DEEP LEARNING

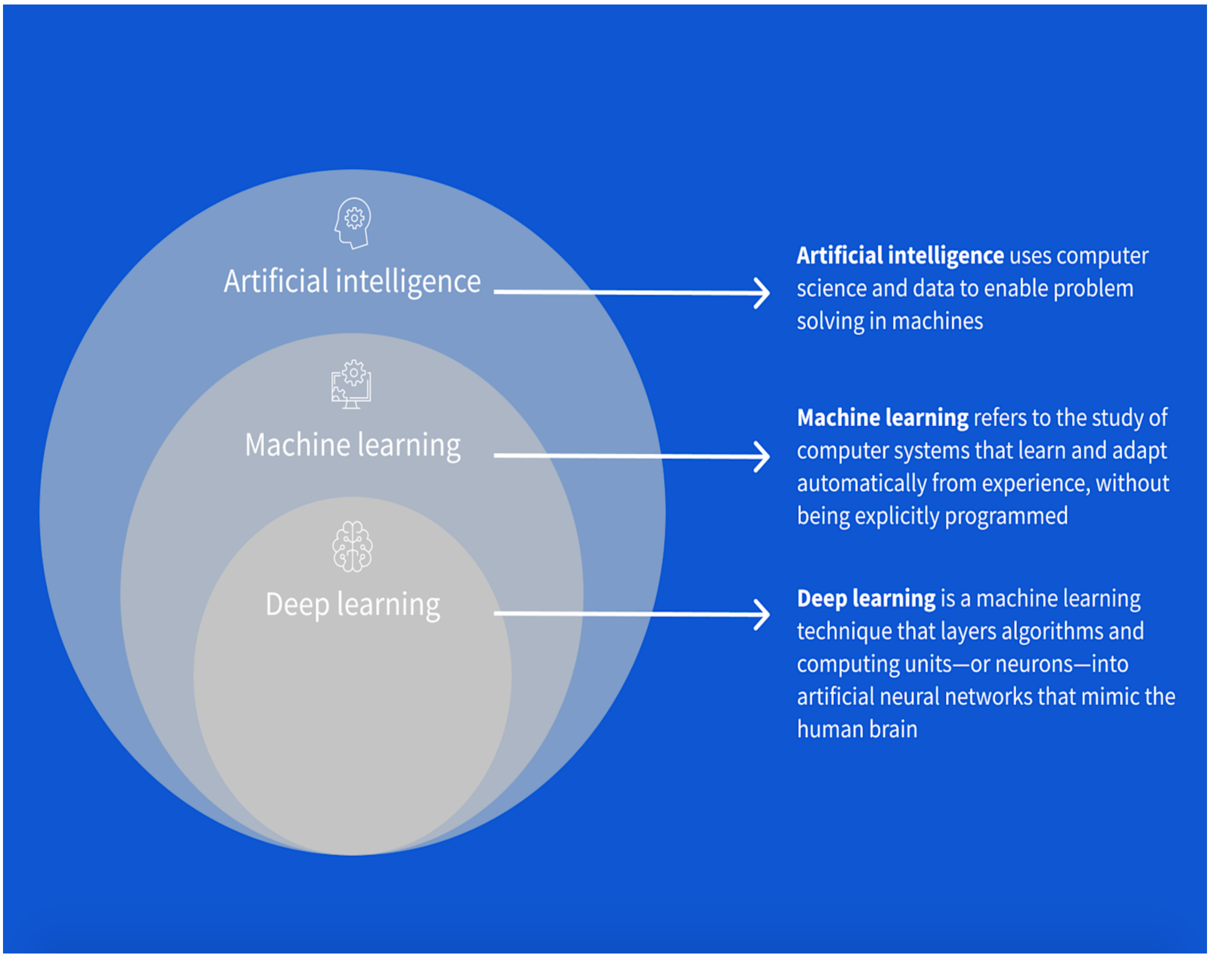
HANDOUT: INTRODUCTION

Contents

1. Fundamentals of Neural Networks and Deep Learning

1.1 What is Deep Learning?

Deep Learning is a subset of **Machine Learning** inspired by the structure and function of the human brain called **Neural Networks (NNs)**. It automatically discovers useful representations in data using **multiple layers**.



Task : Image Classification Using Neural Networks

Q: You are assigned to build a model that classifies handwritten digits (0–9) from images. Why should you use a neural network instead of a simple linear model?

A: A neural network can learn **complex, non-linear relationships** in image pixels. A linear model can only draw straight-line boundaries, which are insufficient for recognizing curved digits. Neural networks use **layers and activation functions** to capture these patterns.

1.2 Neural Network Structure

What is a Neural Network?

A Neural Network is a mathematical model designed to recognize patterns and make decisions by mimicking the way the human brain works.

Just like your brain learns from experience, a neural network learns from data.

What is a Neuron in a Neural Network?

A neuron (also called a node or unit) is the basic building block of a neural network just like a cell is the basic unit of the body.

It's a mathematical function that takes some inputs, processes them, and produces an output.

Layers in a Neural Network

A. Input Layer

- This is where data enters the network.
- Each neuron in the input layer corresponds to one feature in the dataset.
- Example: For housing prices, you might have 6 features → 6 neurons.

B. Hidden Layers

- These layers process the inputs and extract features.
- Each neuron here receives input from all neurons in the previous layer.
- The more hidden layers → the deeper the network.
- You can have 1, 2, or even 100+ hidden layers in deep learning.

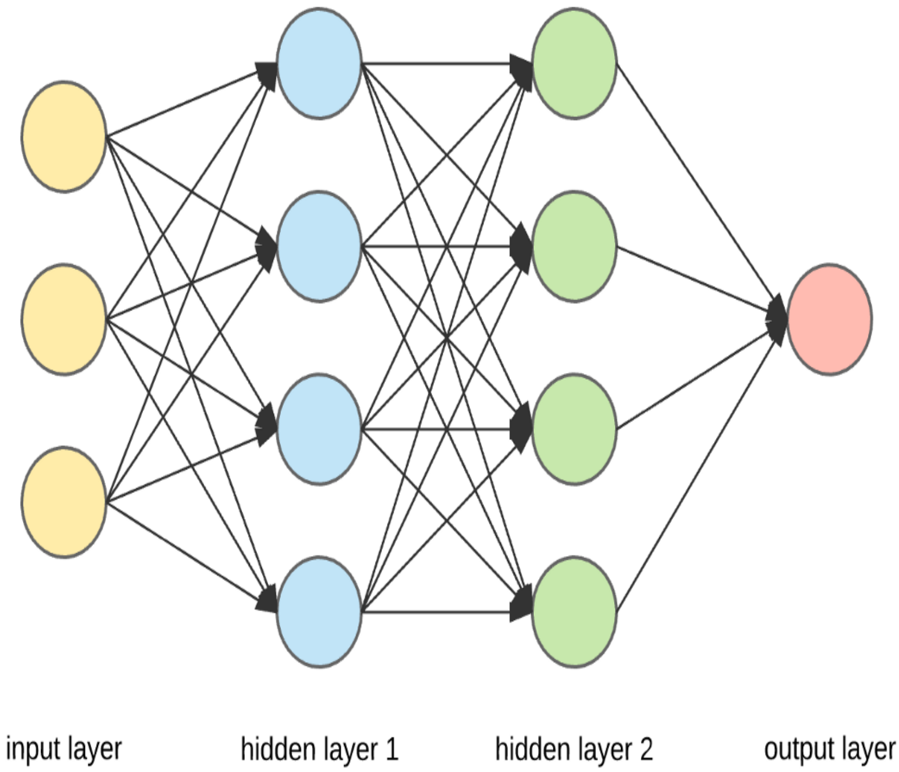
Task : Designing a Multi-layer Network

Q: You are asked why multiple hidden layers are better than one large layer. What would you explain?

A: Multiple hidden layers enable the model to learn **hierarchical features** — low-level features (like edges) in early layers, and high-level features (like shapes or objects) in deeper layers. A single large layer cannot learn such structured representations efficiently.

C. Output Layer

- Produces the final prediction.
- Number of neurons depends on the task:
  - 1 neuron for binary classification or regression
  - Multiple neurons for multi-class classification



- **Input Layer** – receives raw data (features)
- **Hidden Layers** – perform computations via weighted connections
- **Output Layer** – gives final predictions (classification/regression)
- **Activation Functions** – introduce non-linearity (e.g., ReLU, Sigmoid)

Activation Function:

This introduces non-linearity to the network. Without it, the ANN would behave like a simple linear model.

Common activation functions:

Name	Use Case
ReLU	**Hidden layers, fast learning**
Sigmoid	**Binary output**
Tanh	**Outputs between -1 and 1**
Softmax	**Multi-class classification**

Connections (Weights and Biases)

- **Weights** determine the strength of connections between neurons.
- **Bias** is an additional value added to the sum, giving the model flexibility.
- During training, weights and biases are adjusted to reduce error.

Example:

Let's say you have 3 inputs:

- Temperature = 30°C
- Humidity = 70%
- Pressure = 1010 hPa

Each input is multiplied by a weight, and then a bias is added:

$$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + (x_3 \cdot w_3) + b$$

Then, this result z is passed through an activation function (like sigmoid or ReLU) to produce the output.

Let's assume:

- Inputs: x1 = 1.0, x2 = 2.0, x3 = 3.0
- Weights: w1 = 0.5, w2 = -1.0, w3 = 0.8
- Bias: b = 0.2

Step 1: Calculate weighted sum

$$z = (1.0 \cdot 0.5) + (2.0 \cdot -1.0) + (3.0 \cdot 0.8) + 0.2 = 0.5 - 2.0 + 2.4 + 0.2 = 1.1$$

Step 2: Apply activation (say sigmoid):

$$\text{output} = \frac{1}{1 + e^{-1.1}} \approx 0.75$$

So, the neuron’s output is 0.75 — this will be passed to the next layer.

Why Is It Important?

Neurons learn features from data.

Multiple neurons work together to form layers that can detect patterns.

The network adjusts weights inside neurons during training to improve predictions.

1.3 Forward Propagation

Each neuron computes:

Output = Activation(Wx + b)

Where:

- W = weights,
- x = input,
- b = bias

1.4 Loss Function

Measures the difference between predicted and actual value. Examples:

- Mean Squared Error (MSE)
- Cross-Entropy Loss

1.5 Backpropagation

Backpropagation updates the model weights by minimizing loss using optimization techniques like **Gradient Descent**.

1.6 The Core Building Block: The Neuron

Think of a neuron as a tiny calculator.

What it does:

1. Takes one or more inputs
2. Multiplies them by weights
3. Adds a bias
4. Applies a function to decide the output (called activation)

Mathematically:

For inputs x_1, x_2, x_3 with weights w_1, w_2, w_3 and bias b:

$$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + (x_3 \cdot w_3) + b$$

Then apply an **activation function**, like ReLU or sigmoid:

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

How a Neural Network Learns (In Steps)

1. Initialization: Start with random weights
2. Forward Pass: Data flows through the network → prediction
3. Loss Calculation: Compare prediction to actual result
4. Backward Pass: Adjust weights using error (called backpropagation)
5. Repeat: Do this over many rounds (epochs) until the error becomes very small

Why Neural Networks Are Powerful

1. They learn patterns from data — you don't need to program rules manually
2. They handle non-linear relationships
3. They improve with more data
4. They're the foundation of Deep Learning (like CNNs, RNNs, Transformers)

1. Simple Neural Network in Python using NumPy

This is a very basic neural network with 1 hidden layer, built from scratch using only NumPy — no libraries like TensorFlow or Keras. It helps students see what's happening behind the scenes.

Goal: Learn XOR logic using a neural network

```
import numpy as np
# XOR input and output
```

```
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])
```

```
#X contains all possible input combinations of XOR.
#y contains the correct output labels (truth table of XOR).

# Set seed for reproducibility
```

```
np.random.seed(1)
```

```
# Sigmoid activation function and its derivative
```

```
def sigmoid(x):
   return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
   return x * (1 - x)
```

#**Sigmoid** maps values between 0 and 1.

#**Sigmoid Derivative** is used during training (backpropagation) to update the weights.

Initialize weights randomly

```
input_layer_neurons = 2
hidden_layer_neurons = 2
output_neurons = 1
```

#We use 2 input neurons (since XOR has 2 inputs).

#We'll use 2 neurons in the **hidden layer**.

#We'll have 1 **output neuron**.

Weights

```
wh = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
bh = np.random.uniform(size=(1, hidden_layer_neurons))
wo = np.random.uniform(size=(hidden_layer_neurons, output_neurons))
bo = np.random.uniform(size=(1, output_neurons))
```

#wh: Weights between input and hidden layer

#bh: Bias for hidden layer

#wo: Weights between hidden and output layer

#bo: Bias for output layer

Training loop

```
for epoch in range(10000):
```

#This loop trains the network **10,000 times**. For each round:

Forward pass

```
    hidden_input = np.dot(X, wh) + bh
    hidden_output = sigmoid(hidden_input)
```

```
    final_input = np.dot(hidden_output, wo) + bo
    output = sigmoid(final_input)
```

#Multiply inputs by weights → add bias → apply activation → get output

This simulates how neurons fire in the brain

```
    # Backpropagation
    error = y – output
#How far is the prediction from the actual output?
    d_output = error * sigmoid_derivative(output)
#This calculates how much the output layer needs to change.
    error_hidden = d_output.dot(wo.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)
#Backpropagate the error to the hidden layer.
```

Update weights

```
    wo += hidden_output.T.dot(d_output)
    bo += np.sum(d_output, axis=0, keepdims=True)
    wh += X.T.dot(d_hidden)
    bh += np.sum(d_hidden, axis=0, keepdims=True)
```

#The network **learns by adjusting weights and biases** to reduce error.

Final predictions

```
print("Predicted Output:")
print(np.round(output, 3))
```

#After training, the model should print predictions close to:

#After 10,000 iterations, the network should learn the XOR function:

```
Predicted Output:
[[0.01]
 [0.98]
 [0.98]
 [0.02]]
```

This shows the network has learned XOR:

- **0 XOR 0 = 0**
- **0 XOR 1 = 1**
- **1 XOR 0 = 1**
- **1 XOR 1 = 0**

What are we trying to do?

We're training a neural network to learn the XOR logic function:

Input A	Input B	Output (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

Q1:What is input_dim?

When creating a neural network (especially in libraries like Keras), input_dim tells the network how many input features each data sample has.

Example:

Suppose your data looks like this:

```
X = [[65, 1.75, 22],
     [70, 1.80, 24],
     [60, 1.65, 20]]
```

Here, each sample has 3 values:

- weight (kg)

- height (m)
- age (years)

So, input_dim = 3

Why is it needed?

- **The neural network must know how wide the input is to build the correct number of weights.**
- **The first layer only needs this — later layers infer shape from previous ones.**

Q2. Why do we use .dot() instead of * in NumPy while doing matrix multiplication?

Answer:

*** in NumPy does element-wise multiplication, not matrix multiplication.**

.dot() or @ is used for matrix product, which is what we need in neural networks to combine inputs with weights.

Q3. What is the shape of the output after this code?

```
X = np.array([[1, 2],
              [3, 4],
              [5, 6]])
W = np.array([[0.1],
              [0.2]])

output = np.dot(X, W)
```

Answer:

- **X: shape (3, 2) → 3 samples, 2 features**
- **W: shape (2, 1) → 2 input weights, 1 output neuron**
- **output: shape (3, 1)**

So each input gets passed through a weighted sum → single output per row.

Q4. Why do we use activation functions?

Answer:

Without activation functions, the network would just be doing linear math — like a basic regression.

Activation functions let it learn non-linear patterns — like curves, boundaries, XOR, etc.

Q5. What's the purpose of sigmoid_derivative() during training?

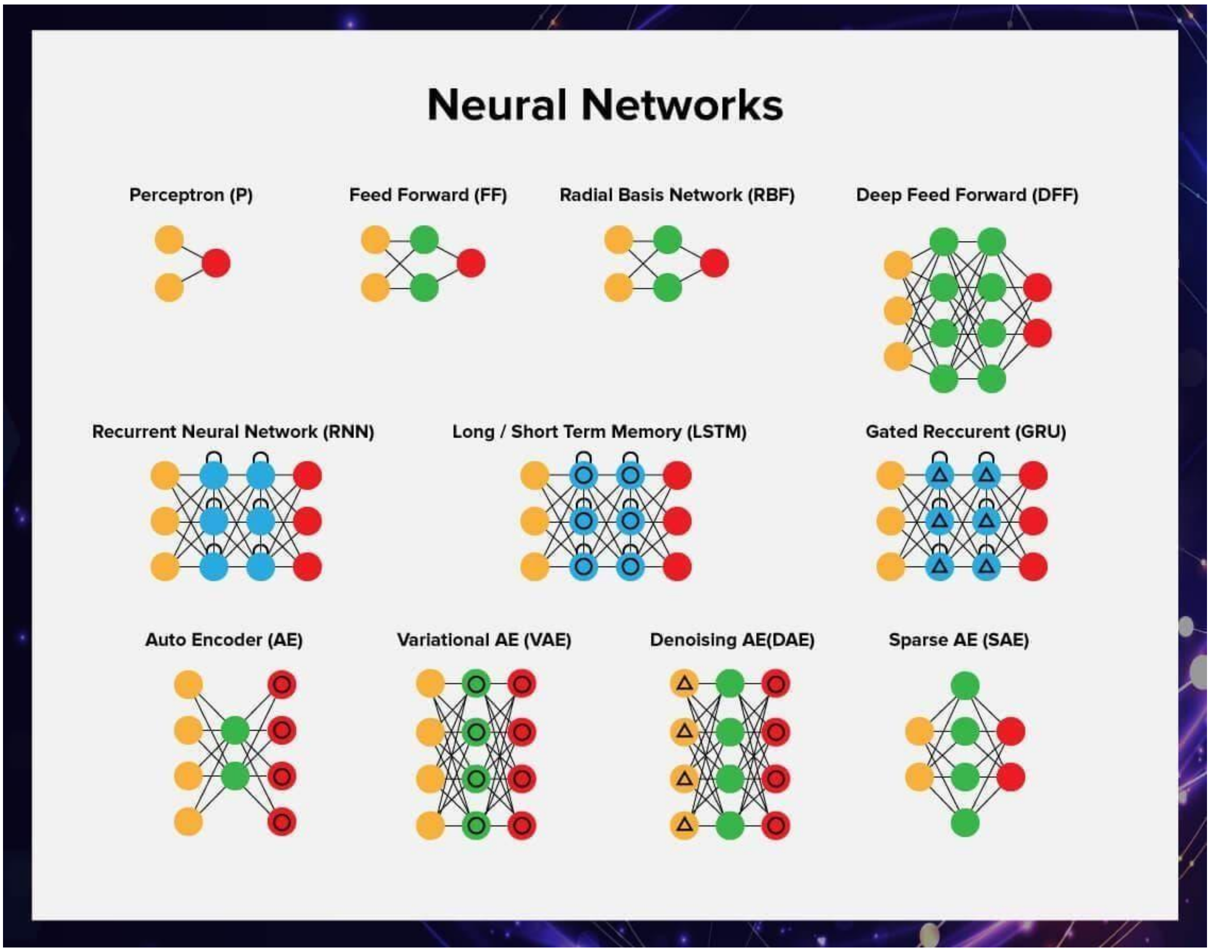
Answer:

To update weights, the network needs to know how sensitive the output is to changes — this is called the gradient.

The derivative tells us how much to adjust weights to reduce the error.

1.6 Types of Neural Networks

Network	**Description**	**Use Cases**
Feedforward NN	Basic ANN, no cycles	Simple classification
CNN	Convolutional Neural Network	Image recognition
RNN	Recurrent Neural Network	Time series, NLP
LSTM/GRU	Variants of RNN	Long sequences
GAN	Generative Adversarial Network	Image synthesis



1. Perceptron (The First Building Block)

What is it?

A **perceptron** is the **simplest type of neural network** — a single neuron.

How it works:

- Takes input features (like age, height)
- Multiplies them by weights, adds a bias
- Passes through an activation (like step or sigmoid)
- Gives a **binary output** (0 or 1)

Use Case: Classifying something into 2 categories (like spam/not spam)

Limitation:

Can't solve complex problems like XOR — because it's linear.

2. Feedforward Neural Network (FFN) (Basic Multi-layer)

What is it?

A **neural network with multiple layers** where data flows **only forward** from input to output.

Structure:

Input → Hidden Layer(s) → Output

- No loops or memory
- Each layer learns **features** from data

Use Case: Any supervised task like prediction, classification

3. Deep Feedforward Network (DFF)

What is it?

An **FFN with multiple hidden layers** — this depth allows it to learn **more complex patterns**.

Why "deep"?

Because it has **more than 1 hidden layer** (could be 10, 100, etc.)

Use Case: Image recognition, fraud detection, etc.

4. Artificial Neural Network (ANN)

What is it?

A **general term** for all neural networks built using artificial neurons.

Includes:

- Perceptrons
- FFNs
- DFFs
- CNNs, RNNs, etc.

You can think of ANN as the **family name**.

5. Convolutional Neural Network (CNN) (Sees like a human)

What is it?

A specialized ANN designed to **process images**.

How it works:

- Uses **convolutional layers** that scan images for patterns (edges, colors, textures)
- Learns spatial features: what's near what, where edges occur, etc.
- Later layers learn complex patterns (like shapes, faces, objects)

Use Case: Image classification, face detection, medical imaging

6. Recurrent Neural Network (RNN) (Remembers)

What is it?

A neural network that has **loops** — it can **remember previous inputs**.

Key Feature:

It uses its **own previous output as input**, like short-term memory.

Use Case:

- Time-series prediction (like stock prices)
- Language modeling
- Text generation
- Speech recognition

Limitation:

- Struggles with long-term memory (solved by LSTM and GRU)

7. Autoencoder (Learns compressed versions)

What is it?

A neural network that tries to **rebuild its input** — used for **data compression and noise reduction**.

Structure:

Input → Encoder → Bottleneck → Decoder → Output

- The bottleneck forces the network to **compress the input**
- The decoder tries to reconstruct the original data

Use Case:

- Dimensionality reduction
- Image denoising
- Feature extraction
- Anomaly detection

8. Generative Adversarial Network (GAN) (The Artist + The Critic)

What is it?

Two networks:

- **Generator**: creates fake data (like fake images)
- **Discriminator**: tries to tell if it's real or fake

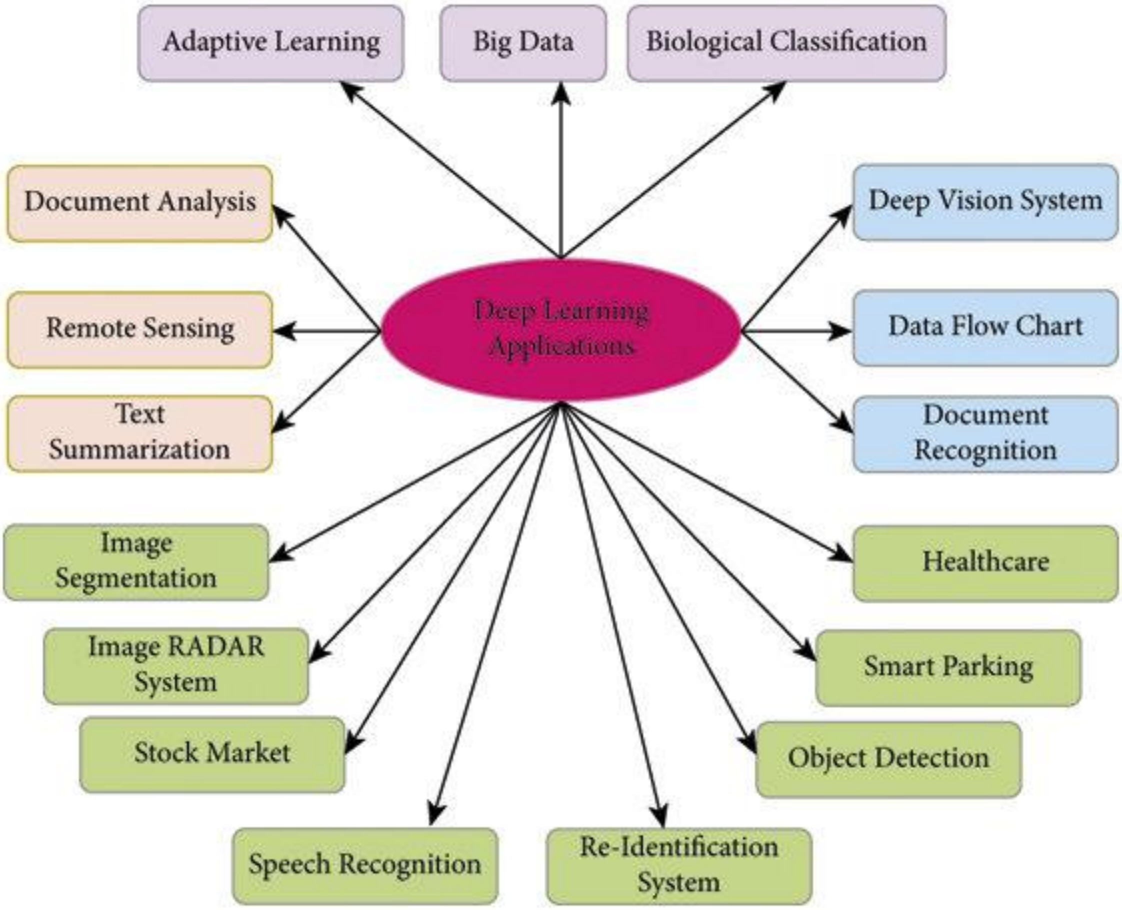
They **compete** — like a forger and a detective. Over time, the generator gets better at making realistic data.

Use Case:

- Image generation (e.g., deepfakes)
- Style transfer
- Super-resolution
- Data augmentation

2. Deep Learning Applications

Domain	Applications
Computer Vision	Image classification, Object detection, Face recognition
Natural Language Processing	Machine translation, Sentiment analysis, Chatbots
Healthcare	Disease detection, Radiology reports generation
Finance	Fraud detection, Stock prediction
Autonomous Vehicles	Lane detection, Obstacle avoidance
Gaming	Strategy learning, Environment interaction



Examples:

- **Google Translate** – Uses sequence-to-sequence RNNs/LSTMs
- **Tesla Autopilot** – Combines CNNs with sensors for road detection
- **AlphaFold** – Predicts 3D protein structures (DL + Reinforcement Learning)

Task: Build a Cancer Detection System

Q: You are working on a deep learning model to detect cancer from MRI scans. Why is deep learning suitable for this task?

A: Deep learning (CNNs) can automatically extract important visual features from medical images without manual intervention. It's capable of detecting **minute differences** and **complex patterns** in MRI scans that are often missed by traditional methods.

Task: Translate English to French

Q: You've been asked to develop an English-to-French translation system using deep learning. What type of model will you use and why?

A: RNNs or **Transformers** are appropriate. These models understand **context** and **word dependencies**, which are essential for translation. Transformers are better due to **attention mechanisms** and parallel processing.

3. Deep Learning Frameworks

3.1 What Are Deep Learning Frameworks?

Definition:

A deep learning framework is a software library or toolkit that provides pre-built tools to help you:

- Design neural networks
- Train them using data
- Perform predictions and evaluations

You can think of them like "Lego kits" for building AI models — they handle the math, optimization, and hardware for you.

Why Do We Need Frameworks?

Training deep learning models involves:

- Complex matrix math
- Backpropagation
- GPU acceleration
- Optimizers (SGD, Adam, etc.)
- Custom architectures (CNNs, RNNs, etc.)

Simple Analogy

If deep learning is cooking, then frameworks like TensorFlow or PyTorch are the microwave + oven + recipe book — they give you the tools to do it faster, better, and at scale.

Doing all this from scratch is:

- Time-consuming
- Error-prone
- Hard to scale

Frameworks make it easier, faster, and more reliable to build and deploy deep learning solutions.

Key Features of a Deep Learning Framework:

Feature	**Description**
Automatic differentiation	Handles gradients & backprop for you
GPU/TPU support	Runs faster on special hardware
Model building tools	Easy layers, activations, loss functions
Training utilities	Optimizers, batch processing, early stopping
Pretrained models	Load and fine-tune models like ResNet, BERT, GPT
Visualization tools	Track training progress (loss, accuracy)
Export & Deployment	Convert models to run on mobile, web, edge devices

Most Popular Deep Learning Frameworks

Framework	**Language**	**Highlights**	**Best For**
TensorFlow	Python, C++	Backed by Google, powerful & production-ready	Deployment, scalability
PyTorch	Python	Dynamic graphs, Pythonic, easy to debug	Research, education, prototyping
Keras	Python	High-level API, runs on TF or Theano	Beginners, rapid prototyping
JAX	Python	High-performance NumPy + auto-diff	Research, fast math on GPUs
MXNet	Python	Efficient and scalable, used by AWS	Large-scale training
Caffe	C++	Fast for vision tasks	Image classification
ONNX	N/A	Model format, not a framework	Interoperability across platforms

Q1: You are a research student who needs flexibility in model development. Which framework do you choose and why?

A: PyTorch, because it offers dynamic computation graphs and easier debugging.

Q2: You're working on a mobile app using DL. Which framework suits deployment?

A: TensorFlow (with TensorFlow Lite) for efficient deployment on edge devices.

Q3: Why use Keras for beginners in Deep Learning?

A: Keras abstracts complex operations and makes it easy to build models with fewer lines of code.

Comparison: PyTorch vs TensorFlow Implementations

Problem Statement:

A neural network with:

- Input: 10 features
- Hidden Layer: 64 neurons, ReLU
- Output: 1 neuron, Sigmoid (binary output)

Key Differences: PyTorch vs TensorFlow

Feature	PyTorch	TensorFlow / Keras
Graph Mode	Dynamic (eager)	Static + Eager via tf.function
Code Style	More Pythonic, OOP-style (custom classes)	High-level API (Sequential/Functional)
Training Loop	Manual control (flexible)	Built-in .fit() API
Debugging	Easy via Python tools	Harder unless using Eager
Popularity in Research	Preferred (customization)	Preferred in production
Deployment	TorchScript, ONNX	TensorFlow Lite, TF.js, TF Serving

What Is a Computation Graph?

A **computation graph** is a **map of operations** that a neural network needs to perform to compute an output.

It's like a **flowchart** of math operations — a graph where:

- **Nodes** = operations (like add, multiply, sigmoid)
- **Edges** = data flow between operations

It shows how inputs (like x1, x2) move through weights, biases, and activations to become outputs.

Simple Example

Let's say your model does:

$$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + b$$

$$a = \text{sigmoid}(z)$$

Each step (multiply, add, activate) is a **node** in the graph. The values **flow** from inputs to outputs — that's the **graph flow**.

Why Do We Use Computation Graphs?

Because:

- Neural networks are just **math equations**
- We need a way to **organize the sequence** of operations
- Backpropagation (used for learning) needs to **trace the path of calculations backward**

Types of Computation Graphs

Type	How It's Built	Example Frameworks
Static Graph	Built *before* running	TensorFlow 1.x
Dynamic Graph	Built *as you run the code*	PyTorch, TF 2.x

Analogy

Static Graph (TensorFlow 1.x):

It's like planning your entire route in Google Maps **before** starting your trip — no re-routing on the way.

Dynamic Graph (PyTorch):

It's like using real-time GPS you can **change the route** while you're driving.

Which Should You Use?

If you want to...	Use
Prototype fast with simple syntax	**TensorFlow / Keras**
Customize training loop, layers, loss	**PyTorch**
Focus on research papers, academic work	**PyTorch**
Deploy on mobile/web with tools	**TensorFlow**
Learn the basics of DL quickly	**Keras**
Build explainable or modular ML pipelines	**PyTorch** (easier to integrate)

Task : Choose Between TensorFlow and PyTorch

Q: Your team needs a framework for experimenting with new models. Should you pick PyTorch or TensorFlow? Justify your answer.

A:PyTorch is preferred in research settings due to its dynamic computational graph, which supports faster experimentation and easier debugging. TensorFlow is better for production deployment but has a steeper learning curve for research use.

Summary

Topic	Key Takeaways
Neural Networks	Simulate brain function; made of layers of neurons
Deep Learning	Learns hierarchical representations of data
Applications	Found in CV, NLP, medicine, finance, etc.
Frameworks	TensorFlow & PyTorch lead the ecosystem