DEEP LEARNING

what is deep learning?

Deep Learning is a way for computers to learn by themselves using layered neural networks, similar to the human brain. It helps machines understand images, text, speech, and other complex data.

Limitations of Machine Learning

1. Requires Manual Feature Engineering
   Performance depends heavily on human-designed features.
2. Struggles with Unstructured Data
   Not effective for images, audio, video, or raw text without heavy preprocessing.
3. Limited Ability to Learn Complex Patterns
   Cannot automatically learn hierarchical or deep representations.
4. Scalability Issues
   Training becomes inefficient as data volume and dimensionality increase.
5. Poor Handling of Sequential Data
   Not suitable for time-series, speech, or language without specialized models.
6. High Dependence on Domain Expertise
   Requires manual tuning, preprocessing, and expert knowledge.
7. No End-to-End Learning
   Cannot work directly on raw data; needs transformations first.
8. Generalization Limitations
   Often overfits small datasets and struggles with complex real-world patterns.

Machine Learning vs Deep Learning

Aspect	Machine Learning (ML)	Deep Learning (DL)
Definition	Subset of AI that uses algorithms to learn patterns from data.	Subset of ML that uses multi-layered neural networks to learn complex patterns.
Data Requirement	Works well with small to medium datasets.	Requires large amounts of data to perform well.
Feature Engineering	Manual feature extraction needed; domain expertise required.	Automatic feature extraction; learns directly from raw data.
Type of Data	Best for structured/tabular data.	Excellent for unstructured data (images, audio, text).
Training Time	Faster training; less computationally heavy.	Slower training; requires GPUs/TPUs and high computation.
Model Complexity	Simple to moderately complex models.	Very complex models with many layers (deep networks).
Interpretability	Easier to interpret (e.g., Decision Trees, Linear Models).	Black-box models; difficult to interpret.
Examples of Algorithms	SVM, Decision Trees, Random Forest, Naive Bayes, KNN.	CNNs, RNNs, LSTMs, GRUs, Transformers, GANs.
Performance on Complex Problems	Limited ability; plateaus with complexity.	High accuracy on complex tasks like vision, speech, NLP.
Computational Requirement	Low to moderate.	High (needs GPU/TPU).
End-to-End Learning	Not end-to-end; requires preprocessing and feature design.	End-to-end learning from raw input to output.
Use Cases	Fraud detection, credit scoring, small-scale prediction.	Image recognition, self-driving cars, chatbots, speech recognition.

Deep Learning models

Deep Learning Model	Description	Best For / Applications
Artificial Neural Networks (ANNs)	Basic neural networks with input, hidden, and output layers.	General prediction, classification, regression.
Convolutional Neural Networks (CNNs)	Use convolution layers to detect spatial patterns.	Image classification, object detection, medical imaging, face recognition.
Recurrent Neural Networks (RNNs)	Capture sequential dependencies using memory of previous steps.	Time-series forecasting, text processing, speech recognition.
LSTM (Long Short-Term Memory)	Special RNN that handles long-term dependencies and avoids vanishing gradient.	Language modeling, translation, chatbots, financial forecasting.
GRU (Gated Recurrent Unit)	Simplified LSTM with fewer parameters; faster to train.	Text generation, speech tasks, sequence modeling.
Transformers	Use self-attention to process sequences in parallel; state-of-the-art model.	NLP tasks (GPT, BERT), translation, Q&A, Vision Transformers.
Autoencoders	Learn compressed representations; reconstruct input.	Anomaly detection, noise removal, dimensionality reduction.
Variational Autoencoders (VAEs)	Probabilistic autoencoders; generate smooth latent representations.	Image generation, data augmentation, drug discovery.
Generative Adversarial Networks (GANs)	Generator + Discriminator competing to create realistic data.	Deepfakes, image synthesis, art generation, super-resolution.
Deep Reinforcement Learning Models (DQN, PPO, Actor–Critic)	Combine deep learning with reinforcement learning for decision-making.	Robotics, autonomous vehicles, gaming (AlphaGo).

Applications

Domain / Industry	Applications
Computer Vision	Image classification, object detection, face recognition, medical imaging, self-driving car perception, video surveillance
Natural Language Processing (NLP)	Chatbots, machine translation, sentiment analysis, text summarization, question answering
Speech Processing	Speech recognition, speech synthesis, voice assistants, noise reduction, emotion detection from voice
Healthcare	Disease detection (cancer, COVID-19), medical image analysis, drug discovery, personalized medicine, patient monitoring
Autonomous Systems	Self-driving cars, drones, robotics, smart manufacturing, industrial automation
Finance	Fraud detection, algorithmic trading, risk modeling, credit scoring, stock price prediction
Entertainment & Media	Deepfakes, AI-generated art, video generation, music creation, image enhancement
E-commerce & Retail	Recommendation systems, personalized ads, demand forecasting, customer behavior prediction
Cybersecurity	Malware detection, intrusion detection, anomaly detection, fraud detection
Agriculture	Crop disease detection, yield prediction, drone-based monitoring, automated harvesting

Frameworks

Framework	Developer	Key Features	Best Used For
TensorFlow	Google	Scalable, graph-based, TensorBoard, supports GPUs/TPUs	Research + Production, large-scale training
Keras	Google (as part of TensorFlow)	Simple API, easy model building, quick prototyping	Beginners, rapid experimentation
PyTorch	Meta (Facebook)	Dynamic computation graphs, easy debugging, strong community	Research, NLP, CV, Transformer models
JAX	Google	XLA compilation, super fast on GPUs/TPUs, automatic differentiation	High-performance ML, scientific computing
MXNet	Apache	Distributed training, hybrid programming	Cloud-based ML, large datasets
CNTK	Microsoft	Efficient DNN training, supports CNN/RNN/LSTM	Production models, enterprise ML
Caffe	Berkeley AI Research	Very fast for CNNs, optimized for images	Computer vision, image classification
Theano **(discontinued)**	MILA (Montreal)	First symbolic computation library, historical importance	Research foundation; legacy Keras backend
ONNX	Microsoft + Partners	Model interchange format, cross-platform portability	Deploying models trained in TF/PyTorch

Introduction to Neural Networks

A **Neural Network** is a computational model inspired by the working of the **human brain**. It is made up of interconnected processing units called **neurons**, which work together to learn patterns from data.

They learn by adjusting internal parameters (**weights and biases**) through training.

Neurons – Artificial Neuron / Perceptron

What is an Artificial Neuron?

An **Artificial Neuron** (also called a *node* or *unit*) is the basic building block of a neural network.

It takes multiple inputs, applies weights, adds a bias, and passes the result through an activation function to produce an output.

Mathematical representation:

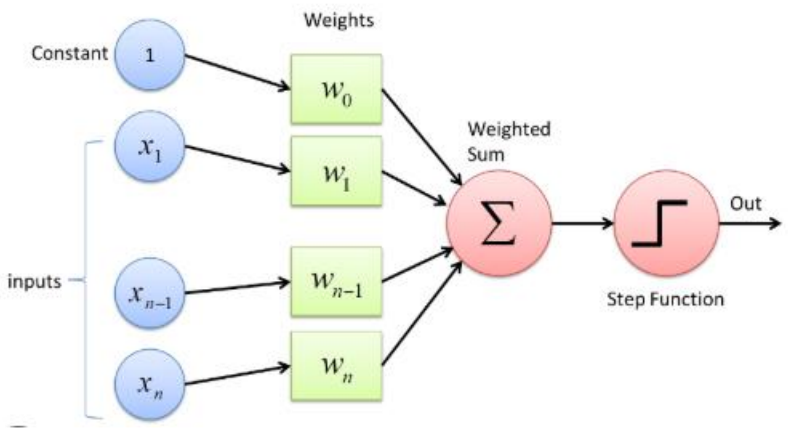
$$\text{Output} = f(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$$

Where:

- $x_1, x_2 \ldots x_n$: Inputs
- $w_1, w_2 \ldots w_n$: Weights
- b: Bias
- f: Activation function

Perceptron

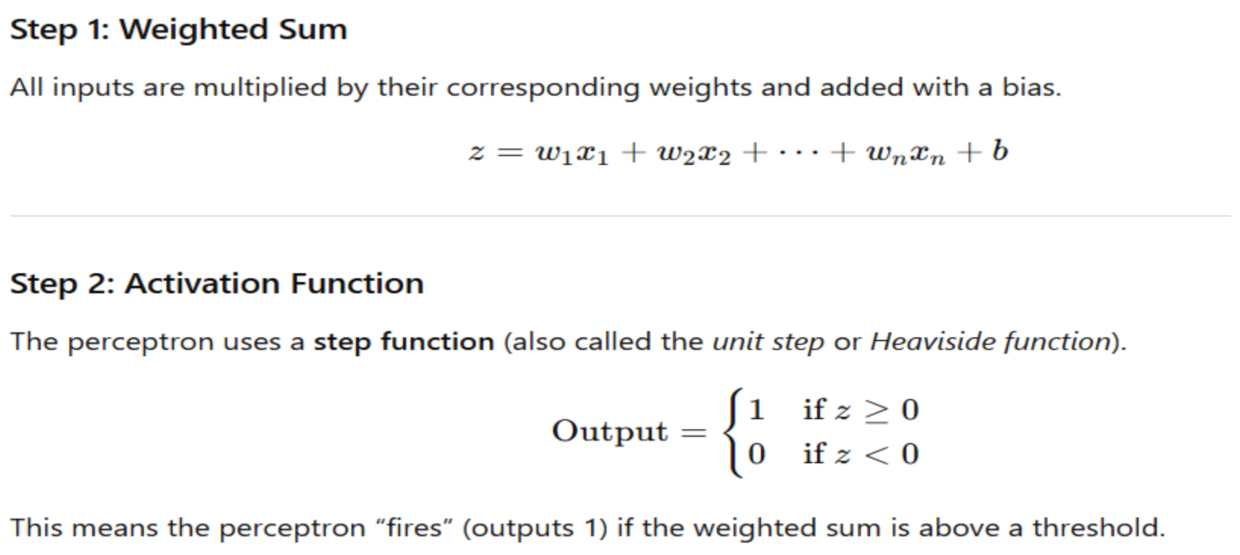
- A simple algorithm for supervised learning of binary classifiers, used as a building block for neural networks.
- It functions as an artificial neuron that takes numerical inputs, weights them, and uses an activation function to produce a binary output ("yes" or "no").
- Perceptrons are useful for tasks like image classification where data can be linearly separated into two distinct categories.
- A **Perceptron** introduced by Frank Rosenblatt in 1957.



- **Inputs:** $x_1, x_2, x_3, \ldots$
- **Weights:** $w_1, w_2, w_3, \ldots$ applied to each input
- **Summation unit:** Computes

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

- **Activation function:** Step function for perceptron
- **Output:** Binary (0 or 1)



Step 1: Weighted Sum

All inputs are multiplied by their corresponding weights and added with a bias.

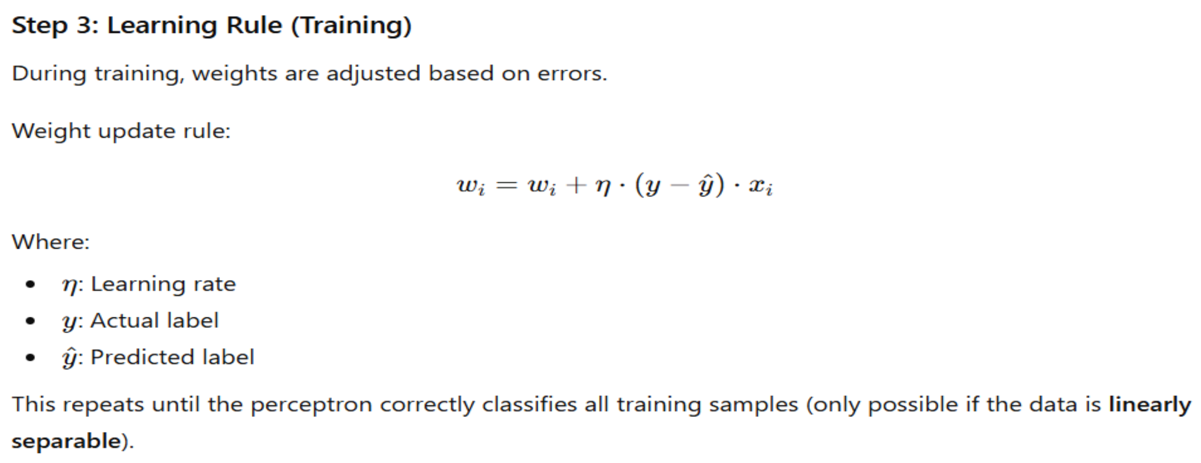
$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

Step 2: Activation Function

The perceptron uses a **step function** (also called the *unit step* or *Heaviside function*).

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

This means the perceptron “fires” (outputs 1) if the weighted sum is above a threshold.



Step 3: Learning Rule (Training)

During training, weights are adjusted based on errors.

Weight update rule:

$$w_i = w_i + \eta \cdot (y - \hat{y}) \cdot x_i$$

Where:

- η: Learning rate
- y: Actual label
- $\hat{y}$: Predicted label

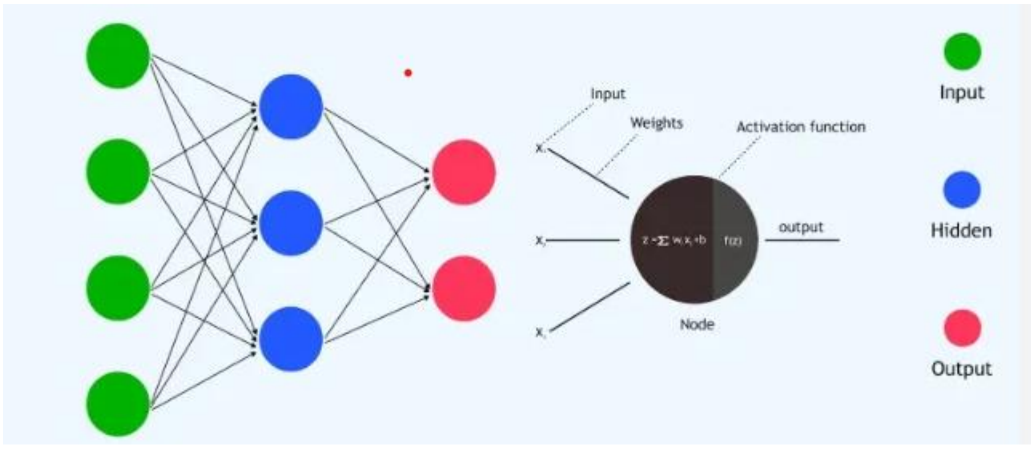
This repeats until the perceptron correctly classifies all training samples (only possible if the data is **linearly separable**).

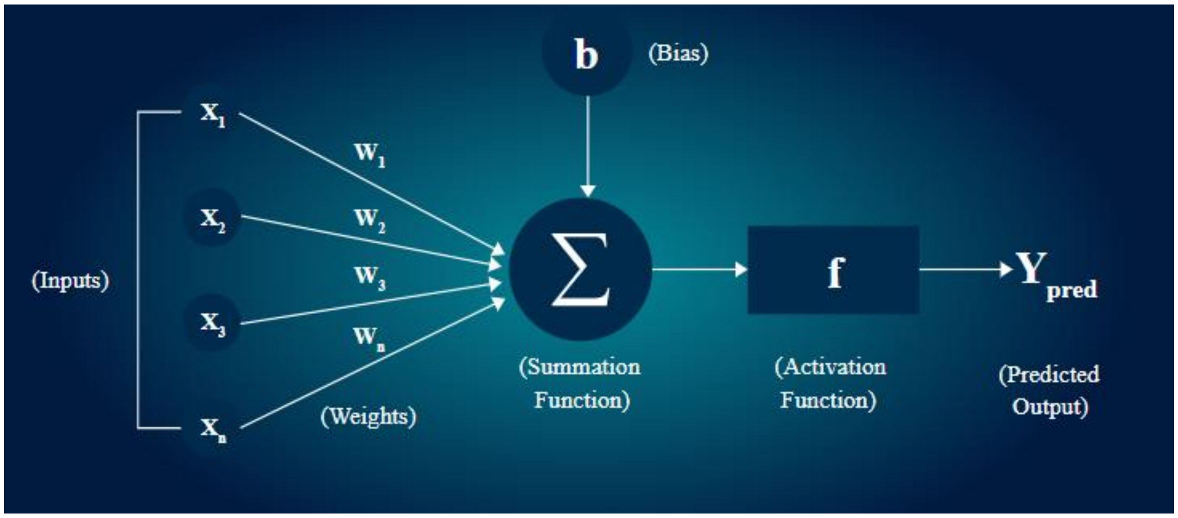
Learning and limitations

- **Learning:** The algorithm learns by adjusting the weights and bias to minimize errors in its classifications. This process is often called backpropagation.
- **Linear separability:** The original perceptron can only solve problems where the data is linearly separable—meaning a straight line can be drawn to separate the two classes.
- **Limitations:** For problems that are not linearly separable, more complex architectures like multi-layer perceptrons are needed.

Neural Network Architecture

A **Neural Network (NN)** is made up of layers of interconnected artificial neurons.





```
Input Layer → Hidden Layer 1 → Hidden Layer 2 → ... → Output Layer
```

Basic Architecture:

1. **Input Layer**
    - Takes raw features (e.g., pixel values, sensor readings).
2. **Hidden Layers**
    - One or more layers where neurons process information.
    - Each neuron performs:

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

    then applies an activation function.

3. **Output Layer**
    - Produces the final prediction (e.g., class label, probability).

Key Components

- **Weights (w)**: Strength of connection between neurons.
- **Bias (b)**: Extra parameter added to the weighted sum.
- **Activation Functions**: ReLU, Sigmoid, Tanh, Softmax.
- **Loss Function**: Measures error (e.g., MSE, cross-entropy).

Working of a Neural Network

The NN works in two major phases:

A) Forward Propagation

- Input → Output
- Computes predictions based on current weights.

B) Backpropagation

- Output → Input
- Adjusts weights based on the error between predicted and actual values.

Training cycle

1. Give inputs to the network.
2. Perform **forward pass** to get prediction.
3. Compute **loss** (error).
4. Perform **backpropagation** to update weights.
5. Repeat for many iterations/epochs.

Forward Propagation (Forward Pass)

Forward propagation means sending the input layer's data through the network to compute the output.

Steps:

1. **Weighted Sum**

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

2. **Activation Function**

   Apply activation:

$$a = f(z)$$

3. **Repeat for all layers**

   Output of one layer becomes input to the next.

4. **Final Output**
   - Regression → numeric output
   - Classification → probabilities (Softmax)

```
Input → (Weighted Sum + Activation) → Hidden Layer → Output Layer
```

Backpropagation (Backward Pass)

Backpropagation is the algorithm used to **train** neural networks.

Goal:

Reduce error by updating weights using gradient descent.

Steps in Backpropagation:

Step 1: Compute Loss

Loss = difference between predicted output ($\hat{y}$) and actual output (y).

- Example:

$$L = \frac{1}{2}(y - \hat{y})^2$$

Step 2: Calculate Gradients

Use **partial derivatives** to determine how much each weight contributed to the error.

$$\frac{\partial L}{\partial w}$$

Step 3: Update the Weights

Using Gradient Descent:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}$$

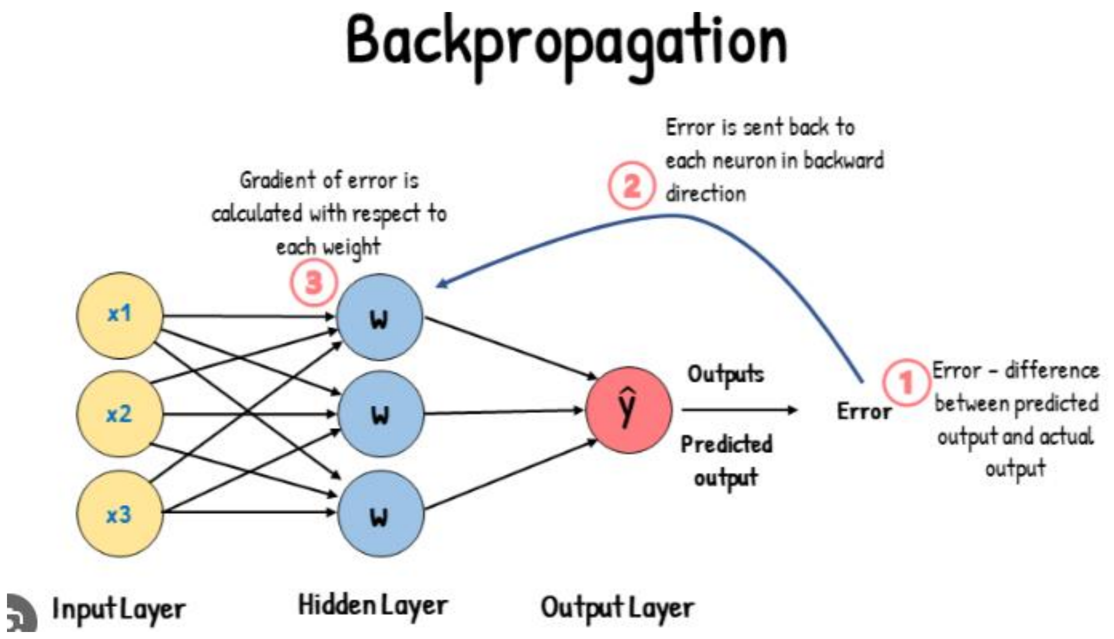
Where:

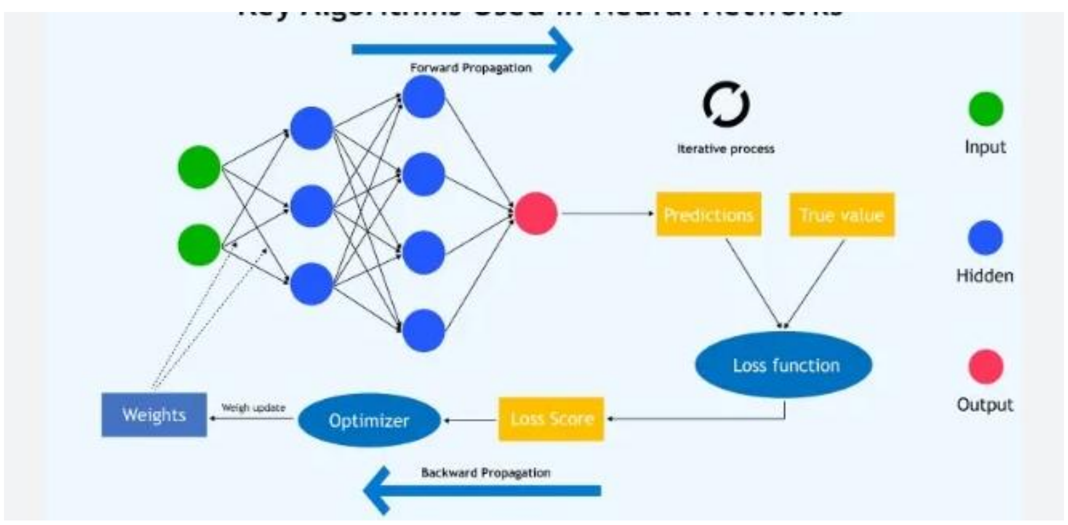
- η = learning rate
- $\frac{\partial L}{\partial w}$ = gradient

Step 4: Repeat

Do this for all weights in all layers.

Repeat for many epochs until error is minimized.


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer



SUMMARY

Concept	Description
Architecture	Input layer, hidden layers, output layer
Forward Propagation	Computes output using weighted sums + activations
Loss Function	Measures how wrong the output is
Backpropagation	Computes gradients and updates weights to reduce error
Training	Repeating forward & backward passes until accuracy improves

Activation Functions

Activation functions introduce **non-linearity** into a neural network, allowing it to learn complex patterns.

They decide **whether a neuron should be activated or not** based on the input.

Type	Definition	Activation Functions	Used For
Linear	The output is directly proportional to the input.	f(x) = x	Regression
Non-linear	These introduce **non-linearity**, allowing neural networks to learn complex relationships.	Sigmoid, Tanh, ReLU, Leaky ReLU, ELU, GELU	Hidden layers, deep networks
Probability-based	Converts raw output scores into a **probability distribution**.	Softmax	Multi-class classification

Why Activation Functions Are Important

Activation functions play a **critical role** in making neural networks powerful, expressive, and capable of learning complex patterns.

- Activation functions introduce **non-linearity** into neural networks, enabling them to learn complex patterns and decision boundaries.
- Activation functions provide gradients that allow weights to be updated.
  With activation functions:

        - Layer outputs become transformed and richer
        - Each layer learns a different level of abstraction
- Hierarchical learning is impossible without activation functions.

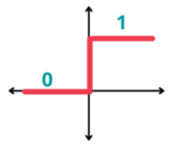
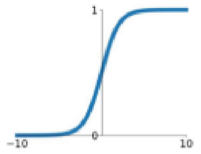
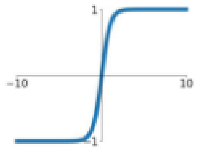
  Without activation:

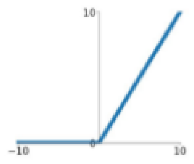
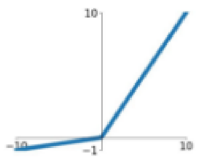
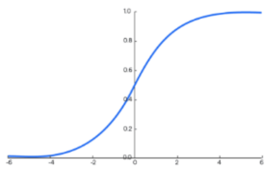
    - Gradients are constant
    - Learning is not possible

  With activation:

    - Gradients flow through layers
    - Network learns through iteration
- Control Output Ranges

Without activation functions, the entire network would behave like a simple linear model, regardless of how many layers it has. They help in controlling output ranges, enabling backpropagation, and making deep learning possible.

Activation Function		**Formula**	**Output Range**	**Graph**	**Pros**	**Cons**	**Used In**
Step function		$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$	0 or 1		1. Simple and easy to compute. 2. Good for **binary decisions** (Yes/No, True/False).	1. Not differentiable → Cannot be used in backpropagation. 2. Outputs are only 0 or 1 → not suitable for deep networks. 3. No probability-like output.	1. Perceptron 2. Simple binary classifiers 3. NOT used in modern deep learning (because differentiation is required)
Sigmoid		$\sigma(x) = \frac{1}{1 + e^{-x}}$	0 to 1		1. Good for probability **outputs.** 2. Smooth gradient.	1. Vanishing gradient problem 2. Outputs **not zero-centered**	Binary classification
Tanh		$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	−1 to 1		1. Zero-centered → faster convergence. 2. Stronger gradients than sigmoid.	1. Still suffers from **vanishing gradients.** 2. Can saturate for very high/low inputs.	Hidden layers (older models)

ReLU ReLU (Rectified Linear Unit)		$\mathrm{ReLU}(x) = \max(0, x)$	0 to ∞		1. **Solves vanishing gradient** for positive values. 2. Computationally efficient. 3. Leads to faster training.	1. **Dying ReLU problem**: neurons may get stuck at 0 forever if weights update negatively.	CNNs, deep networks
Leaky ReLU		$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$	−∞ to ∞		1. Allows gradient flow even for negative values. 2. Prevents neurons from dying.	1. α is a hyperparameter → needs tuning. 2. Not zero-centered for negative values.	Advanced deep networks
Softmax		$\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$	0 to 1		1. Converts outputs to probabilities 2. Used for multi-class classification 3. Differentiable	1. Can be affected by very large values 2. Can produce over-confident predictions.	1. Output layer of multi-class neural networks 2. CNNs for image classification 3. NLP models

Cost Function

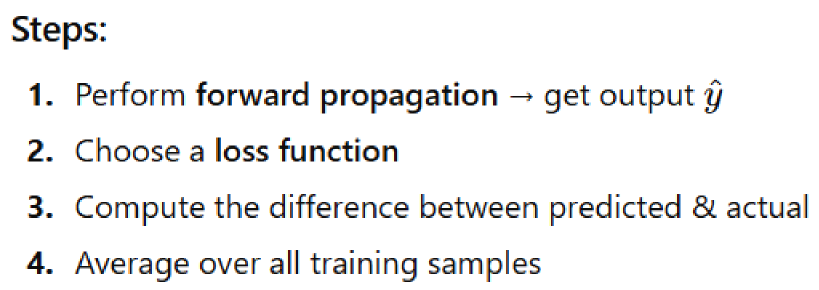
A Cost Function **(also called** Loss Function **or** Error Function**) measures** how far the neural network's predictions are from the actual (true) values**.**
It evaluates the performance of the model.

Purpose

- Lower cost = better model
- Higher cost = poor predictions

How to Measure Loss?

Loss is measured by comparing **predicted output (y^\hat{y}y^)** with **actual output (y)**.



Steps:

1. Perform **forward propagation** → get output $\hat{y}$
2. Choose a **loss function**
3. Compute the difference between predicted & actual
4. Average over all training samples

How to Reduce Loss?

Loss is reduced by:
✓ Updating the weights and bias

- The model adjusts weights so that future predictions are closer to the correct values.

✓ Using Gradient Descent

- A mathematical optimization technique to minimize loss.

✓ Performing Backpropagation

- Computes how much each weight contributed to the error.

✓ Training for multiple epochs

- Repeatedly reducing error after each forward + backward pass.

✓ Using good learning rate

- Small learning rate → slow learning
- Large learning rate → unstable learning

Gradient Descent

Gradient Descent is an **optimization algorithm** that reduces the cost function by **updating weights in the direction of steepest descent** (negative gradient).

Goal

Minimize the loss function **J(w)**.

Formula

$$w_{\text{new}} = w_{\text{old}} - \eta \frac{\partial J}{\partial w}$$

Where:

- w = weight
- η = learning rate
- $\frac{\partial J}{\partial w}$ = gradient (slope of cost function)

How Gradient Descent Works

Steps:

1. Start with random weights
2. Compute predictions using forward propagation
3. Calculate loss
4. Compute gradients (partial derivatives)
5. Update weights using gradient descent formula
6. Repeat until loss is minimized

Concept	**Meaning**
Cost Function	Measures how wrong the model is
Loss Measurement	Compare predicted vs actual values
Reduce Loss	Update weights using backpropagation
Gradient Descent	Optimization method to minimize loss

What is TensorFlow?

Topic	**Summary**
What is TensorFlow?	Open-source deep learning framework using computational graphs
Why TensorFlow?	Scalable, fast, supports GPUs/TPUs, strong ecosystem
Ecosystem	Core TF, Keras, TFLite, TFJS, TensorBoard, TFX
Architecture	Tensors, computational graph, execution engine, hardware
Program Elements	Tensors, Variables, Ops, Layers, Optimizers, Loss functions

Tensors

- Basic data type
- A tensor is a multi-dimensional array used to represent data in machine learning and deep learning frameworks
- Tensors are the core data structure used for inputs, outputs, weights, and everything inside a neural network

Why Tensors?

- Efficiently represent **N-dimensional data**
- Are optimized for **GPU/TPU computation**
- Can store all types of data (images, audio, text embeddings)
- Enable fast mathematical operations (matrix multiplication, convolution)

Types of Tensors

Tensor Type	Dimension	Example	Shape
Scalar	0-D	3.14	()
Vector	1-D	[1, 2, 3]	(3,)
Matrix	2-D	[[1, 2], [3, 4]]	(2, 2)
3-D Tensor	3-D	Color image	(height, width, channels)
4-D Tensor	4-D	Batch of images	(batch, h, w, c)
n-D Tensor	n-D	Any multi-dimensional array	Depends

Creation of Tensors in TensorFlow

```
import tensorflow as tf

# Constant tensor
t1 = tf.constant([[1, 2], [3, 4]])

# Variable tensor (trainable)
t2 = tf.Variable([1.0, 2.0, 3.0])

# Random tensor
t3 = tf.random.normal(shape=(3, 3))
```

Tensor Attributes

Every tensor has:

1. **Shape** → size of each dimension
2. **Rank** → number of dimensions
3. **Data type** → int32, float32, etc.
4. **Device** → CPU/GPU location

Tensor Operations

- Addition, subtraction
- Matrix multiplication
- Reshape
- Transpose
- Broadcasting

What is Keras?

Keras **is a** high-level deep learning **API** written in Python and integrated with TensorFlow **(as tf.keras).**

It allows **developers to** build, train, evaluate, and deploy neural networks easily **with** minimal code.

Key features of Keras

- User-friendly and modular
- Supports multiple backends (primarily TensorFlow)
- Fast prototyping
- Easy to debug (eager execution in TF 2.x)
- Large collection of pre-built layers and models

Keras provides three powerful programming models to build deep learning architectures:

1. Sequential Model
2. Functional API Model
3. Model Subclassing API

Each model type offers different flexibility and is used based on the complexity of the neural network.

◈ 1. Sequential Model

Definition

The **Sequential model** allows you to stack layers in a **linear, step-by-step** fashion.

It is best for models where each layer has **one input and one output**.

```
Input → Layer1 → Layer2 → Layer3 → Output
```

When to Use Sequential

Use when:

- Model has no branching
- Model has no skip connections
- Model is simple feed-forward (MLP, simple CNN)

Advantages

- Very easy to use
- Compact code
- Good for beginners
- Suitable for rapid prototyping

Limitations

- Cannot handle complex models (multi-input/output)
- Cannot create architectures like ResNet, U-Net, Transformers

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

◆ 2. Functional API Model

Definition

The **Functional API** allows the creation of **complex, non-linear architectures** by directly connecting layers like functions.

This enables models with:

- Multiple inputs
- Multiple outputs
- Skip connections
- Shared layers
- Graph-like architectures

When to Use Functional API

- Autoencoders
- Multi-input networks
- Multi-output networks
- ResNet, Inception models
- Attention models
- Graph or branching models

Advantages

- More flexible than Sequential
- Can build any DAG (directed acyclic graph) model
- Maintains clarity with visual model summary

Limitations

- Slightly more complex than Sequential
- Requires care in connecting layers correctly

◈ 3. Model Subclassing API

Definition

Model Subclassing allows you to **create your own model class** by inheriting from `tf.keras.Model`.

This method offers **maximum flexibility** and supports dynamic architectures.

When to Use Model Subclassing

- Implementing research papers
- Building RNNs manually

- Custom training loops (GANs, RL)
- Highly custom forward-pass logic
- Models with loops or conditional layers

Advantages

- Most flexible
- Supports dynamic computation graphs
- Great for experimenting with new architectures

Limitations

- Harder to debug
- No automatic layer connectivity checks
- Model summary may be less detailed

kERAS LAYERS

Category	**Examples**
Dense Layers	Dense()
Convolution Layers	Conv2D, Conv1D, Conv3D
Pooling Layers	MaxPooling2D, AvgPooling2D
Recurrent Layers	LSTM, GRU, SimpleRNN
Normalization Layers	BatchNormalization, LayerNormalization
Embedding Layers	Embedding()
Dropout	Dropout()
Flatten / Reshape	Flatten(), Reshape()

Custom Keras Layers

Definition

Custom layers allow developers to implement **new operations** not available in built-in Keras layers.

Why do we have to Flatten the Input Data?

Definition of Flattening

Flattening converts multi-dimensional input data (e.g., images) into a **1D vector** before passing it into a Dense Layer.

Example:

An image of shape **(28, 28)** → flattened to **(784)**.

Why Flatten? (Reason)

Dense (fully connected) layers expect **1D input vectors**, not matrices or tensors.

Keras Dense Layer

Overview

A **Dense layer** (fully connected layer) connects every input neuron to every output neuron.

It is represented as:

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

Where:

- W = weights
- b = bias
- f = activation function

Parameters of Dense Layer

Parameter	Meaning
units	Number of neurons in the layer
activation	Activation function (relu, sigmoid, softmax, etc.)
kernel_initializer	How weights are initialized
bias_initializer	How bias is initialized
kernel_regularizer	L1/L2 regularization
use_bias	Whether to include bias (True/False)
input_dim/input_shape	Define input dimension for the first layer

```
Dense(units=64, activation='relu', input_shape=(784,))
```

How Dense Layer Works (Operation)

Step-by-step:

1. Takes an input vector:

$$x = [x_1, x_2, ..., x_n]$$

2. Every input is multiplied by a weight.
3. All weighted inputs are added with bias.
4. Activation function is applied.

Mathematically:

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

Dense Layers = **matrix multiplication + activation**

Shallow vs Deep Neural Networks

Feature	Shallow Neural Network	Deep Neural Network
Number of Hidden Layers	1 hidden layer	2 or more hidden layers
Complexity	Low `Input → Hidden Layer → Output`	High `Input → H1 → H2 → H3 → ... → Output`
Ability to Learn Patterns	Limited, simple patterns	Learns complex, hierarchical patterns
Data Requirement	Works with small datasets	Requires large datasets
Training Time	Fast	Slow
Computational Power	Low requirement	High (GPU/TPU often needed)
Risk of Overfitting	Lower	Higher (needs regularization)
Performance on Images/Text	Poor	Excellent
Suitability	Simple tasks	Complex tasks (vision, NLP, speech)
Examples	Basic MLP	CNNs, LSTMs, Transformers

Keras Optimizers

An **optimizer** updates the weights of a neural network during training to **minimize the loss**.

➞ *Optimizer adjusts weights so the model learns better.*

Optimizer	Good For	Notes
SGD	Simple models	Slow but stable
Momentum	Faster SGD	Adds direction memory
RMSprop	RNNs/LSTMs	Adaptive LR
Adam	Almost all DL tasks	Most used
Adagrad	Sparse data	NLP
Nadam	Fast training	Adam + Nesterov

Keras Metrics

A **metric** measures the performance of the model.

➞ It does **not affect training**, only used for evaluation/monitoring.

Example: accuracy, precision, recall.

Problem Type	**Metrics**
Binary Classification	accuracy, binary_accuracy, precision, recall
Multi-Class Classification	categorical_accuracy, top_k_accuracy
Regression	MSE, MAE, RMSE

Keras Losses

Loss function measures **how wrong** the model's predictions are.

➞ Optimizer tries to **minimize loss**.

Task	Loss Function	Notes
Binary Classification	Binary Crossentropy	Sigmoid output
Multi-class (one-hot)	Categorical Crossentropy	Softmax output
Multi-class (integer labels)	Sparse Categorical Crossentropy	Faster
Regression	MSE, MAE	Numeric prediction
SVM style	Hinge	Margin-based
Generative models	KL Divergence	VAEs

Keras Callbacks

A **callback** is a function or object that is executed **at specific stages of training**, such as:

- At the end of each epoch
- At the beginning/end of training
- When the model improves

Callbacks help in monitoring, debugging, saving models, early stopping, etc.

Callback	Purpose
ModelCheckpoint	Saves model during training
EarlyStopping	Stops when model stops improving
ReduceLROnPlateau	Reduces LR automatically
TensorBoard	Visualization of training
CSVLogger	Saves logs to file
TerminateOnNaN	Stops on NaN loss

What is TensorBoard?

TensorBoard is a **visualization tool** built into TensorFlow.
It allows developers to **monitor**, **analyze**, and **understand** the behavior of neural networks during training.