Activation Functions

In neural networks (especially CNNs in computer vision), activation functions decide whether a neuron should be activated or not. They introduce non-linearity, allowing the model to learn complex patterns like edges, textures, and objects in images.

Common Activation Functions (with comparison)

Function		Use Case / Notes
Sigmoid	(0, 1)	Good for binary classification but vanishes gradients for large values.
Tanh	(-1, 1)	Better than sigmoid. Still suffers from vanishing gradients.
ReLU	[0, ∞)	Most popular in CNNs. Fast, simple, sparsity-inducing.

1. Sigmoid Activation Function

The **Sigmoid activation function** is a mathematical function that maps any input value (from negative infinity to positive infinity) into a value between **0 and 1**. It is defined by the equation:

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

This function takes real-valued input and transforms it into a value between 0 and 1, making it ideal for **binary classification problems**, where you want to predict whether something is true or false (e.g., "Is there a cat in this image?").

Sigmoid introduces **non-linearity**, allowing the neural network to learn more complex patterns in the data. However, it shrinks large input values towards 0 or 1, which can cause the **vanishing gradient problem** in deep networks—meaning it becomes difficult for the network to learn and update weights effectively.

Advantages:

- Output is between 0 and 1 (great for probabilities).
- Useful in the **final output layer** of a binary classifier.

❌ **Disadvantages:**

- **Vanishing gradients** for large positive or negative inputs.
- Outputs are **not zero-centered**, which can slow down learning.
- Computationally is more expensive than ReLU.

What is the Vanishing Gradient Problem?

The **Vanishing Gradient Problem** is a major issue in training deep neural networks. It happens when the **gradients become too small** during backpropagation, causing **early layers to stop learning**. This leads to **very slow training**, **low accuracy**, or a **stuck model**. It is mostly caused by activation functions like **Sigmoid** and **Tanh**, which squash the output and make gradients very small. To solve this problem, we use activation functions like **ReLU, Leaky ReLU, ELU, Swish, and GELU**, which help keep gradients strong and allow better learning in deep networks. **Sigmoid Activation Function:**

- Gives output between 0 and 1
- Looks like an S-shaped curve
- When input is very big or very small, output becomes close to 1 or 0
- The **slope (gradient)** in these areas becomes **almost zero**
- So, when we do backpropagation, the gradients become very **tiny**, and earlier layers **get no update**

  Solutions: (Leaky RELU , GELU , ELU)

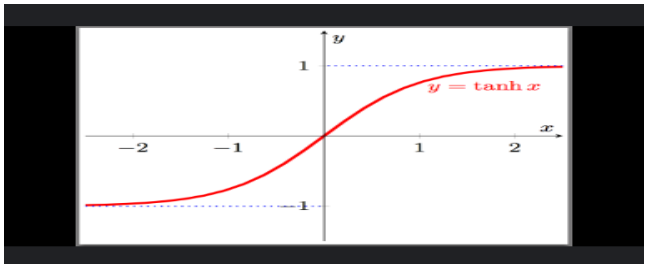
2. Tanh (Hyperbolic Tangent) Activation Function

The **Tanh (hyperbolic tangent)** function is another widely used activation function. It is mathematically defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The Tanh function transforms input values into the range **between -1 and +1**. Like the sigmoid, it's **non-linear**, but unlike sigmoid, it is **zero-centered**, meaning its output has a mean of 0. This generally leads to **better and faster convergence** during training.

Because of its symmetrical shape and wider range (-1 to 1), it often performs better than sigmoid for the hidden layers.



✅ **Advantages:**

- **Zero-centered**, which is better for convergence in optimization.
- Better gradient flow than sigmoid.

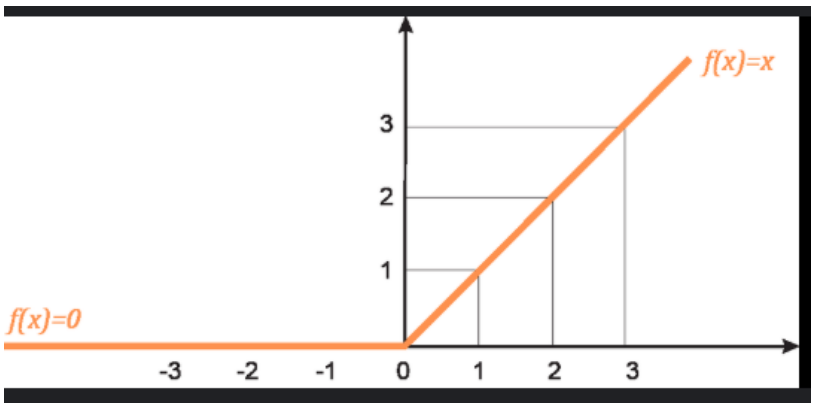
❌ **Disadvantages:**

- Still suffers from the **vanishing gradient problem**.
- Not ideal for very deep networks.
- Slightly more expensive to compute than ReLU.

🔹 3. ReLU (Rectified Linear Unit)

The **ReLU (Rectified Linear Unit)** is one of the most popular activation functions, especially in **Convolutional Neural Networks (CNNs)** used for Computer Vision. It is defined as:

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



This means:

- If the input is **positive**, output it as it is.
- If the input is **negative**, output **0**.

✅ **Advantages:**

- **No vanishing gradient** for positive inputs.
- Computing very fast.
- Introduces sparsity (many neurons become inactive).
- Works very well in **deep CNNs** for image classification.

❌ **Disadvantages:**

- Can cause **"Dying ReLU"** problem: neurons stuck at 0 forever. (alternative: leaky RELU)
- Zero gradient for negative input values.
- Not suitable for all types of inputs (e.g., very noisy or negative datasets).

Activation	Range	Zero Centered	Gradient Issue	Ideal For
Sigmoid	(0, 1)	❌ No	Vanishing gradient	Output layer (binary classification)
Tanh	(-1, 1)	✅ Yes	Vanishing gradient	Hidden layers (small networks)
ReLU	[0, ∞)	❌ No	Dying neurons (for negative x)	Hidden layers in deep CNNs

Why Do We Need Zero-Centered Activation Functions?

🔍 What is Zero-Centered?

A function is **zero-centered** if its output is equally balanced around zero, meaning it produces both positive and negative values. For example:

- **Tanh** outputs between **-1 to +1** → ✅ Zero-centered.
- **Sigmoid** outputs between **0 to 1** → ❌ Not zero-centered.
- **ReLU** outputs between **0 to ∞** → ❌ Not zero-centered.

Problem 2: Slower Convergence

Non-zero-centered activations cause an imbalance in how gradients are distributed. This causes:

- Some weights to **update faster**, others **slower**.
- Optimizer needs **more iterations** to reach the optimal values.
- You might need to lower the learning rate, slowing down even more.

Common Issues in Deep Learning Models

- When you're training a deep learning model, you can face several challenges that prevent the model from learning properly or making good predictions. These are some of the **common issues**:

1. **OVERFITTING**

Overfitting happens when the model **memorizes** the training data instead of learning **general patterns**. (more specific)
It performs **very well on training data**, but **badly on new (test) data**.

🧠 Real-life Example:

A student **memorizes all questions** from the textbook but **fails the final exam** because the questions were different.

❌ Causes:

- Model is **too complex** (deep with many layers)
- Not enough training data
- Training for too long
- Noisy or irrelevant data
- No regularization

✅ Solutions for Overfitting:

Solution	Description
Use Regularization	Techniques like **L1/L2**, **Dropout** remove extra dependence on weights
Get More Data	More data = better generalization
Reduce Model Complexity	Use fewer layers/neurons
Early Stopping	Stop training when validation loss starts increasing
Data Augmentation	Modify images (rotate, flip, etc.) to make training data larger

---

📋 Comparison Table

Feature	Underfitting	Overfitting
Training Error	High	Low
Testing Error	High	High

Feature	Underfitting	Overfitting
Model Complexity	Too Simple	Too Complex
Generalization	Poor	Poor
Solution	More Complexity	Regularization, Simpler model

2. underfitting (more general)

3. Gradient Vanishing (already discussed)

4. Exploding Gradient Problem

The exploding gradient problem happens when the gradients (used in backpropagation to update weights) become too large. This causes:

- Unstable training,
- Weights grow extremely large,
- Model outputs become NaN (Not a Number) or Infinity,
- Loss goes out of control (i.e., doesn't decrease but explodes).

Why does it happen?

In deep neural networks, when gradients are propagated backward layer by layer, they are multiplied through the layers. If the numbers are slightly large in each layer, they can grow exponentially in deeper layers.

It's like multiplying numbers like:
1.5 × 1.5 × 1.5 × 1.5 × ... → quickly becomes a huge number.

Results:

- **Training loss = NaN(Not a Number or infinity)**
- **Model doesn't learn anything**

Example: Imagine you're carrying a stack of books upstairs. With every step (layer), the weight gets multiplied. By the 20th step, you loss strength and **fall**

5. Slow Convergence

Slow convergence means the model is learning very slowly, or struggles to reduce the loss during training.

It may look like the training is happening, but the model takes too many struggles, or doesn't reach good accuracy even after training for a long time.

What happens?

- Training takes a long time to converge (reach low loss).
- Loss decreases very slowly.

Example: Imagine walking in the dark toward a door. You take very small or random steps, so it takes a long time to reach your goal.

Optimizers in Deep Learning

Optimization in deep learning refers to the process of minimizing the loss function by adjusting the model's weights and biases during training.

Think of it as a guide that helps your model learn better and faster. The most common technique is gradient descent, but it has many variants.

1. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is one of the simplest and most traditional optimization algorithms used to train machine learning and deep learning models. It updates the model's parameters step by step to reduce the error by calculating how far off the prediction is from the actual result.

But unlike "batch gradient descent," which uses the entire dataset to calculate the update, SGD uses only a small batch (or a single example) for each update. This makes it faster and less memory-heavy.

🧠 Key Idea:

Update the model's parameters using small pieces of data repeatedly and try to reduce the loss (error) little by little.

❌ Limitations:

- Slow and unstable in large or deep models.
- It doesn’t consider the previous steps, so it lacks memory of how the error is changing over time.
- Can get stuck in bad positions, like local minima or flat areas

2. SGD with Momentum

SGD with Momentum improves the basic SGD by giving it a sort of "memory". Instead of just looking at the current update, it considers the past few updates and uses them to guide the current direction. This helps it move more smoothly and consistently.

🧠 Key Idea:

Imagine pushing a ball down a hill — it gains speed and doesn't get stuck in small holes. Momentum works similarly by building speed in directions where the error keeps reducing, helping the optimizer move faster and avoid getting stuck.

🔥 Advantages Over Basic SGD:

- Faster and more stable learning.
- Less chance of getting stuck in local traps.
- Helps move across flat or bumpy areas of the loss surface.

❌ Limitations:

- If momentum is too high, it may overshoot the best values.
- Sensitive to initial starting values in some cases.

3. RMS Propagation (Root Mean Square Propagation)

RMSProp is a more advanced optimizer that adapts the learning rate for each parameter based on how fast it's changing. This means it automatically slows down the updates where things are changing a lot and speeds up where things are slow.

It became popular especially for training deep networks like RNNs (used in language models), where standard SGD struggles a lot.

🧠 Key Idea:

It keeps track of how much each weight/parameter has been changing recently and adjusts how much it should move based on that history. It prevents some weights from getting too big too quickly, and helps others move faster if they're stuck.

🔥 Advantages:

- Automatically adjusts step sizes for different weights.
- Speeds up training and avoids big swings in values.

❌ Limitations:

- If not configured right, it may forget long-term patterns and become unstable.
- It may not perform well on all types of problems, especially very large ones.

Feature	**SGD**	**SGD with Momentum**	**RMSProp**
Speed	Slow	Medium	Fast
Stability	Low (bounces a lot)	**More stable**	Very stable (if tuned well)
Memory of past updates	❌ No	✅ Yes	✅ Yes
Best for	Small/simple models	Deep CNNs, vision tasks	RNNs, complex models
Limitation	Stuck easily, slow	Can overshoot	May forget long-term patterns

Optimization Techniques

1. Regularization

Regularization is used to **prevent overfitting** — when the model performs well on training data but poorly on unseen data.

1. Dropout

- **What it does**: Randomly turns off (i.e., ignores) a fraction of neurons during training.
- **Why it's used**: Prevents neurons from relying too much on each other and forces the model to learn more robust features.
- **Example**: A dropout rate of 0.5 means 50% of the neurons will be dropped in that layer during training.

2. Weight Decay (L2 Regularization)

- **What it does**: Adds a penalty to the loss function based on the size of the weights.
- **Why it's used**: Discourages large weights, making the model simpler and reducing overfitting.
- **Analogy**: Keeps weights "under control" by punishing large values.

3. Data Augmentation

- **What it does**: Creates more training data by transforming existing data (e.g., rotating, flipping, cropping).
- **Why it's used**: Improves generalization by teaching the model to be invariant to changes in orientation, size, lighting, etc.
- **Example**: In image classification, flipping or rotating images increases diversity in training.

---

2. Normalization

Normalization improves training **speed and stability** by ensuring values (activations or features) remain within a predictable range.

1. Batch Normalization

- **What it does**: Normalizes outputs of a layer for each batch during training.
- **Why it's used**: Helps reduce internal shift (changes in distribution of layer inputs), accelerates training, and sometimes reduces need for dropout.
- **When used**: Works well in CNNs and fully connected layers.

2. Layer Normalization

- **What it does**: Normalizes across the features for each data point (rather than across the batch).
- **Why it's used**: Preferred in RNNs and Transformers where batch sizes may vary.
- **Example**: Used in GPT models.

---

3. Training Enhancements

These are techniques to **make training more efficient, stable, or faster**.

1. Learning Rate Scheduling

- **What it does**: Dynamically changes the learning rate during training.
- **Why it's used**: Helps model converge better by starting with a high learning rate (faster learning) and reducing it later (for fine-tuning).

2. Early Stopping

- **What it does**: Stops training when the model's performance stops improving on validation data.
- **Why it's used**: Saves time and prevents overfitting.
- **Example**: If validation loss doesn't improve for 5 consecutive tries, training halts.

3. Mixed Precision Training

- **What it does**: Trains parts of the model using lower precision (like float16 instead of float32).
- **Why it's used**: Increases training speed and reduces memory usage, especially on modern GPUs.
- **Benefit**: Almost same accuracy with much faster training and less resource consumption.

🔍 What is AlexNet?

AlexNet is the first CNN model that gained worldwide attention. It reduced the top-5 error from 26% to 15%, which shocked the AI community.

AlexNet was the **first deep CNN** to achieve **breakthrough accuracy** on the ImageNet dataset in 2012. It basically **popularized deep learning** for computer vision.

🧱 Structure and Layers:

- Total 8 layer (5 convolutional + 3 fully connected laywer)

- 5 convolutional layers: These apply filters to extract patterns like edges, textures, and shapes.
- 3 fully connected layers: These combine learned features and make predictions.
- ReLU activation: Used instead of sigmoid/tanh for faster training.
- Dropout: Randomly disables some neurons during training — prevents overfitting.

🔥 Why It Was Revolutionary:

- Used GPUs
- First to prove deep CNNs work with large datasets
- Introduced ReLU (fast convergence)
- Used data augmentation (rotating, cropping) to simulate more data

💉 Real-World Use Cases:

- Early face recognition models
- Handwriting recognition
- Medical image classification (before more modern networks)
- Served as the base for further models

❌ Limitations:

- Very large model: ~60 million parameters (slows down small devices)
- Training takes time even with GPUs
- Can overfit if not tuned well
- Cannot handle very complex images (e.g. multiple objects, occlusions)

What is VGGNet?

Created by **Visual Geometry Group** at Oxford University, VGG introduced a **simple but powerful idea**: use **very small filters (3x3)** and make the network **very deep** (16 or 19 layers).

Instead of using large filters (like 11x11 in AlexNet), VGG used **stacked small filters**, which allowed it to extract **more detailed and complex features**.

Why It Works:

- Deep structure → learns more complex features
- Repetitive structure → easy to understand and implement
- **Transfer learning**: VGG is heavily used in applications where pretrained models are reused and fine-tuned

Real-World Use Cases:

- **Medical imaging** (tumor detection, X-rays)
- **Security cameras** (object detection, facial recognition)
- **Agriculture** (detecting pests and crop disease)
- **Autonomous driving** (lane and traffic sign detection)

Limitations:

- **Huge model size**: VGG16 has **138 million parameters**
- Requires **large memory** and **high GPU power**
- **Slow training** and **inference time**
- **Not suitable for mobile or embedded devices**

GoogleNet

Inception, also known as GoogLeNet, was developed by Google researchers to solve one big problem: how to make deep networks faster and lighter without losing accuracy.

The main idea is that different filter sizes capture different information, so why not apply multiple filters at the same layer in parallel?

Unique Features:

- **Inception Modules**: Combines 1x1, 3x3, and 5x5 filters in one block
- **Depth + width** → captures both fine and abstract features
- Final structure has 22 layers, but optimized

Why It Was a Breakthrough:

- More **accurate** and **faster** than VGG
- Uses fewer parameters (only 5 million) than VGG's 138 million!
- Learns **multi-scale features**
- **Efficient use of memory and computation**

Real-World Use Cases:

- **Real-time mobile apps** (like Google Lens)
- **Autonomous vehicles**
- **Smart surveillance**
- **Satellite imagery classification**

Summary Table — Extended Comparison

Feature	AlexNet	VGG	Inception
Year Introduced	2012	2014	2014
Depth	8 layers	16–19 layers	22 layers
Innovation	ReLU, dropout	Simple deep stacking	Parallel multi-size filters
Parameters	~60 million	~138 million	~5 million (very efficient)
Training Speed	Medium	Slow	Fast
Model Size	Large	Very large	Small
Used in Phones?	No	Rare	Yes
Best For	Basic classification	Pretrained feature use	Efficient multi-scale detection

What is ResNet?

ResNet stands for **Residual Network**.
It was introduced in **2015** by researchers from Microsoft and **won the ImageNet Challenge** with a **top-5 error rate of only 3.57%**, even beating **human-level accuracy** on the dataset.

🔥 ResNet's biggest idea:

Make neural networks very deep — without causing degradation in performance.

It allowed researchers to **train CNNs with hundreds or even thousands of layers**.

❓ Why ResNet Was Needed – The Degradation Problem

When researchers tried to make CNNs deeper (like 20, 30, 50+ layers), they saw something strange:

- The accuracy **stopped improving**
- In fact, deeper models were performing **worse** than shallower ones
- This wasn't caused by overfitting — it was a **training problem**!

This is called the Degradation Problem:

As the number of layers increases, the training error starts increasing instead of decreasing.

How ResNet Solves It – Skip Connections

What is a Skip Connection?

🔁 Normal Flow in a Neural Network:

In a typical neural network:

- **Layer 1** passes its output → to **Layer 2**
- Layer 2 processes it → passes it to Layer 3
- And so on…

This is called a **sequential or feed-forward** flow.

But this creates problems in **very deep networks**, like:

- **Vanishing gradients**
- Difficulty in learning identity mappings (i.e., passing input forward when no transformation is needed)
- Poor training performance in deeper layers

🚀 Skip Connection in ResNet:

A **skip connection** is a clever trick where:

The input of a layer is **added directly** to the output of a few layers ahead, **skipping one or more layers** in between.

Why Skip Connections Help

1 Makes Learning Identity Easier

Skip connections **create shortcut paths** that allow gradients to flow **directly back** to earlier layers.

This ensures:

- Gradients don't vanish
- Learning continues even in early layers
- Network trains faster and more efficiently

Advantages of ResNet

- **Solves vanishing gradient** and degradation problem
- Allows **very deep networks** to be trained effectively
- Performs **better** than VGG, Inception, and AlexNet in many tasks
- Widely used as **backbone** for other networks (like object detection models)

Types of ResNet

What Are ResNet Models?

ResNet (Residual Networks) are deep Convolutional Neural Networks (CNNs) designed to solve the *vanishing gradient problem* by using **skip connections**. The number in **ResNet18**, **ResNet34**, etc., refers to the **total number of layers** in the network.

🧱 1. ResNet18 — Lightweight and Fast

Property	Details
Total Layers	18
Depth	Shallow (good for smaller problems)
Speed	Very fast; low latency
Model Size	Small (fewer parameters)
Use Case	Mobile devices, IoT, edge computing

📌 Explanation:

- Contains 18 layers with a small number of parameters.
- Good for real-time or **low-power devices** (e.g., phones, sensors, robots).
- Ideal for **basic image classification** and **small datasets**.

⚠️ Limitation:

- May not perform well on **very complex tasks** like large-scale object detection or segmentation.
-

ResNet34 — A Bit Deeper, Better Accuracy

Property	Details
Total Layers	34
Depth	Medium (deeper than ResNet18)
Speed	Still relatively fast
Model Size	Moderate (more than ResNet18)
Use Case	Mid-sized tasks; balanced performance

Property	Details

Limitation:

- Not as fast as ResNet18.
- Slightly larger size may make it less ideal for mobile apps.

ResNet50 — The Popular Workhorse

Property	Details
Total Layers	50 (includes *bottleneck* blocks)
Depth	Deep (for complex visual tasks)
Speed	Slower than 18 and 34 but very powerful
Model Size	Much larger (more parameters)
Use Case	Image classification, medical imaging, object detection, face recognition, etc.

Uses **bottleneck architecture** (3-layer blocks instead of 2) to keep efficiency while going deeper.

 Excellent for **high-accuracy tasks** like:

- Object Detection (e.g., COCO dataset)
- Face Recognition (e.g., VGGFace2)
- Medical image classification

 Most common in research and production use.

Limitation:

- **Needs more compute power**
- Not suitable for mobile or low-power systems without optimization

Comparison Table:

Feature	ResNet18	ResNet34	ResNet50
Layers	18	34	50
Speed	🔵 Fastest	🟡 Moderate	🔴 Slower
Accuracy	🟡 Decent	🟢 Better	🟢🟢 Best
Model Size	🔵 Small	🟡 Medium	🔴 Large
Use Case	Mobile, IoT	Mid-tasks	Complex tasks

Situation	Use
You need fast results or mobile use	**ResNet18** or **ResNet34**
General-purpose classification	**ResNet50** (balanced, very common)
High accuracy in complex problems	**ResNet101** or **ResNet152**
You're building a custom architecture	Start from ResNet50 as your backbone

DenseNet (Densely Connected Convolutional Network)

DenseNet is a **CNN architecture** introduced to solve the problems of **vanishing gradients, feature reuse, and training efficiency**. It builds upon the idea of ResNet but takes it even further.

📌 Core Idea of DenseNet

In **DenseNet**, **each layer is connected to every other layer** in a feed-forward fashion.
Instead of just passing information to the next layer, every layer **receives feature maps from all previous layers** as input, and **passes its own feature maps to all subsequent layers**.

👉 **If there are L layers**, then each layer gets inputs from **all L-1 previous layers**.

- **Traditional CNN**:

Layer 1 → Layer 2 → Layer 3 → Output

- **ResNet**:

Layer 1 → Layer 2 → Layer 3 → Output

↘_____↑ (Skip Connection)

- **DenseNet**:

Layer 1 → Layer 2 → Layer 3 → Layer 4

↘ ↘ ↘ ↘

→ → → → Output

Each layer has **direct access** to gradients and feature maps from previous layers.

Benefits of DenseNet

Fewer Parameters

- Despite many connections, it actually uses **fewer parameters** than ResNet because it doesn't need to relearn redundant features.

Improved Accuracy

- DenseNet achieves **better performance with fewer parameters** compared to VGG and ResNet.