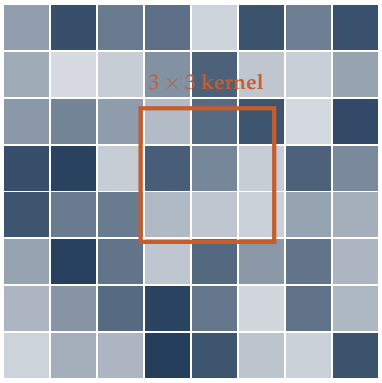
African Institute for Mathematical Sciences — Cameroon
Data Science Curriculum · Module 13

Computer Vision

From Pixels to Perception: Theory, Mathematics, and Practice



Course Notes for Deep Dive Study

Covering image representation, filtering, convolutional networks, generative augmentation, transfer learning, object detection, and segmentation

Compiled Course Companion · 2026

Preface

This companion volume is a deep-dive expansion of Module 13, *Computer Vision*, of the AIMS–Cameroon Data Science curriculum. Where the syllabus lists topics in bullet form — image representation, filtering, augmentation, classification, detection, and segmentation — this text unpacks each of them from first principles: what problem the technique solves, the mathematics that makes it work, the intuition behind the mathematics, and how it is used in practice, with a recurring emphasis on African application contexts such as crop-disease detection, land-use mapping, and medical imaging under limited-data conditions.

The chapters are written for a reader who has already met linear algebra, basic probability, and the fundamentals of supervised machine learning (as covered earlier in this curriculum), and who now wants to see how those tools specialise to the visual domain. Three boxed elements recur throughout:

- **Intuition** boxes translate an equation into a mental picture.
- **Key Formula** boxes isolate the one line worth memorising.
- **Worked Example** boxes carry a small numerical calculation through to completion by hand, so that the arithmetic behind "convolution" or "IoU" or "cross-entropy" is never a black box.

Each chapter closes with a short summary and a set of exercises intended to be solvable with pen, paper, and (where stated) a Python interpreter.

How to read this book. Chapters 1–3 build the classical image-processing vocabulary (pixels, filters, edges) that every deep model still relies on internally. Chapters 4–7 build the deep learning machinery (convolutional networks, architectures, augmentation, transfer learning). Chapters 8–10 apply that machinery to the three canonical computer vision tasks: detection, and segmentation (classification having been covered as the running example throughout 4–7). Chapter 11 grounds the material in African case studies, and Chapter 12 collects the practical OpenCV/PyTorch toolkit and an end-to-end project pipeline.

Contents

Preface i

1 Foundations of Digital Images 1
- 1.1 An Image as a Function, and as a Matrix . . . 1
  - 1.1.1 Bit depth and dynamic range . . . 1
  - 1.1.2 Spatial resolution . . . 2
- 1.2 Colour Images and Colour Spaces . . . 2
  - 1.2.1 Grayscale conversion . . . 2
  - 1.2.2 Other colour spaces: HSV . . . 3
- 1.3 Pixel Relationships and Connectivity . . . 3
- 1.4 Image Formats and Data Types . . . 4
- 1.5 Normalisation . . . 5

2 Image Preprocessing: Noise, Filtering, and Histogram Equalisation 7
- 2.1 Models of Image Noise . . . 7
- 2.2 Convolution and Correlation . . . 7
  - 2.2.1 The one-dimensional intuition . . . 7
  - 2.2.2 The two-dimensional operation . . . 8
  - 2.2.3 Padding, stride, and output size . . . 9
- 2.3 Linear Spatial Filters . . . 9
  - 2.3.1 Smoothing (low-pass) filters . . . 9
  - 2.3.2 Non-linear filters: the median filter . . . 10
- 2.4 Edge-Aware and Frequency Intuition . . . 10
- 2.5 Histogram Equalisation . . . 11
  - 2.5.1 Derivation . . . 11
  - 2.5.2 Adaptive and colour extensions . . . 12

3 Feature Extraction: Gradients, Edges, and Classical Descriptors 14
- 3.1 The Image Gradient . . . 14
  - 3.1.1 Sobel and Prewitt operators . . . 14
  - 3.1.2 The Laplacian: a second-derivative edge detector . . . 15
- 3.2 The Canny Edge Detector . . . 16
- 3.3 Corners and Interest Points . . . 16
- 3.4 From Hand-Crafted to Learned Features . . . 17

4 Convolutional Neural Network Foundations 19
- 4.1 From Fully-Connected to Convolutional Layers . . . 19
  - 4.1.1 The problem with fully-connected layers on images . . . 19
  - 4.1.2 The convolutional layer as a solution . . . 19
  - 4.1.3 Multiple channels and multiple filters . . . 20
- 4.2 Activation Functions . . . 20
- 4.3 Pooling and Receptive Field . . . 21

4.3.1 Pooling . . . 21
4.3.2 Receptive field . . . 22
4.4 Training a CNN: Loss, Gradients, and Backpropagation . . . 23
4.4.1 Loss functions . . . 23
4.4.2 Gradient descent and backpropagation, briefly revisited . . . 23
4.4.3 Batch normalisation and dropout . . . 23

5 Landmark CNN Architectures 25
5.1 LeNet-5: The Original Blueprint (1998) . . . 25
5.2 AlexNet: The 2012 Breakthrough . . . 25
5.3 VGG: Depth Through Simplicity . . . 25
5.4 ResNet: Solving the Degradation Problem . . . 26
5.4.1 The problem: deeper networks that perform worse . . . 26
5.4.2 The solution: residual (skip) connections . . . 26
5.5 EfficientNet: Principled Scaling . . . 27

6 Data Augmentation and Generative Adversarial Networks 30
6.1 Traditional Data Augmentation . . . 30
6.2 Generative Adversarial Networks . . . 31
6.2.1 The two-player game . . . 31
6.2.2 Practical training challenges . . . 32
6.3 Why This Matters for Limited-Data Domains . . . 32

7 Transfer Learning and Fine-Tuning 34
7.1 Why Pretrained Features Transfer . . . 34
7.1.1 What early and late layers learn . . . 34
7.2 Two Transfer Learning Strategies . . . 34
7.2.1 Feature extraction (frozen backbone) . . . 35
7.2.2 Fine-tuning (partially or fully unfrozen) . . . 35
7.3 Deciding Between the Two Strategies . . . 36
7.4 Practical Fine-Tuning Recipe . . . 36

8 Object Detection I: Two-Stage Detectors 38
8.1 Formalising the Problem . . . 38
8.1.1 Intersection over Union . . . 38
8.2 From Sliding Windows to Region Proposals . . . 39
8.2.1 The naive baseline: sliding windows . . . 39
8.2.2 Selective Search: smarter proposals . . . 39
8.3 The R-CNN Family . . . 39
8.3.1 R-CNN (2014) . . . 39
8.3.2 Fast R-CNN (2015): sharing computation . . . 39
8.3.3 Faster R-CNN (2015): learning the proposals too . . . 40
8.4 Non-Maximum Suppression . . . 41

9 Object Detection II: One-Stage Detectors 43
9.1 The One-Stage Idea . . . 43
9.2 The YOLO Output and Loss Function . . . 43
9.2.1 What each grid cell predicts . . . 44
9.2.2 The multi-part loss function . . . 44

9.3 Anchor-Based YOLO and Successive Refinements . . . 45
9.4 Evaluating Detectors: Mean Average Precision . . . 45

10 Image Segmentation **47**
10.1 Semantic versus Instance Segmentation . . . 47
10.2 Fully Convolutional Networks . . . 47
10.3 U-Net: Skip Connections for Precise Localisation . . . 48
10.4 Mask R-CNN: Instance Segmentation . . . 49
10.4.1 RoIAlign: fixing a quantisation problem . . . 49
10.5 Segmentation Loss Functions . . . 49
10.5.1 Pixel-wise cross-entropy . . . 49
10.5.2 Dice loss . . . 50

11 Case Studies: Computer Vision in African Contexts **52**
11.1 Crop Disease and Pest Detection . . . 52
11.1.1 The problem . . . 52
11.1.2 Why this is harder than a typical benchmark . . . 52
11.2 Land-Use Mapping from Satellite Imagery . . . 53
11.2.1 The problem . . . 53
11.2.2 Domain-specific considerations . . . 53
11.3 Medical Image Diagnosis Under Data Scarcity . . . 54
11.3.1 The problem . . . 54
11.3.2 Techniques that matter most in this setting . . . 54
11.4 A Common Thread . . . 55

12 The Practical Toolkit: OpenCV, PyTorch, and an End-to-End Pipeline **57**
12.1 OpenCV: The Classical Vision Workhorse . . . 57
12.2 PyTorch: Building and Training the Model . . . 58
12.3 The End-to-End Project Pipeline . . . 59

A Glossary **62**

Glossary **62**

B Further Reading **65**

Further Reading **65**

CHAPTER 1

Foundations of Digital Images

Every technique in this book, from a 3×3 blur to a hundred-layer neural network, operates on the same underlying object: a grid of numbers. Before we can process, augment, classify, or detect anything, we need a precise vocabulary for what an image *is*, mathematically. That is the purpose of this opening chapter.

1.1 An Image as a Function, and as a Matrix

A grayscale image can be modelled as a two-dimensional function

$$I : \Omega \subset \mathbb{R}^2 \longrightarrow \mathbb{R}, \qquad I(x, y) = \text{light intensity at spatial location } (x, y),$$

where Ω is the rectangular image domain. This continuous model is convenient for deriving calculus-based operators (gradients, Laplacians), but a digital camera never records I at every real-valued (x, y); it records it at a finite grid of locations, and it quantises the intensity to a finite set of levels. These two steps are called **sampling** and **quantisation**.

Intuition

Think of the continuous scene as a smooth painting and the digital sensor as a sheet of graph paper laid over it. *Sampling* decides how fine the graph paper's grid is (how many little squares, i.e. pixels). *Quantisation* decides how many shades of grey you are allowed to use to colour in each square. Coarser paper or fewer shades both throw away information — that lost information is exactly what separates a crisp photo from a blocky, banded one.

After sampling and quantisation, the image becomes a discrete function

$$I[m, n], \qquad m = 0, 1, \ldots, M-1, \quad n = 0, 1, \ldots, N-1,$$

i.e. an $M \times N$ matrix. Each entry $I[m, n]$ is a **pixel** ("picture element"). We will use $I[m, n]$ and $I(x, y)$ interchangeably once we move to the discrete setting, with m indexing rows (the y-direction, top to bottom) and n indexing columns (the x-direction, left to right).

1.1.1 Bit depth and dynamic range

The number of quantisation levels is fixed by the **bit depth** b: an b-bit image stores 2^b possible intensities per channel. The overwhelmingly common choice is $b = 8$, giving intensities in $\{0, 1, \ldots, 255\}$. Medical and scientific imaging often use $b = 12$ or $b = 16$ for a wider **dynamic range** (the ratio between the largest and smallest resolvable signal), which matters when, say, a chest X-ray must show both faint soft tissue and dense bone in the same frame.

Number of quantisation levels

$$L = 2^b, \qquad \text{so an 8-bit image has } L = 256 \text{ levels: } \{0, 1, \ldots, 255\}.$$

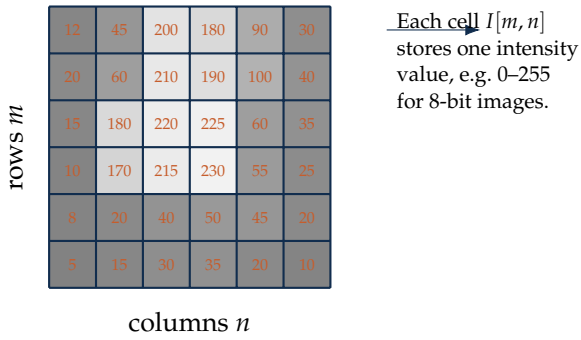


Figure 1.1: A digital image is a matrix of intensity values. Darker cells here correspond to lower pixel values (closer to 0, black); lighter cells correspond to higher values (closer to 255, white).

1.1.2 Spatial resolution

Spatial resolution is the pixel grid size $M \times N$ (commonly reported as width×height, e.g. 1920×1080). Higher resolution captures finer spatial detail but increases storage and compute cost roughly linearly in the pixel count $M \cdot N$, and quadratically in compute for many convolutional operations we will meet in Chapter 4. Resolution and bit depth together fix the raw storage size of an uncompressed image:

$$\text{Storage (bits)} = M \times N \times C \times b,$$

where C is the number of channels (1 for grayscale, 3 for colour).

1.2 Colour Images and Colour Spaces

A colour photograph is not one function but several, stacked together. The most common representation is **RGB**, in which every pixel is a triple

$$I(x, y) = \big(R(x, y),\, G(x, y),\, B(x, y)\big) \in \{0, \ldots, 255\}^3,$$

so the image is a tensor of shape $M \times N \times 3$. The three channels correspond to the response of red-, green-, and blue-sensitive receptors, mimicking the three cone types in the human retina.

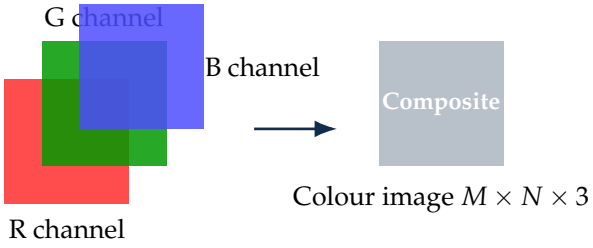


Figure 1.2: An RGB image stacks three single-channel intensity images (red, green, blue). Combining the three channels pixel-by-pixel produces the perceived colour.

1.2.1 Grayscale conversion

Many classical algorithms (edge detectors, corner detectors) operate on a single intensity channel. The standard perceptual conversion weights the channels by the human eye's differing sensitivity to red, green, and blue light:

Luminance-weighted grayscale conversion (ITU-R BT.601)

$$I_{\text{gray}}(x, y) = 0.299\, R(x, y) + 0.587\, G(x, y) + 0.114\, B(x, y).$$

Intuition

The human eye is far more sensitive to green light than to red or blue, because the green-sensitive cones in the retina are the most numerous and their response overlaps most with the eye's overall luminance sensitivity curve. A simple unweighted average $\frac{1}{3}(R + G + B)$ would make a pure blue patch look artificially bright and a pure green patch look artificially dim compared to how a person actually perceives them. The weighted sum corrects for this.

1.2.2 Other colour spaces: HSV

RGB mixes colour and brightness in a way that is inconvenient for many tasks (e.g. "find all red pixels" is fragile in RGB under changing lighting). The **HSV** space (Hue, Saturation, Value) separates *what colour* (hue, an angle on a colour wheel 0°–360°), *how pure* (saturation, 0 = gray to 1 = fully saturated), and *how bright* (value) a pixel is. Converting from RGB (normalised to $[0, 1]$):

$$V = \max(R, G, B), \qquad S = \begin{cases} 0 & V = 0 \\ \dfrac{V - \min(R, G, B)}{V} & \text{otherwise,} \end{cases}$$

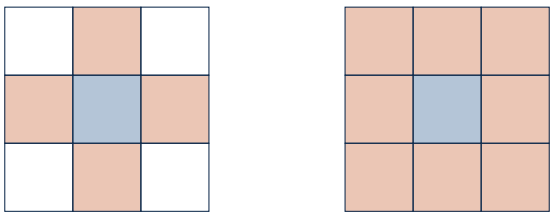
with H computed piecewise depending on which of R, G, B is the maximum (a standard but slightly lengthy formula omitted here for brevity). In practice, thresholding on the H channel in HSV is a far more lighting-robust way to isolate "green crop canopy" or "red ripe fruit" in agricultural imagery than thresholding on the R, G, B channels directly, because hue is largely invariant to changes in illumination intensity.

1.3 Pixel Relationships and Connectivity

Many algorithms — from flood fill to connected-component labelling used in counting objects (e.g. counting cells in a microscopy image, or counting buildings in a satellite tile) — need a formal notion of which pixels are "neighbours."

- **4-connectivity:** pixel (m, n)'s neighbours are $(m \pm 1, n)$ and $(m, n \pm 1)$ — the four pixels sharing an edge.
- **8-connectivity:** adds the four diagonal neighbours $(m \pm 1, n \pm 1)$.

Connectivity choice affects segmentation and labelling results: two diagonally touching foreground pixels are considered part of the same object under 8-connectivity but two *separate* objects under 4-connectivity. A related idea is **distance metrics** between pixels, most commonly the Euclidean distance $d_E\big((m_1, n_1), (m_2, n_2)\big) = \sqrt{(m_1 - m_2)^2 + (n_1 - n_2)^2}$, the city-block (Manhattan, 4-connected) distance $d_4 = |m_1 - m_2| + |n_1 - n_2|$, and the chessboard (8-connected) distance $d_8 = \max(|m_1 - m_2|, |n_1 - n_2|)$, which reappear later when we discuss morphological operations and non-maximum suppression neighbourhoods.



4-connectivity 8-connectivity

Figure 1.3: The center pixel (blue) and its neighbourhood under 4-connectivity (edge-sharing only) versus 8-connectivity (edge- and corner-sharing).

Format	Compression	Typical use
BMP	None	Raw pixel dumps, teaching examples
PNG	Lossless	Screenshots, masks, images needing exact pixel fidelity
JPEG	Lossy (DCT-based)	Photographs where small artefacts are acceptable
TIFF	Lossless / configurable	Scientific, medical, and satellite imagery
WebP	Lossy / lossless	Web delivery, smaller file size than PNG/JPEG

Table 1.1: Common raster image formats and where each is typically used.

1.4 Image Formats and Data Types

Practitioner's Note

JPEG's lossy compression discards high-frequency detail using a Discrete Cosine Transform followed by quantisation — convenient for storage, but dangerous as a source format for training data: repeated JPEG re-saves introduce blocking artefacts that a CNN can learn to key on, causing the model to perform worse on cleanly-encoded deployment images. When curating a dataset (e.g. satellite tiles or medical scans), keep a lossless master copy and only compress a working copy if storage forces you to.

In code, images are almost always loaded as NumPy arrays. Two libraries dominate: OpenCV (`cv2`), which loads colour images in **BGR** channel order for historical reasons, and Pillow/`PIL`, which uses the more intuitive **RGB** order. Mixing the two without converting is one of the most common silent bugs in computer vision pipelines — a model trained on RGB data fed BGR images at inference time will still run without error, but its colour-dependent features will be scrambled, silently degrading accuracy.

Listing 1.1: Loading an image and inspecting its shape and dtype.

```
import cv2
import numpy as np

img = cv2.imread("leaf.jpg")           # returns H x W x 3, BGR, dtype=uint8
print(img.shape, img.dtype)            # e.g. (720, 1280, 3) uint8

img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)   # convert to RGB
gray    = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # collapse to 1 channel
```

1.5 Normalisation

Before an image is passed to almost any learning algorithm, its pixel values are rescaled. The two most common schemes:

Min-max normalisation to the unit interval

$$\hat{I} = \frac{I - I_{\min}}{I_{\max} - I_{\min}}, \qquad \text{typically } I_{\min} = 0,\ I_{\max} = 255 \text{ for 8-bit images.}$$

Standardisation (z-score normalisation)

$$\hat{I} = \frac{I - \mu}{\sigma},$$

where μ, σ are the per-channel mean and standard deviation, often computed over a large reference dataset (for models pretrained on ImageNet, the canonical values are $\mu = (0.485, 0.456, 0.406)$ and $\sigma = (0.229, 0.224, 0.225)$ for the R, G, B channels respectively).

Intuition

Neural network training via gradient descent (Chapter 4) is sensitive to the scale of its inputs. If pixel values range over $[0, 255]$, the very first layer's weight updates are dominated by that scale, gradients can explode or vanish, and different input channels with different scales pull training in inconsistent directions. Standardising to roughly zero mean and unit variance puts every input feature on comparable footing, which in practice makes training faster and more stable — the same reason we standardise features before fitting a linear or logistic regression model.

Standardising a pixel

Suppose a red-channel pixel has raw 8-bit value $I = 180$. First rescale to $[0, 1]$: $I/255 = 0.7059$. Using the ImageNet red-channel statistics $\mu_R = 0.485$, $\sigma_R = 0.229$:

$$\hat{I} = \frac{0.7059 - 0.485}{0.229} = \frac{0.2209}{0.229} \approx 0.965.$$

This standardised value, not the raw 180, is what would actually be fed into a pretrained CNN's first convolutional layer.

Chapter Summary

- A digital image is a discrete matrix (grayscale) or tensor (colour) of intensity values, produced from a continuous scene by *sampling* (spatial grid) and *quantisation* (intensity levels, set by bit depth $b = 2^L$... i.e. $L = 2^b$ levels).
- Colour images stack channels (commonly RGB); grayscale conversion uses perceptually weighted coefficients $0.299R + 0.587G + 0.114B$.
- HSV separates hue/saturation/value and is often more robust to lighting changes than RGB for colour-based thresholding tasks.
- Pixel connectivity (4- vs 8-connected) determines what counts as "touching" for segmentation and labelling.

- Inputs are normalised (min-max or z-score standardisation) before being passed into learning algorithms, for numerical and optimisation reasons.

End-of-Chapter Exercises

1. An 8-bit RGB image has resolution 4000×3000. Compute its uncompressed storage size in megabytes.
2. A pixel has RGB values $(R, G, B) = (220, 40, 30)$. Compute its grayscale value using the ITU-R BT.601 weights.
3. Explain, in your own words, why HSV thresholding for "ripe red tomatoes" is more robust to outdoor lighting changes than RGB thresholding.
4. Two foreground pixels touch only at a corner. Under which connectivity rule(s) are they part of the same connected component?
5. Standardise a green-channel pixel with raw value 90, using $\mu_G = 0.456$, $\sigma_G = 0.224$.

CHAPTER 2

Image Preprocessing: Noise, Filtering, and Histogram Equalisation

Raw images are rarely fed directly into a downstream task. Sensor noise, poor contrast, and unwanted high-frequency detail all degrade performance, so a preprocessing stage cleans and normalises the image first. This chapter builds the single most important mathematical operation in classical (and, as we will see in Chapter 4, modern deep) computer vision: **convolution**.

2.1 Models of Image Noise

Noise is any unwanted variation in pixel intensity not present in the true underlying scene. Different physical processes produce different statistical noise models.

- **Gaussian noise.** Arises from thermal and electronic sensor noise. Each pixel is perturbed by an independent sample from a normal distribution:

$$I_{\text{noisy}}(x, y) = I(x, y) + \eta(x, y), \qquad \eta(x, y) \sim \mathcal{N}(0, \sigma^2).$$

- **Salt-and-pepper noise.** Individual pixels are randomly forced to the extreme values (0 or 255), e.g. from bit errors in transmission or dead sensor elements. A pixel is corrupted with probability p, and if corrupted, set to 0 or 255 with equal chance.
- **Poisson (shot) noise.** Arises from the discrete, random arrival of photons; the number of photons counted at a pixel is itself a Poisson random variable, so the noise variance scales with signal intensity (Var $= \lambda$, the mean photon count). This matters especially in low-light and medical imaging (e.g. X-ray, PET).

Intuition

Gaussian noise looks like a fine, even "grain" over the whole image, because every pixel gets a small independent nudge. Salt-and-pepper noise looks like scattered pure-white and pure-black dots, because a small fraction of pixels are catastrophically wrong while the rest are untouched. Because the two noise types have different statistical "signatures," the best filter for one is often a poor choice for the other, as we will see below.

2.2 Convolution and Correlation

2.2.1 The one-dimensional intuition

Before the 2D case, consider smoothing a noisy 1D signal $f[n]$ by averaging each sample with its neighbours: $g[n] = \frac{1}{3}\big(f[n-1] + f[n] + f[n+1]\big)$. This is a special case of a general weighted local combination

$$g[n] = \sum_{k=-a}^{a} h[k]\, f[n-k],$$

where $h[k]$ is a small **kernel** (or filter) of weights. This operation is **convolution**, written $g = f * h$.

2.2.2 The two-dimensional operation

For images, the kernel is a small 2D matrix h (commonly 3×3 or 5×5), and convolution slides it over every position of the image, at each position computing a weighted sum of the pixels it currently overlaps:

2D discrete convolution

$$g(x,y) = (I * h)(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} h(i,j)\, I(x-i,\, y-j).$$

Strictly, true convolution flips the kernel ($I(x-i, y-j)$) before sliding, whereas **cross-correlation** does not flip it:

$$g(x,y) = (I \star h)(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} h(i,j)\, I(x+i,\, y+j).$$

For symmetric kernels (Gaussian, box blur) the flip makes no difference. For asymmetric kernels (Sobel, learned CNN filters in Chapter 4) it does mathematically, but in practice deep learning frameworks implement cross-correlation and simply call it "convolution" by convention, since the network learns the kernel weights anyway and is indifferent to whether they are pre-flipped.

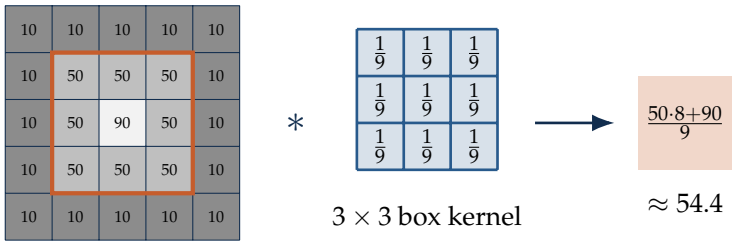


Figure 2.1: Convolution slides a kernel over the image. At each position, the overlapping pixel values are multiplied element-wise by the kernel weights and summed to produce one output pixel; the kernel then moves (by the *stride*) to the next position.

Convolving a 3×3 patch

Take the highlighted patch from Figure 2.1:

$$\begin{bmatrix} 50 & 50 & 50 \\ 50 & 90 & 50 \\ 50 & 50 & 50 \end{bmatrix}$$

convolved with the normalised box kernel $h = \frac{1}{9}\mathbf{1}_{3\times3}$. The output pixel is the average of all nine values:

$$g = \frac{8(50) + 90}{9} = \frac{400 + 90}{9} = \frac{490}{9} \approx 54.4.$$

The centre pixel (90, a bright spot) has been pulled down toward its neighbours' average — this is exactly what a blur does: it smooths a locally extreme value toward the local mean.

2.2.3 Padding, stride, and output size

Two choices govern how a kernel is applied at the image boundary and how far it moves between applications:

- **Padding** p: extra border pixels (commonly zeros, or reflected/replicated values) added around the image so the kernel can be centred on edge and corner pixels too. Without padding ("valid" convolution), the output shrinks; with padding chosen so the output matches the input size, this is called "same" padding.
- **Stride** s: the step size the kernel moves between applications. Stride 1 moves one pixel at a time (dense output); stride 2 skips every other position, halving each output dimension and reducing compute.

Output spatial size of a convolution

$$O = \left\lfloor \frac{N + 2p - K}{s} \right\rfloor + 1,$$

where N is the input size (per dimension), K is the kernel size, p the padding, and s the stride. This formula is applied independently to height and width and will resurface constantly in Chapter 4 when we design CNN layers.

Output size

A 32×32 input, a 5×5 kernel, padding $p = 2$, stride $s = 1$:

$$O = \left\lfloor \frac{32 + 4 - 5}{1} \right\rfloor + 1 = \lfloor 31 \rfloor + 1 = 32.$$

Padding of 2 with a 5×5 kernel exactly preserves the spatial size — this is the "same" padding case, and in general same padding for an odd kernel size K requires $p = (K-1)/2$.

2.3 Linear Spatial Filters

2.3.1 Smoothing (low-pass) filters

Box (mean) filter. Every kernel weight equals $1/K^2$; replaces each pixel with the local average. Cheap, but blurs edges indiscriminately and does not model the physical spread of real optical/sensor blur well.

Gaussian filter. Weights follow a 2D Gaussian bump, giving nearby pixels more influence than far pixels:

2D Gaussian kernel

$$h(i, j) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{i^2 + j^2}{2\sigma^2} \right),$$

normalised so the discretised kernel weights sum to 1. The single parameter σ controls the amount of blur: larger σ smooths over a wider neighbourhood.

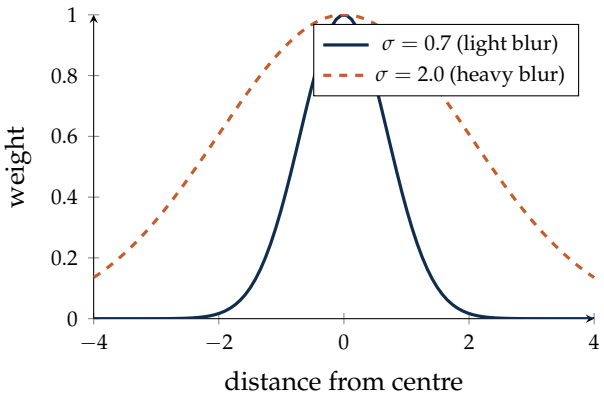


Figure 2.2: Cross-section of a 1D Gaussian weighting profile for two values of σ. A larger σ spreads weight over a wider neighbourhood, producing stronger blur.

Intuition

The Gaussian filter is the smoothing filter that treats "nearness" the most gently: a pixel two steps away contributes noticeably less than a pixel one step away, rather than the box filter's abrupt all-or-nothing cutoff. This is also why the Gaussian is the natural noise-suppressor for *Gaussian* noise: averaging many independent Gaussian perturbations with smoothly decaying weights reduces variance while introducing the least "ringing" or blocky artefacts.

2.3.2 Non-linear filters: the median filter

The **median filter** replaces each pixel by the *median* (not mean) of its neighbourhood. Because it is not a weighted sum, it cannot be written as a convolution — it is non-linear.

Practitioner's Note

The median filter is the standard choice against *salt-and-pepper* noise. A single corrupted pixel forced to value 255 inside an otherwise smooth 3×3 patch is an outlier: the mean (as computed by a box or Gaussian filter) is dragged noticeably toward 255, whereas the median simply ignores it, since it is only one of nine values and does not occupy the middle rank. This is the same reason the median income is reported instead of the mean in the presence of extreme outliers.

2.4 Edge-Aware and Frequency Intuition

Every image can be thought of as a mixture of frequency components: slowly varying regions (sky, skin, walls) are **low frequency**; sharp boundaries, fine textures, and noise are **high frequency**. Smoothing filters like Gaussian and box blur are **low-pass filters** — they preserve slow variation and suppress high-frequency content, which is why they blur edges as an unavoidable side effect of removing noise. Chapter 3 introduces filters designed to do the opposite: preserve or amplify high-frequency edge information.

2.5 Histogram Equalisation

An image's **histogram** $h(r_k) = n_k$ counts how many pixels take each intensity value r_k, $k = 0, \ldots, L-1$. A photograph taken in poor lighting often has a histogram concentrated in a narrow intensity band (e.g. everything is dark and low-contrast), even though the sensor's full 0–255 range is available. **Histogram equalisation** redistributes intensities to use the full range more uniformly, improving perceptual contrast.

2.5.1 Derivation

Model pixel intensities as a continuous random variable $r \in [0, L-1]$ with probability density $p_r(r) = n_r/n$ (fraction of pixels at intensity r, with n the total pixel count). We seek a monotonic transformation $s = T(r)$ such that the *output* intensities s are uniformly distributed. A standard result from probability theory is that applying the **cumulative distribution function (CDF)** of a random variable to itself yields a uniform distribution. Hence:

Histogram equalisation transform

$$s = T(r) = (L-1)\int_0^r p_r(w)\,dw \qquad \Longrightarrow \qquad \text{(discrete form)} \qquad s_k = (L-1)\sum_{j=0}^{k}\frac{n_j}{n}.$$

Intuition

Picture the histogram as a crowd of people (pixels) standing at various heights (intensities) along a narrow section of a number line, most of them bunched close together. The CDF transform is like telling each person to move to a new position determined by "what fraction of the crowd is at or below your current height." People in the crowded regions get spread apart (their density is high, so the CDF rises quickly there and reassigns them to a wide range of new positions), while people in the sparse regions barely move. The net effect is a crowd that is now spread roughly evenly along the whole line — higher contrast, because tonal differences that used to be perceptually invisible (all crammed into a narrow input range) now occupy a visibly wider output range.

Equalising a tiny 8-level image

Suppose intensities range over $L = 8$ levels (0–7) and a small image has 64 pixels with the following counts: $n_0 = 8, n_1 = 6, n_2 = 10, n_3 = 0, n_4 = 14, n_5 = 2, n_6 = 18, n_7 = 6$ (sums to 64). The normalised histogram $p_k = n_k/64$ and cumulative sums:

k	0	1	2	3	4	5	6	7
n_k	8	6	10	0	14	2	18	6
CDF (cum. sum /64)	0.125	0.219	0.375	0.375	0.594	0.625	0.906	1.000
$s_k = 7 \times$ CDF	0.9	1.5	2.6	2.6	4.2	4.4	6.3	7.0
rounded	1	2	3	3	4	4	6	7

Level 6, which held 18 of the 64 pixels (the single largest group), is mapped to output level 6 — but notice how the transform has pulled apart levels that were close together in the crowded lower-middle range (0,1,2 mapping to 1,2,3) more than in the sparse tail. The image now spans the full 0–7 range rather than clustering, increasing contrast.

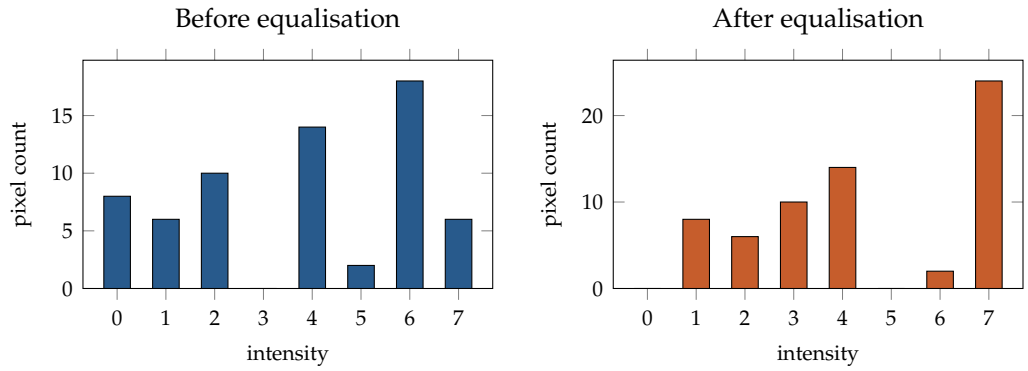


Figure 2.3: Histogram equalisation redistributes intensity counts so the cumulative distribution approaches a straight line (uniform density), spreading out the most crowded intensity bands.

2.5.2 Adaptive and colour extensions

Global equalisation can over-amplify noise in flat regions and under-serve local contrast in images with very different lighting across regions (e.g. a satellite tile spanning both bright bare soil and shadowed forest canopy). **CLAHE** (Contrast-Limited Adaptive Histogram Equalisation) applies equalisation independently on small tiles of the image, with a clip limit that caps how much any single intensity bin can be amplified, then blends tile boundaries smoothly. For colour images, equalising each of R, G, B independently distorts hue (since the three channels are stretched by different amounts); the standard practice is to convert to a space that separates luminance from colour (e.g. HSV or YCbCr), equalise only the intensity/luminance channel, and convert back.

Listing 2.1: Histogram equalisation and CLAHE in OpenCV.

```
import cv2

gray = cv2.imread("scan.png", cv2.IMREAD_GRAYSCALE)
eq = cv2.equalizeHist(gray)                          # global equalisation

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
eq_adaptive = clahe.apply(gray)                      # local, contrast-limited
```

Chapter Summary

- Noise is modelled by physically motivated distributions: Gaussian (sensor/thermal), salt-and-pepper (transmission errors, dead pixels), Poisson (photon shot noise).
- Convolution/correlation slides a small kernel over the image, producing a weighted local combination at each position; padding and stride together determine the output size via $O = \lfloor (N + 2p - K)/s \rfloor + 1$.
- Gaussian and box filters are linear low-pass (smoothing) filters; the median filter is a robust, non-linear alternative that is far better suited to impulsive (salt-and-pepper) noise.
- Histogram equalisation applies the cumulative distribution function of the intensity histogram as a contrast-stretching transform, spreading crowded intensity bands over a wider output range; CLAHE applies this locally with a clip limit to avoid

noise amplification.

End-of-Chapter Exercises

1. Derive the output size for a 224×224 input, kernel size 7, stride 2, padding 3.
2. Explain mathematically why a 3×3 mean filter cannot fully remove salt-and-pepper noise with corruption probability $p = 0.3$, while a 3×3 median filter can remove most of it.
3. Given kernel $h = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ (a discrete Laplacian), is this a low-pass or high-pass filter? Justify using the frequency intuition of Section 2.5.
4. A 6×6-pixel image has all pixels equal to intensity 128 except a single pixel at 255. Compute the effect of a 3×3 box filter versus a 3×3 median filter centred on that pixel (assume its neighbours are all 128).
5. Why must colour histogram equalisation be performed on a luminance channel rather than independently on R, G, and B?

CHAPTER 3

Feature Extraction: Gradients, Edges, and Classical Descriptors

Chapter 2's filters smoothed images by suppressing high-frequency content. This chapter goes the other way: we build filters that *highlight* the high-frequency content that corresponds to meaningful structure — edges, corners, and textures — because these structures are the building blocks that every downstream task (classification, detection, segmentation) ultimately keys on, whether extracted by hand-designed filters or, as in Chapter 4 onward, learned automatically.

3.1 The Image Gradient

An **edge** is a location where intensity changes sharply. The natural mathematical tool for measuring the rate of change of a function is its derivative; for the 2D function $I(x, y)$, this is the **gradient**, a vector of the two partial derivatives:

$$\nabla I = \begin{bmatrix} \dfrac{\partial I}{\partial x} \\ \dfrac{\partial I}{\partial y} \end{bmatrix}.$$

Gradient magnitude and direction

$$\|\nabla I\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}, \qquad \theta = \arctan\left(\frac{\partial I/\partial y}{\partial I/\partial x}\right).$$

The magnitude tells us *how strong* an edge is at a pixel; the direction tells us the orientation of greatest intensity change (perpendicular to the edge itself).

Since images are discrete, partial derivatives are approximated by finite differences, which themselves can be written as small convolution kernels.

3.1.1 Sobel and Prewitt operators

The **Sobel operator** approximates $\partial I/\partial x$ and $\partial I/\partial y$ using 3×3 kernels that combine a derivative in one direction with a smoothing average in the perpendicular direction (making the estimate more robust to noise than a naive one-pixel difference):

Sobel kernels

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}.$$

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

G_x (vertical edges)

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

G_y (horizontal edges)

Figure 3.1: The two Sobel kernels. G_x responds strongly where intensity changes left-to-right (i.e. it detects vertical edges); G_y responds where intensity changes top-to-bottom (horizontal edges). Blue cells are positive weights, orange cells negative.

Applying G_x

Consider a 3×3 patch straddling a vertical dark-to-light boundary:

$$\begin{bmatrix} 20 & 20 & 200 \\ 20 & 20 & 200 \\ 20 & 20 & 200 \end{bmatrix}.$$

Convolving (cross-correlating) with G_x:

$$G_x * \text{patch} = -1(20) + 0(20) + 1(200) - 2(20) + 0(20) + 2(200) - 1(20) + 0(20) + 1(200)$$

$$= (180) + (360) + (180) = 720.$$

A large positive response confirms a strong vertical edge (a left-to-right brightness jump) at this location; applying G_y to the same patch gives 0, since intensity does not change top-to-bottom here.

3.1.2 The Laplacian: a second-derivative edge detector

Where Sobel measures the first derivative (rate of change), the **Laplacian** measures the second derivative (rate of change *of the rate of change*):

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \approx \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * I.$$

Intuition

A first derivative peaks *at* an edge, producing a thick ridge of high-magnitude response that must later be thinned. A second derivative crosses zero *exactly at* the edge location (a "zero-crossing"), because intensity rises before the edge, peaks in rate-of-change at the edge, and the rate of increase itself starts falling immediately after — like the difference between velocity (first derivative) peaking mid-slope and acceleration (second derivative) crossing zero exactly at the inflection point. This gives the Laplacian sharper edge localisation, at the cost of much higher sensitivity to noise, since second derivatives amplify high-frequency noise more aggressively than first derivatives. In practice the Laplacian is almost always applied after a Gaussian blur (the combined "Laplacian of Gaussian," LoG) to control this noise sensitivity.

3.2 The Canny Edge Detector

Sobel and the Laplacian each produce a raw response map; turning that into a clean, one-pixel-wide, well-connected edge map requires more machinery. The **Canny edge detector** (1986) remains the classical gold standard and proceeds in five stages.



Figure 3.2: The five stages of the Canny edge detection pipeline.

1. **Gaussian smoothing.** Reduce noise before differentiating (recall: differentiation amplifies noise).
2. **Gradient computation.** Apply Sobel to obtain magnitude $\|\nabla I\|$ and direction θ at every pixel.
3. **Non-maximum suppression.** At each pixel, check the two neighbours along the gradient direction θ; keep the pixel only if its magnitude is a local maximum along that direction, else suppress it to 0. This thins the thick ridges from step 2 down to single-pixel-wide edges.
4. **Double thresholding.** Classify surviving pixels as *strong* (magnitude $> T_{\text{high}}$), *weak* ($T_{\text{low}} <$ magnitude $\leq T_{\text{high}}$), or *suppressed* (magnitude $\leq T_{\text{low}}$).
5. **Hysteresis.** Keep every strong edge. Keep a weak edge *only if* it is connected (via 8-connectivity) to a strong edge, directly or through a chain of other weak edges; discard isolated weak edges.

Intuition

Hysteresis exists because a single global threshold is a bad compromise: set it high and you lose faint-but-real edge segments (e.g. a low-contrast boundary in a hazy satellite image); set it low and you flood the output with noise-driven false edges. Using two thresholds lets Canny be strict about *starting* an edge (only strong responses qualify) but lenient about *continuing* one already started (a weak response is trusted if it is part of a chain anchored by strong evidence) — much like a metal detector operator who only investigates faint beeps if they occur right next to a location that already gave a strong, confirmed hit.

Listing 3.1: Canny edge detection in OpenCV.

```
import cv2
gray = cv2.imread("field.jpg", cv2.IMREAD_GRAYSCALE)
blurred = cv2.GaussianBlur(gray, (5,5), sigmaX=1.4)
edges = cv2.Canny(blurred, threshold1=50, threshold2=150)
```

3.3 Corners and Interest Points

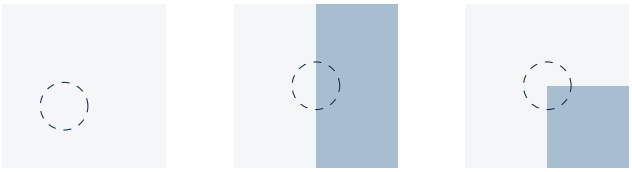
Edges alone are ambiguous: a point on a straight edge looks locally identical to every other point along that edge (this is the **aperture problem** — you cannot tell how far along the edge you have moved by looking through a small window). **Corners**, where two edges meet, are far more distinctive and repeatable, which is why classical matching and tracking algorithms (and, historically, camera calibration and panorama stitching) are built on corner detectors.

The **Harris corner detector** formalises "cornerness" by asking: if we shift a small window by (u, v) in *any* direction, does the intensity inside change a lot? Define

$$E(u,v) = \sum_{(x,y)\in W} \big[I(x+u, y+v) - I(x,y)\big]^2 \;\approx\; \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}, \qquad M = \sum_{W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix},$$

where I_x, I_y are the Sobel gradients and M is the local **structure tensor**. The eigenvalues λ_1, λ_2 of M summarise how E changes with shift direction:

- Both eigenvalues small ⇒ flat region (no change in any direction) ⇒ not a feature.
- One eigenvalue large, one small ⇒ an edge (change only perpendicular to the edge).
- Both eigenvalues large ⇒ a corner (change in every direction) ⇒ a good interest point.



flat: both λ small edge: $\lambda_1 \gg \lambda_2 \approx 0$ corner: both λ large

Figure 3.3: The Harris "cornerness" test: slide a small window (dashed circle) in every direction and check whether the intensity underneath changes. Flat regions and edges each fail to change in at least one direction; only corners change in all directions.

In practice, computing eigenvalues at every pixel is avoided by using the Harris response score $R = \det(M) - k\,(\mathrm{trace}(M))^2$ (with $k \approx 0.04$–0.06), which is large and positive only at corners, and can be thresholded and locally maximised directly.

Practitioner's Note

Classical descriptors such as SIFT and ORB build on this corner-detection idea by additionally encoding a small patch of gradient orientations around each detected keypoint into a fixed-length vector, so that the same physical corner can be re-identified ("matched") across two different photos of a scene taken from different viewpoints or lighting conditions. These hand-engineered descriptors were the state of the art for image matching, panorama stitching, and 3D reconstruction (structure from motion) for roughly two decades, and remain useful today whenever no labelled training data exists to learn features instead — for example, matching aerial photographs against archival maps with no annotated dataset to train on.

3.4 From Hand-Crafted to Learned Features

Every operator in this chapter — Sobel, Laplacian, Harris — shares one property: a human designed the kernel weights in advance, based on a mathematical model of what an edge or corner "should" look like. This works well for generic structure but cannot, by itself, tell a healthy cassava leaf from a diseased one, or a pedestrian from a lamppost: those distinctions require patterns too complex and too task-specific to hand engineer. Chapter 4 introduces **convolutional neural networks**, which keep exactly the same mathematical operation — convolution with a small kernel, sliding across the image — but *learn* the kernel weights from labelled data rather

than designing them by hand, and stack many such learned filters in sequence to build up increasingly abstract, task-specific features.

Chapter Summary

- Edges are locations of sharp intensity change, measured by the image gradient ∇I; the Sobel operator approximates the two partial derivatives with noise-robust 3×3 kernels.
- The Laplacian (second derivative) localises edges at zero-crossings but is noisier; it is normally paired with Gaussian smoothing (LoG).
- Canny edge detection chains Gaussian smoothing, gradient computation, non-maximum suppression, double thresholding, and hysteresis to produce clean, thin, well-connected edge maps.
- Corners are more distinctive than edge points because intensity changes in every shift direction, not just one; the Harris detector formalises this via the eigenvalues of a local structure tensor.
- All of these are *hand-designed* filters; CNNs (next chapter) replace hand design with learned filters while reusing the same convolution operation.

End-of-Chapter Exercises

1. Apply G_y (Sobel) to the patch in the Worked Example above and confirm it evaluates to 0.
2. Explain why the Laplacian operator is rarely applied to a raw, unsmoothed image in practice.
3. In the Canny pipeline, what would you expect to happen to the output if T_{low} were set to 0? What if $T_{\text{high}} = T_{\text{low}}$?
4. Sketch (by reasoning, no need to compute) the eigenvalues of the Harris structure tensor M at the centre of a uniformly coloured region, at the middle of a long straight edge, and at the tip of a checkerboard corner.
5. Why can a hand-designed Sobel kernel not, by itself, distinguish a healthy leaf from a diseased one, even though both contain plenty of edges?

CHAPTER 4

Convolutional Neural Network Foundations

Chapter 3 ended on a deliberate cliffhanger: hand-designed filters like Sobel and Harris can find edges and corners, but they cannot, by themselves, know what a *diseased cassava leaf* or a *malignant lesion* looks like. This chapter builds the machinery that lets a model *learn* its own filters from labelled examples: the **convolutional neural network** (CNN).

4.1 From Fully-Connected to Convolutional Layers

4.1.1 The problem with fully-connected layers on images

A standard ("fully-connected" or "dense") neural network layer connects every input unit to every output unit, with an independent learnable weight for each connection. Applied naively to a modest $224 \times 224 \times 3$ colour image (150,528 input values), a single fully-connected layer with just 1000 output units already needs over 150 million weights. This is problematic for three reasons:

1. **Parameter explosion.** That many weights invites overfitting and demands prohibitive memory and data.
2. **No spatial structure.** A dense layer treats the input as an unordered list of numbers; it has no built-in notion that pixel $(10, 10)$ is close to pixel $(10, 11)$ and far from pixel $(200, 200)$. Useful visual patterns are inherently *local* (an eye, a wheel, a leaf vein), and a dense layer must painstakingly relearn this locality from scratch, wastefully, for every possible position.
3. **No translation invariance.** If the network learns to detect a cat's ear in the top-left of the image, that knowledge does not automatically transfer to detecting the same ear in the bottom-right; a dense layer has entirely separate weights for each position.

4.1.2 The convolutional layer as a solution

A **convolutional layer** replaces the dense connection with the local, weight-shared operation from Chapter 2: a small kernel slides across the image, and *the same kernel weights are reused at every spatial position*. This directly fixes all three problems: the kernel only has $K \times K \times C_{\text{in}}$ weights (tiny compared to a dense layer), it only looks at a local neighbourhood, and because the same weights are applied everywhere, a pattern learned in one position is automatically detected everywhere else in the image.

Intuition

Think of a convolutional filter as a small rubber stamp with a learned pattern printed on it (e.g. "diagonal edge" or, in a deeper layer, "leaf-vein texture"). Convolution presses this

same stamp at every location in the image and records how well the stamp matches at each spot. Because it is literally the *same* stamp everywhere, a shape the stamp recognises in the top-left corner will be recognised identically if it appears in the bottom-right — this built-in property is called **translation equivariance**, and it is the central architectural idea that makes CNNs so much more data-efficient than dense networks for images.

4.1.3 Multiple channels and multiple filters

A real convolutional layer processes multi-channel input (e.g. 3 colour channels, or many feature-map channels deeper in the network) and produces multi-channel output by learning several independent filters simultaneously. If the input has C_{in} channels and the layer has C_{out} filters, each filter has shape $K \times K \times C_{\text{in}}$ (it spans the *full* depth of the input, but only a small $K \times K$ spatial window), and the layer output has C_{out} channels — one per filter.

Parameter count of a convolutional layer

$$\#\text{params} = \underbrace{(K \times K \times C_{\text{in}} + 1)}_{\text{weights + bias, per filter}} \times C_{\text{out}}.$$

Counting parameters

A layer with 3×3 kernels, $C_{\text{in}} = 64$ input channels, and $C_{\text{out}} = 128$ output filters:

$$\#\text{params} = (3 \times 3 \times 64 + 1) \times 128 = (576 + 1) \times 128 = 577 \times 128 = 73{,}856.$$

Compare this to a dense layer connecting a modest $56 \times 56 \times 64$ feature map (200,704 values) to 128 outputs: $200{,}704 \times 128 \approx 25.7$ million weights — roughly 350 times more parameters for the dense alternative, illustrating why convolution scales so much better to realistic image sizes.

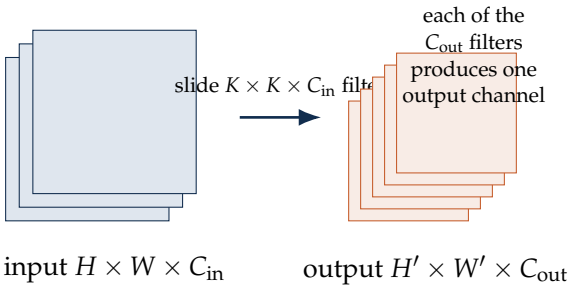


Figure 4.1: A convolutional layer maps an input volume of depth C_{in} to an output volume of depth C_{out}, by learning C_{out} independent filters, each spanning the full input depth but only a small $K \times K$ spatial footprint.

4.2 Activation Functions

A convolution (or dense layer) computes only a weighted sum — a linear operation. Stacking linear operations produces another linear operation, no matter how many layers are used, so

without something non-linear in between, a deep network would collapse to the expressive power of a single linear layer. **Activation functions** inject the needed non-linearity.

Common activation functions

$$\text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \text{ReLU: } f(z) = \max(0, z), \qquad \text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

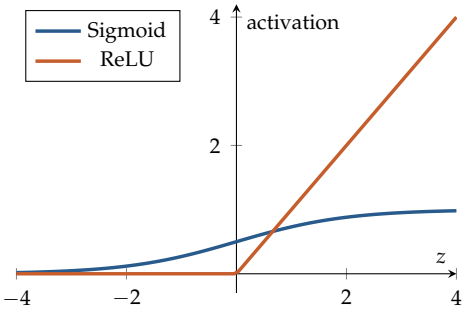


Figure 4.2: Sigmoid squashes any input into $(0, 1)$ but saturates (flattens, gradient $\to 0$) for large $|z|$. ReLU is unbounded above, computationally trivial, and only saturates on the negative side.

Intuition

ReLU's dominance in modern CNNs comes down to the **vanishing gradient problem**. Training uses backpropagation (Section 4.4), which repeatedly multiplies gradients through each layer, including the derivative of the activation function. Sigmoid's derivative is at most 0.25 and decays toward 0 for large $|z|$, so in a deep network these small factors multiply together across many layers and shrink toward zero, leaving early layers with almost no learning signal. ReLU's derivative is exactly 1 for any positive input, so a chain of ReLU layers can pass gradient backward largely undiminished, which is a major reason networks with dozens or hundreds of layers (Chapter 5) became trainable at all. Softmax is reserved for the final layer of a classifier, converting raw scores ("logits") into a probability distribution over classes that sums to 1.

4.3 Pooling and Receptive Field

4.3.1 Pooling

A **pooling** layer reduces the spatial size of a feature map by summarising each local neighbourhood with a single statistic — most commonly the maximum (**max pooling**) or the average (**average pooling**):

$$\text{MaxPool}(x)_{i,j} = \max_{(p,q)\in \text{window}(i,j)} x_{p,q}.$$

Pooling has no learnable parameters; it simply applies a fixed reduction. A 2×2 max pool with stride 2 halves both spatial dimensions.

Intuition

Max pooling keeps the strongest activation in each small neighbourhood and discards the rest, which achieves two things simultaneously: it reduces computation for all subsequent layers (fewer spatial positions to process), and it introduces a small amount of translation *invariance* — if a detected feature (e.g. an edge) shifts by one pixel within the pooling window, the max-pooled output is often unchanged, since the same maximum value is still likely to be captured. This complements the translation *equivariance* of convolution: convolution says "the same pattern is detected wherever it appears," and pooling says "I don't much care exactly where within this small window it appeared."

4.3.2 Receptive field

The **receptive field** of a unit in a given layer is the region of the *original input image* that can influence that unit's value. A single unit in the first convolutional layer (kernel size K_1) has a receptive field of exactly $K_1 \times K_1$ input pixels. As layers stack, receptive fields grow, because each unit in layer 2 depends on a $K_2 \times K_2$ patch of layer 1's units, each of which already depends on a $K_1 \times K_1$ patch of the input.

Receptive field growth across stacked layers

$$r_\ell = r_{\ell-1} + (K_\ell - 1) \prod_{i<\ell} s_i,$$

where r_ℓ is the receptive field size after layer ℓ, K_ℓ is that layer's kernel size, and s_i is the stride of layer i. With stride 1 throughout, this simplifies to $r_\ell = r_{\ell-1} + K_\ell - 1$, i.e. receptive field grows additively with each layer.

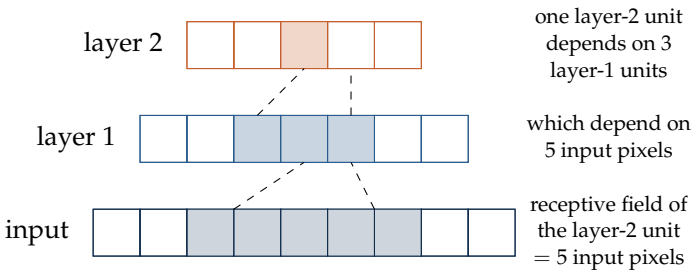


Figure 4.3: Stacking two 3×3 convolutions (stride 1) grows the receptive field of a single output unit from 3×3 (one layer) to 5×5 (two layers) — one reason deep networks with small kernels can still capture large-scale context.

Practitioner's Note

This is also why two stacked 3×3 convolutions are often preferred over one 5×5 convolution: both give a 5×5 receptive field, but two 3×3 layers have fewer parameters ($2 \times (9C^2) = 18C^2$ versus $25C^2$ for a single 5×5 layer, for C channels in and out) *and* include an extra non-linear activation in between, giving the network more representational flexibility for the same or lower cost. This insight, popularised by the VGG architecture (Chapter 5), is now standard practice.

4.4 Training a CNN: Loss, Gradients, and Backpropagation

4.4.1 Loss functions

Training requires a scalar measure of how wrong the network's current predictions are. For multi-class classification (e.g. "healthy" vs. "diseased" vs. "pest-damaged" leaf), the standard choice is **categorical cross-entropy**:

Cross-entropy loss

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c),$$

where $y_c \in \{0, 1\}$ is the true one-hot label (1 for the correct class, 0 otherwise) and $\hat{y}_c$ is the model's predicted probability for class c (from a softmax output).

Intuition

Because y_c is zero for every class except the true one, the sum collapses to a single term: $\mathcal{L} = -\log(\hat{y}_{\text{true class}})$. If the model assigns high probability (close to 1) to the correct class, $-\log(\hat{y})$ is close to 0 (low loss, as desired). If the model is confidently *wrong*, assigning the correct class a probability near 0, then $-\log(\hat{y}) \to \infty$ — the loss punishes confident mistakes far more severely than timid ones, which pushes the model not just toward correct answers but toward well-calibrated confidence.

4.4.2 Gradient descent and backpropagation, briefly revisited

Training minimises the loss by repeatedly nudging every weight in the network in the direction that most decreases the loss, using the gradient of the loss with respect to that weight:

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w},$$

where η is the learning rate (covered in depth, alongside momentum-based variants like Adam, in the curriculum's Optimisation Theory module). **Backpropagation** is simply the repeated application of the calculus chain rule to compute $\partial \mathcal{L}/\partial w$ efficiently for *every* weight in the network in a single backward pass, by propagating an error signal from the output layer back toward the input.

Intuition

For a convolutional layer, backpropagation has an elegant property: the gradient of the loss with respect to the layer's input turns out to be computable as a convolution of the upstream gradient with a flipped version of the same kernel, and the gradient with respect to the kernel weights is itself a convolution of the layer's input with the upstream gradient. In other words, the machinery needed to *train* a convolutional layer reuses the same convolution operation used in the *forward* pass — which is one reason convolutional networks were computationally tractable on the hardware of the 1990s (LeNet) long before the deep architectures of the 2010s.

4.4.3 Batch normalisation and dropout

Two regularisation and stabilisation techniques appear in nearly every modern CNN:

- **Batch normalisation.** Normalises the activations of a layer (across a mini-batch) to zero mean and unit variance, then applies a learned scale γ and shift β: $\hat{x} = \dfrac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$, output $= \gamma\hat{x} + \beta$. This keeps activation distributions stable as they flow through many layers, which allows much higher learning rates and faster, more stable convergence.
- **Dropout.** During training, randomly sets a fraction p of a layer's activations to zero at each forward pass, forcing the network to not over-rely on any single unit. At test time, dropout is disabled and activations are scaled to compensate. This is a computationally cheap and highly effective regulariser against overfitting, particularly valuable when labelled image data is limited — a common situation in African application settings, discussed further in Chapter 11.

Chapter Summary

- Convolutional layers replace dense connections with small, weight-shared kernels, drastically reducing parameter count while building in local connectivity and translation equivariance.
- Activation functions (ReLU chief among them) introduce the non-linearity without which stacking layers would collapse to a single linear transform; ReLU's constant gradient for positive inputs mitigates the vanishing-gradient problem that plagued sigmoid-based networks.
- Pooling reduces spatial resolution and adds a degree of translation invariance; receptive field grows as layers stack, letting deep networks of small kernels see large image regions.
- Training minimises a loss (cross-entropy for classification) via gradient descent, with backpropagation computing all gradients efficiently through the chain rule; batch normalisation and dropout stabilise and regularise this process.

End-of-Chapter Exercises

1. A convolutional layer has 5×5 kernels, 32 input channels, and 64 output filters. Compute its parameter count, and compare it to a dense layer mapping a $28 \times 28 \times 32$ input to 64 outputs.
2. Explain why replacing ReLU with sigmoid throughout a 50-layer CNN would likely make training substantially harder.
3. Two stacked 3×3 convolutions (stride 1, no pooling) are applied to an input. What is the receptive field, in input pixels, of a unit after the second layer? What about after three such layers?
4. A softmax classifier assigns probability 0.02 to the correct class on one example, and 0.9 to the correct class on another. Compute the cross-entropy loss for each and explain the practical implication of the difference for gradient-based learning.
5. Explain, in your own words, why dropout is disabled at test/inference time but active during training.

CHAPTER 5

Landmark CNN Architectures

Chapter 4 established the building blocks — convolution, activation, pooling, normalisation. This chapter tells the story of how those blocks have been *arranged* over roughly three decades of research, and why each new arrangement solved a concrete problem with its predecessor. Understanding this history is not merely trivia: every architecture here is still in active use as a backbone for classification, detection, and segmentation, so recognising their design trade-offs directly informs which one to reach for on a new project.

5.1 LeNet-5: The Original Blueprint (1998)

Yann LeCun's LeNet-5, built for handwritten digit recognition, established the pattern that every subsequent CNN still follows: alternate convolutional and pooling layers to extract progressively higher-level spatial features, then flatten and finish with a small number of fully-connected layers for classification.

$$\text{Input } (32\times32) \to \text{Conv} \to \text{Pool} \to \text{Conv} \to \text{Pool} \to \text{FC} \to \text{FC} \to \text{Output (10 classes)}.$$

At roughly 60,000 parameters, LeNet-5 is minuscule by modern standards, but it proved the core idea worked, on hardware of its era, decades before the data and compute needed to scale it became available.

5.2 AlexNet: The 2012 Breakthrough

AlexNet's landslide win in the 2012 ImageNet competition (a roughly 10-percentage-point error reduction over the next-best, non-deep-learning approach) is widely credited with triggering the deep learning era in computer vision. Architecturally it is a deeper, wider LeNet (5 convolutional layers, 3 fully-connected), but three practical choices mattered enormously:

- **ReLU** activations throughout, instead of the previously standard sigmoid/tanh, dramatically speeding up training (Section 4.2).
- **Dropout** in the fully-connected layers, controlling overfitting on a still-limited (by today's standards) labelled dataset.
- **GPU training**, splitting the network across two GPUs — an early demonstration that the compute budget, not just the algorithm, was a binding constraint on what deep learning could achieve.

5.3 VGG: Depth Through Simplicity

VGGNet's central contribution was a design discipline: use *only* 3×3 convolutions (stride 1, same padding) and 2×2 max pooling, and get depth (16 or 19 weight layers, VGG-16/VGG-19) purely by stacking many of these simple, uniform blocks. As shown in Section 4.3, stacking small kernels grows the receptive field just as effectively as using large kernels, but with fewer

parameters and more non-linearities. VGG's simplicity and uniformity made it enormously popular as a backbone for transfer learning (Chapter 7) for years, even after more parameter-efficient architectures emerged.

Practitioner's Note

VGG's main practical drawback is size: VGG-16 has roughly 138 million parameters, the vast majority in its fully-connected layers, making it slow and memory-hungry relative to its accuracy by modern standards. It remains a reasonable teaching architecture and a solid, well-understood transfer-learning backbone, but ResNet or EfficientNet variants (below) are usually preferred for new production systems.

5.4 ResNet: Solving the Degradation Problem

5.4.1 The problem: deeper networks that perform worse

Naively stacking more and more convolutional layers eventually makes training *harder*, not easier — and not merely due to overfitting: even *training* accuracy degrades once networks get sufficiently deep (tens to hundreds of layers), because gradients must flow backward through every layer, and even with ReLU, very long chains of transformations can still distort or shrink the gradient signal.

5.4.2 The solution: residual (skip) connections

Rather than asking a stack of layers to learn a full transformation $H(x)$ directly, ResNet reformulates the target as a **residual** $F(x) = H(x) - x$, and adds the original input back on via a **skip connection**:

Residual block

$$y = F(x, \{W_i\}) + x,$$

where $F(x, \{W_i\})$ is the output of a small stack of convolutional layers (typically two or three), and x is added back *unchanged* via an identity skip connection (with a 1×1 convolution to match dimensions if channel counts differ).

Intuition

If the extra two layers in a block are not actually needed to improve the representation, the easiest thing for the network to learn is $F(x) \approx 0$, which forces $y \approx x$ — the block simply passes its input through unchanged, acting as an identity mapping. Critically, learning $F(x) = 0$ (driving weights toward zero) is numerically much easier for gradient descent than learning an exact identity mapping $H(x) = x$ through an ordinary (non-residual) stack of non-linear layers, which would require an unusual, delicately balanced combination of weights. Skip connections thus give every block the *option* to be a no-op, which means adding more blocks can never make the network's representational capacity worse, only equal or better — directly resolving the degradation problem and enabling networks like ResNet-152 (152 layers) to train successfully.

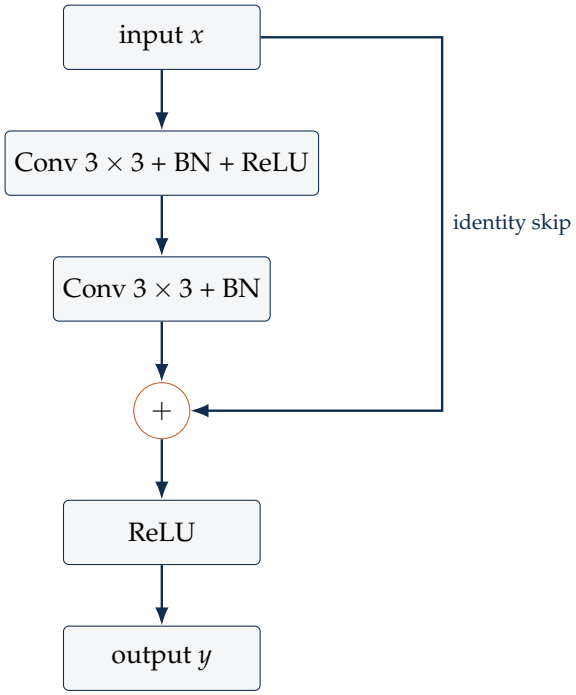


Figure 5.1: A ResNet residual block. The skip connection carries x forward unchanged and adds it to the block's learned output $F(x)$; the block only needs to learn the *residual* correction, which is far easier when the ideal correction is small or even zero.

There is a second, equally important benefit for optimisation: during backpropagation, the identity skip connection provides a direct path for the gradient to flow backward unimpeded (its local derivative with respect to x is exactly 1), supplementing whatever gradient flows through the convolutional branch F. This substantially alleviates the vanishing gradient problem even in networks over 100 layers deep.

5.5 EfficientNet: Principled Scaling

Once ResNet made very deep networks trainable, an obvious question arose: to make a CNN more accurate, should you make it *deeper* (more layers), *wider* (more channels per layer), or feed it *higher-resolution* images? Prior work typically scaled just one of these dimensions, informally. EfficientNet's contribution was to show, empirically, that scaling all three dimensions *together*, in a fixed ratio, gives a better accuracy-to-compute trade-off than scaling any one in isolation.

Compound scaling

$$\text{depth: } d = \alpha^{\phi}, \qquad \text{width: } w = \beta^{\phi}, \qquad \text{resolution: } r = \gamma^{\phi},$$

subject to $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$, where ϕ is a single user-chosen compound coefficient controlling how much extra compute budget to spend, and α, β, γ are constants found by a small grid search on a baseline network.

Intuition

Think of a CNN's capacity as a rectangular box: depth is its height, width is its channel-thickness, and resolution is its footprint. Stretching only the height (depth) without also

widening the footprint (resolution) eventually produces a network that has plenty of abstraction layers but not enough spatial detail surviving to feed them, and vice versa. Compound scaling is the recognition that these three dimensions are not independent knobs but interacting ones, and the constraint $\alpha\beta^2\gamma^2 \approx 2$ is chosen so that each increment of ϕ roughly doubles the network's total computational cost (FLOPs) while growing all three dimensions in a balanced ratio, rather than over-investing in just one.

The baseline network (EfficientNet-B0) was itself found via neural architecture search, using a building block called the **MBConv** (mobile inverted bottleneck) block, which first expands channels with a 1×1 convolution, applies a memory-efficient depthwise 3×3 convolution (one filter per channel, rather than one filter spanning all channels), then projects back down — a design inherited from mobile-efficient architectures (MobileNet) and combined here with the compound scaling idea to produce a family (B0 through B7) spanning a wide range of accuracy/compute trade-offs from a single principled recipe.

Architecture	Year	Depth (approx.)	Key idea
LeNet-5	1998	7 layers	Conv $\rightarrow$ pool $\rightarrow$ FC blueprint
AlexNet	2012	8 layers	ReLU, dropout, GPU training at scale
VGG-16	2014	16 layers	Uniform 3×3 convs, depth via stacking
ResNet-50/152	2015	50–152 layers	Residual (skip) connections
EfficientNet-B0–B7	2019	Variable	Compound depth/width/resolution scaling

Table 5.1: A brief architectural timeline. Each entry solved a specific limitation of its predecessors.

Practitioner's Note

For a new project today, ResNet-50 remains an excellent, well-understood default backbone — fast, robust, and supported everywhere. EfficientNet variants typically offer better accuracy per FLOP, which matters when deploying to constrained hardware (e.g. a low-power device for field-based crop scouting). VGG is mostly retained for teaching and for older transfer-learning pipelines. Chapter 7 discusses how to actually adapt any of these pretrained backbones to a new task with limited labelled data.

Chapter Summary

- LeNet established the conv–pool–FC blueprint; AlexNet scaled it up with ReLU, dropout, and GPU training to win ImageNet 2012, catalysing the deep learning era.
- VGG showed that stacking small, uniform 3×3 convolutions is a simple and effective way to build depth.
- ResNet's residual (skip) connections let each block default to an identity mapping, solving the degradation problem and enabling networks over 100 layers deep, while also providing a direct path for gradients during backpropagation.
- EfficientNet scales depth, width, and input resolution together in a fixed, empirically-derived ratio (compound scaling), yielding better accuracy-per-FLOP than scaling any single dimension alone.

End-of-Chapter Exercises

1. Explain why a residual block that has learned $F(x) \approx 0$ behaves like an identity mapping, and why this matters for training very deep networks.
2. VGG-16 uses only 3×3 convolutions. Using the receptive-field formula from Chapter 4, how many stacked 3×3, stride-1 convolutional layers are needed to match the receptive field of a single 11×11 convolution (as used in AlexNet's first layer)?
3. In your own words, explain why scaling only network depth (with fixed width and resolution) eventually gives diminishing returns, motivating EfficientNet's compound scaling.
4. A ResNet block's skip connection carries x forward via a 1×1 convolution instead of a pure identity. Under what circumstance is this necessary?
5. For a low-power field device used for crop-disease scouting with intermittent connectivity, would you recommend VGG-16 or an EfficientNet-B0 backbone? Justify your answer using the trade-offs discussed in this chapter.

CHAPTER 6

Data Augmentation and Generative Adversarial Networks

Deep CNNs, as built in Chapters 4–5, are data-hungry: millions of parameters need commensurately large, diverse training sets to generalise well rather than memorise. Real-world image datasets, especially in specialised domains like African crop diagnostics or rural clinic imaging, are rarely as large or as diverse as ImageNet. This chapter covers two complementary strategies for stretching limited data further: classical **data augmentation**, and learned, generative augmentation via **Generative Adversarial Networks (GANs)**.

6.1 Traditional Data Augmentation

Data augmentation applies label-preserving transformations to training images, generating new, plausible training examples on the fly without collecting a single additional real photograph.

Category	Transformations	Simulates
Geometric	Flip, rotate, crop, scale, shear	Camera angle, framing variation
Photometric	Brightness, contrast, hue jitter	Lighting and sensor variation
Noise injection	Gaussian noise, blur	Sensor noise, motion, focus
Occlusion	Cutout, random erasing	Partial obstruction, leaf overlap
Mixing	Mixup, CutMix (blend two images/labels)	Ambiguous, boundary-case inputs

Table 6.1: Common data augmentation categories and the real-world variation each approximates.

Intuition

Augmentation works because a CNN's translation equivariance (Chapter 4) does not automatically extend to *rotation*, *scale*, or *lighting* invariance — those must be taught by example. A model that has only ever seen upright, well-lit, centred leaf photographs will not reliably recognise the same leaf photographed at a slight tilt in harsh midday sun, even though a human would consider this a trivial variation. By exposing the model to systematically perturbed versions of the same labelled example, augmentation teaches the network which pixel-level differences are irrelevant to the label (a rotated healthy leaf is still healthy) and which are not (a diseased lesion is still diseased regardless of rotation) — effectively multiplying the diversity of a fixed dataset without changing its ground truth.

Mixup

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j, \qquad \tilde{y} = \lambda y_i + (1-\lambda)y_j, \qquad \lambda \sim \text{Beta}(\alpha, \alpha),$$

blending two training images *and* their labels linearly. This forces the network to behave linearly *between* training examples, which has been shown empirically to improve

generalisation and robustness to slightly corrupted or ambiguous inputs.

Practitioner's Note

Augmentation must respect the semantics of the label. A horizontal flip is harmless for a leaf-disease classifier (a mirrored leaf is still the same class) but is dangerous for a text-recognition task (a mirrored digit "2" is not a valid digit) and must be handled carefully in medical imaging (flipping a chest X-ray left-right may be acceptable, but flipping it top-to-bottom typically is not, since anatomy is not vertically symmetric). Always ask: does this transformation preserve the ground-truth label?

Listing 6.1: A typical augmentation pipeline using torchvision.

```
from torchvision import transforms

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.2),
    transforms.RandomRotation(degrees=15),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]),
])
```

6.2 Generative Adversarial Networks

Traditional augmentation only *transforms* existing images. **GANs** take a more ambitious approach: they learn to *synthesise entirely new, realistic images* from scratch, by pitting two networks against each other in a competitive training game.

6.2.1 The two-player game

A GAN consists of a **generator** G, which maps random noise $z \sim p_z$ (typically a vector sampled from a simple distribution like $\mathcal{N}(0, I)$) to a synthetic image $G(z)$, and a **discriminator** D, which takes an image (real or synthetic) and outputs the probability that it is real. The two are trained simultaneously with opposing objectives: D tries to correctly distinguish real images from G's fakes, while G tries to fool D into thinking its fakes are real.

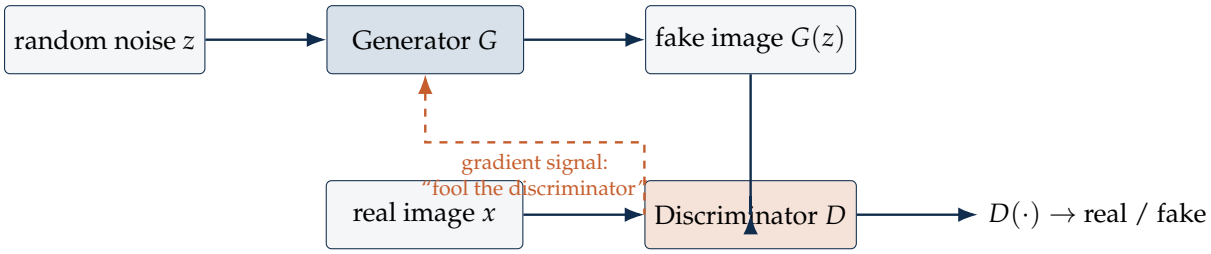


Figure 6.1: The GAN architecture. The generator maps random noise to a synthetic image; the discriminator learns to distinguish real training images from the generator's output; the generator is updated using the discriminator's feedback, pushing it to produce increasingly convincing fakes.

The GAN minimax objective

$$\min_G \max_D \; V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}\big[\log D(x)\big] + \mathbb{E}_{z\sim p_z}\big[\log(1 - D(G(z)))\big].$$

Intuition

Read this formula from the discriminator's side first: D wants to *maximise* V, which means making $D(x)$ close to 1 for real images (so $\log D(x)$ is close to 0, its maximum) and making $D(G(z))$ close to 0 for fakes (so $\log(1 - D(G(z)))$ is also close to 0). In short, D is simply being trained as an ordinary binary classifier: "real" versus "fake." The generator, by contrast, wants to *minimise* V — and since it has no control over the first term (which doesn't involve G), it can only act through the second term, by making $D(G(z))$ as close to 1 as possible (fooling D into thinking the fake is real), which drives $\log(1 - D(G(z)))$ toward $-\infty$. This is a genuine adversarial game: every improvement in D's ability to spot fakes creates a harder training signal that forces G to produce more convincing fakes, and vice versa, ideally converging to a point where G's outputs are indistinguishable from real data and D can do no better than random guessing ($D(x) = 0.5$ everywhere).

6.2.2 Practical training challenges

GAN training is notoriously less stable than standard supervised CNN training, for reasons directly traceable to the minimax formulation:

- **Mode collapse.** The generator discovers a small number of outputs that reliably fool the current discriminator and stops exploring the full diversity of the real data distribution, producing a narrow range of near-identical fakes instead of diverse samples.
- **Vanishing gradients.** If the discriminator becomes too good too quickly, $D(G(z)) \approx 0$ for essentially all generator outputs, which makes $\log(1 - D(G(z)))$ saturate and provides almost no useful gradient to improve G — the generator effectively stops learning even though it is still far from producing realistic images.
- **Oscillation.** Because both networks are moving targets for each other, training can oscillate rather than converge, without the more predictable monotonic loss decrease typical of supervised training.

Practitioner's Note

In practice, DCGAN (Deep Convolutional GAN) established the standard convolutional recipe still used as a starting point today: strided convolutions instead of pooling, batch normalisation in both G and D, and no fully-connected hidden layers. For most applied computer vision work, off-the-shelf pretrained GANs or diffusion-based generators are used for augmentation rather than training a GAN from scratch, since GAN training remains a specialised skill; the exception is domain-specific augmentation (e.g. synthesising additional examples of a rare crop disease) where enough real examples exist to fine-tune a generator but not enough to train a full downstream classifier directly.

6.3 Why This Matters for Limited-Data Domains

Both traditional augmentation and GAN-based synthesis address the same underlying constraint — too few labelled examples relative to model capacity — but at different points on a cost/realism

trade-off. Traditional augmentation is cheap, fast, and always label-preserving by construction, but only explores variations the transformation library can express (rotations, crops, colour jitter). GAN-based augmentation can, in principle, synthesise entirely novel-looking but plausible examples (e.g. a disease lesion pattern not present in any single training photo but consistent with the disease's visual characteristics), at the cost of a much more complex and fragile training process, and the risk of the generator introducing subtle artefacts that a downstream classifier could latch onto as spurious "shortcut" features.

Chapter Summary

- Data augmentation applies label-preserving geometric, photometric, noise, and mixing transformations to multiply the effective diversity of a fixed training set, teaching invariances a CNN does not acquire automatically.
- Mixup linearly blends pairs of images and labels, encouraging locally linear model behaviour between training examples.
- GANs train a generator and discriminator adversarially via the minimax objective $\min_G \max_D V(D, G)$; the discriminator is trained as a standard real/fake classifier, and the generator is trained to fool it.
- GAN training is prone to mode collapse, vanishing gradients (from an overly strong discriminator), and oscillation, all traceable to the adversarial (rather than purely cooperative) nature of the training objective.
- Choose augmentation strategy based on the cost/realism trade-off: traditional augmentation for cheap, reliable, label-safe diversity; generative augmentation for synthesising harder-to-express variation, at higher implementation cost and risk.

End-of-Chapter Exercises

1. List two augmentation transformations that would be inappropriate for a road-sign classification task, and explain why.
2. Using the minimax objective, explain what value of $D(G(z))$ the generator is implicitly trying to drive the discriminator toward, and why.
3. Explain, in terms of the loss landscape, why an overly strong discriminator early in training can stall the generator's learning.
4. A dataset contains only 40 confirmed photographs of a rare cassava disease. Propose a combined augmentation strategy (traditional + potentially generative) and justify the order in which you would try each technique.
5. Why is mode collapse a particularly serious problem if GAN-generated images are to be used as additional training data for a downstream classifier?

CHAPTER 7

Transfer Learning and Fine-Tuning

Training a ResNet or EfficientNet from random initialisation, as sketched in Chapters 4–5, typically requires hundreds of thousands to millions of labelled images and substantial compute. Very few real-world projects — and almost none in specialised, data-scarce domains such as African crop diagnostics or rural clinical imaging — have access to datasets of that scale. **Transfer learning** resolves this mismatch by reusing a network already trained on a large, general dataset (typically ImageNet, with 1.2 million images across 1000 categories) as the starting point for a new, much smaller, task-specific dataset.

7.1 Why Pretrained Features Transfer

7.1.1 What early and late layers learn

Empirically, and consistent with the receptive-field discussion in Chapter 4, the layers of a trained CNN learn a hierarchy of increasingly abstract, increasingly task-specific features:

- **Early layers** (small receptive field) learn generic, low-level features: oriented edges, colour blobs, simple textures — remarkably similar to the hand-designed Sobel and Gabor-like filters from Chapter 3, but learned automatically. These are almost universally useful, regardless of whether the final task is recognising dogs, diseases, or satellite land cover.
- **Middle layers** learn more complex, still fairly generic, textures and part-like patterns (fur, weave patterns, curved contours).
- **Late layers** (large receptive field, close to the output) learn highly task-specific, high-level combinations of parts (e.g. "dog face," "car wheel") tuned to the exact 1000 ImageNet categories the network was trained to distinguish.

Intuition

Imagine training two people: one to identify dog breeds, another to identify diseases in maize leaves. Both people, in the course of their very different training, will independently learn to notice edges, colours, and textures — this low-level visual vocabulary is shared and does not need to be relearned from scratch by the second person. What differs is the high-level combination of these primitives into a final judgement. Transfer learning formalises this insight: keep the network's early, generic "visual vocabulary" layers as they are, and only retrain the later, task-specific layers (or retrain everything, but starting from a far better initialisation than random weights) for the new task.

7.2 Two Transfer Learning Strategies

7.2.1 Feature extraction (frozen backbone)

Remove the pretrained network's final classification layer (which outputs 1000 ImageNet class scores), replace it with a new, randomly initialised layer sized for the new task (e.g. 3 outputs for "healthy / diseased / pest-damaged"), and **freeze** every other layer — meaning their weights are held fixed and receive no gradient updates during training. Only the new final layer is trained.

7.2.2 Fine-tuning (partially or fully unfrozen)

Unfreeze some or all of the pretrained layers (typically starting from the layers closest to the output and working backward) and continue training them, alongside the new final layer, on the new dataset — usually with a substantially lower learning rate than would be used for training from scratch, so as not to destroy the useful pretrained weights with large, disruptive updates.

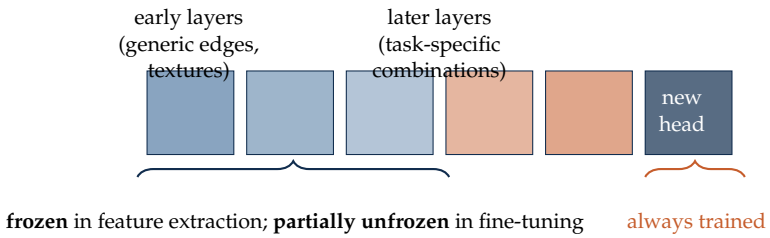


Figure 7.1: A pretrained backbone (blue/orange blocks) has its final classification head replaced with a new, task-specific head (navy). In feature extraction, everything except the new head is frozen; in fine-tuning, some or all backbone layers are unfrozen and trained too, typically at a much lower learning rate.

Discriminative (layer-wise) learning rates

$$\eta_\ell = \eta_{\text{base}} \times \rho^{L-\ell}, \qquad 0 < \rho < 1,$$

where ℓ indexes layers from input ($\ell = 1$) to output ($\ell = L$), η_{base} is the learning rate for the final (new) layer, and ρ discounts earlier layers, giving them smaller updates than later layers.

Intuition

The new final layer starts from random weights and needs to learn quickly. The early backbone layers, by contrast, already encode broadly useful features and only need gentle adjustment to specialise slightly to the new domain — large updates risk catastrophically "forgetting" the useful generic representations learned from a much larger dataset than the new task provides. Discriminative learning rates formalise this asymmetry directly into the optimiser: a high rate for the new head, progressively smaller rates the deeper (earlier) into the pretrained backbone you go.

	Small, similar dataset	Large, similar dataset	Small/large, dissimilar dataset
Recommended	Feature extraction	Fine-tune all layers	Fine-tune later layers only
Why	Too little data to safely update backbone; overfitting risk high	Enough data to safely adapt every layer for extra accuracy	Some new low/mid-level features needed, but full fine-tuning risks overfitting

Table 7.1: A practical decision matrix. "Similar" means the new domain shares low-level visual statistics with the pretraining domain (e.g. natural RGB photographs); "dissimilar" means it does not (e.g. grayscale medical scans, radar imagery).

7.3 Deciding Between the Two Strategies

Choosing a strategy for crop disease detection

Suppose a project has 800 labelled leaf photographs across 4 disease classes, using a ResNet-50 pretrained on ImageNet. ImageNet contains many natural outdoor photographs including plants, so the domain is reasonably *similar*; but 800 images is a *small* dataset relative to ResNet-50's roughly 25 million parameters. Per the decision matrix, feature extraction (freeze the backbone, train only a new final layer) or fine-tuning only the last one or two residual blocks is the appropriate choice — attempting to fine-tune all 50 layers on 800 images would very likely overfit, since the pretrained weights would receive enough gradient signal to drift substantially away from their useful, generically-learned starting point without enough data to safely guide that drift.

7.4 Practical Fine-Tuning Recipe

Listing 7.1: Feature extraction followed by gradual fine-tuning in PyTorch.

```
import torch, torch.nn as nn
from torchvision import models

model = models.resnet50(weights="IMAGENET1K_V2")

# Stage 1: feature extraction -- freeze the backbone
for param in model.parameters():
    param.requires_grad = False

# Replace the final layer for our 4-class problem
model.fc = nn.Linear(model.fc.in_features, 4)   # only this layer trains

# ... train for a few epochs with a normal learning rate (e.g. 1e-3) ...

# Stage 2: fine-tune the last residual block too, at a lower rate
for param in model.layer4.parameters():
    param.requires_grad = True

optimizer = torch.optim.Adam([
    {"params": model.layer4.parameters(), "lr": 1e-5},
    {"params": model.fc.parameters(),     "lr": 1e-3},
```

```
])
# ... continue training ...
```

Practitioner's Note

A few field-tested rules of thumb: (1) always freeze batch normalisation statistics (or use a very small learning rate for them) when fine-tuning on a small dataset, since mini-batch statistics computed from a small or imbalanced dataset can be unreliable; (2) monitor validation loss closely once you unfreeze any backbone layers — a validation loss that starts rising while training loss keeps falling is the classic signature of overfitting the newly-unfrozen weights; (3) when the new domain is visually very different from natural photographs (e.g. hyperspectral or radar satellite bands, ultrasound), even the earliest pretrained layers may need adaptation, and it can be worth experimentally comparing frozen vs. fully fine-tuned early layers rather than assuming generic low-level features will transfer well.

Chapter Summary

- Transfer learning reuses a network pretrained on a large, general dataset as a starting point for a new, typically much smaller, task-specific dataset, exploiting the fact that early CNN layers learn broadly reusable low-level features.
- Feature extraction freezes the entire pretrained backbone and trains only a new final layer; fine-tuning additionally updates some or all backbone layers, usually at a reduced learning rate.
- Discriminative (layer-wise) learning rates apply progressively smaller updates to earlier, more generic layers and larger updates to later, more task-specific layers (and the new head).
- The right strategy depends on both dataset size and domain similarity to the pre-training data; small, similar datasets favour feature extraction, while larger or more dissimilar datasets can support progressively more fine-tuning.

End-of-Chapter Exercises

1. A team has 50,000 labelled X-ray images (a large but visually dissimilar-to-ImageNet dataset) and a pretrained ResNet-50. Using the decision matrix, what strategy would you recommend, and why?
2. Explain, using the concept of receptive field from Chapter 4, why early convolutional layers are more likely to transfer across unrelated domains than late layers.
3. Why might fine-tuning batch normalisation layers on a small, class-imbalanced dataset be risky?
4. Sketch a discriminative learning rate schedule (in words, not code) for a 4-block CNN backbone plus a new head, given $\eta_{\text{base}} = 10^{-3}$ and $\rho = 0.3$.
5. A validation loss begins increasing three epochs after unfreezing the last two residual blocks of a fine-tuned ResNet, while training loss keeps decreasing. What is the most likely explanation, and what two remedies would you try first?

CHAPTER 8

Object Detection I: Two-Stage Detectors

Image classification (Chapters 4–7) answers "what is the dominant object in this image?" **Object detection** answers a strictly harder question: "what objects are present, and *where* is each one?" This chapter and the next build up the two major detection paradigms, starting with the family that solves the problem in two distinct stages: propose candidate regions, then classify and refine each one.

8.1 Formalising the Problem

A detector must output, for a given image, a variable-length list of **bounding boxes** $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$, each with an associated class label and confidence score. This is fundamentally different from classification's fixed-size output (one label per image), and it introduces a new central question: given a predicted box and a ground-truth box, how do we measure whether they "match"?

8.1.1 Intersection over Union

Intersection over Union (IoU)

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\text{area of overlap}}{\text{area of union}} \in [0, 1].$$

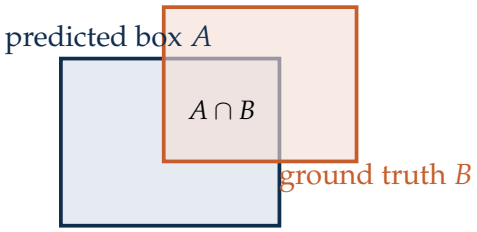


Figure 8.1: IoU measures the overlap between a predicted box A and a ground-truth box B as the ratio of their intersection area to their union area. IoU $= 1$ means a perfect match; IoU $= 0$ means no overlap at all.

Computing IoU

Predicted box $A = (10, 10, 90, 90)$ (an 80×80 square) and ground truth $B = (50, 50, 130, 130)$ (also 80×80). The intersection spans $x \in [50, 90]$, $y \in [50, 90]$, a 40×40 square: $|A \cap B| = 1600$. The union is $|A| + |B| - |A \cap B| = 6400 + 6400 - 1600 = 11{,}200$. So:

$$\text{IoU} = \frac{1600}{11{,}200} \approx 0.143.$$

This low IoU indicates a poor match; detection evaluation protocols typically require IoU ≥ 0.5 (or higher, for stricter benchmarks) for a prediction to be counted as a correct detection

of that ground-truth object.

Intuition

IoU is preferred over simpler alternatives (like centre-point distance) because it is scale-invariant and directly penalises both under- and over-sized predictions: a predicted box that is either too small (misses parts of the object) or too large (includes too much background) suffers a lower IoU, exactly matching the intuitive notion of "how well does this box outline the object."

8.2 From Sliding Windows to Region Proposals

8.2.1 The naive baseline: sliding windows

The most direct approach to detection is to slide a classifier over the image at every position and at multiple scales, classifying each window as "object" or "background." This is prohibitively expensive: a 1000×600 image at even a modest set of window sizes and strides produces tens of thousands of windows, each requiring a full forward pass through a CNN.

8.2.2 Selective Search: smarter proposals

R-CNN (Region-based CNN) sidesteps brute-force sliding windows using **Selective Search**, a classical (non-learned) algorithm that over-segments the image into small regions based on colour, texture, and intensity similarity, then hierarchically merges similar adjacent regions to produce roughly 2000 candidate **region proposals** per image — far fewer than exhaustive sliding windows, while still capturing the vast majority of true object locations.

8.3 The R-CNN Family

8.3.1 R-CNN (2014)

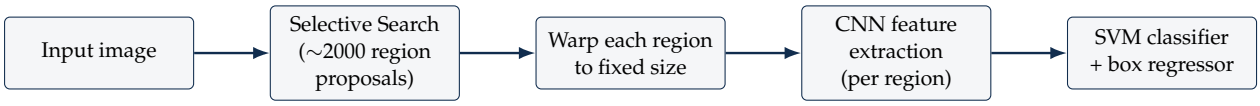


Figure 8.2: The original R-CNN pipeline. Each of roughly 2000 region proposals is independently warped and passed through a full CNN forward pass — the single biggest bottleneck of this design.

R-CNN established the paradigm but is extremely slow: each of the ∼2000 proposed regions is independently cropped, resized, and pushed through a full CNN forward pass, meaning the same image's overlapping regions have their shared visual content processed redundantly thousands of times.

8.3.2 Fast R-CNN (2015): sharing computation

Fast R-CNN's key insight is that this redundancy is wasteful: instead of running the CNN separately on every region proposal, run it *once* on the whole image to produce a single shared feature map, and then, for each region proposal, extract the corresponding sub-region *from that shared feature map* rather than from the raw image.

RoI Pooling

Given a region proposal and a shared convolutional feature map, RoI (Region of Interest) Pooling divides the proposal's corresponding feature-map region into a fixed $H' \times W'$ grid and max-pools each cell, producing a fixed-size output regardless of the original proposal's size or aspect ratio.

Intuition

RoI Pooling solves a very concrete engineering problem: region proposals come in every conceivable size and aspect ratio, but the fully-connected classification layers that follow need a fixed-size input. RoI Pooling is essentially an adaptive max-pooling operation — rather than pooling with a fixed window size and stride, it pools with a window size *computed on the fly* so that any input region, however large or small, is squeezed down to exactly the same fixed output grid (e.g. 7×7), making the downstream fully-connected layers usable for any proposal shape.

By sharing one CNN pass across all region proposals for a given image, Fast R-CNN is roughly two orders of magnitude faster than R-CNN at inference, and can be trained end-to-end (the CNN, classifier, and box regressor are jointly optimised) rather than R-CNN's separate, disjoint training stages.

8.3.3 Faster R-CNN (2015): learning the proposals too

Fast R-CNN still depends on Selective Search, a slow, non-learned, CPU-bound algorithm that becomes the new bottleneck once the CNN itself has been sped up. **Faster R-CNN** replaces Selective Search with a **Region Proposal Network (RPN)** — a small additional CNN, sharing the same backbone feature map, that itself learns to propose regions.

Anchor boxes

At every position of the shared feature map, the RPN considers k pre-defined *anchor boxes* — reference boxes of several fixed scales and aspect ratios, centred at that position — and for each anchor predicts (1) an objectness score (is there an object here at all, foreground vs. background) and (2) four regression offsets refining the anchor into a tighter box around the actual object.

Intuition

Anchors reformulate an open-ended search problem ("find boxes of any possible size and position") into a bounded classification-and-regression problem ("for this fixed, finite set of candidate boxes, which are objects, and how should each be nudged to fit better?"). This is directly analogous to how a multiple-choice exam is easier to grade than an open-ended one: rather than the network having to freely predict arbitrary (x, y, w, h) coordinates from scratch (numerically difficult, since the output space is unbounded and the loss landscape is poorly behaved), it only has to select from and slightly adjust a curated set of plausible reference shapes.

Because the RPN reuses the same shared backbone feature map that the final classification stage also uses, Faster R-CNN's proposal generation costs only a small additional CNN head,

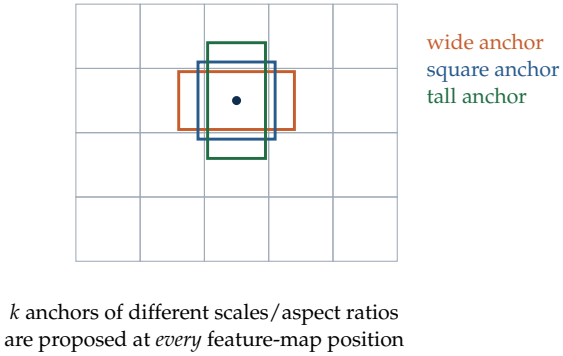


Figure 8.3: At each position of the shared convolutional feature map, the RPN evaluates k anchor boxes of pre-set scales and aspect ratios (here, $k = 3$: wide, square, tall), scoring each as foreground/background and regressing a refined box.

making the entire pipeline (backbone $\rightarrow$ RPN $\rightarrow$ RoI pooling $\rightarrow$ classification/regression) trainable end-to-end and fast enough for many practical applications, though still slower than the one-stage detectors of Chapter 9.

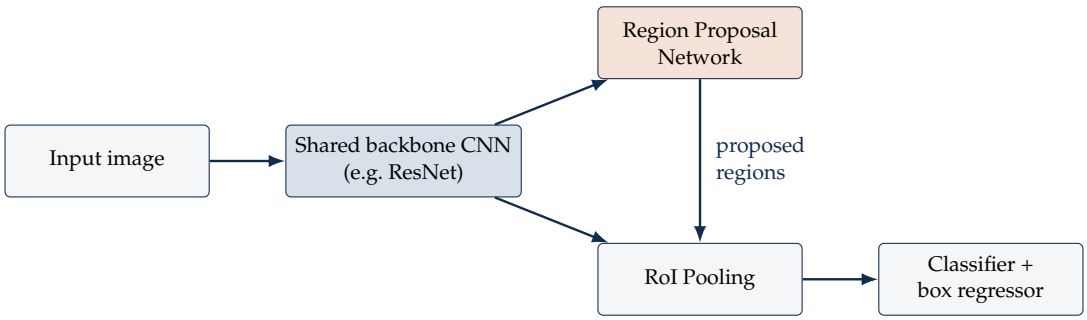


Figure 8.4: The Faster R-CNN pipeline. A single shared backbone feeds both the Region Proposal Network and the final RoI pooling stage, eliminating the separate, slow Selective Search step of earlier R-CNN variants.

8.4 Non-Maximum Suppression

Any of the pipelines above will typically fire multiple overlapping, highly-confident detections around the same physical object (e.g. several nearby anchors all correctly flagging the same car). **Non-Maximum Suppression (NMS)** is the standard post-processing step that reduces this redundancy to a single detection per object.

Non-Maximum Suppression algorithm

1. Sort all candidate boxes for a class by confidence score, descending.
2. Select the highest-scoring box; add it to the final output list; remove it from the candidate list.
3. Compute IoU between this selected box and every remaining candidate; discard any candidate with IoU above a threshold (e.g. 0.5) with the selected box.
4. Repeat from step 2 until no candidates remain.

Intuition

NMS operationalises a simple rule: "trust the most confident detection in a cluster, and assume any other box that overlaps it heavily is describing the *same* object, not a second, genuinely distinct one." The IoU threshold controls how aggressive this suppression is — too low a threshold risks merging two genuinely separate, closely-spaced objects (e.g. two adjacent plants in a densely-planted field) into one; too high a threshold leaves duplicate detections uncleaned.

Chapter Summary

- Object detection requires localising a variable number of objects, evaluated via Intersection over Union (IoU) between predicted and ground-truth boxes.
- R-CNN pioneered region-proposal-based detection but is slow, since it runs a full CNN forward pass independently on each of $\sim$2000 proposed regions.
- Fast R-CNN shares one CNN pass across all proposals via RoI Pooling, which adaptively pools any proposal's feature-map region into a fixed-size output.
- Faster R-CNN replaces the slow, non-learned Selective Search step with a learned Region Proposal Network built on anchor boxes, making the full pipeline trainable end-to-end.
- Non-Maximum Suppression removes duplicate, overlapping detections of the same object, keeping only the highest-confidence box in each cluster.

End-of-Chapter Exercises

1. Compute the IoU between predicted box $(0, 0, 100, 50)$ and ground-truth box $(20, 10, 120, 70)$.
2. Explain, step by step, why Fast R-CNN is faster than R-CNN despite using an almost identical underlying CNN backbone.
3. An RPN uses 3 scales and 3 aspect ratios at every feature-map position. How many anchors are evaluated per position? If the shared feature map is 40×60, how many total anchors are evaluated across the whole image?
4. Two ground-truth cassava plants are spaced closely together in a field photograph, and NMS with IoU threshold 0.3 is applied. Explain the risk this poses and suggest a mitigation.
5. Why can the Region Proposal Network share the same backbone feature map used later for final classification, rather than needing its own separate CNN?

CHAPTER 9

Object Detection II: One-Stage Detectors

Faster R-CNN (Chapter 8) still separates detection into two conceptual stages: propose regions, then classify and refine them. Even sharing a backbone, this two-stage structure imposes a speed ceiling that makes real-time applications (video, live camera feeds, robotics, drone-based crop scouting) difficult. **YOLO** ("You Only Look Once") and its one-stage relatives reframe detection as a single, direct regression problem, trading a small amount of accuracy for a large gain in speed.

9.1 The One-Stage Idea

Rather than first proposing candidate regions and then classifying each one, YOLO divides the input image into a coarse grid and asks each grid cell to *directly predict* bounding boxes and class probabilities in a single forward pass — no separate proposal stage at all.

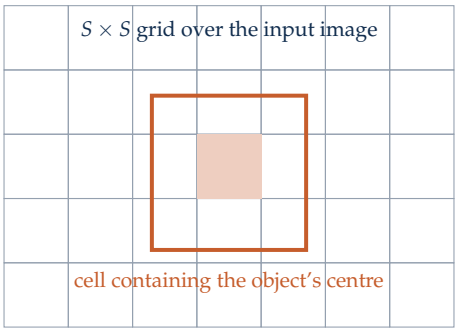


Figure 9.1: YOLO divides the image into an $S \times S$ grid (here 7×5 for illustration). The grid cell that contains an object's centre is made "responsible" for predicting that object.

Intuition

This is a genuine change of problem formulation, not just an engineering shortcut. Faster R-CNN asks "*where might* objects be?" as a distinct first question, then "what is each proposed region?" as a second question. YOLO instead asks a single, uniform question at every grid cell simultaneously: "if an object's centre is here, what box and class would best describe it?" This lets a single CNN forward pass produce every prediction at once, rather than requiring a second network pass conditioned on first-stage proposals — the "you only look once" in the name refers exactly to this single-pass design.

9.2 The YOLO Output and Loss Function

9.2.1 What each grid cell predicts

Each of the $S \times S$ grid cells predicts B candidate bounding boxes (each with 4 coordinates plus a confidence score) and C class probabilities, giving a total output tensor of shape $S \times S \times (B \times 5 + C)$.

Box confidence

$$\text{confidence} = \Pr(\text{object}) \times \text{IoU}_{\text{pred}}^{\text{truth}},$$

a single scalar that is high only when the cell both believes an object is present *and* its predicted box tightly matches that object's true extent.

9.2.2 The multi-part loss function

Training YOLO requires balancing several different objectives (box coordinates, objectness, classification) in a single loss:

YOLO loss, simplified

$$\begin{aligned}\mathcal{L} = \lambda_{\text{coord}} \sum_{i,j} \mathbb{1}_{ij}^{\text{obj}} \Big[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \Big] \\ + \sum_{i,j} \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \;+\; \lambda_{\text{noobj}} \sum_{i,j} \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \\ + \sum_{i} \mathbb{1}_{i}^{\text{obj}} \sum_{c} (p_i(c) - \hat{p}_i(c))^2,\end{aligned}$$

combining a localisation term (box coordinates, only for cells actually responsible for an object), a confidence term (both for object-containing and empty cells, weighted differently), and a classification term.

Intuition

Three design choices in this loss are worth dwelling on. **First**, coordinates are only penalised for the cell responsible for an object ($\mathbb{1}_{ij}^{\text{obj}}$) — empty cells are not asked to regress meaningless box coordinates. **Second**, width and height errors are computed on their *square roots*, not the raw values: this is because an error of, say, 10 pixels matters far more for a small 20×20 box than for a large 200×200 box, and the square root compresses large values more than small ones, which partially equalises the loss's sensitivity across object scales. **Third**, the vast majority of grid cells in any real image contain no object at all, so the "no object" confidence term is down-weighted by $\lambda_{\text{noobj}} < 1$ (typically 0.5) relative to the localisation terms (λ_{coord}, typically 5); without this rebalancing, the overwhelming number of background cells would dominate training and drown out the comparatively rare, but far more important, object-containing cells — a direct illustration of the general class-imbalance problem that recurs throughout detection.

9.3 Anchor-Based YOLO and Successive Refinements

Early YOLO (v1) predicted box dimensions directly, which struggled with unusual aspect ratios and localisation precision. From YOLOv2 onward, the architecture adopted **anchor boxes** (Chapter 8) at each grid cell, predicting offsets *relative to* a small set of pre-defined anchor shapes (often chosen via k-means clustering on the training set's actual box shapes, rather than hand-picked) instead of predicting raw coordinates from scratch — combining YOLO's single-pass speed with the improved localisation stability that anchors provide.

	Two-stage (Faster R-CNN)	**One-stage (YOLO)**
Speed	Slower (separate proposal + classification passes)	Faster (single forward pass)
Small objects	Generally stronger (region proposals adapt to scale)	Historically weaker, though later versions (YOLOv3+, multi-scale prediction) substantially closed the gap
Typical use	Offline batch processing, applications prioritising accuracy over latency	Real-time video, live camera feeds, edge/embedded deployment

Table 9.1: A practical comparison of two-stage and one-stage detection paradigms.

9.4 Evaluating Detectors: Mean Average Precision

Classification accuracy is not an adequate metric for detection, since a detector must be right about both *what* and *where*. The standard metric is **mean Average Precision (mAP)**.

Precision and Recall at a given IoU/confidence threshold

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN},$$

where a predicted box counts as a true positive (TP) if its IoU with some ground-truth box of the same class exceeds a threshold (commonly 0.5) and it is the highest-confidence such match; otherwise it is a false positive (FP). A ground-truth object with no matching prediction is a false negative (FN).

Intuition

Average Precision (AP) for a class is computed by varying the confidence threshold used to accept a detection, plotting the resulting precision against recall at each threshold, and measuring the area under this precision-recall curve. Intuitively, a good detector should maintain high precision (few false alarms) even as it is pushed to achieve higher recall (find more of the true objects) — a curve that stays close to the top-right corner of the precision-recall plot. **mAP** averages this AP score across all object classes, and is often reported at a specific IoU threshold (e.g. mAP@0.5) or averaged across a range of IoU thresholds (e.g. mAP@[0.5:0.95], the stricter COCO-style protocol), giving progressively harsher credit for looser localisation as the IoU threshold rises.

Practitioner's Note

For a field deployment such as real-time weed detection from a moving tractor-mounted camera, YOLO (or a similarly efficient one-stage architecture) is almost always the correct choice, since the application has a hard latency budget the two-stage alternative cannot meet. For an offline task such as auditing a large archive of satellite tiles for building counts, where processing can run overnight in a batch, the higher accuracy of a two-stage detector may be worth the extra compute time. Always match the detector family to the deployment constraint, not simply to whichever architecture currently tops a public benchmark leaderboard.

Chapter Summary

- YOLO reframes detection as a single-pass regression problem: an $S \times S$ grid, with each cell directly predicting bounding boxes, confidence, and class probabilities.
- The YOLO loss combines localisation, confidence, and classification terms, using square-root box dimensions and down-weighted "no object" confidence terms to address scale sensitivity and class imbalance respectively.
- From YOLOv2 onward, anchor boxes (often derived by clustering the training set's box shapes) improve localisation while retaining single-pass speed.
- Two-stage detectors (Faster R-CNN) generally favour accuracy, especially for small or unusual objects; one-stage detectors (YOLO) favour speed and are the standard choice for real-time applications.
- Detection quality is evaluated with mean Average Precision (mAP), which jointly credits correct localisation (via IoU) and correct classification across a range of confidence thresholds.

End-of-Chapter Exercises

1. Explain why YOLO penalises box width/height error using square roots rather than raw pixel differences.
2. A 7×7 grid YOLO model predicts $B = 2$ boxes per cell over $C = 20$ classes. What is the shape of the full output tensor?
3. Why is the "no object" confidence loss term typically down-weighted relative to the localisation loss term? What would likely happen during training if it were not?
4. For each scenario, state whether a one-stage or two-stage detector is the more appropriate choice, and justify briefly: (a) counting livestock in nightly-uploaded drone survey images; (b) flagging pedestrians from a moving vehicle's live camera feed.
5. A detector achieves high precision but low recall on a rare-disease detection task. Explain, in plain language, what this means for a user of the system, and what practical trade-off adjusting the confidence threshold would involve.

CHAPTER 10

Image Segmentation

Object detection (Chapters 8–9) localises objects with rectangular boxes — useful, but coarse: a bounding box around an irregularly-shaped leaf lesion includes plenty of healthy tissue, and a box around a person includes background between their arm and torso. **Segmentation** pushes localisation to its finest possible granularity: a decision for *every single pixel*.

10.1 Semantic versus Instance Segmentation

- **Semantic segmentation** assigns every pixel a class label (e.g. "leaf," "lesion," "background"), but does *not* distinguish between different instances of the same class — two overlapping leaves both labelled "leaf" are indistinguishable in the output.
- **Instance segmentation** additionally separates individual object instances, so two adjacent leaves of the same class receive distinct instance masks, answering not just "which pixels are leaf" but "which pixels belong to *this particular* leaf."

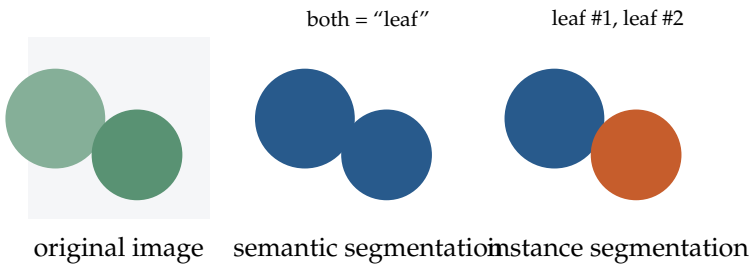


Figure 10.1: Semantic segmentation labels every pixel by class only; instance segmentation additionally separates individual object instances of the same class.

10.2 Fully Convolutional Networks

Classification CNNs (Chapters 4–5) end with fully-connected layers that collapse all spatial information into a single fixed-length vector — appropriate for "what is the dominant object," but useless for a task that needs to preserve and output a decision *per spatial location*. **Fully Convolutional Networks (FCNs)** replace those fully-connected layers with convolutional layers throughout, and finish with an **upsampling** (or **transposed convolution**) stage that restores the feature map from its downsampled, coarse resolution back to (or near) the original input resolution, producing a dense, per-pixel class prediction map.

Intuition

A standard classification CNN's pooling and strided convolutions repeatedly downsample the spatial resolution (Chapter 4) precisely because spatial precision is irrelevant to "what object is this." Segmentation needs the opposite: spatial precision *is* the entire out-

put. FCNs resolve this tension with an encoder-decoder shape: an encoder (a normal classification-style backbone) progressively downsamples while building increasingly abstract, semantically rich features, and a decoder progressively upsamples this abstract representation back to full resolution, converting "what kind of thing is generally present, somewhere in this broad receptive field" into "what is present, at this exact pixel."

10.3 U-Net: Skip Connections for Precise Localisation

Naively upsampling a heavily downsampled feature map loses fine spatial detail that cannot be recovered by upsampling alone — after several pooling layers, information about exactly *where* an edge was located has been discarded, not just compressed. **U-Net**, originally developed for biomedical image segmentation (a domain that, like much African agricultural and medical imaging, often has scarce labelled training data), solves this with **skip connections** that directly copy feature maps from the encoder to the corresponding decoder stage, at matching resolution.

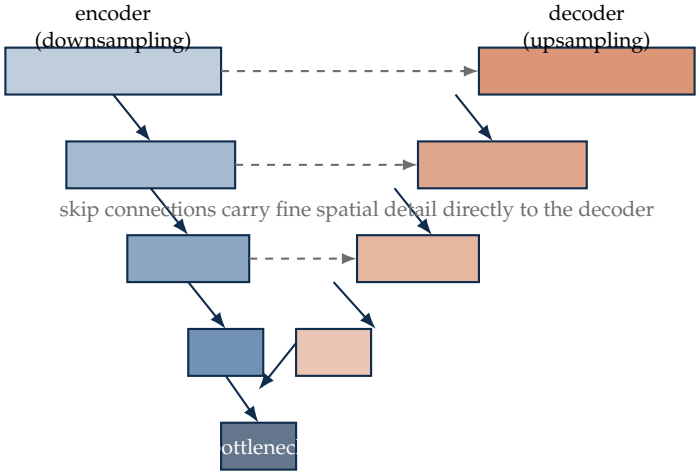


Figure 10.2: The U-Net architecture. The encoder (left) progressively downsamples while extracting abstract features, down to a bottleneck; the decoder (right) progressively upsamples back to full resolution. Skip connections (dashed) copy encoder feature maps directly to the matching-resolution decoder stage, restoring fine spatial detail that pure upsampling cannot recover.

Intuition

Skip connections here play a conceptually similar role to ResNet's skip connections (Chapter 5), but for a different purpose: rather than easing gradient flow through depth, U-Net's skip connections re-inject spatial detail that was necessarily discarded by downsampling. Concretely, the decoder at a given resolution receives *two* sources of information: the coarse, semantically rich signal arriving from the deeper (more downsampled) decoder stage below it ("this general region is probably a lesion"), and the fine-grained, high-resolution signal copied directly across from the matching encoder stage ("and here is exactly where the sharp edges are"). Combining both lets the network produce segmentation masks that are both semantically correct *and* pixel-accurate at object boundaries, which is precisely what a downsampling-only architecture cannot achieve.

10.4 Mask R-CNN: Instance Segmentation

Mask R-CNN extends Faster R-CNN (Chapter 8) with a third parallel output branch: alongside the existing classification and box-regression heads, a small FCN head predicts a binary segmentation mask for each detected region proposal, producing full instance segmentation (per-object detection *and* a pixel-accurate mask) from a single, unified architecture.

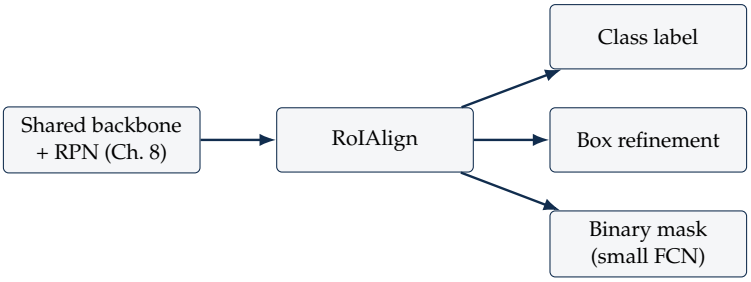


Figure 10.3: Mask R-CNN adds a third, parallel mask-prediction branch to the Faster R-CNN detection pipeline, run independently for each detected region alongside the classification and box-regression branches.

10.4.1 RoIAlign: fixing a quantisation problem

Recall RoI Pooling (Chapter 8) divides a region proposal into a fixed grid for pooling. Region proposal boundaries, however, are floating-point coordinates (e.g. $x = 17.6$), and RoI Pooling rounds these to the nearest integer feature-map cell — a small misalignment that barely affects a bounding box (already an approximate rectangle) but is catastrophic for a pixel-precise mask, where a one- or two-pixel shift visibly degrades the mask's boundary accuracy. **RoIAlign** fixes this by using bilinear interpolation to sample feature values at exact, non-rounded floating-point locations, avoiding the quantisation misalignment entirely.

Intuition

RoI Pooling's rounding is a bit like reading an analogue ruler but only being allowed to record the nearest whole centimetre — fine for a rough estimate, but insufficient if you need millimetre precision. RoIAlign instead reads the exact position, using interpolation (a weighted average of the nearest known values) exactly the way you would estimate a value between two ruler markings by eye. This single change was responsible for a substantial jump in mask accuracy over the naive RoI Pooling approach used in earlier detection architectures.

10.5 Segmentation Loss Functions

10.5.1 Pixel-wise cross-entropy

The most direct segmentation loss simply applies the classification cross-entropy loss (Chapter 4) independently to every pixel, then averages: $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_{\text{CE}}(y_i, \hat{y}_i)$. This is simple and effective when classes are reasonably balanced across pixels, but struggles when one class vastly outnumbers another in pixel count (e.g. a small tumour occupying 2% of a scan, surrounded by 98% healthy tissue) — the loss can be minimised almost entirely by getting the majority class right and largely ignoring the rare, but clinically critical, minority class.

10.5.2 Dice loss

Dice loss, derived from the Dice similarity coefficient (closely related to IoU), directly optimises mask overlap rather than independent per-pixel correctness, and is far more robust to severe class imbalance.

Dice coefficient and Dice loss

$$\text{Dice} = \frac{2|P \cap G|}{|P| + |G|}, \qquad \mathcal{L}_{\text{Dice}} = 1 - \text{Dice},$$

where P is the predicted mask (as soft probabilities, for a differentiable loss) and G is the ground-truth mask.

Intuition

The Dice coefficient (like IoU) measures overlap as a *ratio*, so it is naturally insensitive to the absolute number of background pixels — a tiny 20-pixel tumour mask predicted almost perfectly still scores a high Dice coefficient, because both the numerator and denominator scale with the size of the region actually being compared, not the size of the whole (mostly irrelevant) image. This is precisely the property pixel-wise cross-entropy lacks, which is why Dice loss (often combined with cross-entropy, as a weighted sum of both) is the standard choice for medical image segmentation, where the structure of clinical interest is frequently a small minority of the total pixel count.

Practitioner's Note

For satellite land-use segmentation (e.g. distinguishing cropland, forest, and bare soil across a tile), classes are often more balanced, and pixel-wise cross-entropy alone may suffice. For medical image segmentation (tumour, lesion) or fine agricultural structures (thin leaf veins, small pest damage spots), where the class of interest occupies a small fraction of total pixels, Dice loss (or a combined Dice + cross-entropy loss) is strongly preferred, and is worth trying even when cross-entropy alone seems to be working, since class imbalance can silently produce misleadingly high pixel-accuracy scores while the actual region of interest is poorly segmented.

Chapter Summary

- Semantic segmentation labels every pixel by class; instance segmentation additionally separates individual object instances within the same class.
- Fully Convolutional Networks replace classification's fully-connected layers with an encoder (downsampling) followed by a decoder (upsampling), producing dense, per-pixel predictions.
- U-Net's skip connections copy fine-grained spatial detail directly from encoder to decoder at matching resolution, correcting for detail lost during downsampling.
- Mask R-CNN adds a parallel mask-prediction branch to Faster R-CNN, using RoIAlign (bilinear interpolation) instead of RoI Pooling's rounding to preserve the pixel-level precision masks require.
- Dice loss (versus plain pixel-wise cross-entropy) directly optimises mask overlap and is markedly more robust to the severe class imbalance common in medical and

fine-structure segmentation tasks.

End-of-Chapter Exercises

1. Explain, in your own words, the practical difference between semantic and instance segmentation using an example of overlapping maize plants in a drone image.
2. Why can a segmentation network not simply reuse a classification network's architecture unchanged, replacing only the final layer?
3. A predicted mask and ground-truth mask overlap in 180 pixels; the predicted mask has 220 pixels total, and the ground truth has 200 pixels total. Compute the Dice coefficient and the corresponding Dice loss.
4. Explain why RoIAlign's bilinear interpolation matters more for mask prediction than for bounding-box regression.
5. A tumour segmentation model trained with plain pixel-wise cross-entropy achieves 97% pixel accuracy, yet clinicians report it frequently misses small tumours entirely. Explain this discrepancy and propose a concrete change to the training loss.

CHAPTER 11

Case Studies: Computer Vision in African Contexts

The preceding chapters built the mathematics and architectures of modern computer vision in general terms. This chapter grounds that machinery in three application areas of direct relevance across the African continent: crop disease detection, satellite-based land-use mapping, and medical image diagnosis. Each case study shares a common thread with the curriculum's broader emphasis: how to build reliable systems when labelled data, compute, and connectivity are all more constrained than the conditions under which most published benchmarks (ImageNet, COCO) were assembled.

11.1 Crop Disease and Pest Detection

11.1.1 The problem

Smallholder farmers, who produce the majority of staple crops across much of Sub-Saharan Africa, often lack easy access to agricultural extension officers who can diagnose plant diseases in the field. A smartphone-based or low-cost-camera system that can classify a photographed leaf as healthy, diseased (and identify which disease), or pest-damaged offers a scalable diagnostic aid, provided it works reliably outside a controlled lab setting.

11.1.2 Why this is harder than a typical benchmark

- **Field-condition variability.** Benchmark datasets are frequently photographed against clean, uniform backgrounds under controlled lighting; field photographs include cluttered backgrounds, variable and often harsh sunlight, motion blur, and leaves at varying growth stages.
- **Fine-grained, overlapping symptoms.** Several diseases can produce visually similar early-stage lesions on the same crop, requiring the network to learn subtle, fine-grained distinctions rather than the coarse category differences typical of general object classification.
- **Limited labelled data per class.** Rare diseases, or diseases specific to under-studied regional crop varieties, may have only a few dozen to a few hundred confirmed, expert-labelled photographs available.

Practitioner's Note

A practical pipeline for this setting typically combines several techniques from earlier chapters rather than relying on any single one: (1) HSV-based colour thresholding (Chapter 1) as a lightweight pre-filter to segment the leaf from a cluttered background before classification; (2) aggressive data augmentation (Chapter 6), particularly photometric jitter, to compensate for uncontrolled field lighting; (3) transfer learning from an ImageNet-pretrained backbone

(Chapter 7), using feature extraction or light fine-tuning given the typically small labelled dataset; and (4) careful attention to *class imbalance* in both the loss function and the evaluation metric, since a model can achieve deceptively high overall accuracy by simply always predicting the majority "healthy" class while missing rare disease classes almost entirely.

Why accuracy alone misleads

Suppose a dataset has 900 healthy leaf photographs and 100 diseased ones (a realistic imbalance for an early-stage disease). A model that predicts "healthy" for *every* image achieves $900/1000 = 90\%$ accuracy while being completely useless for its actual purpose — it never once correctly identifies a diseased leaf. This is exactly why detection and segmentation tasks use precision, recall, and mAP (Chapter 9) rather than raw accuracy, and why a classification task with severe class imbalance should be evaluated with per-class recall and, often, a class-weighted or focal loss variant of cross-entropy (Chapter 4) that penalises misclassifying the rare class more heavily than the common one.

11.2 Land-Use Mapping from Satellite Imagery

11.2.1 The problem

Government planning bodies, NGOs, and researchers frequently need up-to-date maps of land cover and land use — distinguishing cropland from forest, informal settlements from planned urban development, or tracking deforestation and urban expansion over time — across areas too large for manual ground survey. Satellite imagery (from freely available sources such as Sentinel-2 or higher-resolution commercial providers) combined with semantic segmentation (Chapter 10) automates this at continental scale.

11.2.2 Domain-specific considerations

- **Multispectral, not just RGB, input.** Satellite sensors often capture additional bands beyond visible RGB (e.g. near-infrared, which is highly informative for vegetation health via indices like NDVI). A segmentation CNN's first convolutional layer must be adapted to accept these extra input channels, and ImageNet-pretrained weights (trained on 3-channel RGB) only partially transfer — the extra channels' filters must be learned from scratch or carefully initialised.
- **Very large images, tiled processing.** A single satellite scene can be tens of thousands of pixels per side — far larger than any single CNN forward pass can process at once. Standard practice tiles the scene into overlapping patches, runs segmentation independently per tile, and stitches results back together, with care taken at tile boundaries to avoid visible seams or double-counted objects.
- **Coarse temporal cadence and cloud cover.** Unlike a controlled photography setting, satellite revisit times are fixed by orbital mechanics, and a substantial fraction of imagery for any given date can be unusable due to cloud cover, requiring either cloud-masking preprocessing or compositing across several dates.

Practitioner's Note

For land-cover segmentation specifically, U-Net (Chapter 10) and its variants remain a strong default: the technique's origins in biomedical imaging (another domain with large images, fine boundary detail, and limited labels) transfer naturally to satellite tiles. Kriging and other classical spatial interpolation techniques (covered in the curriculum's Time Series & Spatial Analysis module) remain useful for filling small gaps left by cloud masking, complementing rather than being replaced by the deep learning segmentation pipeline.

11.3 Medical Image Diagnosis Under Data Scarcity

11.3.1 The problem

Diagnostic imaging — X-ray, ultrasound, and increasingly point-of-care devices — offers enormous potential to extend specialist-level diagnostic capability to clinics that lack a radiologist or specialist on site. But medical imaging datasets, particularly from African clinical settings, are typically far smaller than natural-image benchmarks, both because patient data is (rightly) subject to strict privacy protections limiting large-scale collection, and because expert annotation (a radiologist's time) is expensive and scarce.

11.3.2 Techniques that matter most in this setting

- **Transfer learning is close to essential**, not merely helpful (Chapter 7): training a randomly-initialised CNN from scratch on a few hundred or even a few thousand medical images is rarely viable. A pretrained backbone, fine-tuned carefully (often only the last few layers, given small dataset sizes), is the standard starting point.
- **Dice loss and related overlap-based losses** (Chapter 10) matter enormously here, since the structure of clinical interest (a tumour, a fracture line, a fluid collection) is frequently a small minority of an image's total pixels, and plain pixel-wise cross-entropy can produce a model that looks accurate by every naive metric while systematically missing the pathology that matters.
- **Explainability is not optional.** A clinician cannot responsibly act on a model's output without some indication of *why* the model reached its conclusion. Techniques such as SHAP and LIME (covered in the curriculum's Ethics & Responsible AI module), or simpler saliency/activation-map visualisations specific to CNNs, should accompany any diagnostic-support deployment, not be treated as an optional add-on.
- **Rigorous, prospective validation before deployment.** A model that performs well on a retrospective test set drawn from the same hospital and imaging equipment as its training data can still fail badly when deployed on a different device, patient population, or imaging protocol elsewhere — a phenomenon called **dataset shift**. Validation on genuinely external data, ideally from the actual deployment setting, is essential before any diagnostic tool is trusted in practice.

Intuition

Dataset shift is easy to underestimate because a model can be simultaneously *highly accurate* and *dangerously unreliable*: high accuracy on a test set drawn from the same source as training data only confirms the model learned *some* predictive pattern in that specific

data distribution — it says nothing about whether that pattern is the genuine clinical signal (e.g. the actual disease pathology) or a spurious correlate specific to that hospital's equipment, patient demographics, or even image file metadata (a well-documented failure mode in early medical imaging AI research was models that partly "cheated" by picking up on scanner-specific image artefacts correlated with which hospital, and therefore which patient population, an image came from). This is precisely why external validation, on data the model has never seen and ideally from a different source entirely, is treated as non-negotiable in responsible medical AI development.

11.4 A Common Thread

Across all three case studies, the same handful of techniques from earlier chapters recur, applied to the specific constraints of the domain: transfer learning to compensate for limited labelled data (Chapter 7); careful choice of loss function and evaluation metric to address class imbalance (Chapters 4, 9, 10); data augmentation tuned to the domain's real sources of variation (Chapter 6); and validation practices designed to catch failure modes — class imbalance masking poor minority-class performance, or dataset shift masking genuine generalisation failure — that a single headline accuracy number will not reveal. Building computer vision systems for real-world African deployment contexts is less about needing fundamentally different algorithms, and more about applying the full toolkit of this book with unusually careful attention to exactly these failure modes.

Chapter Summary

- Crop disease detection must contend with uncontrolled field conditions, fine-grained symptom overlap, and limited per-class labelled data; HSV pre-filtering, aggressive augmentation, transfer learning, and imbalance-aware evaluation together form a robust practical pipeline.
- Satellite land-use mapping requires adapting CNNs to multispectral (not just RGB) input, tiling very large scenes for processing, and handling cloud cover and coarse revisit cadence; U-Net-style segmentation architectures transfer naturally from their biomedical origins.
- Medical image diagnosis under data scarcity depends heavily on transfer learning, overlap-based losses (Dice) for small-structure segmentation, mandatory explainability, and rigorous external validation to guard against dataset shift.
- A model can be highly accurate on its own test set while being dangerously unreliable in deployment if that test set does not represent the true diversity of deployment conditions — a risk that is best mitigated by careful metric choice and genuinely external validation, not by chasing a higher accuracy number on the original test set.

End-of-Chapter Exercises

1. A cassava disease classifier reports 94% overall accuracy but a per-class recall of only 38% on the rarest disease category. Explain what likely happened and propose two concrete fixes, referencing techniques from earlier chapters.
2. Why can ImageNet-pretrained weights only be partially reused for a 6-channel

(RGB + 3 additional spectral bands) satellite segmentation model? Which part of the network is affected?

3. Describe, in your own words, a scenario in which a medical imaging model could achieve high accuracy on its own hospital's test set purely due to dataset shift, without learning any genuine clinical signal.
4. Explain why Dice loss (Chapter 10) is particularly well-suited to tumour segmentation, using the specific numbers from the Chapter 10 worked example as a reference point.
5. You are advising a team building a land-cover segmentation model for a region with frequent cloud cover. Propose two complementary strategies (one from computer vision, one from spatial/temporal statistics) for handling missing or cloud-obscured imagery.

CHAPTER 12

The Practical Toolkit: OpenCV, PyTorch, and an End-to-End Pipeline

The preceding eleven chapters built theory: the mathematics of convolution, the architectures that use it, and the tasks (classification, detection, segmentation) built on top of it. This closing chapter assembles the practical software toolkit and shows how the pieces fit together into a single, coherent project workflow, from raw image to deployed model.

12.1 OpenCV: The Classical Vision Workhorse

OpenCV (Open Source Computer Vision Library) remains the standard library for classical image operations — everything from Chapters 1–3 — and is frequently used as a preprocessing front-end even in deep-learning-centric pipelines.

Listing 12.1: A representative OpenCV preprocessing sequence, chaining several Chapter 1–3 operations.

```
import cv2
import numpy as np

img = cv2.imread("leaf_field_photo.jpg")                # BGR, uint8
img = cv2.resize(img, (512, 512))

# Colour-space based background segmentation (Chapter 1)
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
lower_green = np.array([25, 40, 40])
upper_green = np.array([95, 255, 255])
leaf_mask = cv2.inRange(hsv, lower_green, upper_green)

# Morphological cleanup of the mask (small extension of Chapter 1 connectivity
    ideas)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
leaf_mask = cv2.morphologyEx(leaf_mask, cv2.MORPH_CLOSE, kernel)

# Denoising (Chapter 2)
denoised = cv2.medianBlur(img, 5)

# Edge map for a quick sanity check on lesion boundaries (Chapter 3)
gray = cv2.cvtColor(denoised, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150)

masked_leaf = cv2.bitwise_and(denoised, denoised, mask=leaf_mask)
cv2.imwrite("preprocessed_leaf.png", masked_leaf)
```

Practitioner's Note

Even in a project built around a deep CNN, OpenCV preprocessing steps like the background segmentation above are often worth keeping: isolating the leaf from a cluttered field background *before* the CNN ever sees the image removes a source of irrelevant variation the network would otherwise have to learn to ignore, which is especially valuable when labelled training data is limited (Chapter 11).

12.2 PyTorch: Building and Training the Model

Where OpenCV handles classical image operations, deep learning frameworks (PyTorch, used throughout this section, or TensorFlow/Keras equivalently) handle model definition, automatic differentiation for backpropagation (Chapter 4), and GPU-accelerated training.

Listing 12.2: A minimal but complete transfer-learning training loop, combining Chapters 6 and 7.

```
import torch, torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import models, datasets, transforms

device = "cuda" if torch.cuda.is_available() else "cpu"

# --- Data pipeline: augmentation (Chapter 6) + normalisation (Chapter 1) ---
train_tf = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(0.3, 0.3, 0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225]),
])
train_ds = datasets.ImageFolder("data/train", transform=train_tf)
train_loader = DataLoader(train_ds, batch_size=32, shuffle=True, num_workers
    =4)

# --- Model: pretrained backbone + new head (Chapter 7) ---
model = models.resnet50(weights="IMAGENET1K_V2")
for p in model.parameters():
    p.requires_grad = False
model.fc = nn.Linear(model.fc.in_features, len(train_ds.classes))
model = model.to(device)

# --- Loss (Chapter 4) and optimiser ---
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.fc.parameters(), lr=1e-3)

# --- Training loop ---
model.train()
for epoch in range(10):
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()          # backpropagation (Chapter 4)
        optimizer.step()
        running_loss += loss.item()
```

```
    print(f"Epoch {epoch+1}: loss = {running_loss/len(train_loader):.4f}")

torch.save(model.state_dict(), "leaf_classifier.pt")
```

12.3 The End-to-End Project Pipeline

Every technique in this book fits into one of six stages of a repeatable project workflow. Skipping or rushing any single stage is the most common cause of a computer vision project underperforming in deployment relative to its reported training-time metrics.

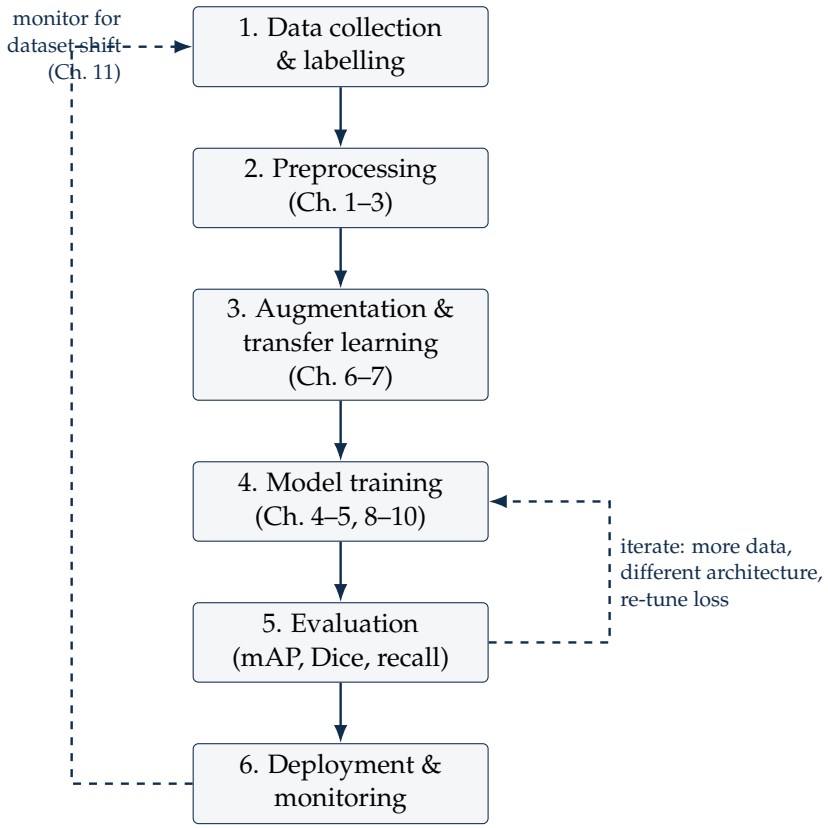


Figure 12.1: The end-to-end computer vision project pipeline. Evaluation routinely sends the project back to model training (a different architecture, loss, or more augmentation), and deployment monitoring can send it all the way back to data collection if dataset shift is detected in production.

1. **Data collection and labelling.** The ceiling on any model's performance is set here, not at the architecture stage; time invested in careful, consistent labelling (and in capturing the true diversity of deployment conditions, per Chapter 11) usually pays off more than time invested in trying more architectures.
2. **Preprocessing.** Classical operations (Chapters 1–3): colour space conversion, denoising, normalisation, and any domain-specific filtering (e.g. background segmentation).
3. **Augmentation and transfer learning setup.** Choosing an augmentation policy (Chapter 6) and a pretrained backbone plus a feature-extraction or fine-tuning strategy (Chapter 7), matched to dataset size and domain similarity.
4. **Model training.** Selecting and training the appropriate architecture for the task: a classification CNN (Chapters 4–5), a detector (Chapters 8–9), or a segmentation network (Chapter

10).

5. **Evaluation.** Using task-appropriate metrics, not just accuracy: mAP for detection, Dice/IoU for segmentation, and per-class precision/recall for any classification task with class imbalance (Chapter 11).
6. **Deployment and monitoring.** Packaging the trained model for its target environment (cloud API, mobile app, edge device), and continuously monitoring live performance for dataset shift — since a model's true test is not the held-out test set, but the real, evolving distribution of images it encounters after deployment.

Practitioner's Note

The two dashed feedback arrows in Figure 12.1 are deliberate: a real project is rarely a single pass through these six stages. A disappointing evaluation result almost always sends you back to model training (try a different loss, more augmentation, a different backbone) rather than immediately to more data collection; and production monitoring that reveals dataset shift sends you all the way back to data collection, since no amount of retraining on the *old* data distribution will fix a model failing on a genuinely new one.

Listing 12.3: A minimal inference-time deployment wrapper.

```python
import torch
from torchvision import transforms
from PIL import Image

class LeafDiseaseClassifier:
    def __init__(self, weights_path, class_names, device="cpu"):
        self.device = device
        self.classes = class_names
        self.model = models.resnet50(weights=None)
        self.model.fc = nn.Linear(self.model.fc.in_features, len(class_names))
        self.model.load_state_dict(torch.load(weights_path, map_location=
            device))
        self.model.eval().to(device)
        self.transform = transforms.Compose([
            transforms.Resize((224,224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225]),
        ])

    @torch.no_grad()
    def predict(self, image_path):
        img = Image.open(image_path).convert("RGB")
        x = self.transform(img).unsqueeze(0).to(self.device)
        logits = self.model(x)
        probs = torch.softmax(logits, dim=1)[0]
        idx = probs.argmax().item()
        return self.classes[idx], probs[idx].item()
```

Chapter Summary

- OpenCV remains the standard toolkit for classical preprocessing (colour space work, denoising, morphology, edge detection), and is frequently retained as a front-end even in deep-learning-centric pipelines.

- PyTorch (or an equivalent framework) handles model definition, automatic differentiation, and GPU-accelerated training, implementing the loss functions, architectures, and transfer-learning strategies covered in Chapters 4–10.
- A complete project pipeline spans data collection, preprocessing, augmentation/transfer-learning setup, training, task-appropriate evaluation, and deployment with ongoing monitoring — with realistic feedback loops between stages, not a single linear pass.
- Data quality and evaluation metric choice are usually higher-leverage than architecture selection; a well-labelled dataset evaluated with the right metric will expose problems that no amount of architecture tuning can fix.

End-of-Chapter Exercises

1. Extend the OpenCV preprocessing listing to also apply CLAHE (Chapter 2) to the value channel before background segmentation, and explain why you would place this step before, rather than after, the HSV thresholding.
2. In the PyTorch training loop, identify which line corresponds to the forward pass, which to the loss computation, and which to backpropagation.
3. A team reports 96% training accuracy but poor field performance. Using Figure 12.1, identify at least three distinct pipeline stages that could plausibly explain this gap, and one diagnostic check for each.
4. Explain why the inference wrapper's `predict` method must apply exactly the same `transforms.Normalize` statistics used during training, and what would go wrong if it used different values.
5. Propose a concrete, lightweight production monitoring signal (something measurable without new ground-truth labels) that could indicate dataset shift for a deployed leaf-disease classifier.

APPENDIX A

Glossary

Anchor box A pre-defined reference bounding box (fixed scale and aspect ratio) used by detectors such as Faster R-CNN and YOLO as a starting point for predicting object locations.

Backpropagation The repeated application of the chain rule to efficiently compute the gradient of a loss function with respect to every weight in a neural network.

Batch normalisation A layer that normalises activations to zero mean and unit variance across a mini-batch, then applies a learned scale and shift, stabilising and accelerating training.

Bounding box A rectangle, typically specified by its corner coordinates, that localises an object within an image.

CLAHE Contrast-Limited Adaptive Histogram Equalisation; applies histogram equalisation locally, on small image tiles, with a clip limit to avoid over-amplifying noise.

Convolution The operation of sliding a small kernel across an image (or feature map) and computing a weighted local sum at each position; the core computational primitive of both classical filtering and CNNs.

Cross-entropy loss A loss function for classification that penalises the negative log-probability the model assigns to the true class, punishing confident mistakes severely.

Data augmentation Label-preserving transformations (flips, crops, colour jitter, etc.) applied to training images to increase the effective diversity of a fixed dataset.

Dataset shift A mismatch between the data distribution a model was trained/tested on and the distribution it encounters in deployment, which can cause silent performance degradation.

Dice loss A segmentation loss based on the Dice overlap coefficient, more robust to class imbalance than plain pixel-wise cross-entropy.

Discriminator In a GAN, the network trained to distinguish real data from the generator's synthetic output.

Fine-tuning Continuing to train some or all of a pretrained network's weights on a new, typically smaller, task-specific dataset.

Fully Convolutional Network (FCN) A CNN with no fully-connected layers, using only convolutional (and pooling/upsampling) layers so it can produce dense, per-pixel output for segmentation.

GAN Generative Adversarial Network; a generator and discriminator trained together in an adversarial game to synthesise realistic data.

Generator In a GAN, the network trained to map random noise to synthetic data that fools the discriminator.

Gradient descent An iterative optimisation algorithm that updates parameters in the direction that most decreases a loss function.

Histogram equalisation A contrast-enhancement technique that redistributes pixel intensities using the cumulative distribution function of the image's intensity histogram.

Instance segmentation Pixel-level labelling that both classifies and distinguishes individual object instances of the same class.

Intersection over Union (IoU) The ratio of the overlap area to the union area of two regions (typically bounding boxes or masks); the standard measure of localisation accuracy.

Kernel A small matrix of weights used in a convolution operation; hand-designed in classical filtering (Sobel, Gaussian) or learned in CNNs.

Mean Average Precision (mAP) The standard detection evaluation metric, averaging Average Precision (area under the precision-recall curve) across classes and/or IoU thresholds.

Mixup An augmentation technique that linearly blends pairs of training images and their labels.

Mode collapse A GAN training failure in which the generator produces only a narrow range of outputs rather than the full diversity of the real data distribution.

Non-Maximum Suppression (NMS) A post-processing step that removes duplicate, overlapping detections of the same object, keeping only the highest-confidence box in each cluster.

Pooling A downsampling operation (typically max or average) that reduces the spatial size of a feature map without learnable parameters.

Receptive field The region of the original input that can influence a given unit's value in a particular layer of a CNN.

Region Proposal Network (RPN) The learned component of Faster R-CNN that proposes candidate object regions using anchor boxes on a shared feature map.

ReLU Rectified Linear Unit; the activation function $f(z) = \max(0, z)$, widely used in CNNs for its favourable gradient properties.

Residual (skip) connection A connection that adds a layer's (or block's) input directly to its output, easing optimisation in very deep networks (ResNet).

RoI Align / RoI Pooling Operations that extract a fixed-size feature representation from a variable-sized region proposal; RoIAlign uses bilinear interpolation for sub-pixel accuracy, RoI Pooling rounds to the nearest feature-map cell.

Semantic segmentation Pixel-level classification that labels every pixel by class but does not distinguish individual object instances.

Stride The step size by which a convolution or pooling kernel moves between successive applications.

Transfer learning Reusing a network pretrained on a large, general dataset as the starting point for a new, related task.

Translation equivariance The property that shifting the input shifts the output correspondingly; the defining property of convolutional layers.

U-Net A segmentation architecture with a downsampling encoder, upsampling decoder, and skip connections that preserve fine spatial detail.

YOLO "You Only Look Once"; a family of one-stage object detectors that directly regress bounding boxes and class probabilities from a grid over the input image in a single forward pass.

APPENDIX B

Further Reading

This companion volume deliberately favours intuition and worked derivation over exhaustive citation. Readers looking to go deeper on any single topic are encouraged to consult the following categories of source material.

- **Foundational papers.** The original papers introducing AlexNet, VGG, ResNet, EfficientNet, R-CNN/Fast R-CNN/Faster R-CNN, YOLO, U-Net, and Mask R-CNN are all short, well-written, and substantially more approachable after working through this book's derivations first.
- **Framework documentation.** The official PyTorch and OpenCV documentation, including their respective tutorial series, are the most reliable source for API details and remain current as libraries evolve — always preferred over secondhand summaries for exact usage.
- **Open datasets for practice.** ImageNet (general classification), COCO (detection and segmentation), and domain-specific collections such as PlantVillage (crop disease) offer realistic practice material once the concepts in this book are in hand.
- **The rest of the AIMS–Cameroon Data Science curriculum.** In particular, the Optimisation Theory module (gradient descent variants, convexity, Bayesian optimisation), the Statistical Inference and Biostatistics modules (evaluation methodology, confounding, causal inference), and the Ethics & Responsible AI module (fairness metrics, explainability, data governance) all directly deepen material that this book necessarily covers only briefly.

This book has traced a single throughline from the first pixel in Chapter 1 to a deployed, monitored model in Chapter 12: every technique introduced along the way exists to answer one of two questions a computer vision system must eventually answer about an image — *what is in it*, and *where*. The mathematics changes from chapter to chapter, but that throughline does not.