



ANNAMACHARYA UNIVERSITY

Estd. under Andhra Pradesh
Private Universities (Establishment and Regulation) Act, 2016

(University listed in UGC as per section 2(f) of the UGC Act, 1956)

INTRODUCTION TO DEEP LEARNING

Presented By

K.Reddemma

Assistant Professor,

Artificial Intelligence and Machine Learning

Annamacharya University,

Rajampet.



ANNAMACHARYA UNIVERSITY

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016)

(UNIVERSITY LISTED IN UGC AS PER THE SECTION 2(f) OF THE UGC ACT, 1956)

RAJAMPET, Annamayya District, AP – 516126, INDIA

Title of the Course	:	INTRODUCTION TO DEEP LEARNING			
Category	:	PC			
Couse Code	:	24ACAI52T			
Year	:	III			
Semester	:	I			
Branch	:	CSE(AI)			
Lecture Hours		Tutorial Hours	Practice Hours	Credits	Total Hours
3		0	0	3	51

Course Objectives:

- Demonstrate the major technology trends driving Deep Learning
- Build, train, and apply fully connected deep neural networks
- Implement efficient (vectorized) neural networks
- Analyse the key parameters and hyper parameters in a neural network's architecture

Course Outcomes

1. Demonstrate the mathematical foundation of neural network
2. Describe the machine learning basics
3. Differentiate architecture of deep neural network
4. Build a convolution neural network
5. Build and train RNN and LSTMs

UNIT – 1 LINEAR ALGEBRA

10

Linear Algebra: Scalars, Vectors, Matrices and Tensors, Matrix operations, types of matrices, Norms, Eigen decomposition, Singular Value Decomposition, Principal Components Analysis. Probability and Information Theory: Random Variables, Probability Distributions, Marginal Probability, Conditional Probability, Expectation, Variance and Covariance, Bayes' Rule, Information Theory. Numerical Computation: Overflow and Underflow, Gradient-Based Optimization, Constrained Optimization, Linear Least Squares.

UNIT – 2 MACHINE LEARNING

10

Machine Learning: Basics and Under fitting, Hyper parameters and Validation Sets, Estimators, Bias and Variance, Maximum Likelihood, Bayesian Statistics, Supervised and Unsupervised Learning, Stochastic Gradient Descent, Challenges Motivating Deep Learning. Deep Feed forward Networks: Learning XOR, Gradient-Based Learning, Hidden Units, Architecture Design, Back-Propagation and other Differentiation Algorithms.

UNIT – 3 REGULARIZATION FOR DEEP LEARNING

9

Regularization for Deep Learning: Parameter Norm Penalties, Norm Penalties as Constrained Optimization, Regularization and Under-Constrained Problems, Dataset Augmentation, Noise Robustness, Semi-Supervised Learning, Multi-Task Learning, Early Stopping, Parameter Tying and Parameter Sharing, Sparse Representations, Bagging and Other Ensemble Methods, Dropout, Adversarial Training, Tangent Distance, Tangent Prop and Manifold Tangent Classifier.

UNIT – 4 CONVOLUTION NETWORKS

10

Convolution Networks: The Convolution Operation, Pooling, Convolution, Basic Convolution Functions, Structured Outputs, Data Types, Efficient Convolution Algorithms, Random or Unsupervised Features, Basis for Convolution Networks.



Sequence Modelling: Recurrent and Recursive Nets: Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive Neural Networks, Echo State Networks, LSTM, Gated RNNs, Optimization for Long-Term Dependencies, Auto encoders, Deep Generative Models.

TEXT BOOK:

1. Ian Good fellow, Joshua Bagnio, Aaron Carville, —Deep Learning, MIT Press, 2016.

REFERENCE BOOKS:

1. Josh Patterson and Adam Gibson, —Deep learning: A practitioner's approach||, O'Reilly Media, First Edition, 2017
2. 1Fundamentals of Deep Learning, Designing next-generation machine intelligence algorithms, Nikhil Baume, O'Reilly, Sheriff Publishers, 2019.
3. Deep learning Cook Book, Practical recipes to get started Quickly, Douse Using, O'Reilly, Sheriff Publishers, 2019

ONLINE LEARNING RESOURCES

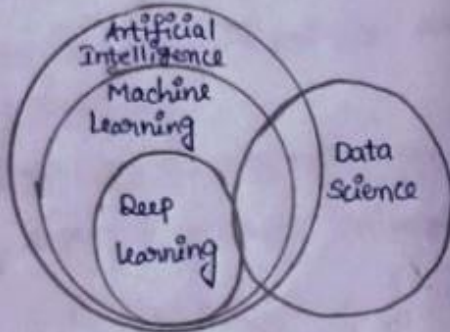
1. <https://keras.io/datasets/>
2. <http://deeplearning.net/tutorial/deeplearning.pdf>
3. <https://arxiv.org/pdf/1404.7828v4.pdf>
4. <https://www.cse.iitm.ac.in/~miteshk/CS7015.html>
5. <https://www.deeplearningbook.org>
6. <https://nptel.ac.in/courses/106105215>

CO-PO MAPPING

Course Outcomes	Engineering Knowledge	Problem Analysis	Design/Development of solutions	Conduct investigations of complex problems	Engineering tool usage	The engineer and the world	Ethics	Individual and Collaborative Team work	Communication	Project management and finance	Life-long learning
24ACAI52T.1	3	3	2	2	2	-	-	-	-	2	1
24ACAI52T.2	3	2	2	2	2	-	-	-	-	2	1
24ACAI52T.3	3	2	3	2	3	-	-	-	-	2	2
24ACAI52T.4	3	2	3	3	3	-	-	-	-	2	2
24ACAI52T.5	3	2	3	3	3	-	-	-	-	2	2

Introduction to Deep learning

What is Deep learning :



Deep learning is a subset of machine learning that uses multilayered neural networks, known as deep neural networks, to simulate the complex decision-making power of the human brain. (or)

Deep learning is a branch of Artificial Intelligence that uses multi-layered neural network to automatically learn representations and features from raw data like images, audio or text to perform tasks such as classification, prediction and pattern recognition.

* Structure of Deep learning Models :

1. Input layer - Accepts raw data - ex: pixel values of an image
2. Hidden layer - Multiple layers that process input data through weighted connections and activation functions.
3. Output layer - produces the final prediction - ex: class label (or) numerical output

* popular Deep learning Architectures :

1. Feedforward Neural Network (FNN) - Basic pattern recognition
2. Convolutional Neural Network (CNN) - Image and video processing
3. Recurrent Neural Networks (RNN) - Sequence Data
4. Long-short-Term Memory (LSTM) - Time-series Data
5. Autoencoders - Data compression & denoising
6. Generative Adversarial Networks (GAN) - Data generation.

Features of Deep Learning

1. Automatic Feature Extraction
2. Non-linear processing
3. End-to-End learning
4. Scalability

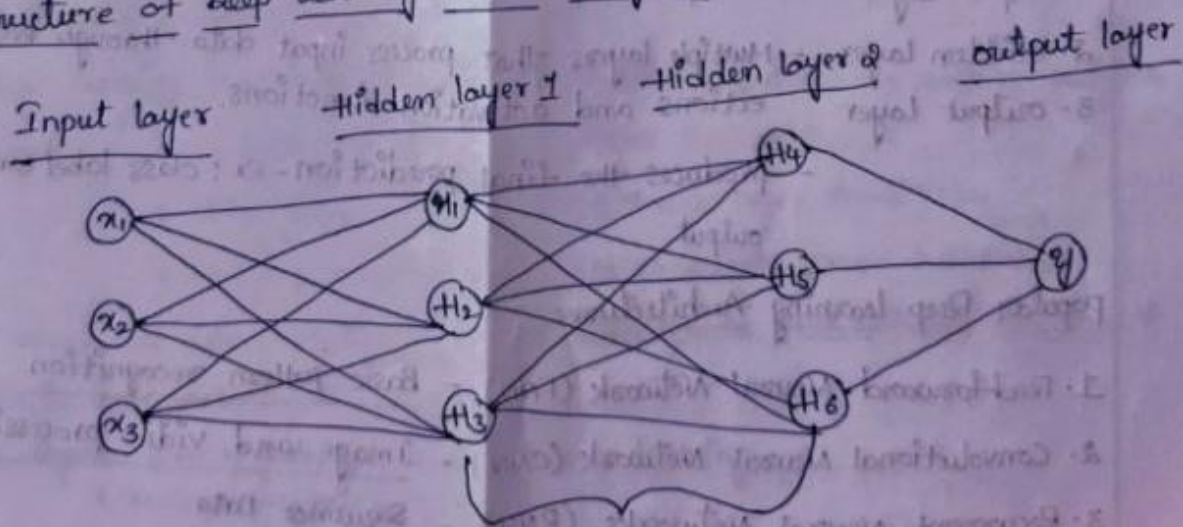
Applications

1. Computer vision
2. Natural Language processing
3. Speech Recognition
4. Healthcare
5. Finance
6. Entertainment

Advantages :

1. Automatic Feature Extraction
2. High accuracy for complex problems
3. Reduce Human Intervention

Structure of Deep learning models diagram :



* Linear Algebra

1. Scalars
2. Vectors
3. Matrices
4. Tensors
5. Matrix operations
6. Types of Matrices
7. Norms
8. Eigen Decomposition
9. Singular value Decomposition
10. principal components Analysis

* What is Linear Algebra :

Linear Algebra is the foundation of Deep learning and Machine learning. It provides the mathematical framework to represent and manipulate data using vectors, matrices, and tensors. Every deep learning model from simple neural networks to large-scale transformers relies on linear algebra operations for computation.

Linear Algebra is a branch of (mathematical) mathematics that deals with linear equations, vectors and matrices and how they transform data. In deep learning, it is used to represent inputs, weights, activations and gradients efficiently.

* For Example :

1. An image $\rightarrow$ Matrix of pixel values
2. A Sentence (in NLP) $\rightarrow$ Vector of embeddings
3. Neural Network weights $\rightarrow$ Matrices connecting layers.

* Why Linear Algebra is important in Deep learning

Deep learning models perform millions of operations involving

1. Dot products $\rightarrow$ for similarity and weight sums
2. Matrix Multiplications $\rightarrow$ for layer Transformations
3. Vector addition/subtraction $\rightarrow$ for bias adjustments.
4. Tensors reshaping and transposition $\rightarrow$ for multidimensional data.

Linear Algebra makes these operations mathematically precise and computationally efficient using optimized libraries like Numpy, Tensorflows, and Pytorch.

* Core Linear Algebra concepts used in Deep Learning :

- a. Scalars
- b. Vectors
- c. Matrices
- d. Tensors
- e. Transpose
- f. Identity Matrices (Matrix)
- g. Inverse of a Matrix
- h. Determinant and Eigenvalues

These are used in principal component analysis, weight initialization, and stability analysis.

* How Linear Algebra powers Neural Networks :

Every layer in a Neural Network performs

$$y = f(Wx + b)$$

Where : x = Input vector (features)

W = Weight matrix

b = Bias vector

f = Activation function $\rightarrow$ like (ReLU, sigmoid)

Each forward pass involves matrix multiplication, addition and non-linear activation - all built from linear algebra.

* Linear Algebra in Backpropagation :

During Training, deep learning models use backpropagation to adjust weights.

This involves

- * Computing gradients $\rightarrow$ partial derivatives

- * Using Jacobian matrices to track layer sensitivities

- * Applying matrix calculus to propagate error backward.

All these are direct applications of Linear Algebra and calculus combined.

* Geometric Interpretation :

Linear Algebra gives a geometric view of deep learning.

- * Vectors $\rightarrow$ points in space.

- * Matrices $\rightarrow$ Linear Algebra transformations (Rotation, Scaling, Translation)

- * Neural Networks $\rightarrow$ sequences of transformations mapping input space to output space.

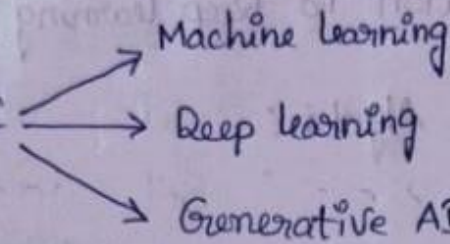
Each layer transforms input data to a new representation that helps in classification (or) prediction.

Introduction of Deep learning :

* What is AI

* What is ML

* Types of AI



* Example Applications

* What is Artificial Intelligence (AI) :

Artificial Intelligence (AI) is a technology that enables machines and computers to perform tasks that typically require human intelligence. It helps systems learn from data, recognize patterns and make decisions to solve complex problems. It is used in healthcare, finance, e-commerce and transportation offering personalized recommendations and enabling self driving cars.

* Core Concepts of AI : 1. Machine learning (ML)

2. Generative AI

3. Natural Language processing (NLP)

4. Expert Systems

Working of Artificial Intelligence :

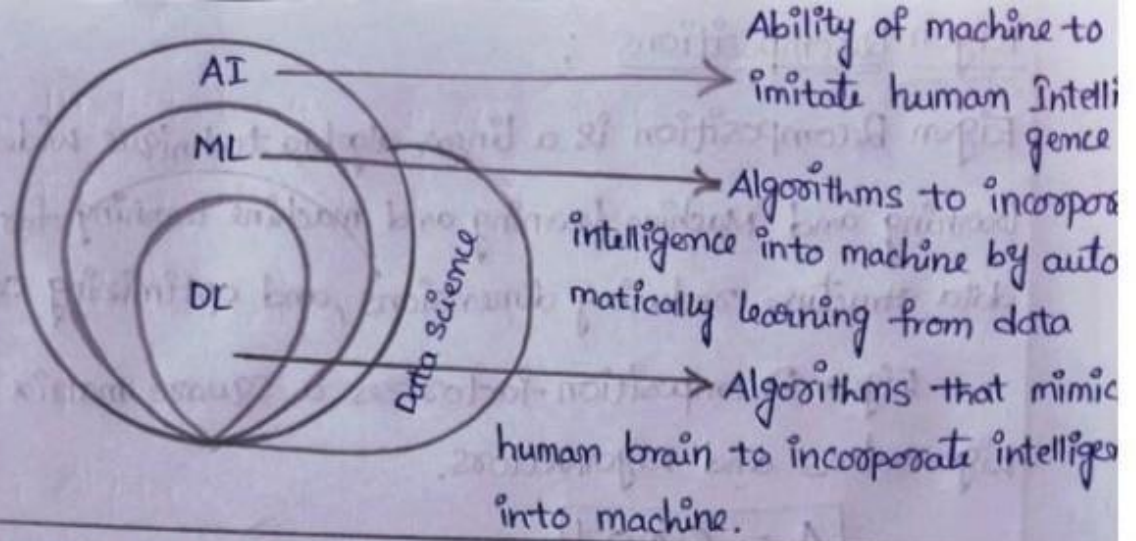
1. Data collection

2. Processing And learning

3. Model Training

4. Decision Making

5. Feedback and Improvements



What is Machine Learning :

Machine learning is a branch of AI that enables algorithms to uncover hidden (layers) patterns within datasets. It allows them to predict new, similar data without explicit programming for each task.

Artificial Intelligence, Machine learning, Deep learning does :

* Artificial Intelligence : Artificial Intelligence is the broad concept of making machines behave like humans.

AI systems :

1. Think logically
2. Make decisions
3. Solve problems
4. Mimic human intelligence

Examples :

1. Chatbots
2. Expert systems
3. Game-playing programs (chess AI)
4. Voice assistants

* Machine Learning :

1. Learn patterns from data
2. Improve performance automatically
3. Make predictions (or) classifications

Examples :

1. Spam email detection
2. Recommendation systems (Amazon, Netflix)
3. Student result prediction

* Deep Learning :

1. Learn complex features automatically
2. Work well with large data
3. Handle images, audio, and text effectively.

Examples :

1. Face Recognition
2. Speech Recognition
3. Self-driving cars
4. Medical image diagnosis

What are Norms in Deep Learning (DL)

Definition

In **Deep Learning**, a **norm** is a mathematical function used to **measure the size, length, or magnitude** of vectors, matrices, or tensors.

Norms help quantify **how large weights, inputs, gradients, or errors** are in a neural network.

In simple words:

Norm = "How big is this vector or matrix?"

Why Norms are Important in Deep Learning

Norms are widely used in DL for:

1. **Regularization** (to avoid overfitting)
 2. **Weight decay**
 3. **Gradient clipping**
 4. **Loss function design**
 5. **Measuring distances & similarities**
 6. **Controlling model complexity**
-

◆ General Mathematical Definition

A function $\| \cdot \|$ is a **norm** if it satisfies:

1. **Non-negativity**

$$\|x\| \geq 0$$

2. **Zero vector property**

$$\|x\| = 0 \iff x = 0$$

3. **Scaling (homogeneity)**

$$\|\alpha x\| = |\alpha| \|x\|$$

4. **Triangle inequality**

$$\|x + y\| \leq \|x\| + \|y\|$$

Manhattan distance measures the distance between two points as the sum of the absolute differences of their coordinates.

It is called *Manhattan* distance because it resembles moving along city blocks (only horizontal and vertical moves).

Mathematical Formula

For two points (vectors)

$$\mathbf{x} = (x_1, x_2, \dots, x_n), \quad \mathbf{y} = (y_1, y_2, \dots, y_n)$$

$$D_{\text{Manhattan}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n |x_i - y_i|$$

This is exactly the L1 norm of the difference vector:

$$\|\mathbf{x} - \mathbf{y}\|_1$$

Numerical Problems (Step-by-Step)

• Example 1: 2D Points

Let

$$A = (2, 3), \quad B = (7, 1)$$

Step 1: Take absolute differences

$$|2 - 7| = 5, \quad |3 - 1| = 2$$

Step 2: Add them

$$D = 5 + 2 = \boxed{7}$$

✅ Manhattan distance = 7

Euclidean distance is the **straight-line distance** between two points (vectors) in Euclidean space. In **deep learning**, it is mainly used to **measure similarity or dissimilarity** between data points, feature vectors, or embeddings.

It is also called **L2 distance**.

Use in DL

- L1 Regularization

- Encourages **sparsity** (many weights become zero)
- Feature selection

Numerical Problems (Step-by-Step)

♦ Example 1: 2D Points

$$A = (2, 3), \quad B = (6, 7)$$

Step 1: Differences

$$(2 - 6) = -4, \quad (3 - 7) = -4$$

Step 2: Squares

$$(-4)^2 = 16, \quad (-4)^2 = 16$$

Step 3: Sum & square root

$$\sqrt{16 + 16} = \sqrt{32} = \boxed{5.66}$$

L2 Norm (Euclidean Norm)

✦ Definition

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

✦ Example

$$x = [1, 2, 2]$$

$$\|x\|_2 = \sqrt{1 + 4 + 4} = 3$$

Use in DL

- **L2 Regularization (Weight Decay)**
- Prevents very large weights
- Produces smoother models

3 L_∞ Norm (Maximum Norm)

✦ Definition

$$\|x\|_\infty = \max(|x_i|)$$

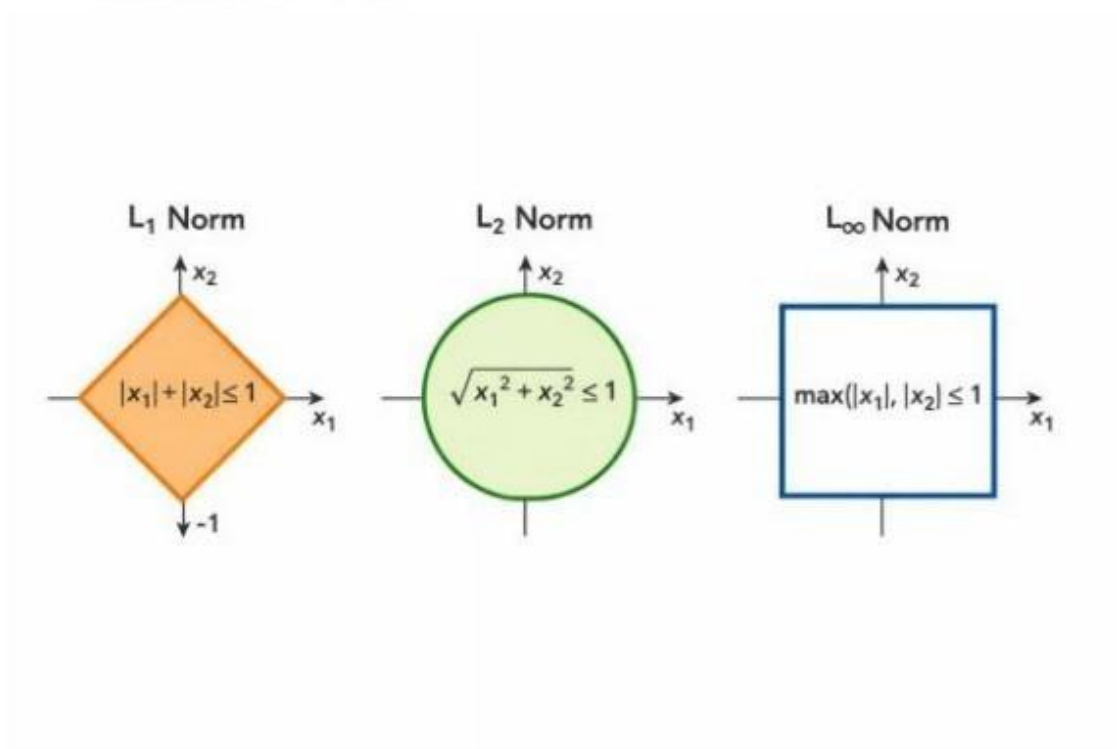
✦ Example

$$x = [-5, 3, -7, 2]$$

$$\|x\|_\infty = 7$$

Use in DL

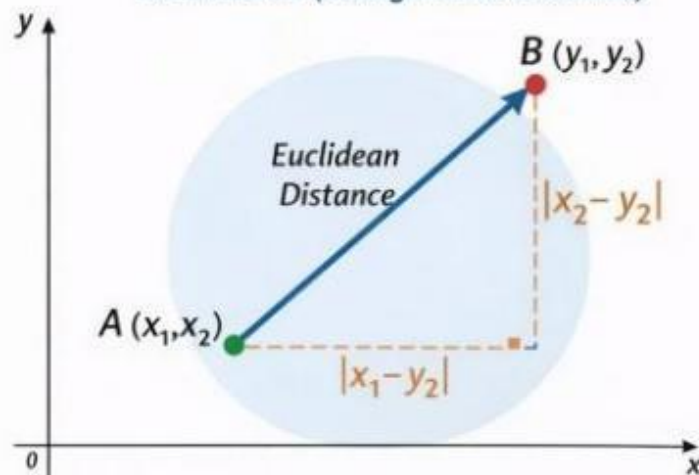
- Gradient clipping
- Stability in RNNs and LSTMs



Euclidean Distance

$$D_{\text{Euclidean}}(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

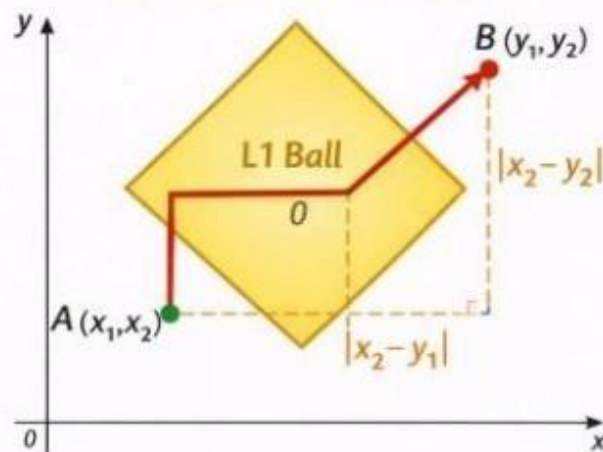
L2 Distance (Straight-Line Distance)



Manhattan Distance

$$D_{\text{Manhattan}}(x, y) = \sum_{i=1}^n |x_i - y_i|$$

L1 Distance (Taxicab Distance)



◆ L1 NORMALIZATION (Manhattan Normalization)

✔ Definition

L1 normalization scales a vector so that the **sum of absolute values** = 1.

$$\hat{x}_i = \frac{x_i}{\sum |x_i|}$$

📅 Problem 1: L1 Normalization (Basic)

Given vector:

$$\mathbf{x} = [2, -3, 5]$$

Step 1: Compute L1 norm

$$\|\mathbf{x}\|_1 = |2| + |-3| + |5| = 2 + 3 + 5 = 10$$

Step 2: Divide each element

$$\hat{\mathbf{x}} = \left[\frac{2}{10}, \frac{-3}{10}, \frac{5}{10} \right]$$

✔ Final Answer

$$[0.2, -0.3, 0.5]$$

✔ Sum of absolute values = 1

◆ L2 NORMALIZATION (Euclidean Normalization)

✔ Definition

L2 normalization scales a vector so that its Euclidean length = 1.

$$\hat{x}_i = \frac{x_i}{\sqrt{\sum x_i^2}}$$

📋 Problem 3: L2 Normalization (Basic)

Given vector:

$$\mathbf{x} = [3, 4]$$

Step 1: Compute L2 norm

$$\|\mathbf{x}\|_2 = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

Step 2: Normalize

$$\hat{\mathbf{x}} = \left[\frac{3}{5}, \frac{4}{5} \right]$$

✔ Final Answer

$$\boxed{[0.6, 0.8]}$$

✔ Vector length = 1

Eigen Decompositions :

Eigen Decomposition is a linear algebra technique widely used in deep learning and machine learning for understanding data structure, reducing dimensions, and optimizing computations.

Eigen Decomposition factorizes a square matrix into its Eigenvalues and Eigenvectors.

$$A = Q \Lambda Q^{-1} \quad (\text{or}) \quad A = V \Lambda V^{-1}$$

Where : $A \rightarrow$ Square matrix

$V, Q \rightarrow$ Matrix of Eigenvectors

$\Lambda \rightarrow$ Diagonal matrix of Eigenvalues

$Q^{-1} \rightarrow$ Inverse of Eigenvector matrix

Simple Words : To Break down a square matrix into simpler components called Eigenvalues and Eigenvectors. This process helps us understand how a matrix behaves and how it transforms data.

How to perform Eigen decomposition :

To perform Eigen decomposition on a matrix, follow these steps

Step 1 : Find the Eigenvalues

$$\det [A - \lambda I] = 0$$

Where : A is the Square matrix

λ is the Eigenvalue

I is the identity matrix of the same dimension as A .

Step 2 : Find the Eigenvectors

For each eigenvalue λ , substitute it back into the equation $(A - \lambda I)V = 0$

Step 3 : Construct the Eigenvector Matrix V

place all the Eigenvectors as columns in the matrix V . If there are n distinct Eigenvalues, V will be an $n \times n$ matrix.

Step 4 : Form the Diagonal Matrix Λ

Construct a diagonal matrix Λ by placing the eigenvalues on its diagonal

Step 5 : Calculate the Inverse of V

Find V^{-1} , the inverse of the Eigenvector matrix V , if the matrix is invertible.

Example problem :

Define the matrix $A = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$

Find the Eigenvalues by solving $\det(A - \lambda I) = 0$

The characteristic equation is : $A - \lambda I = 0$

$$\begin{vmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{vmatrix} = 0$$

$$(4-\lambda)(3-\lambda) - 2(1) = 0$$

$$\lambda^2 - 7\lambda + 10 = 0$$

$$\lambda_1 = 5 \text{ and } \lambda_2 = 2$$

Find Eigenvectors corresponding to each Eigenvalue.

For $\lambda_1 = 5$, solve $(A - 5I)V = 0$

$$\begin{bmatrix} -1 & 1 \\ 2 & -2 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

$$v_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

For $\lambda_2 = 2$, solve: $(A - 2I)v = 0$

$$\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} = 0 \Rightarrow \begin{vmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{vmatrix} = 0 \Rightarrow \begin{vmatrix} 4-2 & 1 \\ 2 & 3-2 \end{vmatrix} = 0 \Rightarrow \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

$$v_2 = \begin{bmatrix} -\frac{1}{2} \\ 1 \end{bmatrix}$$

Form the matrix V of eigenvectors and Λ of eigenvalues

$$V = \begin{bmatrix} 1 & -\frac{1}{2} \\ 1 & 1 \end{bmatrix}, \quad \Lambda = \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}$$

perform the Eigen decomposition

$$A = V\Lambda V^{-1}$$

optionally compute the inverse of V

$$V^{-1} = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & \frac{2}{3} \end{bmatrix}$$

Importance of Eigen decomposition:

Eigen decomposition is widely used because it makes complex tasks simpler.

1. Simplifying Matrix power
2. Data simplification
3. physics
4. Image processing.

Eigen values calculation :

Eigenvalues for a square matrix A , a scalar λ is called an eigenvalue if $AV = \lambda V$.

Where : $V \neq 0$ is the Eigenvector

λ is the Eigenvalue

Formula : $\det(A - \lambda I) = 0$

Where : A = Given Matrix

I = Identity matrix

λ = Eigen values

Steps :

Step 1 : Check Matrix

Matrix must be square ($2 \times 2, 3 \times 3, \dots$)

Step 2 : Write $A - \lambda I$

Subtract λ from diagonal elements.

Step 3 : Find determinant

compute : $\det(A - \lambda I)$

Step 4 : Solve the polynomial

solve the equation $\det(A - \lambda I) = 0$

Roots are the Eigenvalues.

Example problem :

Given Matrix $A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$

Step 1 : Identity Matrix

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Step 2 : Form $A - \lambda I$

$$A - \lambda I = \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix}$$

Step 3 : Find determinant

$$\begin{aligned} \det(A - \lambda I) &= (4 - \lambda)(3 - \lambda) - 2(1) \\ &= \lambda^2 - 7\lambda + 10 \end{aligned}$$

Step 4 : $\lambda^2 - 7\lambda + 10 = 0$

$$(\lambda - 5)(\lambda - 2) = 0$$

Eigenvalues $\lambda_1 = 5, \lambda_2 = 2$

How to calculate the Eigenvector Value :

Eigenvector : A non-zero vector v is an eigenvector of matrix A if $Av = \lambda v$

$$(A - \lambda I)v = 0$$

Steps :

Step 1 : Find Eigenvalues — $\det(A - \lambda I) = 0$

Step 2 : Substitute one Eigenvalue λ — put the value into $A - \lambda I$

Step 3 : Solve the homogeneous system $(A - \lambda I)v = 0$. This gives linear equations.

Step 4 : Find Non-zero solution — Any Non-zero solution is an Eigenvector.

Example problem :

Given Matrix $A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$

Eigenvalues (already found)

$$\lambda_1 = 5, \lambda_2 = 2$$

Eigenvector for $\lambda = 5$

Step 1 : Form $A - 5I$

$$A - 5I = \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix}$$

$$= \begin{bmatrix} 4 - 5 & 2 \\ 1 & 3 - 5 \end{bmatrix}$$

$$= \begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix}$$

Step 2 : Solve $(A - 5I)v = 0$

$$v = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Step 3 : Write equations

$$-x + 2y = 0$$

$$x - 2y = 0$$

Step 4 : $x = 2y$

choose $y = 1$

$$x = 2(1) \Rightarrow \boxed{x = 2}$$

Eigenvector for $\lambda = 5$

$$v_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

(Any scalar multiple is also valid)

Eigenvector for $\lambda = 2$

step 1 : Form $A - \lambda I$

$$A - \lambda I = \begin{bmatrix} 4-2 & 2 \\ 1 & 3-2 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}$$

step 2 : Solve $(A - \lambda I)v = 0$

$$2x + 2y = 0$$

$$x + y = 0$$

step 3 : Solve

$$\begin{aligned} \text{choose } y &= 1 & x &= -y \\ & & x &= -1 \end{aligned}$$

Eigenvector for $\lambda = 2$

$$v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Final output values are

$$\lambda = 5 \rightarrow \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\lambda = 2 \rightarrow \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Probability & Information Theory for Deep Learning

A Comprehensive Exam Revision & Concept Guide



Suitable for APPGECET, B.Tech, and Machine Learning Fundamentals

NotebookLM

Scanned with OKEN Scanner



Scanned with OKEN Scanner

The Landscape of Uncertainty

Why do we need Probability in AI?

Deep Learning models the real world. The real world is uncertain.
Probability is the tool we use to quantify that **uncertainty**.



Model Uncertainty

Handling noise and incomplete information.



Represent Data

Understanding the underlying structure of datasets.



Predict Classes

Assigning likelihoods (e.g., '80% chance this is a cat').



Generative Models

Creating new data from learned distributions.

Examples: Bernoulli (Binary), Gaussian (Continuous), Categorical (Multi-class).

NotebookLM

Scanned with OKEN Scanner

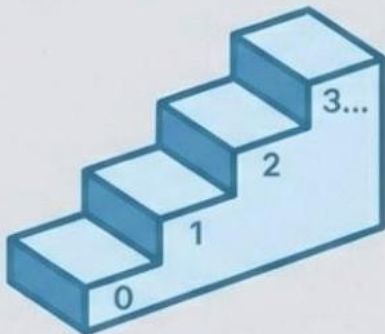


Scanned with OKEN Scanner

The Building Block: Random Variables

Definition: A variable whose value depends on the outcome of a random experiment.
Maps **Outcomes** $\rightarrow$ **Numbers**.

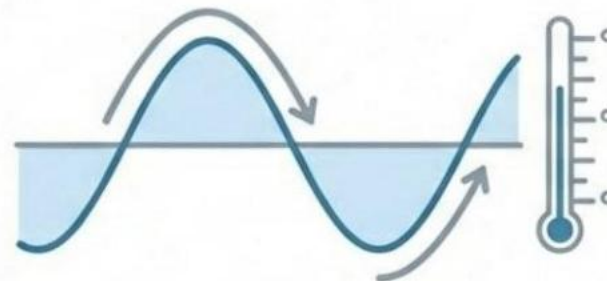
Discrete Random Variable



Takes countable values (0, 1, 2...).

X = Number of heads in a coin toss.

Continuous Random Variable



Takes real values & measurements.

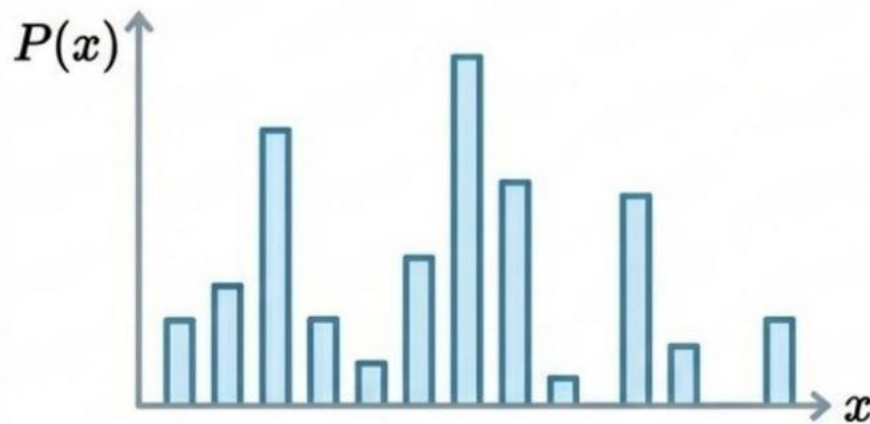
Y = Temperature tomorrow.

In Neural Networks: Inputs (pixels) and Outputs (classes) are treated as **Random Variables**.

Mapping Outcomes with Distributions

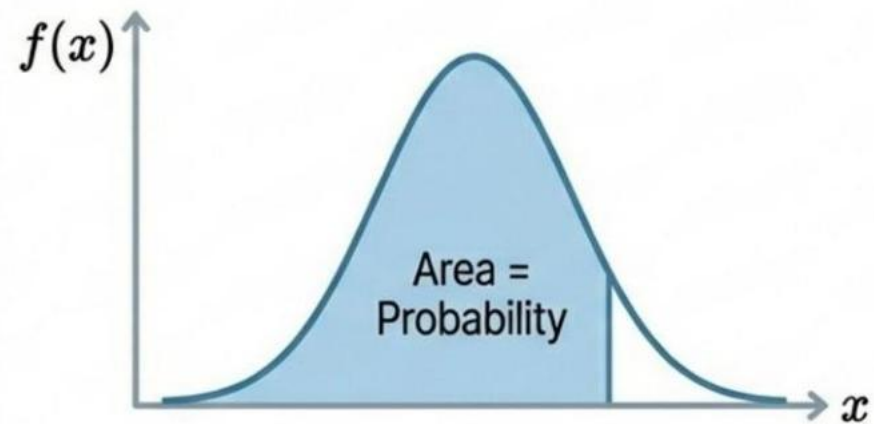
It tells us which values are possible and how likely each value is.

Probability Mass Function (PMF)



Used for **Discrete variables** (e.g., Bernoulli).

Probability Density Function (PDF)



Used for **Continuous variables** (e.g., Gaussian).

Academic Definition: A **Probability Distribution** describes how **probabilities** are assigned to all possible values of a random variable.

Relationships: Marginal & Conditional Probability

Marginal Probability

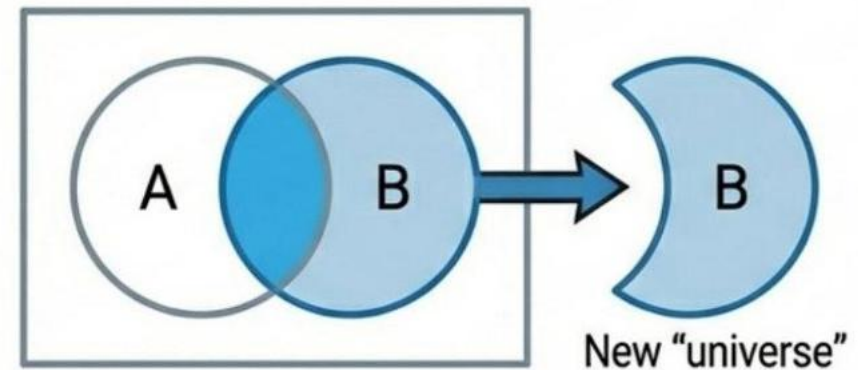
	X_1	X_2	X_3
Y_1			
Y_2			
Y_3			
Sum over Y	$P(X_1)$	$P(X_2)$	$P(X_3)$

Definition: Probability of one variable without considering others.

$$P(X) = \sum P(X, Y) \text{ (Summing over all } Y)$$

Intuition: The "individual" probability.

Conditional Probability



Definition: Probability of an event given another event has occurred.

$$P(A | B) = \frac{P(A, B)}{P(B)}$$

Intuition: Probability of A when B is known to be true (e.g., Rain given Cloudy).

Introduction to Information Theory

Probability asks “How likely?” Information Theory asks
“How much surprise does this event contain?”

Entropy

Measuring
Uncertainty.

Cross-Entropy

Measuring
Error.

KL Divergence

Measuring
Similarity.

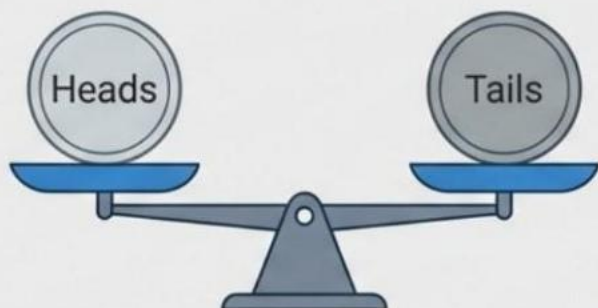
Essential for Loss Functions, Optimization, Feature Selection, and Model Evaluation.

Entropy: The Measure of Uncertainty



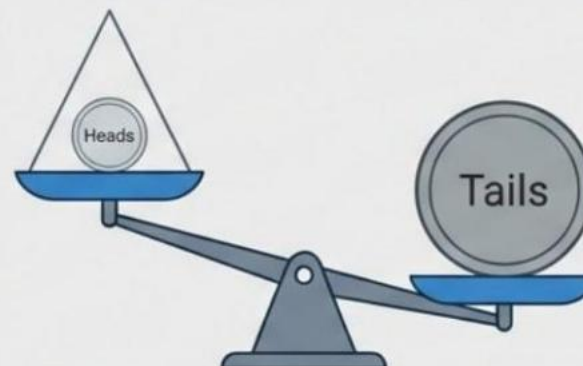
$$H(P) = - \sum_i P(i) \log P(i)$$

High Entropy



Fair Coin (50/50)
Maximum Uncertainty.
High Entropy.

Low Entropy



Biased Coin (99/1)
Very Predictable.
Low Entropy.

Intuition: How unpredictable is the data? Used to measure impurity in Decision Trees.

NotebookLM

Scanned with OKEN Scanner



Scanned with OKEN Scanner

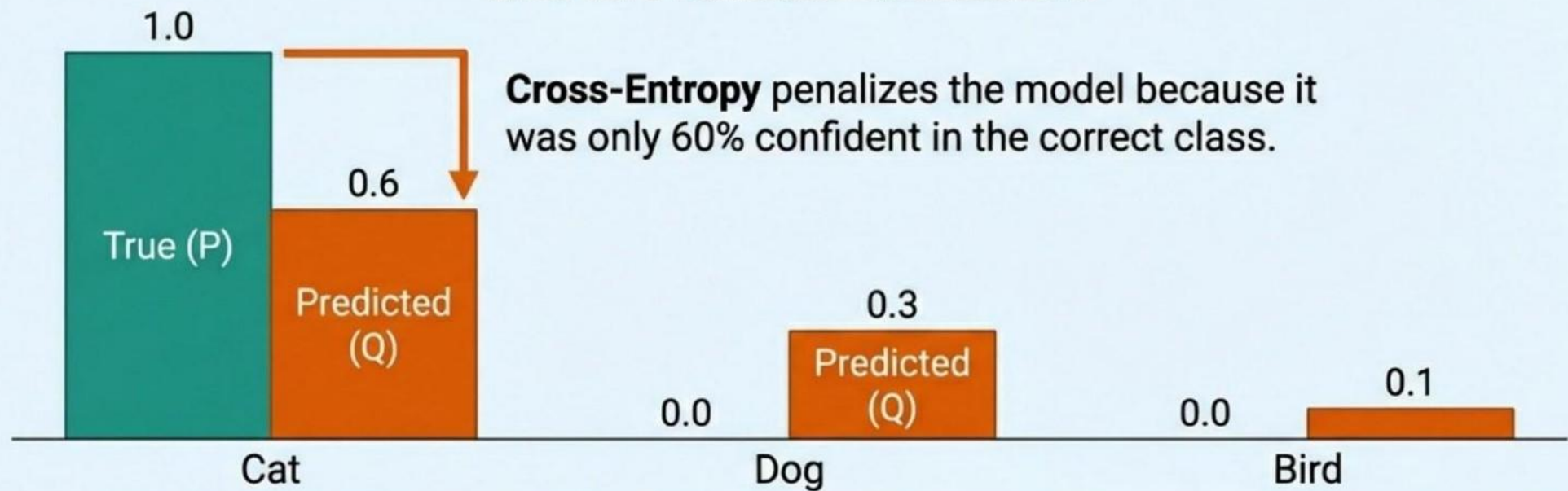
Cross-Entropy: The Measure of Prediction Error

 $H(P, Q)$

$$H(P, Q) = - \sum_i P(i) \log Q(i)$$

P is True Distribution Q is Predicted Distribution

True vs. Predicted Distributions



Deep Learning Anchor: The Gold Standard Loss Function for Classification.

NotebookLM

Scanned with OKEN Scanner

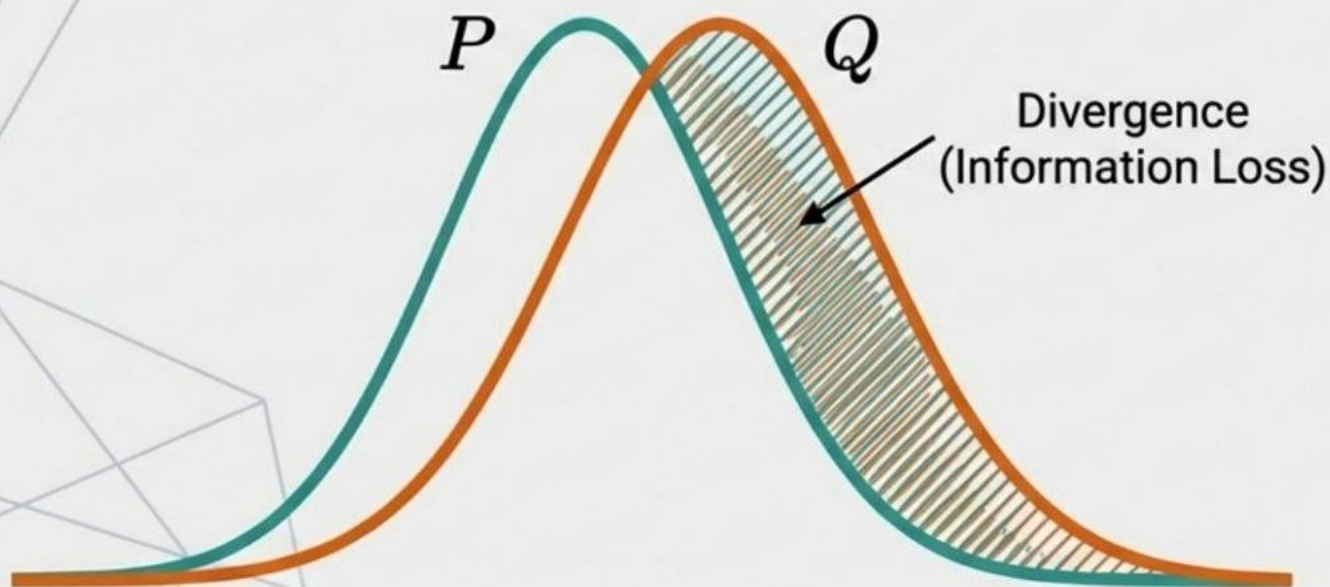


Scanned with OKEN Scanner

KL Divergence: The Distance Metric

$D_{KL}(P||Q)$

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$



Key Properties

1. Always ≥ 0
2. Equals 0 only if $P = Q$
3. **Not Symmetric:** The distance from P to Q is not the same as Q to P .

Intuition: How far is the prediction from the truth?

NotebookLM

Scanned with OKEN Scanner



Scanned with OKEN Scanner

The Relationship Framework

$H(P, Q)$

The “Holy Trinity” of Information Theory

Cross Entropy = Entropy + KL Divergence

$$H(P, Q) = H(P) + D_{KL}(P \parallel Q)$$

Entropy	Measures: Uncertainty in data	Used for: Measuring randomness
Cross Entropy	Measures: Error between True & Predicted	Used for: Loss Functions
KL Divergence	Measures: Distance between distributions	Used for: Comparing distributions

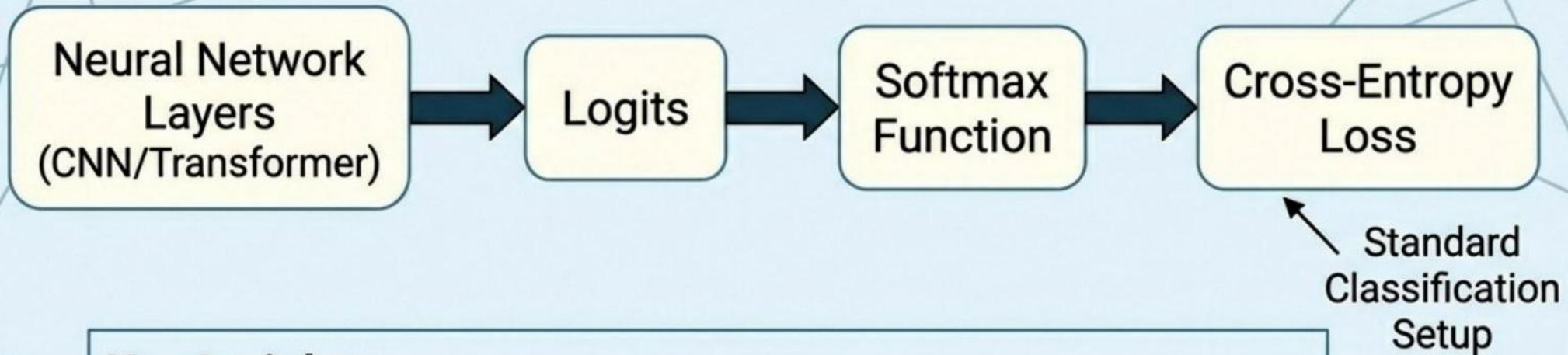
Deep Learning Anchor:

NotebookLM

Scanned with OKEN Scanner

Integration in Deep Learning Models

$$H(P, Q)$$



Key Insight:

Since the Entropy of the truth ($H(P)$) is fixed, **Minimizing Cross Entropy is mathematically identical to Minimizing KL Divergence.**

Used in CNNs, RNNs, Transformers, VAEs.

Exam Dojo: Calculating Entropy

$H(P, Q)$

$$H(X) = - \sum P(x) \log_2 P(x)$$

Problem A (Fair Coin)

Given: $P(0)=0.5$, $P(1)=0.5$

Calculation:

$$\begin{aligned} & - [0.5 \log_2(0.5) + 0.5 \log_2(0.5)] \\ & = - [0.5(-1) + 0.5(-1)] \\ & = -[-1] \end{aligned}$$

Result = 1 bit (Max Uncertainty)

Problem B (Unequal/Biased)

Given: $P(A)=0.8$, $P(B)=0.2$

Calculation:

$$\begin{aligned} & - [0.8 \log_2(0.8) + 0.2 \log_2(0.2)] \\ & = - [0.8(-0.322) + 0.2(-2.322)] \\ & = -[-0.257 - 0.464] \end{aligned}$$

Result $\approx$ 0.72 bits (Lower Uncertainty)

Exam Dojo: Calculating Cross-Entropy

$H(P, Q)$

Scenario:

True Distribution (P): (1, 0, 0) [Class A is correct]

Predicted (Q): (0.7, 0.2, 0.1) [Model is 70% confident]

Step-by-Step Calculation:

1. Formula: $H(P, Q) = - \sum P(i) \log_2 Q(i)$
2. Substitute: $- [1 \cdot \log_2(0.7) + 0 \cdot \log_2(0.2) + 0 \cdot \log_2(0.1)]$
3. Simplify: $- [1 \cdot (-0.515) + 0 + 0]$
4. Final Answer: **0.515 bits**

Pro Exam Tip: When P is one-hot, Cross Entropy is simply $-\log(\text{predicted probability of correct class})$.

Exam Dojo: KL Divergence & Verification

$H(P, Q)$

Scenario: $P = (0.5, 0.5)$ vs $Q = (0.8, 0.2)$

Calculate KL (D_{KL})

$$\begin{aligned}\text{Formula: } & \sum P(i) \log_2(P(i)/Q(i)) \\ &= 0.5 \log_2(0.625) + 0.5 \log_2(2.5) \\ &= 0.5(-0.678) + 0.5(1.322) \\ &= -0.339 + 0.661\end{aligned}$$

Result ≈ 0.322

The Verification ($CE = E + KL$)

We know Entropy $H(P) = 1$ (from Slide 12).

We know KL Divergence

$$D_{KL} = 0.322.$$

$$\text{Sum: } 1 + 0.322 = 1.322.$$

(This matches the Cross-Entropy calculation for these values!)

The “One-Line” Revision Sheet

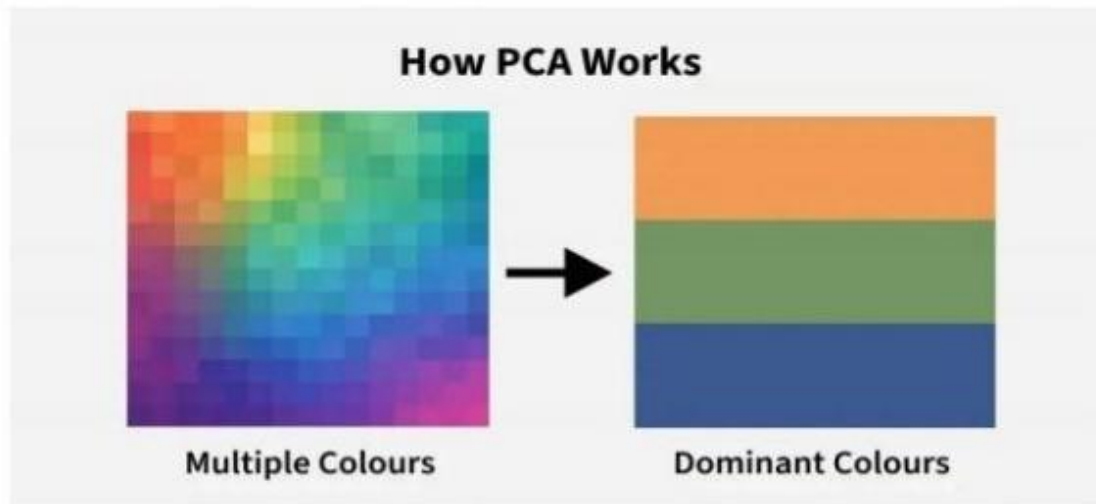
$$H(P, Q)$$

- ✓ **Random Variable:** "Value depends on random outcomes."
- ✓ **Probability Distribution:** "Describes likelihood of all possible values."
- ✓ **Marginal Probability:** "Probability of one variable ignoring others."
- ✓ **Conditional Probability:** "Probability of an event given another event occurred."
- ✓ **Entropy:** Measures uncertainty in a probability distribution.
- ✓ **Cross-Entropy:** Measures difference between true and predicted distributions.
- ✓ **KL Divergence:** Measures how one probability distribution differs from another.

**“Probability quantifies uncertainty;
Information Theory measures the surprise.”**

Principal Component Analysis (PCA)

PCA (Principal Component Analysis) is a dimensionality reduction technique and helps us to reduce the number of features in a dataset while keeping the most important information. It changes complex datasets by transforming correlated features into a smaller set of uncorrelated components.



Principal Component Analysis (PCA)

It helps us to remove redundancy, improve computational efficiency and make data easier to visualize and analyze.

How Principal Component Analysis Works

PCA uses linear algebra to transform data into new features called principal components. It finds these by calculating eigenvectors (directions) and eigenvalues (importance) from the covariance matrix. PCA selects the top components with the highest eigenvalues and projects the data onto them to simplify the dataset.

Note: It prioritizes the directions where the data varies the most because more variation = more useful information.

Imagine you're looking at a messy cloud of data points like stars in the sky and want to simplify it. PCA helps you find the "most important angles" to view this cloud so you don't miss the big patterns. Here's how it works step by step:

Step 1: Standardize the Data

Different features may have different units and scales like salary vs. age. To compare them fairly PCA first standardizes the data by making each feature have:

- A mean of 0
- A standard deviation of 1

$$Z = \frac{X - \mu}{\sigma}$$

where:

- μ is the mean of independent features $\mu = \{\mu_1, \mu_2, \dots, \mu_m\}$
- σ is the standard deviation of independent features $\sigma = \{\sigma_1, \sigma_2, \dots, \sigma_m\}$

Step 2: Calculate Covariance Matrix

Next PCA calculates the [covariance matrix](#) to see how features relate to each other whether they increase or decrease together. The covariance between two features x_1 and x_2 is:

$$\text{cov}(x_1, x_2) = \frac{\sum_{i=1}^n (x_{1i} - \bar{x}_1)(x_{2i} - \bar{x}_2)}{n - 1}$$

Where:

- $\bar{x}_1$ and $\bar{x}_2$ are the mean values of features x_1 and x_2
- n is the number of data points

The value of covariance can be positive, negative or zeros.

Step 3: Find the Principal Components

PCA identifies new axes where the data spreads out the most:

- **1st Principal Component (PC1):** The direction of maximum variance (most spread).
- **2nd Principal Component (PC2):** The next best direction, *perpendicular to PC1* and so on.

These directions come from the eigenvectors of the covariance matrix and their importance is measured by eigenvalues. For a square matrix A an eigenvector X (a non-zero vector) and its corresponding eigenvalue λ satisfy:

$$AX = \lambda X$$

This means:

- When A acts on X it only stretches or shrinks X by the scalar λ .
- The direction of X remains unchanged hence eigenvectors define "stable directions" of A .

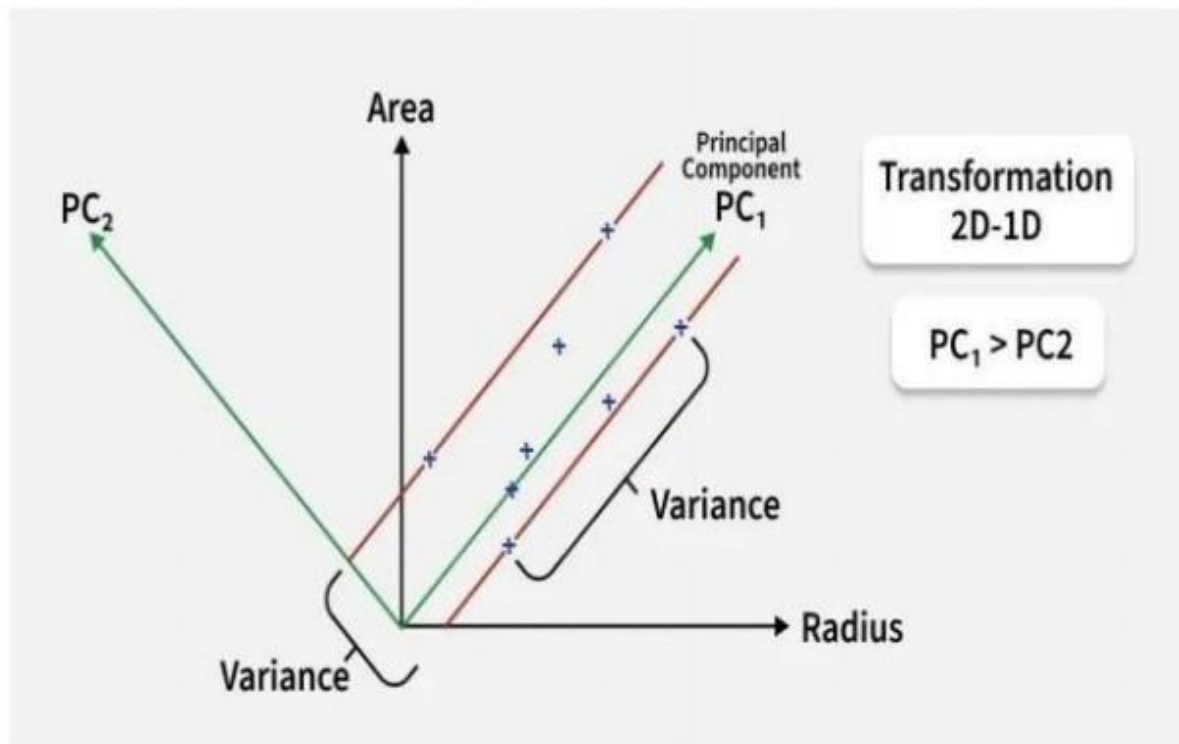
Eigenvalues help rank these directions by importance.

Step 4: Pick the Top Directions & Transform Data

After calculating the eigenvalues and eigenvectors PCA ranks them by the amount of information they capture. We then:

1. Select the top k components that capture most of the variance like 95%.
2. Transform the original dataset by projecting it onto these top components.

This means we reduce the number of features (dimensions) while keeping the important patterns in the data.



Transform this 2D dataset into a 1D representation while preserving as much variance as possible.

In the above image the original dataset has two features "Radius" and "Area" represented by the black axes. PCA identifies two new directions: PC₁ and PC₂ which are the principal components.

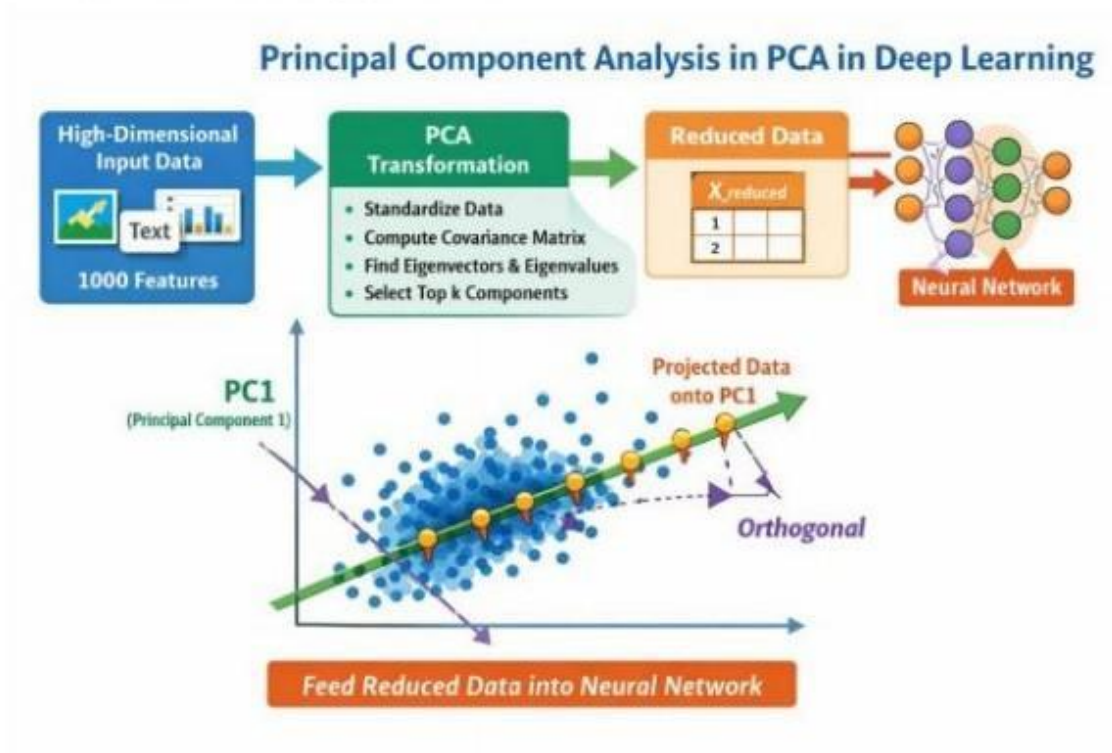
- These new axes are rotated versions of the original ones. PC₁ captures the maximum variance in the data meaning it holds the most information while PC₂ captures the remaining variance and is perpendicular to PC₁.
- The spread of data is much wider along PC₁ than along PC₂. This is why PC₁ is chosen for dimensionality reduction. By projecting the data points (blue crosses) onto PC₁ we effectively transform the 2D data into 1D and retain most of the important structure and patterns.

Advantages of Principal Component Analysis

1. **Multicollinearity Handling:** Creates new, uncorrelated variables to address issues when original features are highly correlated.
2. **Noise Reduction:** Eliminates components with low variance enhance data clarity.
3. **Data Compression:** Represents data with fewer components reduce storage needs and speeding up processing.
4. **Outlier Detection:** Identifies unusual data points by showing which ones deviate significantly in the reduced space.

Disadvantages of Principal Component Analysis

1. **Interpretation Challenges:** The new components are combinations of original variables which can be hard to explain.
2. **Data Scaling Sensitivity:** Requires proper scaling of data before application or results may be misleading.
3. **Information Loss:** Reducing dimensions may lose some important information if too few components are kept.
4. **Assumption of Linearity:** Works best when relationships between variables are linear and may struggle with non-linear data.
5. **Computational Complexity:** Can be slow and resource-intensive on very large datasets.
6. **Risk of Overfitting:** Using too many components or working with a small dataset might lead to models that don't generalize well.



Numerical Example :**Numerical Example: Principal Component Analysis (PCA)****Given Data**

Consider a dataset with 2 features (X_1, X_2) and 4 observations.

Observation	X_1	X_2
1	2	0
2	0	2
3	3	1
4	1	3

Step 1: Find the Mean of Each Feature

$$\mu_{X_1} = \frac{2 + 0 + 3 + 1}{4} = 1.5$$

$$\mu_{X_2} = \frac{0 + 2 + 1 + 3}{4} = 1.5$$

Step 2: Mean-Centered Data

Subtract the mean from each value.

Obs	$X_1 - 1.5$	$X_2 - 1.5$
1	0.5	-1.5
2	-1.5	0.5
3	1.5	-0.5
4	-0.5	1.5

Let the centered matrix be X .

Step 3: Covariance Matrix

$$Cov = \frac{1}{n-1} X^T X$$

$$Cov = \begin{bmatrix} \text{Var}(X_1) & \text{Cov}(X_1, X_2) \\ \text{Cov}(X_2, X_1) & \text{Var}(X_2) \end{bmatrix}$$

Calculations:

$$\text{Var}(X_1) = \frac{0.25 + 2.25 + 2.25 + 0.25}{3} = 1.67$$

$$\text{Var}(X_2) = 1.67$$

$$\text{Cov}(X_1, X_2) = \frac{-0.75 - 0.75 - 0.75 - 0.75}{3} = -1$$

Covariance Matrix

$$\begin{bmatrix} 1.67 & -1 \\ -1 & 1.67 \end{bmatrix}$$

Step 4: Eigenvalues

Solve:

$$|Cov - \lambda I| = 0$$

$$\begin{vmatrix} 1.67 - \lambda & -1 \\ -1 & 1.67 - \lambda \end{vmatrix} = 0$$

$$(1.67 - \lambda)^2 - 1 = 0$$

$$\lambda_1 = 2.67, \quad \lambda_2 = 0.67$$

Step 5: Eigenvectors

For $\lambda_1 = 2.67$

Eigenvector:

$$v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

For $\lambda_2 = 0.67$

Eigenvector:

$$v_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Step 6: Select Principal Component

- Highest eigenvalue = 2.67
- Corresponding eigenvector = **Principal Component (PC1)**

Step 7: Project Data onto PC1

$$Z = X \cdot v_1$$

This gives the **reduced 1-D representation** of the data.

Final Result

- Principal Component selected
- Dimensionality reduced from 2 $\rightarrow$ 1
- Maximum variance preserved

Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a factorization method in linear algebra that decomposes a matrix into three other matrices, providing a way to represent data in terms of its singular values.

SVD helps you split that table into three parts:

- **U**: This part tells you about the people (like their general preferences).
- **Σ** : This part shows how important each factor is (how much each rating matters).
- **V^T** : This part tells you about the products (how similar they are to each other)

Lets understand this with help of an example: Suppose you have a small table of people's ratings for two movies,

Name	Movie 1 Rating	Movie 2 Rating
Amit	5	3
Sanket	4	2
Harsh	2	5

- SVD breaks this table into three smaller parts: one that shows people's preferences, one that shows the importance of each movie, and one that shows how similar the movies are to each other
- Mathematically, the SVD of a matrix A (of size $m \times n$) is represented as: $A = U\Sigma V^T$

Here:

- U : An $m \times m$ orthogonal matrix whose columns are the left singular vectors of A .
- Σ : A diagonal $m \times n$ matrix containing the singular values of A in descending order.
- V^T : The transpose of an $n \times n$ orthogonal matrix, where the columns are the right singular vectors of A .

How to perform Singular Value Decomposition

To perform Singular Value Decomposition (SVD) for the matrix $A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}$, let's break it down step by step.

Step 1: Compute AA^T

First, we need to calculate the matrix AA^T (where A^T is the transpose of matrix A):

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 3 & 2 \\ 2 & 3 \\ 2 & -2 \end{bmatrix}$$

Now, compute AA^T :

$$AA^T = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 \\ 2 & 3 \\ 2 & -2 \end{bmatrix} = \begin{bmatrix} 17 & 8 \\ 8 & 17 \end{bmatrix}$$

Step 2: Find the Eigenvalues of AA^T

To find the eigenvalues of AA^T , we solve the characteristic equation:

$$\begin{aligned} \det(AA^T - \lambda I) &= 0 \\ \det \begin{bmatrix} 17-\lambda & 8 \\ 8 & 17-\lambda \end{bmatrix} &= 0 \\ (\lambda - 25)(\lambda - 9) &= 0 \end{aligned}$$

Thus, the eigenvalues are $\lambda_1 = 25$ and $\lambda_2 = 9$. These eigenvalues correspond to the singular values $\sigma_1 = 5$ and $\sigma_2 = 3$, since the singular values are the square roots of the eigenvalues.

Step 3: Find the Right Singular Vectors (Eigenvectors of $A^T A$)

Next, we find the eigenvectors of $A^T A$ for $\lambda = 25$ and $\lambda = 9$.

For $\lambda = 25$:

Solve $(A^T A - 25I)v = 0$:

$$A^T A - 25I = \begin{bmatrix} -12 & 12 & 2 \\ 12 & -12 & -2 \\ 2 & -2 & -17 \end{bmatrix}$$

Row-reduce this matrix to:

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

The eigenvector corresponding to $\lambda = 25$ is:

$$v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

For $\lambda = 9$:

Solve $(A^T A - 9I)v = 0$:

The eigenvector corresponding to $\lambda = 9$ is:

$$v_2 = \begin{bmatrix} \frac{1}{\sqrt{18}} \\ \frac{-1}{\sqrt{18}} \\ \frac{4}{\sqrt{18}} \end{bmatrix}$$

For the third eigenvector v_3 :

Since v_3 must be perpendicular to v_1 and v_2 , we solve the system $v_1^T v_3 = 0$ and $v_2^T v_3 = 0$, leading to:

$$v_3 = \begin{bmatrix} \frac{2}{3} \\ \frac{3}{3} \\ \frac{-2}{3} \\ \frac{-1}{3} \\ \frac{3}{3} \end{bmatrix}$$

Step 4: Compute the Left Singular Vectors (Matrix U)

To compute the left singular vectors U, we use the formula $u_i = \frac{1}{\sigma_i} A v_i$. This results in:

$$U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

Step 5: Final SVD Equation

Finally, the Singular Value Decomposition of matrix A is:

$$A = U \Sigma V^T$$

Where:

$$U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{18}} & \frac{-1}{\sqrt{18}} & \frac{4}{\sqrt{18}} \\ \frac{2}{3} & \frac{-2}{3} & \frac{1}{3} \end{bmatrix}$$

Thus, the SVD of matrix A is:

$$A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{18}} & \frac{-1}{\sqrt{18}} & \frac{4}{\sqrt{18}} \\ \frac{2}{3} & \frac{-2}{3} & \frac{1}{3} \end{bmatrix}$$

This is the Result SVD matrix of matrix A.

Applications of Singular Value Decomposition (SVD)

1. Calculation of Pseudo-Inverse (Moore-Penrose Inverse)

- The pseudo-inverse is a generalization of the matrix inverse, applicable to non-invertible matrices like low-rank matrices. For an invertible matrix, it equals the inverse.
- Denoted as M^+ , it is calculated using the SVD $M = U\Sigma V^T$, where U and V are orthogonal matrices of left and right singular vectors, and Σ is a diagonal matrix of singular values.
- Pseudo-inverse formula: $M^+ = V\Sigma^{-1}U^T$, where Σ^{-1} inverts non-zero singular values.

2. Solving a Set of Homogeneous Linear Equations

- For $Mx = b$, if $b = 0$, use SVD to choose a column of V associated with a zero singular value.
- If $b \neq 0$, solve by multiplying both sides by M^+ : $x = M^+b$.

3. Rank, Range, and Null Space

The rank, range, and null space of a matrix M can be derived from its SVD.

- **Rank:** The rank of matrix M is the number of non-zero singular values in Σ .
- **Range:** The range of matrix M is the span of the left singular vectors in matrix U corresponding to the non-zero singular values.
- **Null Space:** The null space of matrix M is the span of the right singular vectors in matrix V corresponding to the zero singular values.

4. Curve Fitting Problem

Singular Value Decomposition can be used to minimize the [least square error](#) in the curve fitting problem. By approximating the solution using the pseudo-inverse, we can find the best-fit curve to a given set of data points.

5. Applications in Digital Signal Processing (DSP) and Image Processing

- **Digital Signal Processing:** SVD can be used to analyze signals and filter noise.
- **Image Processing:** SVD is used for image compression and denoising. It helps in reducing the dimensionality of image data by preserving the most significant singular values and discarding the rest.

- ① Random Variables
- ② probability Distributions
- ③ Marginal probability
- ④ Conditional probability

① Random Variables :

A Random Variables is a Variable that can take different possible values, where the specific values is determined by chance.

* The Random Variables itself represents all possible outcomes to numerical values, allowing for mathematical operations

Types : ① Discrete Random Variables
② Continuous Random Variables

① Discrete Random Variables : Finite (or) countable values. Described by probability Mass functions like $p(X=x)$

② Continuous Random Variables :

Real values like Image intensity, sensor readings. Described by probability density functions.

[Role in DL - Represent inputs (features), outputs (predictions) and internal model states (weights, activations)]

probability Distributions

- ① Discrete probability Distribution
- ② Continuous probability Distribution

A Random Variable X Represents the number of heads obtained when tossing a fair coin two times.

- ① Find the probability distribution of X
- ② Verify that it is a valid probability distribution.

Ans Step 1 : List all possible outcomes

outcome	Number of Heads (X)
TT	0
HT	1
TH	1
HH	2

$$\text{Formula: } P(X=x_i) = \frac{1}{k}$$
$$\sum_{i=1}^k P(X=x_i) = \sum_{i=1}^k \frac{1}{k} = \frac{k}{k} = 1$$

Step 2 : Find probabilities

Total outcomes = 4

X	$P(X)$
0	$\frac{1}{4}$
1	$\frac{2}{4} = \frac{1}{2}$
2	$\frac{1}{4}$

x = Number of heads obtained

$P(x)$ = Number of outcomes where $X=x$

Total No. of outcomes

Step 3 : probability Distribution

Case 1 : $x=0$

* Means 0 heads

* Favorable outcomes : TT (1 outcome)

$$P(x=0) = 0 \Rightarrow \frac{1}{4}$$

Case 2 : $x=1$

* Means 1 head

* Favorable outcomes : HT, TH (2 outcomes)

$$P(x=1) = \frac{2}{4} = \frac{1}{2}$$

Case 3 : $x=2$

* Means 2 heads

* Favorable outcomes : HH (1 outcome)

$$P(x=2) = \frac{1}{4}$$

x	Meaning	No. of favorable outcomes	$P(X=x)$
0	No Heads	1	$\frac{1}{4}$
1	one head	2	$\frac{1}{2}$
2	Two heads	1	$\frac{1}{4}$

Step 3 : possible values of x

When we toss a coin twice, the number of heads can be

$$x \in \{0, 1, 2\}$$

Step 4 : Write the probability distributions

$$P(X=x) = \begin{cases} P(X=0), & x=0 \\ P(X=1), & x=1 \\ P(X=2), & x=2 \end{cases}$$

$$P(X=x) = \begin{cases} \frac{1}{4}, & x=0 \\ \frac{1}{2}, & x=1 \\ \frac{1}{4}, & x=2 \end{cases}$$

This is the correct and standard probability distribution formula.

Step 5: Verification.

$$\sum_x P(X=x) = \frac{1}{4} + \frac{1}{2} + \frac{1}{4} = 1$$

The probability distribution of a random variable gives the probability associated with each possible value of the random variable and satisfies $\sum P(X=x) = 1$.

- ① Random variable: Value depends on Random outcomes.
- ② probability Distribution: Describes likelihood of all possible values.
- ③ Marginal probability: probability of one variable ignoring others.
- ④ Conditional probability: probability of an event given another event occurred.
- ⑤ Entropy: Measure the uncertainty in a probability distribution.
- ⑥ Cross-Entropy: Measures difference b/w true and predicted distributions.
- ⑦ Kullback-Divergence: Measures how one probability distribution differs from another.

① Random Variable :

A Random variable is a function that assigns a real numerical value to each outcomes of a Random experiment.

Formula : $X : S \rightarrow R$

Where S is the Sample space.

Example : Toss a coin 2 times, let X = number of heads

* Sample space : $S = \{TT, TH, HT, HH\}$

* Assign values

outcome	X
TT	0
TH	1
HT	1
HH	2

Here X is the random variable mapping outcomes to numbers.

Numerical computations

overflow :

overflow occurs when a numerical result is too large to be represented by the computer, producing infinity (∞).

under-flow :

Under-flow occurs when a numerical result is too small to be represented accurately and is rounded to zero (0).

Gradient-Based optimization.

Gradient-Based optimization minimizes a function by iteratively updating parameters in the opposite direction of the gradient.

Constrained optimization :

Constrained optimization finds the best solution of a function while satisfying given equality (~~or~~) inequality constraints.

Linear least squares :

Linear least squares estimator model parameters by minimizing the sum of square differences b/w predicted and actual values.

* over-flow & under-flow

Detects instability — Importance in stable Training

* Gradient based optimization

Fast learning — used in AI systems — Trains Networks

* Constrained optimization

valid solutions — used in SVM, — Regularization

* Linear least squares

Simple fitting — used in Regression — Loss minimization.
Control model complexity.

Loss stability → Weight updater → Regularization →

Error minimization

UNIT - 2

MACHINE LEARNING

What is Machine Learning ?

Machine learning is a branch of artificial intelligence that enables algorithms to uncover hidden patterns with in datasets. It allows them to predict new, similar data without explicit programming for each task. Machine learning finds applications in driver self do such as image and speech recognition, natural language processing, recommendation systems, fraud detection, portfolio optimization, and autonomous vehicles.



Types of Machine Learning



Machine learning algorithms can be broadly categorized into three main types based on the learning approach and the nature of the data they work with.



Supervised Learning

- Models are trained using labeled datasets. Both input and output variables are provided during training.
- The aim is to establish a mapping function that predicts outcomes for new, unseen data.
- Common applications include classification, regression, and forecasting.

Unsupervised Learning

- Works with unlabeled data where outputs are not known or desired.
- The model identifies hidden structures, relationships, or groupings in the data.
- Useful for clustering, dimensionality reduction, and anomaly detection.
- Focuses on discovering inherent patterns within datasets.

Reinforcement Learning

- Based on decision-making through interaction with an environment.
- An agent performs actions and receives rewards or penalties as feedback.
- The goal is to learn an optimal strategy that maximizes long-term rewards.
- Widely applied in robotics, autonomous systems, and strategic game playing.

Real-world applications of Machine Learning

Here are some specific areas where machine learning is being used:

- **Predictive modeling**: Machine learning can be used to build predictive models that can help businesses make better decisions. For example, machine learning can be used to predict which customers are most likely to buy a particular product, or which patients are most likely to develop a certain disease.
- **Natural language processing**: Machine learning is used to build systems that can understand and interpret human language. This is important for applications such as voice recognition, chatbots, and language translation.
- **Computer vision**: Machine learning is used to build systems that can recognize and interpret images and videos. This is important for applications such as self-driving cars, surveillance systems, and medical imaging.
- **Fraud detection**: Machine learning can be used to detect fraudulent behavior in financial transactions, online advertising, and other areas.
- **Recommendation systems**: Machine learning can be used to build recommendation systems that suggest products, services, or content to users based on their past behavior and preferences.



What is Underfitting and Overfitting

Underfitting :

Underfitting happens when the model fails to learn important patterns. It performs poorly on both training and testing data. Underfitting happens due to:

- Model is too simple
- Very high regularization
- Features are weak or missing
- Not enough training
- High bias

Bias: it is like assuming all birds can only be small and fly, so the model fails to recognize big birds like ostriches or penguins that can't fly and get biased with predictions.

Bias-Variance Tradeoff Underfitting

Underfitting mainly occurs due to high bias:

- High bias means model makes strong assumptions
- Ignores patterns
- Learns an overly simple representation
- Variance is low because the model gives similar outputs even if the data changes

Underfitting = High Bias + Low Variance



What is Overfitting?

Overfitting happens when the model learns too much from the training data, including noise and outliers. It performs very well on training data but poorly on test data. Overfitting happens due to:

- Model too complex
- Too many features
- Very little data
- No regularization
- High variance

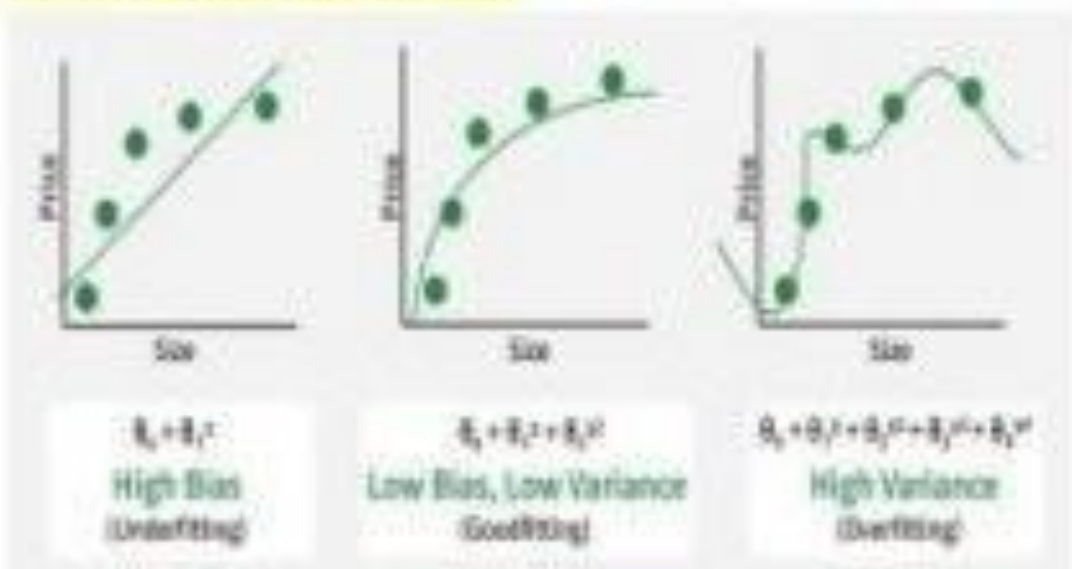
Variance: Error that happens when a machine learning model learns too much from the data, including random noise

Bias-Variance Inside Overfitting

Overfitting is mainly caused by high variance:

- High variance means model reacts too strongly to training data
- Learns noise as patterns
- Low bias because the model is extremely flexible

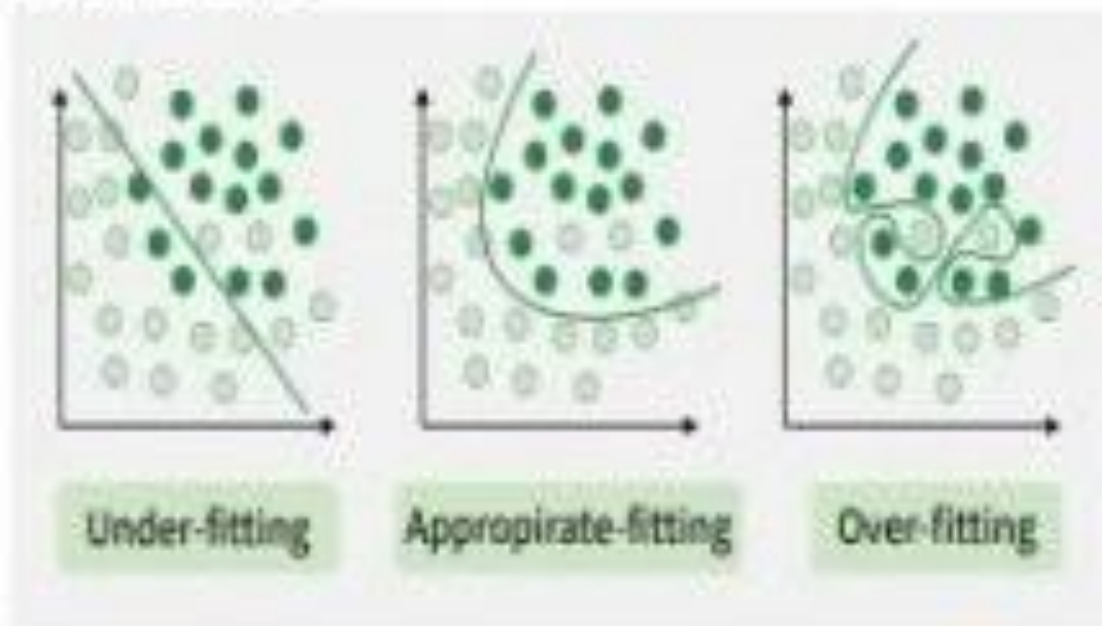
Overfitting = Low Bias + High Variance



Underfitting and Overfitting

- **Underfitting:** Straight line trying to fit a curved data set but cannot capture the data's patterns, leading to poor performance on both training and test sets.
- **Overfitting:** A squiggly curve passing through all training points, failing to generalize performing well on training data but poorly on test data.
- **Appropriate Fitting:** Curve that follows the data trend without overcomplicating to capture the true patterns in the data.

Bias-Variance Tradeoff



The relationship between bias and variance is often referred to as the [bias-variance tradeoff](#), which highlights the need for balance:

- Increasing model complexity reduces bias but increases variance (risk of overfitting).
- Simplifying the model reduces variance but increases bias (risk of underfitting).



The goal is to find an optimal balance where both bias and variance are minimized, resulting in good generalization performance.

Imagine predicting house prices based on size. You plot the data and try to draw a curve that represents the trend. How well this curve fits depends on the complexity of the model.

Underfitting and Overfitting

- Underfitting (High Bias): A model that is too simple (like a straight line for curved data) misses key patterns and performs poorly on both training and testing data.
- Overfitting (High Variance): A model that is too complex (like a high-degree polynomial) learns noise, fits training data too closely, and performs poorly on new data.
- Ideal Fit (Balanced): A moderately complex model captures the main trend without following noise, giving good performance on both training and testing data.

How to Address Overfitting and Underfitting?

Techniques to Reduce Underfitting

- Use a more complex model
- Add new features and perform feature engineering
- Reduce regularization
- Train for more epochs
- Scale features properly

Techniques to Reduce Overfitting

- Collect more training data
- Reduce model complexity
- Use regularization (L1/L2)
- Apply dropout (for neural networks)
- Use early stopping
- Clean noisy data



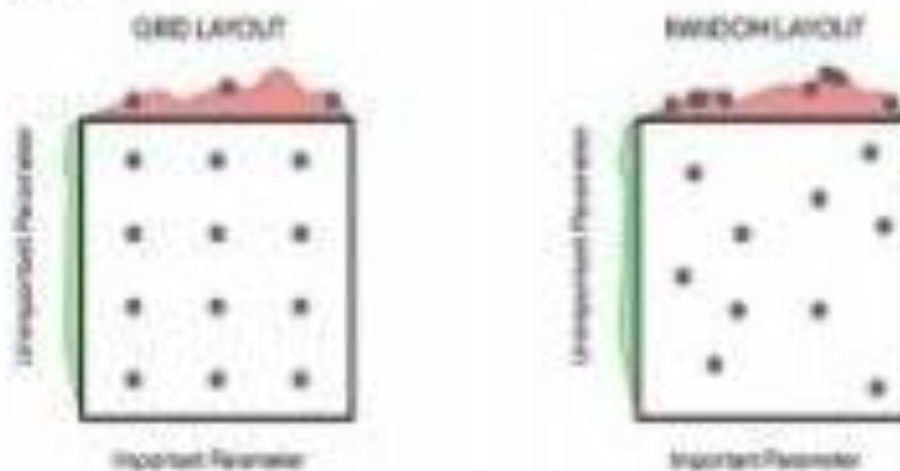
What are the Hyperparameters?

Hyperparameters are those parameters that we set for training. Hyperparameters have major impacts on accuracy and efficiency while training the model. Therefore it needed to be set accurately to get better and efficient results. Let's discuss some Hyperparameters Optimization Methods to optimize the hyperparameters.

Hyperparameters Optimization Techniques

Exhaustive Search Methods

- **Grid Search:** In Grid Search, the possible values of hyperparameters are defined in the set. Then these sets of possible values of hyperparameters are combined by using Cartesian product and form a multidimensional grid. Then we try all the parameters in the grid and select the hyperparameter setting with the best result.
- **Random Search:** This is another variant of Grid Search in which instead of trying all the points in the grid we try random points. This solves a couple of problems that are in Grid Search such as we don't need to expand our search space exponentially every time add a new hyperparameter.



Drawbacks

Random Search and Grid Search are easy to implement and can run in parallel but here are few drawbacks of these algorithms:

- If the hyperparameter search space is large, it takes a lot of time and computational power to optimize the hyperparameter.
- There is no guarantee that these algorithms find best solution if the sample is not meticulously done.

Bayesian Optimization

Instead of random guess, In bayesian optimization we use our previous knowledge to guess the hyper parameter. They use these results to form a probabilistic model mapping hyperparameters to a probability function of a score on the objective function. This probability function is defined below.

$$P\left(\frac{score(y)}{hyperparameters(x)}\right)$$

This function is also called "surrogate" of objective function. It is much easier to optimize than Objective function. Below are the steps for applying Bayesian Optimization for hyperparameter optimization:

1. Build a surrogate probability model of the objective function
2. Find the hyperparameters that perform best on the surrogate
3. Apply these hyperparameters to the original objective function
4. Update the surrogate model by using the new results
5. Repeat steps 2-4 until a number of iteration

Sequential Model-Based Optimization

Sequential Model-Based Optimization (SMBO) is a method of applying Bayesian optimization. Here sequential refers to running trials one after another, each time improving hyperparameters by applying Bayesian probability model (surrogate).



There are 5 important parameters of SMBO:

- Domain of the hyperparameter over which.
- An objective function which outputs a score which we want to optimize.
- A surrogate distribution of objective function
- A selection function to select which hyperparameter to choose next. Generally we take Expected Improvement into the consideration
- A data structure contains history of previous (score, hyperparameter) pairs which are used in previous iterations.

There are many different version SMBO hyperparameter optimization algorithm. These common difference between them is the surrogate functions. Some surrogate function such as Gaussian Process, Random Forest Regression, Tree Parzen Estimator.

validation set

A validation set is a portion of the training data used to tune hyperparameters and select models, while remaining unseen by the learning algorithm during parameter training.

The validation set is a separate subset of data used to tune model hyperparameters and make design decisions during training. Unlike the training set, it is not used to update model weights directly. Instead, it provides an unbiased estimate of model performance during development.

- Used during model selection and optimization
- Helps detect overfitting early
- Often involved in techniques like early stopping

Use Cases

- Selecting optimal hyperparameters (e.g., learning rate, depth of trees)
- Comparing multiple models or architectures
- Deciding when to stop training in neural networks



et al

Advantages

- Prevents the model from being overly tailored to training data
- Improves generalization by guiding model tuning

Limitations

- Repeated tuning on validation data can still cause validation overfitting
- Reduces the amount of data available for training



What is Deep learning :

Deep learning is a specialized branch of machine learning that uses algorithms called Artificial neural networks, especially those with many layers (hence "deep") to learn complex patterns from large amounts of data.

Why it is "Deep" :

Deep refers to the multiple layers in the network.

- * Each hidden layer transforms the input into a more abstract representation.

For example in image classification :

- * Layer 1 may detect edges
- * Layer 2 may detect shapes
- * Layer 3 may detect objects like faces

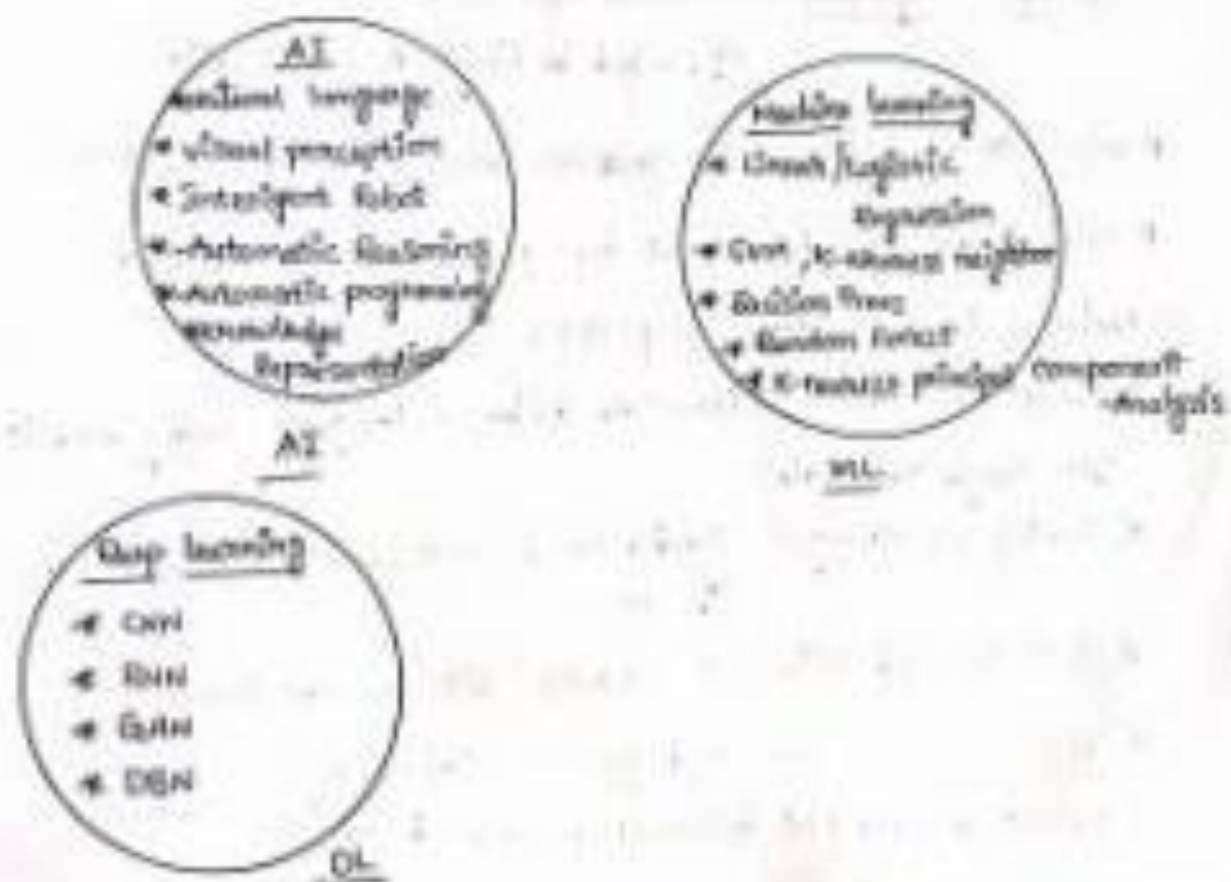


* Machine learning is the subset of AI that provides the ability to learn to the machine without being explicitly programmed.

- * Deep learning is a type of machine learning inspired by the human brain. The idea of Deep learning is to build learning algorithms or models that can mimic the human brain. As humans have neurons in their brain to process something, in the same way deep learning algorithms have Artificial Neural Networks to process the data.

Examples of Deep Learning Applications :

1. Health Care
 2. Self-Driving Cars
 3. Fraud Detection
 4. Detection of Growth Delays in children
 5. Deep Drawing
 6. Language Translations
 7. Visual Recognition
 8. Automatic Handwriting Generation
- * AI → Enabling Machines to think like humans
 - * ML → Training Machines to get better at a task without explicit programming
 - * DL → Using multi-layered networks for machine learning



1. Computer vision :

* Image classification : Identify objects in images

eg : cats, cars

* Object detection : Detect and locate multiple objects in images

eg : YOLO, SSD

* Facial Recognition : Used in security systems, smartphones and logging in social media

* Medical Imaging : Detect tumors, fractures and diseases like COVID-19 from X-Rays / MRI's

2. Speech and Audio processing :

* Speech Recognition : Convert speech to text

eg : used in Google Assistant, Alexa

* Voice cloning : Generate synthetic voices mimicking real people

* Noise Reduction : Filter out background sounds in Audio.

3. Natural language processing (NLP) :

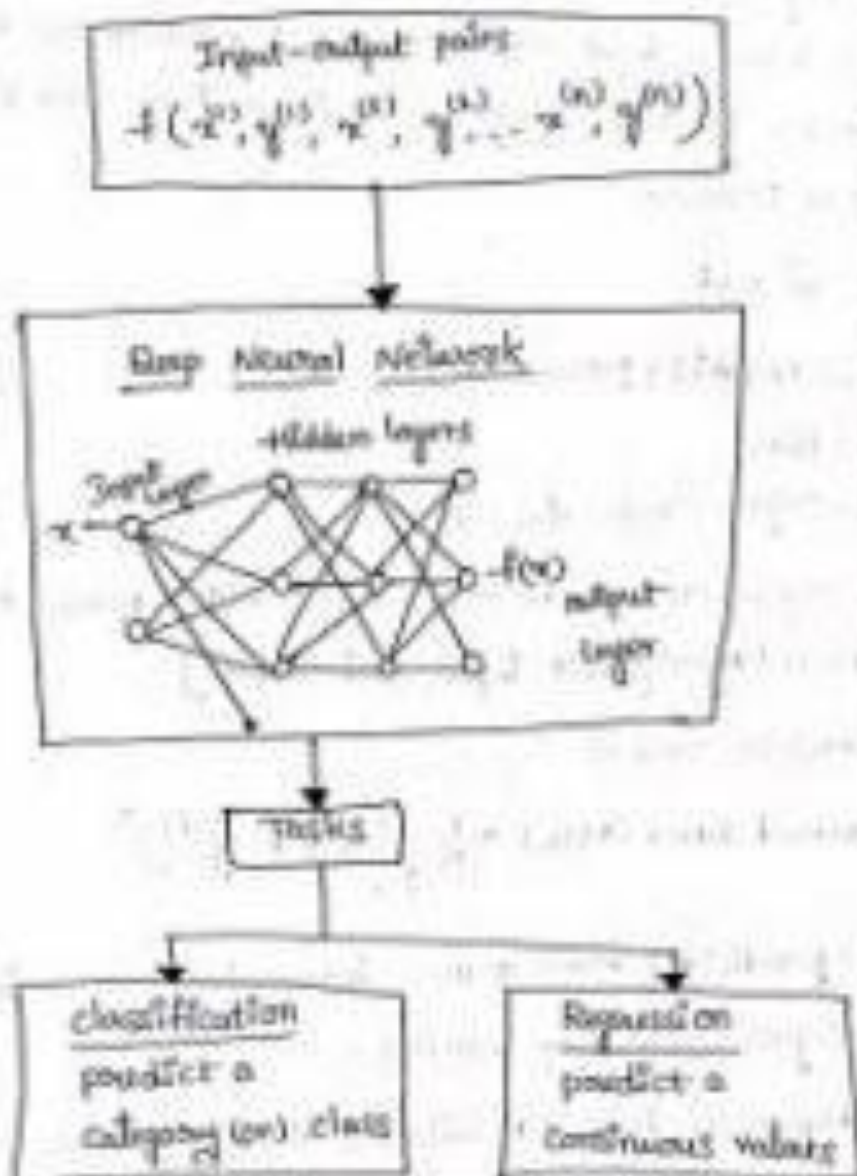
* Text Translation : Translate between languages using models like Google Translate

* Sentiment Analysis : Understand emotions from reviews (or) tweets.

* Chatbots and Virtual Assistants : like Siri or chatGPT

* Text Summarization and Question Answering :
Extract meaningful information from documents.



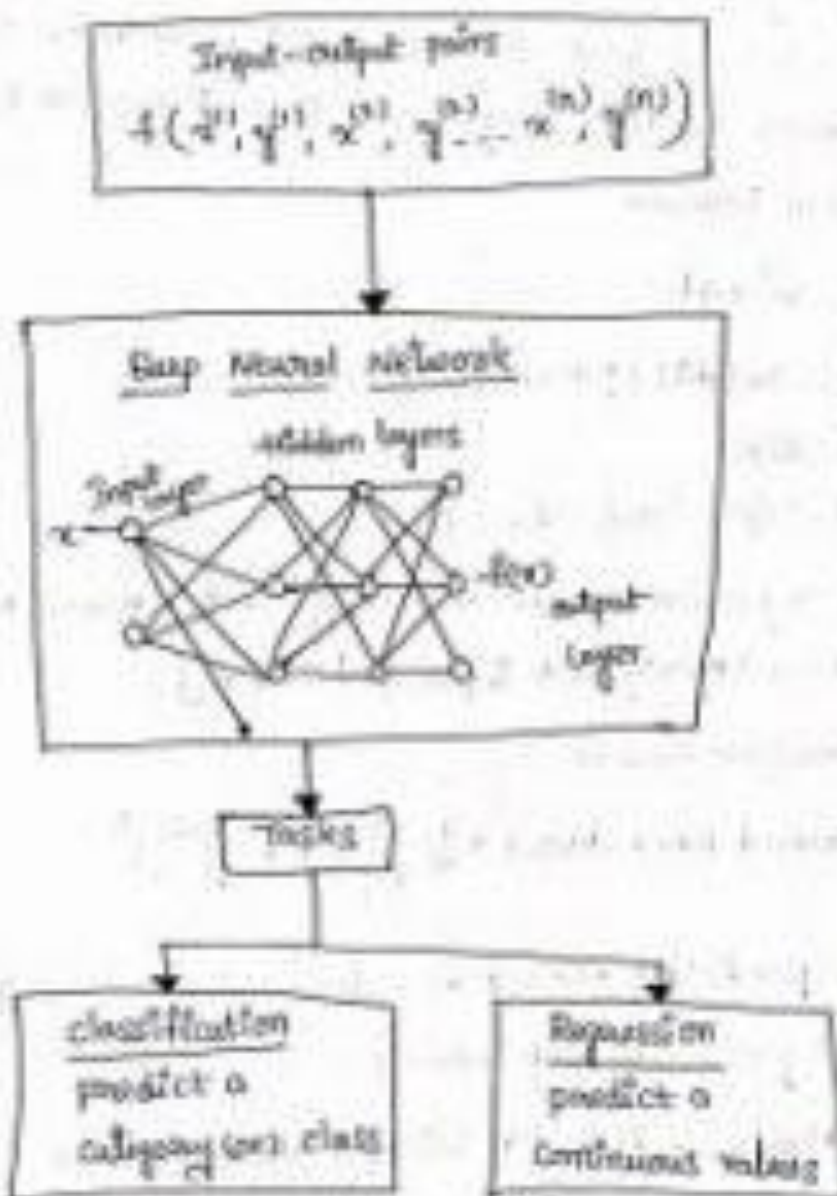


Probabilistic Supervised Learning :

Most supervised learning methods estimate conditional probability distributions $p(y|x)$.

The aim to find a parameter vector θ that best-fits this distribution using maximum likelihood estimation (MLE).

Ex : Linear Regression
 Logistic Regression

probabilistic supervised learning :

Most supervised learning methods estimate conditional probability distributions $p(y|x)$.

The aim is to find a parametric vector θ that best fits this distribution using maximum likelihood estimation (MLE).

Ex : Linear Regression
Logistic Regression

1. Linear Regression :

Linear Regression is a regression algorithm that models the relationship between input features x and a continuous output y . using a linear function, range of the Linear Regression is $-\infty$ to $+\infty$.

Mathematical Formula :

$$y = w^T x + b$$

where : w - weights (parameters)

b : bias

y : output (real-valued)

Linear regression can be implemented with a neural network with no hidden layers [just input and output].

The loss function used is :

$$\text{Mean Squared Error (MSE)} = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Examples : predicting house prices, Temperature forecasting etc.

2. Logistic Regression in Deep learning :

Logistic Regression is a classification algorithm that predicts a binary outcome (0 or 1) by modeling the probability of a class using the logistic (sigmoid) function.

Mathematical Formula :

$$P(y=1|x) = \sigma(w^T x + b)$$

$$\text{where } \sigma(x) = \frac{1}{1 + e^{-x}}$$

- * used in binary classification tasks with neural networks.
- * output layer uses sigmoid activation
- * Loss function used



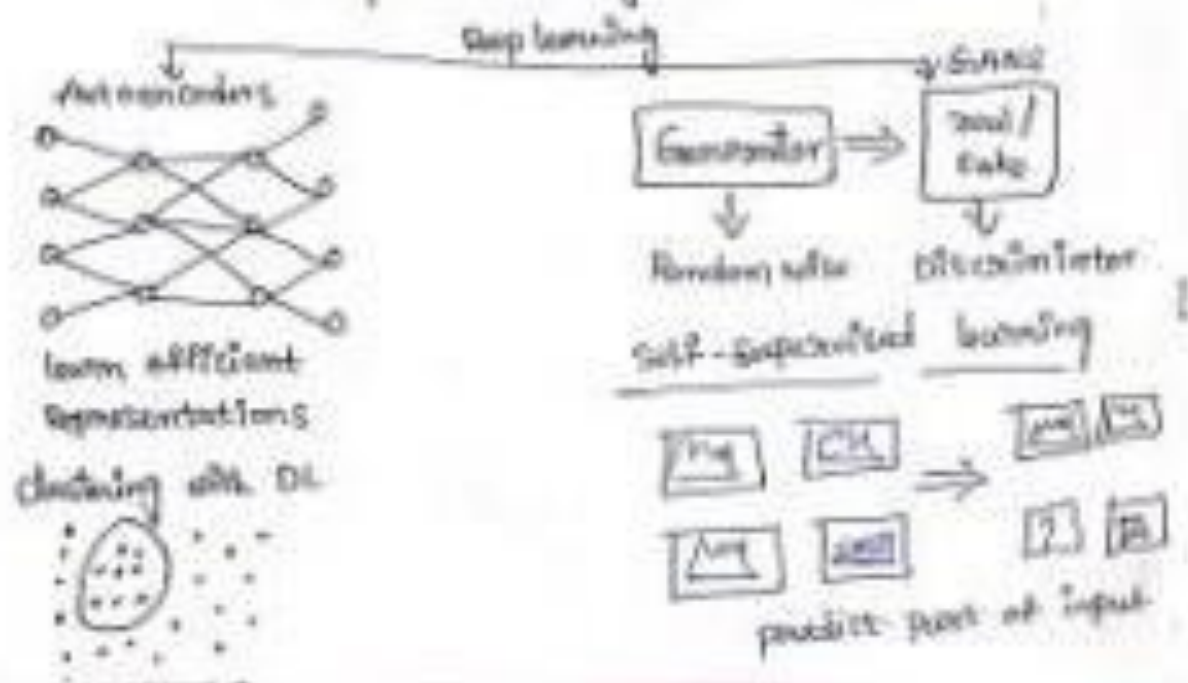
Unsupervised Learning Algorithm

(ii)

Unsupervised learning algorithm in deep learning refers to training neural networks without labeled outputs only with input data x . The model tries to discover patterns, features or grouping in the data.

1. Autoencoders - feature extraction
2. Variational Autoencoders (VAE)
3. Generative Adversarial Network (GAN)
4. Self-organizing maps (SOM) - clustering in deep learning
5. Deep Belief Networks
6. Restricted Boltzmann Machines (RBM)
7. clustering with deep learning
(Deep embedded clustering)
8. Self-Supervised learning - pre-training / Representation learning

Unsupervised learning algorithms in



Scanned with OKEN Scanner



Scanned with OKEN Scanner



Scanned with OKEN Scanner

Label Encoding

Label Encoding is a categorical data preprocessing technique used to convert non-numerical categorical values (labels) into numeric form, so that machine learning algorithms can process them.

Label Encoding assigns each unique category in a column a unique integer value.

Example:

color	Encoded
Red	0
Blue	1
Green	2
Red	0

* Label Encoding

categorical values $\rightarrow$ numeric form

* one-hot Encoding

categorical variables as binary vectors (0's and 1's)

Why it's used in Data Wrangling

1. Many ML Algorithms requires numerical data

[like Linear Regression, SVM, Neural Networks]

2. Label Encoding helps in converting categorical data into machine-readable form.

3. It is simple and memory-efficient.

(to convert categorical values into numeric form)

one-hot Encoding in Data Wrangling:

In Data Wrangling one-hot Encoding is a categorical data preprocessing technique used for represent categorical variables as binary vectors (0s and 1s).

one-hot encoding creates a new column for each category and assigns

* 1 if the record belongs to that category

* 0 otherwise



Suppose we have a dataset with a column colour

colour
Red
Blue
Green
Red

After one-hot Encoding :

col-Red	col-Blue	col-Green
1	0	0
0	1	0
0	0	1
1	0	0

Feature	Label Encoding	one-hot Encoding
Representation	Integer values	Binary vectors (0 and 1)
Suitable for	ordinal data	Nominal data
problem	Imposes order	high dimensionality

Ex: Data set

EmpID	Name	DepID	Department	Salary
101	Alice	D1	HR	5000
102	Bob	D2	Finance	6000
103	Charlie	D3	IT	6500
104	David	D4	Marketing	6800



convert one-hot result to DataFrame

```
dept_encoded_df = pd.DataFrame(  
    dept_encoded,  
    columns = onehot_encoder.get_feature_names_out(['Department'])  
)
```

Concatenate with original DataFrame

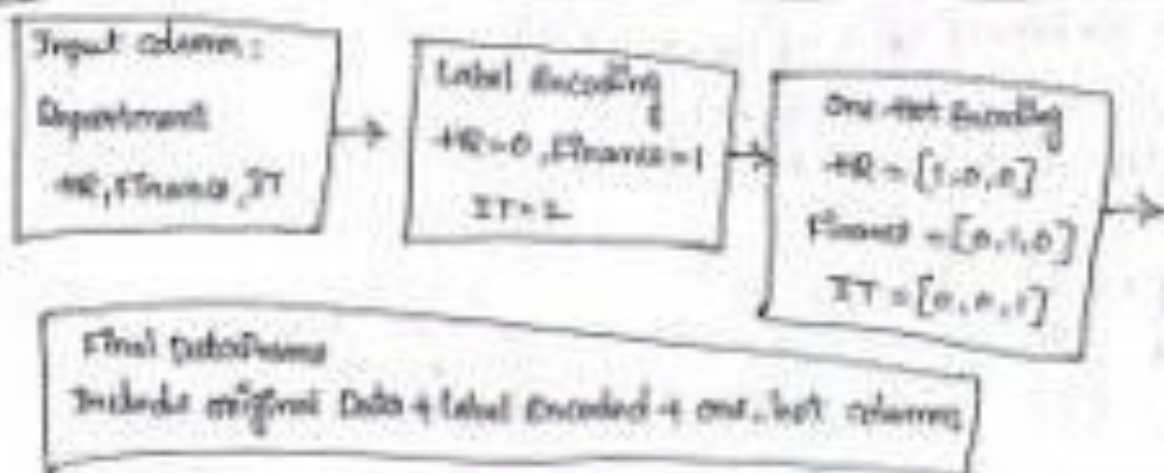
```
df_encoded = pd.concat([df, dept_encoded_df], axis=1)
```

Export to csv :

```
df_encoded.to_csv("c:/Users/Abhishek/Desktop/encoded_output.  
csv", index=False)
```

```
print("Transformed DataFrame exported to 'c:/Users/Abhishek/  
Desktop/encoded_output.csv'")
```

- ① Load data
- ② Check missing values
- ③ Convert categories into numbers
[Department → label encoding]
- ④ create separate binary columns for each category (one-hot Encoding)
- ⑤ Save to transformed dataset



Scanned with OKEN Scanner



Scanned with OKEN Scanner



Scanned with OKEN Scanner

output

EmpID	Name	Emp_ID	Department	Salary
101	Alfred	D1	HR	5000
102	Bob	D2	Finance	6000
103	Charlie	D3	IT	7000
104	David	D4	Marketing	8000

Department_Label Encoded	Department_ Finance	Department_ HR	Dept_ IT	Dept_ Mkt
1	0	1	0	0
0	1	0	0	0
2	0	0	1	0
3	0	0	0	1

Program :

Import pandas as pd

```
df = pd.read_csv('C:/Users/Adarsh/Desktop/merged_data.csv')
```

```
df = df.dropna(subset=['Department'])
```

```
label_encoder = LabelEncoder()
```

```
df['Department_LabelEncoded'] = label_encoder.fit_transform(df['Department'])
```

Apply one-hot encoding :

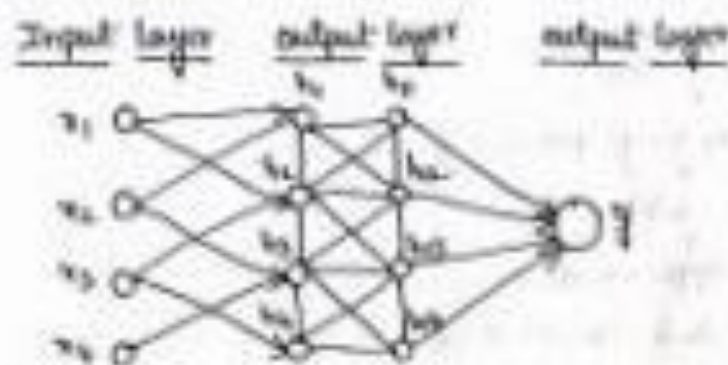
```
onehot_encoder = OneHotEncoder(sparse_output=False, drop=None)
```

```
dept_encoded = onehot_encoder.fit_transform(df[['Department']])
```



Deep Feedforward Networks :

A Deep Feedforward Networks, also known as a deep neural network (or) multilayer perceptron is a type of Artificial Neural Network where information flows in only one direction, from the input layer through multiple hidden layers, to the output layer. It's a fundamental building block in deep learning, used for tasks like image recognition, natural language processing and more.



Backbones :

Feedforward : Information flows in a single direction without feedback loops or cycles.

Deep : The network has multiple hidden layers, allowing it to learn complex, hierarchical representations of data.

Layers :

A deep feedforward network typically consists of an input layer, one or more hidden layers and an output layer.

Input layer : Receives the initial data.

Hidden layers : perform computations and feature extractions.

Output layer : produces the final results (or) prediction.



function Approximation :

Deep-feedforward networks aim to approximate a target function by learning the optimal parameters (weight and bias) for each layer.

Training :

During Training the network learns by adjusting its parameters to minimize the difference b/w its predictions and the actual target values (using backpropagation and gradient descent).

Applications :

1. Image Recognition
2. Natural language processing
3. Pattern Recognition
4. Function Approximation
5. Regression And classification

1. Image Recognition : Identifying objects in images

2. Natural language processing : understanding and generating human language

3. Pattern Recognition :

Identifying patterns in a data

4. Function Approximation :

Approximating complex mathematical functions.

5. Regression And classification :

predicting continuous (or) categorical values



Learning XOR :

The learning XOR (Exclusive OR) function is not linearly separable, meaning it cannot be solved using a single-layer perceptron. This motivates the use of hidden layers and non-linear activation function.

Truth Table XOR

Input x_1	Input x_2	Output $y (x_1 \text{ XOR } x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

Solving XOR with MLP

We need - 2 Input Neurons

1 hidden layer with at least 2 Neurons

1 output Neuron

XOR Neural Network Calculation Example :

Input $x_1 = 1, x_2 = 0$,

Expected output = 1

Hidden layer weights & biases :

Hidden Neuron	w_1	w_2	Bias
h_1	0.5	0.5	-0.2
h_2	0.5	0.5	-0.7



x	y	$\hat{y}$	$y - \hat{y}$
1	2	0.5	1.5
2	3	1.0	2.0
3	4	1.5	2.5

Now compute values

$$\begin{aligned}\frac{\partial L}{\partial w} &= \frac{1}{n} \left[x_1 (y_1 - \hat{y}_1) + x_2 (y_2 - \hat{y}_2) + x_3 (y_3 - \hat{y}_3) \right] \\ &= \frac{1}{3} [1(1.5) + 2(2.0) + 3(2.5)] \\ &= \frac{1}{3} [1.5 + 4 + 7.5] \\ &= \frac{1}{3} (13)\end{aligned}$$

$$\boxed{\frac{\partial L}{\partial w} = -8.67}$$

$$\begin{aligned}\frac{\partial L}{\partial b} &= \frac{1}{n} [(y_1 - \hat{y}_1) + (y_2 - \hat{y}_2) + (y_3 - \hat{y}_3)] \\ &= \frac{1}{3} [1.5 + 2.0 + 2.5] \\ &= \frac{1}{3} (6)\end{aligned}$$

$$\boxed{\frac{\partial L}{\partial b} = -4}$$

Step 4 : Update parameters :

$$\begin{aligned}w &= w - \eta \cdot \frac{\partial L}{\partial w} \\ &= 0.5 - 0.01(-8.67) \\ &= 0.5 + 0.0867\end{aligned}$$

$$\boxed{w = 0.5867}$$

$$\begin{aligned}b &= b - \eta \cdot \frac{\partial L}{\partial b} \\ &= 0 - 0.01(-4) \\ &= 0 + 0.04\end{aligned}$$

$$\boxed{b = 0.04}$$

Updated parameters are now $w = 0.5867$ and $b = 0.04$



Hidden Neuron h_2 :

$$Z_2 = x_1 w_1 + x_2 w_2 + B_2$$

$$x_1 = 1, w_1 = 0.5, x_2 = 0, w_2 = 0.5, B_2 = -0.7$$

$$Z_2 = 1 \times 0.5 + 0 \times 0.5 + (-0.7)$$

$$Z_2 = -0.2$$

$$h_2 = \sigma(-0.2)$$

$$= \frac{1}{1 + e^{-x}}$$

$$\approx 0.450$$

$$h_2 \approx 0.450$$

Step 2 : output neuron

$$Z_{\text{output}} = (1.0 \times h_1) + (-1.0 \times h_2) + B_3$$

$$= (0.531) - (0.450)$$

$$= 0.081$$

$$\hat{y} = \sigma(Z)$$

$$= \frac{1}{1 + e^{-x}}$$

$$= \sigma(0.081)$$

$$= \frac{1}{1 + e^{-0.081}}$$

$$\hat{y} \approx 0.531$$

predicted output $\hat{y} \approx 0.531$

After thresholding (≥ 0.5) $\rightarrow 1$

Expected output = 1



output layer :

output neuron $\hat{y}$	w_1 from h_1 1.0
w_2 from h_2	Bias
-1.0	0

Step 1 : Compute hidden layer outputs

Activation function use sigmoid function $\sigma(x) = \frac{1}{1+e^x}$

hidden neuron h_1 :

$$z_1 = x_1 w_1 + x_2 w_2 + \text{Bias}(b_1)$$

$$x_1 = 1, w_1 = 0.5, x_2 = 0, w_2 = 0.5, b_1 = -0.2$$

$$z_1 = 1 \times 0.5 + 0 \times 0.5 + (-0.2)$$

$$= 0.5 + 0.5 - 0.2$$

$$z_1 = 0.8$$

$$h_1 = \sigma(0.8)$$

$$h_1 = \frac{1}{1+e^{-0.8}}$$

$$h_1 \approx 0.694$$

Gradient Based Learning :

Gradient Based learning is a method used to train machine learning models, especially neural networks by minimizing a loss function using gradient descent techniques.

* Gradient : The slope (or) derivative of a function. It tells us the direction and rate of the steepest increase.

* Learning : Adjusting model parameters (weights & bias) to reduce prediction error.

It's like finding the lowest point in a valley by always walking in the direction of the steepest downhill.

Step-by-step process of Gradient-Based Learning :

1. Initialization
2. Forward pass (Forward propagation)
3. Compute loss
4. Backward Pass (Backward propagation)
5. Update parameters

It relies on computing gradients (partial derivatives) of an objective function with respect to model parameters.

The gradients are then used to update the parameters iteratively to minimize (or) maximize the function depending on the learning task.

This approach is fundamental to many ML learning Algorithms

1. Linear Regression
2. Logistic Regression
3. Support Vector Machines (SVMs)
4. Neural Networks
5. Deep learning Architectures



Optimization Techniques in Gradient Based Learning:

1. Gradient Descent (GD)
2. Stochastic Gradient Descent (SGD)
3. Mini-Batch Gradient Descent (MBGD)
4. Momentum
5. Adaptive Learning Rate methods (AdaGrad, RMS, prop, Adam)

1. Gradient Descent (GD):

Gradient Descent most commonly used optimization method. parameters are updated in the $-ve$ direction Gradient of loss function.

$$\theta(\text{new}) = \theta(\text{old}) - \eta \nabla \theta L$$

(η is learning rate)

2. Stochastic Gradient Descent (SGD):

Variant of Gradient Descent, parameters are updated using gradient computed from single point.

$$\theta(\text{new}) = \theta(\text{old}) - \eta \nabla \theta L (\theta = \eta \nabla \theta L)$$

3. Mini-Batch Gradient Descent (MBGD):

Variant of Gradient Descent, parameters are updated using a small subset.

4. Momentum:

Add momentum term to accelerate convergence & reduce oscillations.

$$V_{\text{new}} = V_{\text{old}} + \eta \nabla \theta L$$

$$\theta_{\text{new}} = \theta_{\text{old}} - V_{\text{new}}$$



6. Adam : (Adaptive moment estimation)

Combines momentum and adaptive Root mean square propagation (Rmsp) learning rates for efficient optimization.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left(\frac{\partial L}{\partial \theta} \right)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial L}{\partial \theta} \right)^2$$

Adam Advantages :

1. Adaptive learning rate

2. Robustness

3. Efficiency

Example problem :

We want to train a model to fit a line by $y = wx + b$ using Gradient descent data points are

x_1	x_2	x_3	y_1	y_2	y_3
1	2	3	2	3	4

Initial weights are $w = 0.5$, $b = 0$ and learning rate $\eta = 0.01$ using these parameters values to find the updated parameters?

Answer : Given data points are

x_1	x_2	x_3	y_1	y_2	y_3
1	2	3	2	3	4

Initial weights are $w = 0.5$, $b = 0$

learning rate $\eta = 0.01$

We will perform one iteration of gradient descent



Step 1 : Compute predictions

x	y (True)	$\hat{y} = 0.5x + 0$
1	2	0.5
2	3	1.0
3	4	1.5

Step 2 : Compute Loss (Mean Square Error)

$$L = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$$

$$n = (\text{No. of Input values})^2$$

$$L = \frac{1}{n} \left[(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + (y_3 - \hat{y}_3)^2 \right]$$

$$n = 3, y_1 = 2, \hat{y}_1 = 0.5, y_2 = 3, \hat{y}_2 = 1.0, y_3 = 4, \hat{y}_3 = 1.5$$

$$L = \frac{1}{3} \left[(2 - 0.5)^2 + (3 - 1.0)^2 + (4 - 1.5)^2 \right]$$

$$= \frac{1}{3} [2.25 + 4 + 6.25]$$

$$= \frac{12.5}{3}$$

$$L = 4.17$$

Step 3 : Compute Gradients

$$\frac{\partial L}{\partial w} = -\frac{2}{n} \sum (y_i - \hat{y}_i)$$

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum (y_i - \hat{y}_i)$$

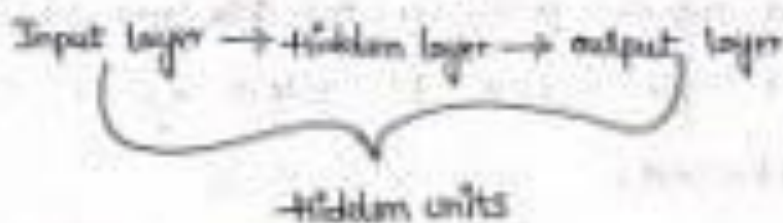
Let's compute Errors



Hidden units in Deep learning:

Hidden units are the individual neurons (or nodes) located in the hidden layers of a neural network. These units receive inputs from the previous layers, apply a weighted sum followed by a bias and then pass the result through an activation function. Hidden units are responsible for learning complex patterns and representations in the data that help the network make accurate predictions.

- * Takes inputs from the previous layers
- * Applies a weighted sum and a bias
- * Passes the result through an activation function



Example : Suppose you have a Neural network.

Input - $[x_1, x_2]$

Hidden - layer 1 - $[h_1, h_2, h_3]$

layer 2 - $[h_4, h_5]$

output $[y]$

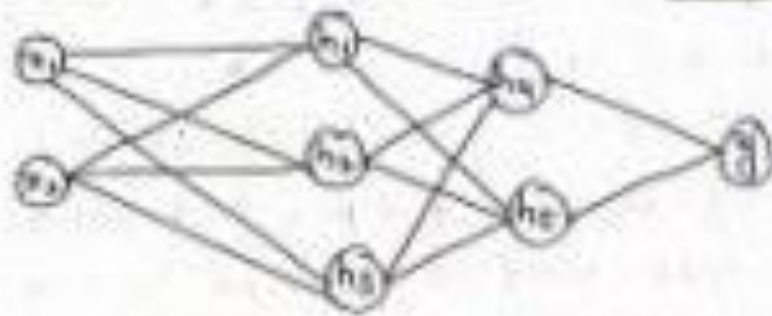
* h_1 to h_5 are hidden units

$$h_1 = (w_1 x_1 + w_2 x_2 + b) f$$

$$h_1 = f(w_1 x_1 + w_2 x_2 + b)$$



Input layer - hidden layers - output layer



* Input layers - 2 - $[x_1, x_2]$

* hidden layers - 3 - $[h_1, h_2, h_3, h_4, h_5]$

* output layer - 1 - $[y]$

Overall the first process (or) the data layer of the process
 i.e. Input layer - hidden layer - output layer is called the
 unit hidden unit.

Input layer - ^{layer} hidden unit - output layer

hidden unit

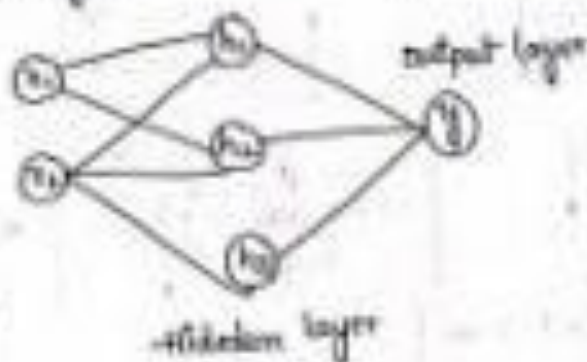
Architecture Design :

Architecture Design is the process of organizing and defining the structure of a Neural Network including,

- * Number and types of layers (input, hidden, output)
- * Number of neurons per layer
- * Activation functions
- * Layer connectivity (feedforward, skip connections etc)

Simple feedforward Neural Network :

Input layer



- * Input layers - 2 (x_1, x_2)
- * Hidden layers - 3 (h_1, h_2, h_3)
- * Output layer - 1 (sigmoid Neuron)

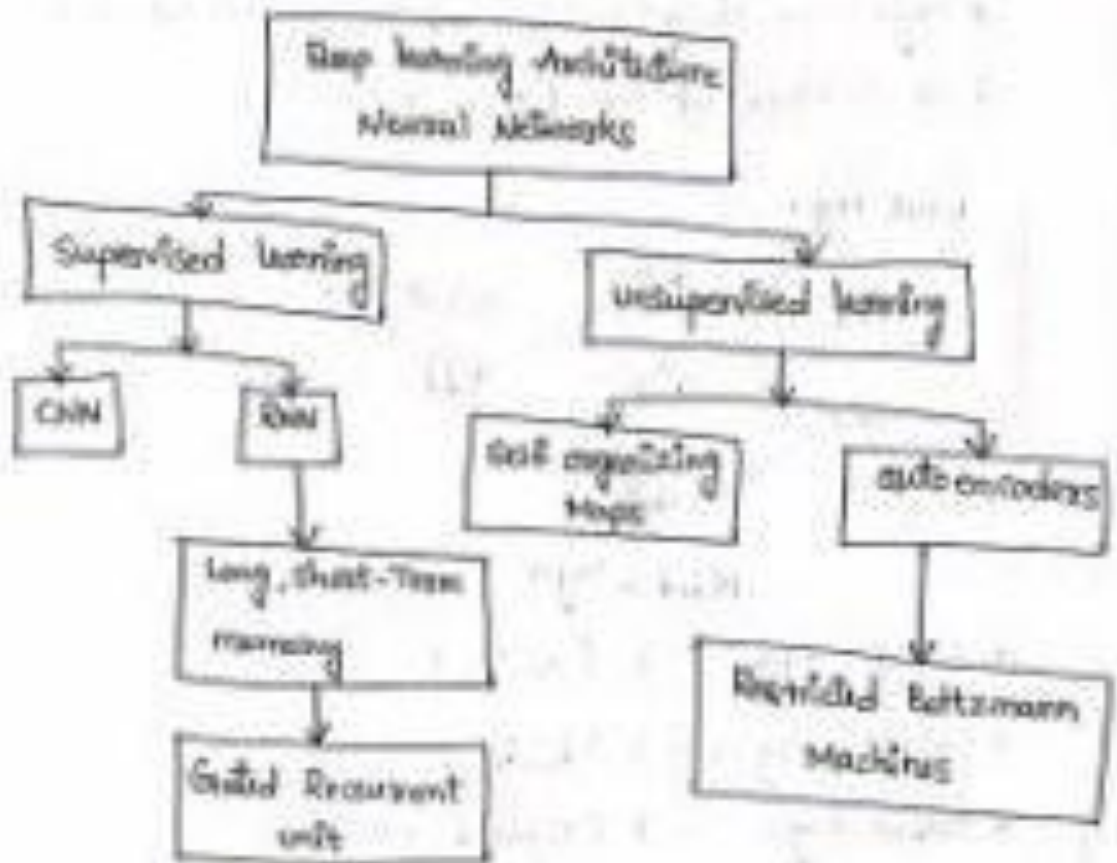
Architecture main purpose is to find a network structure that balances.

- * Accuracy
- * Generalization
- * Computational efficiency
- * Overfitting control.



USES :

1. Designing CNNs for image classification
2. Building RNNs (or) LSTM for sequence data like language (or) Time Series.
3. Creating deep-forward networks for regression (or) classification problems.



Backpropagation And other differentiation Algorithms

Backpropagation is the standard Algorithm for computing gradients in deep networks. It is based on the chain rule of calculus.

Steps :

1. Forward pass - Compute activations layer by layer with output
2. Loss computations - calculate error b/w prediction and target
3. Backward pass -
 - * Compute gradient of loss w.r.t output
 - * propagate gradients backward layer by layer using the chain rule.

4. Parameter update - Apply optimization

Formula : If loss is L and final output is y

$$\frac{\partial L}{\partial w^{(l)}} = y^{(l)} (y^{(l+1)})^T$$

$$y^{(l)} = (w^{(l+1)})^T \frac{\partial L}{\partial z^{(l+1)}} \odot f'(z^{(l)})$$

Where $y^{(l)}$ = Error Term for layer l

$f'(z^{(l)})$ = Derivative of activation function

Intuition :

- * Start at output error
- * push error back layer by layer
- * Each layer's gradient depends on the next layer.

[Below steps explanation in Gradient Based Learning Topic Notes]



Other Differentiation Algorithms in Deep Learning :

There are 5 types of Differentiation Algorithms as there in Deep learning.

1. Symbolic Differentiation
2. Numerical Differentiation (Finite Differences)
3. Automatic Differentiation
4. Hybrid Differentiation
5. Higher-Order Differentiation

1. Symbolic Differentiation :

Uses algebraic manipulation of mathematical expression to compute derivatives symbolically.

Ex : Differentiating $f(x) = x^2 + 3x$ gives $f'(x) = 2x + 3$

Advantages :

1. Gives exact derivatives
2. Works with the simple expressions.

Disadvantages :

1. Becomes inefficient and memory-heavy for large models
2. May cause expression swell
(very large intermediate formulas)

2. Numerical Differentiation :

Numerical Differentiation Approximates derivatives by perturbing inputs slightly.



* Forward Difference $f'(x) \approx \frac{f(x+h) - f(x)}{h}$

* Central Difference $f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$

Advantages :

1. Simple to implement
2. Useful for gradient checking [Debugging backpropagation]

Disadvantages :

1. Very slow for high-dimensional models
2. Sensitive to choice of h
3. Only for training mainly for validating gradients.

3. Automatic Differentiation (AD)

AD is the core algorithm in modern deep learning libraries. It decomposes functions into elementary operations and applies the chain rule systematically.

(a) Forward Mode AD

(b) Reverse Mode AD (Backpropagation)

4. Hybrid Differentiation :

combines forward-mode and reverse-mode AD to handle functions with many inputs and outputs efficiently.

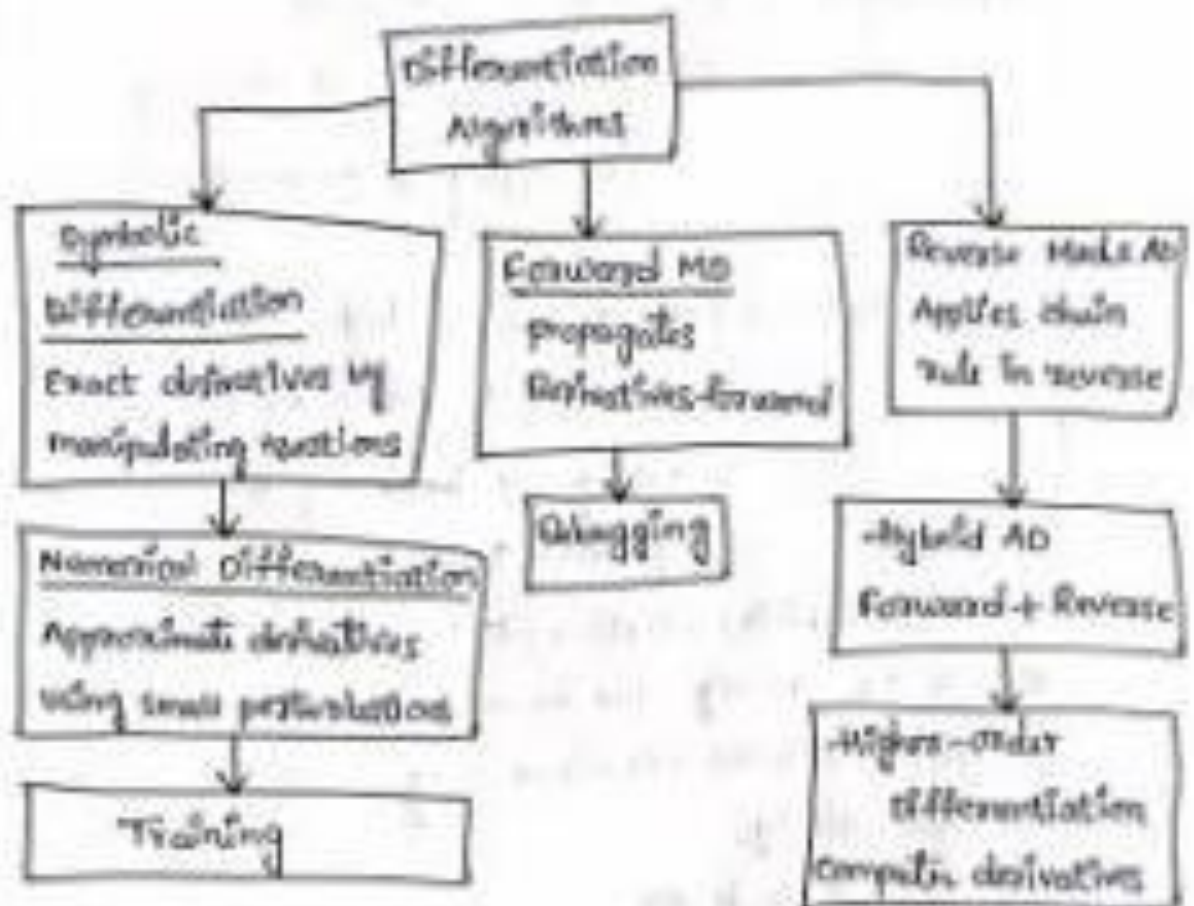
Ex : Computing higher-order derivatives often uses forward-over-reverse or reverse-over-forward schemes.

5. Higher-Order Differentiation : Involves computing derivatives of derivatives. Algorithms include.



Real-Time Applications :

1. Routly used directly, helpful in symbolic maths tools
2. Gradient checking in neural networks
3. physics-based ML, scientific computing, sensitivity analysis.



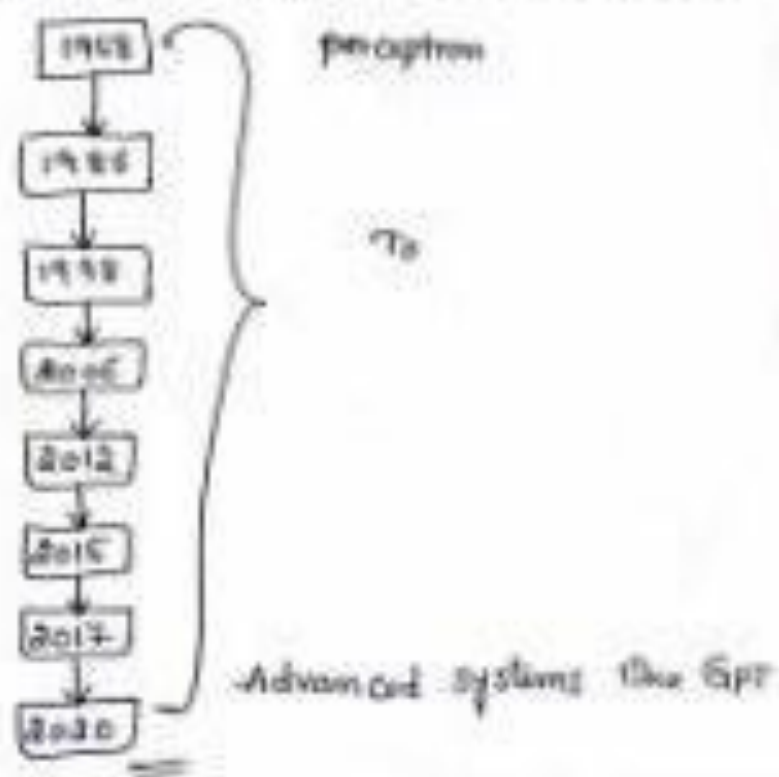
Differentiation Algorithms in Deep Learning



Historical (units) milestones in Deep learning:

A summary of the evolution of Deep learning from early models like the perceptron to advanced systems like GPT and Trans-former (Generative pre-trained Trans-formers)

- ① Early foundations → 1940's - 1960's → McCulloch & Pitts (1943)
- ② The first AI winter → 1969 - 1980's → Minsky & Papert (1969)
- ③ Backpropagation → multi-layer perceptrons & neural networks
- ④ Convolutional networks & Early Applications → 1990's - 2000's
Deep (Belated) belief networks (2006) → Geoffrey Hinton
- ⑤ Modern Deep learning 2013 - present → Architectures CNN's, RNN's & LSTM's, GAN's (2014), Trans-formers (2018)
- ⑥ Large Language Models dominate NLP Tasks → 2020 +



Scanned with OKEN Scanner



Scanned with OKEN Scanner



Scanned with OKEN Scanner

Building a Machine Learning Algorithms

Building a machine learning algorithm involves creating a set of rules or a computational procedure that enables a computer to learn from data and make predictions or decisions without explicit programming. Essentially, it's about designing a process that allows a system to identify patterns in data, adapt to new information, and improve its performance over time.

There are 5 steps are involved in building a Machine Learning Algorithms.

- 1: Data Collection
- 2: Data Preprocessing and Cleaning
- 3: Model Training
- 4: Model Evaluation and Model Performance
- 5: Model Deployment

Step 1: Data Collection for Machine Learning

Data collection is a crucial step in the creation of a machine learning model, as it lays the foundation for building accurate models. In this phase of machine learning model development, relevant data is gathered from various sources to train the machine learning model and enable it to make accurate predictions. The first step in data collection is defining the problem and understanding the requirements of the machine learning project. This usually involves determining the type of data we need for our project like structured or unstructured data, and identifying potential sources for gathering data.

Once the requirements are finalized, data can be collected from a variety of sources such as databases, APIs, web scraping, and manual data entry. It is crucial to ensure that the collected data is both relevant and accurate, as the quality of the data directly impacts the generalization ability of our machine learning model. In other words, the better the quality of the data, the better the performance and reliability of our model in making predictions or decisions.

Step 2: Data Preprocessing and Cleaning

preprocessing and preparing data is an important step that involves transforming raw data into a format that is suitable for training and testing the machine learning model. This phase aims to clean, remove null values and organize

Scanned with OKEN Scanner



Scanned with OKEN Scanner



Scanned with OKEN Scanner

values, and normalize and preprocess the data to achieve greater accuracy and performance of our machine learning models.

As Chirp Humby said, "Data is the new oil. It's valuable, but if unrefined it cannot be used." This quote emphasizes the importance of refining data before using it for analysis or modeling. Just like oil needs to be refined to unlock its full potential, raw data must undergo preprocessing to enable its effective utilization in ML tasks. The preprocessing process typically involves several steps, including handling missing values, encoding categorical variables (i.e. converting into numerical), scaling numerical features, and feature engineering. This ensures that the model's performance is optimized and also our model can generalize well to unseen data and finally get accurate predictions.

Step 3: Model Training

In this phase of building a machine learning model, we have all the necessary ingredients to train our model effectively. This involves utilizing our prepared data to teach the model to recognize patterns and make predictions based on the input features. During the training process, we begin by feeding the preprocessed data into the selected machine-learning algorithm. The algorithm then iteratively adjusts its internal parameters to minimize the difference between its predictions and the actual target values in the training data. This optimization process often employs techniques like gradient descent.

As the model learns from the training data, it gradually improves its ability to generalize to new or unseen data. This iterative learning process enables the model to become more adept at making accurate predictions across a wide range of scenarios.

Step 4: Model Evaluation and Model Performance

Once you have trained your model, it's time to assess its performance. There are various metrics used to evaluate model performance, categorized based on the type of task: regression/numerical or classification.

For regression tasks, common evaluation metrics are:

- **Mean Absolute Error (MAE):** MAE is the average of the absolute
- **Mean Squared Error (MSE):** MSE is the average of the squared differences between the predicted and actual values.

Scanned with OKEN Scanner



Scanned with OKEN Scanner



Scanned with OKEN Scanner

- **Root Mean Squared Error (RMSE):** It is a square root of the MSE, providing a measure of the average magnitude of error.
- **R-squared (R2):** It is the proportion of the variance in the dependent variable that is predictable from the independent variables.

For classification tasks, common evaluation metrics are:

- **Accuracy:** Proportion of correctly classified instances out of the total instances.
- **Precision:** Proportion of true positive predictions among all positive predictions.
- **Recall:** Proportion of true positive predictions among all actual positive instances.
- **F1-score:** Harmonic mean of precision and recall, providing a balanced measure of model performance.
- **Area Under the Receiver Operating Characteristic curve (AUC-ROC):** Measure of the model's ability to distinguish between classes.
- **Confusion Metrics:** It is a matrix that summarizes the performance of a classification model, showing counts of true positives, true negatives, false positives, and false negatives instances.

Step 5: Model Deployment and Making Predictions

Deploying the model and making predictions is the final stage in the journey of creating an ML model. Once a model has been trained and optimized, it's to integrate it into a production environment where it can provide real-time predictions on new data.

During model deployment, it's essential to ensure that the system can handle high user loads, operate smoothly without crashes, and be easily updated. Tools like Docker and Kubernetes help make this process easier by packaging the model in a way that makes it easy to run on different computers and manage efficiently. Once deployment is done our model is ready to predict new data, which involves feeding unseen data into the deployed model to enable real-time decision making.



Challenges Mitigating In Data Learned

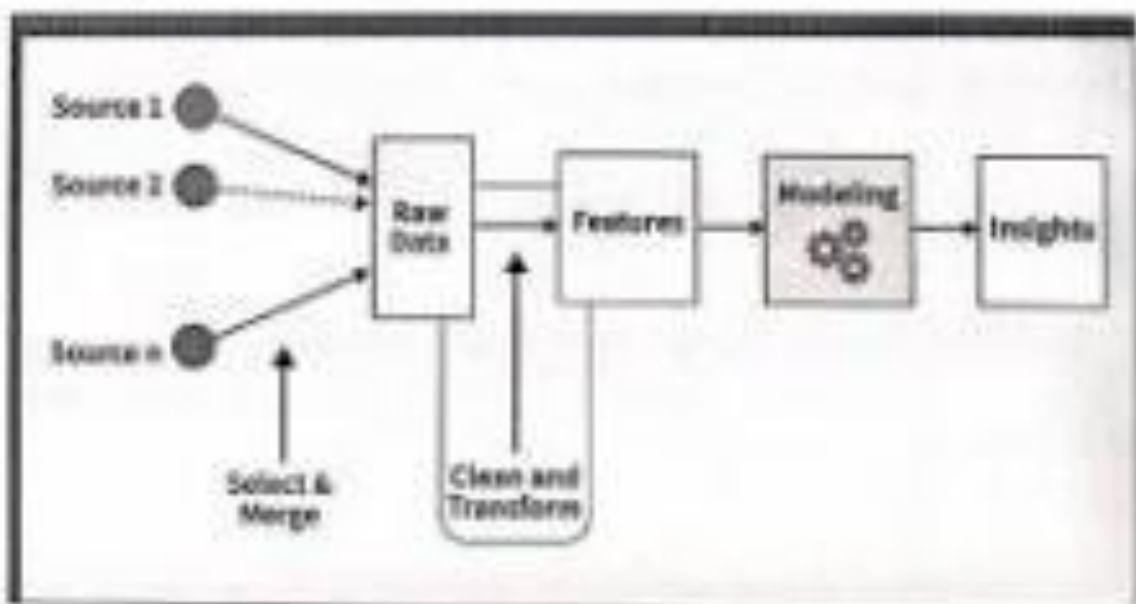
1: Feature engineering

2: Scalability

1: Feature engineering

Feature engineering is the process of turning raw data into useful features that help improve the performance of machine learning models. It includes choosing, creating and adjusting data attributes to make the model's predictions more accurate. The goal is to make the model better by providing relevant and easy-to-understand information.

A feature or attribute is a measurable property of data that is used as input for machine learning algorithms. Features can be numerical, categorical or text-based representing essential data aspects which are relevant to the problem. For example in housing price prediction, features might include the number of bedrooms, location and property age.



Importance of Feature Engineering

Feature engineering can significantly influence model performance. By refining features, we can:

- **Improve accuracy:** Choosing the right features helps the model learn better, leading to more accurate predictions.
- **Reduce overfitting:** Using fewer, more important features helps the model avoid memorizing the data and perform better on new data.
- **Boost interpretability:** Well-chosen features make it easier to understand how the model makes its predictions.
- **Enhance efficiency:** Focusing on key features speeds up the model's training and prediction process, saving time and resources.

Processes Involved in Feature Engineering

Let's see various features involved in feature engineering:

1. Feature Creation: Feature creation involves generating new features from domain knowledge or by observing patterns in the data. It can be:

- **Domain-specific:** Created based on industry knowledge like business rules.
- **Data-driven:** Derived by recognizing patterns in data.
- **Synthetic:** Formed by combining existing features.

2. Feature Transformation: Transformation adjusts features to improve model learning:

- **Normalization & Scaling:** Adjust the range of features for consistency.
- **Encoding:** Converts categorical data to numerical form i.e one-hot encoding.
- **Mathematical transformations:** Like logarithmic transformations for



3. Feature Extraction: Extracting meaningful features can reduce dimensionality and improve model accuracy:

- **Dimensionality reduction:** Techniques like PCA reduce features while preserving important information.
- **Aggregation & Combination:** Summing or averaging features to simplify the model.

4. Feature Selection: Feature selection involves choosing a subset of relevant features to use:

- **Filter methods:** Based on statistical measures like correlation.
- **Wrapper methods:** Select based on model performance.
- **Embedded methods:** Feature selection integrated within model training.

5. Feature Scaling: Scaling ensures that all features contribute equally to the model:

- **Min-Max scaling:** Rescales values to a fixed range like 0 to 1.
- **Standard scaling:** Normalizes to have a mean of 0 and variance of 1.

Steps in Feature Engineering

Feature engineering can vary depending on the specific problem but the general steps are:

1. **Data Cleansing:** Identify and correct errors or inconsistencies in the dataset to ensure data quality and reliability.
2. **Data Transformation:** Transform raw data into a format suitable for modeling including scaling, normalization and encoding.
3. **Feature Extraction:** Create new features by combining or deriving information from existing ones to provide more meaningful input to the model.



- **Feature Iteration:** Continuously refine features based on model performance by adding, removing or modifying features for improvement.

2 : Scalability

Scalability is the capacity of a system to support growth or to manage an increasing volume of work.

- When a system's workload or scope rises, it should be able to maintain or even improve its performance, efficiency, and dependability. This is known as scalability.
- A system must be scalable in order to accommodate growing user traffic, data volumes, or computing demands without suffering a major performance hit or necessitating a total redesign.

Importance of Scalability in System Design

Scalability is an important factor in system design, which offers many benefits such as:

- **Managing Growth:** When a system is scalable, it can handle more users, data, and traffic without sacrificing speed or reliability. Businesses that want to grow over time and draw in more customers need to do this.
- **Increasing Performance:** By dividing up the load among several servers or resources, a scalable system increases overall performance. This leads to faster processing speeds, faster reaction times, and an intuitive user interface.
- **Ensuring Availability:** By guaranteeing that systems continue to operate even in the event of unexpected spikes in traffic or component failures, scalability ensures availability. This is



be operational to guarantee that clients can get the services they require.

- **Cost-effectiveness:** Scalable systems are able to adjust their resources to meet demand variations by adding or subtracting resources as needed. This flexibility helps avoid oversupply and leads to significant cost savings by using just the resources that are required.
- **Encouraging Innovation:** Scalability encourages the development of new features or services by lowering infrastructure obstacles. Because of this flexibility, businesses can swiftly adapt to changes in the market and keep a competitive edge.

Architecture Design

Architecture Design in Deep Learning

Architecture design in deep learning refers to the structured layout or blueprint of a neural network, which defines how the layers, nodes (neurons), and connections are organized to learn patterns from data effectively.

Definition

Architecture design in deep learning is the process of selecting and organizing the components (like layers, activation functions, and connections) of a neural network to solve a specific learning problem efficiently.

Key Elements of Architecture Design

1. Number of Layers:
 - Shallow Network: Few layers.
 - Deep Network: Many hidden layers.
2. Number of Neurons per Layer:
 - Determines the capacity to learn patterns.
3. Type of Layers:
 - Input Layer - Receives data.
 - Hidden Layers - Extract features.
 - Output Layer - Produces final prediction.
 - Special layers: Convolutional, Recurrent, Dropout, etc.
4. Activation Functions:
 - Like Sigmoid, ReLU, Tanh.
 - Add non-linearity to the model.
5. Connection Patterns:
 - Fully connected (Dense), Convolutional, Recurrent, Residual, etc.
6. Regularization Techniques:
 - Dropout, Batch Normalization to prevent overfitting.



Examples:

A typical deep learning architecture for image classification might look like:

Input Layer (Image pixels)



Convolutional Layers



Activation (ReLU)



Pooling Layers



Fully Connected Layers



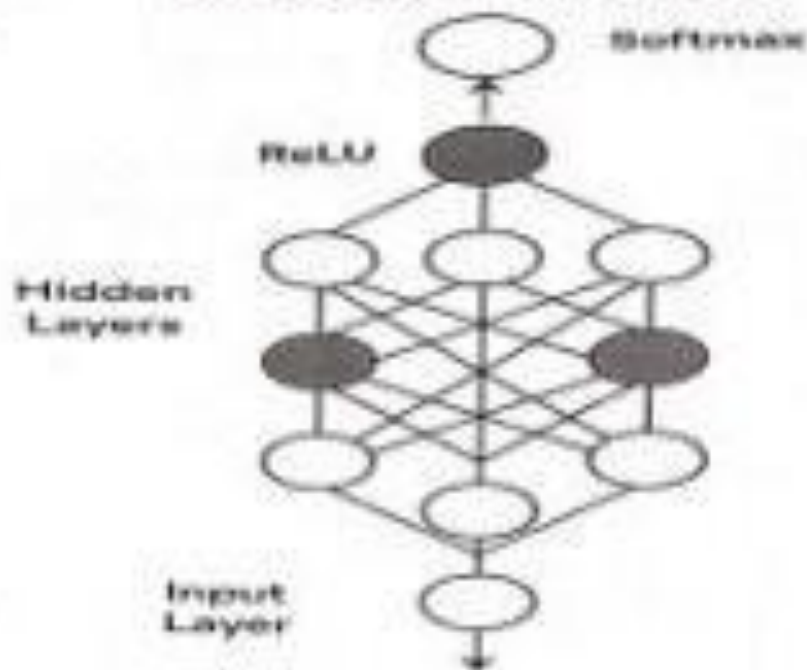
Output Layer (Softmax)

Importance of Architecture Design:

- Determines accuracy, training time, and generalization.
- A well-designed architecture avoids underfitting and overfitting.
- Impacts model's ability to capture complex features.



ARCHITECTURE DESIGN IN DEEP LEARNING



Supervised deep learning

Supervised learning refers to the problem space wherein the target to be predicted is clearly labelled within the data that is used for training.

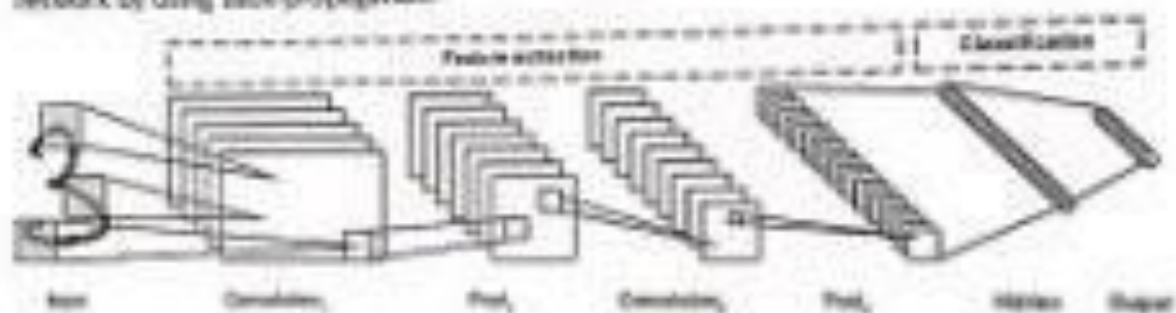
In this section, we introduce at a high-level two of the most popular supervised deep learning architectures - convolutional neural networks and recurrent neural networks as well as some of their variants.

Convolutional neural networks



A CNN is a multilayer neural network that was biologically inspired by the animal visual cortex. The architecture is particularly useful in image-processing applications. The first CNN was created by Yann LeCun, at the time, the architecture focused on handwritten character recognition, such as postal code interpretation. As a deep network, early layers recognize features (such as edges), and later layers recombine these features into higher-level attributes of the input.

The latest CNN architecture is made up of several layers that implement feature extraction and then classification (see the following image). The image is divided into receptive fields that feed into a convolutional layer, which then extracts features from the input image. The next step is pooling, which reduces the dimensionality of the extracted features (through down-sampling) while retaining the most important information (typically, through max pooling). Another convolution and pooling step is then performed that feeds into a fully connected multilayer perceptron. The final output layer of this network is a set of nodes that identify features of the image (in this case, a node per identified number). You train the network by using back-propagation.



The use of deep layers of processing, convolutions, pooling, and a fully connected classification layer opened the door to various new applications of deep learning neural networks. In addition to image processing, the CNN has been successfully applied to video recognition and various tasks within natural language processing.

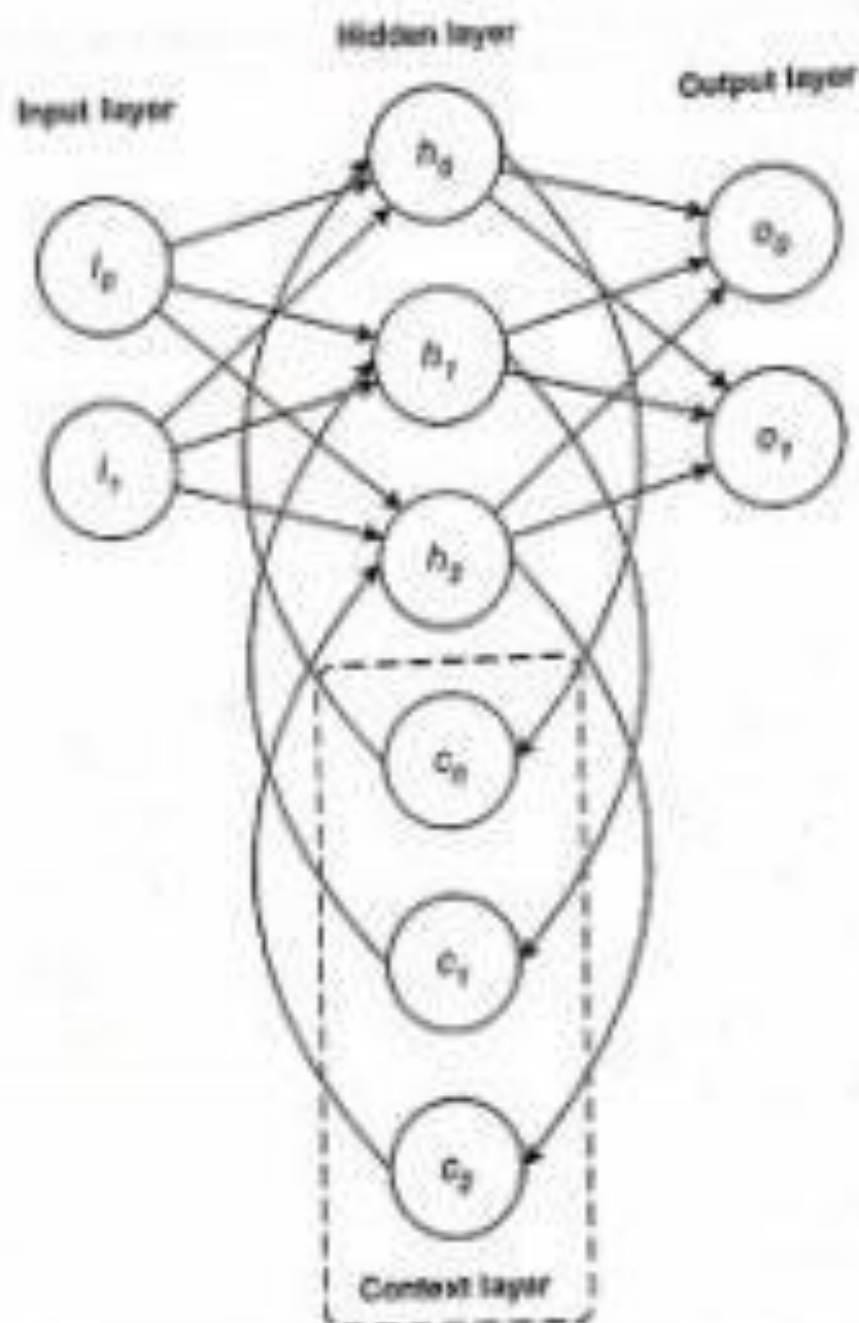
Example applications: Image recognition, video analysis, and natural language processing

Recurrent neural networks

The RNN is one of the foundational network architectures from which other deep learning architectures are built. The primary difference between a typical multilayer network and a recurrent network is that rather than completely feed-forward connections, a recurrent network might have connections that feed back into prior layers (or into the same layer). This feedback allows RNNs to maintain memory of past inputs and model problems in time.

RNNs consist of a rich set of architectures (we'll look at one popular topology called LSTM next). The key differentiator is feedback within the network, which could manifest itself from a hidden layer, the output layer, or some combination thereof.





NNs can be updated in time and trained with standard back-propagation or by using a variant of back-propagation that is called back-propagation in time (BPTT).

Example applications: Speech recognition and handwriting recognition.

LSTM networks

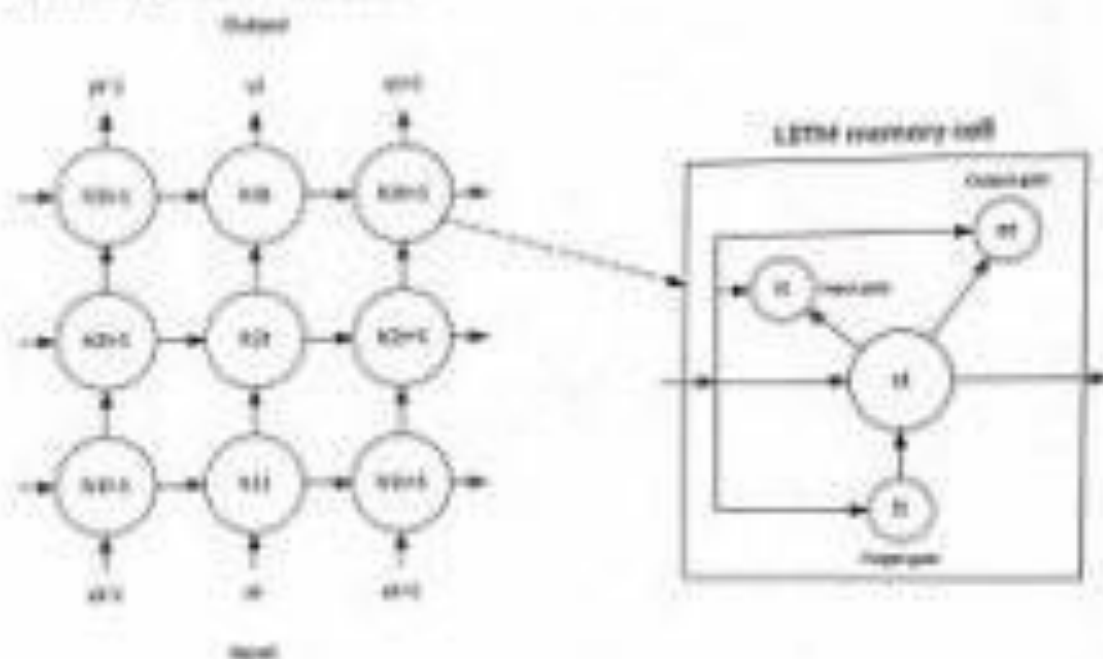
The LSTM was created in 1987 by Hochreiter and Schmidhuber, but it has grown in popularity in recent years as an RNN architecture for various applications. You'll find LSTMs



in products that you use every day, such as smartphones. IBM applied LSTMs in IBM Watson⁴ for milestone-winning conversational speech recognition.

The LSTM departed from typical neuron-based neural network architectures and instead introduced the concept of a memory cell. The memory cell can retain its value for a short or long time as a function of its inputs, which allows the cell to remember what's important and not just its last computed value.

The LSTM memory cell contains three gates that control how information flows into or out of the cell. The input gate controls when new information can flow into the memory. The forget gate controls when an existing piece of information is forgotten, allowing the cell to remember new data. Finally, the output gate controls when the information that is contained in the cell is used in the output from the cell. The cell also contains weights, which control each gate. The training algorithm, commonly BPTT, optimizes these weights based on the resulting network output errors.



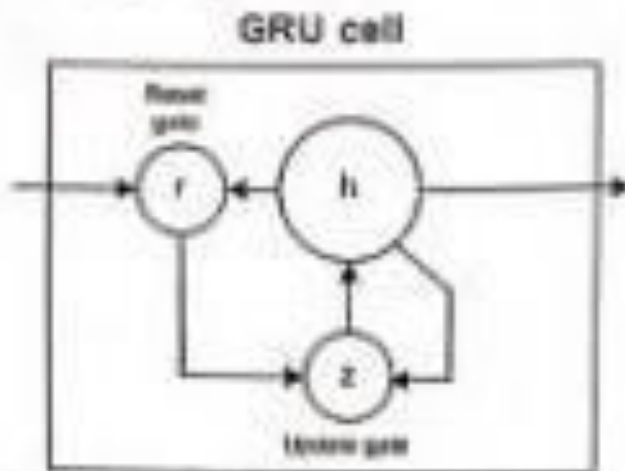
Recent applications of CNNs and LSTMs produced image and video captioning systems in which an image or video is captured in natural language. The CNN implements the image or video processing, and the LSTM is trained to convert the CNN output into natural language.

Example applications: Image and video captioning systems

GRU networks

In 2014, a simplification of the LSTM was introduced called the gated recurrent unit. This model has two gates, getting rid of the output gate present in the LSTM model. These gates are an update gate and a reset gate. The update gate indicates how much of the previous cell contents to maintain. The reset gate defines how to incorporate the new input with the

previous cell contents. A GRU can model a standard RNN simply by setting the reset gate to 1 and the update gate to 0.



The GRU is simpler than the LSTM, can be trained more quickly, and can be more efficient in its execution. However, the LSTM can be more expressive and with more data can lead to better results.

Example applications: Natural language text compression, handwriting recognition, speech recognition, gesture recognition, image captioning

Unsupervised deep learning

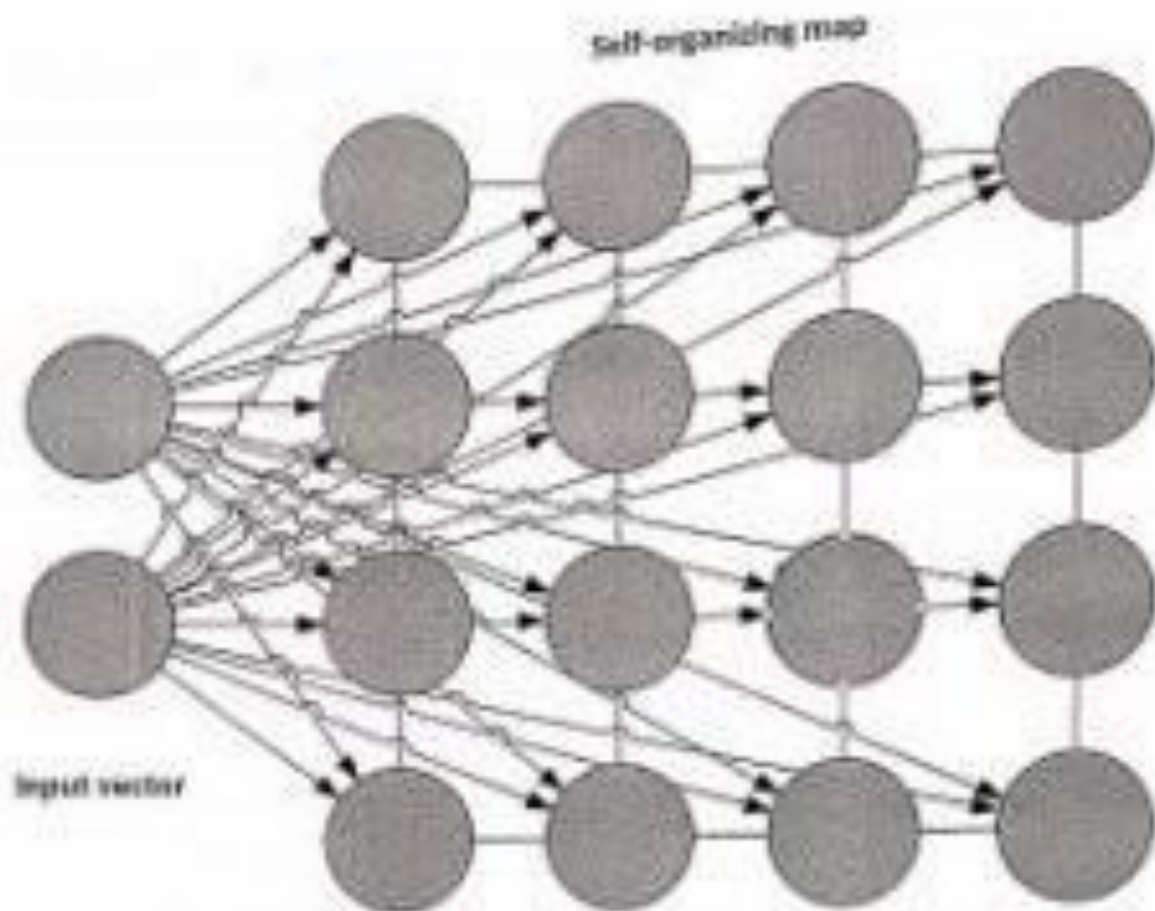
Unsupervised learning refers to the problem space wherein there is no target label within the data that is used for training.

This section discusses three unsupervised deep learning architectures: self-organised maps, autoencoders, and restricted boltzmann machines. We also discuss how deep belief networks and deep stacking networks are built based on the underlying unsupervised architecture.

Self-organised maps

Self-organised map (SOM) was invented by Dr. Teuvo Kohonen in 1982 and was popularly known as the Kohonen map. SOM is an unsupervised neural network that creates clusters of the input data set by reducing the dimensionality of the input. SOMs vary from the traditional artificial neural network in quite a few ways.





The first significant variation is that weights serve as a characteristic of the node. After the inputs are normalized, a random input is first chosen. Random weights close to zero are initialized to each feature of the input record. These weights now represent the input node. Several combinations of these random weights represent variations of the input node. The euclidean distance between each of these output nodes with the input node is calculated. The node with the least distance is declared as the most accurate representation of the input and is marked as the best matching unit or BMU. With these BMUs as center points, other units are similarly calculated and assigned to the cluster that it is the distance from. Radius of points around BMU weights are updated based on proximity. Radius is shrunk.

Next, in an SOM, no activation function is applied, and because there are no target labels to compare against there is no concept of calculating error and back propagation.

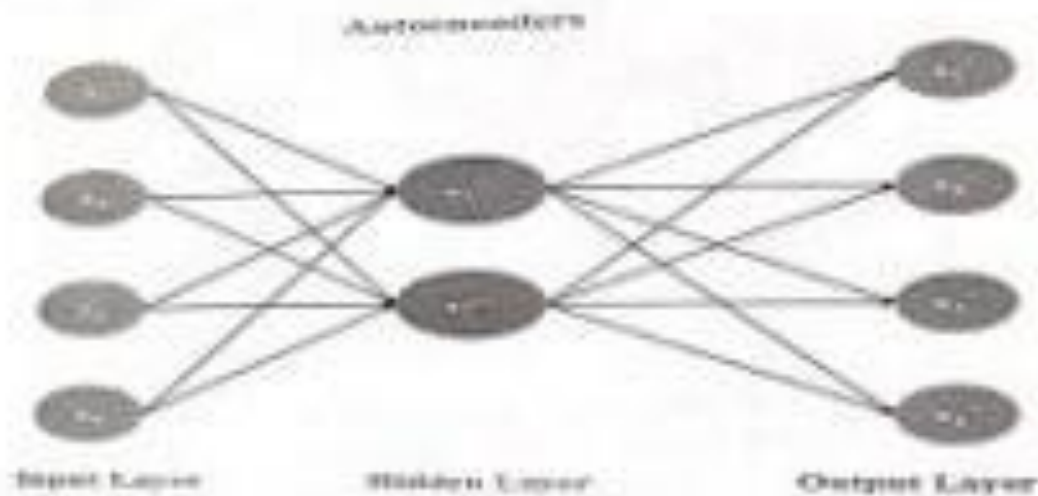
Example applications: Dimensionality reduction, clustering high-dimensional inputs to 2-dimensional output, feature space search, and cluster visualization

Autoencoders

Though the history of when autoencoders were invented is hazy, the first known usage of autoencoders was found to be by LeCun in 1987. This variant of an ANN is composed of 3 layers: input, hidden, and output layers.



First, the input layer is encoded into the hidden layer using an appropriate encoding function. The number of nodes in the hidden layer is much less than the number of nodes in the input layer. This hidden layer contains the compressed representation of the original input. The output layer aims to reconstruct the input layer by using a decoder function.



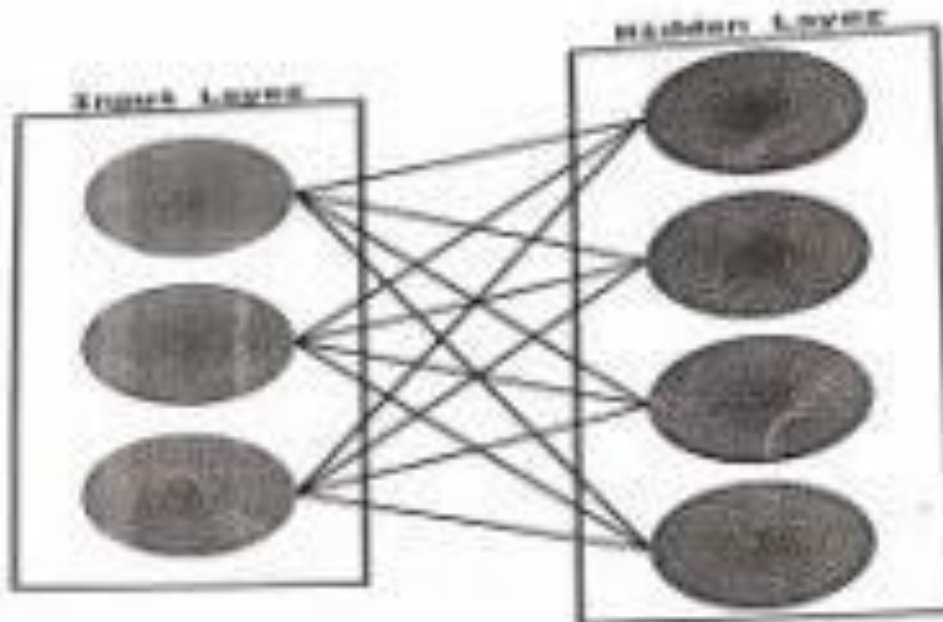
During the training phase, the difference between the input and the output layer is calculated using an error function, and the weights are adjusted to minimize the error. Unlike traditional unsupervised learning techniques, where there is no data to compare the outputs against, autoencoders learn continuously using backward propagation. For this reason, autoencoders are classified as self-supervised algorithms.

Example applications: Dimensionality reduction, data interpolation, and data compression/decompression

Restricted Boltzmann Machines

Though RBMs became popular much later, they were originally invented by Paul Smolensky in 1986 and was known as a *stochastic*.

An RBM is a 2-layered neural network. The layers are input and hidden layers. As shown in the following figure, in RBMs every node in a hidden layer is connected to every node in a visible layer. In a traditional Boltzmann Machine, nodes within the input and hidden layer are also connected. Due to computational complexity, nodes within a layer are not connected in a Restricted Boltzmann Machine.



During the training phase, RBMs calculate the probability distribution of the training set using a stochastic approach. When the training begins, each neuron gets activated at random. Also, the model contains respective hidden and visible bias. While the hidden bias is used in the forward pass to build the activation, the visible bias helps in reconstructing the input.

Because in an RBM the reconstructed input is always different from the original input, they are also known as generative models.

Also, because of the built-in randomness, the same predictions result in different outputs. In fact, this is the most significant difference from an autoencoder, which is a deterministic model.

Example applications: Dimensionality reduction and collaborative filtering

Deep belief networks

The DBN is a typical network architecture, but includes a novel training algorithm. The DBN is a multilayer network (typically deep and including many hidden layers) in which each pair of connected layers is an RBM. In this way, a DBN is represented as a stack of RBMs.

In the DBN, the input layer represents the raw sensory inputs, and each hidden layer learns abstract representations of this input. The output layer, which is treated somewhat differently than the other layers, implements the network classification. Training occurs in two steps: unsupervised pretraining and supervised fine-tuning.

Hyperparameters And Validation sets :

In ML, hyperparameters are parameters whose values are set before the learning process begins and are not learned directly from the training data. They control the behaviour, capacity, and learning process of a model.

Examples of Hyperparameters include ~~scale~~ learning rate, number of hidden layers, number of neurons, regularization parameters, batch size, and number of training epochs.

- * Model complexity
- * Speed of learning
- * Risk of over-fitting and under-fitting

Working steps :

1. Split dataset into training, validation and test data sets
2. Select initial hyperparameter values.
3. Train the model using training data
4. Evaluate performance using validation data
5. Modify hyperparameters
6. Repeat training and evaluation
7. Select best model
8. Test final model on test data.

Grid Search in Hyperparameters Tuning :

Grid search is a systematic hyperparameter optimization technique used in ML to find the best combination of hyperparameters by exhaustively trying all possible values from a predefined set and evaluating model performance using cross-validation.

Working steps :

1. Choose the ML model
2. Define hyperparameter grid
3. Apply Grid Search with cross-validation.
4. Train model on all parameter combinations
5. Compute performance score
6. Select best parameters
7. Retrain final model with best parameters

Random Search :

Random Search is a hyperparameter optimization technique in ML where random combinations of hyperparameters are selected from a predefined range and evaluated using cross-validation to find the best performing model.

Working steps :

1. Define the Range/distribution of hyperparameters
2. Randomly select a combination.
3. Train the model
4. Evaluate using cross-validation
5. Repeat for a fixed number of trials
6. Select the best performing combinations.

Grid Search Example problem :

Cross validation	Kernel	Accuracy(%)
0.1	Linear	82
0.1	RBF	84
1	Linear	86
1	RBF	89
10	Linear	88
10	RBF	91

① Number of model Trained

Total combinations = $3 \times 2 = 6$ models

② Best parameter combination

High accuracy = 91%

occurs at $c=10$, Kernel = RBF

③ Reason - Grid Search selects the combination with maximum cross validation accuracy.

Random Search :

Trial	K	Accuracy(%)
1	3	85
2	7	80
3	5	90
4	9	86

* Highest accuracy = 90%

* Corresponding value of $K=5$

* Optimal hyperparameters (K) = 5

* Best cross-validation accuracy
= 90%

Estimators :

In machine learning and statistics, an estimator is a rule, method (or) mathematical function used to estimate unknown parameters of a model based on observed data. The estimator takes a dataset as input and produces an estimated values of a parameter such as mean, variance, regression coefficients, (or) probability distributions.

Estimators work by applying mathematical formulae (or) optimization techniques to the available data. Based on assumptions about the underlying data distribution, the estimator calculates parameter values that best represent the data. Different estimators may produce different estimates depending on their design and assumptions.

Types of Estimators :

1. point Estimators : provide a single value estimate of a parameter [ex: Sample Mean]

2. Interval Estimators : provide a range within which the parameter is expected to lie

Common Estimators :

1. Mean Estimators
2. Variance Estimators
3. Maximum Likelihood Estimator
4. Bayesian Estimator
5. Least Square Estimators

Working steps :

1. Collect sample data
2. Assume a statistical model (or) distribution
3. Apply the Estimator formula
4. Compute parameter values
5. Use Estimated parameters in the model.

Estimator formula :

$$\hat{\theta} = f(x_1, x_2, \dots, x_n)$$

General formula

$$\hat{\theta} = \text{Estimator}(X)$$

Where : $X = (x_1, x_2, \dots, x_n) \rightarrow$ Training Data

θ = True parameter (Unknown)

$\hat{\theta}$ = Estimated parameter

* Mean Estimator :

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$$

* Variance Estimator :

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

* Maximum Likelihood Estimator (MLE) :

$$\hat{\theta}_{MLE} = \arg \max_{\theta} p(X|\theta)$$

* Linear Regression Estimator (OLS) :

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

Bayesian Estimator (MAP)

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta} P(\theta/x)$$
$$P(\theta/x) \propto P(x/\theta) P(\theta)$$

(Using Bayes Theorem)

* Estimator $\rightarrow$ Formula (or) Rule

* Estimate $\rightarrow$ Numerical value after applying Estimator

Numerical Example 1:

$$X = \{2, 4, 6, 8, 10\}$$

Step 1 : Estimator Formula

The Mean Estimator is : $\hat{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$

Where : n = Number of samples

x_i = Sample values

Step 2 : Substituting values

No. of. Samples $n = 5$

Step 3 : Final Estimate

$$\hat{\mu} = 6$$

The estimated population mean is 6

Variance Estimator :

$$\text{Variance Estimator Formula} = \hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

Step 1 : compute Mean

$$\bar{x} = 6$$

Step 2 : Compute squared differences

x_i	$x_i - \bar{x}$	$(x_i - \bar{x})^2$	
2	-4	16	$\Rightarrow x_i - \bar{x} = 2 - 6 = -4$
4	-2	4	$\Rightarrow 4 - 6 = -2$
6	0	0	$\Rightarrow 6 - 6 = 0$
8	2	4	$\Rightarrow 8 - 6 = 2$
10	4	16	$\Rightarrow 10 - 6 = 4$

$$\sum (x_i - \bar{x})^2 = 40$$

Step 3 : calculate Variance

$$(\hat{\sigma})^2 = \frac{40}{5-1} = \frac{40}{4} = 10$$

$$(\hat{\sigma})^2 = 10$$

Final values are $\hat{\mu} = 6, (\hat{\sigma})^2 = 10$

Bias And Variance :

Bias : Bias is the error due to overly simple assumptions in the learning algorithms.

- * Model is too simple

- * fails to capture the true pattern

Result : Under-fitting , High Training Error, High Testing Error

Example : Using linear regression for a highly Non-linear dataset

Variance : Variance is the error due to excessive sensitivity of the model to training data.

- * Model too complex

- * learns noise along with data

Result : overfitting , very low training error, High Testing Error

Example : Very deep decision tree on a small dataset

Bias-Variance Trade-off :

1. Increasing model complexity $\downarrow$ Bias , But variance $\uparrow$
2. Decreasing model complexity $\uparrow$ Bias , But $\downarrow$ variance
3. Goal : Balance both for best generalization.

Mathematical Insight : $\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Noise}$

How to reduce :

① Bias

- * Use complex models
- * Add features
- * Reduce Regularization.

② Variance

- * More Training Data
- * Regularization (L_1/L_2)
- * pruning / Dropout / Bagging.

Bayesian statistics :

Bayesian statistics is a framework that updates the probability of a hypothesis as more data becomes available, using Bayes' Theorem.

$$\text{Bayes' Theorem : } P(\theta|D) = \frac{P(D|\theta) P(\theta)}{P(D)}$$

Where : $P(\theta) \rightarrow$ prior [Initial belief about parameters]

$P(D|\theta) \rightarrow$ Likelihood [Data given parameters]

$P(\theta|D) \rightarrow$ posterior [updated belief after seeing data]

$P(D) \rightarrow$ Marginal likelihood / Evidence.

- Advantages :
1. Incorporates prior knowledge
 2. Handles uncertainty well
 3. Works well with small datasets
 4. Probabilistic Interpretation

Example problem :

A Machine learning system is used to detect spam mails.
prior probability that an email is spam

$$P(\text{spam}) = 0.30$$

$$P(\text{Not-spam}) = 0.70$$

* probability that the word 'Free' appears in spam mail

$$P(\text{Free}|\text{spam}) = 0.80$$

Find the probability that an email is spam given that it contains the word "free".

Step 1 : Write Baye's Theorem

$$P(\text{spam} | \text{free}) = \frac{P(\text{free} | \text{spam}) \cdot P(\text{spam})}{P(\text{free})}$$

Step 2 : Compute Evidence

$$P(\text{free}) = P(\text{free} | \text{spam}) P(\text{spam}) + P(\text{free} | \text{Not spam}) P(\text{Not spam})$$

$$P(\text{free}) = (0.80)(0.30) + (0.10)(0.70)$$

$$P(\text{free}) = 0.24 + 0.07$$

$$P(\text{free}) = 0.31$$

Step 3 : Compute posterior

$$P(\text{spam} | \text{free}) = \frac{0.80 \times 0.30}{0.31}$$

$$P(\text{spam} | \text{free}) = \frac{0.24}{0.31} \approx 0.774$$

$$P(\text{spam} | \text{free}) \approx 77.4\%$$

Maximum Likelihood Estimators :

Maximum Likelihood Estimators is a statistical method used to estimate the parameters of a model by maximizing the probability of the observed training data, given those parameters.

Mathematical formula :

$$\hat{\theta}_{MLE} = \arg \max_{\theta} \sum_{i=1}^n \log P(x_i / \theta)$$

Training Data $D = x_1, x_2, \dots, x_n$

Model parameter $= \theta$

$$L(\theta) = P(D/\theta) = \prod_{i=1}^n P(x_i / \theta)$$

problem : Given the following observations assumed to be drawn from Gaussian (Normal) distribution. $X = \{2, 4, 6, 8\}$

Estimate : Mean () using MLE

Variance () using MLE

step 1 : MLE formulas for Gaussian Distribution

For Normal Distribution

$$\mu_{MLE} = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\sigma^2_{MLE} = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

step 2 : Compute Mean (μ_{MLE})

$$\mu_{MLE} = \frac{2+4+6+8}{4}$$

$$\boxed{\mu_{MLE} = \frac{20}{4} = 5}$$

Step 3 : compute Variance (σ_{MLE}^2)

x_i	$x_i - \mu$	$(x_i - \mu)^2$
2	$2 - 5 = -3$	9
4	$4 - 5 = -1$	1
6	$6 - 5 = 1$	1
8	$8 - 5 = 3$	9

$$\sum (x_i - \mu)^2 = 9 + 1 + 1 + 9 = 20$$

$$\sigma_{MLE}^2 = \frac{20}{4} = 5$$

$$\mu_{MLE} = 5, \sigma_{MLE}^2 = 5$$

Stochastic Gradient Descent :

Stochastic Gradient Descent is an optimization algorithm used to minimize a loss function by updating model parameters using the gradient computed from a single training example at each iteration.

- * Gradient Descent : Uses entire dataset to compute Gradient
 - * Stochastic GD : Uses one random data point
 - * Mini-Batch GD : Uses small subset (batch)
- SGD introduces randomness making updates faster and more frequent.

Algorithm steps :

1. Initialize parameters θ randomly
2. Shuffle training data
3. For Each training sample (x_i, y_i)
 - * compute gradient of loss
 - * update parameters
4. Repeat for multiple epochs

Linear Regression - Example problem

Hypothesis - $y = wx + b$

$$\text{Loss (MSE)} - L = (y - \hat{y})^2$$

Update Rules - $W = W - \eta$

Example problem :

Given Training Data (single feature)

X	Y
1	2
2	4

Assume a linear model

$$y = wx$$

Loss function : Mean Squared Error for one sample

$$J(w) = \frac{1}{2} (wx - y)^2$$

Learning Rate : $\eta = 0.1$

Initial weight : $w = 0$

prediction : $\hat{y} = wx$
 $\hat{y} = 0 \times 1$
 $\boxed{\hat{y} = 0}$

Error :

$$\hat{y} - y = 0 - 2$$

$$\boxed{\hat{y} - y = -2}$$

Gradient : $\frac{\partial J}{\partial w} = (wx - y)x$
 $= (-2)(1)$

$$\boxed{\frac{\partial J}{\partial w} = -2}$$

Weight update :

$$w = w - \eta \cdot \text{gradient}$$

$$w = 0 - 0.1(-2)$$

$$\boxed{w = 0.2}$$

$$x=2, y=4$$

prediction :

$$\hat{y} = 0.2 \times 2 = 0.4$$

Error : $0.4 - 4$

$$= -3.6$$

Gradient = $(-3.6)(2)$

$$= -7.2$$

Weight update

$$W = 0.2 - 0.1(-7.2)$$

$$W = 0.92$$

Final Weight after one Epoch

$$W = 0.92$$

Advantages :

1. Efficient for large-scale datasets
2. Faster updates and lower memory usage
3. Can escape local minima due to noisy updates
4. Suitable for online learning.

Disadvantages :

1. Loss function fluctuates
2. Sensitive to learning rate
3. May not converge exactly to the minimum.

UNIT-IV

Convolution Networks

① The convolutional operations

- * Mathematical definition of convolution

- * Discrete convolution in images

- * Concept for Filter / kernel

- * Stride and padding

② pooling

- * Max pooling

- * Average pooling

- * Global pooling

purpose : Dimensionality reduction & Translation invariance

③ Convolution

- * 2D Convolution

- * Multi-channel convolution

- * Feature map generation

- * parameter sharing

④ Basic convolutional functions

- * ReLU after convolution

- * Linear Convolutional layer

- * Activation + convolution combination

⑤ structured outputs

- * Pixel-wise prediction

- * Segmentation outputs

- * Dense prediction models

⑥ Data types

- * Image data representation

- * Tensor formats (NCHW/NHWC)

- * Feature map dimensions

⑦ Efficient Convolution Algorithms

- * Im2col method

- * FFT-based convolution

- * Depthwise separable convolution

- * 1×1 convolution

⑧ Random (or) Unsupervised Features

- * Random filter initialization

- * Unsupervised feature learning

- * pre-training methods

⑨ Basis for Convolution Networks

- * Local connectivity

- * parameter sharing

- * Translation invariance

- * Sparse interactions

① What is convolutional networks ?

Ans A Convolutional Neural Networks is a deep learning Architecture specially designed for processing grid-structured such as images.

Instead of full connections, CNNs a mathematical operation called convolution, which extracts spatial features using small learnable filters (kernels).

The convolutional operations

The convolutional operations in convolutional Neural Networks It is used to extract features like edges, textures, shapes from input data such as images.

A standard convolution applies a small filters over an input image (or) feature map to compute a weighted sum at each position.

* The same filter is used across the entire image

* Each filter learns to detect a specific pattern

Ex : Vertical Edge, Curve

* The result is a feature map (or) Activation map

Formula :

$$(I * K)(i, j) = \sum_m \sum_n I(m, n) K(i-m, j-n)$$

Where : $I \rightarrow$ Input image

$K \rightarrow$ Kernel Filter

$(i, j) \rightarrow$ output pixel location

It performs element-wise multiplication + summation

output size formula (Stride & padding)

$$\text{output} = ((N - F + 2P) / S) + 1$$

Where

N - Input size

F - Filter size

P - padding

S - Stride

Example problem for convolution with stride & padding

Consider the following input image matrix I of size 4×4

$$I = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 4 & 1 & 2 & 1 \\ 1 & 3 & 1 & 0 \\ 2 & 2 & 1 & 4 \end{bmatrix}$$

A convolutional operation is performed using

the kernel K of size 2×2 : $K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. The stride $S=2$ and padding $P=1$.

Ans Given Input image (4×4)

$$I = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 4 & 1 & 2 & 1 \\ 1 & 3 & 1 & 0 \\ 2 & 2 & 1 & 4 \end{bmatrix}$$

$$\text{Kernel } (2 \times 2) : K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\text{Stride } (S) = 2$$

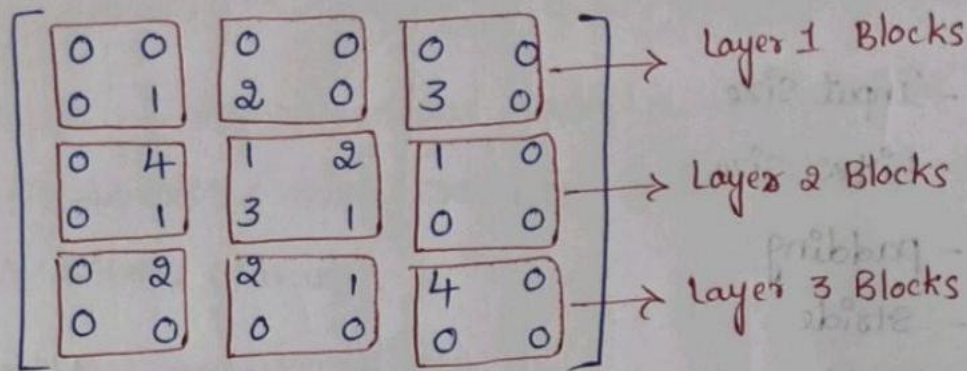
$$\text{padding} = 1$$

$$\text{New size} = N + 2P$$

$$= 4 + 2(1) = 6 \quad \text{So padded image size } 6 \times 6$$

Step 1 : Add padding ($p=1$)

After zero padding 6×6 matrix



Step 2 : Calculate output size :

$$\text{output} = ((N - F + 2p) / S) + 1$$

$$N=4, F=2, p=1, S=2$$

$$\text{output} = ((4 - 2 + 2(1)) / 2) + 1$$

$$= (4/2) + 1$$

$$= 2 + 1$$

$$= 3$$

$$\boxed{\text{output} = 3}$$

$$\boxed{\text{output} = 3 \times 3}$$

Step 3 : perform convolution (stride = 2) (1,1)

Take first 2×2 block $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

multiply : $(0 \times 1) + (0 \times 0) + (0 \times 0) + (1 \times -1)$

$$= -1$$

$$\boxed{\text{Block 1} = -1}$$

Block 2 : (1,2) :

$$\begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\text{multiply} = (0 \times 1) + (0 \times 0) + (2 \times 0) + (0 \times -1) \\ = 0$$

$$\boxed{\text{Block 2} = 0}$$

Block 3 : (1,3)

$$\begin{bmatrix} 0 & 0 \\ 3 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\text{multiply} = (0 \times 1) + (0 \times 0) + (3 \times 0) + (0 \times -1) \\ = 0$$

$$\boxed{\text{Block 3} = 0}$$

Block 4 : (2,1) (2,1)

$$\begin{bmatrix} 0 & 4 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$(0 \times 1) + 4(0) + 0 \times 0 + 1 \times -1 \\ = -1$$

$$\boxed{\text{Block (2,1)} = -1}$$

Block (2,2) :

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$= 0$$

Block (2,3) :

$$\begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = 2$$

$$\boxed{\text{Block (2,3)} = 2}$$

Block (3,1) :

$$\begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = -2$$

$$\text{Block (3,1)} = -2$$

Block (3,2) :

$$\begin{bmatrix} 3 & 1 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = 2$$

$$\text{Block (3,2)} = 2$$

$$\text{Block (3,3) : } \begin{bmatrix} 1 & 0 \\ 1 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = -3$$

$$\text{Block (3,3)} = -3$$

Final output :

$$\begin{bmatrix} -1 & 0 & 0 \\ -1 & 0 & 2 \\ -2 & 2 & -3 \end{bmatrix}$$

Convert Image to Column Matrix :

Step 1 :

Extract 2x2 patches $X = \begin{bmatrix} 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 4 & 1 & \dots \\ 1 & 2 & 3 & \dots \end{bmatrix}$

Each column = flattened patch

Step 2 :

Flatten Kernel : $K = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$

Step 3 :

Matrix Multiplication $Y = K^T \cdot X$

Advantages of convolution

1. Parameter sharing
2. Sparse connectivity
3. Translation invariance
4. Automatic feature learning

Applications :

1. Image Classification
2. Face Recognition
3. Medical imaging
4. Object detection
5. self driving cars

Real Time Examples :

1. Mobile Unlock
2. Amazon product Recommendation
3. YouTube & Net-flix Thumbnail Detection
4. Instagram & Snapchat Filters

Convolutional Neural Networks use the convolutional operation to extract spatial features from images using learnable kernels. The output size depends on stride and padding. Convolutional can also be represented as matrix multiplication for computational efficiency. CNNs are highly effective in computer vision applications due to parameter sharing and local connectivity.

* stride : stride is the number of pixels by which the convolution filter (kernel) moves over the input image

* padding : Padding means adding extra pixels around the border of an input image before applying convolution. (Adding zeros)
(Around image edges)

Efficient Convolutional Algorithms:

Efficient convolutional algorithms are methods and techniques designed to optimize the computation of **convolution operations**, which are at the core of many signal processing tasks and deep learning models, such as Convolutional Neural Networks (CNNs). The goal of these algorithms is to reduce the time complexity, memory usage, and computational cost of convolutions while maintaining or improving the quality of results.

Since convolutions are often used for filtering, pattern recognition, and feature extraction (especially in CNNs), optimizing this operation is critical to make models faster and more efficient—especially when working with large datasets, real-time applications, or devices with limited resources (e.g., mobile phones, embedded systems).

Key Goals of Efficient Convolution:

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.

Improve scalability: Efficient algorithms ensure that convolutions can scale better as the input size increases. **Key Goals of Efficient Convolution:**

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.
4. **Improve scalability:** Efficient algorithms ensure that convolutions can scale better as the input size increases.

Common Approaches for Efficient Convolution:

1. Fourier Transform Approach (FFT-based Convolution)

The Fourier Transform approach leverages the Fast Fourier Transform (FFT) to convert a convolution operation into a multiplication operation in the frequency domain, which is often more efficient computationally.

How It Works:

Convolution in the spatial domain (direct summation over the sliding window) can be computationally expensive, especially for large kernels. The Fourier Transform approach converts

Key Goals of Efficient Convolution:

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.
4. **Improve scalability:** Efficient algorithms ensure that convolutions can scale better as the input size increases.

both the input signal (or image) and the kernel (filter) into the frequency domain using FFT.

- **Convolution Theorem:** The convolution of two signals in the time (or spatial) domain is equivalent to the pointwise multiplication of their respective Fourier transforms:

$$y(t) = (x * h)(t) \quad \text{in time domain}$$

is equivalent to

$$Y(f) = X(f) \cdot H(f) \quad \text{in frequency domain}$$

Where $X(f)$ and $H(f)$ are the Fourier Transforms of the input and filter, respectively, and $Y(f)$ is their product. After multiplication, we take the inverse Fourier Transform (IFFT) to get the result back in the spatial domain.

Why It's Efficient:

- In the frequency domain, convolution becomes multiplication, which is computationally faster. Using FFT reduces the computational complexity from $O(N^2)$ to $O(N \log N)$, making it much faster for large kernels.

Use Case:

- **Large convolutions** where the kernel is large, and direct spatial-domain convolutions would be too slow. For example, in deep learning when you're working with larger image inputs or larger kernels.

2. Depthwise Convolutions

Depthwise convolutions are a variant of convolutions where each filter operates on only one input channel at a time, rather than combining multiple input channels as in a standard convolution. This greatly reduces the number of computations.

How It Works:

- **Standard Convolution:** In a standard convolution, a single filter slides over the input and computes outputs by combining information across all input channels. This involves a full matrix of filters spanning all input channels.

- **MobileNets** (which use depthwise separable convolutions) and other lightweight networks use separable convolutions to reduce computation without sacrificing performance.

4. Structured Convolutions

Structured convolutions refer to the use of convolution patterns or constraints that are designed to reduce computation, improve performance, or better fit the problem domain. These structures often focus on optimizing the way filters are applied or reducing redundant computations by enforcing certain patterns on the filter design or the way they are applied.

How It Works:

Structured convolutions can take many forms, such as:

- **Low-rank convolutions:** Using low-rank matrices to approximate convolutions. These matrices have fewer parameters, which reduces computation.
- **Group convolutions:** Partitioning the input channels into groups, where each group gets its own set of filters. This reduces the computational load by limiting the number of channels that interact with each filter.
- **Block or Toeplitz convolutions:** Some algorithms focus on using block-wise structures, where the filter or input is divided into smaller blocks, and only the non-zero parts are computed.
- **Factorized convolutions:** Factorizing the convolution operation into smaller convolutions, similar to separable convolutions, but with more complex factorizations (e.g., using two 3×3 convolutions instead of one 5×5).

Why It's Efficient:

- **Reduces redundancy:** By enforcing a structure, such as low-rank approximations, group convolutions, or others, redundant computations are avoided, and the number of parameters and operations is minimized.
- **Better generalization:** Structured convolutions can lead to models that generalize better because they reduce overfitting (by limiting the number of parameters or enforcing constraints on the filters).

Use Case:

- **ResNeXt, Inception-v4, and MobileNetV2** utilize structured convolutions (group convolutions or factorized convolutions) to improve both efficiency and model performance. **EfficientNet** uses this idea of "compound scaling" to balance depth, width, and resolution of the model while applying structured operations like depthwise separable convolutions.

Summary of Each Approach:

1. **Fourier Transform Convolution:**
 - Transforms convolution into frequency domain multiplication (more efficient for large convolutions).
2. **Depthwise Convolution:**

- Applies a separate filter for each input channel (reduces parameter count and computation).
- 3. **Separable Convolution:**
 - Decomposes 2D convolutions into two 1D convolutions (reduces computation significantly).
- 4. **Structured Convolutions:**
 - Utilizes specific constraints or patterns (low-rank, group convolutions, etc.) to reduce computation, enhance performance, or adapt to problem-specific needs.

Each of these methods targets different efficiency goals, and often in practical implementations, a combination of them is used to achieve a balance between computational cost and model performance. Let me know if you'd like more details or practical examples on any of them!

Fourier Transform Approach problem.

Formula : $x * k \longleftrightarrow F^{-1}(F(x) \cdot F(k))$

Where

x = input

k = kernel

F = Fourier Transform, F^{-1} = Inverse Fourier Transform

$*$ = Convolution

$$F = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4 + x_5 y_5 + x_6 y_6$$

We have to consider 5x5 matrix perform a 2D convolution of the given input matrix $\hat{I}$ with the kernel K . show step-by-step calculations for at least two positions of the kernel and then write the final output matrix.

The Input matrix is $\hat{I} = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 2 \\ 7 & 8 & 9 & 0 & 1 \\ 1 & 3 & 5 & 2 & 4 \\ 2 & 4 & 6 & 3 & 5 \end{bmatrix}$ Kernel (K) = $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Solution: pointwise calculation

$$\text{Given Input } \hat{I} = \begin{bmatrix} 1 & 2 & 3 & 0 & 2 \\ 4 & 5 & 6 & 0 & 1 \\ 7 & 8 & 9 & 0 & 2 \\ 2 & 3 & 4 & 0 & 4 \\ 2 & 4 & 6 & 3 & 5 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Step 1: Elementwise product with K , output position $(1,1)$, Top-left using input Rows 1-3, column 1-3.

$$\text{patch} : \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{kernel } (K) = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{Formula} = x * K = F^{-1}(F(x) \cdot F(y))$$

$$\begin{bmatrix} x_1 y_1 & x_2 y_2 & x_3 y_3 \\ x_4 y_4 & x_5 y_5 & x_6 y_6 \\ x_7 y_7 & x_8 y_8 & x_9 y_9 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

(1,1)

$$F = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4 + x_5 y_5 + x_6 y_6 + x_7 y_7 + x_8 y_8 + x_9 y_9$$
$$= 1 \times 1 + 2 \times 0 + 3 \times (-1) + 4 \times 1 + 5 \times 0 + 6 \times (-1) + 7 \times 1 + 8 \times 0 + 9 \times (-1)$$

$$\text{Sum } (F) = 1 + 0 - 3 + 4 + 0 - 6 + 7 + 0 - 9$$
$$= -6$$

$$\boxed{\text{output } (x_1) \ (1,1) = -6}$$

: output position (1,2), input Rows (2-4), column (1-3)

$$= \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 0 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

product $\begin{bmatrix} 2 \times 1 & 3 \times 0 & 0 \times -1 \\ 5 \times 1 & 6 \times 0 & 0 \times -1 \\ 8 \times 1 & 9 \times 0 & 0 \times -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 5 & 0 & 0 \\ 8 & 0 & 0 \end{bmatrix}$

Sum = $2+0+0+5+0+0+8+0+0 = 15$

output (1,2) = 15

Step 3 : output position (1,3), input Rows (3,5), column (1,3)

$$(1,3) = \begin{bmatrix} 3 & 0 & 2 \\ 6 & 0 & 1 \\ 9 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

product $\begin{bmatrix} 3 \times 1 & 0 \times 0 & 2 \times -1 \\ 6 \times 1 & 0 \times 0 & 1 \times -1 \\ 9 \times 1 & 0 \times 0 & 2 \times -1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & -2 \\ 6 & 0 & -1 \\ 9 & 0 & -2 \end{bmatrix}$

Sum = $3+0+2+6+0-1+9+0-2 = 13$

output (1,3) = 13

Step 4 : output position (2,1), input Rows (2,4), columns (1,3)

patch $\begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 2 & 3 & 4 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

product $\begin{bmatrix} 4 \times 1 & 5 \times 0 & 6 \times -1 \\ 7 \times 1 & 8 \times 0 & 9 \times -1 \\ 2 \times 1 & 3 \times 0 & 4 \times -1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & -6 \\ 7 & 0 & -9 \\ 2 & 0 & -4 \end{bmatrix}$

Sum = $4+0-6+7+0-9+2+0-4 = -6$

output (2,1) = -6

Step 5: output position (2,2), Input Rows (2,4), columns (2,4)

$$\text{patch} \begin{bmatrix} 5 & 6 & 0 \\ 8 & 9 & 0 \\ 3 & 4 & 0 \end{bmatrix} \quad \text{kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 5 \times 1 & 6 \times 0 & 0 \times -1 \\ 8 \times 1 & 9 \times 0 & 0 \times -1 \\ 3 \times 1 & 4 \times 0 & 0 \times -1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 \\ 8 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

$$\text{Sum} = 5 + 0 + 0 + 8 + 0 + 0 + 3 + 0 + 0 = 16$$

$$\boxed{\text{output}(2,2) = 16}$$

Step 6: output position (2,3), Input Rows (2,4), col (3,5)

$$\text{patch} \begin{bmatrix} 6 & 0 & 1 \\ 9 & 0 & 2 \\ 4 & 0 & 4 \end{bmatrix} \quad \text{kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 6 \times 1 & 0 \times 0 & 1 \times -1 \\ 9 \times 1 & 0 \times 0 & 2 \times -1 \\ 4 \times 1 & 0 \times 0 & 4 \times -1 \end{bmatrix} = \begin{bmatrix} 6 & 0 & -1 \\ 9 & 0 & -2 \\ 4 & 0 & -4 \end{bmatrix}$$

$$\text{Sum} = 6 + 0 - 1 + 9 + 0 - 2 + 4 + 0 - 4 = 12$$

$$\boxed{\text{output}(2,3) = 12}$$

Step 7: output position (3,1), Input Rows (3,5), columns (1,3)

$$\boxed{\text{output}(3,1) = -8}$$

Step 8: output position (3,2), Input Rows (3,5) column (2,4)

$$\boxed{\text{output}(3,2) = 12}$$

Step 9: output position (3,3), Input (3,5) col (3,5)

$$\boxed{\text{output}(3,3) = 8}$$

$$\text{Final output matrix}(3 \times 3) \begin{bmatrix} -6 & 15 & 13 \\ -6 & 16 & 12 \\ -8 & 12 & 8 \end{bmatrix}$$

- **Depthwise Convolution:** In depthwise convolution, each input channel has its own filter. For example, if you have 3 input channels and 3 output channels, instead of using a $3 \times 3 \times 3 \times 3$ filter for each output channel, you will use 3 separate $3 \times 3 \times 3$ filters, one for each input channel.
 - If the input has 3 channels and we use a $3 \times 3 \times 3$ filter, instead of performing $3 \times 3 = 9 \times 3 = 27$ operations per output channel, we would perform only 3 operations (one per input channel).

Why It's Efficient:

- **Reduces the number of parameters:** Since each channel is processed independently, the number of filters required is much smaller.
- **Reduces computations:** Depthwise convolution significantly cuts down on the number of operations, especially for deeper networks where the number of channels increases.

Use Case:

- **MobileNets** and other lightweight networks use depthwise convolutions to reduce computation and memory usage, especially for mobile or edge devices.

3. Separable Convolutions

Separable convolutions break down a 2D convolution into two smaller 1D convolutions — one in the horizontal direction and another in the vertical direction. This reduces computational cost while maintaining the effectiveness of the convolution.

How It Works:

- **Standard 2D Convolution:** A standard 2D convolution, such as a $5 \times 5 \times 5$ kernel, requires 25 multiplications for each output pixel (if using a single filter). For each output, you compute a weighted sum of the neighboring pixels.
- **Separable Convolution:** A $5 \times 5 \times 5$ filter can be decomposed into two 1D filters: one $5 \times 1 \times 1$ filter and one $1 \times 5 \times 5$ filter. First, the input image is convolved with the $5 \times 1 \times 1$ filter, and then the result is convolved with the $1 \times 5 \times 5$ filter. This reduces the number of multiplications from 25 to $5 + 5 = 10$, which is much more efficient.

This is possible because convolution is a linear operation, and thus separable filters still approximate the original 2D convolution well.

Why It's Efficient:

- **Reduces computational cost:** By breaking down 2D convolutions into two 1D convolutions, the number of operations is drastically reduced. For an $k \times k \times k$ kernel, instead of k^3 operations, you only need $k + k = 2k$ operations.

Use Case:

Input

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 2 \\ 7 & 8 & 9 & 0 & 1 \\ 1 & 3 & 5 & 2 & 4 \\ 2 & 4 & 6 & 3 & 5 \end{bmatrix}$$

$$\text{Kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Step 1 : (1,1) Top left path And Kernel values product

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{Kernel} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = -6$$

$$\text{Step 2} : \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 1 \\ 8 & 9 & 0 \end{bmatrix} \quad \text{Kernel} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 14$$

$$\text{Step 3} : \begin{bmatrix} 3 & 0 & 1 \\ 6 & 1 & 2 \\ 9 & 0 & 1 \end{bmatrix} \quad \text{Kernel} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 14$$

$$\text{Step 4} : \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 3 & 5 \end{bmatrix} \quad \text{Kernel} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = -8$$

$$\text{Step 5} : \begin{bmatrix} 5 & 6 & 1 \\ 8 & 9 & 0 \\ 3 & 5 & 2 \end{bmatrix} \quad \text{Kernel} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 13$$

step 6 : $\begin{bmatrix} 6 & 1 & 2 \\ 9 & 0 & 1 \\ 5 & 2 & 4 \end{bmatrix}$ Kernel $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 13$

step 7 : $\begin{bmatrix} 7 & 8 & 9 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ Kernel $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = -10$

step 8 : $\begin{bmatrix} 8 & 9 & 0 \\ 3 & 5 & 2 \\ 4 & 6 & 3 \end{bmatrix}$ Kernel $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 10$

step 9 : $\begin{bmatrix} 9 & 0 & 1 \\ 5 & 2 & 4 \\ 6 & 3 & 5 \end{bmatrix}$ Kernel $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = 10$

Convolutional Networks

- ① The Convolutional operations
- ② Motivation
- ③ Pooling
- ④ Convolution and Pooling as an Infinitely strong prior
- ⑤ Variants of the Basic Convolution Function
- ⑥ structured outputs
- ⑦ Data Types
- ⑧ Efficient Convolutional Algorithms
- ⑨ Random or unsupervised Features
- ⑩ The Neuro-scientific Basis for Convolutional Networks
- ⑪ Convolutional Networks and the History of Deep learning.

What is convolutional Neural Networks [CNNs]

Convolutional Neural Networks (CNNs) are specialized type of deep Neural Networks primarily used for processing structured arrays of data such as images. They are especially powerful for image classification, object detection, and computer vision tasks.

Main components of a CNN :

1. Input layer
2. Convolutional layer
3. Activation Function [Relu]
4. Pooling Layer
5. Fully Connected layers [Dense layer]
6. output layer

① Input layer :

Input layer Takes input in the form of an image

Eg : $32 \times 32 \times 3$ for a color image

② Convolutional layer :

* Applies a set of filters (kernels) that slide over the input to produce feature maps.

* Each filter detects different features

Eg : Edges, corners, textures

* Uses operation called convolution.

③ Activation Function [ReLU] :

* Applies a non-linearity like ReLU [Rectified linear unit]

$$\text{ReLU}(x) = \max(0, x)$$

* Add non-linearity to the Network.

④ Pooling layer :

* Downsamples the feature maps to reduce dimensionality and computation.

* Common Types - max pooling

Average pooling

⑤ Fully Connected layers [Dense layers] :

* Flattens the output from convolutional / pooling layers and makes predictions.

* Final layer often uses softmax for classification.

⑥ output layer :

Gives the final prediction.

Eg : class probabilities for image classification.

Advantages of CNNs :

1. parameter sharing reduces the number of parameters
2. Translation invariance [Detects patterns regardless of location]
3. Effective in image and video recognition, medical image analysis and more.

Real Time Applications :

1. Facial Recognition [Eg - Face ID]
2. Self Driving Cars [object detection]
3. Medical diagnosis [Eg - X-Ray Analysis]
4. Image classification [Eg - cat vs dog]

History of Deep learning And CNN Architectures :

Deep learning is a subfield of Machine learning that uses multi-layered neural networks. Here is how it evolved over time.

Major Milestones in Deep Learning :

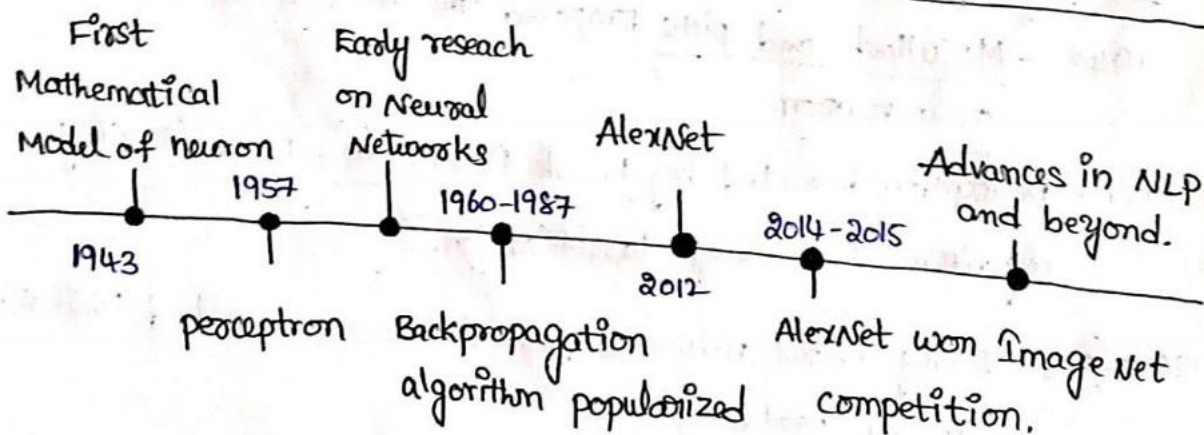
- | <u>Year</u> | <u>Event</u> |
|-------------|---|
| ① 1943 | - <u>McCulloch and Pitts</u> proposed the first mathematical model of a neuron |
| ② 1957 | - Perceptron invented by <u>Frank Rosenblatt</u> - first learning algorithm for binary classification. |
| ③ 1960s-70s | - Early neural networks gained interest but had limitations
Eg: XOR problems |
| ④ 1986 | - Backpropagation algorithm popularized by <u>Rumelhart, Hinton and Williams</u> - made deep training possible. |
| ⑤ 1998 | - LeNet-5 by Yann LeCun used CNNs for digit recognition using Bank for Reading cheques. |

Yann LeCun introduced Deep Belief Networks, reviving interest in deep models.

- ⑦ 2012 - AlexNet won ImageNet competition - deep learning boom begins [used GPUs and ReLU]
- ⑧ 2014-2015 - Advanced models like VGG, GoogLeNet, and ResNet revolutionized deep learning.
- ⑨ 2016 - Now - Deep learning dominates in NLP (Transformers), speech, Robotics, and healthcare. Tools like TensorFlow, PyTorch helped rapid growth.

Why deep learning became successful (post-2012)

- * Huge labeled datasets
- * Fast GPUs for parallel computation
- * Improved architectures
- * Effective training algorithms [Adam, Dropout, Batch Normalization]
- * Open source frameworks [TensorFlow, PyTorch]

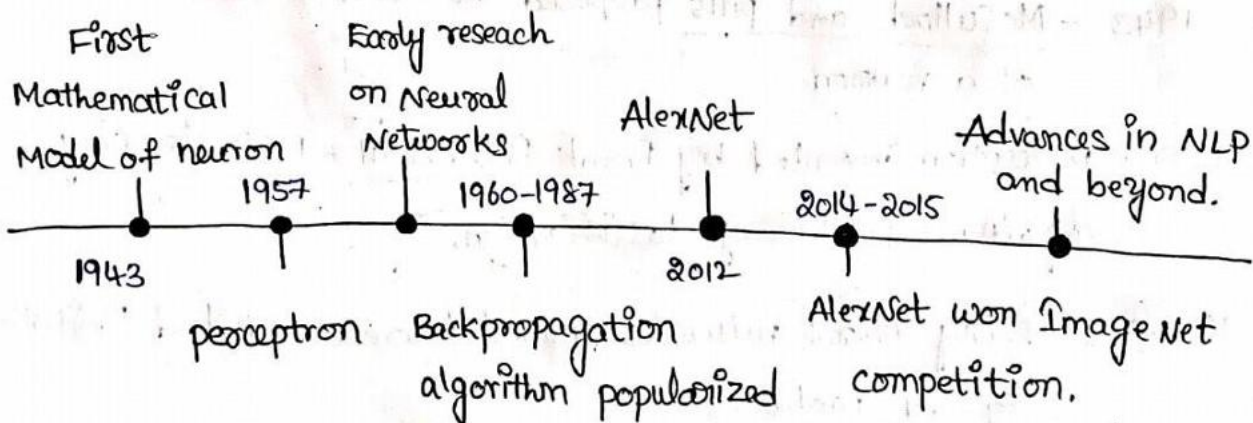


Time line chart of deep learning History

- ⑥ 2006 - Geoffrey Hinton introduced Deep Belief Networks; reviving interest in deep models.
- ⑦ 2012 - AlexNet won ImageNet competition - deep learning boom begins [used GPUs and ReLU]
- ⑧ 2014-2015 - Advanced models like VGG, GoogLeNet, and ResNet revolutionized deep learning.
- ⑨ 2016 - Now - Deep learning dominates in NLP (Transformers), speech Robotics, and healthcare. Tools like TensorFlow, PyTorch helped rapid growth.

Why deep learning Became Successful (post-2012)

- * Huge labeled datasets
- * Fast GPUs for parallel computation
- * Improved architectures
- * Effective training algorithms [Adam, Dropout, Batch Normalization]
- * Open source frameworks [TensorFlow, PyTorch]



Time line chart of deep learning History

operations

- ① standard convolution - feature detection - edges, patterns - AlexNet
- ② Depthwise - Efficient per-channel filtering - MobileNet
- ③ pointwise - channel combination (1x1 conv) - MobileNet, ResNet
- ④ Dilated - large receptive field - Segmentation
- ⑤ Transposed - up sampling - Image Generations, GANs
- ⑥ Grouped - ResNext, Efficient parallel conv
- ⑦ Strided - learnable down sampling
- ⑧ padding - everywhere in CNNs.

* The convolutional operations :

The convolutional operations are the building blocks of CNNs used especially in image and video data. The operation is designed to extract features like edges, textures, shapes etc, from input data by applying filters (kernels).

A convolution is a mathematical operation where a small matrix called a filter (or) kernel slides over the input like an image and at each position, compute a dot product b/w the filter and overlapping input values.

* convolutional parameters :

1. Kernel - The small matrix used to scan input
2. Stride - Number of steps filter moves at a time
3. padding - Adding zeros around input to preserve size
... border area.

① The standard Convolution operation :

The standard convolution is the core operation in convolutional Neural Networks (CNNs). It is used to extract features (like edges, textures, shapes) from input data such as images.

A standard convolution applies a small filter over an input image (or) feature map to compute a Weighted Sum at each position.

- * The same filter is used across the entire image.

- * Each filter learns to detect a specific pattern

Eg : Vertical edge, curve

- * The result is a feature map (or) Activation map

Mathematical Representation :

For a 2D image I and a kernel (K), the convolution at position (i, j) is $S(i, j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m, n)$

Where $I(i+m, j+n)$ - pixel values from the input

$K(m, n)$ - Kernel values (weights)

$S(i, j)$ - output value at position (i, j)

Example problem :

Let's say you have a 3×3 input and a 2×2 filter.

Input $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ Filter (Kernel) $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Ans Standard 2D Convolution - Text Diagram

Input Image (3x3) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

Convolution - Edge Detection

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \rightarrow -4$
kernel output

Kernel (Filter) (2x2) $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Input Image Filter

slide the filter over the input

Step 1 : Top-left corners (covering values 1, 2, 4, 5)

Input patch $\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$

Apply kernel = $(1 \times 1) + (2 \times 0) + (4 \times 0) + (5 \times -1)$
 $= 1 - 5$
 $= -4$

Step 2 : Top Right corners (covering values 2, 3, 5, 6)

Input patch $\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$

Apply kernel = $(2 \times 1) + (3 \times 0) + (5 \times 0) + (6 \times -1)$
 $= 2 - 6$
 $= -4$

Step 3 : Bottom left corners (covering values 4, 5, 7, 8)

Input patch $\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}$

Apply kernel = $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

$= (4 \times 1) + (5 \times 0) + (7 \times 0) + (8 \times -1)$
 $= 4 - 8$
 $= -4$

Step 4 : Bottom-Right corner (covered values (5, 6, 8, 9))

$$\text{Input patch} \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

$$\text{Apply Kernel} \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$(5 \times 1) + (6 \times 0) + (8 \times 0) + (9 \times -1) = 5 - 9 \\ = -4$$

$$\text{output feature Map (2x2)} = \begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

Result :

1. Each output value is the result of a dot product between the kernel and the region it overlaps.
2. This output map highlights where the pattern defined by the kernel appears in the input.

Aspect

Standard Convolution

- ① Type - Linear Transformation
- ② Filter - Small matrix of learnable weights
- ③ Operation - sliding dot-product
- ④ output - Feature Map
- ⑤ use - Feature extraction from spatial data.

Dot product formula : $A = [x_1, x_2, x_3, \dots, x_n]$

$$B = [y_1, y_2, y_3, \dots, y_n]$$

$$(x_1 \times y_1) + (x_2 \times y_2) + (x_3 \times y_3) + \dots + (x_n \times y_n)$$

② Depthwise operations :

In depthwise convolution, each input channel is convolved with its own separate filters, instead of applying one filter across all channels.

Example :

Suppose you have an input of size $H \times W \times 3$ (Height $\times$ width $\times$ channels) and you use standard convolution - And Depth convolution.

Applies a separate filter to each input channel instead of combining them.

purpose : Efficient per-channel filtering reducing computations.

Ex : MobileNet (lightweight networks for mobile devices)

③ Pointwise operations :

A 1×1 convolution that combines features across channels without affecting spatial dimensions.

purpose : channel combination and dimensionality change.

(Helps in merging features after depthwise convolutions)

④ Dilated convolution :

convolution with spaces (holes) b/w kernel elements, increasing the receptive field without increasing parameters.

(Captures long-range dependencies.)

Ex : Semantic segmentation, audio processing.

⑤ Transposed convolution : Also called deconvolution (or) upconvolution used to increase spatial resolution. (image generation)

⑥ Grouped Convolution:

splits the input channels into groups and applies convolutions separately within each group.

Reduce computation, enable specialized feature learning.

Benefit: Allows parallel processing, reducing memory usage.

⑦ Strided Convolution:

convolution with a stride > 1 , which downsamples the spatial dimensions.

learnable downsampling instead of pooling.

Reduces size while learning features.

⑧ padding: Adding extra pixels around the input before convolution.

purpose: control output size and preserve spatial information.

Types: valid padding: No padding, size shrinks

same padding: padding so output size = input size.

Motivation in CNNs:

The Motivation for using convolutional Neural Networks in deep learning comes from the need to efficiently process and understand data that has spatial structure, especially images and videos.

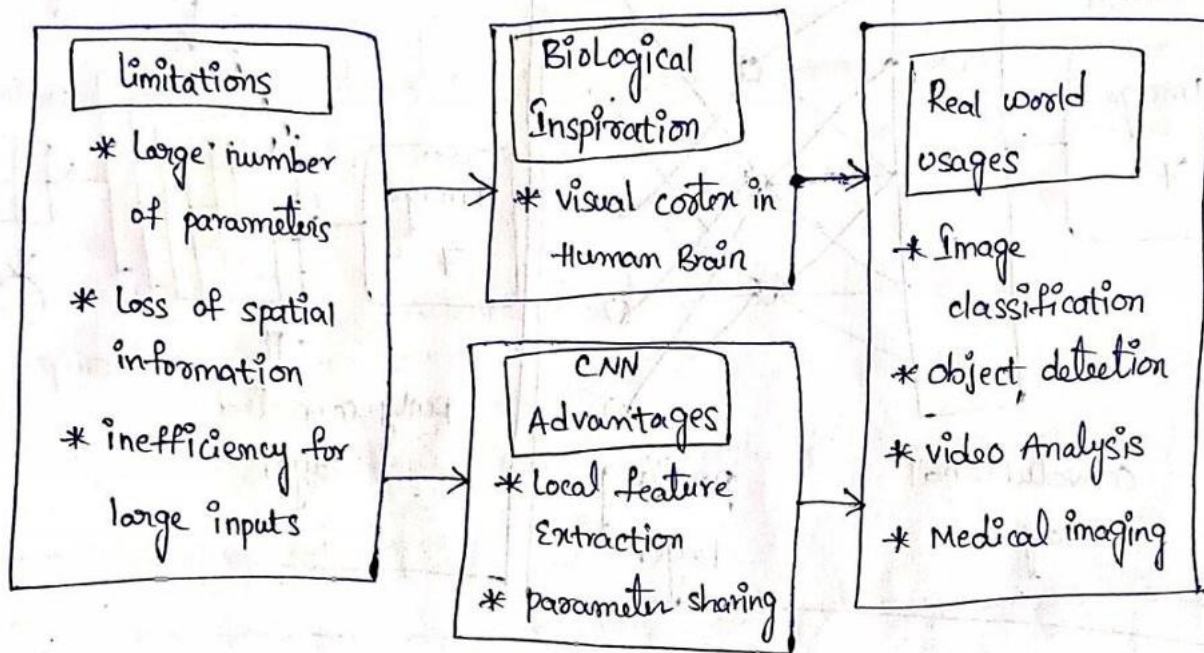
1. Limitations of Traditional Neural Networks

2. Biological Inspiration

3. Core Motivations for CNNs

4. Real world motivation

Motivation in CNNs



1. Limitations of Traditional Neural Networks :

In the context of CNNs, a Traditional Neural Network usually refers to the fully connected neural network that existed before convolutional layers became common for image and spatial data tasks.

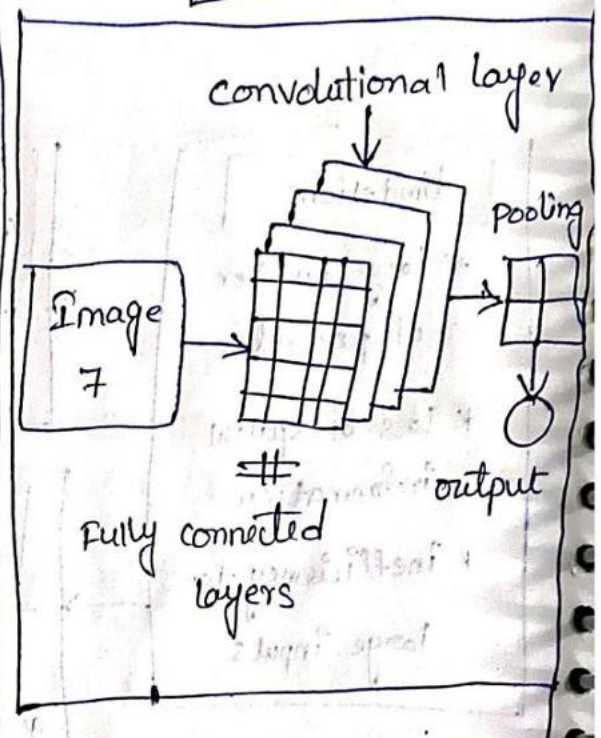
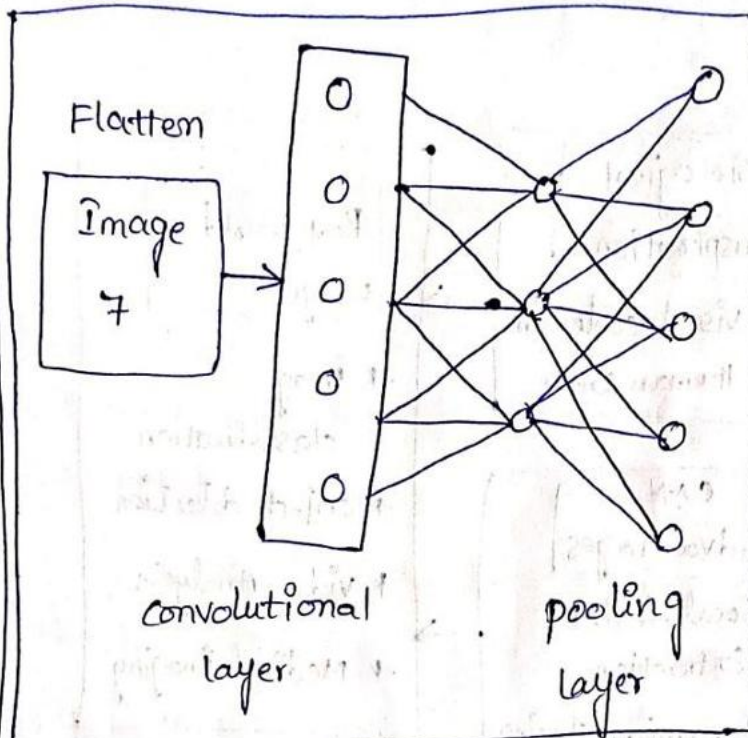
- * Each neuron in one layer is connected to every neuron in the next layer.

- * Input data is flattened into a 1D vector before processing.

- * The network learns by adjusting weights on all those connections.

Traditional Neural Network

CNNs



Example : If you have a 28×28 grayscale image like MNIST

* Traditional NN :

1. Flatten into 784 features (28×28)
2. Fully connected to the first hidden layer say 128 Neurons.
3. continue with more hidden layers until output.

* CNNs :

1. Keep the 2D structure
2. Use convolutional filters to learn local patterns.
3. Far fewer parameters and better spatial feature learning.

2. Biskog

1. Limitations of Traditional Neural Networks:

* High number of parameters:

In a fully connected n/w, each neuron connects to every input pixel, leading to a massive number of parameters when processing image.

* Loss of Spatial information:

Flattening an image into a 1D vector before feeding it to a n/w destroys spatial relationships b/w pixels.

* Inefficiency for large inputs:

The computation and memory requirements grow quickly with input size.

2. Biological Inspiration:

* CNNs are inspired by the visual cortex in the human brain, neurons in the visual cortex respond to small, localized regions of an image.

* This biological mechanism motivated the local connectivity and shared weights in CNN.

pooling in CNNs

In Deep learning (DL), pooling is a downsampling operation used mainly in CNNs to reduce the spatial size (width $\times$ height) of feature maps while retaining the most important information.

pooling is a technique that summarizes the features with in a small region of the input (called the pooling window or filter) into a single representative value. It helps make the network more computationally efficient, less prone to overfitting and translation invariant.

Why pooling is needed :

1. Reduce dimensionality $\rightarrow$ less memory and computation.
2. control overfitting $\rightarrow$ fewer parameters in later layers
(By reducing the number of features)
3. provides translation invariance $\rightarrow$ small shifts in the image won't affect the output much.

Types of pooling :

1. Max pooling
2. Average pooling

1. Max pooling :

Max pooling takes the maximum value from the each patch (region). It's good for capturing strong features (edges, textures)

2. Average pooling

Average pooling takes the calculate the average of all the values inside that region. Replace that region with this single average

2	2	7	3
9	4	6	1
8	5	2	4
3	1	2	6

Max
Pool →

Filter - (2x2)

Stride - (2x2)

9	7
8	6

Step-by-step solution:

① Block 1

2	2
9	4

max → 9

② Block 2

7	3
6	1

max → 7

③ Block 3

8	5
3	1

max → 8

④ Block 4

2	4
2	6

max → 6

② Average pooling:

2	2	7	3
9	4	6	1
8	5	2	4
3	1	2	6

Average pool

Filter - (2x2)

stride - (2,2)

4.25	4.25
4.25	3.5

① Block 1

2	2
9	4

$$\text{Sum} = 2 + 2 + 9 + 4 = 17$$

$$\text{Average} = 17 / 4 = 4.25$$

② Block 2

7	3
6	1

$$\text{Sum} = 7 + 3 + 6 + 1 = 17$$

$$\text{Average} = 17 / 4 = 4.25$$

③ Block 3

8	5
3	1

$$\text{Sum} = 8 + 5 + 3 + 1 = 17$$

$$\text{Average} = 17 / 4 = 4.25$$

(u) Block 4 :

2	4
2	6

 Sum = $2+4+2+6 = 14$
Average = $14/4 = 3.5$

Advantages :

1. Dimensionality Reduction
2. Control overfitting
3. Translation Invariance
4. Highlights Important features
5. Simplifies Representation
6. Improves network Robustness

Disadvantages of pooling :

1. Loss of spatial Information
2. Fixed operation
3. Can remove subtle features
4. Not always needed.

have internal structure (or) relationships among components, rather than being a single scalar (or) independent label.

Examples :

1. Sequence (time series, text)
2. Grids (images, maps)
3. Trees (parse trees, decision trees)
4. Graphs (social networks, molecules)

1. Sequence :

The outputs are ordered elements the position/order matters

- * Text Generation - predicting the next word in a sentence.
- * Speech Recognition - sequence of words from audio.
- * Stock price prediction - time series values.

DL Models :

- ① RNNs
- ② LSTMs & GRUs
- ③ Transformers.

② Grids : outputs are arranged in 2D (or) 3D arrays.

- * Image segmentation - pixel-by-pixel classification.
- * Object Detection - bounding box grids.
- * Medical imaging - Tumor detection in MRI scans.

DL Models :

CNNs
FCNs
U-Net - Biomedical segmentation.

Structured outputs :

In Deep learning structured outputs refers to predictions that have internal structure (or) relationships among components, rather than being a single scalar (or) independent label.

Examples :

1. Sequence (Time series, text)
2. Grids (Images, maps)
3. Trees (parse trees, decision trees)
4. Graphs (Social networks, molecules)

1. Sequence :

The outputs are ordered elements the position/order matters

- * Text Generation - predicting the next word in a sentence
- * Speech Recognition - sequence of words from audio
- * Stock price prediction - time series values.

DL Models :

- ① RNNs
- ② LSTMs & GRUs
- ③ Transformers.

② Grids : outputs are arranged in 2D (or) 3D arrays.

- * Image segmentation - pixel-by-pixel classification.
- * Object Detection - bounding box grids.
- * Medical imaging - Tumor detection in MRI scans.

DL Models :

CNNs

FCNs

U-Net - Biomedical Segmentation

3. Trees - outputs follows a hierarchical structure

- * NLP

- * Decision Trees in structured prediction Tasks.

- * Hierarchical document classification.

4. Graphs - social media (outputs are in the form of nodes and edges with flexible connections).

- * Social Networks - predicting user connections.

- * Traffic Networks - roads and intersections.

DL Models

- * Graph Neural Networks (GINNs)

- * Graph convolutional networks

- * Graph attention Networks

Data Types in CNNs

Input Images - int 8 / float 32

weights & Biases - float 32 / float 16 / int 8

Feature Maps - float 32 / float 16

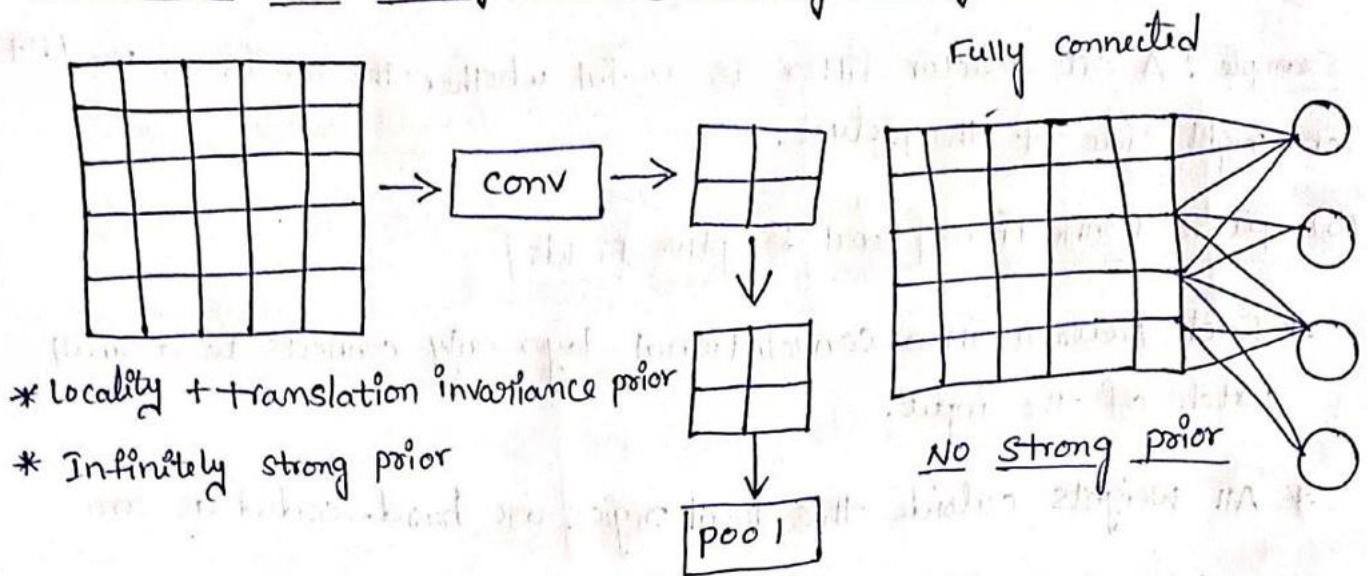
Activations - float 32 / float 16

Gradients - float 32 (Training)

loss values - float 32

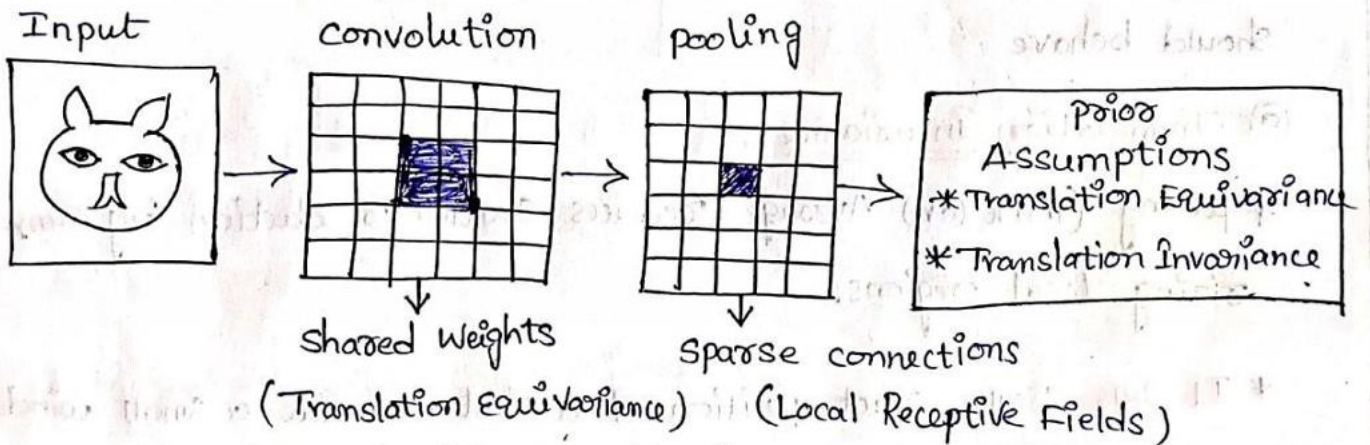
output predictions - float 32 / int 8

Convolution and pooling as an Infinitely Strong prior:



* Convolution = Infinitely strong prior of locality + translation equivariance.

* pooling = Infinitely strong prior of translation invariance.



1. Convolution as an Infinitely strong prior:

Convolution encodes two main assumptions

① shared weights (Translation Equivariance)

② sparse connections (Local Receptive Fields)

① shared weights : (Reduce parameters).

* The same filter (kernel) is applied across different spatial locations of the image.

* This means if the filter detects a vertical edge at the top-left corner, it can also detect the same edge at the bottom Right.

prior assumption : pattern matters, not their exact position.

Example : A cat detector filter is useful whether the cat is on the left (or) right side of the picture.

(b) Sparse Connections [Local Receptive Fields]

* Each neuron in a convolutional layer only connects to a small patch of the input.

* All weights outside this local region are hard-coded as zero

* prior assumption : Local pixels influence each other more than distant pixels.

(2) Pooling as an Infinitely Strong prior :
pooling introduces another hard-coded assumption about how features should behave

(a) Translation Invariance :

* pooling (max (or) Average) reduces spatial resolution by summarizing local regions.

* It says : The exact position of a feature inside a small window doesn't matter, only its presence does.

(b) Dimensionality reduction : fewer computations and faster training

(c) Regularization : reduces overfitting by forcing simpler representations.

Efficient convolutional Algorithms in convolutional Networks

Efficient convolutional Algorithms :

Efficient convolutional Algorithms in convolutional Networks are techniques and mathematical optimizations, that reduce ~~the~~ the computational cost, memory usage, and inference / Training time of the convolutional operations, while preserving accuracy. Since convolution is the most expensive operation in CNNs, (reservations) researchers have developed multiple efficient methods.

① Why Efficiency is needed

1. High computation cost
2. Mobile and Edge devices
3. Training speed
4. Real-Time applications
5. Energy Savings

② 1. Major Efficient convolutional Algorithms - Matrix Multiplication Trick

2. Winograd Minimal Filtering
3. FFT-Based convolutions

① Fourier Transform Approach

4. Depthwise Separable convolution

5. Group convolution

6. Dilated convolution

7. Low Rank Approximations

8. sparse convolutions

9. Strassen-like Algorithms.

② separable convolutional

③ structured convolution

④ Depthwise convolutional

Efficient Convolutional Algorithms:

Efficient convolutional algorithms are methods and techniques designed to optimize the computation of **convolution operations**, which are at the core of many signal processing tasks and deep learning models, such as Convolutional Neural Networks (CNNs). The goal of these algorithms is to reduce the time complexity, memory usage, and computational cost of convolutions while maintaining or improving the quality of results.

Since convolutions are often used for filtering, pattern recognition, and feature extraction (especially in CNNs), optimizing this operation is critical to make models faster and more efficient—especially when working with large datasets, real-time applications, or devices with limited resources (e.g., mobile phones, embedded systems).

Key Goals of Efficient Convolution:

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.

Improve scalability: **Efficient algorithms ensure that convolutions can scale better as the input size increases. Key Goals of Efficient Convolution:**

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.
4. **Improve scalability:** Efficient algorithms ensure that convolutions can scale better as the input size increases.

Common Approaches for Efficient Convolution:

1. Fourier Transform Approach (FFT-based Convolution)

The Fourier Transform approach leverages the Fast Fourier Transform (FFT) to convert a convolution operation into a multiplication operation in the frequency domain, which is often more efficient computationally.

How It Works:

Convolution in the spatial domain (direct summation over the sliding window) can be computationally expensive, especially for large kernels. The Fourier Transform approach converts

Key Goals of Efficient Convolution:

1. **Reduce computational cost:** Convolutions can be computationally expensive, particularly for large kernels and large datasets. Efficient algorithms aim to reduce the number of operations required.
2. **Lower memory usage:** Many convolution operations require storing intermediate results. Optimized algorithms aim to minimize memory footprint.
3. **Speed up inference:** In deep learning, faster convolutions mean faster model inference, which is important for real-time applications.
4. **Improve scalability:** Efficient algorithms ensure that convolutions can scale better as the input size increases.

both the input signal (or image) and the kernel (filter) into the frequency domain using FFT.

- **Convolution Theorem:** The convolution of two signals in the time (or spatial) domain is equivalent to the pointwise multiplication of their respective Fourier transforms:

$$y(t) = (x * h)(t) \quad \text{in time domain} \quad y(t) = (x * h)(t) \quad \text{in time domain}$$

is equivalent to

$$Y(f) = X(f) \cdot H(f) \quad \text{in frequency domain} \quad Y(f) = X(f) \cdot H(f) \quad \text{in frequency domain}$$

Where $X(f)$ and $H(f)$ are the Fourier Transforms of the input and filter, respectively, and $Y(f)$ is their product. After multiplication, we take the inverse Fourier Transform (IFFT) to get the result back in the spatial domain.

Why It's Efficient:

- In the frequency domain, convolution becomes multiplication, which is computationally faster. Using FFT reduces the computational complexity from $O(N^2)$ to $O(N \log N)$, making it much faster for large kernels.

Use Case:

- **Large convolutions** where the kernel is large, and direct spatial-domain convolutions would be too slow. For example, in deep learning when you're working with larger image inputs or larger kernels.

2. Depthwise Convolutions

Depthwise convolutions are a variant of convolutions where each filter operates on only one input channel at a time, rather than combining multiple input channels as in a standard convolution. This greatly reduces the number of computations.

How It Works:

- **Standard Convolution:** In a standard convolution, a single filter slides over the input and computes outputs by combining information across all input channels. This involves a full matrix of filters spanning all input channels.

- **Depthwise Convolution:** In depthwise convolution, each input channel has its own filter. For example, if you have 3 input channels and 3 output channels, instead of using a $3 \times 3 \times 3 \times 3 \times 3$ filter for each output channel, instead of separate $3 \times 3 \times 3$ filters, one for each input channel.
 - If the input has 3 channels and we use a $3 \times 3 \times 3$ filter, instead of performing $3 \times 3 = 9 \times 3 = 27$ operations per output channel, we would perform only 3 operations (one per input channel).

Why It's Efficient:

- **Reduces the number of parameters:** Since each channel is processed independently, the number of filters required is much smaller.
- **Reduces computations:** Depthwise convolution significantly cuts down on the number of operations, especially for deeper networks where the number of channels increases.

Use Case:

- **MobileNets** and other lightweight networks use depthwise convolutions to reduce computation and memory usage, especially for mobile or edge devices.

3. Separable Convolutions

Separable convolutions break down a 2D convolution into two smaller 1D convolutions — one in the horizontal direction and another in the vertical direction. This reduces computational cost while maintaining the effectiveness of the convolution.

How It Works:

- **Standard 2D Convolution:** A standard 2D convolution, such as a $5 \times 5 \times 5$ kernel, requires 25 multiplications for each output pixel (if using a single filter). For each output, you compute a weighted sum of the neighboring pixels.
- **Separable Convolution:** A $5 \times 5 \times 5$ filter can be decomposed into two 1D filters: one $5 \times 1 \times 1$ filter and one $1 \times 5 \times 1$ filter. First, the input image is convolved with the $5 \times 1 \times 1$ filter, and then the result is convolved with the $1 \times 5 \times 1$ filter. This reduces the number of multiplications from 25 to $5 + 5 = 10$, which is much more efficient.

This is possible because convolution is a linear operation, and thus separable filters still approximate the original 2D convolution well.

Why It's Efficient:

- **Reduces computational cost:** By breaking down 2D convolutions into two 1D convolutions, the number of operations is drastically reduced. For an $k \times k \times k$ kernel, instead of $k^2 k^2$ operations, you only need $k + k = 2k$ operations.

Use Case:

- **MobileNets** (which use depthwise separable convolutions) and other lightweight networks use separable convolutions to reduce computation without sacrificing performance.

4. Structured Convolutions

Structured convolutions refer to the use of convolution patterns or constraints that are designed to reduce computation, improve performance, or better fit the problem domain. These structures often focus on optimizing the way filters are applied or reducing redundant computations by enforcing certain patterns on the filter design or the way they are applied.

How It Works:

Structured convolutions can take many forms, such as:

- **Low-rank convolutions:** Using low-rank matrices to approximate convolutions. These matrices have fewer parameters, which reduces computation.
- **Group convolutions:** Partitioning the input channels into groups, where each group gets its own set of filters. This reduces the computational load by limiting the number of channels that interact with each filter.
- **Block or Toeplitz convolutions:** Some algorithms focus on using block-wise structures, where the filter or input is divided into smaller blocks, and only the non-zero parts are computed.
- **Factorized convolutions:** Factorizing the convolution operation into smaller convolutions, similar to separable convolutions, but with more complex factorizations (e.g., using two 3x3 convolutions instead of one 5x5).

Why It's Efficient:

- **Reduces redundancy:** By enforcing a structure, such as low-rank approximations, group convolutions, or others, redundant computations are avoided, and the number of parameters and operations is minimized.
- **Better generalization:** Structured convolutions can lead to models that generalize better because they reduce overfitting (by limiting the number of parameters or enforcing constraints on the filters).

Use Case:

- **ResNeXt, Inception-v4, and MobileNetV2** utilize structured convolutions (group convolutions or factorized convolutions) to improve both efficiency and model performance. **EfficientNet** uses this idea of "compound scaling" to balance depth, width, and resolution of the model while applying structured operations like depthwise separable convolutions.

Summary of Each Approach:

1. **Fourier Transform Convolution:**
 - Transforms convolution into frequency domain multiplication (more efficient for large convolutions).
2. **Depthwise Convolution:**

- Applies a separate filter for each input channel (reduces parameter count and computation).
- 3. **Separable Convolution:**
 - Decomposes 2D convolutions into two 1D convolutions (reduces computation significantly).
- 4. **Structured Convolutions:**
 - Utilizes specific constraints or patterns (low-rank, group convolutions, etc.) to reduce computation, enhance performance, or adapt to problem-specific needs.

Each of these methods targets different efficiency goals, and often in practical implementations, a combination of them is used to achieve a balance between computational cost and model performance. Let me know if you'd like more details or practical examples on any of them!

Fourier Transform Approach problem.

Formula : $x * k \leftrightarrow F^{-1}(F(x) \cdot F(k))$

Where

x = input

k = kernel

F = Fourier Transform, F^{-1} = Inverse Fourier Transform

$*$ = Convolution

$$F = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4 + x_5 y_5 + x_6 y_6$$

We have to consider 5x5 matrix perform a 2D convolution of the given input matrix I with the kernel K . show step-by-step calculations for at least two positions of the kernel and then write the final output matrix.

The Input matrix is $I = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 2 \\ 7 & 8 & 9 & 0 & 1 \\ 1 & 3 & 5 & 2 & 4 \\ 2 & 4 & 6 & 3 & 5 \end{bmatrix}$ Kernel (K) = $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Solution : pointwise calculation

Given Input $I = \begin{bmatrix} 1 & 2 & 3 & 0 & 2 \\ 4 & 5 & 6 & 0 & 1 \\ 7 & 8 & 9 & 0 & 2 \\ 2 & 3 & 4 & 0 & 4 \\ 2 & 4 & 6 & 3 & 5 \end{bmatrix}$ $K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Step 1 : Elementwise product with K , output position (1,1), Top-left using input Rows 1-3, column 1-3.

patch : $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ Kernel (K) = $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Formula = $x * K = F^{-1}(F(x) \cdot F(y))$

$\begin{bmatrix} x_1 y_1 & x_2 y_2 & x_3 y_3 \\ x_4 y_4 & x_5 y_5 & x_6 y_6 \\ x_7 y_7 & x_8 y_8 & x_9 y_9 \end{bmatrix}$ $K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

(1,1)

$$F = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4 + x_5 y_5 + x_6 y_6 + x_7 y_7 + x_8 y_8 + x_9 y_9$$
$$= 1 \times 1 + 2 \times 0 + 3 \times (-1) + 4 \times 1 + 5 \times 0 + 6 \times (-1) + 7 \times 1 + 8 \times 0 + 9 \times (-1)$$

$$\text{Sum}(F) = 1 + 0 - 3 + 4 + 0 - 6 + 7 + 0 - 9$$
$$= -6$$

output (x_1) (1,1) = -6

: output position (1,2), input Rows (2-4), column (1-3)

KRM

$$(1,2) = \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 0 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 2 \times 1 & 3 \times 0 & 0 \times -1 \\ 5 \times 1 & 6 \times 0 & 0 \times -1 \\ 8 \times 1 & 9 \times 0 & 0 \times -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 5 & 0 & 0 \\ 8 & 0 & 0 \end{bmatrix}$$

$$\text{Sum} = 2 + 0 + 0 + 5 + 0 + 0 + 8 + 0 + 0 = 15$$

$$\text{output (1,2)} = 15$$

Step 3 : output position (1,3), input Rows (3,5), column (1,3)

$$(1,3) = \begin{bmatrix} 3 & 0 & 2 \\ 6 & 0 & 1 \\ 9 & 0 & 2 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 3 \times 1 & 0 \times 0 & 2 \times -1 \\ 6 \times 1 & 0 \times 0 & 1 \times -1 \\ 9 \times 1 & 0 \times 0 & 2 \times -1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & -2 \\ 6 & 0 & -1 \\ 9 & 0 & -2 \end{bmatrix}$$

$$\text{Sum} = 3 + 0 + 2 + 6 + 0 + 1 + 9 + 0 + 2 = 13$$

$$\boxed{\text{output (1,3)} = 13}$$

Step 4 : output position (2,1), input Rows (2,4), columns (1,3)

$$\text{patch} \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 2 & 3 & 4 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 4 \times 1 & 5 \times 0 & 6 \times -1 \\ 7 \times 1 & 8 \times 0 & 9 \times -1 \\ 2 \times 1 & 3 \times 0 & 4 \times -1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & -6 \\ 7 & 0 & -9 \\ 2 & 0 & -4 \end{bmatrix}$$

$$\text{Sum} = 4 + 0 + 6 + 7 + 0 + 9 + 2 + 0 + 4 = 32$$

$$\boxed{\text{output (2,1)} = 32}$$

Step 5: output position (2,2), Input Rows (2-4), columns (2,4)

$$\text{patch} \begin{bmatrix} 5 & 6 & 0 \\ 8 & 9 & 0 \\ 3 & 4 & 0 \end{bmatrix} \quad \text{kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 5 \times 1 & 6 \times 0 & 0 \times -1 \\ 8 \times 1 & 9 \times 0 & 0 \times -1 \\ 3 \times 1 & 4 \times 0 & 0 \times -1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 \\ 8 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

$$\text{Sum} = 5 + 0 + 0 + 8 + 0 + 0 + 3 + 0 + 0 = 16$$

$$\boxed{\text{output}(2,2) = 16}$$

Step 6: output position (2,3), Input Rows (2,4), col (3,5)

$$\text{patch} \begin{bmatrix} 6 & 0 & 1 \\ 9 & 0 & 2 \\ 4 & 0 & 4 \end{bmatrix} \quad \text{kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{product} \begin{bmatrix} 6 \times 1 & 0 \times 0 & 1 \times -1 \\ 9 \times 1 & 0 \times 0 & 2 \times -1 \\ 4 \times 1 & 0 \times 0 & 4 \times -1 \end{bmatrix} = \begin{bmatrix} 6 & 0 & -1 \\ 9 & 0 & -2 \\ 4 & 0 & -4 \end{bmatrix}$$

$$\text{Sum} = 6 + 0 - 1 + 9 + 0 - 2 + 4 + 0 - 4 = 12$$

$$\boxed{\text{output}(2,3) = 12}$$

Step 7: output position (3,1), Input Rows (3,5), columns (1,3)

$$\boxed{\text{output}(3,1) = -8}$$

Step 8: output position (3,2), Input Rows (3,5) column (2,4)

$$\boxed{\text{output}(3,2) = 12}$$

Step 9: output position (3,3), Input (3,5) colu (3,5)

$$\boxed{\text{output}(3,3) = 8}$$

Final output matrix (3x3)

$$\begin{bmatrix} -6 & 15 & 13 \\ -6 & 16 & 12 \\ -8 & 12 & 8 \end{bmatrix}$$

Random (or) Unsupervised Features

In Convolutional Neural Networks, the convolutional filters are usually learned from data through backpropagation. But in some cases, instead of training filters, we use random filters (fixed random weights). These features are then passed through the network to the classifier.

- * Initialization - Filters are randomly Generated

- * Fixed - They are not updated during Training.

- * Features Extraction - Input Images are convolved with these

- * classifier - A simple model is trained on top of these feature maps.

Why use Random features :

- * Fast Training → No Backpropagation for convolutional layers

- * Low cost → Avoids expensive gradient updates.

- * Surprisingly effective →

Random CNN features often work well for texture recognition and image classification.

Methods :

- ① Extreme learning Machines (ELM + CNN) →

Random convolution filters, only classifier trained

- ② Random Convolutional Kernel Transform →

extracts texture features.

- ③ Scattering Networks → use fixed wavelet/random filters as feature extractors.

Random Features in Convolutional Neural Networks (CNNs) refer to the use of convolutional filters whose weights are **randomly initialized and kept fixed**, rather than being learned through training. The CNN acts as a **random feature extractor**, generating nonlinear transformations of the input data, while only the final classifier (such as a linear layer or SVM) is trained on these extracted features.

◆ 1. What Are Random Features?

In **convolutional neural networks (CNNs)**, the convolutional filters are usually **learned from data** through backpropagation.

But in some cases, instead of training filters, we use **random filters** (fixed random weights). These features are then passed through the network to the classifier.

◆ 2. How Random Features Work in CNNs

- **Initialization:** Filters (kernels) are randomly generated.
- **Fixed:** They are **not updated** during training.
- **Feature Extraction:** Input images are convolved with these random filters, producing **random feature maps**.
- **Classifier:** A simple model (like logistic regression, SVM, or softmax layer) is trained on top of these feature maps.

◆ 3. Why Use Random Features?

- **Fast training** → no backpropagation for convolutional layers.
- **Low cost** → avoids expensive gradient updates.
- **Surprisingly effective** → Random CNN features often work well for texture recognition and image classification (sometimes competitive with learned features when enough filters are used).

◆ 4. Example Methods

- **Extreme Learning Machines (ELM) + CNN** → random convolution filters, only classifier trained.
- **Random Convolutional Kernel Transform (RCKT)** → extracts texture features.
- **Scattering Networks** → use fixed wavelet/random filters as feature extractors.

5. Comparison: Learned vs Random Features in CNN

Aspect	Learned CNN Features	Random CNN Features
Filters	Optimized via backprop	Randomly initialized & fixed
Training Cost	High	Very low
Adaptability	Adapts to dataset	Generic, not dataset-specific
Performance	Usually higher	Lower, but sometimes competitive
Example Use	Deep learning models	Fast baselines, texture classification

Unsupervised features in convolutional networks

Unsupervised features in **convolutional networks (ConvNets / CNNs)** refer to features that are **learned automatically from data without using labeled outputs**. Instead of training with explicit class labels (supervised), the network discovers useful patterns or representations by analyzing the raw input structure.

Here's a structured explanation:

◆ What are Unsupervised Features?

- Features learned without ground-truth labels.
- CNN learns filters (convolutions) that capture **edges, textures, shapes, or abstract patterns** directly from unlabeled data.
- Goal: find representations that make later tasks (like classification, clustering, anomaly detection) easier.

◆ How CNNs Learn Unsupervised Features

1. Autoencoders

- CNNs can be used in convolutional autoencoders.
- Encoder part learns convolutional filters that reduce input data into compressed representations (unsupervised features).
- Decoder reconstructs the input → training objective is reconstruction loss (not labels).

2. Restricted Boltzmann Machines / Deep Belief Nets

- Early CNN-based unsupervised models stacked convolutional RBMs to discover hierarchical features.

3. Clustering-based Feature Learning

- CNN features (like intermediate layer outputs) can be clustered (e.g., k-means) to form unsupervised feature maps.

4. Self-Supervised Learning (modern approach)

- CNNs learn features by solving **pretext tasks** created from unlabeled data:
 - Predict missing patches of an image (context prediction).
 - Contrastive learning (e.g., SimCLR, MoCo): CNN learns to bring similar images closer and push different ones apart.
 - Rotation prediction: CNN learns to identify the rotation applied to an image (0° , 90° , 180° , 270°).
- These learned unsupervised features can later be fine-tuned with labels.

◆ Examples of Unsupervised Features in CNNs

- **Low-level:** edges, corners, color blobs.
- **Mid-level:** textures, motifs, repeated shapes.
- **High-level:** object parts, abstract semantic patterns.

◆ Advantages

- Works with unlabeled data (huge amount available).
- Improves **generalization** when fine-tuned with few labeled samples.
- Enables transfer learning (pretrained CNNs on unsupervised tasks → used for downstream tasks).

Random (or) Unsupervised Features

In Convolutional Neural Networks, the convolutional filters are usually learned from data through backpropagation. But in some cases, instead of training filters, we use random filters (fixed random weights). These features are then passed through the network to the classifier.

- * Initialization - Filters are randomly generated

- * Fixed - They are not updated during training.

- * Features Extraction - Input Images are convolved with these Random filters, producing random feature maps

- * Classifier - A simple model is trained on top of these feature maps.

Why use Random features :

- * Fast Training $\rightarrow$ No backpropagation for convolutional layers

- * Low cost $\rightarrow$ Avoids expensive gradient updates.

- * Surprisingly effective $\rightarrow$

Random CNN features often work well for texture recognition and image classification.

Methods :

- ① Extreme Learning Machines (ELM + CNN) $\rightarrow$

Random convolution filters, only classifier trained

- ② Random Convolutional Kernel Transform $\rightarrow$
Extracts texture features.

- ③ Scattering Networks $\rightarrow$ use fixed wavelet/random filters as feature extractors.

The Neuro-Scientific Basis for Convolutional Networks :

CNNs are inspired by how the human brain processes visual information, especially the visual cortex. Neuroscience research by Hubel and Wiesel (1960's) on cats and monkeys laid the ~~base~~ foundation

1. Visual Cortex and Feature Detection

2. Receptive Fields

3. Hierarchical processing

4. Weight Sharing

5. Pooling and Invariance

6. Neurobiological Inspiration

1. Visual cortex and Feature Detection :

In the brain's primary visual cortex (V_1), neurons respond to specific patterns. Some neurons fire only for edges in a certain orientation. (Vertical, Horizontal, diagonal)

2. Respective Fields :

A receptive field is the small area of the visual field that a neuron responds to. Early layers in CNNs have small filters [like 3×3 , 5×5] which focus on local patterns.

3. Hierarchical processing :

Human vision works hierarchically

a. Retina $\rightarrow$ detects light intensity

b. $V_1 \rightarrow$ detects edges & orientations

c. Higher layers (V_2, V_4, IT) $\rightarrow$ detect shapes, faces, objects

4. Weight sharing :

In biology, neurons use shared connections to detect the same feature across different locations.

CNNs do the same with convolutional filters - the same filter slides (convolves) across the whole image.

5. Pooling and Invariance :

Visual perception is robust we can recognize an object even if it's shifted, rotated, scaled.

In pooling layers (max/average pooling) reduce sensitivity to exact position, similar to how the brain develops spatial invariance.

6. Neurobiological Inspiration :

Hubel & Wiesel's experiments : Discovered simple cells and complex cells. Simple cells $\approx$ convolutional filters in early CNN layers, and complex cells $\approx$ pooling layers in CNNs.

Unit-5

Sequence Modelling

TOPICS :

1 : Unfolding Computational Graphs

2 : Recurrent Neural Networks (RNNs)

3 : Bidirectional RNNs

4 : Encoder-Decoder Sequence-to-Sequence Architectures

5 : Deep Recurrent Networks and Recursive Neural Networks

6 : Long Short-Term Memory (LSTM)

Sequence Modelling

Definition of Sequence Modelling in Deep Learning

Sequence Modelling in deep learning refers to the process of designing and training models that can understand, predict, or generate data where the order of elements matters.

In other words, it is the task of **learning relationships and dependencies between elements in a sequence**, such as words in a sentence, frames in a video, or values in a time series.

Mathematically, a sequence model learns to map:

$$\left[\begin{array}{l} (x_1, x_2, \dots, x_T) \end{array} \right] \rightarrow (y_1, y_2, \dots, y_T)$$

where each output (y_t) may depend not only on the corresponding input (x_t) but also on **previous elements** in the sequence.

Main Concepts in Sequence Modelling :

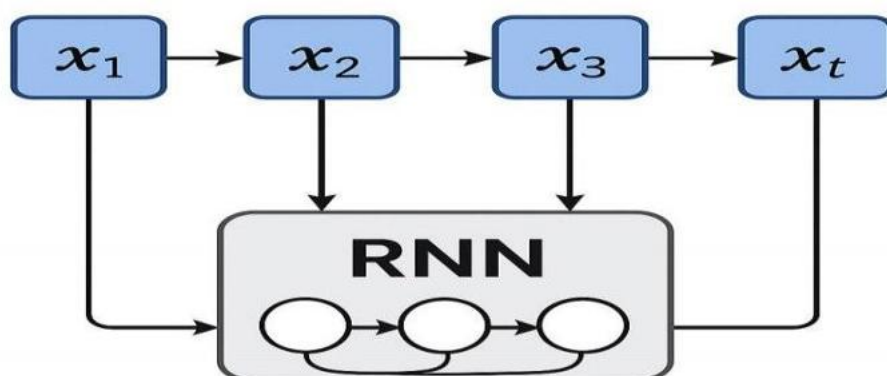
1 : Sequential Data

2 : Tasks in Sequential Modelling

3 : Challenges

1 : Sequential Data in Deep Learning

SEQUENTIAL DATA



Sequential Data refers to data where the order of elements **matters** — each element depends on or is related to the previous ones. In other words, **the sequence or arrangement of data points carries meaningful information**.

Unlike regular datasets (like images or tabular data), where samples are **independent**, sequential data has **temporal or positional dependency**.

This means the **current input depends on past inputs** — and sometimes future inputs too (in bidirectional models).

◆ Examples of Sequential Data:

Domain	Example	Sequence Type
Text	Words in a sentence	Word or character sequence
Speech	Audio waveform	Time-series signal
Finance	Stock prices	Time-series data
IoT / Sensors	Temperature readings	Time-based sequence
Video	Frames over time	Spatial + temporal sequence

◆ Why Order Matters:

In sequential data:

- The **meaning or prediction** often depends on previous steps.
Example:
 - “I am going to the **bank**” → means financial institution.
 - “I am standing near the **bank** of the river” → means river side.
The word “bank” changes meaning based on previous words — **sequence context**.

◆ Mathematical Representation:

A sequence can be represented as:

[
 $X = (x_1, x_2, x_3, \dots, x_T)$
]

where:

- (x_t) = data at time step (t)
- (T) = total number of time steps
- Each (x_t) can be a vector, image, word, or sound frame.

Challenges in Sequential Data:

1. **Variable Lengths:** Sequences may have different lengths (e.g., short vs long sentences).
2. **Long-Term Dependencies:** Some information from far earlier may still be important.
3. **Context Preservation:** Need to remember previous inputs effectively.

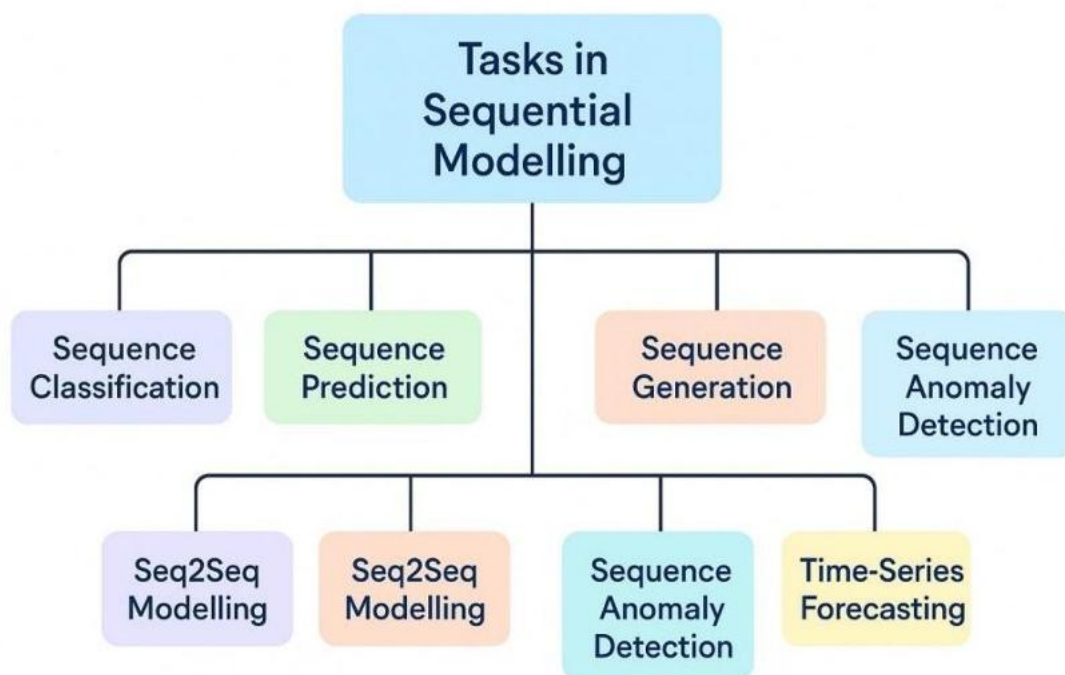
Deep Learning Models for Sequential Data:

Model	Description
Recurrent Neural Networks (RNNs)	Basic architecture for handling sequential dependencies.
LSTM (Long Short-Term Memory)	Solves vanishing gradient problem; remembers long-term dependencies.
GRU (Gated Recurrent Unit)	Simplified LSTM with fewer parameters.
Bidirectional RNNs	Reads sequences from both directions.
Transformers	Uses attention mechanism; captures dependencies without recurrence.

Applications in Real World:

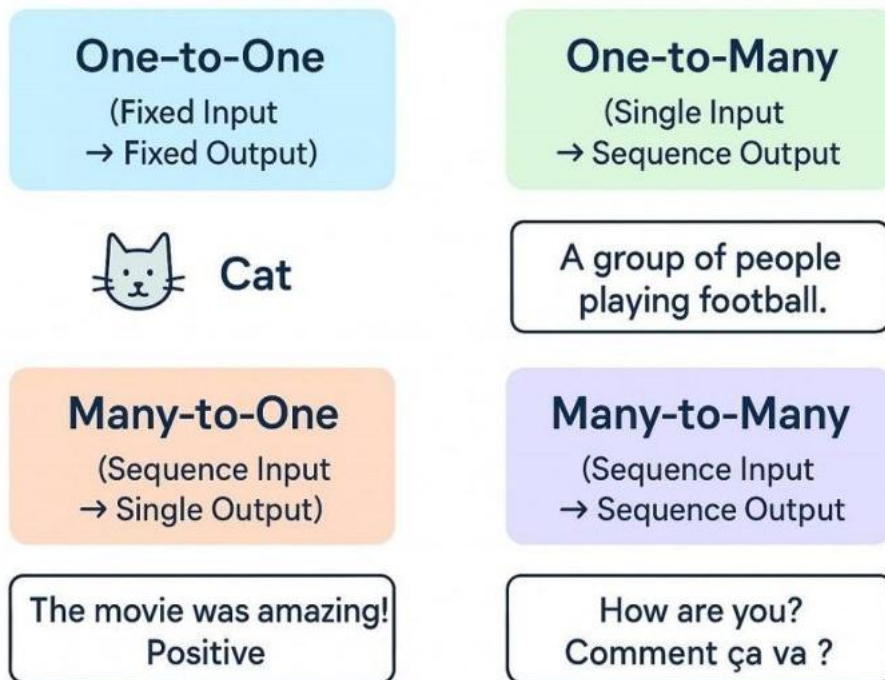
- **Speech Recognition** (e.g., Siri, Alexa)
- **Machine Translation** (English → French)
- **Text Generation** (ChatGPT, GPT models)
- **Stock Price Forecasting**
- **Music Generation**
- **Video Captioning**

Feature	Sequential Data	Non-Sequential Data
Order Importance	High	Low
Examples	Text, speech, time-series	Images, tabular data
Model Type	RNN, LSTM, GRU, Transformer	CNN, MLP
Dependency	Between data points	Independent samples



2 : Tasks in Sequential Modelling

Sequential Modelling tasks categorized by **input-output relationships** such as **one-to-one**, **one-to-many**, **many-to-one**, and **many-to-many**.



1. One-to-One (Fixed Input → Fixed Output)

- **Definition:** Each input corresponds to one output.
- **Use Case:** Traditional feed-forward neural networks, where data is not sequential.
- **Example Tasks:**
 - Image classification (Input: single image → Output: single label)
 - Simple regression (Input: one value → Output: one value)

Example:

Input: [Image of a cat] → Output: "Cat"

2. One-to-Many (Single Input → Sequence Output)

- **Definition:** A single input produces a sequence of outputs.
- **Goal:** Generate sequential data from one input.
- **Example Tasks:**
 - Image captioning (Input: image → Output: sentence)
 - Music generation from a single note

Example:

Input: [Photo] → Output: "A group of people playing football."

Model Used:

Encoder–Decoder models (e.g., CNN + LSTM)

3. Many-to-One (Sequence Input → Single Output)

- **Definition:** A sequence of inputs results in one final output.
- **Goal:** Summarize or classify the entire sequence.
- **Example Tasks:**
 - Sentiment analysis (Input: sentence → Output: positive/negative)
 - Speech emotion recognition
 - Stock market trend prediction (sequence → up/down)

Example:

Input: "The movie was amazing!" → Output: Positive sentiment

Model Used:

LSTM, GRU, Bidirectional LSTM

4. Many-to-Many (Sequence Input → Sequence Output)

- **Definition:** A sequence of inputs produces a sequence of outputs.
- **Goal:** Transform one sequence into another, often of different length.
- **Example Tasks:**
 - Machine translation (English → French)
 - Video frame labeling
 - Speech-to-text conversion

Example:

Input: "How are you?" → Output: "Comment ça va ?"

Model Used:

Seq2Seq with Encoder–Decoder or Transformer models

Special Case: Many-to-Many (Synchronous and Asynchronous)

Type	Description	Example
Synchronous	Input and output sequences have the same length	POS tagging, video frame labeling
Asynchronous	Input and output sequences have different lengths	Machine translation, summarization

Summary Table:

Type	Input	Output	Example Application	Common Model
One-to-One	Single	Single	Image classification	CNN, DNN
One-to-Many	Single	Sequence	Image captioning	CNN + LSTM
Many-to-One	Sequence	Single	Sentiment analysis	LSTM, GRU
Many-to-Many	Sequence	Sequence	Machine translation, speech-to-text	Seq2Seq, Transformer

3 : Challenges

- 1 : Capturing long term dependencies
- 2 : Dealing with variable – length sequences
- 3 : Handling vanishing /exploding gradients
- 4 : Maintaining contextual understanding

Recurrent Neural Networks (RNNs) :

A **Recurrent Neural Network (RNN)** is a type of neural network in which the **output from previous steps is fed as input to the current step**, allowing the network to have a **memory** of previous inputs.

This makes RNNs suitable for tasks where **the order of the data matters**.

⚙ Mathematical Representation:

For each time step t :

$$h_t = f(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

$$y_t = W_y \cdot h_t + c$$

Where:

- h_t : hidden state at time t
- x_t : input at time t
- W_h, W_x, W_y : weight matrices
- b, c : biases
- f : activation function (like tanh or ReLU)

Types of RNNs:

1. **Vanilla RNN**: Basic form (simple recurrent connections)
2. **LSTM (Long Short-Term Memory)**: Handles long-term dependencies using gates
3. **GRU (Gated Recurrent Unit)**: Simplified version of LSTM with fewer parameters
4. **Bidirectional RNN**: Processes data in both forward and backward directions
5. **Deep RNN**: Stacks multiple RNN layers for greater learning capacity

Applications of RNNs:

- **Natural Language Processing (NLP)**:
 - Language modeling
 - Text generation
 - Machine translation

- **Speech Recognition**
- **Music generation**
- **Time-series prediction (e.g., stock prices)**
- **Video analysis**

Limitations:

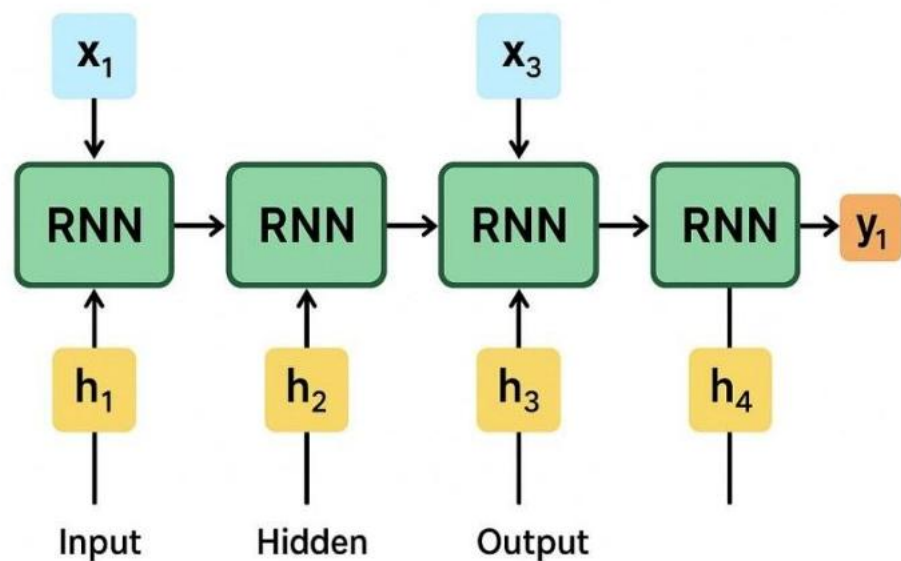
- **Vanishing & Exploding Gradients:** Difficulty learning long-term dependencies.
- **Slow training** due to sequential nature.
- **Limited memory** of earlier inputs.

(These issues are addressed in LSTM and GRU models.)

Advantages:

- Can process sequences of **variable length**.
- Maintains **context** from previous inputs.
- Works well for **temporal or sequential tasks**.

Recurrent Neural Network (RNN)



Problem:

Consider a simple RNN defined by the following equations:

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

$$y_t = W_y \cdot h_t$$

where:

- $W_h = 0.5$
- $W_x = 1.0$
- $W_y = 2.0$
- $b = 0.1$
- Initial hidden state $h_0 = 0$
- Input sequence $x = [1, 2, 3]$

Compute:

1. Hidden states h_1, h_2, h_3
2. Outputs y_1, y_2, y_3

Solution Step-by-Step:

At $t = 1$:

$$h_1 = \tanh(0.5 \cdot h_0 + 1.0 \cdot x_1 + 0.1)$$

$$h_1 = \tanh(0.5 \cdot 0 + 1.0 \cdot 1 + 0.1) = \tanh(1.1) = 0.800$$

$$y_1 = 2.0 \cdot h_1 = 2.0 \cdot 0.800 = 1.6$$

At $t = 2$:

$$h_2 = \tanh(0.5 \cdot h_1 + 1.0 \cdot x_2 + 0.1)$$

$$h_2 = \tanh(0.5 \cdot 0.8 + 2 + 0.1) = \tanh(2.5) = 0.9866$$

$$y_2 = 2.0 \cdot h_2 = 2.0 \cdot 0.9866 = 1.973$$

At $t = 3$:

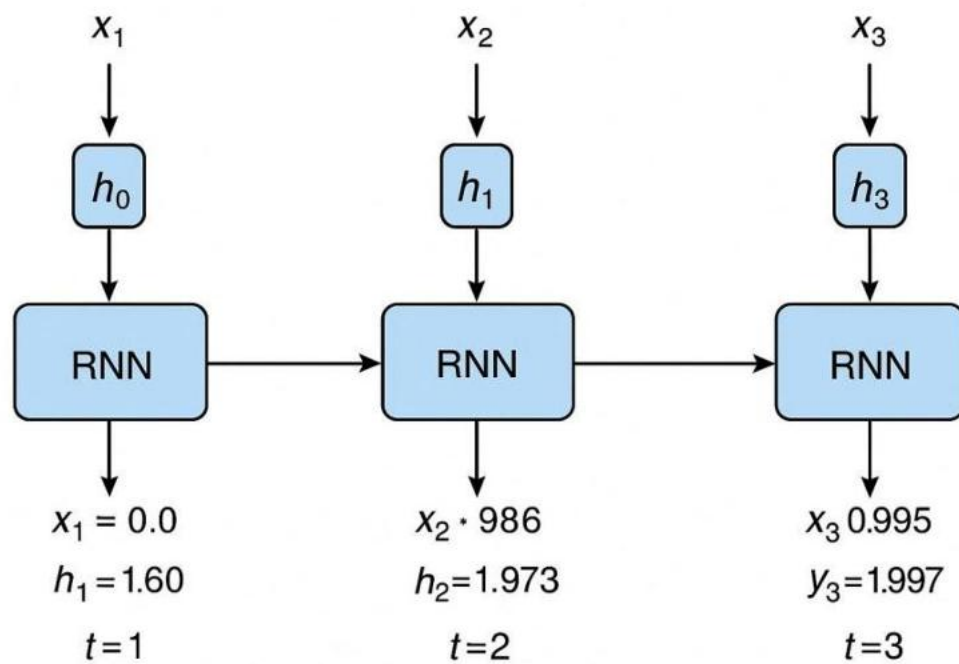
$$h_3 = \tanh(0.5 \cdot h_2 + 1.0 \cdot x_3 + 0.1)$$

$$h_3 = \tanh(0.5 \cdot 0.9866 + 3 + 0.1) = \tanh(3.5933) = 0.9985$$

$$y_3 = 2.0 \cdot h_3 = 2.0 \cdot 0.9985 = 1.997$$

✅ Final Answers:

Time Step (t)	Input (x_t)	Hidden State (h_t)	Output (y_t)
1	1	0.800	1.600
2	2	0.9866	1.973
3	3	0.9985	1.997



Recursive Neural Networks (RecNNs) :

1. Introduction

Recursive Neural Networks (RecNNs) are a class of deep learning architectures designed to process hierarchical or tree-structured data.

Unlike Recurrent Neural Networks (RNNs) that model sequential data (time steps or ordered sequences), RecNNs are used where the structure is nested or compositional, such as natural language parse trees, scene hierarchies, or symbolic expressions.

They are particularly powerful for sequence modelling tasks that involve syntactic or hierarchical relationships (e.g., sentiment analysis using sentence structures).

In Recursive Neural Networks, the same set of weights is applied recursively over a structure, combining smaller parts into larger representations until a complete structure is represented.

Each node in the tree (or structure) computes its hidden representation by recursively combining the representations of its child nodes.

3. Mathematical Formulation

♦ 3. Mathematical Formulation

Consider two child nodes c_1 and c_2 with hidden representations h_{c1} and h_{c2} .

The parent node p computes its representation as:

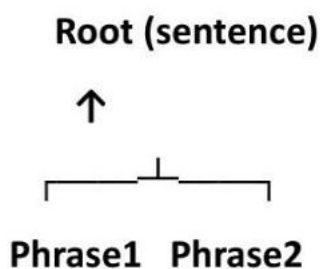
$$h_p = f(W[h_{c1}; h_{c2}] + b)$$

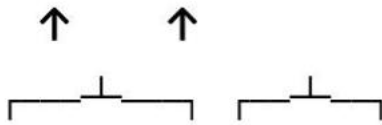
Where:

- $[h_{c1}; h_{c2}] \rightarrow$ concatenation of child vectors
- $W \rightarrow$ shared weight matrix
- $b \rightarrow$ bias term
- $f \rightarrow$ non-linear activation function (e.g., tanh, ReLU)

This process is **repeated recursively** until reaching the root node, producing a **global representation** of the structure.

4. Architecture Diagram





Word1 Word2 Word3 Word4

5. Comparison: Recursive vs Recurrent Networks

Feature	Recursive Neural Network (RecNN)	Recurrent Neural Network (RNN)
Structure	Tree / Hierarchical	Sequential / Temporal
Input Type	Parse trees, expressions, hierarchies	Time series, text, speech
Weight Sharing	Across tree nodes	Across time steps
Example Task	Sentiment analysis using syntax	Language modelling, translation
Representation	Compositional	Sequential

◆ 6. Example Problem

Example: Sentiment Analysis

Sentence: "Not very good"

1. Represent each word with an embedding vector:

$$h_{\text{not}}, h_{\text{very}}, h_{\text{good}}$$

2. Combine child nodes recursively:

$$h_1 = f(W[h_{\text{very}}; h_{\text{good}}] + b)$$

$$h_{\text{root}} = f(W[h_{\text{not}}; h_1] + b)$$

3. The final vector h_{root} represents the overall sentence meaning (sentiment).

7. Types of Recursive Neural Networks

1. Standard RecNN:
Simple recursive composition using fully connected layers.
2. Tree-LSTM (Tree-structured LSTM):
Extends LSTM to tree structures for better gradient flow.
3. Tree-GRU:
Similar to Tree-LSTM but uses gated recurrent units.
4. Graph Recursive Networks:
Generalization for arbitrary graphs (not only trees).

8. Applications in Sequence Modelling

1. Natural Language Processing:

- Sentiment analysis (based on parse trees)
- Syntax-based machine translation
- Semantic composition (phrase and sentence meaning)

2. Computer Vision:

- **Scene parsing (hierarchical scene decomposition)**

3. Knowledge Representation:

- **Logic expressions and symbolic computation**

4. Program Analysis:

- **Understanding code syntax trees (ASTs)**

9. Advantages

- 1 : Captures hierarchical relationships (syntax/semantics).
- 2 : Handles compositionality in structured data.
- 3 : Can generalize to non-sequential inputs (trees, graphs).
- 4 : Reuses parameters efficiently (shared weights).

10. Limitations

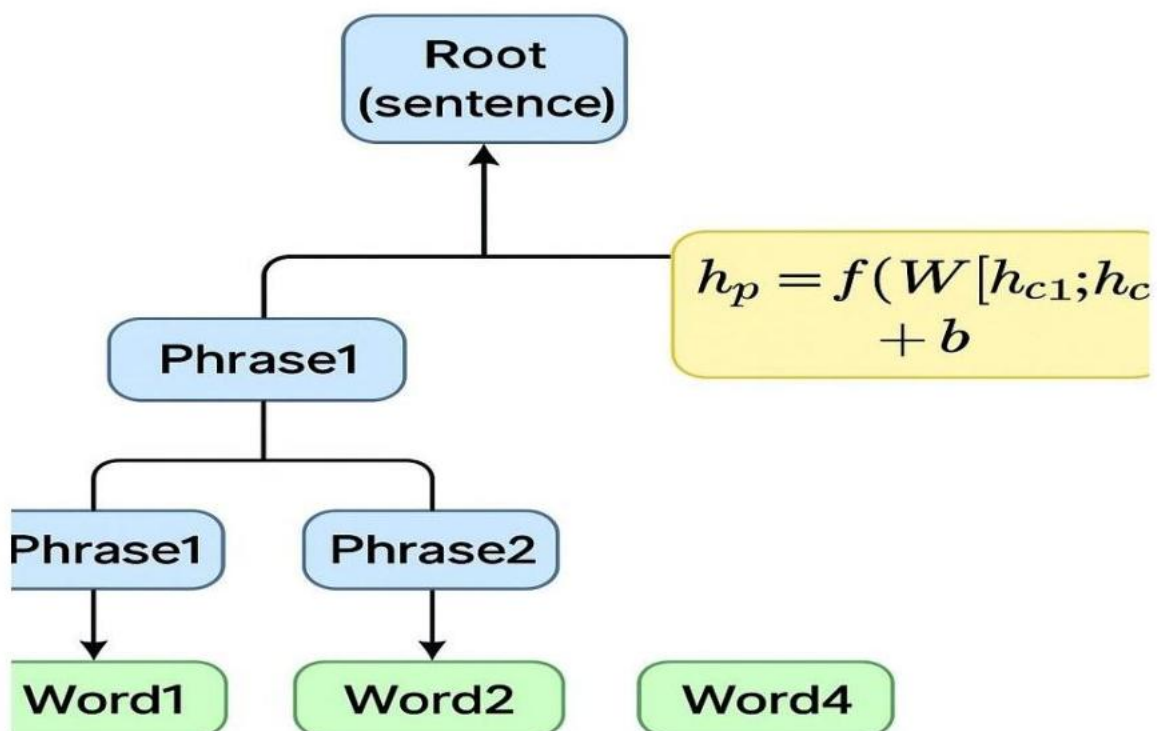
- 1 : Requires structured input (e.g., parse trees).
- 2: Parsing errors can degrade performance.
- 3: Harder to parallelize compared to RNNs.
- 4 : Complex training process (due to non-linear recursion).**

11. Real-Time Applications

- Google's Sentiment Treebank Model for sentiment classification
- Scene Graph Generation in computer vision
- Syntax-aware neural translation
- Code2Vec model (learning code embeddings using AST RecNNs)

12. Summary Table

Aspect	Recursive Neural Network
Data Type	Tree-structured (hierarchical)
Core Equation	$(h_p = f(W[h_{c1}; h_{c2}] + b))$
Typical Use	Compositional semantics
Advantage	Captures structure beyond sequence
Example Model	Tree-LSTM, Tree-GRU
Limitation	Needs structured data (e.g., parse tree)



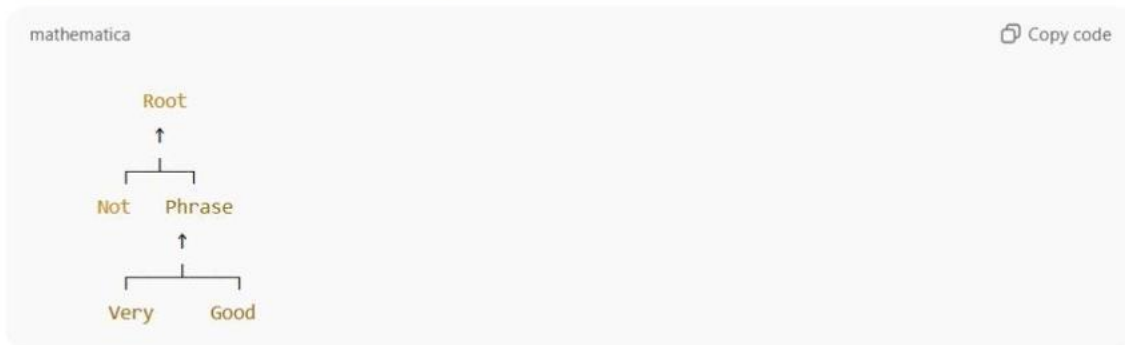
Example Problem — Sentiment Classification using Recursive Neural Network

Problem Statement

Predict the **sentiment** (positive or negative) of the sentence:

“Not very good”

Step 1: Sentence Structure (Parse Tree)



This shows a **hierarchical (tree)** structure where “very” and “good” combine first, and then “not” modifies their meaning.

Step 2: Word Embeddings

Each word is converted into a **vector** (from embedding layer):

Word	Embedding vector (dimension = 3)
not	[0.2, 0.1, -0.3]
very	[0.3, 0.5, 0.7]
good	[0.6, 0.9, 0.8]

Step 3: Recursive Composition Function

The **parent representation** is computed recursively as:

$$h_p = f(W[h_{c1}; h_{c2}] + b)$$

Where:

- h_{c1}, h_{c2} : child node vectors
- $[h_{c1}; h_{c2}]$: concatenation of both vectors
- W : weight matrix
- b : bias vector
- f : activation (e.g., tanh)

Step 4: Compute Intermediate Node (Phrase = "very good")

Let's assume:

$$W = \begin{bmatrix} 0.5 & 0.3 & 0.2 & 0.1 & 0.4 & 0.6 \\ 0.2 & 0.7 & 0.5 & 0.3 & 0.2 & 0.8 \\ 0.4 & 0.6 & 0.3 & 0.2 & 0.5 & 0.7 \end{bmatrix}, \quad b = [0.1, 0.1, 0.1]$$

Concatenate "very" and "good" vectors:

$$[h_{\text{very}}; h_{\text{good}}] = [0.3, 0.5, 0.7, 0.6, 0.9, 0.8]$$

Compute:

$$h_{\text{phrase}} = \tanh(W[h_{\text{very}}; h_{\text{good}}]^T + b)$$

After matrix multiplication and tanh activation:

$$h_{\text{phrase}} = [0.82, 0.76, 0.79]$$

Step 5: Compute Root Node ("not very good")

Combine "not" and the phrase vector:

$$[h_{\text{not}}; h_{\text{phrase}}] = [0.2, 0.1, -0.3, 0.82, 0.76, 0.79]$$

Use the same weights W and bias b :

$$h_{\text{root}} = \tanh(W[h_{\text{not}}; h_{\text{phrase}}]^T + b)$$

Result:

$$h_{\text{root}} = [0.34, -0.42, -0.37]$$

Step 6: Sentiment Prediction Layer

Finally, apply a **softmax classifier** on the root representation:

$$\text{Sentiment} = \text{softmax}(Uh_{\text{root}} + c)$$

If:

$$U = \begin{bmatrix} 0.6 & 0.3 & 0.4 \\ -0.5 & -0.2 & -0.3 \end{bmatrix}, \quad c = [0.1, 0.1]$$

Then the output could be:

$$P(\text{positive}) = 0.28, \quad P(\text{negative}) = 0.72$$

✅ Predicted sentiment: Negative

Step 7: Interpretation

- "very good" alone is **positive**
- The word "not" **reverses** the meaning — RecNN captures this through hierarchical structure
- Hence, the model outputs **negative sentiment**

Step	Process	Output
1	Parse the sentence	Tree structure
2	Embed words	Numeric vectors
3	Combine recursively	Parent nodes via weights
4	Obtain root vector	Represents full sentence
5	Predict label	Softmax output

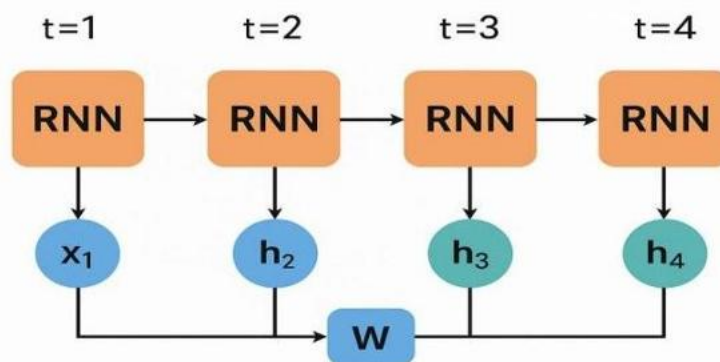
1. Unfolding Computational Graphs in Deep Learning

Definition:

Unfolding a computational graph refers to the process of expanding a **recurrent or recursive model** (like RNNs) over time into a sequence of repeated layers that represent the flow of computations across time steps.

It helps visualize how data moves through time and how parameters (weights, biases) are shared across different time steps.

Unfolding Computational Graphs



Concept Overview:

- In **feedforward networks**, computations flow in one direction — from input to output — with **no recurrence**.
- In **recurrent neural networks (RNNs)**, the output of one step becomes the input for the next step.
To analyze or train such networks, we *unfold* the recurrence over time steps to form a **static computation graph**.

Mathematical Formulation:

Consider a simple RNN:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

$$y_t = g(W_y h_t + c)$$

- x_t : Input at time t
- h_t : Hidden state at time t
- y_t : Output at time t
- W_x, W_h, W_y : Weight matrices (shared across time)
- f, g : Activation functions (e.g., tanh, softmax)

When unfolded for T time steps, the computational graph becomes a chain of T feedforward layers — one per time step — with **shared weights**.

Purpose of Unfolding:

1. Training:

Backpropagation through time (**BPTT**) requires the unfolded graph to compute gradients.

2. Visualization:

Makes the temporal dependencies clear.

3. Implementation:

Deep learning frameworks (like TensorFlow, PyTorch) internally unroll these graphs for gradient calculation.

4. Gradient Computation:

Allows error to propagate backward through all time steps.

Example Problem:

Suppose we have a simple RNN with:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t)$$

and initial $h_0 = 0$.

If we have inputs x_1, x_2, x_3 , the unfolded computation is:

$$h_1 = \tanh(W_x x_1)$$

$$h_2 = \tanh(W_h h_1 + W_x x_2)$$

$$h_3 = \tanh(W_h h_2 + W_x x_3)$$

This chain can now be treated like a **3-layer feedforward network** with shared weights.

Advantages:

- Clarifies **temporal relationships** in sequential data.
- Enables **gradient-based learning** through BPTT.
- Useful for **sequence modeling** (e.g., language, speech, time series).

Disadvantages:

- **Vanishing/exploding gradients** can occur when backpropagating through many time steps.
- **Memory consumption** increases with sequence length (due to storage of intermediate activations).

Real-Time Applications:

- **Speech recognition**
- **Machine translation**
- **Stock market prediction**
- **Text generation**
- **Music composition**

Bidirectional Recurrent Neural Networks (BRNNs) in Deep Learning

A Bidirectional Recurrent Neural Network (BRNN) is an extension of the traditional Recurrent Neural Network (RNN) architecture.

In a standard RNN, information flows only in one direction — from past to future (forward in time).

However, in many real-world tasks (like speech recognition or language translation), context from both past and future is important.

To handle this, BRNNs process data in both forward and backward directions, combining the results for better understanding.

A **BRNN** consists of **two RNNs**:

- One processes the input sequence **from left to right (forward pass)**.
- The other processes the same sequence **from right to left (backward pass)**.

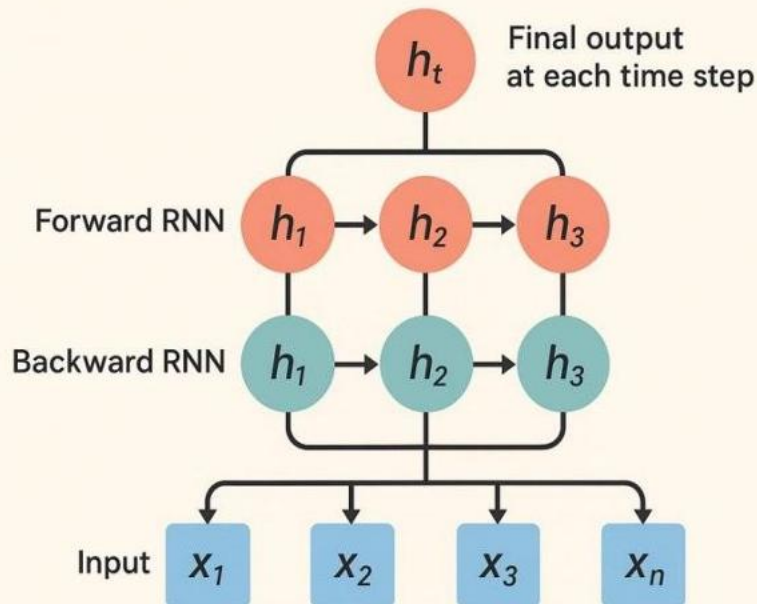
The outputs from both directions are then **combined** (concatenated, summed, or averaged) at each time step.

Visualization:

At each time step t , the model considers:

- **Past context** (from forward RNN)
- **Future context** (from backward RNN)

BIDIRECTIONAL RNN



4. Mathematical Representation

Let the input sequence be:

$$X = (x_1, x_2, \dots, x_T)$$

Forward pass:

$$\vec{h}_t = f(W_{xh}x_t + W_{hh}\vec{h}_{t-1} + b_h)$$

Backward pass:

$$\overleftarrow{h}_t = f(W'_{xh}x_t + W'_{hh}\overleftarrow{h}_{t+1} + b'_h)$$

Output:

$$y_t = g(W_{hy}[\vec{h}_t; \overleftarrow{h}_t] + b_y)$$

where:

- $[\vec{h}_t; \overleftarrow{h}_t]$ is the concatenation of forward and backward hidden states.

5 . Working Example

Suppose we have a sentence:

“The cat sat on the mat.”

To understand the word “**sat**”, we might need:

- Past context: “The cat”
- Future context: “on the mat”

A **unidirectional RNN** only knows “The cat” → limited context.

A **bidirectional RNN** knows **both** sides, giving a **better understanding** of “sat”.

6. Types of BRNN Layers

- **Bidirectional Simple RNN**
- **Bidirectional LSTM** (Long Short-Term Memory)
- **Bidirectional GRU** (Gated Recurrent Unit)

Advantages

- ✓ **Better context understanding:** Uses information from both past and future.
- ✓ **Improved accuracy:** Particularly effective in sequence labelling tasks.
- ✓ **Useful for NLP tasks:** Where understanding meaning depends on full sentence context.

8. Disadvantages

Cannot be used in real-time predictions:

Since future data must be known, BRNNs are not suitable for streaming or online data.

Higher computational cost:

Two RNNs are trained simultaneously.

Memory intensive:

Needs more storage for parameters and hidden states.

9. Real-Time Applications

1. **Speech Recognition** – Understanding full speech context improves accuracy.
2. **Named Entity Recognition (NER)** – Identifying entities in text using both past and future words.
3. **Part-of-Speech Tagging** – Knowing both preceding and succeeding words helps in accurate tagging.
4. **Machine Translation** – Better captures the full meaning of sentences.
5. **Handwriting Recognition** – Context from both directions improves character identification.

10. Example Problem

Task: Predict the part of speech (POS) of each word in the sentence

“Time flies like an arrow.”

Using BRNN:

- **Forward RNN:** reads from “Time” → “arrow.”
- **Backward RNN:** reads from “arrow” → “Time.”
- Final output for “flies” depends on **both directions**, improving accuracy.

11. Comparison Table

Feature	RNN	BRNN
Direction	Forward only	Forward + Backward
Context	Past only	Past + Future
Real-time use	Possible	Not possible
Accuracy	Moderate	High
Computation	Less	Double

12. Summary

Concept	Description
Full Form	Bidirectional Recurrent Neural Network
Purpose	Learn dependencies from both directions in sequence data
Input Flow	Forward and backward simultaneously
Applications	NLP, Speech, Handwriting recognition
Key Limitation	Requires full sequence before prediction

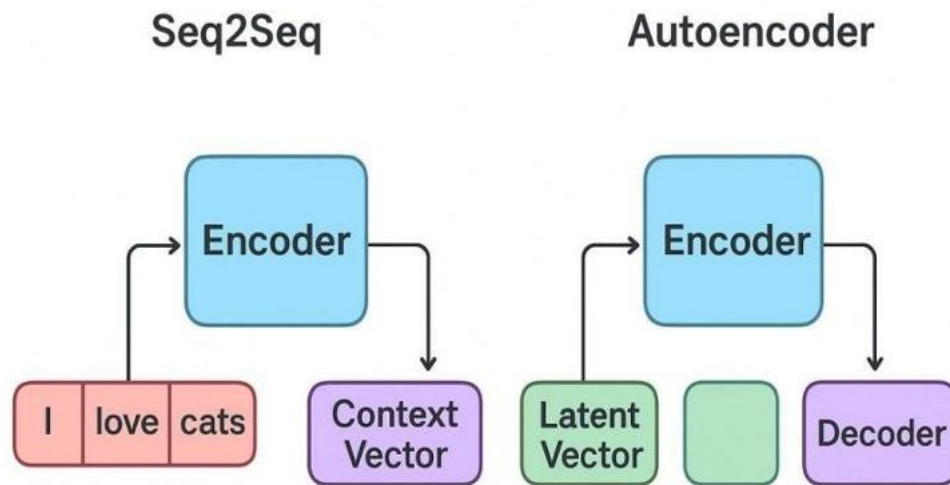
What is an Encoder in Deep Learning (DL)?

◆ 1. Definition

In **Deep Learning**, an **Encoder** is a **neural network module** that **processes input data** (like text, images, or audio) and **converts it into a meaningful internal representation** — usually a **vector or set of vectors** called the **context vector**, **latent representation**, or **feature embedding**.

An encoder compresses the input information into a compact form that captures its essential meaning or features.

Encoder in Deep Learning



◆ 2. Purpose of the Encoder

- To **extract features** or **hidden patterns** from raw data.
- To **reduce dimensionality** while preserving useful information.
- To **represent the input** in a form that can be easily processed by another part of the model (like a Decoder).

◆ 3. Where Encoders Are Used

Encoders are used in many deep learning architectures, including:

Architecture	Role of Encoder
Seq2Seq (Encoder–Decoder)	Converts input sequence into context vector
Autoencoders	Compress input to latent representation
Transformers	Convert input tokens to embeddings with attention

Architecture	Role of Encoder
CNN-based models	Extract spatial features from images
Speech models (ASR)	Encode audio signals into phonetic/semantic features

◆ 4. Working Principle

The encoder reads an **input sequence or data**, processes it step-by-step (or layer-by-layer), and outputs a **representation vector** summarizing the important information.

Example in text (sequence input):

```
[
x_1, x_2, x_3, ..., x_T \rightarrow \text{Encoder} \rightarrow h_T = C
]
```

Where:

- (x_i) = input token
- (h_T) = final hidden state (context vector)
- (C) = encoded representation

5. Encoder in Sequence-to-Sequence Models

In **Encoder–Decoder architectures**, the encoder is usually built using **RNN**, **LSTM**, or **GRU** units.

- It reads the entire input sentence token-by-token.
- Each token updates a **hidden state**.
- The **final hidden state** represents the entire sentence.

Example:

Input: "I love cats"

Output (Encoder): Context vector = [0.52, 0.13, -0.44, 0.67, ...]

This vector is then passed to the **decoder** to generate the translated or predicted output.

◆ 6. Encoder in Autoencoders

An **Autoencoder** consists of:

- **Encoder** → Compresses input data into a **latent vector (bottleneck layer)**.
- **Decoder** → Reconstructs the original input from that latent vector.

Input → Encoder → Latent Vector → Decoder → Output (Reconstruction)

Example:

- Input: 784-dimensional image (28×28 pixels)
- Encoder: Reduces to 64 features
- Decoder: Expands back to 784 pixels

◆ 7. Encoder in Transformers

In **Transformer** models (like BERT, T5, and GPT), the encoder uses **self-attention mechanisms** to process all input tokens simultaneously and understand their relationships.

Each encoder layer:

- Applies **multi-head self-attention**
- Applies **feed-forward neural network**
- Outputs **contextualized embeddings**

8. Mathematical Representation (RNN Encoder)

For time steps ($t = 1, 2, \dots, T$):

$$\begin{bmatrix} h_t = f(Wx_t + Uh_{t-1} + b) \end{bmatrix}$$

Where:

- (x_t): input at time t
- (h_t): hidden state
- (W, U, b): learnable parameters
- (f): non-linear activation (e.g., tanh)

The final hidden state (h_T) represents the entire input — the **encoded vector**.

◆ 9. Types of Encoders

Encoder Type	Input	Processing	Example Use
RNN Encoder	Sequence	Temporal	Text, Speech
CNN Encoder	Image	Spatial	Vision tasks
Transformer Encoder	Sequence	Attention-based	NLP tasks
Autoencoder Encoder	Any data	Compression	Denoising, feature learning

◆ 10. Real-Life Example

Google Translate Encoder:

When you type a sentence in English, the encoder converts each word into embeddings and learns their contextual relationships to form a **semantic vector** that represents the entire sentence's meaning.

This vector is then given to the **decoder**, which generates the corresponding sentence in another language.

11. Key Features of an Encoder

- Captures **contextual meaning**
- Produces **fixed-size representation** from variable-length input
- Learns **hierarchical features** automatically
- Can be **stacked** (multi-layer encoders) for deeper understanding

◆ 12. Advantages

- ✓ Converts complex input into meaningful features
- ✓ Can handle variable-length inputs
- ✓ Helps reduce data redundancy
- ✓ Reusable for multiple downstream tasks (transfer learning)

◆ 13. Disadvantages

- ✗ May lose information if compressed too much
- ✗ Difficult to interpret latent representations
- ✗ Needs large datasets for training effectively

Decoder in Deep Learning (DL) — Sequence-to-Sequence Models

◆ 1. Definition

In **Deep Learning**, a **Decoder** is a neural network component that **generates an output sequence** from a given **encoded representation (context vector)** produced by the **Encoder**.

It's mainly used in **Sequence-to-Sequence (Seq2Seq)** architectures to convert the encoded information back into a readable or interpretable sequence — such as words, text, or tokens.

The Decoder “translates” the compact representation (context vector) into the final output sequence.

◆ 2. Purpose of the Decoder

- To **generate sequential output** (like translated text, summarized content, or captions).
- To **interpret the context vector** created by the encoder.
- To **predict the next element** in the output sequence step by step.

◆ 3. Position in Seq2Seq Architecture

Input Sequence → [Encoder] → Context Vector → [Decoder] → Output Sequence

✿ Example:

Input: “I love cats”

Output: “J’aime les chats”

Here:

- **Encoder:** Understands and compresses “I love cats”
- **Decoder:** Expands the meaning into “J’aime les chats”

◆ 4. How the Decoder Works

The Decoder:

1. **Takes the context vector** from the encoder.
2. **Generates one output token** at a time (e.g., one word).
3. **Uses its previous output** as input for predicting the next token.

Mathematically:

$$\begin{aligned}
 &[\\
 &s_t = f(y_{t-1}, s_{t-1}, C) \\
 &] \\
 &[\\
 &P(y_t | y_{<t}, X) = \text{softmax}(W s_t) \\
 &]
 \end{aligned}$$

Where:

- (s_t) : Decoder's hidden state at time t
- (y_{t-1}) : Previously generated word
- (C) : Context vector from Encoder
- (f) : LSTM/GRU function

◆ 5. Example of Decoding Steps

Step	Input to Decoder	Output (Prediction)
1	Context Vector + <start>	"J"
2	Previous output "J"	"aime"
3	Previous output "aime"	"les"
4	Previous output "les"	"chats"
5	Stop at <end> token	—

◆ 6. Teacher Forcing (During Training)

During training, instead of feeding the decoder's **predicted output**, we feed the **true previous word** from the training data.

✓ This technique helps the model learn faster and stabilize training.

◆ 7. Decoder in Attention Models

In **Attention-based Seq2Seq models**, the decoder doesn't rely on a single context vector.

Instead, it **attends to different parts of the encoder output** dynamically for each word it generates.

$$c_t = \sum_i \alpha_{t,i} h_i$$

Where:

- (h_i) : Encoder hidden states
- $(\alpha_{t,i})$: Attention weights showing focus on each input word

◆ **Benefit:** Decoder can “look back” at relevant input words while generating each output.

◆ 8. Components of a Decoder

Component	Function
Input Embedding	Converts tokens to numeric vectors
RNN/LSTM/GRU Layers	Process sequence step-by-step
Attention Mechanism	Focuses on relevant encoder outputs
Output Layer	Generates probabilities for next token

◆ 9. Decoder Types

Decoder Type	Description	Example Use
RNN Decoder	Sequential generation	Early NMT models
LSTM Decoder	Handles long-term dependencies	Translation, Chatbots
GRU Decoder	Lightweight version of LSTM	Real-time applications
Transformer Decoder	Uses self-attention & cross-attention	GPT, T5, BART

Applications

- ✓ Neural Machine Translation (e.g., English → French)
- ✓ Text Summarization
- ✓ Image Captioning
- ✓ Speech-to-Text
- ✓ Chatbots and Dialogue Systems

Advantages

- ✓ Can generate variable-length output
- ✓ Learns temporal or contextual dependencies
- ✓ Works well with attention and transformers
- ✓ Handles complex sequence tasks

13. Disadvantages

- ✗ Sequential decoding is **slow** (one token at a time)
- ✗ May produce **errors that accumulate** over time
- ✗ Struggles with **long outputs** without attention

Feature	Description
Definition	Neural network that generates output from encoded features
Input	Context vector (from encoder)

Feature	Description
Output	Sequence of predictions (words, tokens, etc.)
Common Models	RNN, LSTM, GRU, Transformer
Key Role	Converts encoded meaning back into sequence form

Encoder Decoder Models

In deep learning the encoder-decoder model is a type of neural network that is mainly used for tasks where both the input and output are sequences. This architecture is used when the input and output sequences are not the same length for example translating a sentence from one language to another, summarizing a paragraph, describing an image with a caption or convert speech into text. It works in two stages:

- **Encoder:** The **encoder** takes the input data like a sentence and processes each word one by one then creates a single, fixed-size summary of the entire input called a **context vector** or **latent space**.
- **Decoder:** The **decoder** takes the context vector and begins to produce the output one step at a time.

*For example, in **machine translation** an encoder-decoder model might take an English sentence as input (like "I am learning AI") and translate it into French ("Je suis en train d'apprendre l'IA").*

Encoder-Decoder Model Architecture

In an encoder-decoder model both the encoder and decoder are separate networks each one has its own specific task. These networks can be different types such as [Recurrent Neural Networks \(RNNs\)](#), [Long Short-Term Memory networks \(LSTMs\)](#), [Gated Recurrent Units \(GRUs\)](#), [Convolutional Neural Networks \(CNNs\)](#) or even more advanced models like [Transformers](#).

Encoder Decoder Architecture

Encoder

The encoder's job is to process the input data and convert it into a form that the model can understand. It does this using two main steps:

1. **Self-Attention Layer:** This layer helps the encoder focus on different parts of the input data that are important for understanding the context. For example in a sentence it allows the model to consider how each word relates to the others.
2. **Feed-Forward Neural Network:** After the self-attention layer this network processes the information further to capture complex patterns and relationships in the data.

Encoder and Decoder

Decoder

The decoder takes the processed information from the encoder and generates the output. It also has three main components:

1. **Self-Attention Layer:** Similar to the encoder this layer allows the decoder to focus on different parts of the output it has generated.
2. **Encoder-Decoder Attention Layer:** This unique layer enables the decoder to focus on relevant parts of the input data help to generate more accurate outputs.
3. **Feed-Forward Neural Network:** Like the encoder the decoder uses this network to process the information and generate the final output.

Working of Encoder Decoder Model

The actual working of the encoder decoder model is shown in below diagram. Now we will understand it stepwise:

Working of Encoder Decoder

Step 1: Tokenizing the Input Sentence

- The sentence "I am learning AI" is first broken into tokens: ["I", "am", "learning", "AI"].
- Each word (token) is converted into a vector that a machine can understand. This process is called embedding.

Step 2: Encoding the Input

- The Encoder processes these embeddings using [self-attention](#).
- Self-attention helps the encoder to focus on important words. For example while encoding "learning", it understands its relation with "I" and "AI."
- After processing the encoder generates a Context Vector which captures the meaning of the entire sentence. For example in the image The arrows show how each word relates to the others during encoding. The final output from the encoder is the context representation

Step 3: Passing the Context to the Decoder

- The Context Vector is passed to the Decoder as shown in image.
- It acts like a summary of the full input sentence.

Step 4: Decoder Generates Output Step-by-Step

- The Decoder uses the context and starts creating the output one word at a time.
- First it predicts the first word then uses that to predict the second word and so on

Step 5: Decoder Attention

- While generating each word the decoder attends to different parts of the input sentence to make better predictions.
- For example when translating "learning," it might pay more attention to the word "learning" in the input.

Step 6: Producing the Final Output

- The decoder continues generating until the full translated sentence is produced.

- Each output token depends on the previous ones and the input context. You finally see the output tokens generated on the right side of the diagram completing the translation.

4 : Encoder–Decoder Sequence-to-Sequence Architectures

☀ What is an Encoder in Deep Learning?

An **Encoder** is a **neural network component** used to **convert input data (like text, images, or audio)** into a **fixed-length numerical representation (vector)** that captures its essential information or meaning.

Encoders are mainly used in **sequence-to-sequence (Seq2Seq)** architectures, **autoencoders**, and **transformer models**.

◆ Definition

An **Encoder** is a neural network that **processes an input sequence** and **encodes** it into a **context vector** (also called a hidden state or latent representation). This **context vector** summarizes all the important information in the input so that another network (the **decoder**) can use it to generate the desired output.



◆ Mathematical Representation

Suppose the input sequence is:

[
 $X = (x_1, x_2, x_3, \dots, x_T)$
]

where (T) is the sequence length.

The encoder processes the sequence step by step using a Recurrent Neural Network (RNN), GRU, or LSTM:

$$\begin{aligned} [\\ h_t = f(Wx_t + Uh_{t-1} + b) \\] \end{aligned}$$

Here,

- (h_t): hidden state at time step t (encodes current and past information)
- (x_t): input at time step t
- (W, U, b): model parameters
- (f): activation function (like tanh or ReLU)

The **final hidden state** (h_T) is often used as the **context vector**, representing the entire input sequence.

◆ Working Steps of an Encoder

1. **Input Representation:**
Converts raw input (e.g., words or pixels) into embeddings (numerical form).
2. **Sequential Processing:**
Processes the sequence one element at a time, capturing temporal or spatial relationships.
3. **Context Vector Formation:**
Generates a compressed representation that summarizes the input data.
4. **Transfer to Decoder:**
Passes this context vector to the decoder for output generation.

◆ Example:

If the input sentence is:

👉 "I love deep learning"

The encoder converts it into:

$$\begin{aligned} [\\ \text{Context Vector} = [0.4, -0.2, 0.8, \dots] \end{aligned}$$

]

—a compact vector summarizing the entire meaning of the sentence.

◆ Encoder Types in Deep Learning

Type	Used In	Description
RNN Encoder	Seq2Seq models	Captures sequential data (e.g., text, time series).
CNN Encoder	Image processing	Extracts spatial features from images.
Transformer Encoder	NLP tasks	Uses self-attention to capture global dependencies.
Autoencoder Encoder	Unsupervised learning	Compresses input into a latent code for reconstruction.

◆ Applications

- **Machine Translation** – Encoder reads input language.
- **Speech Recognition** – Encodes audio features into text representation.
- **Image Captioning** – CNN encoder extracts image features.
- **Text Summarization** – Encodes a document into semantic representation.

◆ Diagram

Input Sequence → [Encoder Network] → Context Vector

Example:

“I love deep learning” → [LSTM Encoder] → [0.4, -0.2, 0.8, ...]

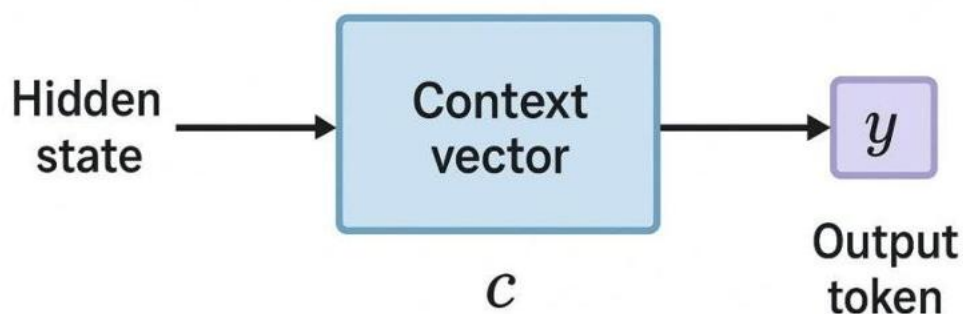
◆ Summary

Feature	Description
Purpose	Convert input sequence into a meaningful numerical representation
Output	Context vector / hidden state
Common Architectures	RNN, GRU, LSTM, CNN, Transformer
Role	Acts as the first part of the Encoder–Decoder framework

What is an Decoder in Deep Learning

In **Deep Learning (DL)**, a **Decoder** is a neural network component that **transforms an internal representation (or encoded feature vector) back into an output sequence, image, or signal**. It is most commonly used in **Encoder–Decoder architectures** like **Sequence-to-Sequence (Seq2Seq)** models, **Autoencoders**, and **Transformers**.

DECODER



Definition

A **Decoder** is a neural network that takes a **context vector** (the encoded information from the encoder) and **generates output data** step by step, often in the form of a sequence.

$$\left[\begin{array}{l} \text{Decoder: } y = f_{\text{dec}}(h_t, c) \end{array} \right]$$

where

- (h_t) = hidden state of the decoder at time step t
- (c) = context vector from the encoder
- (y) = predicted output at time t

Working Principle

1. Input from Encoder:

The decoder receives the **context vector** (summarized information of the input sequence) from the encoder.

2. Initial Hidden State:

The decoder's first hidden state is usually initialized with the encoder's final hidden state.

3. Sequential Output Generation:

The decoder generates the output **one step at a time**, using the previous output as the next input (in models like RNN-based Seq2Seq).

4. Prediction:

Each step produces a token (like a word in a sentence) until an **End-of-Sequence (EOS)** token is reached.

Architecture Example

In Sequence-to-Sequence Model:

- **Encoder:** Processes input sequence → outputs a context vector.
- **Decoder:** Takes context vector → generates target sequence.

Decoder Equations (for RNN-based models):

```
[
s_t = f(s_{t-1}, y_{t-1}, c)
]
[
\hat{y}_t = g(s_t)
]
```

Where:

- (s_t): Decoder hidden state at time t
- (y_{t-1}): Previous predicted output
- (c): Context vector
- ($g(\cdot)$): Output layer (e.g., softmax)

Example Use Cases

Task	Input (Encoder)	Output (Decoder)
Machine Translation	English sentence	French sentence
Text Summarization	Long article	Short summary
Image Captioning	Image features	Sentence description
Speech Recognition	Audio features	Text sequence

Common Types of Decoders

1. **RNN-based Decoder** – Uses LSTM or GRU to produce sequential outputs.
2. **CNN-based Decoder** – Used in image generation (e.g., in autoencoders).
3. **Transformer Decoder** – Uses self-attention and cross-attention (used in GPT, BERT-decoder, etc.).

Example: Machine Translation

Input: "How are you?"

Encoder: Converts it into context vector c

Decoder: Uses c to generate → "Comment ça va ?"

Diagram Description (Text Form)

Input Sequence $\rightarrow$ [Encoder] $\rightarrow$ Context Vector $\rightarrow$ [Decoder] $\rightarrow$ Output Sequence

$(x_1, x_2, \dots, x_n)$ c $(y_1, y_2, \dots, y_m)$

Key Points to Remember

- Decoder **generates** outputs from encoded representations.
- Works **step-by-step** in sequence tasks.
- Often uses **attention mechanism** for focusing on relevant encoder outputs.
- Essential in models like **Seq2Seq**, **Transformers**, and **Autoencoders**.

5 : Deep Recurrent Networks and Recursive Neural Networks

A **Deep Recurrent Network** is an extension of a standard RNN where multiple recurrent layers are stacked on top of each other. Each layer processes sequential data and passes a hidden state to the next layer. This allows the network to learn hierarchical temporal features in sequences.

- Standard RNN: Single hidden layer.
- Deep RNN: Multiple hidden layers stacked.

Advantages

- Captures **complex temporal dependencies**
- Learns **hierarchical sequence representations**
- More expressive than single-layer RNNs

Disadvantages

- Harder to train (vanishing/exploding gradients)
- Computationally expensive
- Requires more data to avoid overfitting

Applications

- Language modeling
- Speech recognition

- Video sequence analysis
- Time series forecasting

Recursive Neural Networks (RecNNs)

Definition

A **Recursive Neural Network** is a type of neural network that works on **structured data**, such as **trees**, rather than flat sequences. It recursively applies the same set of weights to combine child nodes into a parent representation.

- Unlike RNNs, which model sequences linearly, RecNNs **capture hierarchical structure**.
- Commonly used in natural language processing (NLP) for parsing sentences.

Advantages

- Captures **hierarchical structures** in data
- Parameter sharing reduces **model size**
- Good for syntactic and semantic modeling

Disadvantages

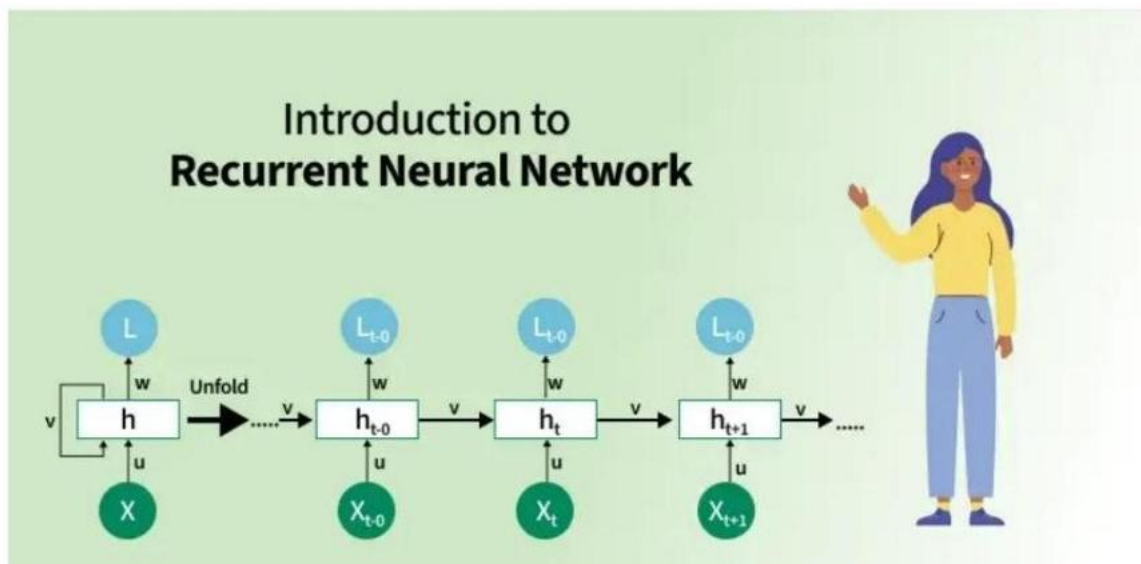
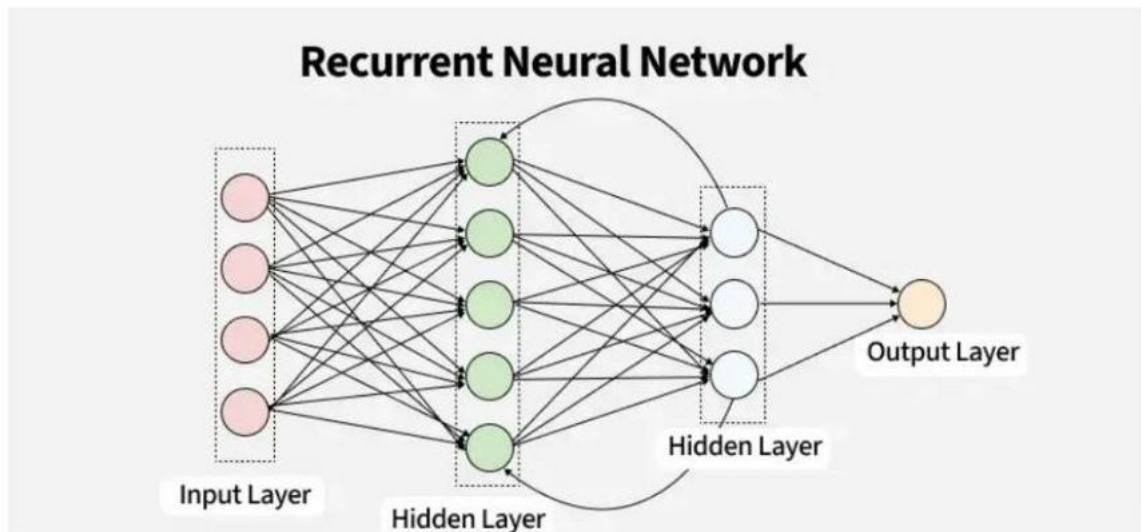
- Harder to parallelize
- Sensitive to **tree structure quality**
- Not suitable for flat sequences without inherent hierarchy

Applications

- Sentiment analysis (e.g., Stanford Sentiment Treebank)
- Parsing natural language
- Program analysis (ASTs)
- Scene understanding in images (hierarchical object composition)

Recurrent Neural Networks (RNNs) differ from regular neural networks in how they process information. While standard neural networks pass information in

one direction i.e. from input to output, RNNs feed information back into the network at each step.



Lets understand RNN with a example:

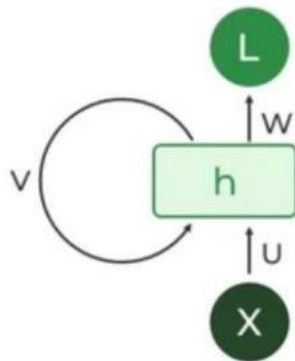
Imagine reading a sentence and you try to predict the next word, you don't rely only on the current word but also remember the words that came before. RNNs work similarly by "remembering" past information and passing the output from one step as input to the next i.e it considers all the earlier words to choose the most likely next word. This memory of previous steps helps the network understand context and make better predictions.

Key Components of RNNs

There are mainly two components of RNNs that we will discuss.

1. Recurrent Neurons

The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

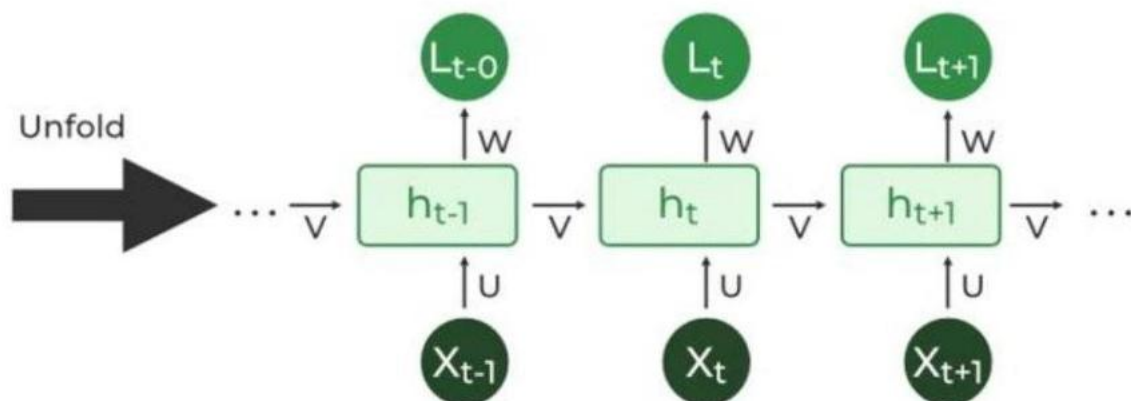


Recurrent Neuron

2. RNN Unfolding

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.

This unrolling enables [backpropagation through time \(BPTT\)](#) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.



RNN Unfolding

Recurrent Neural Network Architecture

RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks where each dense layer has distinct weight matrices. RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs the hidden state H_i is calculated for every input X_i to retain sequential dependencies. The computations follow these core formulas:

1. Hidden State Calculation:

$$h = \sigma(U \cdot X + W \cdot h_{t-1} + B)$$

Here:

- h represents the current hidden state.
- U and W are weight matrices.
- B is the bias.

2. Output Calculation:

$$Y = O(V \cdot h + C)$$

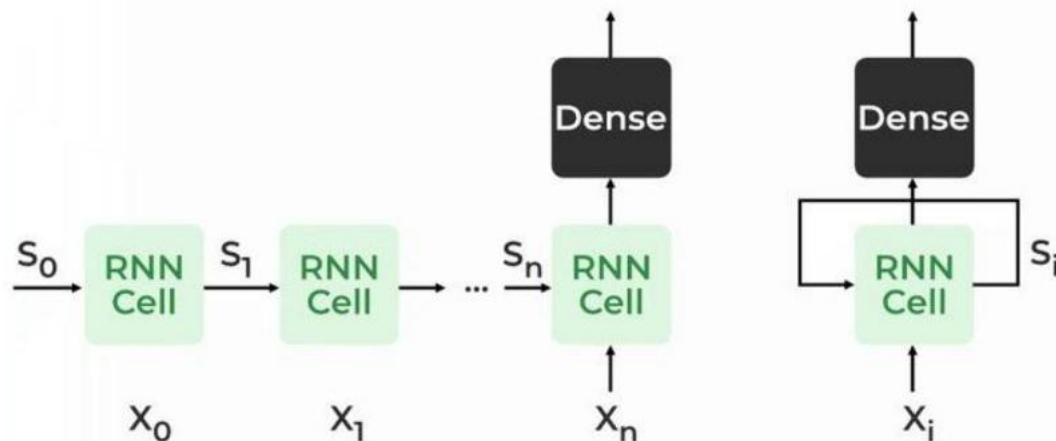
The output Y is calculated by applying O an activation function to the weighted hidden state where V and C represent weights and bias.

3. Overall Function:

$$Y = f(X, h, W, U, V, B, C)$$

This function defines the entire RNN operation where the state matrix S holds each element s_i representing the network's state at each time step i .

RECURRENT NEURAL NETWORKS



Recurrent Neural Architecture

How does RNN work?

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

Updating the Hidden State in RNNs

The current hidden state h_t depends on the previous state h_{t-1} and the current input x_t and is calculated using the following relations:

1. State Update:

$$h_t = f(h_{t-1}, x_t)$$

where:

- h_t is the current state
- h_{t-1} is the previous state
- x_t is the input at the current time step

2. Activation Function Application:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

Here, W_{hh} is the weight matrix for the recurrent neuron and W_{xh} is the weight matrix for the input neuron.

3. Output Calculation:

$$y_t = W_{hy} \cdot h_t$$

where y_t is the output and W_{hy} is the weight at the output layer.

These parameters are updated using backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as backpropagation through time.

Backpropagation Through Time (BPTT) in RNNs

Since RNNs process sequential data [Backpropagation Through Time \(BPTT\)](#) is used to update the network's parameters. The loss function $L(\theta)$ depends on the final hidden state h_3 and each hidden state relies on preceding ones forming a sequential dependency chain:

h_3 depends on h_2 , h_2 depends on h_1 , ..., h_1 depends on h_0 .

Backpropagation Through Time (BPTT) In RNN

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.

1. Simplified Gradient Calculation:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

2. Handling Dependencies in Layers: Each hidden state is updated based on its dependencies:

$$h_3 = \sigma(W \cdot h_2 + b)$$

The gradient is then calculated for each state, considering dependencies from previous hidden states.

3. Gradient Calculation with Explicit and Implicit Parts: The gradient is broken down into explicit and implicit parts summing up the indirect paths from each hidden state to the weights.

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2^+}{\partial W}$$

4. Final Gradient Expression: The final derivative of the loss function with respect to the weight matrix W is computed:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \sum_{k=1}^3 \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This iterative process is the essence of backpropagation through time.

Types Of Recurrent Neural Networks

There are four types of RNNs based on the number of inputs and outputs in the network:

1. One-to-One RNN

This is the simplest type of neural network architecture where there is a single input and a single output. It is used for straightforward classification tasks such as binary classification where no sequential data is involved.

One to One RNN

2. One-to-Many RNN

In a One-to-Many RNN the network processes a single input to produce multiple outputs over time. This is useful in tasks where one input triggers a sequence of predictions (outputs). For example in image captioning a single image can be used as input to generate a sequence of words as a caption.

One to Many RNN

3. Many-to-One RNN

The Many-to-One RNN receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction. In sentiment analysis the model receives a sequence of words (like a sentence) and produces a single output like positive, negative or neutral.

Many to One RNN

4. Many-to-Many RNN

The Many-to-Many RNN type processes a sequence of inputs and generates a sequence of outputs. In language translation task a sequence of words in one language is given as input and a corresponding sequence in another language is generated as output.

Many to Many RNN

Variants of Recurrent Neural Networks (RNNs)

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

1. Vanilla RNN

This simplest form of RNN consists of a single hidden layer where weights are shared across time steps. Vanilla RNNs are suitable for learning short-term dependencies but are limited by the vanishing gradient problem, which hampers long-sequence learning.

2. Bidirectional RNNs

[Bidirectional RNNs](#) process inputs in both forward and backward directions, capturing both past and future context for each time step. This architecture is ideal for tasks where the entire sequence is available, such as named entity recognition and question answering.

3. Long Short-Term Memory Networks (LSTMs)

[Long Short-Term Memory Networks \(LSTMs\)](#) introduce a memory mechanism to overcome the vanishing gradient problem. Each LSTM cell has three gates:

- **Input Gate:** Controls how much new information should be added to the cell state.
- **Forget Gate:** Decides what past information should be discarded.
- **Output Gate:** Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

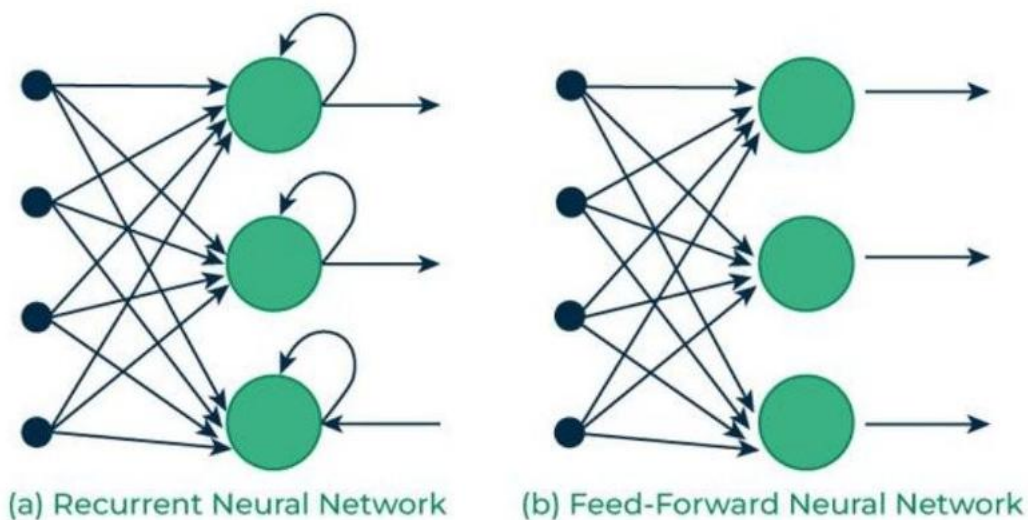
4. Gated Recurrent Units (GRUs)

[Gated Recurrent Units \(GRUs\)](#) simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism. This design is computationally efficient, often performing similarly to LSTMs and is useful in tasks where simplicity and faster training are beneficial.

How RNN Differs from Feedforward Neural Networks?

Feedforward Neural Networks (FNNs) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this by incorporating loops that allow information from previous steps to be fed back into the network. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



Recurrent Vs Feedforward networks

Implementing a Text Generator Using Recurrent Neural Networks (RNNs)

In this section, we create a character-based text generator using Recurrent Neural Network (RNN) in TensorFlow and Keras. We'll implement an RNN that learns patterns from a text sequence to generate new text character-by-character.

Long Short-Term Memory (LSTM) in Sequence Modeling,

1. Introduction

Long Short-Term Memory (LSTM) is a type of **Recurrent Neural Network (RNN)** specially designed to handle the problem of **long-term dependencies** in sequence data. Standard RNNs struggle with **vanishing and exploding gradients**, which prevents them from learning long-range dependencies effectively. LSTMs solve this by introducing a **memory cell** and **gating mechanisms** that control the flow of information.

Sequence modeling refers to tasks where the input data and/or output are sequences, such as:

- Time series forecasting
- Natural language processing (NLP)
- Speech recognition
- Video analysis
- Music generation

2. LSTM Architecture

An LSTM unit consists of three main components:

1. Cell State (C^t):

- Represents the long-term memory of the network.
- Acts as a “conveyor belt” allowing information to flow unchanged across time steps with minimal modifications.

2. Gates: Control the flow of information into and out of the cell.

- **Forget Gate (f^t)**: Decides which information from the previous cell state to discard.
- **Input Gate (i^t)**: Decides which new information to store in the cell state.
- **Output Gate (o^t)**: Decides what information to output based on the cell state.

3. LSTM Equations

Given input at time step x_t , hidden state h_{t-1} , and cell state C_{t-1} :

1. Forget Gate:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. Input Gate:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

3. Cell State Update:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

4. Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Where:

- (σ) is the sigmoid activation function.
- ($\tanh$) is the hyperbolic tangent activation function.
- (W) and (b) are weights and biases learned during training.

4. How LSTM Solves RNN Problems

Problem in RNN	LSTM Solution
Vanishing gradient	Cell state with linear interactions allows gradients to flow unchanged.
Long-term dependencies	Forget and input gates control which information is retained or discarded.
Exploding gradient	Gates help regulate information flow and stabilize learning.

6. Variants of LSTM

1. **Vanilla LSTM** – Standard architecture described above.
2. **Stacked LSTM** – Multiple LSTM layers stacked for deeper learning.
3. **Bidirectional LSTM (BiLSTM)** – Processes sequences in both forward and backward directions.
4. **Peephole LSTM** – Gates also take cell state as input.

7. Applications of LSTM in Sequence Modeling

1. **Natural Language Processing (NLP)**
 - Language modeling
 - Machine translation
 - Sentiment analysis
2. **Time Series Prediction**
 - Stock market forecasting
 - Weather prediction
3. **Speech Recognition**
 - Transcribing audio into text
4. **Video Analysis**

- Action recognition
- Gesture detection

5. Music Generation

- Composing melodies based on input sequences

8. Advantages of LSTM

- Handles long-term dependencies effectively.
- Reduces vanishing/exploding gradient problems.
- Flexible for sequences of variable lengths.
- Works well for diverse sequence data types (text, audio, video).

9. Limitations

- Computationally expensive compared to simple RNNs.
- Requires more memory due to multiple gates.
- Slower training for very long sequences.
- Overfitting if dataset is small.

UNIT – 2: Regularization for Deep Learning

- 1. Regularization and Under Constrained Problems,**
- 2. Dataset Augmentation,**
- 3. Noise Robustness,**
- 4. Semi-Supervised Learning, Multitask Learning,**
- 5. Early Stopping,**
- 6. Parameter Tying and Parameter Sharing,**
- 7. Sparse Representations,**
- 8. Bagging and Other Ensemble Methods,**
- 9. Dropout,**
- 10. Adversarial Training,**
- 11. Optimization for Training:**
- 12. Deep Models challenges in Neural Network Optimization,**
- 13. Basic Algorithms,**
- 14. Optimization Strategies and Meta-Algorithms**

1 : Regularization and Under Constrained Problems

Regularization and Under-Constrained Problems:

1. Regularization

Definition

Regularization is a set of techniques used in machine learning to prevent **overfitting** by adding a penalty (regularization term) to the model's loss function. It discourages the model from becoming too complex and helps it generalize better on unseen data.

- Common types: **L1 (Lasso)**, **L2 (Ridge)**, **Elastic Net**, **Dropout (for neural nets)**.

Advantages

- Prevents **overfitting**.
- Improves **generalization** of the model.
- Reduces **variance** without significantly increasing bias.
- Can perform **feature selection** (e.g., Lasso shrinks some weights to zero).

Disadvantages

- May lead to **underfitting** if penalty is too strong.
- Hyperparameter (regularization strength, e.g., λ) tuning is required.
- Interpretation of coefficients can be harder (especially with L2).

Applications

- Regression problems (Ridge/Lasso regression).
- Deep learning (Dropout, Weight decay).
- Natural language processing (avoiding overfitting on limited text data).
- Computer vision (CNNs with regularization to prevent memorizing training images).

2. Under-Constrained Problems

Definition

An **under-constrained problem** is one where the available information (data or constraints) is **insufficient** to uniquely determine a solution. In machine learning, this happens when:

- There are **fewer training samples** than model parameters.
- Multiple models/solutions fit the data equally well.

Example: Solving linear equations with more variables than equations → infinite possible solutions.

Advantages

- Offers **flexibility** to choose among many possible solutions.
- Useful in **ill-posed problems**, where approximate solutions are acceptable.

- Encourages creativity in solution space (e.g., generative models).

Disadvantages

- Leads to **overfitting**, since the model has too much freedom.
- Difficult to obtain a **unique, stable solution**.
- Sensitive to noise in data.
- Requires additional constraints or regularization to make the problem solvable.

Applications

- Computer Vision (image reconstruction, super-resolution).
- Natural Language Processing (sentence completion, translation with limited data).
- Signal Processing (denoising, audio reconstruction).
- Medical Imaging (MRI reconstruction from limited scans).
- Underdetermined linear systems in mathematics/engineering.

Describe L1 and L2 regularization techniques with mathematical formulations and their impact on weights.

L1 Regularization, also called Lasso (Least Absolute Shrinkage and Selection Operator) Regression, is a regularization technique that adds the absolute values of the model coefficients as a penalty term to the loss function.

It modifies the original cost function as:

$$\mathcal{L}_{L1} = \mathcal{L}_{original} + \lambda \sum_{i=1}^n |w_i|$$

where:

- $\mathcal{L}_{original}$ = original loss function (e.g., mean squared error in regression),
- λ = regularization parameter (controls penalty strength),
- w_i = model weights.

Advantages

1. Produces **sparse models** (many coefficients are zero).
2. Performs **automatic feature selection**.
3. Good when you believe only a few features are important.
4. Useful in **high-dimensional datasets** (e.g., text classification, genomics).

Disadvantages

1. Can be **unstable** when features are highly correlated (randomly picks one and ignores others).
2. May underperform if all features are important.
3. Optimization is harder than L2 because the absolute value is not differentiable at 0.

Applications

- **Feature selection** in machine learning pipelines.
- **Sparse signal reconstruction** (compressed sensing).
- **Genomics and bioinformatics** (where only some genes affect outcomes).
- **Natural Language Processing (NLP)** (e.g., selecting important words in text classification).

Example Problem :

Example Problem: Lasso Regression

Problem Statement

Suppose we are trying to predict a target y using two features x_1 and x_2 .

We fit a linear model:

$$\hat{y} = w_1x_1 + w_2x_2$$

Given:

- Training data:

x_1	x_2	y
1	0	2
0	1	2
1	1	3

- Initial weights: $w_1 = 1.0, w_2 = 1.0$
- Regularization parameter: $\lambda = 0.5$
- Loss function:

$$\mathcal{L}_{L1} = \frac{1}{2m} \sum_{i=1}^m (y_i - \hat{y}_i)^2 + \lambda(|w_1| + |w_2|)$$

Step 1: Compute Predictions

For each data point:

1. $x_1 = 1, x_2 = 0 \Rightarrow \hat{y} = 1(1) + 1(0) = 1$
Error = $y - \hat{y} = 2 - 1 = 1$
2. $x_1 = 0, x_2 = 1 \Rightarrow \hat{y} = 1(0) + 1(1) = 1$
Error = $2 - 1 = 1$
3. $x_1 = 1, x_2 = 1 \Rightarrow \hat{y} = 1(1) + 1(1) = 2$
Error = $3 - 2 = 1$

Step 2: Compute MSE Part

$$\text{MSE} = \frac{1}{2m} \sum (\text{error}^2) = \frac{1}{6}(1^2 + 1^2 + 1^2) = \frac{3}{6} = 0.5$$

Step 3: Add L1 Penalty

$$\lambda(|w_1| + |w_2|) = 0.5(|1| + |1|) = 0.5(2) = 1.0$$

Step 4: Total Loss

$$\mathcal{L}_{L1} = 0.5 + 1.0 = 1.5$$

Step 5: Effect of L1 Regularization

- To reduce total loss, optimization will shrink weights.
- Since L1 encourages sparsity, the optimizer may reduce one weight (say w_2) to 0, while keeping the other nonzero.
- The final model could look like:

$$\hat{y} = 2x_1 \quad (\text{ignoring } x_2)$$

2 : Dataset Augmentatio

Definition

Dataset Augmentation (or **Data Augmentation**) is the process of artificially increasing the size and diversity of a dataset by applying transformations or generating new data points from existing ones.

- It is commonly used in **machine learning** and **deep learning**, especially when the available dataset is small or imbalanced.
- The goal is to improve the **generalization** of models and reduce **overfitting**.

Advantages

1. **Increases dataset size** without needing new data collection (cost-effective).
2. **Improves model generalization** → model learns better and performs well on unseen data.
3. **Reduces overfitting** by providing varied examples.
4. **Balances imbalanced datasets** (e.g., minority classes can be augmented).
5. **Domain-specific flexibility**: different augmentation techniques for text, images, audio, tabular data.

Disadvantages

1. **Risk of unrealistic samples** → excessive augmentation may distort data and reduce accuracy.
2. **Computational overhead** → applying augmentations during training increases training time.
3. **Not a substitute for real data** → synthetic diversity cannot replace genuine variety.
4. **Bias introduction** → poorly chosen augmentation techniques can reinforce unwanted biases.
5. **Domain dependency** → what works in images may not work in text or time-series data.

Real-time Examples

1. **Image Classification (Computer Vision)**
 - Rotating, flipping, scaling, adding noise to images in datasets like CIFAR-10 or ImageNet.
 - Example: In self-driving cars, augmenting road sign images for different lighting conditions.
2. **Natural Language Processing (NLP)**
 - Synonym replacement, back-translation, or paraphrasing sentences to expand text datasets.
 - Example: Chatbot training with paraphrased queries.

3. Speech Recognition / Audio Processing

- Changing pitch, adding background noise, or speed variation to voice datasets.
- Example: Training Alexa/Google Assistant with noisy real-world environments.

4. Healthcare (Medical Imaging)

- MRI/X-ray images augmented by rotations, contrast adjustment, or synthetic generation (GANs).
- Example: Identifying tumors when real medical data is scarce.

5. Finance / Time-Series Data

- Bootstrapping, jittering, or window slicing to augment stock price or sensor data.
- Example: Fraud detection with imbalanced datasets (very few fraud cases).

3.Noise robustness in regularization for deep learning

Definition

Noise robustness in regularization means designing or applying regularization methods so that the deep neural network focuses on the underlying true signal in the data rather than memorizing noisy patterns.

For example, if some training labels are wrong (label noise) or some features are irrelevant (input noise), a good regularization method helps the network generalize well despite such disturbances.

How Regularization Helps with Noise Robustness

1. **L2 Regularization (Weight Decay):**
By penalizing large weights, L2 discourages the model from fitting random noise in the training data, since noise usually requires sharper, high-magnitude weight patterns to fit exactly.
2. **L1 Regularization:**
Promotes sparsity in weights, effectively ignoring noisy or irrelevant features by assigning them zero weights.
3. **Dropout:**
Randomly dropping neurons prevents the network from relying too much on specific features, making it less likely to memorize noisy patterns.
4. **Data Augmentation:**
Introducing transformations (rotations, flips, noise injection) increases robustness by forcing the model to learn invariances instead of noise.
5. **Early Stopping:**
Prevents the network from continuing training until it memorizes noisy details in the dataset.
6. **Adversarial Regularization / Noise Injection:**
Adding small perturbations (Gaussian noise) during training makes the model more stable and less sensitive to noise in inputs.

Advantages

- Improves **generalization** when training data is noisy.
- Prevents memorization of mislabeled data.
- Encourages learning **robust features** that remain stable even under perturbations.
- Makes models more resistant to adversarial attacks or unexpected variations.

Disadvantages

- If applied too strongly, regularization may also suppress useful patterns along with noise.
- Some methods (like dropout or noise injection) may slow down convergence.
- Robustness to one type of noise (e.g., label noise) does not guarantee robustness to another (e.g., adversarial noise).
- Requires careful hyperparameter tuning (e.g., dropout rate, λ in L2).

Real-Time Examples

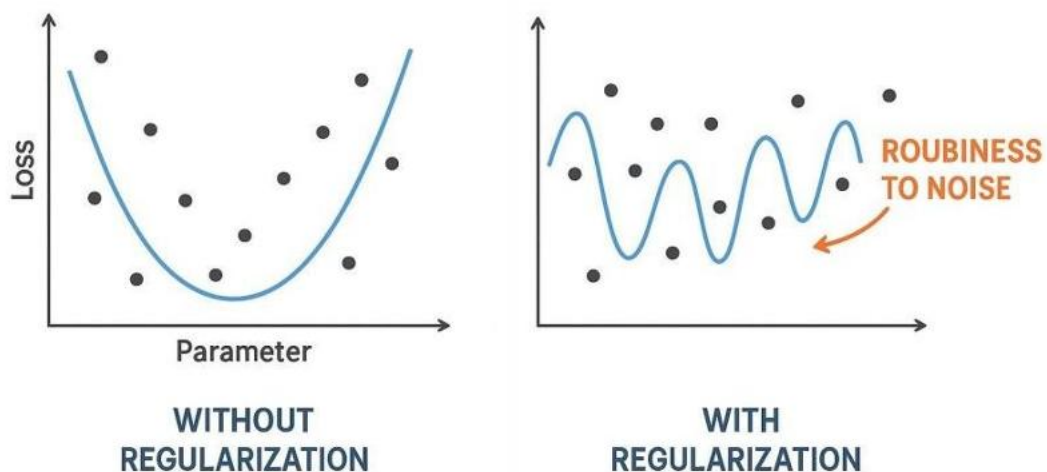
- **Speech recognition systems:** Training data often contains background noise (cars, fans, chatter). Regularization ensures the system recognizes speech robustly despite noise.
- **Medical imaging:** MRI/CT scans may have sensor noise or artifacts. Regularized models extract meaningful patterns without being misled by noisy pixels.
- **Financial fraud detection:** Transaction logs may contain errors or irrelevant attributes. Regularization helps the model focus on real fraud signals.
- **Autonomous vehicles:** Camera/LiDAR inputs have weather noise (rain, fog, dust). Regularized networks learn robust perception.

Example Problem

Suppose we train a neural network to classify handwritten digits (MNIST), but **20% of training labels are flipped randomly (label noise)**.

- Without regularization, the network can **memorize mislabeled samples**, leading to high training accuracy but poor test accuracy.
- With **L2 regularization ($\lambda = 0.001$) and dropout ($p = 0.5$)**, the model **ignores noisy labels** and focuses on the true digit structure (strokes, shapes).
- Result: Test accuracy improves significantly despite noisy training data.

NOISE ROBUSTNESS IN REGULARIZATION FOR DEEP LEARNING



4. Semi-supervised learning in Regularization for Deep learning

Definition

Semi-supervised learning (SSL) in deep learning is a learning paradigm where models are trained using **few labeled samples and many unlabeled samples**.

In regularization terms, SSL methods encourage the model to produce **consistent predictions** on unlabeled data and to learn **smooth decision boundaries**, thus reducing overfitting.

Semi-supervised learning in **Regularization for Deep Learning** is an important idea because it bridges **supervised learning (with labeled data)** and **unsupervised learning (with unlabeled data)**. Since labeled data is expensive and time-consuming to obtain, semi-supervised learning uses both **a small amount of labeled data** and **a large amount of unlabeled data** to improve model generalization.

Regularization plays a central role here: the unlabeled data acts as an **implicit regularizer**, preventing the model from overfitting to the limited labeled data.

How Semi-Supervised Learning Acts as Regularization

1. Consistency Regularization

- The model should give **similar outputs** for an input and its perturbed version (noise, augmentation, dropout, etc.).
- Example: If an image is slightly rotated, the predicted class should remain the same.
- This prevents the model from becoming too sensitive to noise → acts as a regularizer.

2. Entropy Minimization

- For unlabeled data, the model is pushed to make **confident predictions** (low entropy) instead of uncertain ones.
- Helps avoid overly complex boundaries around unlabeled points.

3. Pseudo-Labeling

- The model generates “pseudo labels” for unlabeled data and reuses them for training.
- Encourages the network to self-regularize through its own predictions.

4. Graph-based Regularization

- Data points close in feature space are assumed to have the same label.
- This smoothness constraint regularizes the model to avoid sharp, irregular decision boundaries.

Advantages

- Reduces overfitting when labeled data is scarce.
- Improves generalization by leveraging unlabeled data.
- Acts as a natural **regularizer without adding explicit penalties** (like L1/L2).
- Scales well for real-world problems (medical imaging, NLP, speech recognition).

Disadvantages

- Sensitive to incorrect pseudo labels → may propagate errors.
- Requires careful choice of data augmentation and consistency methods.
- Computationally more expensive than purely supervised training.
- If unlabeled data distribution is very different, it may **mislead** the model.

Real-Time Examples

1. **Medical Imaging**
 - Only a few X-rays/CT scans are labeled by doctors, but thousands of unlabeled scans exist.
 - SSL helps improve classification accuracy while preventing overfitting.
2. **Speech Recognition**
 - Few transcribed audio samples, many unlabeled recordings.
 - Semi-supervised consistency regularization improves robustness to noise.
3. **Text Classification**
 - Few labeled documents, large unlabeled corpus.
 - SSL smooths decision boundaries across unlabeled texts, acting as a regularizer.

Example Problem

Suppose we have **200 labeled images** of handwritten digits and **5000 unlabeled images**.

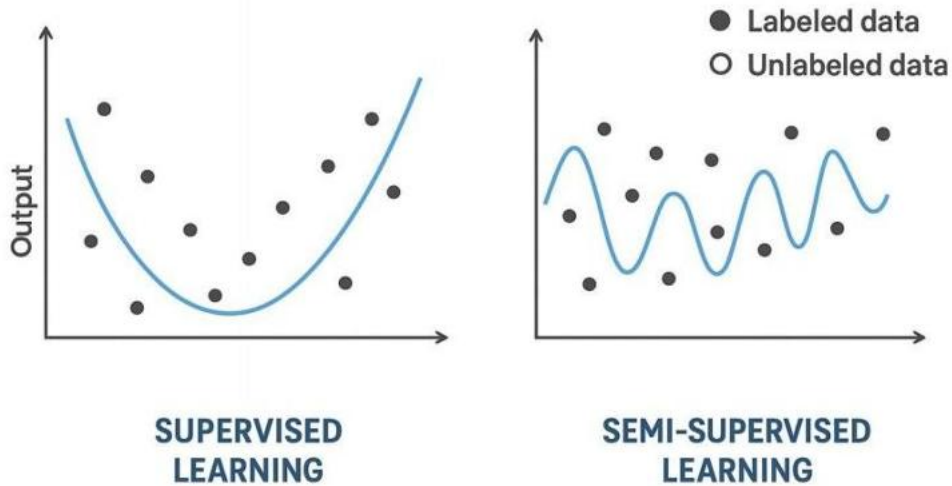
- **Without Regularization (purely supervised):**
Model overfits on 200 images, test accuracy = 60%.
- **With Semi-Supervised Regularization:**
Apply consistency regularization → model trained on labeled + unlabeled.
Model learns smooth decision boundaries, test accuracy rises to 85%.

🔍 Comparison: Semi-Supervised vs L1, L2, Dropout Regularization

Aspect	L1 Regularization (Lasso)	L2 Regularization (Ridge)	Dropout Regularization	Semi-Supervised Regularization
Definition	Adds penalty proportional to sum of absolute weights ($\ w\ _1$).	Adds penalty proportional to sum of squared weights ($\ w\ ^2$).	Randomly drops neurons during training to prevent co-adaptation.	Uses unlabeled data + consistency constraints to guide learning.
Type of Constraint	Sparsity (forces some weights → 0).	Weight shrinkage (reduces magnitude of all weights).	Network thinning (forces redundancy).	Smooth decision boundaries, consistency, entropy minimization.
Use of Data	Only labeled data.	Only labeled data.	Only labeled data.	Labeled + Unlabeled

Aspect	L1 Regularization (Lasso)	L2 Regularization (Ridge)	Dropout Regularization	Semi-Supervised Regularization
				data.
Acts on	Model parameters (weights).	Model parameters (weights).	Neurons/activations.	Predictions and feature space structure.
Goal	Feature selection (sparse models).	Avoid overfitting by controlling large weights.	Avoid overfitting by preventing reliance on specific neurons.	Avoid overfitting by learning smooth functions consistent with unlabeled data.
Computation Cost	Low.	Low.	Moderate (extra randomness).	High (needs large unlabeled set + extra loss terms).
When Useful	High-dimensional data, irrelevant features.	Small dataset, but many correlated features.	Large deep networks prone to overfitting.	Scarcity of labeled data but abundance of unlabeled data.
Risk / Limitation	May discard useful features.	May not produce sparse models.	Slower convergence, may underfit if too much dropout.	Wrong pseudo-labels may propagate errors; needs careful augmentation.
Real-Time Example	Text classification with many irrelevant words.	Predicting house prices with correlated features.	Image recognition (CNNs for CIFAR, ImageNet).	Medical imaging with few labeled scans + many unlabeled ones.

SEMI-SUPERVISED LEARNING IN REGULARIZATION FOR DEEP LEARNING



Example Problem: Predicting House Prices with Early Stopping

Problem Statement

We want to predict house prices based on features such as the number of rooms, area, location score, etc.

We train a neural network, but if we let it train too long, it will **overfit**. So, we apply **early stopping** to avoid that.

Dataset (Simplified)

Suppose we have **2000 house records**, each with:

- rooms
- area (in sq. ft.)
- location_score (rating out of 10)
- price (target variable)

We split into:

- Training data: 70%
- Validation data: 15%
- Test data: 15%

Model Setup

- Neural Network: 2 hidden layers (64, 32 units, ReLU activation)
- Optimizer: Adam
- Loss: Mean Squared Error (MSE)

Training Without Early Stopping

- Train for **200 epochs**.
- **Training Loss**: keeps decreasing (model memorizes training data).
- **Validation Loss**: starts decreasing initially, then after ~50 epochs it increases (overfitting begins).

Result:

- Train MSE: **5.2**
- Validation MSE: **12.8**
- Test MSE: **14.1**

The model is **overfitting**.

Training With Early Stopping

We apply **early stopping** with:

- `patience = 10` (stop if no improvement for 10 epochs).
- At **epoch 52**, validation loss stops improving.
- Training automatically halts at **epoch 62** (after patience).

 Result:

- Train MSE: **6.3**
- Validation MSE: **8.2**
- Test MSE: **8.5**

The model **generalizes better**.

Explanation of Early Stopping Effect

- Without early stopping → trained too long → overfit → poor test accuracy.
- With early stopping → training stopped at best point → balanced train and test performance.

Real-Life Connection

Think of a student preparing for an exam:

- If they **study too little** → underfit (don't learn enough).
- If they **study too much** and memorize books word by word → overfit (fail to apply knowledge).
- If they **study until they understand concepts well** and stop → that's **early stopping**

5 : Multitasking

Multitasking is the ability of an operating system to perform multiple tasks or processes concurrently by sharing CPU time and system resources.

- **The CPU rapidly switches between tasks (via context switching) to give the illusion of parallel execution.**
- **Example: Typing a report while listening to music and downloading files.**

Advantages of Multitasking

1. **Efficient CPU Utilization – CPU does not remain idle; it switches tasks when one is waiting for I/O.**
2. **Increased Productivity – Multiple programs run at the same time (e.g., coding + running browser).**
3. **Better Responsiveness – Real-time interaction with multiple applications (e.g., typing + spell-check running).**
4. **Flexibility – Enables background tasks (e.g., updates) while users work on the foreground.**
5. **Time-Saving – Reduces user waiting time as tasks progress simultaneously.**

Disadvantages of Multitasking

1. **Overhead of Context Switching – Frequent switching consumes CPU cycles.**
2. **Resource Contention – Multiple tasks may compete for CPU, memory, or I/O devices.**
3. **Reduced Performance – Too many tasks can slow down the system.**
4. **Complex OS Design – Scheduling, synchronization, and deadlock handling are difficult.**
5. **Possible System Crashes – A faulty process may affect other running tasks.**

Applications of Multitasking

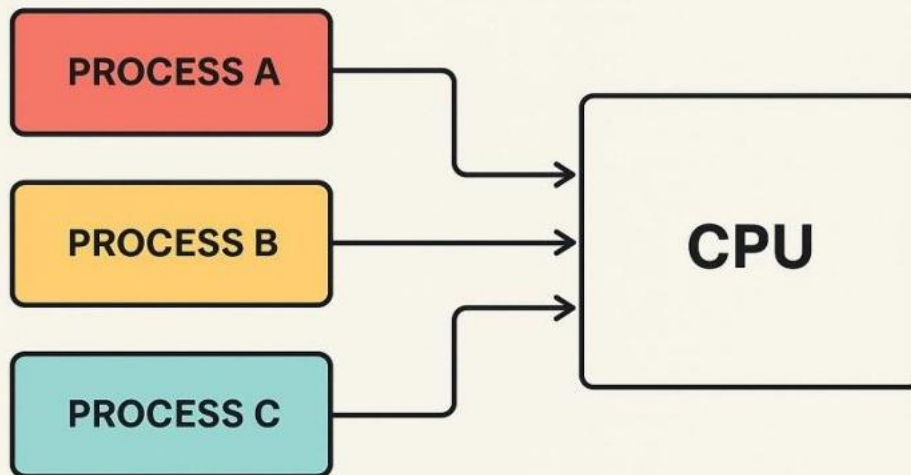
1. **Operating Systems – Windows, Linux, macOS handle multiple processes at once.**
2. **Mobile Phones – Running multiple apps in the background (calls + messaging + music).**
3. **Banking Systems – Handling multiple ATM or online transactions simultaneously.**
4. **Airline & Railway Reservations – Booking tickets for many users at the same time.**
5. **Real-Time Systems – Medical monitoring, traffic control, robotics (simultaneous task execution).**
6. **Servers & Cloud Systems – Web servers handle requests from thousands of users concurrently.**

Real-World Examples of Multitasking

1. **PC Usage – Writing a report in MS Word, downloading files, and running antivirus in the background.**

2. **Smartphones** – Receiving a WhatsApp call while updating apps and streaming music.
3. **Web Browsers** – Watching YouTube in one tab while downloading a PDF in another.
4. **Online Shopping Websites** – Amazon handles millions of purchase, payment, and shipping tasks simultaneously.
5. **ATMs** – Multiple people withdrawing/depositing cash at different machines at the same time.

MULTITASKING



6 : Early Stopping

Definition of Early Stopping

Early stopping is a regularization technique used in machine learning and deep learning to prevent overfitting.

- Training is stopped before the model starts overfitting, based on validation performance.
- Typically, training continues until the validation error stops decreasing for a certain number of iterations (patience).

Advantages of Early Stopping

1. Prevents Overfitting – Stops the model from memorizing training data.
2. Saves Training Time – No need to run unnecessary extra epochs.
3. Computational Efficiency – Reduces resource usage (CPU/GPU, electricity).
4. Easy to Implement – Works with almost any training algorithm.
5. Improves Generalization – Model performs better on unseen/test data.

Disadvantages of Early Stopping

1. Risk of Underfitting – Stopping too early may prevent the model from fully learning patterns.
2. Choice of Patience – Deciding when to stop (patience parameter) is tricky.
3. Validation Dependence – Requires a separate validation dataset, reducing data for training.
4. Not Always Optimal – In some problems, models may benefit from longer training + other regularization.
5. Fluctuating Validation Loss – Noise in validation metrics may cause premature stopping.

Applications of Early Stopping

1. Deep Learning (Neural Networks) – To avoid overfitting in CNNs, RNNs, transformers, etc.
2. Gradient Boosting (XGBoost, LightGBM, CatBoost) – Widely used to stop tree growth early.
3. Reinforcement Learning – Prevents over-training agents on specific environments.
4. Medical Image Processing – Ensures model generalizes well to unseen patient data.
5. Natural Language Processing (NLP) – Improves generalization in tasks like translation, sentiment analysis.

Real-World Examples of Early Stopping

1. Handwriting Recognition (MNIST Dataset) – Training stops once validation accuracy stabilizes, avoiding wasted epochs.
2. Stock Price Prediction – Prevents overfitting to historical noise, improving future prediction reliability.
3. Medical Diagnosis Systems – Prevents models from memorizing training patients' data, ensuring better generalization.
4. Recommendation Systems (Netflix, Amazon) – Helps avoid overfitting on a subset of users/items.
5. Speech Recognition (Google Assistant, Alexa) – Improves robustness by avoiding models memorizing specific accents in training data.

7 : Parameter Tying and Sharing , Sparse Representations in deep learning.

1. Definition

Parameter Tying

- **Parameter tying** means forcing certain parameters in a model to take the **same value** rather than learning them independently.
- Instead of each parameter being unique, multiple parameters are **linked (tied)**, so updating one updates all tied parameters.

Parameter Sharing

- **Parameter sharing** is a special case of parameter tying, where the **same parameter values** are reused in different parts of the model.
- Commonly used in **Convolutional Neural Networks (CNNs)**, where the same filter (weights) is applied across different spatial positions of an image.

2. Advantages

1 : **Reduced number of parameters** → Less memory usage, faster training, and less risk of overfitting.

2 : **Better generalization** → Since tied/shared parameters capture common patterns.

3 : **Efficient learning** → Instead of learning separate weights for each location or feature, one set of weights is reused.

4 : **Translation invariance** (in CNNs) → The same filter detects the same feature (e.g., edge, texture) in different positions.

5 : **Regularization effect** → Parameter tying acts like a constraint, preventing unnecessary complexity.

3. Example Problem

Without Parameter Sharing (Fully Connected Layer on Image)

Suppose we process a **28×28 image (like MNIST)**.

- A fully connected layer needs **784×100 = 78,400 weights** to connect to 100 hidden neurons.
- Each pixel has its **own independent weight** → too many parameters.

With Parameter Sharing (Convolutional Layer)

Now, instead of independent weights:

- We use a **5×5 filter (25 weights)** shared across the whole image.
- This same filter slides over the image (convolution), detecting the same pattern (e.g., edge, corner) anywhere.
- If we use 100 such filters → only **25×100 = 2,500 parameters**, instead of 78,400.

Numerical Illustration

Let's detect **horizontal edges** in a 5×5 patch:

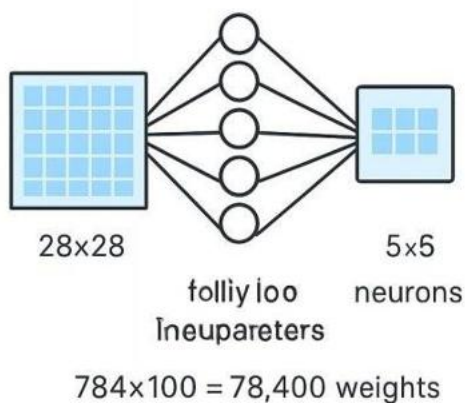
Filter weights (shared across the image):

$$W = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ +1 & +1 & +1 & +1 & +1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

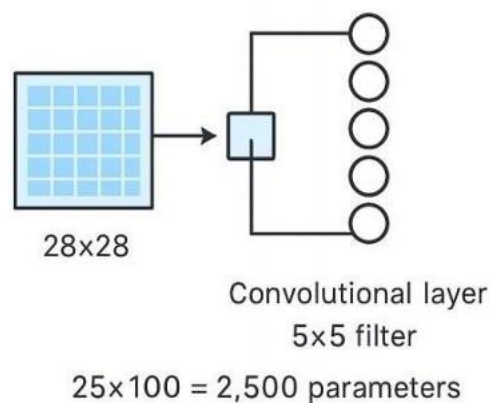
- This same **W** slides over all patches of the image.
- If a horizontal edge appears anywhere, the filter activates strongly.
- **Instead of learning different edge detectors for each location, we share one filter.**

PARAMETER TYING AND SHARING

WITHOUT PARAMETER SHARING



WITH PARAMETER SHARING



8. Sparse Representations in Deep Learning.

1. Definition

A **sparse representation** in deep learning means that out of all the possible features or neurons in a layer, **only a small fraction of them are active (non-zero) at a given time**, while most remain **zero or inactive**.

Formally:

- If a vector has many zeros and only a few non-zero elements → it is **sparse**.
- Sparse representations capture **essential information** using fewer active dimensions, making the representation **efficient and interpretable**.

Example:

Suppose we encode an image as a feature vector of length 1000.

- In a **dense representation**, maybe 800 values are non-zero.
- In a **sparse representation**, maybe only 50 values are non-zero → but they represent the **most relevant features** (e.g., edges, shapes, textures).

Advantages of Sparse Representations

- 1: **Efficiency** – Fewer active features → lower storage & computational cost.
- 2: **Better Generalization** – Reduces overfitting by focusing only on important features.

3: Interpretability – Active units correspond to meaningful patterns (e.g., edges in vision, keywords in NLP).

4: Biological inspiration – Similar to the brain, where only small subsets of neurons fire at a time.

5: Noise robustness – Unimportant/noisy features are suppressed (set to zero).

Applications of Sparse Representations

1. Computer Vision (CNNs)

- **Edge and texture detection** → Sparse activations mean only relevant filters fire (e.g., detecting a corner in an image).
- Used in **object recognition, face detection, image compression**.

2. Natural Language Processing (NLP)

- **Word Embeddings & One-Hot Encoding** → Words are often represented sparsely (most dimensions are 0).
- **Topic modeling** → Sparse representations identify only a few relevant topics per document.
- Used in **sentiment analysis, translation, chatbots**.

3. Recommender Systems

- User-item interaction matrices are extremely sparse (most users don't rate most items).
- Sparse representations help model **preferences** efficiently.
- Example: **Netflix recommendations, Amazon product suggestions**.

4. Autoencoders & Representation Learning

- **Sparse Autoencoders** impose sparsity in the hidden layer → learn compact, meaningful features.
- Used in **dimensionality reduction, anomaly detection, denoising**.

5. Speech & Audio Processing

- Sparse coding extracts only **important frequency components**.
- Used in **speech recognition, music classification, noise reduction**.

6. Medical Imaging

- Sparse representations help identify **only the important regions** (e.g., tumors in MRI scans).
- Reduces noise and improves interpretability for doctors.

7. Compressed Sensing

- Sparse signals can be reconstructed from fewer samples.

- Used in **MRI scans, radar systems, wireless communication.**

Sparse Representations in Deep Learning:

In deep learning, Sparse Representations refers to a form of data representation where only a small number of features (or Neurons) are active (Non-zero) at any given time, while the rest are zero (or) inactive.

This is in contrast to dense Representations, where most of the elements are non-zero.

Why use Sparse Representations:

Sparse Representations help in

1. Reducing computation and memory usage
2. Improving generalization by focusing only on the most relevant features
3. Enhancing interpretability, since fewer neurons are involved.
4. Mimicking how the human brain processes information (neurons fire sparsely)

Sparse Representations are Achieved:

1. L1 Regularization [Lasso]:

1. Adds penalty on the absolute values of weights (or) activations.
2. Encourage many weights/activations to become exactly zero.

2. Sparsity Constraints:

In autoencoders, constraints are applied to hidden layers so that only a few neurons activate for each input.

3. ReLU and Variants:

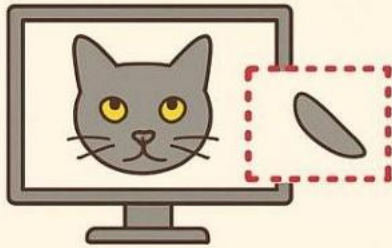
ReLU [Rectified Linear Unit] naturally introduce sparsity by outputting zero for negative values.

4. Dropout:

Temporarily drops [sets to zero] random neurons during training, introducing stochastic sparsity.

APPLICATIONS OF SPARSE REPRESENTATIONS

Computer Vision (CNNs)



Edge and texture detection

Natural Language Processing (NLP)



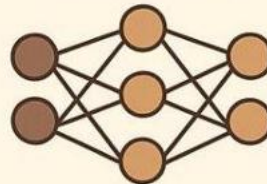
Word embeddings & one-hot encoding

Recommender Systems



User-item interaction

Autoencoders & Representation Learning



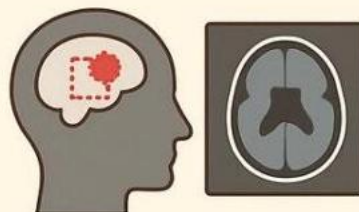
Sparse autoencoders

Speech & Audio Processing



Sparse coding

Medical Imaging



Sparse representations in MRI

9. Bagging and other ensemble methods in regularization definition advantages real time examples

1. Definition

Bagging (Bootstrap Aggregating)

- Bagging is an **ensemble method** where multiple models (usually decision trees) are trained on **different random subsets** of the training data (created via bootstrapping).
- The final prediction is made by **averaging** (for regression) or **majority voting** (for classification).
- It reduces **variance** and prevents **overfitting**, acting as a **regularization technique**.

Other Ensemble Methods for Regularization

1. Boosting

- Trains models sequentially, where each new model focuses on correcting errors of the previous one.
- Examples: AdaBoost, Gradient Boosting, XGBoost, LightGBM.
- Reduces **bias** and improves accuracy.

2. Stacking (Stacked Generalization)

- Combines multiple base models and uses another model (meta-learner) to learn how to best combine them.
- Regularizes by leveraging **strengths of different models**.

3. Random Forests

- Special case of Bagging with Decision Trees, but also introduces **random feature selection** → better regularization.

2. Advantages

Reduces Overfitting → Prevents models (like decision trees) from memorizing training data.

Improves Accuracy → Multiple weak learners combine into a strong learner.

Stable & Robust → Reduces variance (bagging) and bias (boosting).

Handles Noise → Ensemble methods smooth out errors from noisy data.

Parallelizable (Bagging & Random Forests) → Models can be trained independently.

3. Real-Time Examples

Bagging (Random Forest)

- **Healthcare:** Predicting diseases using patient data.
- **Finance:** Fraud detection (robust against noisy transaction patterns).
- **Image Recognition:** Random Forests used in medical image classification.

Boosting (XGBoost, AdaBoost, LightGBM)

- **Search Engines:** Ranking web pages (boosting models power Google's search ranking).
- **Finance:** Credit risk scoring (XGBoost is widely used in banking).
- **E-commerce:** Personalized recommendations (Amazon uses boosting methods).

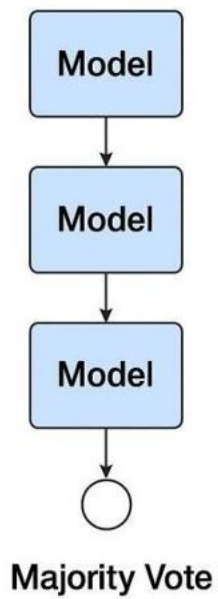
Stacking

- **Kaggle Competitions:** Winning solutions often use stacked ensembles.
- **Weather Forecasting:** Combining models (CNNs, RNNs, Trees) for better prediction.

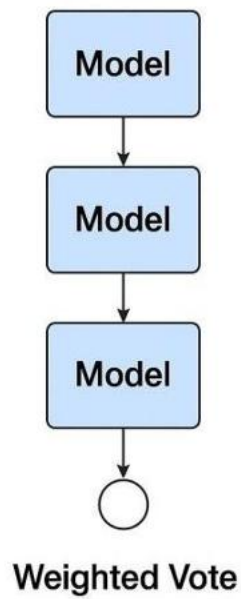
4. Example Problem (Bagging vs Single Tree)

- Suppose we want to classify whether a customer will **buy a product**.
- **Single Decision Tree** → High variance, might overfit.
- **Bagging (Random Forest with 100 trees)** →
 - Each tree sees a different bootstrap sample.
 - Final prediction = majority vote.
 - Much more stable, regularized, and accurate.

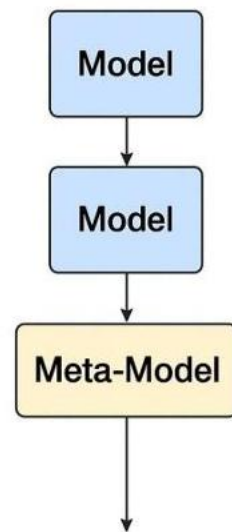
BAGGING



BOOSTING



STACKING



10.Dropout in Deep Learning

Dropout is a regularization technique in deep learning where, during training, some neurons are randomly "dropped" (i.e., set to zero) with a certain probability (e.g., 0.5). This prevents neurons from depending too much on each other (co-adaptation) and reduces overfitting.

Advantages of Dropout

1. **Reduces Overfitting** – prevents the model from memorizing training data.
2. **Improves Generalization** – makes the network more robust on unseen test data.
3. **Simple & Effective** – easy to implement with one line in most deep learning libraries.
4. **Acts like Model Averaging** – since different subnetworks are trained each iteration, it's like training an ensemble of networks.
5. **Boosts Performance in Deep Models** – especially in very large neural networks with many parameters.

Disadvantages of Dropout

1. **Slower Convergence** – training takes longer since neurons keep changing.
2. **May Underfit if Overused** – dropping too many neurons can reduce model capacity.
3. **Not Always Suitable** – sometimes ineffective for small datasets or shallow networks.
4. **Harder Hyperparameter Tuning** – requires adjusting dropout rate (e.g., 0.2, 0.5) carefully.
5. **Not Always Needed with Batch Normalization** – sometimes redundant if BN is used.

Real-Time Examples of Dropout

1. **Image Recognition (CNNs)** – Dropout is widely used in **ImageNet models** to avoid overfitting when classifying millions of images.
2. **Speech Recognition** – helps models generalize across accents and noisy environments.
3. **Fraud Detection** – prevents overfitting in financial transaction data with many features.
4. **Medical Diagnosis AI** – ensures predictions (like tumor detection) are not biased to training data only.
5. **Recommendation Systems** – improves personalization without overfitting to small groups of users.

Example Problem:

Problem Statement

Suppose we train a neural network for **handwritten digit classification (MNIST dataset, 60,000 training samples, 10 classes)**.

- Without dropout, the model achieves **99% training accuracy but only 91% test accuracy** (overfitting).

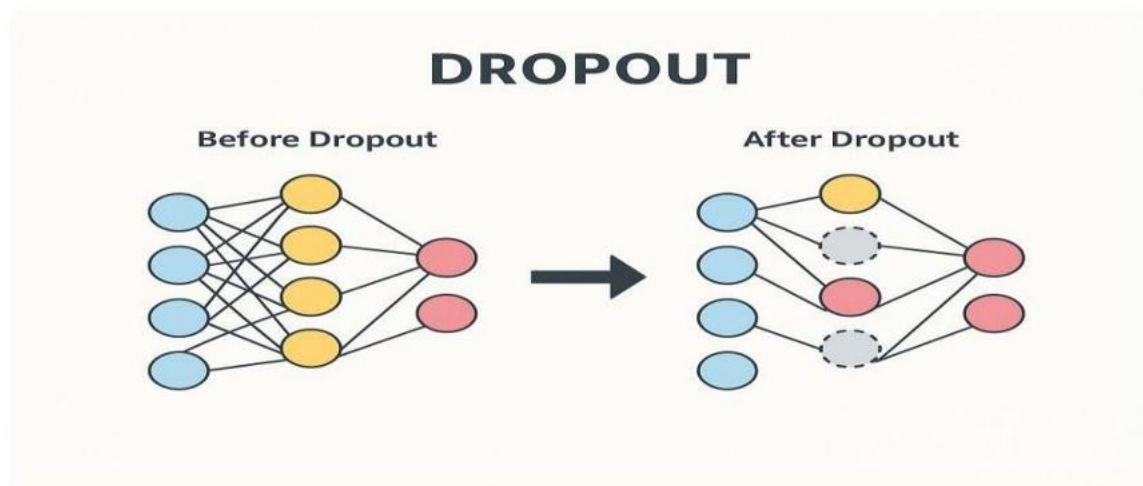
- We apply dropout with a rate of **0.5** on the fully connected layers.

Solution Steps:

1. **Training with Dropout:** Randomly set 50% of hidden neurons to zero during training.
2. **Effect:** Each mini-batch sees a different subnetwork, preventing co-adaptation.
3. **Result:**
 - Training accuracy decreases slightly to **97%** (since some neurons are dropped).
 - Test accuracy improves to **95%** (better generalization).

Final Answer

Using dropout, the network generalizes better, reducing the gap between training and testing accuracy.



11. Adversarial Training

Adversarial Training is a **defense technique in machine learning** where a model is trained not only on normal (clean) data but also on **adversarial examples**—inputs intentionally perturbed to fool the model.

- Goal: Make the model **robust against adversarial attacks** by exposing it to adversarial samples during training.
- It's widely used in **deep learning, computer vision, NLP, and security-sensitive applications**.

In short: *Train with "difficult" or "attacked" examples so the model learns to resist them.*

Advantages of Adversarial Training

1. **Improves Robustness** – Protects against adversarial attacks by learning stable decision boundaries.
2. **Better Generalization** – Models trained with adversarial noise often handle real-world noisy data better.
3. **Security Enhancement** – Critical for AI systems in healthcare, finance, and autonomous vehicles.
4. **Stronger Defenses** – Makes models more resistant to gradient-based attacks.
5. **Simple Concept** – Just augment training with adversarial samples.

Disadvantages of Adversarial Training

1. **High Computational Cost** – Generating adversarial examples during training is expensive.
2. **Reduced Accuracy on Clean Data** – Sometimes the model becomes too focused on adversarial robustness and loses accuracy on normal inputs.
3. **Not Foolproof** – New/stronger attacks can still bypass it.
4. **Complex Implementation** – Needs careful tuning of attack strength (ϵ) and defense.
5. **Longer Training Time** – Because each batch must include adversarial examples.

Real-World Examples of Adversarial Training

1. **Autonomous Vehicles** – Protects self-driving cars from adversarial stop-sign attacks (adding stickers that confuse vision models).
2. **Facial Recognition Systems** – Defends against adversarial attacks where attackers use perturbed images to bypass authentication.
3. **Medical Imaging** – Prevents adversarial noise from causing wrong diagnosis in X-rays or MRIs.
4. **Spam & Fraud Detection** – Makes classifiers resistant to modified malicious inputs (e.g., spam words hidden in emails).
5. **Voice Assistants (Alexa, Siri, Google Assistant)** – Protects against adversarial audio signals (hidden voice commands).

Example Problem (Adversarial Training in Practice)

Problem:

Suppose you train a classifier on MNIST (handwritten digits). Without adversarial training:

- Clean test accuracy = **98%**
- Accuracy on adversarial samples (FGSM attack with $\epsilon=0.2$) = **25%**

You now apply **adversarial training** using FGSM adversarial examples. After retraining:

- Clean test accuracy = **96%**
- Accuracy on adversarial samples = **85%**

Solution Explanation:

- The model sacrificed a small drop in clean accuracy (**98% → 96%**)
- But gained significant robustness against adversarial inputs (**25% → 85%**)
- This shows the trade-off: **slightly lower normal accuracy but much stronger defense.**

12. Deep model challenge in neural network optimization :

A **deep model challenge in neural network optimization** refers to the set of difficulties that arise when training very deep neural networks. As networks get deeper, they become more powerful in theory but also much harder to optimize in practice. Let me explain in detail:

Challenges in Deep Neural Network Optimization

1. Vanishing and Exploding Gradients

- In very deep networks, gradients (error signals during backpropagation) can shrink to near zero (**vanishing**) or grow extremely large (**exploding**) as they pass through many layers.
- This makes weight updates ineffective (learning stops) or unstable (weights diverge).

2. Overfitting

- Deeper models with millions of parameters can memorize the training data rather than generalizing.
- This leads to high training accuracy but poor test accuracy.

3. High Computational Cost

- Deep networks require **large memory** and **long training time**.
- Training on massive datasets needs GPUs/TPUs, which is costly.

4. Difficult Optimization Landscape

- The loss function in deep models has many **local minima** and **saddle points**.
- Gradient-based methods can get stuck in plateaus or poor minima.

5. Hyperparameter Sensitivity

- Deep networks depend on many hyperparameters (learning rate, batch size, dropout, weight initialization).
- Choosing the wrong values can cause non-convergence or poor accuracy.

6. Internal Covariate Shift

- As layers get deeper, the input distribution of each layer keeps changing.
- This slows training and requires techniques like **batch normalization**.

7. Gradient Descent Convergence

- Standard gradient descent struggles with deep models.
- Requires advanced optimizers like **Adam**, **RMSProp**, **Adagrad**.

8. Vanishing Feature Representation

- In very deep models, important information from early layers may get lost.
- Solutions include **skip connections (ResNet)** and **dense connections (DenseNet)**.

Solutions to Deep Model Challenges

- **Activation functions** like ReLU, Leaky ReLU, GELU (reduce vanishing gradients).
- **Batch Normalization / Layer Normalization** (reduce covariate shift).
- **Residual Connections** (help gradient flow in ResNet).
- **Dropout, Early Stopping, Data Augmentation** (reduce overfitting).
- **Advanced Optimizers (Adam, RMSProp)** (better convergence).
- **Weight Initialization techniques (Xavier, He, etc.)** (avoid exploding/vanishing).

Real-Time Example

- **Self-driving cars** use deep CNNs for object detection. Training a model like **YOLO** or **ResNet-50** involves optimizing millions of parameters.
 - If gradients vanish, the car might fail to detect pedestrians properly.
 - If overfitting occurs, the model may only recognize pedestrians from training images but fail in real-world lighting/weather conditions.

Example Problem

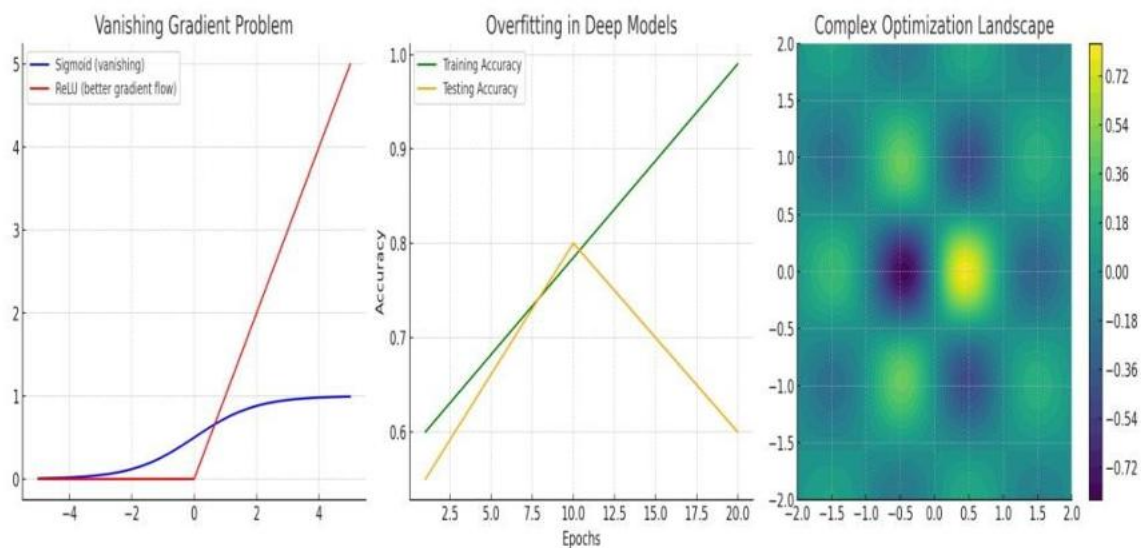
Suppose you train a **20-layer neural network** with sigmoid activation. During training, the loss stops decreasing, and accuracy remains very low even after 50 epochs.

Reason:

- Sigmoid causes **vanishing gradients** in deep networks.
- Gradients approach zero for large positive or negative inputs, making weight updates ineffective.

Solution:

- Replace sigmoid with **ReLU**.
- Add **batch normalization**.
- Use **residual connections** (skip layers).



1. **Vanishing Gradient Problem** – Sigmoid activation shrinks gradients, while ReLU helps gradients flow.
2. **Overfitting** – Training accuracy increases, but test accuracy drops after some epochs.
3. **Optimization Landscape** – Deep models face highly non-convex surfaces with many local minima and saddle points.

Optimization strategies and meta-algorithms are central ideas in **machine learning, deep learning, and artificial intelligence** because they determine **how efficiently and effectively a model learns** from data. Let's go step by step with clear explanations.

1. Optimization Strategies

Optimization strategies and meta-algorithms are central ideas in **machine learning, deep learning, and artificial intelligence** because they determine **how efficiently and effectively a model learns** from data. Let's go step by step with clear explanations.

Optimization strategies are methods used to minimize or maximize an **objective function (loss function or cost function)**. In deep learning, this usually means minimizing the error between predicted and true values.

Common Optimization Strategies:

1. **Gradient Descent (GD)**
 - Iteratively updates model parameters in the opposite direction of the gradient.
 - Types:
 - **Batch Gradient Descent** – uses all data at once.
 - **Stochastic Gradient Descent (SGD)** – updates per sample (faster, noisier).
 - **Mini-Batch Gradient Descent** – compromise between batch and SGD.
2. **Momentum**

- Accelerates gradient descent by adding a fraction of the previous update, avoiding oscillations.
- 3. **Nesterov Accelerated Gradient (NAG)**
 - A lookahead version of momentum for faster convergence.
- 4. **Adaptive Methods**
 - **Adagrad** – adjusts learning rates for each parameter based on past gradients.
 - **RMSProp** – fixes Adagrad's decaying learning rate problem.
 - **Adam** – combines Momentum + RMSProp (most popular today).
- 5. **Learning Rate Scheduling**
 - Adjusts learning rate dynamically (e.g., step decay, cosine annealing, warm restarts).
- 6. **Second-order Methods**
 - Use curvature (Hessian matrix) for optimization (e.g., Newton's method, L-BFGS).
 - Rare in deep learning due to high computation cost.

2. Meta-Algorithms

Meta-algorithms are **higher-level strategies** that **control or combine optimization or learning algorithms** to improve performance. They do not learn directly but **guide or enhance the learning process**.

Common Meta-Algorithms:

1. **Ensemble Learning**
 - Combines multiple models for better accuracy.
 - Techniques: Bagging (Random Forests), Boosting (XGBoost, AdaBoost), Stacking.
2. **Metaheuristic Optimization**
 - Algorithms inspired by nature/evolution for global optimization:
 - Genetic Algorithms (GA)
 - Particle Swarm Optimization (PSO)
 - Simulated Annealing (SA)
 - Ant Colony Optimization (ACO)
3. **Hyperparameter Optimization**
 - Finds best hyperparameters for models.
 - Techniques: Grid Search, Random Search, Bayesian Optimization, Hyperband.
4. **AutoML & Meta-Learning**
 - **AutoML**: Automates model selection, feature engineering, hyperparameter tuning.
 - **Meta-Learning ("Learning to Learn")**: Uses experience from previous tasks to speed up learning on new tasks.
5. **Boosting Meta-Algorithm**
 - Converts weak learners into strong ones by focusing on misclassified samples (e.g., AdaBoost, Gradient Boosting).
6. **Bagging Meta-Algorithm**
 - Reduces variance by training models on bootstrapped datasets (Random Forest).

3. Key Differences

Aspect	Optimization Strategies	Meta-Algorithms
Purpose	Minimize loss / maximize objective	Improve, combine, or guide learning
Level	Works at parameter update level	Works at model/algorithm level
Examples	SGD, Adam, RMSProp	Bagging, Boosting, Hyperparameter tuning

4. Real-Time Examples

- **Optimization Strategy:** Training a neural network with **Adam optimizer** to classify handwritten digits (MNIST).
- **Meta-Algorithm:** Using **Random Forest (bagging)** or **XGBoost (boosting)** to improve classification accuracy in fraud detection.

5. Example Problem

Suppose we want to **train a deep learning model for house price prediction**:

- **Without Optimization Strategies:** If we use only plain gradient descent with a fixed learning rate, training may be slow or stuck in local minima.
- **With Optimization Strategy:** Using Adam with learning rate scheduling speeds up convergence and reduces error.
- **With Meta-Algorithm:** Instead of one model, we use **stacking** (ensemble of multiple regressors) or **Bayesian Optimization** to tune hyperparameters automatically, achieving much higher accuracy.

13. Optimization: Basic Algorithms

Optimization in deep learning is the process of minimizing (or maximizing) a loss (or objective) function by updating the model parameters (weights and biases).

The most common approach is **gradient-based optimization**, where we use derivatives to guide the parameter updates.

1. Gradient Descent (GD)

- **Definition:** Updates parameters in the opposite direction of the gradient of the loss function.
- **Update Rule:**

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta_t)$$

where

- θ = model parameters,
- η = learning rate,
- $J(\theta)$ = loss function.
- **Advantages:** Simple, mathematically grounded.
- **Disadvantages:** Requires full dataset → computationally expensive for large datasets.

2. Stochastic Gradient Descent (SGD)

- **Definition:** Instead of using the whole dataset, it updates parameters using one sample at a time.
- **Update Rule:**

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta; x_i, y_i)$$

- **Advantages:** Faster, works well for large datasets.
- **Disadvantages:** Updates are noisy, may oscillate around minima.

3. Mini-Batch Gradient Descent

- **Definition:** Uses a small batch of data instead of the full dataset or a single sample.
- **Update Rule:** Same as SGD, but gradient computed on a batch.
- **Advantages:** Balances efficiency and stability.
- **Disadvantages:** Needs good batch size selection.

4. Momentum

- **Definition:** Accelerates gradient descent by adding a fraction of the previous update to the current one.
- **Update Rule:**

$$v_t = \beta v_{t-1} + \eta \nabla_{\theta} J(\theta)$$

$$\theta_{t+1} = \theta_t - v_t$$

- **Advantages:** Reduces oscillations, speeds convergence.
- **Disadvantages:** Needs tuning of momentum factor β .

5. Nesterov Accelerated Gradient (NAG)

- **Definition:** A refined momentum method that looks ahead before updating.
- **Update Rule:**

$$v_t = \beta v_{t-1} + \eta \nabla_{\theta} J(\theta - \beta v_{t-1})$$

$$\theta_{t+1} = \theta_t - v_t$$

- **Advantages:** Better convergence than standard momentum.
- **Disadvantages:** Slightly more complex.

6. Adagrad

- **Definition:** Uses adaptive learning rates based on past gradients (good for sparse data).
- **Update Rule:**

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_{\theta} J(\theta)$$

where G_t is the sum of past squared gradients.

- **Advantages:** Good for sparse data.
- **Disadvantages:** Learning rate shrinks too much over time.

7. RMSProp

- **Definition:** Fixes Adagrad's shrinking learning rate by using exponential moving averages of squared gradients.
- **Update Rule:**

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta)(\nabla_{\theta} J(\theta))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_{\theta} J(\theta)$$

- **Advantages:** Works well for non-stationary problems.
- **Disadvantages:** Needs tuning of decay rate.

8. Adam (Adaptive Moment Estimation)

- **Definition:** Combines **Momentum** + **RMSProp**, most widely used optimizer.
- **Update Rule:**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} J(\theta)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_{\theta} J(\theta))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

- **Advantages:** Fast convergence, widely applicable.
- **Disadvantages:** Can generalize poorly in some cases.

14. Optimization strategies and Meta algorithms

Optimization strategies and meta-algorithms are central ideas in **machine learning, deep learning, and artificial intelligence** because they determine **how efficiently and effectively a model learns** from data. Let's go step by step with clear explanations.

1. Optimization Strategies

Optimization strategies are methods used to minimize or maximize an **objective function (loss function or cost function)**. In deep learning, this usually means minimizing the error between predicted and true values.

Common Optimization Strategies:

1. Gradient Descent (GD)

- Iteratively updates model parameters in the opposite direction of the gradient.
- Types:
 - **Batch Gradient Descent** – uses all data at once.
 - **Stochastic Gradient Descent (SGD)** – updates per sample (faster, noisier).
 - **Mini-Batch Gradient Descent** – compromise between batch and SGD.

2. Momentum

- Accelerates gradient descent by adding a fraction of the previous update, avoiding oscillations.

3. Nesterov Accelerated Gradient (NAG)

- A lookahead version of momentum for faster convergence.

4. Adaptive Methods

- **Adagrad** – adjusts learning rates for each parameter based on past gradients.
- **RMSProp** – fixes Adagrad's decaying learning rate problem.
- **Adam** – combines Momentum + RMSProp (most popular today).

5. Learning Rate Scheduling

- Adjusts learning rate dynamically (e.g., step decay, cosine annealing, warm restarts).

6. Second-order Methods

- Use curvature (Hessian matrix) for optimization (e.g., Newton's method, L-BFGS).
- Rare in deep learning due to high computation cost.

2. Meta-Algorithms

Meta-algorithms are **higher-level strategies** that **control or combine optimization or learning algorithms** to improve performance. They do not learn directly but **guide or enhance the learning process**.

Common Meta-Algorithms:

1. Ensemble Learning

- Combines multiple models for better accuracy.
- Techniques: Bagging (Random Forests), Boosting (XGBoost, AdaBoost), Stacking.

2. Metaheuristic Optimization

- Algorithms inspired by nature/evolution for global optimization:
 - Genetic Algorithms (GA)
 - Particle Swarm Optimization (PSO)
 - Simulated Annealing (SA)
 - Ant Colony Optimization (ACO)

3. Hyperparameter Optimization

- Finds best hyperparameters for models.
- Techniques: Grid Search, Random Search, Bayesian Optimization, Hyperband.

4. AutoML & Meta-Learning

- **AutoML:** Automates model selection, feature engineering, hyperparameter tuning.

- **Meta-Learning ("Learning to Learn"):** Uses experience from previous tasks to speed up learning on new tasks.

5. Boosting Meta-Algorithm

- Converts weak learners into strong ones by focusing on misclassified samples (e.g., AdaBoost, Gradient Boosting).

6. Bagging Meta-Algorithm

- Reduces variance by training models on bootstrapped datasets (Random Forest).

3. Key Differences

Aspect	Optimization Strategies	Meta-Algorithms
Purpose	Minimize loss / maximize objective	Improve, combine, or guide learning
Level	Works at parameter update level	Works at model/algorithm level
Examples	SGD, Adam, RMSProp	Bagging, Boosting, Hyperparameter tuning

4. Real-Time Examples

- **Optimization Strategy:** Training a neural network with **Adam optimizer** to classify handwritten digits (MNIST).
- **Meta-Algorithm:** Using **Random Forest (bagging)** or **XGBoost (boosting)** to improve classification accuracy in fraud detection.

5. Example Problem

Suppose we want to **train a deep learning model for house price prediction**:

- **Without Optimization Strategies:** If we use only plain gradient descent with a fixed learning rate, training may be slow or stuck in local minima.
- **With Optimization Strategy:** Using Adam with learning rate scheduling speeds up convergence and reduces error.

- **With Meta-Algorithm:** Instead of one model, we use **stacking** (ensemble of multiple regressors) or **Bayesian Optimization** to tune hyperparameters automatically, achieving much higher accuracy.