Deep-Learning-Series-1-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

1. Different types of Activation functions

What Are Activation Functions?

Imagine a light switch. When you flip it, the light turns on or off. An **activation function** in a neural network works kind of like that switch—it decides if a neuron should "fire" (activate) or not based on the input it gets.

But it’s not just an on/off switch. Sometimes, it lets signals through in varying strengths, like dimming a light instead of just turning it on or off.

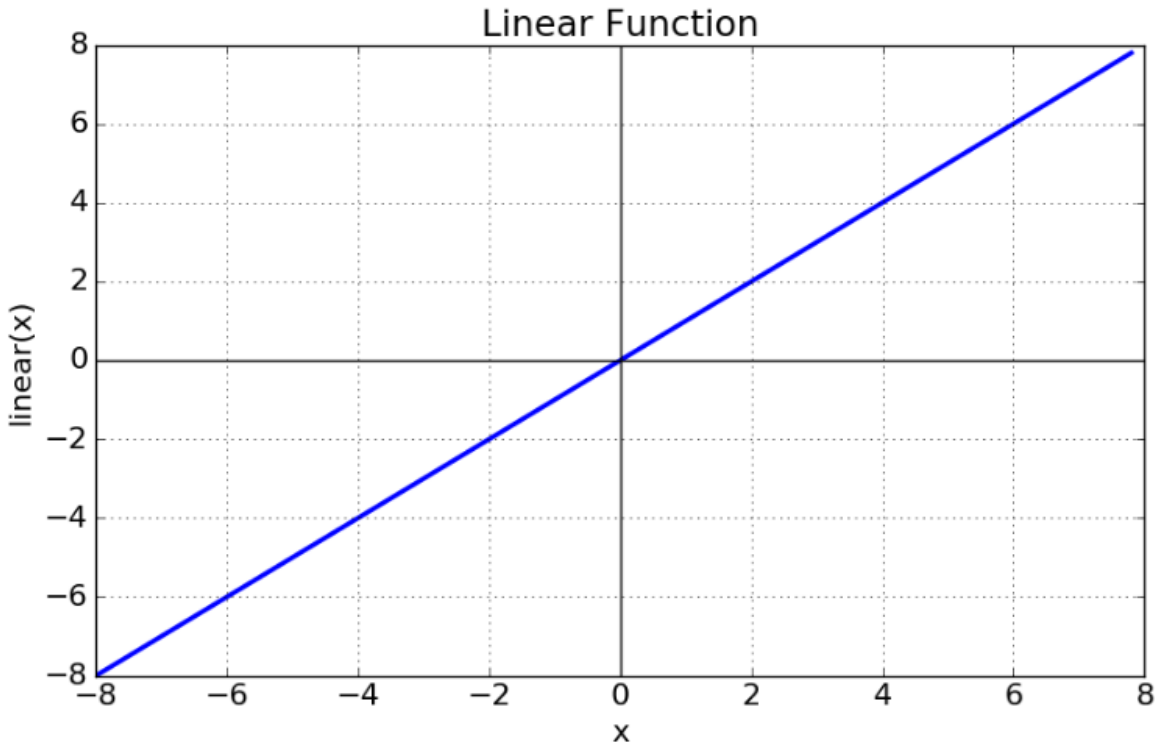
Why Do We Use Activation Functions?

Without activation functions, a neural network would just be doing simple math, like adding and multiplying. This would make it **impossible to learn complex patterns** like recognizing faces, understanding speech, or predicting trends. Activation functions introduce **non-linearity**, allowing networks to solve more complex problems.

Types of Activation Functions

1. Linear Activation Function (Identity Function)

- **What it does:** It doesn’t really change the input. Whatever goes in, comes out the same.



Equation : f(x) = x
Range : (-infinity to infinity)
It doesn’t help with the complexity or various parameters of usual data that is fed to the neural networks.

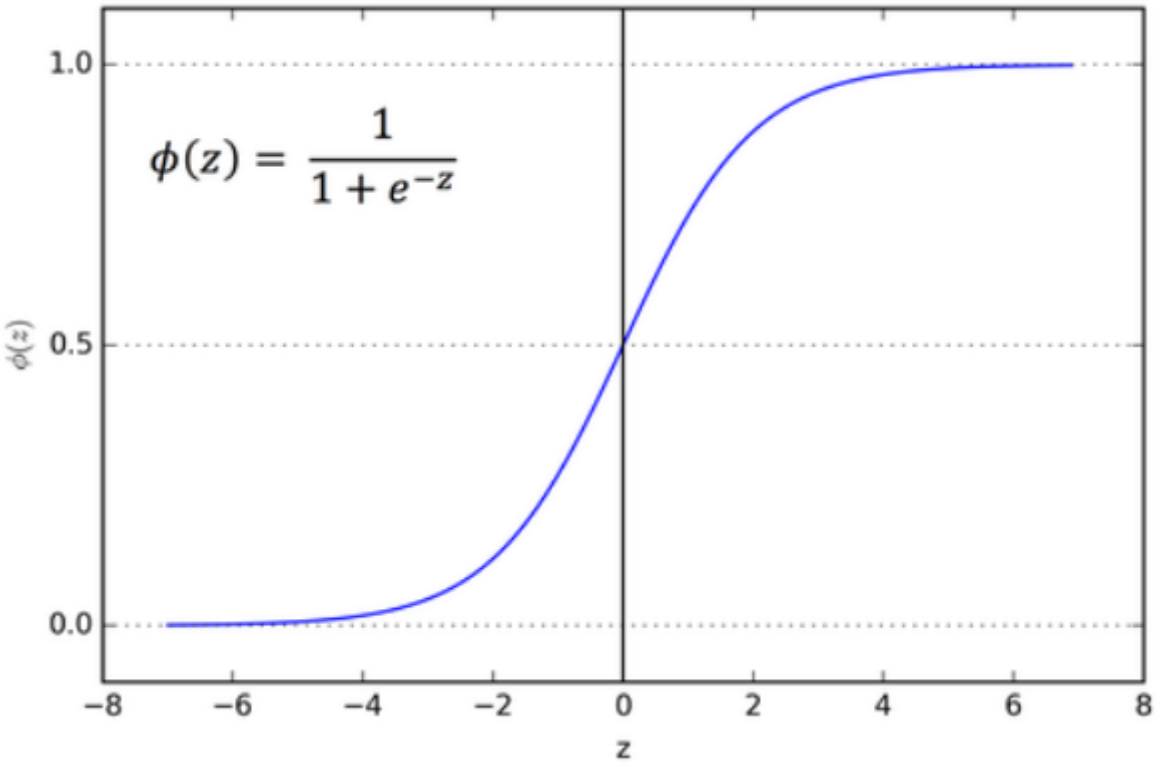
- **Why it’s not great:** It can't handle complex data because it doesn't add any "twist" to the input. It treats everything in a straight line, making it **useless for learning complex patterns**.

2. Non-Linear Activation Functions

These are the real stars because they help neural networks learn and handle complex data.

a) Sigmoid (Logistic) Function

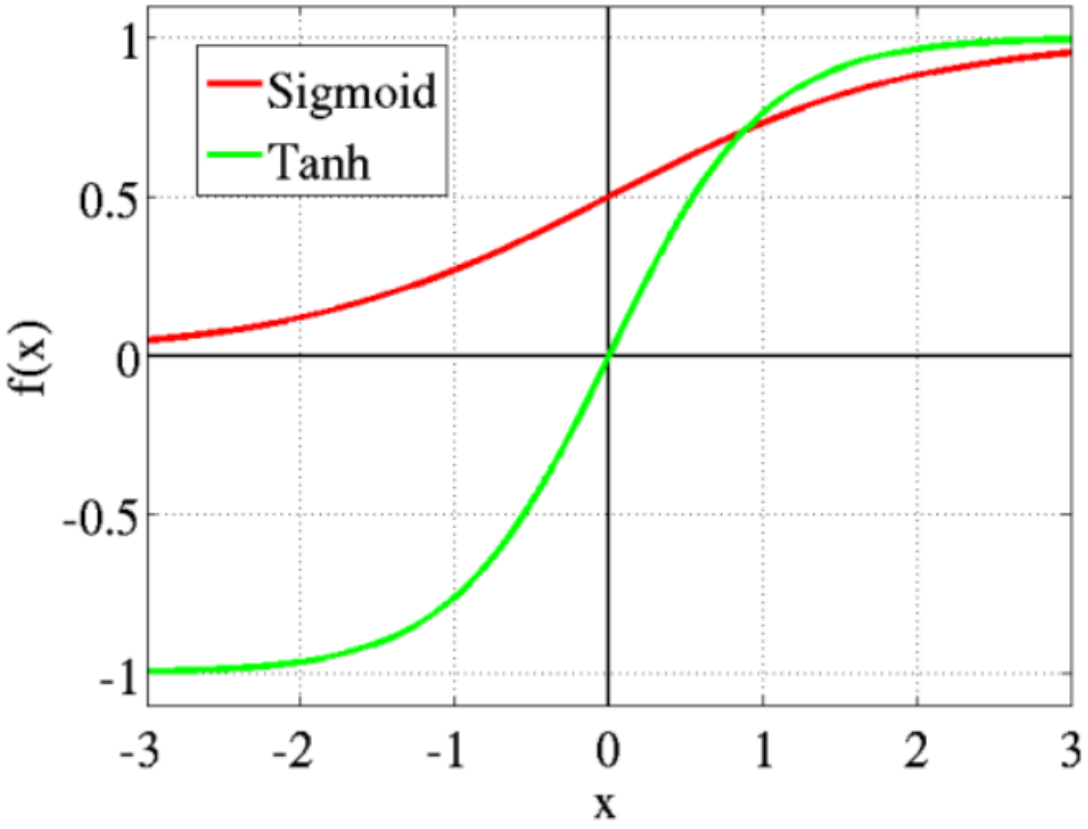
- **Shape:** Looks like an **S**.



- **Why it's useful:**
  Perfect for **probability predictions**. If you want to predict whether something is **yes or no**, this is a good choice because outputs stay between 0 and 1 (like probability).
- **Problems:**
  When the values are too high or too low, the function becomes really flat, and the network **stops learning**—this is called the **vanishing gradient problem**.

b) Tanh (Hyperbolic Tangent) Function

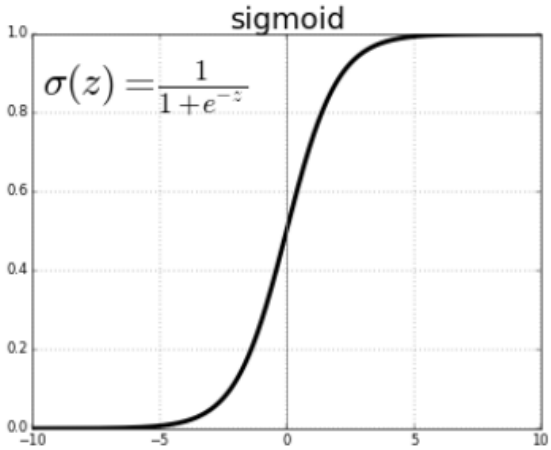
- **Shape:** Also **S-shaped**, but steeper.

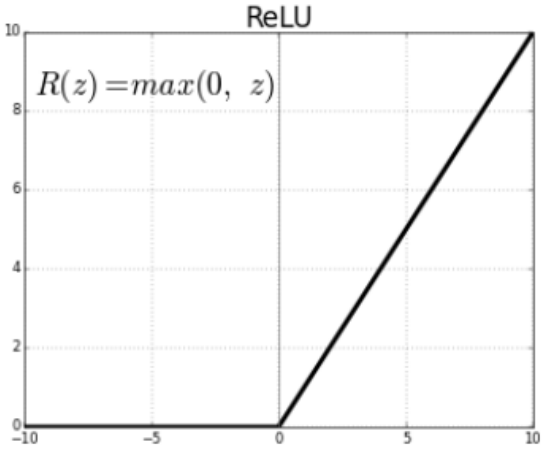


- Range: -1 to 1
- **Why it's better than Sigmoid:**
  It centers the data around **zero**, which helps the model learn faster. Negative inputs become negative outputs, and positive inputs become positive outputs.
- **Problems:**
  Still suffers from the **vanishing gradient problem**, though less than Sigmoid.

c) ReLU (Rectified Linear Unit)

- **Shape:** Looks like a **hockey stick**.





-
- Range: 0 to infinity
- **Why it's super popular:**
  It's simple and works really well for deep learning models. **If the input is positive, it stays the same; if it's negative, it turns into zero.** This makes computations fast and helps networks learn quickly.
- **Problems:**
  Sometimes, if too many neurons output **zero** (because of negative inputs), the network might stop learning in those parts. This is known as the **dying ReLU problem**.

Activation Function	Range	Good For	Problems
Linear	−∞-\infty to +∞+\infty	Simple tasks (but rarely used)	Can't handle complex patterns
Sigmoid	0 to 1	Binary classification, probabilities	Vanishing gradient, slow learning
Tanh	-1 to 1	Binary classification (better than Sigmoid)	Vanishing gradient (but less severe)
ReLU	0 to +∞+\infty	Deep learning, CNNs	Dying ReLU (neurons stop learning)

Activation functions help neural networks learn and make decisions. **ReLU** is widely used today because of its simplicity and performance, while **Sigmoid** and **Tanh** are more suited for specific tasks like classification.

2. Forward pass problem

A **forward pass** is simply feeding inputs into a neural network to get outputs. Think of it like putting ingredients (inputs) into a blender (the network), pressing start, and getting a smoothie (output).

Formulas Used

1. **Weighted Sum (for neurons):** Each neuron computes a weighted sum of its inputs plus a bias:

$$z = (input_1 \times weight_1) + (input_2 \times weight_2) + \ldots + bias$$

2. **Sigmoid Activation Function:** To introduce non-linearity, we apply the sigmoid function to the weighted sum:

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

3. **Error Calculation (optional for later steps):** Once outputs are generated, the error is often calculated using:

$$Error = \frac{1}{2}(Target - Output)^2$$

Example Problem

- **Inputs:**
  $X_1 = 0.05$, $X_2 = 0.10$
- **Weights (control how much influence inputs have):**
  $W_1 = 0.15$, $W_2 = 0.20$, $W_3 = 0.25$, $W_4 = 0.30$ (for hidden layer)
  $W_5 = 0.40$, $W_6 = 0.45$, $W_7 = 0.50$, $W_8 = 0.55$ (for output layer)
- **Biases (extra push for neurons):**
  $b_1 = 0.35$ (hidden layer), $b_2 = 0.60$ (output layer)
- **Targets (desired outputs):**
  $T_1 = 0.01$, $T_2 = 0.99$

Step-by-Step Forward Pass

1. Calculate Hidden Layer Outputs (H_1 and H_2)

For H_1:

$$H_1 = (X_1 \times W_1) + (X_2 \times W_2) + b_1$$

$$H_1 = (0.05 \times 0.15) + (0.10 \times 0.20) + 0.35 = 0.3775$$

Apply the **sigmoid activation function**:

$$H_1 = \frac{1}{1 + e^{-0.3775}} \approx 0.593$$

For H_2:

$$H_2 = (X_1 \times W_3) + (X_2 \times W_4) + b_1$$

$$H_2 = (0.05 \times 0.25) + (0.10 \times 0.30) + 0.35 = 0.3925$$

Apply sigmoid:

$$H_2 = \frac{1}{1 + e^{-0.3925}} \approx 0.597$$

2. Calculate Output Layer (Y_1 and Y_2)

Now, H_1 and H_2 become the new inputs.

For Y_1:

$$Y_1 = (H_1 \times W_5) + (H_2 \times W_6) + b_2$$

$$Y_1 = (0.593 \times 0.40) + (0.597 \times 0.45) + 0.60 = 1.106$$

Apply sigmoid:

$$Y_1 = \frac{1}{1 + e^{-1.106}} \approx 0.751$$

For Y_2:

$$Y_2 = (H_1 \times W_7) + (H_2 \times W_8) + b_2$$

$$Y_2 = (0.593 \times 0.50) + (0.597 \times 0.55) + 0.60 = 1.225$$

Apply sigmoid:

$$Y_2 = \frac{1}{1 + e^{-1.225}} \approx 0.773$$

3. Compare Outputs to Target Values (Error Calculation)

Now we see how far off our outputs are from what we *wanted*.

For Y_1:
We wanted **0.01**, but got **0.751**. That's a big miss.

For Y_2:
We wanted **0.99**, but got **0.773**. Closer, but still off.

Since our outputs don't match the targets, we'll use **backpropagation** to adjust the weights and biases so the network improves.

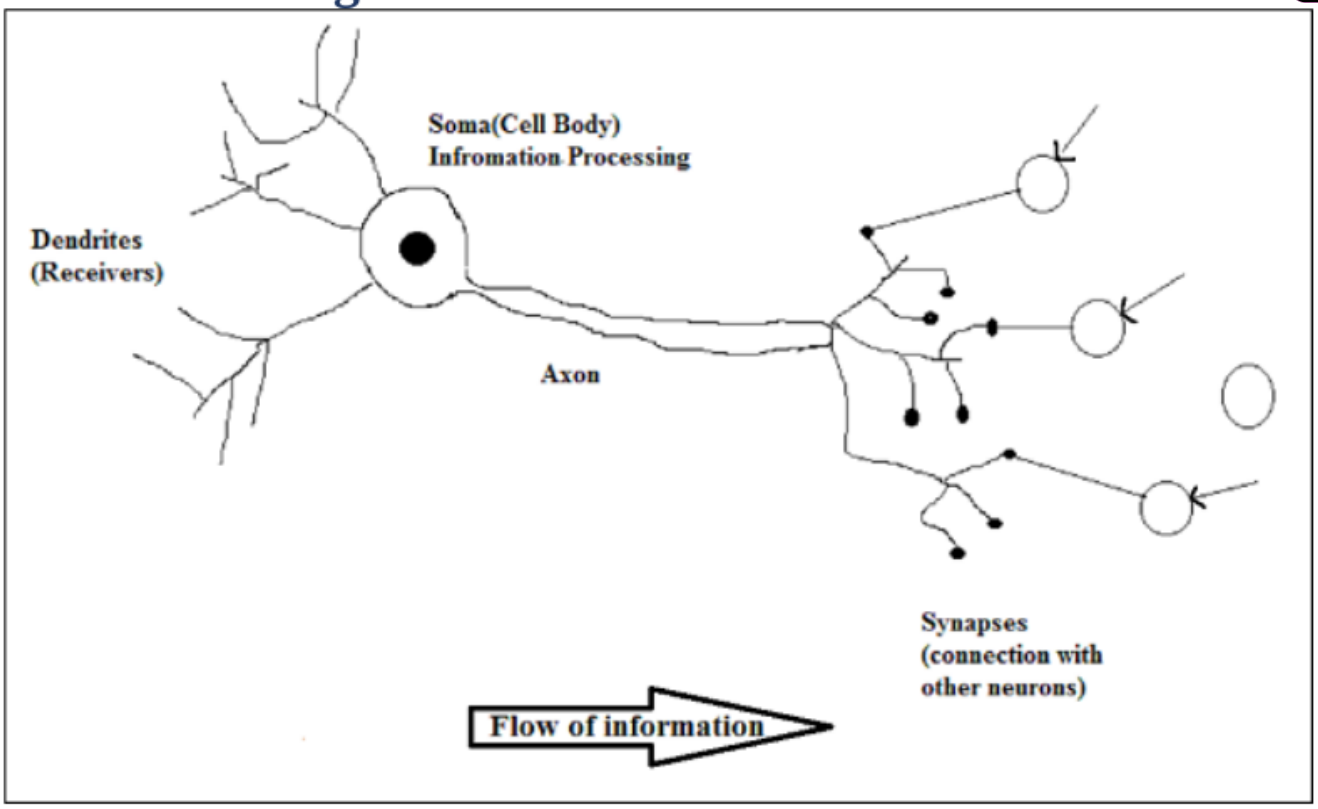
3. Difference between biological and artificial neural network

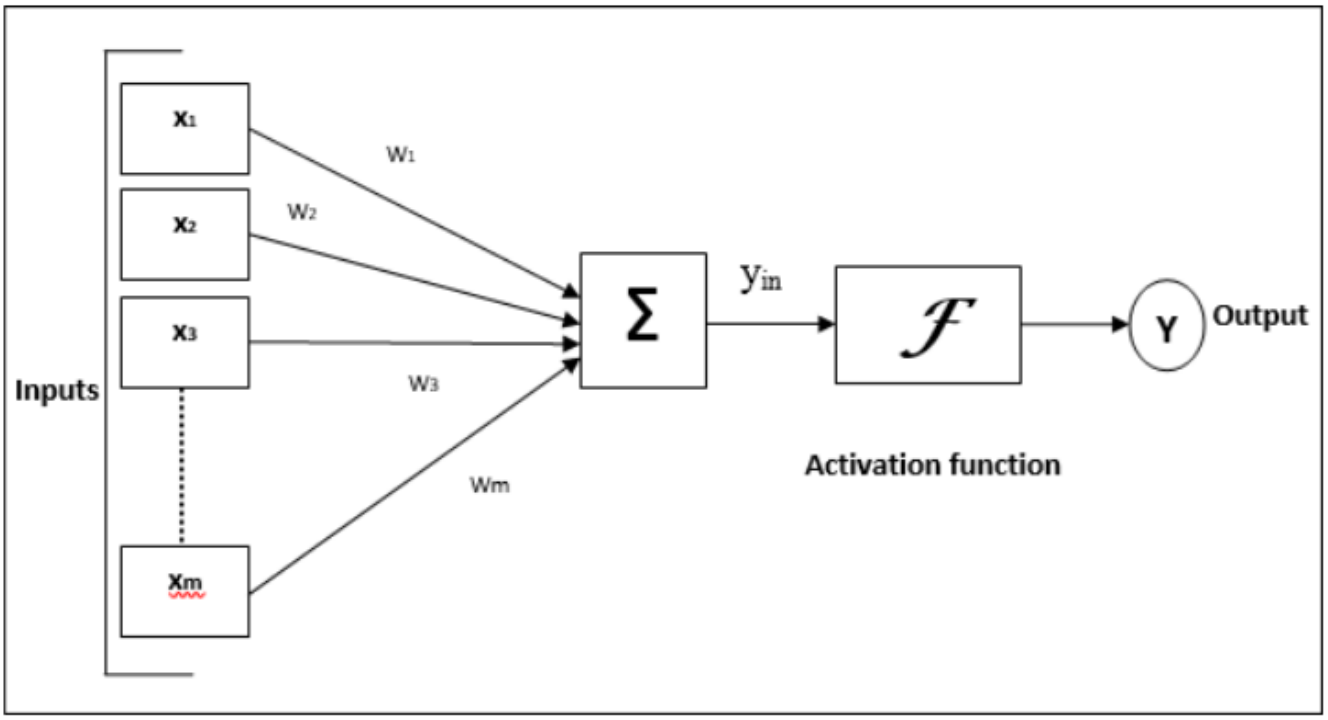
1. Basic Idea

- **Biological Neural Network (BNN):**
  This is the network of neurons in your brain and nervous system. It processes information from the world around you—like what you see, hear, or feel.
- **Artificial Neural Network (ANN):**
  This is a **computer program** designed to **mimic how the brain works**. It helps machines recognize patterns, like identifying faces in photos or translating languages.

2. Structure Comparison

Schematic Diagram





Biological Neural Network (BNN)	Artificial Neural Network (ANN)
Neuron (Nerve Cell)	Node (Processor Unit)
Dendrites (Receives signals)	Inputs (Data fed into the system)
Soma (Processes signals)	Activation Function (Processes data)

Biological Neural Network (BNN)	Artificial Neural Network (ANN)
Axon (Sends signals)	**Output (Result after processing)**
Synapse (Connections between neurons)	**Weights (Adjustable connections between nodes)**

3. How They Work

Biological Neuron (BNN):

1. **Dendrites** receive signals from other neurons (like ears listening to sounds).
2. The signal goes to the **Soma** (the cell body), which **processes** the information.
3. If the information is strong enough, it travels down the **Axon** (like a wire).
4. The signal jumps across the **Synapse** to the next neuron, passing on the message.

Artificial Neuron (ANN):

1. The system gets **inputs** (like numbers, images, or text).
2. Each input is multiplied by a **weight** (which decides how important that input is).
3. The neuron processes the weighted input using an **activation function** (like deciding if the input is strong enough to pass along).
4. It sends the **output** to the next layer of neurons in the network, continuing the process.

4. Key Differences

Criteria	Biological Neural Network (BNN)	Artificial Neural Network (ANN)
Processing Speed	Slow, but incredibly **efficient and adaptable**.	Fast, but less adaptable than the human brain.
Size	About **100 billion neurons** and **1 quadrillion connections**.	Usually between **100 to 10,000 nodes**, depending on the model.
Learning Ability	Learns from **ambiguous** and **unstructured data** (like messy real-world situations).	Needs **structured and clean data** to learn effectively.
Fault Tolerance	If some neurons get damaged, the brain can **adapt and reroute** signals.	ANNs can handle errors, but **large failures** may affect performance.
Storage	Stores information in **synapses** through chemical signals.	Stores data in **memory** (like files on a computer).

5. Examples

- **BNN Example:**
  If you see a blurry image of your friend, your brain can still recognize them because it can handle **uncertainty** and past experiences.
- **ANN Example:**
  A facial recognition app might **struggle** if the image is blurry unless it has been trained with lots of similar blurry photos.

4. Back propagation algorithm

Imagine you're learning how to throw a basketball into a hoop. The first time you throw, you miss. But after each attempt, you adjust your throw—maybe you throw a little softer or aim a bit higher. Over time, these small adjustments help you get better and better until you can make the shot consistently.

Backpropagation is how a **neural network learns from its mistakes**, just like how you adjust your basketball shot.

How Does Backpropagation Work?

Think of a neural network as a **big math machine** with **layers of neurons** (nodes). Here's how backpropagation helps it learn:

1. **Make a Guess (Forward Pass):**
   The network takes some input data, does a bunch of math (using **weights** and **biases**), and makes a prediction or guess (the **output**).
2. **Check the Error (Compare with Reality):**
   After making the guess, the network compares it to the actual answer (the **target**). The difference between the guess and the correct answer is called the **error**.
3. **Learn from Mistakes (Backward Pass):**
   Now, the network works **backwards** from the output to figure out **which parts of the network caused the error**. It uses a bit of calculus (called the **chain rule**) to understand how much each weight contributed to the mistake.
4. **Adjust and Improve (Update Weights):**
   Once it knows which weights are responsible for the error, it **adjusts them** slightly to reduce

the error the next time. This process is repeated over and over until the network gets better at making predictions.

Breaking It Down with a Simple Example

Let’s say we have a neural network that’s trying to predict the price of a house based on its size

1. **Forward Pass:**
   - The network takes the **size of the house** as input.
   - It multiplies this size by some **weights** (which are random at first) and adds a **bias**.
   - It spits out a **predicted price**.
2. **Calculate Error:**
   - The predicted price is $200,000, but the actual price is $250,000.
   - The **error** is $50,000.
3. **Backpropagate the Error:**
   - The network now asks, *“How much did each weight contribute to this $50,000 mistake?”*
   - It calculates this using the **chain rule** from calculus.
4. **Update Weights:**
   - If the weight was too low, the network **increases** it slightly.
   - If the weight was too high, the network **decreases** it slightly.
5. **Repeat:**
   - The network tries again with new data, and the process repeats.
   - Over time, the network becomes **better at predicting prices** because it keeps adjusting itself based on past mistakes.

Why is Backpropagation Important?

- **It’s Efficient:**
  Backpropagation is a fast way for neural networks to learn from mistakes. Instead of guessing randomly, it uses math to figure out **exactly what needs to change**.
- **It Works for Big Problems:**
  Whether you’re recognizing faces, translating languages, or playing video games, backpropagation helps deep learning models improve.

Steps of Backpropagation

1. **Input:** Feed data into the network.
2. **Forward Pass:** Calculate the output.
3. **Error Calculation:** Compare output with the actual result.
4. **Backward Pass:** Send the error backward through the network to calculate how much each weight contributed to the error.
5. **Update Weights:** Adjust weights to reduce the error.
6. **Repeat:** Do this over and over until the network makes accurate predictions.

Analogy to Help Understand Backpropagation

Imagine you're baking cookies for the first time:

1. **First Attempt (Forward Pass):**
   You follow a random recipe and bake the cookies.
2. **Taste Test (Error Calculation):**
   You taste the cookies and realize they’re too salty.
3. **Figure Out the Mistake (Backward Pass):**
   You think back: *“Did I add too much salt? Or maybe not enough sugar?”*
4. **Adjust the Recipe (Update Weights):**
   You reduce the salt and add more sugar for the next batch.
5. **Try Again (Repeat):**
   You bake another batch, taste again, and keep tweaking until the cookies taste perfect!

5. What is Deep learning?

What is Deep Learning?

Deep learning is a type of machine learning that teaches computers to learn from large amounts of data, much like how humans learn from experience. Instead of giving the computer exact instructions, we provide examples, and the computer figures out patterns on its own.

How Does It Work?

- **Inspired by the Brain**: Deep learning uses **artificial neural networks**, which are inspired by how the human brain works. Just like our brain has neurons connected to each other, these networks have "nodes" (or artificial neurons) connected in layers.
- **Layers of Learning**:
  - **Input Layer**: This is where data (like an image or text) enters the network.
  - **Hidden Layers**: These layers process the data. The more layers there are, the "deeper" the network is. Each layer learns different features. For example, in an image, early layers might detect edges, and deeper layers might recognize shapes or objects.
  - **Output Layer**: This layer gives the final result, like identifying if an image is a cat or a dog.

Why Is It Useful?

- **Learns Complex Patterns**: Deep learning can figure out very complicated patterns that traditional programming or simple machine learning might miss.
- **No Manual Instructions Needed**: You don’t have to manually program features (like detecting edges in images); the model learns them automatically.

Where Is It Used?

- **Computer Vision**: Recognizing faces in photos, self-driving cars detecting obstacles.
- **Natural Language Processing (NLP)**: Translating languages, chatbots, and virtual assistants.
- **Speech Recognition**: Turning spoken words into text, like in Siri or Google Assistant.
- **Game Playing**: Beating humans in games like Chess or Go.

Machine Learning vs. Deep Learning:

Machine Learning	Deep Learning
Needs less data to work.	Needs a lot of data.
Works well for simple tasks.	Better for complex tasks like image or speech recognition.
Features are manually chosen.	Learns features automatically.
Can run on regular computers.	Needs powerful computers (GPUs).

Challenges in Deep Learning:

1. **Needs Lots of Data**: The more data, the better it performs.
2. **Expensive to Run**: Training deep models requires powerful computers.
3. **Hard to Understand**: It’s like a black box—you can’t always explain how it made a decision.
4. **Overfitting**: Sometimes, it learns the training data too well and doesn’t perform well on new data.

Advantages of Deep Learning:

- **High Accuracy**: Can outperform traditional methods in tasks like image recognition.
- **Automatically Learns Features**: No need for manual tweaking.
- **Can Handle Complex Data**: Works well with images, audio, text, etc.