AI & Machine Learning for Genomic Data Science

Master in Genomic Data Science – Università di Pavia

Barbara Tarantino

AA 2025-2026

Course Outline

- Day 1 → Python Foundations & AI in Medicine
- Day 2 → Core Machine Learning for Genomics
- Day 3 → Deep Learning Foundations (PyTorch)
- Day 4 → Computer Vision for Medicine
- Day 5 → Large Language Models & Clinical Text

Objective: to acquire theoretical and practical bases to apply AI to genomics, clinical images, medical text.

Program - Day 4

Computer Vision for Medicine

- Types of medical images and their representation (X-RAY, CT, MRI, histology)
- Fundamentals of CNNs (convolution, filters, pooling, feature maps)
- Python Ecosystem for Computer Vision

Practical organization – Day 4

Morning (9:30 – 12:30)

- Introduction to Computer Vision in Medicine
  - Types of medical images (radiology, histopathology, microscopy)
  - How to represent a digital image (pixels, channels, tensors)
- Key concepts of CNNs
  - Convolution, filters, pooling, feature maps

Afternoon (14:30 – 17:00)

- Python Ecosystem for Computer Vision
- Notebook: CNN "from scratch" for image classification

Recap + Q&A

Section 1

Introduction to Computer Vision in Medicine

Why Computer Vision in Medicine?

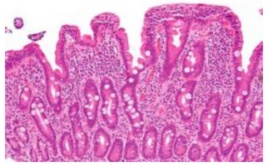
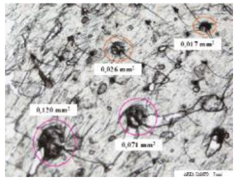
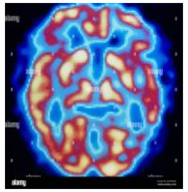
- Medical images are among the richest sources of information in the clinical field.

- Typical tasks:
  - **Diagnosis** (e.g. detecting pulmonary nodules in an X-ray).
  - **Prognosis** (i.e. predicting progression from histological images).
  - **Screening** (e.g. population mammograms).

- Challenge: Large amounts of data → impossible to analyze all manually.

- **Computer Vision (CV)** allows you to build algorithms that "see" and recognize visual patterns → support to the doctor, not a replacement.

Imaging Modalities in Medicine

Modality	Example	Image Type
Radiology 	X-ray, CT scan, MRI	2D or 3D grayscale
Istopathology 	Digitized biopsies	Gigapixels, cellular details
Microscope 	Subcellular cells/structures	Ultra-high resolution images
Other 	Ultrasound, PET	Dynamic or functional images

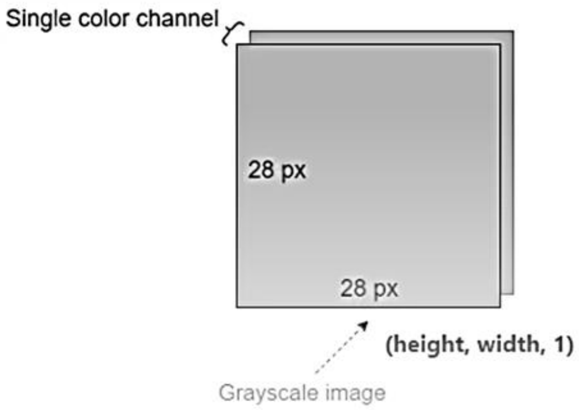
All of these modes produce **arrays of numbers (pixels)** that the AI can work with.

What is a Digital Image?

- A digital image is not "a photo", but a **matrix of numbers**.
- Each **pixel** (point in the image) has an **intensity value**.

Example 1 – Grayscale image

- Each pixel has **only one number** that indicates brightness:
- **0 = black**, **255 = white**, intermediate values = gray.



[[0, 128, 255],
[64, 200, 90]]

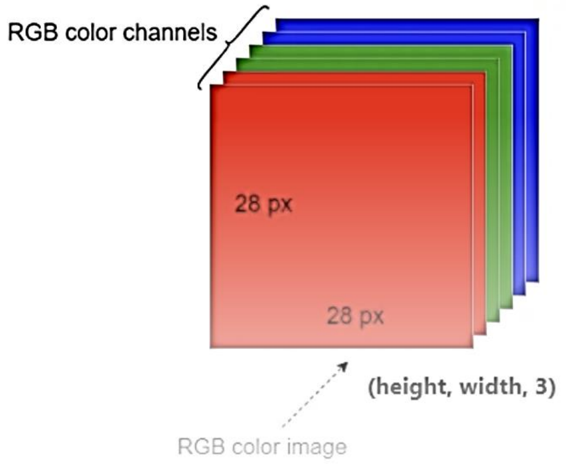
- → the higher the number, the clearer the pixel.
  → this is a **2D (H×W) matrix** (AHeight × Width)

What is a Digital Image?

Example 2 – Color Image (RGB)

- Each pixel has **3 numbers**, one for each color channel:

Pixel	R (red)	G (green)	B (blue)
1	255	0	0
2	0	255	0
3	0	0	255
4	255	255	255
5	0	0	0



- → this is a **3D matrix (H×W×C),** where **C = 3 channels**.

An image (gray or colored) is just **a table of numbers** that the computer can read and manipulate — and that neural networks will use as input.

How we represent an image in AI models

- Images are represented as **tensors**, i.e. multi-dimensional numerical structures.
- Most common facilities:

Type	Form	Meaning
Grayscale	**2D → (H×W)**	Height × Width
Color (RGB)	**3D → (H × W × C)**	Height × width × channels (C=3)
Image datasets	**4D → (N×C ×H×W)**	Number of Images × Channels × Height × Width

In **PyTorch**, images are handled as **4D tensors**, which are perfect for **convolutional neural network (CNN) input**

The Problem: How to Interpret Millions of Pixels

- A 1024×1024 pixel X-ray contains **over 1 million numerical values**.

- If we read them "in a row" as a vector → **we would completely lose the spatial structure** (i.e. *where* the pixels are located between them).

- But in the pictures, **location matters**:
  - A border, lesion, or cell has **a local shape and context**.

- We therefore need a model that:
  - Scan **small local regions (patch),**
  - **Recognize patterns** that are repeated,
  - **Maintains the geometry** of the image.

This is exactly what **Convolutional Neural Networks (CNN) do**.

From images to patterns: the idea behind CNNs

- CNNs are designed to **learn from pixels in a hierarchical way**:
  - **Low levels** → recognize **simple patterns** (edges, lines, textures).
  - **Intermediate levels** → combine patterns in **shapes or structures** (e.g. tissues, cells).
  - **High levels** → recognize **clinical objects or regions** (e.g., lung, injury).
- Each layer "extrapolates" more abstract information → from pixel → to medical concepts.

Key idea: CNNs start from **local details** to build a **global vision**.

Pixels → Edges → Textures → Structures → Lesion / Organ

What does a neural network need to learn from an image?

- From an image, a network must learn to:
  - **Recognize local patterns** (edges, lines, textures).
  - **Combining simple patterns into more complex structures** (tissues, anatomical regions).
  - **Associate clinical patterns with meanings** (e.g., injury, inflammation, normality).

- To do this, you must:
  - Analyze the image **locally** (neighboring pixels).
  - Summarize information into increasingly abstract **features.**
  - Maintain the **spatial relationship** between the parts.

Convolutional Neural Networks (CNNs) are designed to do just that:
they learn hierarchical visual features → from pixels → to clinical concepts.

Section 2

MLP vs CNN

From MLP to CNN: how the vision of an image changes

- In an **MLP (Multilayer Perceptron)** the image is **flattened** into a single 1D long vector.$H \times W \times C$ → means that **all the pixels** are placed "one after the other", as if it were a single list of numbers.

- Each neuron in the first layer is **connected to all the pixels** in the image → so each neuron has **a lot of weights to learn**.

$$y = f(Wx + b), W \in \mathbb{R}^{1\times(HWC)}$$

where:

- x: Input vector (all pixels)
- W: Neuron weights
- b:Bias
- $f(\cdot)$: activation function (e.g. ReLU, sigmoid)

MLP Issues

- **Main problems:**

- **Loss of spatial structure:**
  The network no longer knows that pixels close to each other **are part of the same shape or edge**.
  → Does not recognize local patterns.

- **Too many parameters:**
  Each neuron has a weight for each pixel → huge number of connections → inefficient for large images.

An MLP does not "see" the image as an image, but as a list of numbers.

CNN's Key Idea: Maintain Image Structure

- In **the CNN (Convolutional Neural Network)** the image remains a **3D** *tensor*

$$I \in \mathbb{R}^{H \times W \times C}$$

- Each neuron does not look at all the pixels, but **only at a small local region** of the image (receptive field).
- The weights are no longer all different: they are **shared** in a small array called kernel K.

$$y = f(I * K + b)$$

where:

- I: Image (input)
- K: kernel (shared filter)
- $*$: Convolution operation (the filter **slides** over the image by combining neighboring pixels)
- b:Bias
- $f(\cdot)$: activation (e.g. ReLU)

What does convolution mean

- The **kernel runs over the image**, combining neighboring pixels → extraction of **local patterns** (edges, textures).
- Same filter → fewer parameters, more generalization.
- **Spatial relations** (geometry) are maintained.
- Each filter produces a **feature map** → where the pattern appears.

Training: from weights to filters

- **In MLP** , we train weights that tie *global features* to an output.

- **In CNN** we train **filters (kernels):** small sets of weights that learn to **recognize local patterns**.

  - Each **filter** (e.g. 3×3×3) produces a **feature map**: where that pattern appears.
  - Training optimizes the filter coefficients so that it activates strongly **only** when the right pattern is present.
  - All filters train in parallel with **backpropagation**, but each one "specializes" its function (edges, curves, textures…).

In an MLP, weights connect pixels → classroom;

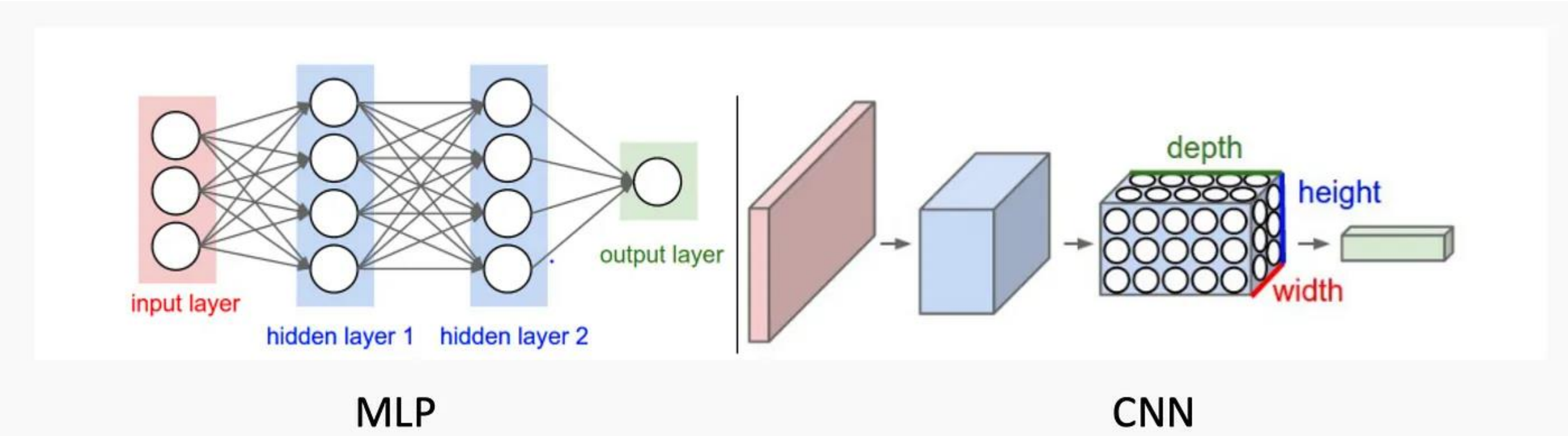
In a CNN, weights define a filter that connects pixels → pattern.

MLP vs CNN Summary Comparison

Aspect	**MLP**	**CNN**
Input form	Flattened in 1D	3D Tensor (H×W×C)
Connections	All Linked Pixels	Local pixels only
Weights	All different	Shared (kernel)
Space structure	Lost	Kept woman
What he learns	Global patterns (loses local details)	Local patterns → shapes → objects
What the parameters represent	Global connections	Shared local filters

Visual comparison

- **MLP** → flattened image, each neuron connected to all the pixels → **many connections, loss of spatial structure**.
- **CNN** → image as a 3D cube (height × width × channels), neurons connected only to small regions → **less weight, geometry maintained, extraction of local patterns**.



From conceptual differences to CNN architecture

- MLP connects *all pixels to all neurons* → learns **global relationships**, but loses geometry.

- **CNN** connects *only local regions* with **shared filters** → learns **spatial patterns**.

- A CNN is not a single layer, but a **hierarchy of blocks**:
  - extracts **increasingly complex features** (from edges → shapes → objects)
  - **reduces** the spatial dimension
  - **Keeps** only relevant information

CNNs progressively transform the image into a **feature vector**, which is then passed to a **final MLP for classification**.

Section 3

Key concepts of CNNs

Spatial structure of images (RGB and 3D)

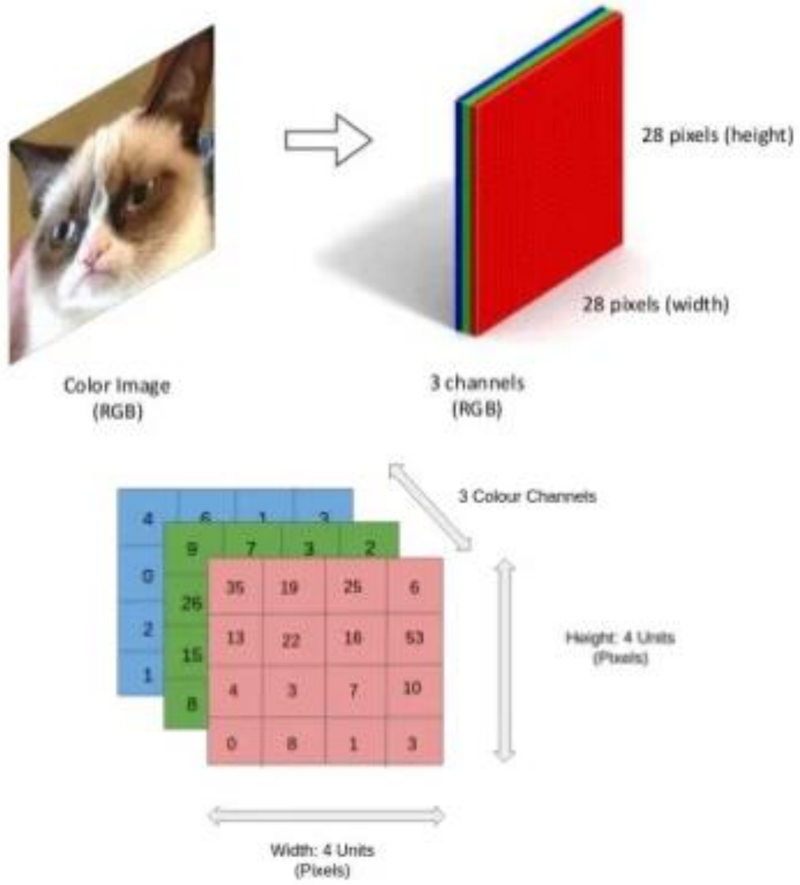
- An image is not a simple list of numbers,
  but a **three-dimensional block of pixels** → a **3D tensor**:
  - **Height (H):** Number of lines (pixels vertically)
  - **Width (W):** Number of columns (pixels horizontal)
  - **Depth (C):** Number of **color channels**

Color images (RGB)

- Each pixel has a **position in space** and a **color** described by three values: **R**bone, **G**green, **B**blue.
- Each channel (R, G, B) is a **matrix** that represents the intensity of a color.
- By adding the three channels, we get the final color image.

Examples of pixels:

- (255, 0, 0) → pure red
- (0, 255, 0) → green
- (128, 128, 128) → grey

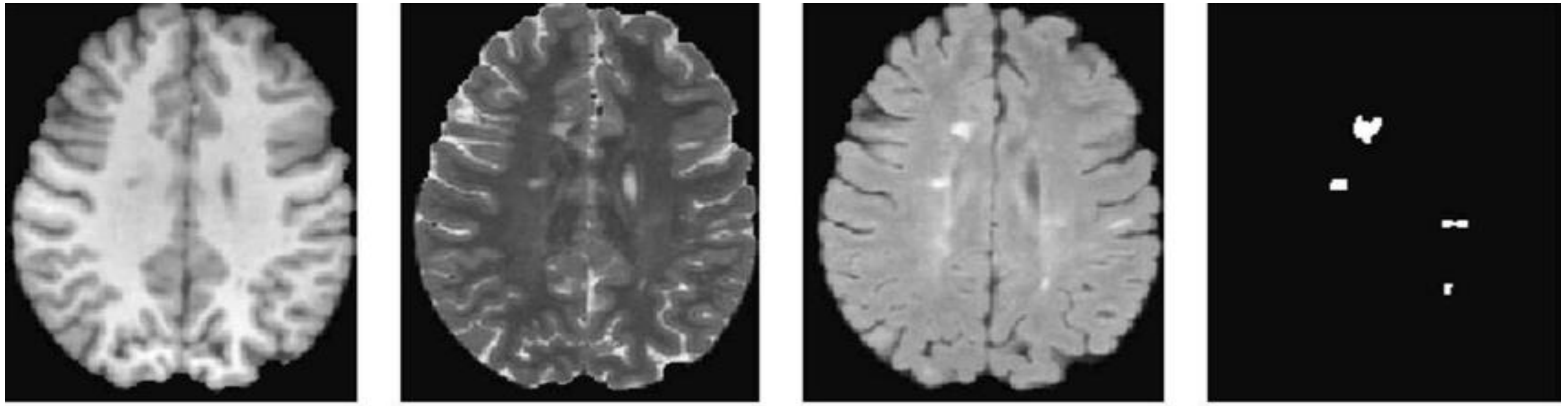


Each channel (R, G, B) is an intensity matrix: added together, they form the color image.

Spatial structure of images

Biomedical Imaging (MRI):

- **In the medical field, different magnetic resonance sequences** are used instead of RGB channels:
  - **T1** → provides fine anatomical details.
  - **T2** → highlights fluids and disease areas.
  - **FLAIR** → suppresses cerebrospinal fluid, making lesions more visible.
- Each voxel/pixel is then described by **multiple values**, one for each sequence.
- CNNs integrate this multi-channel information to improve injury recognition.
- The **Ground Truth image** shows manual segmentation (e.g. areas of injury) used as a reference during training.



T1 T2 FLAIR Ground truth

Role of CNNs

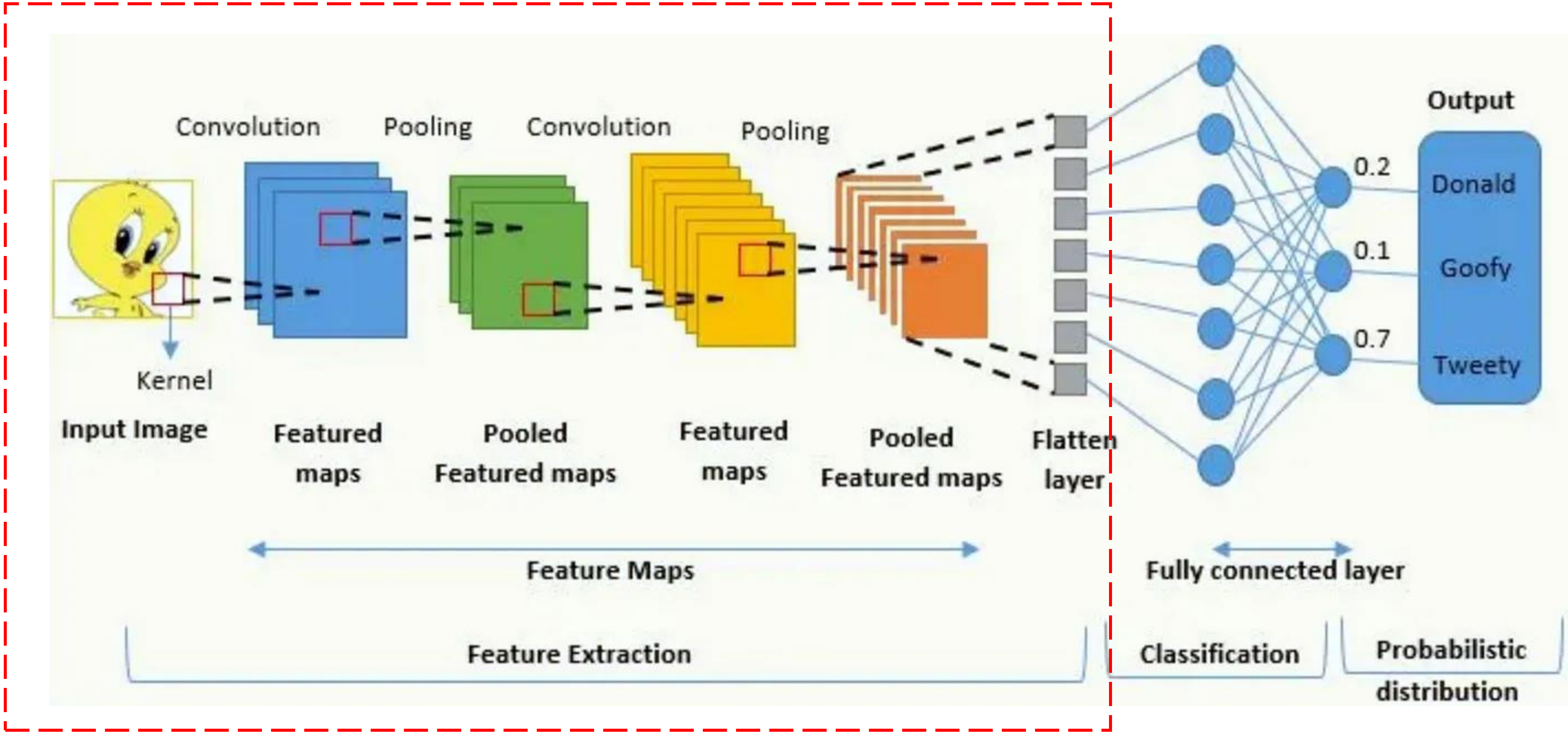
- CNNs work directly on images as **3D tensors** $.H \times W \times C$
- They maintain **spatial relationships** between neighboring pixels → local patterns have meaning (edges, corners, shapes).
- They integrate the different **channels** (RGB or MRI sequences such as T1, T2, FLAIR) for a richer representation.
- They construct **hierarchical representations**:
  - Edges → textures → complex shapes → structures → objects/lesions.

CNNs don't see an image as a list of pixels, but as a **spatial structure rich in relationships** from which to extract information.

General structure of a CNN

- A CNN consists of two main parts:
  - **Feature Extraction** → transforms pixels into more useful representations (visual patterns).
  - **Prediction** → uses these representations to give a final result (e.g. classification).
- Now we will focus on the first part (Feature Extraction).
- We will see how, through convolution and pooling, the network builds:
  - **Feature maps** → new intermediate representations.
  - **Pooled feature maps** → smaller, more robust versions.

General structure of a CNN



From spatial structure to convolution

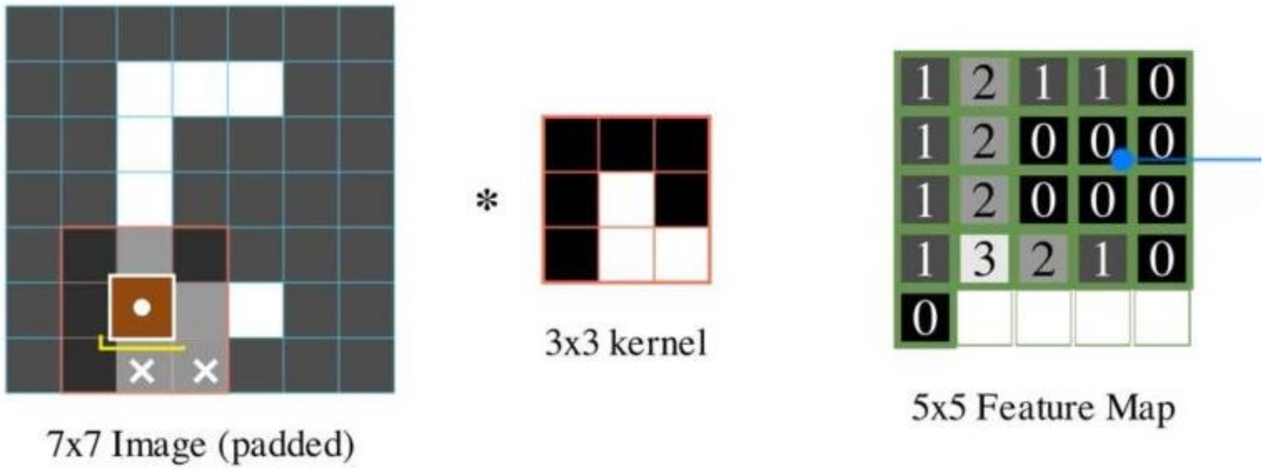
- We have seen that an image is a **3D tensor** .$(H \times W \times C)$
- CNNs **do not flatten** this structure, but analyze it in small portions (**local patches**).
- Each patch is processed by a **filter (kernel)** → matrix of shared weights.
- The goal: to transform raw pixels into **new representations (feature maps)** that highlight useful patterns (edges, lines, textures).

The two basic operations in a CNN

- Each **convolutional layer** always performs two fundamental functions:

1. **Spatial filtering**
   - Convolution applies a filter (kernel) to small local regions.
   - Each filter detects a specific pattern (edges, corners, textures).
   - Output = a feature map that shows "where" that pattern appears.
   - It is the basis for spatial feature extraction.

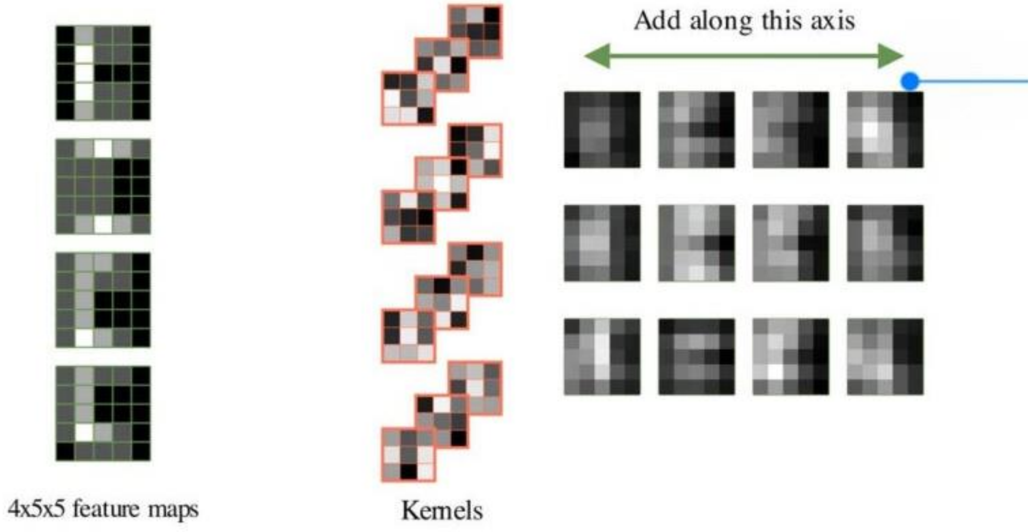
   It is used to extract spatial information (edges, corners, textures).



The two basic operations in a CNN

2. Channel Combination

- Images have multiple **channels** (RGB, MRI T1/T2/FLAIR…).
- CNNs combine information from all channels → **multi-channel** feature maps.
- This allows you to capture patterns that only emerge from channel integration.

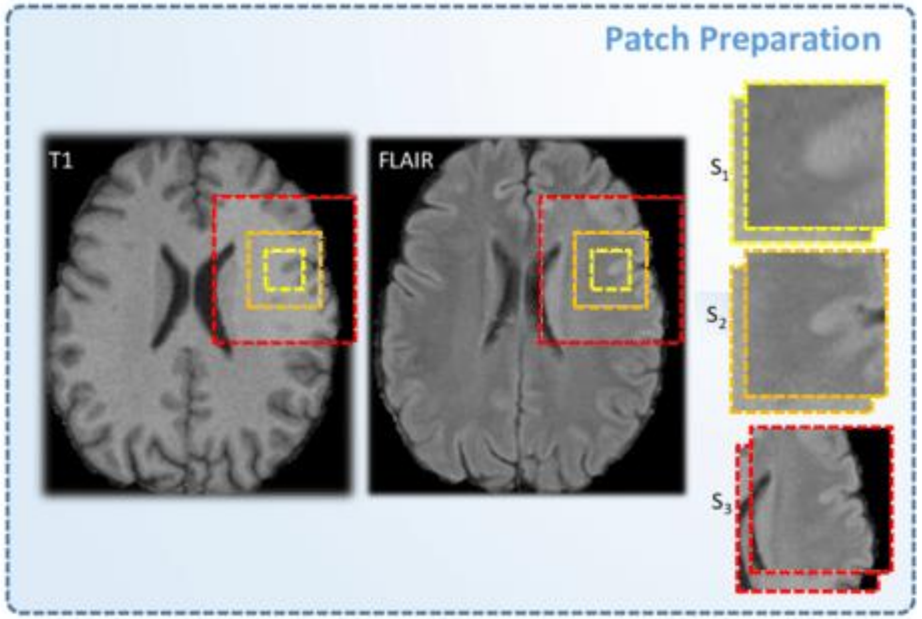


Integrate the different channels into a new feature map.

Patches and Locations

- CNNs analyze **local patches** $K \times K$ (e.g. or):$3 \times 35 \times 5$
    - Each neuron is connected to only a small patch in the image (not all pixels).
    - This drastically reduces the number of connections and takes advantage of the spatial nature of the images.
    - Capture **local information** such as edges, angles, contrasts.
- This locality reduces parameters and reflects how the brain processes images: from small details → complex structures.

Patch + kernel = basic mechanism for extracting features.



Shared filters and weights

- A **filter (kernel)** is a matrix of weights that runs over the entire image (sliding window).$(K \times K)$

- This allows the same pattern (e.g. an edge) to be detected in different positions.

- The same weights are **shared everywhere** → parameters and **spatial invariance**.

- Each filter produces a different **feature map**.

CNNs learn filters that recognize the same pattern wherever it appears.

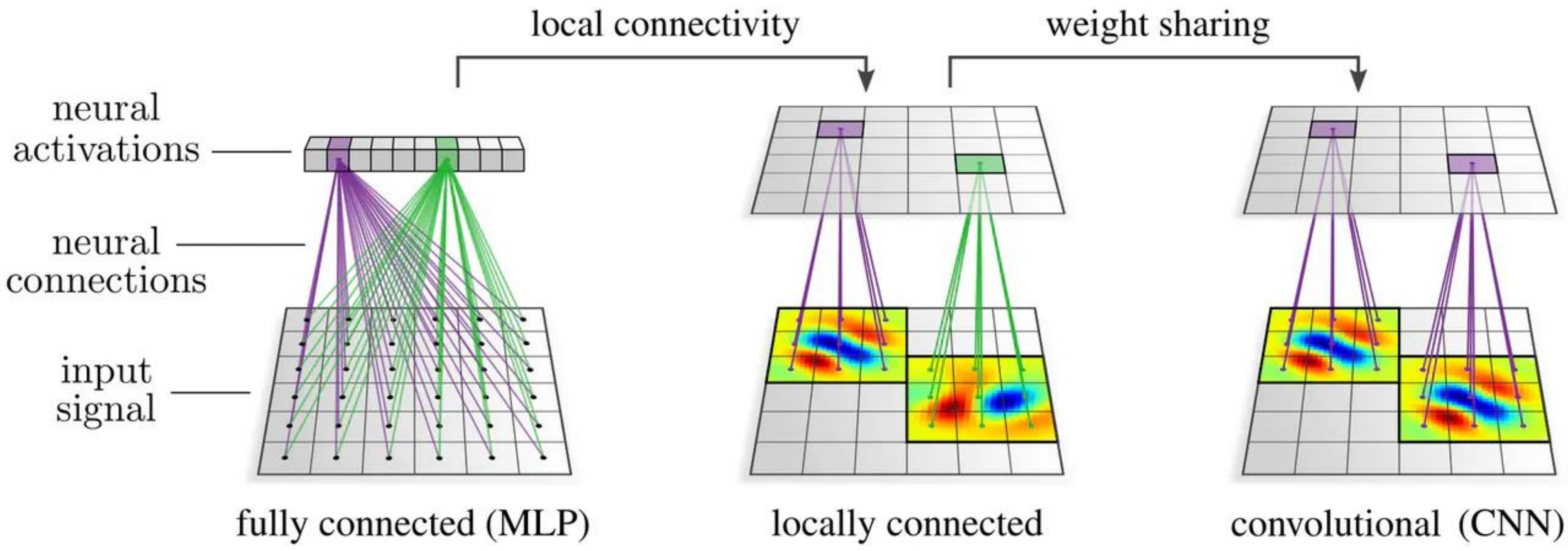
Filter examples

- Examples of classic filters (historical, unlearned):
  - Vertical border: $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$
  - Horizontal border: $\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
- In modern CNNs the filter weights are **randomly initialized** and **learned with backpropagation**.
- The model learns on its own *which* patterns are useful for the task.

Locations and Shared Weights (MLP vs CNN Comparison)

- **Fully connected networks (MLPs):** each neuron is connected to all pixels → too many parameters.
- **Local connectivity:** each neuron processes only one **local patch** → reduces complexity.
- **CNN:** They add the concept of **weight sharing** → the same filter runs over the entire image.

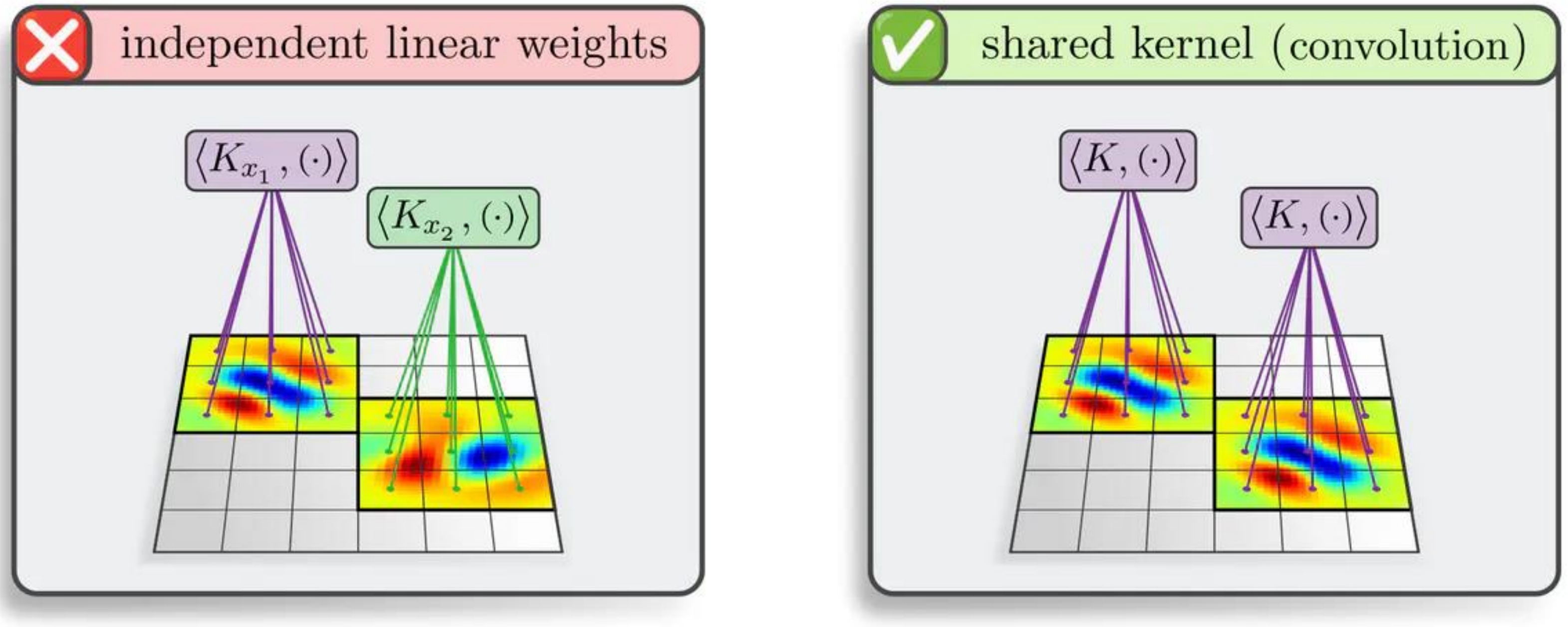
The result: fewer parameters, more efficiency, the ability to recognize patterns anywhere.



Independent vs shared weights

Independent weights: Each patch has its own filter → inefficient, no generalizations.

- **Shared weights (CNNs):** A **single filter** applied to all positions → recognizes the same pattern in every part of the image.
- **Advantage:** Reduced parameters + robustness → if an edge appears on the top left or bottom right, the filter still recognizes it.

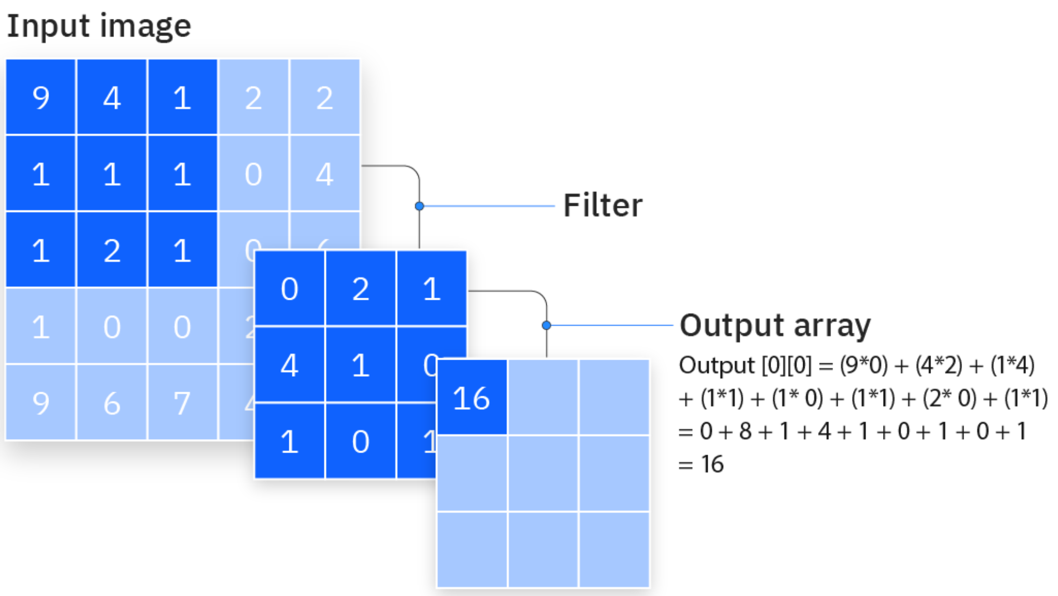


Convolution: basic concept

- A **convolution** takes a small pixel window (**patch**) from the image, such as $.K \times K$
- On this window we apply a **filter (kernel),** which is also a small matrix of numbers (weights) $K \times K$.
- Operation:
  - multiply each pixel by the corresponding number of the filter,
  - we add up all the results,
  - we get a single value.
- This value becomes a "new pixel" in an image transformed → the **feature map**.
- By repeating the operation on the entire image we get a whole map that highlights a certain **pattern** (edges, lines, textures…).

Numerical example

- Patch 3×3 taken from the image.
- Kernel 3×3 applied → element-by-element multiplication + sum.
- Output = 16 (in the first pixel of the feature map).
- Proceeding through the entire image → complete feature map.



Convolution formula

- Convolution calculates how well a filter "fits" a small area of the image.
- Mathematically, for each position (,), the value of the new image (feature map) is obtained by multiplying and adding the pixels of the image with the weights of the filter.

$$S(i,j) = \sum_{m}\sum_{n} I(i+m, j+n) \cdot K(m,n)$$

- I= Input Image
- K= filter (kernel)
- $S(i, j)$= value of the feature map in place (i, j)

Each filter produces **a different feature map**, because its weights are learned autonomously during training.$K(m, n)$

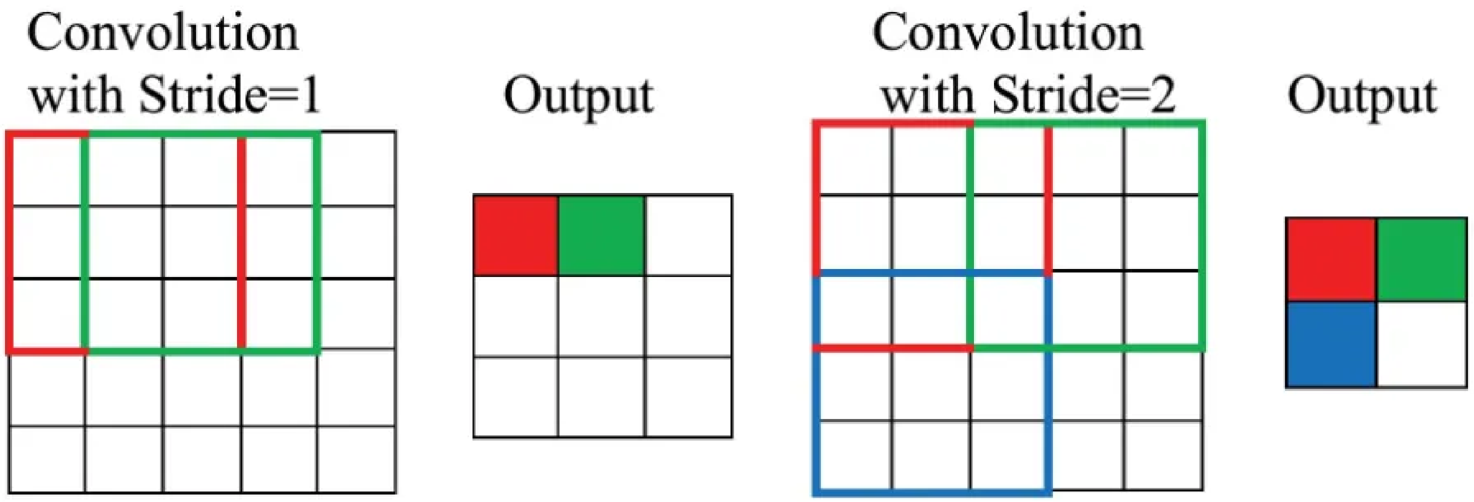
Stride and Padding: rules of convolution

- **Stride (S):** The pitch at which the filter moves.
  - **Role—** Controls the "resolution" of the feature map.
  - **Advantages:** high screeches → less calculations, compression.
  - **Disadvantages:** Loss of fine details.
  - **Use:** S=1 for small/detailed images, S>1 to reduce size and complexity.

- **Padding (P):** Dummy pixels around the edges.
  - **Role: Controls** whether edges are preserved.
  - **Benefits:** Preserves information at the margins.
  - **Disadvantages:** introduces artificial pixels (zero-padding).
  - **Use:**
    - *Same* padding when it is necessary to maintain size;
    - *Valid (no padding)* for more compact output.

Stride: Filter pitch effect

- **Stride = 1** → high resolution, captures fine details.
- **Stride = 2 (or >1)** → more compact output, less detail.
- **Compromise:**
    - small stride → more information but more calculations;
    - screech large → fewer calculations but risk of losing local patterns.



Padding: Adding "space" around the image

- You add a frame of dummy pixels (typically 0) to the edges.
- **Same padding (with padding):** dimensions are preserved → edges also contribute to convolution.
- **Practical role:** allows filters to "see" edges as well, → useful when details at the edges are important (e.g. medical images).

Stride and Padding: Summary

Parameter	Role	Advantages	Detriments
Stride (S)	Check the resolution of the feature map	Reduces size, less compute	Loss of detail if too high
Padding (P)	Preserve image edges	Retains edge info, useful for deep convolutions	Adds dummy pixels (may introduce noise)

Feature Maps

- The output of the convolution is a **feature map**.
- Each filter generates a **feature map**, which highlights where the pattern you are looking for is present.
- Using multiple filters we obtain a **stack of feature maps**, i.e. many parallel representations of the same image.
- In the first few layers, feature maps capture **simple local patterns** (edges, lines).
- In subsequent layers, by combining multiple convolutions, feature maps represent **more complex patterns** (shapes, structures).

Output Size Formula

When we apply a convolution, the size of the output (per side) is:

$$O = \frac{W - K + 2P}{S} + 1$$

- W= input size (width/height)
- K= kernel size
- P= padding
- S= stride

This formula tells us **how many neurons the feature map will have** after convolution.

- If **S increases**, the output decreases (larger jumps).
- If **P increases**, the output increases ("protected" edge).
- If **K increases**, the output decreases (the filter covers more area).

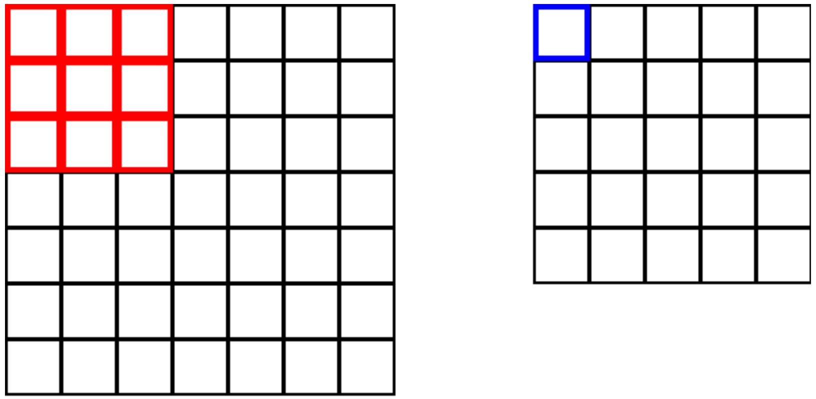
Output Size Formula

Example:

- Input = , Kernel = , Stride = 1, Padding = 07 $\times$ 73 $\times$ 3

$$O = \frac{7 - 3 + 0}{1} + 1 = 5$$

→ The output is .5 $\times$ 5



Each neuron of this sees a patch 3 × 3 of the original image.5 $\times$ 5

Receptive Field

- Each neuron in a feature map "sees" only a small region of the original image → its **local receptive field**.
- In the deeper layers, the receptive field grows → neurons combine more local information → **global patterns**
- Recursive formula (per layer):l

$$RF_l = RF_{l-1} + (K_l - 1) \times \prod_{i=1}^{l-1} S_i$$

Where:

- K_l= kernel size at layer l
- S_i= stride del layer i
- $RF_0 = 1$

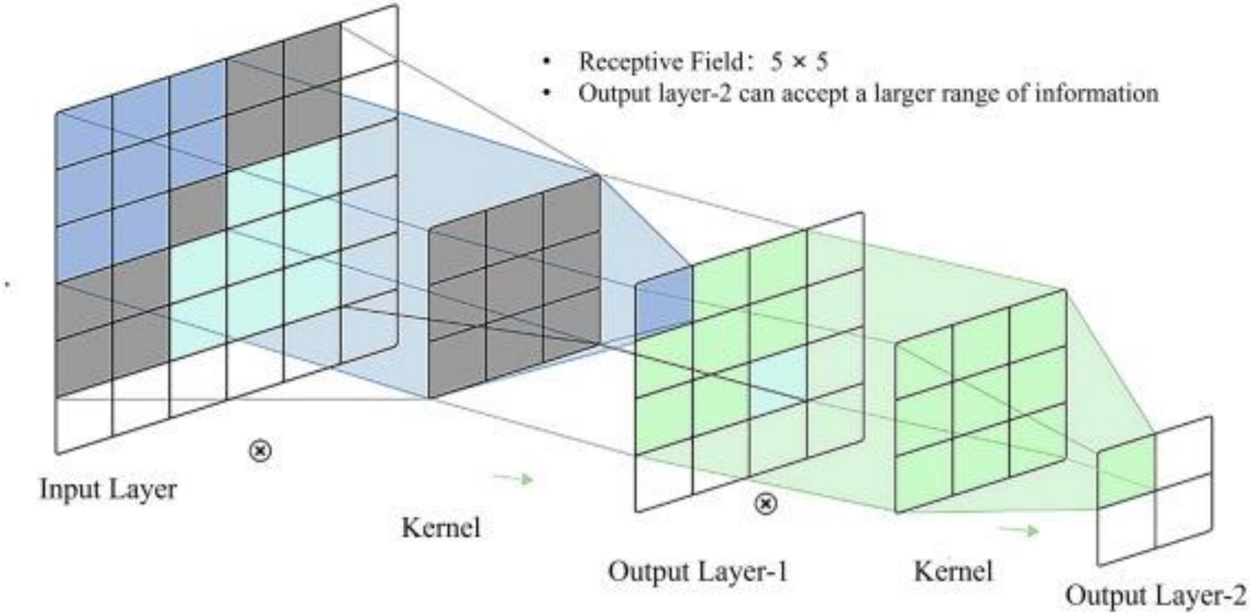
Example of RF growth

1.First layer

1. Kernel , stride 13×3
2. Output: feature map5×5
3. Each neuron sees a patch.3×3

2.Second layer

1. Kernel sulla feature map $3 \times 35 \times 5$
2. Output: feature map3×3
3. But each neuron now indirectly "covers" pixels of the original image.5×5



Each layer sees a larger portion of the original input.

Why the Receptive Field Grows

- Each layer sees a larger portion of the original input.
- More layers = larger receptive field even if kernels remain small.
- Stride > 1 make the receptive field grow even faster.

Deep layers don't look at pixels → look at **increasingly large and complex patterns**.

Combining channels in CNNs

- So far, we've only seen convolution on **one channel** (a single image).
- In reality, images have **multiple channels**:
  - Natural → 3 (RGB)
  - Biomedical → multiple sequences (e.g. T1, T2, FLAIR…)
- CNN's filters are therefore **3D**:
  - Height × Width → spatial part of the kernel
  - Depth → number of input channels
- Each filter:
  - **"sees" all the channels** of the image,
  - combines its values,
  - and produces **a single output feature map**.

$$S(i,j) = \sum_{\ell=1}^{C_{in}} \sum_{m} \sum_{n} I_\ell(i+m, j+n) \cdot K_\ell(m,n)$$

Each filter learns to combine **all channels of the input** into a new representation.

Channel Combination (RGB)

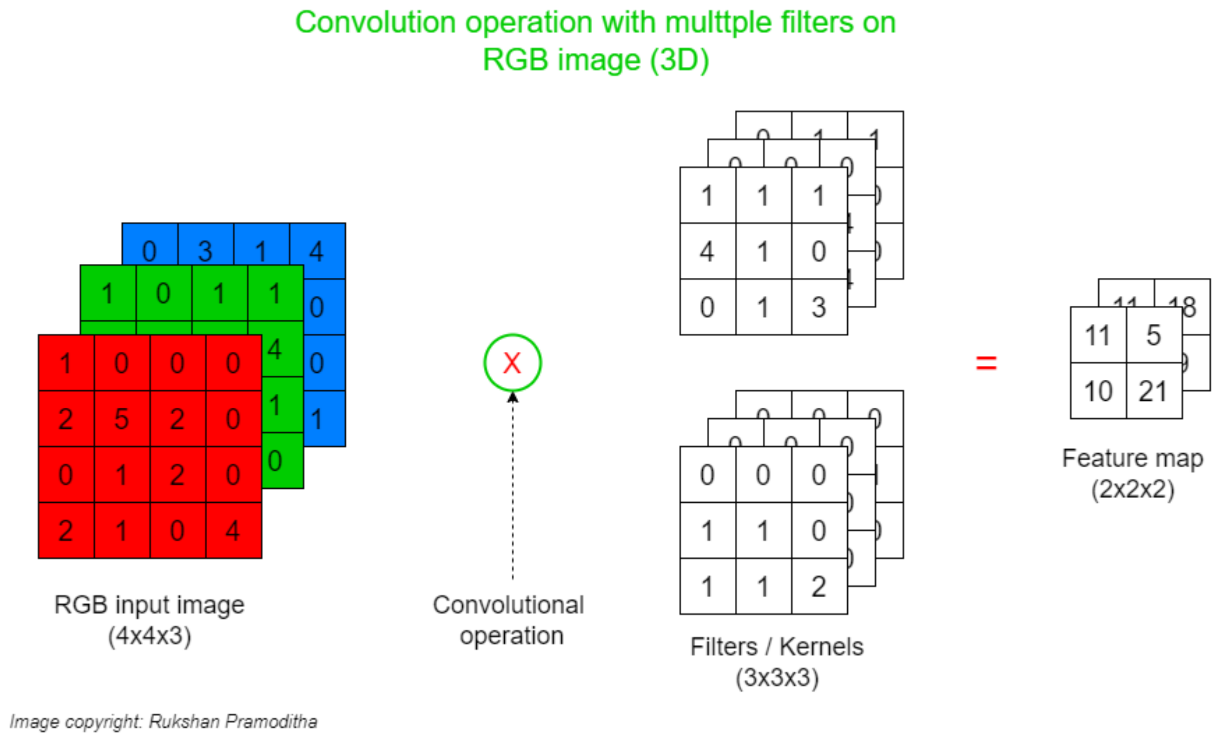
- Each convolutional filter "sees" **all** channels of the image (e.g. R, G, B).
- The output is a **feature map** that combines color and shape information.

Practical example

- Input: **4×4×3**
  (height × width × 3 RGB channels)
- Filter (kernel): **3×3×3**
  → moves 1 pixel at a time
- Input , kernel , padding , $W = 4K = 3P = 0$
  stride $S = 1$:

$$O = \frac{4 - 3 + 2(0)}{1} + 1 = 2$$

- The output is **2×2.**



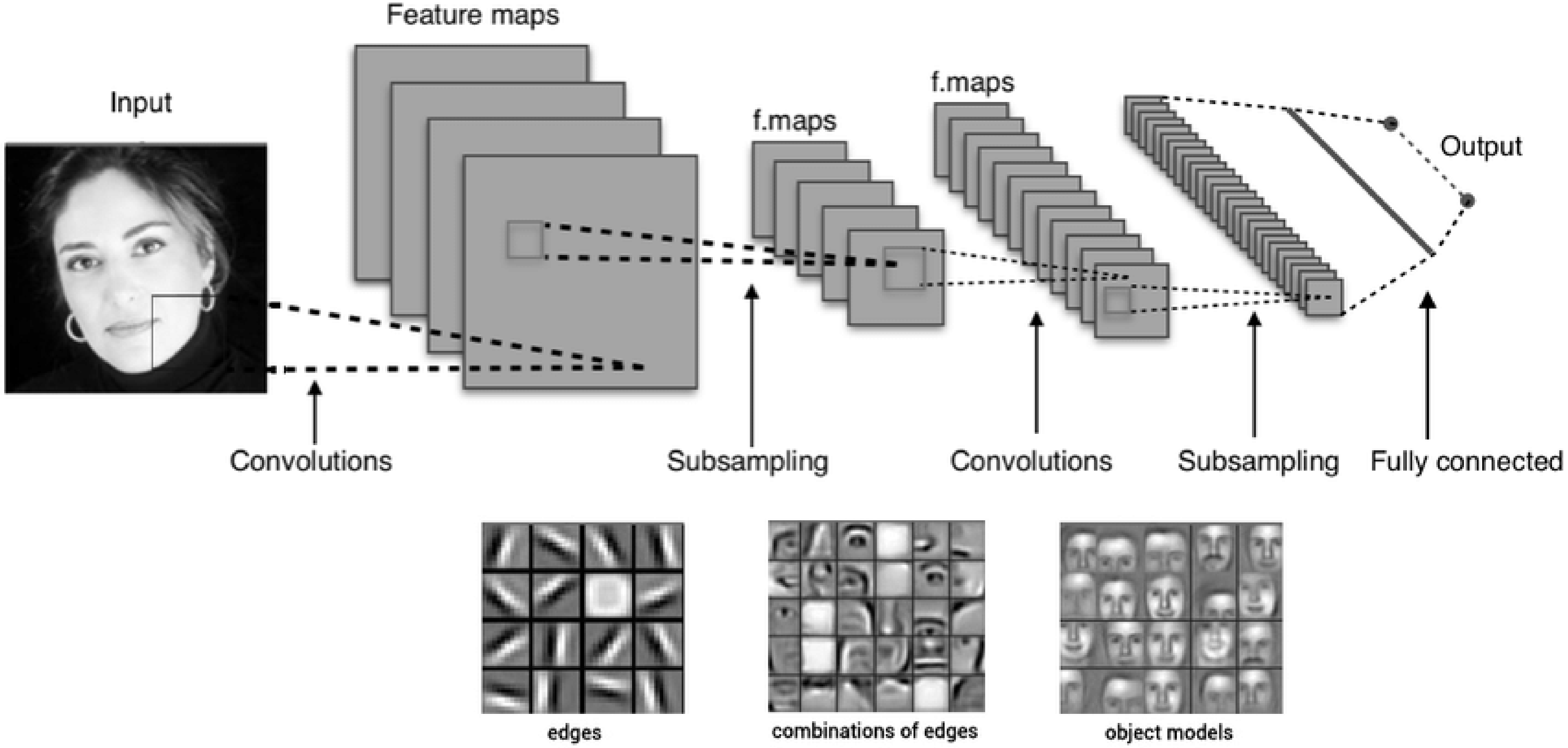
Summary: From Pixel to Feature Maps

- **Convolution** → filters (kernels) analyze small **local patches**.
- **Shared weights** → same weights over the entire image → fewer parameters, spatial invariance.
- **Feature maps** → each filter generates a new representation, highlighting specific patterns.
- **Stride & Padding** → control the size of the output and how much the field of view grows.
- **Receptive field** → with multiple layers you go from local details (edges) to global structures (shapes, objects, lesions).

CNNs construct **hierarchical representations**: from pixels → to edges → to

textures → up to complex structures.

From Local Features to Hierarchical Representations



Activation after convolution

- **A nonlinear trigger function** is applied after each convolution.
- Because?
  - Without →, CNN would be just a linear combination of filters.
  - With →, the network can learn **complex, non-linear patterns**.
- La più comune: **ReLU (Rectified Linear Unit)**.

Effect of ReLU

- Definition: $f(x) = \max(0, x)$
- Effect: negative values → 0; positive → unchanged.
- Key benefits:
  - It only keeps significant activations.
  - Computationally simple and efficient.
  - Reduces the problem of gradient saturation.
- Less used alternatives: sigmoid, tanh.

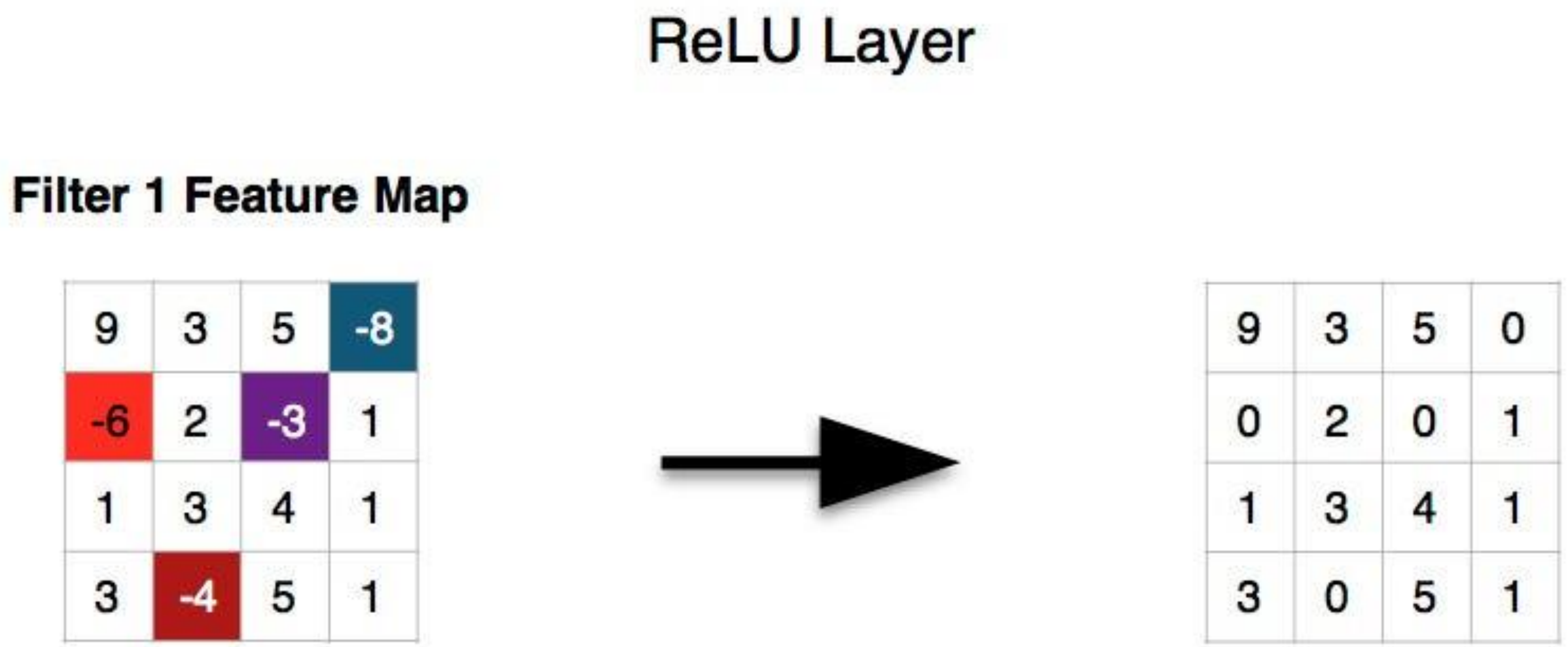
Effect of ReLU

- Negative values indicate the same pattern but with **opposite polarity** (e.g. white→black vs black→white border).

- ReLU simplifies the representation, keeping only positive and stable activations.

- If both polarities are needed, the network learns **more dedicated filters**.

Effect of ReLU



From convolution to pooling

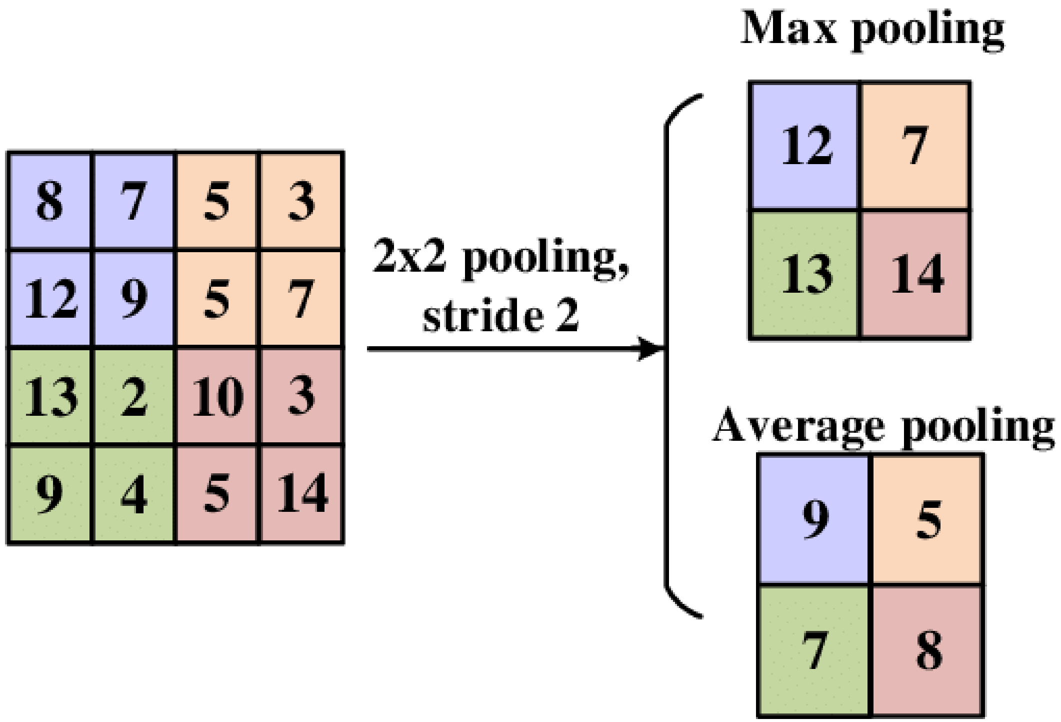
- After Convolution + ReLU, feature maps contain detailed local information.

- Pooling is used to **reduce dimensionality** and make features more **robust and invariant**.

- The most common:
    - **Max Pooling** → takes the maximum value.
    - **Average Pooling** → takes the average.

Pooling Example (2x2, Stride 2)

- Input: 4×4 matrix.
- Pooling with 2×2 window stride 2 → 2×2 output.
- Max pooling → takes the highest value in each region.
- Average pooling → calculates the average in each region.
- Reduces size while retaining the most important information.



Max vs Average Pooling

- **Max pooling**
  - Highlights the strongest features (e.g. hard edges).
  - Most used in modern CNNs.
  - It can lose "weak" but useful information.

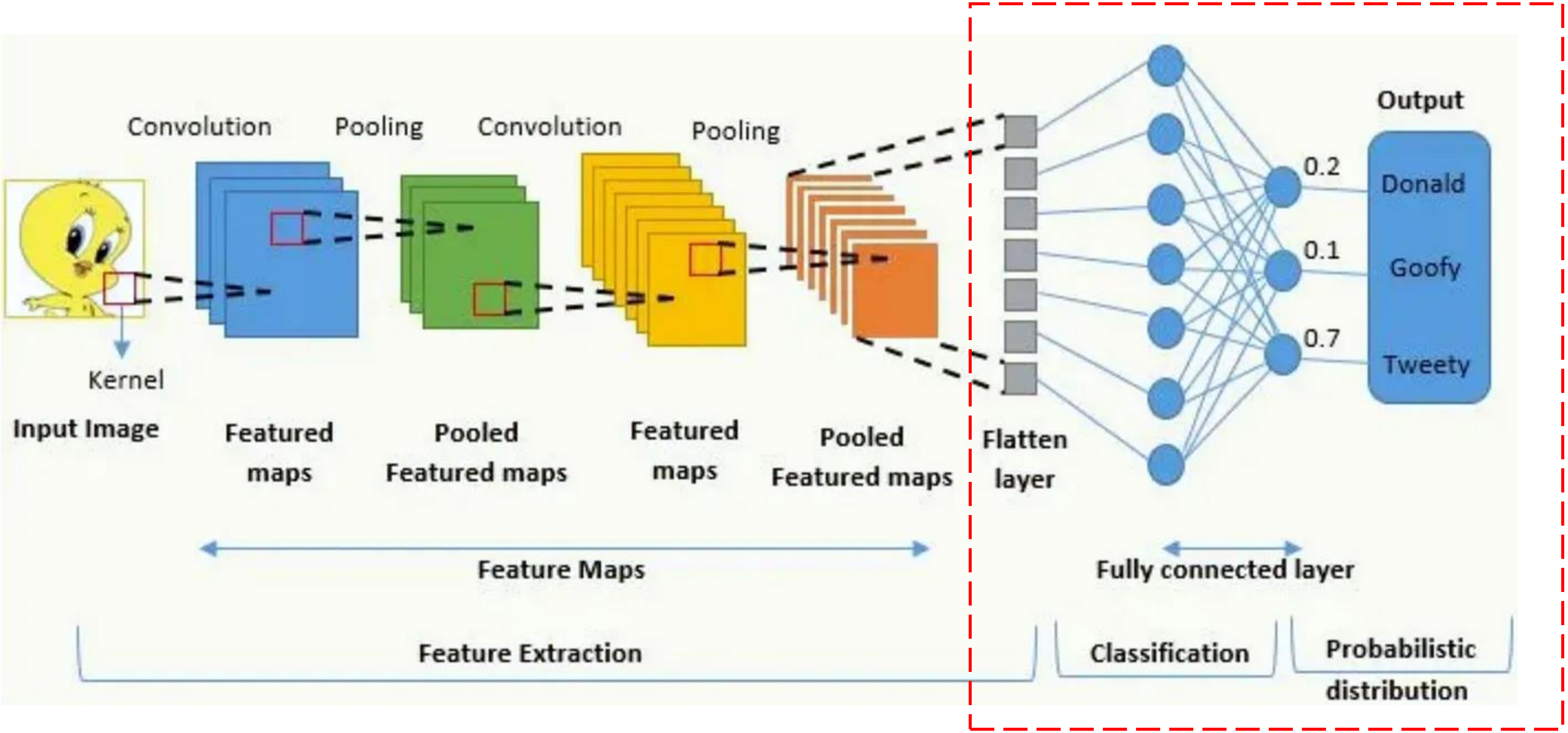
- **Average pooling**
  - It maintains a "softer" and more distributed information.
  - It risks blurring the important details.
  - Used less today, but useful in some contexts (e.g. noise reduction).

- **Today**
  - Pooling **is not mandatory**.
  - Many modern CNNs use **Global Average Pooling** (each feature map is reduced to a single value, global average, before the classifier).
  - In other architectures, you prefer **to reduce pooling** and use convolutions with *stride*.

Typical architecture of a CNN

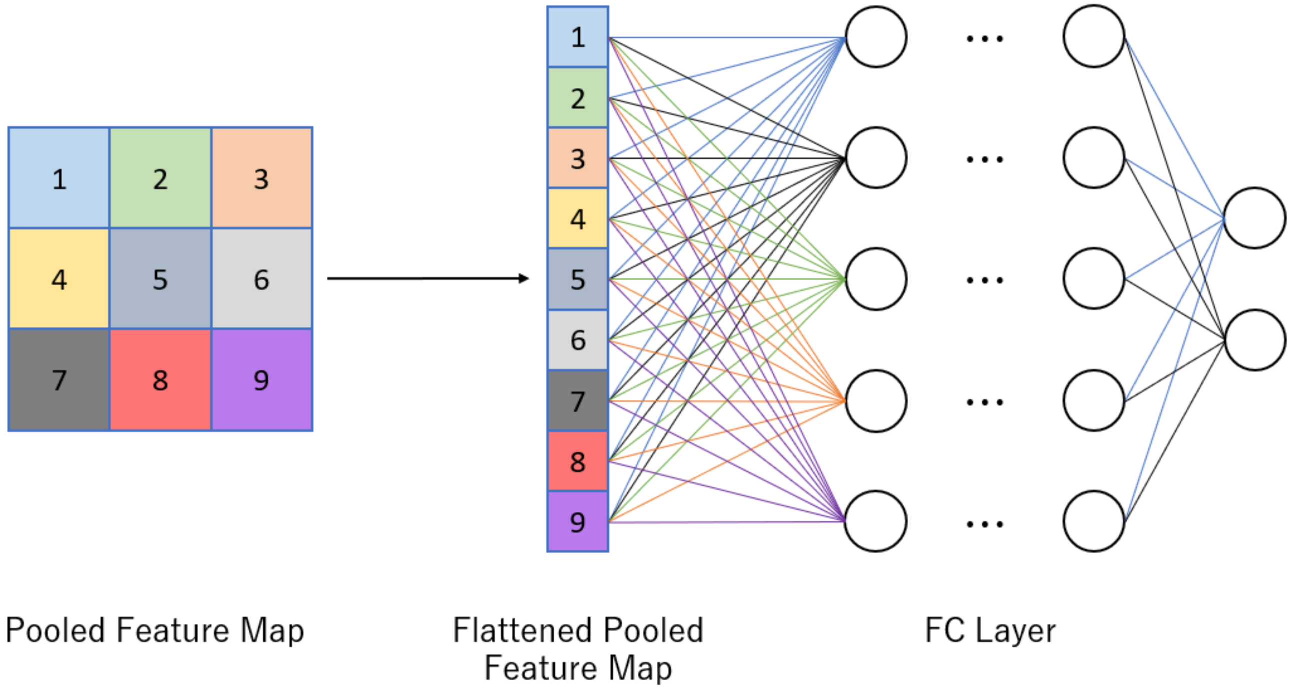
- **Repeated blocks:**
  - Convolution → Activation (ReLU) → Pooling
  - They extract increasingly abstract and hierarchical features.
- **Final phase:**
  - Flatten → Fully Connected Layer → Output
- Key concept: **automatic feature extraction** → **final decision**.

Typical architecture of a CNN



Fully Connected Layer (FC)

- Definition: Each neuron is connected to *all* values of the "flattened" feature map.
- Role:
  - Combines local features → global decision.
  - It works as an "MLP classifier" on top of the extracted features.
- Difference with Convolutional Layer:
  - Local → conv, shared weights.
  - HR → global, independent weights for each connection.
- Advantage: It allows the network to integrate **all the learned patterns** into a single decision.



Output layer

1. **Classification**
    1. Softmax (multi-class) o Sigmoid (multi-label).
    2. Output = probability per class.

2. **Regression**
    1. Output = continuous numeric values.
    2. E.g. age, size, risk score.

3. **Segmentation**
    1. Output = mappa (pixel-wise).
    2. Each pixel classified in a class.

4. **Object Detection**
    1. Output = coordinates bounding box + class probability.
    2. Used in advanced architectures (YOLO, Faster R-CNN).

Key message: **The same CNN structure can be adapted to different tasks by modifying the final layer**.

At a glance — CNN Architecture

- A CNN is a **modular network** that processes images in several stages: **Convolution → ReLU → Pooling → Flatten → Fully Connected → Output**
- Each block learns **increasingly complex patterns**: from → borders to shapes → objects → classes.
- Filters (kernels) are the parameters that the network **learns by itself** through **backpropagation** and **optimization**.

A CNN transforms raw pixels into meaningful features, all the way to a final decision (class or value).

Section 3

CNN in Python

Python Ecosystem for Computer Vision

- Images are not tables or vectors, but **3D structures of pixels (H× W × C).**

- To manage them, we need a Python ecosystem that allows us to:
  - **read** and **transform** images;
  - **convert them into tensors** for the neural network;
  - **build and train** convolutional models (CNNs);
  - **Visualize** what the model is learning.

- **The two pillars of the PyTorch ecosystem for Computer Vision:**
  - **Torch** → creates tensors, models, and calculates gradients.
  - **TorchVision** → manages datasets, images, and pre-trained models.

Required libraries

```
pip install torchvision
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets, transforms
```

Key concepts:

- `torch` = mathematical brain (handles tensors and operations)
- `nn` = network building blocks (layers, activations)
- `Optim` = engine that updates weights (optimizers)
- `torchvision` = "bridge" between images and PyTorch

Image tensors: the shape of the data changes

- In an MLP we had 2D tensors:

```
(batch_size, n_features)
```

- In CNNs, each image is **a matrix of pixels with depths (channels)** → 3D tensor.
- And when we work on batches of images:

```
(batch_size, channels, height, width)
```

Image tensors: the shape of the data changes

Example:

```
images, labels = next(iter(train_loader))
print(images.shape)
# torch. Size([64, 3, 224, 224])
```

This means:

- 64 images per batch
- 3 color channels (RGB)
- 224x224 pixels in size

Didactic note:

- In PyTorch, **channels** always come before height and width.
  (in NumPy it was H×W×C, in PyTorch it is C×H×W)
- It is essential to remember this to avoid mistakes in convolutions!

Il modulo `torchvision.transforms`

- Convolutional neural networks **cannot work directly on image files** (`.png`, `.jpg`, `.tif`, etc.).

- The models are trained only on **number tensors**:
  each pixel must be converted into a number (0–255 → 0–1) and made **uniform in size** (same H×W for all images).

- `Torchvision.transforms` is a library that allows you to create **transformation pipelines** to be applied to each image *before* sending it to the network.

Il modulo `torchvision.transforms`

Transformation	Function	Example of use
transforms. Resize((H,W))	Resize all images to the same shape	from 100×80 → 32×32
transforms. CenterCrop(28)	Crop the center area (for images with a black border)	Useful for X-rays
transforms. ToTensor()	Converts a PIL or NumPy image to a **3D tensor** (C×H×W) with values [0,1]	Required for CNN
transforms. Normalize(mean, std)	Apply normalization (z-score) to pixels for each channel	improves numerical stability
transforms. RandomHorizontalFlip()	Randomly flip the image	Data Augmentation
transforms. RandomRotation(10)	Rotate the image ±10° randomly	Data Augmentation
transforms. ColorJitter()	Slightly varies brightness/contrast	Useful in histopathology
transforms. Compose([...])	Combine multiple transformations in sequence	Full pipeline

Each transformation returns a new modified image → Compose() is used to concatenate them into a single logical flow.

Complete example with `transforms. Compose()`

```
from torchvision import transforms

# Transformation pipeline definition
transform = transforms. Compose([
    Transforms. Resize((32, 32)), # resize all images
    transforms. RandomHorizontalFlip(),   # flip casuale (solo train set)
    transforms. ToTensor(),               # converte in tensore C×H×W
    Transforms. Normalize((0.5,), (0.5,)) # normalize pixel values
])
```

Example Explanation

- **Resize**
  All images must be the same size, → CNNs do not accept variable inputs.

- **RandomHorizontalFlip**
  Increases dataset diversity (data augmentation).
  Each era sees slightly different versions of the same images → helps generalize.

- **ToTensor**
  Converts an RGB image (3 channels) to a shape tensor (3, H, W)
  and divides the pixel values by 255 → from [0–255] to [0–1].

- **Normalize**
  Applica la formula:

$$x' = \frac{x - \mu}{\sigma}$$

- for each channel of the image.
- It helps the network converge **faster**, because the average values approach 0 → **more stable gradient** during training.

Il modulo `torchvision.datasets`

- The `torchvision.datasets` **module provides ready-to-use** or easily customizable datasets.

- Each dataset returns **pairs of data** `(image, label)` that can be:
  - real images (x-rays, histology, photos, etc.),
  - numerical labels (class, pathology, category…).

It integrates seamlessly with `transforms` and `DataLoader modules.`

Dataset predefiniti in `torchvision.datasets`

PyTorch includes **ready-made, standardized datasets** that are ideal for practice.
Each dataset automatically provides:

- data downloads,
- subdivision into train/test,
- integration with DataLoader.

Dataset	Image Type	Classes	Canals	Dimension
MNIST	Handwritten digits	10	1	28×28
FashionMNIST	Clothing	10	1	28×28
CIFAR10	Common Objects	10	3	32×32
ImageNet	Complex objects	1000	3	224×224
MedMNIST	Real clinical data (e.g. Chest, Path, Blood, OCT…)	Variable	1 or 3	28×28 or 64×64

Example with MNIST

```
from torchvision import datasets

train_data = datasets. MNIST(
    root='data', # where to save files
    train=True,           # True = dataset di training
    transform=transform, # transformation pipeline
    download=True # download automatically
)
```

How it works behind the scenes:

- Download images and labels from the web.
- Automatically applies the transformations you define.
- Returns pairs (`image, label`) each time it is called.

DataLoader: the bridge between Dataset and Model

The `DataLoader` is used to:

- create data batches (`batch_size`),
- mix them (`shuffle=True`),

```
from torch.utils.data import DataLoader
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
```

1. The `DataLoader` takes 64 images from the dataset.
2. Pack them in a 4D tensor `(64, C, H, W)`.
3. It returns batches ready to send to the network with a simple `for`.

Full example: Dataset to batch

```
for images, labels in train_loader:
    print(images.shape)
    print(labels.shape)
    break
```

Output:

```
torch. Size([64, 1, 32, 32]) #64 grayscale images
torch. Size([64]) # 64 numeric labels (0–9)
```

You can now directly switch `images` to the CNN model:

```
outputs = model(images)
```

Data augmentation in medical images

In the case of **X-rays, CT scans or histologies**, transformations are essential to make the model robust:

```
train_transform = transforms. Compose([
    Transforms. RandomRotation(15), # light rotations
    transforms. RandomResizedCrop(224),      # ritagli casuali
    transforms. RandomHorizontalFlip(),      # flip orizzontale
    transforms. ColorJitter(brightness=0.2, contrast=0.2),
    transforms. ToTensor(),
    transforms. Normalize(mean=[0.5], std=[0.5])
])
```

Clinical motivation:

- Medical images may vary in contrast, location, or lighting.
- Augmentation simulates these variations → the network learns robust patterns, not irrelevant details.

PyTorch ecosystem for data management

Object	Form	Role
transforms	torchvision	defines the transformations to be applied to images
datasets	torchvision	provides standard or custom datasets
DataLoader	torch.utils.data	Batch Handles and Optimizes Upload
ToTensor()	transforms	converts image → 3D tensor
Normalize()	transforms	Center and scale pixel values

Images (from files or public datasets) are → transformed into **4D normalized tensors (batch, channels, height, width→**
ready to be fed into a convolutional network.

CNN architecture construction

Convolutional Neural Networks (CNNs) are networks designed to process images.

- They are composed of **modular blocks** that are repeated several times.
- Each block learns to recognize **increasingly complex patterns** (from edges → shapes → objects).

```
[ Convolution → ReLU → Pooling ] × N → Flatten → Fully Connected →
Output
```

Each block reduces the **size of the image** (fewer pixels) but increases **the number of maps** that describe what it has learned.
More blocks → the network recognizes more and more complex details.

What each component does

Layer	Function	Output
Conv2d	Extracts local patterns via **filters**	Feature maps
ReLU	Makes triggers **non-linear**	Positive activations
Pooling	Reduces size and noise	Synthetic Features
Flatten	Transform 3D → 1D tensors	Feature vector
Linear	Combine Features in Final Decision	Classes or numeric values

Each **filter** (e.g. 3×3) slides over the image and **recognizes a specific pattern** (edges, lines, textures).
Subsequent layers learn **increasingly complex patterns** (shapes → objects → anatomical structures).

What inputs each component needs

Layer	Main Parameters	What they mean
Conv2d	`in_channels, out_channels, kernel_size`	how many channels enter (1 for B&W, 3 for RGB), how many filters to learn, size of filters
ReLU	—	Activation: Used after each convolution
MaxPool2d	`kernel_size, stride`	How Much To Reduce Size (2×2 Half Image)
Flatten	`start_dim=1`	flattens all features in a vector
Linear	`in_features, out_features`	how many numbers go in (depends on Flatten), how many outgoing neurons (classes)

Some numbers are **fixed (depending on the data),** others **are chosen** based on the network you want to build.

How the dimensions change inside the network

- Each layer changes **the height, width, and depth** of the image.

Operation	**What it does**	**Effect on shape**
`Conv2d`	Apply local filters	reduces height/width
`Pooling`	summarizes the information	halves the size
`Flatten`	turn to list	3D → 1D
`Linear`	combina le feature	da 1D → final output

- **Typical tensor shape:** `(batch, channels, height, width)`

Example:

`torch. Size([64, 1, 28, 28])` → 64 28×28 grayscale images.

Calculate the size after a convolution

General formula:

$$\text{Output} = \frac{(Input - Kernel + 2 \times Padding)}{Stride} + 1$$

Example:

```
Input = 28, Kernel = 3, Padding = 0, Stride = 1
→ (28 - 3 + 0) / 1 + 1 = 26
```

Image changes from **28×28 → 26×26**

If you then apply `MaxPool2d(2,2)` → halved → **13×13**

Simple rule:

Conv → "tightens" the image.

Pool → "reduces" further (summarizes).

From image to vector: the Flatten

- After convolutional blocks, the image is made up of **many maps** (one for each filter).
- To connect it to the classifier, we "spread" them in a single vector:

$$\mathtt{Dimensione\ finale} = \mathtt{Canali} \times \mathtt{Altezza} \times \mathtt{Larghezza}$$

Example:

- Output after Pooling: (8, 13, 13)

  → 8 × 13 × 13 = 1352

  ```
  self.fc1 = nn. Linear(1352, 2)
  ```

Now the network knows that 1352 numbers arrive from the convolutional block to be connected to 2 classes.

How to choose the right numbers

Parameter Type	Example	How to choose it
`in_channels`	1 or 3	Depends on the type of image
`out_channels`	8, 16, 32…	How many patterns do you want to learn
`kernel_size`	3 or 5	how "large" the area the filter looks at
`pool_size`	2	almost always 2 (half image)
`Linear(..., num_classes)`	2, 10, 14	depends on the dataset

Practical rules for setting up a CNN

1. Number of filters (`out_channels`)

- Start **small** → 8 or 16 filters in the first layer.
- It gradually doubles with each block (8 → 16 → 32 → 64).
- More filters = more details, but also more parameters → attention to the risk of *overfitting*.

Typical example:

- `Conv1: 1 → 8`
- `Conv2: 8 → 16`
- `Conv3: 16 → 32`

Golden rule: *more deep layers → more abstract features (shapes, organs, structures)*

Practical rules for setting up a CNN

2. Filter size (`kernel_size`)

- Most used standard: **3×3**
  → captures local details but keeps costs low.
- Sometimes 5×5 in the first layers for larger patterns (x-rays, macrostructures).
- Rarely >7×7: Becomes too heavy and loses local accuracy.

Synthesis:

3×3 for all layers → simple, fast, works almost all the time.

Practical rules for setting up a CNN

3. Pooling (MaxPool2d)

- Always use **2×2** with stride=2 to halve size.
- Alternate with convolutional blocks (after every 1–2 convolutions).
- Too much pooling → the network "loses" fine details.
  Too little → the network remains too large and slow.

Example:

- `[Conv → ReLU → Conv → Pool] × 2`

Practical rules for setting up a CNN

4. Activation function (`ReLU`)

- Always after each convolution.
- It has no hyperparameters → any complications.
- It only maintains positive activations → helps the network learn nonlinearities.

In few words:

`Conv` → `ReLU` is an inseparable pair.

Practical rules for setting up a CNN

5. Flatten e Fully Connected

- The `Flatten` is not set: it automatically takes the output of the last convolutional block.
- The first `Linear` must receive **all the values that come out of the Flatten**.
  → calculate it as: `num_filtri_finali × altezza_finale × larghezza_finale`
- The last `Linear` outputs **the number of classes** (`num_classes`).

Example:

```
Last Conv: (32, 7, 7)
Flatten: 32×7×7 = 1568
→ Linear(1568, num_classes)
```

Practical summary

Element	Typical choice	Motivation
Kernel	3×3	Scale Detail and Speed
Stride	1	No loss of information
Padding	1	Maintains the same size
Pooling	2×2	halves the size
Filters	8 → 16 → 32 → 64	Increases depth
ReLU	after each Conv	introduces nonlinearity

Start simple (few filters, 3×3, 2×2 pool)
→ if the network doesn't learn, add depth or filters.
→ if overfitted, add dropouts or reduce capacity.

Building a CNN

Basic example (MNIST 28×28, black/white, 2 classes):

```
class SimpleCNN(nn. Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nos. Conv2d(1, 8, kernel_size=3) # block 1
        self.pool = nos. MaxPool2d(2, 2) # half size
        self.fc1 = nos. Linear(8*13*13,2) #2 Output Classes

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        return x
```

Building a CNN

Why these numbers?

- `in_channels=1` → black and white image
- `out_channels=8` → 8 filters = 8 patterns that the network learns
- `kernel_size=3` → standard, capture small details
- `MaxPool2d(2,2)` → halved to reduce parameters
- `Linear(1352, 2)` → because after the Flatten we have $8 \times 13 \times 13 = 1352$ values

Understanding the Calculation of Dimensions

General formula of the convolution:

$$\text{Output} = \frac{(Input - Kernel + 2 \times Padding)}{Stride} + 1$$

In our case:

- Input = 28, Kernel = 3, Padding = 0, Stride = 1
- → (28 - 3 + 0)/1 + 1 = 26

Then pooling **halves**:

- 26 / 2 = 13

Final Dimensional Flow:

- (1,28,28) → (8,26,26) → (8,13,13) → (1352,) → (2,)

How CNN trains

The training is identical to that seen for the MLP.

Definition of loss and optimizer

```
criterion = nos. CrossEntropyLoss() # rating
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Training Cycle

```
for epoch in range(10):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoca {epoch+1}, Loss: {running_loss/len(train_loader):.4f}")
```

The network updates the **filter weights** to reduce loss.

Validation and testing

```
model.eval()
correct, total = 0, 0

with torch.no_grad():
    for images, labels in val_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = correct / total
print(f"Accuracy: {accuracy:.2f}")
```

Full CNN stream summary

Phase	What happens	PyTorch Items
1. Preprocessing	Image Transformation	`torchvision.transforms`
2. Dataset & Loader	Upload and batch	`torch.utils.data`
3. Model definition	CNN Structure	`nn. Module`
4. Training	Weight update	`optimizer.step()`
5. Validation	Accuracy calculation	`model.eval()`

Dataset → Transform → DataLoader → CNN → Loss → Optimizer → Validation

Practical conclusion

Now we know how to:

- Building a CNN from scratch in PyTorch
- Calculate and verify internal dimensions
- Choose the right parameters (`kernel, filters, pooling`)
- Train and evaluate the model

In summary:

CNNs learn *spatial patterns* from pixels,
combining local features into global representations.
They are the foundation of all modern Computer Vision.

From theory to practice

In **Google Colab** we open the notebook:

`ComputerVision_CNN.ipynb`

Objective:

Put into practice all the concepts seen so far:

- Image Upload and Preprocessing
- Step-by-step construction of a **simple CNN**
- Training, validation, and visualization of results

Conclusion Day 4

Today we have introduced the fundamental principles of **Computer Vision applied to medicine**:

- Representation of images as pixel tensors.
- Key concepts of CNNs (convolution, pooling, feature maps).
- Convenient workflow: preprocessing, training and evaluation.

On Day 5 we will see how the same principles extend to the **clinical text with Large Language Models (LLMs).**

See you tomorrow!