KALINGA UNIVERSITY NAYA RAIPUR

FACULTY OF CS & IT

Programme: -BCAAIML
Course Name: - Machine Learning Techniques
Course Code: -BCAAIML501 **Sem : 5th**

Unit-II

Linear Models

Linear models are a class of models that make predictions using a linear function of the input features. The general form is:

y=w1x1+w2x2+⋯+wnxn+by = w_1x_1 + w_2x_2 + \dots + w_nx_n + b

- **Applications:** Regression (Linear Regression) and classification (Logistic Regression).
- **Advantage:** Simple, fast, and interpretable.
- **Limitation:** Can't capture complex, non-linear patterns in data.

Multi-Layer Perceptron (MLP)

An MLP is a type of **feedforward artificial neural network** with one or more **hidden layers** between input and output layers.

- **Structure:** Input layer → Hidden Layer(s) → Output Layer
- **Each layer:** Performs weighted sum followed by a non-linear activation function (e.g., ReLU, sigmoid).

Output=f(Wx+b)\text{Output} = f(Wx + b)

- **Training:** Uses **Backpropagation** to minimize error using gradient descent.
- **Capabilities:** Can model non-linear relationships and complex patterns.

Key Points:

Concept	Description
Linear Model	Predicts output using a linear combination of inputs.
MLP	A neural network with hidden layers capable of learning non-linear functions.
Activation Function	Adds non-linearity (e.g., ReLU, sigmoid) to model complex relationships.
Backpropagation	Training algorithm to minimize error by adjusting weights using gradients.

Going Forwards in Multi-Layer Perceptrons (MLP)

"Going Forwards", also known as the **forward pass** or **forward propagation**, is a fundamental step in the operation of a neural network such as the Multi-Layer Perceptron (MLP). It refers to the process by which input data is passed through the network layer by layer to produce an output. This step occurs during both the training and inference phases of a neural network.

Overview of the Process

In a typical MLP, the network consists of an input layer, one or more hidden layers, and an output layer. Each layer contains several neurons, and each neuron in one layer is connected to all neurons in the subsequent layer. Every connection has an associated weight, and each neuron has a bias.

The **forward pass** begins at the input layer, where the input data is fed into the network. The input is then processed by each neuron in the subsequent layers using a weighted sum followed by an activation function.

Mathematically, for a given neuron, the operation is represented as:

z=w·x+bz = w \cdot x + b a=f(z)a = f(z)

Where:

- xx is the input vector
- ww is the weight vector
- bb is the bias
- zz is the weighted input
- f(z)f(z) is the activation function applied to zz, producing the output aa

This process is repeated through all the layers of the network, hence the term "going forwards."

Example of Forward Propagation in MLP

Consider a simple MLP with:

- 2 input neurons
- 1 hidden layer with 2 neurons
- 1 output neuron

Assume the input vector is [x1,x2][x_1, x_2]. The input is fed into the two neurons in the hidden layer. Each neuron calculates the weighted sum of the inputs, adds a bias, and applies an activation function (e.g., sigmoid or ReLU). The outputs from the hidden layer become inputs to the output neuron, which again performs a similar computation to produce the final output.

Activation Functions

Activation functions are applied after the weighted sum to introduce non-linearity into the network. Common activation functions include:

- **Sigmoid:** Squeezes input to a value between 0 and 1.
- **ReLU (Rectified Linear Unit):** Outputs zero for negative input and the input itself if positive.
- **Tanh:** Similar to sigmoid but outputs between -1 and 1.

These functions are essential for the network to learn complex patterns and make accurate predictions.

Purpose and Importance

The forward pass is crucial for determining how well the neural network is performing. During training, the output generated from the forward pass is compared to the actual target value, and the error (or loss) is computed. This error is then used in the backward pass (backpropagation) to update the weights and biases in the network.

In inference mode (after training is complete), the forward pass is used to make predictions on new, unseen data. The network uses the learned weights and biases to process the input and generate output predictions.

Efficiency Considerations

The forward pass is computationally intensive, especially in deep networks with many layers and neurons. Optimizations such as vectorized operations using libraries like NumPy or TensorFlow can speed up the computation.

Moreover, techniques like batch processing (processing multiple input examples at once) and using hardware acceleration (like GPUs) can significantly enhance the efficiency of forward propagation.

Multi-Layer Perceptron in Practice

The **Multi-Layer Perceptron (MLP)** is one of the most fundamental architectures in the field of deep learning and neural networks. While the theoretical concepts of MLP involve neurons, layers, and activation functions, its practical implementation introduces considerations related to model design, training, evaluation, and application. Understanding how to apply MLPs effectively in real-world scenarios is key to solving complex machine learning problems.

1. Architecture Design

In practice, an MLP consists of:

- **Input layer:** Receives raw data (e.g., image pixels, features from a dataset).
- **Hidden layers:** One or more layers where learning happens through weights, biases, and non-linear activations.

- **Output layer:** Produces final predictions or classifications.

Key architectural decisions include:

- **Number of layers:** More layers allow for learning deeper representations, but too many can cause overfitting or vanishing gradients.
- **Number of neurons per layer:** Depends on the complexity of the task. Too few may lead to underfitting; too many can make the model unnecessarily complex.
- **Activation functions:** Common choices are ReLU (for hidden layers), softmax (for classification outputs), or linear (for regression tasks).

2. Data Preprocessing

In practical applications, raw data must be prepared before feeding into an MLP:

- **Normalization/standardization:** Ensures that input features are on a similar scale, improving convergence speed and stability.
- **Handling missing values:** Techniques such as imputation or data cleansing may be needed.
- **Encoding categorical variables:** Use one-hot encoding or label encoding for non-numeric inputs.

3. Training an MLP

The training of an MLP involves:

- **Loss function:** Measures the error between predicted output and actual labels. Examples include:
  - Mean Squared Error (MSE) for regression.
  - Cross-Entropy Loss for classification.
- **Optimization algorithm:** Gradient descent and its variants (SGD, Adam, RMSprop) are used to minimize the loss by updating weights and biases.
- **Backpropagation:** A key algorithm that calculates the gradient of the loss function with respect to each weight using the chain rule.

4. Evaluation and Validation

To ensure the model generalizes well:

- **Split the dataset** into training, validation, and test sets.
- **Use metrics** like accuracy, precision, recall, F1-score (for classification), or RMSE (for regression).
- **Cross-validation** can be used to get a more reliable performance estimate.

5. Overfitting and Regularization

Overfitting occurs when the MLP performs well on training data but poorly on new data. Practical techniques to prevent overfitting include:

- **Dropout:** Randomly disables neurons during training to prevent reliance on specific nodes.
- **Early stopping:** Stops training when the validation loss stops improving.
- **L1/L2 Regularization:** Adds a penalty term to the loss function based on weights.

6. Applications of MLP

In real-world scenarios, MLPs are applied in:

- **Image recognition**
- **Speech and audio processing**
- **Natural Language Processing (NLP)**
- **Time-series forecasting**
- **Finance, healthcare, and customer behavior prediction**

While MLPs have been surpassed by more specialized models (like CNNs for images or RNNs for sequences), they remain powerful for general-purpose tasks.

Examples of Using the Multi-Layer Perceptron (MLP)

The **Multi-Layer Perceptron (MLP)** is a type of feedforward artificial neural network that maps sets of input data onto appropriate outputs. While it is a fundamental neural network architecture, MLPs are highly versatile and have been successfully applied across a broad range of applications in different industries. Below are several practical examples that illustrate the use of MLPs.

1. Handwritten Digit Recognition

One of the classic use cases of MLPs is the **MNIST digit recognition problem**, where the goal is to classify images of handwritten digits (0–9). Each digit image is 28x28 pixels, flattened into a 784-length input vector. The MLP is trained on thousands of these examples to learn patterns that differentiate digits based on stroke shapes, sizes, and curves.

- **Input Layer:** 784 neurons (one for each pixel).
- **Hidden Layers:** One or more layers with ReLU activation.
- **Output Layer:** 10 neurons (for digits 0–9) with softmax activation.

This example demonstrates how MLPs can perform well in image classification when input size is small and manageable.

2. Credit Risk Prediction in Banking

In finance, MLPs are used for **credit scoring**—predicting the likelihood that a loan applicant will default on payments. The input features may include income, credit history, age, employment status, and more.

- **Input Layer:** Contains numeric and categorical features.
- **Output Layer:** A single neuron with sigmoid activation for binary classification (approve/reject loan).

MLPs help in automating credit decisions with high accuracy and consistency, reducing manual workload and risk.

3. Medical Diagnosis

MLPs are applied in healthcare for **predictive diagnostics**, such as identifying whether a tumor is malignant or benign based on features like size, shape, and texture from medical imaging.

For example, in **breast cancer classification**, the model is trained using labeled data from biopsy results. MLPs help doctors make informed decisions by flagging high-risk cases for further review.

4. Stock Price Prediction

Although predicting the stock market is complex, MLPs are used for **time series forecasting** by learning patterns in historical stock data. Features may include daily open/close prices, trading volume, and technical indicators.

While MLPs don't consider sequence (unlike RNNs), they still perform decently on short-term forecasting when sequence length is small or pre-processed properly.

5. Spam Detection in Emails

In natural language processing (NLP), MLPs can be used to classify emails as spam or not spam. The input to the MLP consists of word count vectors or TF-IDF values representing email content.

- **Input Layer:** Feature vector of email terms.
- **Hidden Layers:** Capture non-linear patterns in text.
- **Output Layer:** Sigmoid function for binary classification.

This application is essential for filtering out unwanted messages and improving user experience.

6. Optical Character Recognition (OCR)

MLPs are also used in OCR systems to convert scanned documents into machine-encoded text. MLPs take feature vectors derived from character images and classify them into known alphabets or symbols.

Here is a detailed **500-word explanation** for the topic:

Overview – Deriving Backpropagation

Overview of Backpropagation

Backpropagation is a fundamental algorithm used for training **Multi-Layer Perceptrons (MLPs)** and other types of neural networks. It enables the network to learn from data by adjusting its weights and biases to minimize the error between the predicted output and the

actual target. This process is essential for **supervised learning**, where the goal is to learn a function that maps inputs to desired outputs.

The basic idea behind backpropagation is to **propagate the error backward** from the output layer to the input layer, using the **chain rule of calculus** to compute gradients. These gradients tell us how much each weight in the network contributes to the error. Using this information, we update the weights using optimization algorithms like **gradient descent**.

Structure of a Neural Network

Consider a simple feedforward neural network with:

- An **input layer**
- One or more **hidden layers**
- An **output layer**

Each layer is composed of **neurons**, and each neuron applies a **weighted sum** of inputs followed by an **activation function** (such as sigmoid, tanh, or ReLU).

The goal of training is to minimize a **loss function** (such as Mean Squared Error or Cross-Entropy Loss), which measures the difference between the predicted output and the actual label.

Deriving Backpropagation: Step-by-Step

Let's consider a single training example (x,y)(x, y), where xx is the input and yy is the target output. The forward pass computes the output y^\hat{y} of the network. The backpropagation algorithm then proceeds in the following steps:

1. Forward Pass

- For each layer ll, compute:

  z(l)=W(l)a(l−1)+b(l)z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)} a(l)=f(z(l))a^{(l)} = f(z^{(l)})

Where:

- W(l)W^{(l)}: Weight matrix at layer ll
- b(l)b^{(l)}: Bias vector
- a(l−1)a^{(l-1)}: Output from the previous layer (or input xx for the first layer)
- f()f(): Activation function (e.g., sigmoid)

2. Compute Loss

For example, using Mean Squared Error:

$L=12(y−y^)2L = \frac{1}{2} (y - \hat{y})^2$

3. Backward Pass: Compute Gradients

Let $\delta(l)$ be the error term for layer l. For the output layer:

$\delta(L)=(a(L)−y)\cdot f'(z(L))\delta^{(L)} = (a^{(L)} - y) \cdot f'(z^{(L)})$

For hidden layers:

$\delta(l)=(W(l+1)T\delta(l+1))\cdot f'(z(l))\delta^{(l)} = \left(W^{(l+1)T} \delta^{(l+1)}\right) \cdot f'(z^{(l)})$

Here, $f'(z)$ is the derivative of the activation function.

4. Update Weights and Biases

Use the gradients to update weights and biases:

$W(l):=W(l)−\eta\cdot\delta(l)\cdot a(l−1)TW^{(l)} := W^{(l)} - \eta \cdot \delta^{(l)} \cdot a^{(l-1)T}$
$b(l):=b(l)−\eta\cdot\delta(l)b^{(l)} := b^{(l)} - \eta \cdot \delta^{(l)}$

Where η is the learning rate.

Benefits and Importance

- **Efficient:** Backpropagation is computationally efficient for deep networks.
- **Scalable:** Works for networks with many layers and millions of parameters.
- **Widely used:** It is the foundation for training deep neural networks.

Radial Basis Functions and Splines

Radial Basis Functions (RBFs) and **splines** are powerful mathematical tools used for function approximation, interpolation, and pattern recognition. These techniques are especially useful when we want to fit a model to scattered, multidimensional data. Both are used to create models that can generalize well to unseen data by learning smooth functions from examples.

1. Radial Basis Functions (RBFs)

A **Radial Basis Function** is a real-valued function whose output depends only on the distance from a center point. That is, the function is radially symmetric and can be written as:

ϕ(‖x−c‖)\phi(\|x - c\|)

Where:

- xx is the input vector,
- cc is the center,
- ‖·‖\|\cdot\| is typically the Euclidean norm.

The most commonly used RBF is the **Gaussian function**:

ϕ(x)=e−β‖x−c‖2\phi(x) = e^{-\beta \|x - c\|^2}

Where β\beta is a parameter controlling the spread or "width" of the function.

RBF Networks

An **RBF network** is a type of artificial neural network that uses radial basis functions as activation functions. It typically consists of three layers:

1. **Input Layer**: Passes the input features.
2. **Hidden Layer**: Applies RBFs centered at specific points.
3. **Output Layer**: Produces a weighted sum of RBF outputs.

Training an RBF network involves:

- Selecting centers (either randomly or using clustering like K-means),
- Determining the spread (β\beta),
- Training the weights (usually via least squares or gradient descent).

Advantages of RBFs

- RBF networks learn very fast.
- They are universal approximators—able to model any continuous function with enough basis functions.
- They are especially effective for **interpolation** problems in high dimensions.

2. Splines

Splines are piecewise polynomial functions used in approximation and interpolation tasks. They are particularly useful when we want to model a smooth function that passes through or near a given set of data points.

The most common type is the **cubic spline**, which connects a set of points using cubic polynomials in such a way that the overall function is smooth (continuously differentiable) at the joints (called knots).

Types of Splines

- **Linear spline**: Piecewise linear.
- **Quadratic/Cubic spline**: Higher-order polynomials for smoother approximations.
- **B-splines** (Basis splines): Generalize splines using basis functions, useful for large datasets.
- **Thin-plate splines**: A specific kind of RBF used for multidimensional interpolation.

Applications of Splines

- Curve fitting and smoothing in statistics.
- Path planning in robotics.
- Animation in computer graphics.
- Image morphing and transformation.

RBFs vs. Splines

Feature	Radial Basis Functions	Splines
Basis	Distance from center	Piecewise polynomial
Smoothness	Controlled by RBF width	Controlled by continuity at knots
Use in ML	Neural networks, interpolation	Regression, smoothing, graphics
Dimensionality	Good for high-dimensional data	More common in 1D–3D data

Concepts – RBF Network

A **Radial Basis Function (RBF) Network** is a type of **artificial neural network** that uses **radial basis functions** as activation functions. These networks are especially effective for tasks involving **function approximation**, **pattern recognition**, and **interpolation**. RBF networks are considered **feedforward** networks and are generally simpler and faster to train compared to deep networks.

1. Core Concept

At the heart of an RBF network lies the idea of mapping inputs into a **higher-dimensional space** using **radial basis functions**, then performing a **linear combination** of these transformed inputs to produce the output.

The general structure is:

- **Input Layer**: Receives the feature vector.
- **Hidden Layer**: Applies RBFs (commonly Gaussian) to each input. Each neuron in this layer computes the distance between the input and a "center" and transforms it using the RBF.
- **Output Layer**: Combines the weighted outputs of the hidden layer linearly to produce the final result.

2. Mathematical Model

Given an input vector xx, the output of an RBF network is:

$$y(x) = \sum_{i=1}^{N} w_i \cdot \phi(\|x - c_i\|)$$

Where:

- ϕ is the radial basis function (e.g., Gaussian: $\phi(r) = e^{-\beta r^2}$),
- c_i is the **center** of the i^{th} RBF unit,
- w_i is the **weight** applied to the i^{th} RBF unit output,
- N is the number of RBF units in the hidden layer.

3. Components of an RBF Network

a. Centers (c_i)

- Typically determined by clustering methods like **K-Means**.
- Represent reference points in input space where the RBF is centered.

b. Spread or Width Parameter (β)

- Controls the "sharpness" of the RBF.
- A large β makes the function flatter; a small β makes it peakier.

c. Weights (w_i)

- Learned using **linear regression**, **least squares**, or **gradient descent** after the centers and spread are fixed.

4. Training an RBF Network

Training usually occurs in two stages:

1. **Unsupervised phase**: Determine RBF centers (e.g., with K-means clustering).
2. **Supervised phase**: Learn output layer weights using methods like least squares.

5. Applications

- **Function approximation**
- **Time-series prediction**
- **Pattern classification**
- **Image and signal processing**
- **Control systems**

6. Advantages

- **Fast training** due to linear output layer.
- **Good generalization** for many practical problems.

- **Universal approximator**: With enough RBF units, it can approximate any continuous function.

7. Limitations

- Choosing the **right number of RBF units** is critical.
- Sensitive to **choice of centers and spread**.
- Performance may degrade in **high-dimensional spaces** due to the **curse of dimensionality**.

Curse of Dimensionality – Explained

The **Curse of Dimensionality** refers to a set of problems that arise when working with data in **high-dimensional spaces**. As the number of dimensions (features) increases, the data becomes **increasingly sparse**, and many algorithms that work well in low dimensions become **inefficient**, **unreliable**, or **computationally expensive**.

Why Is It Called a "Curse"?

Because increasing the number of dimensions:

- Exponentially **increases computation time**,
- **Decreases model performance** due to sparse data,
- Makes it harder to find meaningful patterns or clusters.

The "curse" implies that beyond a certain point, **adding more features** may hurt rather than help.

Intuition with Example

Imagine you're trying to fill a line (1D), a square (2D), and a cube (3D) with points.

- In **1D**, you can cover the space fairly quickly.
- In **2D**, you need many more points.
- In **3D**, the number of points required **explodes**.

So, in **100 dimensions**, the volume becomes enormous, and data points are **extremely far apart**, making clustering or learning difficult.

Impact on Machine Learning

1. **Distance Metrics Become Less Meaningful**
    - In high dimensions, the **difference between the nearest and farthest neighbor** becomes negligible.
    - Algorithms like **k-NN** and **RBF Networks** that rely on distance lose effectiveness.

2. **Overfitting**
    - With more features, models can fit the noise instead of the actual pattern, leading to poor generalization.
3. **Need for More Data**
    - The number of training samples required to maintain the same performance **grows exponentially** with the number of dimensions.
4. **Model Complexity Increases**
    - The computational cost for searching, clustering, or classifying in high dimensions is extremely high.

Techniques to Combat the Curse

Method	Description
Feature Selection	Choosing only the most relevant features.
Dimensionality Reduction	Using techniques like PCA (Principal Component Analysis), t-SNE, or Autoencoders to reduce feature space.
Regularization	Penalizing complex models to prevent overfitting (e.g., L1/L2 regularization).
Using Simpler Models	Models with fewer parameters tend to generalize better in high-dimensional spaces.

In the Context of RBF Networks

In RBF networks:

- The **Euclidean distance** between the input and the center is critical.
- In high-dimensional input spaces, these distances become unreliable, and the network struggles to learn effective representations.
- Hence, **dimensionality reduction** is often applied before training RBFs.

Interpolations and Basis Functions – Explained

In the context of machine learning and approximation theory, **interpolations** and **basis functions** are foundational concepts used to approximate complex functions or datasets. They are especially relevant in models such as **Radial Basis Function (RBF) networks**, **splines**, and other regression techniques.

What is Interpolation?

Interpolation is the process of estimating unknown values that fall **between known data points**.

Example:

If we know that:

- $f(1)=3$
- $f(3)=7$

Interpolation estimates f(2)f(2), a value between 1 and 3.

There are many interpolation techniques:

- **Linear interpolation**
- **Polynomial interpolation**
- **Spline interpolation**
- **RBF interpolation**

Each has trade-offs in terms of **smoothness**, **accuracy**, and **computational efficiency**.

What are Basis Functions?

A **basis function** is a building block used to construct more complex functions. In approximation and interpolation, a **set of basis functions** is combined (usually linearly) to represent the target function.

Mathematically:

f(x)=∑i=1nwi⋅ϕi(x)f(x) = \sum_{i=1}^{n} w_i \cdot \phi_i(x)

Where:

- ϕi(x)\phi_i(x) are the basis functions,
- wiw_i are weights or coefficients,
- nn is the number of basis functions.

Why Use Basis Functions?

- They allow **flexible modeling** of complex patterns.
- You can choose different basis functions depending on the nature of the data.
- When using **RBFs** or **splines**, the basis functions determine the **shape** and **smoothness** of the interpolation.

Common Types of Basis Functions

Type	Description	Example
Polynomial	Use polynomials like x,x2,x3,…x, x^2, x^3, \ldots	Used in polynomial regression
Trigonometric	Use sine and cosine functions	Used in Fourier analysis
Radial Basis Functions (RBF)	Centered functions, typically Gaussian	ϕ(r)=e−βr2\phi(r) = e^{-\beta r^2}
Splines	Piecewise polynomial functions	Cubic splines are common

Type	Description	Example
Wavelets	Localized wave-like basis	Used in signal processing

Interpolation Using Basis Functions

To interpolate using basis functions:

1. **Choose basis functions** appropriate to your data.
2. **Set up equations** so that the function passes through the known data points.
3. **Solve for weights wiw_i**.
4. Use the resulting function to predict intermediate values.

For example, in **RBF interpolation**, we place a Gaussian at each data point and calculate the weighted sum to estimate values in between.

Applications

- **Curve fitting**
- **Signal reconstruction**
- **Image interpolation**
- **Function approximation**
- **Machine learning models (e.g., RBF networks)**

Support Vector Machines (SVM) – Explained

Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms used for **classification** and **regression** tasks. They are particularly effective in high-dimensional spaces and are known for their robustness and accuracy, especially in binary classification problems.

Core Idea Behind SVM

SVMs work by finding a **hyperplane** (a decision boundary) that best separates data points of different classes with the **maximum margin**.

Intuition:

- Imagine two classes of points in a 2D plane.
- SVM tries to draw a **line (or hyperplane in higher dimensions)** between these classes such that:
    - The margin between the two classes is as **wide as possible**.
    - The closest points to the hyperplane (called **support vectors**) are maximally distant from it.

Key Concepts in SVM

Concept	Description
Hyperplane	The decision boundary separating different classes.
Margin	The distance between the hyperplane and the nearest data point of any class.
Support Vectors	The data points closest to the hyperplane; these define the position of the hyperplane.
Kernel Trick	A technique used to transform non-linearly separable data into higher dimensions where it becomes linearly separable.

Mathematical Formulation

Given training data points (x1,y1),(x2,y2),…,(xn,yn)(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) where yi∈{−1,1}y_i \in \{-1, 1\}, SVM tries to solve the following optimization problem:

minw,b12‖w‖2subject to yi(w·xi+b)≥1\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{subject to } y_i(\mathbf{w} \cdot x_i + b) \geq 1

This means it wants to:

- Minimize the norm (magnitude) of the weight vector w\mathbf{w},
- Ensure each data point is correctly classified with a margin of at least 1.

Kernel Functions in SVM

SVMs can perform **non-linear classification** using kernels, which implicitly map input data into higher-dimensional spaces without explicitly computing coordinates in that space.

Common Kernels:

Kernel Type	Function
Linear	K(x,x′)=x·x′K(x, x') = x \cdot x'
Polynomial	K(x,x′)=(x·x′+c)dK(x, x') = (x \cdot x' + c)^d
RBF (Gaussian)	K(x,x′)=exp(−γ‖x−x′‖2)K(x, x') = \exp(-\gamma |x - x'|^2)
Sigmoid	K(x,x′)=tanh(αx·x′+c)K(x, x') = \tanh(\alpha x \cdot x' + c)

Advantages of SVM

- Works well in **high-dimensional** spaces.
- Effective even when the number of dimensions exceeds the number of samples.
- Uses a **subset of training points** (support vectors), so it’s memory efficient.

- Offers **robust performance** with a well-defined mathematical foundation.

Limitations of SVM

- Poor performance on large datasets due to high training time.
- Less effective when data is heavily **noisy** or **overlapping**.
- Choosing the right **kernel** and **parameters** (like CC, γ\gamma) can be challenging.

Applications of SVM

- Text classification (e.g., spam detection)
- Image classification
- Handwriting recognition
- Bioinformatics (e.g., protein classification)
- Face detection