upGrad

Introduction to Neural Networks

upGrad

Agenda

1. Why neural networks are needed. Discuss how classical ML struggles with complex, non-linear data patterns
2. Understanding the Neuron
3. Network Architecture and Layers
4. Forward & Backward Propagation, Activation Function
5. Practical Intution from ML to NN

Poll Question

upGrad

Poll 1: Decision Boundaries
Question:
Which type of decision boundary can a basic linear classifier learn?
A) Circular
B) Arbitrary curved
C) Straight line (hyperplane)
D) Multi-layer hierarchical

Poll 2: Feature Engineering
Question:
Why do classical ML models rely heavily on feature engineering?
A) They automatically learn deep features
B) They cannot handle raw data effectively
C) They perform better with large datasets
D) They are inherently non-linear

Poll 3: Image Recognition
Question:
Why do classical ML models struggle with image classification?
A) Images are too small
B) Pixels are independent
C) Images have high dimensional, non-linear structure
D) Classical models are too fast

Poll 4: Curse of Dimensionality
Question:
What happens to distance-based models (like k-NN) as dimensions increase?
A) They become more accurate
B) Distances become less meaningful
C) Computation becomes faster
D) Overfitting disappears

Poll 5: Learning Representations
Question:
Which limitation prevents classical ML from learning complex abstractions?
A) Lack of data
B) Lack of compute
C) Shallow model structure
D) Too many hyperparameters

Poll 6: Non-Linear Data
Question:
Which classical ML approach attempts to handle non-linearity but still has limits?
A) Logistic Regression
B) k-Means Clustering
C) Kernel SVM
D) Naive Bayes

Poll 1. C Poll 2. B Poll 3. C Poll 4. B Poll 5. C Poll 6. C

Why we need Neural Networks?

upGrad

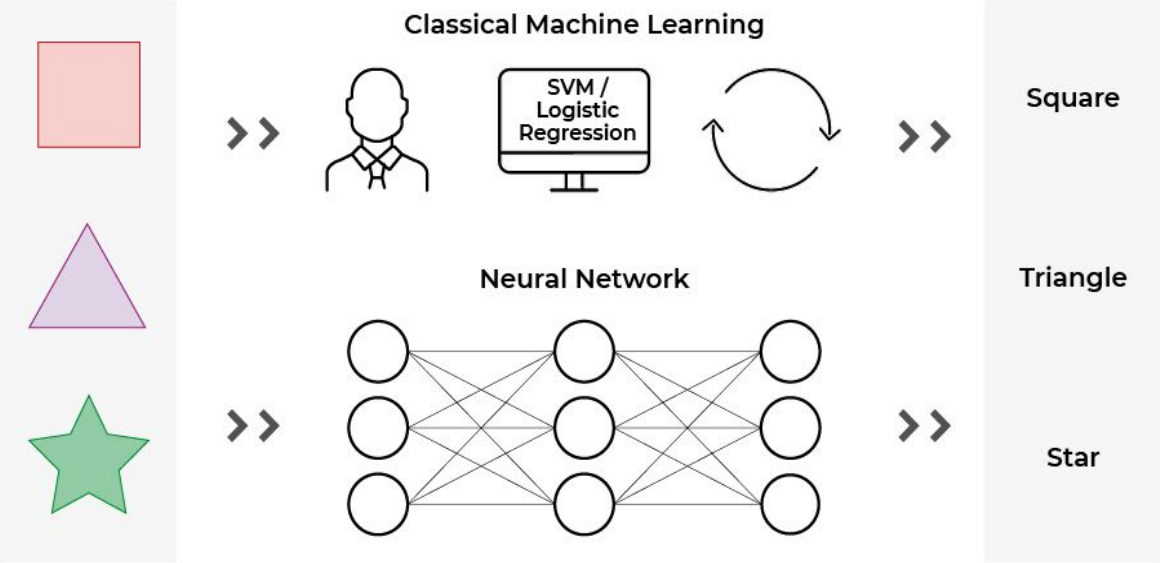
Neural networks were developed to overcome the fundamental limitations of traditional machine learning models when dealing with **complex, real-world data**. As data grew larger, noisier, and more unstructured, classical approaches became insufficient.

What Classical ML Models Do

Traditional machine learning includes models like:

- **Linear Regression / Logistic Regression**
- **Support Vector Machines (SVM)**
- **Decision Trees / Random Forests**
- **k-Nearest Neighbors (kNN)**

These models map **input features to outputs** using relatively simple mathematical relations. They work well when the pattern in the data is **simple or can be made simple with feature engineering** (e.g., transformations, polynomial features).



However:

- Many real-world problems *involve complicated relationships* that can't be neatly described by simple math formulas.
- Data like **images, text, and speech** have structure that's difficult to capture with shallow rules or summary statistics

Limitation: Linear Decision Boundaries

For example, a **single linear classifier** like logistic regression can only draw a straight line (or a hyperplane) to separate classes:

- Works if classes are **linearly separable**
- Fails if the boundary is **curved, tangled, or higher-order**

Even enhanced models (e.g., polynomials or kernels) require careful *manual feature design* and still struggle on truly complex patterns

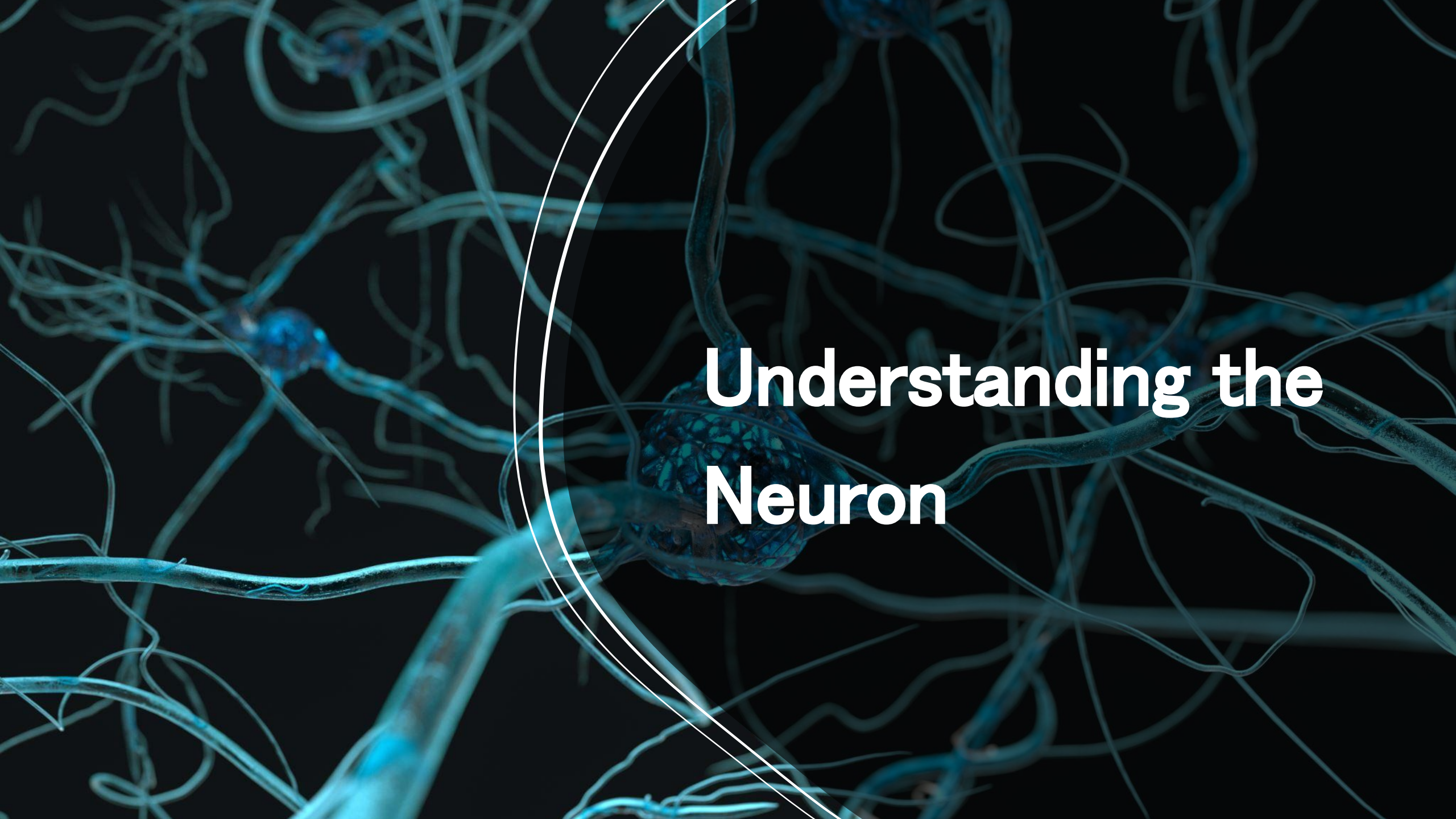
Non-Linear Complexity and Real Data

✨ **Real Data Is Hard**

In many domains, data relationships are **not linear**:

- Image pixels interact in hierarchical and spatial ways.
- Speech and language reflect temporal and contextual patterns.
- Signals and sensor data have cyclic or chaotic patterns.

Traditional models can only approximate non-linear behavior if you **pre-engineer features** to capture these effects. That's labor-intensive and often not effective enough for high-complexity tasks.



Understanding the Neuron

Artificial Neuron

upGrad

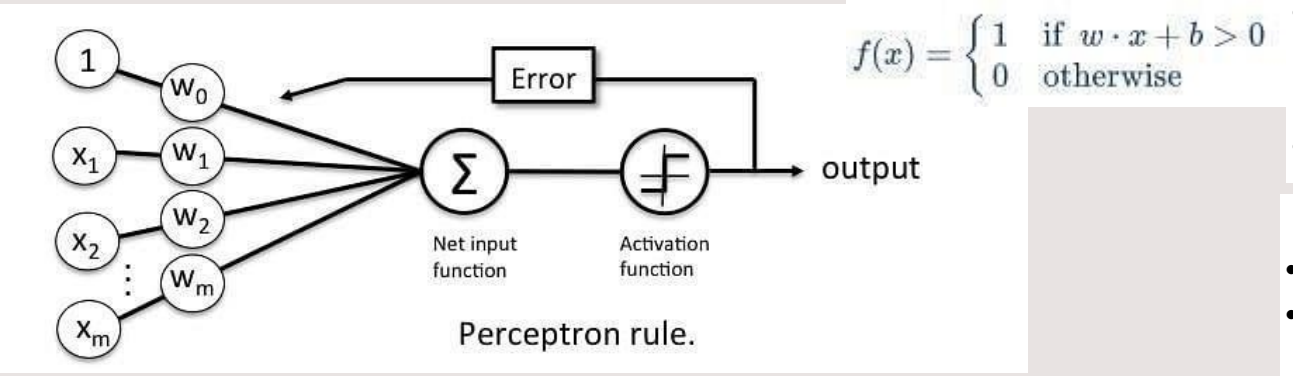
Researchers Warren McCullock and Walter Pitts published their first concept of simplified brain cell in 1943. This was called McCullock-Pitts (MCP) neuron. They described such a nerve cell as a simple logic gate with binary outputs.

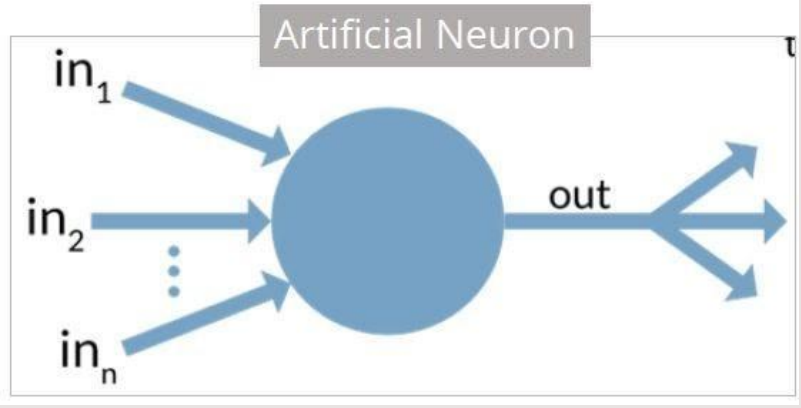
Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers.

Neuron and Perceptron

This algorithm enables neurons to learn and processes elements in the training set one at a time.

An artificial neuron is a mathematical function based on a model of biological neurons, where each neuron takes inputs, weighs them separately, sums them up and passes this sum through a nonlinear function to produce output.





Structure of Neuron

1. Inputs (x_1, x_2, …, x_n)

- Inputs represent **features** or signals coming from data
- Example:
  - In image recognition: pixel values
  - In spam detection: word frequencies
- Each input carries **information** to the neuron

2. Weights (w_1, w_2, …, w_n)

- Each input is multiplied by a **weight**
- Weights represent the **importance** of each input
- Larger weight → stronger influence
- Negative weight → inhibitory effect

3. Weighted Sum

The neuron computes a linear combination:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

This represents the total input signal received by the neuron.

Artificial Neuron

upGrad

4. Bias (b)

- Bias is an **additional adjustable parameter**
- It shifts the activation function left or right
- Allows the neuron to activate even when inputs are zero

Updated equation:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

5. Activation Function (f)

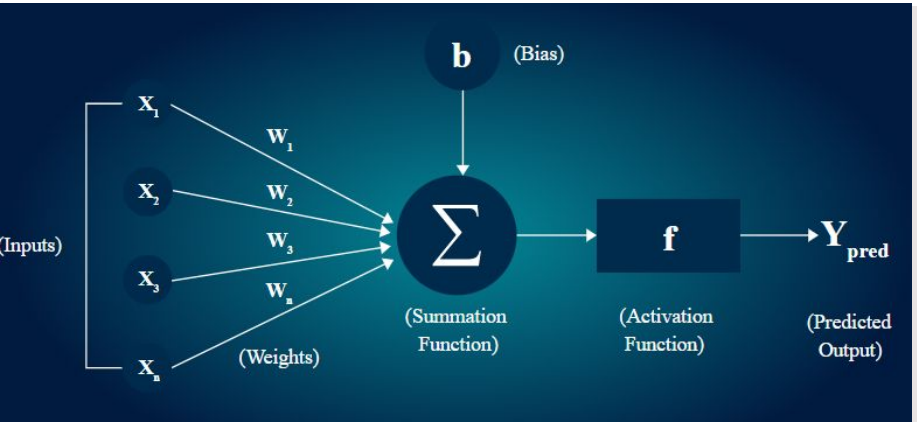
- The activation function introduces **non-linearity**
- Determines whether and how strongly the neuron "fires"

Common activation functions:

- **Step function** (early perceptron)
- **Sigmoid** – outputs values between 0 and 1
- **ReLU** – outputs max(0, z)
- **Tanh**

Final output:

$$\text{output} = f(z)$$



Neural networks are the building blocks of deep learning systems.

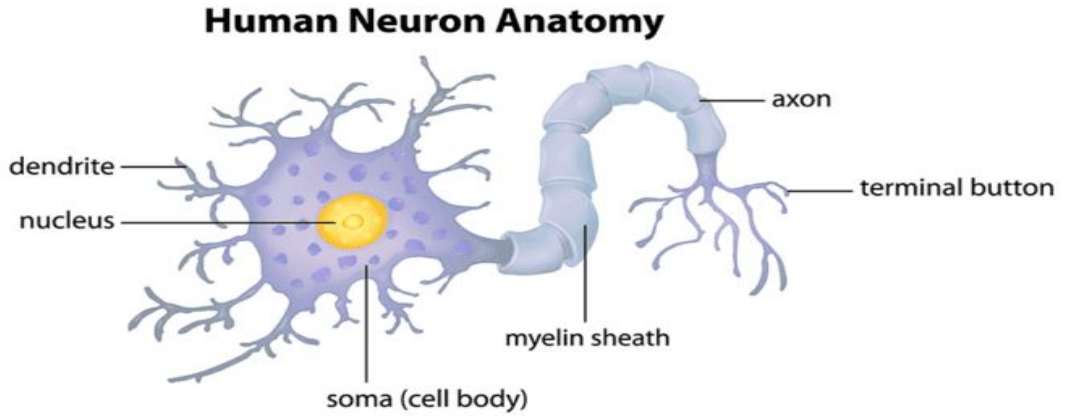
The word *"neural"* is the adjective form of *"neuron,"* and *"network"* denotes a graph-like structure;

Our brains are composed of approximately 86 billion neurons, each connected to about 10,000 other neurons. The cell body of the neuron is called the *soma*, where the inputs (*dendrites*) and outputs (*axons*) connect soma to other soma

Each neuron receives electrochemical inputs from other neurons at their dendrites. If these electrical inputs are sufficiently powerful to activate the neuron, then the activated neuron transmits the signal along its axon, passing it along to the dendrites of other neurons. These attached neurons may also fire, thus continuing the process of passing the message along.

The key takeaway here is that a neuron firing is a **binary operation — the neuron either *fires* or it *doesn't fire***.

Simply put, a neuron will only fire if the total signal received at the soma exceeds a given threshold.



What is Neural Network?

upGrad

A neural network is a **connected structure of artificial neurons** arranged in layers that processes input data to produce an output (prediction or decision). The network learns **patterns and relationships** by adjusting weights during training.

Basic Layers in a Neural Network

A typical feedforward neural network has three main types of layers:

1. **Input Layer**
2. **Hidden Layer(s)**
3. **Output Layer**

1 Input Layer

🎯 **Purpose**:

Receives the raw input data and feeds it into the network.

📌 **Key Points**

- It does *not* perform any computation (no weights or activation).
- Each neuron in the input layer represents one feature of the input.
- If your data has *n* features, you have *n* input nodes.

Example:

- Tabular data with 10 columns → 10 input neurons
- Image 28×28 pixels → 784 input neurons (one per pixel)
- Text encoded by word vectors → one neuron per feature dimension

📌 **Important**:

The input layer simply **passes values forward** — it doesn't transform them.

2 Hidden Layer(s)

🎯 **Purpose**:

Perform the main computation — detecting patterns and transforming the input into meaningful internal representations.

📌 **Why "Hidden"?**

- These layers are *not seen by the outside world*
- They sit between input and output — hence "hidden"

What Happens in Hidden Layers?

Each hidden layer consists of neurons that:

1. Take weighted sums of previous layer outputs
2. Add a bias
3. Apply an activation function

$$\text{output} = f(Wx + b)$$

This process allows the network to **learn non-linear relationships**.

🔹 **Activation Functions**

Hidden layers almost always use **non-linear activation functions** such as:

- ReLU (Rectified Linear Unit)
- Sigmoid
- Tanh

Non-linearity lets the network learn *complex patterns* — something classical models struggle with.

Depth Matters

- **Shallow network:** Few hidden layers
  Good for simple problems
- **Deep network:** Many hidden layers
  Enables learning of **hierarchical features**, e.g.:
  - Vision: edges → shapes → objects → concepts
  - Language: letters → words → phrases → meaning

3 Output Layer

🎯 **Purpose**:

Produces the final result of the network — a prediction, class label, value, etc.

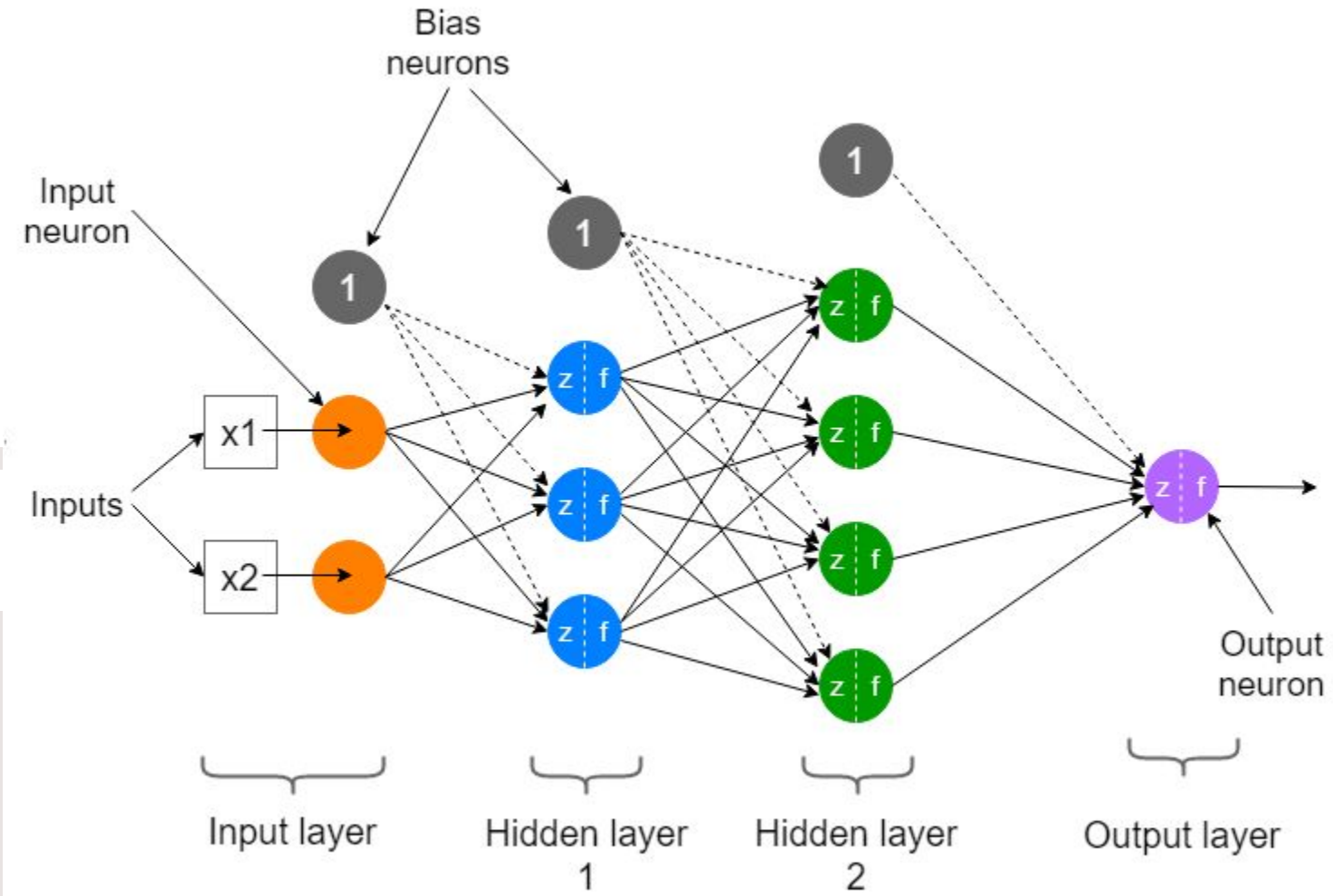
What is Neural Network?

upGrad

Task type	Output shape	Activation
Regression	Single continuous value	Linear
Binary classification	Single probability	Sigmoid
Multi-class classification	Vector of class scores	Softmax
Multi-label	Vector of probabilities	Sigmoid

Example:

- Classifying cats vs dogs → 1 output with sigmoid
- Recognizing digits (0–9) → 10 outputs with softmax



Discuss how stacking layers learn different layers of abstraction

upGrad

When multiple layers are stacked in a neural network (especially convolutional neural networks), each layer transforms its input into a **different representation**. As information flows forward, the network builds **a hierarchy of features** — from very simple to highly abstract — enabling it to understand and make complex predictions

What Each Layer Learns — Explanation With Example

To understand how stacking enables abstraction, let's use a **simple image classification** example (e.g., recognizing objects in a photo):

🔹 **Input Layer**

•The input layer receives raw data (e.g., pixel values of an image).

•It does *no processing* except passing values to the next layer.

Example: A 224×224 color image enters the network as pixel values.

🔹 **First Hidden Layer — Low-Level Features**

🧩 **What it learns:**

•Very basic features such as **edges**, **lines**, and **simple textures**.

These basic patterns are the building blocks of visual understanding — such as vertical/horizontal edges or simple contrast differences.

🔹 **Second Hidden Layer — Mid-Level Features**

🧩 **What it learns:**

•Combinations of low-level signals → **corners**, **curves**, **simple shapes**.

By taking patterns like edges and recombining them, the network starts forming *slightly complex visual primitives*.

🔹 **Deeper Hidden Layers — High-Level Features**

🧩 **What they learn:**

•Larger and more structured patterns, such as:

- **Object parts** (e.g., eyes, wheels, leaves)
- **Textural patterns**
- **Semantic meaning**

Here, the network stitches mid-level features together into more recognizable concepts.

🔹 **Final Layers — Class-Specific Concepts**

🧩 **What they do:**

•Integrate all learned features

•Map them to target classes (e.g., "cat," "dog," "car")

•Often implemented with fully connected layers and softmax for classification

These final layers act as **decision-making layers**, interpreting the features built by earlier layers.

Layer Level	Typical Features Learned
Input	Raw data (pixels)
Early Hidden Layers	Edges, gradients, simple textures
Middle Hidden Layers	Corners, curves, combinations of patterns
Deep Hidden Layers	High-level object parts/semantic patterns
Output Layer	Final decision/class prediction

What is ANN?

upGrad

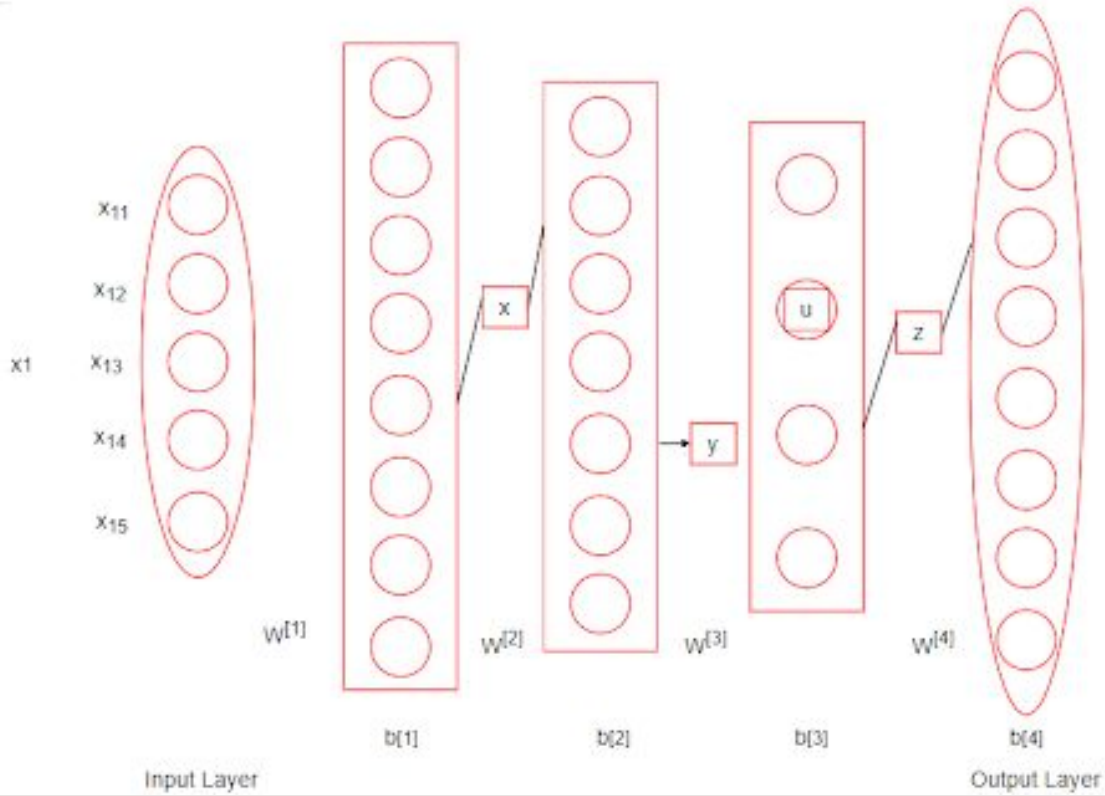
Some of the important hyperparameters to consider to decide the network structure are given below:

- Number of layers
- Number of neurons in the input, hidden and output layers
- Learning rate (the step size taken each time we update the weights and biases of an ANN)
- Number of epochs (the number of times the entire training data set passes through the neural network)

The purpose of training the learning algorithm is to obtain optimum weights and biases that form the **parameters** of the network.

Neural Network

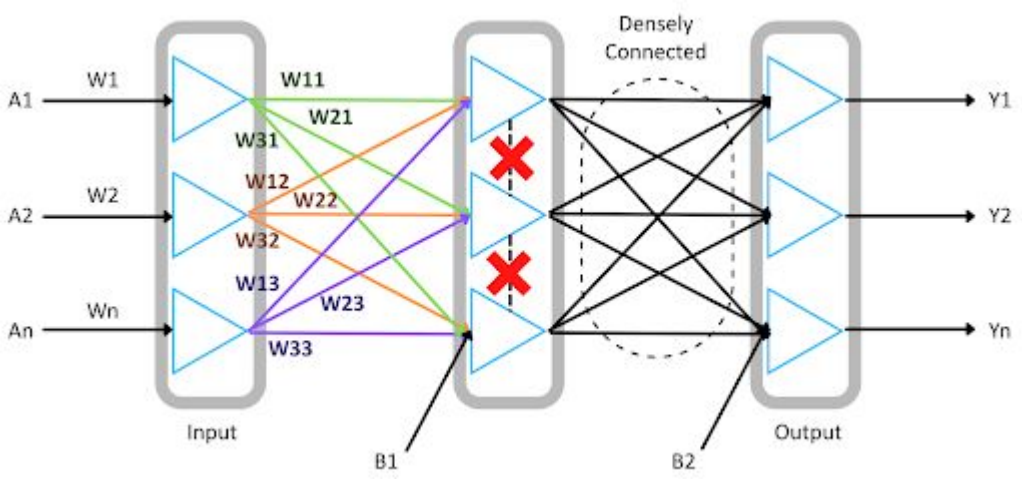
1. **W** represents the weight matrix.
2. **b** stands for bias.
3. **x** represents the input.
4. **y** represents the ground truth label.
5. **p** represents the probability vector of the predicted output for the classification problem



What is ANN?

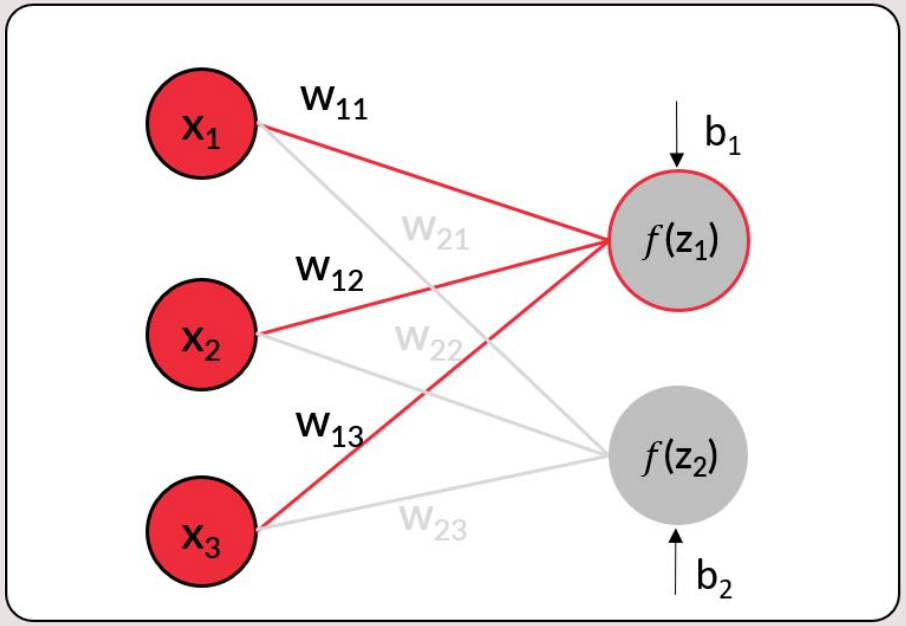
upGrad

Assumptions of ANN



1. The neurons in an ANN are **arranged in layers**, and these layers are arranged **sequentially.**
2. The neurons within the same layer **do not interact** with each other.
3. The inputs are fed into the network through the **input layer**, and the outputs are sent out from the **output layer.**
4. Neurons in **consecutive layers** are **densely connected**, i.e., all neurons in layer l are connected to all neurons in layer l+1.
5. Every neuron in the neural network has a **bias** value associated with it, and each interconnection has a **weight** associated with it.
6. All neurons in a particular hidden layer use the **same activation function**. Different hidden layers can use different activation functions, but in a hidden layer, all neurons use the same activation function.

FeedForward Propagation



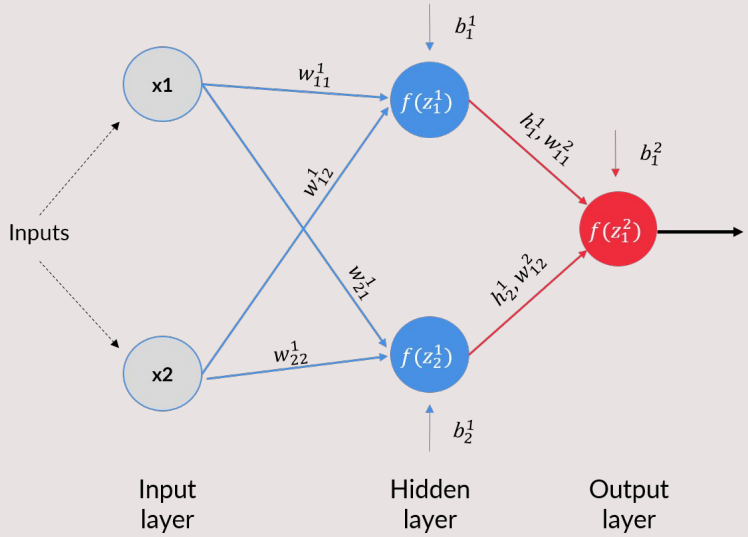
$$[z_1] = [w_{11} \quad w_{12} \quad w_{13}] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + [b_1]$$

$$\begin{bmatrix} z_1^1 \\ z_2^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1 \\ w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^3 x_3 + b_2^1 \end{bmatrix}$$

$$h^1 = \begin{bmatrix} h_1^1 \\ h_2^1 \end{bmatrix} = \sigma(W^1 . x^1 + b^1) = \begin{bmatrix} \sigma(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1) \\ \sigma(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^3 x_3 + b_2^1) \end{bmatrix}$$

Std. Number of Rooms	Std. House Size (sq. ft.)	Price ($)
3	1,340	313,000
5	3,650	2,384,000
3	1,930	342,000
3	2,000	420,000
4	1,940	550,000
2	880	490,000

Std. Number of Rooms	Std. House Size (sq. ft.)	Price ($)
-0.32	-0.66	-0.54
1.61	1.80	2.03
-0.32	-0.03	-0.51
-0.32	-0.03	-0.41
0.65	-0.02	-0.25
-1.29	-1.15	-0.32



$$Layer1:$$

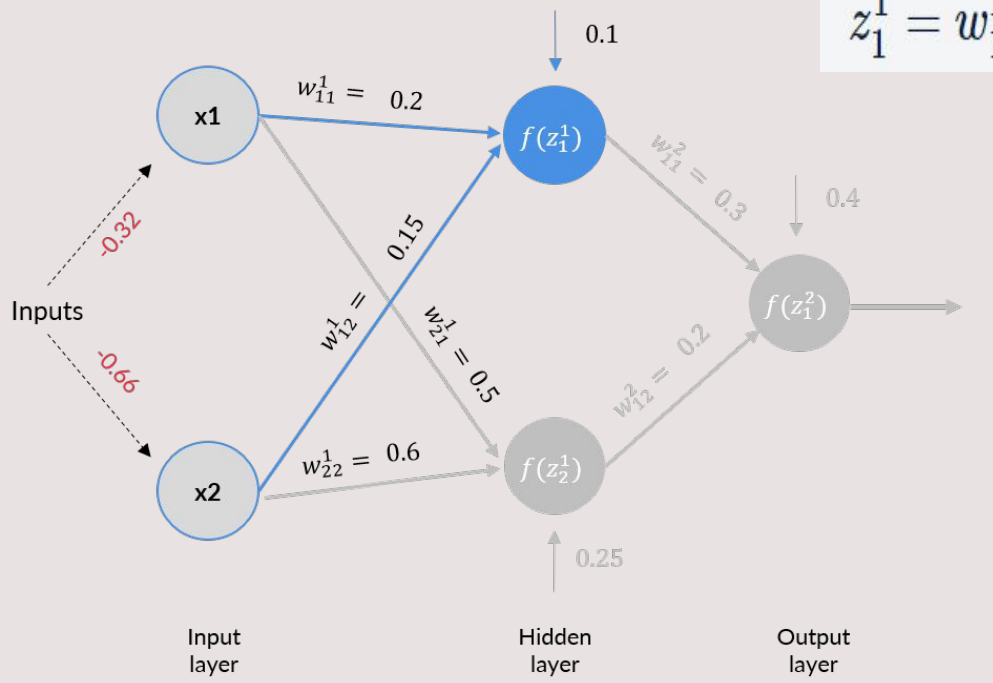
$$W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.15 \\ 0.5 & 0.6 \end{bmatrix}$$

$$b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.25 \end{bmatrix}$$

$$Layer2:$$

$$W^2 = \begin{bmatrix} w_{21}^2 \\ w_{22}^2 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix}$$

$$b^2 = [b_1^2] = [0.4]$$



$$z_1^1 = w_{11}^1 x_1 + w_{12}^1 x_2 + b_1^1 = 0.2 * (-0.32) + 0.15 * (-0.66) + 0.1 = -0.063$$

$$h_1^1 = \sigma(-0.063) = \frac{1}{1+e^{-z_1^1}} = \frac{1}{1+e^{-(-0.063)}} = 0.484$$

$$z_2^1 = w_{21}^1 x_1 + w_{22}^1 x_2 + b_2^1 = 0.5 * (-0.32) + 0.6 * (-0.66) + 0.25 = -0.306$$

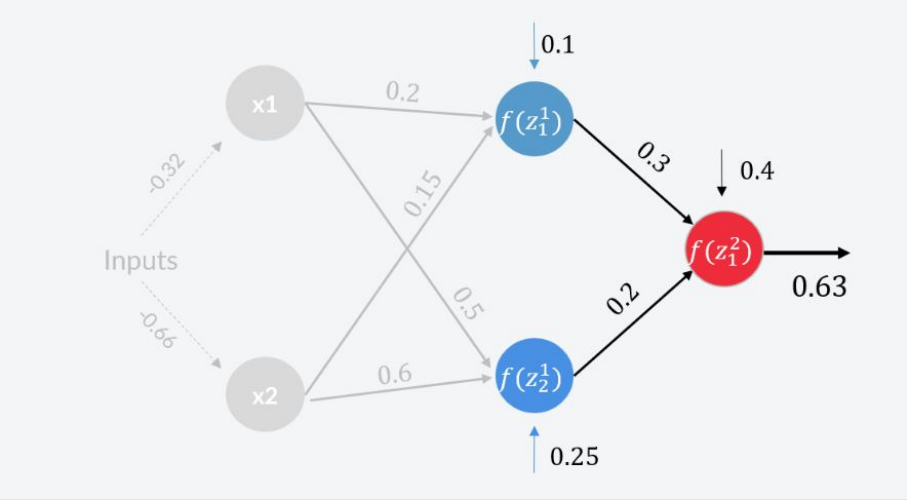
$$h_1^1 = \sigma(-0.306) = \frac{1}{1+e^{-z_2^1}} = \frac{1}{1+e^{-(-0.306)}} = 0.424$$

$$h^1 = \begin{bmatrix} h_1^1 \\ h_2^1 \end{bmatrix} = \sigma(W^1 . x_i + b)$$

$$h^1 = \sigma(\begin{bmatrix} w_{11}^1 x_1 + w_{12}^1 x_2 + b_1^1 \\ w_{21}^1 x_1 + w_{22}^1 x_2 + b_2^1 \end{bmatrix})$$

$$h^1 = \sigma(\begin{bmatrix} 0.2 * (-0.32) + 0.15 * (-0.66) + 0.1 \\ 0.5 * (-0.32) + 0.6 * (-0.66) + 0.25 \end{bmatrix}) = \sigma(\begin{bmatrix} -0.063 \\ -0.306 \end{bmatrix})$$

$$h^1 = \begin{bmatrix} 0.484 \\ 0.424 \end{bmatrix}$$

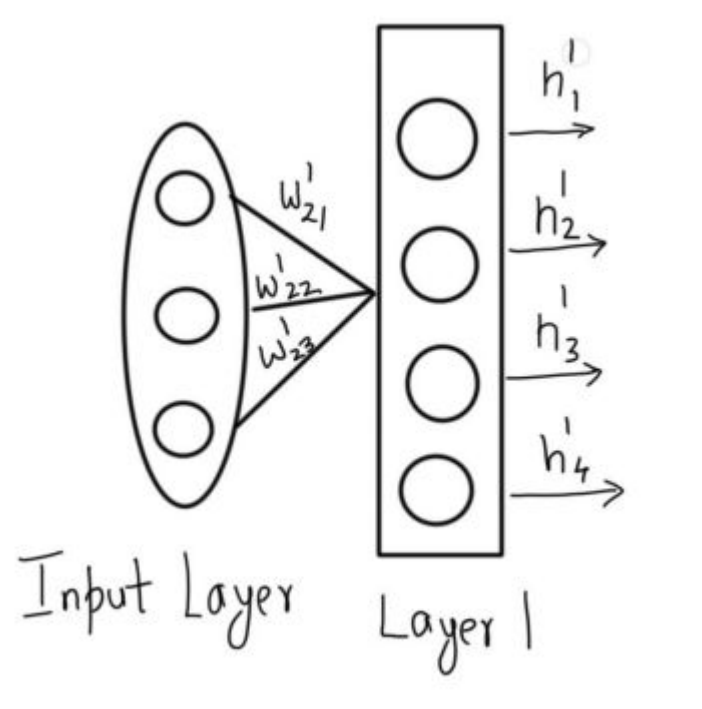


$$z_1^2 = w_{11}^2 h_1^1 + w_{12}^2 h_2^1 + b_1^2 = 0.3 * 0.484 + 0.2 * 0.424 + 0.4 = 0.63$$

$$h^2 = (W^2 h^1 + b^2) = (\begin{bmatrix} w_{11}^2 & w_{12}^2 \end{bmatrix} \begin{bmatrix} h_1^1 \\ h_2^1 \end{bmatrix}) + b^2$$

$$h^2 = (W^2 h^1 + b^2) = (\begin{bmatrix} 0.3 & 0.2 \end{bmatrix} \begin{bmatrix} 0.484 \\ 0.424 \end{bmatrix}) + 0.4$$

$$h^2 = [0.63]$$



$$x_i = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix}, b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ b_4^1 \end{bmatrix}, h^1 = \begin{bmatrix} h_1^1 \\ h_2^1 \\ h_3^1 \\ h_4^1 \end{bmatrix}$$

$$h^1 = \begin{bmatrix} h_1^1 \\ h_2^1 \\ h_3^1 \\ h_4^1 \end{bmatrix} = \sigma(W^1 . x_i + b) = \begin{bmatrix} \sigma(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1) \\ \sigma(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1) \\ \sigma(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1) \\ \sigma(w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1) \end{bmatrix}$$

Hence, the feed forward algorithm for a neural network with L hidden layer and a softmax output becomes:

1. $h^0 = x_i$
2. for l in [1,2,.....,L]:
    1. $h^l = \sigma(W^l . h^{l-1} + b^l)$
3. $p_i = e^{W^o . h^L}$
4. p_i = normalize(p_i)

$$p_i = \begin{bmatrix} p_{i1} \\ p_{i1} \\ . \\ . \\ p_{ic} \end{bmatrix} \text{ where } p_{ij} = \frac{e^{w_j . h^L}}{\sum_{t=1}^{c} e^{w_t . h^L}} \text{ for } j = [1, 2, \ldots, c] \text{ \& } c = \text{number of classes}$$

$$B = \begin{bmatrix} | & | & | & | & | \\ x_i & x_{i+1} & . & . & x_{i+m-1} \\ | & | & | & | & | \end{bmatrix}$$

1. $H^0 = B$
2. for l in $[1, 2, \ldots\ldots, L]$:
    1. $H^l = \sigma(W^l . H^{l-1} + b^l)$
3. P = normalize(exp($W^o . H^L + b^o$))

Loss Function

Total Loss = $L = L_1 + L_2 + L_3 + \ldots\ldots\ldots + L_{1000000}$

$G(w, b) = \frac{1}{n} \sum_{i=1}^{n} L(F(x_i), y_i)$

The loss used for a multiclass classification is the Cross-Entropy loss which can written as follows:

$$-y^T . log(p) = - \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} . \begin{bmatrix} log(p_1) \\ log(p_2) \\ log(p_3) \end{bmatrix} = -(y_1 log(p_1) + y_2 log(p_2) + y_3 log(p_3))$$

$$Loss(L) = RSS = \sum (actual - h^L)^2$$

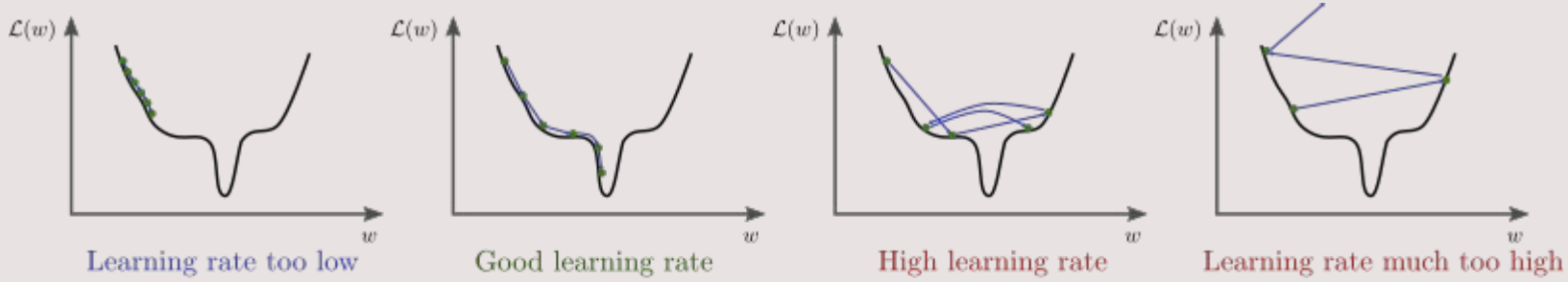
$$Loss(L) = f(W, b)$$

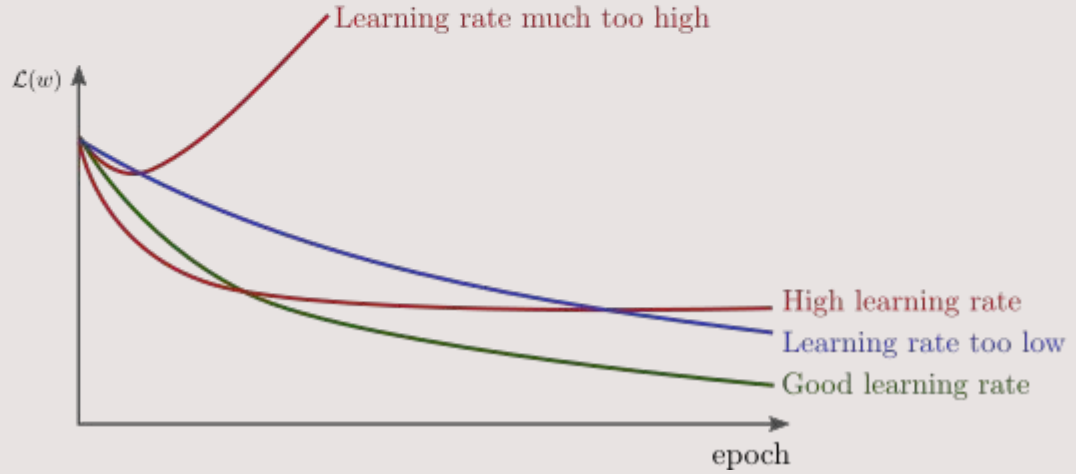
The **learning rate** is a hyperparameter used in the training process of neural networks (or any machine learning model) that controls how much to change the model's weights and biases with respect to the loss gradient. It plays a crucial role in the optimization process, especially when using gradient-based optimization algorithms like **gradient descent**.

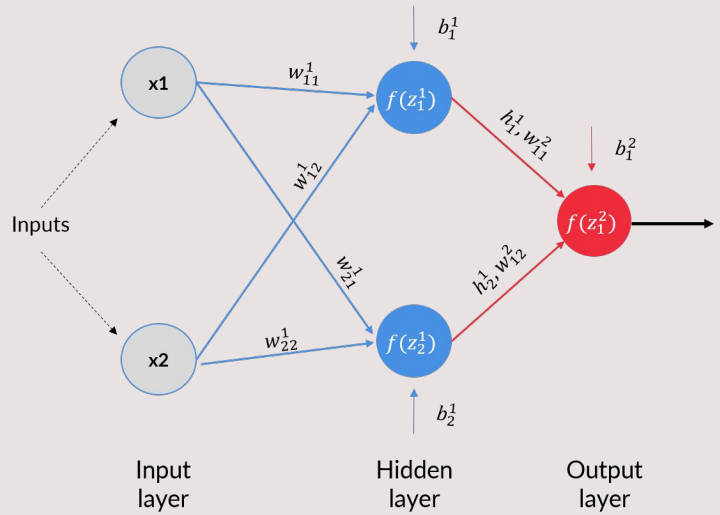
The learning rate, typically denoted by η , α, determines the step size at each iteration while moving toward the optimal solution (minimizing the loss function). It defines how fast or slow the model learns and adjusts its weights.

- **Small Learning Rate**: A small learning rate makes tiny adjustments to the weights and biases. This leads to a slower convergence, but the advantage is that it reduces the risk of overshooting the optimal value. However, too small a learning rate might make the training process extremely slow and potentially cause the model to get stuck in local minima.
- **Large Learning Rate**: A large learning rate makes more significant adjustments to the weights, leading to faster convergence. However, if the learning rate is too large, the model might overshoot the optimal solution and fail to converge, causing the loss to oscillate.

$$w_{kj}^{l} = w_{kj}^{l} - \eta \frac{\partial L}{\partial w_{kj}^{l}}$$



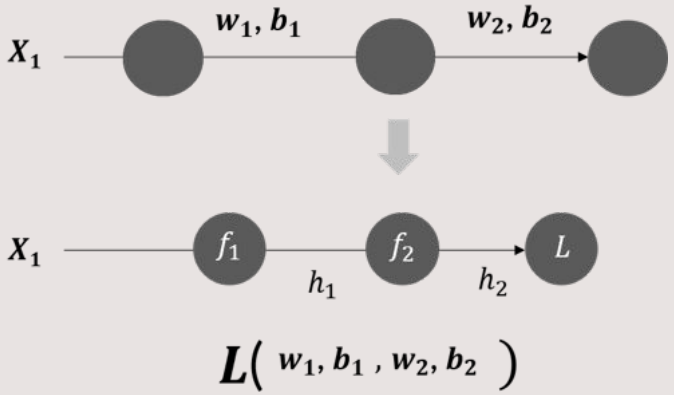




$$w_{11}^1 = w_{11}^1 - \eta \frac{\partial L}{\partial w_{11}^1} \qquad w_{21}^1 = w_{21}^1 - \eta \frac{\partial L}{\partial w_{21}^1}$$

$$w_{12}^1 = w_{12}^1 - \eta \frac{\partial L}{\partial w_{12}^1} \qquad w_{22}^1 = w_{22}^1 - \eta \frac{\partial L}{\partial w_{22}^1}$$

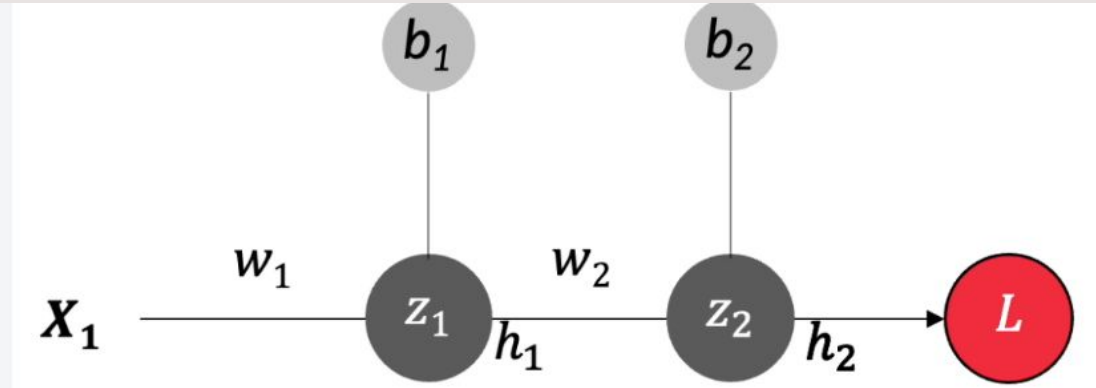
$$b_1^1 = b_1^1 - \eta \frac{\partial L}{\partial b_1^1} \qquad b_2^1 = b_2^1 - \eta \frac{\partial L}{\partial b_2^1}$$



Loss function: $Loss = \frac{1}{2}(y - h_2)^2$

Cumulative Input: $z_i = w_i h_{i-1} + b_i$

Output using tanh activation function: $h_i = tanh(z_i)$



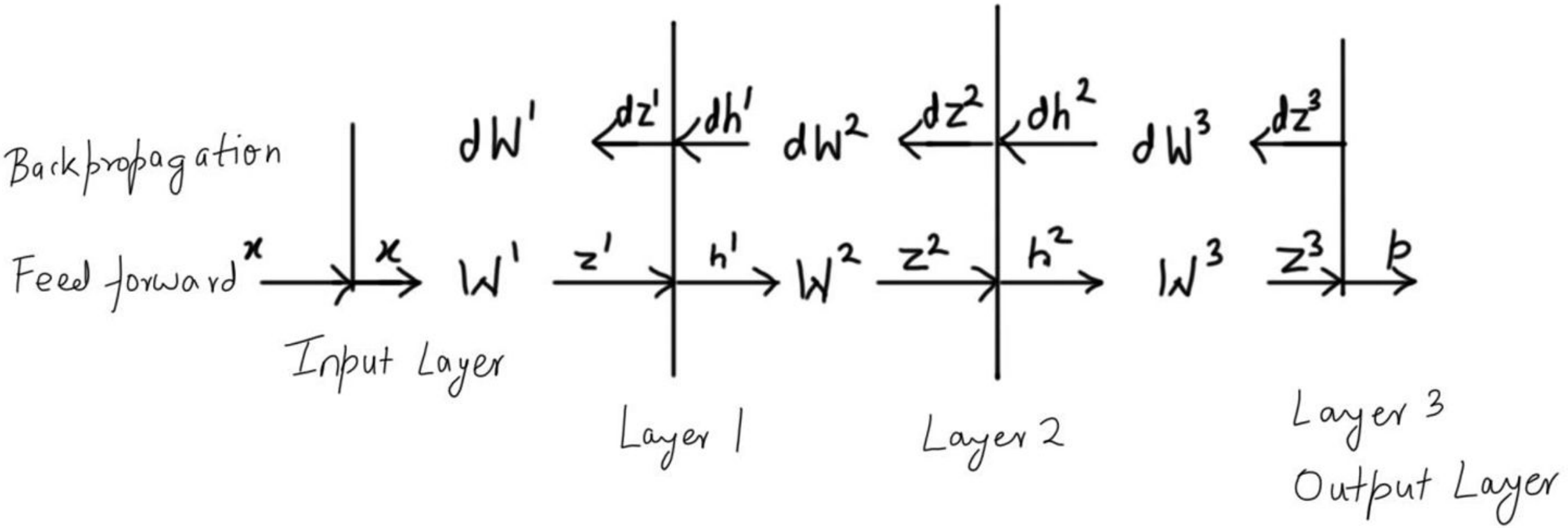
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial w_2}$$

Back Propagation

From the last layer to the first layer, for each layer, compute the gradient of the loss function with respect to the weights at each layer and all the intermediate gradients.

Once all the gradients of the loss with respect to the weights (and biases) are obtained, use an optimisation technique like gradient descent to update the values of the weights and biases.

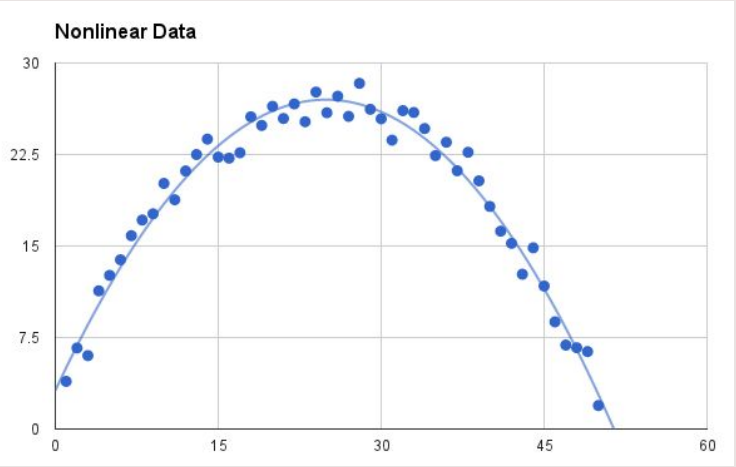
Repeat the process for a specified number of iterations or until the predictions made by the model are acceptable.

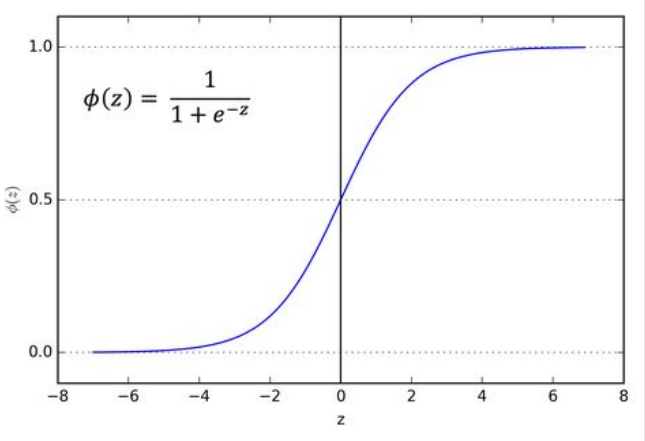

Backpropagation
Feed forward
x
x
W1
z1
h1
W2
z2
h2
W3
z3
p
dW1
dz1
dh1
dW2
dz2
dh2
dW3
dz3
Input Layer
Layer 1
Layer 2
Layer 3
Output Layer

Single layer vs. multi-layer perceptron

There are **two types of Perceptron:**

- A **single layer Perceptron** can learn only separable linear functions.
- A **multi-layer Perceptron**, also known as a feed-forward neural network, overcomes this limitation and offers superior computational power. It is also possible to combine several Perceptrons to create a **powerful mechanism**.





The main terminologies needed to understand for nonlinear functions are:

Derivative or Differential: *Change in y-axis w.r.t. change in x-axis.It is also known as slope.*

Monotonic function: *A function which is either entirely non-increasing or non-decreasing.*

The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves**-

1. Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape.

The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output.Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

The logistic sigmoid function can cause a neural network to get stuck at the training time.

The **softmax function** is a more generalized logistic activation function which is used for multiclass classification.

The function is **monotonic** but function's derivative is not.

Prediction: For a given input vector x, the Perceptron computes the weighted sum of the inputs plus the bias:

$$z = w^T \cdot x + b$$

Where:

- w is the weight vector.
- x is the input vector.
- b is the bias term.
- z is the linear combination of inputs.

Weight Update Rule:

- When the prediction is incorrect, the Perceptron updates its weights and bias to reduce the error. The weight update rule is:

$$w = w + \eta \cdot e \cdot x$$

$$b = b + \eta \cdot e$$

Where:

- η is the learning rate (a small positive number).
- e is the error.
- x is the input vector.

- The learning rate η controls how large the updates are and helps in converging to a solution.

- **Error**: The error for a single training example is the difference between the actual label y and the predicted label $\hat{y}$:

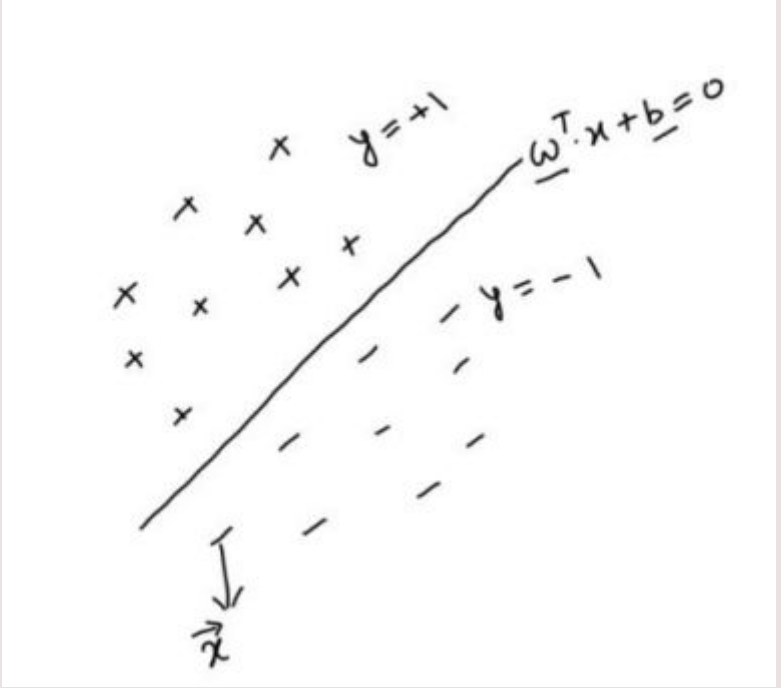
$$e = y - \hat{y}$$

  - If $e = 0$, the prediction is correct.
  - If $e \neq 0$, the prediction is incorrect, and the Perceptron will adjust the weights.

- **Activation Function**: The Perceptron uses a step function (also known as the Heaviside step function) to make a binary classification:

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

Where $\hat{y}$ is the predicted label.

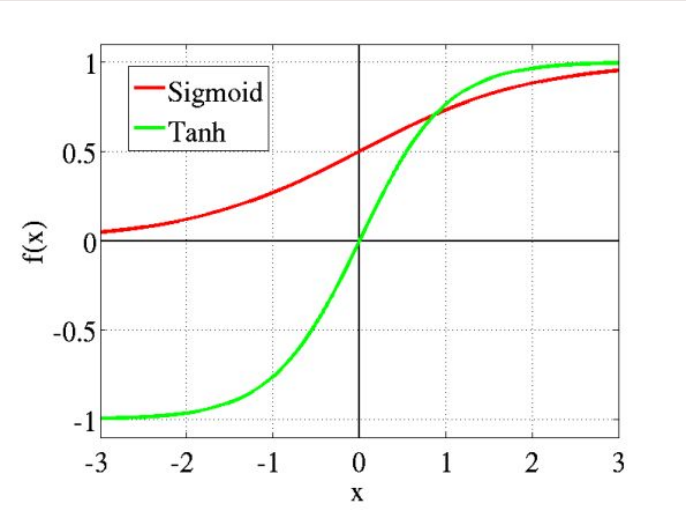


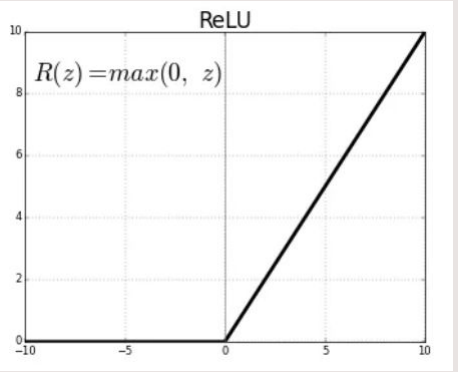
2. Tanh or hyperbolic tangent Activation Function
tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).

The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

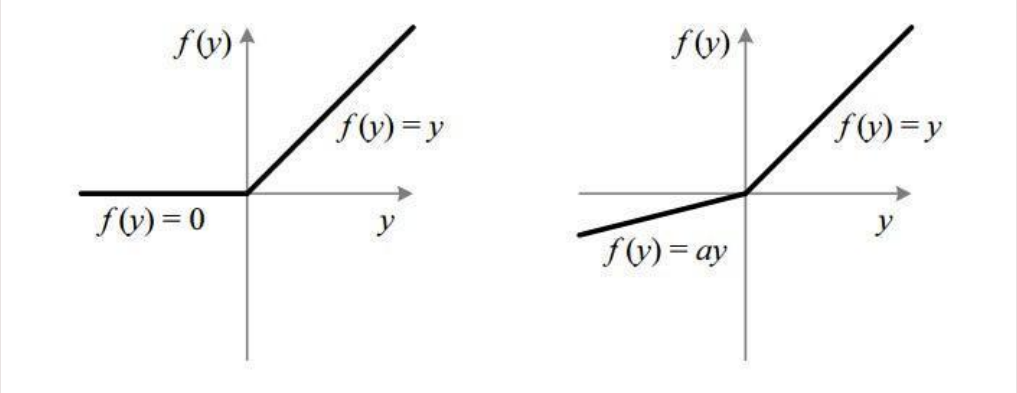
The tanh function is mainly used classification between two classes.
Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

The function is **monotonic** but function's derivative is not.





3. ReLU (Rectified Linear Unit) Activation Function
The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning.



As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.
Range: [0 to infinity)
The function and its derivative **both are monotonic**.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

Summary of When to Use Each Activation Function:

Activation Function	Use Case	Where to Use	Limitations
Sigmoid	Binary classification	Output layer	Vanishing gradient problem
Softmax	Multiclass classification	Output layer	Not used in hidden layers
ReLU	General deep networks	Hidden layers	Dying ReLU problem
Tanh	Normalized inputs (-1, 1)	Hidden layers	Vanishing gradient problem
Leaky ReLU	Deep networks, avoid dead neurons	Hidden layers	Slope for negative inputs might still be too small

Practical Examples:

- **Sigmoid**: If you're building a neural network to predict whether an image contains a cat or not (binary classification), you would use sigmoid in the output layer.
- **Softmax**: For classifying handwritten digits (0-9), you would use softmax in the output layer.
- **ReLU**: Use ReLU in hidden layers for tasks like image classification using a CNN.
- **Tanh**: For RNNs dealing with sequential data, where you need outputs centered around zero.
- **Leaky ReLU**: If you're noticing the dying ReLU problem in deep layers of a network, switch to Leaky ReLU to allow a small gradient for negative inputs.

Why Non-Linearity Is Essential for Learning Complex Mappings

Non-linearity is the **core reason neural networks can model complex real-world relationships**. Without it, even very large networks collapse into simple linear models and lose most of their power.

1. What Happens Without Non-Linearity?

Consider a neural network with multiple layers **but no activation functions**:

$$y = W_3(W_2(W_1 x))$$

This simplifies to:

$$y = Wx$$

➞ **Multiple linear layers = one linear transformation**

Consequence

- The network can only learn **straight-line (linear) relationships**
- Depth becomes meaningless
- Complex patterns cannot be represented

📌 **Key Insight:**

Stacking linear operations does *not* increase expressive power.

2. Real-World Data Is Inherently Non-Linear

Most real-world problems involve **non-linear interactions**:

- Image pixels combine spatially and hierarchically
- Speech depends on time, frequency, and context
- Language meaning depends on word order and semantics
- Financial, biological, and physical systems have feedback loops

Linear models **cannot bend or curve decision boundaries** to fit such data.

3. Non-Linearity Enables Curved Decision Boundaries

Linear Model

- Can only separate data using a **straight line or hyperplane**

Non-Linear Model

- Can create **curved, twisted, and nested boundaries**

This is why problems like XOR are **impossible** for linear models but trivial for neural networks with non-linear activations.

📌 Example:

- XOR data cannot be separated by a straight line
- A single hidden layer with a non-linear activation solves it

4. Activation Functions Introduce Non-Linearity

Activation functions apply a **non-linear transformation** after each neuron's weighted sum:

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

Common non-linear activations:

- **ReLU** → enables sparse and efficient learning
- **Sigmoid** → smooth probability-like outputs
- **Tanh** → zero-centered non-linearity

These functions allow the network to **bend and shape** the learned function.

5. Non-Linearity Enables Hierarchical Feature Learning

Non-linearity allows each layer to learn **a different level of abstraction**:

Example: Image Recognition

- Layer 1: edges (non-linear contrast detection)
- Layer 2: corners and shapes
- Layer 3: object parts
- Layer 4: full objects

Without non-linearity:

- Each layer would just compute another linear mix of pixels
- No hierarchy would emerge

📌 **Non-linearity makes depth meaningful**

6. Function Composition Creates Expressive Power

Neural networks work by **composing functions**:

$$f_3\left(f_2\left(f_1(x)\right)\right)$$

When each function is non-linear:

- The composition becomes exponentially more expressive
- Complex mappings can be built from simple pieces

This is why deep networks can approximate extremely complicated functions efficiently.

7. Theoretical Perspective

Universal Approximation Theorem

A neural network with:

- At least one hidden layer
- A **non-linear activation function**

can approximate **any continuous function** (given enough neurons).

Without non-linearity:

- The theorem does not hold
- The network has strictly limited capacity

8. Intuition: A Simple Analogy

Imagine trying to draw:

- A **circle** using only straight lines → impossible
- A **complex shape** using only straight cuts → impossible

Non-linearity allows the model to **curve, fold, and warp space**, making complex shapes learnable.

With Linear Layers Only	With Non-Linearity
Only straight boundaries	Curved and complex boundaries
Depth adds no power	Depth adds expressiveness
Cannot solve XOR	Easily solves XOR
No feature hierarchy	Hierarchical abstractions
Limited to simple mappings	Learns real-world complexity

Practical Intuition: From Classical ML to Neural Networks

1. How Classical Machine Learning Thinks

In classical ML, we assume that the **input features already describe the problem well**.

Typical ML Pipeline

1.Collect data

2.Manually design features

3.Choose a model

4.Train the model

5.Make predictions

Example: House Price Prediction

$$\text{Price} = w_1 \cdot Area + w_2 \cdot Rooms + b$$

Key assumption:

If we give the model good features, a simple model can learn the task.

This works well when:

- Relationships are simple or mostly linear
- Data is structured and low dimensional
- Domain knowledge is available

2. Where Classical ML Starts to Break Down

As problems become more realistic:

- Relationships become **non-linear**
- Features interact in complex ways
- Data becomes **high-dimensional** (images, text, audio)

Example: Image Data

- One image = thousands of pixels
- Pixel interactions are spatial and hierarchical
- Writing rules or features manually becomes impractical

📌 **Problem:**

Classical ML can *combine features*, but it **cannot learn features automatically**.

3. First Bridge: Logistic Regression → Single Neuron

A key insight:

A single artificial neuron is mathematically equivalent to logistic regression.

Logistic Regression

$$y = \sigma(w \cdot x + b)$$

Artificial Neuron

- Inputs → weights → bias → activation

Important intuition:

Neural networks do not start from something completely new — they **generalize classical ML models**.

4. What Actually Changes in Neural Networks

The difference is **not the neuron** — it's the **stacking of neurons into layers**.

Classical ML

Input ⟶ Model ⟶ Output

Neural Network

Input ⟶ Layer ⟶ Layer ⟶ Layer ⟶ Output

Each layer:

- Receives inputs
- Transforms them
- Produces a new representation

📌 **This is the core leap from ML to NN.**

5. Feature Engineering vs Feature Learning (Core Intuition)

Classical ML: Feature Engineering

You manually create features such as:

- Polynomial terms
- Interaction terms
- Domain-specific transformations

Example:

$x \longrightarrow x^2$, $\log(x)$, $x_1 \cdot x_2$

Neural Networks: Feature Learning

The network:

- Learns useful transformations
- Learns interactions automatically
- Adjusts them based on data

📌 **Neural networks replace manual feature engineering with learned representations.**

6. Why Depth (Multiple Layers) Matters

Each layer performs:

$$\text{output} = f(Wx + b)$$

Stacking layers means:

$$f_3\left(f_2\left(f_1(x)\right)\right)$$

Intuition

- First layer: learns simple patterns
- Next layers: combine patterns into more complex ones
- Final layers: make decisions

📌 **Depth allows progressive abstraction**, something classical ML cannot do naturally.

7. Practical Example: Image Recognition

Classical ML Approach

1. Detect edges manually
2. Compute textures
3. Extract shapes
4. Train SVM or logistic regression

❌ Requires expert knowledge
❌ Hard to scale
❌ Breaks on complex images

Neural Network Approach

1. Input: raw pixels
2. Early layers: edges
3. Middle layers: shapes
4. Deep layers: objects

✔ End-to-end learning
✔ Scales with data
✔ Adapts automatically

📌 **Neural networks learn the same hierarchy humans design — automatically.**

8. Non-Linearity: Why ML Intuition Must Change

In classical ML:

- Linear models ⟶ straight boundaries
- Non-linearity must be added manually

In neural networks:

- Activation functions provide built-in non-linearity
- Every layer bends the representation space

📌 **Without non-linearity, deep networks collapse into linear models.**

9. How to Think Practically When Moving from ML to NN

Classical ML Thinking	Neural Network Thinking
Design features	Design architecture
Choose kernel	Choose activation
Simplify data	Add capacity
Small datasets	Large datasets

10. When to Use ML vs Neural Networks

Scenario	ML	NN
Small structured data	✅	❌
Simple relationships	✅	❌
Images, text, audio	❌	✅
Complex interactions	❌	✅

Deep learning models can analyze human speech despite varying speech patterns, pitch, tone, language, and accent. Virtual assistants such as Amazon Alexa and automatic transcription software use speech recognition to do the following tasks:

- Assist call center agents and automatically classify calls.
- Convert clinical conversations into documentation in real time.

Google Translate

Computers use deep learning algorithms to gather insights and meaning from text data and documents. This ability to process natural, human-created text has several use cases, including in these functions:

- Automated virtual agents and chatbots
- Automatic summarization of documents or news articles

A recommendation engine is a system that gives customers recommendations based upon their behavior patterns and similarities to people who might have shared preferences. These systems, also known as recommenders, use statistical modeling, machine learning, and behavioral and predictive analytics algorithms to personalize the web experience.