DEEP LEARNING

Dr Soumya Ranjan Mishra

Introduction to Neural Networks and McCulloch-Pitts Neuron

Introduction to Neural Networks

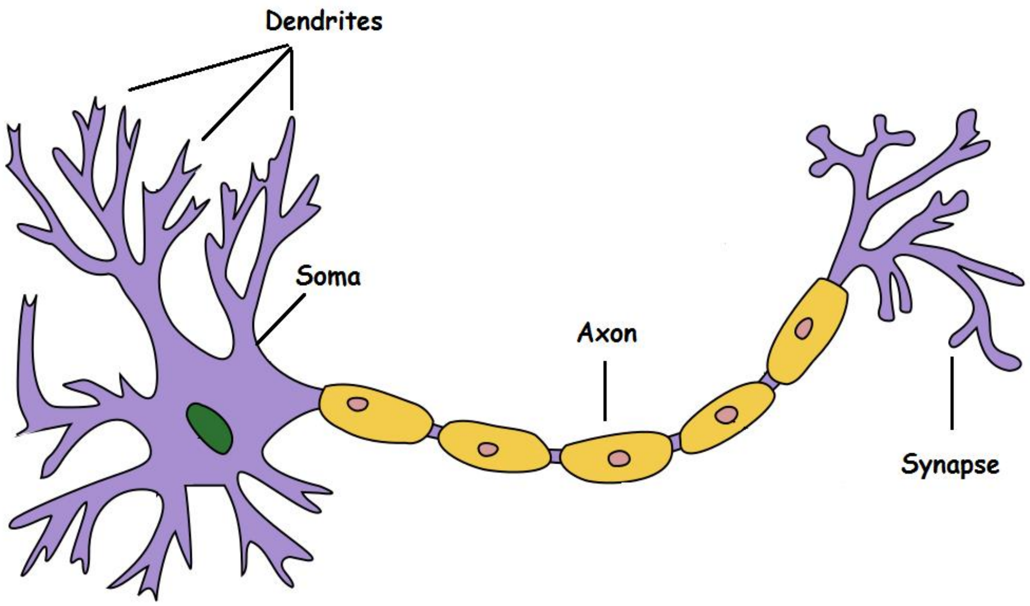
- Neural networks are computational models inspired by the structure and functioning of the human brain.

- They consist of interconnected processing units called neurons, which work together to solve complex problems.

- Neural networks are widely used in **pattern recognition, machine learning, and artificial intelligence applications.**

Biological Inspiration

- The human brain consists of billions of neurons connected by synapses.
- Each neuron receives input signals, processes them, and transmits an output signal.
- **Neural networks mimic this biological process** in a simplified mathematical form.



Dendrite: Receives signals from other neurons

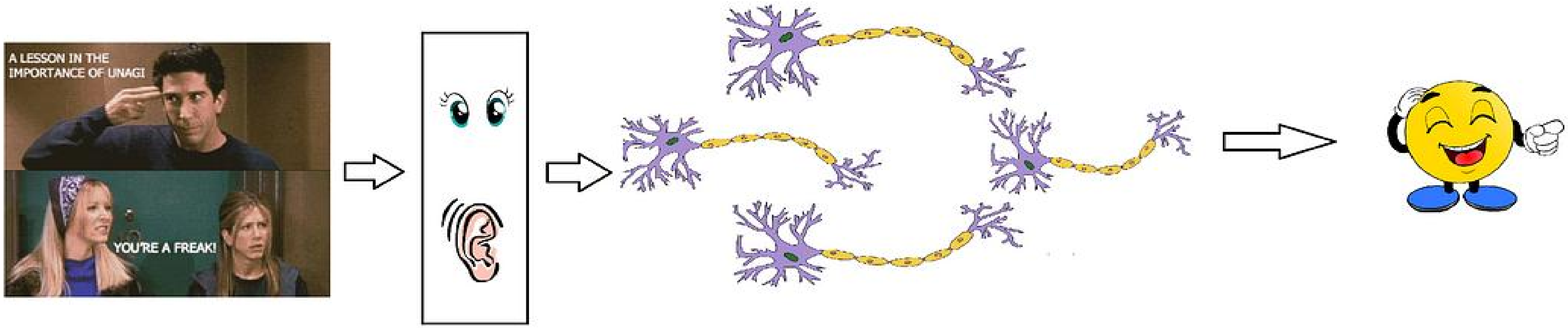
Soma: Processes the information

Axon: Transmits the output of this neuron

Synapse: Point of connection to other neurons

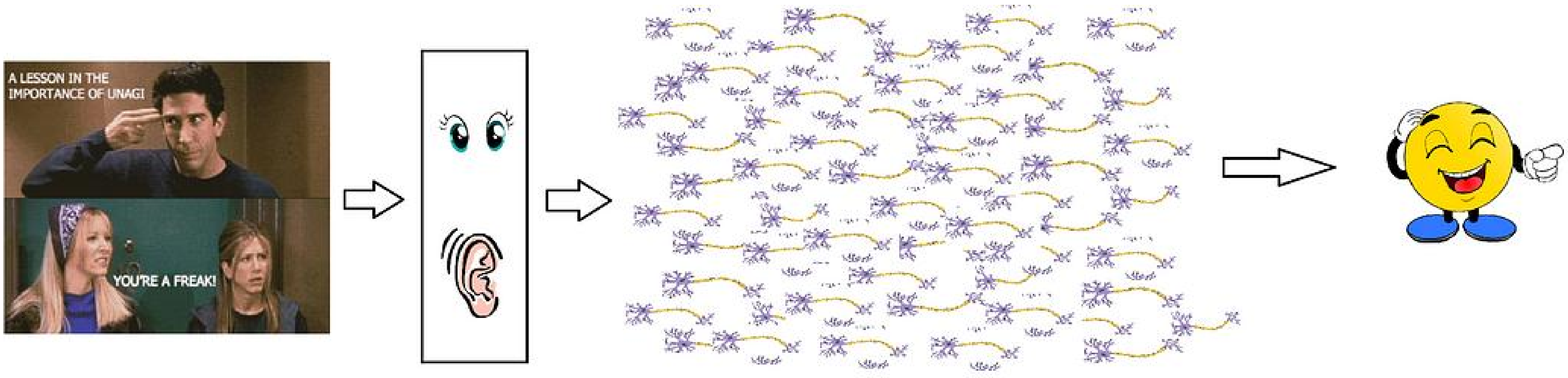
Our sense organs interact with the outer world and send the visual and sound information to the neurons.

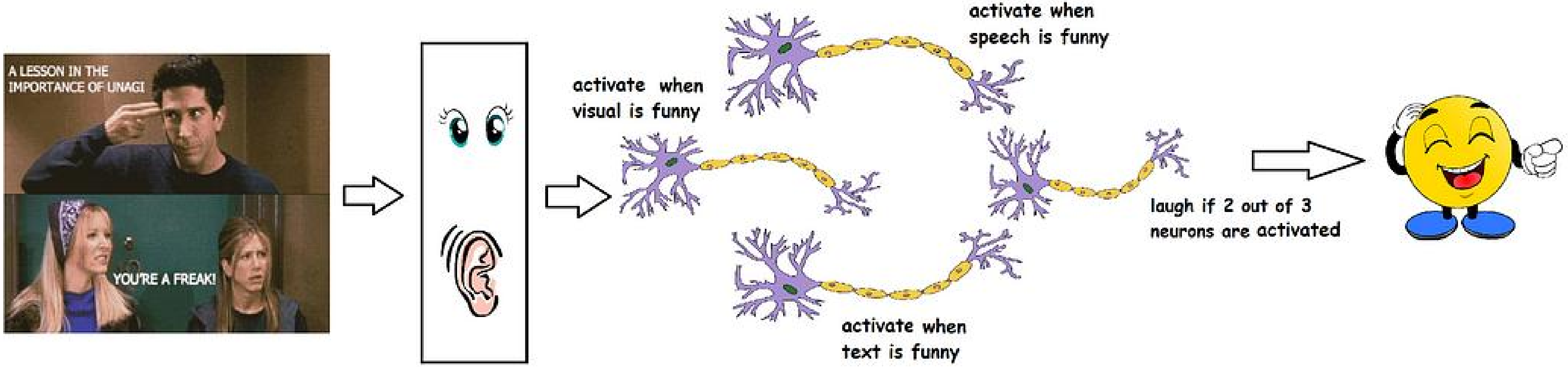
Let's say you are watching Friends. Now the information your brain receives is taken in by the “laugh or not” set of neurons that will help you make a decision on whether to laugh or not.



In reality, it is not just a couple of neurons which would do the decision making.

There is a massively parallel **interconnected network of 10^{11} neurons (100 billion) in our brain** and their connections are not as simple as I showed you above. It might look something like this:




A LESSON IN THE
IMPORTANCE OF UNAGI
YOU'RE A FREAK!
activate when
visual is funny
activate when
speech is funny
activate when
text is funny
laugh if 2 out of 3
neurons are activated

Artificial Neural Networks (ANNs)

An artificial neural network consists of:

Input Layer: Receives input signals (features).

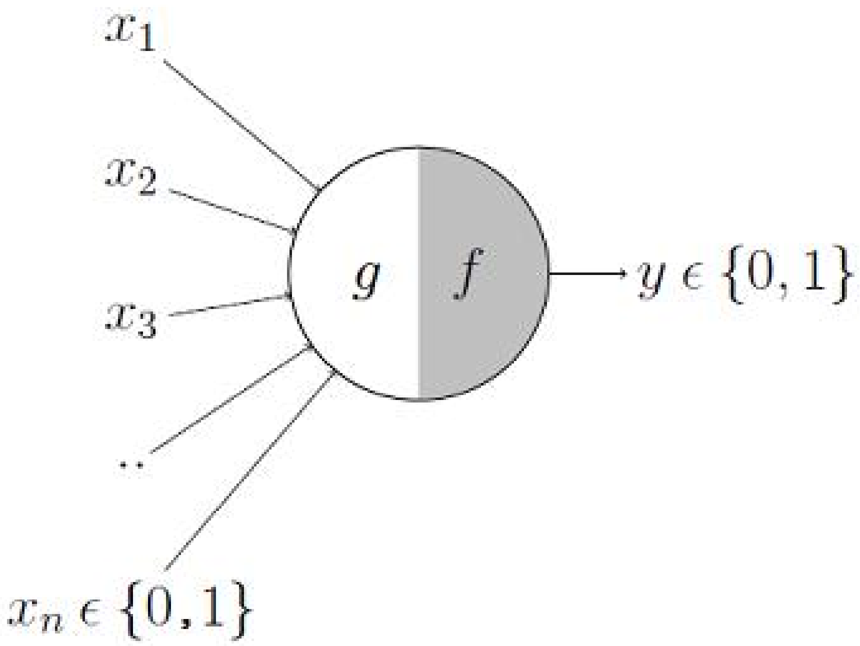
Hidden Layers: Processes information using weighted connections.

Output Layer: Produces the final result.

Each connection in the network has an associated weight, which determines the strength of the connection. The neuron applies an activation function (such as step function, sigmoid, ReLU) to decide the output.

McCulloch-Pitts Neuron

The first computational model of a neuron was proposed by **Warren MuCulloch** (neuroscientist) and Walter Pitts (logician) in 1943.



It may be divided into 2 parts.

The first part, ***g*** takes an input (ahem dendrite ahem), performs an aggregation and based on the aggregated value the second part, ***f*** makes a decision.

Lets suppose that I want to predict my own decision, whether to watch a random football game or not on TV.

The inputs are all boolean i.e., {0,1} and my output variable is also boolean {0: Will watch it, 1: Won't watch it}.

So, x1 could be is *PremierLeagueOn* (I like Premier League more)

x2 could be isIt *AFriendlyGame* (I tend to care less about the friendlies)

x3 could be is *NotHome* (Can't watch it when I'm running errands. Can I?)

x4 could be is *ManUnitedPlaying* (I am a big ManUnited fan) and so on.

These inputs can either be **excitatory or inhibitory**.

Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs i.e., if x3 is 1 (not home) then my output will always be 0 i.e., the neuron will never fire, so x3 is an inhibitory input.

Excitatory inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together. Formally, this is what is going on:

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad if \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad if \quad g(\mathbf{x}) < \theta \end{aligned}$$

Structure of M-P Neuron

An M-P neuron consists of:

Inputs: $x_1, x_2, \ldots, x_n$

Weights: $w_1, w_2, \ldots, w_n$ (Each input has an associated weight)

Summation Function: Computes weighted sum S

Threshold (θ)**:** A fixed value that determines neuron activation

Activation Function: Step function

Mathematical Model

The M-P neuron calculates the weighted sum of inputs:

$$S = \sum_{i=1}^{n} w_i x_i$$

The output Y is determined by a step function:

$$Y = \begin{cases} 1, & \text{if } S \geq \theta \\ 0, & \text{if } S < \theta \end{cases}$$

Example of M-P Neuron

Let's consider a neuron with two inputs x_1 and x_2, weights $w_1 = 1, w_2 = 1$, and threshold $\theta = 1.5$.

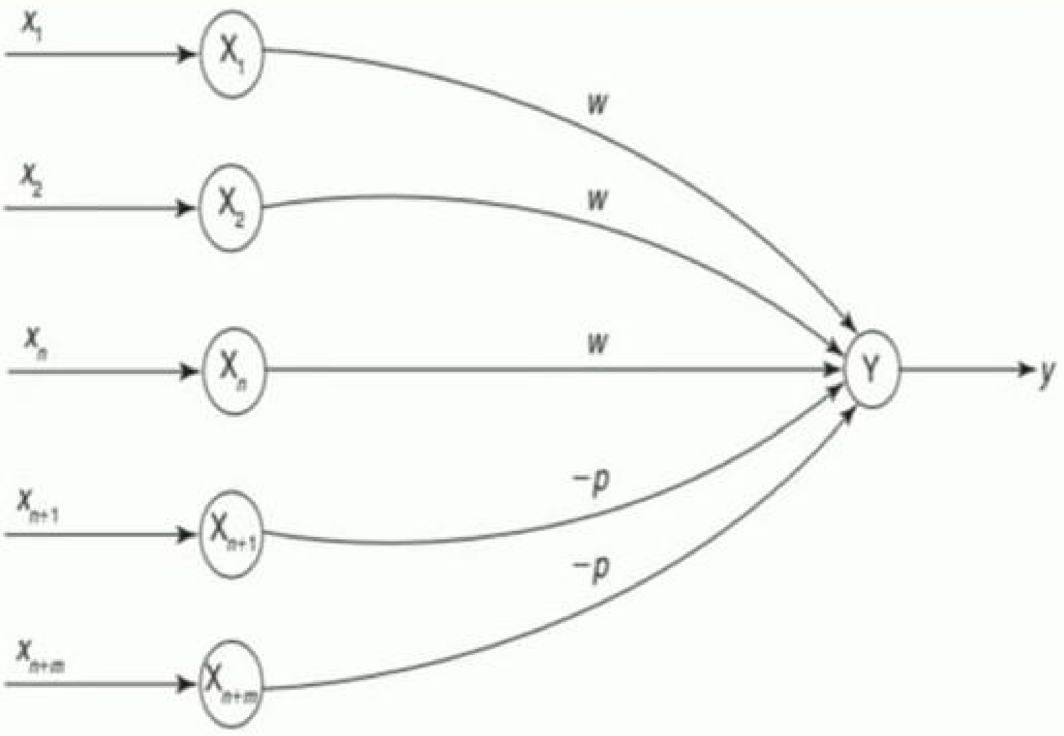
x_1	x_2	$S = w_1x_1 + w_2x_2$	Output Y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

Implement AND function using McCulloch–Pitts Neuron

- The McCulloch–Pitts neuron was the earliest neural network discovered in 1943.
- It is usually called as M–P neuron.
- Since the firing of the output neuron is based upon the threshold, the activation function here is defined as

$$f(y_{in}) = \begin{cases} 1 & \text{if} \quad y_{in} \geq \theta \\ 0 & \text{if} \quad y_{in} < \theta \end{cases}$$

- The threshold value should satisfy the following condition: $\theta > nw - p$



Implement AND function using McCulloch–Pitts Neuron

- Consider the truth table for AND function
- The M–P neuron has no particular training algorithm
- In M-Pneuron, only analysis is being performed.
- Hence, assume the weights be w1 = 1 and w2 = 1.

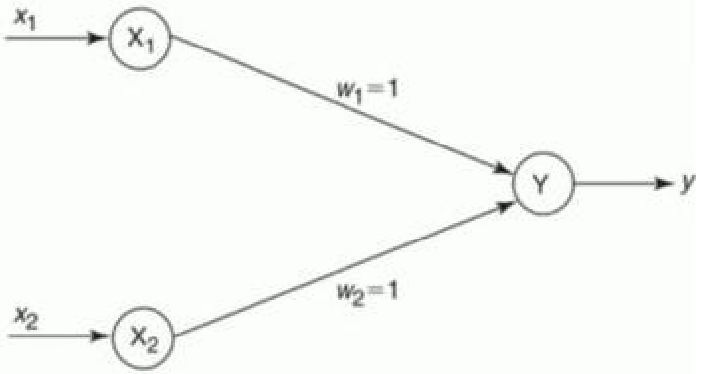
x_1	x_2	y
1	1	1
1	0	0
0	1	0
0	0	0

$$(1, 1),\ y_{in} = x_1w_1 + x_2w_2 = 1\times 1 + 1\times 1 = 2$$

$$(1, 0),\ y_{in} = x_1w_1 + x_2w_2 = 1\times 1 + 0\times 1 = 1$$

$$(0, 1),\ y_{in} = x_1w_1 + x_2w_2 = 0\times 1 + 1\times 1 = 1$$

$$(0, 0),\ y_{in} = x_1w_1 + x_2w_2 = 0\times 1 + 0\times 1 = 0$$



Implement AND function using McCulloch–Pitts Neuron

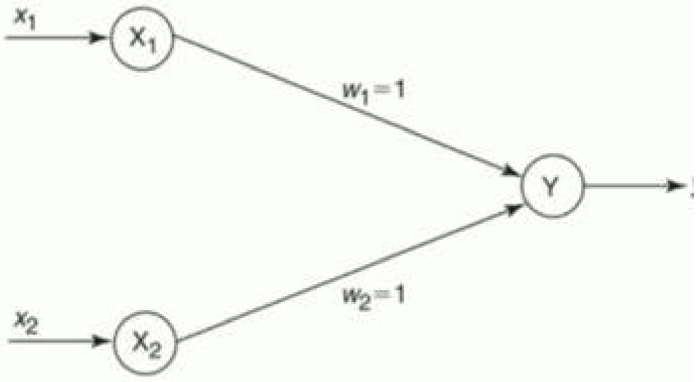
- This can also be obtained by

$$\theta \geq nw - p$$

- Here, n = 2, w = 1 (excitatory weights) and p = 0 (no inhibitory weights).
- Substituting these values in the above-mentioned equation we get $\theta \geq 2 \times 1 - 0 \Rightarrow \theta \geq 2$
- Thus, the output of neuron Y can be written

$$y = f(y_{in}) = \begin{cases} 1 & \text{if} \quad y_{in} \geq 2 \\ 0 & \text{if} \quad y_{in} < 2 \end{cases}$$

x_1	x_2	y
1	1	1
1	0	0
0	1	0
0	0	0



Least Mean Squares (LMS) Algorithm in Neural Networks

The Least Mean Squares (LMS) algorithm is a supervised learning algorithm used for training single-layer neural networks (also known as adaptive linear neurons, or ADALINE).

It is a type of gradient descent algorithm that minimizes the mean squared error between the actual and predicted output.

Mathematical Formulation

The LMS algorithm updates the weight vector w to minimize the Mean Squared Error (MSE):

Initialization:

Initialize weights w randomly or set them to small values.
Set a learning rate η (small positive value).

Compute Net Input:

$$y = w^T x$$

where:

- y = predicted output
- x = input vector
- w = weight vector

Compute Error:

$$e = d - y$$

where:

- d = desired (actual) output
- e = error

Update Weights Using LMS Rule:

$$w_{new} = w_{old} + \eta e x$$

This is based on gradient descent to minimize the error.

Repeat: Iterate until convergence (i.e., error is minimized below a threshold or after a fixed number of iterations).

Problem Statement

Consider a neural network with **two inputs** and a single output. The goal is to train the weights using LMS to approximate a given function.

Given Data

Input x_1	Input x_2	Desired Output d
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

Initial Conditions

- Learning Rate $\eta = 0.1$
- Initial Weights: $w_1 = 0.2$, $w_2 = -0.3$
- Initial Bias: $b = 0.1$

Step-by-Step Calculation for First Epoch

For the first training example $(x_1 = 1, x_2 = 1, d = 1)$:

1. **Compute Net Input**

$$y = (0.2 \times 1) + (-0.3 \times 1) + 0.1 = 0.2 - 0.3 + 0.1 = 0$$

2. **Compute Error**

$$e = d - y = 1 - 0 = 1$$

3. Update Weights and Bias

$$w_1^{new} = w_1 + \eta e x_1 = 0.2 + (0.1 \times 1 \times 1) = 0.3$$

$$w_2^{new} = w_2 + \eta e x_2 = -0.3 + (0.1 \times 1 \times 1) = -0.2$$

$$b^{new} = b + \eta e = 0.1 + (0.1 \times 1) = 0.2$$

Repeat for remaining training samples

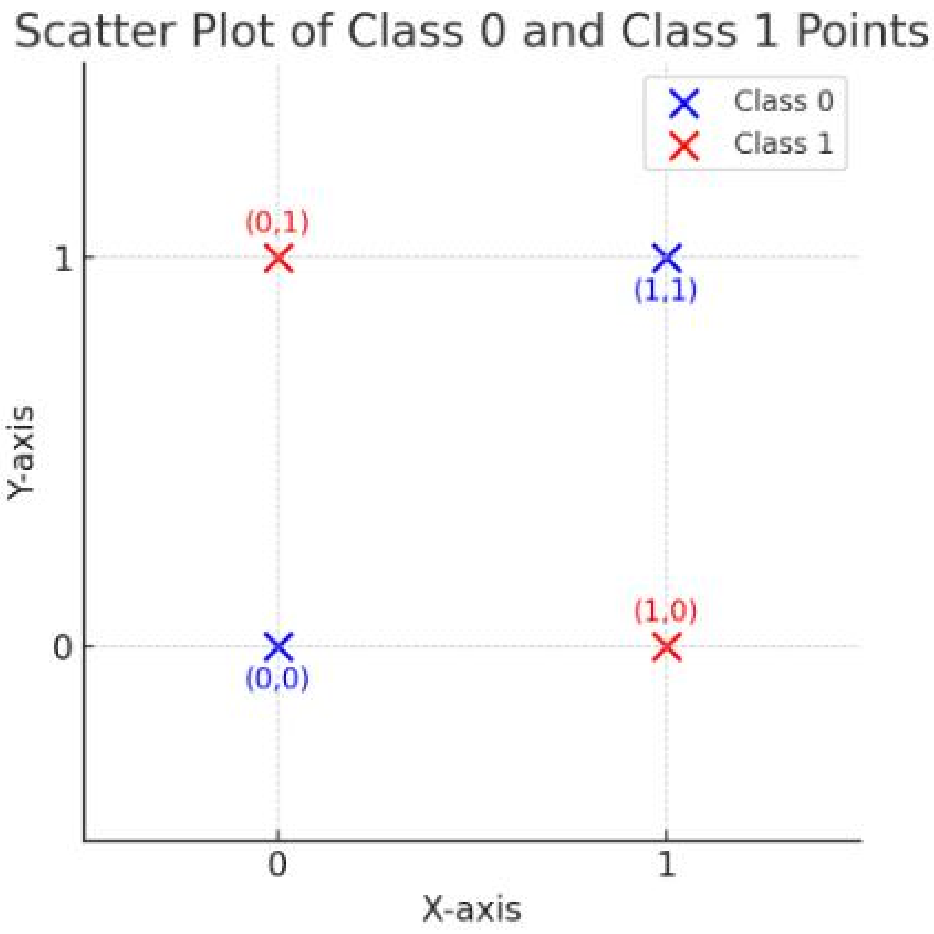
After multiple iterations, the weights stabilize, and the neural network converges to a solution that minimizes the error.

Example: XOR Classification

The XOR (exclusive OR) truth table is:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Example: XOR Classification



Class 0: (0,0) and (1,1)

Class 1: (0,1) and (1,0)

These points are not linearly separable. That is, you can't draw a straight line that separates the two classes.

Let’s design a neural network by hand that can solve XOR.

Hidden Layer:

$$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + b_1)$$

$$h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + b_2)$$

Let’s take:

Layer	Weights	Bias
h_1	$w_{11} = 20$, $w_{12} = 20$	$b_1 = -10$
h_2	$w_{21} = -20$, $w_{22} = -20$	$b_2 = 30$

Output Layer:

$$y = \sigma(w_{o1}h_1 + w_{o2}h_2 + b_o)$$

Take:

- $w_{o1} = 20, w_{o2} = 20, b_o = -30$

Case 1: $x_1 = 0, x_2 = 0$

$$h_1 = \sigma(20 \times 0 + 20 \times 0 - 10) = \sigma(-10) \approx 0$$

$$h_2 = \sigma(-20 \times 0 - 20 \times 0 + 30) = \sigma(30) \approx 1$$

$$y = \sigma(20 \times 0 + 20 \times 1 - 30) = \sigma(-10) \approx 0$$

Case 2: $x_1 = 0, x_2 = 1$

$$h_1 = \sigma(20 \times 0 + 20 \times 1 - 10) = \sigma(10) \approx 1$$

$$h_2 = \sigma(-20 \times 0 - 20 \times 1 + 30) = \sigma(10) \approx 1$$

$$y = \sigma(20 \times 1 + 20 \times 1 - 30) = \sigma(10) \approx 1$$

Case 3: $x_1 = 1, x_2 = 0$

$$h_1 = \sigma(20 \times 1 + 20 \times 0 - 10) = \sigma(10) \approx 1$$

$$h_2 = \sigma(-20 \times 1 - 20 \times 0 + 30) = \sigma(10) \approx 1$$

$$y = \sigma(20 \times 1 + 20 \times 1 - 30) = \sigma(10) \approx 1$$

Case 4: $x_1 = 1, x_2 = 1$

$$h_1 = \sigma(20 \times 1 + 20 \times 1 - 10) = \sigma(30) \approx 1$$

$$h_2 = \sigma(-20 \times 1 - 20 \times 1 + 30) = \sigma(-10) \approx 0$$

$$y = \sigma(20 \times 1 + 20 \times 0 - 30) = \sigma(-10) \approx 0$$

- A 2-layer neural network **(1 hidden layer with nonlinear activations)** can learn XOR, while a linear model cannot.
- This clearly shows the power of hidden layers and nonlinear activation functions in neural networks.

Perceptron Model

- The perceptron is one of the simplest types of artificial neural networks used for binary classification.
- It is a type of linear classifier that updates its weights based on the errors made during training.
- The perceptron is inspired by biological neurons in the human brain, which receive inputs, process them, and produce an output.
- It attempts to mimic the way neurons fire when the total input exceeds a certain threshold.

Structure of Perceptron

A perceptron consists of:

Inputs (x1,x2,...,xn): Features of the dataset.

Weights (w1,w2,...,wn): Determines the importance of each input.

Bias (b): Helps shift the decision boundary.

Summation Function: Computes the weighted sum of inputs.

Activation Function: Applies a threshold to produce the final output

Mathematical Representation

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where $f(\cdot)$ is a step activation function:

$$f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Multilayer Perceptron (MLP) and Hidden Layer Representation

A Multilayer Perceptron (MLP) is a type of artificial neural network (ANN) that consists of multiple layers of neurons, enabling it to learn complex patterns and perform classification and regression tasks.

MLP consists of three types of layers:

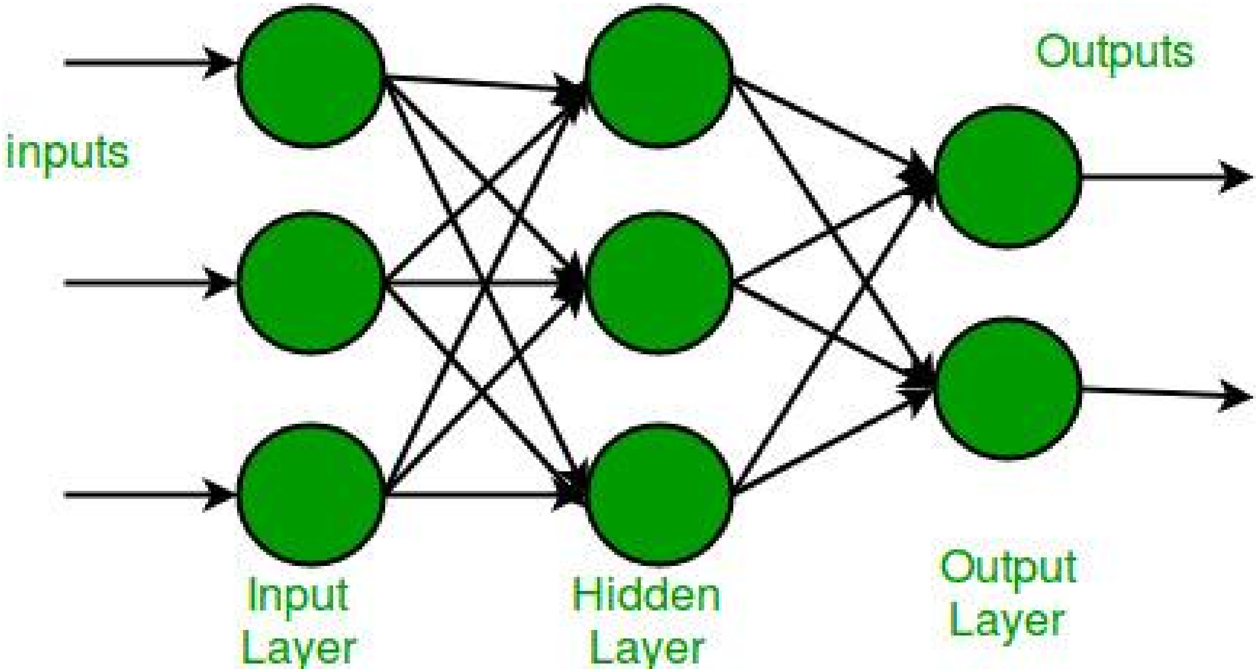
Input Layer: Receives the input features.

Hidden Layer(s): Performs computations and feature extraction.

Output Layer: Produces the final prediction or classification.

Each neuron in a layer is connected to all neurons in the next layer, making MLP a fully connected network.

Multilayer Perceptron (MLP) and Hidden Layer Representation



Role of Hidden Layers in MLP

- The hidden layers help in learning hierarchical representations of data.
- Each neuron in a hidden layer applies a weighted sum followed by an activation function to introduce non-linearity.
- More hidden layers allow the network to learn complex and abstract patterns, enabling deep learning capabilities.

For a given input $X = [x_1, x_2, ..., x_n]$, the output of a hidden neuron is computed as:

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

$$a = f(z)$$

Activation Functions in MLP

Sigmoid: Used in binary classification, but suffers from vanishing gradient issues.

Tanh: Similar to sigmoid but ranges from -1 to 1.

ReLU (Rectified Linear Unit): Popular for deep networks due to its efficiency.

Softmax: Used in the output layer for multi-class classification.

Backpropagation and Learning in MLP

- MLP is trained **using backpropagation**, which adjusts weights using gradient descent.
- The loss function (e.g., Mean Squared Error) helps in computing errors.
- The learning rate controls the step size for weight updates.

Advantages of MLP

✓ Can learn non-linear functions.

✓ Effective for both classification and regression problems.

✓ Works well with structured data.

Limitations of MLP

✗ Requires large amounts of data for training.

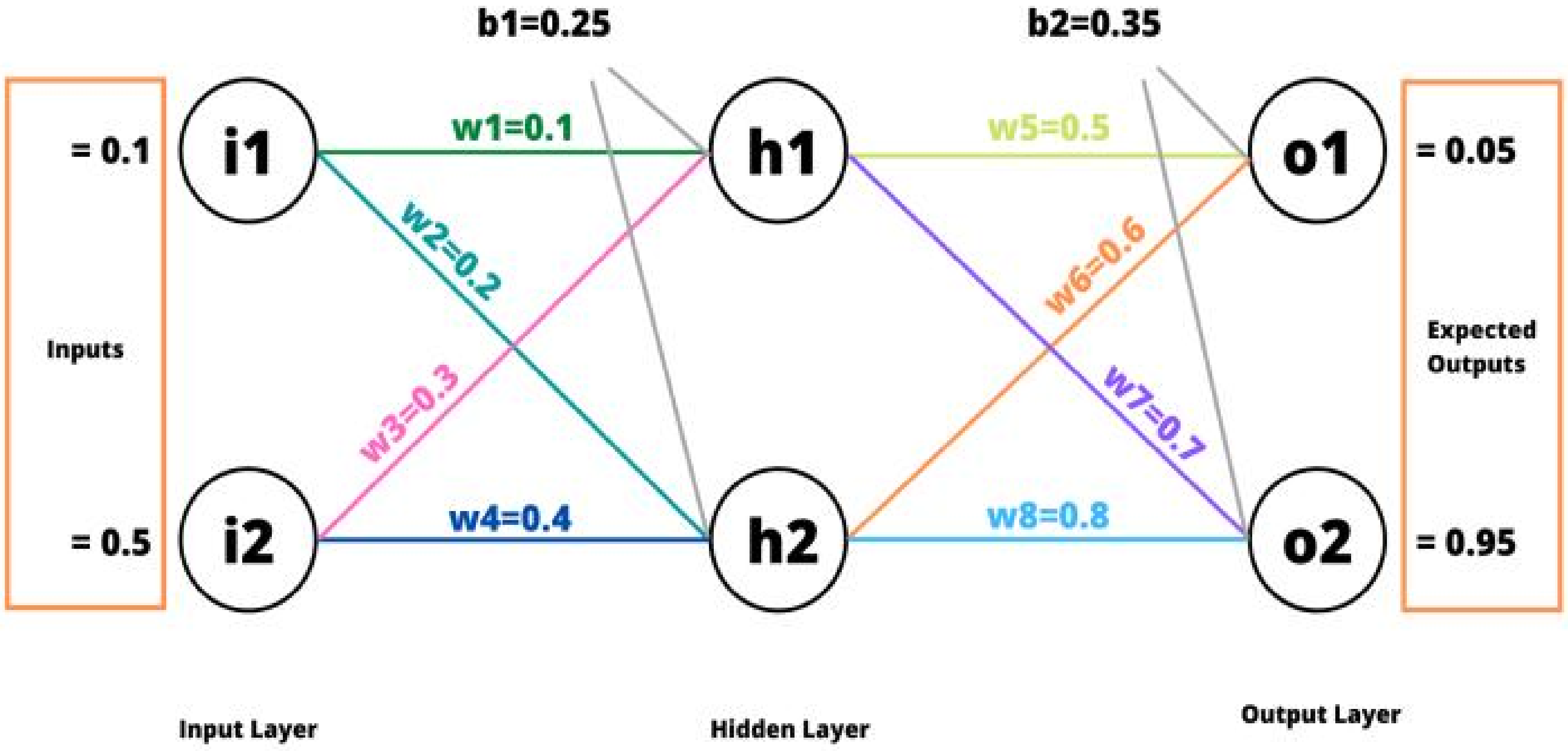
✗ Prone to overfitting if not regularized.

✗ Computationally expensive for deep networks.

Back propagation Algo

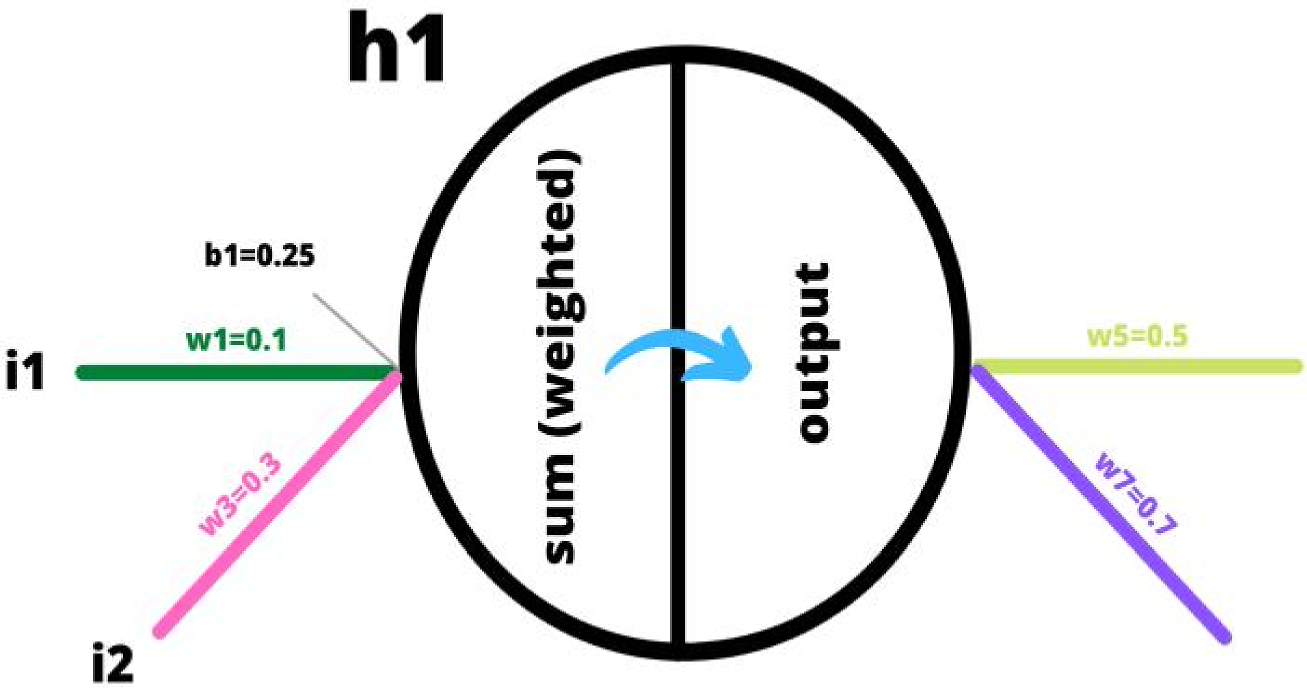
There are two units in the Input Layer, two units in the Hidden Layer and two units in the Output Layer.

The w1,w2,w2,…,w8 represent the respective weights. b1 and b2 are the biases for Hidden Layer and



we'll be passing two inputs i1 and i2, and perform a forward pass to compute total error

Then a backward pass to distribute the error inside the network and update weights accordingly.



- Computation of weighted sum
- Squashing of the weighted sum using an activation function.
- The result from the activation function becomes an input to the next layer (until the next layer is an Output Layer).
- In this example, we'll be using the Sigmoid function (Logistic function) as the activation function. The Sigmoid function basically takes an input and squashes the value between 0 and +1.

Chain Rule in Calculus

If we have $y = f(u)$ and $u = g(x)$ then we can write the derivative of y as:

$$\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx}$$

Neural Network Design and Forward Propagation Calculation

Design a feedforward neural network with:

Input Layer: 2 neurons (x_1 = 0.05, x_2 = 0.10)
Hidden Layer: 2 neurons (h_1, h_2)
Output Layer: 2 neurons (o_1, o_2)

Given Weights and Biases:

Weights from Input to Hidden Layer:
w_1 = 0.15, w_2 = 0.20 (Connected to h_1)
w_3 = 0.25, w_4 = 0.30 (Connected to h_2)

Bias for Hidden Layer: b_1 = 0.35

Weights from Hidden to Output Layer:
w_5 = 0.40, w_6 = 0.45 (Connected to o_1)
w_7 = 0.50, w_8 = 0.55 (Connected to o_2)

Bias for Output Layer: b_2 = 0.60

Tasks:

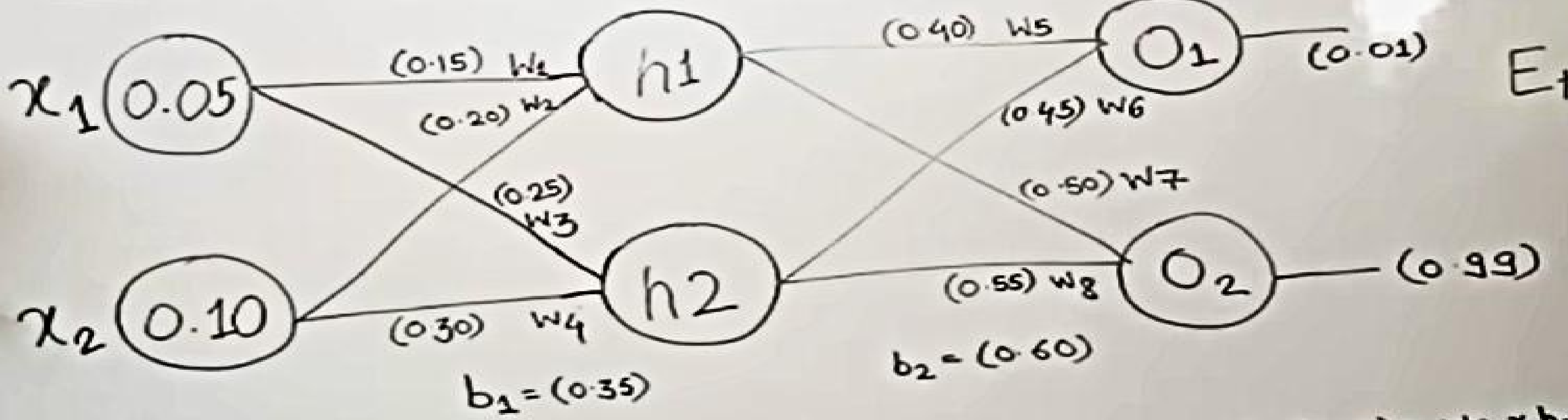
Calculate the net input and activation for each hidden neuron (h_1 and h_2) using the

sigmoid activation function:

Compute the net input and activation for the output neurons (o_1 and o_2) using the same activation function.

Determine the final output values for o_1 and o_2 after forward propagation.

Backpropagation / Backward Propagation of error



$E_{total} = EO_1 + EO_2$
$= 0.2983$

$h_1(in) = w_1 \times x_1 + w_2 \times x_2 + b_1$
$= (0.15 \times 0.05 + 0.2 \times 0.1 + 0.35)$
$= 0.377$

$h_1(out) = \frac{1}{1+e^{-h(in)}} = 0.5932$

$h_2(out) = 0.5968$

$O_1(in) = w_5 \times h_1(out) + w_6 \times h_2(out) + b_2$
$= (0.4 \times 0.593 + 0.45 \times 0.596 + 0.6)$
$= 1.105$

$O_1(out) = \frac{1}{1+e^{-O_1(in)}} = 0.7513$

$O_2(out) = 0.7729$

$E_{Total} = \sum \frac{1}{2}(\text{target} - o/p)^2$

$EO_1 = 0.274$ $EO_2 = 0.0235$

$E_{total} = 0.298371109$

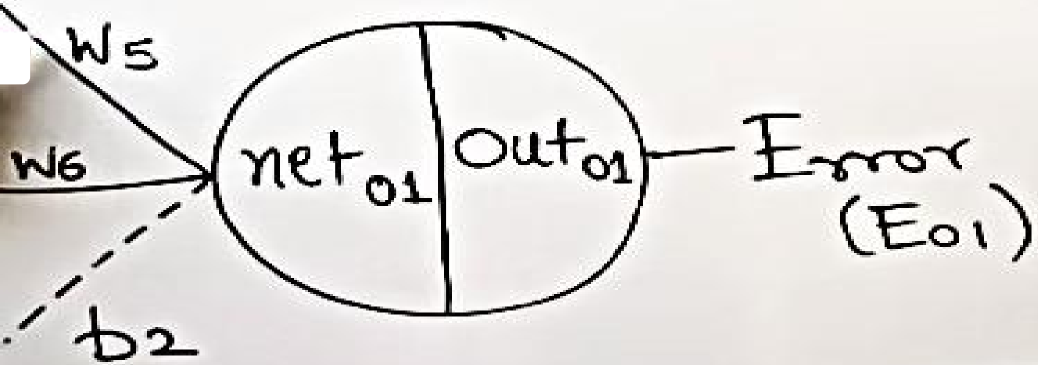
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

- $\frac{\partial E_{total}}{\partial out_{o1}} = Out_{o1} - Target_{o1}$
  $= 0.751365 - 0.01$
  $= 0.741365$

- $\frac{\partial out_{o1}}{\partial net_{o1}} = Out_{o1}(1 - Out_{o1})$
  $= 0.751365(1 - 0.751365)$
  $= 0.186815602$

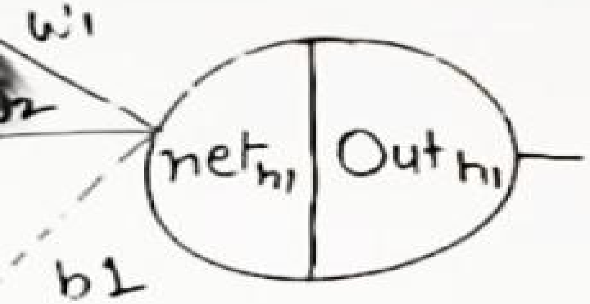
- $\frac{\partial net_{o1}}{\partial w_5} = Out_{h1} = 0.593269992$

- $\frac{\partial E_{total}}{\partial w_5} = 0.08216704$



$$W_5^* = W_5 - \alpha \times \frac{\partial E_{total}}{\partial W_5} = 0.4 - 0.6 * 0.08216704$$
$$= 0.350699776$$

Hidden layer



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{o2}}{\partial out_{o2}} = (Out_{o1} - target_{o1})$$

$$= 0.772928465 - 0.99$$

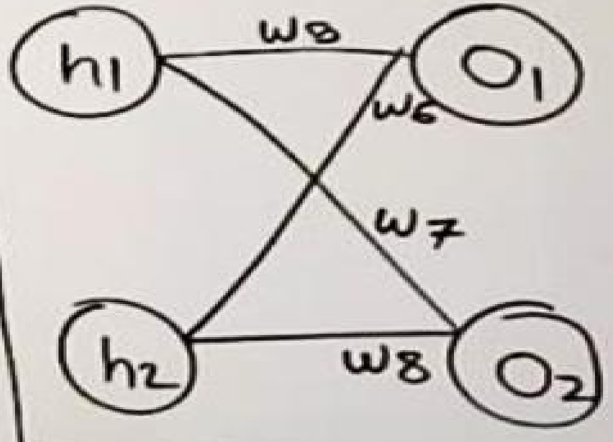
$$\boxed{= -0.217071535}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial net_{o2}} * \frac{\partial net_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} \qquad w_5 \qquad \frac{\partial E_{o2}}{\partial out_{o2}} * \frac{\partial out_{o2}}{\partial net_{o2}} \qquad w_7$$

$$\boxed{0.13849856} \qquad \boxed{0.4} \qquad \boxed{-0.0380982} \qquad \boxed{0.50}$$



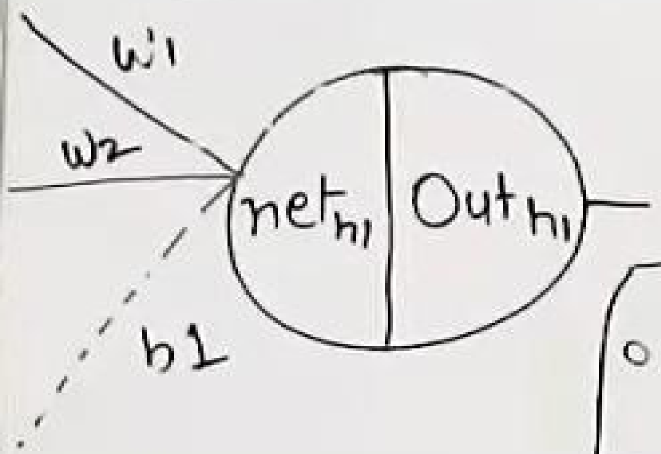
$$\Rightarrow 0.055399425 + (-0.019049119)$$

$$= 0.036350306$$

$$Out_{o1}(1 - Out_{o1})$$

$$= 0.175510052$$

Hidden layer



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

- $$\frac{\partial out_{h_1}}{\partial net_{h_1}} = out_{h_1}(1 - out_{h_1}) = 0.241300709$$

- $$net_{h_1} = w_1 x_1 + w_2 x_2 + b_1 \times 1$$
  $$\frac{\partial net_{h_1}}{\partial w_1} = x_1 = 0.05$$

$$\boxed{\frac{\partial E_{total}}{\partial w_1} = 0.000438568}$$

- $$w_1^* = w_1 - \alpha * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.6 * 0.000438568$$
  $$= 0.1497368592$$

1.1 Compute Net Input to Hidden Layer

The net input to each hidden neuron is calculated as:

$$net_{h1} = x_1 w_1 + x_2 w_2 + b_1$$

$$net_{h2} = x_1 w_3 + x_2 w_4 + b_1$$

Given:

- $x_1 = 0.05, x_2 = 0.10$
- $w_1 = 0.15, w_2 = 0.20, w_3 = 0.25, w_4 = 0.30$
- $b_1 = 0.35$

$$\begin{aligned} net_{h1} &= (0.05 \times 0.15) + (0.10 \times 0.20) + 0.35 \\ &= 0.0075 + 0.020 + 0.35 = 0.3775 \\ net_{h2} &= (0.05 \times 0.25) + (0.10 \times 0.30) + 0.35 \\ &= 0.0125 + 0.030 + 0.35 = 0.3925 \end{aligned}$$

1.2 Apply Activation Function (Sigmoid)

The activation function is the **Sigmoid function**:

$$h_i = \sigma(net_h) = \frac{1}{1 + e^{-net_h}}$$

For h_1:

$$h_1 = \frac{1}{1 + e^{-0.3775}} = 0.59327$$

For h_2:

$$h_2 = \frac{1}{1 + e^{-0.3925}} = 0.59689$$

1.3 Compute Net Input to Output Layer

The net input to each output neuron is calculated as:

$$net_{o1} = h_1 w_5 + h_2 w_6 + b_2$$

$$net_{o2} = h_1 w_7 + h_2 w_8 + b_2$$

Given:

- $w_5 = 0.40, w_6 = 0.45, w_7 = 0.50, w_8 = 0.55$
- $b_2 = 0.60$

$$net_{o1} = (0.59327 \times 0.40) + (0.59689 \times 0.45) + 0.60$$

$$= 0.23731 + 0.26860 + 0.60 = 1.1060$$

$$net_{o2} = (0.59327 \times 0.50) + (0.59689 \times 0.55) + 0.60$$

$$= 0.29664 + 0.32829 + 0.60 = 1.2249$$

1.4 Apply Activation Function (Sigmoid)

For o_1:

$$o_1 = \frac{1}{1 + e^{-1.1060}} = 0.75136$$

For o_2:

$$o_2 = \frac{1}{1 + e^{-1.2249}} = 0.77293$$

Step 2: Compute Error

Given target outputs:

- $t_1 = 0.01$, $t_2 = 0.99$

$$E = \frac{1}{2}[(t_1 - o_1)^2 + (t_2 - o_2)^2]$$

$$E = \frac{1}{2}[(0.01 - 0.75136)^2 + (0.99 - 0.77293)^2]$$

$$E = \frac{1}{2}[(0.74136)^2 + (0.21707)^2]$$

$$E = \frac{1}{2}[0.54961 + 0.04714] = \frac{0.59675}{2} = 0.2984$$

Step 3: Compute Gradients Using Backpropagation

Using **gradient descent**, we update each weight using:

$$w_{new} = w - \eta \frac{\partial E}{\partial w}$$

where **learning rate** $\eta = 0.5$.

3.1 Compute Gradients for Output Layer Weights

Using the chain rule:

$$\frac{\partial E}{\partial w_5} = \delta_{o1} \cdot h_1$$

$$\frac{\partial E}{\partial w_6} = \delta_{o1} \cdot h_2$$

where:

$$\delta_{o1} = (o_1 - t_1) \cdot o_1(1 - o_1)$$

$$= (0.75136 - 0.01) \cdot 0.75136(1 - 0.75136)$$

$$= 0.74136 \cdot 0.18681 = 0.1385$$

Similarly, for o_2:

$$\delta_{o2} = (o_2 - t_2) \cdot o_2(1 - o_2)$$

$$= (0.77293 - 0.99) \cdot 0.77293(1 - 0.77293)$$

$$= -0.21707 \cdot 0.17529 = -0.03807$$

Updating w_5, w_6, w_7, w_8:

$$w_5' = w_5 - \eta(\delta_{o1} \cdot h_1) = 0.40 - 0.5(0.1385 \times 0.59327) = 0.359$$

$$w_6' = w_6 - \eta(\delta_{o1} \cdot h_2) = 0.45 - 0.5(0.1385 \times 0.59689) = 0.4087$$

$$w_7' = w_7 - \eta(\delta_{o2} \cdot h_1) = 0.50 - 0.5(-0.03807 \times 0.59327) = 0.5113$$

$$w_8' = w_8 - \eta(\delta_{o2} \cdot h_2) = 0.55 - 0.5(-0.03807 \times 0.59689) = 0.5613$$

3.2 Compute Gradients for Hidden Layer Weights

$$\delta_{h1} = (\delta_{o1} w_5 + \delta_{o2} w_7) \cdot h_1(1 - h_1)$$

$$= (0.1385 \times 0.40 + (-0.03807) \times 0.50) \times 0.59327(1 - 0.59327)$$

$$= (0.0554 - 0.01903) \times 0.2413 = 0.0088$$

Updating w_1, w_2, w_3, w_4:

$$w_1' = w_1 - \eta(\delta_{h1} \times x_1) = 0.15 - 0.5(0.0088 \times 0.05) = 0.1498$$

$$w_2' = w_2 - \eta(\delta_{h1} \times x_2) = 0.20 - 0.5(0.0088 \times 0.10) = 0.1996$$

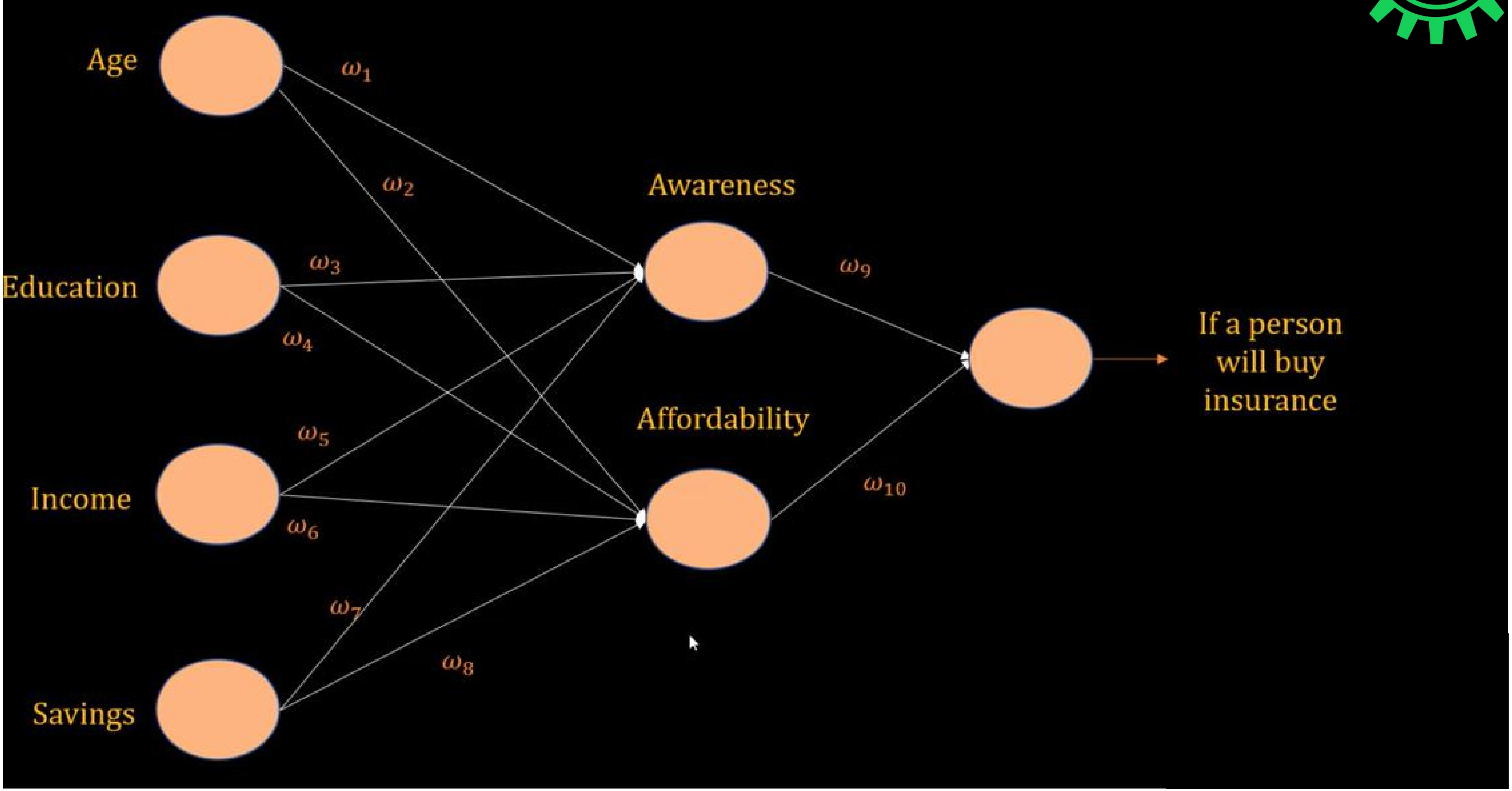
Similarly, update w_3 and w_4.

Final Updated Weights

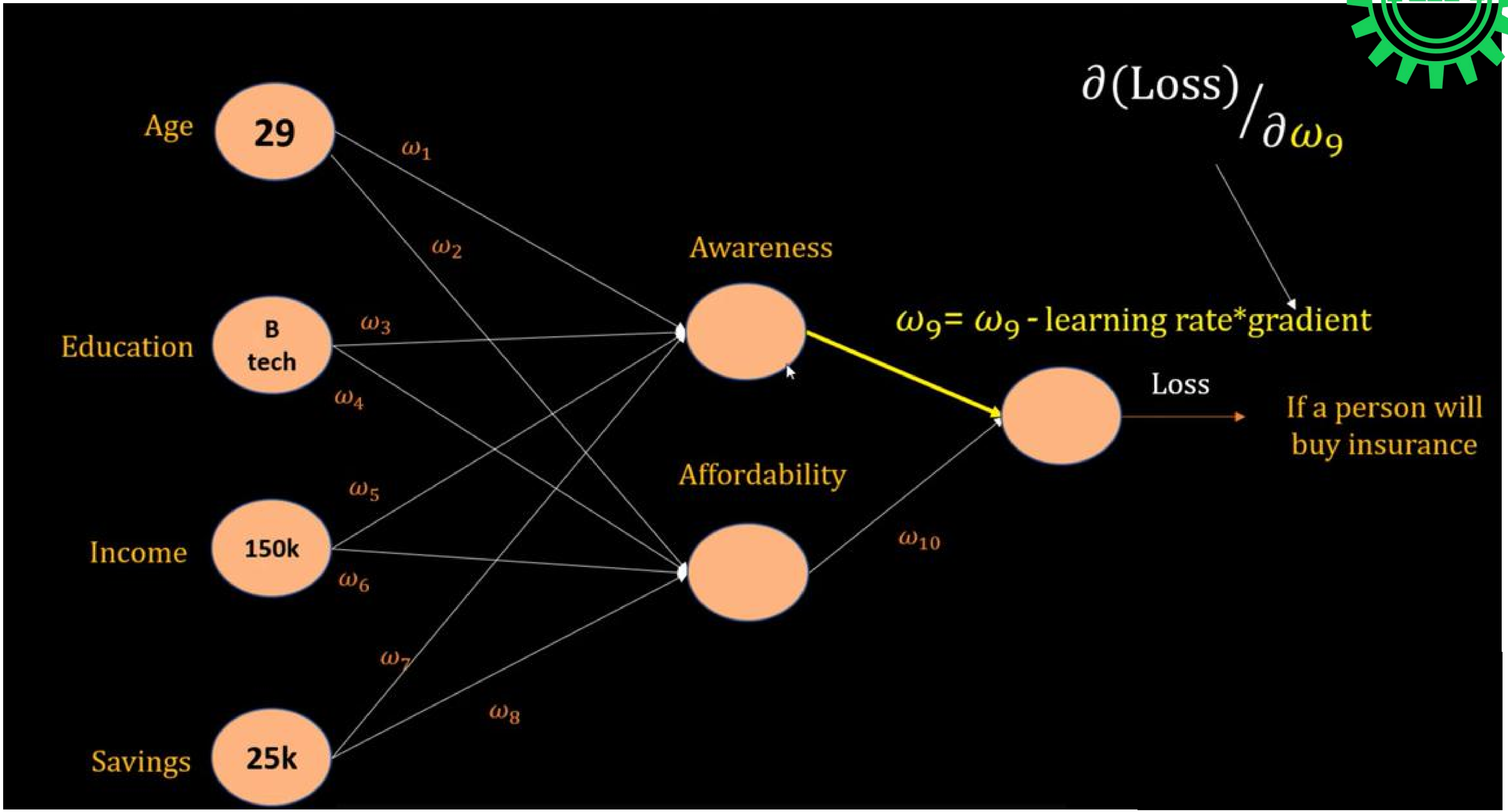
$$w'_1 = 0.1498, \quad w'_2 = 0.1996, \quad w'_3 = 0.2497, \quad w'_4 = 0.2995$$

$$w'_5 = 0.359, \quad w'_6 = 0.4087, \quad w'_7 = 0.5113, \quad w'_8 = 0.5613$$

Exploding Gradient Problem and Vanishing Gradient Problem

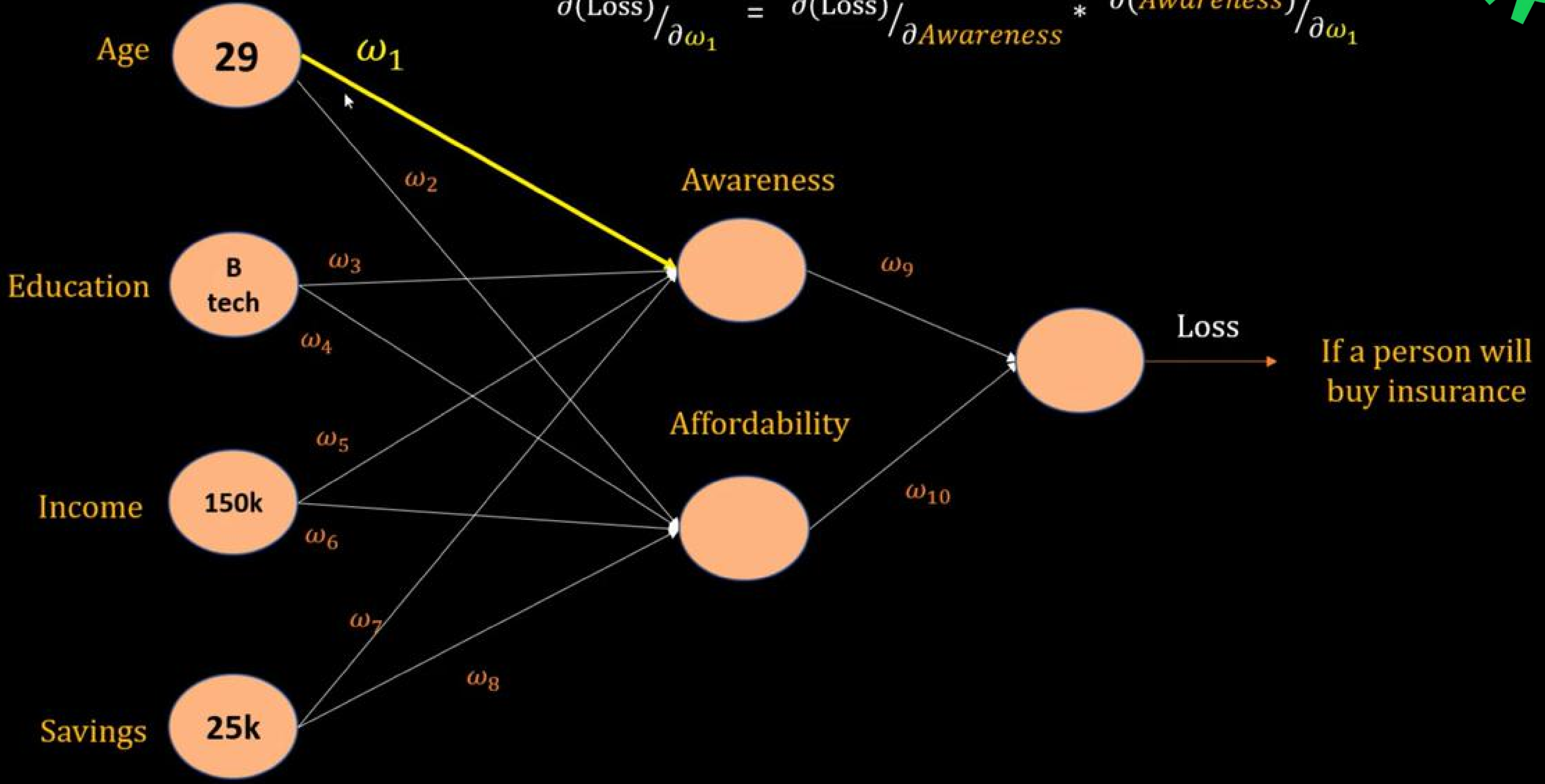


Age
Education
Income
Savings
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
ω_7
ω_8
Awareness
Affordability
ω_9
ω_{10}
If a person
will buy
insurance



$\partial(\text{Loss}) / \partial \omega_9$
Age
29
Education
B tech
Income
150k
Savings
25k
ω_1 ω_2 ω_3 ω_4 ω_5 ω_6 ω_7 ω_8 ω_{10}
Awareness
Affordability
$\omega_9 = \omega_9$ - learning rate*gradient
Loss
If a person will buy insurance

$$\partial(\text{Loss})/\partial\omega_1 = \partial(\text{Loss})/\partial Awareness * \partial(Awareness)/\partial\omega_1$$



$$\partial(\text{Loss}) / \partial \omega_1 = \partial(\text{Loss}) / \partial Awareness * \partial(Awareness) / \partial \omega_1$$

$$gradient = d1 * d2$$

$$gradient = 0.03 * 0.05$$

$$gradient = 0.0015$$

As number of hidden layers grow, gradient becomes very small and weights will hardly change . This will hamper the learning process.

Vanishing Gradients

$$\partial(\text{Loss})/\partial\omega_1 = \partial(\text{Loss})/\partial Awareness * \partial(Awareness)/\partial\omega_1$$

$$gradient = d1 * d2$$

$$gradient = 100 * 500$$

$$gradient = 50000$$

more bigger

$$gradient = d1 * d2 * d3 * d4 * \ldots * dn$$

Vanishing gradient problem is more prominent in very deep neural networks.

Regularization for Deep Learning

- Regularization in deep learning refers to a set of techniques used to prevent overfitting by adding constraints or penalties to the learning process.
- Overfitting occurs when a model performs well on training data but fails to generalize to unseen data.

Why is Regularization Needed?

- Deep neural networks are powerful but highly flexible, meaning they can easily memorize training data, especially when the model is large or the dataset is small.
- Regularization helps the model generalize better by discouraging it from learning overly complex or noisy patterns.

Common Regularization Techniques

L1 and L2 Regularization (Weight Penalty)

- L1 (Lasso): Adds a penalty equal to the absolute value of the weights.
- L2 (Ridge): Adds a penalty equal to the square of the weights (most common in deep learning).

Effect: Prevents weights from becoming too large; encourages simpler models.

Loss Function Example with L2 Regularization:

$$\text{Loss} = \text{Original Loss} + \lambda \sum_i w_i^2$$

where λ is the regularization strength.

Dropout

- Randomly drops (sets to zero) a fraction of neurons during training.
- Forces the network to not rely on any one feature or path, encouraging redundancy and robustness.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(512, activation='relu', input_shape=(784,)),
    Dropout(0.5),  # Drop 50% of neurons during training
    Dense(10, activation='softmax')
])
```

Technique	Key Idea	Helps With
L1/L2 Regularization	Penalize large weights	Weight control
Dropout	Randomly deactivate neurons	Model robustness
Early Stopping	Stop before overfitting	Training control
Data Augmentation	Generate more data variations	Generalization
Batch Norm	Normalize activations	Stabilizing training
Label Smoothing	Soften targets	Prevent overconfidence

Optimization for Training Deep Models: SGD vs Adam

Why Optimization is Needed?

When training a deep learning model, we want to minimize a loss function. Optimization algorithms help adjust the model's weights to reduce the loss during training.

Stochastic Gradient Descent (SGD)

- Updates model weights using gradient of the loss function.
- Instead of computing gradient over the entire dataset (as in batch gradient descent), it uses one data point (or a small batch) at a time → faster updates.

Weight Update Rule:

$$w = w - \eta \cdot \nabla L(w)$$

w: weights

η: learning rate

∇L(w): gradient of the loss w.r.t weights

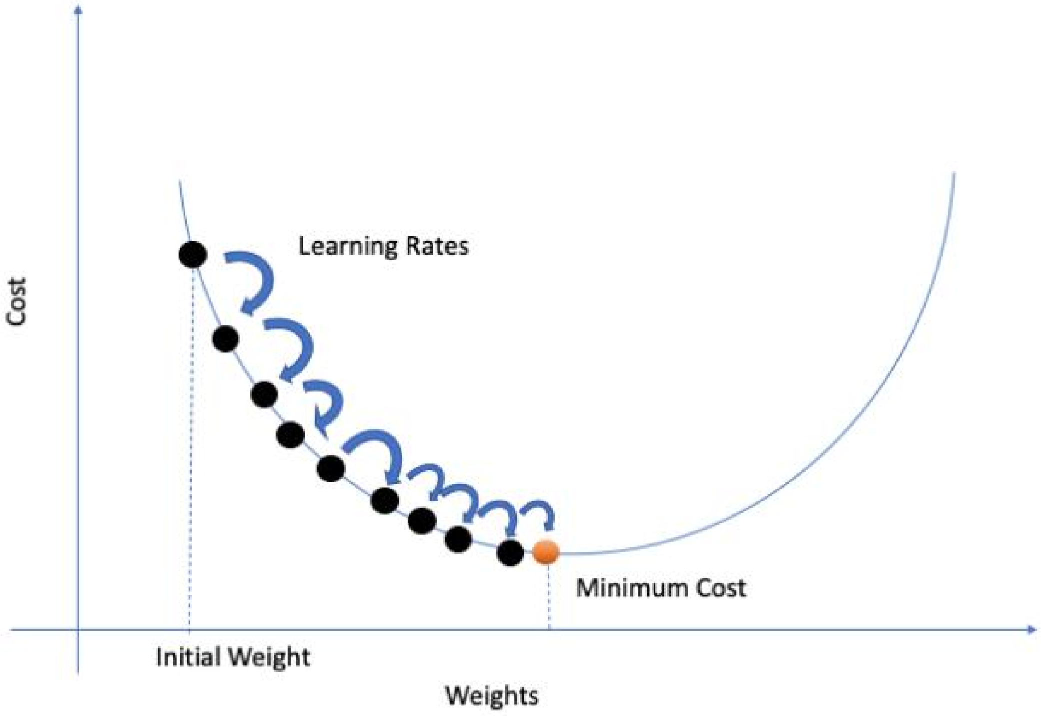
```
model = ...  # your neural network
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

How Adam Optimizer Works

Imagine you're hiking down a mountain (you want to reach the lowest point — this is like minimizing loss in a neural network).

You don't have a map. You only know how steep the slope is at your feet — this is like getting the gradient (direction of error).

Now, you want to reach the bottom smartly, not too fast (you'll fall!) and not too slow (you'll never reach).



1-Gradient tells you which way to go

The gradient tells us how much the output (loss) is changing. Like, how steep the slope is.
→ If it’s steep, we move fast. If it’s flat, we slow down.

2-Remember Past Steps (Momentum)

Adam remembers how the gradient has been changing.

If we’ve been going downhill for a while, let’s go faster (like rolling downhill).

This is called "momentum" (just like a ball rolling gathers speed).

3-Adjust Step Size (Learning Rate) Automatically

Adam also watches how much the gradient is shaking or bouncing.

If it’s bouncing a lot → take smaller steps (careful!).

If it's smooth → take bigger steps.

This is called adaptive learning rate.

4-Combine Both Smart Tricks

Adam combines both:

Moving average of gradients (momentum)

Moving average of gradient squares (adaptive step size)

5-Take a Step

After computing all of this, Adam takes a new step in the right direction and updates the model’s weights.

6-Repeat

It repeats this process for each batch of training data until the model learns well.

Introduction to Convolutional Neural Networks (CNN)

What is a CNN?

A Convolutional Neural Network (CNN) is a deep learning architecture designed to process structured grid data, such as images.

CNNs are widely used in image classification, object detection, and facial recognition due to their ability to automatically extract meaningful features from images.

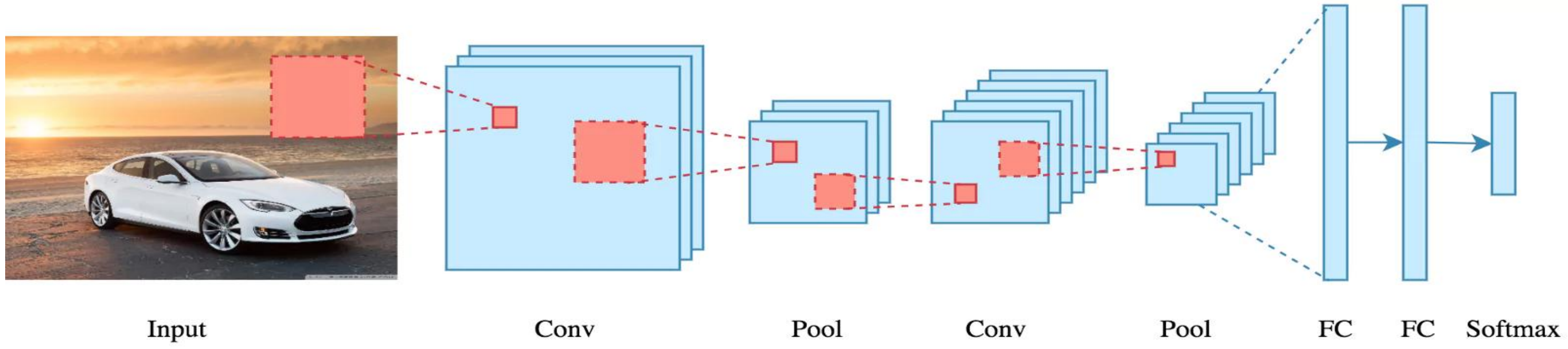
Components of CNN

Convolution Layer: Extracts features from the input image using filters (kernels).

Activation Function (ReLU): Introduces non-linearity to help learn complex patterns.

Pooling Layer (Max/Average Pooling): Reduces dimensionality while preserving important features.

Fully Connected Layer: Makes final predictions using extracted features.



Input Image (5×5 matrix)

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

Filter (3×3 Kernel)

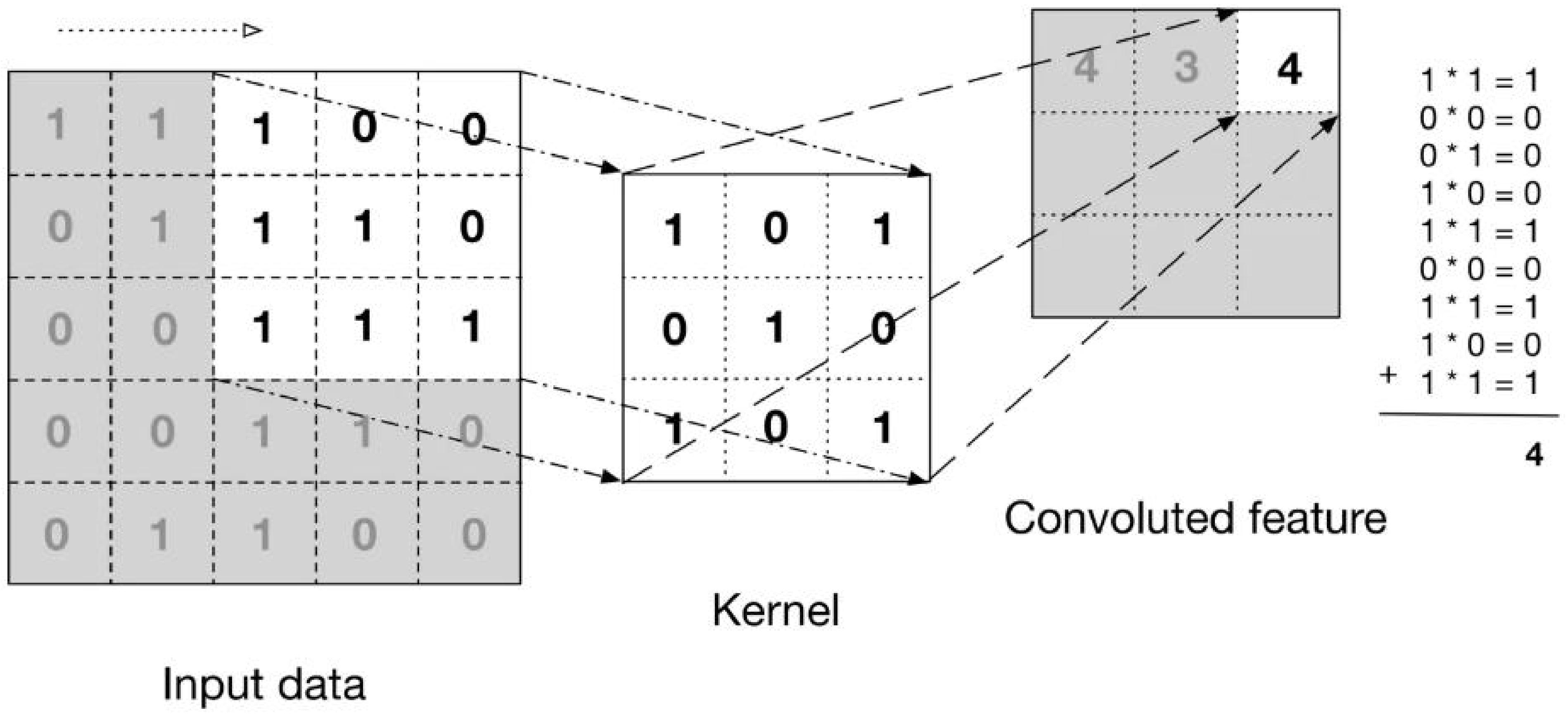
$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\rightarrow \boxed{\begin{bmatrix} -6 & -6 & -6 \\ -6 & -6 & -6 \\ -6 & -6 & -6 \end{bmatrix}}$$

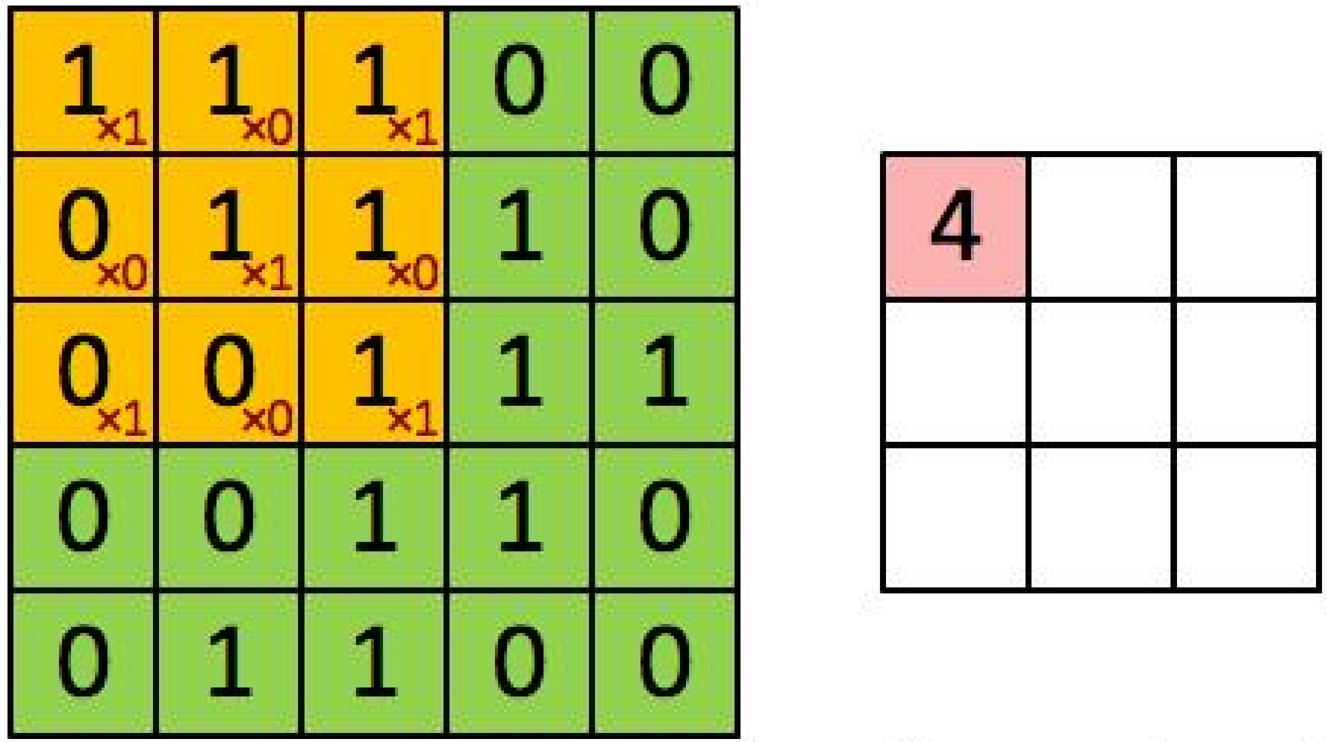
Multiply element-wise with the kernel:

$$(1 \times 1) + (2 \times 0) + (3 \times -1) + (4 \times 1) + (5 \times 0) + (6 \times -1) + (7 \times 1) + (8 \times 0) + (9 \times -1)$$

$$= 1 + 0 - 3 + 4 + 0 - 6 + 7 + 0 - 9 = -6$$

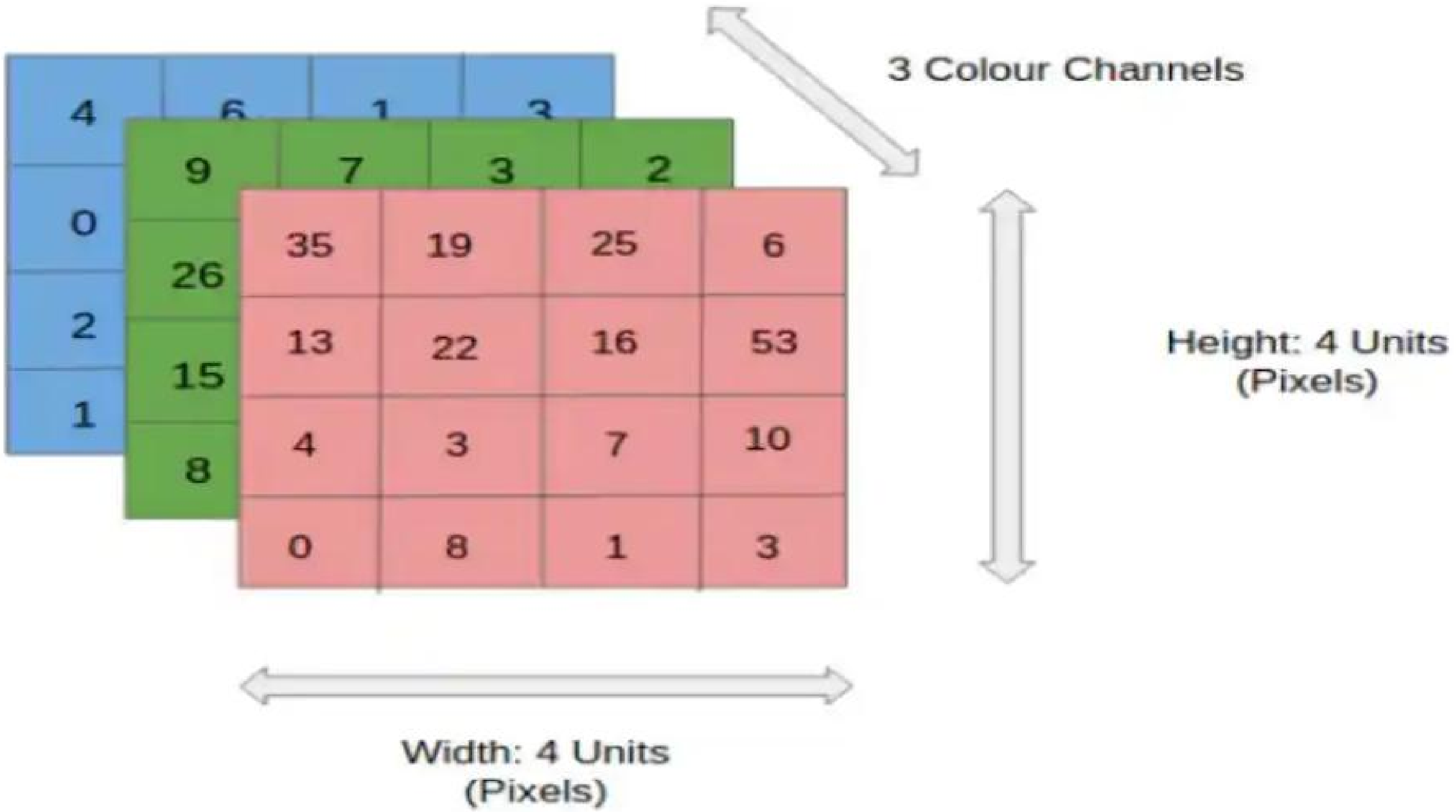


1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
Input data
1 0 1
0 1 0
1 0 1
Kernel
4 3 4
Convoluted feature
1 * 1 = 1
0 * 0 = 0
0 * 1 = 0
1 * 0 = 0
1 * 1 = 1
0 * 0 = 0
1 * 1 = 1
1 * 0 = 0
+ 1 * 1 = 1
4



Image

Convolved Feature



3 Colour Channels
Height: 4 Units
(Pixels)
Width: 4 Units
(Pixels)
4
6
1
3
0
2
1
9
7
3
2
26
15
8
35
19
25
6
13
22
16
53
4
3
7
10
0
8
1
3

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

$$308 \quad + \quad -498 \quad + \quad 164 \quad + 1 = -25$$

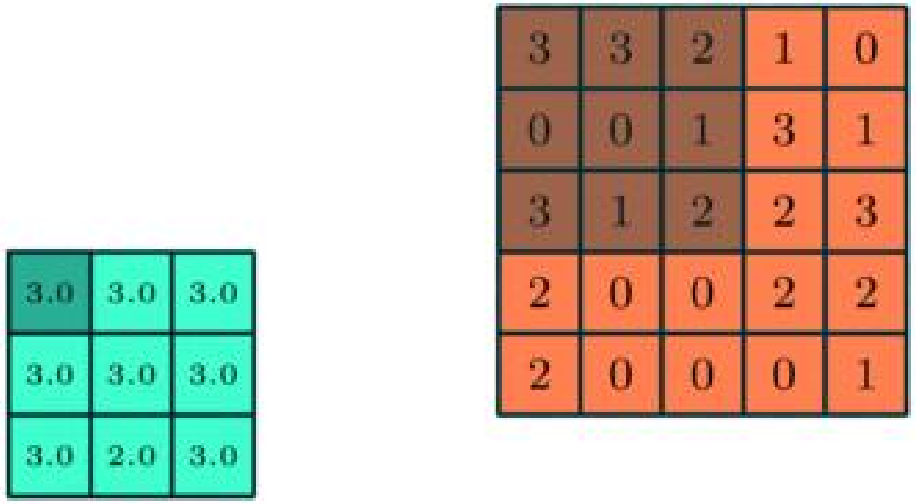
Bias = 1

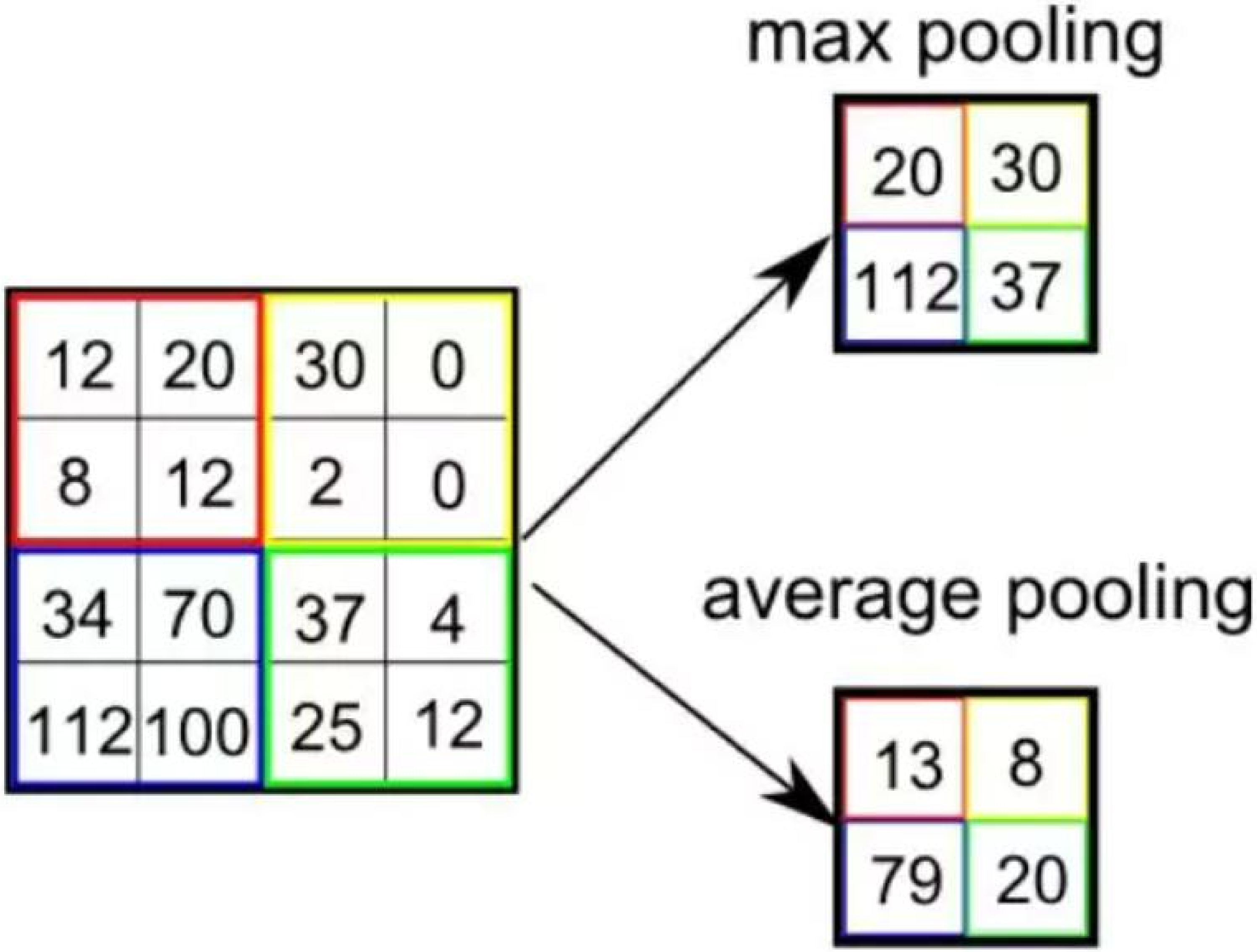
Output

-25				...
				...
				...
				...
...	...	...	...	...

What is a Pooling Layer?

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data by reducing the dimensions. There are two types of pooling average pooling and **max pooling**.




max pooling
average pooling
12
20
30
0
8
12
2
0
34
70
37
4
112
100
25
12
20
30
112
37
13
8
79
20

ReLU (Rectified Linear Unit) Activation Function in CNN

The ReLU (Rectified Linear Unit) activation function is one of the most commonly used activation functions in Convolutional Neural Networks (CNNs).

It introduces non-linearity into the model, helping CNNs learn complex patterns in images.

Mathematical Definition

$$f(x) = \max(0, x)$$

This means:

- If x is positive, $f(x) = x$ (it remains the same).
- If x is negative, $f(x) = 0$ (negative values are replaced with zero).

Example of ReLU in CNN

Consider a **3×3 convolution** operation applied to an image, followed by ReLU activation.

Step 1: Input Feature Map (Before ReLU)

$$\begin{bmatrix} -2 & 3 & -1 \\ 5 & -6 & 2 \\ -3 & 4 & -7 \end{bmatrix}$$

Step 2: Applying ReLU Activation

We replace all negative values with 0:

$$\begin{bmatrix} 0 & 3 & 0 \\ 5 & 0 & 2 \\ 0 & 4 & 0 \end{bmatrix}$$

Concept	Purpose	Example
Convolution	Feature extraction (edges, textures)	3×3 filter on image
Pooling	Dimensionality reduction	Max pool on 2×2 region
Stride	Control movement of filter	Stride 2 → skips 1 pixel
Padding	Keep size same after convolution	Pad image with zeros

Layer Type	Input Shape	Output Shape	Operation
Conv2D (6 filters, 5×5)	(28,28,1)	(24,24,6)	Feature Extraction
MaxPooling2D (2×2)	(24,24,6)	(12,12,6)	Downsampling
Conv2D (16 filters, 5×5)	(12,12,6)	(8,8,16)	Feature Extraction
MaxPooling2D (2×2)	(8,8,16)	(4,4,16)	Downsampling
Flatten	(4,4,16)	(256)	Converts to 1D
Dense (120)	(256)	(120)	Fully Connected Layer
Dense (84)	(120)	(84)	Fully Connected Layer
Dense (10)	(84)	(10)	Output (Classification)

Fourier Transform Convolution

The Fourier Transform breaks down a signal (like an image) into its frequency components.

Idea:

Instead of doing convolution in spatial domain (pixel-wise), we do it in frequency domain using the Fast Fourier Transform (FFT).

Why?

Convolution in time/spatial domain = Multiplication in frequency domain.

Much faster for large kernels and images.

Steps:

Convert input and kernel to frequency domain using FFT.

Multiply them element-wise.

Convert back to spatial domain using Inverse FFT.

What is FFT?

FFT = Fast Fourier Transform

- It’s a fast algorithm to compute the Discrete Fourier Transform (DFT).
- Converts data from spatial domain (pixels) to frequency domain.

So instead of dealing with pixel intensities, we analyze the rate of change (how quickly pixel values change — the frequency).

Images have:

- Low frequencies → smooth regions (sky, walls)
- High frequencies → edges, noise, fine textures

We use FFT to:

- Analyze these frequency patterns
- Apply filters (blur, edge detection)
- Compress or clean images

Separable Convolutions (Spatially Separable Convolutions)

A standard 2D convolution uses a 2D filter (e.g., 3×3) to convolve over the input image.

In a spatially separable convolution, we break a 2D filter into two 1D filters: one for rows and one for columns.

Instead of using a 3×3 kernel:

[a b c
d e f
g h i]

We approximate it with two filters:

A vertical 1D filter (3×1): [v1, v2, v3]

A horizontal 1D filter (1×3): [h1, h2, h3]

Advantages:

- Reduced computation (faster).
- Lower memory usage.
- Only works if the kernel is mathematically separable.

So, instead of doing 9 multiplications per pixel, you do 3 + 3 = 6 multiplications.

Standard Convolution:

Input: 32×32 image with 3 channels (RGB)

Kernel: 64 filters of size 3×3×3

Computation:
$3 \times 3 \times 3 \times 64 = 1728$ multiplications per output pixel

Depthwise Separable Convolution:
Step 1: Depthwise Convolution

3 filters (one for each channel)

Each filter: 3×3

Computation:
$3 \times 3 \times 3 = 27$

Step 2: Pointwise Convolution

1×1 convolutions

64 filters: each processes all 3 input channels

Computation:
$3 \times 1 \times 1 \times 64 = 192$

Total Computation: 27 + 192 = 219

🔹 1. Standard Convolution

Suppose we have:

- Input feature map size: $H \times W \times C_{in}$
- Kernel size: $K \times K$
- Number of filters: C_{out}

Then, for **each output feature map element**:

$$\text{Multiplications per output element} = K \times K \times C_{in}$$

Since there are $H \times W$ output locations and C_{out} filters:

$$\boxed{\text{Total multiplications (Standard Conv)} = H \times W \times C_{out} \times K \times K \times C_{in}}$$

2. Depthwise Convolution

In depthwise convolution, each input channel is convolved separately with its own filter.

- Each channel has a $K \times K$ filter.
- So per output element (per channel): $K \times K$ multiplications.

For all channels and all spatial locations:

$$\boxed{\text{Total multiplications (Depthwise Conv)} = H \times W \times C_{in} \times K \times K}$$

Example

Input feature map: $32 \times 32 \times 3$ (like a small RGB image)

Kernel size: 3×3

Number of filters: 64

So, output feature map = $32 \times 32 \times 64$

◆ Standard Convolution

Formula:

$$\text{Multiplications} = H \times W \times C_{out} \times K^2 \times C_{in}$$

Substitute values:

$$= 32 \times 32 \times 64 \times 3 \times 3 \times 3$$

$$= 32 \times 32 \times 64 \times 27$$

$$= 56,623,104 \ \ \text{multiplications}$$

Depthwise Convolution

Formula:

$$\text{Multiplications} = H \times W \times C_{in} \times K^2$$

Substitute values:

$$= 32 \times 32 \times 3 \times 3 \times 3$$

$$= 32 \times 32 \times 27$$

$$= 27,648 \;\; \text{multiplications}$$

Advantages:

- Faster and lightweight
- Great for mobile and embedded devices
- Used in MobileNet, EfficientNet, Xception

What is a Recurrent Neural Network (RNN)?

A **Recurrent Neural Network (RNN)** is a type of neural network designed to handle sequential data — data where the order matters, like:

- Words in a sentence
- Notes in a song
- Temperatures across days

Unlike traditional neural networks, RNNs have "memory" of past inputs.

“I love machine ___.”

You need to remember the words before "___" to guess what comes next. That’s exactly what an RNN does — it remembers previous words using its hidden state.

RNN Processing:

At each time step t, the RNN takes:

The input at that time step $\mathbf{x_t}$

The previous hidden state $\mathbf{h_{t-1}}$

And computes the new hidden state $\mathbf{h_t}$

$$\mathbf{h_t = \tanh(W_x x_t + W_h h_{t-1} + b)}$$

$\mathbf{W_x}$ **and** $\mathbf{W_h}$ are weights

tanh is the activation function

$\mathbf{h_0}$ is initialized to zero

Input sequence: "I am going to"

Predict: "school"

Here's how an RNN will work:

Input: "I" → Hidden state 1

Input: "am" + Hidden state 1 → Hidden state 2

Input: "going" + Hidden state 2 → Hidden state 3

Input: "to" + Hidden state 3 → Predict "school"

The RNN remembers the context using hidden states.

Simple Numerical Example:

Let's say:

Input: **[1, 2, 3]** (e.g., A=1, B=2, C=3)

Initial hidden state: **$h_0 = 0$**

Weight values:

$W_x = 1$, $W_h = 1$, $b = 0$

At time t = 1:

$x_1 = 1$, $h_0 = 0$

$h_1 = \tanh(1\times1 + 1\times0) = \tanh(1) \approx 0.761$

At time t = 2:

$x_2 = 2$, $h_1 \approx 0.761$

$h_2 = \tanh(1\times2 + 1\times0.761) = \tanh(2.761) \approx 0.992$

At time t = 3:

$x_3 = 3$, $h_2 \approx 0.992$

$h_3 = \tanh(1\times3 + 1\times0.992) = \tanh(3.992) \approx 0.999$

What is an LSTM Network?

LSTM (Long Short-Term Memory) is a special type of Recurrent Neural Network (RNN) that is capable of learning long-term dependencies. **It was designed to avoid the vanishing gradient problem found in traditional RNNs.**

LSTMs are widely used in time-series forecasting, NLP, speech recognition, and more.

Recurrent Neural Networks (RNNs) process sequences by maintaining a **hidden state** across time steps:

At each time step t, the hidden state is updated as:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b)$$

To train RNNs, we use **Backpropagation Through Time (BPTT)** — the gradient is computed from the **last time step backward to the first**.

Vanishing gradient problem found in traditional RNNs.

During training, we compute the **gradient of the loss** w.r.t weights by **chain rule**:

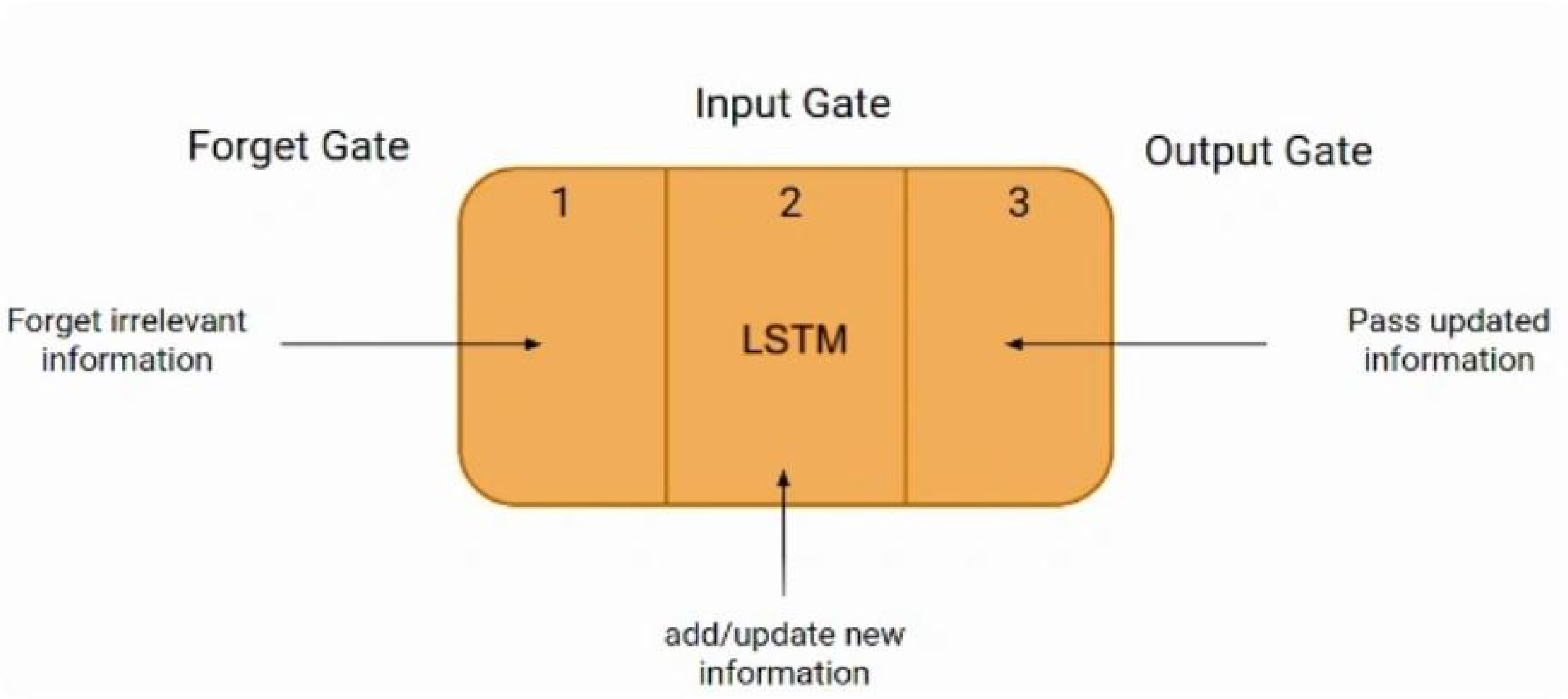
$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdot \ldots \cdot \frac{\partial h_k}{\partial W}$$

But notice:

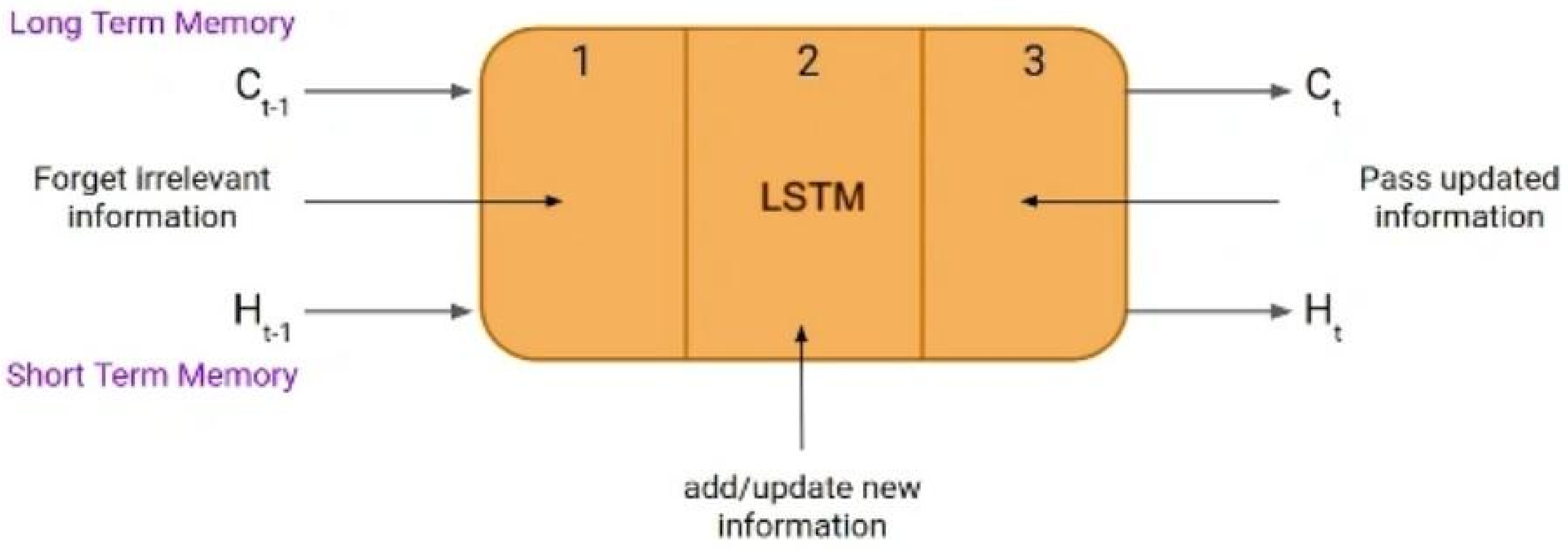
- Each $\frac{\partial h_t}{\partial h_{t-1}}$ is often **less than 1**.
- So, repeated multiplication of these gradients across **many time steps** makes them **shrink rapidly**.

$$\left(\frac{\partial h}{\partial h}\right)^T \approx 0 \quad \Rightarrow \quad \text{Gradients vanish!}$$

LSTM Architecture



LSTM Architecture



LSTM Architecture

Cell state: Keeps track of long-term memory.

Hidden state: Short-term output memory.

3 Gates:

Forget Gate (f_t) – decides what to forget.

Input Gate (i_t) – decides what to update.

Output Gate (o_t) – decides what to output.

Cell State: Like a conveyor belt carrying long-term memory.

Hidden State: Like short-term working memory.

Gates: Think of them like switches or filters that decide what to keep, add, or throw away.

Step 1: Forget Gate

Question: “What old memory should I forget?”

Example: If you're reading “**I am going to the market”**, and you’re at the word **"the"**, you may not need to remember "**I am"** anymore.

So, LSTM forgets irrelevant information from the past.

Step 2: Input Gate

Question: “What new information should I add?”

Now you’re reading a new word, like **"market"**.

LSTM checks whether this is important and decides to add it to memory.

Step 3: Update Memory (Cell State)

Action: Combine:

What you decided to forget (Step 1), and

What you decided to add (Step 2)

→ This becomes your updated long-term memory.

Step 4: Output Gate

Question: “What should I show as the result/output?”

Now, based on the updated memory, LSTM decides:

What should go out (to the next step),

And what should stay in.

This becomes your hidden state, and is used to predict the next word, or passed to the next LSTM cell.

Forget Gate

Purpose: Decide what information to remove from cell state.

Equation:

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

Explanation:

- Combine h_{t-1} (past short-term memory) and x_t (current input).
- Multiply by weight matrix W_f, add bias b_f.
- Pass through **sigmoid** (0 → forget, 1 → keep).

Input Gate

Purpose: Decide what new information to add to cell state.

Equations:

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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Explanation:

- i_t: Filter for which values to update (sigmoid).
- $\tilde{C}_t$: New candidate values (tanh).

Update Cell State

Equation:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Explanation:

- Multiply old cell state by forget gate output (removes unwanted info).
- Add candidate values multiplied by input gate output (adds new info).

Output Gate

Purpose: Decide the next hidden state (output).

Equations:

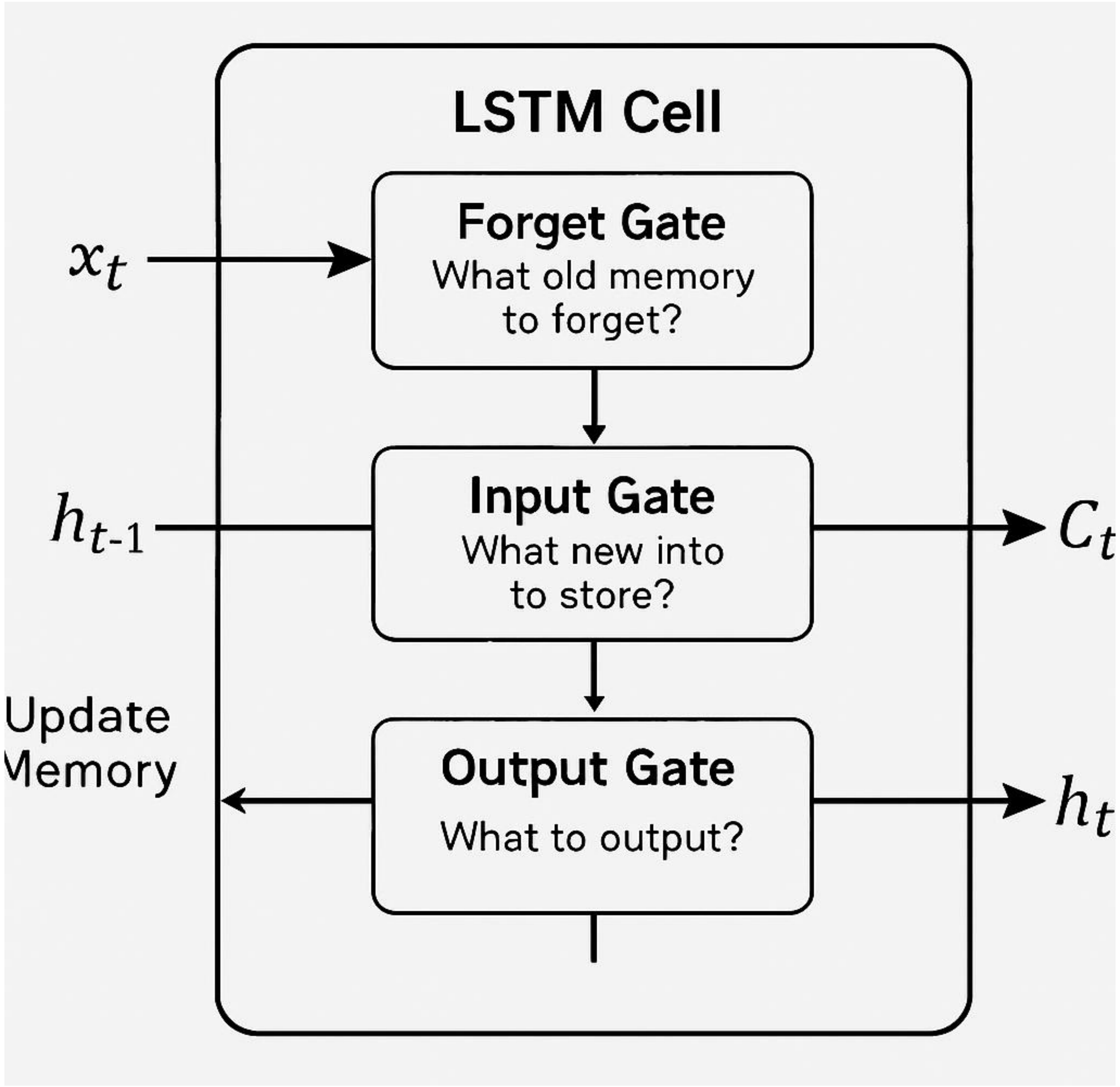
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Explanation:

- o_t filters what part of cell state becomes output.
- Final hidden state h_t is output gate × tanh of cell state.

Step	What it does	Why it matters
1. Forget Gate	Throws away unneeded old info	Keep memory clean
2. Input Gate	Adds new important info	Learn current input
3. Update Memory	Combines old and new info	Update understanding
4. Output Gate	Decides what to output	Use for next prediction


LSTM Cell
Forget Gate
What old memory to forget?
Input Gate
What new into to store?
Output Gate
What to output?
x_t
h_{t-1}
C_t
h_t
Update Memory