**Intern Project**

**Supervised Learning**

Supervised Learning is a type of machine learning where the model is trained on a labelled dataset. Each example in the dataset consists of an input (features) and a corresponding correct output (label). The model learns to map inputs to the correct outputs by minimizing the prediction error.

**Example:**

Imagine teaching a model to distinguish between cats and dogs:

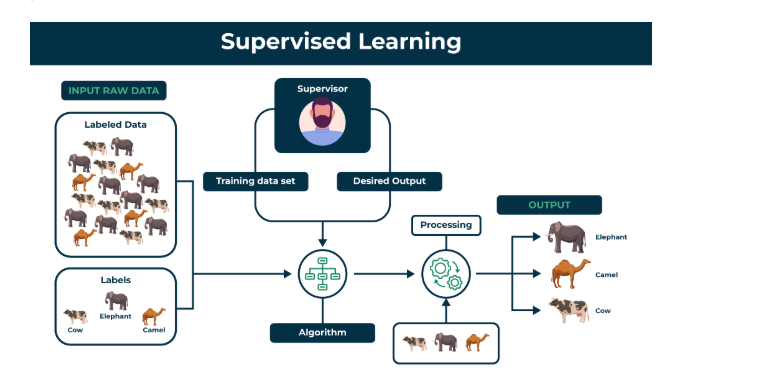
* **Inputs:** Images of animals.
* **Labels:** "Cat" or "Dog."
* The model learns patterns (like shape, color) and predicts the label for new images.

**Advantages of Supervised Learning:**

1. **Clear and Accurate Results:**  
   Since the model learns from labelled data (where answers are already known), it usually gives very accurate results.
2. **Easy to Understand:**  
   You know what the model is learning because you control the input and the correct output.
3. **Easy to Measure Performance:**  
   You can easily check how well the model is doing by comparing its predictions to the correct answers.
4. **Works Well for Specific Tasks:**  
   It is great when you have a specific goal, like classifying emails as spam or predicting house prices

**Cons of Supervised Learning:**

1. **Needs Labelled Data:**  
   You need a lot of labelled examples to train the model. Getting and labeling this data can be expensive and time-consuming.
2. **Not Flexible:**  
   It only works well for the specific problem it was trained on. It may not adapt easily to new problems without retraining.
3. **Risk of Overfitting:**  
   The model might learn the training data too well, including noise or irrelevant details, and then perform poorly on new, unseen data.
4. **Takes Time and Effort:**  
   Preparing clean, labelled data and training the model properly requires effort and expertise.



**Unsupervised Learning**

Unsupervised Learning is a machine learning approach where the model is trained on **unlabelled data**. The system tries to identify inherent patterns, relationships, or structures without any specific guidance

**Example:**

Given a dataset of customer purchase behaviour:

* The model might group customers with similar buying patterns without knowing who they are.
* Useful for targeted marketing or recommendation systems.

**Advantages of Unsupervised Learning:**

* No need for labelled data (cost-effective, scalable).
* Useful for exploratory analysis and discovering hidden patterns.
* Can handle complex and large datasets.

**Challenges:**

* Harder to evaluate the correctness of results.
* Risk of finding patterns that don't have real-world meaning (**interpretability** issue).

A diagram of a training process

AI-generated content may be incorrect.

**What is Overfitting**

Overfitting happens when your model **learns too much** from the training data, including the **noise and random fluctuations**. It memorizes the data instead of understanding the general patterns. So, it performs **very well on training data** but **poorly on new, unseen data** (like test data).

**Analogy:**

Imagine you're preparing for an exam by memorizing all the answers from last year's question paper. If the exact same questions appear, you'll ace it! But if the questions change slightly, you'll struggle because you only memorized and didn't understand the concepts.

**Indicators of Overfitting:**

* **Training Accuracy = High**
* **Test Accuracy = Low**

**What is Underfitting**

Underfitting happens when your model is **too simple** and fails to capture the patterns in the data. It performs **poorly on both training and test data**.

**Analogy:**

It's like preparing for a math exam but only learning basic arithmetic. If the exam includes algebra, geometry, or advanced topics, you'll fail because you didn't learn enough.

**Indicators of Underfitting:**

* **Training Accuracy = Low**
* **Test Accuracy = Low**

**Example:**

You use a simple linear model to classify complex ad images. It might only get **50-60% accuracy** on both training and test sets because it can't capture the complexity in the data.

**Train/Test Split**

It is a simple method where you divide your dataset into two separate parts:

* **Training Set**: Used to train the machine learning model.
* **Test Set**: Used to evaluate the trained model’s performance on unseen data.

**🔹 Typical Split Ratios:**

* 70% training / 30% testing
* 80% training / 20% testing
* Sometimes, also split into Train/Validation/Test (60/20/20).

**🔹 Why do we need it?**

* To ensure that the model does not just memorize the training data (overfitting).
* To evaluate if the model generalizes well to new, unseen data.

**Cross-Validation (CV)**

**🔹 What is it?**

Cross-validation splits the dataset into **K equal-sized parts (folds)**. The model is trained and tested **K times**, each time using a different fold as the validation set and the rest as training data.

**🔹 Most common type:**

* **K-Fold Cross-Validation** (e.g., K=5 or K=10)

**🔹 How it works (5-Fold Example):**

1. Split dataset into 5 folds.
2. Train model on 4 folds, validate on the remaining 1 fold.
3. Repeat this process 5 times, each time changing the validation fold.
4. Average the performance over all 5 runs.

Dataset ---> [ Train | Test ] (Train/Test Split)

Dataset ---> [ Fold1 | Fold2 | Fold3 | Fold4 | Fold5 ] (Cross-Validation)

Train on Fold2+3+4+5, Test on Fold1

Train on Fold1+3+4+5, Test on Fold2

...

**What is Deep Learning?**

**Deep Learning** is a **subset of Machine Learning (ML)** that uses **artificial neural networks** with **many layers** to learn patterns from data.

Think of it like:

**Deep Learning = Machine Learning + Neural Networks with multiple layers**

It’s called "deep" because these networks have **many layers (depth)** between input and output.

A diagram of a network

AI-generated content may be incorrect.

A screenshot of a black screen

AI-generated content may be incorrect.

**Forward Propagation (FP)**

Forward Propagation is the process of passing input data through the network layers to produce an output (prediction).

**What happens step by step?**

1. Input Data → First Layer (Input Layer)
2. For each neuron:
   * Multiply **inputs by weights**
   * Add **bias**
   * Apply **activation function (like ReLU, Sigmoid, etc.)**
3. Pass the output to the **next layer**.
4. Repeat until the **Output Layer** gives the final prediction.

**Backpropagation in Neural Network**

*Backpropagation*is also known as "*Backward Propagation of Errors*" and it is a method used to train [**neural network**](https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/) . Its goal is to reduce the difference between the model’s predicted output and the actual output by adjusting the weights and biases in the network.

**🎯 What Is a Bias in Machine Learning?**

In a neural network (or any linear model), each **neuron** computes a weighted sum of its inputs and adds a **bias** term:

Output=w1x1+w2x2+⋯+wnxn+b\text{Output} = w\_1x\_1 + w\_2x\_2 + \dots + w\_nx\_n + bOutput=w1​x1​+w2​x2​+⋯+wn​xn​+b

* x1,x2,...,xnx\_1, x\_2, ..., x\_nx1​,x2​,...,xn​: input features (like pixel values)
* w1,w2,...,wnw\_1, w\_2, ..., w\_nw1​,w2​,...,wn​: weights (learned during training)
* bbb: **bias** (also learned during training)

**🧠 Why Do We Add a Bias?**

**Bias allows the model to shift the activation function to better fit the data**.

If we **didn't have bias**, the output would always go through the origin (0,0). That limits the flexibility of our model.

**🧮 Math Example Without Bias:**

Let’s say:

* You have only 1 input: x=2x = 2x=2
* Weight w=3w = 3w=3

Then:

Output=3×2=6\text{Output} = 3 \times 2 = 6Output=3×2=6

Now what if you need an output of 7 for this input? You **can’t do it** unless you add a bias:

Output=3×2+1=7\text{Output} = 3 \times 2 + 1 = 7Output=3×2+1=7

So the **bias shifts** the model's prediction up or down to better match the target.

**🏠 Real-World Analogy: The Rent Example**

Let’s say you’re trying to predict **house rent** based on the number of rooms:

| **Rooms** | **Rent** |
| --- | --- |
| 1 | ₹7000 |
| 2 | ₹8000 |
| 3 | ₹9000 |

**🧠 Observation:**

Every extra room increases rent by ₹1000 — that’s your **weight (w = 1000)**.

But even if a house had **0 rooms**, the rent is **not zero**, right? There's a base cost—**like location, electricity, or building access**.

That base cost is the **bias (b)**.

So your model becomes:

Rent=1000×rooms+bias\text{Rent} = 1000 \times \text{rooms} + \text{bias}Rent=1000×rooms+bias

Let’s say the bias = ₹6000 (base rent). Then:

* 1 room → ₹1000×1 + ₹6000 = ₹7000 ✅
* 2 rooms → ₹1000×2 + ₹6000 = ₹8000 ✅

**Without bias**, this wouldn’t be possible.

| **Concept** | **Explanation** |
| --- | --- |
| What is bias? | A constant value added to the weighted sum of inputs |
| Why needed? | To give the model flexibility and shift predictions to better fit data |
| Real-life use | Base rent in housing, base salary in income prediction, etc. |
| Without bias | The model is forced to always go through origin — reduces performance |

**🧠 In Neural Networks**

In a neural network:

* Bias lets the neuron **activate** even if all inputs are zero.
* It increases the **expressive power** of the network.

Each neuron has:

* **Weights**: Learn importance of each input.
* **Bias**: Adjusts the threshold at which the neuron "fires."

**Real-World Analogy:**

Imagine shooting arrows at a target:

* **Forward Propagation:** You shoot the arrow, and it hits somewhere (Prediction).
* **Backward Propagation:** You see how far you missed, adjust your aim (Weights), and try again.

**What is Linearity?**

A **linear relationship** means:

* **Input increases → Output increases (or decreases) at a constant rate.**
* The relationship is like a **straight line**:  
  y = mx + c

**What is Non-Linearity?**

A **non-linear relationship** means:

* The **output doesn’t change proportionally** with input.
* Sometimes small changes in input cause **big changes** in output.
* The relationship could be a curve, zigzag, step, etc.

**Without non-linearity:**

* Neural network behaves like **one big linear equation** → can’t model complex data (like images, voices).

**Activation Functions Add This Non-Linearity!**

* Functions like **ReLU, Sigmoid, Tanh** "bend" the straight line.
* They let the network adapt to complex patterns.

**Why is Non-Linearity Important in Neural Networks?**

* Neural networks consist of neurons that operate using **weights**, **biases**, and **activation functions**.
* In the learning process, these weights and biases are updated based on the error produced at the outut—a process known as **backpropagation**. Activation functions enable backpropagation by providing gradients that are essential for updating the weights and biases.

->Images have curves, textures, lighting variations, so non-linearity is needed

->Voice data includes tone, pitch, speed which cannot be captured by a linear model

**Activation Functions:**

**1. Linear Activation Function**

Linear Activation Function resembles straight line define by y=x. No matter how many layers the neural network contains, if they all use linear activation functions, the output is a linear combination of the input.

* The range of the output spans from (−∞ to +∞)(−∞ to +∞).
* Linear activation function is used at just one place i.e. output layer.
* Using linear activation across all layers makes the network’s ability to learn complex patterns limited.

**Why is it NOT preferred in deep learning?**

| **Problem** | **Reason** |
| --- | --- |
| **No Non-Linearity** | It can't capture complex patterns or decision boundaries. |
| **Entire network behaves linear** | No matter how many layers you stack, it's just one big linear equation. |
| **Poor learning capacity** | It lacks flexibility to model real-world data (images, speech, etc.) |

**Where is it used?**

🔸 **Output layer of regression problems:**

* Because sometimes, you want the output to be **any real value**, not squashed between 0 and 1.

Example:

* Predicting house prices, stock prices, etc.

**Non-linear activation functions**

**1.Sigmoid Function**

It  is characterized by ‘S’ shape.

This formula ensures a smooth and continuous output that is essential for gradient-based optimization methods.

* It allows neural networks to handle and model complex patterns that linear equations cannot.
* The output ranges between 0 and 1, hence useful for binary classification.

**Summary:**

* **Squashes input to (0,1)**.
* Great for **probability outputs**.
* **Non-linear** → Helps network learn complex patterns.
* Can cause **vanishing gradient problem** → less used in deep nets now.

**What is Vanishing Gradient?**

When training deep neural networks using **backpropagation**, the **gradients** (partial derivatives of the loss w.r.t weights) can become **very small (close to zero)** as they propagate backward from output layers to input layers.

This means:

**Early layers (near the input) get very tiny updates** → They stop learning → Training becomes very slow or even completely **stuck**.

The more layers, the more multiplication of small derivatives.

Result: **Early layers receive near-zero gradients** → **Weights stop updating**.

**How to fix this**: Using ReLu and others

**Gradient Recap:**

When we train a neural network, **gradients** are used to update weights in each layer using:

w=w−η⋅∂L∂ww = w - \eta \cdot \frac{\partial L}{\partial w}w=w−η⋅∂w∂L​

Where:

* ∂L∂w\frac{\partial L}{\partial w}∂w∂L​ is the **gradient**
* If this gradient is **very small (close to zero)**, the weight update becomes **negligible**

**🔍 So What Happens With Near-Zero Gradients?**

**It doesn’t mean the layer gets less data.**

📌 Instead, it means:

The layer gets **almost no signal** about **how to improve** its weights.

Let’s explain with a real-world analogy:

**🏠 Real-World Analogy: Whispering Instructions**

Imagine a team of workers passing instructions down a chain — but:

* Each person whispers just **10%** of what they heard to the next person.
* By the time the message reaches the **first worker**, it's just a **faint whisper**.

So the first worker (early layer in the network) has **no clear idea** what to do.

This is what happens with **vanishing gradients**.

**📉 Technical Explanation**

During **backpropagation**, gradients are calculated **layer by layer**, starting from the output layer.

If each layer’s derivative is < 1 (e.g. sigmoid outputs in range 0–1), then multiplying them over many layers:

final gradient=small numbermany times→near zero\text{final gradient} = \text{small number}^{\text{many times}} \rightarrow \text{near zero}final gradient=small numbermany times→near zero

So, early layers get **vanishingly small updates** → their weights don’t change → they stop learning.

**🚫 Why Is This Bad?**

* Early layers learn **low-level features** like edges, textures, etc.
* If they don’t learn well, the entire model suffers, because later layers rely on good base representations.

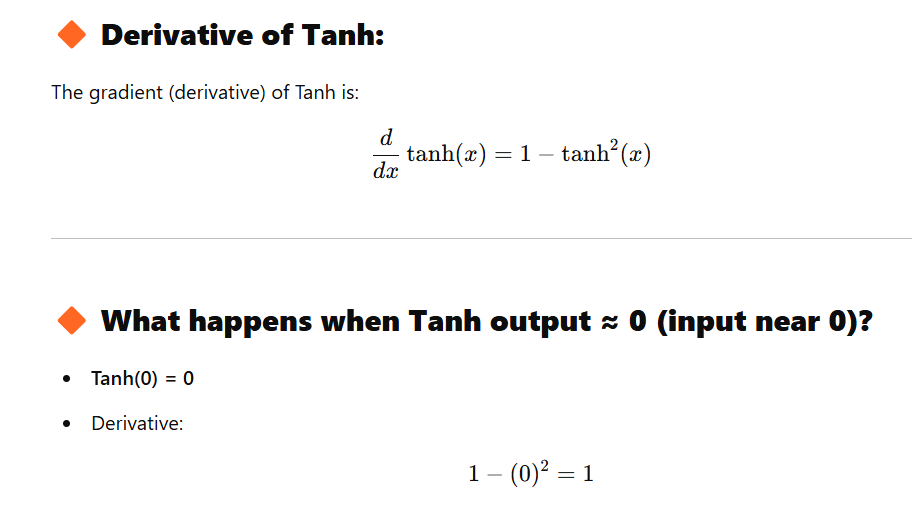
**✅ Summary**

| **Concept** | **Meaning** |
| --- | --- |
| Near-zero gradient | Gradient is almost 0 → weights don’t update |
| Result | Early layers **stop learning** effectively |
| Not about data | They get the full input data — but receive **no learning signal** |
| Caused by | Too many small gradient multiplications (like sigmoid derivatives < 1) |
| Impact | Model **fails to converge** or takes forever to train |

**🛠 How Do We Fix It?**

* ✅ Use better activation functions: **ReLU** (Rectified Linear Unit) → doesn’t squash gradients
* ✅ Use techniques like **Batch Normalization**
* ✅ Use architectures like **ResNet** with **residual connections** to help gradients flow back more directly

**2. Tanh Activation Function**

It has an S-shaped curve (like Sigmoid), but it is centered around zero (which is an improvement over Sigmoid). 

**Output Range:**

* The output values range between **-1 and 1**.

**Advantages:**

Zero-centered data means that the average (mean) value of the data is zero

1. **Zero-centered output:**
   * Helps in optimization because the mean of the activations is closer to zero, making gradient descent more efficient.
2. **Smooth gradients:**
   * Useful for problems where inputs can have negative, zero, and positive values.
3. .

**🎯 Real-Life Analogy**

Imagine you’re steering a car that can only turn **right** (sigmoid behavior). It’s really hard to go straight or reach your target unless you zig-zag awkwardly.

Now, imagine you can turn both **left and right** (tanh behavior). You can reach your destination **faster and more accurately**.

**When to Use:**

* It is often used in **hidden layers** where zero-centered data is preferable.
* Still, it's mostly replaced by **ReLU** nowadays due to the vanishing gradient issue, but in some cases (like RNNs), **Tanh** is still popular.

|  |  |
| --- | --- |
| Tanh output near 0 | Gradient is large → **Good learning** |
| Tanh output near -1 or 1 | Gradient near zero → **Slow/no learning** |
| Deeper network with Tanh layers | **Vanishing gradients issue** becomes worse |

**🔶 Solution?**

* Use **ReLU** or other activation functions in deep networks to avoid vanishing gradients.
* Proper initialization, normalization, or using residual connections (ResNet) also helps.

**ReLU (Rectified Linear Unit) Activation**

It is defined by A(x)=max (0,x)*A*(*x*)=max(0,*x*), this means that if the input x is positive, ReLU returns x, if the input is negative, it returns 0.

* **Value Range**: [0,∞)[0,∞), meaning the function only outputs non-negative values.
* **Nature**: It is a **non-linear** activation function, allowing neural networks to learn complex patterns and making backpropagation more efficient.
* **Advantage over other Activation:**ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

**Why is ReLU Popular?**

1. **Non-linearity:** Allows the neural network to learn complex patterns.
2. **Computationally Simple:** Just max(0, x), no exponentials or divisions.
3. **Avoids Vanishing Gradient Problem (for positive x):**
   * Derivative is **1** for **x > 0**, gradients don’t vanish easily.
4. **Efficient Convergence:**
   * Faster training than Sigmoid/Tanh.

**Problem: Dying ReLU**

* For **x ≤ 0**, gradient = **0**.
* If too many neurons output **0**, they **stop learning (no gradient updates)** → This is called **Dying ReLU** problem.

**Definition:**  
ReLU stands for **Rectified Linear Unit**. It is an activation function used in neural networks, especially in hidden layers.

**Formula:**

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

This means:

* If x > 0, then f(x) = x
* If x <= 0, then f(x) = 0

**🔶 Why Use ReLU?**

ReLU helps solve the **vanishing gradient problem** seen in earlier activation functions like **sigmoid** and **tanh**. These earlier functions squash input values into a small range, leading to very small gradients (derivatives), which slows or stops learning during **backpropagation**.

In contrast, ReLU:

* Outputs **0** for negative inputs (which helps with sparsity)
* Outputs **x** for positive inputs (preserves gradients and speeds up learning)

**🔶 How ReLU Works:**

**🔹 Forward Propagation:**

* Input: xxx
* Output: max⁡(0,x)\max(0, x)max(0,x)
  + Example: x=−5→Output=0x = -5 \rightarrow \text{Output} = 0x=−5→Output=0
  + Example: x=3→Output=3x = 3 \rightarrow \text{Output} = 3x=3→Output=3

**🔹 Backward Propagation (Gradient):**

* Derivative of ReLU:
  + If x>0→dfdx=1x > 0 \rightarrow \frac{df}{dx} = 1x>0→dxdf​=1
  + If x<=0→dfdx=0x <= 0 \rightarrow \frac{df}{dx} = 0x<=0→dxdf​=0

This makes the gradient simple — either **0** or **1**, avoiding the vanishing gradient issue (like in sigmoid, where gradient could go down to 0.01 or lower).

**🔶 Weight Update with ReLU**

During backpropagation, weights are updated using this formula:

Wnew=Wold−learning rate×d(Loss)dWW\_{\text{new}} = W\_{\text{old}} - \text{learning rate} \times \frac{d(\text{Loss})}{dW}Wnew​=Wold​−learning rate×dWd(Loss)​

If the ReLU derivative = **1**, the gradient is passed through properly, and weights are updated.

If the ReLU derivative = **0** (i.e., when input is negative), **no weight update happens** — the weight stays the same.

**🔶 Dead Neurons Problem**

If a neuron always receives **negative inputs**, its output is always **0**, and its gradient is always **0** during backpropagation.

Result:

* No weight updates for this neuron
* The neuron becomes inactive or **"dead"**
* It doesn’t contribute to learning anymore

This is a major **disadvantage of ReLU**.

**🔶 Advantages of ReLU**

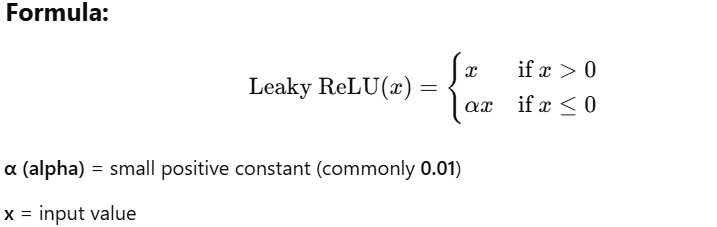
1. ✅ **Solves Vanishing Gradient Problem**  
   Gradients are either 0 or 1 (instead of near-zero values like in sigmoid/tanh).
2. ✅ **Computationally Efficient**  
   It's just a simple max(0, x) operation — very fast to compute.
3. ✅ **Linear-like Behavior**
   * Positive values pass through unchanged.
   * Helps build piecewise linear models that are easy to optimize.
4. ✅ **Sparse Activation**  
   Many neurons output 0, which adds sparsity and improves efficiency.

**🔶 Disadvantages of ReLU**

1. ❌ **Dead Neurons**  
   If input is always negative, the neuron stops updating and becomes useless.
2. ❌ **Not Zero-Centered**  
   Outputs are always ≥ 0, which might cause issues in some optimization techniques.

**4. Leaky ReLU?**

Leaky ReLU (Leaky Rectified Linear Unit) is a **modified version of ReLU** designed to fix one major problem of ReLU: the "**dying ReLU problem**".



**Graph:**

* For **positive inputs**, behaves exactly like **ReLU** (output = input).
* For **negative inputs**, instead of outputting zero, it gives a small negative slope (**αx**), allowing a **small gradient**.

**Why Leaky ReLU?**

🔸 **Problem with ReLU:**

* In ReLU, **negative inputs → output = 0**.
* For large parts of data, neurons might output **zero gradients** → **"Dying ReLU Problem"** (no learning, neuron stuck).

🔸 **Solution with Leaky ReLU:**

* It **allows small negative outputs** → **gradient never becomes zero** → avoids dead neurons.

**Pros:**

1. **Fixes Dying Neurons:** Small negative slope keeps neurons active even for negative inputs.
2. **Fast computation:** Simple and efficient like ReLU.
3. **Good gradient flow:** Reduces risk of vanishing gradient on negative side.

**Cons:**

1. **α (alpha) is manually set:** Needs tuning, not always optimal.
2. **Less biologically motivated.**

**Optimizers**

Optimizers are algorithms or methods used to adjust the weights and biases of a neural network to minimize the loss function.

**Why Do We Need Optimizers?**

* After **forward propagation**, the model calculates an output.
* The difference between the predicted output and the actual output is calculated using a **loss function** (error).
* The **optimizer** uses **gradients (from backpropagation)** to update weights to **reduce this error** step by step.

**How Do Optimizers Work?**

1. **Calculate gradients (using backpropagation).**
2. **Update weights and biases** using the gradients.
3. **Repeat for multiple epochs** until loss is minimized.

**Epoch** refers to **one complete pass** through the **entire training dataset** by the learning algorithm (neural network).

**In Simple Words:**

* You have a dataset with, say, **1000 images**.
* **1 Epoch = Model sees all 1000 images once (forward + backward pass).**
* After 1 epoch, the model updates weights **once for all data points**.
* Typically, training happens over **multiple epochs** (like 10, 50, 100) to improve performance.

**Why Do We Need Multiple Epochs?**

The model **doesn’t learn well in just one pass**.

* **1st Epoch:** Random weights, initial predictions are poor.
* **Subsequent Epochs:** Model adjusts weights based on errors, gets better.
* After **many epochs**, model minimizes the loss and generalizes well.

**What is Transfer Learning?**

Transfer learning involves taking a **pre-trained** model (like ResNet50, which is trained on ImageNet) and adapting it to a new but related task (advertisement detection). This helps in:

* Reducing the training time.
* Improving accuracy with less data.
* Leveraging knowledge from large-scale datasets

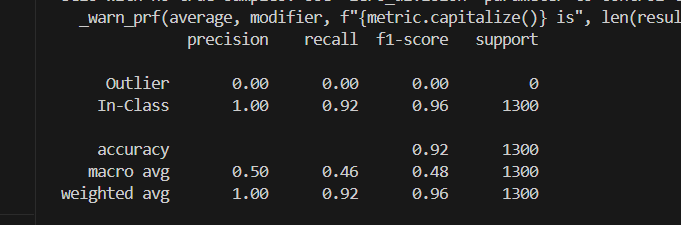
You can load a pre-trained ResNet50 model from TensorFlow/Keras or PyTorch. It has already learned useful **low-level features** (edges, textures) and **high-level features** (shapes, objects) from ImageNet.

**Loss Functions:**

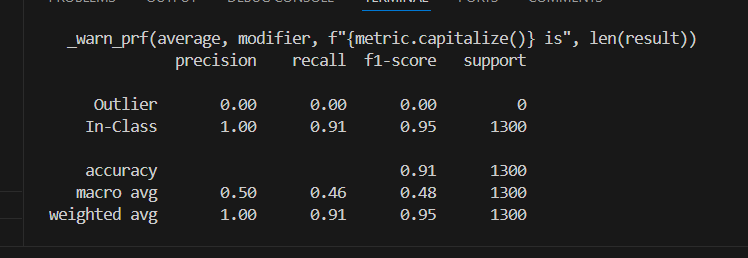
Quantify the deviation of the predicted output by the neural network to the expected output.

A black background with white text

AI-generated content may be incorrect.



For vgg16



Resnet50

**Preprocessing of Input in code**

Each **pretrained deep learning model** (like ResNet50, VGG16, MobileNet) was trained on a specific dataset (e.g., **ImageNet**) using a particular **image normalization method**. When using these models for inference or fine-tuning, **the input images must be preprocessed in the same way** as they were during training.

If we **do not preprocess the input**, the model will receive **incorrectly scaled data**, leading to:

✅ **Poor accuracy**  
✅ **Wrong predictions**  
✅ **Unstable training during fine-tuning**

Yes, but **different models have different preprocessing requirements**:

| **Model** | **Preprocessing Steps** |
| --- | --- |
| **ResNet50** | Subtracts mean RGB values (123.68, 116.78, 103.94) |
| **VGG16/VGG19** | Similar mean subtraction |
| **MobileNet** | Scales pixels between -1 and 1 |
| **EfficientNet** | Scales pixels between 0 and 1 |

**Pooling=Avg :**

**Input Image (224x224x3)**

**│**

**▼**

**ResNet50 (Pretrained CNN)**

**│**

**▼**

**Last Convolutional Layer → Feature Maps (7x7x2048)**

**│**

**▼**

**Global Average Pooling (GAP) → Averages each 7x7 feature map**

**│**

**▼**

**Output Feature Vector (2048,)**

The **last convolutional layer** outputs a **(7×7×2048) feature map**.

pooling="avg" **averages each feature map** across all 7×7 spatial locations.  
The output is a **2048-dimensional feature vector**.

| **Pooling Type** | **Effect** | **Feature Vector Shape** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| pooling="avg" (Global Average Pooling) | Averages the values of each feature map | (2048,) |

|  |  |  |
| --- | --- | --- |
| pooling="max" (Global Max Pooling) | Takes the maximum value from each feature map | (2048,) |

|  |  |  |
| --- | --- | --- |
| pooling=None | Keeps original (7×7×2048) feature maps | (7,7,2048) |

**Feature Vector Shape: (1, 2048)**

**Now, this feature vector can be used for training a classifier like SVM or Logistic Regression.**

**# Function to extract CNN features**

def extract\_features(image\_path, target\_size=(224, 224)):

    img = load\_img(image\_path, target\_size=target\_size)

    img\_array = img\_to\_array(img)

    img\_array = np.expand\_dims(img\_array, axis=0)

    img\_array = preprocess\_input(img\_array)  # Normalize for ResNet

    features = resnet\_model.predict(img\_array)

    return features.flatten()

**Code Explanation:**

def extract\_features(image\_path, target\_size=(224, 224)):

* **Defines a function** named extract\_features that takes:
  + image\_path: The file path of the image.
  + target\_size=(224, 224): Specifies that the image will be resized to **224×224 pixels** (ResNet50 requires this input size).

img = load\_img(image\_path, target\_size=target\_size)

* **Loads the image** from the given path.
* **Resizes** it to **224×224 pixels** to match ResNet50's expected input size.

🔹 *Why?*  
✅ ResNet50 is trained on **224×224 images** (ImageNet dataset), so resizing ensures compatibility.

img\_array = img\_to\_array(img)

* Converts the image into a **NumPy array** format.
* The shape of img\_array is (224, 224, 3), where:
  + **224x224** → Image dimensions.
  + **3** → RGB color channels.

🔹 *Why?*  
✅ Deep learning models require numerical data (not raw image files), so we convert the image into an array.

img\_array = np.expand\_dims(img\_array, axis=0)

* Expands dimensions from (224, 224, 3) → (1, 224, 224, 3).
* Adds a **batch dimension** so the model can process a **single image as a batch of size 1**.

🔹 *Why?*  
✅ TensorFlow models expect inputs in batch format: (batch\_size, height, width, channels).

img\_array = preprocess\_input(img\_array) # Normalize for ResNet

* **Applies ResNet-specific preprocessing**:
  + Rescales pixel values to match ResNet50’s expected format (which uses **ImageNet normalization**).

🔹 *Why?*  
✅ **Normalization ensures the model performs optimally** by matching the data distribution of ImageNet training.

features = resnet\_model.predict(img\_array)

* **Feeds the processed image** into the **pre-trained ResNet50 model**.
* The **model outputs high-level feature vectors** instead of classification labels.
* The output shape is (1, 2048), as ResNet50 with pooling="avg" gives a **2048-dimensional vector**.

🔹 *Why?*  
✅ These **deep features** capture the **content** of the image and can be used for tasks like:

* Image classification
* Object detection
* Clustering
* Anomaly detection (e.g., One-Class SVM)

return features.flatten()

* **Flattens the output** from (1, 2048) → (2048,) (a 1D array).
* The result is a **compact, meaningful representation of the image**.

🔹 *Why?*  
✅ Many ML models (SVM, KNN, etc.) work best with 1D feature vectors.

# Function to load and process dataset

def load\_data\_with\_features(data\_dir):

    feature\_list = []

    image\_paths = []

    # Ensure the directory exists

    if not os.path.exists(data\_dir):

        raise FileNotFoundError(f"Dataset directory '{data\_dir}' not found.")

    # Load images from the advertisement folder

    advertisement\_dir = os.path.join(data\_dir, "advertisement")

    for file\_name in os.listdir(advertisement\_dir):

        file\_path = os.path.join(advertisement\_dir, file\_name)

        if file\_path.lower().endswith(('.jpg', '.png', '.jpeg')):  # Ensure it's an image

            feature\_vector = extract\_features(file\_path)

            feature\_list.append(feature\_vector)

            image\_paths.append(file\_path)

    return np.array(feature\_list), image\_paths

**Line-by-Line Breakdown**

def load\_data\_with\_features(data\_dir):

* Defines a function named load\_data\_with\_features(), which **takes data\_dir as input**.
* data\_dir is expected to be the **path to a dataset folder** (e.g., "./dataset").

feature\_list = []

image\_paths = []

* **feature\_list**: Stores the **feature vectors** of all images.
* **image\_paths**: Stores the **file paths** of images (useful for tracking which image corresponds to which features).

# Ensure the directory exists

if not os.path.exists(data\_dir):

raise FileNotFoundError(f"Dataset directory '{data\_dir}' not found.")

* **Checks if the dataset directory exists**.
* If data\_dir does not exist, it raises a FileNotFoundError to avoid errors later in the function.

# Load images from the advertisement folder

advertisement\_dir = os.path.join(data\_dir, "advertisement")

* Joins the dataset path (data\_dir) with "advertisement", assuming the dataset contains a subfolder named "advertisement" where images are stored.
* This means the dataset structure is expected to be:

dataset/

├── advertisement/

│ ├── image1.jpg

│ ├── image2.jpg

│ ├── ...

for file\_name in os.listdir(advertisement\_dir):

* **Iterates over all files** in the "advertisement" folder.

file\_path = os.path.join(advertisement\_dir, file\_name)

* **Constructs the full path** of the image file.

if file\_path.lower().endswith(('.jpg', '.png', '.jpeg')): # Ensure it's an image

* **Checks if the file is an image** by looking at its extension (.jpg, .png, .jpeg).
* Ensures non-image files (like .txt or .csv) are **ignored**.

feature\_vector = extract\_features(file\_path)

* **Extracts features** from the image using the extract\_features() function (which likely uses ResNet50).
* Converts the image into a **feature vector** (e.g., a 2048-dimensional array).

feature\_list.append(feature\_vector)

image\_paths.append(file\_path)

* **Stores the extracted features** in feature\_list.
* **Stores the corresponding image path** in image\_paths for reference.

return np.array(feature\_list), image\_paths

* **Converts the feature list to a NumPy array** (for efficient processing).
* **Returns the extracted features** and **corresponding image paths**.

X\_train, X\_test = train\_test\_split(data\_scaled, test\_size=0.2, random\_state=42)

**Why is random\_state Important?**

1. **Ensures Reproducibility**
   * If random\_state is **not set**, the train-test split changes every time you run the code.
   * Setting random\_state=42 (or any fixed value) ensures the **same split occurs** every time.
2. **Consistency for Model Comparison**
   * If you're testing different models, you want them to be trained on the **same training set** and tested on the **same test set**.
   * random\_state ensures **fair comparison** between models.

**Clear CNN(krish)**

**🌈 RGB vs Grayscale Images – Explained**

**📷 1. What is an Image (Digitally)?**

An image is represented as a **matrix of pixel values**. Each pixel contains **intensity information** that determines how that pixel appears on a screen.

**⚫ 2. Grayscale Images**

* **Definition**: A grayscale image contains shades of gray only, without any color.
* **Pixel Values**: Each pixel value ranges from **0 to 255**:
  + 0 = black
  + 255 = white
  + values in between = different shades of gray
* **Shape**: The image array will have shape:  
  height × width × 1 (e.g., 28×28×1)
* **Channels**: Only **1 channel** (gray).

✅ **Used In**:

* Simpler models
* MNIST dataset (handwritten digits)
* When color is not important for the task

**🌈 3. RGB (Color) Images**

* **Definition**: RGB images use 3 color channels: **Red**, **Green**, and **Blue**
* Each pixel is represented by 3 values:
  + One for Red (R)
  + One for Green (G)
  + One for Blue (B)
* These values are also in the range **0–255**
* Final color is a **combination** of these three channels
* **Shape**: The image array will have shape:  
  height × width × 3 (e.g., 32×32×3)
* **Channels**: 3

✅ **Used In**:

* Most modern datasets (CIFAR-10, ImageNet, etc.)
* Applications involving object detection, facial recognition, etc.

**🔍 4. Example – Dimensions of Images**

| **Type of Image** | **Example Dimensions** | **Meaning** |
| --- | --- | --- |
| Grayscale | 6×6×1 | 6x6 pixels, 1 color channel |
| RGB | 4×4×3 | 4x4 pixels, 3 color channels |

**🔧 5. Pixel Values and Channels**

For RGB:

* The **same pixel location** across R, G, and B channels can have **different values**
* Combining these three values gives the **final color** of that pixel

For Grayscale:

* Only **one value** per pixel defines the brightness/shade

**🔄 6. Conversion: RGB to Grayscale**

Sometimes, RGB images are converted to grayscale for simplicity:

* The common formula (approximate) is:  
  Gray = 0.299 \* R + 0.587 \* G + 0.114 \* B
* This weighted average reflects human sensitivity to different colors (green is perceived stronger than red or blue)
* **Value = 0** → completely black
* **Value = 255** → completely white
* **Value = 128** → medium gray
* **Value = 75, 150, etc.** → various shades of gray in between black and white

Each **"shade"** is just a different **brightness level**. More the value, the **lighter** the gray; lower the value, the **darker** the gray.

**📘 Example:**

Let’s say you have a pixel with the following values:

| **Channel** | **Value** |
| --- | --- |
| Red | 255 |
| Green | 0 |
| Blue | 0 |

✅ This pixel will appear **pure red**, because:

* Red is at its maximum intensity,
* Green and Blue are turned off (value = 0).

Now another example:

| **Channel** | **Value** |
| --- | --- |
| Red | 255 |
| Green | 255 |
| Blue | 0 |

✅ This pixel will appear **yellow**, because:

* Red + Green = Yellow when combined in light
* Blue is off

It uses light logic not paint logic 🡪imp

**✅ 1. Image Basics:**

* A **grayscale image** of size 6x6 means it has:
  + **6 rows** and **6 columns**
  + Only **1 channel** (gray), so it's represented as 6x6x1
  + Each pixel value ranges from **0 (black)** to **255 (white)**

**✅ 2. Normalization:**

* Pixel values are **normalized to the range [0, 1]**
* This is done by dividing each pixel by **255**
  + So:  
    0 → 0.0 (black)  
    255 → 1.0 (white)
* This is essential because neural networks perform better on normalized data.

**✅ 3. Convolution Operation:**

* This is the **core operation** in CNNs.
* A **filter (or kernel)** — in this case, a 3x3 matrix — is slid across the image.
* At each position, a **dot product** is calculated between:
  + The **filter values**
  + The **underlying 3x3 portion** of the image
* The results are summed up to produce **a single number** at that position in the output feature map.

**✅ 4. Stride:**

* **Stride** defines how much the filter moves at each step.
* A **stride of 1** means the filter moves **one pixel at a time**.
* A larger stride reduces the size of the output (faster but less detailed).

**✅ 5. Output Dimensions:**

* If a 3x3 filter with **stride = 1** and **no padding** is applied to a 6x6 image, the output is:

(6−3)+1=4⇒4x4(6 - 3) + 1 = 4 \Rightarrow 4x4(6−3)+1=4⇒4x4

* This 4x4 matrix is the **feature map**.

**✅ 6. Purpose of Filters:**

* The filter shown:

text

CopyEdit

[[ 1, 0, -1],

[ 2, 0, -2],

[ 1, 0, -1]]

is a **Sobel filter** used for **detecting vertical edges**.

* It produces high values where there's a **vertical contrast** between pixels (like a black-white boundary).

**✅ 7. Output Interpretation:**

* The resulting values can be **positive, zero, or negative**
* These are then either:
  + Left as-is
  + Passed through **activation functions** like **ReLU** to remove negative values
* The output highlights **edges or patterns** in the image
  + In your example, the black/white edge got converted into a **distinct edge-detected map**

**✅ 8. Multiple Filters in Real CNNs:**

* Real CNNs don’t use fixed filters like Sobel manually.
* Instead, filters are **learned during training**
  + They're initialized randomly
  + Updated through **backpropagation** to extract useful features (edges, textures, shapes)

**📌 What Is Padding in CNN?**

When we apply a convolution operation to an image using a filter (also called kernel), the **size of the output (called the feature map)** often becomes **smaller** than the input image.

🔴 This reduction means we’re **losing edge information** of the image, which might be important for learning.

To **prevent this loss of information**, we use **padding**, which is simply adding extra rows and columns (usually with zeros) around the input image.

**🧠 Example Without Padding**

Let's say:

* Input image size: **6 × 6**
* Filter size: **3 × 3**
* No padding
* Stride = 1 (default)

**Formula for Output Size:**

Output size=n−f+1\text{Output size} = n - f + 1Output size=n−f+1

Where:

* n = input size
* f = filter size

Output size=6−3+1=4\text{Output size} = 6 - 3 + 1 = 4Output size=6−3+1=4

So the **output will be 4 × 4** — smaller than the original input (6 × 6). That means **we lost some information** from the edges.

**🛡️ How Padding Helps**

To **preserve the original input size**, we use **padding** by adding extra rows and columns around the image.

**🔧 Updated Formula (with Padding)**

When we apply padding (denoted by p), the formula becomes:

Output size=n−f+2p+1\text{Output size} = n - f + 2p + 1Output size=n−f+2p+1

We want the output size to be **equal to the input size n**, so:

n−f+2p+1=nn - f + 2p + 1 = nn−f+2p+1=n

Subtract n from both sides:

−f+2p+1=0⇒2p=f−1⇒p=f−12-f + 2p + 1 = 0 \Rightarrow 2p = f - 1 \Rightarrow p = \frac{f - 1}{2}−f+2p+1=0⇒2p=f−1⇒p=2f−1​

**✅ Example 1: 6×6 Image, 3×3 Filter**

p=3−12=22=1p = \frac{3 - 1}{2} = \frac{2}{2} = 1p=23−1​=22​=1

→ Add **1 layer of padding** on all sides.

So the image becomes **(6 + 2 × 1) = 8 × 8**, but only the center 6×6 is your actual data — the rest is the padded border.

Apply 3×3 filter → output will be 6×6.

🎯 So padding helped us **preserve the original image size** in the output.

**🔁 Two Types of Padding**

1. **Zero Padding**
   * Add zeros around the border.
   * Simple and most common.
   * Keeps computation fast and simple.
2. **Neighbor Padding**
   * Copy values from the border of the image (e.g., if the top-left pixel is 4, add 4 above it too).
   * Less common, but sometimes used for edge-sensitive tasks.

**1. Structure and Input**

* **ANN**:
  + Input: Usually a **1D vector** (e.g., features of a dataset).
  + Layers: Input layer → Hidden layer(s) → Output layer.
  + Operation: Every input node is connected to every neuron in the next layer with a weight.
* **CNN**:
  + Input: Usually a **2D image (or 3D if RGB)**.
  + Layers: Instead of fully connected layers, uses **filters/kernels** to scan the image and extract spatial features.
  + Operation: Uses **convolutional layers** that apply filters over the input image.

**2. Operation: How they compute**

* **ANN Operation**:
  + Take the input vector x.
  + Multiply by weights W and add bias b.
  + Compute:

z=WTx+bz = W^T x + bz=WTx+b

* + Then apply activation function (e.g., ReLU):

a=ReLU(z)a = \text{ReLU}(z)a=ReLU(z)

* + Repeat for layers until output layer.
* **CNN Operation**:
  + Take the input image (e.g., 6x6 pixels).
  + Apply a **filter (kernel)** (e.g., 3x3) that slides over the image — this is called **convolution**.
  + At each position, multiply overlapping pixel values by filter weights and sum them up → this forms one element of the output feature map.
  + Use **multiple filters** (F1, F2, ..., Fn), each detecting different features like edges, textures, or shapes.
  + Output is a set of **feature maps** (one per filter).

**3. Filters in CNN**

* The filters in CNN work somewhat like neurons in ANN but are **spatially localized** and scan across the image.
* Filters **start with random values** initially.
* During training (using forward and backward propagation), these filter weights get **updated** so that they detect meaningful patterns in images (edges, colors, shapes, etc.).

**4. Activation Function: ReLU**

* Both ANN and CNN use activation functions like **ReLU** after their main operations.
* ReLU (Rectified Linear Unit) is defined as:

ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0, x)ReLU(x)=max(0,x)

* Why ReLU?
  + It introduces **non-linearity** so the network can learn complex patterns.
  + It helps with **backpropagation** because its derivative is simple (either 0 or 1), which avoids vanishing gradient problems and makes training easier.

**5. Forward and Backward Propagation**

* In **ANN**, forward pass: multiply inputs by weights, add bias, activate → repeat for layers.
* In **CNN**, forward pass: convolve image with filters, add bias, activate (ReLU) → get feature maps.
* In both, during backpropagation, errors are used to update weights (ANN weights or CNN filter weights).

**✅ 1. What is a Convolution Layer?**

A **convolution layer** is the core building block of a **Convolutional Neural Network (CNN)**.

It includes:

* **Convolution Operation**: Applying a **filter (or kernel)** on the image to extract features like edges, textures, etc.
* **Activation Function** (like ReLU): Applied to the result to introduce **non-linearity** and enable the model to learn complex patterns.

🔁 You can **stack multiple convolution layers** one after another to extract **deeper features**.

**✅ 2. What is Pooling?**

**Pooling** is a downsampling technique applied **after** a convolution layer to:

* Reduce the **spatial size** (width × height) of the output.
* Make the network **faster and less prone to overfitting**.
* Focus on the **most important features**.

There are 3 main types:

**🟦 A. Max Pooling:**

* Takes the **maximum value** in each patch (e.g., 2×2 window).
* Helps to **retain the strongest features**.
* **Example**:  
  Suppose after convolution + ReLU, the output is:

CopyEdit

1 2 3

4 3 6

2 8 4

Applying 2×2 **max pooling** with stride 2:

* + First 2×2 window: [1, 2, 4, 3] → max = **4**
  + Next 2×2 window: [3, 6, 8, 4] → max = **8**

Resulting output:

CopyEdit

4 6

8 4

✅ **Why Max Pooling?**  
It helps the model become **location invariant** — it doesn't matter exactly **where** a feature appears, as long as it **does appear**.

**🟨 B. Min Pooling:**

* Takes the **minimum value** in each 2×2 window.
* Rarely used, but useful when **low-intensity features** are important.
* Using the same example, we get:

CopyEdit

1 3

2 4

**🟩 C. Average Pooling (Mean Pooling):**

* Takes the **average of all values** in each 2×2 window.
* Smoothens the image.
* Example output:

CopyEdit

(1+2+4+3)/4 = 2.5

(3+6+8+4)/4 = 5.25

**✅ 3. What is Location Invariance?**

* Location invariance means the model should recognize a pattern **no matter where** it appears in the image.
* **Max pooling helps** with this by reducing the spatial size and keeping only the strongest features (like cat faces, car wheels, etc.).
* This is important because an object might appear in **different positions** in different images.

**2. Why is this useful?**

* Suppose you're building a CNN to recognize a **cat’s face** in an image.
* A cat face could be in the **top-left**, **center**, or **bottom-right** of different pictures.
* You don't want your model to only recognize the cat if it's in one fixed location.

**3. How max pooling helps:**

* After the convolution operation (which detects features like edges, corners, etc.), **max pooling reduces the size** of the feature map by **picking the most important (max) value** in each region.
* This reduces sensitivity to **small shifts and movements** in the input.

Example:  
If a cat’s ear shifts slightly in position in two images, the convolution output might change slightly. But after max pooling, the key feature (the ear) is still captured — just in a slightly different place in a **smaller version** of the image.

**🔁 So what does this sentence mean?**

“It doesn’t matter exactly where a feature appears, as long as it does appear”

It means:  
CNN with pooling layers **does not care about the exact position** of the feature (like a cat’s eye or a car’s wheel).  
**If the feature is present anywhere in the image**, the model can detect it and use it to make predictions.

CNNs are used to classify images (like handwritten digits in the MNIST dataset) by automatically learning the important features in those images. Here's the breakdown:

**📷 Step 1: Input Image**

* Dataset: **MNIST**
* Image: A grayscale image of a digit (like 2, 7, or 9)
* Size: **28 × 28 × 1**
  + 28 x 28 is the width and height
  + 1 means it's grayscale (only 1 color channel)

**🔍 Step 2: Convolution Layer**

* You apply a **filter** (also called a **kernel**) of size **5 × 5**
* This filter "slides" over the image and performs a small matrix multiplication at each step to detect features like edges, curves, etc.
* No **padding** is used here (called **valid padding**), so the output becomes:

java

CopyEdit

Output Size = (28 - 5 + 1) × (28 - 5 + 1) = 24 × 24

* You usually use **multiple filters** (e.g., 10 filters), so the final output is:

CopyEdit

24 × 24 × 10 → 10 feature maps

**⚡ Step 3: Activation Function (ReLU)**

* **ReLU (Rectified Linear Unit)** is applied after convolution.
* It removes negative values and keeps only positive features (important for non-linearity).

**🧊 Step 4: Max Pooling**

* You apply a **2 × 2** filter that takes the **maximum value** from each 2×2 block.
* This reduces the size by half:

css

CopyEdit

From 24 × 24 → 12 × 12

* Output after max pooling:

CopyEdit

12 × 12 × 10

* Why max pooling? It keeps the most important feature and **makes the network location invariant** — it doesn’t matter where the feature appears, as long as it’s detected.

**🔁 Step 5: Repeat (Conv + ReLU + Pool)**

* Apply **another convolution** (again with 5 × 5 filters) on the 12 × 12 input:

CopyEdit

12 - 5 + 1 = 8 → Output becomes 8 × 8

* Then apply **another max pooling (2 × 2)**:

CopyEdit

8 × 8 → 4 × 4

* Final output at this stage:

scss

CopyEdit

4 × 4 × (number of filters)

**🔄 Step 6: Flattening**

* You now have multiple 4 × 4 matrices (one for each filter).
* Flattening means converting all these 4 × 4 matrices into a **single 1D vector**.

Example:

If you had 10 filters:

* Each filter → 4 × 4 = 16 values
* Total = 16 × 10 = **160 values**
* Final shape: **[160]**

**🔗 Step 7: Fully Connected Layer (Dense Layer)**

* This layer behaves like a traditional neural network.
* Each neuron is connected to **every value from the flattened layer**.
* Usually, there can be one or more dense (hidden) layers.

**🎯 Step 8: Output Layer**

* Since MNIST has **10 digits (0 to 9)**, the output layer has **10 neurons**.
* Each neuron gives a **probability score** for each digit.
* The **digit with the highest score is the predicted digit**.

**🔁 Step 9: Training**

* The model calculates the **loss** (how wrong the prediction is).
* It performs **backpropagation**:
  + Gradients are calculated using **chain rule**
  + Weights in **filters** and **dense layers** are updated using optimizers like **Adam**
  + This helps the model improve over time
* **✅ Summary**

| **Step** | **Action** | **Output** |
| --- | --- | --- |
| Input | 28×28×1 (grayscale digit image) | - |
| Conv1 + ReLU | 5×5 filter, no padding | 24×24×10 |
| Max Pooling | 2×2 | 12×12×10 |
| Conv2 + ReLU | 5×5 filter | 8×8×10 |
| Max Pooling | 2×2 | 4×4×10 |
| Flatten | All values into a vector | 160 |
| Dense (FC) | Fully connected layers | Varies |
| Output | 10 neurons (for 0–9 digits) | Predicted digit |
|  |  |  |

<https://developersbreach.com/convolution-neural-network-deep-learning/>

:

**🧠 High-Level Overview of CNN**

CNNs are mainly used for **image classification tasks**—like determining whether an image is a **horse, zebra, or dog**.

CNNs work in **two main phases**:

1. **Feature Extraction**
2. **Classification**

**🖼️ 1. Input: RGB Image**

* The input image is in **RGB format** (i.e., it has 3 color channels: Red, Green, and Blue).
* Example: A zebra image.

**🧩 2. Feature Extraction (Convolution + ReLU + Pooling)**

This part detects patterns like edges, textures, shapes, etc.

**⛓️ Layers in sequence:**

**🔁 Repeated block (3 times):**

* **Convolution Layer**: Applies filters (kernels) to extract patterns.
* **ReLU Activation**: Introduces non-linearity (replaces negative values with zero).
* **Pooling Layer**: Reduces spatial dimensions (height & width) using max/average pooling.

➡️ This sequence is repeated **three times**:

nginx

CopyEdit

Conv + ReLU → Pooling

Conv + ReLU → Pooling

Conv + ReLU → Pooling

These steps generate what's called **Feature Maps**—they highlight important patterns from the image.

**📏 3. Flattening**

* Converts the multi-dimensional feature maps into a **single 1D vector**.
* Purpose: Prepare the data for the **Fully Connected Neural Network**.

**🔗 4. Fully Connected Layers (Dense Layers)**

* These layers learn to combine features into predictions.
* If you're predicting whether an image is a **Horse**, **Zebra**, or **Dog**, the final layer will have **3 neurons** (one per class).
* A **Softmax Activation Function** is applied in the final layer to output **probabilities** for each class.

**🧮 5. Classification & Loss Calculation**

* The model's output (called **ŷ** or y-hat) is compared to the true label (**y**).
* A **loss function** (e.g., cross-entropy loss) is used to compute the error.

**🔁 6. Backpropagation**

* The model uses **backpropagation** to update its internal weights:
  + Derivatives (gradients) are calculated.
  + Filters and weights in all layers are updated using **gradient descent** to reduce the loss.
  + ReLU is used because it helps easily compute gradients (derivatives are either 1 or 0).