UNIT-1

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.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

What Is Deep Learning

- ✓ Deep learning is a **subset of machine learning** that uses **multi-layered artificial neural networks** to learn complex patterns from large datasets. Inspired by the human brain, these models automatically extract features from raw data.
- ✓ It works technically in the same way as machine learning does, but with different capabilities and approaches.
- ✓ Deep learning models are capable enough to focus on the accurate features themselves by requiring a little guidance from the programmer.
- ✓ Deep learning is implemented with the help of Neural Networks, and the idea behind the motivation of neural network is the biological neurons, which is nothing but a brain cell

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Architectures

Shallow neural network

The Shallow neural network has only **one hidden layer between the input and output.**

Deep Neural Networks

It is a neural network that incorporates the complexity of a certain level, which means **several numbers of hidden layers** are encompassed in **between the input and output layers.** They are highly proficient on model and process non-linear associations.

Historical Trends in Deep Learning

1950s–1980s: Foundations and Early Neural Models

- ✓ **McCulloch-Pitts Neuron (1943):** Theoretical model of an artificial neuron.
- ✓ **Perceptron (Rosenblatt, 1958):** First algorithm for supervised learning of binary classifiers.
- ✓ **Limitations Identified (Minsky & Papert, 1969):** Showed that perceptrons couldn't solve non-linearly separable problems (e.g., XOR), leading to reduced interest.

1980s–1990s: Revival through Backpropagation

- ✓ **Backpropagation Algorithm (Rumelhart, Hinton, Williams, 1986):** Enabled training of multilayer neural networks (MLPs).
- ✓ **Multilayer Perceptron's (MLPs):** Introduced the ability to model complex, non-linear functions.
- ✓ Interest remained mostly academic due to computational limitations and small datasets.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2000s: Rise of Alternative ML Algorithms

- ✓ **Support Vector Machines (SVMs), Decision Trees, Random Forests, Boosting** became more popular due to better performance and interpretability.
- ✓ **Kernel Methods** addressed non-linear problems more efficiently.
- ✓ **Neural networks fell out of favor** in most applications due to overfitting and training difficulties.

2010s–Present: Deep Learning Revolution

Driven by three main factors:

- ✓ **Increased Computational Power**

  GPUs enabled fast matrix/tensor operations.

  GPU stands for **Graphics Processing Unit** — originally designed to accelerate image rendering and gaming graphics. But today, **GPUs are the backbone of modern deep learning** because they can process large blocks of data in parallel, much faster than traditional CPUs.

Linear Algebra

Scalar (0D)

✓ A **scalar** is a single number. It has **magnitude only**, no direction.

Example:

✓ Temperature = **36.6°C**

✓ Age = **25**

✓ Price = **₹499**

Notation

✓ x=5

Real-World Analogy

✓ The **temperature on a thermometer** is a scalar — it's just a number.

2. Vector (1D)

- ✓ A **vector** is an ordered **1D array** of numbers. It has **both magnitude and direction**.
- ✓ Velocity vector: [60,45] → 60 km/h East, 45 km/h North
- ✓ Student scores in 3 subjects: [85,78,92]
- ✓ **Notation:**

$$\vec{v}=[85,78,92]$$

Real-World Analogy

A **GPS direction vector** showing movement along x and y axes.

3. Matrix (2D)

✓ A **matrix** is a **2D array** of numbers arranged in rows and columns. It can represent **linear transformations**..

Example:

✓ Student scores for 3 students in 4 subjects:

$$\text{Scores} = \begin{bmatrix} 85 & 90 & 88 & 92 \\ 78 & 80 & 75 & 85 \\ 92 & 88 & 95 & 94 \end{bmatrix}$$

4. Tensor (3D or more)

✓ A **tensor** is a **multi-dimensional array**. Scalars (0D), vectors (1D), and matrices (2D) are all special cases of tensors.

Example:

✓ **Color image** = 3D tensor (Height × Width × Channels(RGB)

✓ A 100×100 image with RGB channels:

$$\textbf{\textit{Image Tensor}} = R^{100 \times 100 \times 3}$$

✓ **Video** = 4D tensor (Frames × Height × Width × Channels)

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2.2 Multiplying Matrices and Vectors

- ✓ Matrix and vector multiplication is a **core operation** in linear algebra and is used extensively in **machine learning**, **deep learning**, **computer graphics**, and **data transformations**.
- ✓ **1. Matrix × Vector Multiplication**

- ✓ If you have a matrix A of shape $m \times n$ and a vector x of size n, the product $A \cdot x$ results in a new vector y of size m

$$y = A \cdot x$$

- ✓ Dimensions: If $A = [m \times n]$ and $x = [n]$, then $y = [m]$

Let

$$A = \begin{bmatrix} 2 & 3 \\ 4 & 1 \\ 5 & 0 \end{bmatrix}, \quad x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Now, multiply:

$$A \cdot x = \begin{bmatrix} (2)(1) + (3)(2) \\ (4)(1) + (1)(2) \\ (5)(1) + (0)(2) \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \\ 5 \end{bmatrix}$$

2. Matrix × Matrix Multiplication

✓ Two matrices A and B can be multiplied only if the number of columns in A equals the number of rows in B

✓ If $A=[m\times n]$,$B=[n\times p]$,then $C=A\cdot B=[m\times p]$

Let:

$$A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix},\quad B=\begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}$$

$$A\cdot B=\begin{bmatrix}(1)(5)+(2)(7) & (1)(6)+(2)(8)\\ (3)(5)+(4)(7) & (3)(6)+(4)(8)\end{bmatrix}=\begin{bmatrix}19 & 22\\ 43 & 50\end{bmatrix}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Identity and Inverse Matrices

- The **identity matrix** is like the number **1** in matrix math.
- Multiplying any matrix A with the identity matrix **does not change** it:

$$A \cdot I = I \cdot A = A$$

📃 **Example:**

If

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

then

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

and

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

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- The **inverse** of a matrix is like dividing — it "undoes" the effect of a matrix.
- If a matrix A has an inverse A^{-1}, then:

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

> Not all matrices have an inverse. A matrix must be **square** and **non-singular** (i.e., determinant ≠ 0) to be invertible.

📄 **Example:**

Let

$$A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$$

Then the inverse is:

$$A^{-1} = \frac{1}{(4 \cdot 6 - 7 \cdot 2)} \cdot \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \frac{1}{10} \cdot \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

TYPES OF MATRICES

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1) Singleton Matrix

- A matrix that has only one element is called a singleton matrix.
- In this type of matrix number of columns and the number of rows is equal to 1.
- A singleton matrix is represented as $[a]_{1.}$
- Example of Singleton Matrix $[5]_{1\times1}$

2) Null Matrix

- A matrix whose all elements are zero is called a Null Matrix.
- A null matrix is also called a Zero Matrix because its all elements are zero.
- Example of Null Matri $\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}_{3\times3}$

3) Row Matrix

- A matrix that contains only one row and any number of columns is known as a row matrix.
- A row matrix is represented as $[a]_{1\times n}$ where 1 is the number of row and n is the number of columns present in a row matrix.
- Example of Row Matri: $[1 \quad 3 \quad 7]_{1\times3}$

4) Column Matrix

- A matrix that contains only one Column and any number of Rows is known as a row matrix.
- A row matrix is represented as $[a]_{n*1}$ where n is the number of rows and 1 is the number of column.
- Example of Column Matrix $\begin{bmatrix}1\\15\\4\\5\end{bmatrix}_{4\times1}$

5) Horizontal Matrix

- A matrix in which the number of rows is lower than the number of columns is called a Horizontal Matrix. $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}_{2\times4}$
- Example of Horizontal Matrix

6) Vertical Matrix

- A matrix in which the number of rows exceeds the number of columns is called a Vertical Matrix
- Example of Vertical Matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}_{4\times2}$

7) Rectangular Matrix

- A matrix that does not have an equal number of rows and columns is known as a Rectangular Matrix.
- Example of Rectangular Matrix $\begin{bmatrix} 1 & 3 & 7 & 15 \\ 3 & 4 & 6 & 11 \\ 5 & 2 & 9 & 8 \end{bmatrix}_{3\times4}$

8) Square Matrix

- A matrix that has an equal number of rows and an equal number of columns is called a Square Matrix.
- Example of Square Matrix $\begin{bmatrix} 8 & 3 & 2 \\ 6 & 4 & 6 \\ 5 & 7 & 9 \end{bmatrix}_{3\times3}$

9) Diagonal Matrix

- A matrix that has all elements as 0 except diagonal elements is known as a diagonal matrix.
- Example of Diagonal Matrix $\begin{bmatrix} 8 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 9 \end{bmatrix}_{3\times3}$

10) Scalar Matrix

- A diagonal matrix whose all diagonal elements are non-zero and the same is called a Scalar Matrix.
- Scalar Matrix is a kind of diagonal matrix where all diagonal elements are the same.
- Example of Scalar Matrix $\begin{bmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{bmatrix}_{3\times3}$

11) Identity Matrix

- A diagonal matrix where all the diagonal elements are 1 and all non-diagonal elements are 0 is called an Identity Matrix.
- The Identity Matrix is called Unit Matrix.
- The identity matrix or unit matrix always has an equal number of rows and columns.Example of Rectangular Matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{3\times3}$

12) Triangular Matrix

- A square matrix in which the non-zero elements form a triangular below and above the diagonal is called a Triangular Matrix.
- Based on the triangle formed below or above the diagonal, the triangular matrix is classified as:
  - Upper Triangular Matrix
  - Lower Triangular Matrix

Upper Triangular Matrix

- A square matrix in which all the elements below the diagonal are zero and the elements from the diagonal and above are **non-zero elements is called an Upper Triangular Matrix**.
- In an Upper Triangular Matrix, the non-zero elements form a triangular-like shape..

$$\begin{bmatrix} 8 & 5 & 6 \\ 0 & 4 & 7 \\ 0 & 0 & 9 \end{bmatrix}_{3\times3}$$

Lower Triangular Matrix

- A square matrix in which all the elements above the diagonal are zero and the elements from the diagonal and below **are non-zero elements is called a Lower Triangular Matrix**
- In a Lower Triangular Matrix, the non-zero elements form a triangular-like shape from the diagonal and below...

$$\begin{bmatrix} 8 & 0 & 0 \\ 6 & 4 & 0 \\ 5 & 7 & 9 \end{bmatrix}_{3\times3}$$

15) Symmetric Matrix

- A square matrix "A" of any order is defined as a symmetric matrix if the transpose of the matrix is equal to the original matrix itself, i.e., $A^T = A$.

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

16) Skew Symmetric Matrix

- A square matrix "A" of any order is defined as a skew-symmetric matrix if the transpose of the matrix is equal to the negative of the original matrix itself, i.e., $A^T = -A$.

$$\begin{bmatrix} 0 & 3 & 5 \\ -3 & 0 & -2 \\ -5 & 2 & 0 \end{bmatrix}$$

17) Orthogonal Matrix

- A square matrix whose transpose is equal to its inverse is called Orthogonal Matrix. In an Orthogonal Matrix if $A^T = A^{-1}$ then $AA^T = I$ where I is the Identity Matrix.

$$A = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

and $$A^T = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$\Rightarrow A \times A^T = \begin{bmatrix} \cos^2(\theta) + \sin^2(\theta) & \cos(\theta)\sin(\theta) - \cos(\theta)\sin(\theta) \\ \sin(\theta)\cos(\theta) - \cos(\theta)\sin(\theta) & \cos^2(\theta) + \sin^2(\theta) \end{bmatrix} \quad \Rightarrow A \times A^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_{(2\times2)}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

NORMS

- ✓ **L1 Norm (Manhattan Norm)**
- ✓ **L2 Norm (Euclidean Norm)**
- ✓ **L∞ Norm (Infinity Norm)**
- ✓ **p-Norm (p=3)**

✓ L2 Norm (Euclidean Norm)

L2 Norm (Euclidean Norm) is the measure of the straight-line distance of a vector from the origin.
It is calculated by taking the square root of the sum of the squares of all the elements in the vector.
It is commonly used to measure distance or magnitude in machine learning, geometry, and optimization.

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

$$\|\mathbf{x}\|_2 = \sqrt{3^2 + (-4)^2 + 5^2} = \sqrt{9 + 16 + 25} = \sqrt{50} \approx 7.07$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

L1 Norm (Manhattan Norm)

- ✓ is the sum of the absolute values of all elements in a vector.
- ✓ It measures distance by counting total movement along each axis, like navigating a grid of city blocks.
- ✓ It is commonly used in machine learning for feature selection and regularization (like Lasso regression).

$$\|\mathbf{x}\|_1 = |3| + |-4| + |5|$$

$$\|\mathbf{x}\|_1 = 3 + 4 + 5 = 12$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

L∞ Norm (Infinity Norm)

- ✓ is the largest absolute value among the elements of a vector.
- ✓ It measures the maximum impact or magnitude in any single dimension.
- ✓ It is useful in optimization and error analysis when the worst-case difference matters.

$$\|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, |x_3|)$$

$$\|\mathbf{x}\|_\infty = \max(3, 4, 5) = 5$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

EIGEN DECOMPOSITION

What is Eigen Decomposition?

Eigen decomposition is a method of decomposing a square matrix **A** into three components:

$$A = PDP^{-1}$$

Where:

- **P** = matrix of eigenvectors
- **D** = diagonal matrix of eigenvalues
- **P^{-1}** = inverse of P

This decomposition is only possible if **A is a square matrix and has enough linearly independent eigenvectors** (i.e., A is *diagonalizable*).

Eigen Values and Eigen Vectors

An **eigenvector is a non-zero vector** that remains unchanged in direction **when a linear transformation is applied to it**, though it may be scaled by a scalar factor, known as its corresponding eigenvalue.

- Normally, when a linear transformation is applied to a vector, its **length and direction both change**.
- But there are some special vectors that **do not change their direction**; they only **stretch or shrink** by a constant factor.
- These special vectors are called **eigenvectors**.
- To find eigenvectors, we must **first calculate their corresponding eigenvalues**.

Eigenvalues and Eigenvectors

1. An eigenvalue of A is a scalar λ s.t. $det(A - \lambda I) = 0$ *or* $det(\lambda I - A) = 0$.
2. The eigenvectors of A corresponding to λ are the nonzero solutions of $(A - \lambda I)\vec{x} = \vec{0}$ *or* $(\lambda I - A)\vec{x} = \vec{0}$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

How to Find Eigenvectors of a 2 × 2 matrix?

To find the eigenvectors, we first have to compute the eigenvalues using the above-mentioned steps. Let us understand the process by an example.

Example: Find the eigenvalues and eigenvectors of matrix $A = \begin{bmatrix} 5 & 4 \\ 1 & 2 \end{bmatrix}$.

Solution:

Let λ represent the eigenvalue(s) and $\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix}$ represent the eigenvector(s).

Then the characteristic equation is:

$|A - \lambda I| = 0$

$$\begin{vmatrix} 5-\lambda & 4 \\ 1 & 2-\lambda \end{vmatrix} = 0$$

$(5 - \lambda)(2 - \lambda) - (4)(1) = 0$

$10 - 5\lambda - 2\lambda + \lambda^2 - 4 = 0$

$\lambda^2 - 7\lambda + 6 = 0$

$(\lambda - 6)(\lambda - 1) = 0$

$\lambda = 6, \lambda = 1.$

Thus, the eigenvalues are 1 and 6. Let us find the corresponding eigenvector to each eigenvalue in each case.

When $\lambda = 1$:

Substitute $\lambda = 1$ in the equation:

$(A - \lambda I)\,\mathbf{v} = O$

$$\begin{bmatrix} 5-1 & 4 \\ 1 & 2-1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 4 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

To solve this system, we apply the elementary row transformations (Alternatively, we can use Cramer's rule as well) on the coefficient matrix:

Apply $R_2 \to 4R_2 - R_1$,

$$\begin{bmatrix} 4 & 4 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Expanding as equations,

$4x + 4y = 0$

We have one equation in two variables. So assume $y = t$. Then

$4x + 4t = 0 \Rightarrow 4x = -4t \Rightarrow x = -t$

So the solution is, $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -t \\ t \end{bmatrix} = t\begin{bmatrix} -1 \\ 1 \end{bmatrix}$

Thus, the eigenvector is $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$.

When λ = 6:

Substitute λ = 1 in the equation:

$(A - \lambda I)\,\mathbf{v} = O$

$$\begin{bmatrix} 5-6 & 4 \\ 1 & 2-6 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 4 \\ 1 & -4 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

To solve this system, we apply the elementary row transformations (Alternatively, we can find Cramer's rule as well) on the coefficient matrix:

Apply $R_2 \rightarrow R_2 + R_1$,

$$\begin{bmatrix} -1 & 4 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Expanding as equations,

$-x + 4y = 0$

Let $y = t$. Then $-x + 4t = 0 \Rightarrow x = 4t$.

So the solution is, $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4t \\ t \end{bmatrix} = t\begin{bmatrix} 4 \\ 1 \end{bmatrix}$

Thus, the eigenvector is $\begin{bmatrix} 4 \\ 1 \end{bmatrix}$.

Thus, the eigenvectors of the given 2 × 2 matrix are $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 4 \\ 1 \end{bmatrix}$.

How to Find Eigenvectors of a 3 × 3 matrix?

We follow the same steps as above for the 3 × 3 matrix as well. Here is an example.

Example: Find the eigenvectors of 3 × 3 matrix $A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$.

Solution:

Let λ be the eigenvalue and $\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ be the eigenvector of A.

Finding eigenvalues:

The characteristic equation is:

$|A - \lambda I| = 0$

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

Calculating the determinant, we get

$-\lambda^3 + 3\lambda^2 = 0$

$\lambda^2 (-\lambda+3) = 0$

$\lambda = 0, \lambda = 3.$

Thus, the eigenvalues are 0 and 3. Let us find the corresponding eigenvectors.

When λ = 0:

$(A - \lambda I)\,\mathbf{v} = 0$

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

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

Apply $R_2 \rightarrow R_2 - R_1$ and $R_3 \rightarrow R_3 - R_1$,

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

Expanding the above matrix, we get

$x + y + z = 0$

We have only one equation with two unknowns. So let us assume two of the variables to be $y = t_1$ and $z = t_2$. Then the above equation becomes:

$x + t_1 + t_2 = 0 \Rightarrow x = - t_1 - t_2$.

Thus, the eigenvector is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \(\left[\begin{array}{l} - t_1 - t_2 \\ t_1 \\ t_2 \end{array}\right]\) = t_1\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} + t_2\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

Thus, the eigenvectors that correspond to $\lambda = 0$ are $\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$.

When λ = 3:

$(A - \lambda I)\,\mathbf{v} = 0$

$$\begin{bmatrix} 1-3 & 1 & 1 \\ 1 & 1-3 & 1 \\ 1 & 1 & 1-3 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

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

Apply $R_2 \rightarrow 2R_2 + R_1$ and $R_3 \rightarrow 2R_3 + R_1$,

$$\begin{bmatrix} -2 & 1 & 1 \\ 0 & -3 & 3 \\ 0 & 3 & -3 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, apply $R_3 \rightarrow R_3 + R_2$,

$$\begin{bmatrix} -2 & 1 & 1 \\ 0 & -3 & 3 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Expanding the above matrix, we get

$-2x + y + z = 0$

$-3y + 3z = 0$

Let $z = t$. Then

$-3y + 3z = 0 \Rightarrow z = y = t$.

$-2x + y + z = 0 \Rightarrow -2x + t + t = 0 \Rightarrow x = t$.

Thus, the eigenvector is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} t \\ t \\ t \end{bmatrix} = t\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Thus, the eigenvector that correspond to $\lambda = 3$ is $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Thus, the eigenvectors of the given 3 × 3 matrix are $\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$, and $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Singular Value Decomposition

Singular Value Decomposition (SVD) is a fundamental concept in linear algebra and numerical analysis. It decomposes a matrix into three simpler matrices, providing insights into its structure and allowing for various computations and analyses.

Given a matrix A, its Singular Value Decomposition is represented as:

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

Where:

- U is an orthogonal matrix containing the left singular vectors.
- Σ is a diagonal matrix containing the singular values.
- V^T is the transpose of an orthogonal matrix containing the right singular vectors.

$$U = AA^T$$

$$V = A^T A$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Singular Value Decomposition

SVD has numerous applications, including:

1.Dimensionality Reduction: SVD can be used to reduce the dimensionality of data while preserving important information.

2.Matrix Approximation: By truncating the singular values and corresponding singular vectors, you can approximate the original matrix.

3.Pseudoinverse: SVD can be used to compute the pseudoinverse of a matrix, which is useful in solving linear least squares problems.

4.Principal Component Analysis (PCA): SVD is closely related to PCA, where the singular vectors represent the principal components of the data.

5.Image Compression: SVD can be used for compressing images by representing them in terms of their singular vectors and values.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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.**
- **Correlated = two things move together**
- If one **increases**, the other **also increases** → **positively correlated**
- If one **increases**, the other **decreases** → **negatively correlated**
- **Uncorrelated = no relationship**
- **shoe size** and **exam marks**
  - Big shoe size does NOT mean high marks
  - Small shoe size does NOT mean low marks
- So, shoe size and marks are **uncorrelated**.
- it helps us to **remove redundancy**, improve **computational efficiency** and make **data easier to visualize and analyze**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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 simplify the dataset.
- 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.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- Data has **many features**, often **correlated** (related to each other).
- PCA **rotates the data** to look at it from better directions.
- It finds the **direction with maximum spread** → **Principal Component 1 (PC1)**.
- It then finds another direction **at 90°** to PC1 → **Principal Component 2 (PC2)**.
- These new directions are **uncorrelated**.
- Keeping only the **top few components** removes noise and reduces dimensions.
- Important information is **kept**, less important information is **discarded**.

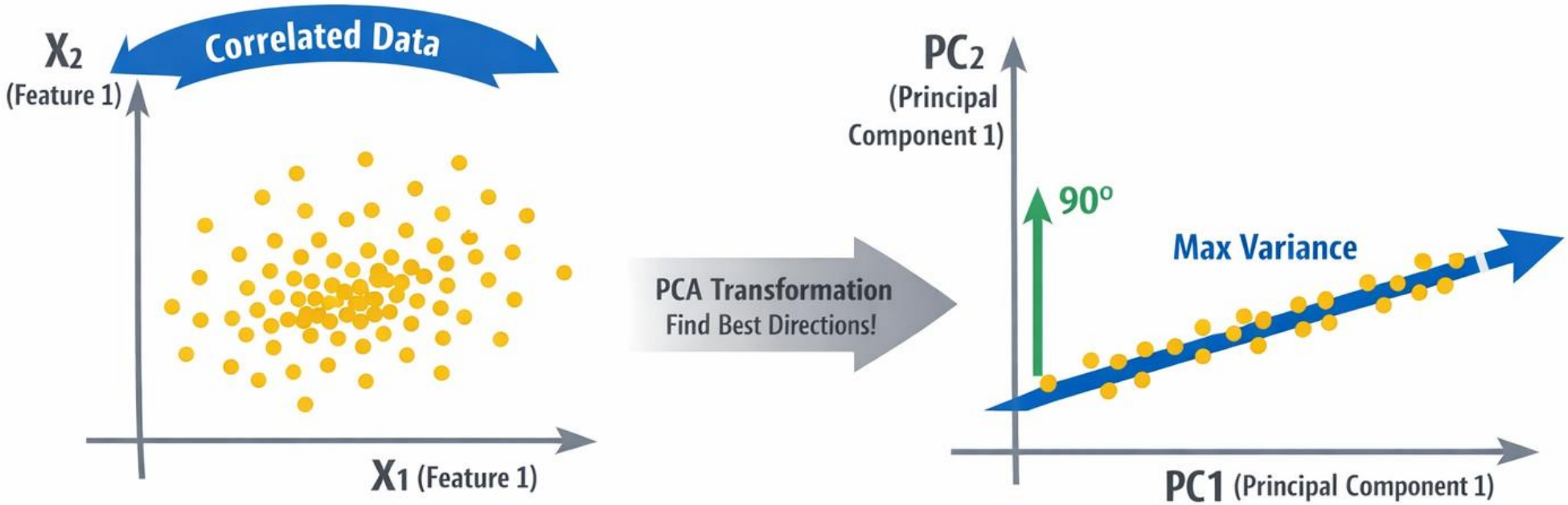
One-line clarity

👉 **PCA keeps the most informative directions and removes redundant ones.**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

BEFORE PCA

AFTER PCA



- Many Correlated Features
- Redundant Information

- Uncorrelated Components
- Reduced Dimensions

Key Patterns, Less Noise!

step 1: Standardize the Data

Different features may have different units and scales (for example, salary vs. age). To ensure that all features contribute equally, PCA first **standardizes the data** so that each feature has:

Mean = 0

Standard deviation = 1

The standardization is done using the Z-score formula:

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

Where:

X= original feature value

$\mu = \{\mu_1, \mu_2, \ldots, \mu_m\}$is the mean of each feature

$\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$is the standard deviation of each feature

This step removes bias caused by different scales.

step 2: Calculate the Covariance Matrix

After standardization, PCA computes the **covariance matrix** to understand how features vary together.

The covariance between two features x_1 and x_2 is:

Where:

x_1, x_2 are the mean values of features

n is the number of data points

Covariance values:

Positive → features increase together

Negative → one increases, the other decreases

Zero → no relationship

Step 3: Find the Principal Components

PCA identifies **new axes (directions)** along which the data has maximum variance:

PC1: Direction of maximum variance

PC2: Next highest variance, perpendicular to PC1

And so on

These directions are obtained from the **eigenvectors** of the covariance matrix, and their importance is given by **eigenvalues**.

For a matrix A:

$$AX = \lambda X$$

This means:

Eigenvectors define **important directions**

Eigenvalues indicate **how much variance** each direction captures

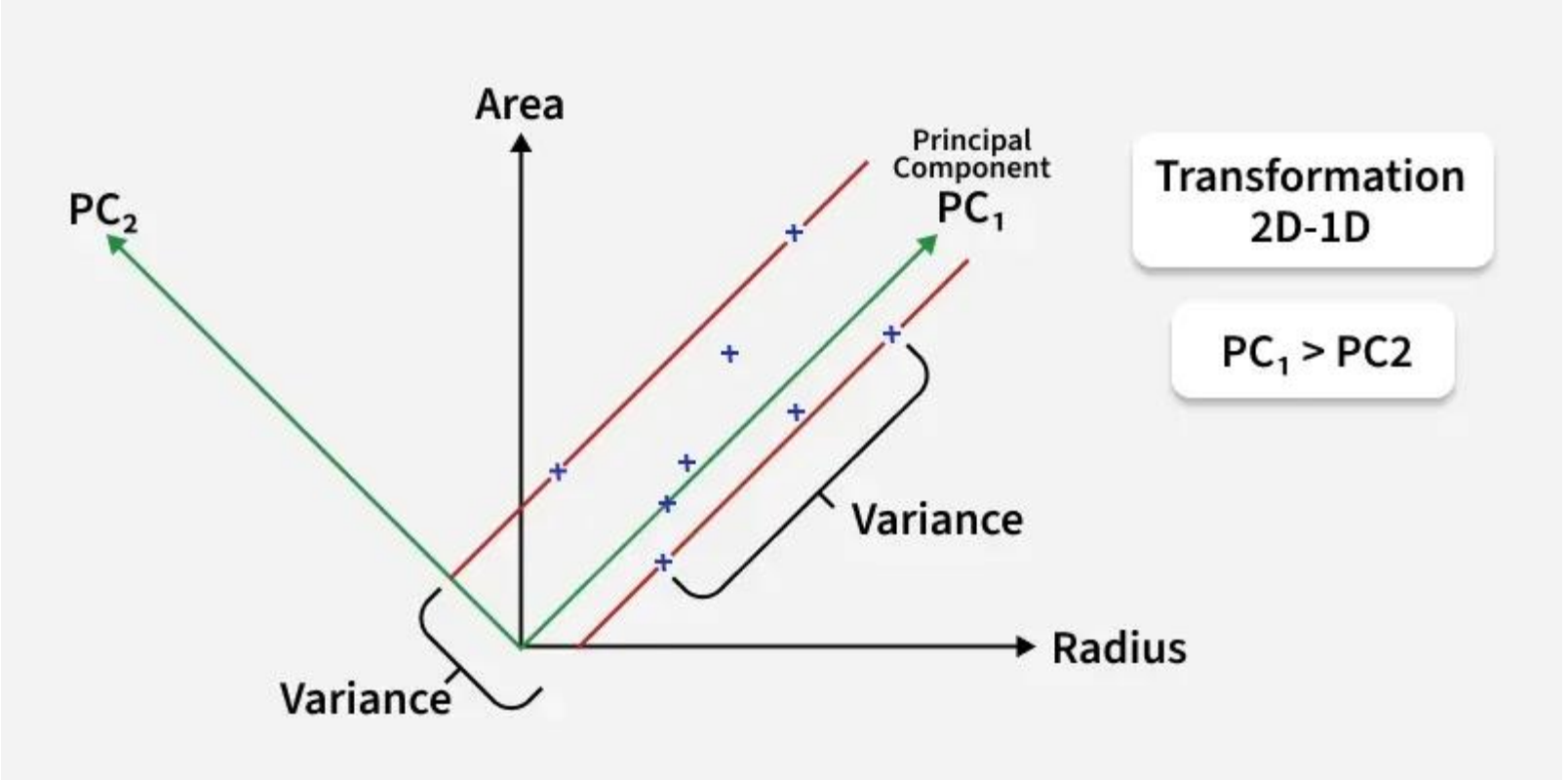
step 4: Select Top Components and Transform Data

Eigenvalues are sorted in **descending order**

The top **k components** that capture most variance (e.g., 95%) are selected

The original data is **projected onto these components**

This reduces the number of dimensions while preserving important patterns.



- In the above image the original dataset has two features "Radius" and "Area" represented by the black axes. PCA identifies two new directions: PC_1 and PC_2 which are the principal components.
- These new axes are rotated versions of the original ones. PC_1 captures the maximum variance in the data meaning it holds the most information while PC_2 captures the remaining variance and is perpendicular to PC_1.
- The spread of data is much wider along PC_1 than along PC_2. This is why PC_1 is chosen for dimensionality reduction. By projecting the data points (blue crosses) onto PC_1 we effectively transform the 2D data into 1D and retain most of the important structure and patterns.

Part-2

Probability and Information Theory: Random Variables, Probability Distributions, Marginal Probability, Conditional Probability, Expectation, Variance and Covariance, Bayes‘ Rule, Information Theory.

Probability and Information Theory

- Probability theory is a mathematical framework for representing uncertain statements.
- In artificial intelligence applications, we use probability theory in two major ways.

1. The laws of probability tell us how AI systems should reason, so we design our algorithms to compute or approximate various expressions derived using probability theory.

2. we can use probability and statistics to theoretically analyze the behavior of proposed AI systems.

Probability theory is a fundamental tool of many disciplines of science and engineering.

Why Probability?

- Many branches of computer science deal mostly with entities that are entirely deterministic and certain. A programmer can usually safely assume that a CPU will execute each machine instruction flawlessly.
- It can be surprising that machine learning makes heavy use of probability theory. This is because machine learning must always deal with uncertain quantities, and sometimes may also need to deal with stochastic (non-deterministic) quantities.
- Uncertainty and stochasticity can arise from many sources.

Introduction to Random Variable

Experiment

- ✓ Is any Physical action or process that is observed and the result is noted

Random Experiment

- ✓ An Experiment in which **all possible outcomes** are known and the **exact outcomes cannot be predicted in advance is called "Random Experiment".**
- ✓ Eg1 : Tossing a coin is an example of Random experiment because in this experiment all possible outcomes {H,T} are known but exact outcome cannot be predicted.
- ✓ Eg2: Boiling Water – Outcome is Predicted i.e it gets Evaporated . So its not an Random experiment

Sample Space

- ✓ Set of all Possible outcomes of an Experiment is called “Sample Space”.
- ✓ Examples :
  1) Sample Space of “Tossing a Coin experiment is “ **S = {H,T}**
  2) Sample Space of “Tossing 2 Coins Experiment is “ **S = {HH,HT,TH,TT}**
  3) Sample Space of “ Rolling a Dice Experiment is “ **S = {1,2,3,4,5,6}**

Random Variable

Random Variable

- ✓ A random variable is a function that assigns a numerical value to each possible outcome of a random experiment.
- ✓ Random Variables are denoted by capital Letters. Eg : X, Y etc..
- ✓ Example :
  - ✓ **Experiment : "Tossing 2 Coins "**
  - ✓ **Sample Space : S = {HH,HT,TH,TT}**
  - ✓ **Random Variable : X { no. of Heads }**
  - ✓ **So, X = {2,1,1,0} (Numerical Values assigned to HH ,HT, TH and TT)**

Types of Random Variable

Types of Random Variable

Random variable are classified into 2 Types

1) **Discrete Random variable**
2) **Continuous Random variable**

Types of Random Variable

1) Discrete Random Variable :

- ✓ **If a Random variable takes only a finite or a countable number of values , it is called Discrete Random Variable.**
- ✓ **Example : when 2 coins are tossed , the sample space S = {HH,HT,TH,TT} and if the function is No.of Heads, then the variable can take finite or countable values like 2,1,0 and such variable is called Discrete Random variable.**

2) Continuous Random Variable :

- ✓ **A Random Variable X which can take any Value between certain Interval is called Continuous Random Variable.**
- ✓ **Example : The Height of students in a particular class lies between 4 to 6 feet. We write this as**

$$X = \{x | 4 \leq x \leq 6\}$$

Probability Distribution of a Random Variable

1) Probability Distribution

- ✓ **Distribution of Probabilities of each Outcome of the random variable is called Probability Distribution**
- ✓ **Example :**

Experiment : 2 coins are tossed ,

Sample Space S = {HH,HT,TH,TT}

function : No.of Heads, then

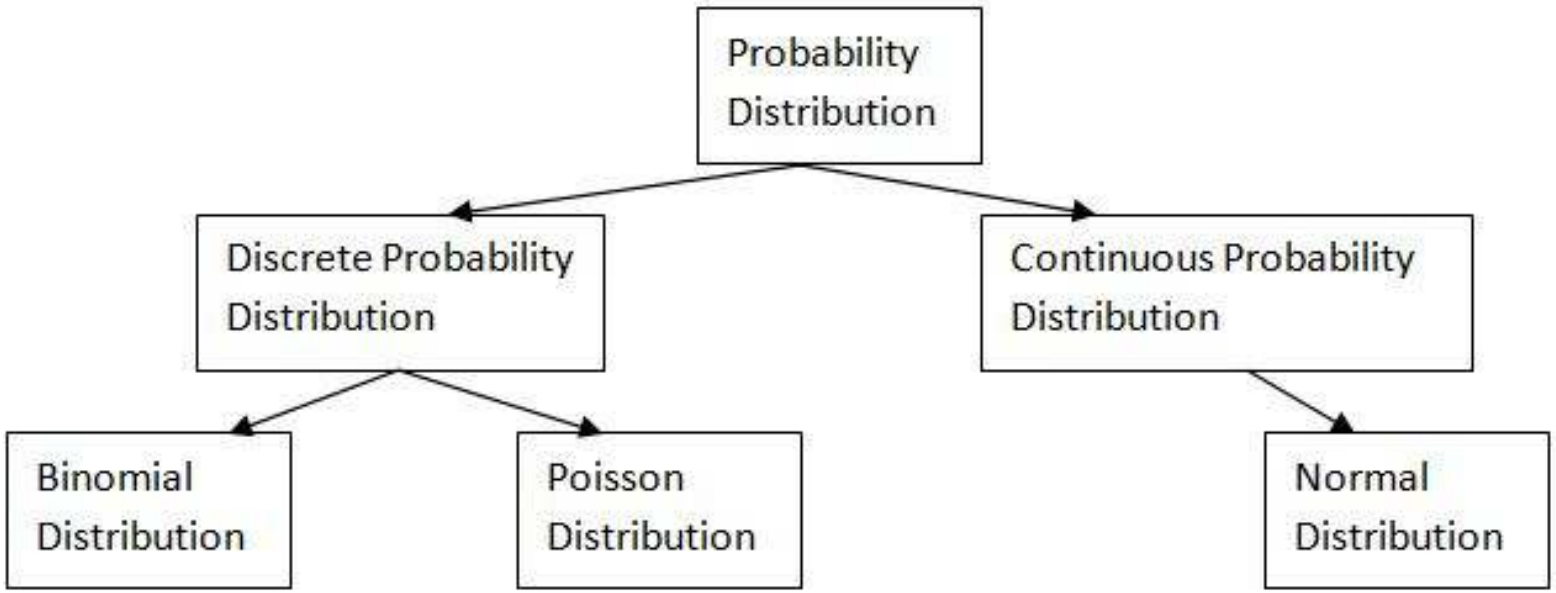
the Discrete Random variable X = {2,1,0}

Outcome of the Variable Corresponding Probabilities

X=x	0	1	2
f(x) = P(x=x)	1/4	2/4	1/4

Probability Distribution of the Discrete Random variable and f(X) is called Probability Mass Function

Types of Probability Distribution



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Discrete Probability Distribution (or) Probability Mass Function

- Probability Distribution of a Discrete Random Variable is called **Discrete Probability Distribution** and
- The **Discrete Probability Distribution Function** P(X=x) or Pi is called **Probability Mass Function**
- The Properties of Probability Mass Function are

i. $P_i \geq 0$ *for all i, i.e* $P_i's$ *are all non – negative.*

ii. $\sum P_i = P_1 + P_2 + \cdots\cdots\cdots\cdots\cdots\cdots + P_n = 1$ *i.e the total probability is one*

1) TO check the given function is Probability Mass Function or Not.

Solution : Experiment = Tossing 2 coins

Sample space = {HH, HT, TH, TT}

X = {No. of Heads}

X = {2, 1, 1, 0}

↓

Discrete Random Variable because it takes finite (or) countable values.

The Probability Distribution of the outcomes of the Random variable are

x	0	1	2
P(X=x) (or) f(x) or P_i	1/2 (Prob of getting 0 Heads out of 4 outcomes)	2/4	1/4

is a PMF

TO check given function is PMF or not, the f(x) should satisfy 2 conditions

① Each $P_i > 0$

② Sum of all probabilities should be Equal to 1

In our Example, all three P_i are greater than zero and

1/4 + 2/4 + 1/4 = 1. So f(x) is a PMF

Problem 2:

The Probabilities that a customer will buy 1,2,3,4 or 5 items in a grocery store are 3/10,1/10,1/10,2/10,3/10 respectively. What is the average number of items that a customer will buy

Solution :

The Probability Distribution Table for the Above problem is

X	P(x)	x.Px)
1	3/10	3/10
2	1/10	2/10
3	1/10	3/10
4	2/10	8/10
5	3/10	15/10
E x.P(X) (or) E(x)		**31/10 = 3.1**

The Average Number of Items the customer can buy is 3.1

Problem 3:

The Following Probability Distribution tells us the Probability that a certain Soccer Team Scores certain number of goals in a given time. **Find Mean, Variance and SD**

Goals x	Probabilities P(x)
0	0.18
1	0.34
2	0.34
3	0.11
4	0.02

Solution :

The Probability Distribution Table for the Above problem is

Goals x	Probabilities P(x)	x.P(x)	X^2	$X^2.P(X)$
0	0.18	0	0	0
1	0.34	0.34	1	0.34
2	0.34	0.68	4	1.36
3	0.11	0.33	9	0.99
4	0.02	0.08	16	0.32
$E(X) = \sum x.P(X) = 1.43$			$E(X^2) = \sum X^2.P(X) = 3.01$	

$$\text{Mean } \mu = E(X) = 1.43$$

$$\text{Variance } \sigma^2 = E(X^2) - (E(X))^2$$

$$= 3.01 - (1.43)^2$$

$$= 0.96$$

Step 1: Mean (Expected Value)

$$E(X) = \sum xP(x)$$

x	P(x)	x·P(x)
0	0.18	0
1	0.34	0.34
2	0.34	0.68
3	0.11	0.33
4	0.02	0.08

$$E(X) = 0 + 0.34 + 0.68 + 0.33 + 0.08 = \boxed{1.43}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Step 2: Find $E(X^2)$

$$E(X^2) = \sum x^2 P(x)$$

x	x^2	$x^2P(x)$
0	0	0
1	1	0.34
2	4	1.36
3	9	0.99
4	16	0.32

$$E(X^2) = 0 + 0.34 + 1.36 + 0.99 + 0.32 = \boxed{3.01}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Step 3: Variance

$$\text{Var}(X) = E(X^2) - [E(X)]^2$$

$$\text{Var}(X) = 3.01 - (1.43)^2$$

$$\text{Var}(X) = 3.01 - 2.05 = \boxed{0.96}$$

Step 4: Standard Deviation

$$\text{SD} = \sqrt{\text{Var}(X)} = \sqrt{0.96} \approx \boxed{0.98}$$

Binomial Distribution in Probability

Binomial Distribution is a probability distribution used to model the **number of successes in a fixed number of independent trials,** where each trial has only two possible outcomes: **success or failure**.

This distribution is useful for calculating the probability of a specific number of successes in scenarios like flipping coins, quality control, or survey predictions.

Binomial Distribution is based on Bernoulli trials, where each trial has an independent and identical chance of success.

Conditions for Binomial Distribution

The Binomial distribution can be used in scenarios where the following conditions are satisfied:

1.Fixed Number of Trials: There is a set number of trials or experiments (denoted by n), such as flipping a coin 10 times.

2.Two Possible Outcomes: Each trial has only two possible outcomes, often labeled as "success" and "failure." For example, getting heads or tails in a coin flip.

3.Independent Trials: The outcome of each trial is independent of the others, meaning the result of one trial does not affect the result of another.

4.Constant Probability: The probability of success (denoted by p) remains the same for each trial. For example, if you're flipping a fair coin, the probability of getting heads is always 0.5.

Binomial Distribution Formula

The Binomial Distribution Formula, which is used to calculate the probability, for a random variable X = 0, 1, 2, 3,....,n is given as

$$P(X = r) = {}^{n}C_{r}\, p^{r}\, (1-p)^{n-r},\ r = 0, 1, 2, 3....$$

Where,

- **n** = Total number of trials
- **r** = Number of successes
- **p** = Probability of success

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Binomial Experiments

A **binomial experiment** is a probability experiment that satisfies the following conditions.

1. The experiment is repeated for a fixed number of trials, where each trial is independent of other trials.
2. There are only two possible outcomes of interest for each trial. The outcomes can be classified as a success (S) or as a failure (F).
3. The probability of a success $P(S)$ is the same for each trial.
4. The random variable x counts the number of successful trials.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Notation for Binomial Experiments

Symbol	*Description*
n	The number of times a trial is repeated.
$p = P(S)$	The probability of success in a single trial.
$q = P(F)$	The probability of failure in a single trial. ($q = 1 - p$)
x	The random variable represents a count of the number of successes in n trials: $x = 0, 1, 2, 3, \ldots, n$.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Binomial Experiments

Example:

Decide whether the experiment is a binomial experiment. If it is, specify the values of n, p, and q, and list the possible values of the random variable x. If it is not a binomial experiment, explain why.

- You randomly select a card from a deck of cards, and note if the card is an Ace. You then put the card back and repeat this process 8 times.

This is a binomial experiment. Each of the 8 selections represent an independent trial because the card is replaced before the next one is drawn. There are only two possible outcomes: either the card is an Ace or not.

$$n = 8 \qquad p = \frac{4}{52} = \frac{1}{13} \qquad q = 1 - \frac{1}{13} = \frac{12}{13} \qquad x = 0,1,2,3,4,5,6,7,8$$

Binomial Experiments

Example:
Decide whether the experiment is a binomial experiment. If it is, specify the values of n, p, and q, and list the possible values of the random variable x. If it is not a binomial experiment, explain why.

- You roll a die 10 times and note the number the die lands on.

This is not a binomial experiment. While each trial (roll) is independent, there are more than two possible outcomes: 1, 2, 3, 4, 5, and 6.

Binomial Probability Formula

In a binomial experiment, the probability of exactly x successes in n trials is

$$P(x) = {}_nC_x p^x q^{n-x} = \frac{n!}{(n-x)!\,x!} p^x q^{n-x}.$$

Example:

A bag contains 10 chips. 3 of the chips are red, 5 of the chips are white, and 2 of the chips are blue. Three chips are selected, with replacement. Find the probability that you select exactly one red chip.

p = the probability of selecting a red chip $= \frac{3}{10} = 0.3$

$q = 1 - p = 0.7$

$n = 3$

$x = 1$

$$P(1) = {}_3C_1(0.3)^1(0.7)^2$$
$$= 3(0.3)(0.49)$$
$$= 0.441$$

Mean, Variance and Standard Deviation

Population Parameters of a Binomial Distribution

Mean: $\mu = np$

Variance: $\sigma^2 = npq$

Standard deviation: $\sigma = \sqrt{npq}$

Example:

One out of 5 students at a local college say that they skip breakfast in the morning. Find the mean, variance and standard deviation if 10 students are randomly selected.

$n = 10$

$p = \frac{1}{5} = 0.2$

$q = 0.8$

$\mu = np$

$= 10(0.2)$

$= 2$

$\sigma^2 = npq$

$= (10)(0.2)(0.8)$

$= 1.6$

$\sigma = \sqrt{npq}$

$= \sqrt{1.6}$

≈ 1.3

Poisson Distribution

The Poisson distribution is a **discrete probability distribution** that calculates the likelihood of a certain number of **events happening in a fixed interval of time**, assuming the events occur independently.

Example: *Emails Per Hour*

If you receive emails randomly at an average rate of 5 per hour (λ = 5), the Poisson distribution can tell you the probability of receiving 0 emails, exactly 3 emails, and so on.

, λ (lambda), which represents the event's average occurrence rate in an interval(not a subinterval).

When do we use it?

Poisson works **only if these are true**:

We count events

Example: number of phone calls, accidents, emails, arrivals, etc.

Fixed interval

The interval can be:

- Time (per minute, per hour)
- Area
- Volume

Same average rate

The event happens at a **constant average rate**.

Independence

What happens now **does not affect** what happens later

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Poisson Distribution

The **Poisson distribution** is a discrete probability distribution of a random variable x that satisfies the following conditions.

1. The experiment consists of counting the number of times an event, x, occurs in a given interval. The interval can be an interval of time, area, or volume.
2. The probability of the event occurring is the same for each interval.
3. The number of occurrences in one interval is independent of the number of occurrences in other intervals.

The probability of exactly x occurrences in an interval is

$$P(x) = \frac{\mu^x e^{-\mu}}{x!}$$

where $e \approx 2.71818$ and μ is the mean number of occurrences.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Poisson Distribution

Example:

The mean number of power outages in the city of Brunswick is 4 per year. Find the probability that in a given year,

a.) there are exactly 3 outages,

b.) there are more than 3 outages.

a.) $\mu = 4, \quad x = 3$

$$P(3) = \frac{4^3(2.71828)^{-4}}{3!}$$

$$\approx 0.195$$

b.) $P(\text{more than } 3)$

$$= 1 - P(x \le 3)$$

$$= 1 - [P(3) + P(2) + P(1) + P(0)]$$

$$= 1 - (0.195 + 0.147 + 0.073 + 0.018)$$

$$\approx 0.567$$

Mean and variance of a Poisson distribution

The Poisson distribution has only one parameter, called λ.

- The mean of a Poisson distribution is λ.
- The variance of a Poisson distribution is also λ.

In most distributions, the mean is represented by μ (mu) and the variance is represented by σ^2 (sigma squared). Because these two parameters are the same in a Poisson distribution, we use the λ symbol to represent both.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Poisson distribution formula

The probability mass function of the Poisson distribution is:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

Where:

- X is a random variable following a Poisson distribution
- k is the number of times an event occurs
- $P(X = k)$ is the probability that an event will occur k times
- e is Euler's constant (approximately 2.718)
- λ is the average number of times an event occurs
- ! is the factorial function

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Example: Applying the Poisson distribution formula

An average of 0.61 soldiers died by horse kicks per year in each Prussian army corps. You want to calculate the probability that exactly two soldiers died in the VII Army Corps in 1898, assuming that the number of horse kick deaths per year follows a Poisson distribution.

Calculation

The specific army corps (VII Army Corps) and year (1898) don't matter because the probability is constant.

k = 2 deaths by horse kick

λ = 0.61 deaths by horse kick per year

e = 2.718

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

$$P(X = 2) = \frac{(2.718^{-0.61})(0.61^2)}{2!}$$

$$P(X = 2) = \frac{(0.54339)(0.3721)}{2}$$

$$P(X = 2) = 0.101$$

The probability that exactly two soldiers died in the VII Army Corps in 1898 is 0.101.

Practice questions

1 → At a small walk-in clinic, an average of five patients arrive at the clinic per hour during opening hours. What is the probability that exactly three patients will arrive in the next hour?

$k = 3$ patients
$\lambda = 5$ patients/hour
$e = 2.718$

$$P(X = k) = \frac{e^{-\lambda}\lambda^{k}}{k!}$$

$$P(X = 3) = \frac{(2.718^{-5})(5^{3})}{3!}$$

$$P(X = 3) = \frac{(0.00674)(125)}{6}$$

$$P(X = 3) = 0.14$$

2 → If you receive an average of two emails per week from your statistics professor, what is the probability that you will receive exactly one email from your statistics professor on Monday?

k = 1 email
λ = 2 emails per week = 2/7 emails per day = 0.286 emails per day (The question asks for the probability in units of days, so λ needs to be converted from units of weeks to days.)
e = 2.718

$$P(X = k) = \frac{e^{-\lambda}\lambda^{k}}{k!}$$

$$P(X = 1) = \frac{(2.718^{-0.286})(0.286^{1})}{1!}$$

$$P(X = 1) = \frac{(0.75128)(0.286)}{1}$$

$$P(X = 1) = 0.215$$

3 → Over the last 300 years, there were 87 floods in Statsville. Assuming that the number of floods per year follows a Poisson distribution, what is the probability that there will be no floods in Statsville next year?

k = 0 floods
λ = 87 floods per 300 years = 87/300 floods per year = 0.29 floods per year
e = 2.718

$$P(X = k) = \frac{e^{-\lambda}\lambda^{k}}{k!}$$

$$P(X = 0) = \frac{(2.718^{-0.29})(0.29^{0})}{0!}$$

$$P(X = 0) = \frac{(0.74829)(1)}{1}$$

$$P(X = 0) = 0.748$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Discrete Random Variable

① if a Random variable takes only a finite or Countable number of values, then the variable is called DRV.

② Probability Distribution of a Discrete Random variable is called Discrete Probability Distribution or Probability Mass Function

③ Properties of PMF are

① $P_i \geq 0$ for all i,
i.e P_i's are all non-Negative

② $\sum P_i = 1$
i.e Sum of Probabilities must be Equal to 1

Continuous Random Variable

if a Random variable takes any value b/w certain interval, then the variable is called CRV.

Probability Distribution of a Continuous Random variable is called continuous Probability Distribution or Probability Density Function.

Properties of PDF is

① $f(x) \geq 0$

② $\int_{-\infty}^{\infty} f(x) \cdot d(x) = 1$

Discrete Probability Distribution

Binomial Distribution Poisson Distribution

Continuous Probability Distribution

↓

Normal Distribution

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Marginal Probability

Marginal probability is the probability of a **single event occurring without considering the outcomes of other related events**. It is obtained from a joint probability distribution by summing (or integrating) over the possible values of the other variables.
They are widely used in fields such as statistics, economics, engineering, and social sciences.

For example, in a deck of 52 cards, there are 26 red cards (hearts and diamonds).
So, the chance of picking a red card is:
$26 \div 52 = 1/2 = 0.5$

This means there is a 50% chance of drawing a red card.

In simple words, marginal probability answers this question:

“What is the chance of this one thing happening, no matter what else happens?”

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Formula for Marginal Probability

1. Discrete Random Variables

Let X and Y be discrete random variables with joint PMF $P(X = x, Y = y)$.

Marginal PMF of X:

$$P(X = x) = \sum_{y} P(X = x, Y = y)$$

Marginal PMF of Y:

$$P(Y = y) = \sum_{x} P(X = x, Y = y)$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Continuous Random Variables

Let $f_{X,Y}(x, y)$ be the joint PDF.

Marginal PDF of X:

$$f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x, y)\, dy$$

Marginal PDF of Y:

$$f_Y(y) = \int_{-\infty}^{\infty} f_{X,Y}(x, y)\, dx$$

How to Calculate Marginal Probability (Steps)

1.Identify the **joint probability distribution** (table or PDF).

2.Eliminate the other variable:

1. Discrete → Sum
2. Continuous → Integrate

3.Interpret the result as the probability of a single variable.

problem:

If two fair dice are rolled, calculate the marginal probability of getting a 3 on the first die.

Solution:

Total number of outcomes = $6 \times 6 = 36$

Favourable outcomes (first die = 3) = 6

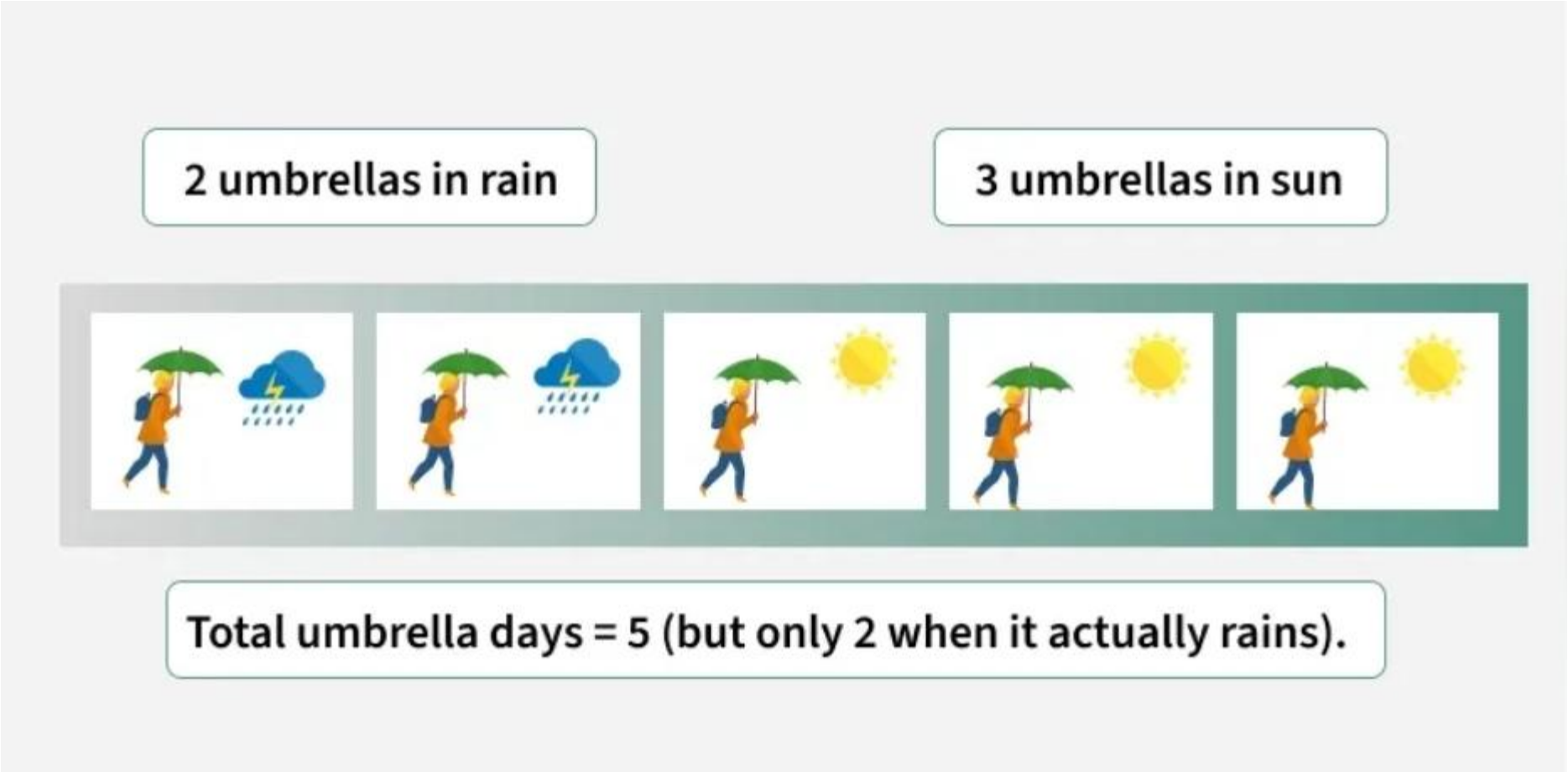
$$P(First\ die = 3) = \frac{6}{36} = \frac{1}{6}$$

3. Conditional Probability

Conditional probability defines the probability of **an event occurring based on a given condition or prior knowledge of another event.**

It is **an event occurring**, given that another event has already occurred. In probability, this is denoted as A given B, expressed as P(A | B), indicating the probability of event A when the event B has already occurred.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

Conditional Probability

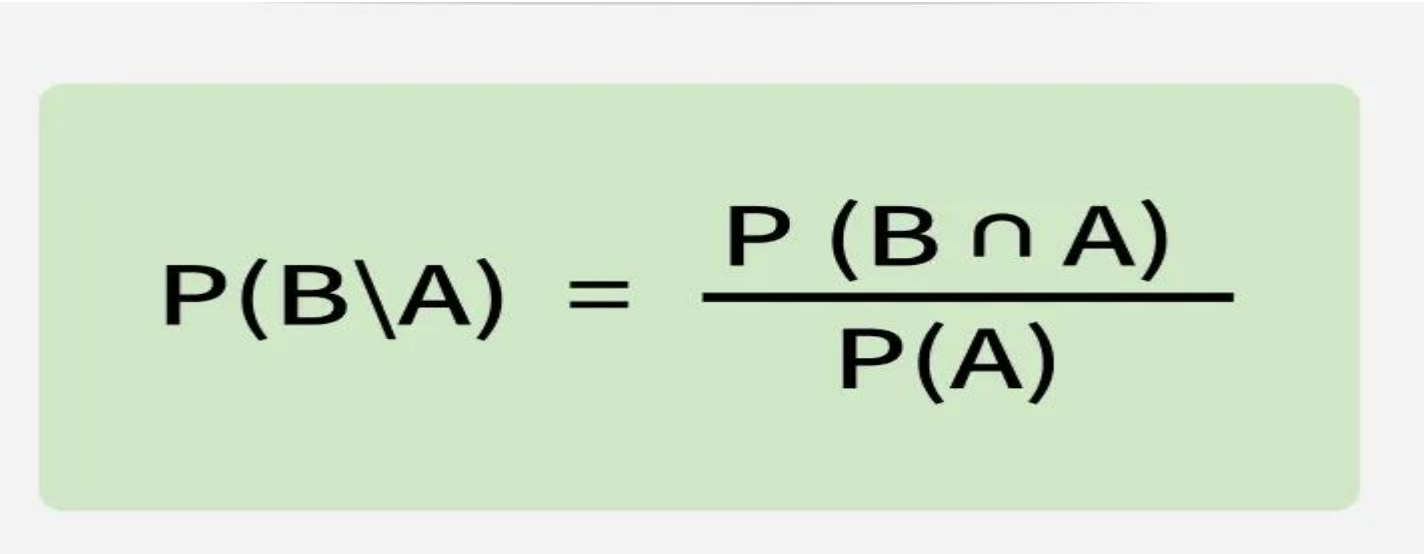
Chance it's raining given that you carry an umbrella?

2 out of 5 (40%)!

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Conditional Probability Formula

Let's consider two events A and B, then the **formula for the conditional probability** of B when A has already occurred is given by:



Where,

- **P (A ∩ B)** represents the probability of both events A and B occurring simultaneously.
- **P(A)** represents the probability of event A occurring.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Steps to Find Probability of One Event Given Another Has Already Occurred

To **calculate** the conditional probability, we can use the following step-by-step method:

Step 1: *Identify the Events. Let's call them Event A and Event B.*

Step 2: *Determine the Probability of Event A i.e.,* ***P(A)***

Step 3: *Determine the Probability of Event B i.e.,* ***P(B)***

Step 4: *Determine the Probability of Event A and B i.e.,* ***P(A ∩ B).***

Step 5: *Apply the Conditional Probability Formula and calculate the required probability.*

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

1) Tossing a Coin

Let's consider two events in tossing two coins,

•A: Getting a head on the first coin.

•B: Getting a head on the second coin. Sample space for tossing two coins is:

S = {HH, HT, TH, TT}

The conditional probability of getting a head on the second coin (B) given that we got a head on the first coin (A) is = P(B|A).

Since the coins are independent (one coin's outcome does not affect the other),
P(B|A) = P(B) = 0.5 (50%), which is the probability of getting a head on a single coin toss.

Conditional Probability vs Joint Probability vs Marginal Probability

:

Parameter	Conditional Probability	Joint Probability	Marginal Probability
Definition	The probability of an event occurring is given. That another event has already occurred.	The probability of two or more events occurring simultaneously.	The probability of an event occurring without considering any other events.
Calculation	$P(A \mid B)$	$P(A \cap B)$	$P(A)$
Variables involved	Two or more events	Two or more events	Single event.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Expectation (Expected Value or Mean)

The expectation (or expected value) is the **average value** of a random variable.
It tells us what value we expect **on average** if we repeat an experiment many times.

Discrete Case (simple explanation)

If a variable can take values x_1, x_2, x_3, …with probabilities $P(x_1), P(x_2)$, …,
then the expectation is:

$$E[x] = \sum x_i \ P(x_i)$$

Example:

Roll a fair die → 1,2,3,4,5,6 each with probability 1/6.

$$E[x] = \frac{1+2+3+4+5+6}{6} = 3.5$$

Continuous Case

For continuous variables, we replace the sum with an integral:

$$E[x] = \int x \ p(x)\, dx$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Variance

Variance measures **how much the values of a random variable differ from the mean (average)**.
It tells us **how spread out** the numbers are.

- **Low variance** → values are close to the mean
- **High variance** → values are widely spread out

Formula (General Definition)

Variance is defined as:

$$\mathrm{Var}(x) = E\left[(x - E[x])^2\right]$$

Meaning:

1. Find the mean of x
2. Subtract the mean from each value (measure the difference)
3. Square these differences
4. Take the expected value (average) of the squared differences

Discrete Case

$$\mathrm{Var}(x) = \sum P\,(x)\,\left(x - E[x]\right)^2$$

Continuous Case

$$\mathrm{Var}(x) = \int p\,(x)\,\left(x - E[x]\right)^2\,dx$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Given

A discrete random variable X with PMF:

x	P(x)
0	0.2
1	0.5
2	0.3

Step 1: Find Mean $E(X)$

$$E(X) = \sum xP(x)$$

$$E(X) = 0(0.2) + 1(0.5) + 2(0.3) = 0 + 0.5 + 0.6 = 1.1$$

Step 2: Find Variance

Formula:

$$\mathrm{Var}(X) = \sum P(x)(x - E[X])^2$$

x	P(x)	$x - E(X)$	$(x - E(X))^2$	$P(x)(x - E(X))^2$
0	0.2	−1.1	1.21	0.242
1	0.5	−0.1	0.01	0.005
2	0.3	0.9	0.81	0.243

$$\mathrm{Var}(X) = 0.242 + 0.005 + 0.243 = \boxed{0.49}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Continuous random variable X with PDF:

$$p(x) = 2x, \quad 0 < x < 1$$

Step 1: Find Mean $E(X)$

$$E(X) = \int_0^1 x \cdot 2x \, dx$$

$$E(X) = 2\int_0^1 x^2 dx = 2\left[\frac{x^3}{3}\right]_0^1 = \frac{2}{3}$$

Step 2: Find Variance

Formula:

$$\text{Var}(X) = \int p(x)(x - E[X])^2 dx$$

$$\text{Var}(X) = \int_0^1 2x \left(x - \frac{2}{3}\right)^2 dx$$

After integration:

$$\text{Var}(X) = \boxed{\frac{1}{18}}$$

Final Answer (Continuous)

- **Mean** $E(X) = \frac{2}{3}$
- **Variance** $= \frac{1}{18}$

Covariance

- **Covariance** is a measure of **the relationship between two random variables** and to what extent, they change together.
- Or in other words, it defines the changes between the two variables, such that change in one variable is equal to change in another variable.
- Mathematically, the covariance between two random variables X and Y, denoted as $\text{cov}(X,Y)$.
- **Types of Covariance**

  Covariance can have both positive and negative values. Based on this, it has two types:

  1.Positive Covariance

  2.Negative Covariance

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Covariance

1) **Positive Covariance**

   If $\text{cov}(X,Y)>0$, it indicates a positive relationship, meaning that as the value of one variable increases, the other variable tends to increase as well.

2) **Negative Covariance**

   If $\text{cov}(X,Y)<0$, it indicates a negative relationship, meaning that as the value of one variable increases, the other variable tends to decrease.

Covariance

$$Cov\,(X,Y) = \frac{\sum(X_i - \bar{X})\,(Y_i - \bar{Y})}{N-1}$$

Where:

- X_i and Y_i represent the observed values of X and Y.
- $\bar{X}$ and $\bar{Y}$ denote their respective means.
- N is the number of observations.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Covariance

Suppose we want to evaluate the relationship between the number of hours studied (X) and the test scores (Y) obtained by a group of five students. The data are below.

Hours (X)	Score (Y)
3	70
5	80
2	60
7	90
4	75

	Hours (X)	Score (Y)	$(X_i - \bar{X})$ 1	$(Y_i - \bar{Y})$	Product	2
	3	70	-1.2	-5	6	
	5	80	0.8	5	4	
	2	60	-2.2	-15	33	
	7	90	2.8	15	42	
	4	75	-0.2	0	0	
Average	4.2	75		Total	85	3
				N-1	4	
				Covariance	21.25	4

The positive covariance (21.25) suggests a positive association between the number of hours studied and exam scores. This result implies that as the number of study hours increases, the scores tend to increase.

Bayes' Rule

Bayes' Rule helps us **update a probability** when we get **new information**.
It answers this question:
"What is the probability of A happening, given that B has happened?"

Formula

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

Where:
- $P(A)$ =prior probability (before new information)
- $P(B \mid A)$ =likelihood (how likely B is if A is true)
- $P(A \mid B)$ =posterior probability (updated probability after seeing B)
- $P(B)$ =total probability of B

Example (Medical Test)
1% of people have a rare disease.
A test detects the disease 99% of the time.
You take the test → it shows **positive**.
What is the chance you actually have the disease?
We use Bayes' rule to update the probability.

Part-3

Numerical Computation: Overflow and Underflow, Gradient-Based Optimization, Constrained Optimization, Linear Least Squares.

Overflow and Underflow

1. What is Overflow?

Overflow happens when a number becomes **too large** for the computer to represent.

Example

- e^{1000} =a number with **hundreds of digits**, which a computer cannot store.
- The result becomes:

∞ (infinity) or **NaN (Not a Number)**.

Why it happens in Deep Learning

Deep learning uses:

- **Exponentials** (softmax, probability distributions)
- **Large weights or activations**
- **Accumulation of very large values**

So models may produce values that exceed floating-point limits.

2. What is Underflow?

Underflow happens when a number becomes **too small** (close to zero) and the computer rounds it **to 0**.

Example

- $e^{-1000} \approx 0$
- Floating point cannot hold such a tiny number → becomes **0**.

Why it happens in Deep Learning

Deep learning often uses:

- **Very small probabilities**
- **Log-likelihoods**
- **Tiny gradients**

Tiny numbers underflow to **0**, causing:

- Dead gradients
- Wrong probability calculations

gradient based Learning

- Gradient-based learning is a type of machine learning in which the **optimization algorithm** uses gradients to **update the model parameters during training.**
- This approach is commonly used in **deep learning and neural networks** because it allows the model to learn **complex representations of the input data**.
- It works by **iteratively adjusting the model's weights and biases** to minimize the cost function (error).

  At each step, the parameters are updated in the direction of the negative gradient of the cost function until the error becomes **very small or zero.**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Working of gradient descent:

Step 1 – Define the Goal:

•The goal of **Gradient Descent** is to **minimize the cost function** (also called the loss function), which measures the difference between:

- **Actual output (y)** (the ground truth label).
- **Predicted output (ŷ)** (what the model predicts).

Step 2 – Calculate the Cost Function:

•The **Cost Function (J)** is computed as:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Loss(y_i, \hat{y}_i)$$

•where:

- m= total number of training examples.
- θ= model parameters (e.g., weights).
- The **Loss function** measures error for a single data point.

Step 3 – Compute the Gradient (Direction):

- Calculate the **partial derivative** of the cost function with respect to each parameter θ:

$$\frac{\partial J(\theta)}{\partial \theta}$$

- This tells us the direction in which the cost function increases the most.

Step 4 – Take a Step in the Opposite Direction (Steepest Descent):

- To **reduce the cost function**, we move in the **negative direction of the gradient**.
- The **update rule** is:

$$\theta := \theta - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta}$$

- where:
  - α= learning rate (a small positive number).
  - θ= current parameter values.

Step 5 – Learning Rate Consideration:

•If α(learning rate) is:

- **Too high** → Might overshoot the minimum → Risk divergence.
- **Too small** → Takes a lot of time → More computational effort.
- Proper choice of αhelps in fast yet stable convergence.

Step 6 – Iterate Until Convergence:

•Repeat steps 3 and 4 iteratively.

•The model keeps adjusting θin each iteration.

•The process continues until:

- The **change in cost function becomes negligible**.
- Or **maximum number of iterations is reached**.
- Or the **cost function approaches zero**.

Step 7 – Termination:

•Once the cost is minimized and the parameters stabilize, the model stops updating.

•At this point, the model is considered **trained**.

Important Note:

- **Cost Function vs Loss Function:**
  - **Loss function** → Error for a **single example**.
  - **Cost function** → Average error across **all training examples**.

Types of gradient descent

There are three types of gradient descent learning algorithms

1. Batch gradient descent,
2. Stochastic gradient descent
3. Mini-batch gradient descent.

Batch Gradient Descent (Vanilla Gradient Descent)

- In this method, the **error is calculated for every sample in the dataset**,

  but the model parameters are **not updated until the entire dataset has been processed**.
- One full pass through the dataset is called an **epoch**.
- This gives a **stable error gradient** and usually leads to **stable convergence**.

Advantages

1.Fewer updates → more **efficient** than updating after each sample.

2.Updates are **stable** and smoother, which helps in convergence.

3.Easy to use with **parallel processing** (faster calculations on modern hardware).

Disadvantages

1.May **stop early at a not-so-good solution** (suboptimal).

2.Needs to **store the whole dataset in memory**.

3.Slow training for large datasets (updates happen only once per epoch).

4.Collecting all errors before updating adds extra complexity.

5.Needs **more time and computing power**.

Stochastic Gradient Descent (SGD)

- In SGD, the model updates **after every single training sample**, not after the whole dataset.
- This makes updates **frequent** and sometimes **faster** than batch gradient descent.
- But, because updates happen so often, the error path is **noisy** (it jumps up and down instead of moving smoothly).

Advantages

1. Frequent updates show **immediate progress** in learning.
2. Very **easy to understand and implement**.
3. Frequent updates let the model **learn quickly in the beginning**.
4. Noise in updates helps avoid getting stuck in **local minima**.
5. **Faster** and needs **less memory** (doesn't require whole dataset at once).
6. Works well for **large datasets**.

Disadvantages

1. Too many updates can be **computationally heavy** for large datasets.
2. Frequent updates create **noisy gradients** (error jumps instead of going down smoothly).
3. The noise can make it **hard for the model to settle at the true minimum**.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Mini-Batch Gradient Descent

•It’s a mix of **Batch Gradient Descent** and **Stochastic Gradient Descent (SGD)**.

•The training dataset is split into **small groups (mini-batches)**, usually between **50–256 samples**.

•The model updates its parameters after each mini-batch, not after the whole dataset (like Batch) and not after every single sample (like SGD).

•This makes it the **most popular method in deep learning**.

Advantages

1.Updates happen more often than Batch, so the model **learns faster** and avoids getting stuck in bad solutions (**local minima**).
2.More **computationally efficient** than SGD (since updates are done in batches, not sample by sample).
3.Doesn’t need the **whole dataset in memory at once**, which is good for large datasets.

Disadvantages

1.Requires setting an extra parameter: **mini-batch size**.
2.Like Batch, it still needs to **accumulate errors** for a batch before updating.
3.Can lead to **more complex behavior** compared to the other methods.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Challenges with Gradient Descent

1. Local Minima and Saddle Points

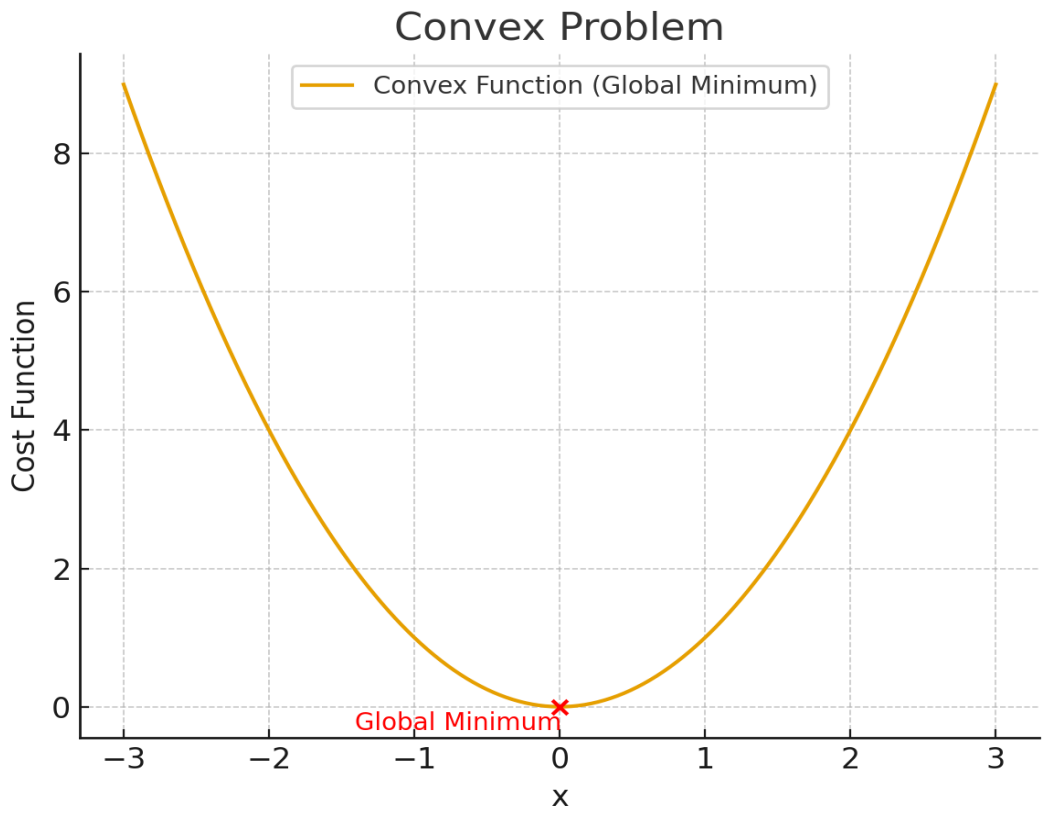
•**A convex problem** is an optimization problem where the cost function (error curve) is shaped like a bowl (U-shape).

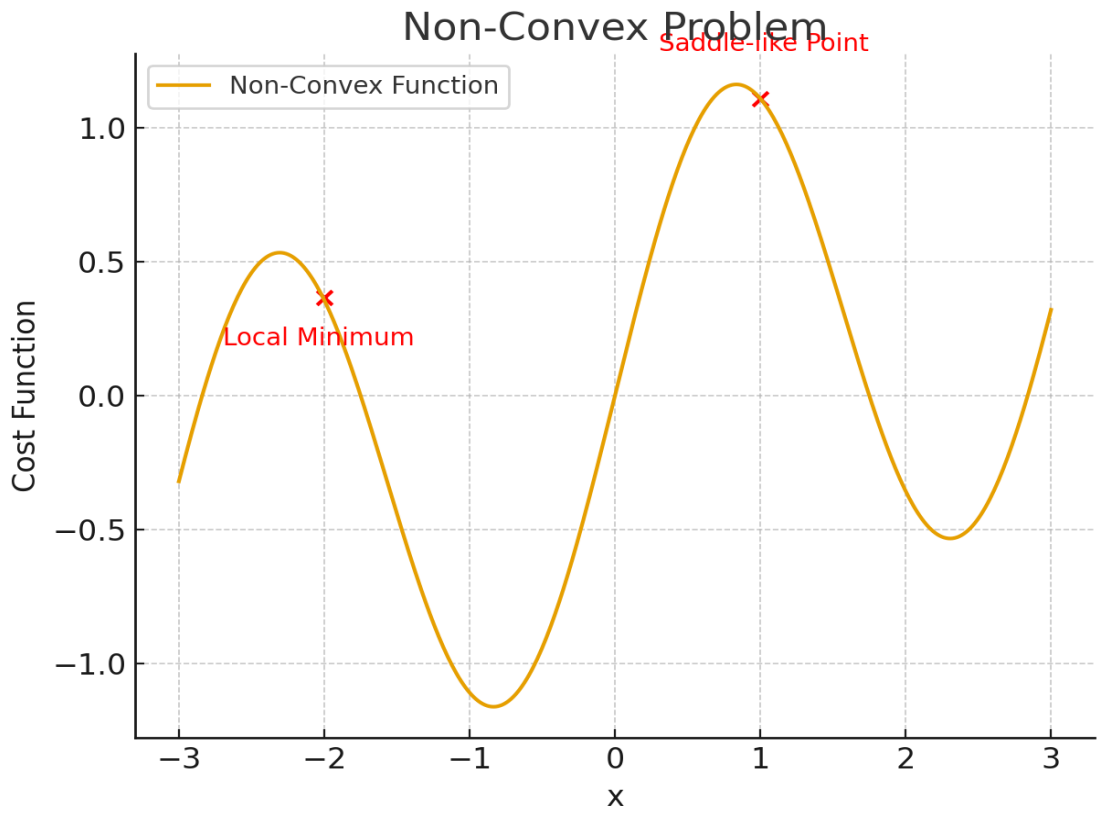
•In this shape, there is only one minimum point, called the global minimum.

•But in **non-convex problems** (more complex functions, common in deep learning), gradient descent can:

- **Get stuck in local minima** → looks like the lowest point but isn't the true best solution.
- **Get stuck at saddle points** → a point where the slope is zero, but it's not the minimum (like sitting in the dip of a horse's saddle: downhill in one direction, uphill in another).

•**Noisy gradients** (like in SGD) sometimes help escape these traps.





Here's a diagram to illustrate the **challenges with Gradient Descent**

•**Left:** Convex problem (U-shaped curve) → only one **global minimum**. Gradient descent always finds it.

•**Right:** Non-convex problem → has **local minima** and **saddle-like points** where gradient descent can get stuck.

This shows why gradient descent works perfectly for convex problems but struggles with non-convex ones.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

2. Vanishing and Exploding Gradients

These are common in **deep neural networks**, especially in **RNNs**:

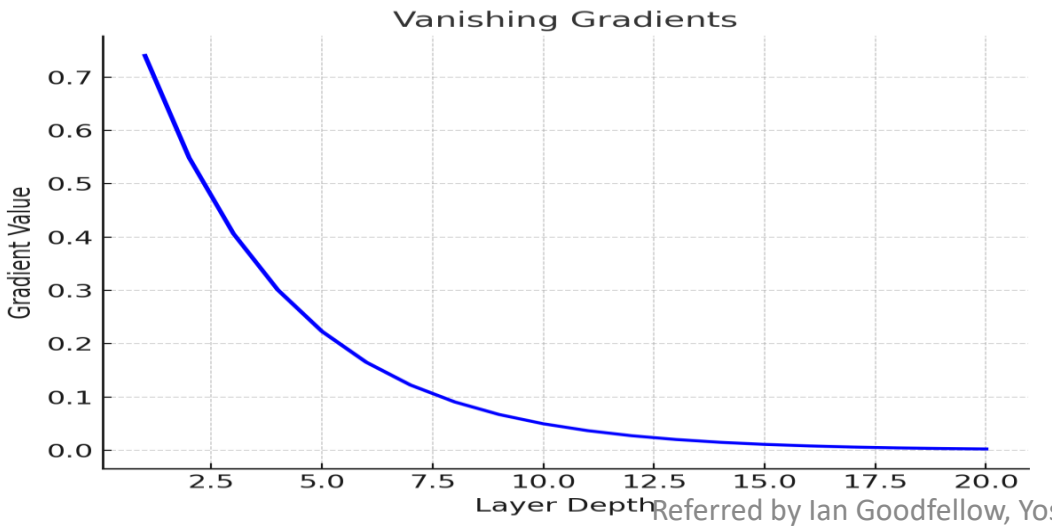
•**Vanishing Gradients**

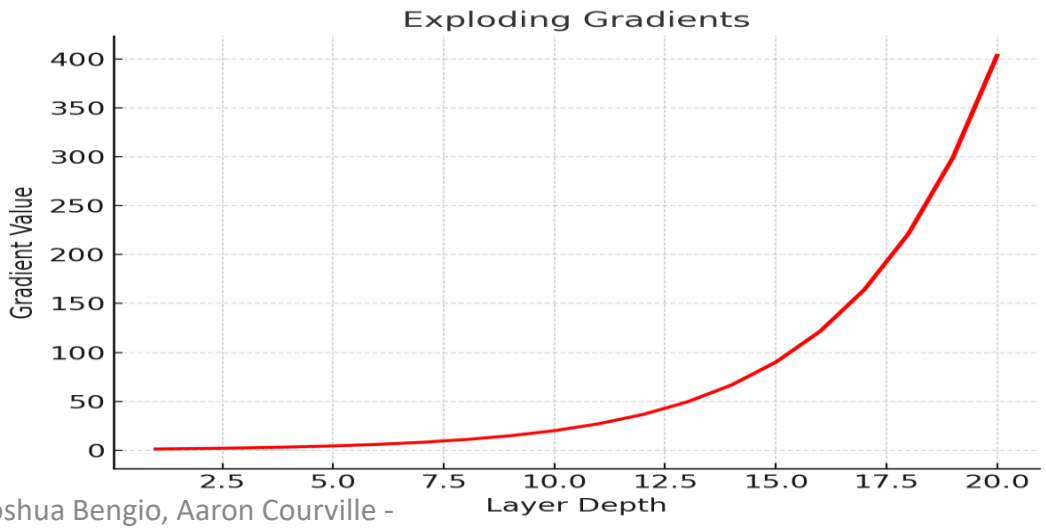
- The gradient becomes **too small** as it moves backward during training.
- Early layers update very little → they learn **very slowly or stop learning**.
- Weights shrink close to zero, and the network **stops improving**.

•**Exploding Gradients**

- The gradient becomes **too large**.
- Weights grow very big → the model becomes **unstable**.
- Sometimes weights blow up so much they turn into **NaN (Not a Number)**.

✅ **Fix:** Techniques like **gradient clipping**, **better weight initialization**, **activation functions (ReLU instead of sigmoid/tanh)**, and **dimensionality reduction** can help.





Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

In numerical computation many optimization problems involve **constraints** that restrict the set of allowable solutions. Such problems are known as **constrained optimization problems**.
Constrained optimization is the process of finding the maximum or minimum value of a function **subject to one or more constraints** on the variables.
Mathematically:

$$\min/\max \quad f(x) \textit{ subject to } x \in S$$

$S$is called the **feasible set**
Points inside $S$are called **feasible points**

Types of Constraints
Constraints are written as:

$$S = \{x, \mid g^{(i)}(x) = 0 \;\; h^{(j)}(x) \leq 0\}$$

Equality constraints: $g(x) = 0$
Inequality constraints: $h(x) \leq 0$

3. Norm Constraint

A commonly used constraint is the norm constraint:

$$\| x \| \leq 1$$

It is used to:

- Control model complexity
- Prevent overfitting
- Keep solutions small

4. Projected Gradient Descent (PGD)

Projected Gradient Descent is an optimization method used for constrained problems where, after taking a gradient descent step, the solution is projected back into the feasible region if it violates the constraint.

Algorithm Steps

1 Take a gradient descent step

$$x' = x - \eta \nabla f(x)$$

2 Check feasibility

If $x' \in S$ →keep it

If $x' \notin S$ →project it back

3 Projection step

$$x_{new} = \Pi_S(x')$$

where $\Pi_S$is the projection onto the feasible set S.

5. Re-parameterization Method

Definition (Exam-ready)

The **re-parameterization method** converts a constrained optimization problem into an **unconstrained optimization problem** by expressing the constrained variable in terms of new free parameters.

Key Idea

Instead of optimizing x**with constraints**, we:

- Express x using a new variable
- The constraint is **automatically satisfied**
- Optimize only the new variable

Example: Unit Norm Constraint

Original constrained problem:

$$\min f\,(x)\ subject\ to\ \|\,x\,\|_2 = 1$$

Re-parameterization

For 2-D vectors:

$$x = [\cos\theta\,,\sin\theta]$$

Why?

$$\|\,x\,\|_2 = \sqrt{\cos^{\ 2}\theta + \sin^{\ 2}\theta} = 1$$

✓Constraint is **always satisfied**

New unconstrained problem:

$$\min g\,(\theta) = f([\cos\theta\,,\sin\theta])$$

- Optimize over θ
- No constraints required

Karush–Kuhn–Tucker (KKT) Conditions

1. Definition

The **KKT conditions** provide **necessary conditions** for a solution to be optimal in a constrained optimization problem involvin

- Equality constraints: $g^{(i)}(x) = 0$
- Inequality constraints: $h^{(j)}(x) \leq 0$

It generalizes the **Lagrange multiplier method** to handle inequality constraints.

2. Problem Formulation

$$\min f\ (x)\ \textit{subject to}\ g^{(i)}(x) = 0,\ \ h^{(j)}(x) \leq 0$$

- x →decision variables
- $f(x)$ →objective function
- $g(x)$ →equality constraints
- $h(x)$ →inequality constraints

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

3. KKT Multipliers

For each constraint, introduce:

- λ_i →multiplier for equality constraint
- $\alpha_j \geq 0$ →multiplier for inequality constraint

These measure how strongly each constraint affects the optimal solution.

4. Generalized Lagrangian

$$L(x, \lambda, \alpha) = f(x) + \sum_i \lambda_i \, g^{(i)}(x) + \sum_j \alpha_j \, h^{(j)}(x)$$

- Combines **objective function + constraints**
- Converts constrained problem → unconstrained optimization

5. Active vs Inactive Constraints

- **Active constraint**: $h^{(j)}(x^*) = 0$ →affects solution
- **Inactive constraint**: $h^{(j)}(x^*) < 0$ →multiplier $\alpha_j = 0$

UNIT-2

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.

Fitting: Overfitting and Underfitting in ML

- Overfitting and Underfitting are the two main problems that occur in machine learning and degrade the performance of the machine learning models.
- The main goal of each machine learning model is **to generalize well**. Here **generalization** defines the ability of an ML model to provide a suitable output by adapting the given set of unknown input.
- It means after providing training on the dataset, it can produce reliable and accurate output.
- Hence, the **underfitting** and **overfitting** are the two terms that need to be checked for the performance of the model and whether the model is generalizing well or not.

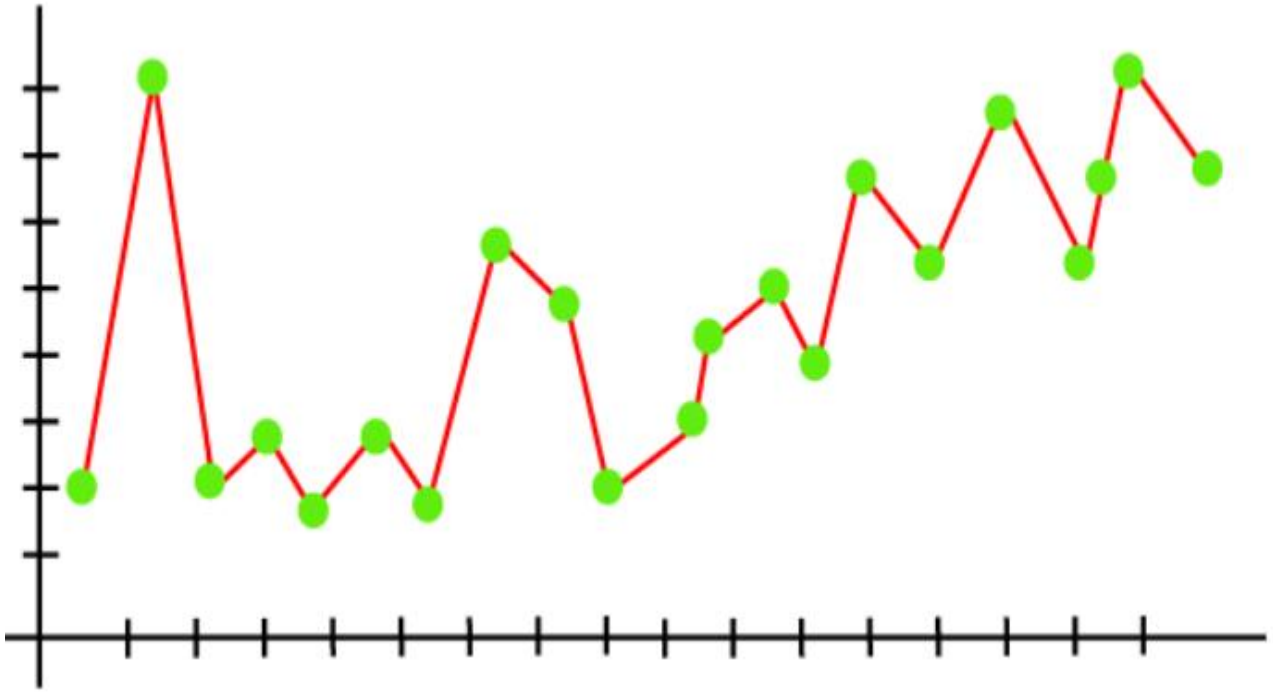
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Overfitting

- Overfitting occurs when our machine learning model **tries to cover all the data points or more than the required data points present in the given dataset.**
- Because of this, the model starts **caching noise and inaccurate values present in the dataset**, and all these factors reduce the efficiency and accuracy of the model.
- The overfitted model has **low bias** and **high variance.**
- The chances of occurrence of overfitting increase as much we provide training to our model. It means the more we train our model, the more chances of occurring the overfitted model.
- Overfitting is the main problem that occurs in S-L.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Ex: Linear Regression O/P



As we can see from the above graph, the model tries to cover all the data points present in the scatter plot. It may look efficient, but in reality, it is not so.

Because the goal of the regression model to find the best fit line,

but here we have not got any best fit, so, it will generate the prediction errors

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

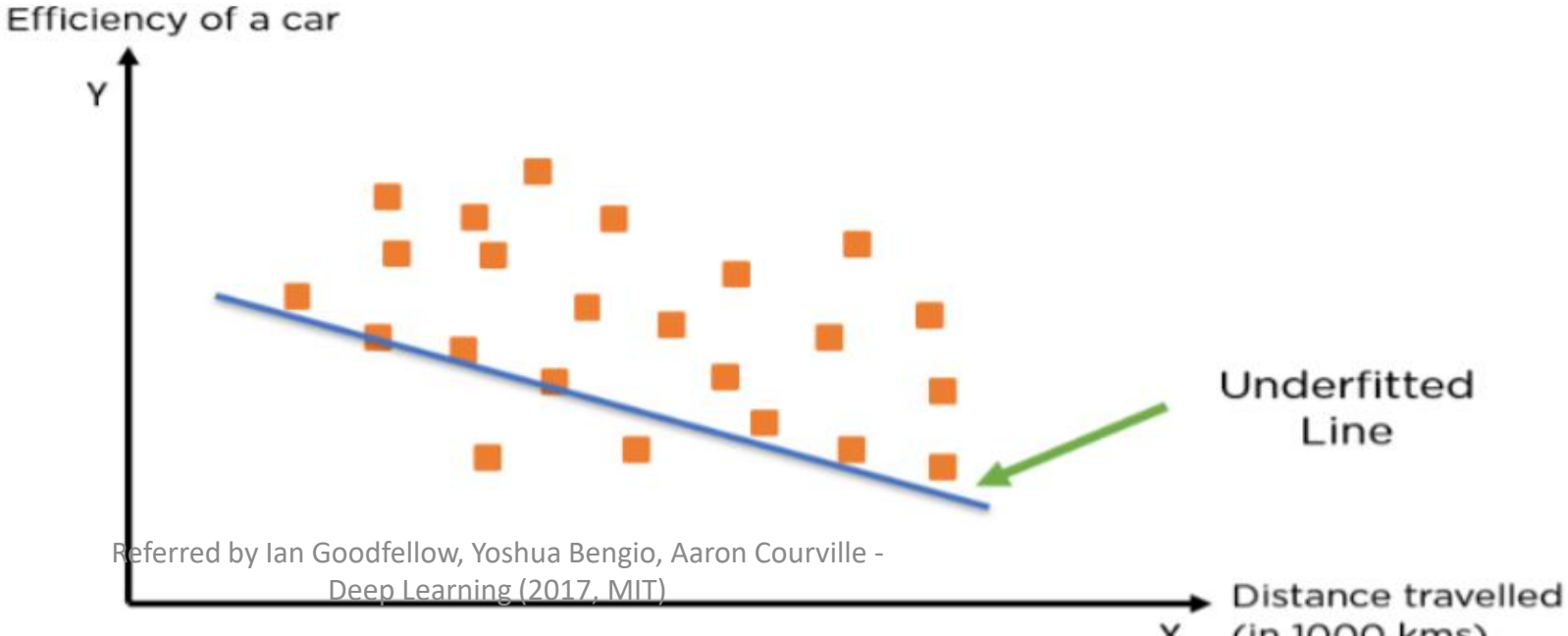
How to avoid the overfitting in model

- Both overfitting and underfitting cause the degraded performance of the machine learning model. But the main cause is overfitting, so there are some ways by which we can reduce the occurrence of overfitting in our model.
- **Cross-Validation**
- **Training with more data**
- **Removing features**
- **Early stopping the training**
- **Regularization**
- **Ensembling**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

What is Underfitting?

- When a model has not learned the patterns in the training data well and is unable to generalize well on the new data, it is known as underfitting.
- An underfit model has poor performance on the training data and will result in unreliable predictions.
- Underfitting occurs due to **high bias and low variance.**



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Reasons for Underfitting

- Data used for training is not cleaned and contains noise (garbage values) in it
- The model has a high bias
- The size of the training dataset used is not enough
- The model is too simple

Ways to Tackle Underfitting

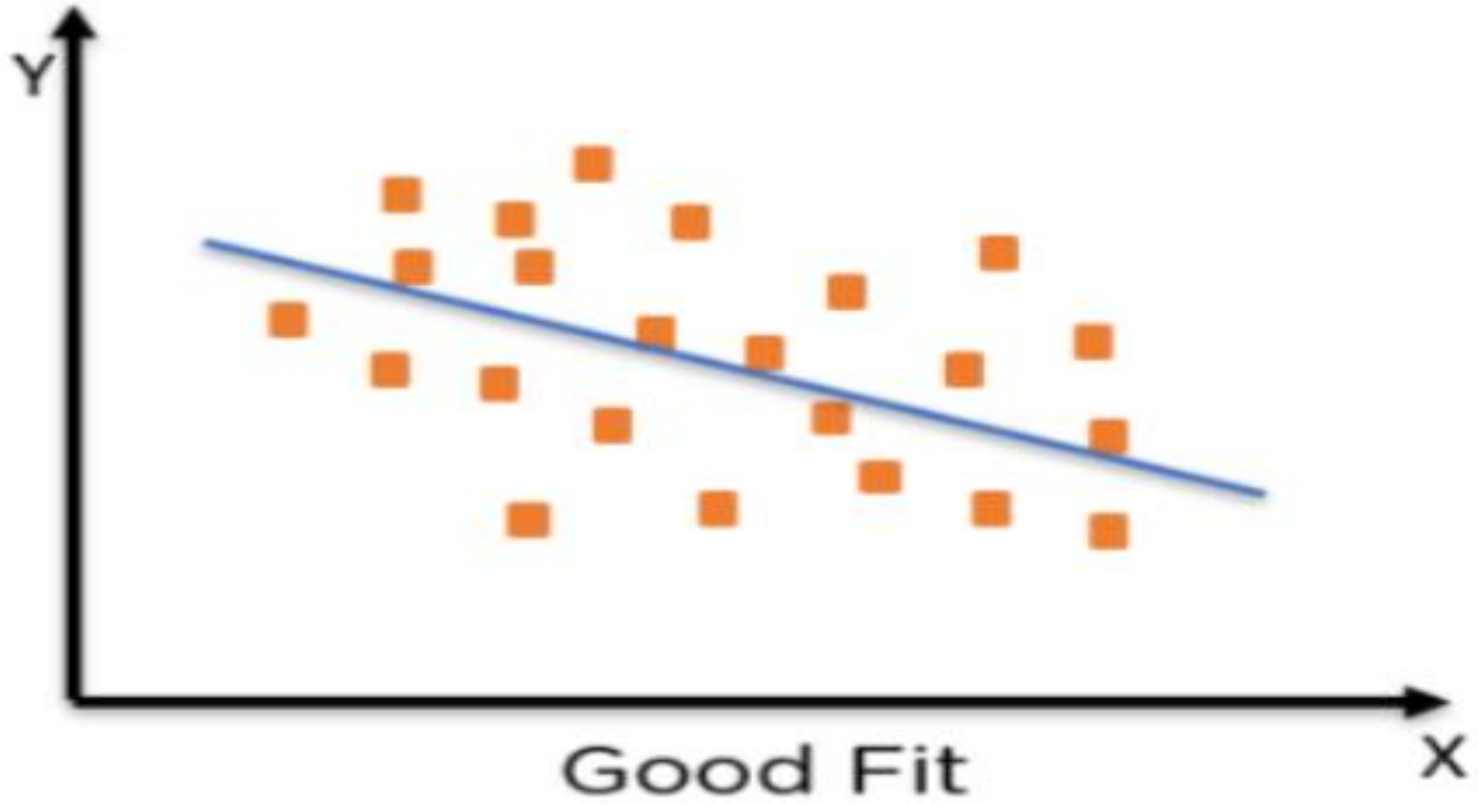
- Increase the number of features in the dataset
- Increase model complexity
- Reduce noise in the data
- Increase the duration of training the data

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

What Is a Good Fit In Machine Learning?

- To find the good fit model, you need to look at the performance of a machine learning model over time with the training data.
- As the algorithm learns over time, the error for the model on the training data reduces, as well as the error on the test dataset.
- If you train the model for too long, the model may learn the unnecessary details and the noise in the training set and hence lead to overfitting.
- In order to achieve a good fit, you need to stop training at a point where the error starts to increase.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

Estimators

- An estimator is a **function or algorithm** that tries to estimate a target variable from given input features.
- Point Estimation is the attempt to provide the single best prediction of some quantity of interest.
- Quantity of interest can be:
  - A single parameter
  - A vector of parameters — e.g., weights in linear regression
  - A whole function

Point estimator

- A point estimator is just a **formula or function** that gives you a best guess for an unknown value (parameter) based on data.
- We write this guess as:

q = true value

$\hat{q}$ (read as "q hat") = estimated value (from data)

Example:

- You want to estimate the **average exam score** of all students in a school.
  But you only collect the scores of **5 students**:
- Scores: 78, 85, 90, 82, 75
- Point estimate of the average score (mean):

$$\hat{\mu} = \frac{78+85+90+82+75}{5} = 82$$

- So, $\hat{\mu} = 82$ is your **point estimate** of the true average score.

- ✓ **Example 2:**
- ✓ Out of the past **30 days**, it rained on **12 days**.
- ✓ Point estimate of the **probability of rain**:
- ✓ p^=12/30=0.4
- ✓ So, $\hat{p} = 0.4$ is your **point estimate** of the true probability of rain on a given day.

Bias and Variance

- These are one of the ways to evaluate a machine-learning model.
- There are two types of error in machine learning:
- **1.Reducible error and 2.Irreducible error.**
- **Bias and Variance come under reducible error**.
- Bias is the error occurring between the model's **predicted value** and the **actual value.**
- Let *Y* be the true value of a parameter, and
- let $\hat{Y}$ be an estimator of Y based on a sample of data.
- Then, the bias of the estimator $\hat{Y}$ is given by: $Bias(\hat{Y})=E(\hat{Y})-Y$

- where E(Y^) is the expected value of the estimator *Y*^.
- It is the measurement of the model that how well it fits the data.
- **Low Bias:** Low bias value means **fewer assumptions** are taken to build the target function. In this case, the model will closely **match the training dataset.**
- **High Bias:** High bias value means **more assumptions** are taken to build the target function. In this case, the model will **not match the training dataset closely.**

To Reduce High Bias

- **Use a more complex model**
  - **Polynomial Regression, CNN, RNN & Image Processing**
- **Increase the number of features**
  - adding more features to train the dataset will increase the complexity of the model.
- **Reduce Regularization of the model**
  - **To prevent overfitting problem and to increase generalization ability of the model**
- **Increase the size of the training data**

Variance

- **Variance** measures how much the model's predictions **change** when trained on **different parts** of the training data.
- Example:
- Imagine training a model 3 times with different random splits of the same dataset.You get 3 different predictions for the same input:
- Prediction 1: 80
- Prediction 2: 90
- Prediction 3: 100
- Here, the model has **high variance** because the output changes a lot.

  - E[Y^] :expected prediction (average over many models)
  - Y^: prediction from model

- Variance=$E[(Y\hat{} - E[Y\hat{}])2]$

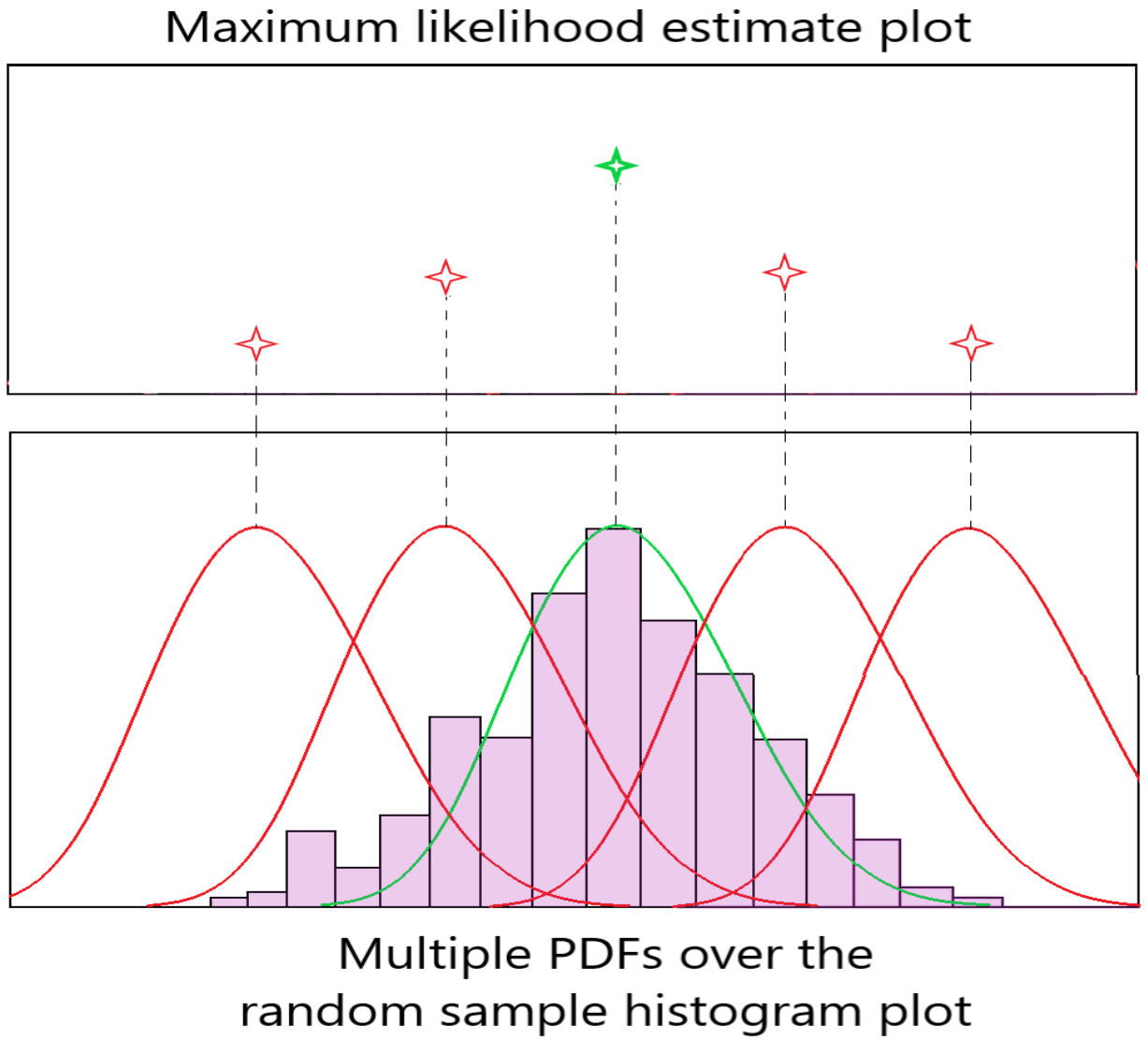
- where E[Y^] is the expected value of the predicted values. Here expected value is averaged over all the training data.
- Variance errors are either low or high-variance errors
- **Low variance:** Low variance means that the model is less sensitive to changes in the training data. This is the case of **underfitting** when the model fails to generalize on both training and test data.
- **High variance:** High variance means that the model is very sensitive to changes in the training data. This is the case of **overfitting** when the model performs well on the training data but poorly on test data

Type	Meaning	Result
Low Variance	Model doesn't change much with different training data	**Underfitting** — poor on both training and test data
High Variance	Model changes too much with different training data	**Overfitting** — good on training, bad on test data

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Maximum Likelihood Estimation (MLE)

- **Maximum Likelihood Estimation** is a method of determining the parameters (mean, standard deviation, etc) of normally distributed random sample data or a method of finding the **best fitting Probability Density Function** over the **random sample data.**
- This is done by maximizing the likelihood function so that the PDF fitted over the random sample.
- Another way to look at it is that **Maximum Likelihood Estimation** function gives the mean, the standard deviation of the random sample is most similar to that of the whole sample**.**



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- The above figure shows multiple attempts at fitting the **Parametric Density Estimation** bell curve over the random sample data.
- **Red bell curves indicate poorly fitted Probability Density Function** and the **green bell curve shows the best fitting Parametric Density Estimation over the data.**
- We obtained the optimum bell curve by checking the values in Maximum Likelihood Estimate plot corresponding to each Parametric Density Estimation.
- As observed in Fig 1, the red plots poorly fit the normal distribution, hence their *'maximum likelihood estimate'* is also lower. The green PDF curve has the maximum likelihood estimate as it fits the data perfectly. This is how the maximum likelihood estimate method works.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

In the intuition, we discussed the role that Likelihood value plays in determining the optimum PDF curve. Let us understand the math involved in Maximum Likelihood Estimation Method.

We calculate Likelihood based on conditional probabilities. See the equation given below.

$$L = F(\text{,}[X_1 = x_1]\text{,}[X_2 = x_2]\text{,}\ldots[X_n = x_n] \mid P) = \Pi_{i=1}^{n} P^{x_i}\left(1 - P\right)^{1-x_i}$$

where,

L -> Likelihood value

F -> Probability distribution function

P -> Probability

$\mathbf{X_{1, X2, \ldots Xn \text{ ->}}}$ random sample of size n taken from the whole population.

$\mathbf{x_{1, x2, \ldots xn \text{ ->}}}$ values that these random sample (Xi) takes when determining the PDF.

Π -> product from 1 to n.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Bayesian Statistics

Bayesian Statistics is a framework where we use **prior knowledge** and **observed data** together to update our belief about unknown parameters.
It is based on **Bayes' theorem**:

$$P(\theta \mid X) = \frac{P(X \mid \theta)\, P(\theta)}{P(X)}$$

Where:
- **θ** → parameter
- **X** → observed data
- **P(θ)** → *prior* (belief before seeing data)
- **P(X|θ)** → *likelihood* (how likely the data is, given θ)
- **P(θ|X)** → *posterior* (updated belief after seeing data)
- **P(X)** → *evidence* or *normalizing constant*

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1. Components of Bayesian Statistics

a) Prior (P(θ))

Represents what we believe *before* seeing any data.

Example:

Belief that a coin is almost fair, so

$$P(\theta) = Beta(5, 5)$$

b) Likelihood (P(X|θ))

Tells us how probable the data is under different values of θ.

Example:

If you flip a coin 10 times and get 7 heads:

$$P(X \mid \theta) = \theta^7 \left(1 - \theta\right)^3$$

c) Posterior (P(θ|X))

Updated belief after combining **prior + data**:

$$\text{Posterior} \propto Likelihood \times Prior$$

Posterior always shifts in the direction supported by data.

challenges Motivating Deep Learning

Deep Learning emerged because traditional machine learning techniques faced several limitations when dealing with modern data and real-world problems. The major challenges that motivated the development of deep learning are described below.

1. High-Dimensional and Complex Data

Modern applications involve **images, videos, speech, text, and sensor data**.

Traditional ML models (like SVMs, logistic regression, decision trees) struggle with:

- High dimensionality
- Spatial and temporal correlations
- Complex non-linear patterns

Deep learning models (CNNs, RNNs, Transformers) handle these naturally by learning hierarchical feature representations.

2. Manual Feature Engineering

Earlier ML pipelines required **hand-crafted features** (SIFT, HOG, MFCC).

This was:

- Time-consuming
- Problem-specific
- Dependent on expert knowledge

Deep learning performs **automatic feature extraction**, reducing human effort and improving accuracy.

3. Limited Model Capacity of Traditional ML

Classical models usually assume **linear or simple decision boundaries**.
They fail to capture:
•Non-linear relationships
•Complex interactions
•Hierarchical patterns in data
Deep learning networks, with multiple layers and non-linear activations, can approximate any function (Universal Approximation Theorem).

4. Need for Scalability with Big Data

Huge datasets are generated today from:
•Social media
•E-commerce
•IoT and sensors
•Medical imaging
Traditional models do not scale well with millions of samples.
Deep learning models *improve their performance* as data increases.

5. Advances in Computation (GPUs & TPUs)

Earlier, deep models were not practical due to limited computing power.
The introduction of:

- GPUs
- TPUs
- Distributed training
- Cloud computing

made large-scale deep learning feasible.

6. Overcoming the "Curse of Dimensionality"

In high-dimensional spaces:

- Data becomes sparse
- Distances become meaningless
- Classical algorithms fail

Deep networks reduce dimensionality through layers and embeddings (e.g., CNN pooling, word embeddings).

7. End-to-End Learning

Traditional ML requires separate stages:

1.Preprocessing

2.Feature engineering

3.Model selection

4.Classification/Regression

Deep learning performs **end-to-end training**, learning everything jointly from raw data → output.

This simplifies pipelines and reduces errors introduced by manual steps.

8. Success in Difficult Real-World Tasks

Deep learning achieved breakthrough performance in tasks where conventional ML struggled:

- Image recognition (ImageNet)
- Speech recognition
- Machine translation
- Self-driving cars
- Game playing (AlphaGo)

These success stories accelerated adoption.

PART-2

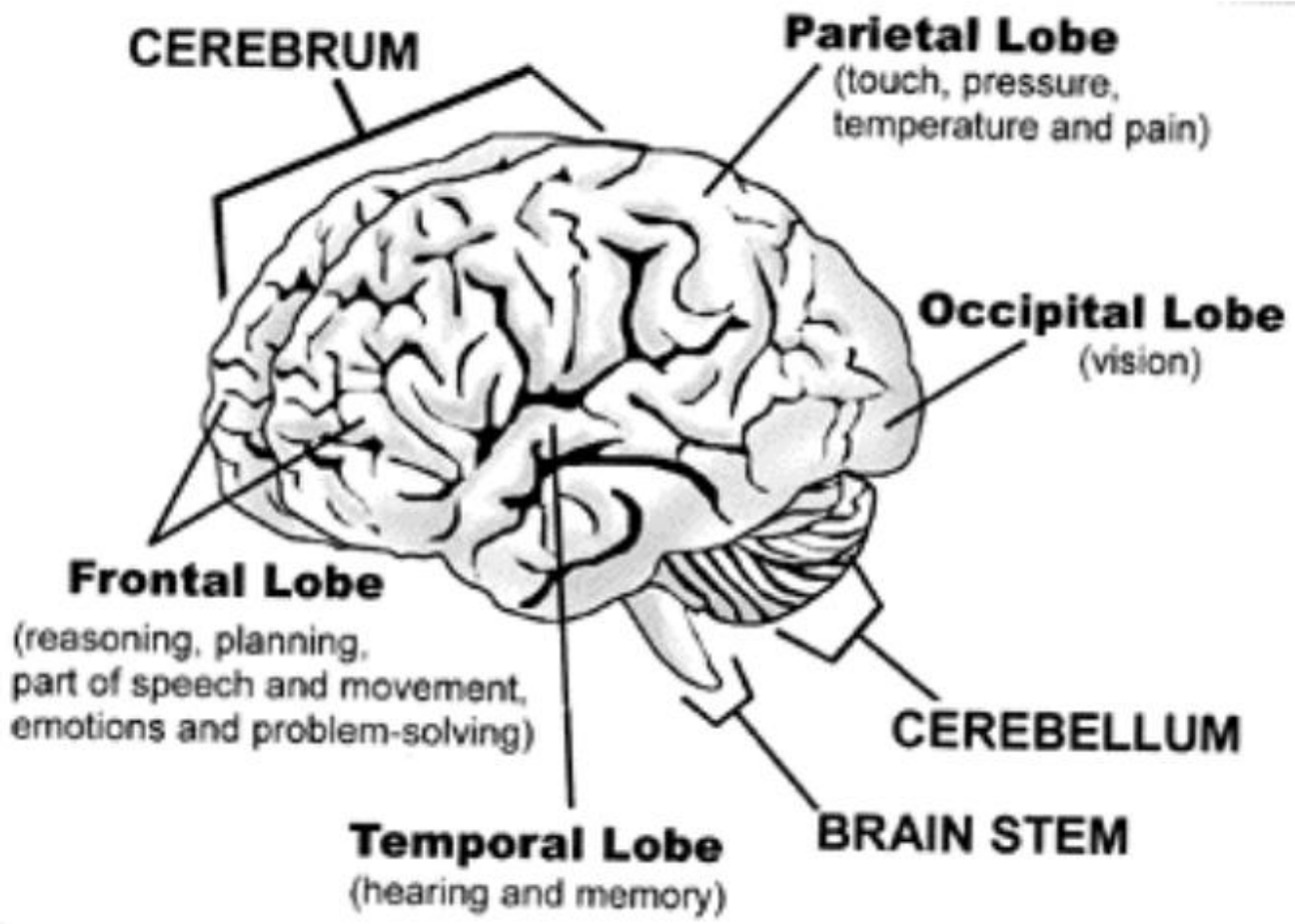
Deep Feed forward Networks: Learning XOR, Gradient-Based Learning, Hidden Units, Architecture Design, Back-Propagation and other Differentiation Algorithms.

Biological Neuron – Idea of Computational units

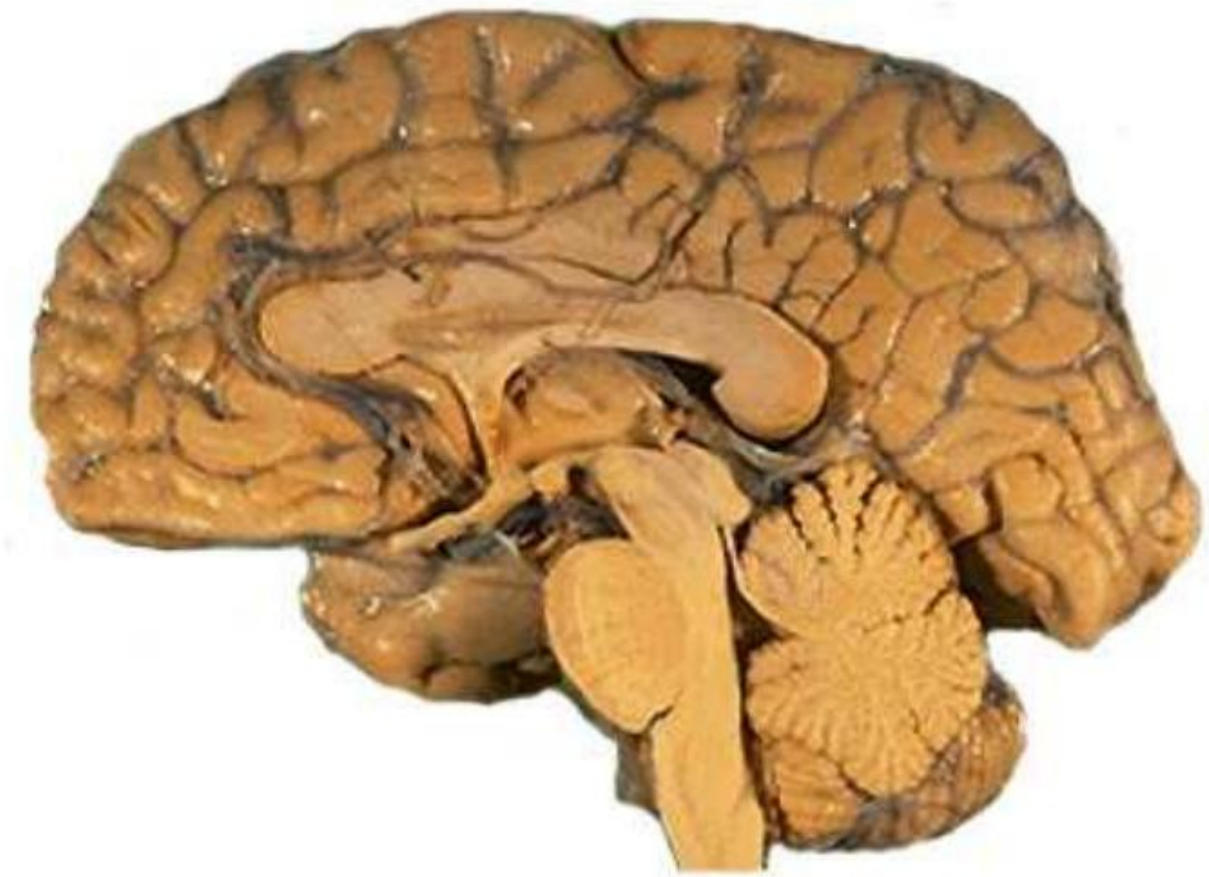
- ✓ **It is a simple implementation of how our brains might work.**
- ✓ It's not a very accurate representation but it tries to replicate some of the methods our brain uses to learn from it's mistakes.
- ✓ **The brain is essentially a bunch of neurons connected to each other in a huge interconnected network.**
- ✓ There are a **lot of neurons** and even more connections.
- ✓ These neurons pass a small amount of electrical charge to each other as a way to transmit information.
- ✓ Another important feature of these neural connections is that the connection between two neurons can be vary between **strong and weak.**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Structure of a Human Brain



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

- **For example we touch a hot pan.**
- The nerves from our hand transmits info to certain neurons in our brain.
- Now there is a **pathway** from these neurons to the neurons which control our hand.
- And in these cases our brain has **learnt** that the best option is to move our hand from the pan ASAP.
- Hence this certain **pathway** between the neurons taking input from the hand and the neurons controlling the hand will be **strong.**
- ***Neural pathways become stronger upon frequent usage, and our brain essentially tries to use pathways which have proven to give us better results over time.***
- ***All neurons have three main parts:***

    1) Dendrites
    2) cell body or soma and
    3) axons.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1. Dendrites

1. These are **branch-like structures** that receive messages from **other neurons** and allow the transmission of messages to the cell body.
2. Acquiring chemical impulse from other cells and neurons
3. Converting the chemical signals into electrical impulses
4. Carrying electrical impulses towards the next part of the neuron, the cell body

2. Cell Body or SOMA

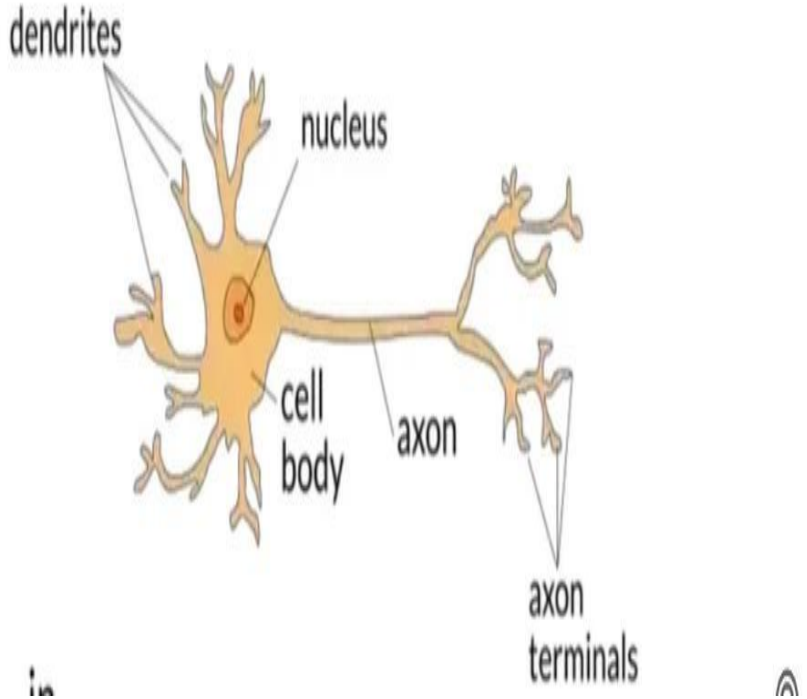
1. Joining the signals received by the dendrites and passing them to the axons, the next part of the neuron

3. Axon

1. Transmit the outflow of the message to the adjacent connected neurons

4. Synapse or axon terminals

It is the **chemical junction** between the **terminal of one neuron** and the dendrites of **another neuron**



EXPLORING THE ARTIFICIAL NEURON

- An "artificial neural network" is a computation system that attempts to mimic the neural connections in our brain.
- ***Artificial neurons*** (also called ***Perceptrons***, ***Units*** or ***Nodes***) are the simplest elements or building blocks in a Artificial neural network.
- They are inspired by biological neurons that are found in the human brain.
- Artificial neuron is a mathematical model inspired by a biological neuron.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Biological Neuron vs Artificial Neuron

1) A **biological neuron** receives its input signals from **other neurons** through **dendrites** (small fibers).

Likewise, a perceptron receives its data from other perceptrons through input neurons that take numbers.

1) The connection points between dendrites and biological neurons are called **synapses**.

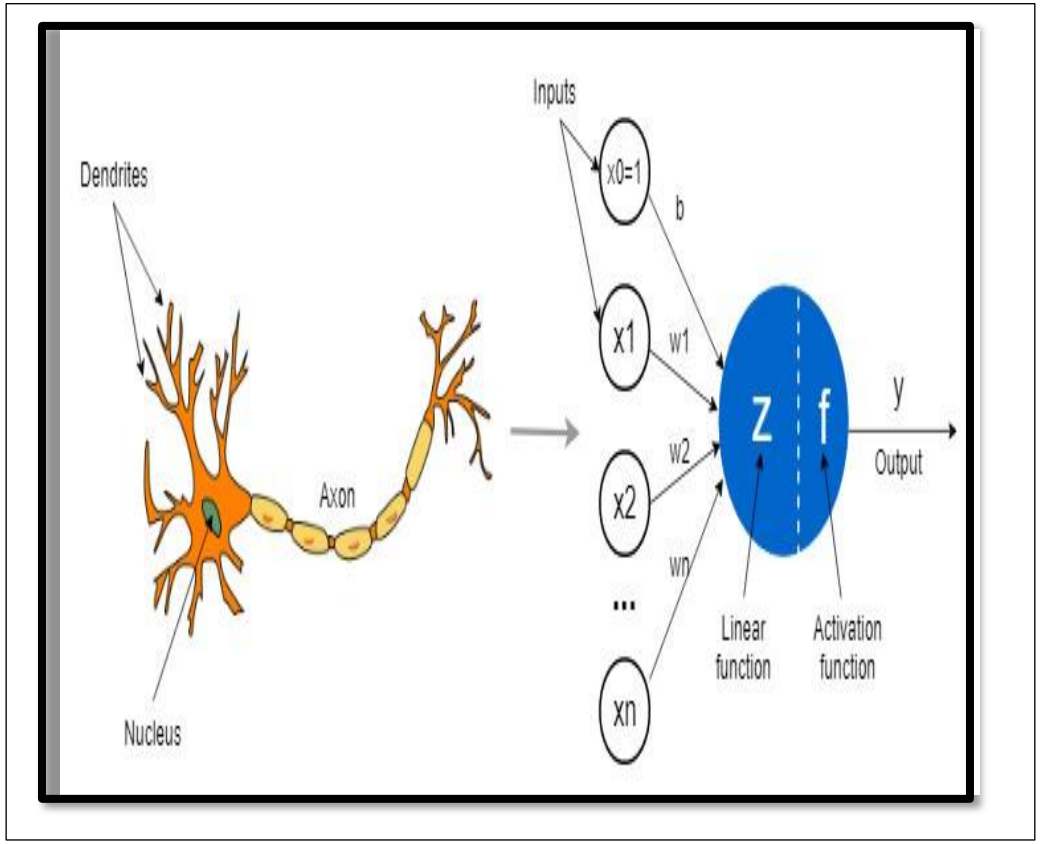
Likewise, the connections between inputs and perceptrons are called weights.

1) In a biological neuron, the ***nucleus*** produces an output signal based on the signals provided by dendrites.

Likewise, the *nucleus* (colored in blue) in a perceptron performs some calculations based on the input values and produces an output.

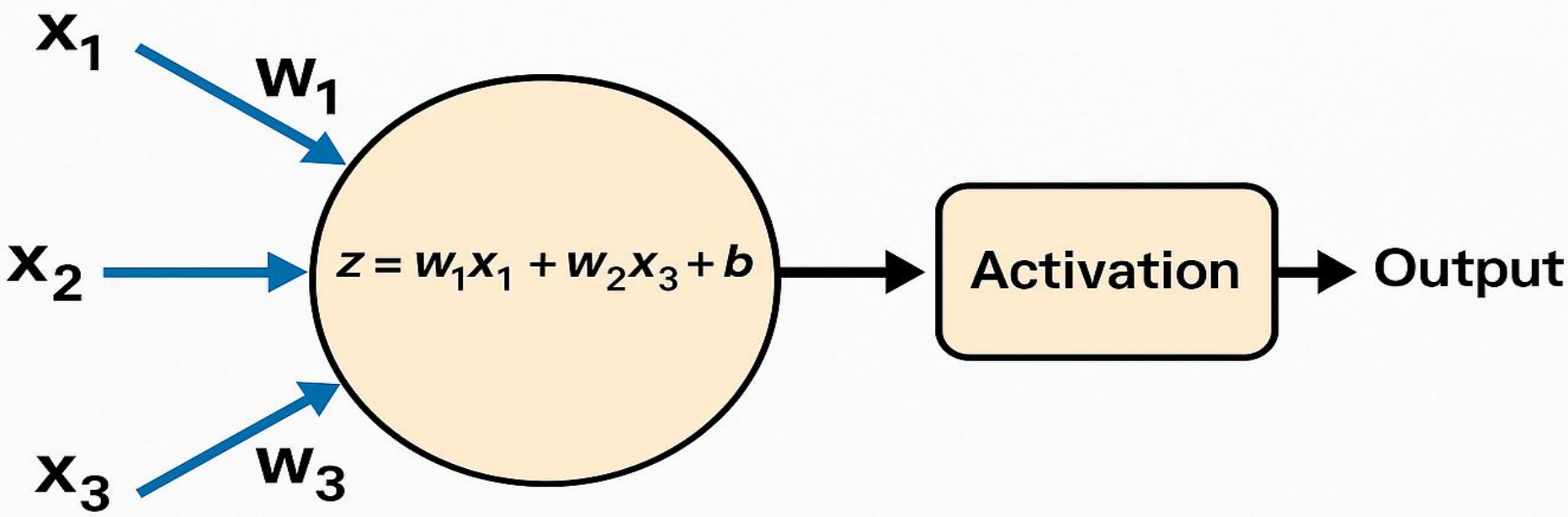
1) . In a biological neuron, the output signal is carried away by the ***axon***.

Likewise, the axon in a perceptron is the output value which will be the input for the next perceptrons



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

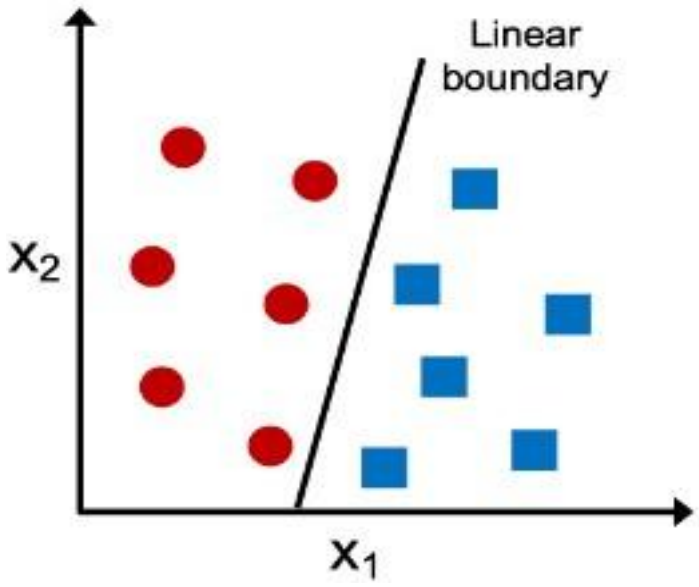
Computational Unit (Artificial Neuron)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

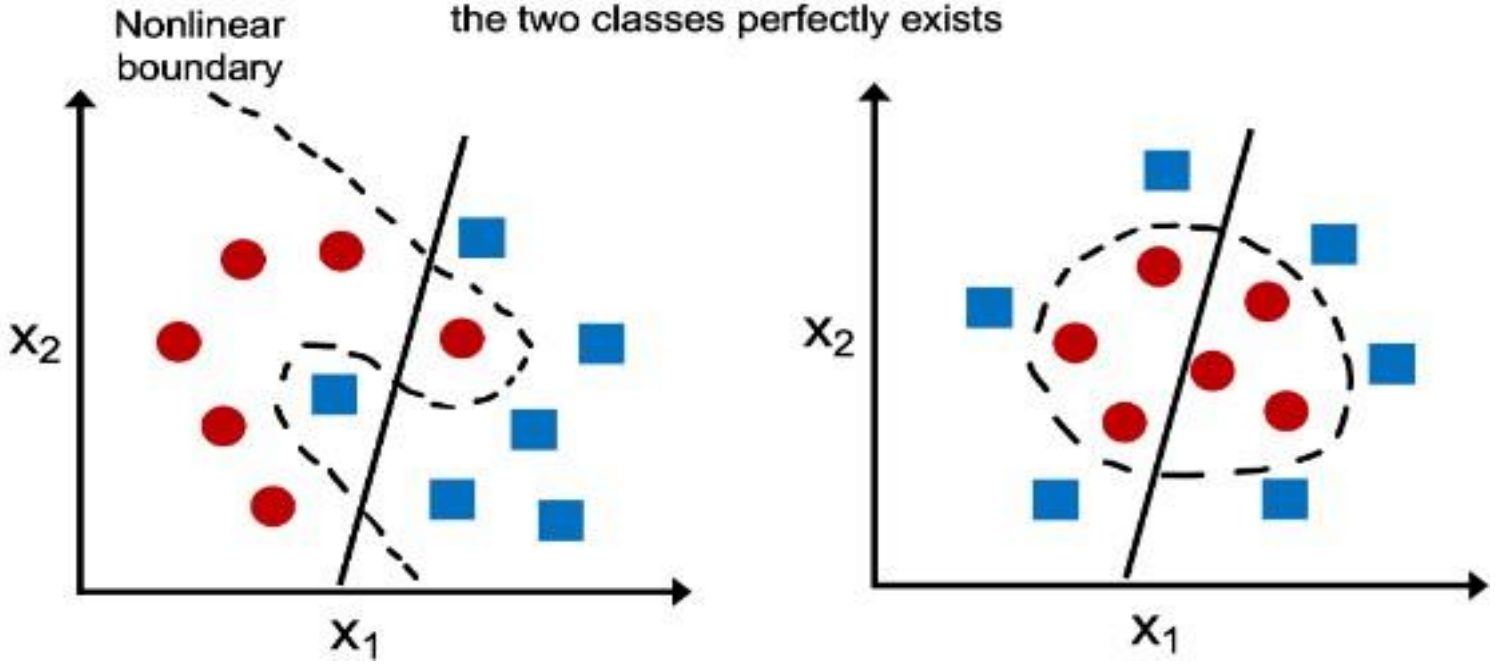
Linearly separable

A linear decision boundary that separates the two classes exists



Not linearly separable

No linear decision boundary that separates the two classes perfectly exists



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

ACTIVATION FUNCTIONS

•Linear Functions

1)If the data points can be separated by a **straight line** , then such data is called **Linearly separable data**

1)Linearly separable data produces **Linear Functions**

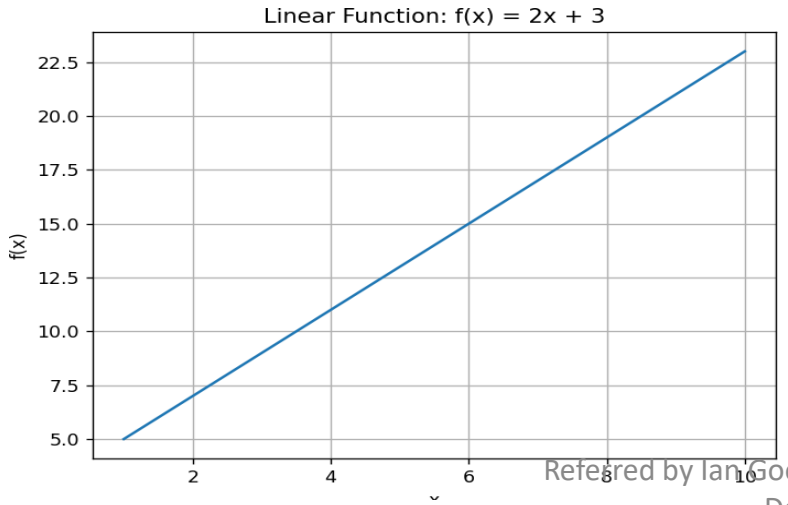
2)Eg : f(x) = y = 2x+3 ---→ Linear Function (***x and y has linear relationship)***

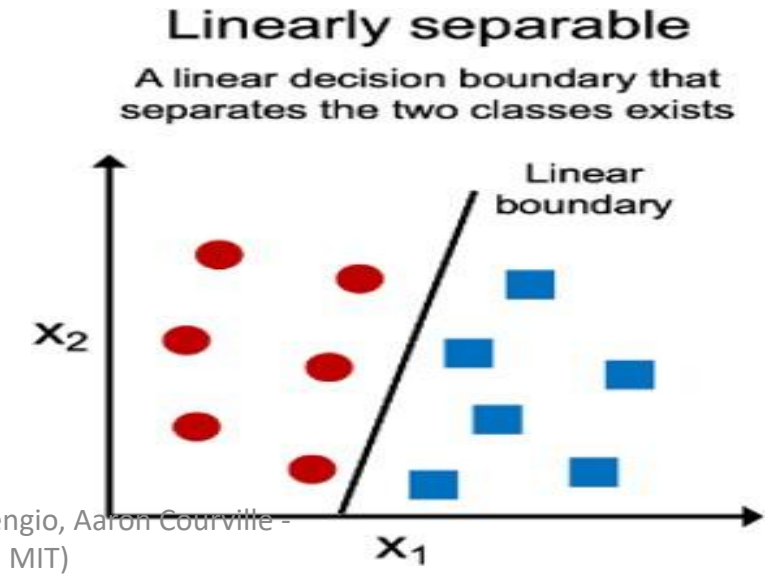
3)In linear function,

1)the output (y) is directly proportional to the input (x) .

- ❖Each increment in x leads to a constant increment in y.
- ❖When plotted on a graph, this function forms a straight line.

4)To Train Linear Functions , No need to have activation functions



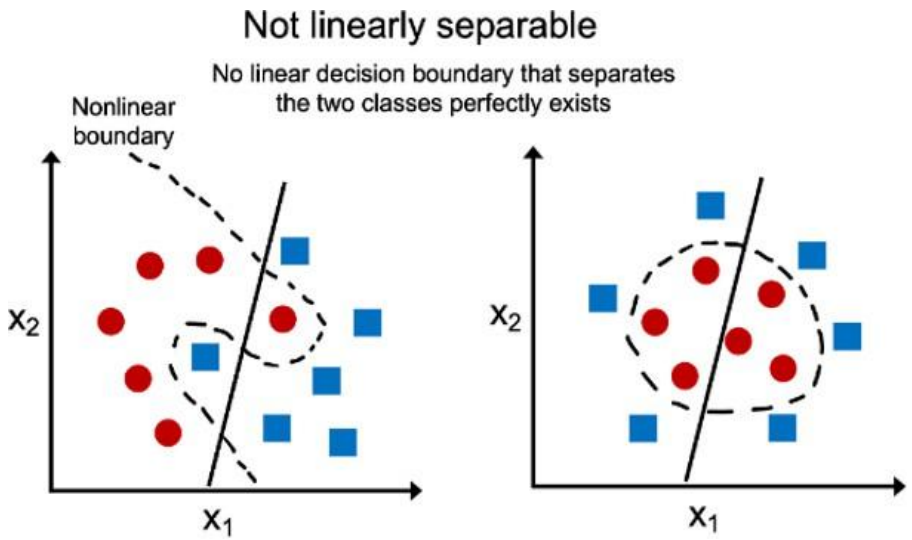


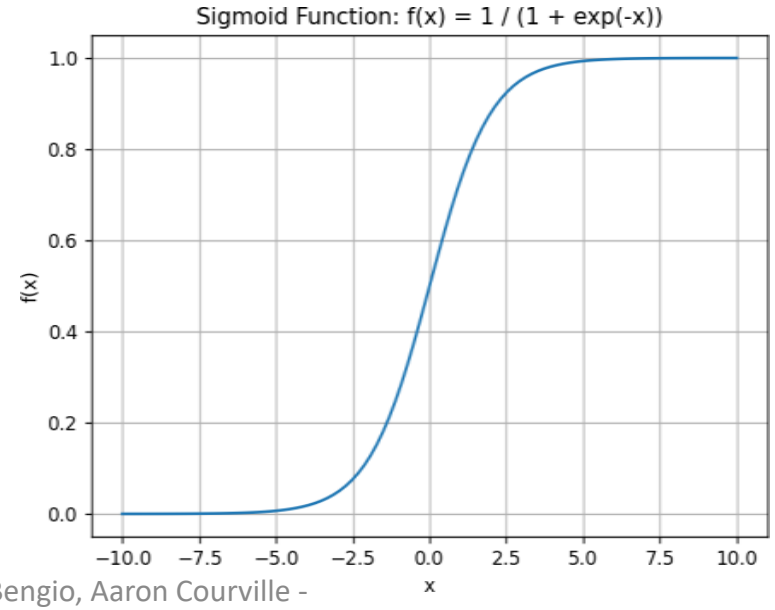
Non- Linear Functions

- If the data points are not possible to separate by a **straight line** , then such data is called **Non-Linearly separable data**
- **Non- Linearly separable data** produces **Non-Linear Functions**
- Eg : f(x) = y = 1/(1+exp(-x)) or X_5 ---→ Linear Function (***x and y has Non -linear relationship)***
- **In Non-linear function,**
  - ✓ Unlike the linear function, the output (y) is not directly proportional to the input (x).
  - ✓ An Increment in X doesn't produce a constant change in the other variable
  - ✓ When plotted on a graph, this function forms a parabolic curve.

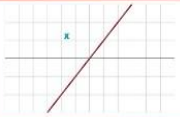
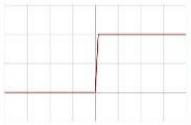
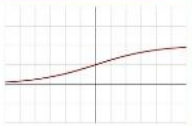
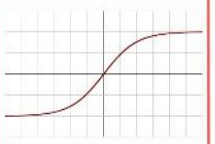
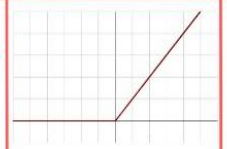
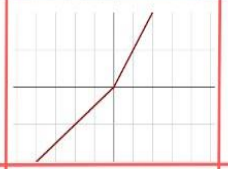
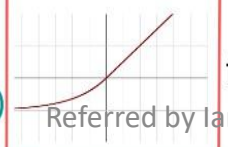
1)Thus, Activation Functions helps To Train Non-Linear Functions or to introduce Non-Linearity in Neural Network

- In a neural network context, ***activation functions are crucial for capturing and representing non-linear relationships like the quadratic function***.
- Without activation functions, neural networks would only be able to approximate linear functions, limiting their ***ability to model complex patterns and relationships in data***.
- Activation functions are a critical component of neural networks.
- They introduce non-linearity into the network, enabling it to learn complex patterns in data.





Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

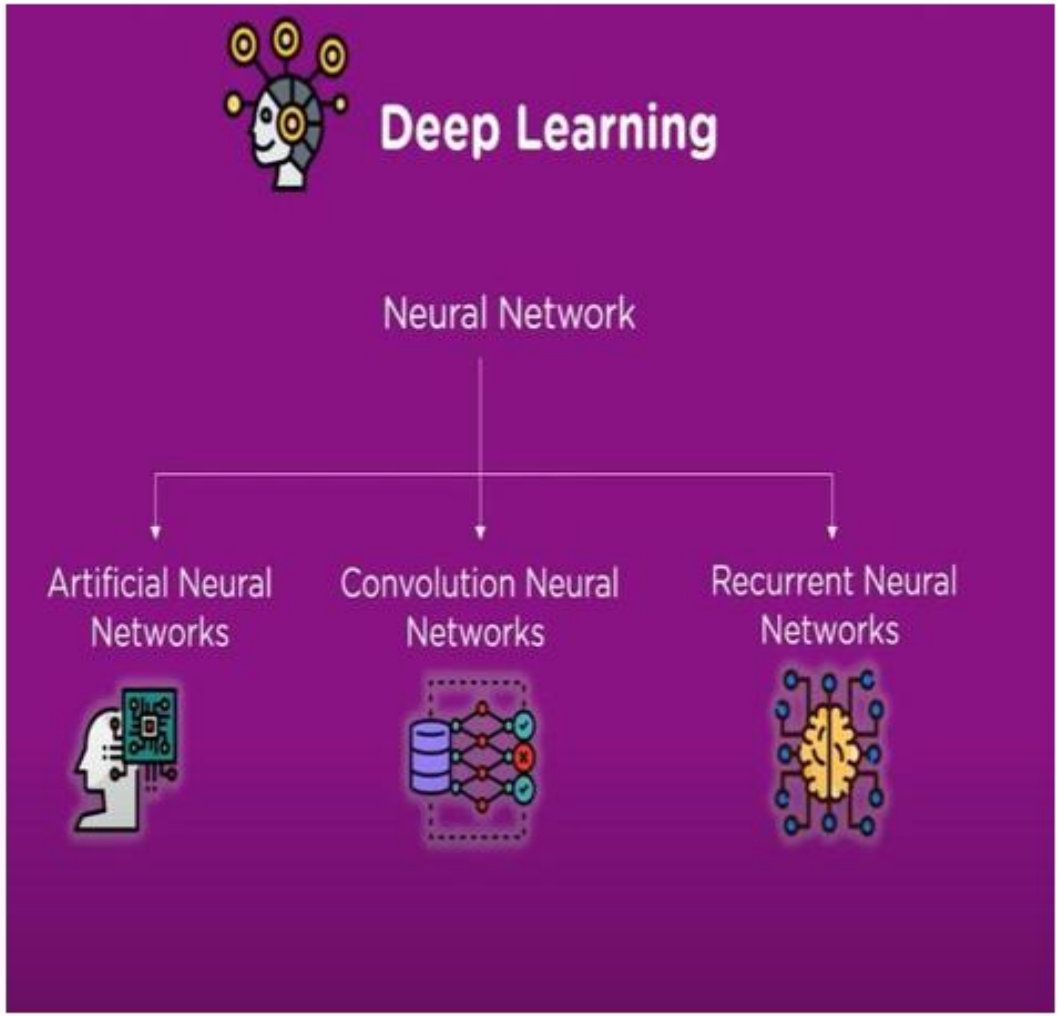
ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } x\geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1 + e^{x})$	$f'(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^{x} - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^{x} & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

ARCHITECTURE OF NEURAL NETWORK

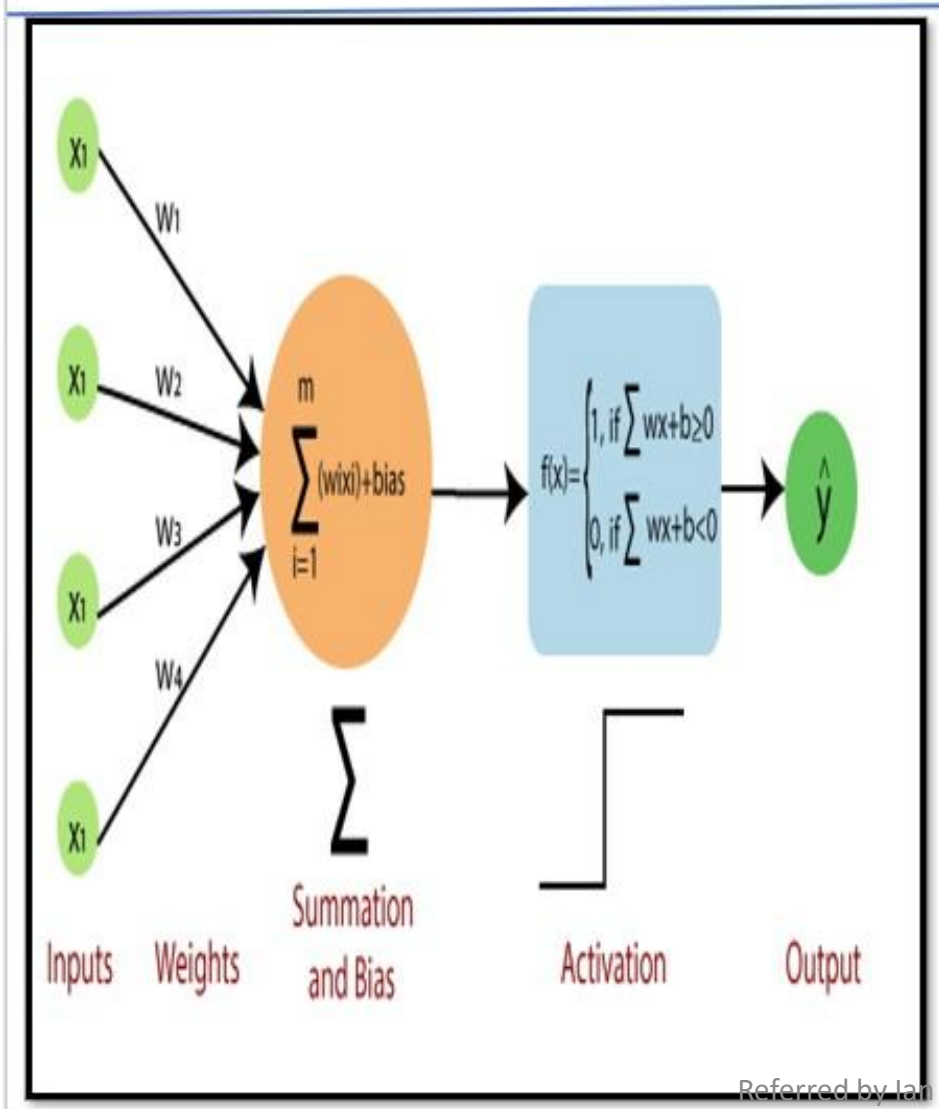
- Neural networks can be used for various tasks, including **classification, regression, clustering, and reinforcement learning.**
- They have achieved remarkable success in areas such as **image recognition, natural language processing, and autonomous driving.**
- There are different types of neural networks, including

1) **Feedforward Neural Networks** - where information flows in one direction)
   - **1.1) Single Layer Feed Forward Neural Network (or) Single Layer Perceptron**
   - **1.2) Multi-Layer Feed Forward Neural Network (or) Multi-Layer Perceptron**
2) **Recurrent Neural Networks** - which have connections that form cycles
3) **Convolutional Neural Networks** - specifically designed for processing grid-like data, such as images, and
4) more complex architectures like **Generative Adversarial Networks** (GANs) and transformers. Each type is suited to different types of data and tasks.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Artificial Neural Network Architecture



Layers

- ✓ It's a collection of interconnected nodes, called neurons, organized in layers
- ✓ A neural network usually consists of an input layer, one or more hidden layers, and an output layer.
- ✓ Information flows from the input layer through the hidden layers to the output layer.

Weights and Biases

- ✓ Each connection between neurons has associated weights and biases.
- ✓ These parameters are learned during the training process and determine the strength of connections between neurons.

Activation Function

- ✓ The activation function of a neuron defines its output based on the weighted sum of inputs. Common activation functions include sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville
Deep Learning (2017, MIT)

NOTE BIAS

Adding bias in a neural network is essential for several reasons.

- Bias allows for the **reduction of errors during computation** by activation functions and **ensures a non-null output in case of null input.**
- In neural networks, **bias acts as a constant term** that shifts the activation function, enhancing the model's flexibility and adaptability to capture complex patterns in the data effectively

ACTIVATION FUNCTION

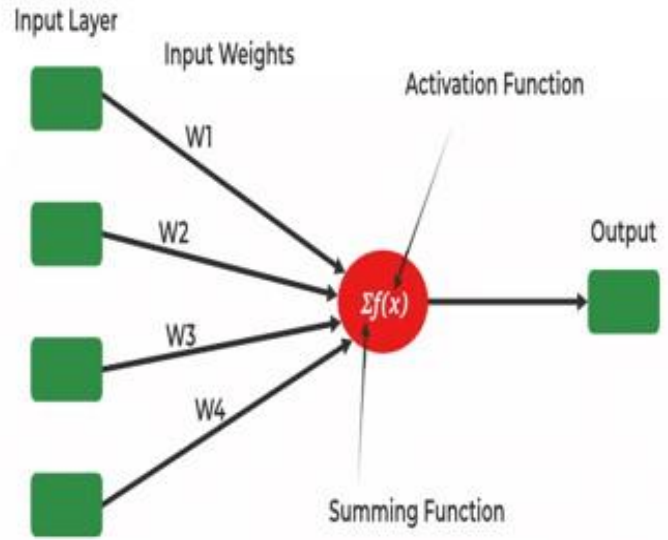
- Activation functions introduce **non-linearity** to the network, enabling it to learn complex relationships in the data.
- Common activation functions include **ReLU (Rectified Linear Unit), Sigmoid, Tanh, and softmax, each with its advantages and use cases.**

Step 1	Get the Inputs
Step 2	Initialize weights and bias randomly and send to hidden layers
	Weights and biases are **initialized randomly at the beginning of training** and are **updated during the training process to minimize the error**..
Step 3	Each neuron in a hidden layer receives input from all neurons in the previous layer and **computes its output using weighted sums and activation functions**.
	Activation functions introduce **non-linearity into the network**, allowing it to learn complex patterns in the data.
Step 4	The final layer of the neural network is the output layer, **which produces the Actual Output.**
Step 5	***Once the output is computed, it is compared with the Expected Output(or targets) to compute the loss***, which quantifies the difference between the Expected and Actual Output.
Step 6	Backpropagation is the process of computing the gradients of the loss function with respect to the **weights and biases of the network**.
Step 7	***The process of forward propagation, loss computation, and backpropagation is repeated iteratively for fixed number of epochs or until convergence***.
Step 8	Once training is complete, the trained model can be **evaluated on a separate validation or test dataset to assess its performance**
Step 9	Metrics such as **accuracy, mean squared error, or F1 score** can be used to evaluate the model's performanc depending on the nature of the task.
Step 10	Finally, the trained model can be deployed for **making predictions on new, unseen data in real-world applications.**

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Single Layer Feed Forward Neural Network

- It is the simplest and most basic architecture of ANN's.
- It consists of only two layers- the input layer and the output layer.
- ***The input layer consists of 'm' input neurons connected to each of the 'n' output neurons.***
- The connections carry weights w_{11} and so on.
- **The input layer of the neurons doesn't conduct any processing – they pass the i/p signals to the o/p neurons.**
- The computations are performed in the output layer.
- So, though it has 2 layers of neurons, only one layer is performing the computation.
- This is the reason why **the network is known as SINGLE** layer.
- Also, the signals always flow from the input layer to the output layer. Hence, the **network is known as FEED FORWARD.**
- Such simple ANN Architecture is called Perceptron
- ***Single Layer Feed Forward Neural Network or Single Layer Perceptron*** is limited to learning ***linearly separable patterns.*** effective for tasks where the data can be divided into distinct categories through a straight line.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Single Layer Feed Forward Neural Network

Basic Components of Perceptron

Input Features: The perceptron takes multiple input features, each input feature represents a characteristic or attribute of the input data.

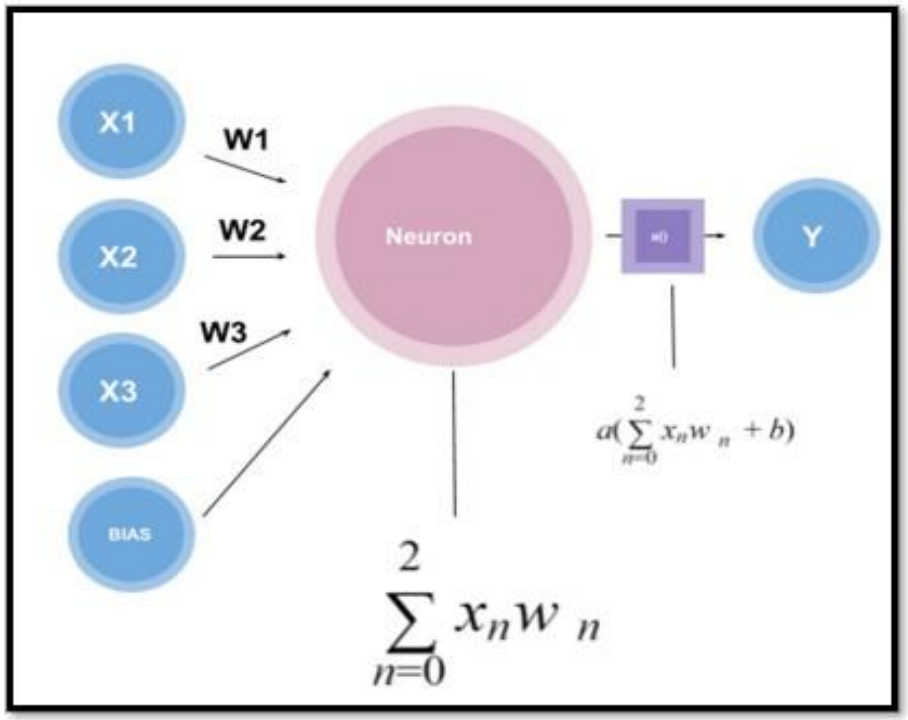
Weights: Each input feature is associated with a weight, determining the significance of each input feature in influencing the perceptron's output. During training, these weights are adjusted to learn the optimal values.

Summation Function: The perceptron calculates the weighted sum of its inputs using the summation function. The summation function combines the inputs with their respective weights to produce a weighted sum.

Activation Function: The weighted sum is then passed through an activation function. Perceptron uses Heaviside step function functions. which take the summed values as input and compare with the threshold and provide the output as 0 or 1.

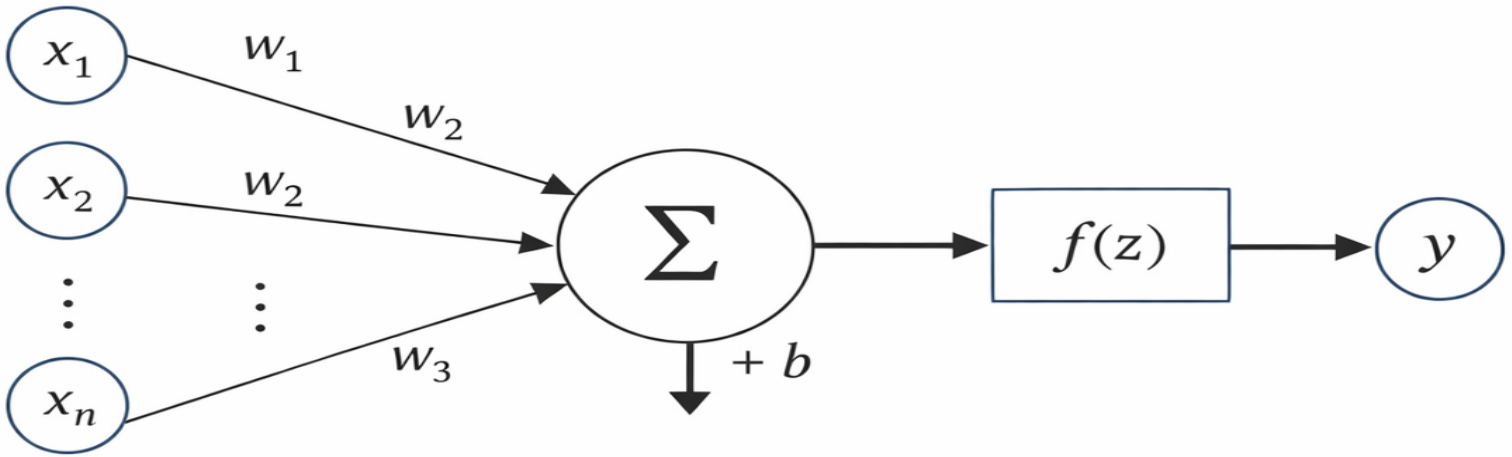
Output: The final output of the perceptron, is determined by the activation function's result. For example, in binary classification problems, the output might represent a predicted class (0 or 1).

Bias: A bias term is often included in the perceptron model. The bias allows the model to make adjustments that are independent of the input. It is an additional parameter that is learned during training.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Single Layer Feed-Forward Neural Network



$$z = \sum w_i x_i + b$$

$$y = f(z)$$

Inputs $(x_1, x_2, \ldots, x_n)$	*Weights* $(w_1, w_2, \ldots, w_n)$	*Weighted Sum* (z)	*Bias* (b)	*Activation Function*	*Output* (y)

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Single Layer Feed Forward Neural Network

Learning Algorithm (Weight Update Rule):

- ✓ During training, the perceptron learns by adjusting its weights and bias based on a learning algorithm. A common approach is the perceptron learning algorithm, which updates weights based on the difference between the predicted output and the true output.
- ✓ These components work together to enable a perceptron to learn and make predictions. While a single perceptron can perform binary classification, more complex tasks require the use of multiple perceptrons organized into layers, forming a neural network.

Multi Layer Feed Forward Neural Network

- A multilayer feedforward network, also known as a multilayer perceptron (MLP), is a type of artificial neural network architecture commonly used in deep learning.
- It consists of multiple layers of nodes (neurons) arranged in a sequence, where each node in a layer is connected to every node in the subsequent layer, but not to nodes within the same layer or to nodes in previous layers.
- The information flows in one direction, from the input layer through one or more hidden layers to the output layer, hence the term "feedforward."

- Here's a breakdown of the components:

1) Input Layer:

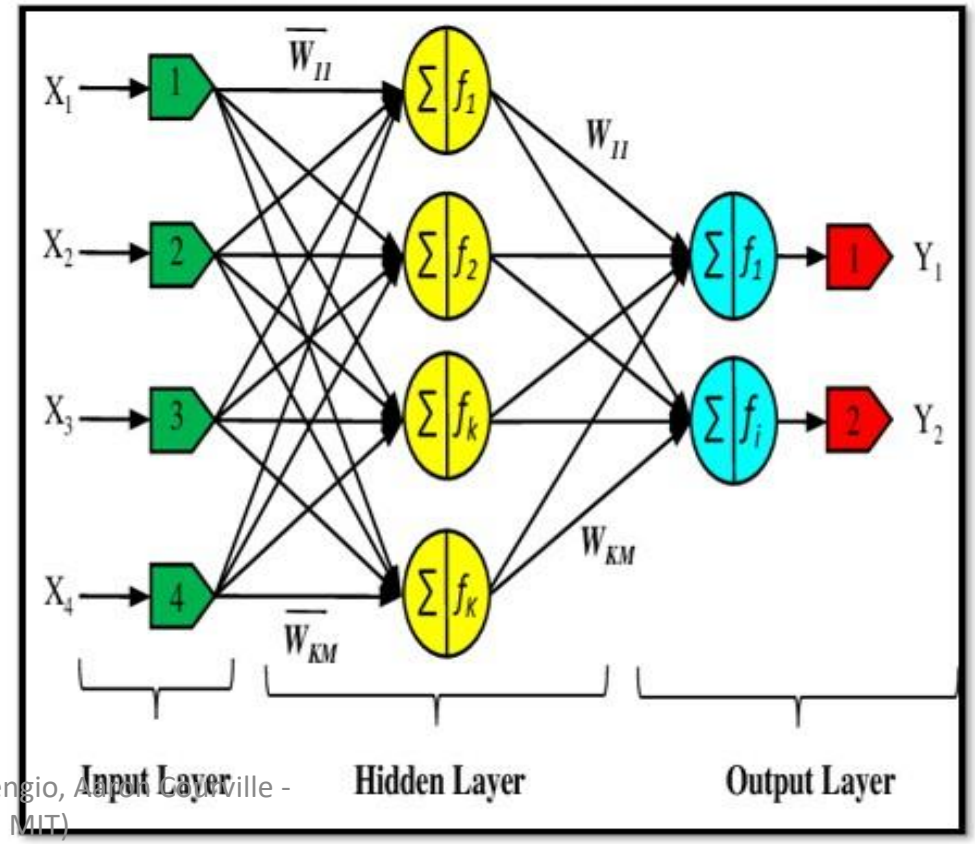
- ✓ The input layer consists of neurons representing the input features of the data.
- ✓ Each neuron in this layer represents one feature.

2) Hidden Layers:

- ✓ These are intermediate layers between the input and output layers.
- ✓ Each hidden layer consists of neurons that perform computations based on the weighted sum of inputs from the previous layer, followed by the application of an activation function.
- ✓ Multiple hidden layers allow the network to learn complex patterns in the data.

3) Output Layer:

- ✓ The output layer produces the final output of the network.
- ✓ The number of neurons in this layer depends on the nature of the problem (e.g., classification, regression) and the desired output.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- During training, the network adjusts the weights of the connections between neurons using optimization algorithms like gradient descent in order to minimize the difference between the predicted outputs and the actual outputs (i.e., minimize the loss function).
- This process is known as backpropagation.
- Multilayer feedforward networks are capable of approximating a wide range of functions and are often used in tasks such as classification, regression, and function approximation.
- However, they may suffer from issues like overfitting if not properly regularized, and choosing an appropriate architecture (number of layers, number of neurons per layer, etc.)

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1.Forward Propagation

In **forward propagation** the data flows from the input layer to the output layer, passing through any hidden layers. Each neuron in the hidden layers processes the input as follows:

1. **Weighted Sum**: The neuron computes the weighted sum of the inputs:

$$z = \sum_i w_i x_i + b$$

Where:

- x_i is the input feature.
- w_i is the corresponding weight.
- b is the bias term.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Activation Function: The weighted sum z is passed through an activation function to introduce non-linearity. Common activation functions include:

- **Sigmoid**: $\sigma(z) = \frac{1}{1+e^{-z}}$
- **ReLU (Rectified Linear Unit)**: $f(z) = \max(0, z)$
- **Tanh (Hyperbolic Tangent)**: $\tanh(z) = \frac{2}{1+e^{-2z}} - 1$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

3. Loss Function

Once the network generates an output the next step is to calculate the loss using a loss function. In supervised learning this compares the predicted output to the actual label.

For a classification problem the commonly used **binary cross-entropy** loss function is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

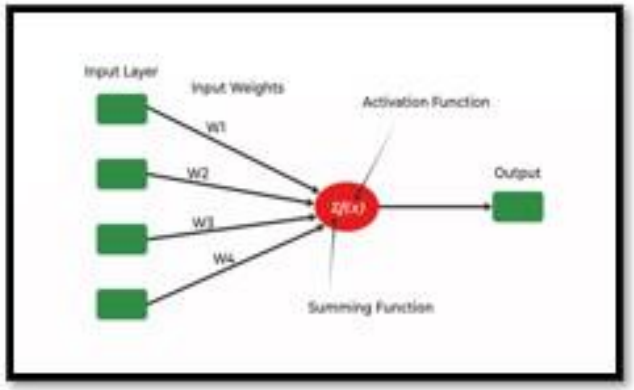
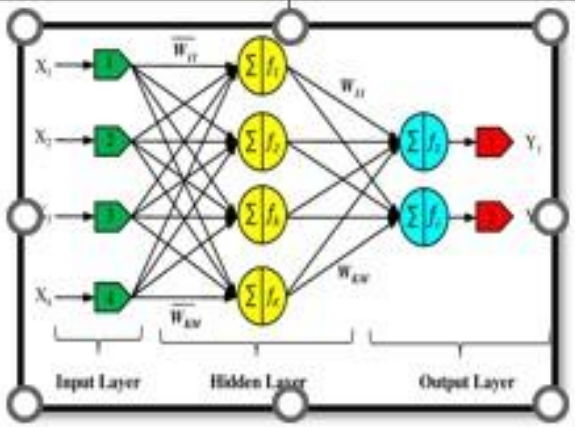
Where:

- y_i is the actual label.
- $\hat{y}_i$ is the predicted label.
- N is the number of samples.

For regression problems the **mean squared error (MSE)** is often used:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

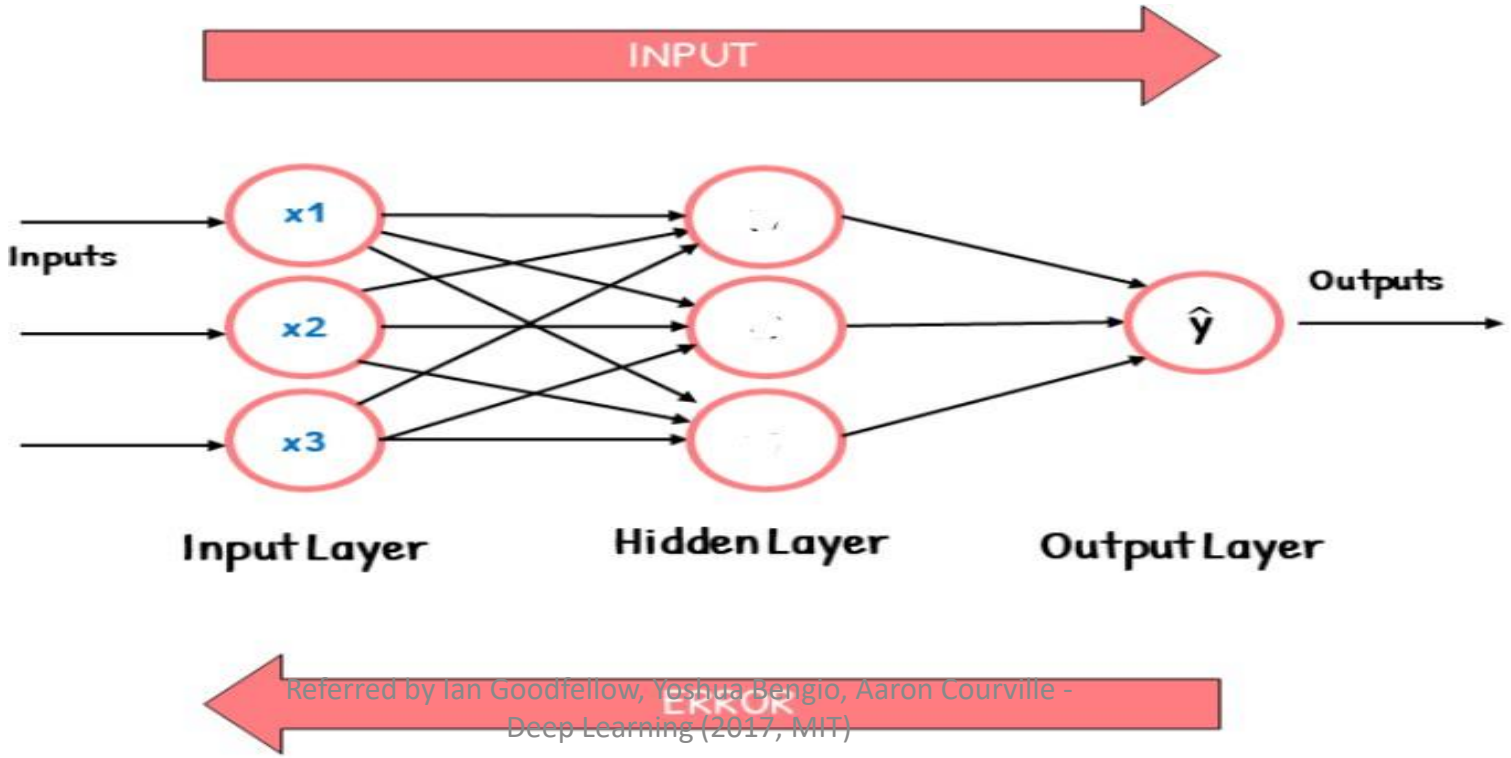
Single Layer Perceptron	Multi Layer Perceptron
Single Layer Perceptron (SLP) consists of a single layer of input nodes directly connected to the output nodes.	Multi- Layer Perceptron (MLP) consists of multiple layers of nodes, including an input layer, one or more hidden layers, and an output layer.
	
It's primarily used for binary classification problems where the data is linearly separable.	They are commonly used for a variety of tasks including classification, regression, and even in reinforcement learning.
SLPs use a simple activation function, such as the step function Identity Functions	MLP uses complex activation functions like RELU, Leaky Relu, tanh, sigmoid for introducing Non linearity in the network
They are limited in their ability to handle complex patterns and cannot model nonlinear relationships between input and output.	Hidden layers allow MLPs to learn complex patterns and relationships within the data by introducing nonlinearity through activation functions like ReLU (Rectified Linear Unit), tanh (Hyperbolic Tangent), or sigmoid. MLPs are capable of approximating any continuous function given enough hidden units and training data. This property is known as the Universal Approximation Theorem

- **Architecture**: SLP has only input and output layers, while MLP has at least one hidden layer in addition to input and output layers.
- **Complexity**: SLPs are simpler and limited in handling complex patterns, whereas MLPs can model complex relationships.
- **Function approximation**: MLPs can approximate any function, while SLPs are limited to linear separable functions.
- **Activation functions**: SLPs typically use simpler activation functions like step or Identity, while MLPs can use a variety of activation functions, allowing for nonlinear transformations.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville -
Deep Learning (2017, MIT)

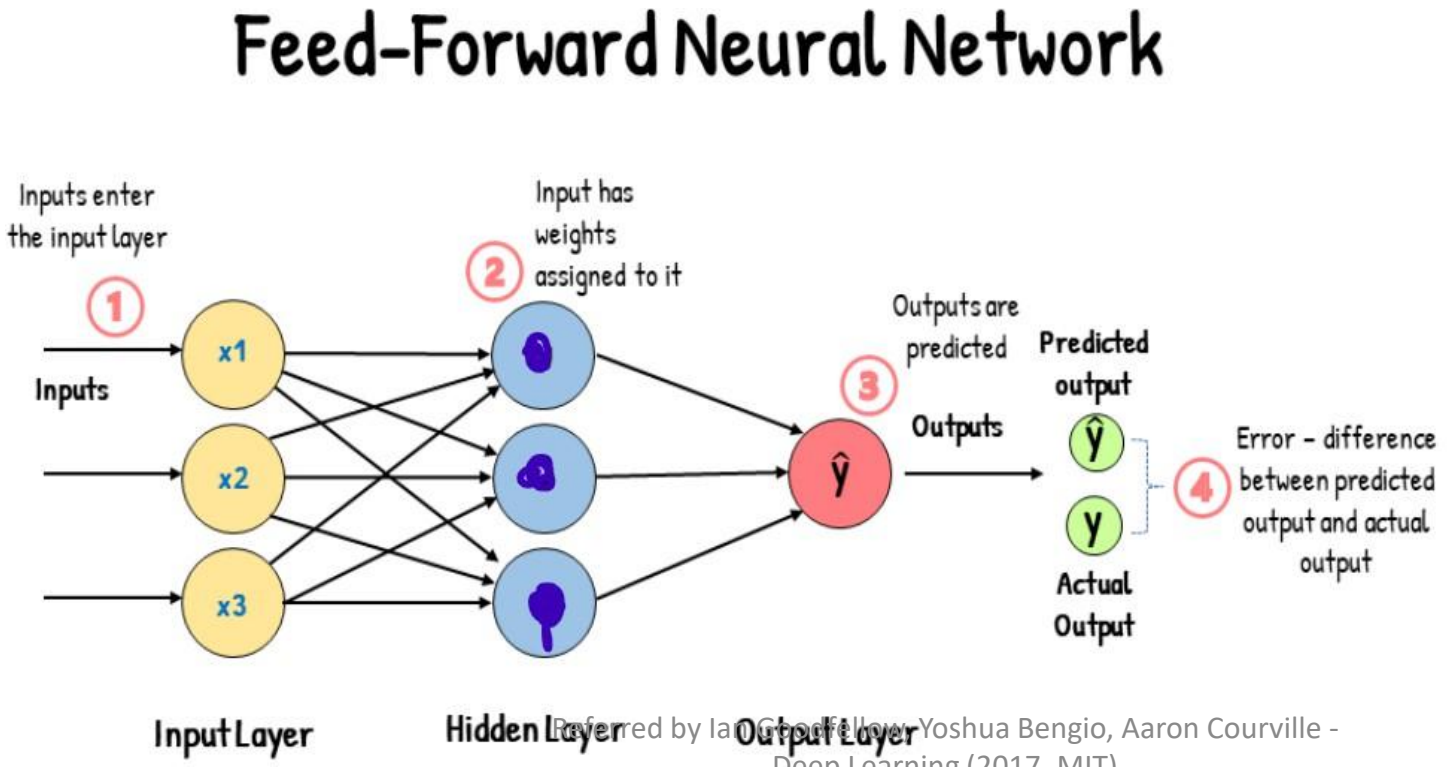
LEARNING PROCESS IN ANN – GRADIENT DESCENT AND BACK PROPOGATION

- The Backpropagation tries to **minimize** the cost function by **adjusting** the **weights and biases** of the network based on the **gradient** calculated with the gradient descent.
- It plays an important part in improving the predictions made by neural networks.
- In a feedforward neural network, the input moves forward from the input layer to the output layer.
- Backpropagation helps improve the neural network's output.
- It does this by propagating the error backward from the output layer to the input layer.

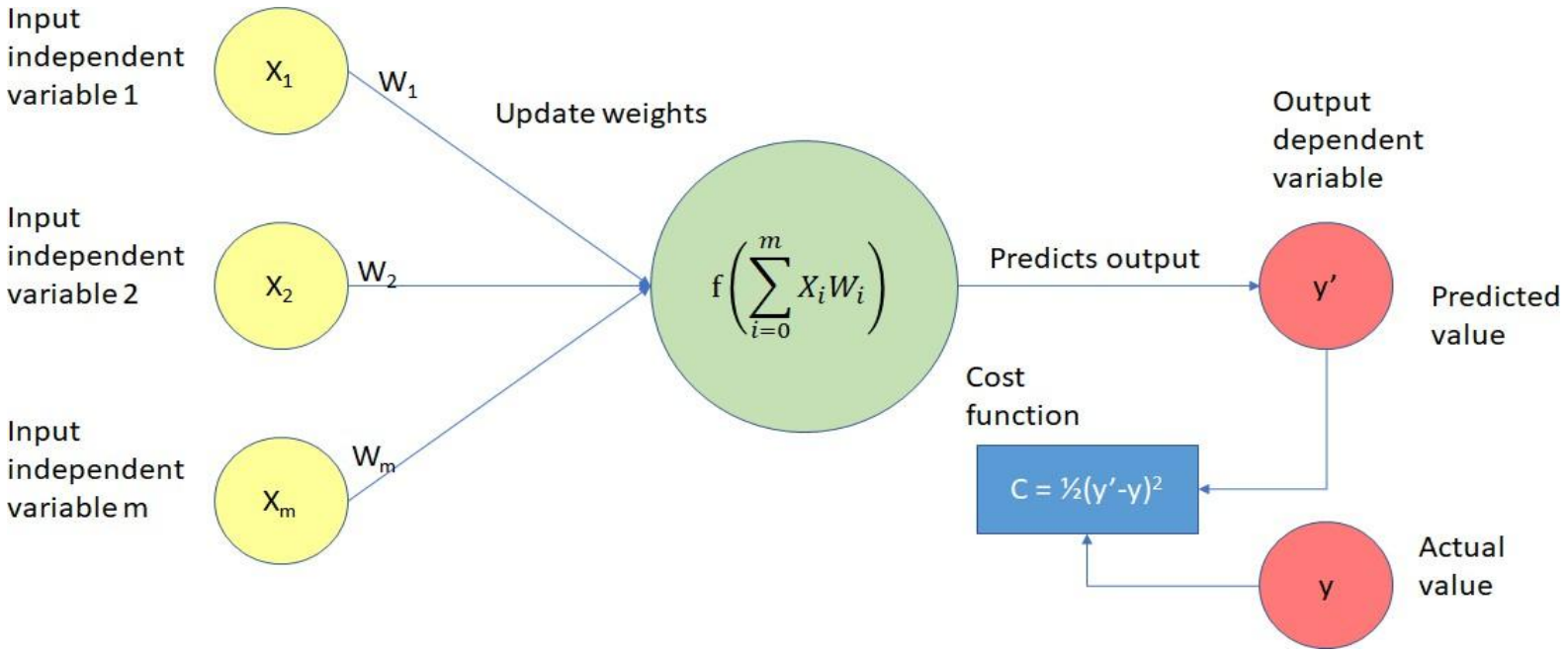


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

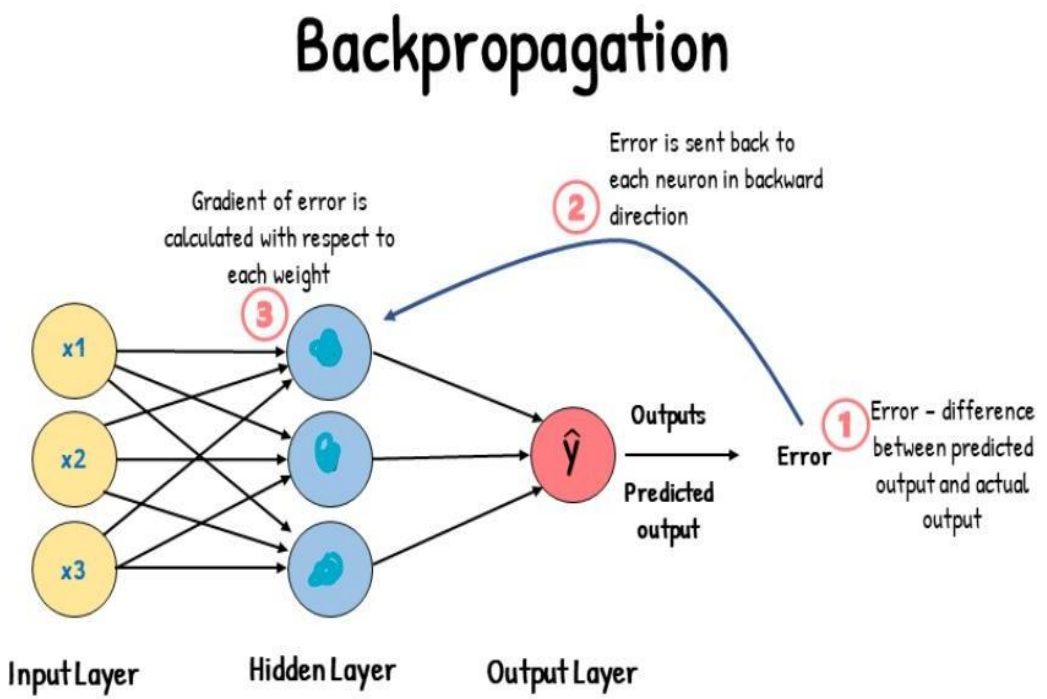
- When a neural network is first trained, it is first fed with input.
- Since the neural network isn't trained yet, we don't know which weights to use for each input.
- And so, each input is randomly assigned a weight.
- Since the weights are randomly assigned, the neural network will likely make the wrong predictions.
- It will give out the incorrect output.



❖When the neural network gives out the incorrect output, this leads to an output error.
❖This error is the difference between the actual and predicted outputs.
❖***A cost function measures this error.***



- **This is where Backpropagation comes in…**
- Backpropagation allows us to readjust our weights to reduce output error.
- The error is propagated backward during backpropagation from the output to the input layer.
- This error is then used to calculate the gradient of the cost function with respect to each weight.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- Essentially, ***backpropagation aims to calculate the negative gradient of the cost function.***
- This negative gradient is what helps in adjusting of the weights.
- ***It gives us an idea of how we need to change the weights so that we can reduce the cost function.***

Gradient Descent

- ***The weights are adjusted using a process called gradient descent.***
- Gradient descent is an optimization algorithm that is used to find the weights that minimize the cost function.
- Minimizing the cost function means getting to the minimum point of the cost function.
- So, gradient descent aims to find a weight corresponding to the cost function's minimum point.
- To find this weight, we must navigate down the cost function until we find its minimum point.

Navigating towards minimum cost function depends on 2 things

Direction - Gradient Descent or Gradient Ascent

Direction is determined by calculating the **Gradients.**

Specifically, we aim to find the negative gradient.

This is because a negative gradient indicates a decreasing slope.

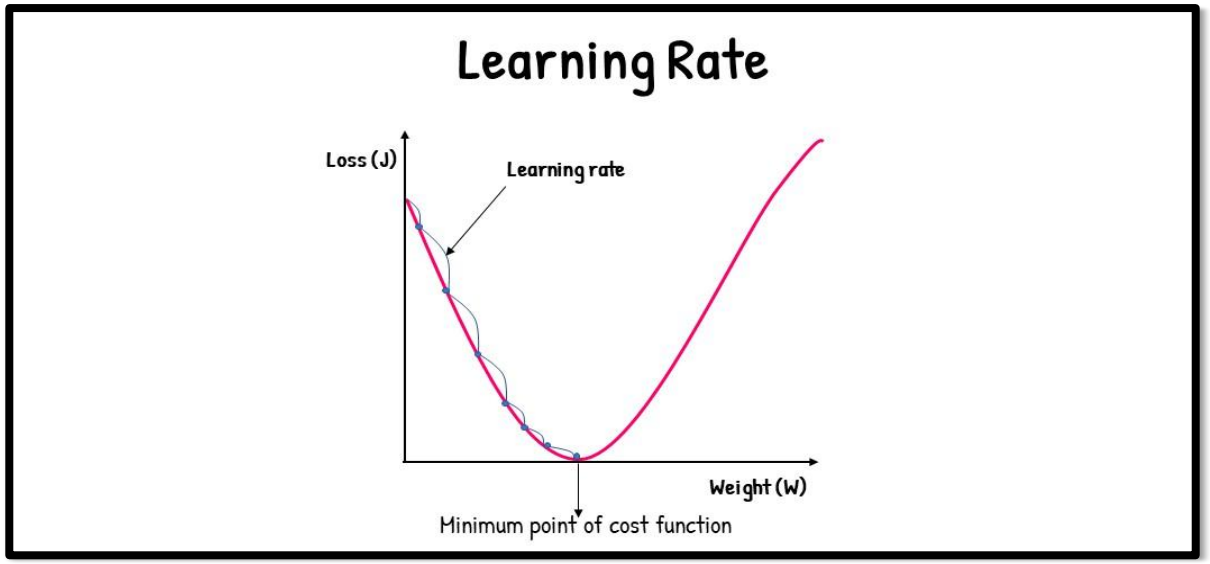
A decreasing slope means that moving downward will lead us to the minimum point.

Step Size – Learning Rate

Step Size is determined by **Learning Rate.**

The learning rate is a tuning parameter that determines the step size at each iteration of gradient descent.

1t determines the speed at which we move down the slope.



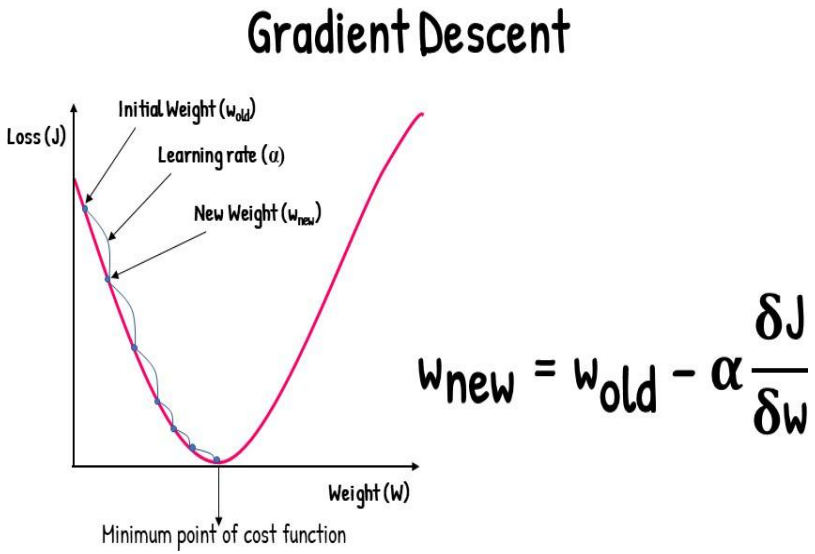
- Let's say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?
- The best way is to observe the ground and find where the land descends. From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

$$w_{new} = w_{old} - \alpha \underbrace{\frac{dJ}{dw}}_{\text{Gradient}}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Using the initial weight and the gradient and learning rate, we can determine the subsequent weights.



$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$

From the graph of the cost function, we can see that:

1. To start descending the cost function, we first initialize a random weight.
2. Then, we take a step down and obtain a new weight using the gradient and learning rate. With the gradient, we can know which direction to navigate. We can know the step size for navigating the cost function using the learning rate.
3. We are then able to obtain a new weight using the gradient descent formula.
4. We repeat this process until we reach the minimum point of the cost function.
5. Once we've reached the minimum point, we find the weights that correspond to the minimum of the cost function.

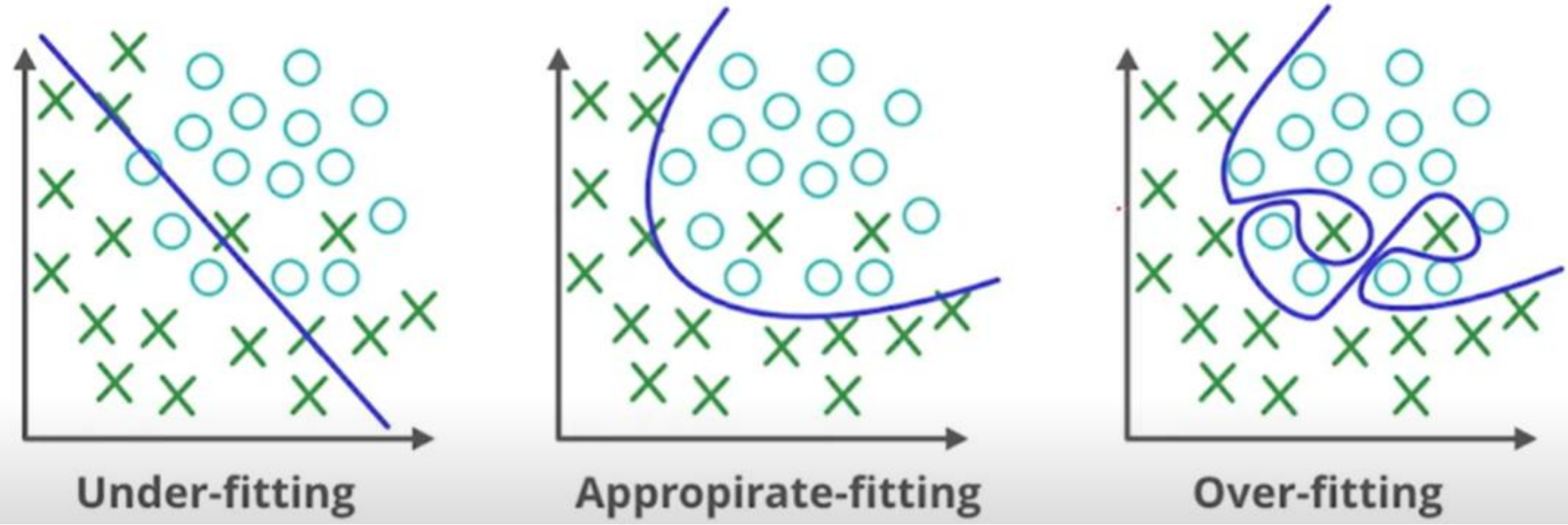
UNIT-III: REGULARIZATION AND OPTIMIZATION OF DL MODELS

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.**

Optimization for Training Deep Models: Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta-Algorithms.

Regularization

- Overfitting and underfitting are common challenges in neural networks.
- Overfitting occurs when a model performs very well on training data but fails to generalize to new or unseen data, while underfitting happens when a model performs poorly on both training and validation datasets.



Regularization in deep Learning

- Regularization is a technique used to reduce errors by fitting the function appropriately on the given training set and avoiding overfitting.
- The commonly used techniques are:

1. Lasso Regularization – L1 regularization
2. Ridge Regularization – L2 regularization

Lasso Regularization

- A regression model which uses the L1 regularization technique is called LASSO (**Least Absolute Shrinkage and Selection Operator**) regression
- **L1 Regularization** is a technique used to **reduce overfitting** in neural networks by **adding a penalty** to the loss function during training. This penalty is the **sum of the absolute values** of the weights in the model.
- In each training step, the model tries to **minimize the total loss**.
- Because large weights increase the loss (due to the penalty), the optimizer **shrinks the weights**.
- Some weights become **exactly 0** — meaning those neurons or input features are **ignored**.
- This makes the model **simpler and more efficient**.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- Let’s say the original loss function (like Mean Squared Error or Cross-Entropy) is:
- **Loss = L(y_true, y_pred)**
- With **L1 Regularization**, the new loss becomes:
-
- **Loss = L(y_true, y_pred) + λ * Σ |w|**
- **Where:**
- L(y_true, y_pred) = original loss (error between actual and predicted)
- λ (lambda) = regularization strength (a small positive number, like 0.01)
- Σ |w| = sum of absolute values of all weights in the network

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

3. $\sum |w|$

This is the **sum of the absolute values of the weights**.

- If your model has 3 weights:
  - $w_1 = 2.0$
  - $w_2 = -1.5$
  - $w_3 = 0.0$

Then:

$$\sum |w| = |2.0| + |-1.5| + |0.0| = 2.0 + 1.5 + 0.0 = 3.5$$

Let's say:

- Original loss = 0.25
- $\lambda = 0.01$
- Sum of weights ($|w|$) = 3.5

Then:

$$\text{Total Loss} = 0.25 + 0.01 \cdot 3.5 = 0.25 + 0.035 = 0.285$$

Ridge Regression

- A regression model that uses the L2 regularization technique is called Ridge regression.
- **L2 Regularization** is a technique used to **prevent overfitting** by discouraging the model from having **large weight values**.
- It adds a **penalty** to the loss function based on the **square of the weights**.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} w_i^2$$

- where
- ‘m’ number of feature
- ‘n’ number of examples
- Y_i actual target value
- $Y_i\hat{}$ predicted target value
- ‘w’ weight

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

$$\text{Loss} = L(y_{\text{true}}, y_{\text{pred}}) + \lambda \cdot \sum w^2$$

Where:

Term	Meaning
$L(y_{\text{true}}, y_{\text{pred}})$	Original loss (like MSE or Cross-Entropy)

Let's say:

- Weights: $w = [2.0, -1.5, 0.0]$
- Then: $\sum w^2 = 2.0^2 + (-1.5)^2 + 0.0^2 = 4.0 + 2.25 + 0 = 6.25$
- Lambda (λ) = 0.01
- Original Loss (e.g., MSE) = 0.25

Then:

$$\text{Total Loss} = 0.25 + 0.01 \cdot 6.25 = 0.25 + 0.0625 = 0.3125$$

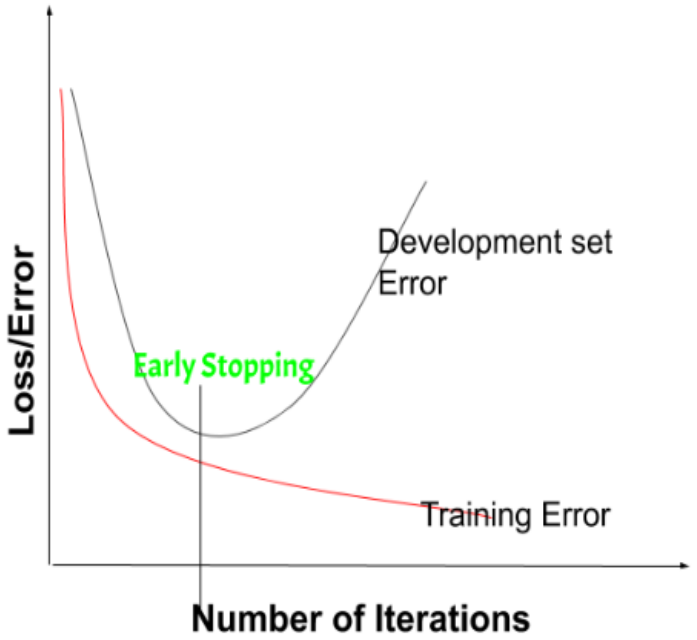
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Early Stopping

- Early stopping is a smart trick to **stop training at the right time**, before overfitting happens
- A problem with training neural networks is in the choice of the number of training epochs to use.
- Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model.

How It Works (Step-by-Step):

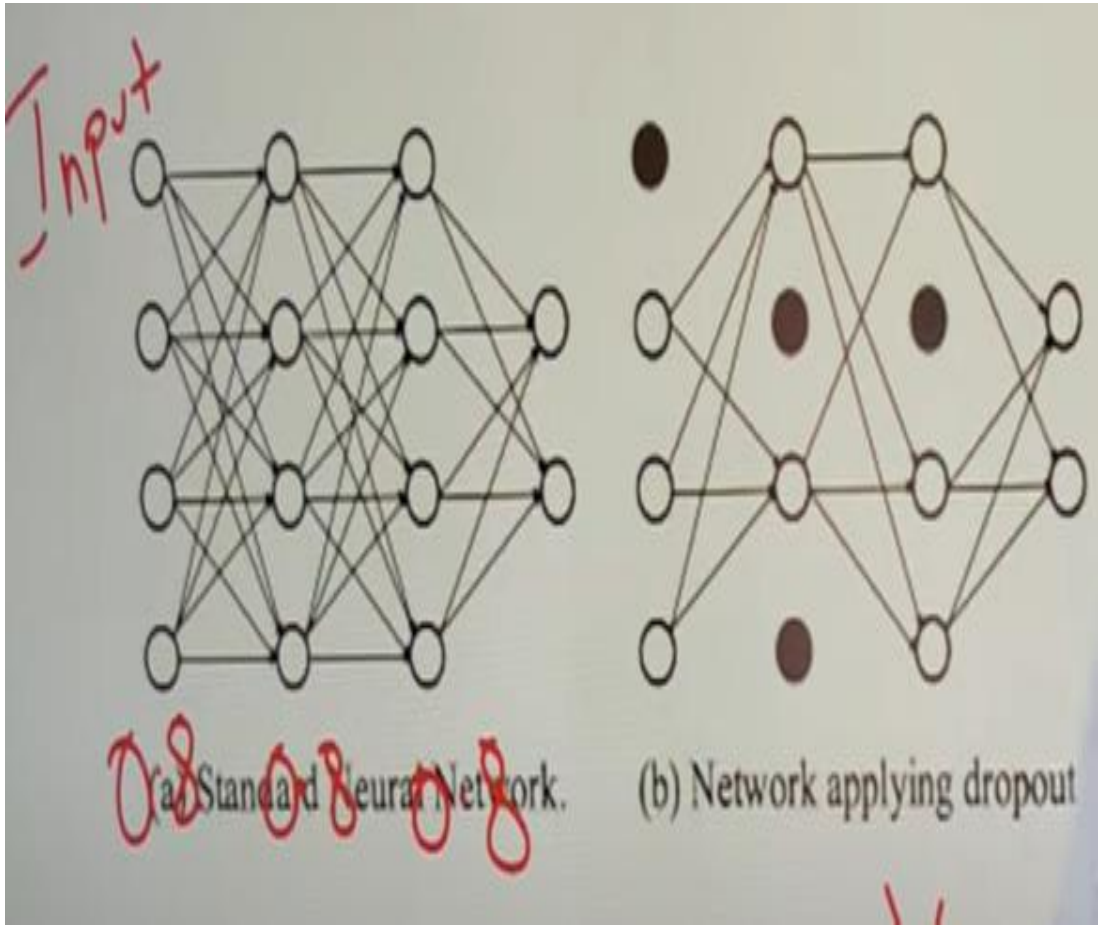
- Train your model with **many epochs** (say 100 or 200).
- After **each epoch**, check how the model performs on the **validation dataset**.
- Keep track of **validation error** (or loss).
- If the validation error **starts increasing**, it means the model is **starting to overfit**.
- Stop training at the point where validation error is **lowest**.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

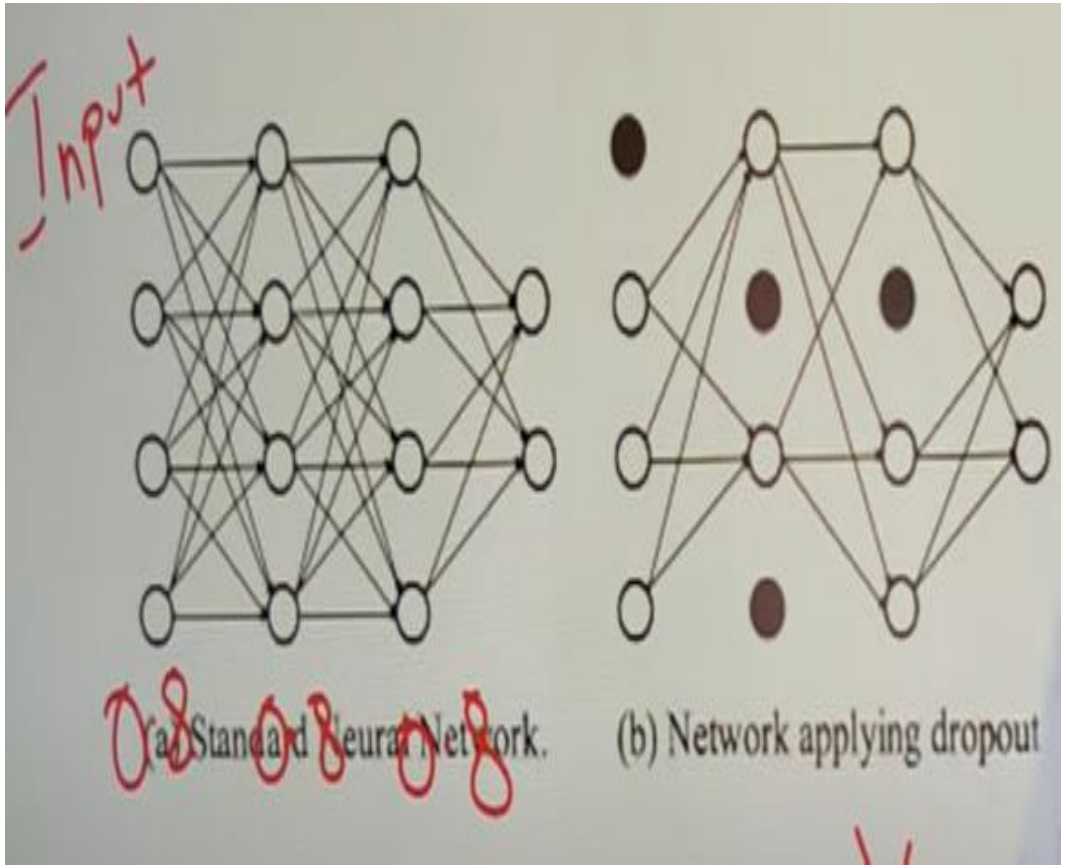
Drop Out Regularization

- Dropout regularization is a widely used technique in deep learning to prevent overfitting in neural networks.
- It works by randomly "dropping out," or setting to zero, a fraction of the units (i.e., neurons) in the neural network during training.
- This means that at each training iteration, certain neurons are temporarily removed from the network with a probability defined by the dropout rate.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Drop Out Regularization



- For n neurons , we get 2^n different Thinned Networks.
- During training, all 2^n different Thinned Networks are trained for each batch.
- While testing the original network is tested

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Drop Out Regularization

Here's how dropout regularization typically works:

- **During Training**: In each training iteration, a fraction of neurons in the hidden layers are randomly selected to be "dropped out" or temporarily removed from the network. The dropout rate is a hyperparameter that determines the probability of any particular neuron being dropped out. For example, a dropout rate of 0.5 means that each neuron has a 50% chance of being dropped out.
- **Forward Pass**: During the forward pass, the network behaves as usual, but with the selected neurons set to zero. This means that the activations and outputs of these neurons are ignored for that particular training iteration.
- **Backward Pass**: During the backward pass (backpropagation), only the active neurons (i.e., those that were not dropped out) contribute to the gradient computation. This means that the network's parameters are updated based only on the active neurons.
- **Testing Phase**: During the testing or inference phase, dropout is typically turned off, and all neurons are used. However, the weights of the connections are usually scaled by the dropout rate to account for the fact that more units are active during testing than during training.

Data Augmentation

- Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data.
- It includes making minor changes to the dataset or using deep learning to generate new data points.

Augmented vs. Synthetic data

- **Augmented data** is driven from original data with some minor changes. In the case of image augmentation, we make geometric and color space transformations (flipping, resizing, cropping, brightness, contrast) to increase the size and diversity of the training set.
- **Synthetic data** is generated artificially without using the original dataset. It often uses DNNs (Deep Neural Networks) and GANs (Generative Adversarial Networks) to generate synthetic data.

Data Augmentation

When Should You Use Data Augmentation?

1.To prevent models from overfitting.

2.The initial training set is too small.

3.To improve the model accuracy.

Data Augmentation

Image Augmentation

1.Geometric transformations: randomly flip, crop, rotate, stretch, and zoom images. You need to be careful about applying multiple transformations on the same images, as this can reduce model performance.

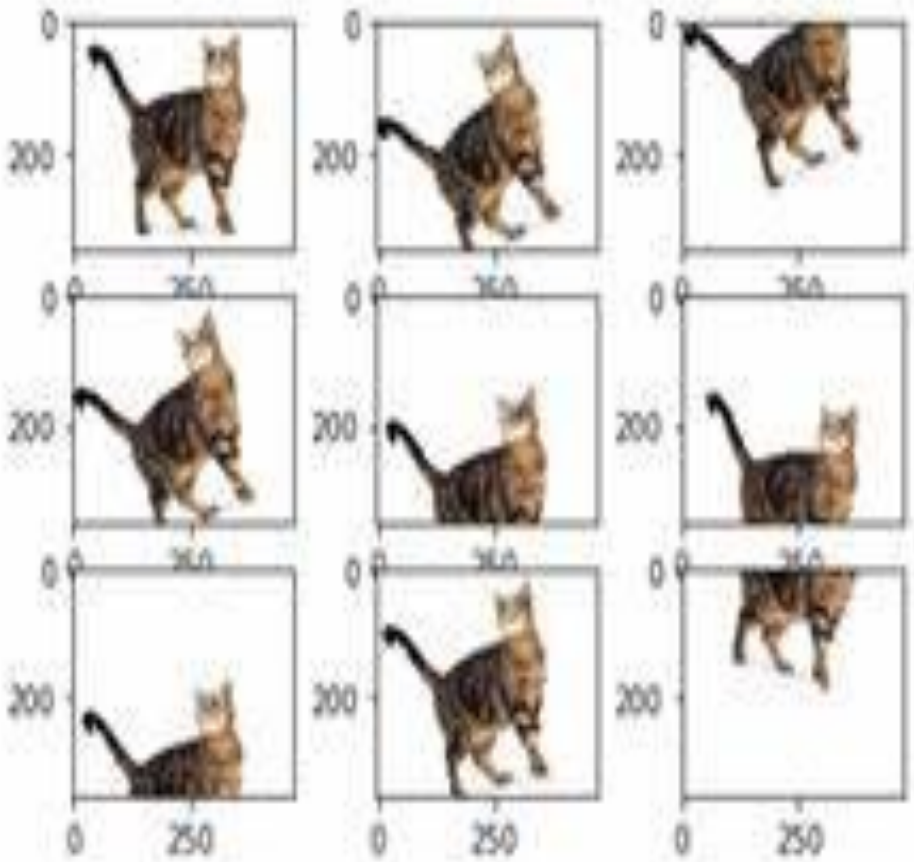
2.Color space transformations: randomly change RGB color channels, contrast, and brightness.

3.Kernel filters: randomly change the sharpness or blurring of the image.

4.Random erasing: delete some part of the initial image.

5.Mixing images: blending and mixing multiple images.

Data Augmentation



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Norm Penalties as Constrained Optimization

- In deep learning, instead of only **adding a penalty to the loss**, we can also think of regularization as **putting a limit on the size of the weights**.

- In deep learning, **regularization is usually written like this:**

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta)$$

That means:

- $J(\theta)$ =normal loss (error)
- $\Omega(\theta)$ =penalty term (like L2 norm)
- α= controls strength of regularization
- So we **add a penalty** if weights become too large.

Regularization as a Constraint

Instead of adding a penalty, we can say "**Weights must stay small.**"
So we impose a rule:

$$\Omega(\theta) \leq k$$

Meaning:
$\Omega(\theta)$is the norm of weights
$k$is the maximum allowed size

What is Lagrangian Optimization?

Lagrangian optimization is a **mathematical method used to solve problems where We want to minimize (or maximize) something**.But we also have **some restriction (constraint**)

1. General Form

Optimization Problem:

$$\min_{\theta} \quad J(\theta)$$

Subject to constraint:

$$g(\theta) \leq 0$$

Where:

$J(\theta)$ =objective function (loss)

$g(\theta)$ =constraint

2. Lagrangian Function

To solve this, we combine objective + constraint into one function:

$$\mathcal{L}(\theta, \alpha) = J(\theta) + \alpha g(\theta)$$

Where:

$\mathcal{L}$= Lagrangian

α= Lagrange multiplier (controls constraint)

Deep Learning Regularization Example Constraint form:

We want small weights:

$$\Omega(\theta) \leq k$$

So constraint function:

$$g(\theta) = \Omega(\theta) - k$$

Now Lagrangian becomes:

$$\mathcal{L}(\theta, \alpha) = J(\theta) + \alpha(\Omega(\theta) - k)$$

Role of the Lagrange Multiplier (α)

In constrained regularization, we want:

$$\Omega(\theta) \leq k$$

That means:

The norm of the weights must not exceed a limit k.

To solve this, we form the Lagrangian:

$$\mathcal{L}(\theta, \alpha) = J(\theta) + \alpha(\Omega(\theta) - k)$$

Here, **α (alpha)** is called the **Lagrange multiplier**.

How α Controls the Constraint

Case 1: When weights become too large

If:

$$\Omega(\theta) > k$$

Then:

The constraint is violated

The term $(\Omega(\theta) - k)$becomes positive

The optimizer increases α

So:

α becomes large

This creates a **strong penalty force** that pushes the weights down:

→ *weights shrink*

Case 2: When weights are already small

If:

$$\Omega(\theta) < k$$

Then:

- The constraint is satisfied
- No violation happens
- α does not need to grow

So:

$$\alpha \textit{ becomes small}$$

Thus:

$$\rightarrow \textit{ no unnecessary shrinking}$$

Final Key Idea

So, α acts like an automatic controller:

- Large α → strong regularization (weights reduced)
- Small α → weak regularization (weights unchanged)

Why Constraints Are Better Than Large Penalties

1. Avoid Dead Neurons

If penalties are very large, weights may become almost zero.

Then some neurons stop learning and become **dead**.

Constraints do not force weights to zero.

They only **limit how large the weights can be**.

2. Better Training Stability

With large learning rates:

weights may grow very big

gradients also become large

this can cause numerical problems

Constraints prevent this by:

limiting weight size

projecting weights back into the allowed region

This keeps training **stable**.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Regularization and Under-Constrained Problems

Under-Constrained Problem

A learning problem is called **under-constrained** when the model has **too many parameters but t**he dataset has **too little information.**So the model is not forced to choose one best solution.

Example:

If a neural network has:

1 million weights

But only:

10,000 training examples

Then the model has too much freedom.

Simple Example

Imagine you have only **2 training points**:

Now you ask:

“Draw a curve that passes through both points.”

You can draw: a straight line or a curve or a zigzag or infinite shapes but All of them fit perfectly.so the solution is **not unique**.

2. Why Does Under-Constrained Happen in Deep Learning?

- Deep learning models are usually:
- Very large
- Highly flexible
- So they can easily achieve:

$$\text{Training } error \approx 0$$

Even without learning the true pattern.

3. Problem: Many Possible Solutions

In an under-constrained system:

$$J(\theta)$$

can be minimized by many different values of θ

So the solution is not unique:

$$\theta_1, \theta_2, \theta_3, \ldots \textit{ all give same loss}$$

4. What Happens Without Regularization?

Without regularization:

- Weights may become very large
- Model may memorize training data :**Instead of learning patterns, it just remembers answers**.
- Poor performance on test data

This leads to:

Overfitting

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Data Augmentation

- Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data.
- It includes making minor changes to the dataset or using deep learning to generate new data points.

Augmented vs. Synthetic data

- **Augmented data** is driven from original data with some minor changes. In the case of image augmentation, we make geometric and color space transformations (flipping, resizing, cropping, brightness, contrast) to increase the size and diversity of the training set.
- **Synthetic data** is generated artificially without using the original dataset. It often uses DNNs (Deep Neural Networks) and GANs (Generative Adversarial Networks) to generate synthetic data.

Data Augmentation

Image Augmentation

1.Geometric transformations: randomly flip, crop, rotate, stretch, and zoom images. You need to be careful about applying multiple transformations on the same images, as this can reduce model performance.

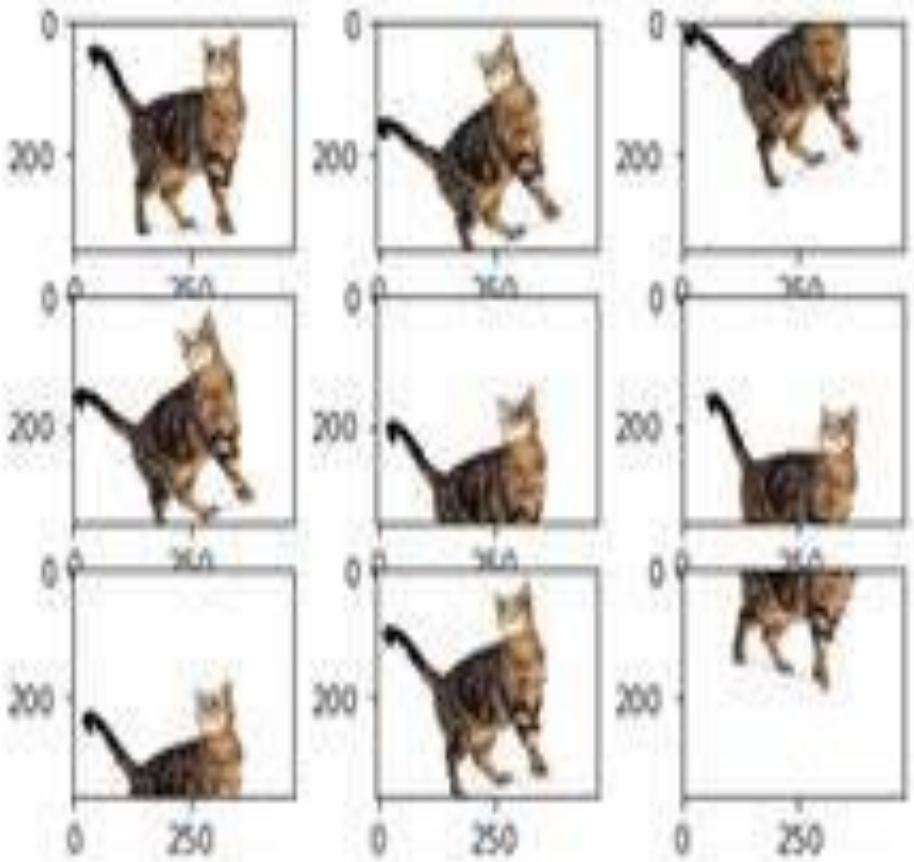
2.Color space transformations: randomly change RGB color channels, contrast, and brightness.

3.Kernel filters: randomly change the sharpness or blurring of the image.

4.Random erasing: delete some part of the initial image.

5.Mixing images: blending and mixing multiple images.

Data Augmentation



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Noise Robustness (Noise Injection)

1. Noise Applied to Inputs (Data Augmentation)

- Applying noise to the **input data** is a form of **data augmentation**.
  For some models, adding noise with **very small variance** to the input is equivalent to adding a **penalty on the norm of the weights**.

This means:

•The model learns to be less sensitive to small changes in input

•It encourages smoother functions

•It improves generalization

- Thus, input noise acts as a **regularizer**.

2. Noise Applied to Hidden Units

- Noise can also be applied to the **hidden units** of a neural network.
  Injecting noise at hidden layers is **more powerful than simply shrinking the parameters**.
- This is because:
- It forces neurons to learn robust representations
- It prevents strong dependence on specific hidden units
- This idea is so important that **Dropout** was developed as a main method based on noise injection in hidden units.

3. Adding Noise to Weights

- Noise can also be added directly to the **model weights**.
  This technique is mainly used in **Recurrent Neural Networks (RNNs)**.
- Adding noise to weights can be interpreted as a **stochastic implementation of Bayesian inference** over the weights.
- In Bayesian learning:

•**Weights are considered uncertain**

•They are represented using a probability distribution $p(w)$

•This distribution reflects uncertainty in the model parameters

- Injecting noise into weights is a **practical and stochastic way** to represent this uncertainty during training.

4. Injecting Noise at the Output Targets (Label Noise)

- Most real-world datasets contain **errors in their labels**.
  Maximizing $\log p\,(y \mid x)$is harmful when the label $y$is incorrect.
- To handle this, noise can be injected into the **output targets** through the cost function

Semi-Supervised Learning

Meaning

Semi-supervised learning is a training approach where a model learns from:

- A small amount of **labeled data**
- A large amount of **unlabeled data**

Thus, it combines both supervised and unsupervised learning.

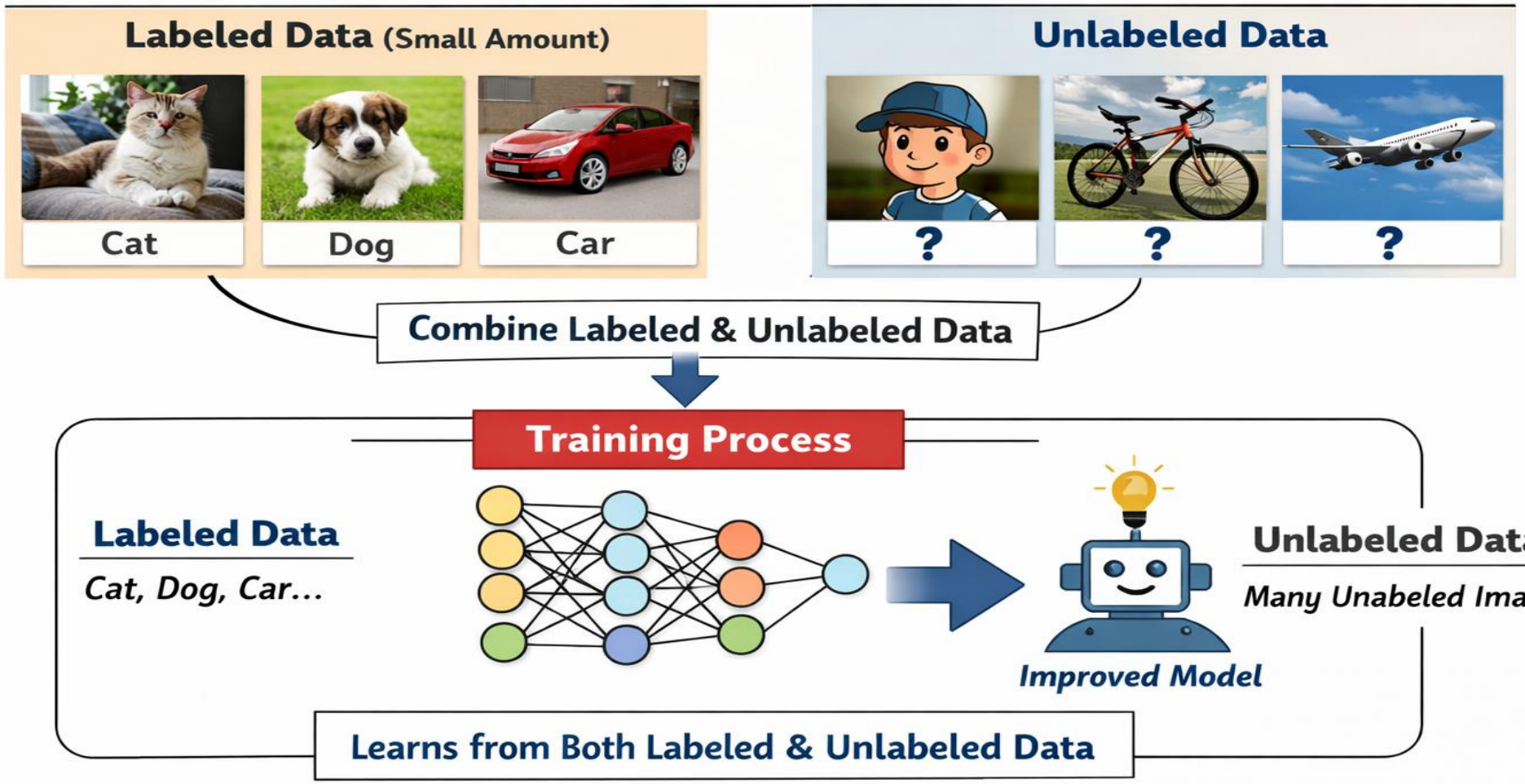
Why is it Useful?

- Labeling data requires time, effort, and cost.
- Unlabeled data is easily available in large quantities.

So, semi-supervised learning improves model performance by using extra unlabeled examples to learn better patterns and generalize well.

Key Point: It helps reduce overfitting when labeled data is limited.

Semi-Supervised Learning



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Multi-Task Learning(MTL) for Deep Learning

Multi-Task Learning (MTL) is a **sub-field of Machine Learning/Deep Learning**.

Multi-Task Learning (MTL) is a deep learning approach in which a **single model is trained** to perform **multiple tasks simultaneously**.

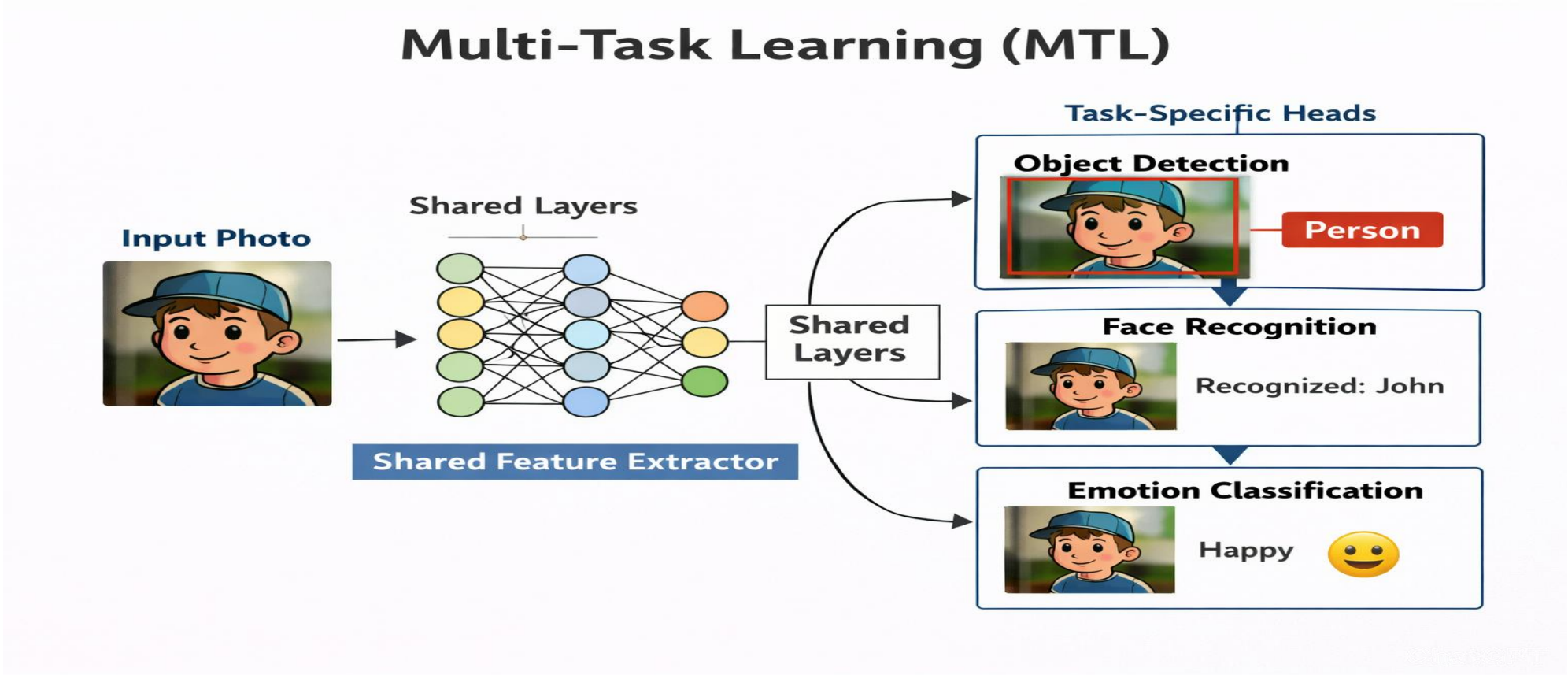
In deep learning, MTL refers to training a neural network to **solve multiple related tasks** by **sharing some layers and parameters** across tasks.

The main goal is to improve the **generalization performance** of the model by leveraging shared information between tasks.
Instead of learning each task separately, the model learns a common representation that benefits all tasks.

Intuition Behind MTL:

- Deep learning models aim to **learn good representations of input features** to predict outputs.
- Standard approach: optimize a function for a **single task** by training and fine-tuning the model.
- MTL: **forces the model to learn generalized representations** because weights are updated for multiple tasks simultaneously.
- Analogy to humans: **we learn better when studying multiple related tasks instead of focusing on only one**.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Shared Feature Extractor

A part of the network shared across all tasks to extract useful features.

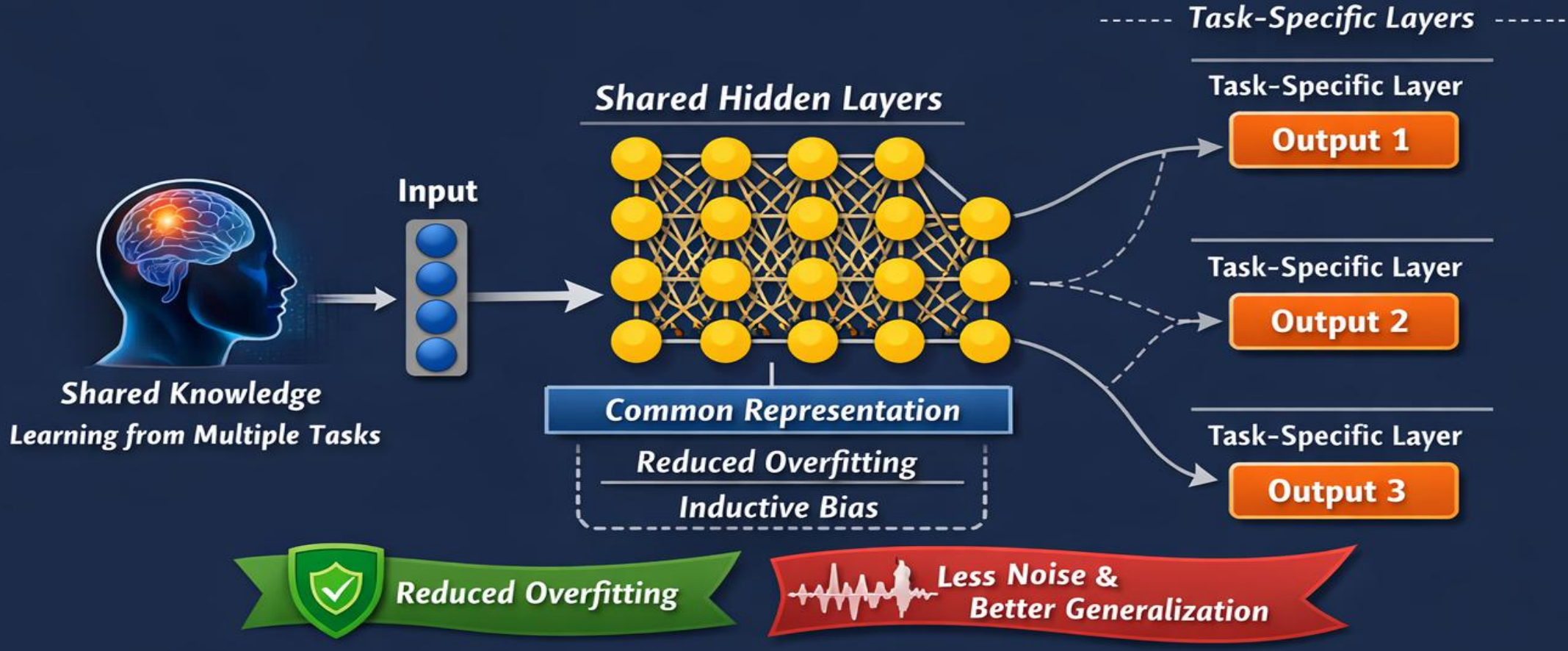
Task-Specific Heads

Separate output layers for each task, used to make predictions.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Multi-Task Learning (MTL)

Hard Parameter Sharing



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Common Architecture of Multi-Task Learning (MTL)

The most common implementation of Multi-Task Learning uses two main components:

1. Shared Feature Extractor

This part of the neural network is **common for all tasks**.

It learns shared features from the input data such as:

- edges
- shapes
- patterns
- high-level representations

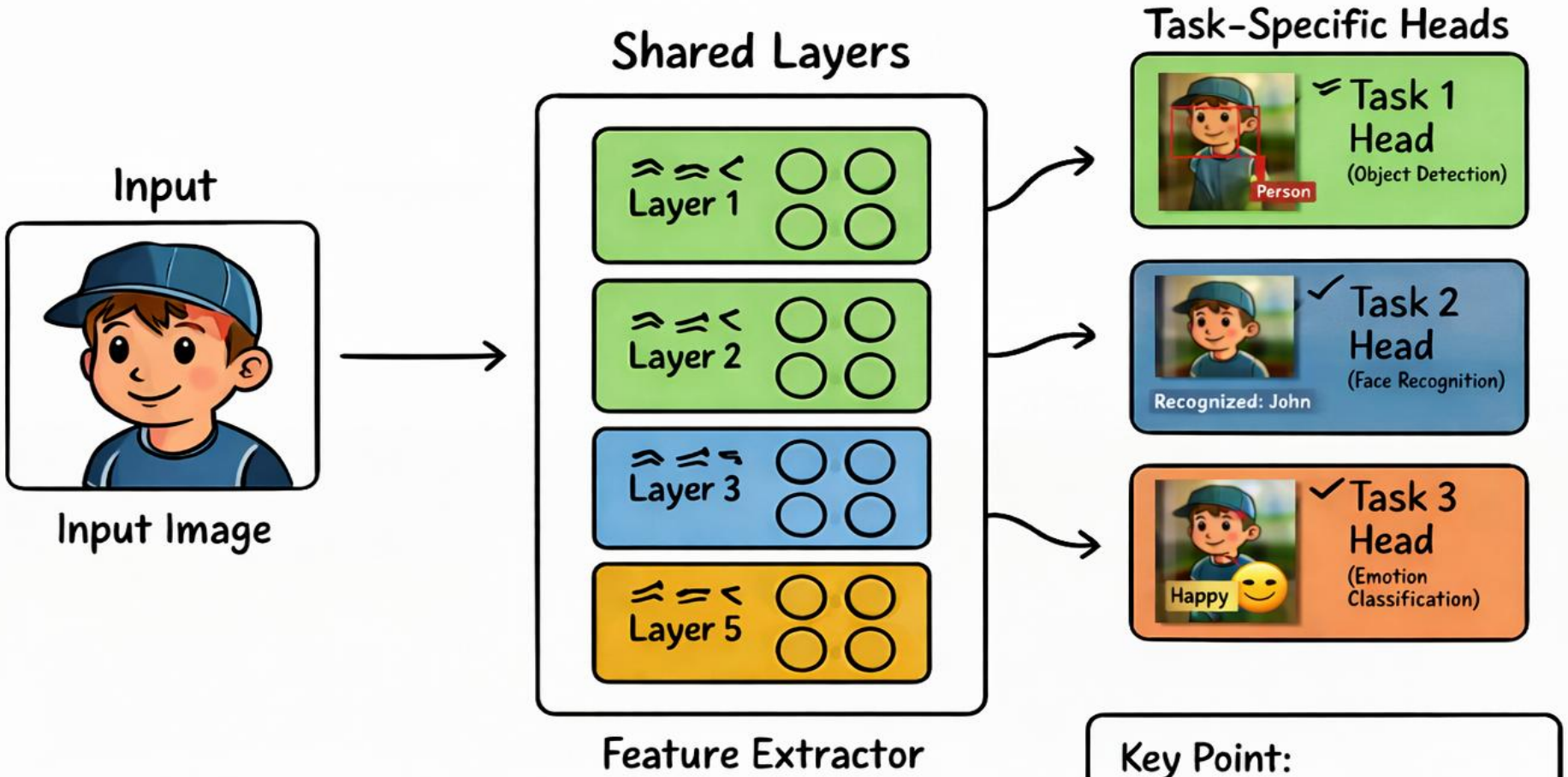
So, instead of learning separately for each task, the model learns a **common feature space**.

2. Task-Specific Heads

- After shared layers, the network splits into multiple branches.
- Each branch is called a **task-specific head**.
- Each head is responsible for:
- making predictions for one particular task
- producing separate outputs

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Common Architecture of Multi-Task Learning (MTL)



Input ⟶ Shared Layers ⟶ { Task 1 Head, Task 2 Head, Task 3 Head }

Key Point:
Shared layers reduce overfitting and improve generalization.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Major Techniques in Multi-Task Learning

1.Hard Parameter Sharing

- A common set of hidden layers is shared across all tasks.
- Only the final output layers are task-specific.

Architecture

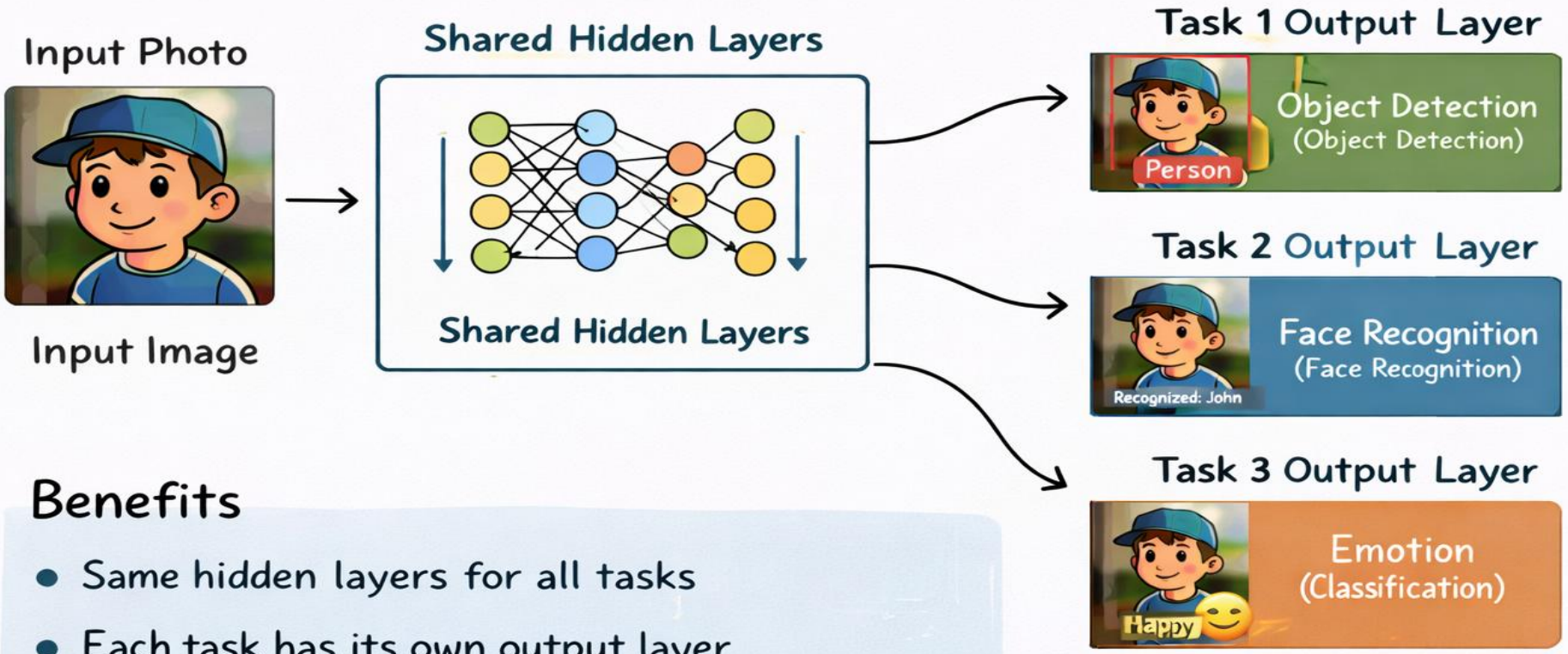
$$\text{Input} \rightarrow \textit{Shared Hidden Layers} \rightarrow \begin{cases} \textit{Task 1 Output Head} \\ \textit{Task 2 Output Head} \\ \textit{Task 3 Output Head} \end{cases}$$

Advantages

Strongly reduces overfitting
Forces tasks to learn a shared representation
Encourages features useful across multiple related tasks
Most commonly used form of MTL

Hard Parameter Sharing

A common hidden layer is shared between tasks while each task has its own task-specific output layer.



Benefits

- Same hidden layers for all tasks
- Each task has its own output layer
- Reduces overfitting and encourags learning common features

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2.Soft Parameter Sharing (Multi-Task Learning)

- Soft Parameter Sharing is a Multi-Task Learning technique where:
- Each task has its **own separate neural network**
- Each network has its **own weights and biases**
- So, unlike hard sharing, tasks do **not** use the exact same layers.

Mechanism

Suppose two tasks have parameters:

Task 1: θ_1

Task 2: θ_2

We add a penalty:

$$|| \theta_1 - \theta_2 ||$$

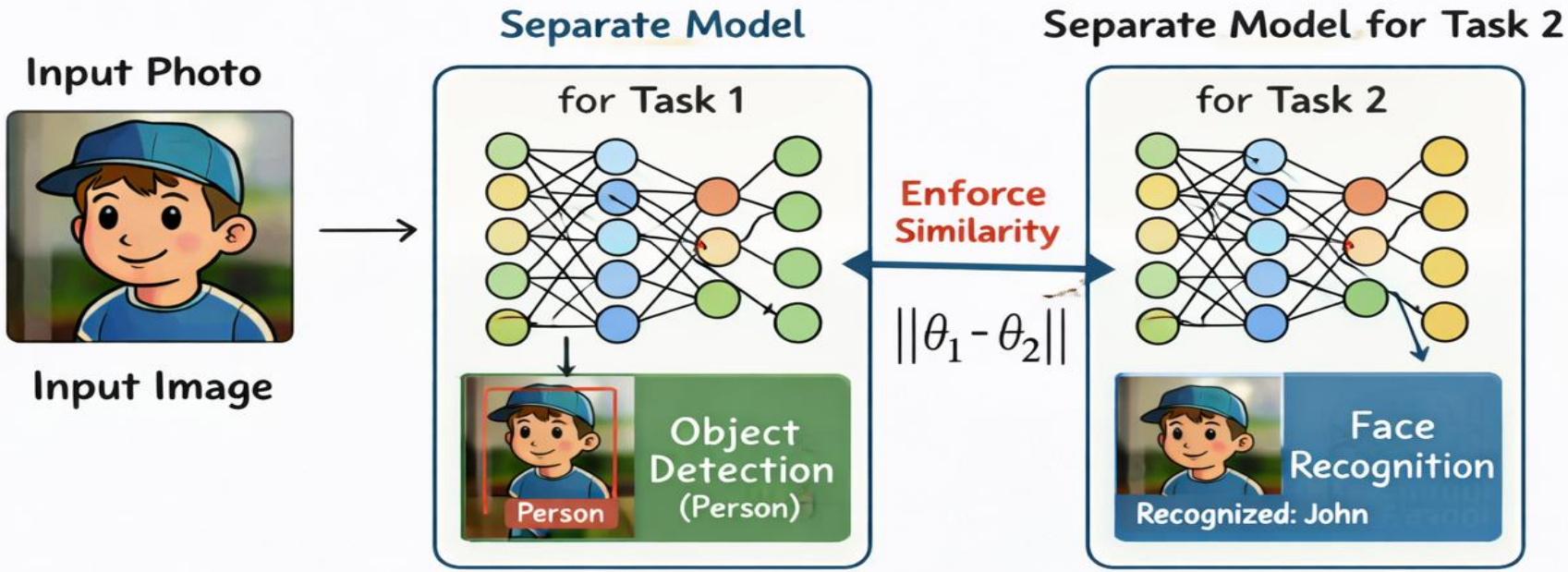
This forces:

Parameters to stay close

Knowledge to be shared across tasks

Soft Parameter Sharing

Each task has its own separate model, but similarity between task parameters is encouraged.



Benefits

- Allows separate models for each task
- Promotes knowledge sharing
- Acts as a reglarizer
- Reduces overfitting

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Parameter Sharing and Parameter Tying

1. Parameter Sharing

•**Definition**: Using the **same parameters (weights and biases)** across **multiple parts of a network** or **across multiple tasks.**

•**Purpose**:

- Reduces the **number of parameters** in the model.
- Helps the model **generalize better** by forcing it to learn representations that are useful for multiple tasks.

•**Example in MTL**:

- **Hard Parameter Sharing**: shared hidden layers across multiple tasks.
- **Soft Parameter Sharing**: separate models with parameters regularized to be similar.

2. Parameter Tying

•**Definition**: **Parameter Tying** means **using the same parameters (weights) in multiple parts of a neural network** instead of learning separate weights for each part.

•**Purpose**:

- Ensures **structural or functional symmetry** in the network.
- Reduces model complexity while enforcing **stronger constraints** than general sharing.

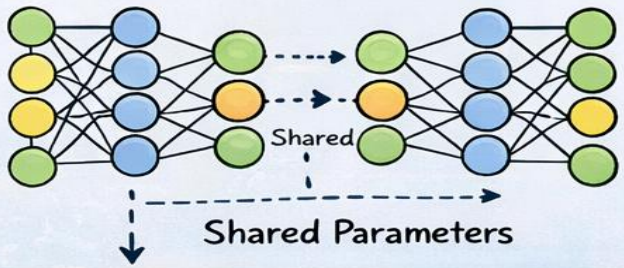
•**Example**:

- **Convolutional Neural Networks (CNNs)**: the same filter (kernel) weights are applied across different regions of an image (spatially tied weights).
- **Recurrent Neural Networks (RNNs)**: same weights are used across different time steps.

Parameter Sharing vs Parameter Tying

A common hidden layer is shared between tasks while each task has its own task-specific output layer.

Parameter Sharing



✓ Separate copies of the same network layers

θ_1

0.8	-0.2	0.5
0.1	0.6	-0.1
-0.5	0.4	0.9

$\rightarrow \theta_2$

0.8	-0.2	0.5
0.1	0.6	-0.1
-0.5	0.4	0.9

Separate Copies $\theta_1 \neq \theta_2$

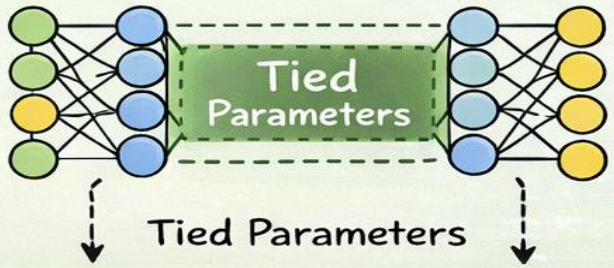
$\theta_1 \neq \theta_2 \quad \theta_1 \neq \theta_2$

Separate Copies

$\theta_1 \neq \theta_2$

VS

Parameter Tying



✓ Single shared instance of network layers

0,8	0,	
0,8	0,	

$\rightarrow$

0.8	-0.2	0.5
0.1	0.6	-0.1
-0.5	0.4	0.9

Single Instance

$\theta_1 = \theta_2 = \theta$

Single Instance

$\theta_1 = \theta_2 = \theta$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Sparse representation

- A **sparse representation** means A model represents data using only a small number of active (non-zero) features or neurons. So, instead of using **many neurons**, only **a few important ones** are activated.

Simple Example

- Imagine you have a vocabulary of 10,000 words.
- A sentence like:"Deep learning is powerful" .will use only **3–4 words**, not all 10,000. That is a **sparse representation**.

In Neural Networks

In sparse representation means Most neurons output **0 ,**Only a few neurons output **strong values**

$$h = [0, 0, 0, 5.2, 0, 0, 1.8, 0, 0]$$

Here, only 2 neurons are active → **sparse**

Why Sparse Representations are Useful?

A **sparse representation** means:

Out of many neurons/features, **only a few are active (non-zero)** at a time.

1. Efficient Feature Learning

Sparse models learn only the **most useful features** instead of using everything.

Focuses on key patterns

Ignores noise/unimportant details

Example:

Instead of learning 100 random features, it learns 5 strong ones.

2. Better Generalization (Less Overfitting)

- When fewer neurons are active:
- Model becomes simpler
- Doesn’t memorize training data
- Performs better on new data
- Sparse = less chance of overfitting

3. Interpretability

- Sparse features are easier to understand.
- Each neuron often learns a **meaningful pattern**, like:
- One neuron → detects **edges**
- Another → detects **eyes**
- Another → detects **faces**
- Another → detects **objects**
- So sparse models are more explainable.

4. Reduced Complexity

- Since most values are zero:
- Less computation
- Less memory/storage
- Faster processing
- This is very useful in big neural networks.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Bagging and Other Ensemble Methods

What is an Ensemble Method?

- An **ensemble method** is a machine learning technique in which **Multiple models (learners) are combined** to solve the same problem.
- Instead of relying on a single model, ensemble learning builds several models and merges their predictions to improve overall performance

1. Bagging (Bootstrap Aggregating)

Bagging is an ensemble technique where Many models are trained **independently .**Each model uses a different random sample of the dataset .Final output is the **average or majority vote**

Steps of Bagging

- Create multiple datasets using **bootstrap sampling**

  (random sampling with replacement)

- Train separate models on each sample
- Combine predictions:
- **Classification → Majority Voting**
- **Regression → Averaging**

Why Bagging is Useful?

✓ Reduces variance

✓ Prevents overfitting

✓ Improves stability

✓ Works well with noisy data

Example: Random Forest

Random Forest is the most famous bagging method.

•Uses many decision trees

•Combines their results

Decision Tree alone → overfits

Random Forest → strong and stable

2. Boosting

Boosting trains models **sequentially**, not independently.
Each new model focuses on correcting the mistakes of the previous one.

Steps:

1.Train first weak model
2.Identify errors
3.Train next model to fix errors
4.Combine all models

Popular Boosting Algorithms

- AdaBoost
- Gradient Boosting
- XGBoost

Boosting Advantage

✓ Reduces bias
✓ High accuracy
⚠ But more prone to overfitting if not controlled

3. Stacking

Stacking combines different types of models:

- Model 1 → Decision Tree
- Model 2 → SVM
- Model 3 → Neural Network

Then a **meta-model** learns how to best combine them.

Advantage

✓ Very powerful

✓ Uses strengths of multiple algorithms

4. Voting Ensemble

Multiple models predict, and final result is chosen by voting.

Types:

- Hard Voting → majority class
- Soft Voting → average probability

Tangent Distance

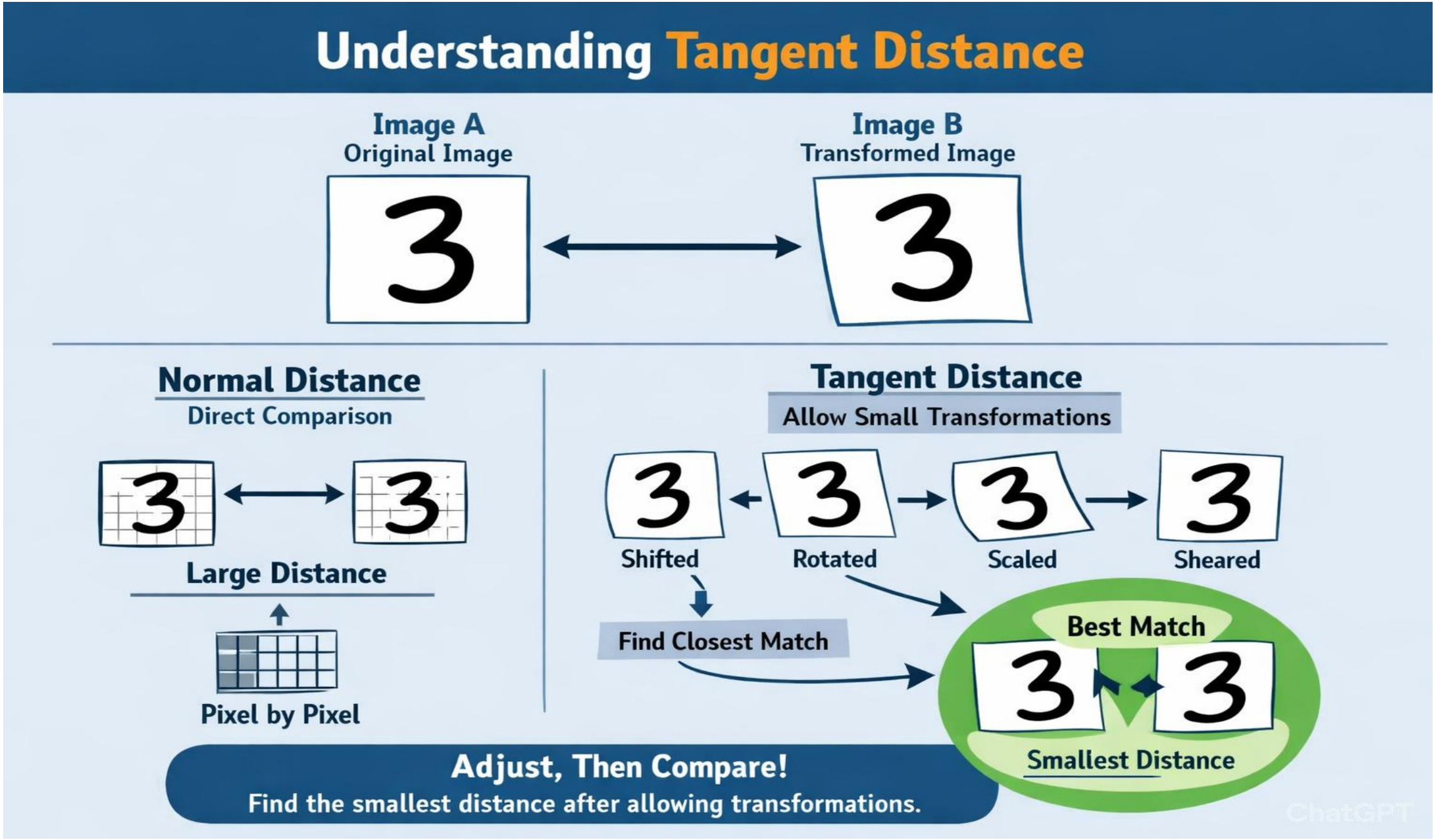
What is the Problem?

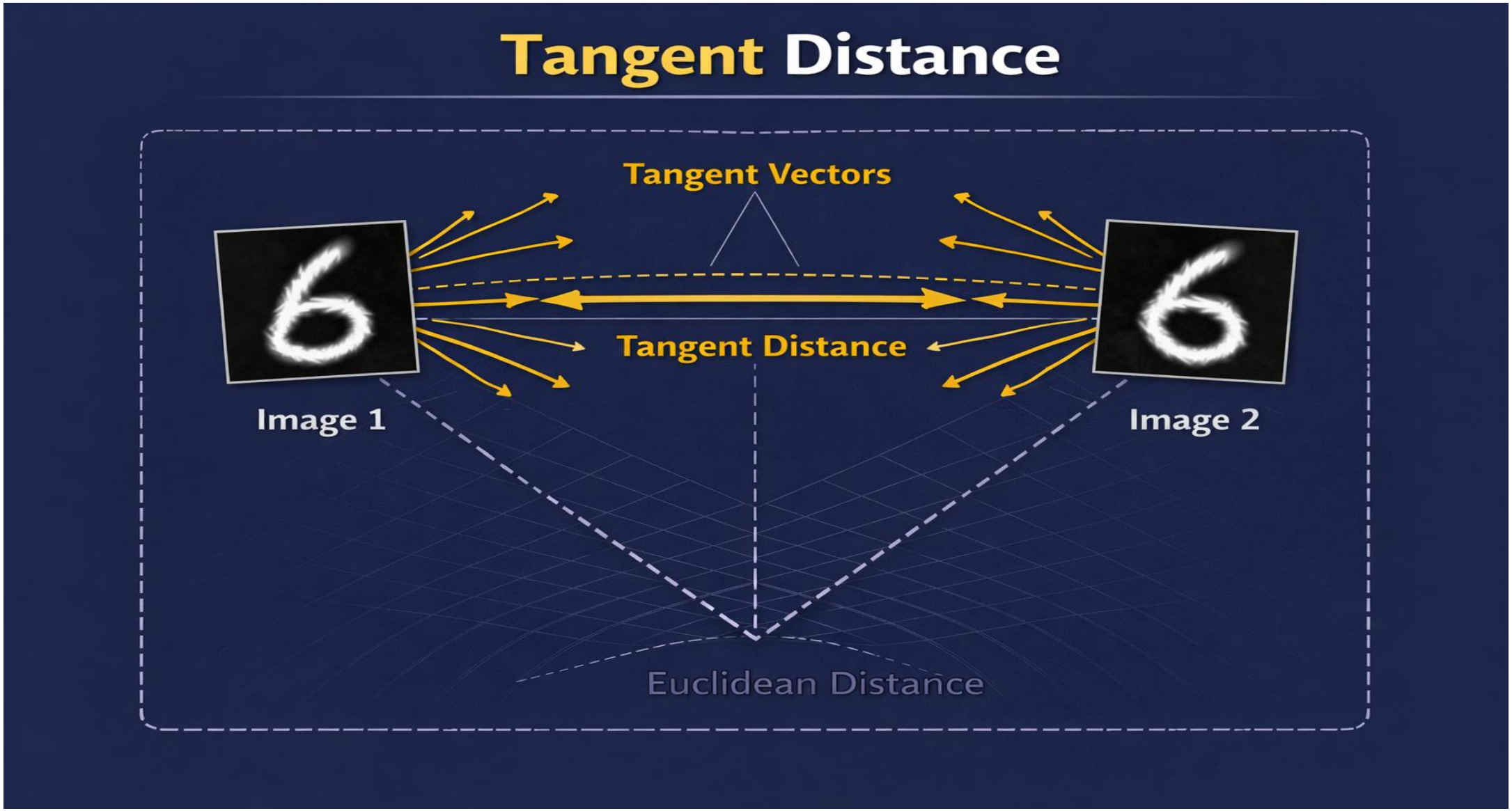
- Suppose you have two handwritten digit images:
- Image A → digit **“3”**
- Image B → same digit **“3”**, but slightly rotated or shifted
- If we use **normal Euclidean distance**, the computer compares pixels **one by one**.
- Even if both are the same digit,because one is slightly rotated or shifted → the pixel values change → the distance becomes large **But for humans, they are clearly the same digit**

Tangent Distance

- Tangent Distance is a similarity measure mainly used in image recognition.
- Unlike Euclidean distance, it **does not directly compare images pixel by pixel**.
- It allows **small transformations** such as **rotation, translation, scaling, and shearing before computing distance**.
- It assumes **two images may represent the same object** but look slightly different due to minor distortions.
- It generates slightly transformed versions of one image for comparison.
- It computes the minimum distance between the other image and these transformed versions.
- It uses a tangent (linear) approximation to model small transformations efficiently.
- It avoids checking all possible transformations, which would be computationally expensive.
- It is more robust to small variations in images.
- It is commonly used in handwriting recognition and pattern classification tasks.

Understanding Tangent Distance





Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Tangent Propagation (Tangent Prop)

1.Definition:

1. **Tangent Propagation** is a training method used in neural networks to make the model **invariant to small transformations** like rotation, translation, or scaling.

Why Do We Need It?

- In image recognition:
- A digit “3” slightly rotated is still “3”
- A shifted face is still the same person
- But neural networks may change predictions for small input changes ❌
- Tangent Propagation helps the model **ignore small variations**.

How It Works

During training:

- Define allowable small transformations (rotation, shift, etc.)
- Compute **tangent vectors** representing these transformations
- Add a penalty term in the loss function
- This penalty forces the model to be insensitive to those transformation

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Mathematical Intuition

Normal training minimizes:

$$Loss = Classification\ Error$$

Tangent Prop adds:

$$Loss = Classification\ Error + \lambda \times Invariance\ Penalty$$

The penalty reduces sensitivity to transformation directions.

Tangent Distance	Tangent Propagation
Used during comparison	Used during training
Adjusts images before measuring distance	Adjusts learning to enforce invariance
Post-processing idea	Training regularization method

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Manifold Tangent Classifier (MTC)

1.Definition:

- Manifold Tangent Classifier (MTC) is a way of training a neural network so that it **does not change its decision when the input changes slightly in natural ways** (like small rotations, shifts, or thickness changes).

I**magine**:

- A sheet of paper lying in a 3D room.
- The room is 3D space.
  The paper is 2D surface inside that 3D space.
- The paper is called a **manifold**.
- It is lower-dimensional but exists inside higher-dimensional space.

Train Classifier with Regularization

Loss function becomes:

$$Loss = Classification\ Loss + \lambda \times Manifold\ Penalty$$

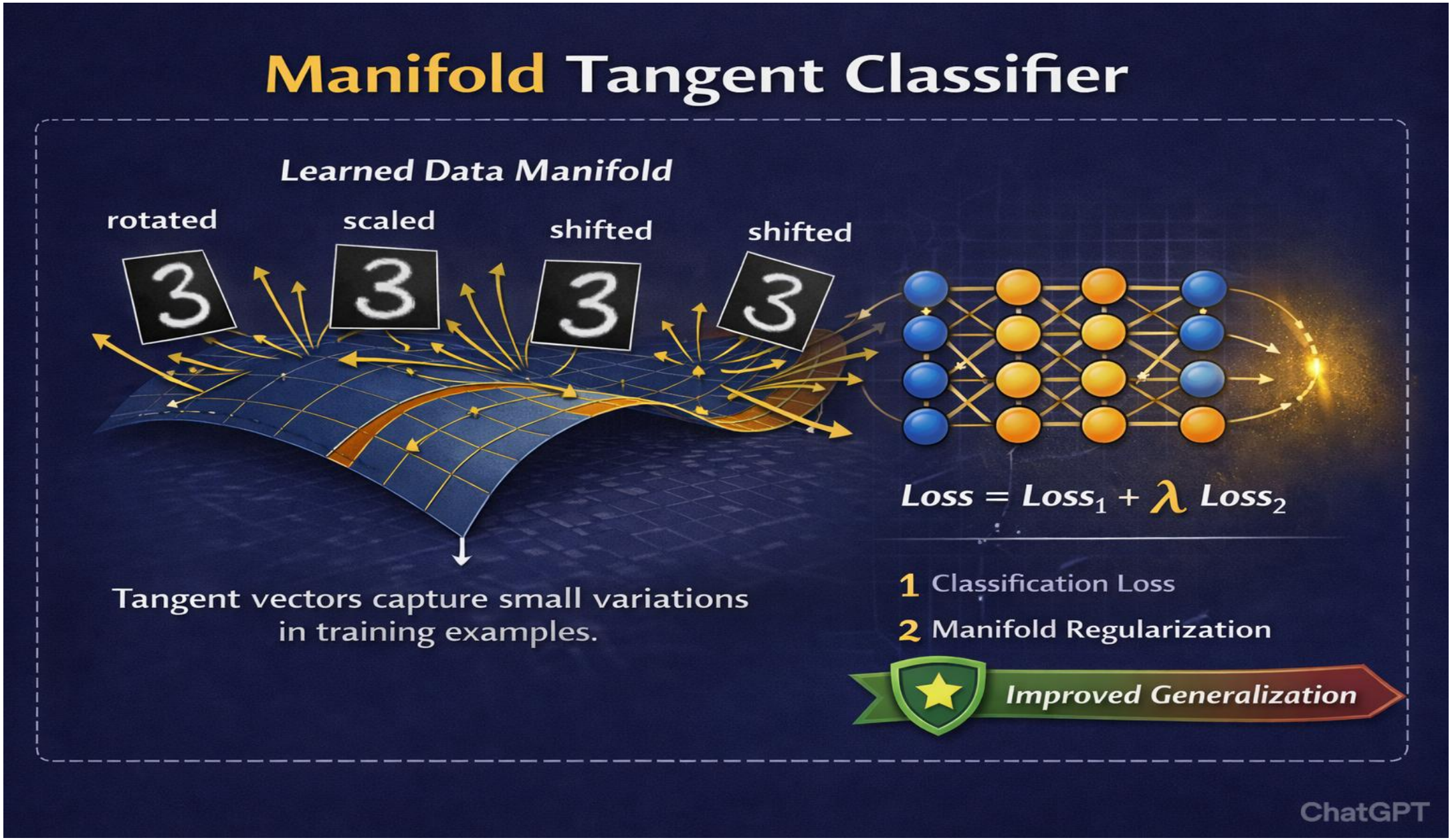
Where:

First term → normal cross-entropy loss

Second term → penalizes output change along tangent directions

λ → controls strength

1. **Goal**:
    1. Use the **manifold structure of data** to improve robustness and classification accuracy.
2. **Use Cases**:
    1. Image recognition (handwritten digits, faces)
    2. Speech recognition
    3. Any high-dimensional data with meaningful local transformations
3. **Benefit**:
    1. Captures **invariances naturally present in data**
    2. Reduces overfitting and improves **generalization**


Manifold Tangent Classifier
Learned Data Manifold
rotated
scaled
shifted
shifted
3
3
3
3
Tangent vectors capture small variations in training examples.
Loss = Loss1 + λ Loss2
1 Classification Loss
2 Manifold Regularization
Improved Generalization
ChatGPT

Optimization for Training Deep Models: Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta-Algorithms.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Optimization for Training Deep Models (Deep Learning Notes)

1. Pure Optimization (Deep Learning)

In deep learning, **training a neural network** means finding the best values of the model parameters (weights and biases) that minimize the error.
Mathematically, training is written as:

$$\theta^* = \arg\min_{\theta} \quad J(\theta)$$

1. Parameters (θ)
θrepresents all the learnable values in the network, such as:
Weights (W)
Biases (b)
So,

$$\theta = \{W, b\}$$

These parameters control how the network makes predictions.
2. Loss Function ($J(\theta)$)
$J(\theta)$is the **loss (cost) function**, which measures:
How wrong the model's predictions are.
Smaller loss → better model
Larger loss → poor model

3. Optimal Parameters (θ^*)

$\theta^*$is the best set of parameters that gives the minimum possible loss.

2. Challenges in Neural Network Optimization

Deep networks are hard to optimize because the loss surface is complex.

(a) Non-Convex Optimization

Neural network loss is **not convex**, so it has:

- many valleys
- many peaks
- No guarantee of global minimum.

b) Local Minima

A **local minimum** is a point where the loss is low compared to nearby points……but not the lowest overall.

Problem: Training can get stuck there.

Example:

- ✓ Global minimum = deepest valley
- ✓ Local minimum = small valley
- ✓ Gradient descent stops because:
- ✓ slope becomes zero
- ✓ it thinks it reached the best point
- ✓ But actually, a better solution exists elsewhere.

(c) Saddle Points

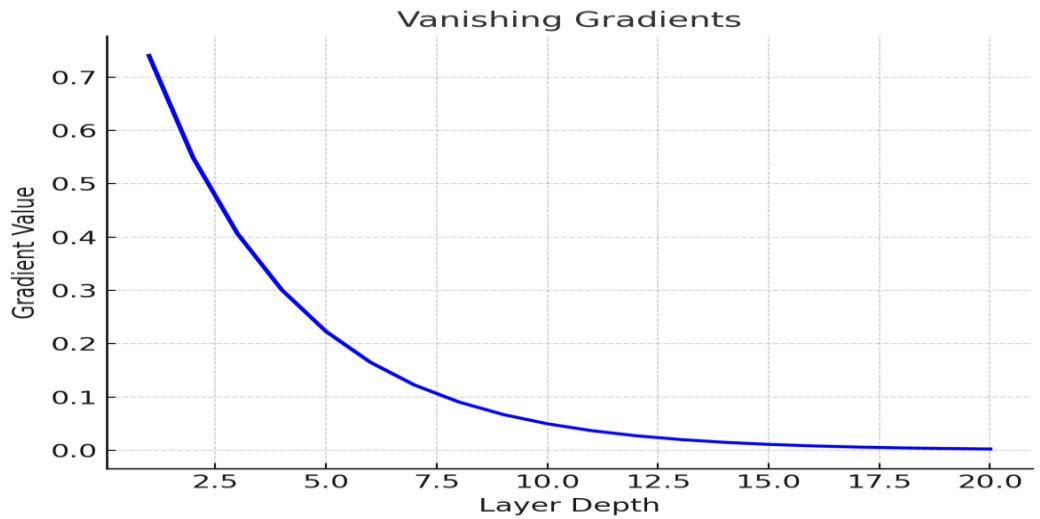
What is a saddle point?

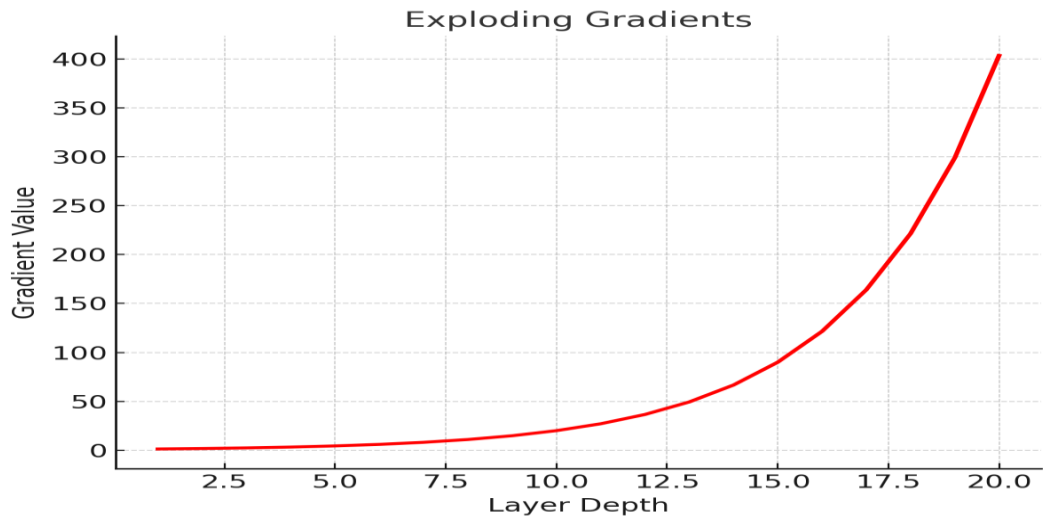
- ✓ A saddle point is a point where:gradient (slope) = 0
- ✓ but it is **not a minimum.**It is like a horse saddle shape:
- ✓ down in one direction.up in another direction

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

(d) Vanishing & Exploding Gradients

Vanishing gradient
Gradients become very small → slow learning
Exploding gradient
Gradients become very large → unstable training





Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Parameter Initialization Strategies

Before training starts, neural network weights must be initialized properly.

Bad initialization can cause:

- very slow learning
- vanishing/exploding gradients
- failure to converge

So, good initialization is essential for faster and stable training.

Zero Initialization

All weights are initialized to zero:

$$W = 0$$

Problem: Symmetry Issue

- If all neurons start with the same weights, then:
- all neurons produce the same output
- all gradients become identical
- neurons learn the same features
- So, the network fails to learn properly.

Result: No effective learning.

4.2 Random Initialization

Weights are initialized with small random values:

$$W \sim \mathcal{N}(0, \sigma^2)$$

Advantage

•breaks symmetry

•different neurons learn different features

Problem

1. If random values are too large:gradients may explode
2. If too small:gradients may vanish
3. So, smarter initialization is needed.

4.3 Xavier Initialization (Glorot Initialization)

Xavier Initialization is a weight initialization method designed to keep the activations and gradients stable across layers during training.It was introduced by: Glorot and Bengio (2010)

why Xavier Initialization is Needed?

- In deep neural networks, poor initialization can cause:
- **Vanishing gradients** (gradients become too small)
- **Exploding gradients** (gradients become too large)
- This slows down or stops learning.
- Xavier initialization solves this by keeping the variance of signals almost constant through layers.

When is Xavier Used?

Xavier initialization is best suited for activation functions like:

- **Sigmoid**
- **Tanh**

Because these functions are sensitive to large input values.

Formula

Weights are initialized such that:

$$Var(W) = \frac{1}{n_{in} + n_{out}}$$

Where:

n_{in} =number of input neurons (fan-in)

n_{out} =number of output neurons (fan-out)

He Initialization (Kaiming Initialization)

He Initialization is a weight initialization strategy designed specifically for deep neural networks that use **ReLU activation functions**.

It was introduced by:He et al. (2015)

Why He Initialization is Needed?

In deep networks with ReLU, poor initialization can lead to:

- **Vanishing gradients**
- **Dead ReLU neurons**
- Slow or unstable training
- ReLU sets all negative inputs to zero, so fewer neurons remain active.
- He initialization compensates for this by using a larger variance.

Formula

Weights are initialized such that:

$$Var(W) = \frac{2}{n_{in}}$$

Where:

n_{in} =number of input connections (fan-in)

Algorithms with Adaptive Learning Rates

- In basic gradient descent, the learning rate $\eta$is fixed:
  - $\theta = \theta - \eta \nabla J(\theta)$
- But choosing a single learning rate is difficult because:
- Too large → training becomes unstable
- Too small → training becomes very slow
- So adaptive learning rate algorithms automatically adjust the learning rate for each parameter during training.

Algorithms with Adaptive Learning Rates

Adagrad (Adaptive Gradient Descent)

- Unlike normal gradient descent, **Adagrad changes the learning rate for each parameter** during training.
- Parameters that change a lot → get a **smaller learning rate**.
- Parameters that change less → get a **larger learning rate**.
- This is helpful because **real-world datasets** often have:
- **Sparse features** (appear rarely, like unusual words in NLP).
- **Dense features** (appear often). <Features that frequently take meaningful values

So, it's not fair to use the same learning rate for all features.

Benefits of using AdaGrad

Easy to use – Simple optimization method, works with many models.
No manual tuning – Automatically adjusts learning rates, no need to set them manually.
Adaptive learning rate –

Big gradients → smaller learning rate (slows updates).
Small gradients → larger learning rate (speeds updates).
Helps avoid overshooting and improves convergence.

Good with noisy data – Reduces the impact of noisy features by lowering their learning rates.
Handles sparse data well – Very effective for **NLP (natural language processing)** and **recommendation systems**, because rare features get higher learning rates and learn faster.

Adam optimizer

- Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning.
- It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.
- The name “Adam” is derived from “adaptive moment estimation,” highlighting its ability to adaptively adjust the learning rate for each network weight individually.
- Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.
- Adam optimizer is an optimization algorithm that extends SGD by dynamically adjusting learning rates based on individual weights. It combines the features of AdaGrad and RMSProp to provide efficient and adaptive updates to the network weights during deep learning training.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Adam Optimizer Formula

The **Adam optimizer** updates the weights of a neural network using both the **first moment (mean of gradients)** and the **second moment (variance of gradients)**. The general update formula is:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where:

- θ_t →current weight
- η→ learning rate
- $\hat{m}_t$ →bias-corrected first moment (average of past gradients)
- $\hat{v}_t$ →bias-corrected second moment (average of squared gradients)
- ϵ→ small number to avoid division by zero
- $\beta_1$and β_2 →decay rates for the moving averages of the gradients

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Adam Optimizer Formula

The **Adam optimizer** updates the weights of a neural network using both the **first moment (mean of gradients)** and the **second moment (variance of gradients)**. The general update formula is:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where:

- θ_t →current weight
- η→ learning rate
- $\hat{m}_t$ →bias-corrected first moment (average of past gradients)
- $\hat{v}_t$ →bias-corrected second moment (average of squared gradients)
- ϵ→ small number to avoid division by zero
- $\beta_1$and β_2 →decay rates for the moving averages of the gradients

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Adversarial Training

1.Definition:

1. A technique to make neural networks **robust against adversarial examples**, which are inputs **slightly modified to fool the model**.
2. Goal: improve the model's **generalization and reliability** under malicious or small perturbations.

2.Adversarial Examples:

1. Inputs x'that are **almost identical** to the original input $x$but lead the model to **wrong predictions**.
2. Generated using techniques like **FGSM (Fast Gradient Sign Method)** or **PGD (Projected Gradient Descent)**.

3.Mechanism of Adversarial Training:

1. Generate adversarial examples x'during training.
2. Include both **original** and **adversarial examples** in the training set.
3. Optimize the **loss function** so the model performs well on both.

4.Benefits:

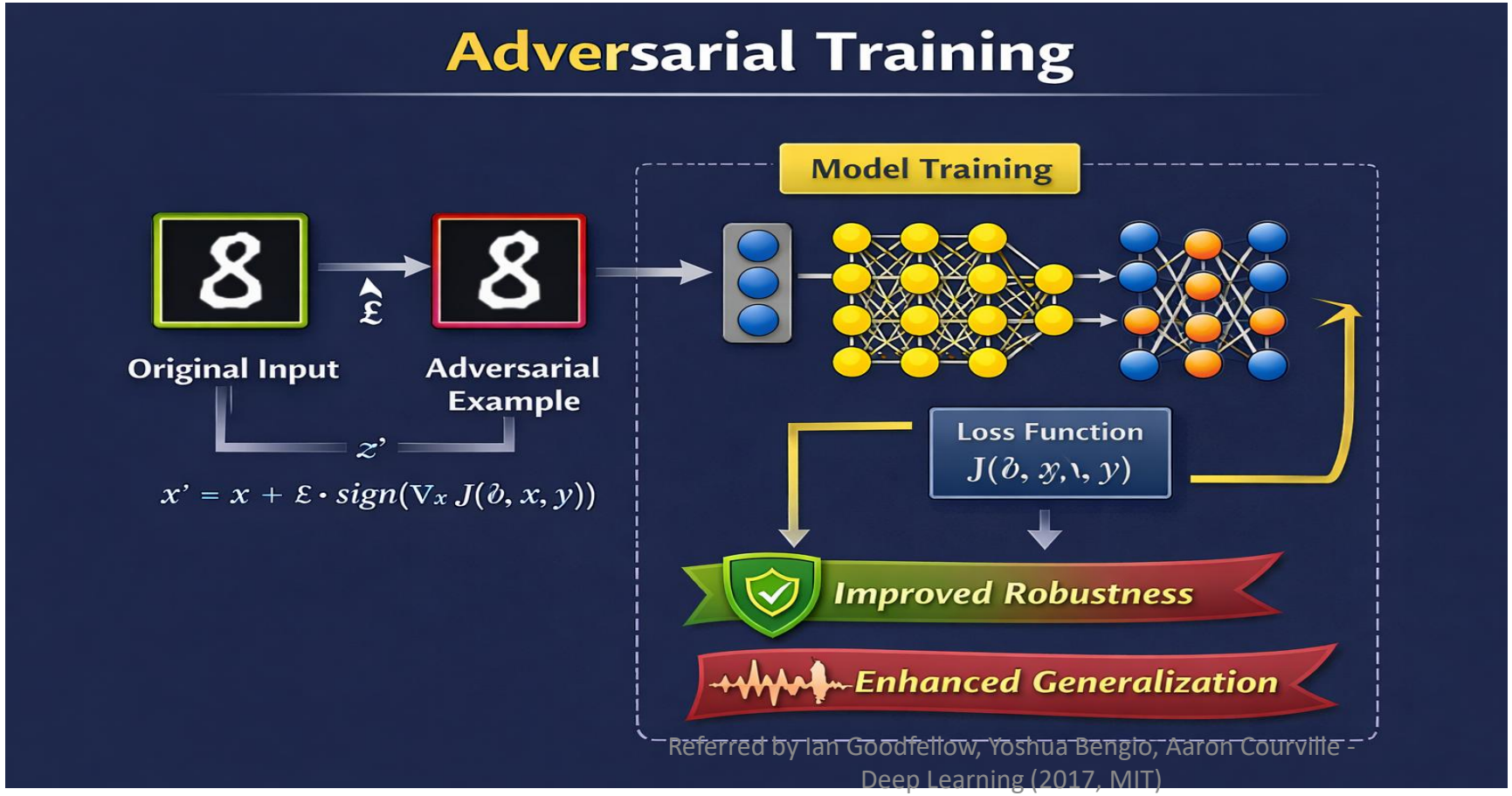
1. Improves **robustness** against attacks or perturbations.
2. Helps the model **generalize better** on slightly altered or noisy inputs.
3. Reduces **vulnerability** to adversarial attacks.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

5.Key Formula (for FGSM):

$$x' = x + \epsilon \cdot sign\left(\nabla_x J(\theta, x, y)\right)$$

1. x =original input
2. y= true label
3. ϵ= small perturbation
4. $J(\theta, x, y)$ =loss function



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

UNIT-IV: CONVOLUTIONAL NETWORKS

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

Introduction

- **CNN (Convolutional Neural Network)** is a class or type of deep neural network which is usually applied for image classification.
- It takes an input image, applies filters, flatten the image, and classify the image.
- Convolutional networks (LeCun, 1989), also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology.
- The first CNN was created by **Yann LeCun;** The architecture is particularly useful in image-processing applications. at the time, the architecture focused on handwritten character recognition.
- A digital image is a **binary representation** of visual data. It contains a series of **pixels** arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

Convolutional Operation in CNNs

1. What is Convolution?

•**Convolution** is a mathematical operation between:

- **Input** (I) → e.g., image (2D array of pixels).
- **Kernel/Filter** (K) → small matrix that detects features (edges, textures, patterns).

•The result is a new 2D array called a **Feature Map** (activation map).

In CNNs, convolution extracts features like edges, corners, and shapes.

2. Mathematical Definition

For 2D convolution:

where:

•$I(m, n)$ =input image pixel

•K= kernel (weights)

•$S(i, j)$ =output feature map value

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

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

4. Cross-Correlation in CNNs

•In practice, most CNNs use **Cross-Correlation**, not pure convolution.

•Formula:

$$S(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$$

Difference:

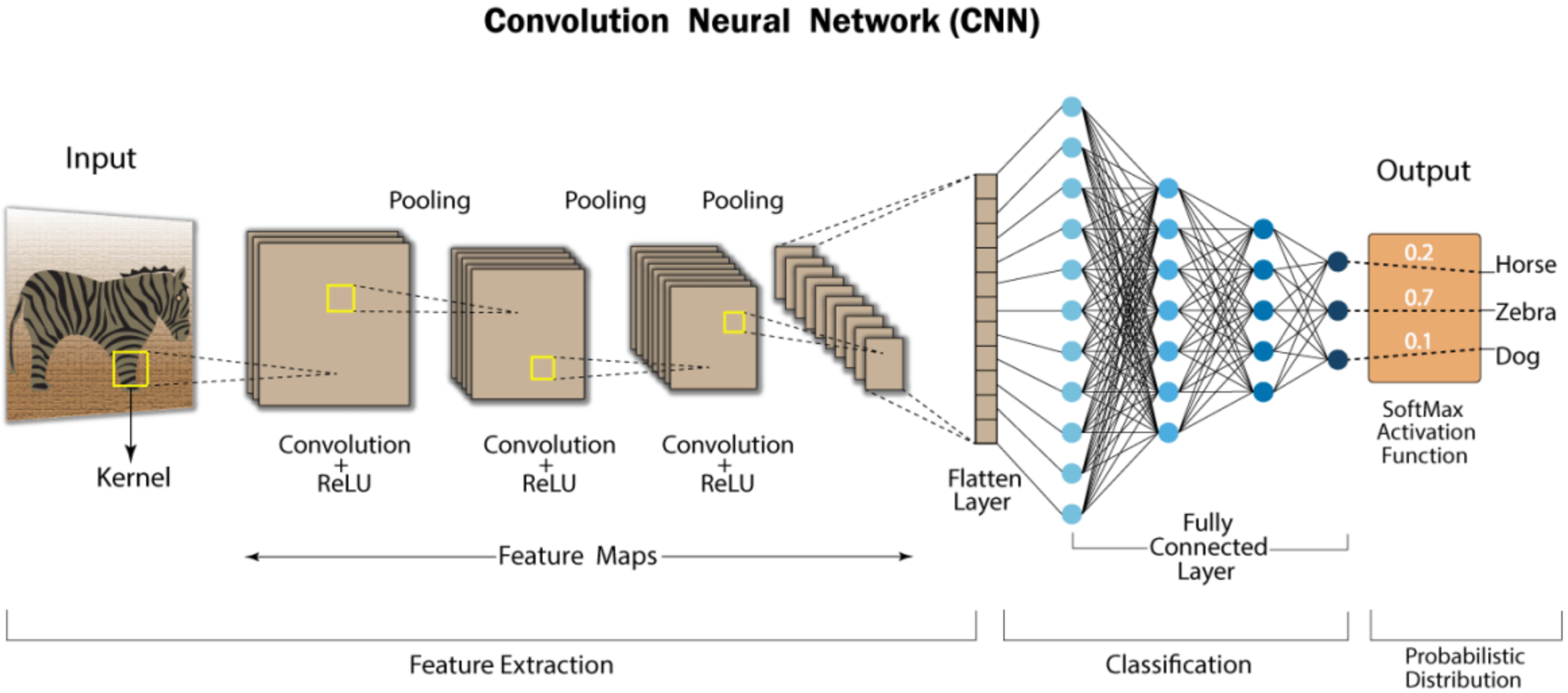
•In **true convolution**, the kernel is flipped before sliding.

•In **cross-correlation**, the kernel is applied directly (no flip).

•For learning in CNNs → it doesn't matter, because kernels are learned automatically.

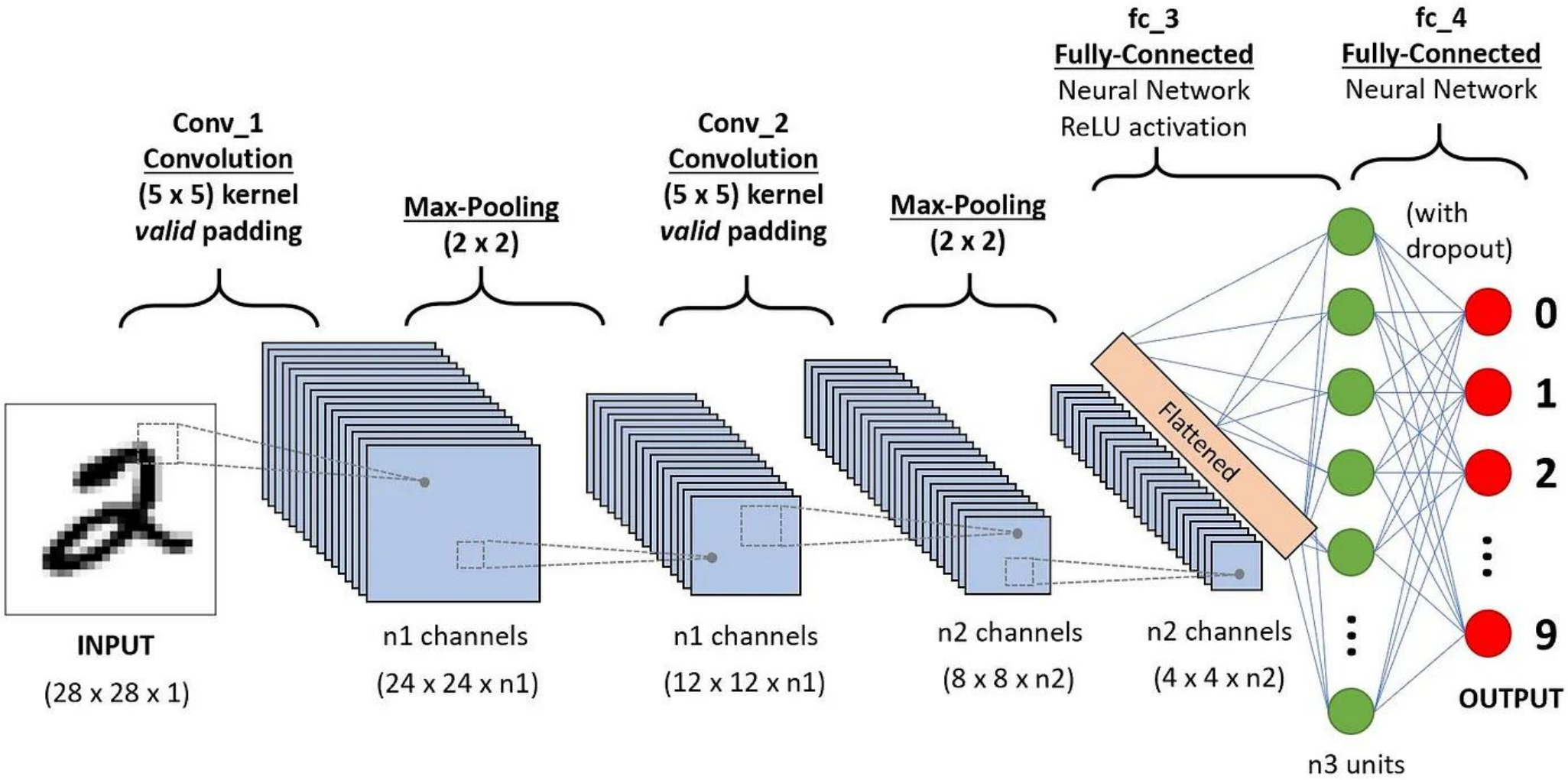
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

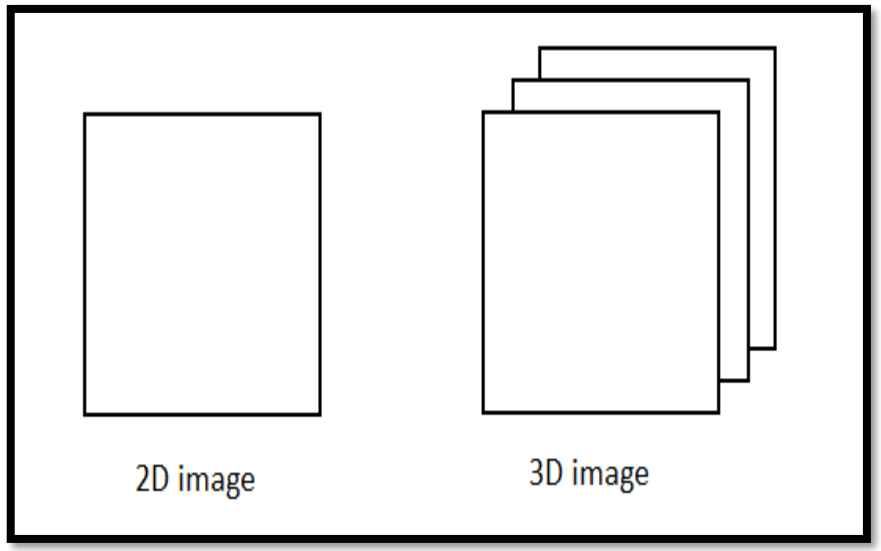
1) Input Image
2) Convolution Layer
3) Pooling Layer
4) Fully Connected Input Layer(Flatten)
5) Fully Connected Layer

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

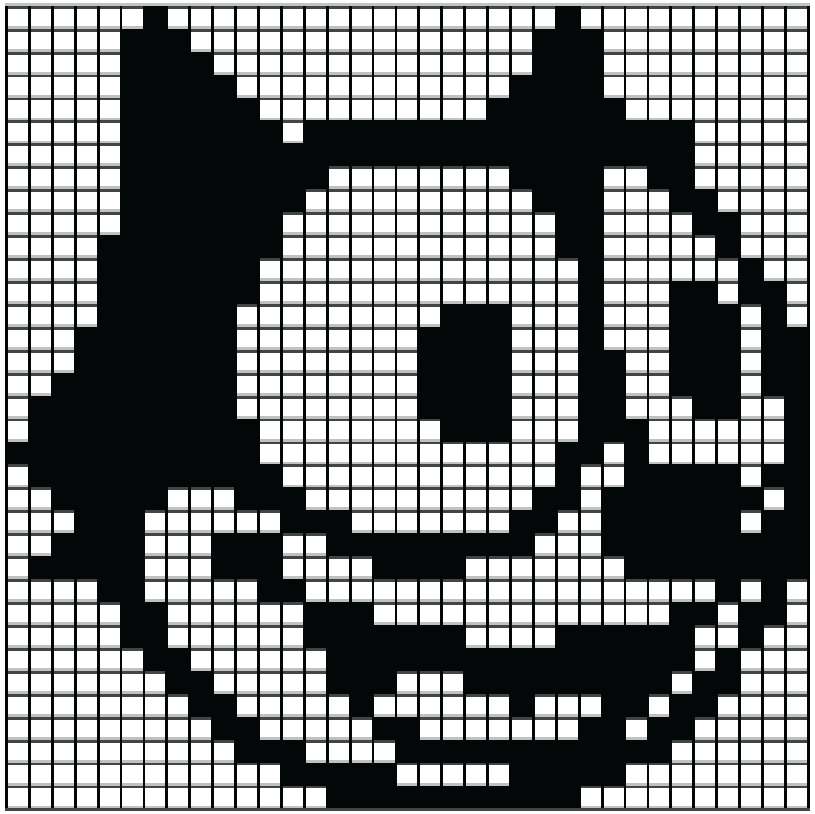
1) INPUT IMAGE

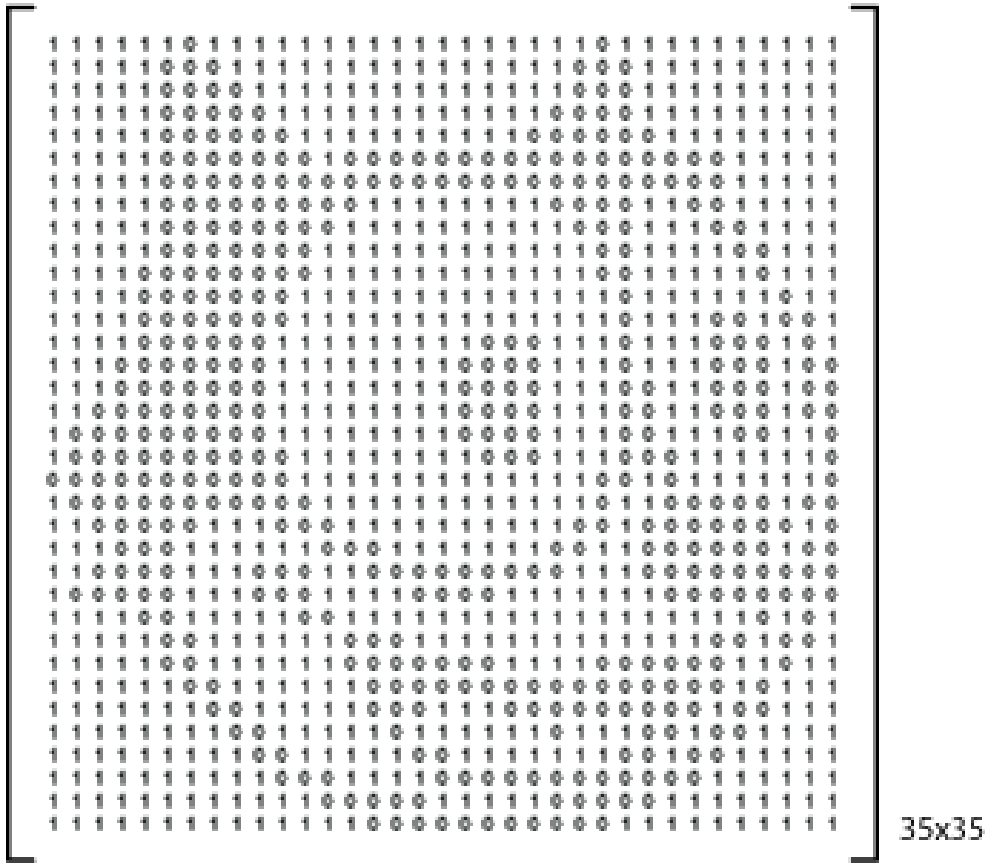
- The input image will be broken down into pixels.
- If it is a **black and white image**, it will only have **one layer** and pixels will be interpreted as **2D array with the value from 0 to 255.**
- If it is **colored image**, it will have **3 layers (red, green, blue) and will be interpreted as 3D array.**



Building Blocks of CNN

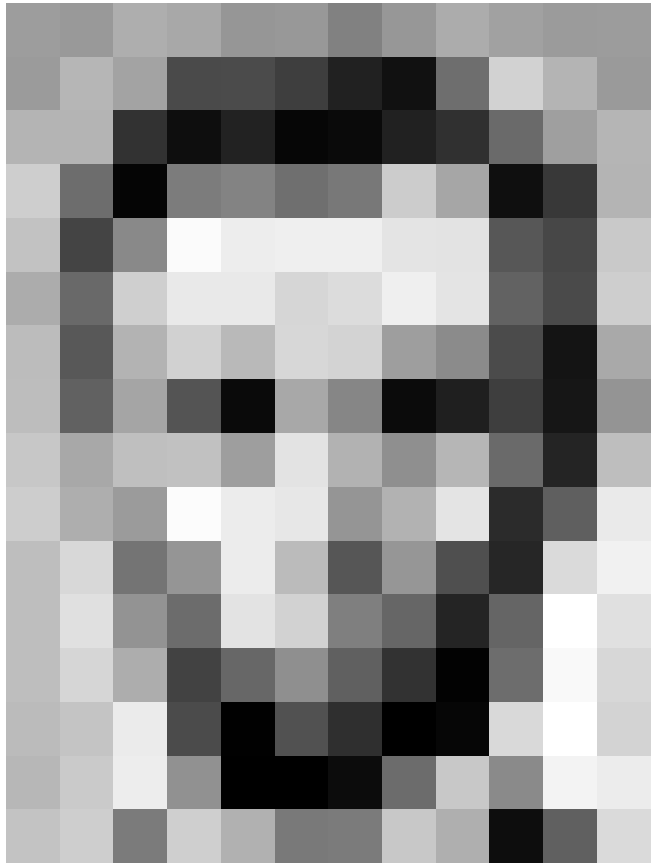
Example of 2D Black and White Image





Building Blocks of CNN

Example of 2D Gray Scale Image



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

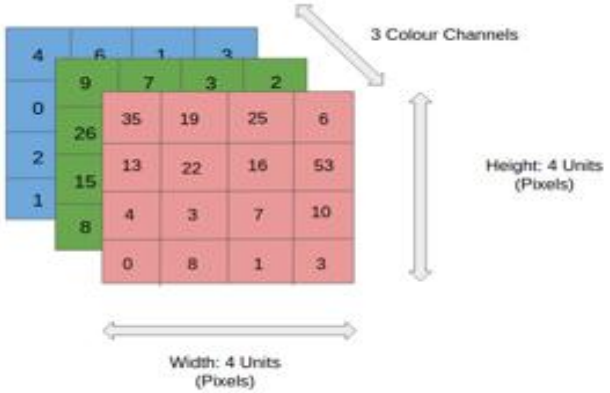
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

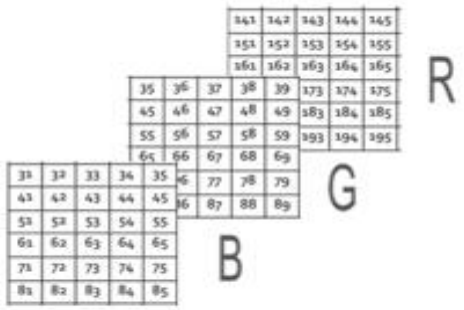
Building Blocks of CNN

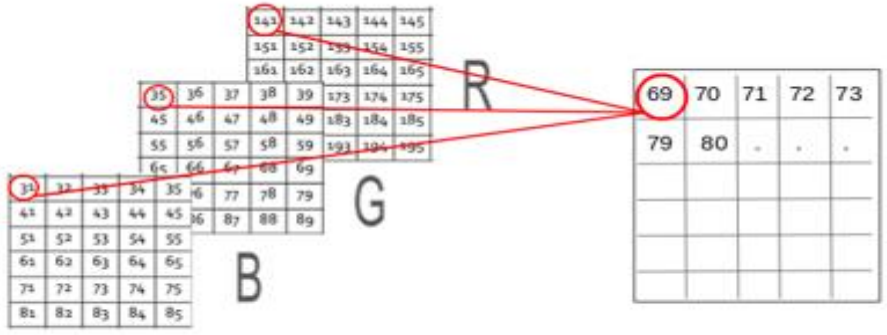
Example of 3D Color Image

In Convolutional Neural Networks (CNNs) images are fed as Three Dimensional Arrays (Tensors) of pixel values corresponding to Red, Green, and Blue channels.



Colour Image



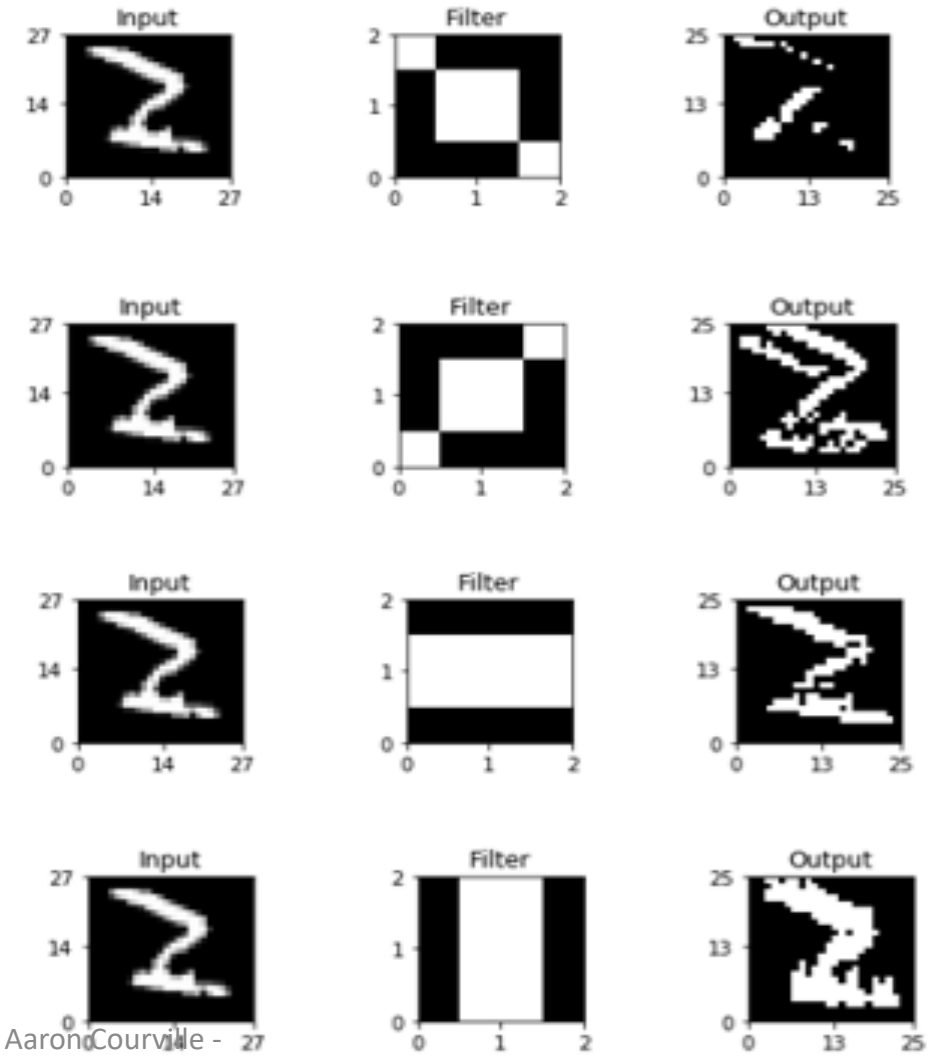


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

2) Convolution Layer

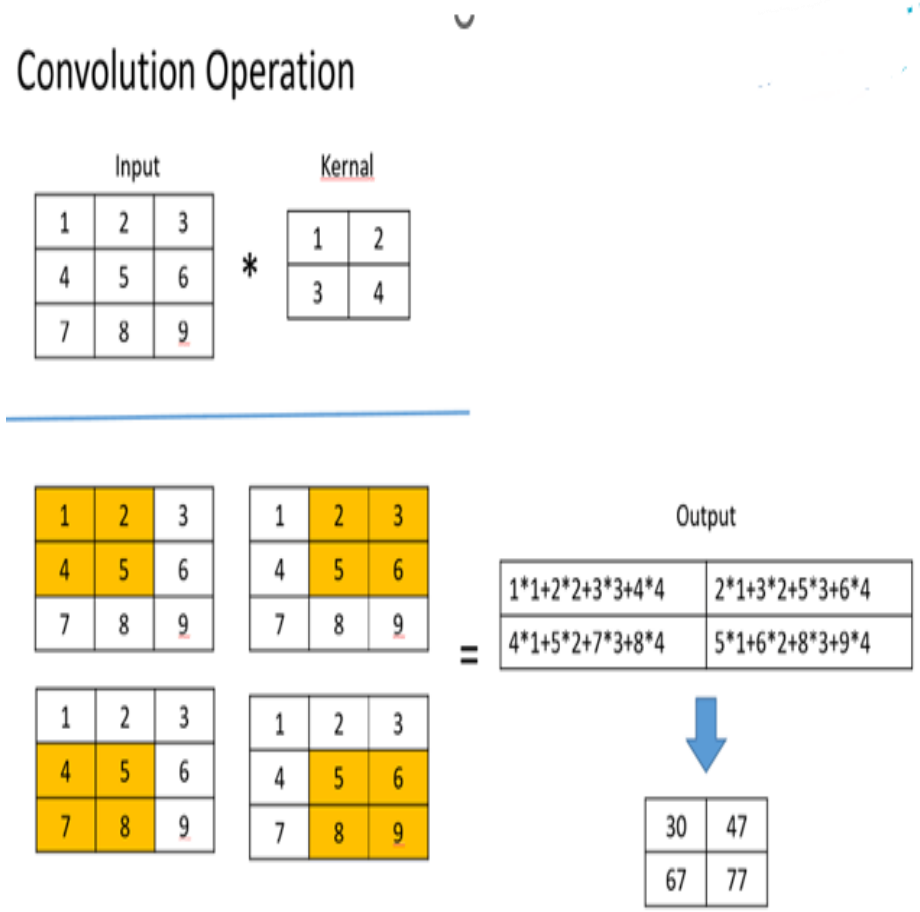
- In the convolution layer, several filters of equal size are applied, and each filter is used to recognize a specific pattern from the image, such as the curving of the digits, the edges, the whole shape of the digits, and more.
- For example, one filter might be good at finding straight lines, another might find curves, and so on. By using several different filters, the CNN can get a good idea of all the different patterns that make up the image.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

- **Using these two matrices, we can perform the convolution operation by applying the dot product, and work as follows:**
  1) **Apply the kernel matrix from the top-left corner to the right.**
  2) **Perform element-wise multiplication.**
  3) **Sum the values of the products.**
  4) **The resulting value corresponds to the first value (top-left corner) in the convoluted matrix.**
  5) **Move the kernel down with respect to the size of the sliding window.**
  6) **Repeat steps 1 to 5 until the image matrix is fully covered.**
- The dimension of the convoluted matrix depends on the size of the sliding window. The higher the sliding window, the smaller the dimension.



What is a Kernel in CNN?

- A **kernel** (also called **filter** or **convolutional matrix**) is a **small-sized matrix of numbers (weights)** used to detect features in the input data (like images).
- It is much smaller than the input image (for example, 3×3 or 5×5 instead of 28×28).
- The kernel "slides" across the image and performs a **convolution operation** to create a **feature map**.

How the Kernel Works (Step by Step)

1. **Take a small region of the image** (same size as the kernel).
2. **Multiply each pixel value** with the corresponding kernel value.
3. **Add them up** → result is a single number.
4. Place that number in the output (feature map).
5. **Slide the kernel** across the whole image (left to right, top to bottom).

This process = **convolution**.

Building Blocks of CNN

Input Image

252	251	246	207	90
250	242	236	144	41
252	244	228	102	43
250	243	214	59	52
248	243	201	44	54

Mark C. F. Sousa

Feature map

Kernel

1	0	-1
1	0	-1
1	0	-1

Receptive field

X

=

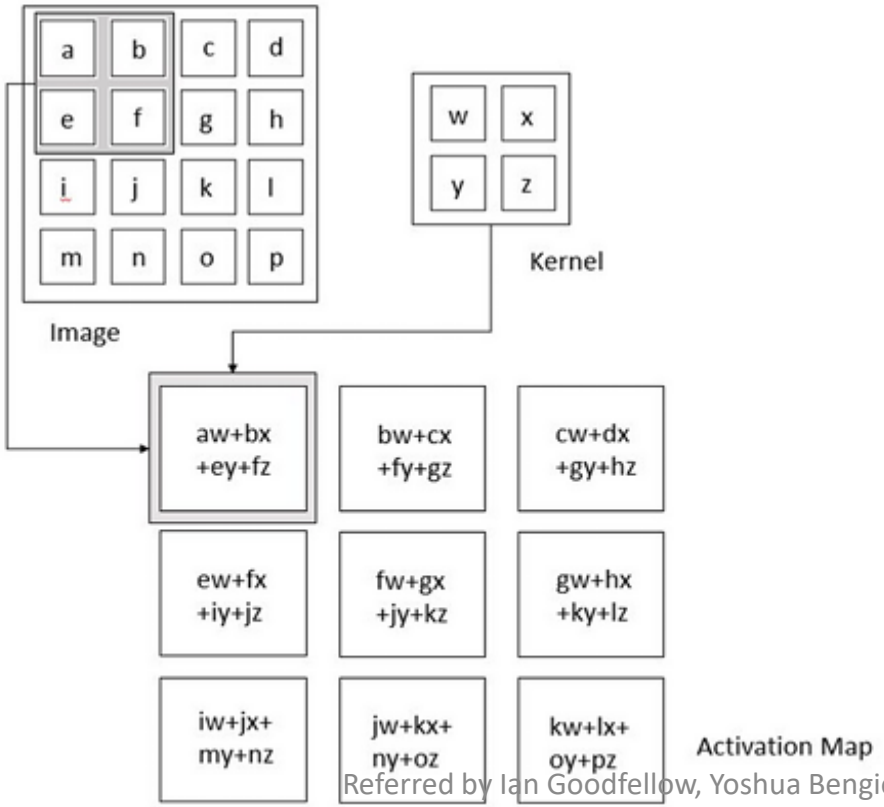
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Fig: Formula for Convolution Layer

This will yield an output volume of size ***Wout* x *Wout* x *Dout*.**



Note:padding is the process of adding extra pixels (usually zeros) around the border of an input image or feature map before applying convolution.

Building Blocks of CNN

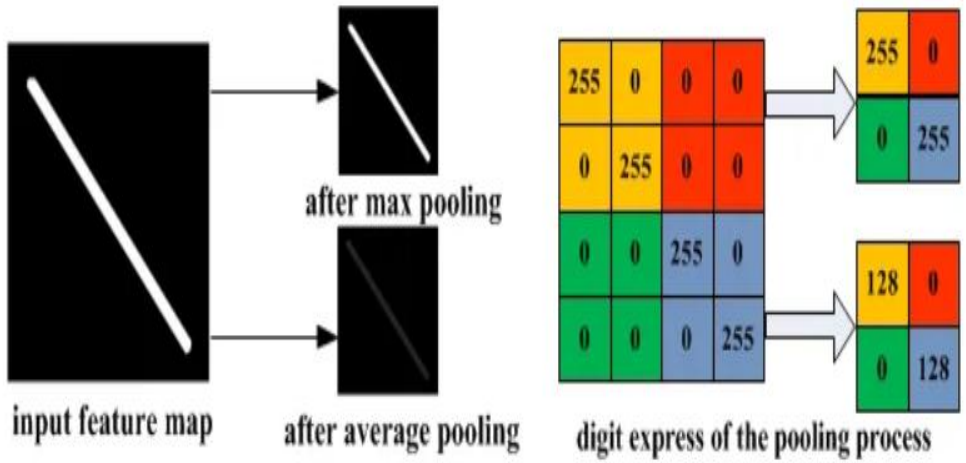
3) Pooling Layer

The goal of the pooling layer is to pull the most significant features from the convoluted matrix.

This is done by applying some aggregation operations, which reduce the dimension of the feature map (convoluted matrix), hence reducing the memory used while training the network.

The most common aggregation functions that can be applied are:

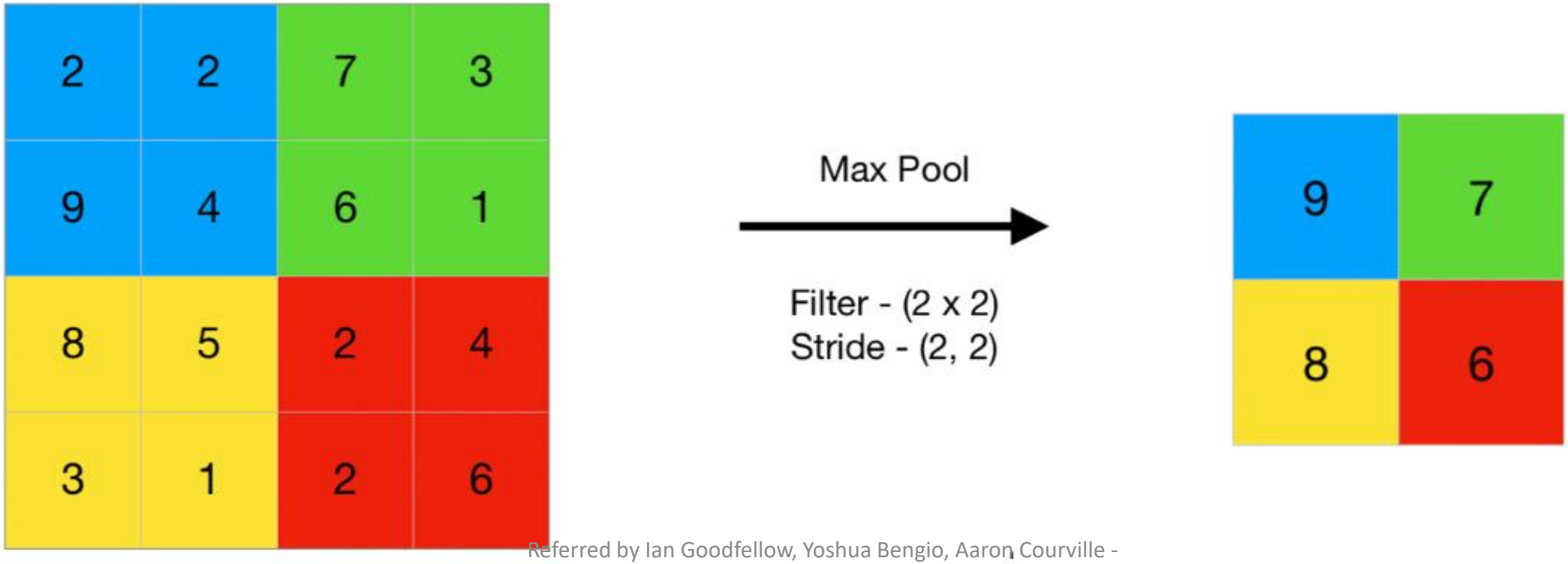
1) Max pooling, which is the maximum value of the feature map
2) Average pooling is the average of all the values.



1. Max Pooling

Max pooling selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

Max pooling layer preserves the most important features (edges, textures, etc.) and provides better performance in most cases.

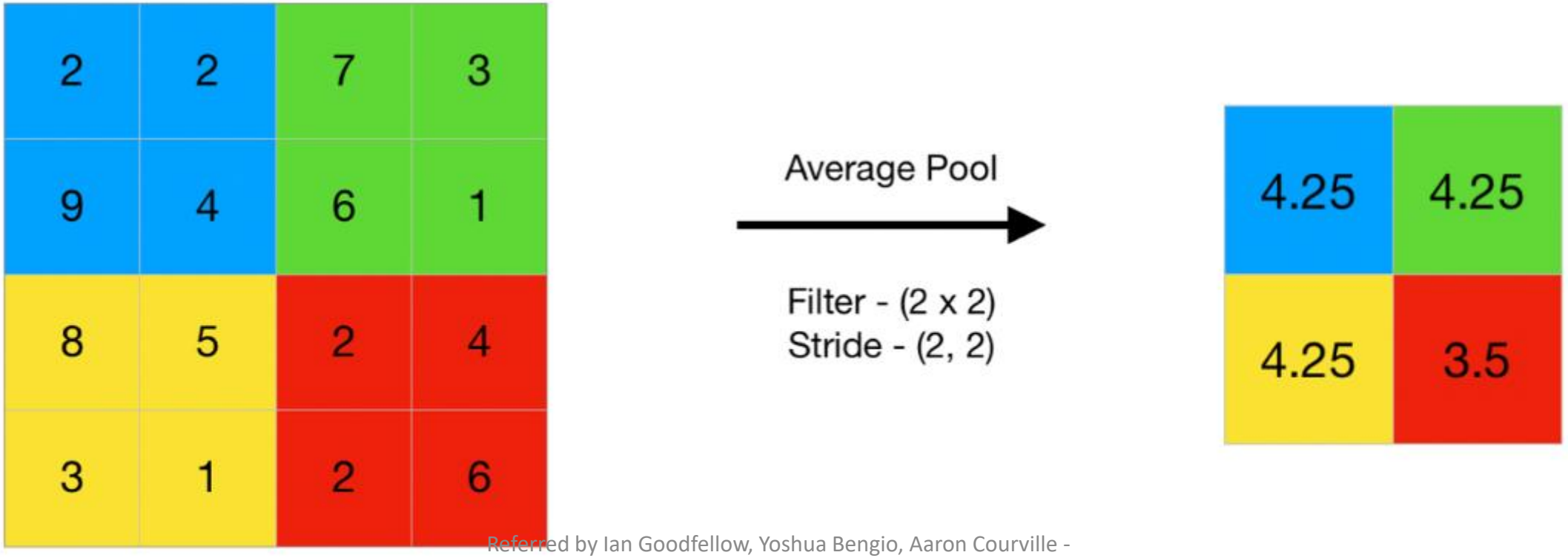


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

2. Average Pooling

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.

Average pooling provides a more generalized representation of the input. It is useful in the cases where preserving the overall context is important.

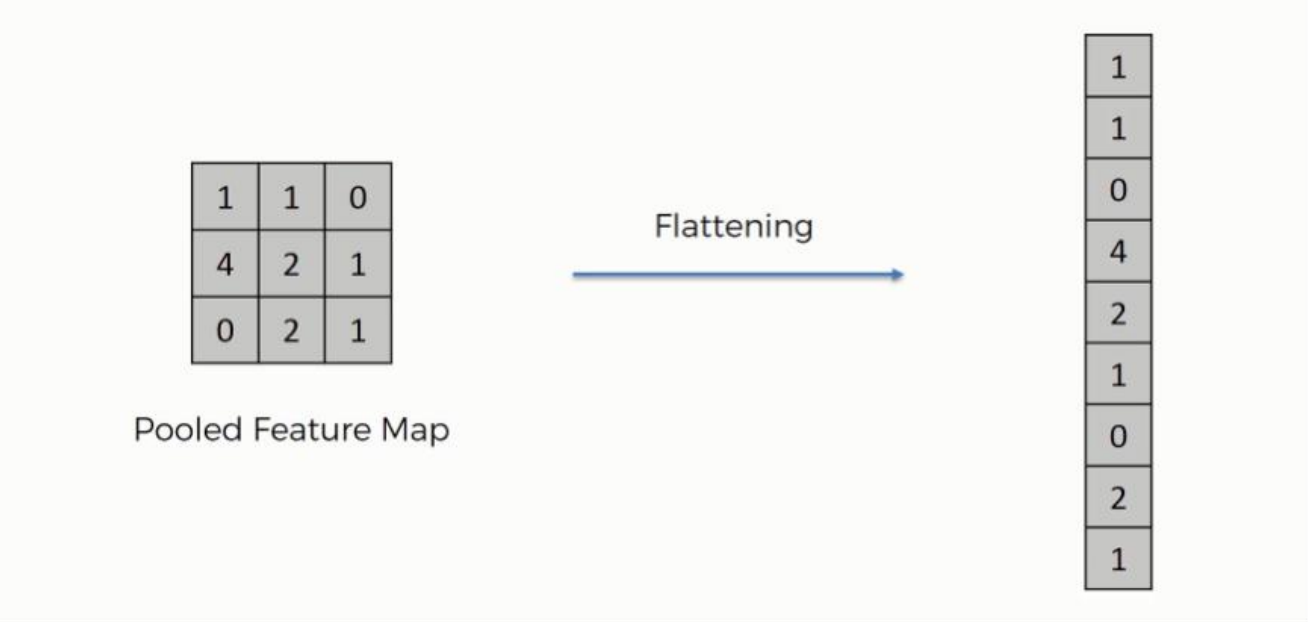


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

3) Fully Connected Input Layer(Flatten)

The last pooling layer flattens its feature map so that it can be processed by the fully connected layer.

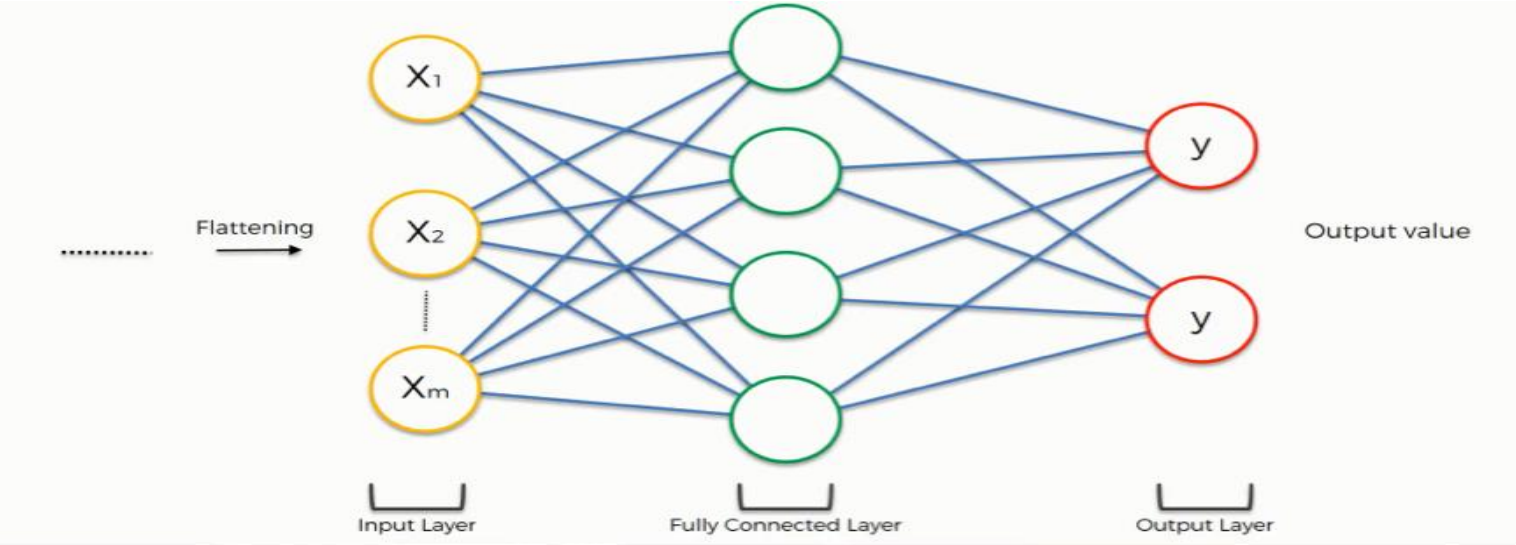


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Building Blocks of CNN

4) Fully Connected Layer

- The flattened matrix goes through a fully connected layer to classify the images.
- The purpose of this layer is to classify the image into a label. It takes the output of previous layer and predicts the best label by applying weights and “voting”.
- The final output will be the probabilities for each label.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Basic Convolution Functions

In Convolutional Neural Networks, convolution functions are different ways of applying filters (kernels) to extract features from input data like images.
These functions help CNNs learn patterns such as:

- edges
- textures
- shapes
- Objects

1. Standard Convolution

A filter slides over the input image and performs element-wise multiplication and sum.

Formula:

$$S(i, j) = \sum X\,(i, +mj + n) \cdot W(m, n)$$

Where:

X = input

W = kernel

S = feature map

Example:

3×3 kernel applied on image → produces feature map.

2. 1×1 Convolution (Pointwise Convolution)

Uses a kernel of size **1×1**.

Purpose:

- Reduces number of channels
- Combines features across channels
- Improves efficiency

Example:

Input: 64 channels

After 1×1 conv → Output: 32 channels

Used in:

- Inception Networks
- ResNet Bottlenecks

3. Strided Convolution

Meaning

Instead of moving filter one step at a time, it moves by **stride > 1**.

Example:

Stride = 1 → detailed output

Stride = 2 → output size reduces

Benefit:

✓ Down sampling without pooling

1. Structured Outputs

In many deep learning tasks, the output is not a single value, but has a **structured form** such as:

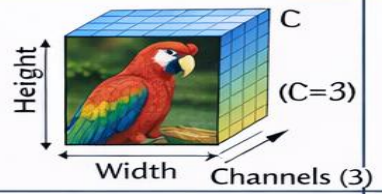
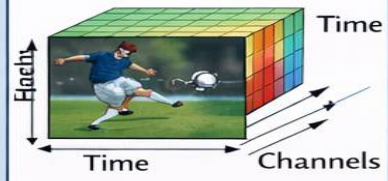
- sequences
- images
- maps
- grids

So CNNs often produce **structured outputs**.

Task	Input	Output Structure
Image Classification	Image	Single label
Object Detection	Image	Bounding boxes + labels
Semantic Segmentation	Image	Pixel-wise label map
Depth Estimation	Image	Depth map

Data Types in Convolutional Networks

CNNs work with different data formats.

Data Type	Shape / Representation	Example Applications	Notes
1D Data		$x \in \mathbb{R}^T$	• Convolutions applied along temporal dimension.
2D Data		$x \in \mathbb{R}^{H \times W}$	• Standard 2D convolution.
3D Data		$x \in \mathbb{R}^{H \times W \times C}$	• RGB images (C=3), Medical scans (CT, MRI)
4D+ Data		$x \in \mathbb{R}^{T \times} \in \mathbb{R}^{Y \times} H \times W \times C$ • Video frames, Spatiotemporal data	• Convolution applied across width, height, and channels. • Convolutions can operate over space + time.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Efficient Convolution Algorithms

Convolution is expensive because it involves many multiplications.
So efficient methods are needed.

(a) Fast Fourier Transform (FFT)

Convolution in spatial domain becomes multiplication in frequency domain:

$$Conv(x, w) = FFT^{-1}\big(FFT(x) \cdot FFT(w)\big)$$

✓Faster for large kernels

b) Im2Col + Matrix Multiplication

Converts convolution into matrix multiplication:
highly optimized on GPUs
Used in deep learning libraries.

(c) Winograd Algorithm

Reduces multiplications for small kernels like:

$$3 \times 3$$

✓ Used in modern CNN accelerators

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

(d) Depthwise Separable Convolution

Used in MobileNet:

•reduces computation drastically

Instead of full convolution:

1.Depthwise convolution

2.Pointwise convolution

✓ Efficient for mobile devices

4. Random or Unsupervised Features

CNN filters are usually learned through backpropagation.

But sometimes features are obtained without supervision.

(a) Random Features

Filters are randomly initialized and not trained.

Surprisingly, random CNN features can still work for simple tasks.

✓ Fast, no training needed

(b) Unsupervised Feature Learning

Features are learned without labeled data.

Methods include:

- Autoencoders
- Sparse coding
- Self-supervised learning

Example:

CNN learns edges, textures automatically.

Why Useful?

- When labeled data is scarce
- For pretraining networks

Basis for Convolutional Networks

CNNs are based on three key principles:

(a) Local Connectivity

Each neuron connects only to a small region of input.

Example:

$$3 \times 3$$

filter sees only local pixels.

✓ captures local patterns like edges

(b) Parameter Sharing

Same filter is applied across the entire image.

So number of parameters is reduced.

Example:

One edge detector works everywhere.

✓ efficient learning

(c) Translation Equivariance

If input shifts, output feature map also shifts.

CNN recognizes patterns regardless of position.

UNIT-V: SEQUENCE MODELING

Sequence Modeling: 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.

Sequence Modelling

- **Sequence Modelling** is a type of Deep learning Instead of treating inputs independently, sequence models learn from **previous data points** to predict the next one.

Why Sequence Modelling is Important?

Sequence modelling is important because **many real-world data sources are sequential and dependent on context**. Ignoring order leads to loss of critical information.

Captures Temporal Dependencies

In sequential data, the **current output depends on previous inputs**.

Example:

- In the sentence: "The movie was not good‟ The word **"not"** changes the meaning of **"good"**.
- A normal model (like logistic regression or simple ANN) treats words independently.A sequence model understands **context**.

Why Sequence Modelling is Important?

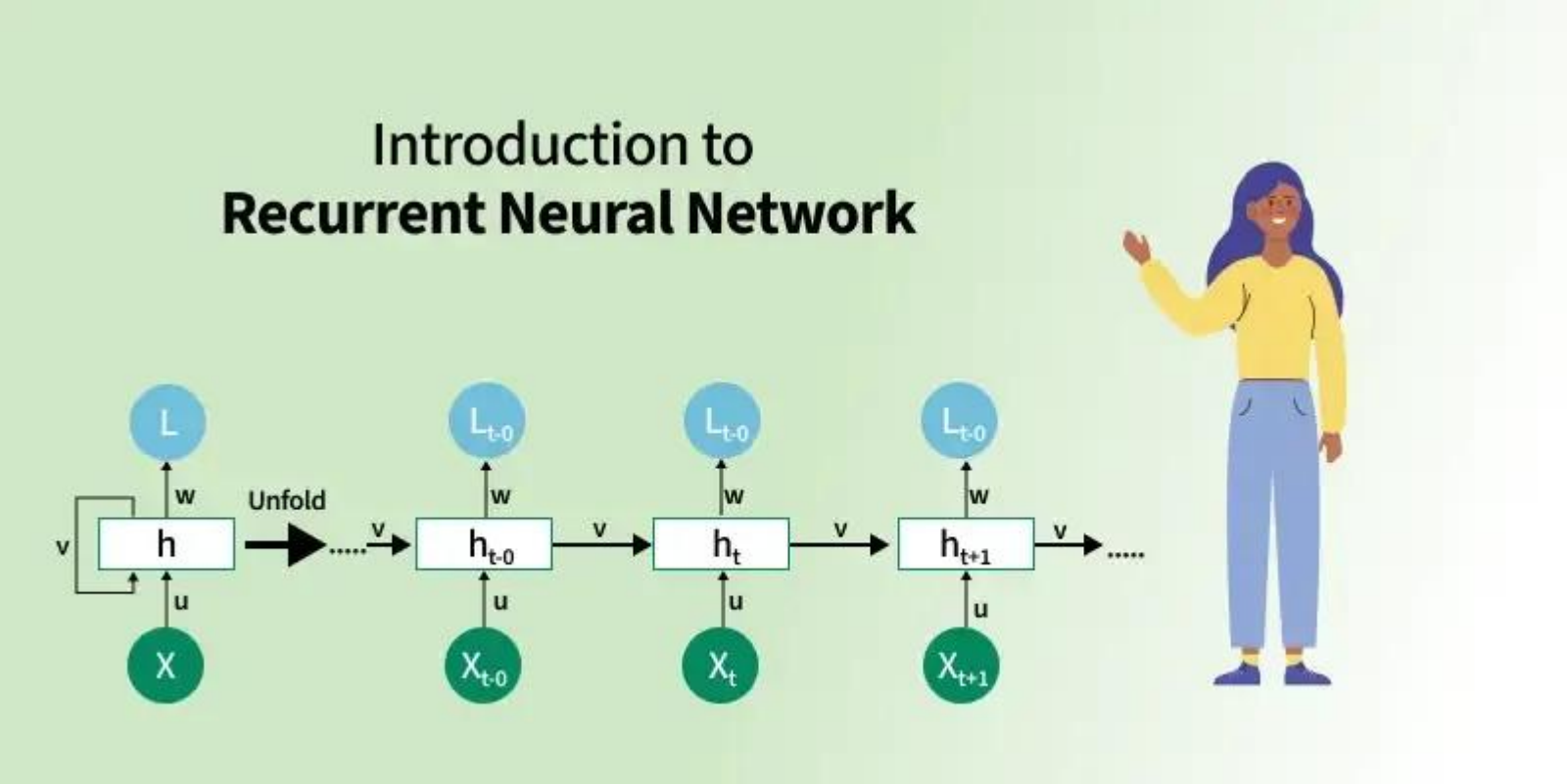
Many real-world problems involve **ordered data**:

- Text (word sequences)
- Speech (audio signals over time)
- Time-series data (stock prices, weather)
- Video (frame sequences)
- DNA sequences

Traditional models (like simple feedforward neural networks) assume inputs are independent, but sequence data has **context dependency**.

Introduction to Recurrent Neural Networks

- **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.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

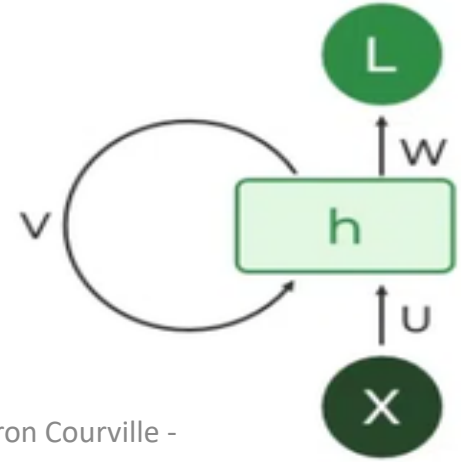
- Imagine reading a sentence and you try to predict the next word, you don't depend only on the **current word** but also **remember the** previous information
- RNNs work similarly by "remembering" past information and passing the **output from one step as input to the next** i.e it **considers all 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.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

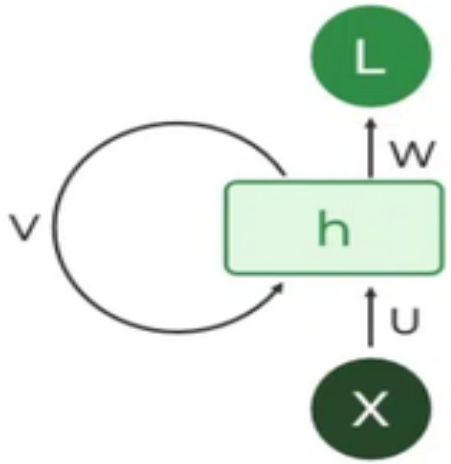
- ✓ **X** → The input at the current time step (e.g., the current word in a sentence).
- ✓ **h** → The **hidden state**. This is the "memory" that carries information from the past. It is updated at every step.
- ✓ **L** → The output (e.g., the predicted next word, classification label, etc.).
- ✓ **u, v, w** → These are the **weight matrices** (learnable parameters of the network):
- ✓ **u**: connects the input (**X**) to the hidden state (**h**).
- ✓ **v**: connects the previous hidden state back into the current hidden state (this is how memory is maintained).
- ✓ **w**: connects the hidden state to the output (**L**).

How it works:

- **Input step**: Current input (**X**) goes into the network, transformed by weight **u**.
- **Memory step**: The previous hidden state (**h**) is combined with this new input, transformed by weight **v**.
- **Hidden update**: A new hidden state (**h**) is produced → this contains both current input info + past memory.
- **Output step**: The hidden state is mapped to an output (**L**) using weight **w**.
- Mathematically:

$$h_t = f(U \cdot x_t + V \cdot h_{t-1})$$
$$y_t = g(W \cdot h_t)$$

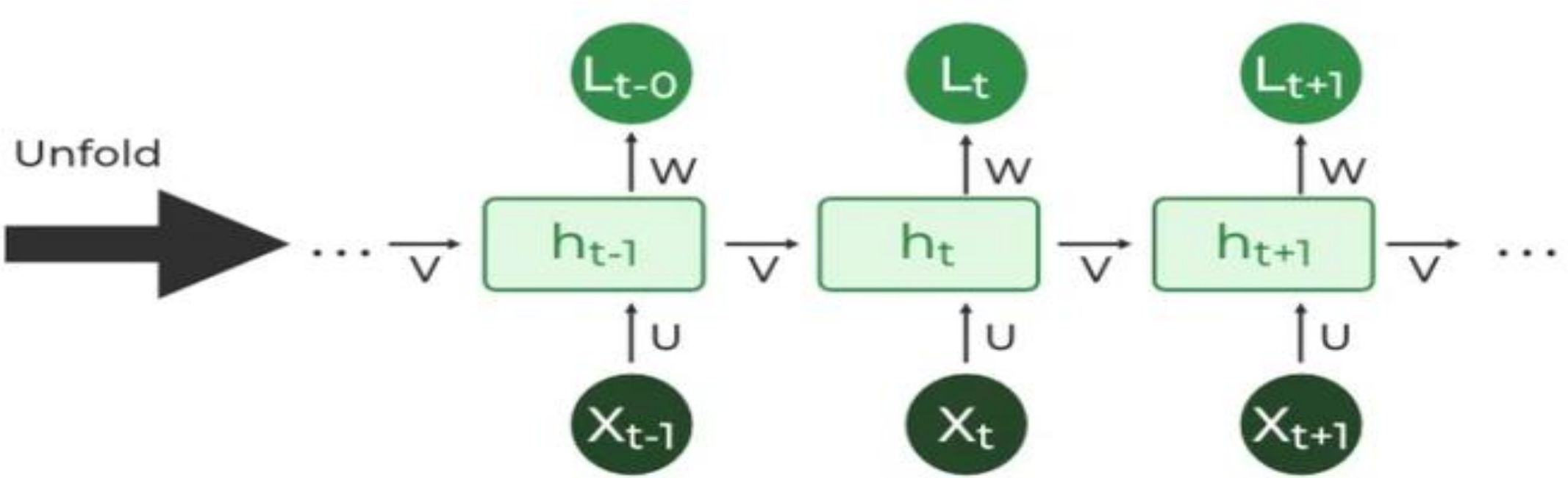
Where:
f= usually **tanh** or **ReLU** (activation for hidden state).
g= usually **softmax** (for output probabilities).



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Explanation of the image:

•Each green rectangle (**h**) is a hidden state at a timestep:

- h_{t-1} :hidden state from the previous timestep.
- h_t :current hidden state.
- h_{t+1} :next hidden state.

•Each dark green circle at the bottom (**X**) is the input at that timestep:

- X_{t-1}, X_t, X_{t+1}.

•Each dark green circle at the top (**L**) is the output at that timestep:

- L_{t-1}, L_t, L_{t+1}.

•**U, V, W** = weight matrices (same as before, shared across timesteps).

Flow:

1.Input $X_t$enters → combined with previous hidden state h_{t-1}using weights $U$and V.

2.A new hidden state $h_t$is generated.

3.That hidden state produces an output $L_t$through weight W.

4.The process repeats for the next input (X_{t+1} (with $h_t$carried forward.

👉 **Why unfolding?**

An RNN cell is actually the same structure repeated at each timestep. Unrolling just makes it easier to see how the sequence flows across time.

Would you like me to also draw this as a **sentence prediction example** (like showing “The cat ___” → predicting the next word)?

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:

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.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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.

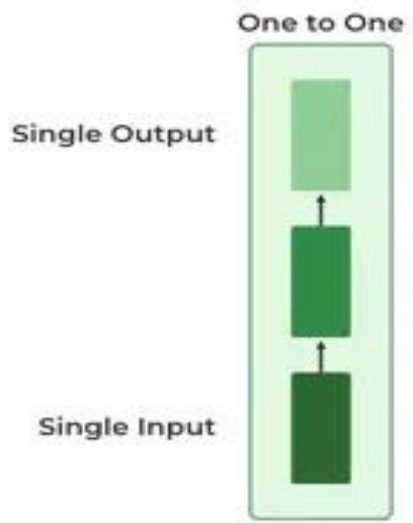
Types Of Recurrent Neural Networks

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.

use case: Basic classification (e.g., image → label).

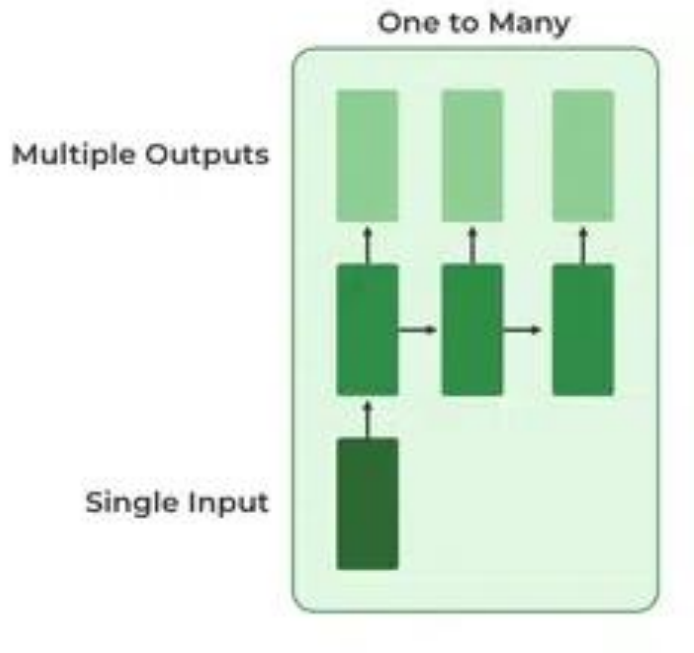


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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.

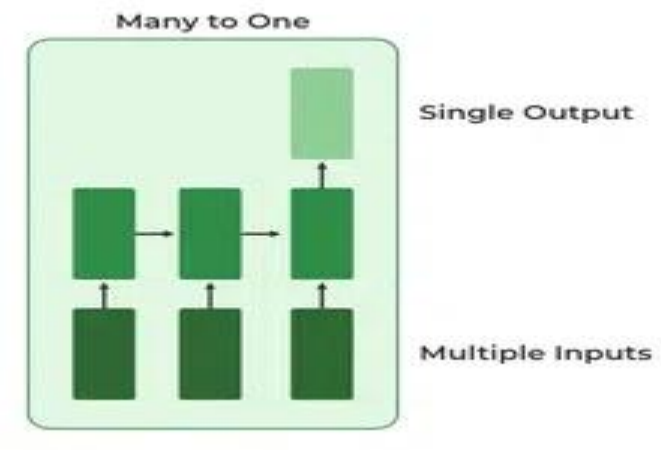


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

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.

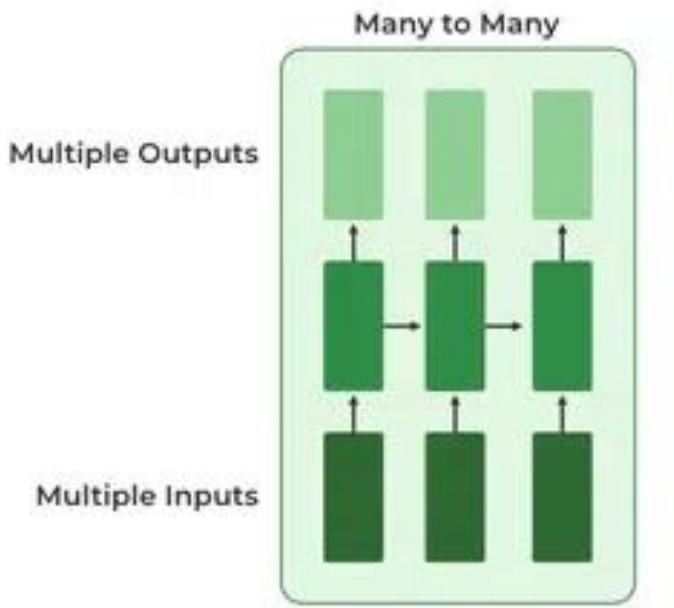


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

4. Many-to-Many RNN

The **Many-to-Many RNN** type processes a sequence of inputs and generates a sequence of outputs.

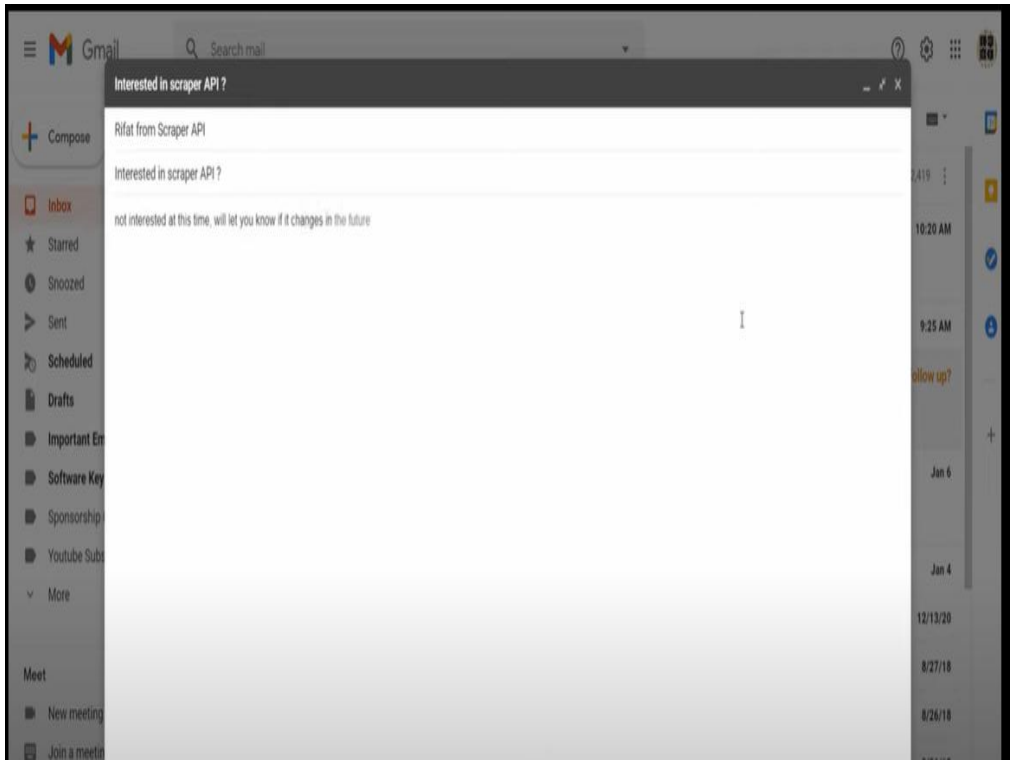
In language translation task a sequence of words in one language is given as input and a corresponding sequence in another language is generated as output.

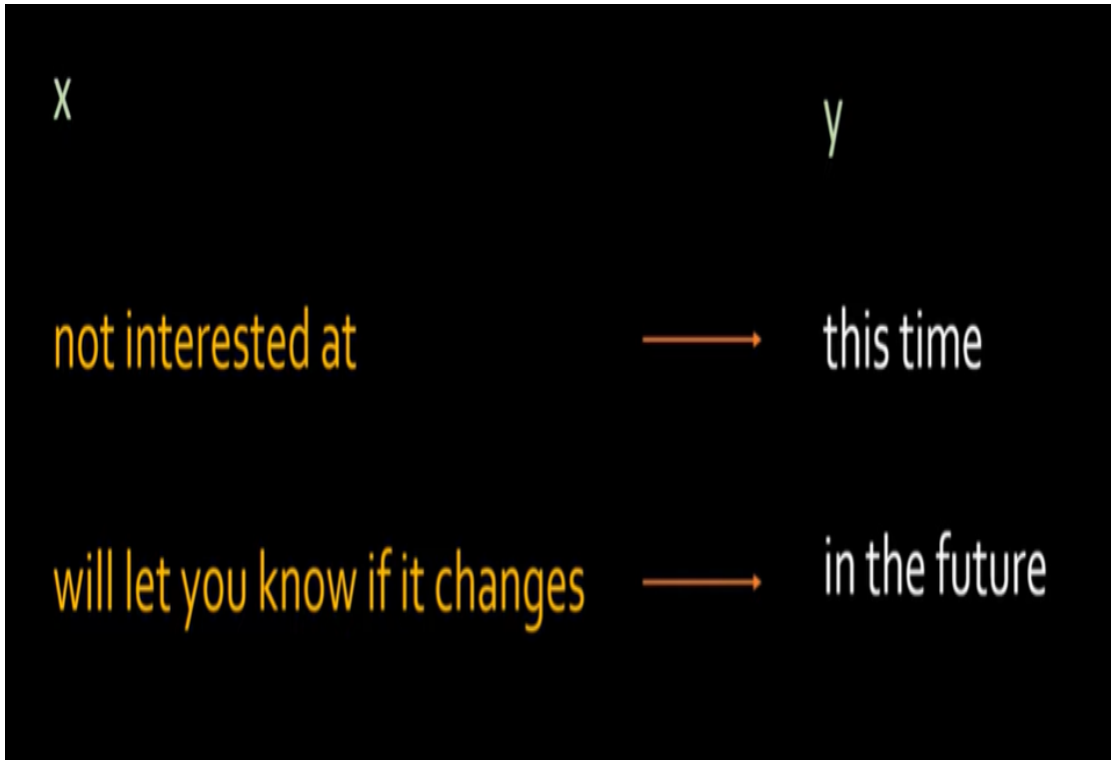


Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

RNN APPLICATIONS

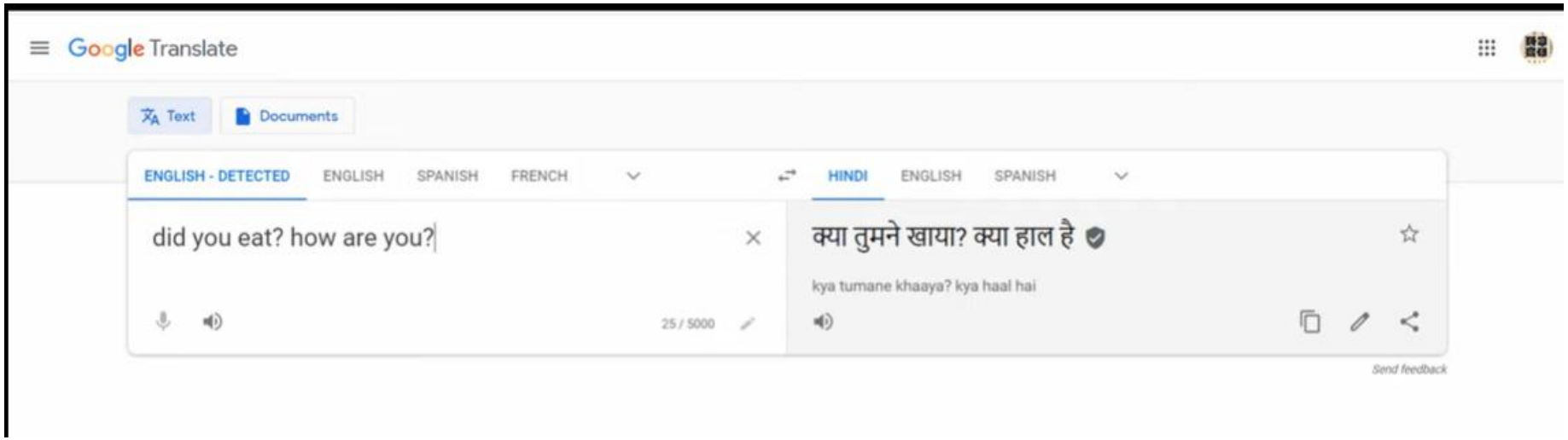
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)





RNN is implemented in Gmail for auto Completion

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



RNN is implemented in Google Transalation

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- In the case of auto-completion, a Recurrent Neural Network (RNN) takes a sequence of characters as input and predicts the next likely character in the sequence.
- Here's an example of how an RNN would handle auto-completion:
- Let's say we have a partially typed word "hel" and we want the RNN to predict the next character.
- The RNN will process the input sequence character by character:

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1. The RNN takes the first character "h" as input. It processes this input and generates an output, which represents the probability distribution of the next character.
2. The RNN then takes the second character "e" as input, along with the previous hidden state. It processes this input and updates its hidden state accordingly.
3. The RNN then takes the third character "l" as input, along with the previous hidden state. It processes this input and updates its hidden state again.
4. After processing the entire input sequence "hel", the RNN will output a probability distribution over the possible next characters (e.g., "l", "p", "m", etc.).
5. The character with the highest probability (let's say "l") is chosen as the predicted next character for auto-completion.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Bidirectional Recurrent Neural Network

- Recurrent Neural Networks (RNNs) are designed to effectively process **sequential data** such as speech, text, and time series.
- Unlike traditional feedforward neural networks that handle inputs as fixed-length vectors, RNNs can work with **variable-length sequences** by maintaining a **hidden state** that carries information from previous time steps.
- This hidden state acts as a form of **memory**, allowing RNNs to capture temporal dependencies and patterns within the sequence.
- However, traditional RNNs encounter difficulties during training, particularly the **vanishing gradient problem**, where gradients become extremely small during backpropagation.
- This leads to poor learning of long-term dependencies.
- To overcome these limitations, advanced architectures such as the **Bidirectional Recurrent Neural Network (BRNN)** have been developed. In the following section, we will explore the structure, working, and advantages of BRNNs in detail.

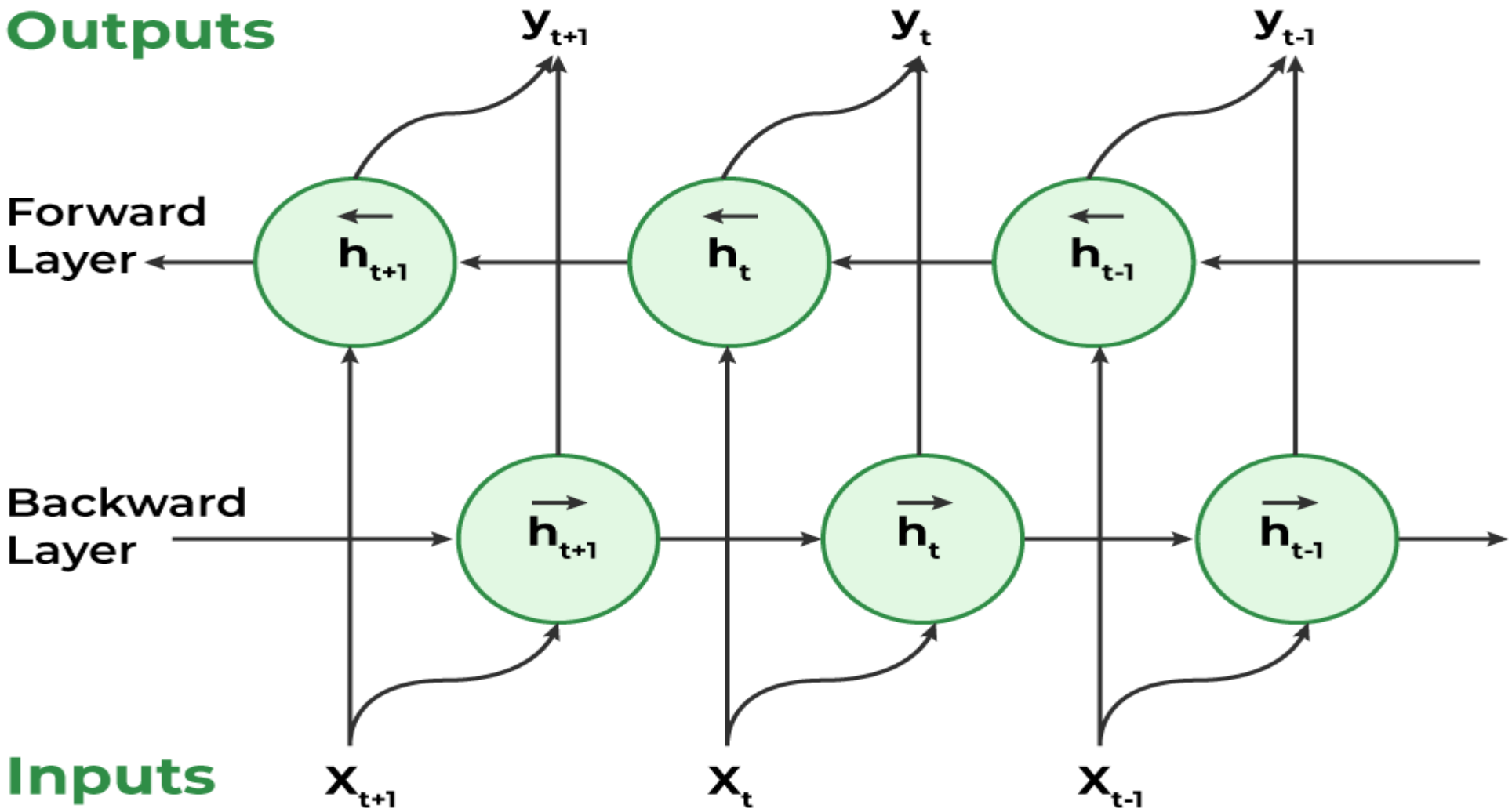
- A **Bidirectional Recurrent Neural Network (BRNN)** is an enhanced form of RNN that processes data in both **forward and backward directions**
- This enables the model to use information from both **past and future contexts**, improving sequence understanding and prediction accuracy.
- While a traditional RNN updates its hidden state only from past inputs, a BRNN includes an additional **backward layer** that processes the sequence in reverse. The outputs from both directions are then **combined**, allowing the network to make more accurate and context-aware predictions.

Example:

Consider the sentence: "I like apple. It is very healthy."

- In a traditional unidirectional RNN the network might struggle to understand whether "apple" refers to the fruit or the company based on the first sentence.
- However a BRNN would have no such issue. By processing the sentence in both directions, it can easily understand that "apple" refers to the fruit, thanks to the future context provided by the second sentence ("It is very healthy.").

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Working of Bidirectional Recurrent Neural Networks (BRNNs)

1. **Inputting a sequence:** A sequence of data points, each represented as a vector with the same dimensionality, are fed into a BRNN. The sequence might have different lengths.
2. **Dual Processing:** Both the forward and backward directions are used to process the data. On the basis of the input at that step and the hidden state at step t-1, the hidden state at time step t is determined in the forward direction. The input at step t and the hidden state at step t+1 are used to calculate the hidden state at step t in a reverse way.
3. **Computing the hidden state:** A non-linear activation function on the weighted sum of the input and previous hidden state is used to calculate the hidden state at each step. This creates a memory mechanism that enables the network to remember data from earlier steps in the process.
4. **Determining the output:** A non-linear activation function is used to determine the output at each step from the weighted sum of the hidden state and a number of output weights. This output has two options: it can be the final output or input for another layer in the network.
5. **Training:** The network is trained through a supervised learning approach where the goal is to minimize the discrepancy between the predicted output and the actual output. The network adjusts its weights in the input-to-hidden and hidden-to-output connections during training through backpropagation.

❖ Applications of Bidirectional Recurrent Neural Networks (BRNNs)

❖ BRNNs are widely used in various natural language processing (NLP) tasks, including:

❖ **Sentiment Analysis**: By considering both past and future context they can better classify the sentiment of a sentence.

❖ **Named Entity Recognition (NER)**: It helps to identify entities in sentences by analyzing the context in both directions.

❖ **Machine Translation**: In encoder-decoder models, BRNNs allow the encoder to capture the full context of the source sentence in both directions hence improving translation accuracy.

❖ **Speech Recognition**: By considering both previous and future speech elements it enhance the accuracy of transcribing audio.

Advantages of BRNNs

•**Enhanced Context Understanding**: Considers both past and future data for improved predictions.

•**Improved Accuracy**: Particularly effective for NLP and speech processing tasks.

•**Better Handling of Variable-Length Sequences**: More flexible than traditional RNNs making it suitable for varying sequence lengths.

•**Increased Robustness**: Forward and backward processing help filter out noise and irrelevant information, improving robustness.

Challenges of BRNNs

•**High Computational Cost**: Requires twice the processing time compared to unidirectional RNNs.

•**Longer Training Time**: More parameters to optimize result in slower convergence.

•**Limited Real-Time Applicability**: Since predictions depend on the entire sequence hence they are not ideal for real-time applications like live speech recognition.

•**Less Interpretability**: The bidirectional nature of BRNNs makes it more difficult to interpret predictions compared to standard RNNs.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Long Short-Term Memory (LSTM).

- Long Short-Term Memory is an advanced version of **recurrent neural network (RNN) architecture**.
- LSTM networks are an **extension** of recurrent neural networks (RNNs) mainly introduced to handle situations where **RNNs fail.**
- The problems of RNNs, namely, the **vanishing and exploding gradients**, and provides a convenient solution to these problems in the form **of Long Short Term Memory (LSTM).**
- It fails to store information for a **longer period of time**. At times, a reference to certain information stored quite a long time ago is required to predict the current output. But RNNs are absolutely incapable of handling such "long-term dependencies".
- There is **no finer control** over which part of the **context needs to be carried** forward and how much of the **past needs to be 'forgotten'.**

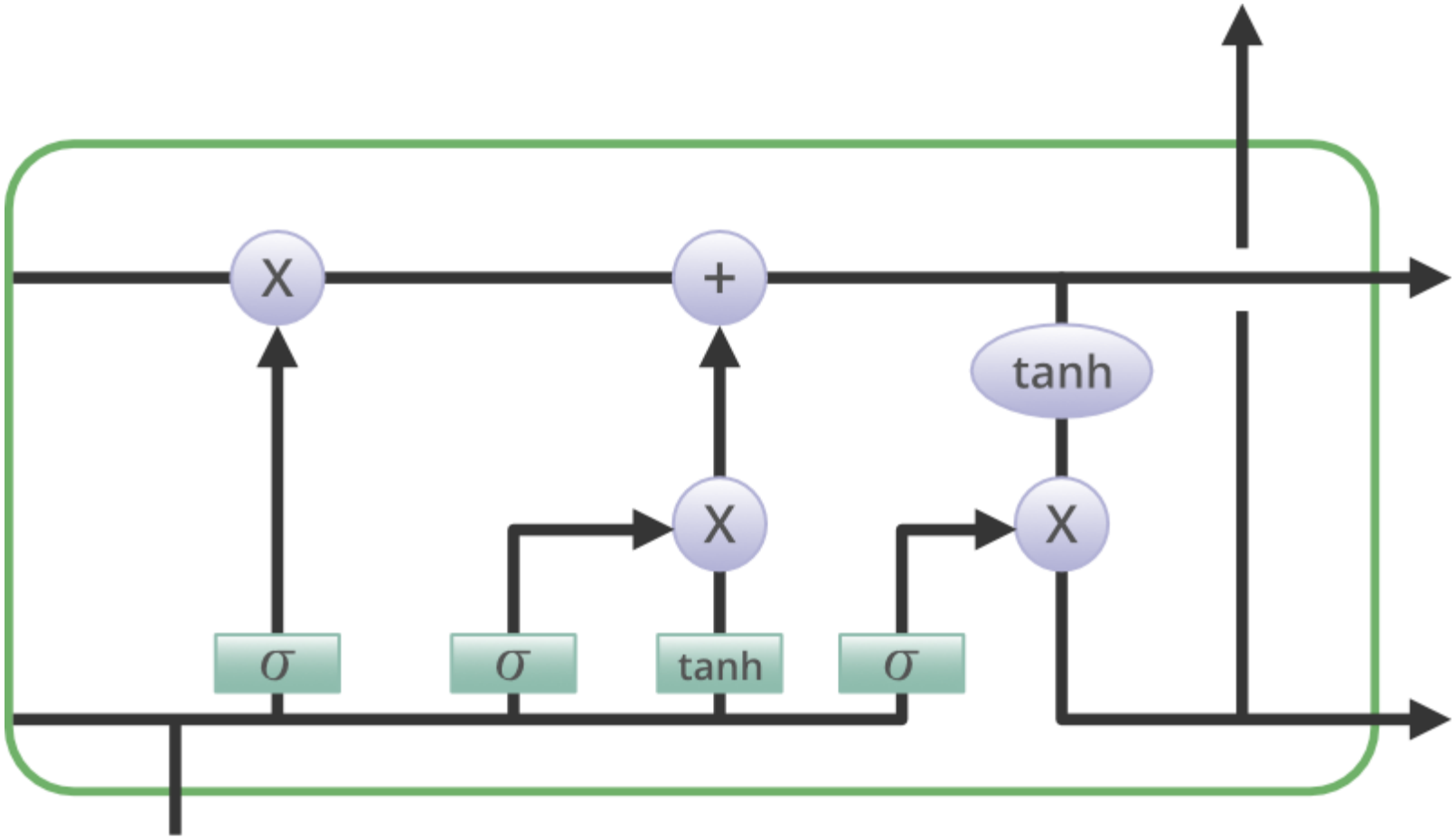
Structure of LSTM

- The main difference between RNNs and LSTMs is in their **hidden layer design**. In RNNs, the hidden layer has only one simple neural network layer using the **tanh** function.
- But in LSTMs, the hidden layer is more advanced. It has a **gated unit (cell)** made up of **four layers** that work together. These layers produce two things:

  1 .The **output** of the current cell

  2.The **cell state**, which carries important information to the next cell
- LSTMs have **three gates** (forget gate, input gate, and output gate) that use the **sigmoid** function, and one **tanh** layer.
- These **gates control the flow of information** — they decide what to remember and what to forget.
- Each gate gives an output between **0 and 1**:

1. **0 means "reject all" (ignore the information)**
2. **1 means "accept all" (keep the information)**

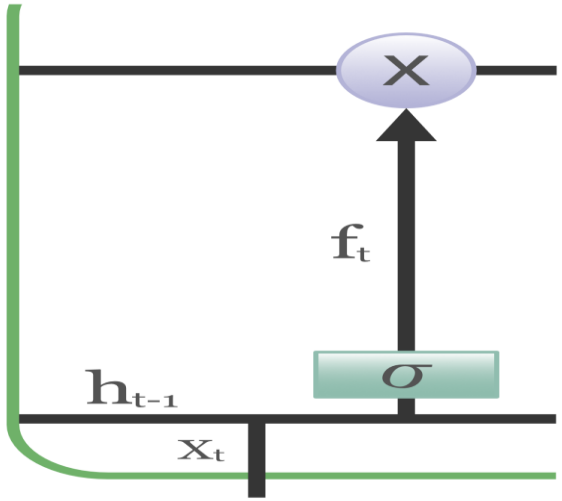

X
+
tanh
X
X
σ
σ
tanh
σ

Gate	Activation	Role
Forget Gate	σ	Remove unneeded memory
Input Gate	σ + tanh	Add new information
Cell State	+	Maintain long-term memory
Output Gate	σ + tanh	Produce output for next step

Forget Gate

- ✓ The **Forget Gate** helps the LSTM decide what information to **remove** from the cell state because it is no longer useful.
- ✓ **It takes two inputs:**
- ✓ $\mathbf{x_t}$ – the input at the current time step
- ✓ $\mathbf{h_{t-1}}$ – the output from the previous cell
- ✓ These inputs are multiplied by their **weights** and a **bias** is added. The result then goes through a **sigmoid activation function**, which gives an output between **0 and 1**.
- ✓ If the output is **0**, it means the information should be **forgotten**.
- ✓ If the output is **1**, it means the information should be **kept** for the future.

$$ft=\sigma(Wf\cdot[ht-1,xt]+bf)$$



Input Gate

The **Input Gate** controls how much **new information** should be added to the cell state.

It works in three steps:

Filtering important information:

Like the forget gate, it takes the previous output ($\mathbf{h_{t-1}}$) and the current input ($\mathbf{x_t}$) and passes them through a **sigmoid function**.

This helps decide which information is **important** (values close to 1) and which should be **ignored** (values close to 0).

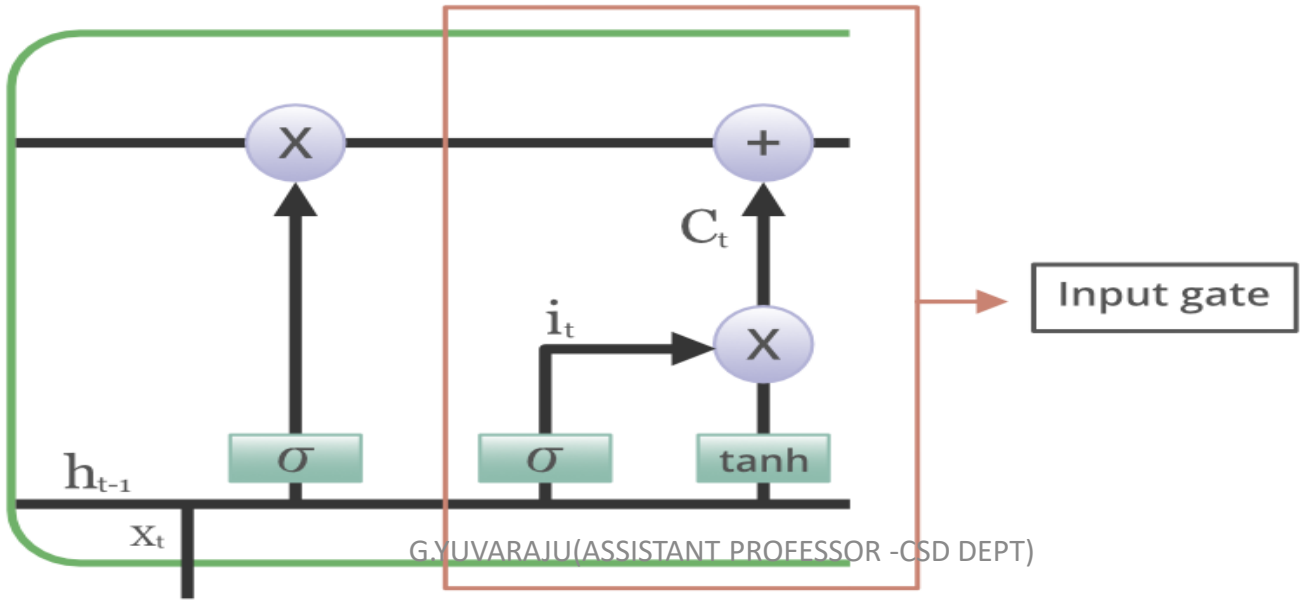
Creating new candidate values:

A **tanh function** is then used to create a **vector** of new possible values (ranging from **–1 to +1**) that could be added to the cell state.

Combining the results:

The filtered values from the sigmoid function are **multiplied** with the new candidate values from the tanh function.

This gives the **useful new information** that will be **added to the cell state**.



Output Gate

The **Output Gate** controls what information from the current cell state should be sent as **output** to the next step.
It works as follows:

1.Creating a candidate output:

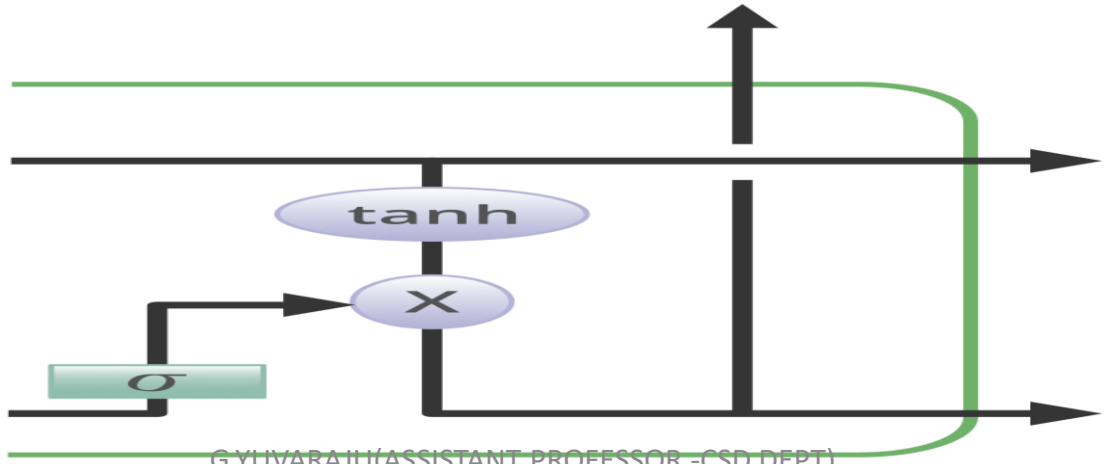
The current **cell state** is passed through the **tanh function**, which gives a vector of values between **–1 and +1**. This represents the possible output information.

2.Filtering important information:

The previous output ($\mathbf{h_{t-1}}$) and the current input ($\mathbf{x_t}$) are passed through a **sigmoid function**. This helps decide which parts of the information should be **sent out**.

3.Producing the final output:

The results from the **tanh** and **sigmoid** functions are **multiplied** to produce the **final output** ($\mathbf{h_t}$).
This output is then sent both **to the next cell** and **as the output of the current time step**.



Step-by-step working of one LSTM cell

Let's look at what happens when the LSTM reads the word **"very."**

1. Forget Gate

•It looks at the previous word (**"feeling"**) and decides what old information to keep or forget.

•For example, it may **forget small details** like the word "am," but **keep the emotional context** from "feeling."

•If the forget gate gives 0, that info is **removed**; if it gives 1, it is **kept**.

So, it might keep "emotion-related" information and forget unrelated grammar words.

2. Input Gate

•Now, it looks at the current word (**"very"**) and decides **what new information** to add.

•The sigmoid layer decides **which parts** of "very" are important (e.g., intensity of emotion).

•The tanh layer creates **possible new values** (like "strong emotion").

•These two are multiplied to update the cell state with new, useful information.

So now, the LSTM's cell state stores something like "strong feeling."

3. Output Gate

•Finally, it decides **what to show as output** and **what to pass to the next word.**

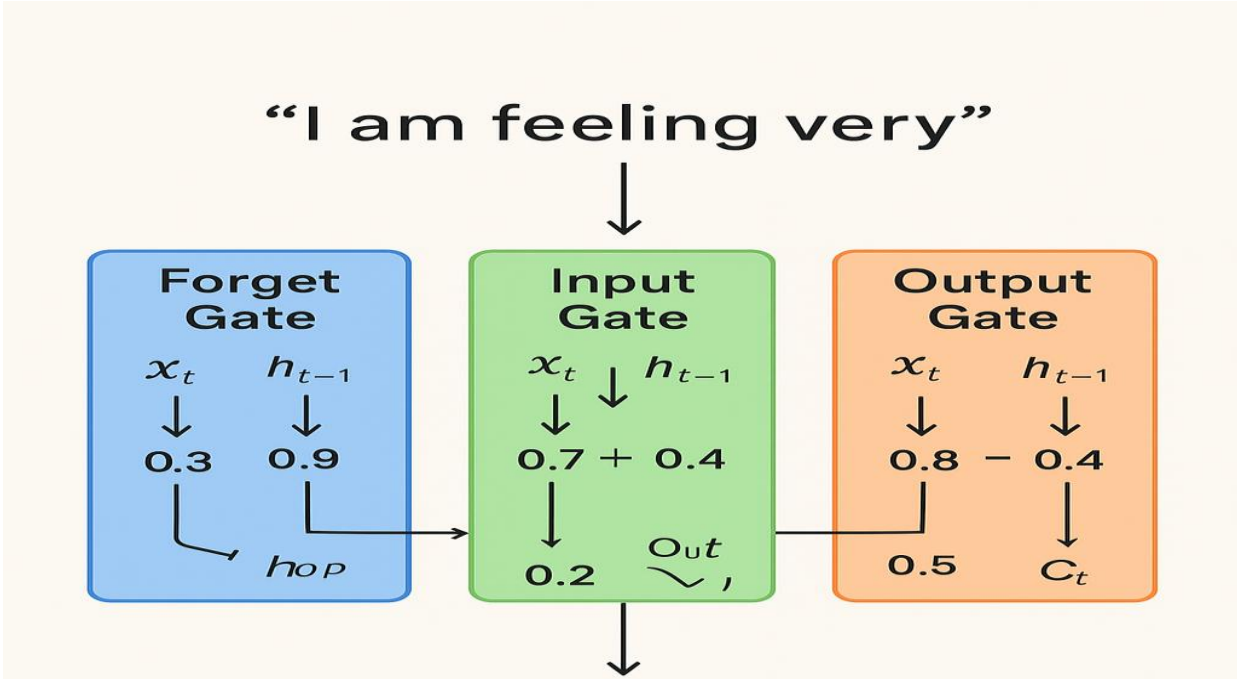
•The cell state goes through **tanh** (to get values between –1 and +1).

•Then, it's filtered by a **sigmoid** function that decides which part of the information to output.

•The result is the hidden output ($\mathbf{h_t}$) that goes to the next word (**"happy"** or **"tired"** prediction).

So, the LSTM might output something like "strong positive emotion," helping the model choose **"happy"** as the next word.

Gate	Purpose	Example Action
Forget Gate	Remove unimportant info	Forget "am"
Input Gate	Add new useful info	Add "strong" from "very"
Output Gate	Decide what to show	Output "strong emotion"



Forget Gate	Forget "am"
Input Gate	Add "strong" from "very"
Output Gate	Output "strong emotion"

Applications of LSTM:

Language Modeling: LSTMs have been used to build language models that can generate natural language text, such as in machine translation systems or chatbots.

Time series prediction: LSTMs have been used to model time series data and predict future values in the series. For example, LSTMs have been used to predict stock prices or traffic patterns.

Sentiment analysis: LSTMs have been used to analyze text sentiments, such as in social media posts or customer reviews.

Speech recognition: LSTMs have been used to build speech recognition systems that can transcribe spoken language into text.

Image captioning: LSTMs have been used to generate descriptive captions for images, such as in image search engines or automated image annotation systems.

Bidirectional LSTM

- Bidirectional Long Short-Term Memory (BiLSTM) is an extension of traditional LSTM network.
- Unlike conventional LSTMs that process sequences in only one direction, BiLSTMs allow information to flow from both forward and backward enabling them to capture more contextual information.
- This makes BiLSTMs particularly effective for tasks where understanding both past and future context is crucial.

Understanding Bidirectional LSTM (BiLSTM)

A Bidirectional LSTM (BiLSTM) consists of two separate LSTM layers:

- **Forward LSTM**: Processes the sequence from start to end
- **Backward LSTM**: Processes the sequence from end to start

The outputs of both LSTMs are then combined to form the final output. Mathematically, the final output at time **t** is computed as:

$$p_t = p_{t_f} + p_{t_b}$$

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

- It is usually used in NLP-related tasks. The intuition behind this approach is that by processing data in both directions, the model is able to better understand the relationship between sequences (e.g. knowing the following and preceding words in a sentence).

- For example, The first statement is "**Server can you bring me this dish"** and the second statement is
- "**He crashed the server**".
- In both these statements, the word server has different meanings and this relationship depends on the following and preceding words in the statement.
- The bidirectional LSTM helps the machine to understand this relationship better than compared with unidirectional LSTM.
- This ability of BiLSTM makes it a suitable architecture for tasks like sentiment analysis, text classification,
- and machine translation.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

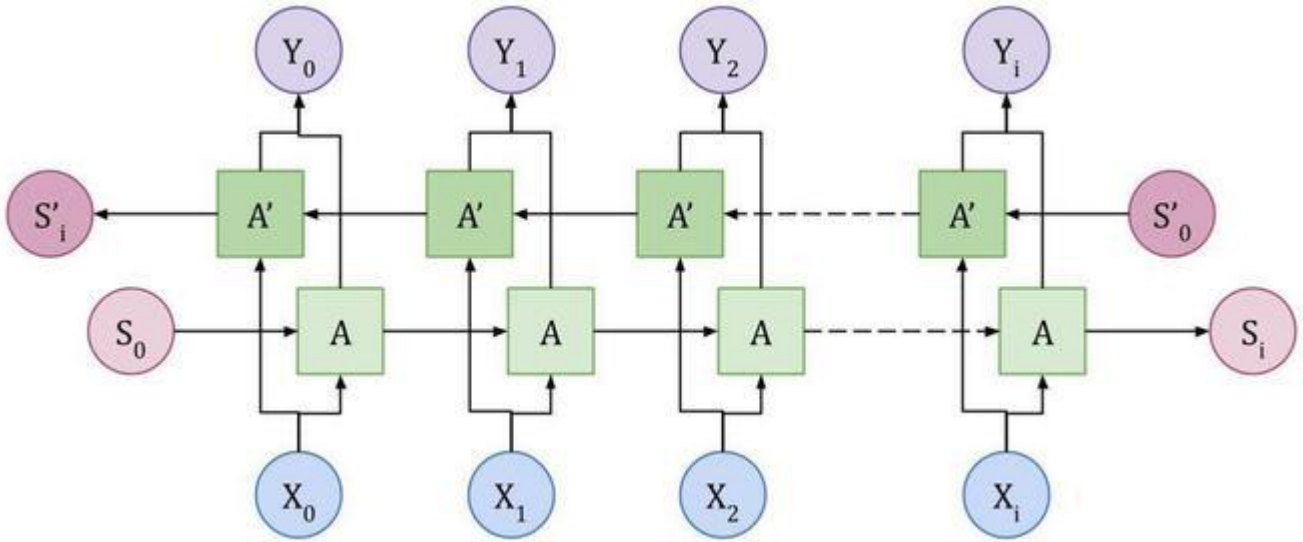
Architecture:

- The architecture of bidirectional LSTM comprises of two unidirectional LSTMs which process the sequence in both forward and backward directions.
- This architecture can be interpreted as having two separate LSTM networks, one gets the sequence of tokens as it is while the other gets in the reverse order.
- Both of these LSTM network returns a probability vector as output and the final output is the combination of both of these probabilities.it can be represented as
- p_t = p_{t_f} + p_{t_b}

Where:

- p_t: Final probability vector of the network.
- p_{tf}: Probability vector from the forward LSTM network.
- p_{tb}: Probability vector from the backward LSTM network.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Here:

•X_i is the input token

•Y_i is the output token

•A and A' are Forward and backward LSTM units

•The final output of Y_i is the combination of A and A' LSTM nodes.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Deep Recurrent Networks (Deep RNNs)

A **Deep Recurrent Network** is formed by **stacking multiple RNN layers** on top of each other instead of using a single RNN layer.

Mathematical Representation

For a 2-layer Deep RNN:

First Layer:

$$h_t^{(1)} = f\left(W^{(1)} h_{t-1}^{(1)} + U^{(1)} x_t\right)$$

Second Layer:

$$h_t^{(2)} = f\left(W^{(2)} h_{t-1}^{(2)} + U^{(2)} h_t^{(1)}\right)$$

General form for layer l:

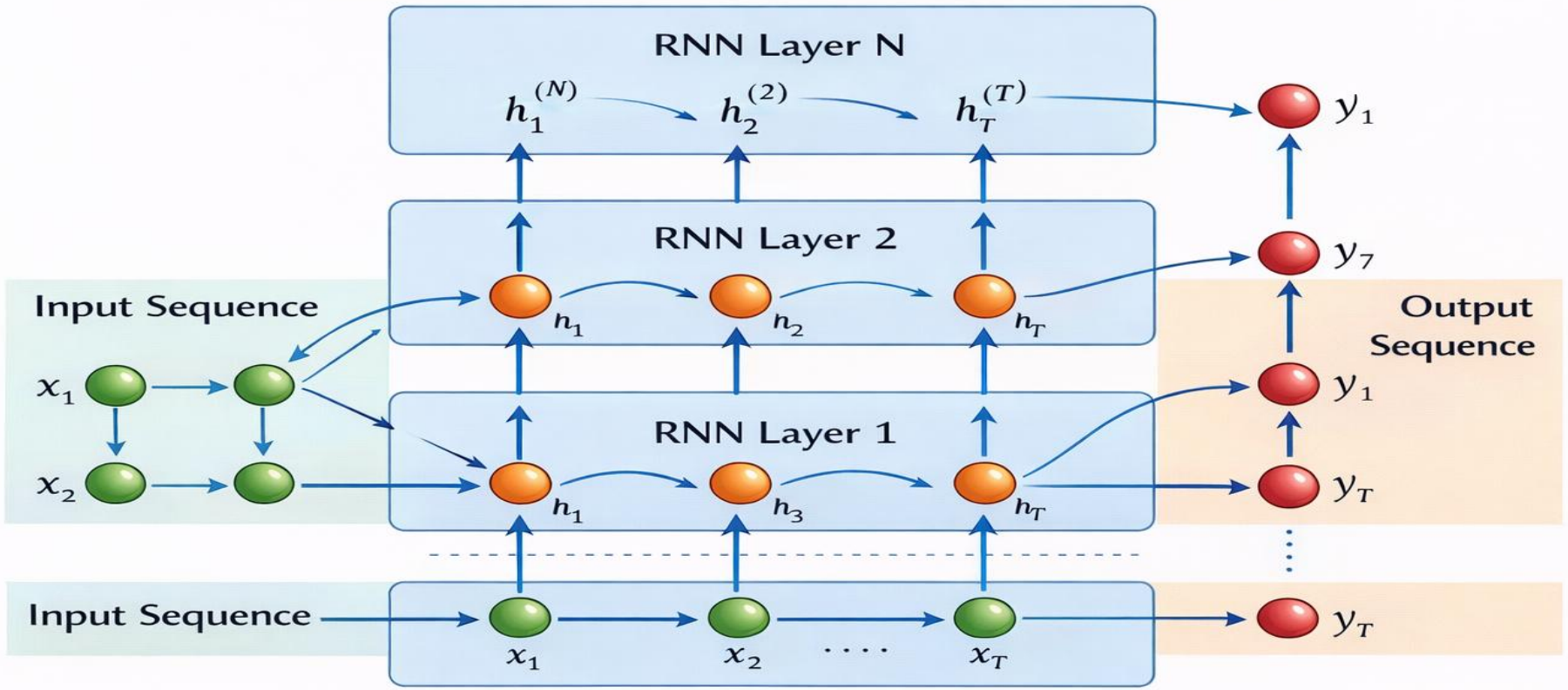
$$h_t^{(l)} = f\left(W^{(l)} h_{t-1}^{(l)} + U^{(l)} h_t^{(l-1)}\right)$$

Where:

- t= time step
- l= layer number

Deep Recurrent Networks

Stack multiple RNN layers to learn temporal features across multiple levels.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

How It Works

At each time step:

1.Input goes to first RNN layer

2.First layer produces hidden state

3.That hidden state becomes input to next layer

4.Final layer produces output

So depth is added in the **vertical direction**, while time unfolds horizontally

Why Use Deep RNNs?

Single-layer RNN:

Learns simple temporal patterns

Deep RNN:

Learns hierarchical temporal features

Captures complex patterns

Improves modeling power

Example:

Lower layers → capture short-term patterns

Higher layers → capture long-term structure

.

Gated RNN

A **Gated RNN** is a type of Recurrent Neural Network that uses **gating mechanisms** to control how information flows through time.

These gates help the network:

- Remember important past information
- Forget irrelevant information
- Solve the **vanishing gradient problem**

The two most common gated RNNs are:

- **Long Short-Term Memory (LSTM)**
- **Gated Recurrent Unit (GRU)**

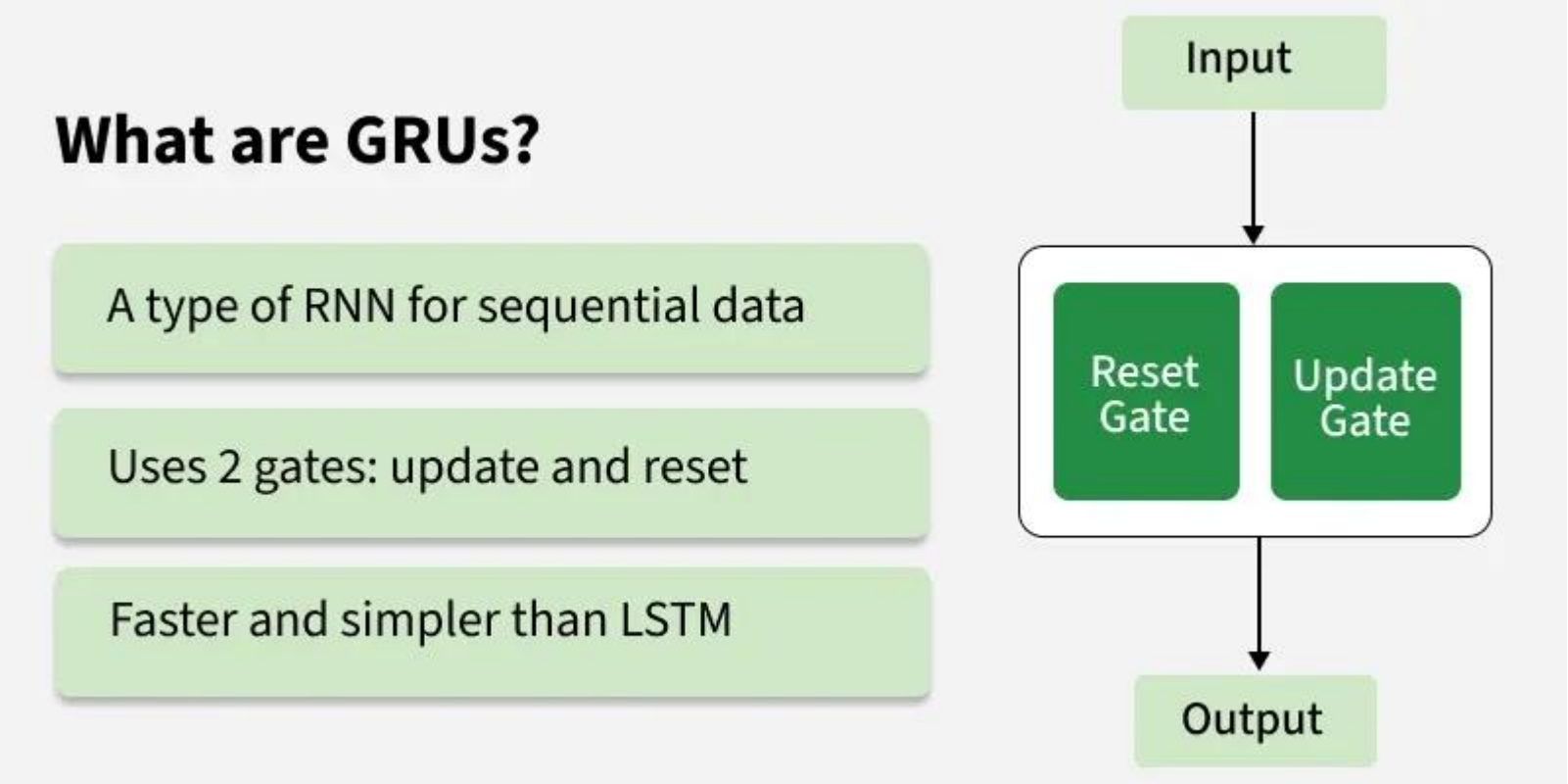
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Gated Recurrent Unit Networks

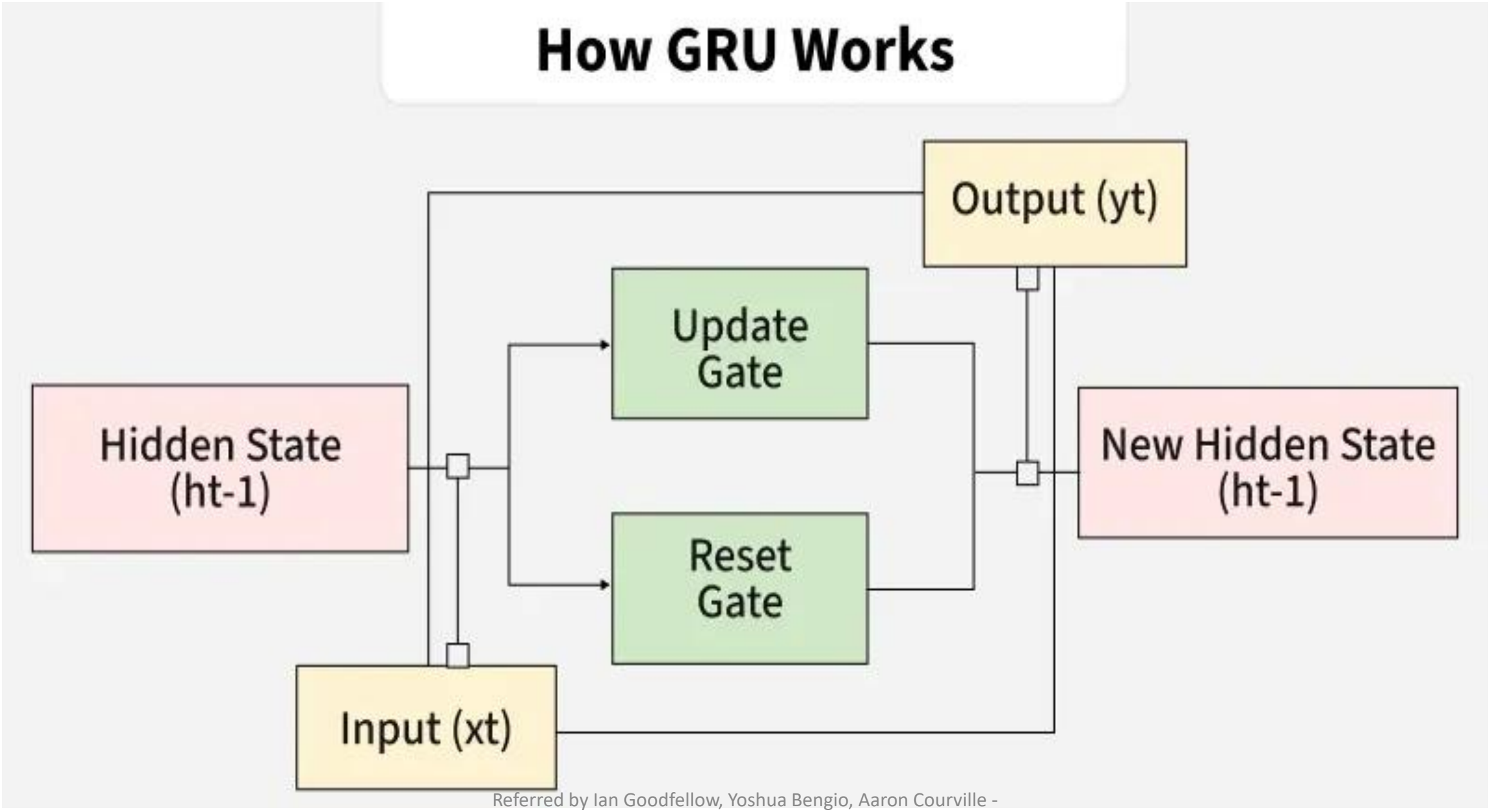
Gated Recurrent Unit (GRU) networks are a type of recurrent neural network designed to handle sequential data while reducing the complexity of traditional RNNs.

GRUs are a simplified advancement of LSTM, where they merge multiple gates into update and reset gates, hence learning long-term dependencies with faster training and fewer parameters.

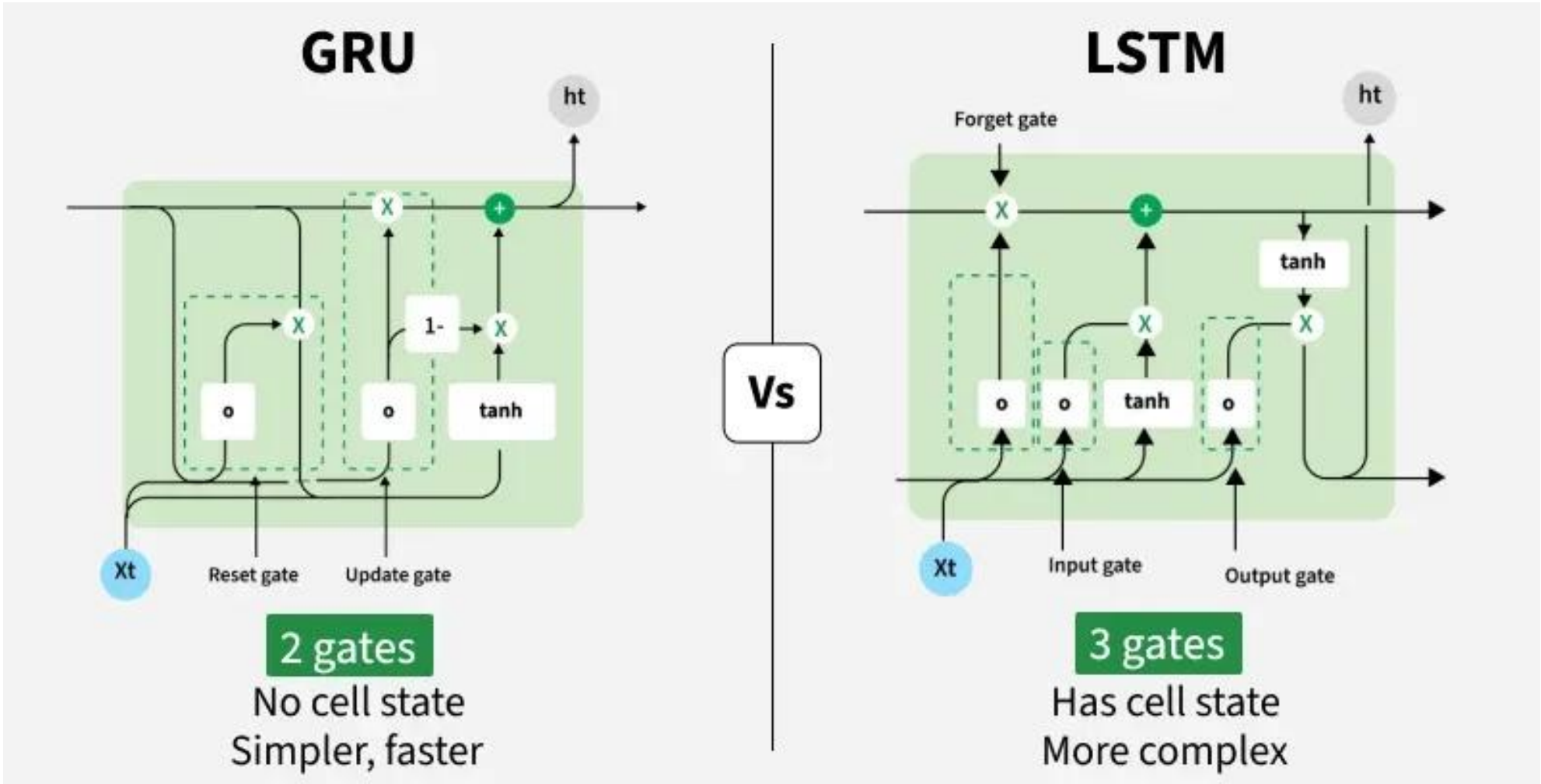
- Simplified alternative to LSTM
- Handles sequence and time-series data effectively
- Widely used in NLP, speech and forecasting tasks



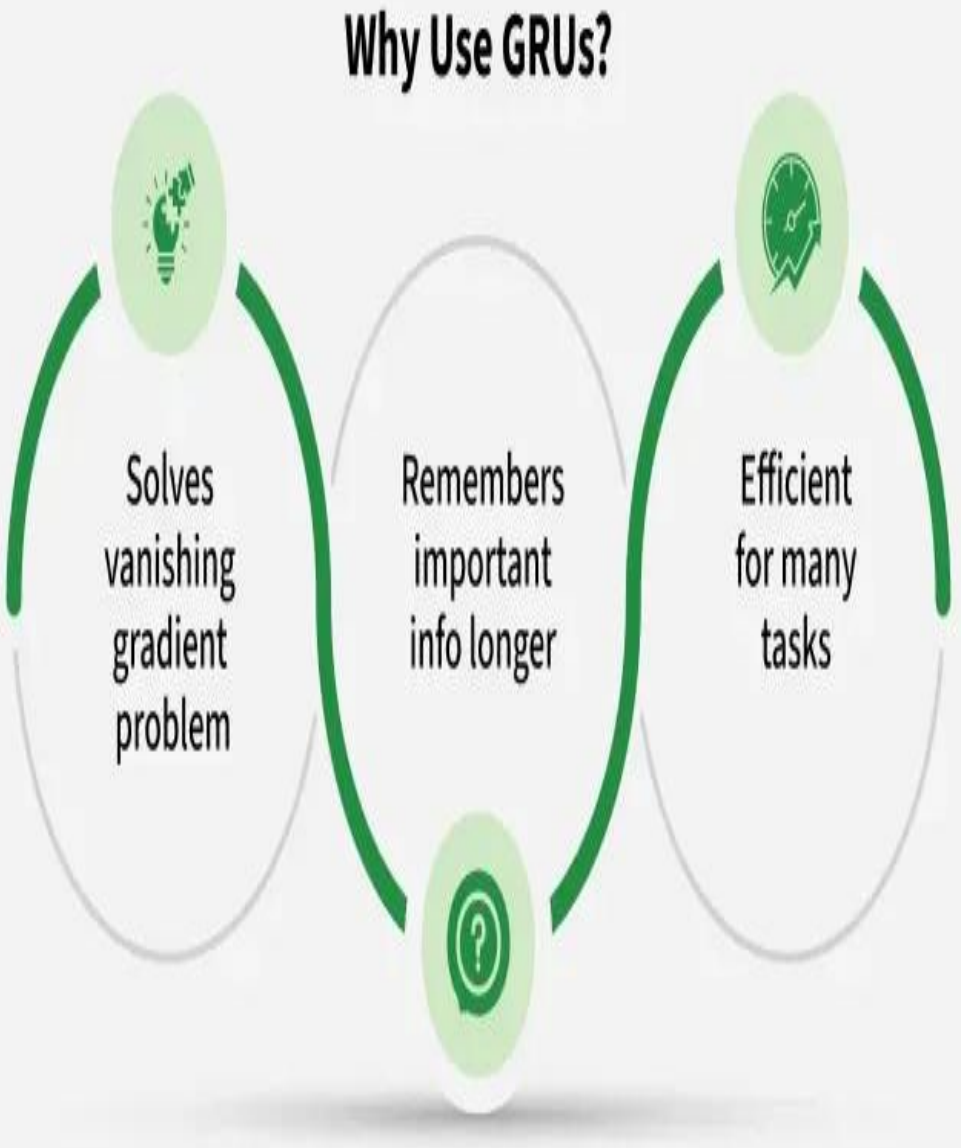
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)



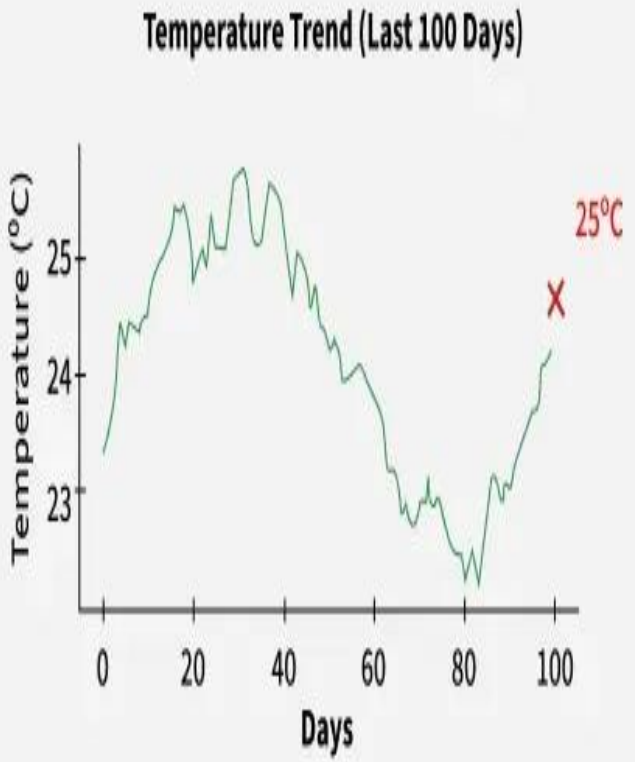
GRU Model & Prediction

Load and preprocess data

Build and train GRU model

Predict next day temperature using last 100 days

Example output: 25°C



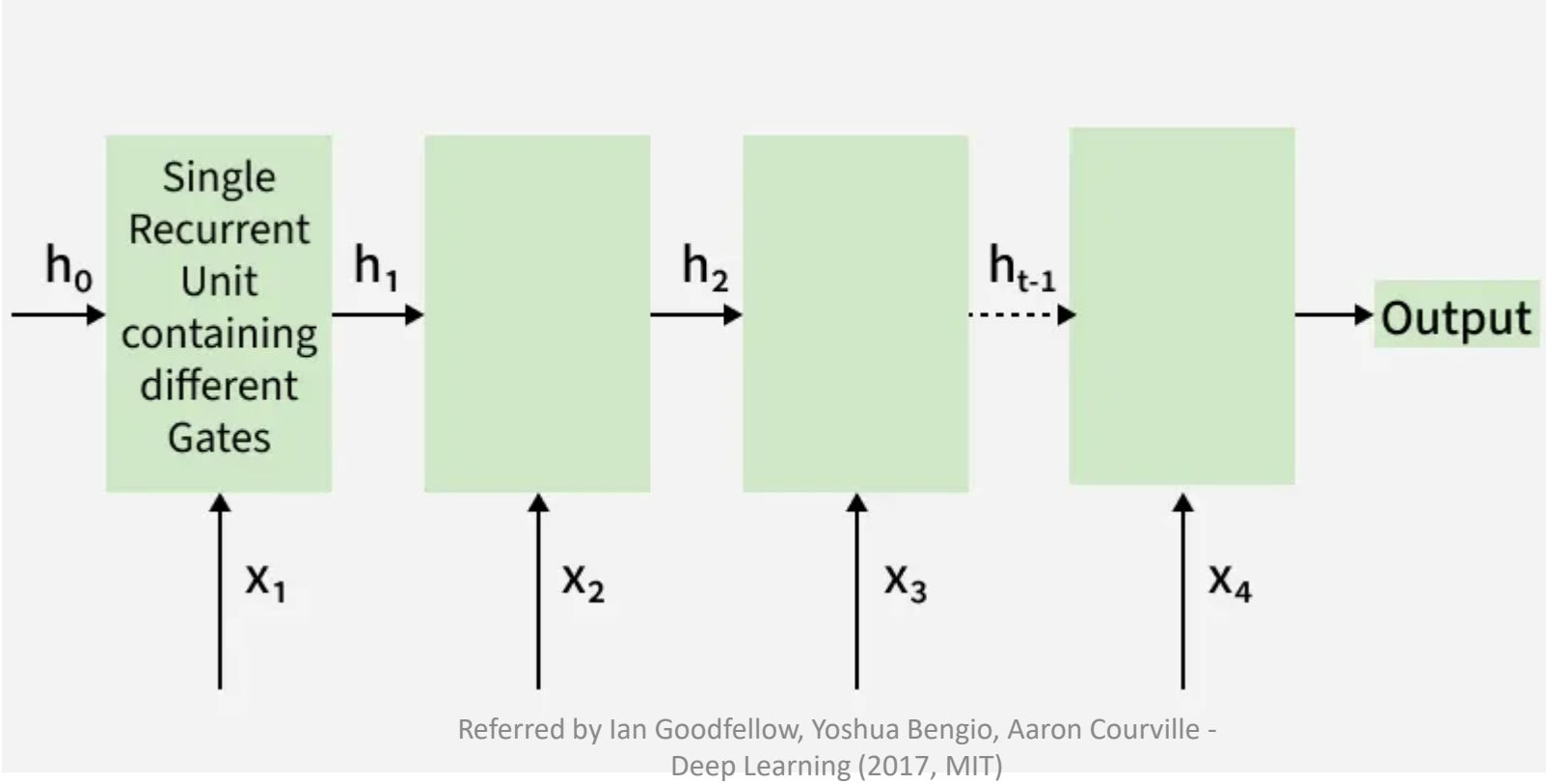
Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

What are Gated Recurrent Units (GRU)?

Gated Recurrent Units (GRUs) are a type of RNN introduced by Cho et al. in 2014.

The core idea behind GRUs is to use gating mechanisms to selectively update the hidden state at each time step allowing them to remember important information while discarding irrelevant details.

GRUs aim to simplify the LSTM architecture by merging some of its components and focusing on just two main gates: the update gate and the reset gate.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1.Update Gate (z_t :(This gate decides how much information from previous hidden state should be retained for the next time step.

2.Reset Gate (r_t :(This gate determines how much of the past hidden state should be forgotten.

These gates allow GRU to control the flow of information in a more efficient manner compared to traditional RNNs which solely rely on hidden state.

These gates allow GRU to control the flow of information in a more efficient manner compared to traditional RNNs which solely rely on hidden state.

Equations for GRU Operations

The internal workings of a GRU can be described using following equations:

1. Reset gate:

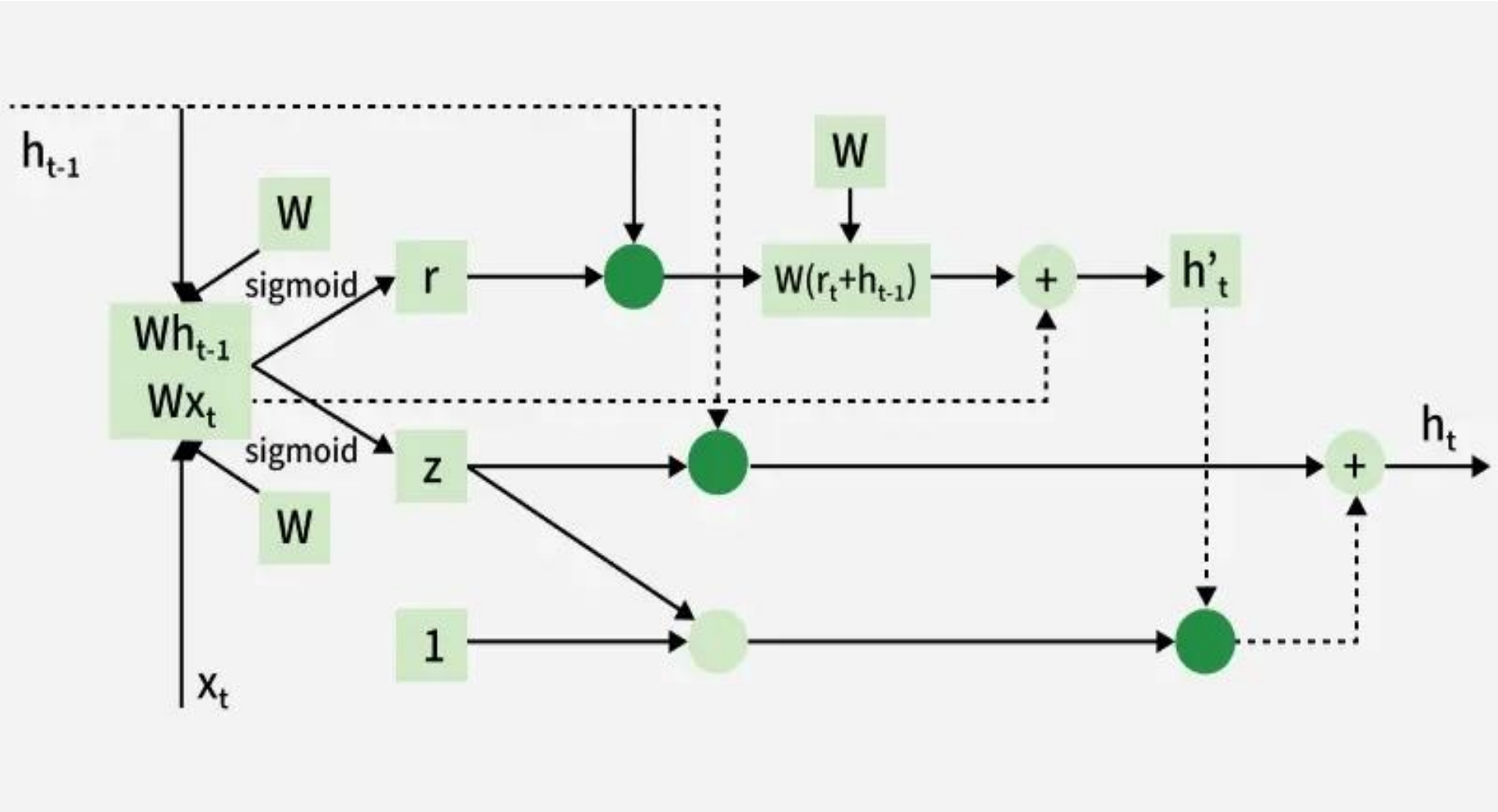
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

The reset gate determines how much of the previous hidden state h_{t-1} should be forgotten.

2. Update gate:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

The update gate controls how much of the new information x_t should be used to update the hidden state.



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

3. Candidate hidden state:

This is the potential new hidden state calculated based on the current input and the previous hidden state.

4. Hidden state:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot h'_t$$

The final hidden state is a weighted average of the previous hidden state h_{t-1} and the candidate hidden state h'_t based on the update gate z_t.

How GRUs Solve the Vanishing Gradient Problem

Like LSTMs, GRUs were designed to address the vanishing gradient problem which is common in traditional RNNs.

GRUs help mitigate this issue by using gates that regulate the flow of gradients during training ensuring that important information is preserved and that gradients do not shrink excessively over time.

By using these gates, GRUs maintain a balance between remembering important past information and learning new, relevant data.

GRU vs LSTM

GRUs are more computationally efficient because they combine the forget and input gates into a single update gate. .

GRUs do not maintain an internal cell state as LSTMs do, instead they store information directly in the hidden state making them simpler and faster

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Feature	LSTM (Long Short-Term Memory)	GRU (Gated Recurrent Unit)
Gates	3 (Input, Forget, Output)	2 (Update, Reset)
Cell State	Yes it has cell state	No (Hidden state only)
Training Speed	Slower due to complexity	Faster due to simpler architecture
Computational Load	Higher due to more gates and parameters	Lower due to fewer gates and parameters
Performance	Often better in tasks requiring long-term memory	Performs similarly in many tasks with less complexity

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Optimization for Long-Term Dependencies

What are Long-Term Dependencies?
In sequence problems (text, speech, time-series), sometimes the model must remember information from **far back** in the sequence.
Example:
👉 *"I grew up in France… so I speak fluent ___."*
To predict **"French"**, the model must remember **France**, even though it appeared many words earlier.
That's a **long-term dependency**.
1. Vanishing Gradient Problem
2. Exploding Gradient Problem

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Optimization Strategies for Long-Term Dependencies

1. Use LSTM or GRU (Best Solution)

They have **gates** that control memory:

- Forget gate
- Input gate
- Output gate

So they can remember important info for a long time.

✓ Solves vanishing gradient better than plain RNN.

2. Gradient Clipping

Used to stop exploding gradients.

Instead of allowing huge updates:

✓ Keeps training stable.

3. Better Weight Initialization

Proper initialization prevents gradients from shrinking too fast.

Common methods:

- Xavier Initialization
- He Initialization

✓ Helps faster convergence.

4. Use Shorter Sequences (Truncated BPTT)

Instead of backpropagating through the full sequence:

- break it into smaller chunks

✓ Reduces computation and gradient issues.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

5. Skip Connections / Residual Connections

Adding shortcut paths helps gradient flow.

Used in deep sequence models.

✓ Prevents vanishing gradients.

6. Attention Mechanism

Instead of remembering everything, the model can:

•directly focus on relevant past words

Example:

Transformer models use attention completely.

✓ Best modern solution for long dependencies.

7. Use Adaptive Optimizers

Optimizers like:

•Adam

•RMSProp

adjust learning rates automatically.

✓ Works better for sequence training.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Autoencoders

Autoencoders are a special type of neural networks that learn to compress data into a compact form and then reconstruct it to closely match the original input. They consist of an:

•Encoder that captures important features by reducing dimensionality.

•Decoder that rebuilds the data from this compressed representation.

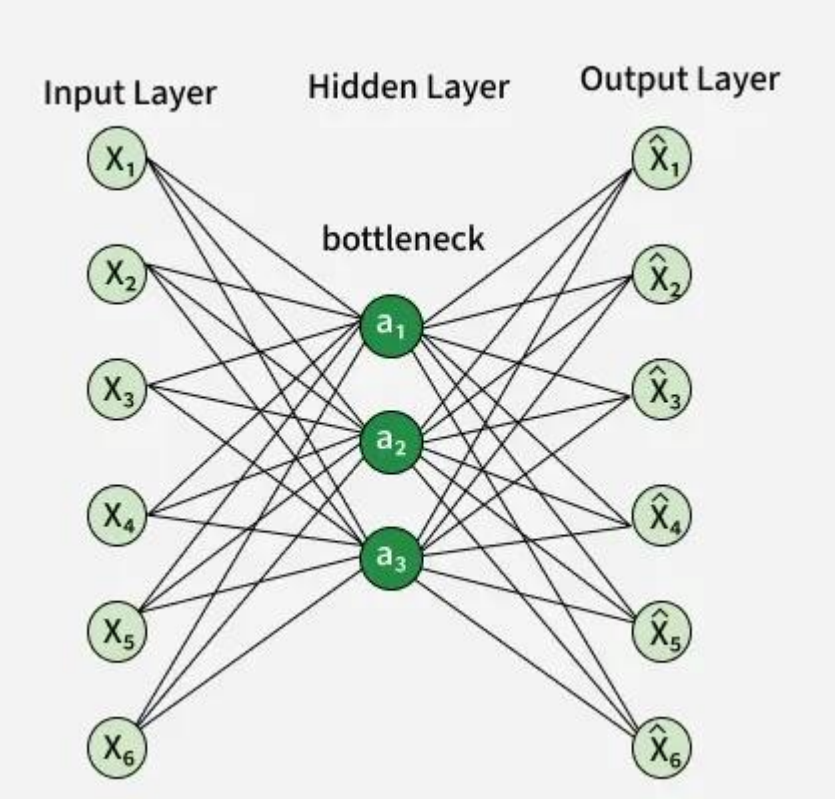
The model trains by minimizing reconstruction error using loss functions like Mean Squared Error or Binary Cross-Entropy.

These are applied in tasks such as noise removal, error detection and feature extraction where capturing efficient data representations is important.

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Architecture of Autoencoder

An autoencoder's architecture consists of three main components that work together to compress and then reconstruct data which are as follows:



Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

1. Encoder

It compress the input data into a smaller, more manageable form by reducing its dimensionality while preserving important information. It has three layers which are:

•**Input Layer**: This is where the original data enters the network. It can be images, text features or any other structured data.

•**Hidden Layers**: These layers perform a series of transformations on the input data. Each hidden layer applies weights and activation functions to capture important patterns, progressively reducing the data's size and complexity.

•**Output(Latent Space)**: The encoder outputs a compressed vector known as the latent representation or encoding. This vector captures the important features of the input data in a condensed form helps in filtering out noise and redundancies.

2. Bottleneck (Latent Space)

It is the smallest layer of the network which represents the most compressed version of the input data. It serves as the information bottleneck which force the network to prioritize the most significant features. This compact representation helps the model learn the underlying structure and key patterns of the input helps in enabling better generalization and efficient data encoding.

3. Decoder

It is responsible for taking the compressed representation from the latent space and reconstructing it back into the original data form.

•**Hidden Layers**: These layers progressively expand the latent vector back into a higher-dimensional space. Through successive transformations decoder attempts to restore the original data shape and details

•**Output Layer**: The final layer produces the reconstructed output which aims to closely resemble the original input. The quality of reconstruction depends on how well the encoder-decoder pair can minimize the difference between the input and output during training

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

Efficient Representations in Autoencoders

Constraining an autoencoder helps it learn meaningful and compact features from the input data which leads to more efficient representations. After training only the encoder part is used to encode similar data for future tasks. Various techniques are used to achieve this are as follows:

- **Keep Small Hidden Layers**: Limiting the size of each hidden layer forces the network to focus on the most important features. Smaller layers reduce redundancy and allows efficient encoding.

- **Regularization**: Techniques like L1 or L2 regularization add penalty terms to the loss function. This prevents overfitting by removing excessively large weights which helps in ensuring the model to learns general and useful representations.

- **Denoising:** In denoising autoencoders random noise is added to the input during training. It learns to remove this noise during reconstruction which helps it focus on core, noise-free features and helps in improving robustness.

- **Tuning the Activation Functions**: Adjusting activation functions can promote sparsity by activating only a few neurons at a time. This sparsity reduces model complexity and forces the network to capture only the most relevant features.

Types of Autoencoders
Lets see different types of Autoencoders which are designed for specific tasks with unique features:

1. Denoising Autoencoder

Denoising Autoencoder is trained to handle corrupted or noisy inputs, it learns to remove noise and helps in reconstructing clean data. It prevent the network from simply memorizing the input and encourages learning the core features.

2. Sparse Autoencoder

Sparse Autoencoder contains more hidden units than input features but only allows a few neurons to be active simultaneously. This sparsity is controlled by zeroing some hidden units, adjusting activation functions or adding a sparsity penalty to the loss function.

Deep Generative Models (DGM)

What are they?

Deep Generative Models are neural networks that learn the pattern of data and can **generate new similar data**.

Instead of saying:

“This is a cat”

They say:

“I can create a new cat image!”

What can they generate?

- Deep generative models can create:
- Images (faces, objects)
  Text (stories, code)
  Music
  Videos
  Synthetic medical data
  New designs

Types of Deep Generative Models

1. Autoencoders (AE)

Idea:

Compress → Reconstruct

•Encoder: reduces input to latent space

•Decoder: rebuilds original input

Used for: feature learning, denoising

2. Variational Autoencoders (VAE)

Idea:

Generate new data by learning probability distribution.

•Latent space is smooth

•Can sample new points

📌 Used for: image generation, anomaly detection

Referred by Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning (2017, MIT)

3. **Generative Adversarial Networks (GANs)**

Idea:

Two networks compete:

Generator → creates fake data

Discriminator → detects real vs fake

Training is like a game:

•Generator improves

•Discriminator improves

Used for: realistic image generation

4. Autoregressive Models

Generate step-by-step

Example:

•Predict next word based on previous words

Used for: language models like GPT

5. Diffusion Models (Most modern)

Idea:

Noise → Remove noise → Create image

They generate very high-quality outputs.

Used in: DALL·E, Stable Diffusion