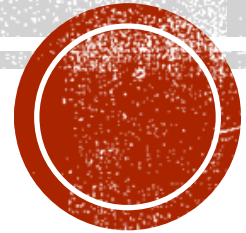
CONVOLUTIONAL NEURAL NETWORKS

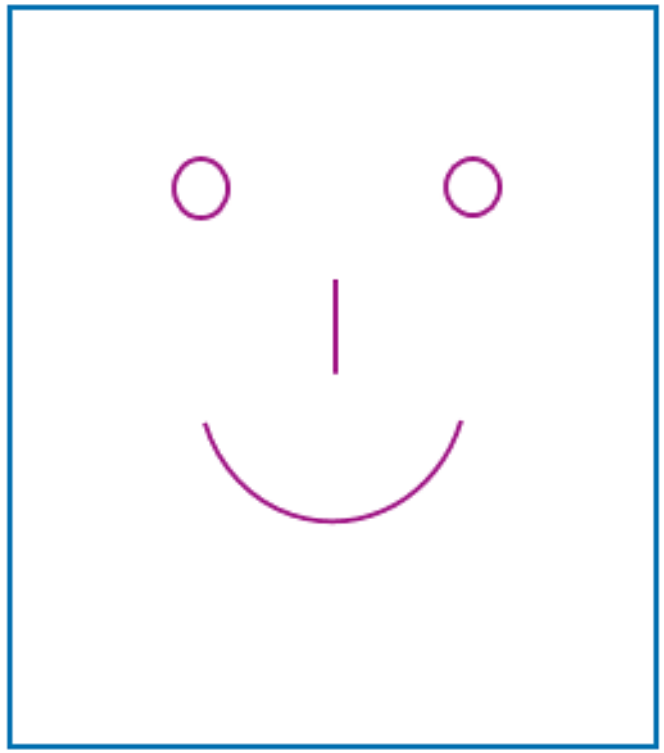


Dr Divya Meena Sundaram

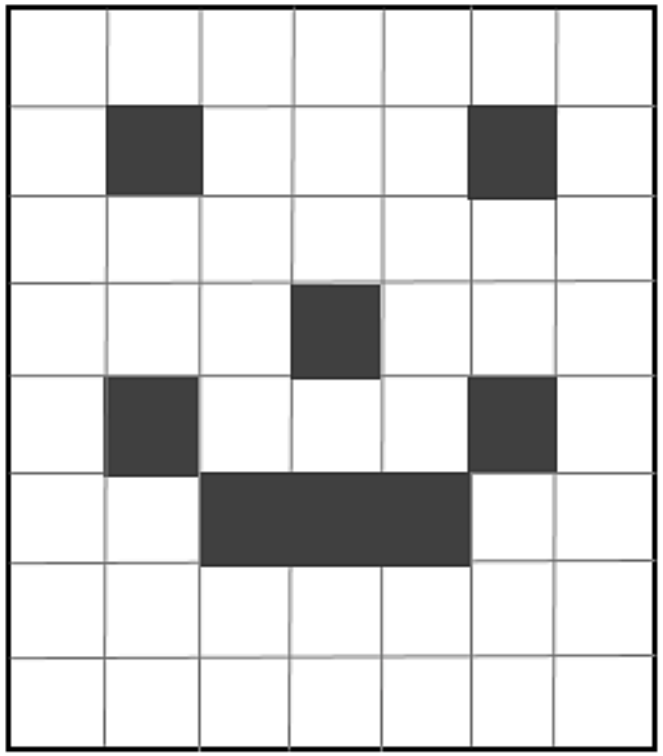
Sr. Assistant Prof. Grade 2

SCOPE

VIT-AP University



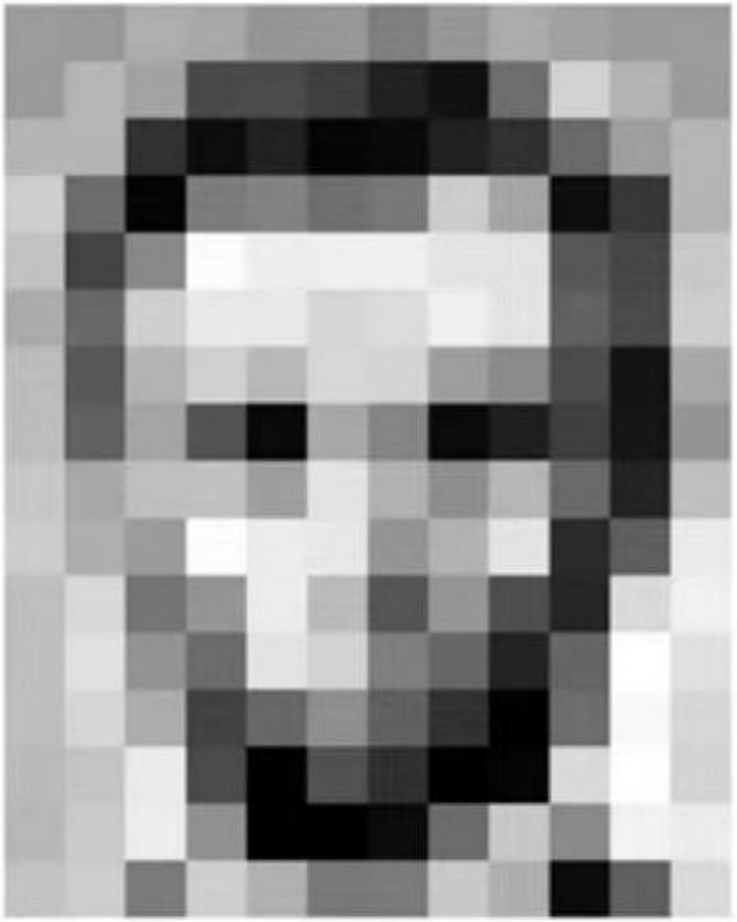
Real Image



Represented in the form of black and white pixels

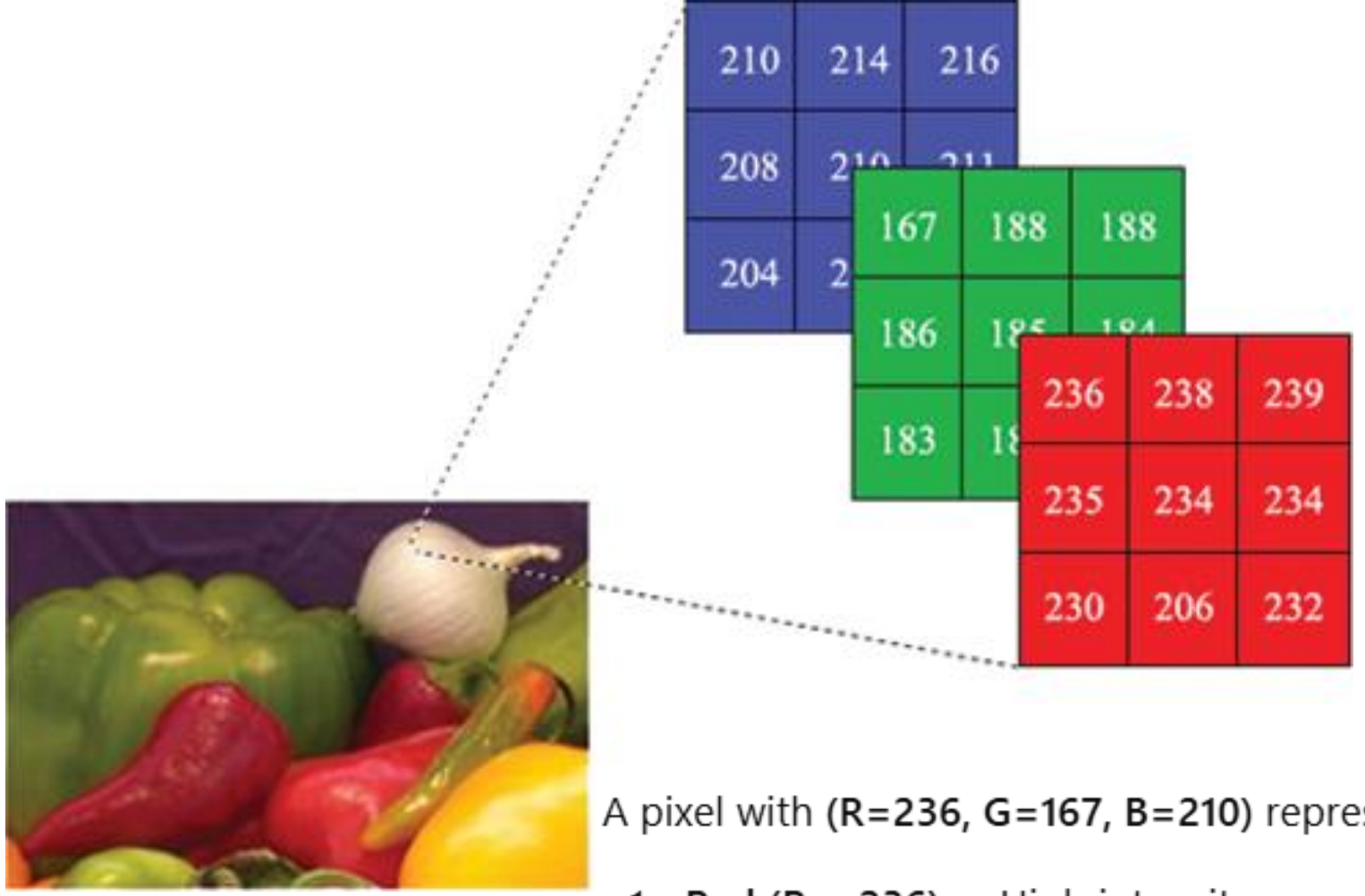
0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Image represented in the form of a matrix of numbers



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

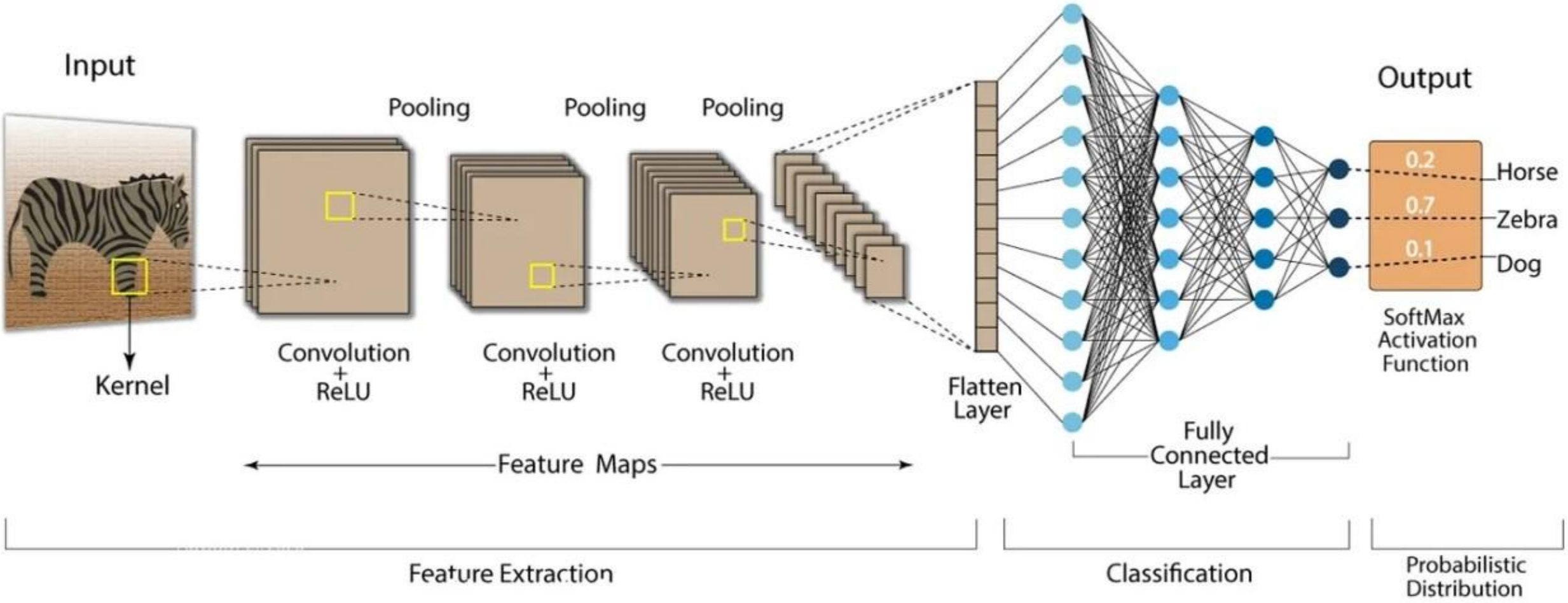
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218



A pixel with (**R=236, G=167, B=210**) represents a **light pink or lavender-like color** because:

1. **Red (R = 236)** → High intensity, meaning the pixel has a strong red component.
2. **Green (G = 167)** → Medium intensity, contributing some green to the color mix.
3. **Blue (B = 210)** → High intensity, making the pixel have a significant bluish tone.

Convolution Neural Network (CNN)



Layer	Purpose
Input Layer	Takes RGB image as (**Height, Width, 3**)
Conv Layer	Extracts **features** (edges, textures) from each channel
ReLU Activation	Introduces **non-linearity**
Pooling Layer	Reduces **dimensions**, keeps key features
Flatten Layer	Converts feature maps into a **1D vector**
Fully Connected	Final **classification layer**

1. Kernel (Filter)

A **kernel (filter)** is a small matrix (e.g., 3×3, 5×5) that slides over the input image to perform convolutions.

- It extracts features like **edges, textures, and patterns**.
- Each kernel learns a different feature representation.

🔹 **Example of a 3×3 Edge Detection Kernel:**

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

- This detects edges by emphasizing differences between adjacent pixels.

🔹 **Multiple Kernels per Layer**

- If we use **32 filters** in a Conv layer, the output has **32 feature maps**.
- More filters allow CNNs to learn **richer** and **more complex** features.

2. Stride (Step Size of Kernel Movement)

Stride controls how much the kernel shifts across the input image.

- **Stride = 1** (default) → Kernel moves **one pixel at a time** → **More detailed** features.
- **Stride = 2** → Kernel moves **two pixels at a time** → **Smaller output (more downsampling)**.

- **Effect of Stride:**
- **Lower stride (1)** → More detailed, larger output.
- **Higher stride (2 or more)** → Faster computation, but smaller output size.

- **Formula for Output Size with Stride S and Kernel Size K:**

$$O = \frac{(I - K)}{S} + 1$$

where O is output size, I is input size, K is kernel size, and S is stride.

3. Padding (Handling Image Borders)

Padding adds extra pixels around the input to control how much the kernel can slide over the image.

- **Types of Padding:**

1. **Valid Padding ("No Padding")**
    - The kernel only processes within the image.
    - Output size is **reduced** after convolution.
2. **Same Padding (Zero Padding)**
    - Adds zeros around the image to maintain the **same output size**.
    - Helps in preserving **spatial dimensions**.

- **Formula for Output Size with Padding P:**

$$O = \frac{(I - K + 2P)}{S} + 1$$

- P = padding size (e.g., 1 for **3×3 kernels** to maintain size).
- **Example: If input is 32×32, using a 3×3 kernel, stride 1, and padding 1, the output remains 32×32.**

4. Pooling (Downsampling the Image)

Pooling reduces spatial dimensions while keeping important features.

- **Types of Pooling:**

1. **Max Pooling:** Takes the **largest** value in a region.
2. **Average Pooling:** Takes the **average** of the values.

Example of **2×2 Max Pooling:**

$$\begin{bmatrix} 1 & 3 \\ 7 & 9 \end{bmatrix} \quad \Rightarrow \quad 9$$

- **Purpose:**

- **Reduces overfitting** by making CNNs focus on dominant features.
- **Speeds up computation** by reducing image size.

5. Fully Connected Layers (Dense Layers)

- Convert extracted features into a **1D vector**.
- The final layer uses **Softmax (for classification)** or **Sigmoid (for binary tasks)**.

- **Example: Classifying an image as Awake vs. Drowsy**

$$\text{Softmax:} \quad P_{\text{awake}} = \frac{e^{z_{\text{awake}}}}{e^{z_{\text{awake}}} + e^{z_{\text{drowsy}}}}$$

- The higher probability determines the **final prediction**.

6. Dropout (Preventing Overfitting)

- **Randomly drops neurons** during training to prevent **overfitting**.
- Ensures CNN **doesn't memorize** training data.

Given an input image of size $(H \times W)$, a filter of size $(f \times f)$, stride s, and padding p, the output size is:

$$O = \frac{(H - f + 2p)}{s} + 1$$

where:

- O = Output size
- H, W = Input height and width
- f = Filter size
- p = Padding
- s = Stride

Example:

- Input: 5×5, Kernel: 3×3, Stride: 1, Padding: 0
- Output size = $(5 - 3 + 2(0))/1 + 1 = 3 \times 3$

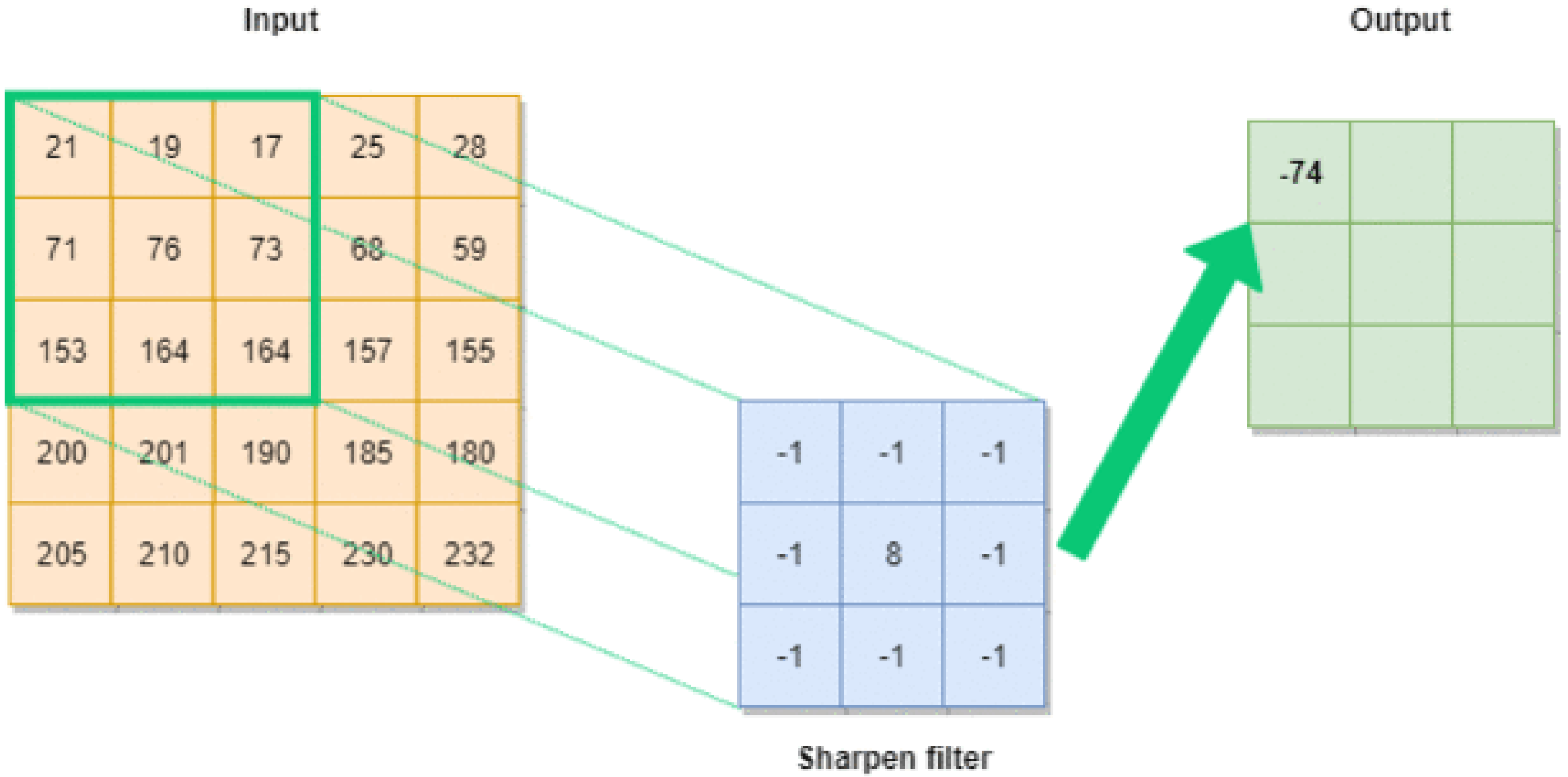
STRUCTURE OF A CONVOLUTIONAL LAYER

A **single convolutional layer** consists of:

- **Input Image** (e.g., 64×64×3 for RGB)
- **Filters (Kernels)** (e.g., 3×3, 5×5 matrices)
- **Stride** (movement of the filter)
- **Padding** (optional, keeps size same)
- **Activation Function** (e.g., ReLU)

HOW CONVOLUTION WORKS

1. A filter, also called a kernel, is applied to the input data
2. The filter is slid over the input image, one pixel at a time
3. At each location, a matrix multiplication is performed
4. The results are summed onto a feature map
5. The feature map indicates the locations and strengths of detected features



*

-1	0	1
-2	0	2
-1	0	1

Filter

SAMPLE PROBLEM ON CONVOLUTION OPERATION

1. Given Input Image (3×3)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

2. Given Filter (2×2 Kernel)

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

3. Stride = 1, No Padding

We'll slide the 2×2 filter across the image with a **stride of 1**.

Element-wise multiplication:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (1 \times 1) + (2 \times 0) + (4 \times 0) + (5 \times -1) = 1 + 0 + 0 - 5 = -4$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (2 \times 1) + (3 \times 0) + (5 \times 0) + (6 \times -1) = 2 + 0 + 0 - 6 = -4$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 6 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$(3 \times 1) + (0 \times 0) + (6 \times 0) + (0 \times -1) = 3 + 0 + 0 + 0 = 3$$

Fourth position (Middle-left)

$$\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}$$

$$(4 \times 1) + (5 \times 0) + (7 \times 0) + (8 \times -1) = 4 + 0 + 0 - 8 = -4$$

Fifth position (Middle-center)

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

$$(5 \times 1) + (6 \times 0) + (8 \times 0) + (9 \times -1) = 5 + 0 + 0 - 9 = -4$$

Sixth position (Middle-right)

$$\begin{bmatrix} 6 & 0 \\ 9 & 0 \end{bmatrix}$$

$$(6 \times 1) + (0 \times 0) + (9 \times 0) + (0 \times -1) = 6 + 0 + 0 + 0 = 6$$

Seventh position (Bottom-left)

$$\begin{bmatrix} 7 & 8 \\ 0 & 0 \end{bmatrix}$$

$$(7 \times 1) + (8 \times 0) + (0 \times 0) + (0 \times -1) = 7 + 0 + 0 + 0 = 7$$

Eighth position (Bottom-center)

$$\begin{bmatrix} 8 & 9 \\ 0 & 0 \end{bmatrix}$$

$$(8 \times 1) + (9 \times 0) + (0 \times 0) + (0 \times -1) = 8 + 0 + 0 + 0 = 8$$

Ninth position (Bottom-right)

$$\begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix}$$

$$(9 \times 1) + (0 \times 0) + (0 \times 0) + (0 \times -1) = 9 + 0 + 0 + 0 = 9$$

Given Input Image (3×3)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Given Filter (2×2 Kernel)

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Output Feature Map (2×2 Convolution Output)

$$\begin{bmatrix} -4 & -4 & 3 \\ -4 & -4 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

HOW MANY FILTERS CAN BE USED IN A CONVOLUTIONAL LAYER?

The number of filters (kernels) in a convolutional layer is a hyper parameter, meaning you can choose it based on the complexity of the problem and computational resources.

- Small models: 16, 32, 64 filters
- Large models: 128, 256, 512 filters
- More filters → More feature extraction (edges, textures, shapes)
- Too many filters → More computation, risk of overfitting, possible overfitting

General Rule:

- Start with fewer filters (e.g., 32 or 64) in the first layers.
- Increase filters in deeper layers (e.g., 128, 256, 512).

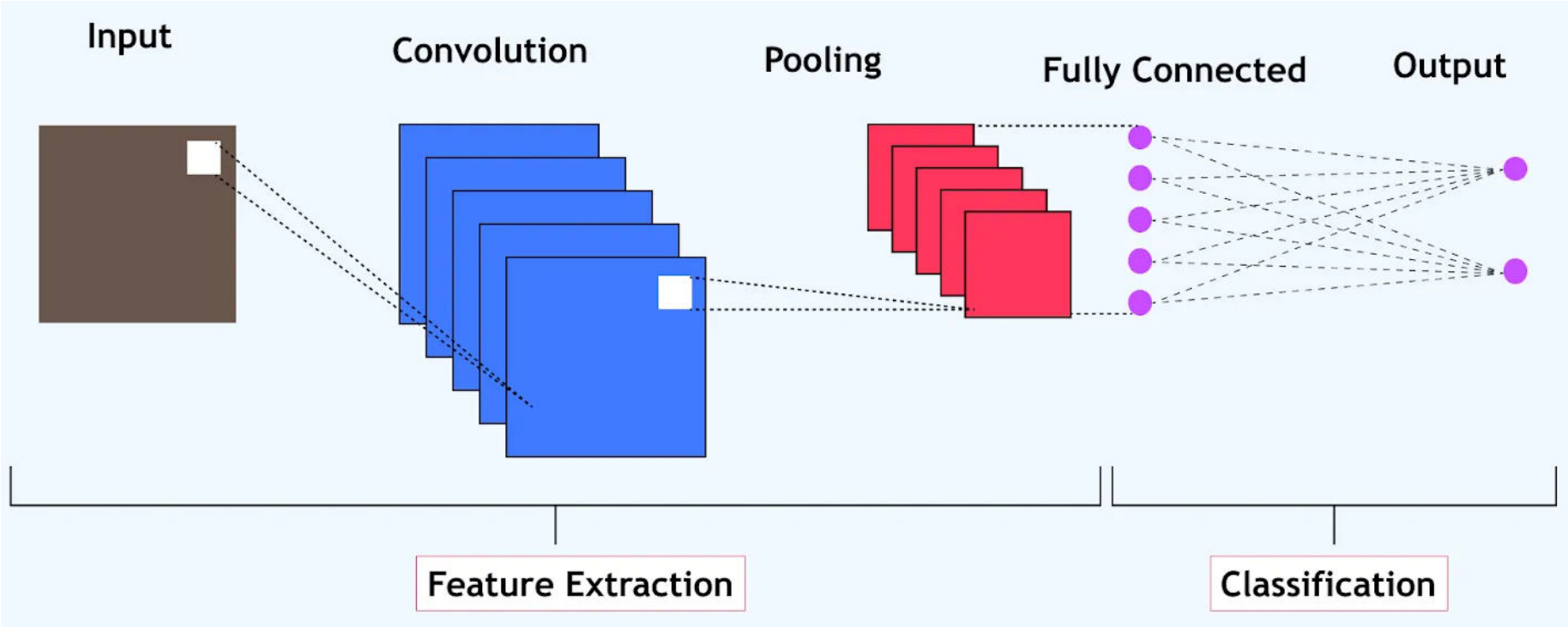
WHAT COMES AFTER ONE CONVOLUTIONAL LAYER?

- More convolutional layers → Extract deeper features
- Pooling layer → Reduce dimensions
- Fully connected layer → Make final predictions

CONVOLUTIONAL NEURAL NETWORK

CONVOLUTIONAL NEURAL NETWORK

- A Convolutional Neural Network (CNN) is a **type of Deep Learning neural** network architecture and an **extended version of artificial neural networks** (ANN) which is predominantly used to extract the feature from the grid-like matrix dataset.
- The convolutional neural networks (CNN) **introduced by LeCun(1998).**
- CNN **act directly on matrices, or even on tensors** for images with three RGB color channels.
- CNN are now widely used for image classification, image segmentation, object recognition, face recognition, etc.,



- The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction.
- The network learns the optimal filters through backpropagation and gradient descent.

WHY CNN AND WHY NOT ANN ?

(a) Spatial Structure is Lost in ANN

- **ANNs flatten images** into 1D vectors, losing information about **edges, textures, and spatial hierarchies.**
- **CNNs preserve spatial structure** using convolutional filters.

(b) Too Many Parameters in ANN

- A 256×256 RGB image = **196,608 input neurons** (3 channels).
- Fully connected layers require **millions of parameters** → causes **overfitting** & **huge memory consumption.**
- **CNNs reduce parameters** using convolution & pooling.

(c) No Feature Extraction in ANN

- ANN learns from **raw pixel values**, requiring more data.
- CNN **automatically extracts features** like edges, corners, textures.

(d) Computation Inefficiency

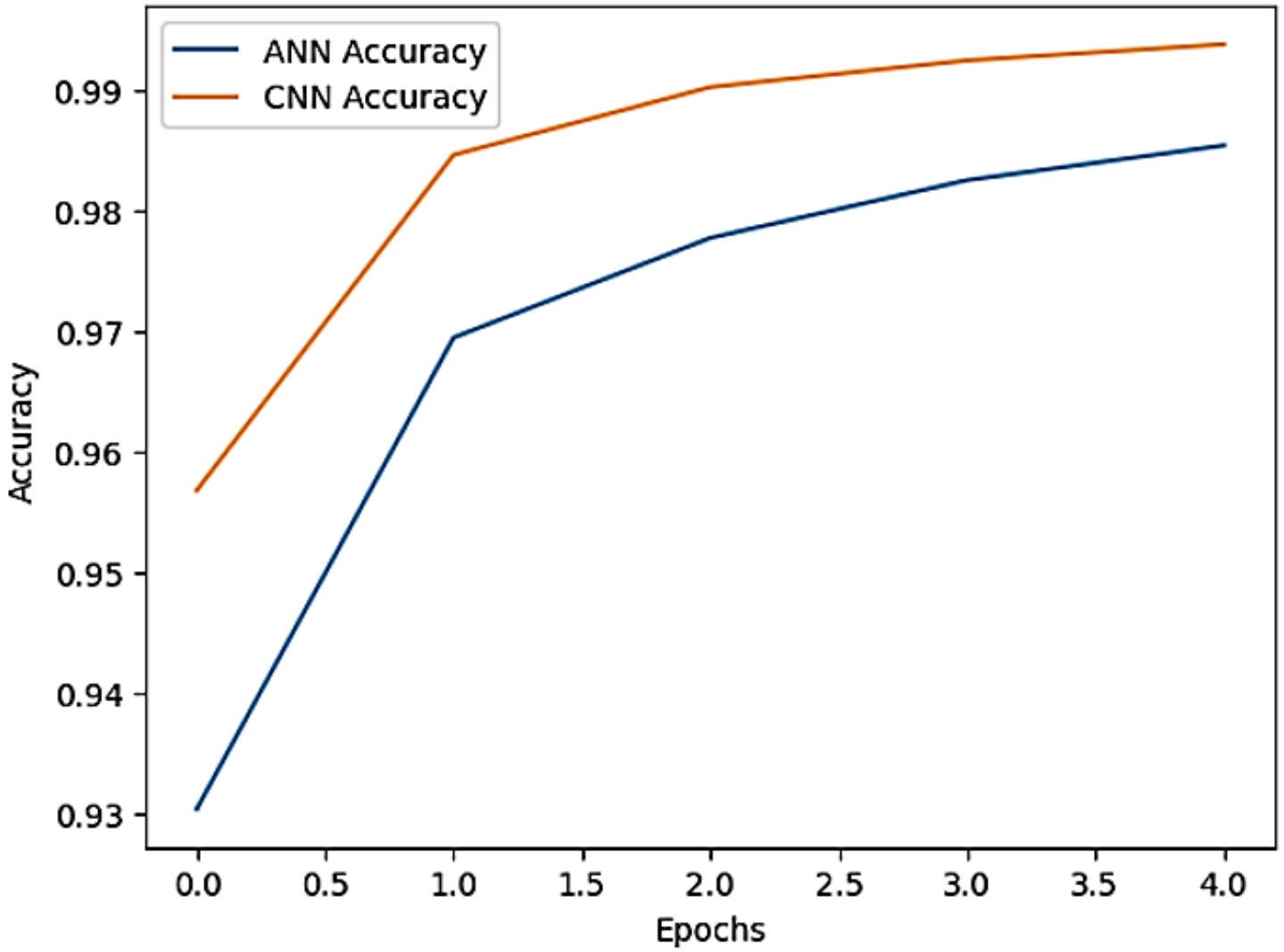
- **ANN:** Fully connected layers = **quadratic growth** in parameters.
- **CNN:** Uses **shared weights** (kernels), making it **faster & scalable**.

Conclusion: CNNs are the Best for Image Recognition

If spatial structure matters (which it does in images), use CNNs.

ANNs are **too inefficient for large images.**

For real-time detection (like YOLO), ANN is impractical.




```
Total parameters in ANN: 109386
Total parameters in CNN: 121930
```

1. CONVOLUTION LAYER

CONVOLUTION LAYER

- The **first layer** of a Convolutional Neural Network is always a Convolutional Layer.
- This is the first step in the process of **extracting valuable features** from an image.
- A convolution **converts all the pixels in its receptive field into a single value.**
- Convolutional filters (kernels) are **small matrices that slide over an image** to extract specific features.
- **Different filters** are used to **detect edges, textures, shapes, and patterns** at various levels of a CNN.

Filter Type	Purpose	Kernel Matrix (3×3)
Edge Detection (Sobel X)	Detects vertical edges	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
Edge Detection (Sobel Y)	Detects horizontal edges	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Sharpening	Enhances fine details	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
Blurring (Smoothing)	Reduces noise, softens image	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
Embossing	Highlights edges with a 3D effect	$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
Gaussian Blur	Smooths image to remove fine details	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$
Laplacian Filter	Detects edges in all directions	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

HOW CNN USE FILTERS AT DIFFERENT LAYERS

1. First Layers: Detect Basic Features (Edges & Corners)

✅ Filters detect **low-level features** like **edges, lines, and corners**.

✅ Edge-detection filters (like **Sobel, Prewitt, or Laplacian**) are commonly learned.

✅ These filters help identify **sharp intensity changes** in the image.

Example:

- A **3×3 filter** detecting vertical edges may activate strongly on object boundaries.

2. Middle Layers: Identify Shapes & Textures

✅ Deeper layers **combine detected edges** to recognize **shapes, textures, and patterns**.

✅ Filters detect **curves, blobs, and repeating patterns**.

✅ Helps in identifying **facial features, fur textures, or specific geometric patterns**.

Example:

- A **dog's fur texture** or **the rounded shape of a human eye** is identified.

3. Deeper Layers: Recognize High-Level Features (Objects & Faces)

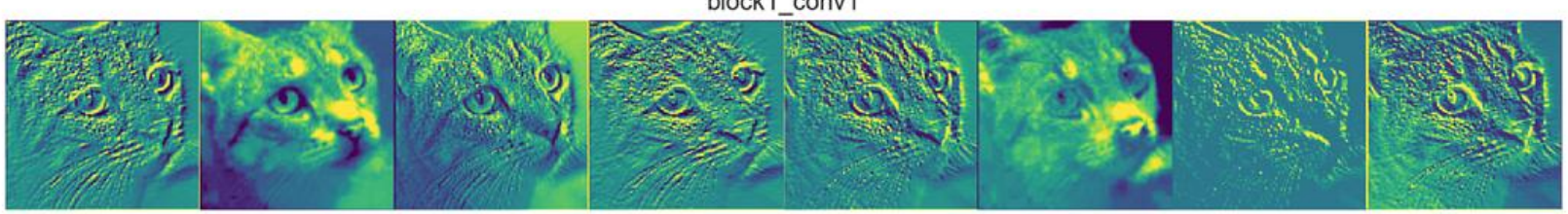
✅ High-level layers extract **complex structures** like **faces, digits, and full objects**.

✅ Filters detect **combinations of shapes** to form object representations.

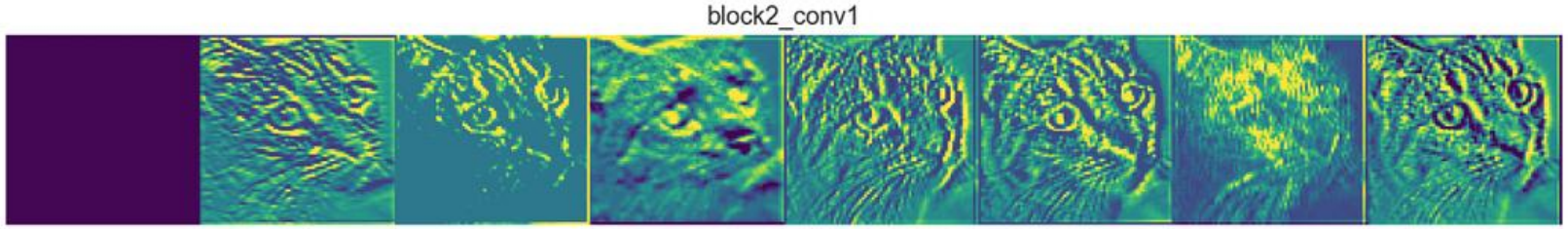
✅ This is where CNNs learn **what a cat, car, or face looks like**.

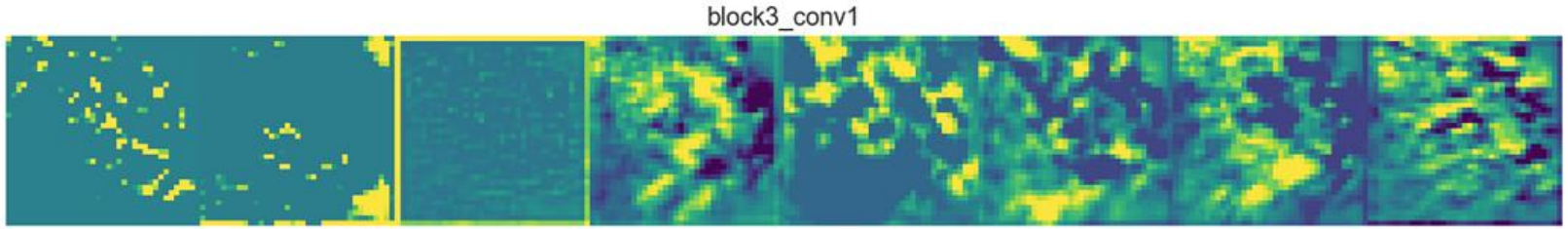
Example:

- A CNN **recognizing a dog's face** by combining fur texture, eyes, ears, and nose features.

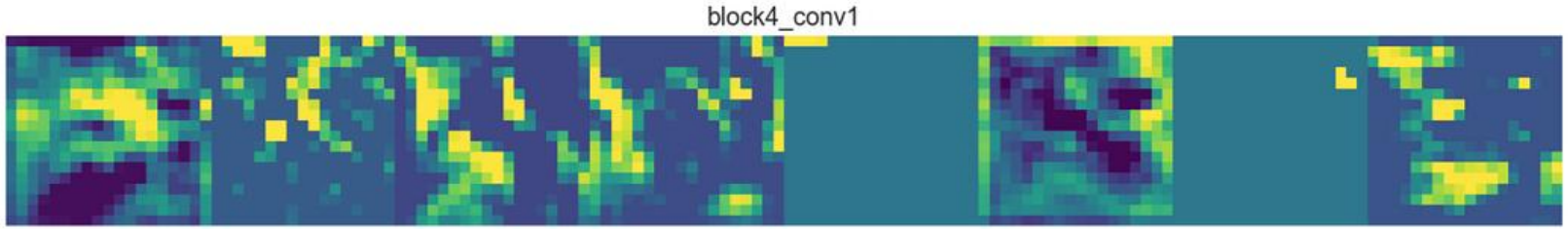


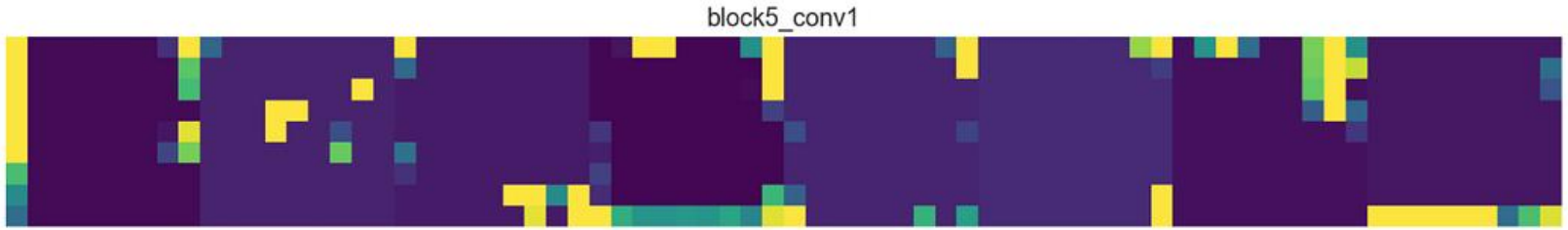
- **Early layers keep image-like structure** because they focus on edges.
- Detect **edges, lines, and simple textures**.





- In middle layer, feature maps **start losing clear structure**—they no longer resemble the original image.
- Detect **textures, shapes, and more abstract patterns**.

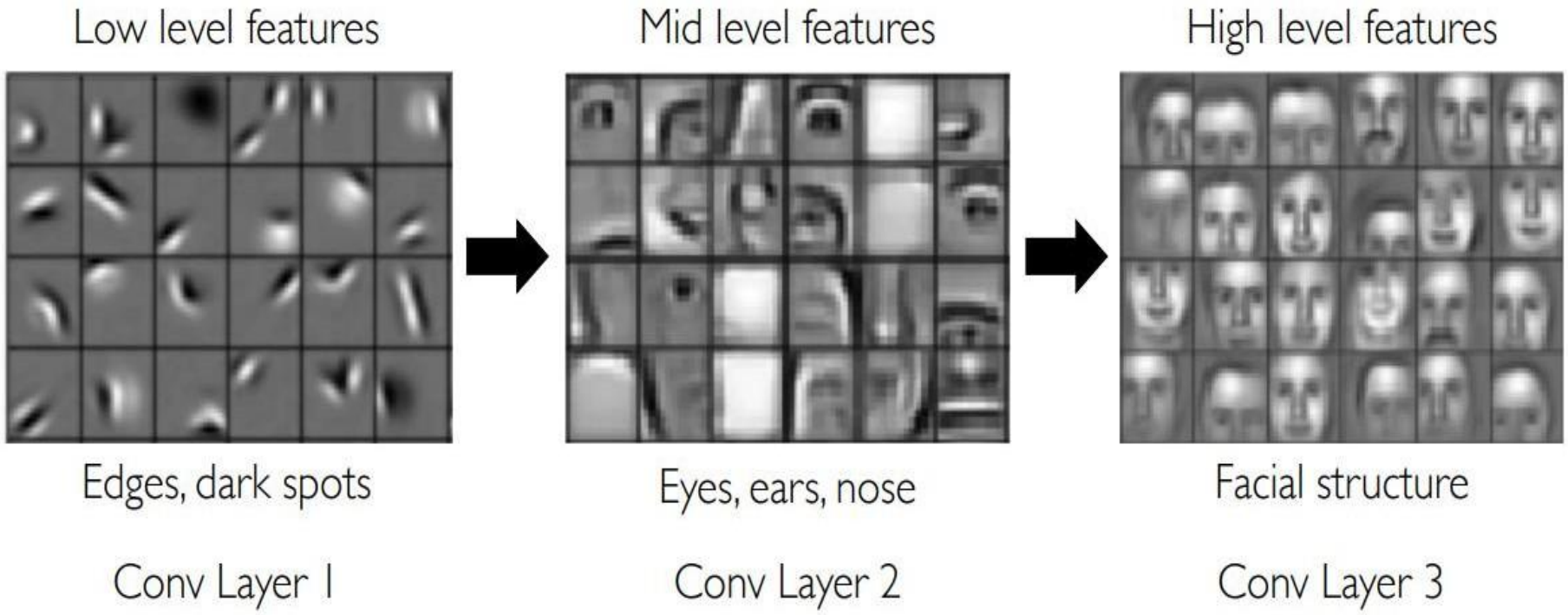




- In the deeper layers, feature maps look **highly abstract.**
- These layers ignore **background details** and focus only on **what makes the object recognizable**.

REPRESENTATION LEARNING

- Representation learning is a technique where a **model automatically learns the best way to represent data** rather than relying on handcrafted features.
- In deep convolutional neural networks (CNNs), representation learning enables models to **capture hierarchical feature representations, from low-level edges to high-level objects.**



HOW CNNS LEARN REPRESENTATIONS

- CNNs progressively learn better and more abstract representations of data as they move deeper in the network.

✓ **Automatically learns features** from data (no manual feature engineering).

✓ **Learns hierarchical abstractions** (edges → textures → objects).

1 Low-Level Features (Shallow Layers)

- The first few layers detect edges, corners, and textures.

2 Mid-Level Features (Intermediate Layers)

- Deeper layers begin to recognize **object parts** (e.g., eyes, nose, wheels, fur).

3 High-Level Features (Deep Layers)

- The final layers detect **entire objects** (faces, cars, animals).

2. POOLING LAYER

POOLING LAYER

- A Pooling Layer is used to reduce the size of feature maps while preserving important information.
- The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. It helps in:
  - Reducing computation
  - Making the network more robust to small shifts & distortions
  - Preventing overfitting
- **Types:** Max Pooling, Min Pooling and Average Pooling, Global Average Pooling, Global Max Pooling

1. Max Pooling (Most Common)

- Takes the maximum value in each window.
- Preserves the most important features.
- Helps in edge detection

Example (2×2 Max Pooling, Stride=2):

Input (4×4 Matrix)

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 9 & 8 & 6 & 5 \end{bmatrix}$$

Pooling Window (2×2 regions)

$$\begin{bmatrix} \max(1,3,5,6) & \max(2,4,7,8) \\ \max(3,2,9,8) & \max(1,0,6,5) \end{bmatrix}$$

Output (2×2 Matrix)

$$\begin{bmatrix} 6 & 8 \\ 9 & 6 \end{bmatrix}$$

2. Min Pooling

- Useful in tasks where detecting **low-intensity patterns** is important.
- Rarely used in deep learning but can be applied in medical imaging and anomaly detection.

Example (2×2 Min Pooling, Stride = 2):

Given a **4×4 feature map**:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Apply 2×2 Min Pooling (Stride=2):

$$\begin{bmatrix} \min(1,2,5,6) & \min(3,4,7,8) \\ \min(9,10,13,14) & \min(11,12,15,16) \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 \\ 9 & 11 \end{bmatrix}$$

3. Average Pooling

- Takes the average value of each window.
- Helps in blurring & smoothing features.

Example (2×2 Average Pooling, Stride=2):

$$\begin{bmatrix} (1+3+5+6)/4 & (2+4+7+8)/4 \\ (3+2+9+8)/4 & (1+0+6+5)/4 \end{bmatrix}$$

Output

$$\begin{bmatrix} 3.75 & 5.25 \\ 5.5 & 3.0 \end{bmatrix}$$

4. Global Pooling (Average and Max)

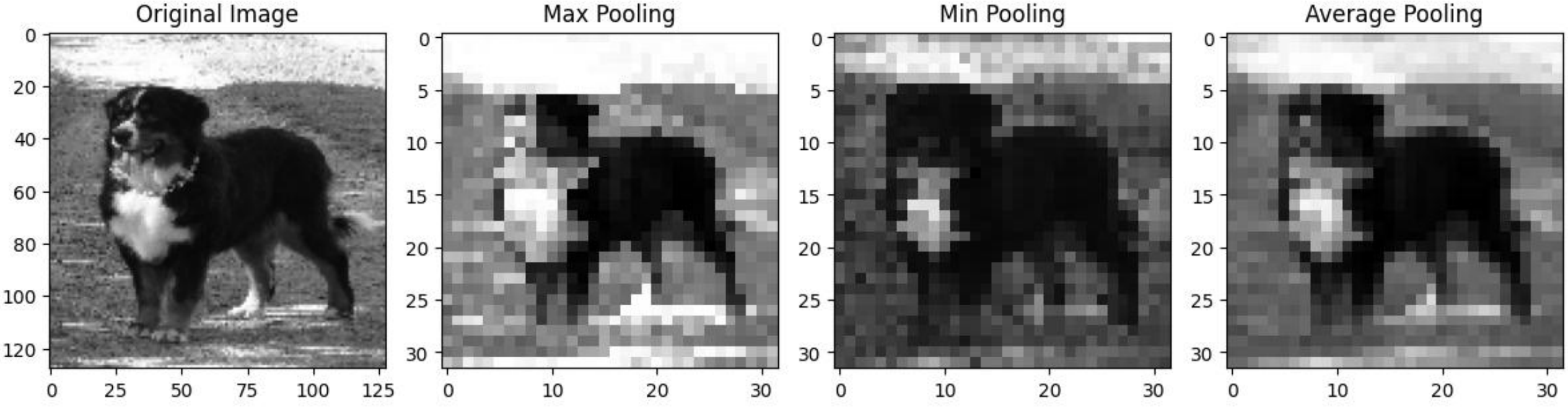
- Reduces an entire feature map to a single value.
- Often used before fully connected layers.
- Helps in reducing parameters.

$$GAP = \frac{(1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16)}{16} = \frac{136}{16} = 8.5$$

$$GMP = \max(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16) = 16$$

Type of Pooling	What it Does	Pros
Max Pooling	Selects the **maximum** value in the pooling window.	✅ Captures the most important/high-activation features. ✅ Enhances edge detection and textures. ✅ Reduces dimensions efficiently.
Min Pooling	Selects the **minimum** value in the pooling window.	✅ Helps in noise reduction by ignoring high activations. ✅ Useful for detecting low-intensity patterns.
Average Pooling	Computes the **mean** of all values in the pooling window.	✅ Retains more overall feature information. ✅ Smooths the feature map, reducing noise.
Global Max Pooling	Takes the **maximum** value from the entire feature map.	✅ Reduces dimensionality significantly. ✅ Helps in detecting the most dominant feature in an image.
Global Average Pooling	Computes the **mean** of the entire feature map.	✅ Enforces better feature generalization. ✅ Used in architectures like ResNet for reducing overfitting.

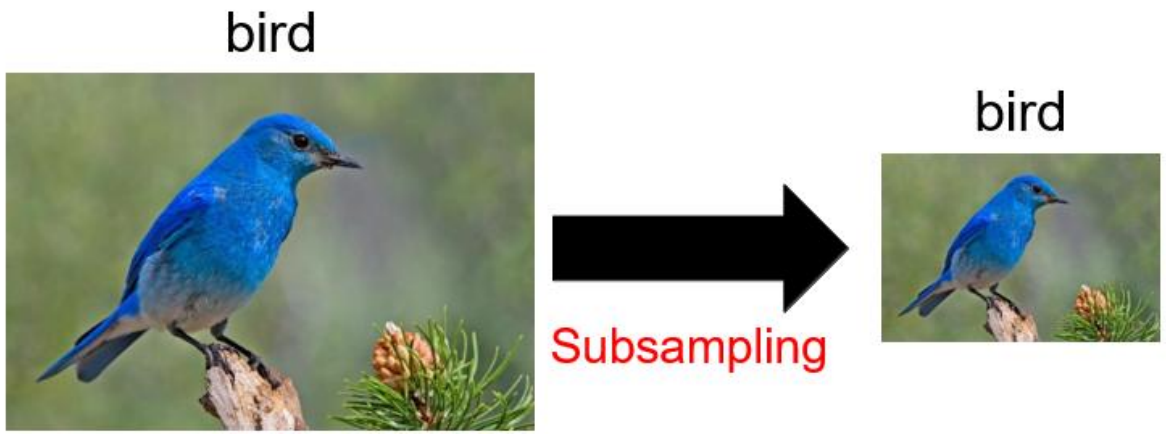
- **Max Pooling** → Best for **detecting strong edges & textures.**
- **Min Pooling** → Helps **remove noise but loses strong details.**
- **Average Pooling** → Smooths features, retains more detail but **blurs edges.**
- **Global Max Pooling** → Keeps **only the strongest feature**, losing spatial info.
- **Global Average Pooling** → Provides **better generalization** but might **blur details.**



WHY POOLING AND WHY NOT SUBSAMPLING?

- Subsampling pixels (simply resizing an image) also makes the image smaller and reduces parameters. However, pooling is more than just subsampling!
- **Retains important features**: Instead of just dropping pixels (like resizing), pooling preserves important information by selecting max/average values within local regions.
- **Increases Robustness (Translation Invariance):** Pooling helps reduce sensitivity to small translations, rotations, or distortions in the image.
- **Reduces Overfitting**: Pooling removes unnecessary fine details, forcing the model to focus on important patterns rather than noise.

Aspect	Subsampling (Resizing)	Pooling (Max/Average)
How it works	Directly drops pixels to shrink the image	Aggregates values (max, avg) to keep important info
Feature Preservation	Some key details may be lost	Keeps important features (edges, textures)
Translation Invariance	No	Yes
Reduces Overfitting?	No	Yes
Used in CNNs?	No	Yes



3. FULLY CONNECTED LAYER

FULLY CONNECTED LAYER

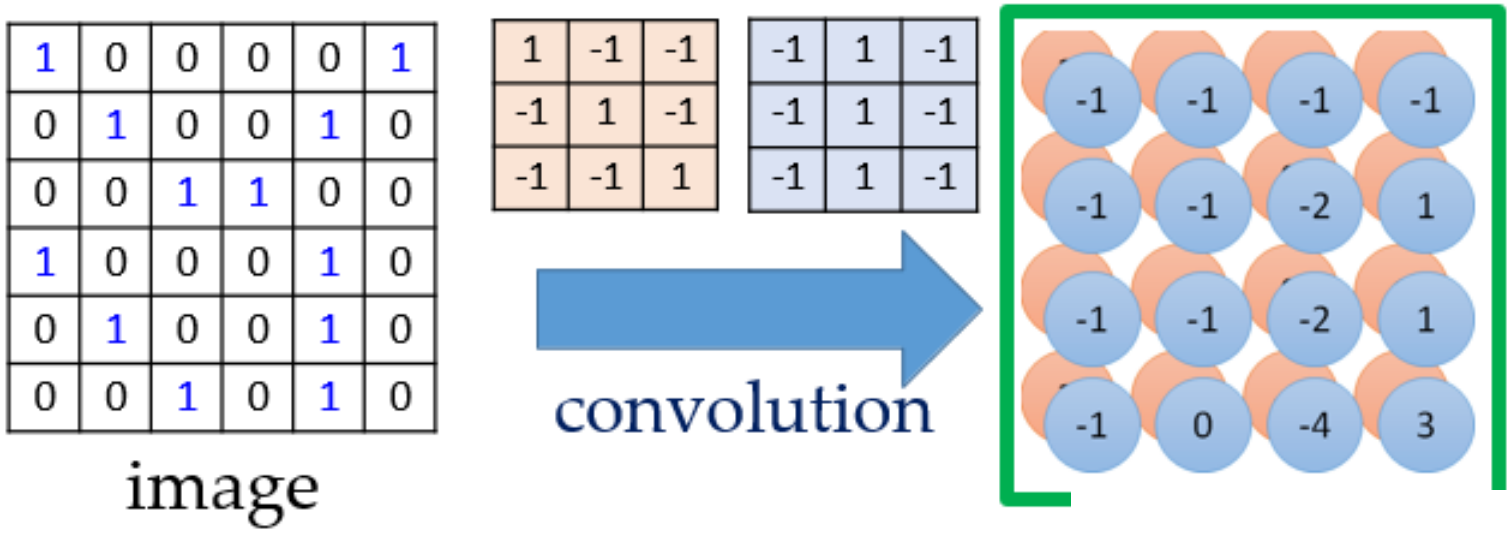
- A fully connected (FC) layer is the final stage of a Convolutional Neural Network (CNN) before making a prediction. It connects every neuron from the previous layer to every neuron in the next layer, just like in an Artificial Neural Network (ANN).
- The FC layer helps to map the representation between the input and the output.
  - ➢ Flattens Features; Learns Complex Patterns; Performs Classification

If the **flattened feature vector** has **n values**, and the **FC layer has m neurons**, the number of parameters is:

$$\text{Parameters} = n \times m + m \quad \text{(including biases)}$$

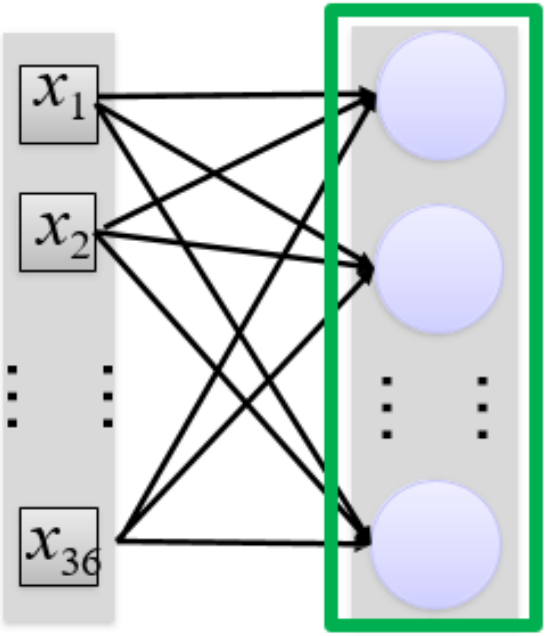
This makes **FC layers computationally expensive**, which is why CNNs often **reduce FC layers** in favor of **Global Average Pooling** (GAP).

CONVOLUTION V.S. FULLY CONNECTED



Fully-connected

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0



Feature	Convolutional Layer	Fully Connected Layer
Connections	**Local connections** (only within a small region)	**Global connections** (each neuron connects to all)
Parameter Efficiency	**Fewer parameters**	**Many parameters** (high computational cost)
Interpretability	Extracts meaningful spatial patterns	Makes final decisions based on learned features
Best for	Feature extraction	Classification

🔹 Why CNNs Reduce Fully Connected Layers?

- **Too Many Parameters** → High computational cost
- **Overfitting Risk** → More parameters = Higher chance of memorizing training data

Convolution

Max Pooling

Convolution

Max Pooling

Can repeat this many times

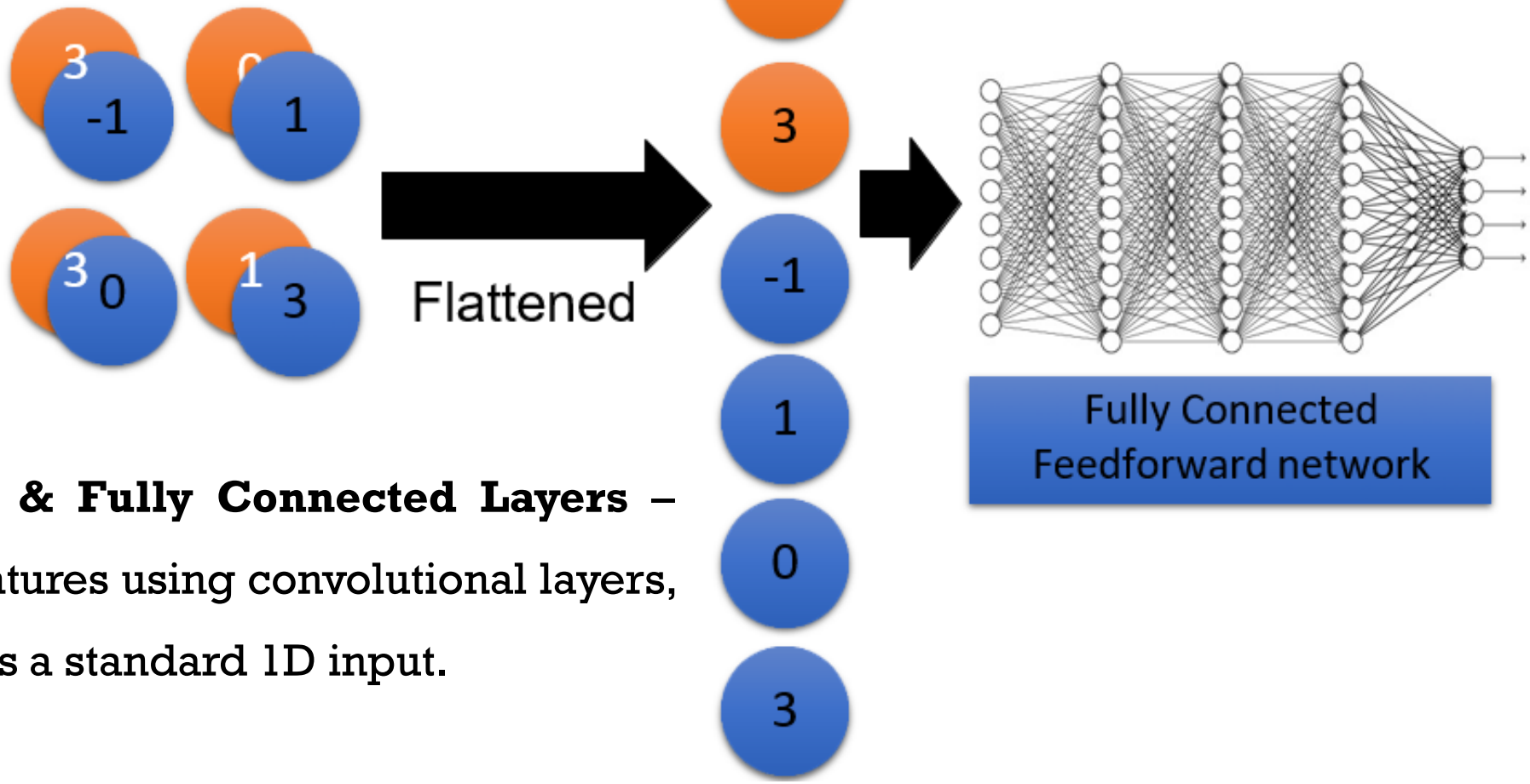
Flattened

Fully Connected Feedforward network

cat dog ……

4. FLATTEN LAYER

- The **Flattening layer** is used to convert a multi-dimensional feature map (output of convolutional or pooling layers) into a **1D vector**, which can then be fed into a **fully connected (dense) layer** for classification.



- **Bridges Convolutional & Fully Connected Layers** – CNNs process spatial features using convolutional layers, but classification requires a standard 1D input.

```python
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same", activation='relu', input_shape=(64, 64, 3)),
    layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),
    layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same", activation='relu'),
    layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),
    layers.Flatten(),
    layers.Dense(units=128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(units=2, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, 64, 32)	896
max_pooling2d (MaxPooling2D)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 64)	0
flatten (Flatten)	(None, 16384)	0
dense (Dense)	(None, 128)	2,097,280
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 2)	258

Total params: 2,116,930 (8.08 MB)

Trainable params: 2,116,930 (8.08 MB)

Non-trainable params: 0 (0.00 B)

CALCULATING TOTAL NO: OF PARAMETERS

```
5    layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same", activation='relu', input_shape=(64, 64, 3)),
```

Step 1: First Conv2D Layer

- **Input Shape**: (64, 64, 3)
- **Filter Size**: (3×3)
- **Strides**: 1 (moves 1 pixel at a time)
- **Padding**: `"same"` (keeps output size same as input)
- **Number of Filters**: 32

$$\text{Output Size} = \frac{\text{Input Size} - \text{Filter Size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

Since **padding="same"**, output size remains **(64, 64, 32)**.

Parameter Calculation

Each filter has:

$$\text{Weights} = \text{Filter Size} \times \text{Filter Size} \times \text{Input Channels} + \text{Bias}$$

$$= (3 \times 3 \times 3) + 1 = 28$$

Since there are **32 filters**, total parameters:

$$32 \times 28 = 896$$

```
6     layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),
```

Step 2: First MaxPooling2D Layer

- **Pool Size**: (2×2)
- **Strides**: 2
- **Padding**: `"valid"` (no padding)

$$\text{Output Size} = \frac{\text{Input Size} - \text{Pool Size}}{\text{Stride}} + 1$$

$$\frac{64 - 2}{2} + 1 = 32$$

New Output Shape: (32, 32, 32)

Parameters: 0 (Pooling has no trainable weights)

```
7    layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same", activation='relu'),
```

Step 3: Second Conv2D Layer

- **Input Shape**: (32, 32, 32)
- **Filter Size**: (3×3)
- **Filters**: 64
- **Padding**: `"same"` (keeps size same)

New Output Shape: (32, 32, 64) (same as input due to "same" padding)

Parameter Calculation

Each filter has:

$$\text{Weights} = (3 \times 3 \times 32) + 1 = 289$$

Since there are **64 filters**, total parameters:

$$64 \times 289 = 18,496$$

```
8     layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),
```

Step 4: Second MaxPooling2D Layer

- Pool Size: (2×2)
- Strides: 2

$$\frac{32-2}{2}+1=16$$

New Output Shape: (16, 16, 64)

Parameters: 0

```
9     layers.Flatten(),
```

Step 5: Flatten Layer

- Input Shape: (16, 16, 64)
- Output Shape: (16 × 16 × 64) = (16384)
- Parameters: 0 (just reshaping)

```
10      layers.Dense(units=128, activation='relu'),
```

Step 6: Fully Connected Dense Layer

- Input Shape: (16384)
- Neurons: 128

New Output Shape: (128)

$$\text{Weights} = \text{Input Features} \times \text{Neurons}$$

$$= 16384 \times 128 = 2,097,152$$

Each neuron has a bias term, so the number of **bias parameters** is:

$$\text{Bias} = \text{Neurons} = 128$$

$$\text{Total Parameters} = \text{Weights} + \text{Bias}$$

$$= 2,097,152 + 128 = 2,097,280$$

```
11      layers.Dropout(0.5),
```

Step 7: Dropout Layer

- No change in shape
- Output Shape: (128)
- Parameters: 0

```
12      layers.Dense(units=2, activation='softmax')
```

Step 8: Output Dense Layer

- Input Shape: (128)
- Output Neurons: 2

$$\text{Weights} = 128 \times 2 = 256$$

$$\text{Bias} = 2$$

$$\text{Total Parameters} = 258$$

Final Output Shape: (2) (Softmax gives two probability scores)

1. **Conv2D layers keep the spatial dimensions the same** (if `padding="same"`) or reduce them (if `padding="valid"`).
2. **Pooling layers reduce the spatial dimensions** by **half**.
3. **Flatten converts feature maps into a 1D vector.**
4. **Dense layers fully connect neurons** for classification.
5. **Softmax outputs class probabilities.**

Layer	Type	Output Shape	Parameters
Conv2D	32 filters, 3×3 kernel	(64, 64, 32)	896
MaxPooling2D	2×2 pooling	(32, 32, 32)	0
Conv2D	64 filters, 3×3 kernel	(32, 32, 64)	18,496
MaxPooling2D	2×2 pooling	(16, 16, 64)	0
Flatten	Convert to 1D vector	(16384)	0
Dense (FC)	128 neurons	(128)	2,097,280
Dropout	50% dropout	(128)	0
Dense (Output)	2 neurons (Softmax)	(2)	258

Total Trainable Parameters = 2,116,930

```
Total params: 2,116,930 (8.08 MB)
Trainable params: 2,116,930 (8.08 MB)
Non-trainable params: 0 (0.00 B)
```

```python
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same", activation=None, input_shape=(64, 64, 3)),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),

    layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same", activation=None),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.MaxPooling2D(pool_size=(2,2), strides=2, padding="valid"),

    layers.Flatten(),
    layers.Dense(units=128, activation=None),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.5),

    layers.Dense(units=2, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

1. First Conv2D Layer

- Input shape: (64, 64, 3)
- Filters: 32
- Kernel size: (3,3)
- Weights: $(3 \times 3 \times 3) \times 32 = 864$
- Biases: 32
- Total: $864 + 32 = 896$

2. Batch Normalization (First)

- BN has **4 parameters per filter** (scale, shift, mean, variance).
- Total BN params: $4 \times 32 = 128$

3. Second Conv2D Layer

- Input shape: (32, 32, 32)
- Filters: 64
- Kernel size: (3,3)
- Weights: $(3 \times 3 \times 32) \times 64 = 18,432$
- Biases: 64
- Total: $18,432 + 64 = 18,496$

4. Batch Normalization (Second)

- Total BN params: $4 \times 64 = 256$

5. Dense Layer (128 Neurons)

- **Input size**: Flattened output size from conv layers:
  - Input to dense: $15 \times 15 \times 64 = 14,400$
- **Weights**: $14,400 \times 128 = 1,843,200$
- **Biases**: 128
- **Total**: $1,843,200 + 128 = 1,843,328$

6. Batch Normalization (Third)

- **Total BN params**: $4 \times 128 = 512$

7. Final Dense Layer (Output Layer)

- **Weights**: $128 \times 2 = 256$
- **Biases**: 2
- **Total**: $256 + 2 = 258$

Final Parameter Count

Layer	Weights	Bias	BN Params	Total Params
Conv2D (32 filters)	864	32	128	**1,024**
Conv2D (64 filters)	18,432	64	256	**18,752**
Dense (128 units)	1,843,200	128	512	**1,843,840**
Dense (2 units)	256	2	0	**258**
Total	**1,862,752**	**226**	**896**	**1,863,874**

Thus, the final model has **1,863,874 parameters**.