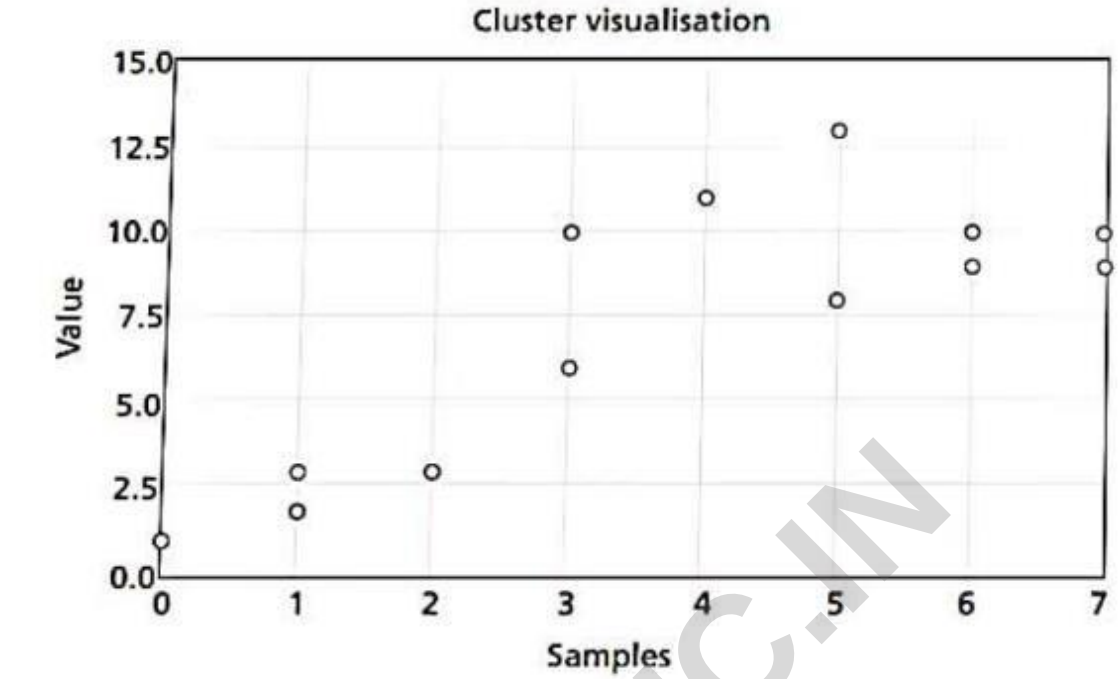
Module-5

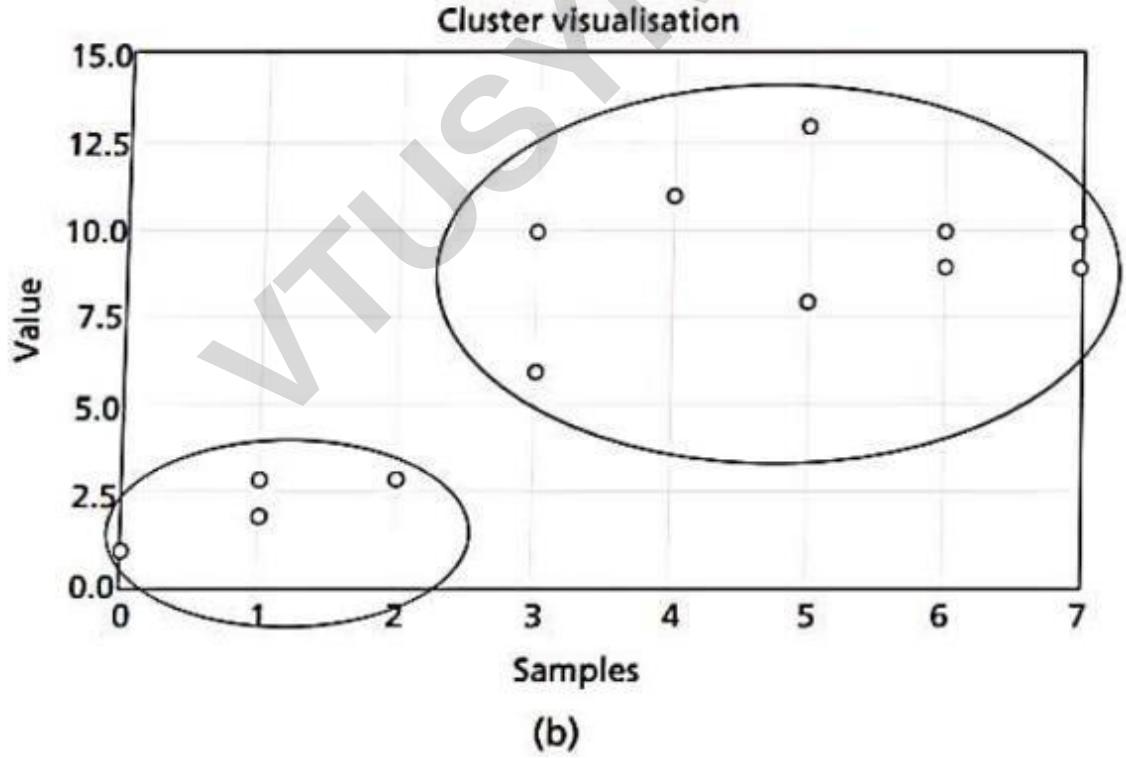
Chapter – 01 - Clustering Algorithms

Introduction to Clustering Approaches

- Cluster analysis is the fundamental task of unsupervised learning. Unsupervised learning involves exploring the given dataset.
- Cluster analysis is a technique of partitioning a collection of unlabelled objects that have many attributes into meaningful disjoint groups or clusters.
- This is done using a trial and error approach as there are no supervisors available as in classification.
- The characteristic of clustering is that the objects in the clusters or groups are similar to each other within the clusters while differ from the objects in other clusters significantly.
- The input for cluster analysis is examples or samples. These are known as objects, data points or data instances.
- All these terms are same and used interchangeably in this chapter. All the samples or objects with no labels associated with them are called unlabelled.
- The output is the set of clusters (or groups) of similar data if it exists in the input.
- For example, the following Figure 13.1(a) shows data points or samples with two features shown in different shaded samples and Figure 13.1(b) shows the manually drawn ellipse to indicate the clusters formed.



(a)



(b)

Figure 13.1: (a) Data Samples (b) Clusters' Description

Visual identification of clusters in this case is easy as the examples have only two features.

But, when examples have more features, say 100, then clustering cannot be done manually and automatic clustering algorithms are required.

Also, automating the clustering process is desirable as these tasks are considered difficult by humans and almost impossible. All clusters are repre- sented by centroids.

Example: For example, if the input examples or data is (3, 3), (2, 6) and (7, 9), then the centroid is given as.

$$\left(\frac{3+2+7, 3+6+9}{3}\right) = (4, 6)$$

The clusters should not overlap and every cluster should represent only one class. Therefore, clustering algorithms use trial and error method to form clusters that can be converted to labels.

Difference between Clustering & Classification

S.No.	Clustering	Classification
1.	Unsupervised learning and cluster formation are done by trial and error as there is no supervisor	Supervised learning with the presence of a supervisor to provide training and testing data
2.	Unlabelled data	Labelled data
3.	No prior knowledge in clustering	Knowledge of the domain is must to label the samples of the dataset
4.	Cluster results are dynamic	Once a label is assigned, it does not change

Applications of Clustering

1. Grouping based on customer buying patterns
2. Profiling of customers based on lifestyle
3. In information retrieval applications (like retrieval of a document from a collection of documents)
4. Identifying the groups of genes that influence a disease
5. Identification of organs that are similar in physiology functions
6. Taxonomy of animals, plants in Biology
7. Clustering based on purchasing behaviour and demography
8. Document indexing
9. Data compression by grouping similar objects and finding duplicate objects

Challenges of Clustering Algorithms

High-Dimensional Data

- As the number of features increases, clustering becomes difficult.

Scalability Issue

- Some algorithms perform well for **small datasets** but fail for **large-scale data**.

Unit Inconsistency

- Different measurement units (e.g., kg vs. pounds) can create problems.

Proximity Measure Design

- Choosing an **appropriate distance metric** is crucial for accurate clustering.

Advantages and Disadvantages of Clustering Algorithms

S.No.	Advantages	Disadvantages
1.	Cluster analysis algorithms can handle missing data and outliers.	Cluster analysis algorithms are sensitive to initialization and order of the input data.
2.	Can help classifiers in labelling the unlabelled data. Semi-supervised algorithms use cluster analysis algorithms to label the unlabelled data and then use classifiers to classify them.	Often, the number of clusters present in the data have to be specified by the user.
3.	It is easy to explain the cluster analysis algorithms and to implement them.	Scaling is a problem.
4.	Clustering is the oldest technique in statistics and it is easy to explain. It is also relatively easy to implement.	Designing a proximity measure for the given data is an issue.

Proximity Measures

Proximity measures determine **similarity or dissimilarity** among objects. **Distance**

measures (dissimilarity) indicate **how different** objects are.

Similarity measures indicate **how alike** objects are. **property**:

More distance → Less similarity, and vice versa. **Properties of**

Distance Measures (Metric Conditions)

The properties of the distance measures are:

1. D_{ij} is always positive or zero.
2. $D_{ij} = 0$, i.e., the distance between the object to itself is 0.
3. $D_{ij} = D_{ji}$. This property is called symmetry.
4. $D_{ij} \leq D_{ik} + D_{kj}$. This property is called triangular inequality.

Types of Distance Measures Based on Data Types **Quantitative**

Variables

1. **Euclidean Distance (L_2 norm)**
   - Formula:
   $$D(x, y) = \sqrt{\sum (x_i - y_i)^2}$$
   - **Advantage**: Simple and widely used.
   - **Disadvantage**: Affected by scale changes (e.g., unit conversion).
2. **Manhattan Distance (L_1 norm / City Block Distance)**
   - Formula:
   $$D(x, y) = \sum |x_i - y_i|$$
   - **Advantage**: Less sensitive to outliers than Euclidean.
3. **Chebyshev Distance (L∞ norm)**
   - Formula:
   $$D(x, y) = \max |x_i - y_i|$$
   - **Used when** max difference in any feature matters.
4. **Minkowski Distance**
   - General form:
   $$D(x, y) = \left(\sum |x_i - y_i|^r\right)^{1/r}$$
   - When **r = 1**, it is **Manhattan Distance**.
   - When **r = 2**, it is **Euclidean Distance**.
   - When **r → ∞**, it is **Chebyshev Distance**.

Binary Attributes

1. **Simple Matching Coefficient (SMC)**
   - Measures similarity between two binary objects.
   - Formula:

$$SMC = \frac{a + d}{a + b + c + d}$$

   where:
   - **a** = count of **(0,0)** matches
   - **d** = count of **(1,1)** matches
   - **b, c** = mismatches
2. **Jaccard Coefficient**
   - Used when **only "1" values** are important.
   - Formula:

$$J = \frac{d}{b + c + d}$$

3. **Hamming Distance**
   - Counts positions where two binary objects differ.
   - Example: Distance between **101** and **110** is **2**.

Categorical Variables

Distance is 1 if different, 0 if same.

Example: Gender (Male, Female) → Distance = 1

Ordinal Variables

- Have an **inherent order** (e.g., low < medium < high).
- Formula:

$$D(X,Y) = \frac{|Position(X) - Position(Y)|}{(n-1)}$$

  - Example: Distance between **Clerk (1)** and **Manager (3)**:

$$\frac{|1-3|}{3} = \frac{2}{3}$$

Vector-Based Distance Measures (For Text & Documents)

Cosine Similarity

- Measures angle between two vectors.
- Formula:

$$\text{sim}(X,Y) = \frac{\sum x_i y_i}{\sqrt{\sum x_i^2} \times \sqrt{\sum y_i^2}}$$

- Used in **text analysis and document clustering**.

Distance Measures

Measure	Formula	Best for
Euclidean	$\sqrt{\sum(x_i - y_i)^2}$	General numerical data
Manhattan	(\sum	x_i - y_i
Chebyshev	(\max	x_i - y_i
Minkowski	(\left(\sum	x_i - y_i
Simple Matching	$\frac{a+d}{a+b+c+d}$	Binary attribute similarity
Jaccard	$\frac{d}{b+c+d}$	Sparse binary attributes
Hamming	Count of differing bits	Binary sequences
Cosine Similarity	$\frac{\sum x_i y_i}{\sqrt{\sum x_i^2} \times \sqrt{\sum y_i^2}}$	Text & document similarity

Hierarchical Clustering Algorithms

Overview

- Produces a nested partition of objects with hierarchical relationships.
- Represented using a **dendrogram**.
- Two main categories: **Agglomerative** and **Divisive** methods.

Types of Hierarchical Clustering

1. **Agglomerative Methods (Bottom-Up)**
    - Each sample starts as an individual cluster.
    - Clusters are merged iteratively until one cluster remains.
    - Once a cluster is formed, it cannot be undone (**irreversible**).
2. **Divisive Methods (Top-Down)**
    - Starts with a single cluster containing all data points.
    - Splits iteratively into smaller clusters.
    - Continues until each sample becomes its own cluster.

Agglomerative Clustering Techniques

Single Linkage (MIN Algorithm)

- Merges clusters based on the **smallest distance** between two points from different clusters.
- Related to the **Minimum Spanning Tree (MST)**.

Complete Linkage (MAX or Clique Algorithm)

In complete linkage algorithm, the distance (x, y), (where x is from one cluster and y is from another cluster), is the largest distance between all possible pairs of the two groups or clusters (or simply the largest distance of two points where points are in different clusters) as given below. It is used for merging the clusters.

$$D_{CL}(C_i, C_j) = \text{maximum}_{a \in C_i,\, b \in C_j} d(a, b) \qquad (13.11)$$

Dendrogram for the above clustering is shown in Figure 13.3.

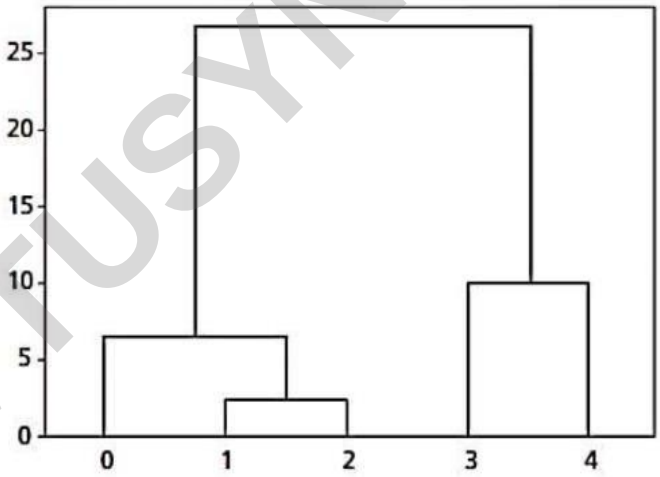


Figure 13.3: Dendrogram for Table 13.4

Average Linkage Algorithm

In case of an average linkage algorithm, the average distance of all pairs of points across the clusters is used to form clusters. The average value computed between clusters c_i, c_j is given as follows:

$$D_{AL}(C_i, C_j) = \frac{1}{m_i m_j} \sum_{a \in C_i, b \in C_j} d(a, b) \quad (13.12)$$

Here, m_i and m_j are sizes of the clusters.

The dendrogram for Table 13.4 is given in Figure 13.4.

Figure 13.4: Dendrogram for Table 13.4

Mean-Shift Clustering Algorithm

- **Non-parametric and hierarchical** clustering technique.
- Also known as **mode-seeking** or **sliding window algorithm**.
- No prior knowledge of cluster count or shape required.
- Moves towards high-density regions in data using a **kernel function** (e.g., **Gaussian window**).

Advantages of Mean-Shift Clustering

No model assumptions

Suitable for all non-convex shapes

Only one parameter of the window, that is, bandwidth is required Robust to noise No

issues of local minima or premature termination

Disadvantages of Mean-Shift Clustering

Selecting the bandwidth is a challenging task. If it is larger, then many clusters are missed. If it is small, then many points are missed and convergence occurs as the problem.

The number of clusters cannot be specified and user has no control over this parameter.

Partitional Clustering Algorithm

- **k-means** is a widely used **partitional clustering** algorithm.
- The user specifies **k**, the number of clusters.
- Assumes **non-overlapping** clusters.
- Works well for **circular or spherical clusters**.

Process of k-means Algorithm

1. **Initialization**
    - Select **k** initial cluster centers (randomly or using prior knowledge).
    - Normalize data for better performance.
2. **Assignment of Data Points**
    - Assign each data point to the **nearest centroid** based on **Euclidean distance**.
3. **Update Centroids**

- Compute the **mean vector** of assigned points to update cluster centroids.
- Repeat this process until no changes occur in cluster assignments.

4. **Termination**
   - The process stops when cluster assignments remain unchanged.

Algorithm 13.3: *k*-means Algorithm

Step 1: Determine the number of clusters before the algorithm is started. This is called *k*.

Step 2: Choose *k* instances randomly. These are initial cluster centers.

Step 3: Compute the mean of the initial clusters and assign the remaining sample to the closest cluster based on Euclidean distance or any other distance measure between the instances and the centroid of the clusters.

Step 4: Compute new centroid again considering the newly added samples.

Step 5: Perform the steps 3–4 till the algorithm becomes stable with no more changes in assignment of instances and clusters.

Mathematical Optimization

- **Objective:** Minimize **Sum of Squared Errors (SSE)**

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} \text{dist}(c_i, x)^2$$

Where:

- c_i = centroid of cluster i
- x = data point
- **dist** = Euclidean distance

Advantages

1. **Simple** and easy to implement.
2. **Efficient** for small to medium datasets.

Disadvantages

1. **Sensitive to initialization** – different initial points may lead to different results.
2. **Time-consuming for large datasets** – requires multiple iterations.

Choosing the Value of k

- No fixed rule for selecting **k**.
- Use **Elbow Method**:
    - Run k-means with different values of **k**.
    - Plot **Within Cluster Sum of Squares (WCSS)** vs. **k**.
    - The optimal **k** is at the **"elbow"** where the curve flattens.

Computational Complexity

O(nkId), where:

- **n** = number of samples
- **k** = number of clusters
- **I** = number of iterations
- **d** = number of attributes

Density-based Methods

- **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)** is a **density-based** clustering algorithm.
- Clusters are **dense regions** of data points separated by areas of **low density (noise)**.
- Works well for **arbitrary-shaped clusters** and datasets with **noise**.

Algorithm 13.4: DBSCAN

Step 1: Randomly select a point p. Compute distance between p and all other points.

Step 2: Find all points from p with respect to its neighbourhood and check whether it has minimum number of points m. If so, it is marked as a core point.

Step 3: If it is a core point, then a new cluster is formed, or existing cluster is enlarged.

Step 4: If it is a border point, then the algorithm moves to the next point and marks it as visited.

Step 5: If it is a noise point, they are removed.

Step 6: Merge the clusters if it is mergeable, $dist\ (c_i, c_j) < \varepsilon$.

Step 7: Repeat the process 3–6 till all points are processed.

Uses two parameters:

1. **ε (epsilon)** – Neighborhood radius.
2. **m (minPts)** – Minimum number of points within **ε** to form a cluster.

Types of Points in DBSCAN

1. **Core Point**
   - A point with at least **m** points in its **ε-neighborhood**.
2. **Border Point**
   - Has **fewer than m** points in its **ε-neighborhood** but is adjacent to a **core point**.
3. **Noise Point (Outlier)**
   - Neither a **core point** nor a **border point**.

Density Connectivity Measures

1. **Direct Density Reachability**
   - Point **X** is directly reachable from **Y** if:
     - **X** is in the **ε-neighborhood** of **Y**.
     - **Y** is a **core point**.
2. **Densely Reachable**

- **X** is densely reachable from **Y** if there exists a **chain of core points** linking them.

3. **Density Connected**
   - **X and Y** are density connected if they are both **densely reachable** from a common **core point Z**.

Advantages of DBSCAN

1. Can detect arbitrary-shaped clusters.
2. Robust to noise and outliers.
3. Does not require specifying the number of clusters (k-means does).

Disadvantages of DBSCAN

1. Sensitive to ε and m parameters – Poor parameter choice can affect results.
2. Fails in datasets with varying density – A single ε may not work for all clusters.
3. Computationally expensive for high-dimensional data.

Grid-based Approach

- **Grid-based clustering** partitions space into a **grid structure** and fits data into **cells** for clustering.
- Suitable for **high-dimensional data**.
- Uses **subspace clustering**, **dense cells**, and **monotonicity property**.

Concepts

Subspace Clustering

- Clusters are formed using a **subset of features (dimensions)** rather than all attributes.
- Useful for **high-dimensional data** like **gene expression** analysis.

- **CLIQUE (Clustering in Quest)** is a widely used **grid-based subspace clustering algorithm**.

Concept of Dense Cells

- CLIQUE partitions dimensions into **intervals (cells)**.
- A cell is **dense** if its data point density **exceeds a threshold**.
- Dense cells are **merged** to form clusters.

Algorithm 13.5: Dense Cells

Step 1: Define a set of grid points and assign the given data points on the grid.

Step 2: Determine the dense and sparse cells. If the number of points in a cell exceeds the threshold value τ, the cell is categorized as dense cell. Sparse cells are removed from the list.

Step 3: Merge the dense cells if they are adjacent.

Step 4: Form a list of grid cells for every subspace as output.

Monotonicity Property

Algorithm 13.6: CLIQUE

Stage 1:

Step 1: Identify the dense cells.

Step 2: Merge dense cells c_1 and c_2 if they share the same interval.

Step 3: Generate Apriori rule to generate $(k + 1)^{th}$ cell for higher dimension. Then, check whether the number of points cross the threshold. This is repeated till there are no dense cells or new generation of dense cells.

Stage 2:

Step 1: Merging of dense cells into a cluster is carried out in each subspace using maximal regions to cover dense cells. The maximal region is an hyperrectangle where all cells fall into.

Step 2: Maximal region tries to cover all dense cells to form clusters.

In stage two, CLIQUE starts from dimension 2 and starts merging. This process is continued till the n-dimension.

- Uses **anti-monotonicity (Apriori property)**:
  - If a **k-dimensional cell** is dense, then **all (k-1) dimensional projections** must also be dense.
  - If a lower-dimensional cell is **not dense**, then higher-dimensional cells containing it are **also not dense**.
- Similar to **association rule mining** in frequent pattern mining.

Advantages of CLIQUE

1. **Insensitive to input order of objects**.
2. **No assumptions about data distribution**.
3. **Finds high-density clusters in subspaces** of high-dimensional data.

Disadvantage of CLIQUE

- **Tuning grid parameters (grid size, density threshold) is difficult**.
- **Finding the optimal threshold** to classify a cell as dense is challenging.

Chapter :- 2

Reinforcement Learning

Overview of Reinforcement Learning

What is Reinforcement Learning?

- **Reinforcement Learning (RL)** is a **machine learning** paradigm that mimics how **humans** and **animals** learn through **experience**.
- Humans **interact with the environment**, receive **feedback (rewards or penalties)**, and adjust their behavior accordingly.
- **Example**: A child touching fire learns to avoid it after experiencing pain (negative reinforcement).

How RL Works in Machines

- RL **simulates** real-world scenarios for a computer program (agent) to **learn** by trial and error.
- The agent **executes actions**, receives **positive or negative rewards**, and **optimizes its future actions** based on these experiences.

Types of Reinforcement Learning

1. **Positive Reinforcement Learning**
    - **Rewards encourage good behavior** (reinforce correct actions).
    - **Example**: A robot gets **+10 points** for reaching a goal successfully.
    - **Effect**: Increases the likelihood of repeating the rewarded action.
2. **Negative Reinforcement Learning**
    - **Negative rewards discourage unwanted actions**.
    - **Example**: A game agent loses **-10 points** for stepping into a danger zone.
    - **Effect**: Helps the agent learn to **avoid negative outcomes**.

Characteristics of RL

- **Sequential Decision-Making**: The agent **makes a series of decisions** to maximize total rewards.
- **Trial and Error Learning**: The agent learns by **exploring different actions** and their consequences.
- **No Supervised Labels**: Unlike **supervised learning**, RL does **not require labeled data**; it learns from experience.

Applications of Reinforcement Learning

- **Robotics**: Teaching robots to **walk, grasp objects, or perform complex tasks**.
- **Gaming**: AI agents in **chess, Go, and video games** (e.g., AlphaGo, OpenAI Five).
- **Autonomous Vehicles**: Self-driving cars learn **optimal driving strategies**.
- **Finance**: AI-based **trading strategies** for stock markets.
- **Healthcare**: Personalized treatment plans based on patient responses.

Scope of Reinforcement Learning

Reinforcement Learning (RL) is well-suited for **decision-making problems** in **dynamic and uncertain environments**. It excels in cases where an agent must **learn through trial and error** and optimize its actions based on delayed rewards.

Situations Where RL Can Be Used

Pathfinding and Navigation

- Consider a **grid-based game** where a **robot** must navigate from a **starting node (E)** to a **goal node (G)** by choosing the optimal path.
- RL can learn the **best route** by exploring different paths and receiving feedback on their efficiency.

- In **obstacle-based games**, RL can identify **safe paths** while avoiding dangerous zones.

Dynamic Decision-Making with Uncertainty

- RL is useful in environments where **not all information is known upfront**.
- It is **not suitable** for tasks like **object detection**, where a classifier with complete labeled data performs better.

Characteristics of Reinforcement Learning

1. **Sequential Decision-Making**
   - In RL, decisions are made in **steps**, and each step influences future choices.
   - Example: In a **maze game**, a wrong turn can lead to failure.
2. **Delayed Feedback**
   - Rewards are **not always immediate**; the agent may need to **take multiple steps** before receiving feedback.
3. **Interdependence of Actions**
   - Each action affects the **next set of choices**, meaning an incorrect move can have long-term consequences.
4. **Time-Related Decisions**
   - Actions are taken in a **specific sequence over time**, affecting the final outcome.

Challenges in Reinforcement Learning

Reward Design

- Setting the **right reward values** is crucial. **Incorrectly designed rewards** may lead the agent to learn undesired behavior.

Absence of a Fixed Model

- Some environments, like **chess**, have fixed rules, but many real-world problems **lack predefined models**.
- Example: Training a **self-driving car** requires **simulations** to generate experiences.

Partial Observability

- Some environments, like **weather prediction**, involve **uncertainty** because **complete state information** is unavailable.

High Computational Complexity

- Games like **Go** involve a **huge state space**, making RL training time-consuming.
- **More possible actions → More training time needed.**

Applications of Reinforcement Learning

1. **Industrial Automation**
   - Optimizing **robot movements** for efficiency.
2. **Resource Management**
   - Allocating resources in **data centers and cloud computing**.
3. **Traffic Light Control**
   - Reducing **congestion** by adjusting signal timings dynamically.
4. **Personalized Recommendation Systems**
   - Used in **news feeds, e-commerce, and streaming services** (e.g., **Netflix recommendations**).
5. **Online Advertisement Bidding**
   - Optimizing **ad placements** for maximum engagement.
6. **Autonomous Vehicles**
   - RL helps in training **self-driving cars** to navigate safely.
7. **Game AI (Chess, Go, Dota 2, etc.)**
   - AI models like **AlphaGo** use RL to master complex games.
8. **DeepMind Applications**

- AI systems that **generate programs, images, and optimize machine learning models**.

Reinforcement Learning as Machine Learning

Reinforcement Learning (RL) is a distinct branch of **machine learning** that differs significantly from **supervised learning**.

While **supervised learning** depends on **labeled data**, **reinforcement learning learns through interaction with the environment**, making decisions based on **trial and error**.

Why RL Is Necessary?

Some tasks **cannot** be solved using **supervised learning** due to the **absence of a labeled training dataset**. For example:

- **Chess & Go:** There is no dataset with all possible game moves and their outcomes. RL allows the agent to **explore** and improve over time.
- **Autonomous Driving:** The car must **learn** from real-world experiences rather than relying on a fixed dataset.

Challenges in Reinforcement Learning Compared to Supervised Learning

- **More complex decision-making** since every action affects future outcomes.
- **Longer training times** due to trial-and-error learning.
- **Delayed rewards**, making it difficult to attribute success or failure to a specific action.

Differences between Supervised Learning and Reinforcement Learning

Reinforcement Learning	Supervised Learning
No supervisor and labelled dataset initially	Presence of supervisor and labelled data
Decisions are dependent and are made sequentially	Decisions are independent of each other and based on input given in the training phase
Feedback is not instantaneous and delayed by time	Usually, the feedback is instantaneous once the model is created
Agent action affects the next input data	Dependent of initial input or the input given at start
No target values, only goal oriented	Target class is predefined by the problem
Example: Chess, GO, Atari Games	Example: Classifiers

Components of Reinforcement Learning

Reinforcement Learning (RL) is based on an **agent** interacting with an **environment** to learn an optimal strategy through **trial and error**.

Basic Components of RL

Figure 14.4: Basic Components of RL

1. **Agent** – The decision-maker (e.g., a robot, self-driving car, AI player in a game).
2. **Environment** – The external world where the agent interacts (e.g., a game board, real-world traffic).
3. **State (S)** – A representation of the environment at a specific time.
4. **Actions (A)** – The possible choices available to the agent.
5. **Rewards (R)** – The feedback signal received by the agent for taking an action.
6. **Policy (π)** – The agent's strategy for selecting actions based on states.
7. **Episodes** – The sequence of states, actions, and rewards from the start state to the goal state.

Types of RL Problems

Learning Problems

- **Unknown environment** – The agent learns by **trial and error**.
- **Goal** – Improve the policy through interaction.
- **Example** – A robot navigating through an unknown maze.

Planning Problems

- **Known environment** – The agent can compute and improve the policy using a model.
- **Example** – Chess AI that plans its moves based on game rules.

Environment and Agent

- The **environment** contains all elements the agent interacts with, including obstacles, rewards, and state transitions.
- The **agent** makes decisions and performs actions to maximize rewards.

Example

In **self-driving cars**,

- The **environment** includes roads, traffic, and signals.
- The **agent** is the AI system making driving decisions.

States and Actions

- **State (S)** – Represents the current situation.
- **Action (A)** – Causes a transition from one state to another.

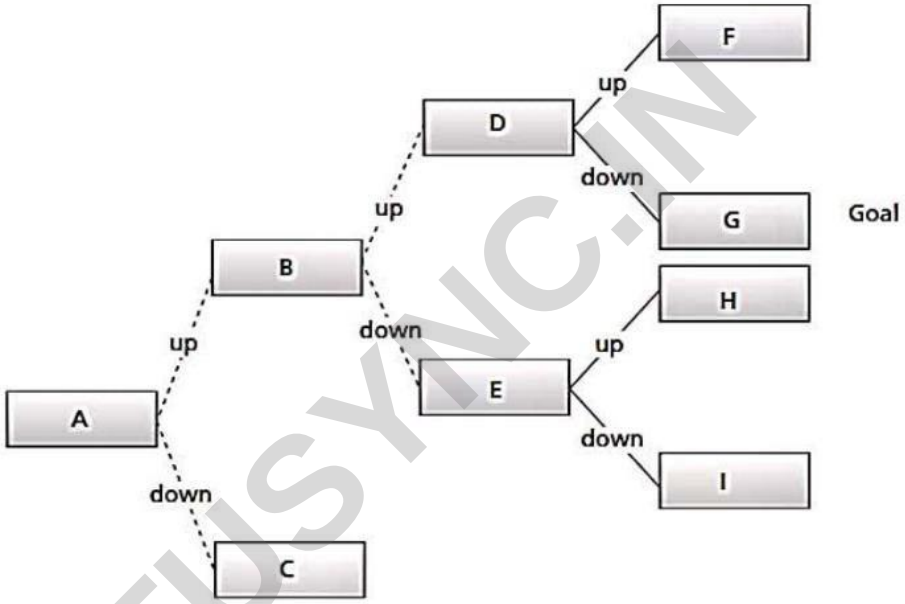


Figure 14.5: A Game: UP Action Results in Darker Line and Down Operation Results in Dotted Lines

Example (Navigation)

In a **grid-based game**, states represent **positions** (A, B, C, etc.), and actions are movements (**UP, DOWN, LEFT, RIGHT**).

Types of States

1. **Start State** – Where the agent begins.
2. **Goal State** – The target state with the highest reward.
3. **Non-terminal States** – Intermediate steps between start and goal.

Types of Episodes

- **Episodic** – Has a definite **start** and **goal state** (e.g., solving a maze).
- **Continuous** – No fixed goal state; the task continues indefinitely (e.g., stock trading).

Policies in RL

A **policy (π)** is the strategy used by the agent to choose actions. **Types of**

Policies

1. **Deterministic Policy** – Always maps a state to the same action:

$$a = \pi(s)$$

2. **Stochastic Policy** – Gives a probability of taking an action in a given state:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

(Probability of choosing action **a** in state **s**).

Choosing the Best Policy

- The **optimal policy** is the one that **maximizes cumulative expected rewards**.

Rewards in RL

- **Immediate Reward (r)** – The instant feedback for an action.
- **Total Reward (G)** – The sum of all rewards collected during an episode.
- **Long-term Reward** – The cumulative future reward.

Discount Factor (γ)

To **prioritize immediate rewards** over distant ones, we use a **discount factor (γ)**:

$$G = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$$

- **γ = 1** → The agent considers **all future rewards equally**.
- **γ < 1** → The agent **prioritizes immediate rewards**.
- **Typical values**: 0.8 or 0.9.

Example: Calculating Discounted Reward

If at **step 5**, reward = **+1**, and **γ = 0.7**, then:

$$(0.7)^5 \times 1 = 0.16807$$

RL Algorithm Categories

- **Model-Based RL** – Uses a predefined model (e.g., Chess AI).
- **Model-Free RL** – Learns by trial and error (e.g., a robot navigating an unknown environment).

Markov Decision Process

A **Markov Chain** is a stochastic process that satisfies the Markov property.

It consists of a sequence of random variables where the probability of transitioning to the next state depends only on the current state and not on the past states.

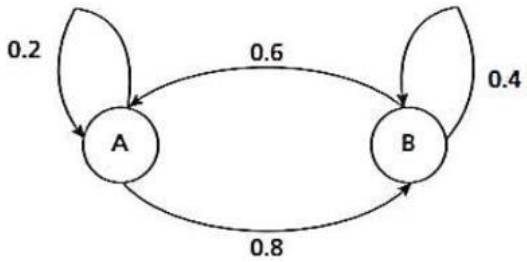


Figure 14.6: Markov Chain

Example: University Transition

Consider two universities:

- **80%** of students from University A move to University B for a master's degree, while **20%** remain in University A.
- **60%** of students from University B move to University A, while **40%** remain in University B.

This can be represented as a Markov Chain, where:

- States represent the universities.
- Edges denote the probability of transitioning between states.

A **transition matrix** at time is defined as:

Each row represents a probability distribution, meaning the sum of elements in each row equals **1**.

Probability Prediction

Let the initial distribution be:

$$u_0 = (0.6, 0.4)$$

To find the state distribution after one time step

$$u_1 = u_0 \times P = (0.6, 0.4) \times \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 0.6 \end{bmatrix} = (0.28, 0.72)$$

:

After two time steps:

$$\nu_2 = \nu_1 \times P = (0.28, 0.72) \times \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 0.6 \end{bmatrix} = (0.344, 0.656)$$

The system stabilizes over time, reflecting the equilibrium distribution.

Markov Decision Process (MDP)

An **MDP** extends a Markov Chain by incorporating rewards. It consists of:

1. **Set of states**
2. **Set of actions**
3. **Transition probability function**
4. **Reward function**
5. **Policy**
6. **Value function**

Markov Assumption

The Markov property states that the probability of reaching a state and receiving a reward depends only on the previous state and action :

$$P(s'|s, a) = P(s'|s, a, s_{t-1}, s_{t-2}, ...)$$

MDP Process

1. **Observe** the current state .
2. **Choose an action** .
3. **Receive a reward** .
4. **Move to the next state** .
5. **Repeat to maximize cumulative rewards**.

State Transition Probability

The probability of moving from state to after taking action is given by:

$$P(s'|s,a)$$

This forms a **state transition matrix**, where each row represents transition probabilities from one state to another.

Expected Reward

The expected reward for an action in state is given by:

$$R(s,a) = E[r|s,a]$$

Training and Testing of RL Systems

Once an MDP is modeled, the system undergoes:

1. **Training:** The agent repeatedly interacts with the environment, adjusting parameters based on rewards.
2. **Inference:** A trained model is deployed to make decisions in real-time.
3. **Retraining:** When the environment changes, the model is retrained to adapt and improve performance.

Goal of MDP

The agent's objective is to maximize **total accumulated rewards** over time by following an optimal policy.

Multi-Arm Bandit Problem and Reinforcement Problem Types Reinforcement

Learning Overview

Reinforcement learning (RL) uses trial and error to learn a series of actions that maximize the total reward. RL consists of two fundamental sub-problems:

Prediction (Value Estimation):

- The goal is to predict the total reward (return), also known as **policy evaluation** or **value estimation**.
- This requires the formulation of a function called the **state-value function**.
- The estimation of the state-value function can be performed using **Temporal Difference (TD) Learning**.

Policy Improvement:

- The objective is to determine actions that maximize returns.
- This process is known as **policy improvement**.
- Both prediction and policy improvement can be combined into **policy iteration**, where these steps are used alternately to find an optimal policy.

Multi-Arm Bandit Problem

A commonly encountered problem in reinforcement learning is the **multi-arm bandit problem** (or **N-arm bandit problem**).

Consider a hypothetical casino with a robotic arm that activates a **5-armed slot machine**. When a lever is pulled, the machine returns a reward within a specified range (e.g., $1 to $10).

The challenge is that each arm provides rewards randomly within this range.

Objective:

Given a limited number of attempts, the goal is to maximize the total reward by selecting the best lever.

A logical approach is to determine which lever has the highest **average reward** and use it repeatedly.

Formalization:

Given **k** attempts on an **N-arm slot machine**, with rewards , the expected reward (action- value function) is:

$$Q(a) = \frac{1}{k} \sum_{i=1}^{k} r_i$$

The best action is defined as:

$$a^* = \arg\max Q(a)$$

This indicates the **action that returns the highest average reward** and is used as an indicator of action quality.

Example:

If a slot machine is chosen five times and returns rewards , the quality of this action is:

$$Q(a) = \frac{1+2+0+7+9}{5} = 3.8$$

Exploration vs Exploitation and Selection Policies

In reinforcement learning, an agent must decide how to select actions:

Exploration:

- Tries all actions, even if they lead to sub-optimal decisions.
- Useful in games where exploring different actions provides better long-term rewards.
- Risky but informative.

Exploitation:

- Uses the **current best-known action** repeatedly.
- Focuses on **short-term gains**.
- Simple but often sub-optimal.

A balance between exploration and exploitation is crucial for optimal decision-making.

Selection Policies

Greedy Method

- Picks the best-known action at any given time.
- Based solely on exploitation.
- Risk: It may miss out on exploring better options. **ε-**

Greedy Method

- Balances exploration and exploitation.
- With probability **ε**, the agent explores a random action.
- With probability **1 - ε**, it selects the best-known action.
- **ε** ranges from 0 to 1 (e.g., ε = 0.1 means a 10% chance of exploration).

ε – *Greedy Method*

This method combines both exploration and exploitation. In ε – *greedy*, ε value ranges from 0 to 1 (refer to Algorithm 14.1).

Algorithm 14.1: ε - Greedy

Step 1: Generate a random number p in the range 0 – 1.

Step 2: If $p \geq \varepsilon$, the algorithm chooses the action with highest reward value.

Step 3: If $p < \varepsilon$, actions are chosen randomly.

Step 4: Update the value of action and repeat for all actions.

With probability ε, the action a is chosen randomly, but rest of the time with probability $1 - \varepsilon$, the best actions based on greedy approach are selected. Thus, there is a balance of exploration as well as exploitation. For example, if ε is 0.2, then 80% best actions are chosen and 20% random actions are chosen.

Reinforcement Learning Agent Types

An RL agent can be classified into different approaches based on how it learns:

1. **Value-Based Approaches**
   - Optimize the value function , which represents the maximum expected future reward from a given state.
   - Uses **discount factors** to prioritize future rewards.
2. **Policy-Based Approaches**
   - Focus on finding the **optimal policy** , a function that maps states to actions.
   - Rather than estimating values, it directly learns which action to take.
3. **Actor-Critic Methods (Hybrid Approaches)**
   - Combine value-based and policy-based methods.
   - The **actor** updates the policy, while the **critic** evaluates it.
4. **Model-Based Approaches**
   - Create a model of the environment (e.g., using **Markov Decision Processes (MDPs)**).
   - Use simulations to plan the best actions.
5. **Model-Free Approaches**

- No predefined model of the environment.
- Use methods like **Temporal Differencing (TD) Learning** and **Monte Carlo methods** to estimate values from experience.

Reinforcement Algorithm Selection

The choice of a reinforcement learning algorithm depends on factors such as:

- **Availability of models**
- **Nature of updates** (incremental vs. batch learning)
- **Exploration vs. exploitation trade-offs**
- **Computational efficiency**

Reinforcement Algorithm Selection

Reinforcement agent algorithms are decided based on the criteria of availability of models, environment and nature of updates. The agent algorithm selections are shown in Table 14.3.

Table 14.3: Algorithm Selection

Model	Environment	On/off	Algorithms Used
Present	Static	Off	Dynamic Programming
Absent	Dynamic	On	SARSA
Absent	Dynamic	-	Temporal Learning, Monte Carlo Methods
Absent	Dynamic	Off	Q-Learning

Model-based Learning

Passive Learning refers to a **model-based environment**, where the environment is **known**. This means that for any given state, the next state and action probability distribution are known.

Markov Decision Process (MDP) and **Dynamic Programming** are powerful tools for solving reinforcement learning problems in this context.

The mathematical foundation for passive learning is provided by **MDP**. These model- based reinforcement learning problems can be solved using **dynamic programming** after constructing the model with MDP.

The primary objective in reinforcement learning is to take an action **a** that transitions the system from the current state to the end state while maximizing rewards. These rewards can be positive or negative.

The goal is to maximize expected rewards by choosing the **optimal policy**: for all

possible values of **s** at time **t**.

Policy and Value Functions

An **agent** in reinforcement learning has multiple courses of action for a given state. The way the agent behaves is determined by its **policy**.

A **policy** is a distribution over all possible actions with probabilities assigned to each action.

Different actions yield different rewards. To quantify and compare these rewards, we use **value functions**.

Value Function Notation

A **value function** summarizes possible future scenarios by averaging expected returns under a given policy π.

It is a prediction of future rewards and computes the expected sum of future rewards for a given state **s** under policy π:

$$v(s) = E[G_t|S_t = s]$$

where **v(s)** represents the **quality of the state** based on a long-term strategy. **Example**

If we have two states with values **0.2** and **0.9**, the state with **0.9** is a better state to be in. Value

functions can be of two types:

- **State-Value Function** (for a state)
- **State-Action Function** (for a state-action pair)

State-Value Function

Denoted as **v(s)**, the **state-value function** of an MDP is the expected return from state **s** under a policy **π**:

This function accumulates all expected rewards, potentially discounted over time, and helps determine the **goodness of a state**.

The **optimal state-value function** is given by:

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

Action-Value Function (Q-Function)

Apart from **v(s)**, another function called the **Q-function** is used. This function returns a real value indicating the total expected reward when an agent:

1. Starts in state **s**
2. Takes action **a**
3. Follows a policy **π** afterward

It is denoted as:

$$Q_\pi(s,a) = E[G_t|S_t = s, A_t = a]$$

The optimal Q-value function is given by:

$$v^*(s) = \max_a Q_\pi(s,a)$$

Bellman Equation

Dynamic programming methods require a **recursive formulation** of the problem. The recursive formulation of the **state-value function** is given by the **Bellman equation**:

$$v(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R_{s,a} + \gamma v(s')]$$

where:

- **P(s' | s, a)**: Probability of transitioning to **s'** from **s** after taking action **a**
- **R(s, a)**: Reward received after taking action **a** in state **s**
- **γ**: Discount factor (0 ≤ γ ≤ 1) controlling the weight of future rewards

Solving Reinforcement Problems

There are two main algorithms for solving reinforcement learning problems using conventional methods:

1. **Value Iteration**
2. **Policy Iteration**

Value Iteration

Value iteration estimates **v(s)** iteratively:

$$v_{t+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R_{s,a} + \gamma v_t(s')]$$

Algorithm

1. Initialize **v(s)** arbitrarily (e.g., all zeros).
2. Iterate until convergence:
    - For each state **s**, update **v(s)** using the Bellman equation.
    - Repeat until changes are negligible.

Policy Iteration

Policy iteration consists of **two main steps**:

1. **Policy Evaluation**
2. **Policy Improvement**

Policy Evaluation

Initially, for a given policy π, the algorithm starts with **v(s) = 0** (no reward). The Bellman equation is used to obtain **v(s)**, and the process continues iteratively until the optimal **v(s)** is found.

Policy Improvement

The policy improvement process is performed as follows:

1. Evaluate the current policy using policy evaluation.
2. Solve the Bellman equation for the current policy to obtain **v(s)**.
3. Improve the policy by applying the **greedy approach** to maximize expected rewards.
4. Repeat the process until the policy converges to the optimal policy.

Algorithm

1. Start with an arbitrary policy π.
2. Perform **policy evaluation** using Bellman's equation.

3. Improve the policy greedily.
4. Repeat until convergence.

Model Free Methods

Model-free methods do not require complete knowledge of the environment. Instead, they learn through experience and interaction with the environment.

The reward determination in model-free methods can be categorized into three formulations:

1. **Episodic Formulation**: Rewards are assigned based on the outcome of an entire episode. For example, if a game is won, all actions in the episode receive a positive reward (+1). If lost, all actions receive a negative reward (-1). However, this approach may unfairly penalize or reward intermediate actions.
2. **Continuous Formulation**: Rewards are determined immediately after an action. An example is the multi-armed bandit problem, where an immediate reward between $1 - $10 can be given after each action.
3. **Discounted Returns**: Long-term rewards are considered using a discount factor. This method is often used in reinforcement learning algorithms.

Model-free methods primarily utilize the following techniques:

- **Monte Carlo (MC) Methods**
- **Temporal Difference (TD) Learning**

Monte-Carlo Methods

Monte Carlo (MC) methods do not assume any predefined model, making them purely experience-driven. This approach is analogous to how humans and animals learn from interactions with their environment.

Algorithm 14.5: MC

Step 1: For a policy π, select state s.
Step 2: Generate an episode.
Step 3: Compute average returns for every first occurrence of state s in an episode.
Step 4: Update the returns and start a new episode. Repeat for all episodes.

Characteristics of Monte Carlo Methods:

- Experience is divided into **episodes**, where each episode is a sequence of states from a starting state to a goal state.
- **Episodes must terminate**; regardless of the starting point, an episode must reach an endpoint.
- **Value-action functions** are computed only after the completion of an episode, making MC an **incremental** method.
- MC methods compute rewards at the end of an episode to estimate maximum expected future rewards.
- **Empirical mean** is used instead of expected return; the total return over multiple episodes is averaged.
- Due to the non-stationary nature of environments, value functions are computed for a fixed policy and revised using **dynamic programming**.

Monte Carlo Mean Value Computation:

The mean value of a state is calculated as:

$$V(s_i) = \frac{1}{N} \sum G_i$$

Incremental Monte Carlo Update:

The value function is updated incrementally using the following formula:

$$V(s_i) = V(s_i) + \alpha(G_i - V(s_i))$$

where α is the learning rate.

Temporal Difference (TD) Learning

Temporal Difference (TD) Learning is an alternative to Monte Carlo methods. It is also a model-free technique that learns from experience and interaction with the environment.

Characteristics of TD Learning:

- **Bootstrapping Method**: Updates are based on the current estimate and future reward.
- **Incremental Updates**: Unlike MC, which waits until the end of an episode, TD updates values at each step.
- **More Efficient**: TD can learn before an episode ends, making it more sample-efficient than MC methods.
- **Used for Non-Stationary Problems**: Suitable for environments where conditions change over time.

Differences between Monte Carlo and TD Learning

Feature	Monte Carlo (MC)	Temporal Difference (TD)
Update Timing	End of episode	After each step
Learning Type	Experience-based	Bootstrapping
Computational Efficiency	Less efficient	More efficient
Suitability	Static environments	Dynamic environments

Monte-Carlo Method	Temporal Difference Method
It is easy to understand and does not exploit Markov principle	Exploits Markov principle
This requires termination and episode	Does not require termination

Eligibility Traces and TD(λ)

TD Learning can be accelerated using **eligibility traces**, which allow updates to be spread over multiple states. This leads to a family of algorithms called **TD(λ)**, where **λ** is the decay parameter ($0 \leq \lambda \leq 1$):

- **λ = 0**: Only the previous prediction is updated.
- **λ = 1**: All previous predictions are updated.

By incorporating eligibility traces, TD(λ) provides an alternative short-term memory mechanism to enhance learning efficiency.

Algorithm 14.6: TD-Learning

1. Initialize policy π.
2. Set $v(s) = 0$ for all states $s \in S$.
3. Repeat for each episode.
4. Initialize s.
5. Repeat for each step of episode.
6. Choose action 'a' and execute it. Observe the new state and new reward.
7. Update value using Bellman Eq. (14.22).
8. Update as s'.
9. Until s is terminal state.

Q-Learning

Q-Learning Algorithm

1. **Initialize Q-table**:
    - Create a table Q(s,a) with states s and actions a.
    - Initialize Q-values with random or zero values.
2. **Set parameters**:
    - Learning rate α\alphaα (typically between 0 and 1).
    - Discount factor γ (typically close to 1).
    - Exploration-exploitation trade off strategy (e.g., ε-greedy policy).
3. **Repeat for each episode**:
    - Start from an initial state s.
    - Repeat until reaching a terminal state:

        1. Choose an action a using an exploration strategy (e.g., ε-greedy).
        2. Perform action a, observe new state s' and reward r.
        3. Update Q-value using the **Bellman equation**:

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

        4. Set $s = s'$.

4. **End the training** once convergence is reached (Q-values become stable).

This iterative process helps the agent learn optimal Q-values, which guide it to take actions that maximize rewards.

Algorithm 14.7: Q-Learning

1. Set $Q(s_t, a) = 0$ where s_t is the terminal node in an episode.
2. For all states s and action a, set Q Value = 0.
3. Repeat.
4. Select s_t randomly.
5. Choose action a_t from Q using ε- greedy.
6. Perform action a_t such that $r(s_t, a_t) > 0$ and reach the next state.
7. Update the action-value function using TD using Eq. (14.24) and Bellman equation:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha \times TD_t(s_t, a_t) \quad (14.25)$$

Until the terminal state s is reached.

Here, α is the learning rate. It is between 0 and 1. If α is zero, then nothing is updated and there is no learning at all. When setting it to a higher value such as 0.9, the learning happens fast. $\gamma$is the discount factor. It is in the range 0 – 1. It should be less so that the algorithm can converge.

The inference is simple, the best action is the maximum of Q function as follows:

$$a_t = \arg\max_a (Q(s_t, a))$$

SARSA Learning

SARSA Algorithm (State-Action-Reward-State-Action)

Algorithm 14.8: SARSA

1. For all states and actions, set $Q(s_t, a) = 0$.
2. Initialize α, ε and γ.
   Initialize the state s_t.
   Choose action a_t based on selection policy ε-greedy strategy and execute it.
   Get a new state s_{t+1}, observe the reward r_{t+1}.
   Choose action a_{t+1} from Q based on ε-Greedy procedure and execute it.
   Update Action-value function of Eq. (14.28).
3. Update the values $s_t = s_{t+1}$, $a_t = a_{t+1}$.
4. If s' is terminal node, then start a new episode else repeat above steps.

Initialize Q-table:

- Create a table Q(s,a) for all state-action pairs.

- Initialize Q-values with random or zero values.

Set parameters:

- Learning rate α (typically between 0 and 1).
- Discount factor γ (typically close to 1).
- Exploration-exploitation strategy (e.g., ε-greedy policy).

Repeat for each episode:

- Start from an initial state s.
- Choose an action a using the ε-greedy policy.

Repeat until the terminal state is reached:

1. Take action a, observe reward r and next state s'.
2. Choose next action a' from s' using ε-greedy policy.
3. Update Q-value using the **SARSA update rule**:

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$$

4. Set $s = s'$ and $a = a'$.

End the training when Q-values converge. **Differences**

between SARSA and Q-Learning

Feature	Q-Learning (Off-Policy)	SARSA (On-Policy)
Policy	Learns from the best possible actions (greedy)	Learns from actions taken (ε-greedy)
Update Rule	Uses maximum future Q-value: $r + \gamma \max Q(s',a')$	Uses estimated future Q-value: $r + \gamma Q(s',a')$
Exploration	More aggressive; optimizes for long-term rewards	More conservative; balances risk and reward