performed by the agent is referred to as "a"
- State occurred by performing the action is "s."
- The reward/feedback obtained for each good and bad action is "R."
- A discount factor is Gamma "γ."

The Bellman equation can be written as:

1. $V(s) = \max [R(s,a) + \gamma V(s`)]$

Where,

V(s)= value calculated at a particular point.

R(s,a) = Reward at a particular state s by performing an action.

γ = Discount factor

V(s`) = The value at the previous state.

In the above equation, we are taking the max of the complete values because the agent tries to find the optimal solution always.

So now, using the Bellman equation, we will find value at each state of the given environment. We will start from the block, which is next to the target block.

For 1st block:

$V(s3) = \max [R(s,a) + \gamma V(s`)]$, here $V(s')= 0$ because there is no further state to move.

$V(s3)= \max[R(s,a)]$=> $V(s3)= \max[1]$=> **$V(s3)= 1$.**

For 2nd block:

V(s2) = max [R(s,a) + γV(s`)], here γ= 0.9(lets), V(s')= 1, and R(s, a)= 0, because there is no reward at this state.

V(s2)= max[0.9(1)]=> V(s)= max[0.9]=> **V(s2) =0.9**

For 3rd block:

V(s1) = max [R(s,a) + γV(s`)], here γ= 0.9(lets), V(s')= 0.9, and R(s, a)= 0, because there is no reward at this state also.

V(s1)= max[0.9(0.9)]=> V(s3)= max[0.81]=> **V(s1) =0.81**

For 4th block:

V(s5) = max [R(s,a) + γV(s`)], here γ= 0.9(lets), V(s')= 0.81, and R(s, a)= 0, because there is no reward at this state also.

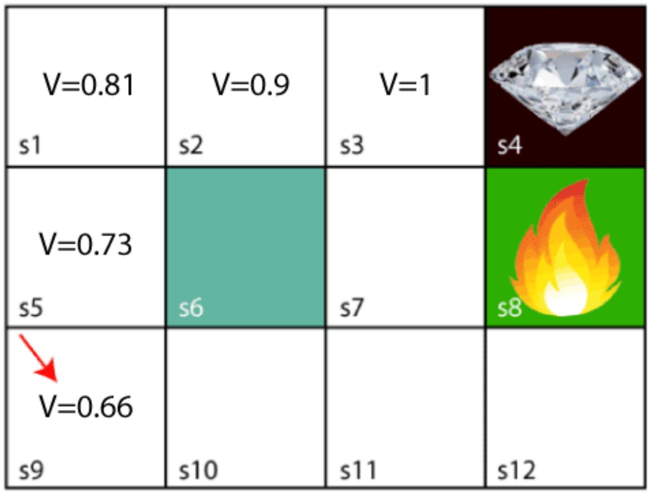
V(s5)= max[0.9(0.81)]=> V(s5)= max[0.81]=> **V(s5) =0.73**

For 5th block:

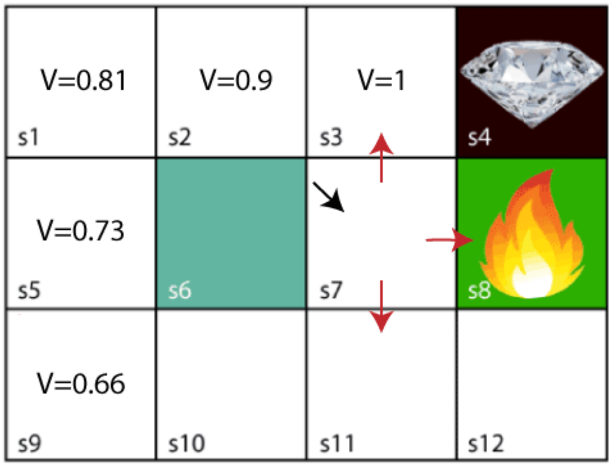
V(s9) = max [R(s,a) + γV(s`)], here γ= 0.9(lets), V(s')= 0.73, and R(s, a)= 0, because there is no reward at this state also.

V(s9)= max[0.9(0.73)]=> V(s4)= max[0.81]=> **V(s4) =0.66**

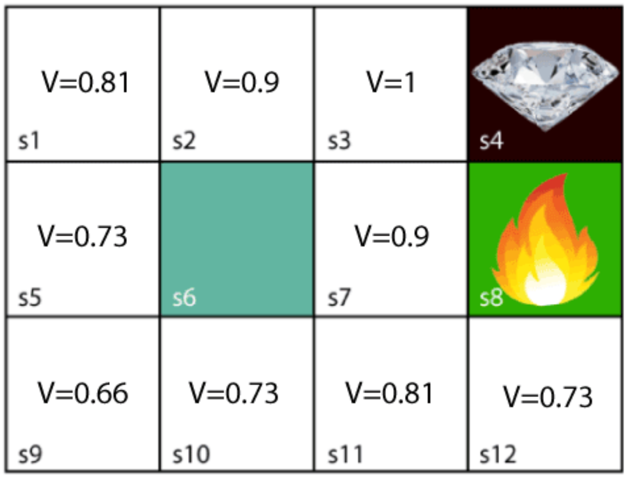
Consider the below image:



Now, we will move further to the 6th block, and here agent may change the route because it always tries to find the optimal path. So now, let's consider from the block next to the fire pit.



Now, the agent has three options to move; if he moves to the blue box, then he will feel a bump if he moves to the fire pit, then he will get the -1 reward. But here we are taking only positive rewards, so for this, he will move to upwards only. The complete block values will be calculated using this formula. Consider the below image:



Types of Reinforcement learning

There are mainly two types of reinforcement learning, which are:

- **Positive Reinforcement**
- **Negative Reinforcement**

Positive Reinforcement:

The positive reinforcement learning means adding something to increase the tendency that expected behavior would occur again. It impacts positively on the behavior of the agent and increases the strength of the behavior.

This type of reinforcement can sustain the changes for a long time, but too much positive reinforcement may lead to an overload of states that can reduce the consequences.

Negative Reinforcement:

The negative reinforcement learning is opposite to the positive reinforcement as it increases the tendency that the specific behavior will occur again by avoiding the negative condition.

It can be more effective than the positive reinforcement depending on situation and behavior, but it provides reinforcement only to meet minimum behavior.

How to represent the agent state?

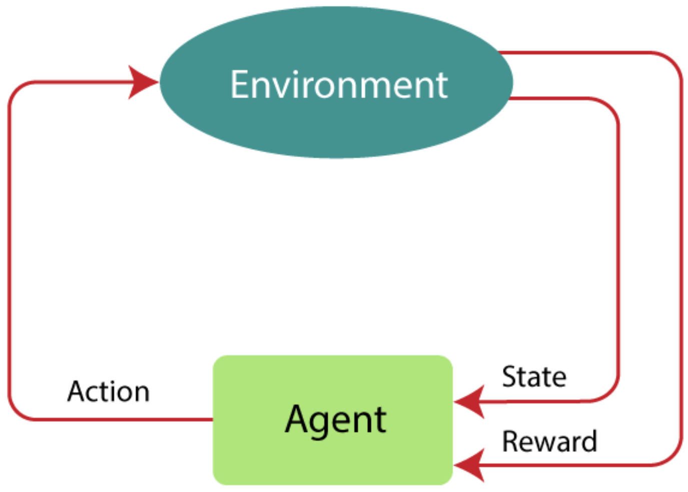
We can represent the agent state using the **Markov State** that contains all the required information from the history. The State St is Markov state if it follows the given condition:

$P[S_t+1 \mid S_t] = P[S_t +1 \mid S_1,......, S_t]$

The Markov state follows the **Markov property**, which says that the future is independent of the past and can only be defined with the present. The RL works on fully observable environments, where the agent can observe the environment and act for the new state. The complete process is known as Markov Decision process, which is explained below:

Markov Decision Process

Markov Decision Process or MDP, is used to **formalize the reinforcement learning problems**. If the environment is completely observable, then its dynamic can be modeled as a **Markov Process**. In MDP, the agent constantly interacts with the environment and performs actions; at each action, the environment responds and generates a new state.



MDP is used to describe the environment for the RL, and almost all the RL problem can be formalized using MDP.

MDP contains a tuple of four elements (S, A, P_a, R_a):

- A set of finite States S
- A set of finite Actions A
- Rewards received after transitioning from state S to state S', due to action a.
- Probability P_a.

MDP uses **Markov property**, and to better understand the MDP, we need to learn about it.

Markov Property:
It says that ***"If the agent is present in the current state S1, performs an action a1 and move to the state s2, then the state transition from s1 to s2 only depends on the current state and future action and states do not depend on past actions, rewards, or states."***

Or, in other words, as per Markov Property, the current state transition does not depend on any past action or state. Hence, MDP is an RL problem that satisfies the Markov property. Such as in a **Chess game, the players only focus on the current state and do not need to remember past actions or states**.

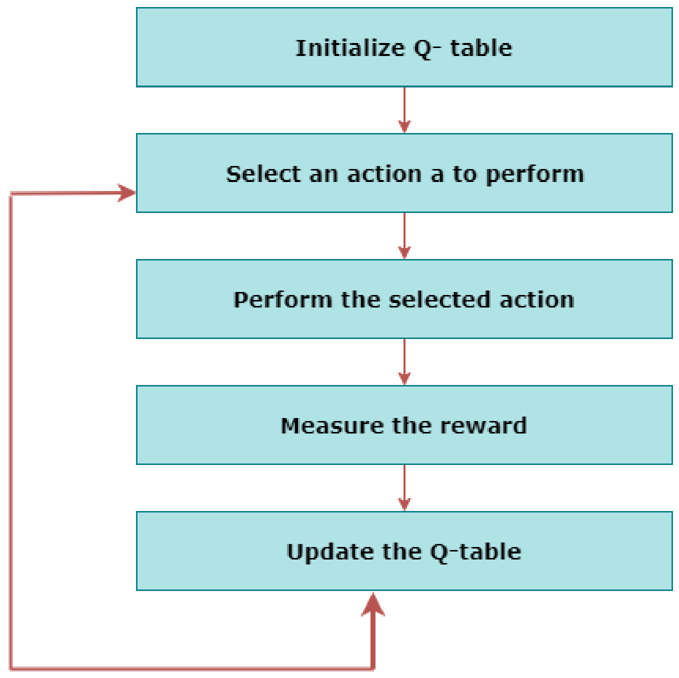
Finite MDP:

A finite MDP is when there are finite states, finite rewards, and finite actions. In RL, we consider only the finite MDP.

Markov Process:
Markov Process is a memoryless process with a sequence of random states S_1, S_2, ..., S_t that uses the Markov Property. Markov process is also known as Markov chain, which is a tuple (S, P) on state S and transition function P. These two components (S and P) can define the dynamics of the system.

Reinforcement Learning Algorithms
Reinforcement learning algorithms are mainly used in AI applications and gaming applications. The main used algorithms are:

- **Q-Learning:**
  - Q-learning is an **Off policy RL algorithm**, which is used for the temporal difference Learning. The temporal difference learning methods are the way of comparing temporally successive predictions.
  - It learns the value function Q (S, a), which means how good to take action "**a**" at a particular state "**s**."
  - The below flowchart explains the working of Q- learning:



- **State Action Reward State action (SARSA):**
    - SARSA stands for **State Action Reward State action**, which is an **on-policy** temporal difference learning method. The on-policy control method selects the action for each state while learning using a specific policy.
    - The goal of SARSA is to calculate the **Q π (s, a) for the selected current policy π and all pairs of (s-a).**
    - The main difference between Q-learning and SARSA algorithms is that **unlike Q-learning, the maximum reward for the next state is not required for updating the Q-value in the table.**
    - In SARSA, new action and reward are selected using the same policy, which has determined the original action.
    - The SARSA is named because it uses the quintuple **Q(s, a, r, s', a').** Where,
      s: original state
      a: Original action
      r: reward observed while following the states
      s' and a': New state, action pair.
- **Deep Q Neural Network (DQN):**
    - As the name suggests, DQN is a **Q-learning using Neural networks**.
    - For a big state space environment, it will be a challenging and complex task to define and update a Q-table.
    - To solve such an issue, we can use a DQN algorithm. Where, instead of defining a Q-table, neural network approximates the Q-values for each action and state.

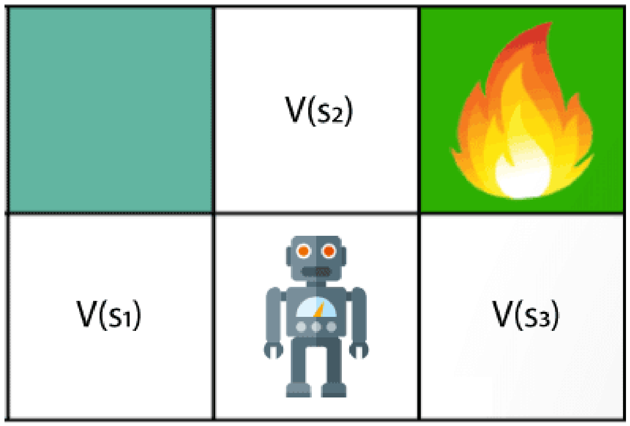
Now, we will expand the Q-learning.

Q-Learning Explanation:

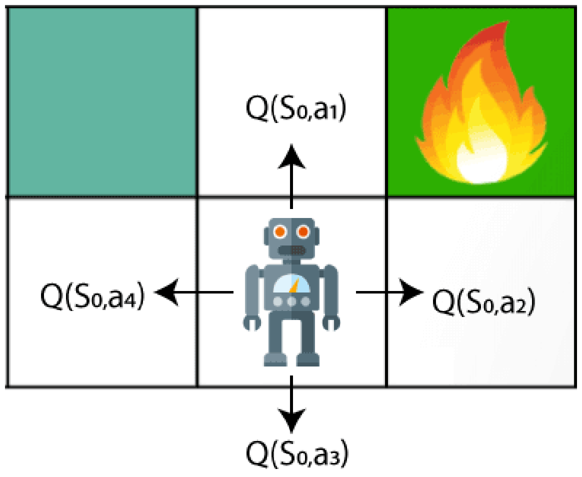
- Q-learning is a popular model-free reinforcement learning algorithm based on the Bellman equation.
- **The main objective of Q-learning is to learn the policy which can inform the agent that what actions should be taken for maximizing the reward under what circumstances.**
- It is an **off-policy RL** that attempts to find the best action to take at a current state.
- The goal of the agent in Q-learning is to maximize the value of Q.
- The value of Q-learning can be derived from the Bellman equation. Consider the Bellman equation given below:

$$V(s) = \max \left[R(s,a) + \gamma \textstyle\sum_{s'} P(s,a,s')V(s`)\right]$$

In the equation, we have various components, including reward, discount factor (γ), probability, and end states s'. But there is no any Q-value is given so first consider the below image:



In the above image, we can see there is an agent who has three values options, $V(s_1)$, $V(s_2)$, $V(s_3)$. As this is MDP, so agent only cares for the current state and the future state. The agent can go to any direction (Up, Left, or Right), so he needs to decide where to go for the optimal path. Here agent will take a move as per probability bases and changes the state. But if we want some exact moves, so for this, we need to make some changes in terms of Q-value. Consider the below image:



Q- represents the quality of the actions at each state. So instead of using a value at each state, we will use a pair of state and action, i.e., Q(s, a). Q-value specifies that which action is more lubricative than others, and according to the best Q-value, the agent takes his next move. The Bellman equation can be used for deriving the Q-value.

To perform any action, the agent will get a reward R(s, a), and also he will end up on a certain state, so the Q -value equation will be:

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

Hence, we can say that, ***V(s) = max [Q(s, a)]***

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

The above formula is used to estimate the Q-values in Q-Learning.

What is 'Q' in Q-learning?

The Q stands for **quality** in **Q-learning**, which means it specifies the quality of an action taken by the agent.

Q-table:
A Q-table or matrix is created while performing the Q-learning. The table follows the state and action pair, i.e., [s, a], and initializes the values to zero. After each action, the table is updated, and the q-values are stored within the table.

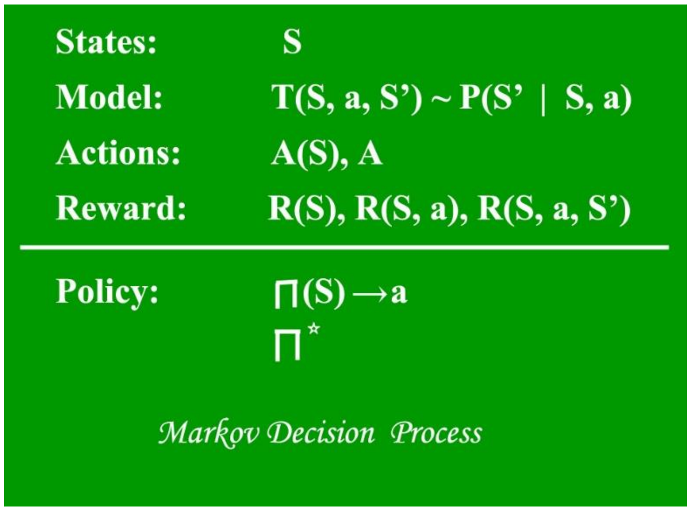
The RL agent uses this Q-table as a reference table to select the best action based on the q-values.

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 is a solution to **Markov Decision Process**.



States:	**S**
Model:	**$T(S, a, S') \sim P(S' \mid S, a)$**
Actions:	**$A(S)$, A**
Reward:	**$R(S)$, $R(S, a)$, $R(S, a, S')$**
Policy:	**$\Pi(S) \rightarrow a$** **Π^***

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 the 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 are Actions?

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

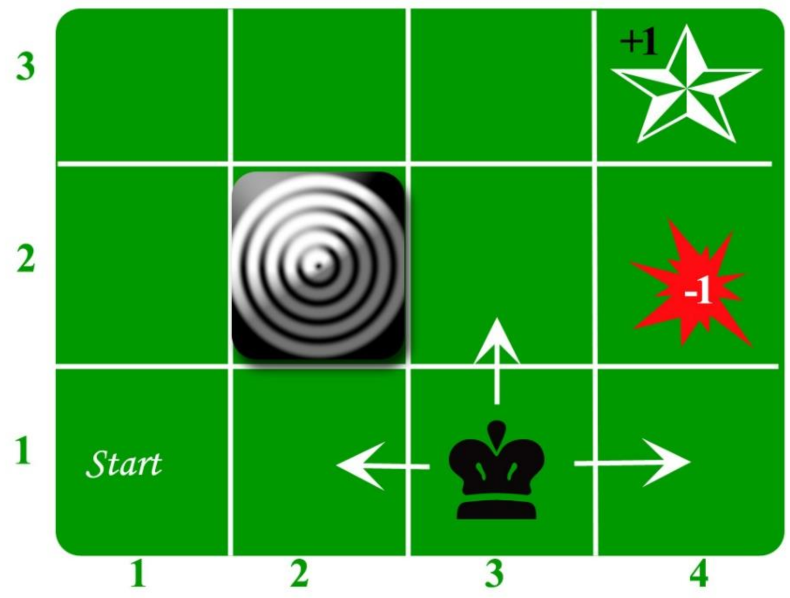
What is a Reward?

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 as 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's 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 UPRIGHT**
- **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 the probability of going RIGHT is 0.1 (since LEFT and RIGHT are right angles to UP).

The agent receives rewards for each time step:-

- Small reward for 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 the sum of rewards.

1. Introduction

A **Markov Decision Process (MDP)** is a mathematical model used to describe an environment in **Reinforcement Learning (RL)**. It provides a formal way to model decision-making in situations where outcomes are partly under the control of the agent and partly random.

It is based on the **Markov Property**, which assumes that the future state depends only on the **current state and action**, not on the past sequence of states.

---

2. Markov Property

The Markov Property is the **memoryless property** of a stochastic process. It is defined as:

The probability of moving to the next state depends only on the current state and action.

P(st+1|st,at)=P(st+1|s1,a1,...,st,at)

This simplifies the decision-making process since only the current state is needed to make optimal decisions.

---

3. Components of an MDP *(5 marks)*

An MDP is defined by a **5-tuple: (S, A, P, R, γ)**

Component	**Description**
S (States)	Set of all possible states in the environment. Example: grid positions in a maze.
A (Actions)	Set of all possible actions the agent can take in a given state. Example: move up, down, left, right.
P (Transition Probability)	(P(s'
R (Reward Function)	R(s,a,s'): Immediate reward received after transitioning from state s to s' using action a.

Component	Description
γ (Discount Factor)	A value between 0 and 1 that determines how much future rewards are considered. A higher γ values future rewards more.

4. Objective of MDP

The goal of an MDP is to find an **optimal policy** $\pi*$, which tells the agent what action to take in each state to **maximize the total expected reward over time**.

The cumulative reward (also called **return**) is defined as:

Gt=Rt+1+γRt+2+γ2Rt+3+…

Where:

- Gt is the return
- R is the reward
- γ is the discount factor

5. Example: Grid World

Consider a robot in a 4x4 grid world:

- **States:** Each grid cell (e.g., (0,0), (2,3))
- **Actions:** Move up, down, left, or right
- **Transition:** 80% chance moves as intended, 10% moves sideways
- **Reward:** +10 for reaching goal cell, -1 per step
- **Objective:** Learn the shortest path with maximum total reward

This shows how MDP helps the agent make decisions despite uncertainty.

6. Applications of MDP

MDPs are widely used in:

- Game AI (e.g., chess, Go)
- Robotics and navigation
- Self-driving cars
- Industrial automation
- Conversational agents and recommendation systems

7. Conclusion

Markov Decision Processes form the **foundation of Reinforcement Learning**. They enable agents to make smart, reward-driven decisions by modeling environments using states, actions, transitions, and rewards.

1. Introduction

The **Bellman Equation** is a fundamental concept in **Reinforcement Learning (RL)** and **Dynamic Programming (DP)**. It provides a recursive decomposition of the value function, helping to **evaluate the long-term return** of actions taken in a stochastic environment.

The equation is named after **Richard Bellman**, who introduced it in the context of dynamic programming.

The Bellman Equation is used to:

- **Evaluate a policy** (how good a given policy is),
- **Improve a policy**, and
- **Find the optimal policy** (one that maximizes the cumulative reward).

It is built upon the **Markov Decision Process (MDP)** framework, which models environments using states, actions, transition probabilities, rewards, and discount factors.

2. Value Functions

To understand the Bellman Equation, we first define two types of **value functions**:

(a) State Value Function Vπ(s):

Expected return when starting from state s and following policy π:

$V\pi(s)=E\pi[\sum t=0\infty\gamma tRt+1 \mid So=s]$

(b) Action Value Function (Qπ(s,a):

Expected return when starting from state s, taking action a, and then following policy π:

$Q\pi(s,a)=E\pi[\sum t=0\infty\gamma tRt+1 \mid S0=s,A0=a]$

Where:

- Rt+1: Reward received at time t+1,

- γ: Discount factor ($0 \leq \gamma \leq 1$),
- π: Policy function (decision rule).

---

3. Bellman Expectation Equations (For Evaluation)

These equations are used to evaluate how good a policy π is.

🧾 *State Value Function:*
$V^\pi(s)=\sum_a \pi(a|s)\sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma V^\pi(s')]$

Meaning:
The value of a state under a policy = expected reward of each action + discounted value of the next state.

🧾 *Action Value Function:*
$Q^\pi(s,a)=\sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma \sum_{a'} \pi(a'|s')Q^\pi(s',a')]$

Meaning:
The value of taking an action from a state = immediate reward + expected future rewards from the next state.

---

4. Bellman Optimality Equations (For Improvement)

These equations help in **finding the best policy** (π^*) by computing the **optimal value function**.

✰ *Optimal State Value Function:*
$V^*(s)=\max_a \sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma V^*(s')]$

✰ *Optimal Action Value Function:*
$Q^*(s,a)=\sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma \max_{a'} Q^*(s',a')]$

These are the foundation for algorithms like **Value Iteration**, **Q-Learning**, and **Deep Q-Networks (DQN)**.

---

5. Intuition and Example

🔁 *Recursive Insight:*

The Bellman Equation breaks down the value of a decision into two parts:

1. The **immediate reward**.
2. The **value of the next state**, assuming we behave optimally or follow a given policy from then on.

📌 *Example (Grid World):*

- A robot moves in a 4×4 grid.
- It receives -1 for every step, +10 for reaching the goal.
- Actions: up, down, left, right.
- The environment is stochastic (80% chance it moves as intended, 20% otherwise).

To calculate the best action at each grid cell, the Bellman Equation is used to **update value estimates** iteratively until they converge.

---

6. Applications of Bellman Equation

The Bellman Equation is used in:

- **Value Iteration** and **Policy Iteration** (Dynamic Programming)
- **Q-Learning**
- **Deep Reinforcement Learning** (e.g., Deep Q-Networks)
- **Robot navigation**
- **Game AI** (e.g., AlphaGo, Chess engines)

---

7. Conclusion

The Bellman Equation is a core mathematical tool in Reinforcement Learning. It enables an agent to learn from its environment by updating its value estimates and gradually converging towards an optimal policy. Its recursive nature makes it highly efficient and applicable in many real-world sequential decision-making problems.

Bellman Equation

Bellman Equation is based on the principle of optimality, which states that the optimal value of a state can be computed based on the immediate reward and the optimal value of the next state.

1. Bellman Equation for State Value Function

State value function, denoted as V(s) under a given policy, represents the expected cumulative reward when starting in state s and the following policy thereafter:

$$V_\pi(s)=E[R(s,a)+\gamma V_\pi(s')]$$

Expanding this equation with transition probabilities, we get:

$$V\pi(s)=\sum a\in A\pi(a|s)\sum s'\in SP(s'|s,a)[R(s,a)+\gamma V\pi(s')]$$

where:

- $V_\pi(s)$: Value function of state **s** under policy .
- $P(s'|s,a)$: Transition probability from state s to state s′ when taking action a.
- $R(s,a)$: Reward obtained after taking action a in state s.

- γ: Discount factor controlling the importance of future rewards.
- $\pi(a|s)$: Probability of taking action a in state s under policy .

2. Bellman Equation for Action Value Function (Q-function)

Q-function (Q(s,a)) represents the expected return for taking action a in state **s** and following policy afterward:

$$Q\pi(s,a)=E[R(s,a)+\gamma V\pi(s')]$$

Expanding it using transition probabilities:

$$Q\pi(s,a)=\sum s'\in SP(s'|s,a)[R(s,a)+\gamma\sum a'\pi(a'|s')Q\pi(s',a')]$$

This equation helps compute the expected future rewards based on both current action a*a* and subsequent policy actions.

Bellman Optimality Equations

For an optimal policy $\pi*$, the Bellman equation becomes:

1. Optimal State Value Function

$$V*(s)=\max a\sum s'P(s'|s,a)[R(s,a)+\gamma V*(s')]$$

2. Optimal Action Value Function (Q-Learning)

$$Q*(s,a)=\sum s'P(s'|s,a)[R(s,a)+\gamma\max a'Q*(s',a')]$$

These equations form the foundation for Dynamic Programming, Temporal Difference (TD) Learning, and Q-Learning.

Solving MDPs with Bellman Equations

Markov Decision Process can be solved using Dynamic Programming (DP) methods that rely on Bellman Equations:

1. **Value Iteration**: Uses Bellman Optimality Equation to iteratively update value functions until convergence.

2. **Policy Iteration**: Alternates between policy evaluation (solving Bellman Expectation Equation) and policy improvement (updating policy based on new value function).

3. **Q-Learning**: Uses the Bellman Optimality Equation for Q-values to learn optimal policies.

Example: Navigating a Maze

Consider a maze as our environment, where an agent's goal is to reach the trophy state (reward **R = 1**) while avoiding the fire state (reward **R = -1**). The agent receives positive reinforcement for reaching the goal and negative reinforcement for failing. The agent must navigate the maze efficiently while considering possible future rewards.

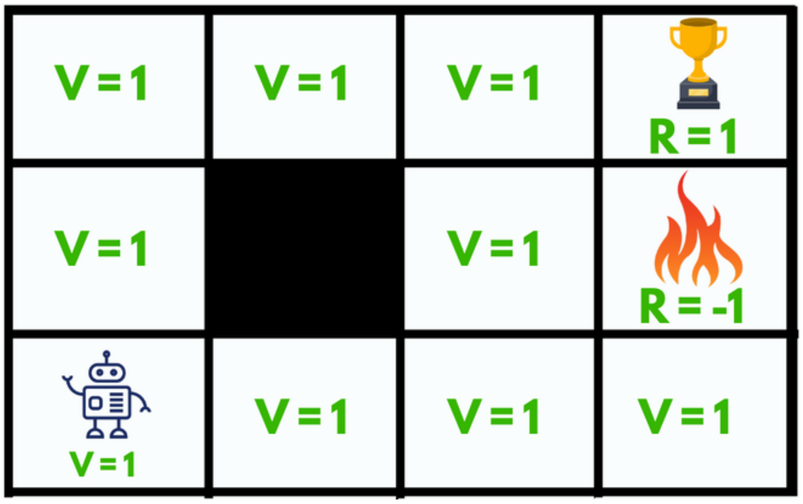


Fig: Without Bellman Equation

What Happens Without the Bellman Equation?
Initially, we allow the agent to explore the environment and find a path to the goal. Once it reaches the trophy state, it backtracks to its starting position and assigns a value of **V = 1** to all states that lead to the goal.
However, if we change the agent's starting position, it will struggle to find a new path since all previously learned state values remain the same. This is where the Bellman Equation helps by dynamically updating state values based on future rewards.

Applying the Concept
Consider a state adjacent to the fire state, where V = 0.9. The agent can move UP, DOWN, or RIGHT but cannot move LEFT due to a wall.
Among the available actions, the agent selects the action leading to the maximum value, ensuring the highest possible reward over time.

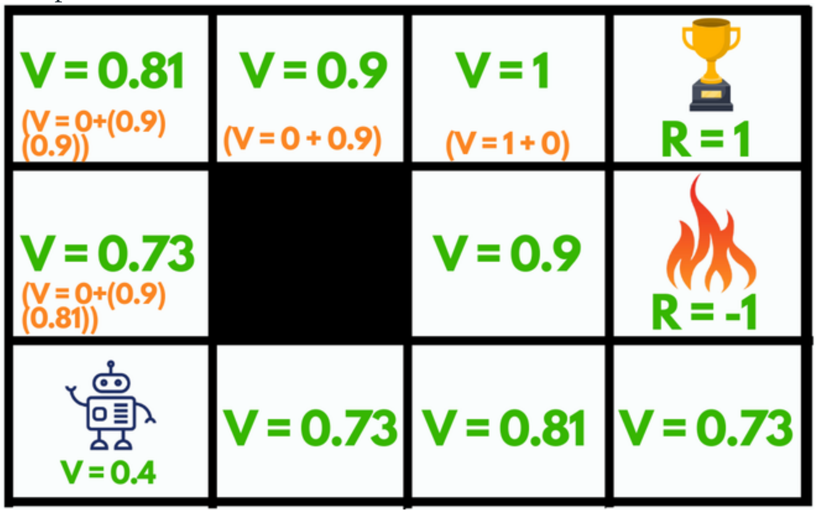


Fig: Using Bellman Equation

By continuously updating state values, the agent systematically calculates the best path while avoiding the fire state. The goal (trophy) and failure (fire) states do not require value updates as they represent terminal states (V = 0).

Understanding the Bellman Equation in Reinforcement Learning

The Bellman Equation is a key concept in reinforcement learning that helps agents make decisions in complex situations by assessing possible future states and rewards. This article examines its mathematical principles, real-world uses, and importance in creating optimal policies within Markov Decision Processes.

In the early 1950s, Richard Bellman, a visionary mathematician and computer scientist, embarked on a journey to unravel the complexities of decision-making in uncertain environments. Driven by the need for efficient problem-solving methods, he introduced the revolutionary concept of dynamic programming, which breaks down intricate problems into manageable subproblems.

As he explored this approach, Bellman formulated what would later be known as the Bellman Equation: ***a mathematical expression that evaluates the value of a decision in a given state based on the potential future states and their associated rewards.*** His groundbreaking work culminated in the publication of his 1957 book, *Dynamic Programming*, which laid the theoretical foundation for optimization and control theory while paving the way for advancements in artificial intelligence and reinforcement learning.

Today, Bellman's ideas continue to shape how machines learn and adapt to their environments, leaving a lasting legacy in mathematics and computer science. In reinforcement learning, the Bellman Equation is crucial in understanding how agents interact with their surroundings, making it essential for solving various problems in Markov Decision Processes (MDPs).

In this article, we will explore what the equation is, how it compares to the Bellman Optimality Equation, how it's used in reinforcement learning, and more.

What is the Bellman Equation?

The Bellman Equation is a recursive formula used in decision-making and reinforcement learning. It shows how the value of being in a certain state depends on the rewards received and the value of future states.

In simple terms, the Bellman Equation breaks down a complex problem into smaller steps, making it easier to solve. The equation helps find the best way to make decisions when outcomes depend on a series of actions.

Mathematically, the Bellman Equation is written as:

$$V(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s') \right]$$

Where:

- *V(s)*: The value of state sss, which represents the long-term reward of being in that state.
- *R(s,a)*: The immediate reward received for taking action aaa in state sss.
- *γ:* The discount factor (between 0 and 1) that determines the importance of future rewards compared to immediate rewards.
- *P(s′/s,a)* The probability of transitioning to state s′ from state sss by taking action *a*.
- *maxa*: The optimal action that maximizes the expected value of future rewards.

This sum multiplies the value of each possible next state by the probability of reaching that state, effectively capturing the expected future rewards.

Bellman Equation vs. Bellman Optimality Equation

The Bellman Equation calculates the value of a state based on the rewards from taking actions and the expected value of future states. It’s a key part of evaluating how good a policy is when an agent follows it in an environment. The equation looks at all the possible actions and outcomes and helps determine the value of being in a certain state.

In contrast, the Bellman Optimality Equation is used when the goal is to find the best or "optimal" policy.

Instead of just evaluating the value of a state based on a specific policy, it finds the maximum value achievable by considering all possible actions and policies. Bellman Optimality Equation does this by taking the maximum expected reward over all actions available in each state. This way, it tells us the highest possible value we can get from each state if we act in the best way possible.

In other words, the main difference between the two equations is their focus.

The standard Bellman Equation evaluates the effectiveness of a policy by estimating the expected value of following it. On the other hand, the Bellman Optimality Equation aims to find the best policy overall by identifying the action that gives the maximum long-term reward in each state.

TLDR: The standard Bellman Equation is used for policy evaluation, while the optimality equation is used for policy improvement and control.

The Bellman Optimality Equation is essential for solving control problems, where the objective is to choose the best sequence of actions to maximize rewards. In these problems, agents must make decisions that maximize immediate rewards and consider future rewards.

The optimality equation provides a way to systematically evaluate and choose actions that lead to the highest possible long-term reward, helping to find the optimal policy. It is a central part of algorithms like Q-learning and value iteration, which are used in reinforcement learning to train agents to act in the best way possible.

Bellman Equation in Reinforcement Learning

The Bellman Equation is a core part of reinforcement learning. It helps agents learn how to make the best decisions by breaking down complex tasks into smaller steps.

Namely, the Bellman Equation enables agents to calculate the value of different states and actions, which guides them in choosing the best path to maximize rewards.

Thus, In reinforcement learning, the Bellman Equation is used to evaluate and improve policies, ensuring agents make better decisions as they gain experience.

Role in Q-Learning

Q-learning is a popular reinforcement learning algorithm. The algorithm leverages the Bellman Equation to estimate the value of state-action pairs, known as Q-values. These values tell the agent how rewarding it will be to take a certain action in a given state and follow the best policy afterward.

The Bellman Optimality Equation for Q-learning finds the maximum expected value for each action, helping agents identify the best moves to maximize their long-term rewards.

For example, in an episodic reinforcement learning problem, imagine an agent navigating a grid world. When it reaches a state, the agent updates its Q-value for that state-action pair using the equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- ***Q(s,a)***: The Q-value for taking action *a* in state *s*.
- **α**: The learning rate, which determines how much new information overrides the old Q-value.
- ***R(s,a)***: The immediate reward received after taking action *a* in state *s*.

- ***γ:*** The discount factor, which weighs future rewards relative to immediate rewards.
- ***$\max_{a'} Q(s',a')$***: The maximum Q-value for the next state *s'*, across all possible actions *a'*.

  This equation updates the Q-value based on the immediate reward and the maximum expected future rewards. Through repeated application, the agent learns to approximate the optimal Q-values and can determine the best actions to maximize long-term rewards.

Bellman Equation for policy evaluation

The Bellman Equation is also used to evaluate policies by estimating value functions. In this context, it calculates the expected long-term reward when an agent follows a given policy from a specific state. This helps determine how effective a policy is by showing the average reward the agent can expect.

The policy iteration algorithm uses this evaluation to improve policies. It starts by estimating the value function for a policy and then updates the policy to choose better actions based on this value. This process repeats until the policy converges to the optimal one, at which point no further improvements can be made.

Bellman Equation Derivation

In this section, we'll break down the Bellman Equation step by step, starting from the first principles. We will then explore how rewards and transitions build value functions, guiding agents in choosing the best actions.

Step-by-step derivation

To derive the Bellman Equation, we start with the concept of a value function. The value function, V(s)V(s)V(s), gives the expected cumulative reward for starting in a state sss and following an optimal policy thereafter. This equation is recursive, meaning it relates the value of a state to the value of subsequent states.

1. Define the value function

The value function ***V(s)*** represents the total expected reward an agent will receive from starting in state sss and following the optimal policy. Mathematically, it can be expressed as:

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s\right]$$

Where:

- ***s*** is the current state,

- ***γ*** is the discount factor (between 0 and 1), which controls the importance of future rewards,
- ***R*(s_t)** is the reward received at each time step *t*.

2. Breaking down future rewards

To express *V(s)* in terms of immediate rewards and the value of future states, we separate the first reward from the sum of all future rewards:

$$V(s) = \mathbb{E}\left[R(s,a) + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} R(s_t)\right]$$

Where:

- ***R(s, a)*** is the immediate reward obtained by taking action ***a*** in state ***s***,
- ***γ Σ*** $(\gamma^{(t}1))$ ***R*(s_t)** represents the discounted sum of future rewards.

3. Incorporate the next state value

Now we add the value of the next state, ***s'***, which the agent reaches after taking action ***a*** in state ***s***. This leads to the equation:

$$V(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')\right]$$

Where:

- ***P(s' | s, a)*** is the probability of transitioning to state ***s'*** from state ***s*** after taking action ***a***,
- ***max$_a$*** represents the choice of the action ***a*** that maximizes the expected reward.

4. The Final Bellman Equation

After simplifying, the Bellman Equation for the value function ***V(s)*** becomes:

$$V(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')\right]$$

This final form of the Bellman Equation decomposes the value of a state into two components: the immediate reward and the expected value of the next state. This recursive structure allows algorithms to compute or approximate ***V(s)*** iteratively, which helps in finding optimal policies over time.

Examples and applications

Let's walk through an example with a simple grid world:

Imagine an agent trying to reach a goal on a grid while avoiding obstacles. The agent moves up, down, left, or right and gets rewards for each move. If it reaches the goal, it gets a higher reward. Using the Bellman Equation, we can calculate the value for each state on the grid by evaluating the immediate rewards and future state values.

For Markov Decision Processes (MDPs), the Bellman Equation helps find the optimal policy. If we have a small MDP, like a delivery problem where an agent must pick the fastest route, we apply the equation to each state-action pair. By doing so, we can calculate the best actions that maximize the total expected reward.

Advanced Topics: Bellman Equation in Q-Learning

In this section, we explore how the Bellman Equation is applied in advanced methods like:

- Value iteration
- Temporal-Difference learning.

These techniques help agents learn optimal actions by updating value estimates, even in complex environments. Understanding these advanced topics is key to mastering reinforcement learning.

Value iteration

Value iteration is a method that uses the Bellman Equation to update value functions iteratively. The goal is to find the optimal value for each state so that the agent can determine the best actions.

In value iteration, the agent updates each state's value based on possible future states and their rewards. This process repeats until the values converge to a stable solution. The convergence properties of value iteration ensure that, given enough time, it will find the optimal policy.

Temporal-difference learning

TD learning is another approach that uses the Bellman Equation but differs in a key way—it doesn't need a model of the environment.

Instead, it updates value estimates based on actual experiences the agent gathers. This makes it more practical when the environment is unknown. TD learning bridges the gap between Monte Carlo methods, which rely on complete episodes, and dynamic programming, which needs a full model.

Applications and Use Cases

The Bellman Equation has a wide range of applications beyond theoretical models. It plays a crucial role in real-world scenarios where decision-making and optimization are needed. This section explores how the Bellman Equation supports these applications.

Reinforcement learning in games

In classic gaming environments like Atari, the Bellman Equation helps optimize an agent's actions to achieve the highest score. The equation guides the agent in selecting the best moves by evaluating the expected rewards for each action.

Real-world examples include using deep reinforcement learning algorithms, like Deep Q-Networks (DQNs), that use the Bellman Equation to improve game AI. These methods have been used to train agents that outperform human players in various games, showing the power of reinforcement learning.

Robotics and navigation

In robotics, the Bellman Equation helps machines make decisions about movement and navigation. Robots use it to evaluate different paths and choose the one that maximizes future rewards, such as reaching a goal in the shortest time or with the least energy.

In other words, robots use the Bellman equation to help them predict outcomes based on their current state and plan their next moves accordingly. This decision-making process is critical for navigating through a maze or avoiding obstacles in dynamic environments.

Conclusion

The Bellman Equation is fundamental in developing intelligent systems – particularly in reinforcement learning and other areas of artificial intelligence. It serves as the basis for algorithms that allow systems to learn, adapt, and optimize their actions in dynamic environments, which leads to more efficient and effective problem-solving in uncertain environments.

Monte Carlo Policy Evaluation

Monte Carlo policy evaluation is a technique within the field of reinforcement learning that estimates the effectiveness of a policy—a strategy for making decisions in an environment. It’s a bit like learning the rules of a game by playing it many times, rather than studying its manual. This approach doesn't require a pre-built model of the environment; instead, it learns exclusively from the outcomes of the episodes it experiences. Each episode consists of a sequence of states, actions, and rewards, much like playing rounds of a game, starting from the initial state and continuing until the game ends.

How Monte Carlo Policy Evaluation Works?

The method works by running simulations or episodes where an agent interacts with the environment until it reaches a terminal state. At the end of each episode, the algorithm looks back at the states visited and the rewards received to calculate what's known as the "return" — the cumulative reward starting from a specific state until the end of the episode. Monte Carlo policy evaluation repeatedly simulates episodes, tracking the total rewards that follow each state and then calculating the average. These averages give an estimate of the state value under the policy being followed.
By aggregating the results over many episodes, the method converges to the true value of each state when following the policy. These values are useful because they help us understand which states are more valuable and thus guide the agent toward better decision-making in the future. Over time, as the agent learns the value of different states, it can refine its policy, favouring actions that lead to higher rewards.

Concepts Related to Monte Carlo Policy Evaluation:
Monte Carlo policy evaluation is like a trial-and-error learning method where you understand the value of actions by repeatedly trying them and observing the outcomes. Imagine you're in a maze and each move either gets you closer to the exit or takes you to a dead end. If you try many different paths, over time, you'll learn which turns are likely to be dead ends and which ones lead to the exit.
In reinforcement learning, each complete walkthrough of the maze is an "episode," and the Monte Carlo method uses many such episodes to figure out how good or bad it is to be in a certain spot in the maze (a "state"). After many walkthroughs, you start to notice patterns: some spots consistently lead to quick exits, so they are given a high value; others tend to lead to dead ends, so they are valued lower.
The Monte Carlo method waits until the end of the episode, then works backwards to assign a value to each state based on the rewards collected. It doesn't make assumptions about the environment or use complex models; it learns purely from experience. By averaging the total rewards that follow each state, it can estimate the state's value, guiding you to make better decisions in the maze in future runs.

Mathematical Concepts in Monte Carlo Policy Evaluation:
In Monte Carlo policy evaluation, the value V of a state "s" under a policy π is estimated by the average return G following that state. The return is the cumulative reward obtained after visiting state "s":
$V(s)=\frac{1}{N(s)}\sum_{i=1}^{N(s)}G_i$
Here, N(s) is the number of times state "s" is visited across episodes, and Gi is the return from the i-th episode after visiting state "s". This average converges to the expected return as N(s) becomes large:
$V(s)\approx E_\pi[G|S=s]$
Each return Gi is calculated by summing discounted rewards from the time state "s" is visited till the end of the episode:
$G_i=\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}$
where, γ is the discount factor (between 0 and 1) and R is the reward at each time step. This

reflects the idea that rewards in the near future are more valuable than rewards further in the future.

Advantages of Monte Carlo Policy Evaluation:

- **No Model Required:** It doesn't need a model of the environment's dynamics, as it learns directly from experience, making it ideal for complex or unknown environments.
- **Simple Implementation:** The algorithm is straightforward to implement since it averages returns from episodes without requiring intricate mathematical calculations or estimations.
- **Flexible to Variability:** It can handle stochastic policies and environments since it considers a range of possible outcomes through sampling.

Disadvantages of Monte Carlo Policy Evaluation:

- **High Variance:** It can exhibit high variance in estimates since outcomes from different episodes may vary widely, especially with fewer episodes.
- **Inefficiency with Long Episodes:** It becomes less efficient with long episodes or delayed rewards, as it must wait until the end of an episode to update values.
- **Lack of Bootstrap:** Unlike other methods, it does not bootstrap (update estimates based on other estimates), which can slow down the learning process in large state spaces.

Conclusion

In conclusion, Monte Carlo policy evaluation is like learning through full experience. It's a hands-on way to measure how effective certain actions are, based on the rewards they yield over many trials. While it's not perfect and can be a bit slow, it's a practical approach, especially when we're stepping into new territory without a guide.

Q. Explain Policy Evaluation using Monte Carlo, Policy Iteration, Value Iteration, Q-Learning, and SARSA.

---

1. Introduction *(1 mark)*

In **Reinforcement Learning (RL)**, **policy evaluation** refers to calculating the **value function** for a given policy π\piπ, i.e., determining how good it is to follow π\piπ from each state.

Different methods are used to evaluate and improve policies, including **model-based methods** (like Policy Iteration and Value Iteration) and **model-free methods** (like Monte Carlo, Q-Learning, and SARSA).

---

2. Monte Carlo Methods *(3 marks)*

➤ Concept:

Monte Carlo (MC) methods estimate the value of a policy by **sampling episodes** (complete sequences from start to terminal state) and averaging returns.

➤ **Key Characteristics:**

- Works only for **episodic tasks**.
- No knowledge of transition probabilities is needed (model-free).
- **Returns** are calculated as:

  $G_t=R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\dots$

- **First-Visit MC**: Averages returns from the first time a state is visited.
- **Every-Visit MC**: Averages returns every time the state is visited.

➤ **Use in Policy Evaluation:**

$V(s)=1/N(s)\sum_{i=1}$ to $N(s)$ G_i

Where $N(s)$is the number of times state sss is visited.

---

3. Policy Iteration (Dynamic Programming)

➤ **Concept:**

Alternates between two steps:

1. **Policy Evaluation**: Compute $V\pi(s)$ for current policy π
2. **Policy Improvement**: Update policy by choosing actions that maximize expected return

➤ **Algorithm:**

- **Initialize** a random policy
- **Repeat** until convergence:
  1. Evaluate $V\pi$ using Bellman expectation equation
  2. Improve π using:

     $\pi(s)=\arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma V\pi(s')]$

➤ **Converges to: Optimal policy $\pi*$**

---

4. Value Iteration (Dynamic Programming)

➤ **Concept:**

Simplified version of policy iteration that **combines evaluation and improvement** into one step using the Bellman Optimality Equation.

➤ **Algorithm:**

- **Initialize** $V(s)$ arbitrarily
- **Repeat** until convergence:

  $V(s)=\max_a \sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma V(s')]$

  After convergence, derive policy:

  $\pi(s)=\arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s')+\gamma V(s')]$

➤ **Faster than full policy evaluation + improvement cycles.**

---

5. Q-Learning (Off-Policy, Model-Free) *(2 marks)*

➤ **Concept:**

Learns the **optimal action-value function** $Q*(s,a)$ by interacting with the environment and updating estimates using the **Bellman Optimality Equation**.

➤ **Update Rule:**

$Q(s,a)\leftarrow Q(s,a)+\alpha[R+\gamma \max_{a'} Q(s',a')-Q(s,a)]$

➤ **Key Features:**

- **Off-policy**: Learns optimal policy even if actions are chosen using a different behavior policy (e.g., ε-greedy).
- Doesn't require a model of the environment.

➤ **Used for: Optimal control and policy evaluation/improvement simultaneously.**

---

6. SARSA (On-Policy, Model-Free)

➤ **Concept:**

SARSA stands for **State-Action-Reward-State-Action**. Unlike Q-learning, it uses the **action actually taken**, not the best possible action.

➤ **Update Rule:**

$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]$

➤ **Key Features:**

- **On-policy**: Evaluates and improves the policy being followed.
- Safer in risky environments (more conservative than Q-learning).

➤ **Policy Evaluation: Learns $Q\pi(s,a)$ for the current policy.**

---

7. Comparison Table

Method	Model-Free	On/Off Policy	Episodic?	Evaluation Style
Monte Carlo	Yes	On-policy	Yes	Average of returns
Policy Iteration	No	On-policy	No	Bellman expectation
Value Iteration	No	On-policy	No	Bellman optimality
Q-Learning	Yes	**Off-policy**	No	Max future value
SARSA	Yes	**On-policy**	No	Value of next action

---

8. Conclusion

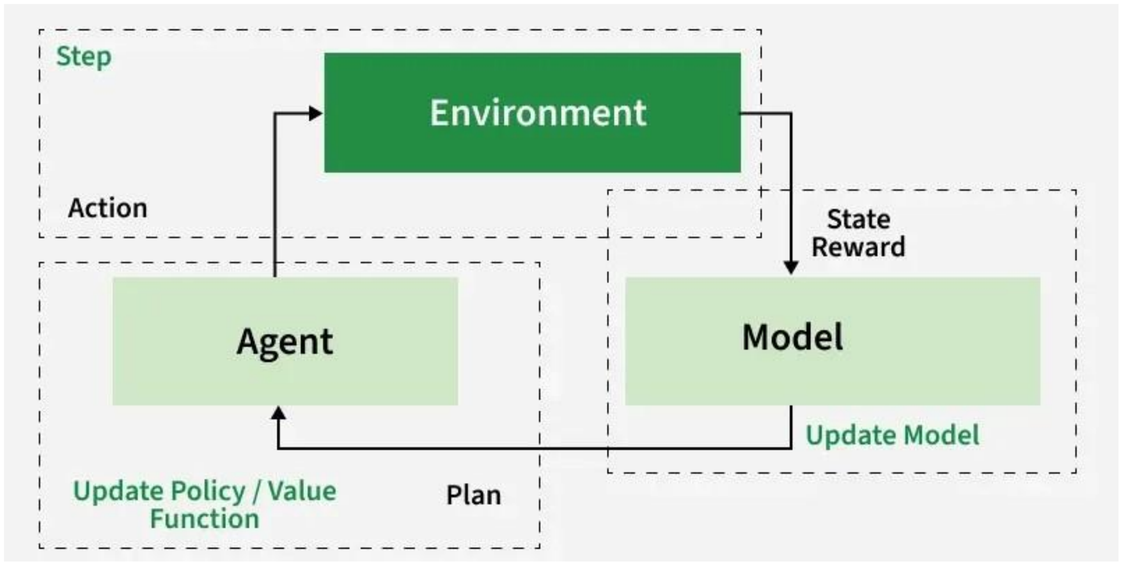
Policy evaluation is a core part of reinforcement learning, helping agents assess how good their current behavior is. Based on the task, we choose **Monte Carlo for episodic tasks**, **DP methods for known environments**, and **Q-Learning/SARSA for online learning without a model**.

Model-Based Reinforcement Learning (MBRL)

Model-based reinforcement learning is a subclass of reinforcement learning where the agent constructs an internal model of the environment's dynamics and uses it to simulate future states, predict rewards, and optimize actions efficiently.

Key Components of MBRL

- **Model of the Environment:** This is typically a predictive model, which can forecast the next state and rewards given the current state and action.
- **Planning Algorithm:** Once a model is learned, a planning algorithm evaluates the model to decide the optimal sequence of actions.



How Does MBRL Work?

MBRL follows these steps:

1. **Model Learning:** The agent collects experience by interacting with the environment and then uses these experiences to learn a model that predicts future states and rewards.
2. **Model-Based Planning:** The learned model is then used to simulate future interactions without needing to interact with the real environment. Planning algorithms like Monte Carlo Tree Search (MCTS) or Dynamic Programming can be used to identify optimal actions.
3. **Policy Optimization:** The agent uses the results from planning to optimize its policy, which is then deployed back into the real environment.
4. **Continuous Learning:** The model is updated regularly as the agent gathers new experiences, improving the model's accuracy and the agent's performance over time.

Popular Approaches in Model-Based Reinforcement Learning

Several popular MBRL methods have emerged in recent years:

1. World Models

- **Description**: This approach involves creating a model that simulates the environment's dynamics using past experiences. It often employs recurrent neural networks (RNNs) to learn the transition function f(s,a)$f(s,a)$ from state ss and action aa.
- **Example**: The "World Models" framework allows an agent to learn to play games solely from camera images without predefined state representations.

2. Model Predictive Control (MPC)

- **Description**: MPC uses a model to predict future states and optimize control actions over a finite horizon. It samples trajectories based on the model and selects actions that maximize expected rewards.
- **Example**: The Model Predictive Path Integral (MPPI) method is a stochastic optimal control technique that iteratively optimizes action sequences by sampling multiple trajectories.

3. Dyna Architecture

- **Description**: This approach combines model learning, data generation, and policy learning in an iterative process. The agent learns a model of the environment and uses it to generate synthetic experiences for training.
- **Example**: Sutton's Dyna algorithm alternates between updating the model and refining the policy based on both real and simulated experiences.

4. Sampling-Based Planning

- **Description**: Involves generating candidate action sequences through sampling and evaluating them using the learned model. Techniques like random shooting or cross-entropy methods help refine action selections.
- **Example**: Algorithms such as PlaNet and PETS utilize sampling-based planning for effective decision-making in complex environments.

5. Analytic Gradient Computation

- **Description**: This method leverages assumptions about the dynamics to compute gradients analytically, facilitating local optimal control solutions.
- **Example**: Linear Quadratic Regulator (LQR) frameworks often utilize this approach for efficient control in linear systems.

6. Value-Equivalence Prediction

- **Description**: This technique predicts value functions using models, allowing for better generalization and improved policy updates.
- **Example**: Incorporating model-generated data into reinforcement learning can enhance learning efficiency, although it must be managed carefully to avoid divergence due to modeling errors.

7. Meta-Reinforcement Learning

- **Description**: This approach focuses on learning how to learn, where agents adapt their strategies based on previous experiences across different tasks.
- **Example**: Context-based meta-reinforcement learning utilizes world models to generalize across various environments and tasks.

Advantages of Model-Based RL

MBRL offers several advantages, making it a compelling choice for specific applications:

- **Sample Efficiency:** Since the agent learns and plans using a model, fewer interactions with the environment are required, which is crucial for real-world problems where interactions can be expensive or slow.
- **Better Generalization:** The learned model allows agents to generalize better to unseen situations because they are trained to predict and adapt to new scenarios.
- **Long-Term Planning:** MBRL facilitates long-term planning by allowing agents to simulate long sequences of actions and their outcomes, which is valuable for tasks like robotic control or game-playing.

Challenges in Model-Based RL

While MBRL holds promise, it comes with its own set of challenges:

- **Model Accuracy:** Building an accurate model of a complex environment is challenging. Any inaccuracies in the model can lead to suboptimal policies and degraded performance.
- **Computational Complexity:** MBRL requires both learning the model and planning over it, which can be computationally intensive, especially in large, high-dimensional environments.

- **Exploration vs. Exploitation Dilemma:** The agent may over-rely on its learned model, leading to limited exploration of the environment, which might hinder the discovery of optimal solutions.

As RL continues to grow, model-based methods offer a promising direction for developing intelligent agents that learn faster, plan better, and act efficiently in complex environments.

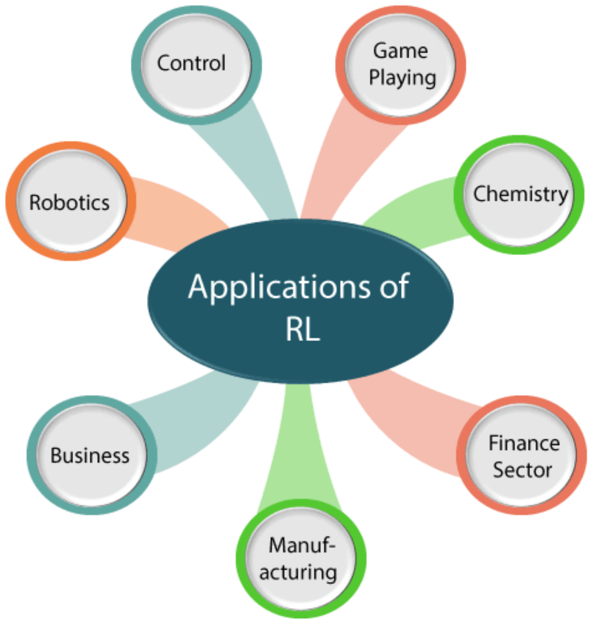
Difference between Reinforcement Learning and Supervised Learning

The Reinforcement Learning and Supervised Learning both are the part of machine learning, but both types of learnings are far opposite to each other. The RL agents interact with the environment, explore it, take action, and get rewarded. Whereas supervised learning algorithms learn from the labeled dataset and, on the basis of the training, predict the output.

The difference table between RL and Supervised learning is given below:

Reinforcement Learning	**Supervised Learning**
RL works by interacting with the environment.	Supervised learning works on the existing dataset.
The RL algorithm works like the human brain works when making some decisions.	Supervised Learning works as when a human learns things in the supervision of a guide.
There is no labeled dataset is present	The labeled dataset is present.
No previous training is provided to the learning agent.	Training is provided to the algorithm so that it can predict the output.
RL helps to take decisions sequentially.	In Supervised learning, decisions are made when input is given.

Reinforcement Learning Applications



1. **Robotics:**
   a. RL is used in **Robot navigation, Robo-soccer, walking, juggling**, etc.
2. **Control:**
   . RL can be used for **adaptive control** such as Factory processes, admission control in telecommunication, and Helicopter pilot is an example of reinforcement learning.
3. **Game Playing:**

. RL can be used in **Game playing** such as tic-tac-toe, chess, etc.

4. **Chemistry:**

. RL can be used for optimizing the chemical reactions.

5. **Business:**

. RL is now used for business strategy planning.

6. **Manufacturing:**

. In various automobile manufacturing companies, the robots use deep reinforcement learning to pick goods and put them in some containers.

7. **Finance Sector:**
   . The RL is currently used in the finance sector for evaluating trading strategies.

Conclusion:

From the above discussion, we can say that Reinforcement Learning is one of the most interesting and useful parts of Machine learning. In RL, the agent explores the environment by exploring it without any human intervention. It is the main learning algorithm that is used in Artificial Intelligence. But there are some cases where it should not be used, such as if you have enough data to solve the problem, then other ML algorithms can be used more efficiently. The main issue with the RL algorithm is that some of the parameters may affect the speed of the learning, such as delayed feedback.