Chapter 4

Artificial Neural Network

1. **Artificial Neural Network [12 hours]**

   4.1 Introduction to Neural Network
   Neural Network Architectures: Feedforward, Convolution, Recurrent
   Perceptron: Single Layer perceptron, Multilayer perceptron, Backpropagation

   4.2 Training Neural Network
   Forward and Backward propagation:
   Forward propagation
   Backpropagation and Gradient Descent
   Loss Functions:
   Role of loss function
   Mean Squared Error (MSE)
   Cross-entropy Loss
   Regularization techniques:
   Overfitting and underfitting
   Regularization methods: L1, L2, Dropout, Batch normalization

   4.3 Advanced Neural Network Architecture
   Convolution Neural Networks (CNNs):
   CNNs and their components
   Convolution, Pooling and fully connected layers
   Application in Image processing and computer vision
   Recurrent Neural Networks:
   Basics of RNNs
   Long Short-term Memory (LSTM)
   Gradient Recurrent Units (GRU)
   Applications of Time-series prediction

4. Artificial Neural Network [12 hours]

4.1 Introduction to Neural Network

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

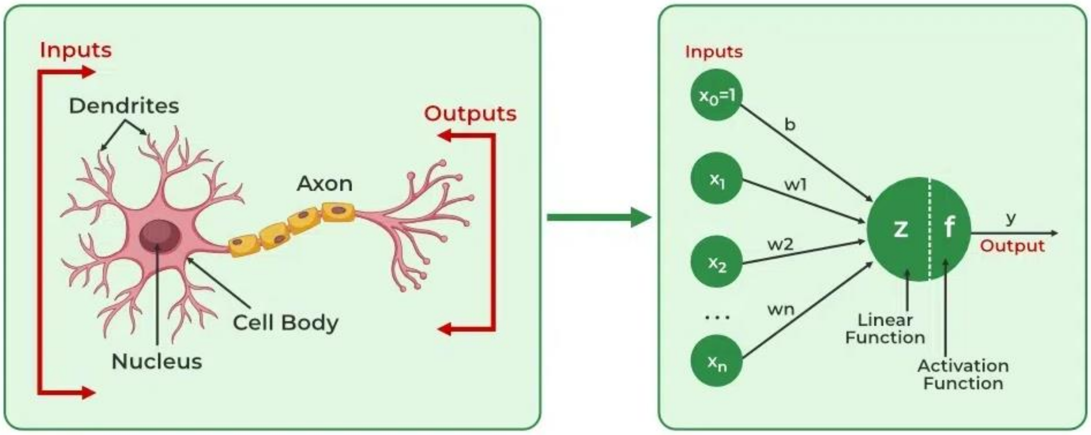
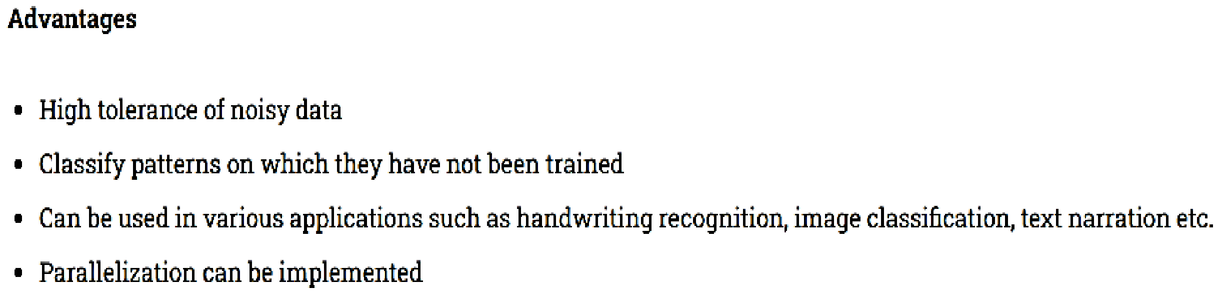


Figure: Illustration of biological neuron and an artificial neuron, showing how inputs are received and processed to produce outputs in both systems

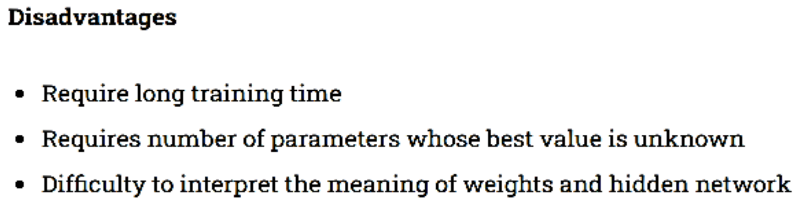
Neural networks are capable of learning and identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

1. **Neurons:** The basic units that receive inputs, each neuron is governed by a threshold and an activation function.
2. **Connections:** Links between neurons that carry information, regulated by weights and biases.
3. **Weights and Biases:** These parameters determine the strength and influence of connections.
4. **Propagation** Functions: Mechanisms that help process and transfer data across layers of neurons.
5. **Learning Rule:** The method that adjusts weights and biases over time to improve accuracy.



Advantages

- High tolerance of noisy data
- Classify patterns on which they have not been trained
- Can be used in various applications such as handwriting recognition, image classification, text narration etc.
- Parallelization can be implemented



Disadvantages

- Require long training time
- Requires number of parameters whose best value is unknown
- Difficulty to interpret the meaning of weights and hidden network

4.1.1 Neural Network Architectures: Feedforward, Convolution, Recurrent

a) Feedforward Neural Networks (FNN)

- This simple neural network variant passes data in a single direction through various processing nodes until the data reaches the output node.
- Feed-forward neural networks are designed to process large volumes of 'noisy' data and create 'clean' outputs.
- This type of neural network is also known as the multi-layer perceptrons (MLPs) model.
- A feed-forward neural network architecture includes the input layer, one or more hidden layers, and the output layer.
- Despite their alternate name, these models leverage sigmoid neurons rather than perceptrons, thus allowing them to address nonlinear, real-world problems.
- Feed-forward neural networks are the foundation for facial recognition, natural language processing, computer vision, and other neural network models. It is mainly used for pattern recognition tasks like image and speech classification.
- It consists of layers of neurons organized sequentially:
  - **Input Layer**: Takes in features from the data (e.g., pixel values in an image).

    - **Hidden Layer(s)**: Intermediate layers that process the input using weighted connections and activation functions.
    - **Output Layer**: Produces the final prediction or classification result.
- **Data Flow**:
    - Data moves **forward only**, from input → hidden → output.
    - There are **no cycles or loops**.
    - Each neuron in a layer is connected to all neurons in the next layer.

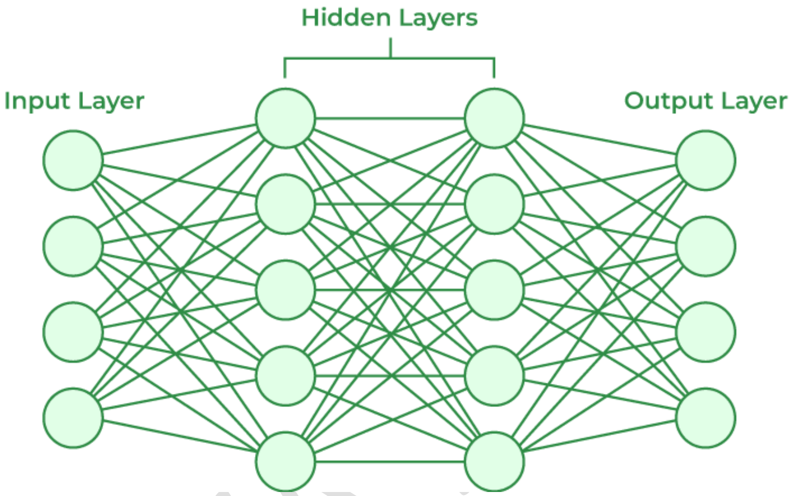


Figure: Feedforward Neural Network

b) Convolutional Neural Network (CNN)

- A **Convolutional Neural Network** is a type of neural network that uses **convolution operations** to extract spatial features from data.
- It is an advanced version of Artificial Neural Networks (ANNs)_primarily designed to extract features from grid-like matrix datasets.
- This is particularly useful for visual datasets such as images or videos, where data patterns play a crucial role.
- CNNs are widely used in computer vision applications due to their effectiveness in processing visual data.
- CNNs consist of multiple layers like the input layer, Convolutional layer, pooling layer, and fully connected layers.

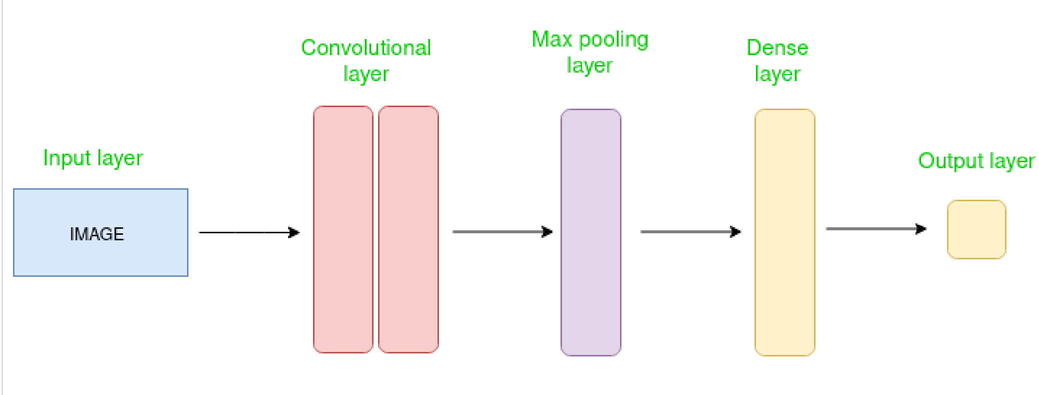


Figure: Convolutional Neural Network

Core Layers:

1. **Convolution Layer**:
   - Applies filters (kernels) to input data to detect patterns (like edges, textures).
   - Each filter slides across the input and generates a **feature map**.
2. **Pooling Layer**:
   - Reduces the spatial size of the feature maps (e.g., via max pooling).
   - Helps in reducing computation and controlling overfitting.

Advantage:

- Preserves spatial relationships.
- Requires fewer parameters than FNNs for image tasks.

CNN's Role in Spatial Data Processing:

Spatial data refers to data that has a **structure or layout in space** — meaning **location and position are important**. **Examples:**

- Images (2D grid of pixels)
- Videos (sequence of image frames)
- Satellite imagery
- Medical scans (e.g., MRI

A **Convolutional Neural Network (CNN)** is specially designed to **process and understand spatial data**, such as **images, videos, or 2D/3D sensor data**, where the arrangement of data points in space matters.

CNNs can **automatically learn spatial patterns** such as edges, corners, textures, and shapes by using **convolutional filters**. These filters slide across the image and detect patterns **locally**, preserving spatial relationships.

c) Recurrent Neural Network (RNN)

- **A Recurrent Neural Network (RNN)** is a type of neural network designed to handle **sequential data**, where the order and past context of data points affects the current output.. Examples include **time series**, **text**, **speech**, and **video**.
- Recurrent neural networks are a powerful and robust type of neural network, and belong to the most promising algorithms in use because they are the only type of neural network with an internal memory.
- RNNs can remember important things about the input they received, which allows them to be very precise in predicting what's coming next.
- This is why they're the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more.
- Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms.

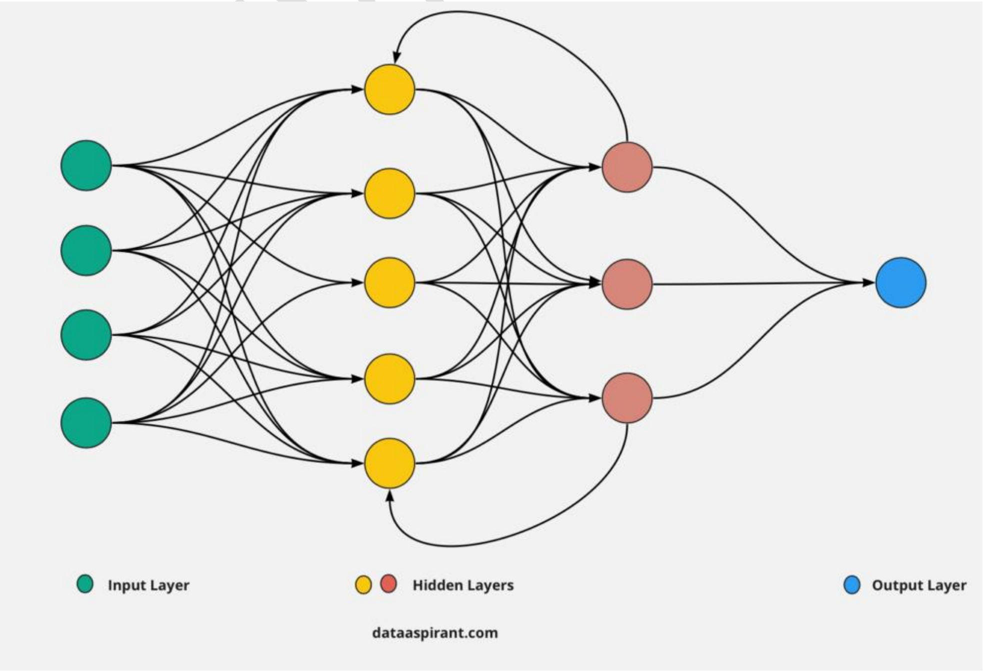


Figure: Recurrent Neural Network

Why Are RNNs Suitable?

Unlike traditional neural networks, **RNNs have memory**. They process input one element at a time while **retaining information from previous steps** through their **hidden state**. Unlike feedforward networks, RNNs have **loops** in their architecture. These loops allow the network to **retain memory of previous inputs**, making them ideal for tasks where **context and history** are important.

Their **recursive structure** gives RNNs the ability to:

- Capture dependencies between elements in a sequence
- Model context and history
- Handle variable-length input (e.g., sentences of different lengths)

4.1.2 Perceptron

A **perceptron** is the **basic unit of a neural network** — often considered the simplest type of artificial neuron.

A perceptron takes multiple **inputs**, multiplies them by **weights**, adds a **bias**, and passes the result through an **activation function** to produce an output.

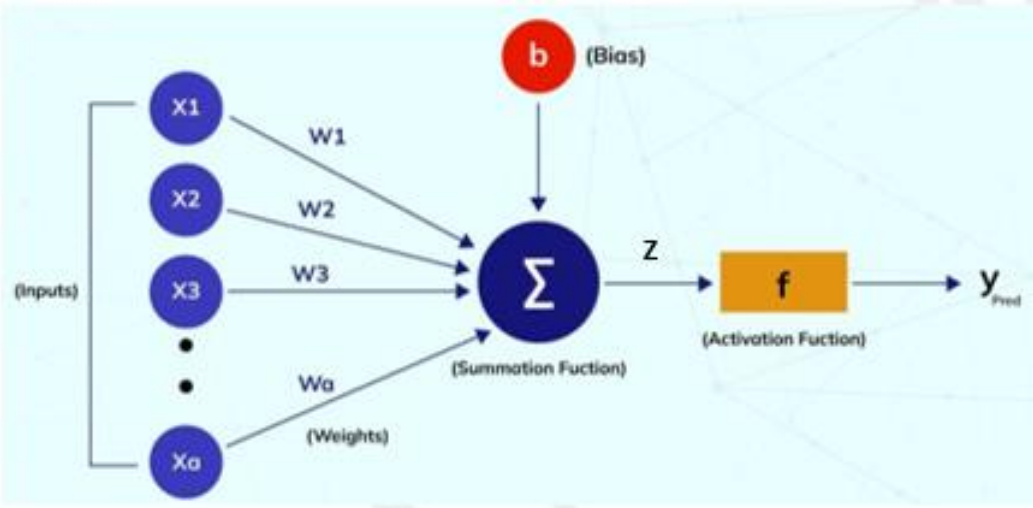


Figure: A Perceptron

Mathematical Representation:

- Given: inputs $\mathbf{x_1, x_2, ..., x_n}$ with weights $\mathbf{w_1, w_2, ..., w_n}$, and bias **b,**
- **For each neuron:**
  - $z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$
  - output $(y_{pred}) = f(z)$
- A common activation function (f) can be ***step function***:

$$\text{output} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

- Other activation functions can be like as ReLU, Softmax, Sigmoid and so on.

Basic components of Perceptron:

1. **Input Values**: A set of values or a dataset for predicting the output value. They are also described as a dataset's features and dataset.
2. **Weights:** The real value of each feature is known as weight. It tells the importance of that feature in predicting the final value.
3. **Bias:** The bias term helps the perceptron make adjustments independent of the input, improving its flexibility in learning.
4. **Summation Function:** The summation function binds the weights and inputs together. It is a function to find their sum.
5. **Activation Function:** The weighted sum is passed through the **Heaviside step function**, comparing it to a threshold to produce a binary output (0 or 1).
6. **Output:** The final output is determined by the activation function, often used for **binary classification** tasks.

What a weight does:

In neural networks, a weight determines how much influence an input feature has on the output:

- **Positive weight** → The higher the input, the more it contributes **positively** to the output.
- **Negative weight** → The higher the input, the more it **lowers** the output.
- **Zero weight** → The input has **no influence** on the output.

Limitations of the Perceptron

- A single-layer perceptron can only solve **linearly separable problems**.
- **Cannot solve** problems like the **XOR problem**, where data classes cannot be separated by a straight line.

Perceptron numerical example:

1. **Implement a perceptron to learn the AND logic gate**.

Input x1	Input x2	Desired Output
0	0	0
0	1	0
1	0	0
1	1	1

Let:

- Initial weights: $w_1 = 0.5$, $w_2 = 0.5$
- Bias: $b = -0.7$
- Activation function: **Step function**

$$\text{output} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Now, perceptron calculation table for AND gate is:

Input x_1	Input x_2	Weighted Sum $z = w_1x_1 + w_2x_2 + b$	Activation Output	Desired Output	Correct?
0	0	$0.5 \times 0 + 0.5 \times 0 - 0.7 = -0.7$	0	0	✓
0	1	$0.5 \times 0 + 0.5 \times 1 - 0.7 = -0.2$	0	0	✓
1	0	$0.5 \times 1 + 0.5 \times 0 - 0.7 = -0.2$	0	0	✓
1	1	$0.5 \times 1 + 0.5 \times 1 - 0.7 = 0.3$	1	1	✓

This table clearly shows how the perceptron correctly classifies all inputs of the AND gate.

2. **Build a perceptron that classifies an item as either a** pen (1) **or a** pencil (0)**.**

Dataset (Example):

Object	Length (cm) x1	Weight (g) x2	Class
A	14	9	0 (Pencil)
B	15	10	0 (Pencil)
C	13	20	1 (Pen)
D	12	21	1 (Pen)

Let’s normalize the features slightly and choose:

- w1 = 0.4 (weight for length)
- w2 = 0.6 (weight for weight)
- b = −15
- Activation: **Step function**

Perceptron Calculation Table:

Object	x1	x2	Weighted Sum z = 0.4 x1 + 0.6 x2 − 15	Output	Expected Class	Correct?
A	14	9	0.4 × 14 + 0.6 × 9 – 15 = −4	0	0 (Pencil)	✓
B	15	10	0.4 × 15 + 0.6 × 10 – 15 = −3	0	0 (Pencil)	✓
C	13	20	0.4 × 13 + 0.6 × 20 – 15 = 2.2	1	1 (Pen)	✓
D	12	21	0.4 × 12 + 0.6 × 21 – 15 = 2.4	1	1 (Pen)	✓

This simple perceptron correctly classifies all examples using linear separation.

Types of Perceptron:

• Single Layer perceptron

- A Single Layer Perceptron is the simplest type of neural network that consists of:
  - One input layer and
  - One output layer (no hidden layer)
  - It is used for binary classification and is based on a linear decision boundary.
- Limitation: Linear Separability
  - A single perceptron can **only solve linearly separable problems** (e.g., AND, OR).
  - It **cannot solve** problems like **XOR**, which require non-linear decision boundaries.

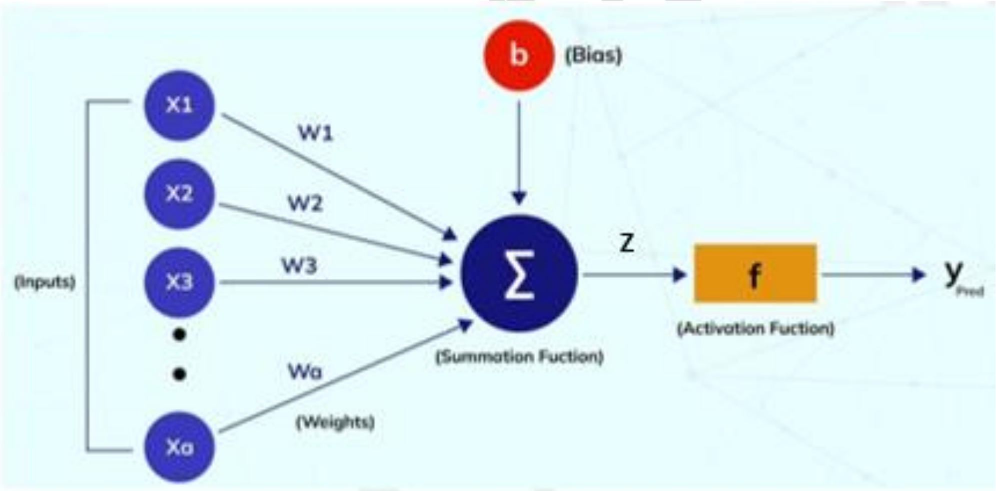


Figure: Single Layer Perceptron

• Multilayer perceptron

- To solve complex (nonlinear) problems, we stack perceptrons into **multiple layers**:
- A Multilayer Perceptron consists of:
  - Input layer
  - One or more hidden layers
  - Output layer
- It can model non-linear decision boundaries and solve complex problems

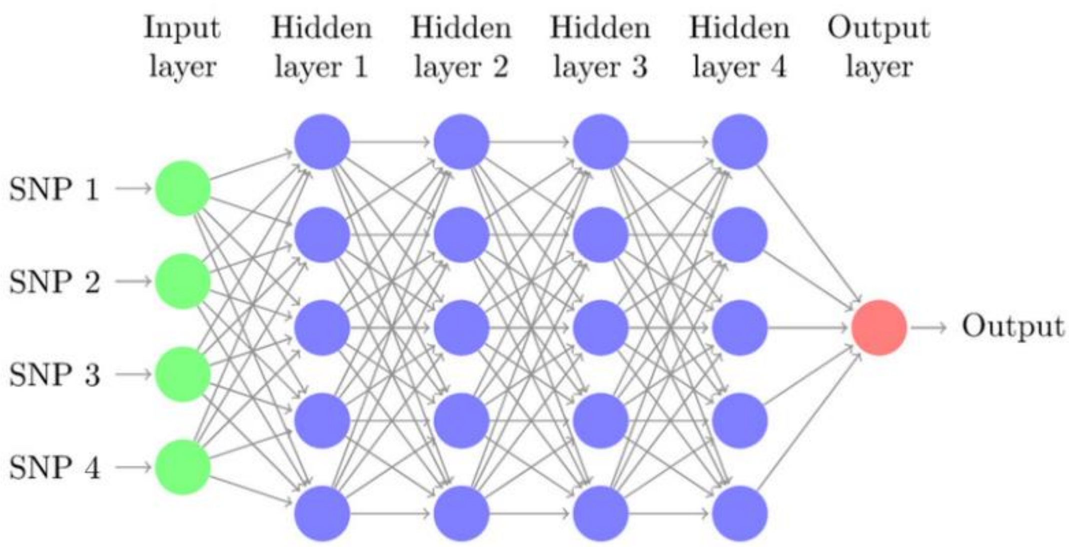
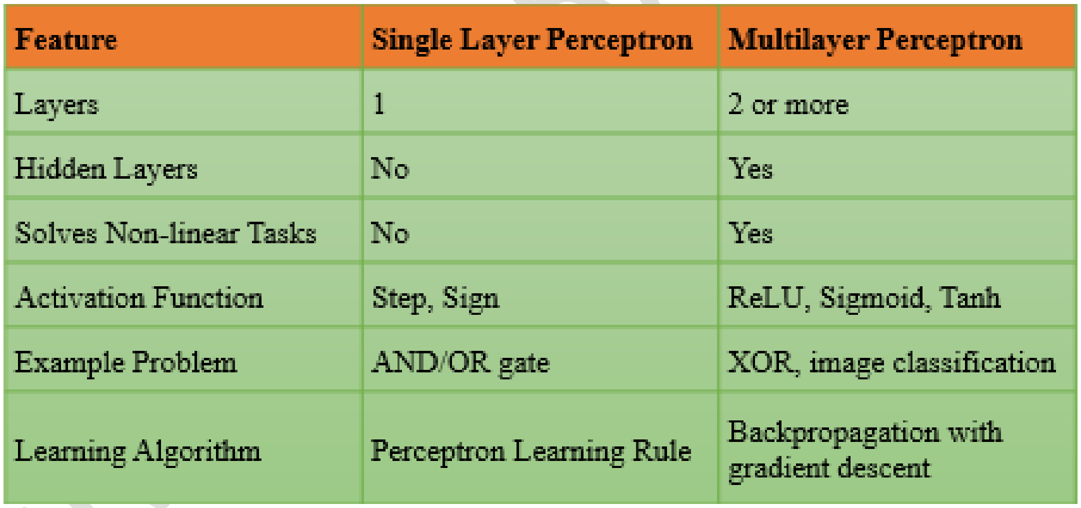


Figure: Multi Layer Perceptron

Comparison:



Feature	Single Layer Perceptron	Multilayer Perceptron
Layers	1	2 or more
Hidden Layers	No	Yes
Solves Non-linear Tasks	No	Yes
Activation Function	Step, Sign	ReLU, Sigmoid, Tanh
Example Problem	AND/OR gate	XOR, image classification
Learning Algorithm	Perceptron Learning Rule	Backpropagation with gradient descent

- ## Backpropagation

  - Backpropagation (short for backward propagation of errors) is the core algorithm used to train artificial neural networks.
  - It efficiently computes how much each weight in the network should be adjusted to reduce the error in predictions.

- Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.
- It works iteratively to adjust weights and bias to minimize the cost function.
- The Back Propagation algorithm involves two main steps: the Forward Pass and the Backward Pass.

1. Forward Pass Work

- In forward pass, the input data is fed into the input layer. These inputs combined with their respective weights are passed to hidden layers.
- Each hidden layer computes the weighted sum of the inputs then applies an activation function like ReLU (Rectified Linear Unit) to obtain the output (`o`).
- The output is passed to the next layer where an activation function such as *softmax* converts the weighted outputs into probabilities for classification.

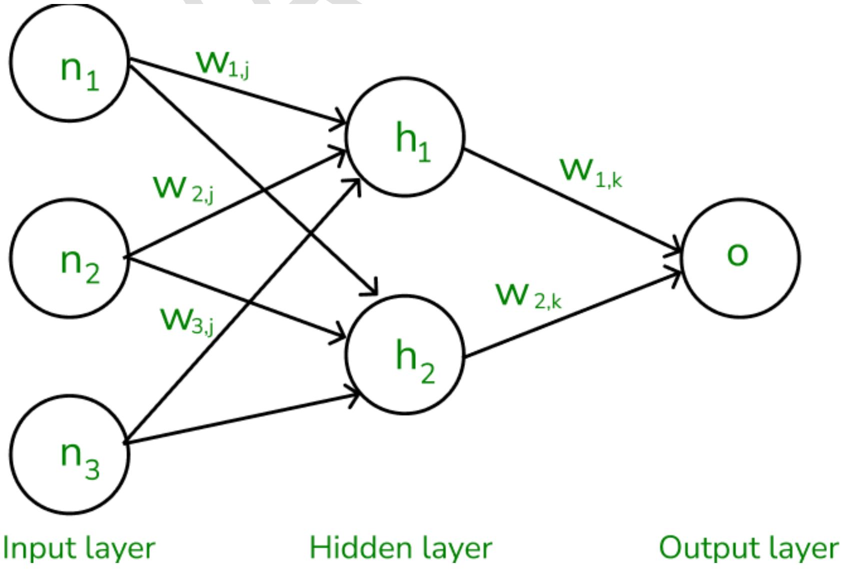


Figure: The forward pass using weights and biases

2. **Backward Pass**

- In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases.
- One common method for error calculation is the Mean Squared Error (MSE) given by: MSE=(Predicted Output−Actual Output)^2
- Once the error is calculated, the network adjusts weights using gradients which are computed with the chain rule.
- The backward pass continues layer by layer ensuring that the network learns and improves its performance.

4.2 Training Neural Network

Neural Network Training is the process of updating the weights and biases of a neural network model through the backpropagation algorithm by passing data through the network to find the appropriate parameters for making accurate predictions.

Training means helping the network **learn from data** by minimizing error between **actual output** and **expected output**. This is done by **adjusting weights** using **gradient descent** during backpropagation.

4.2.1 Forward and Backward propagation:

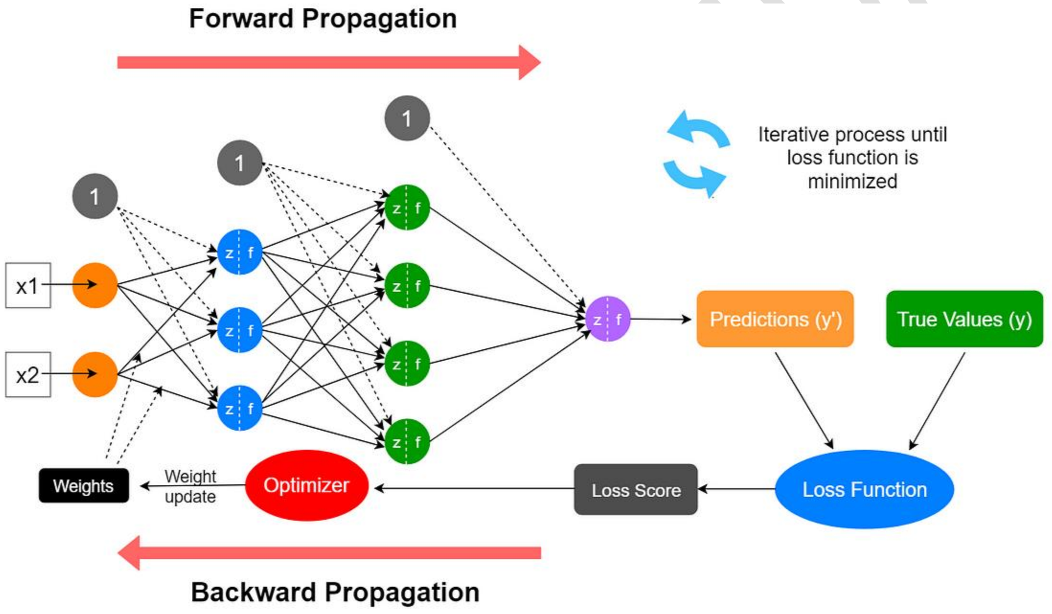


Figure: Forward and Backward Propagation

Forward propagation

- **Forward propagation** is the process of **passing input data through the network** to get the **predicted output** (also called **ŷ**).
- It happens layer by layer—from **input** → **hidden layers** → **output layer**.
- Each neuron performs:
- **Weighted Sum + Bias → Apply Activation Function → Output**

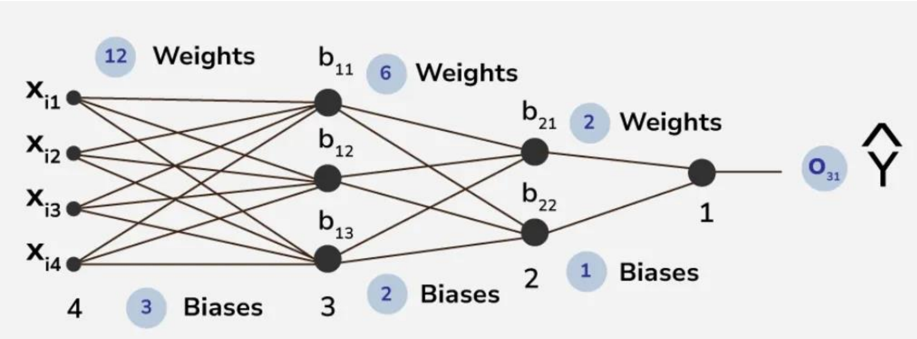


Figure: Forward Propagation

Steps:

1. Input Layer

- The input data is fed into the network through the input layer.
- Each feature in the input dataset represents a neuron in this layer.
- The input is usually normalized or standardized to improve model performance.

2. Hidden Layers

- The input moves through one or more hidden layers where transformations occur.
- Each neuron in hidden layer computes a weighted sum of inputs and applies activation function to introduce non-linearity.
- Each neuron receives inputs, computes: Z=WX+b, where:
    - W is the weight matrix
    - X is the input vector
    - b is the bias term
- The activation function such as ReLU or sigmoid is applied.

3. Output Layer

- The last layer in the network generates the final prediction.
- The activation function of this layer depends on the type of problem:
    - **Softmax** (for multi-class classification)
    - **Sigmoid** (for binary classification)
    - **Linear** (for regression tasks)

4. Prediction

- The network produces an output based on current weights and biases.
- The loss function evaluates the error by comparing predicted output with actual values.

Numerical Example:

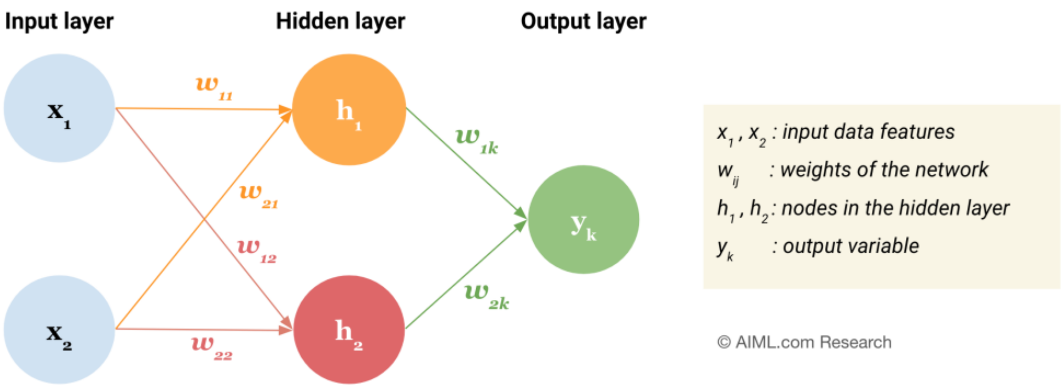
1. **Given a neural network structure as below with sigmoid activation function and parameters as below:**

   x1 = 1.0, x2 = 2.0

   w11 = 0.1, w12 = 0.3, b1 = 0.1

   w21 = 0.2, w22 = 0.4, b2 = 0.2

   w31 = 0.5, w32 = 0.6, b3 = 0.3



Solution:

Step	Calculation
Input values	x1=1.0, x2=2.0

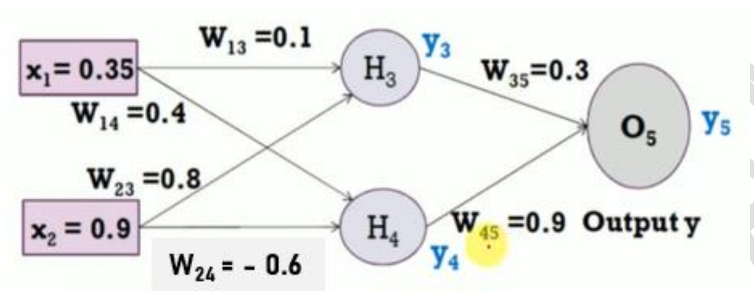
Hidden neuron 1	z = w11 * x1 + w21 * x2 + b1
	= 0.1×1.0+0.2×2.0+0.1
	= 0.6
	h1= σ(0.6) = $\frac{1}{1+e^{-0.6}}$ = 0.6457

Hidden neuron 2	z2 = w12 * x1 + w22 * x2 + b2
	= 0.3×1.0+0.4×2.0+0.2
	= 1.3
	h2 = σ(1.3) = $\frac{1}{1+e^{-1.3}}$ = 0.7858

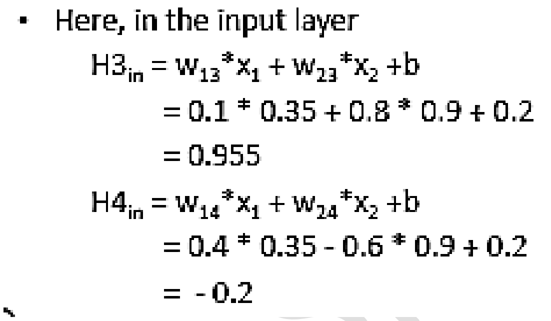
Output neuron	z3 = w1k * h1 + w2k * h2 + b3 = 0.5×0.6457+0.6×0.7858+0.3 = 1.0943 $\hat{y}$ = σ(1.0943) = $\frac{1}{1+e^{-1.0943}}$ = 0.7492

Hence: The final predicted output $\hat{y}$ after forward propagation is **approximately 0.749**.

2. For the given ANN, perform a forward pass and compute Y_{in} for each neuron. Assume that the bias is 0.2 and the activation function for Y_{out} is : $y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$ **If the actual output is 1, will it make back propagation?**



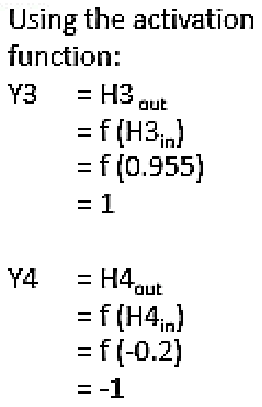
Solution:



- Here, in the input layer

$H3_{in} = w_{13} * x_1 + w_{23} * x_2 + b$
$= 0.1 * 0.35 + 0.8 * 0.9 + 0.2$
$= 0.955$

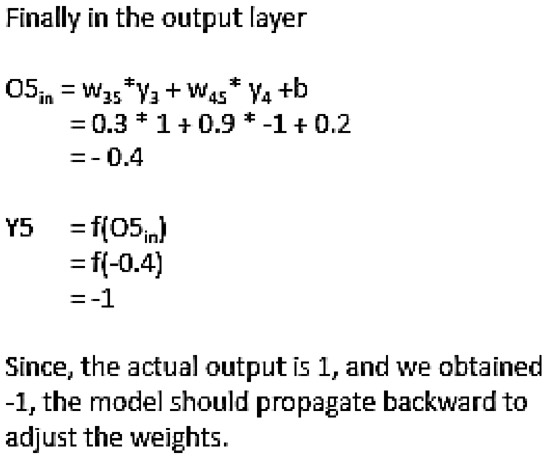
$H4_{in} = w_{14} * x_1 + w_{24} * x_2 + b$
$= 0.4 * 0.35 - 0.6 * 0.9 + 0.2$
$= -0.2$



Using the activation function:

$Y3 = H3_{out}$
$= f(H3_{in})$
$= f(0.955)$
$= 1$

$Y4 = H4_{out}$
$= f(H4_{in})$
$= f(-0.2)$
$= -1$



Finally in the output layer

$O5_{in} = w_{35} * y_3 + w_{45} * y_4 + b$
$= 0.3 * 1 + 0.9 * -1 + 0.2$
$= - 0.4$

$Y5 = f(O5_{in})$
$= f(-0.4)$
$= -1$

Since, the actual output is 1, and we obtained -1, the model should propagate backward to adjust the weights.

Backpropagation and Gradient Descent

Backward propagation is the process of **updating weights** in a neural network to reduce the **error** between the predicted output ($\hat{y}$) and the actual output (y).
It uses **calculus** (chain rule) and **gradient descent** to adjust the weights in the direction that minimizes the error.

Neural networks are trained using Gradient Descent (or its variants) in combination with backpropagation. Backpropagation computes the gradients of the loss function with respect to each parameter (weights and biases) in the network by applying the chain rule.

After **forward propagation**:

- You get a predicted output $\hat{y}$
- You compare it to the actual output y
- You calculate the **error**
- You go **backward** through the network to adjust the **weights** and **biases** to reduce that error.

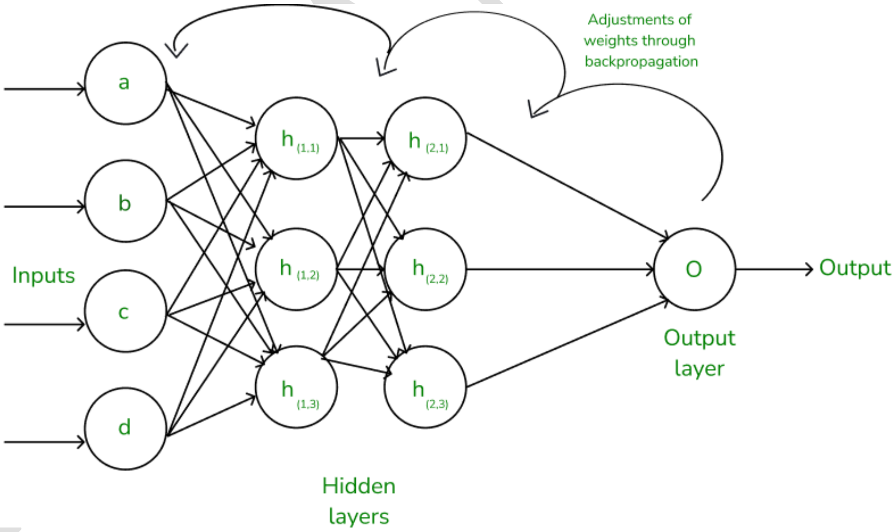


Figure: Backpropagation

Purpose of Backpropagation:

- To minimize the error (loss) between predicted output and true output
- To compute how much each weight and bias contributed to the error
- To adjust weights and biases in the direction that reduces the loss

Steps of Backpropagation Algorithm

Step 1: Perform Forward Propagation

- Pass input data through the network layer by layer to generate output.
  - Compute intermediate weighted sums using:

    $$a_j = \sum(W_{i,j} * X_i + b_j)$$

    Where,

    - a_j is the weighted sum of all the inputs and weights at each node
    - $W_{i,j}$ represents the weights between the ith input and the jth neuron
    - X_i represents the value of the ith input
    - b_j is the bias for jth neuron
  - Apply activation function (like sigmoid, ReLU, etc) to a_j and obtain intermediate outputs and final output.

    $$O_j = \text{activation function}(a_j)$$

Step 2: Calculate Loss using suitable Loss functions

- Calculate the loss using a suitable loss function like MSE or Cross-Entropy

  Error = Actual Output – Predicted Output

Step 3: Perform Backward Propagation

- We move backwards from the output layer to the input layer to compute the gradients and update weights. This is based on the chain rule of derivatives.

a) Gradient Calculation

- The change in each weight is calculated as:

  $$\Delta W_{i,j} = \eta \times \delta_j \times O_j$$

  Where:

  - η (eta) is the learning rate.
  - δ_j is the error term for each unit calculated as:
    - $\delta_{output} = Y(1-Y)(Y_{target}-Y)$ for output layer
    - $\delta_j = Y_j\,(1-Y_j) * W_{j,k} * \delta_{output}$ for hidden layer (applies chain rule)

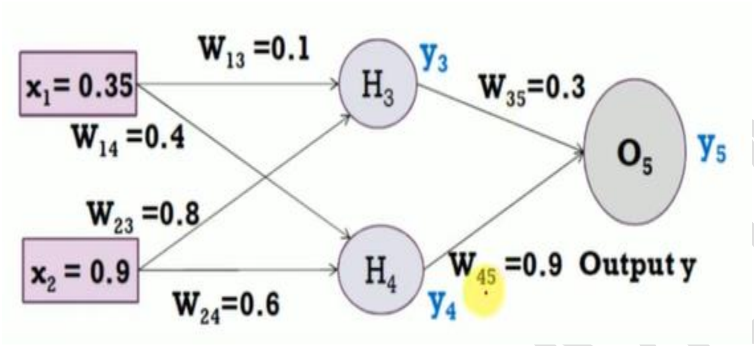
b) Weight Update (Gradient Descent)

- Compute the ΔW for each weights in a chain rule as: $\Delta W_{i,j} = \eta * \delta_j * X_i$
- Compute New weight as: New weight = old Weight + Δ Weight

Step 4: Repeat step 1, 2 and 3 iteratively for n epochs until actual output is obtained (i.e. global minima is reached)

Numerical:

1. **Assume that the neurons have a sigmoid activation function, perform a forward pass and a backward pass on the network. Assume that the actual output of y is 0.5 and learning rate (η) is 1. Perform another forward pass. [Reference video for solution: https://www.youtube.com/watch?v=tUoUdOdTkRw]**



Solution:

Step 1: Forward Propagation

Here in the Hidden layer:

$H3_{in}$ $= w_{13} * x_1 + w_{23} * x_2 = 0.1 * 0.35 + 0.8 * 0.9$ = **0.755**

$H4_{in}$ $= w_{14} * x_1 + w_{24} * x_2 = 0.4 * 0.35 + 0.6 * 0.9$ = **0.68**

Using the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$

Y_3 $= f\,(H3_{in}) = \frac{1}{1+e^{-0.755}}$ = **0.6802**

Y_4 $= f\,(H4_{in}) = \frac{1}{1+e^{-0.68}}$ = **0.6637**

Finally in the output layer:

$O5_{in}$ $= w_{35} * y_3 + w_{45} * y_4 = 0.3 * 0.6802 + 0.9 * 0.6637$ = **0.80139**

Using sigmoid function:

Y $= f\,(O5_{in}) = \frac{1}{1+e^{-0.80139}}$ = **0.69**

Step 2: Compute Error

Error = Target – Y = 0.5 – 0.69 = **- 0.19**

Since there is an error of -0.19, we need to perform back propagation for weight adjustment.

Step 3: Backward Propagation for Computation of δ_3, δ_4 and δ_5 and adjustment for weights:

For output layer:

δ_5 $= Y(1-Y)\ (Y_{target} - Y)$

$= 0.69*(1-0.69)*(0.5-0.69) =$ **-0.0406**

For hidden layer:

δ_3 $= Y_3\ (1-Y_3)*w_{35}*\ \delta_5$

$= 0.68\ (1-0.68) * 0.3 * -0.0406 =$ **-0.00265**

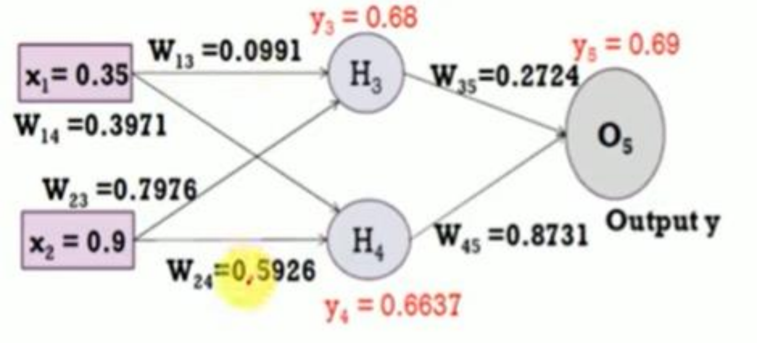
δ_4 $= Y_4\ (1-Y_4)*w_{45}*\ \delta_5$

$= 0.6637\ (1-0.6637) * 0.9 * -0.0406 =$ **-0.0082**

New weights are:

Δ weights	**New weights**
$\Delta w_{45} = \eta\ \delta_5\ Y_4$ $= 1 * -0.0406* 0.6637 = -0.0269$	$W_{45} = \text{old } W_{45} + \Delta w_{45}$ $= 0.9 – 0.0269 = 0.8731$
$\Delta w_{14} = \eta\ \delta_4\ X_1$ $= 1 * -0.0082* 0.35 = -0.0287$	$W_{14} = \text{old } W_{14} + \Delta w_{14}$ $= 0.4 – 0.0287 = 0.3971$
$\Delta w_{35} = \eta\ \delta_5\ Y_3$ $= 1 * -0.0406* 0.6802 = -0.0276$	$W_{35} = \text{old } W_{35} + \Delta w_{35}$ $= 0.3 – 0.0276 = 0.2724$
$\Delta w_{23} = \eta\ \delta_3\ X_2$ $= 1 * -0.00265* 0.9 = -0.002385$	$W_{23} = \text{old } W_{23} + \Delta w_{23}$ $= 0.8 – 0.002385 = 0.7976$
$\Delta w_{24} = \eta\ \delta_4\ X_2$ $= 1 * -0.0082* 0.9 = -0.00738$	$W_{24} = \text{old } W_{24} + \Delta w_{24}$ $= 0.6 – 0.00738 = 0.5926$
$\Delta w_{13} = \eta\ \delta_3\ X_1$ $= 1 * -0.00265* 0.35 = -0.0009275$	$W_{13} = \text{old } W_{13} + \Delta w_{13}$ $= 0.1 – 0.0009275 = 0.0991$

Updated network values are:



Step 4: Second Forward pass

Here in the Hidden layer:

$H3_{in}$ $= w_{13} * x_1 + w_{23} * x_2$

$= 0.0991 * 0.35 + 0.7976 * 0.9$

$= \mathbf{0.7525}$

$H4_{in}$ $= w_{14} * x_1 + w_{24} * x_2$

$= 0.3971 * 0.35 + 0.5926 * 0.9$

$= \mathbf{0.6723}$

Using the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$

$\mathbf{Y_3} = f\,(H3_{in}) = \frac{1}{1+\,e^{-0.7525}} = \mathbf{0.6797}$

$\mathbf{Y_4} = f\,(H4_{in}) = \frac{1}{1+e^{-0.6723}} = \mathbf{0.6620}$

Finally in the output layer:

$O5_{in}$ $= w_{35} * y_3 + w_{45} * y_4$

$= 0.2724 * 0.6797 + 0.8731 * 0.662$

$= \mathbf{0.7631}$

Using sigmoid function:

$\mathbf{Y} = f\,(O5_{in}) = \frac{1}{1+e^{-0.7631}} = \mathbf{0.6820}$

Note: Here we stop as per the question.

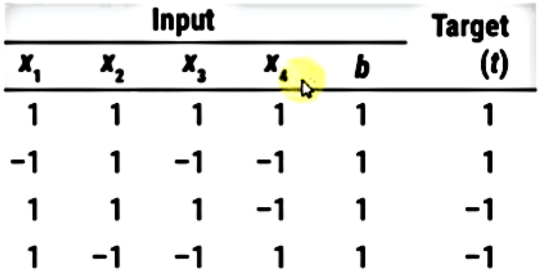
Else, we iterate with forward and backward propagation until the global minima (i.e until we get the exact output).

Assignment:

2. **Find the weights required to perform the following classification using perceptron Network. The target classes are 1 and -1. Assume learning rate (α)=1, initial weights as 0 and bias=0. The activation function is as:**

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

Also, construct the Neural Network with weights and bias. Also classify the entity with features vector [1 1 -1 1] , i.e. which class do they belong to.



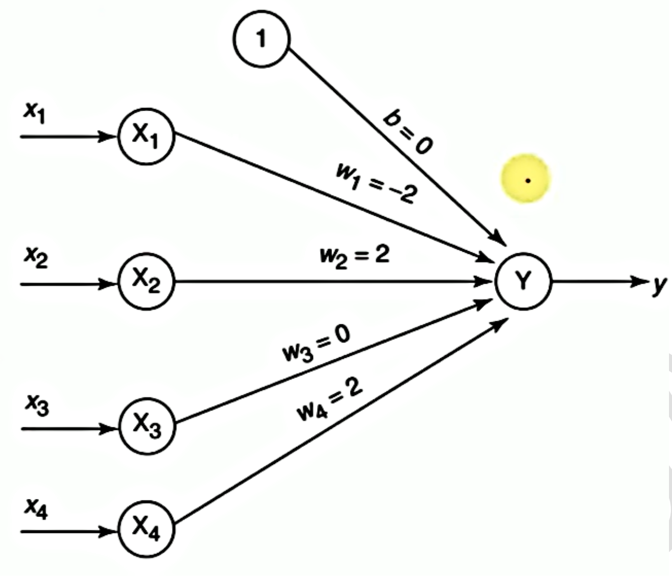
Input					Target
x_1	x_2	x_3	x_4	b	(t)
1	1	1	1	1	1
−1	1	−1	−1	1	1
1	1	1	−1	1	−1
1	−1	−1	1	1	−1

Reference Video: https://www.youtube.com/watch?v=CvbYumf_wSI

Solution:

Epoch	Inputs				Traget (t)	Net Input (Yin)	Output (Y)	Weight changes					Weights					Remarks
	x1	x2	x3	x4				Δw1	Δw2	Δw3	Δw4	Δb	w1	w2	w3	w4	b	
													0	0	0	0	0	Initialization
Epoch 1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	
	-1	1	-1	-1	1	-1	-1	-1	1	-1	-1	1	0	2	0	0	2	
	1	1	1	-1	-1	4	1	-1	-1	-1	1	-1	-1	1	-1	1	1	
	1	-1	-1	1	-1	1	1	-1	1	1	-1	-1	-2	2	0	0	0	
Epoch 2	1	1	1	1	1	0	0	1	1	1	1	1	-1	3	1	1	1	
	-1	1	-1	-1	1	3	1	0	0	0	0	0	-1	3	1	1	1	No weight updates because y=t
	1	1	1	-1	-1	3	1	-1	-1	-1	1	-1	-2	2	0	2	0	
	1	-1	-1	1	-1	-2	-1	0	0	0	0	0	-2	2	0	2	0	No weight updates because y=t
Epoch 3	1	1	1	1	1	2	1	0	0	0	0	0	-2	2	0	2	0	No weight updates because y=t
	-1	1	-1	-1	1	2	1	0	0	0	0	0	-2	2	0	2	0	No weight updates because y=t
	1	1	1	-1	-1	-2	-1	0	0	0	0	0	-2	2	0	2	0	No weight updates because y=t
	1	-1	-1	1	-1	-2	-1	0	0	0	0	0	-2	2	0	2	0	No weight updates because y=t

Hence, required Neural Network is:

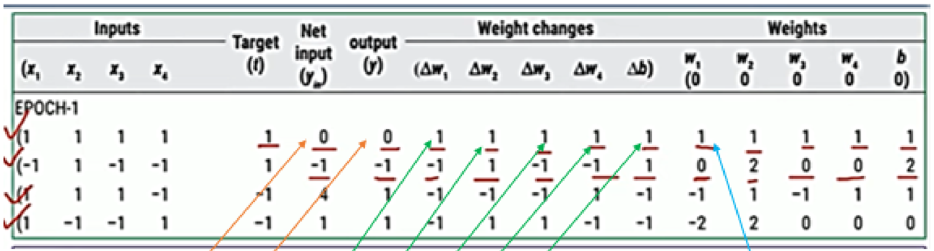


Working note for 1st epoch:

$$y_{in} = b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$$

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

$$\Delta w_1 = \alpha t x_1; \quad \Delta w_2 = \alpha t x_2; \quad \Delta w_3 = \alpha t x_3; \quad \Delta w_4 = \alpha t x_4; \quad \Delta b = \alpha t$$



Inputs				Target	Net input	output	Weight changes					Weights				
(x_1	x_2	x_3	x_4	(t)	(y_{in})	(y)	(Δw_1	Δw_2	Δw_3	Δw_4	Δb)	w_1 (0	w_2 0	w_3 0	w_4 0	b 0)
EPOCH-1																
(1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1
(-1	1	-1	-1	1	-1	-1	-1	1	-1	-1	1	0	2	0	0	2
(1	1	1	-1	-1	4	1	-1	-1	-1	1	-1	-1	1	-1	1	1
(1	-1	-1	1	-1	1	1	-1	1	1	-1	-1	-2	2	0	0	0

Working notes for Row 1 in Epoch 1:

Net input (yin) $= b+x_1w_1+x_2w_2+x_3w_3+x_4w_4$

$= 0+ 1*0+1*0+1*0+1*0$

$= 0$

Output (y) = 0 using the activation function

$\Delta w_1 = \alpha t x_1 = 1*1*1=1$

$\Delta w_2 = \alpha t x_2 = 1* \ 1*1 = 1$

$\Delta w_3 = \alpha t x_3 = 1$

$\Delta w_4 = \alpha t x_4 = 1$

$\Delta b = \alpha t = 1$

New weights:

W_1= old $w_1 + \Delta w_1 = 0 + 1 = 1$

W_2= old $w_2 + \Delta w_2 = 0 + 1 = 1$

W_3= old $w_3 + \Delta w_3 = 0 + 1 = 1$

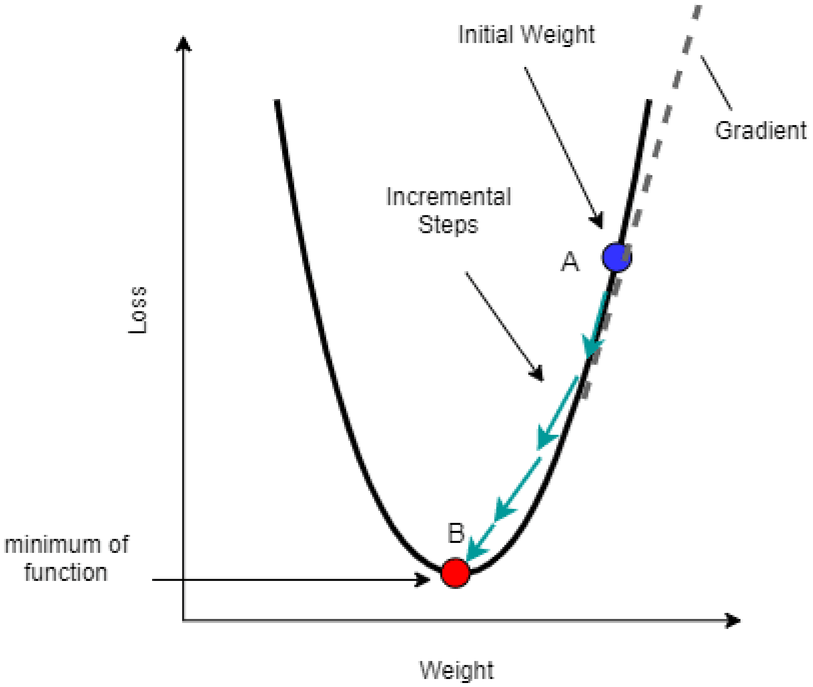
W_4= old $w_4 + \Delta w_4 = 0 + 1 = 1$

B = old b + Δb = 0+1=1

Gradient Descent:

Gradient Descent is a fundamental algorithm in machine learning and optimization. It is used for tasks like training neural networks, fitting regression lines, and minimizing cost functions in models

Gradient Descent is an algorithm used to find the best solution to a problem by making small adjustments in the right direction. It's like trying to find the lowest point in a hilly area by walking down the slope, step by step, until you reach the bottom.



We want to **reduce the error (loss)** of the model by adjusting the weights. The lowest point in the curve (point **B**) represents the **best weight**—where the error is minimum.

Point A – Starting Point (Initial Weight):

- The training process starts here.
- This is a **random guess** for the weight.
- The model is making **high errors** here (high up on the loss curve).

Gradient:

- The slope of the curve at point A.
- It tells us **which direction to go** and **how steep** the hill is.
- We want to move **downhill**, opposite to the gradient, to reduce error.

Incremental Steps:

- The arrows show small steps taken in the direction **that lowers the loss**.
- These steps are calculated using the **gradient**.
- Each step slightly improves the weight and reduces the loss.

Point B – Final Point (Minimum):

- This is the **lowest point on the curve**.
- Here, the **loss is minimum**, and we have found the **best weight** for the model.

Learning Rate:

The learning rate, usually denoted by η (eta), is a key hyperparameter in training machine learning models — especially in algorithms like gradient descent.

It controls how much the model's parameters (like weights) are adjusted during training based on the error.

Learning rate is an important hyper-parameter in gradient descent that controls how big or small the steps should be when going downwards in gradient for updating models parameters. It is essential to determines how quickly or slowly the algorithm converges toward minimum of loss (or cost) function.

Choosing the Right Learning Rate:

- **If Learning rate is too small:** The algorithm will take tiny steps during iteration and converge very slowly. This can significantly increases training time and computational cost especially for large datasets.
- **Learning rate is too big:** The algorithm may take huge steps leading overshooting the minimum of cost function without settling. It fail to converge causing the algorithm to oscillate. This process is termed as exploding gradient problem.

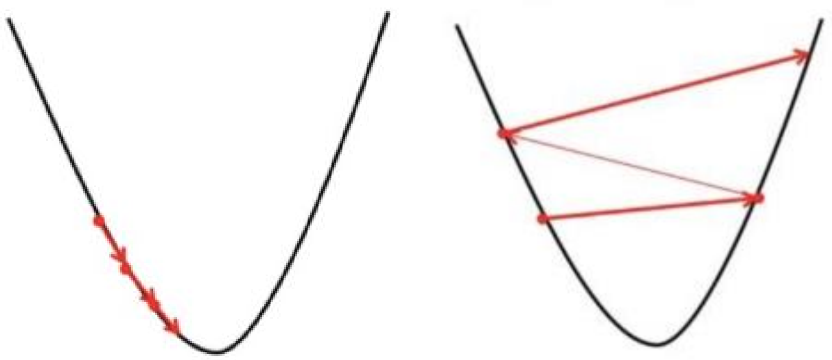


Figure: (a) Learning rate with small step (b) Learning rate with big steps

- Good learning rate:
  - Fast convergence
  - Smooth error reduction
- Bad learning rate:
  - Causes unstable training
  - May never reach minimum

4.2.2 Loss Functions:

Loss and Loss functions

Loss:

- In machine learning, loss is a number that tells us how wrong the model's prediction is compared to the actual (true) value.
- They provide a measure of how well the model's predictions align with the actual data.

Loss function:

- A loss function is a mathematical formula used to calculate the error between:
- The predicted value ($\hat{y}$)
- The actual value (y)
- It helps guide the training process by telling the model how to adjust its weights.

Role of loss function

A **loss function** plays a **central role** in training a machine learning model. It tells the model **how well or poorly it's performing** and guides it on how to improve.

- Measures Error: The loss function quantifies the difference between: model's prediction ($\hat{y}$) and the true/target value (y).
- **Provides Feedback to Learn:** The model uses the value of the loss to decide whether it's doing a good job or how to adjust weights and biases to reduce error.
- **Guides Gradient Descent:** The loss function provides th**e** direction and magnitude of the slope (gradient) to reach global minima.

Categories of Loss Functions:

Loss functions are classified into two classes based on the type of learning task:

- Regression Models: predict continuous values.
- Classification Models: predict the output from a set of finite categorical values.

Regression Loss Functions in Machine Learning

Regression tasks involve predicting continuous values, such as house prices or temperatures. Here are some commonly used loss functions for regression:

a) Mean Squared Error (MSE)

- It is the average of the squared differences between the predicted values and actual values.
- the mean of the squared differences is taken rather than just the sum.
- This ensures that the loss function is independent of the number of data points in the training set, making the metric more reliable across datasets of varying sizes.
- However, MSE is sensitive to outliers

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

Where:

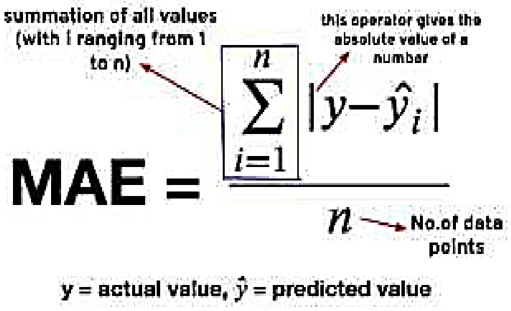
y_i is the actual value for the ith data point.

$\hat{y}$ is the predicted value for the ith data point.

n is the total number of data points.

b) Mean Absolute Error (MAE) / La Loss

- The Mean Absolute Error (MAE) is a commonly used loss function in machine learning that calculates the mean of the absolute values of the residuals (difference between actual output and predicted output) for all datapoints in the dataset.
- It's defined as the average of the absolute difference between actual and predicted values.


summation of all values (with i ranging from 1 to n)
this operator gives the absolute value of a number
$\mathrm{MAE} = \frac{\sum_{i=1}^{n}|y-\hat{y}_i|}{n}$
No.of data points
y = actual value, $\hat{y}$ = predicted value

Classification Loss Functions in Machine Learning

Cross-Entropy Loss, also known as Negative Log Likelihood, is a commonly used loss function in machine learning for classification tasks. This loss function measures how well the predicted probabilities match the actual labels.

Cross-entropy loss punishes the model when it's confident and wrong, and rewards it when it's confident and right**.**

a) Binary Cross-entropy Loss:

- It is used for binary classification.
- The model must choose **one class** from **two possible classes**. Example: Classifying an image as **animal** or **bird**.
- Each output is a **single probability** between 0 and 1.
  - ○ If the true label is 1 and the model predicts close to 1 → low loss.
  - ○ If the true label is 0 and the model predicts close to 1 → high loss.
- **Mathematically:**

$$\mathcal{L}(y, \hat{y}) = -[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})]$$

Where:

- y = actual label (0 or 1)
- $\hat{y}$ = predicted probability of class 1 (between 0 and 1)

- Example:

Suppose:

- Actual: $y = 1$
- Prediction: $\hat{y} = 0.9$

$$\mathcal{L} = -[1 \cdot \log(0.9) + 0 \cdot \log(0.1)] = -\log(0.9) \approx 0.105$$

b) Categorical Cross-entropy Loss:

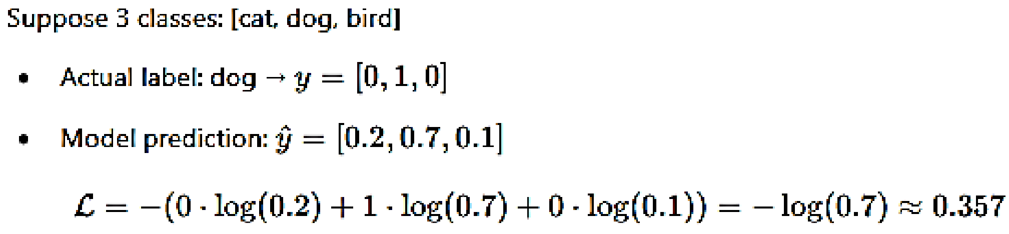
- It is used for multiclass classification (more than 2 classes)
- Each output is a **vector of class probabilities** (e.g., from Softmax)
- The true label is **one-hot encoded**.
- Mathematically:

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)$$

Where:

- C= number of classes
- y_i = 1 for the correct class, 0 otherwise (one-hot encoding)
- $\hat{y}_i$ = predicted probability of class i

- Example:



Suppose 3 classes: [cat, dog, bird]

- Actual label: dog → $y = [0, 1, 0]$
- Model prediction: $\hat{y} = [0.2, 0.7, 0.1]$

$$\mathcal{L} = -(0 \cdot \log(0.2) + 1 \cdot \log(0.7) + 0 \cdot \log(0.1)) = -\log(0.7) \approx 0.357$$

4.2.3 Regularization techniques:

Overfitting and Underfitting

- **Overfitting**:

  Occurs when a model is **too complex** and learns not only the underlying pattern but also the noise in the training data. This leads to excellent performance on training data but poor generalization on new, unseen data (test data).
- **Underfitting**:

  Happens when a model is **too simple** to capture the underlying structure of the data. It results in poor performance on both training and test data because the model cannot learn enough from the data.

Learning Curve To Identify Overfit & Underfit

A learning curve is a graphical representation showing how an increase in learning comes from greater experience. It can also reveal if a model is learning well, overfitting, or underfitting.

Learning curves are graphical representations that illustrate how a model's performance changes with increasing experience, typically measured by the amount of training data it has processed. The x-axis of a learning curve typically represents the amount of training data or the number of training iterations, while the y-axis represents the performance metric, such as error or accuracy.

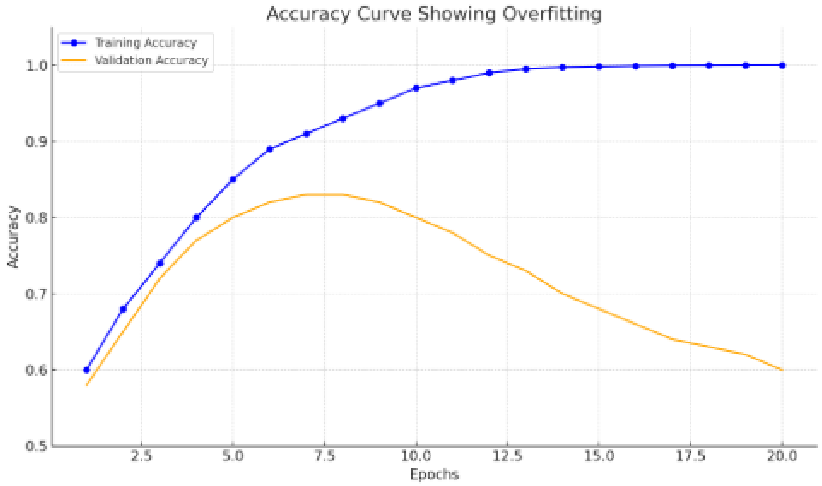


Figure: Accuracy curve illustrating overfitting

This graph shows classic overfitting:

- **Blue Line (Training Accuracy):** keeps increasing steadily, reaching near-perfect accuracy.
- **Orange Line (Validation Accuracy):** increases at first, but then **drops after a certain point**, indicating the model is memorizing the training data rather than learning to generalize.

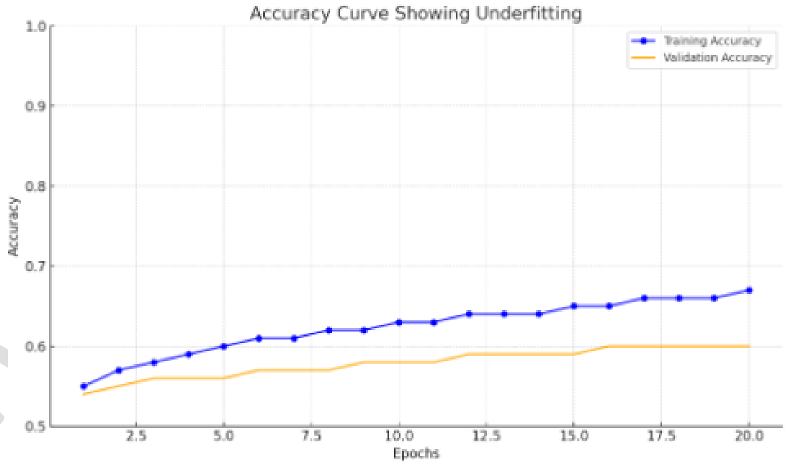


Figure: Accuracy curve illustrating underfitting

This graph shows a typical **underfitting** scenario:

- Both **Training (blue)** and **Validation (orange)** accuracies remain low.
- The model fails to learn patterns in the training data and doesn't generalize well either.

Regularization methods: L1, L2, Dropout, Batch normalization

Regularization is an important technique in machine learning that helps to improve model accuracy by preventing overfitting, which happens when a model learns the training data too well including noise and outliers, and perform poor on new data.

By adding a penalty for complexity, it helps simpler models to perform better on new data.

Common Regularization Methods:

a) Lasso Regression

- A regression model which uses the L1 Regularization technique is called LASSO (Least Absolute Shrinkage and Selection Operator) regression.
- Lasso regression is a type of **linear regression** that uses **L1 regularization** to prevent overfitting and perform feature selection.
- It adds the absolute value of magnitude of the coefficient as a penalty term to the loss function(L).
- This penalty can shrink some coefficients to zero which helps in selecting only the important features and ignoring the less important ones.
- Mathematically, it minimizes the following objective function:

$$\text{Loss} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |w_j|$$

Where:

- y_i is the actual value,
- $\hat{y}_i$ is the predicted value,
- w_j are the model coefficients (weights),
- $\lambda \geq 0$ is the regularization parameter controlling the strength of penalty.

Tuning λ:

The parameter λ controls the trade-off between fitting the data well and keeping the model simple.

- If λ=0, Lasso becomes ordinary least squares (no regularization).
- If λ is very large, all coefficients shrink toward zero.

When to use Lasso Regression?

- When you want **feature selection** along with regression.
- When your data has many features, and you suspect that only a few are important.
- When you want to reduce model complexity and avoid overfitting.

b) Ridge Regression

- A regression model that uses the L2 regularization technique is called Ridge regression.
- It adds the squared magnitude of the coefficient as a penalty term to the loss function (L).

- Ridge regression is a type of **linear regression** that uses **L2 regularization** to prevent overfitting by shrinking the coefficients towards zero but **not exactly zero**.
- Mathematically, Ridge regression minimizes:

$$\text{Loss} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} w_j^2$$

Where:

- y_i = actual value,
- $\hat{y}_i$ = predicted value,
- w_j = model coefficients,
- $\lambda \geq 0$ = regularization parameter controlling penalty strength.

- **Tuning** **λ:**

  Controls the strength of shrinkage.

  - λ = 0 → ordinary least squares (no regularization).

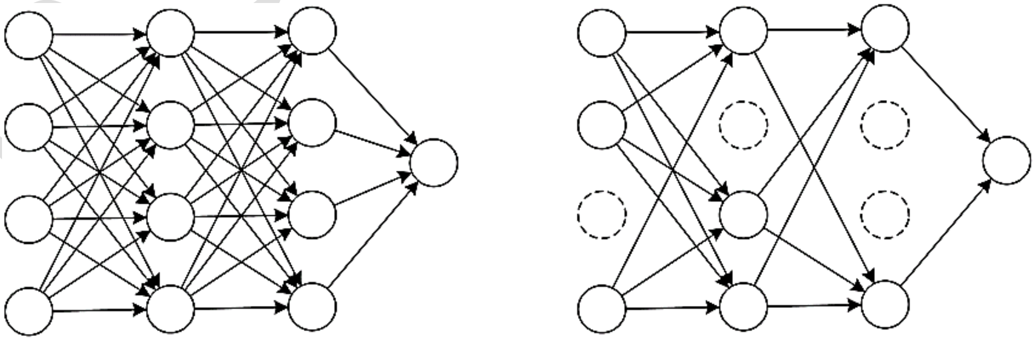
- Large λ → coefficients shrink toward zero, model gets simpler.

When to use Ridge Regression?

- When you want to reduce model complexity but **keep all features** in the model.
- When predictors are highly correlated (multicollinearity problem).
- When you want to prevent overfitting but don't want feature elimination.

c) Dropout

- Dropout is a popular regularization method used mainly in **neural networks** to reduce overfitting and improve generalization.
- When a fully-connected layer has a large number of neurons, co-adaptation is more likely to happen. Co-adaptation refers to when multiple neurons in a layer extract the same, or very similar, hidden features from the input data.
- This poses two different problems to our model:
- Wastage of machine's resources when computing the same output.
- If many neurons are extracting the same features, it adds more significance to those features for our model. This leads to overfitting if the duplicate extracted features are specific to only the training set.
- To solve this problem, we use dropout while training the NN to minimize co-adaptation. In dropout, we randomly shut down some fraction of a layer's neurons at each training step by zeroing out the neuron values.



(a) Standard Neural Network

(b) Network after Dropout

Why Dropout helps?

- **Prevents co-adaptation:** Neurons can't depend on the presence of specific other neurons; they must learn to work more independently and robustly.
- **Creates an ensemble effect:** Because different subsets of neurons are active in each training iteration, it's like training many different smaller networks and averaging their predictions, improving generalization.
- **Reduces overfitting:** By adding noise and randomness during training, Dropout reduces the chance the model memorizes the training data noise.

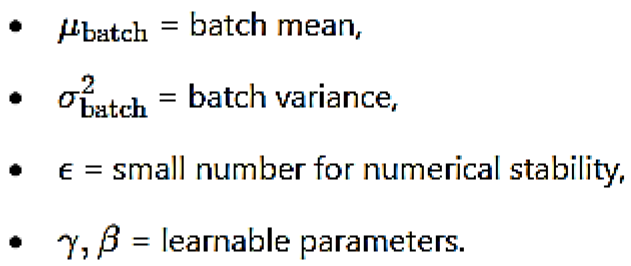
d) Batch Normalization

- Batch Normalization is a technique to **normalize the inputs of each layer** in a neural network during training.
- It helps stabilize and speed up training and also acts as a form of regularization.
- During training, for each mini-batch of data, BatchNorm computes the **mean** and **variance** of the inputs to a layer.
- It then **normalizes** these inputs by subtracting the batch mean and dividing by the batch standard deviation, so they have zero mean and unit variance.
- After normalization, BatchNorm applies **learnable scale (γ) and shift (β) parameters** to allow the network to restore any necessary distribution.
    - Mathematically, for an input xxx in the batch:

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

$$y = \gamma\hat{x} + \beta$$

Where:



- μ_{batch} = batch mean,
- σ^2_{batch} = batch variance,
- ϵ = small number for numerical stability,
- γ, β = learnable parameters.

Why Batch Normalization acts as Regularization?

- **Reduces internal covariate shift:** Normalizing inputs keeps the distributions stable, allowing higher learning rates and faster convergence.
- **Adds noise during training:** Because the mean and variance are calculated per mini-batch, they vary slightly from batch to batch, introducing noise which has a **regularizing effect** (like a mild dropout).
- **Improves generalization:** This noise helps prevent overfitting similarly to other regularization methods.
- Sometimes, models using BatchNorm **require less or no dropout**.

Numerical

Suppose we have a mini-batch of **4 input values** to a neuron (before activation): x = [4, 8, 6, 10]

Assume:

- Small constant for stability: $\epsilon = 10^{-5}$
- Learnable parameters (initialized): $\gamma = 1$, $\beta = 0$

Solution;

Step 1: Compute Batch Mean μ_{batch}

$$\mu_{\text{batch}} = \frac{4+8+6+10}{4} = \frac{28}{4} = 7$$

Step 2: Compute Batch Variance σ^2_{batch}

$$\sigma^2_{\text{batch}} = \frac{(4-7)^2 + (8-7)^2 + (6-7)^2 + (10-7)^2}{4}$$

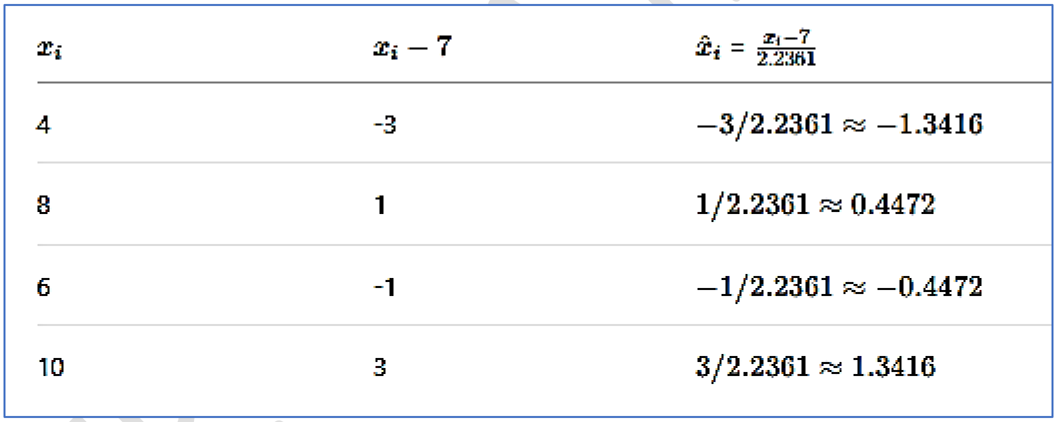
$$= \frac{(-3)^2 + 1^2 + (-1)^2 + 3^2}{4} = \frac{9+1+1+9}{4} = \frac{20}{4} = 5$$

Step 3: Normalize Each Value

For each x_i , calculate:

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}} = \frac{x_i - 7}{\sqrt{5 + 0.00001}} \approx \frac{x_i - 7}{2.2361}$$

Calculate normalized values:



x_i	$x_i - 7$	$\hat{x}_i = \frac{x_i - 7}{2.2361}$
4	-3	$-3/2.2361 \approx -1.3416$
8	1	$1/2.2361 \approx 0.4472$
6	-1	$-1/2.2361 \approx -0.4472$
10	3	$3/2.2361 \approx 1.3416$

Step 4: Scale and Shift

Apply $\gamma = 1$ and $\beta = 0$:

$$y_i = \gamma \hat{x}_i + \beta = \hat{x}_i$$

So final normalized values:

$$y = [-1.3416,\ 0.4472,\ -0.4472,\ 1.3416]$$

4.3 Advanced Neural Network Architecture

4.3.1 Convolution Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized type of neural network designed for processing **grid-like data**, such as images (2D grids of pixels). They are widely used in **computer vision** tasks due to their ability to automatically learn and extract **spatial hierarchies of features** from images.

Convolutional Neural Networks (CNNs) are a type of deep learning neural network architecture that is particularly well suited to image classification and object recognition tasks. A CNN works by transforming an input image into a feature map, which is then processed through multiple convolutional and pooling layers to produce a predicted output.

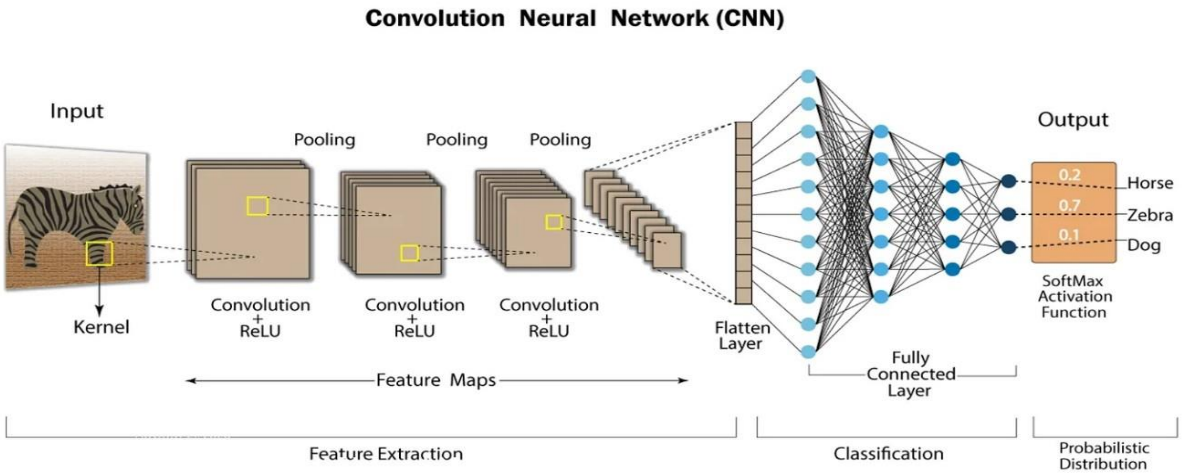


Figure: CNN architecture

Working of Convolutional Neural Network:

A convolutional neural network starts by taking an input image, which is then transformed into a feature map through a series of convolutional and pooling layers. The convolutional layer applies a set of filters to the input image, each filter producing a feature map that highlights a specific aspect of the input image. The pooling layer then downsamples the feature map to reduce its size, while retaining the most important information.

The feature map produced by the convolutional layer is then passed through multiple additional convolutional and pooling layers, each layer learning increasingly complex features of the input image. The final output of the network is a predicted class label or probability score for each class, depending on the task.

CNNs and their components

The layers of a Convolutional Neural Network (CNN) can be broadly classified into the following categories:

1. **Convolutional Layer:**

   The convolutional layer is responsible for extracting features from the input image. It performs a convolution operation on the input image, where a filter or kernel is applied to the image to identify and extract specific features.

   - **Purpose**: Extract features (e.g., edges, textures, shapes).
   - **How**: Uses small matrices called **filters** or **kernels** (e.g., 3×3, 5×5) that slide over the input image and compute dot products.
   - **Feature Maps**: The result is a **feature map** highlighting important visual patterns.
   - **Filters**: Multiple filters detect different types of features — e.g., vertical edges, diagonal lines, etc.
   - Example: A 3×3 filter may detect horizontal edges in an image.

2. **Activation Layer:**

   The activation layer applies a non-linear activation function, such as the ReLU function, to the output of the pooling layer. This function helps to introduce non-linearity into the model, allowing it to learn more complex representations of the input data.

   - **Purpose**: Introduces **non-linearity**.
   - **How**: Applies the Rectified Linear Unit function: ReLU(x) = max(0, x)
   - Helps the model learn complex patterns.

3. **Pooling Layer:**

   The pooling layer is responsible for reducing the spatial dimensions of the feature maps produced by the convolutional layer. It performs a down-sampling operation to reduce the size of the feature maps and reduce computational complexity.

- **Purpose**: **Reduce spatial dimensions** and computational load, while retaining essential features.
- **Types**:
  - **Max Pooling**: Takes the **maximum value** from a region.
  - **Average Pooling**: Takes the **average value**.
- Example: A 2×2 max-pooling reduces a 4×4 feature map to 2×2.

4. **Fully Connected Layer:**

After the convolutional and pooling layers have extracted features from the input image, the dense layer can then be used to combine those features and make a final prediction.
The activations from the previous layers are flattened and passed as inputs to the dense layer, which performs a weighted sum of the inputs and applies an activation function to produce the final output.

The fully connected layer is a traditional neural network layer that connects all the neurons in the previous layer to all the neurons in the next layer. This layer is responsible for combining the features learned by the convolutional and pooling layers to make a prediction.

- **Purpose**: Makes final decisions — e.g., classifying the image as a cat or a dog.
- **How**: Flattens the output from convolution/pooling layers and connects it to fully connected neurons, like a traditional neural network.

Numerical Example:

Given the following 3×3 input image and 2×2 filter,

Input image = $\begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 4 & 6 & 5 \end{bmatrix}$ **and Filter (Kernel)** = $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

1. **Perform the convolution operation with: stride = 1, no padding and activation function = ReLU**
2. **Then, apply 2×2 max pooling with stride = 2 on the resulting feature map.**

Solution:

The typical flow is: **Input → Convolution → ReLU → Pooling**

Step 1: Convolution Operation Table:

Position	Region	Calculation	Result
Top-left (0,0)	$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix} \text{x} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ = (2×1) + (3×0) + (0×0) + (1×1) = 2 + 0 + 0 + 1	**3**
Top-right (0,1)	$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$	$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \text{x} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ =(3×1) + (1×0) + (1×0) + (2×1) = 3 + 0 + 0 + 2	**5**
Bottom-left (1,0)	$\begin{bmatrix} 0 & 1 \\ 4 & 6 \end{bmatrix}$	$\begin{matrix} 0 & 1 \\ 4 & 6 \end{matrix} \text{x} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ = (0×1) + (1×0) + (4×0) + (6×1) = 0 + 0 + 0 + 6	**6**
Bottom-right(1,1)	$\begin{bmatrix} 1 & 2 \\ 6 & 5 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 6 & 5 \end{bmatrix} \text{x} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ = (1×1) + (2×0) + (6×0) + (5×1) = 1 + 0 + 0 + 5	**6**

Feature Map Before ReLU = $\begin{bmatrix} 3 & 5 \\ 6 & 6 \end{bmatrix}$

The ReLU function is: ReLU(x) = max (0, x)

That means:

- If x>0, return x
- If x≤0 , return 0

After ReLU Activation, output = $\begin{bmatrix} 3 & 5 \\ 6 & 6 \end{bmatrix}$

Step 2: Max Pooling:

Applying 2×2 Max Pooling on the output $\begin{bmatrix} 3 & 5 \\ 6 & 6 \end{bmatrix}$, the result is = [6]

Application of CNN:

1. Image Classification:

- Widely used in social media platforms for content tagging or automatic photo organization.
- **Task**: Classify an input image into a predefined category.
- **Example**: Classify whether an image contains a cat, dog, car, etc.
- **How CNN helps**: Learns low-level features like edges in early layers and high-level concepts like faces or shapes in deeper layers.

2. Object Detection:

- Used in self-driving cars to detect other vehicles, people, and road obstacles
- **Task**: Identify what objects are in an image and where they are (bounding boxes).
- **Examples**:
  - YOLO (You Only Look Once)
  - Faster R-CNN
- **Use Cases**:
  - Autonomous vehicles (detecting pedestrians, road signs)
  - Surveillance systems (detecting suspicious behavior)
  - Industrial inspection (detecting defects)

3. Facial Recognition:

- Used in border control, employee attendance, and mobile phone authentication.
- **Task**: Identify or verify a person from a facial image.
- **How CNN helps**: Extracts facial features like eyes, nose, shape, etc., invariant to lighting and pose.
- **Use Cases**:
  - Face unlock in smartphones
  - Biometric security systems
  - Social media tagging

4. Medical Image Analysis:

- CNNs assist doctors in early diagnosis and reduce human error.

- **Task**: Detect abnormalities in medical scans (X-rays, MRIs, CT scans).
- **Use Cases**:
    - Detecting tumors, fractures, infections
    - Segmenting organs for surgery planning
    - Retinal image classification for diabetic retinopathy

5. Scene Understanding / Segmentation:

- Used in agriculture to identify crop health or in smart cities for traffic flow analysis.
- **Semantic Segmentation**: Assign a class label to each pixel (e.g., road, tree, car).
- **Instance Segmentation**: Separate different instances of the same object class (e.g., three different people in a photo).
- **Applications**:
    - Robotics (understanding surroundings)
    - Satellite image processing (detecting land usage, floods)
    - AR/VR applications

6. Document Analysis & OCR (Optical Character Recognition):

- CNNs recognize characters even with noise, skew, or varying fonts.
- **Task**: Recognize printed or handwritten text in images.
- **Use Cases**:
    - Digitizing handwritten notes
    - Automatic number plate recognition
    - Invoice processing in businesses

7. Retail & E-commerce:

- Helps platforms like Amazon or Shopify recommend products visually.
- **Use Cases**:
    - Product recognition from photos (search by image)
    - Virtual try-on using pose estimation
    - Customer behavior analysis via CCTV

8. Art & Style Transfer:

- Used in mobile apps like Prisma or Photoshop Neural Filters.

- **Task**: Apply artistic style of one image to another (e.g., Van Gogh style on your selfie).
- **CNNs** learn content and style features separately, allowing creative applications

9. Gaming and Augmented Reality:

- Powers immersive game experiences and motion control.
- **Use Cases**:
  - Real-time object detection and tracking in AR/VR environments
  - Gesture recognition for gaming controls
  - Scene understanding in AI game agents

10. Security and Surveillance:

- Enhances safety in public places like airports and railway stations.
- **Use Cases**:
  - Intrusion detection
  - Weapon or threat recognition in public spaces
  - Behavioral pattern recognition

Pre-trained CNN Models

Instead of training CNNs from scratch, we can use **pre-trained models** that were trained on large datasets like **ImageNet**.

A **pretrained CNN model** is a convolutional neural network that has already been trained on a large benchmark dataset (like **ImageNet** with 1.2 million images and 1,000 classes). These models have learned general visual features — such as edges, textures, and shapes — that can be **transferred to new tasks**, saving time and computation.

Why Use Pretrained Models?

1. **Faster Development** – Skip training from scratch.
2. **Less Data Required** – You don't need a huge dataset.
3. **Higher Accuracy** – Leverages high-quality models trained on diverse images.
4. **Good for Transfer Learning** – Adapt a model to a different but related task (e.g., classifying medical images using a model trained on animals/vehicles).

Popular Pretrained CNN Models:

Model	Key Features	Parameters	Use Case
VGG16/VGG19	Simple, deep networks with 3×3 conv layers	~138M	Baseline model, easy to use
ResNet	Introduces **residual connections** for deep networks	~25M+	Very deep, avoids vanishing gradients
Inception (GoogLeNet)	Multi-scale convolution (1×1, 3×3, 5×5) in one module	~6.8M	Efficient with good accuracy
MobileNet	Lightweight, optimized for mobile devices	~4M	Real-time apps on edge devices
EfficientNet	Scales depth, width, and resolution efficiently	Varies	High accuracy with fewer params
DenseNet	Dense connections between layers	~8M+	Improves information flow

Benefits of CNNs

Benefit	Description
1. **Automatic Feature Extraction**	CNNs learn features (e.g., edges, textures) from data automatically—no manual feature engineering needed.
2. **Parameter Sharing**	Filters (kernels) are reused across the image, reducing the number of parameters and improving efficiency.
3. **Local Connectivity**	CNNs exploit spatial structure by connecting each neuron to a small region of the input, capturing local patterns.
4. **Scalable and Deep**	Easy to stack more layers (convolution, pooling) to learn complex patterns and hierarchies.
5. **Effective for Image Tasks**	Performs exceptionally well on tasks like image classification, object detection, segmentation, etc.

Benefit	Description
6. **Transfer Learning Friendly**	Works well with pre-trained models on large datasets (like ImageNet) and fine-tuning on new tasks.
7. **Parameter Efficient**	Requires fewer parameters than fully connected networks for large inputs like images.

Limitations of CNNs

Limitation	Description
1. **Data-Hungry**	Requires **large labeled datasets** to perform well. With small data, it can overfit.
2. **High Computational Cost**	Training deep CNNs requires powerful GPUs and large memory, especially for high-resolution images.
3. **Lack of Invariance**	CNNs are **not naturally invariant** to rotation, scale, and viewpoint unless trained on such variations.
4. **Black Box Nature**	CNN decisions are often **hard to interpret** — they don't explain why a prediction was made.
5. **Architecture Tuning Required**	Requires careful tuning of hyperparameters (filter size, number of layers, stride, etc.) for optimal results.
6. **Not Ideal for Non-Grid Data**	CNNs are specialized for **grid-like inputs** (e.g., images); not suitable for sequences (use RNNs) or graphs (use GNNs).
7. **Sensitive to Input Distribution**	Performance may drop if test data differs significantly from training data (domain shift).

Assignment:

Discuss pre-trained models: VGG.

4.3.2 Recurrent Neural Networks (RNNs)

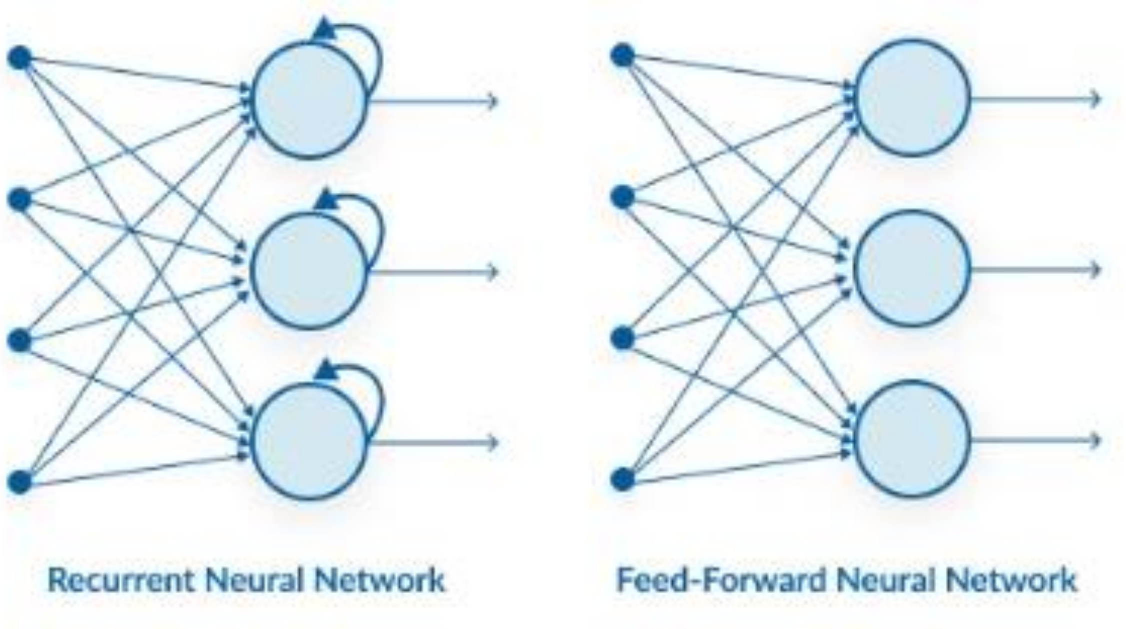
Basics of RNNs

Unlike traditional feedforward neural networks (like CNNs), **RNNs are designed to handle sequential data** — data where order matters. They process inputs **step by step** while **maintaining memory** of previous steps through internal hidden states.

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.

Imagine reading a sentence and you try to predict the next word, you don't rely only on the current word but also remember the words that came before. RNNs work similarly by "remembering" past information and passing the output from one step as input to the next i.e it considers all the earlier words to choose the most likely next word. This memory of previous steps helps the network understand context and make better predictions.

Recurrent Neural Network vs Feed-forward Neural Network:



Feature	Feed-forward Neural Network (FNN)	Recurrent Neural Network (RNN)
Data Flow	Straight through (no loops)	Loops through time (with memory)
Memory	No memory of past inputs	Maintains memory via hidden states
Input Assumption	Inputs are independent	Inputs are sequentially dependent
Architecture	Static layers, no time dimension	Unfolds across time steps
Training	Standard backpropagation	Backpropagation Through Time (BPTT)

Transformation of a Feedforward NN to RNN:

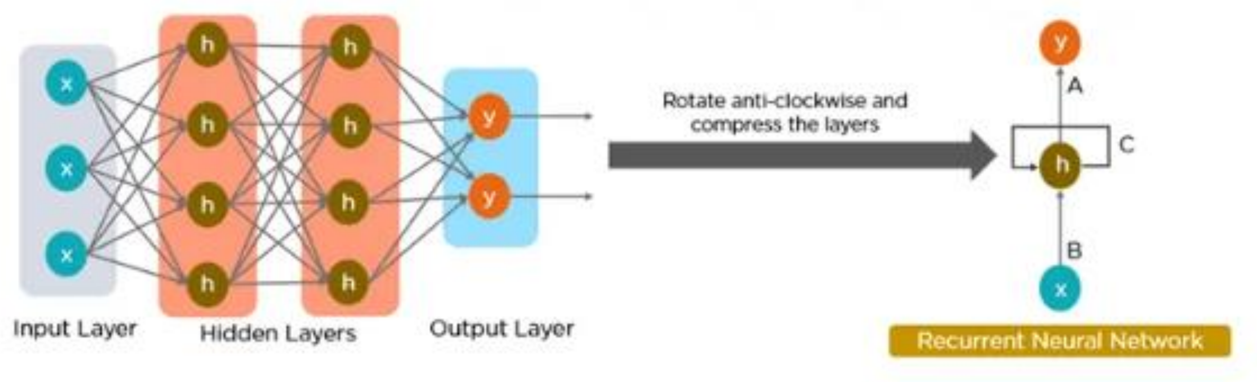


Figure: Transformation of a Feedforward Neural Network into a Recurrent Neural Network (RNN) by Unfolding Over Time

The figure illustrates how a standard feedforward neural network, which processes fixed inputs through layered connections, can be conceptually transformed into a Recurrent Neural Network (RNN) for sequential data. By **rotating the network and compressing the layers**, the figure shows how RNNs maintain a **single hidden layer with a feedback loop**, enabling the model to **retain memory of previous inputs**.

- In a **feedforward network**, each input is processed independently.
- In an **RNN**, the hidden state h is **reused across time steps**, receiving both current input x and previous state h as input.
- The output y is influenced not just by the current input but also by **past context**, making RNNs ideal for sequence modeling tasks like text, time series, or speech.

Key Components of RNNs:

There are mainly two components of RNNs:

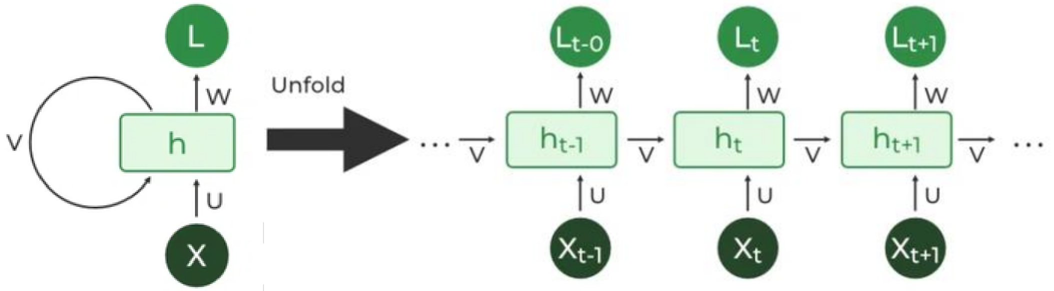


Figure: Unfolding of RNN

1. Recurrent Neurons

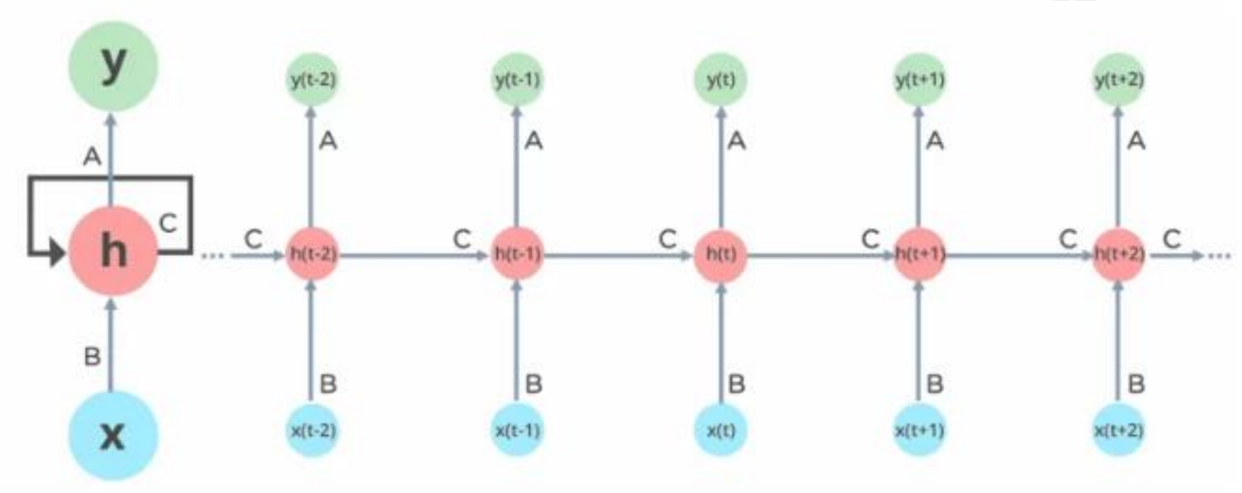
The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

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.

How do RNN work?



Imagine reading a sentence word by word.

When you get to a new word, your brain remembers the previous words to understand the meaning. **RNNs work in a similar way.**

Here’s what happens at each step:

1. **It takes the current input** (like a word or number),
2. **It remembers what it saw before** (this memory is called a "hidden state"),
3. **It gives an output** (like a prediction or classification).

Then it moves to the next input, using both:

- the **new input**, and
- the **memory from the previous step**.

This helps the RNN understand the context and patterns in the sequence.

Example:

Let’s say you give it this sentence: "I love eating chocolate."

It reads one word at a time:

- Reads "I" → stores it in memory
- Reads "love" → remembers "I", combines with "love"
- Reads "eating" → remembers "I love", combines with "eating"
- ...

By the time it gets to "chocolate", it has the full context of the sentence so it can make a smart prediction or generate text.

Numerical example:

Given a simple Recurrent Neural Network (RNN) with the following parameters:

- Initial hidden state h0=0
- Input sequence: x=[1,2,3]
- Input weight W=1.0
- Hidden weight U=0.5
- Output weight (W_o) = 0.8
- No bias
- Activation function: tanh(z) (to keep values in range [-1, 1])

Calculate the hidden states h1, h2, and h3 and the final output $y_{t.}$

Solution:

Time Step (t)	**Input x_t**	**Previous Hidden State h_{t-1}**	**Linear Computation $z_t = W \cdot x_t + U \cdot h_{t-1}$**	**Hidden State $h_t = \tanh(z_t)$**
0	–	**0.0000** (initial)	–	**0.0000**
1	1	0.0000	1.0 × 1 + 0.5 × 0.0000 = 1.0000	**0.7616**
2	2	0.7616	1.0 × 2 + 0.5 × 0.7616 = 2.3808	**0.9830**
3	3	0.9830	1.0 × 3 + 0.5 × 0.9830 = 3.4915	**0.9981**

Final output $y_t = W_o * h_t$

$= 0.8 * 0.9981 = 0.7984$

Vanishing Gradient Problem:

When the sentence or sequence is very long, RNNs **can forget earlier words**. This is called the **vanishing gradient problem**. It means the network struggles to learn from long-term memory.

RNNs suffer from the matter of vanishing gradients.
The gradients carry information utilized in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the training of long data sequences difficult.
To fix this, special types of RNNs were made:

- **LSTM** (Long Short-Term Memory)
- **GRU** (Gated Recurrent Unit)

These networks have little **gates** inside them that help **decide what to remember and what to forget**. This makes them **much better at handling long sequences** like full paragraphs, stock trends, or audio.

4.3.2.1 Long Short-term Memory (LSTM)

Long Short-Term Memory (LSTM) is an enhanced version of the Recurrent Neural Network (RNN) designed by Hochreiter and Schmidhuber.
LSTMs can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting.
LSTM Architecture:

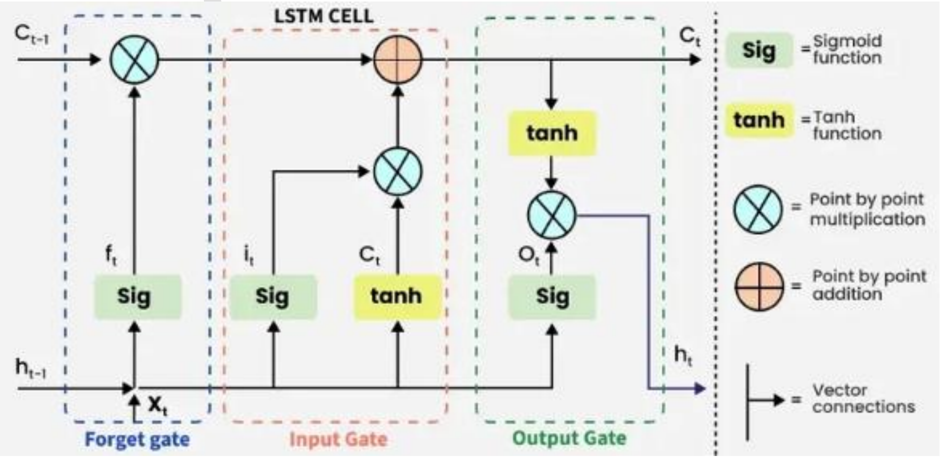


Figure: LSTM model

Long Short-Term Memory Networks (LSTMs) introduce a memory mechanism to overcome the vanishing gradient problem.

Each LSTM cell has three gates:

1. Forget Gate:
   - The information that is no longer useful in the cell state is removed with the forget gate
   - Two inputs X_t (input at the particular time) and h_{t-1} (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias.
   - The resultant is passed through an activation function which gives a binary output.
   - If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.
   - The equation for the forget gate is:

     $f_t = \sigma (W_f \cdot [h_{t-1}, X_t] + b_f)$

2. Input Gate:
   - Controls how much new information should be added to the cell state.
   - The equation for the input gate is:

     $i_t = \sigma(W_I \cdot [h_{t-1},X_t] + b_i)$

     $\widehat{C_\iota} = \tanh (W_c \cdot [h_{t-1},X_t] + b_c)$

     $C_t = f_t \odot C_{t-1} + i_t \odot \widehat{C_t}$

3. Output Gate:
   - Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.
   - The equation for the output gate is:

     $o_t = \sigma (W_o \cdot [h_{t-1}, X_t] + b_o)$

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Applications of LSTM:

1. Language Modeling & Translation
2. Speech Recognition & Synthesis
3. Time Series Forecasting (stock, weather, energy)
4. Video Analysis & Captioning
5. Anomaly Detection in sequences
6. Healthcare monitoring & diagnosis
7. Robotics control
8. Handwriting Recognition
9. Music Generation
10. Recommendation Systems

10.3.2.1 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.
- LSTMs are very complex structure with higher computational cost. 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.

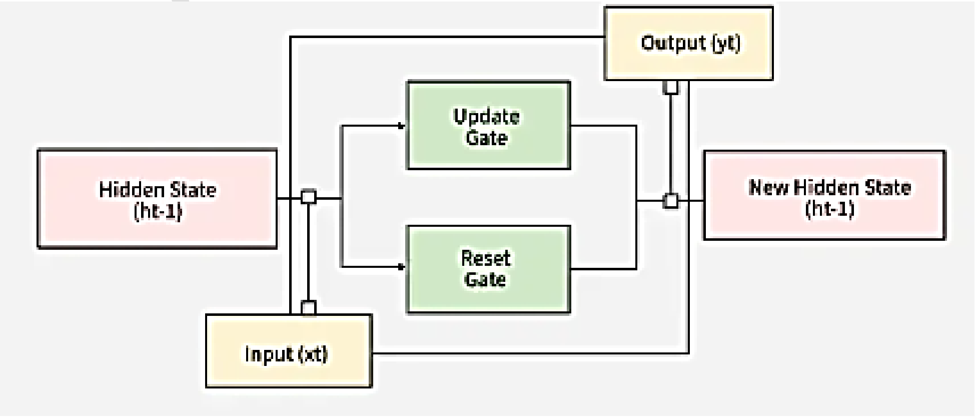


Figure: GRU block

- GRUs simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism.
- This design is computationally efficient, often performing similarly to LSTMs and is useful in tasks where simplicity and faster training are beneficial.

How GRU works?

- Like other recurrent neural network architectures, GRU processes sequential data one element at a time, updating its hidden state based on the current input and the previous hidden state.
- At each time step, the GRU computes a “candidate activation vector” that combines information from the input and the previous hidden state. This candidate vector is then used to update the hidden state for the next time step.
- **Candidate activation vector (h'_t)** is computed using the current input x and a modified version of the previous hidden state that is "reset" by the reset gate:

  $$h'_t = \tanh(W_h * [r_t * h_{t-1}, x_t])$$
- The new hidden state h_t is computed by combining the candidate activation vector with the previous hidden state, weighted by the update gate: $h_t = (1 - z_t) * h_{t-1} + z_t * h'_t$
- The **candidate activation vector** is computed using two gates:

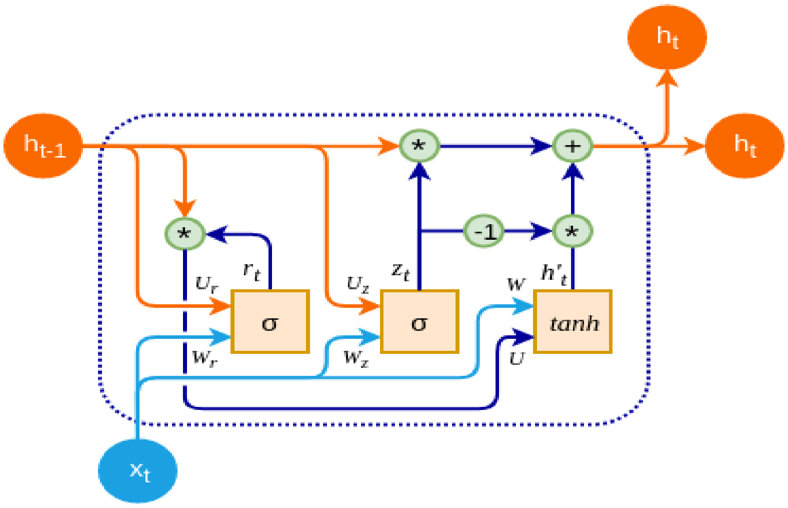


Figure: Components of GRU

1. **Update Gate (z_t):**
    - This gate decides how much information from previous hidden state should be retained for the next time step.
    - The update gate controls how much of the new information xt should be used to update the hidden state.
    - $z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$

      where:

      x_t = current input

      h_{t-1} = previous hidden state,

      W_z is weight matrices during training

      σ = sigmoid function

2. **Reset Gate (r_t):**
    - This gate determines how much of the past hidden state should be forgotten.
    - The reset gate determines how much of the previous hidden state h_{t-1} should be forgotten.
    - $r_t = \sigma (W_r \cdot [h_{t-1} , x_t])$

      where:

      x_t = current input

      h_{t-1} = previous hidden state,

      Wr is weight matrices during training

      σ = sigmoid function

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.

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

Sequence-to-Sequence Models in RNN

Sequence-to-Sequence (Seq2Seq) models are used in tasks where both input and output are sequences, possibly of different lengths.

The Sequence-to-Sequence (Seq2Seq) model is a type of neural network architecture widely used in machine learning particularly in tasks that involve translating one sequence of data into another.

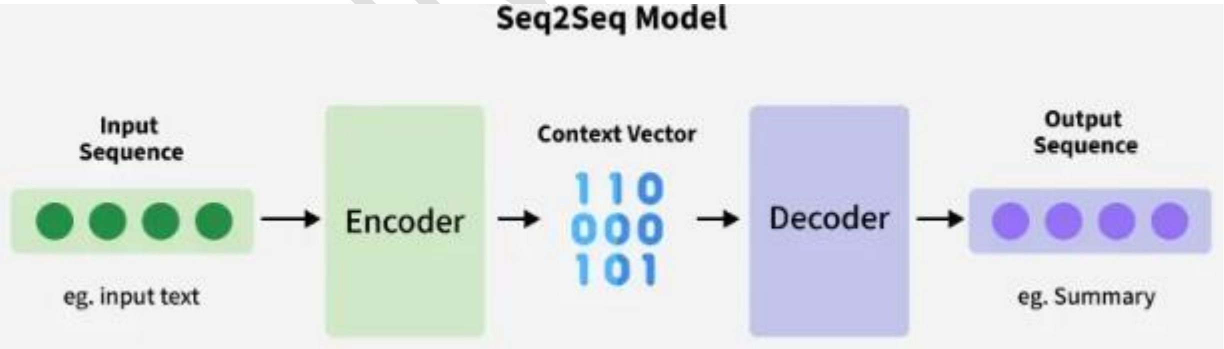
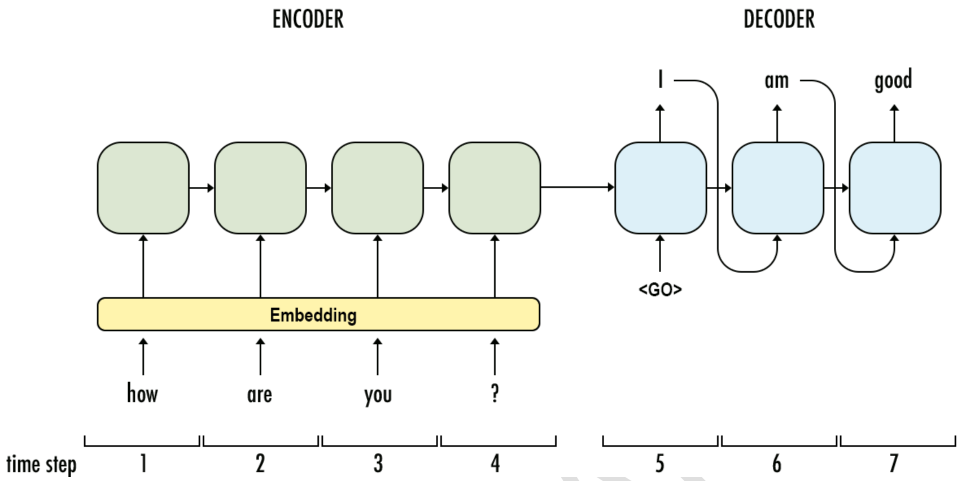


Figure: Encode and Decoder Stack in seq2seq model

A typical **Seq2Seq model** consists of two main parts:

1. **Encoder:**
   - An RNN (or LSTM/GRU) that reads the input sequence one step at a time.
   - It processes the entire input into a **fixed-size context vector** (hidden state).

2. **Decoder:**
   - Another RNN that takes the context vector and generates the output sequence step by step.
   - At each step, it predicts the next token in the sequence.



Common Applications:

- Machine Translation (English → French)
- Chatbots
- Text Summarization
- Speech Recognition
- Image Captioning (with CNN + RNN)
- Code Generation

Enhancements:

- **Attention Mechanism**: Helps decoder focus on specific encoder outputs at each time step.
- **Bidirectional RNN**: Encoder reads the input both forward and backward.

10.3.2.2 Applications of Time-series prediction

Time-series prediction involves forecasting future values based on past sequential data. It's widely used across industries wherever trends, patterns, or sequential behaviors exist.

Major Applications:

1. **Financial Sector**

- **Stock Price Forecasting**: Predict future stock prices using past data trends.
- **Cryptocurrency Trends**: Anticipate crypto value fluctuations.
- **Exchange Rate Prediction**: Forecast currency exchange rates (e.g., USD to EUR).

2. Weather and Environment

- **Weather Forecasting**: Predict temperature, humidity, or rainfall based on historical data.
- **Air Quality Prediction**: Estimate pollution levels using past environmental measurements.
- **Flood or Earthquake Forecasting**: Use sensor and seismic data to forecast natural disasters.

3. Energy Sector

- **Electricity Demand Forecasting**: Anticipate power needs to manage supply efficiently.
- **Load Balancing**: Distribute electricity demand across the grid to prevent overloads.
- **Solar/Wind Energy Prediction**: Predict power output from renewable sources.

4. Healthcare

- **Vital Sign Prediction**: Monitor and predict patient vitals like heart rate, blood pressure.
- **Disease Progression Modeling**: Track and forecast the course of chronic diseases.
- **Event Prediction**: Detect upcoming medical events like epileptic seizures.

5. Industry and Manufacturing

- **Predictive Maintenance**: Forecast machine failures before they happen to reduce downtime.
- **Equipment Failure Prediction**: Monitor usage data to detect wear and potential failures.
- **Production Planning**: Adjust manufacturing schedules based on demand forecasts.

6. Supply Chain & Retail

- **Demand Forecasting**: Predict future product demand to optimize inventory.
- **Inventory Management**: Keep stock levels optimal by forecasting usage patterns.
- **Sales Forecasting**: Predict future sales to plan marketing and production.

7. Transportation

- **Traffic Flow Prediction**: Anticipate congestion using road and sensor data.
- **Flight Delay Prediction**: Use weather and historical flight data to forecast delays.
- **Vehicle Movement Forecasting**: Predict routes or delivery times in logistics.

8. IT & Security

- **Network Load Prediction**: Anticipate internet traffic or server load to ensure uptime.
- **Server Resource Usage**: Forecast CPU, memory, and storage needs.
- **Anomaly Detection**: Spot unusual patterns like fraud or cyber-attacks from usage data.

***