Image Processing and Computer Vision

Sivieri Chiara

2025

Contents

1 Introduction

1.1 Definition and Relationship

Computer Vision (CV) is a field that focuses on the automatic extraction of information from visual data, such as images and videos. In contrast, **Image Processing (IP)** mainly concerns itself with enhancing the quality of images, often as a preprocessing step for Computer Vision. The two fields are closely related and frequently used together.

Image Processing can be viewed as a foundation that enables more advanced Computer Vision tasks. For example, filtering and enhancement techniques can help highlight specific features in an image, making it easier for a Computer Vision algorithm to interpret and analyze the scene.

1.2 Types of Information Extracted in Computer Vision

Computer Vision aims to extract various types of meaningful information from images and videos. These include:

- **Objects and scenes:** Identifying and classifying elements within a scene (e.g., trees, cars, people).
- **Activities and gestures:** Recognizing human actions or specific motions (e.g., walking, waving).
- **Locations and landmarks:** Understanding where something is in space or recognizing specific places.
- **Text and writing:** Extracting written content from images.
- **Faces and emotions:** Detecting faces and interpreting emotional expressions.
- **3D structure:** Estimating spatial properties or geometry from 2D image data (e.g., depth estimation, structure-from-motion).
- **Search and organization:** Enabling systems to search through image or video archives based on visual content.

These applications demonstrate the importance of CV in areas such as robotics, surveillance, medical imaging, and human-computer interaction.

1.3 Challenges in Computer Vision

Despite significant progress, many challenges remain in the field:

- **Loss of depth information:** Images are 2D projections of a 3D world, leading to ambiguities.

- **Variability:** Objects may appear in different scales, lighting conditions, and orientations.
- **Occlusions:** Important parts of objects may be hidden or partially visible.
- **Articulated objects:** Complex deformable shapes (e.g., human bodies) are difficult to model and track.

These factors make it difficult to create algorithms that are robust and generalize well across different contexts.

1.4 Historical Milestones in Computer Vision

Computer Vision has evolved significantly since its early days:

- **1963:** Larry Roberts' PhD thesis focused on perceiving 3D solids from 2D images.
- **1973:** Pictorial structures introduced by Fischler and Elschlager.
- **1982:** David Marr proposed an articulated body model for perception.
- **2001:** Lowe introduced Scale-Invariant Feature Transform (SIFT).
- **2003–2005:** Techniques such as panorama stitching, bag-of-visual-words, pedestrian detection, and Viola-Jones object detection laid foundations for modern CV.

These contributions paved the way for modern techniques and tools that power today's intelligent vision systems.

Applications include inspection, quality control, object tracking, process monitoring, and decision automation.

1.5 The Deep Learning Paradigm Shift

There has been a major paradigm shift in CV, moving from hand-crafted features and rule-based systems to **data-driven learning systems**, primarily due to deep learning:

- Before 2012: Vision pipelines relied on manual feature design and hand-tuned rules.
- After 2012: With the advent of deep learning (e.g., AlexNet), features, representations, and decision functions are learned end-to-end.

This shift was powered by increased computational resources, large labeled datasets (like ImageNet), and new architectures like CNNs (Convolutional Neural Networks).

1.6 ImageNet and ILSVRC

The ImageNet dataset (Over 14 million images across 22K categories) and its associated Large Scale Visual Recognition Challenge (ILSVRC) had a transformative effect: **ILSVRC** Introduced a benchmark for 1000-class classification using 1.4M training images.

- Landmark models: AlexNet (2012), ZFNet (2013), VGG (2014), Inception (2014), ResNet (2015), SENet (2017).
- Top-5 error rates decreased dramatically from 28.1% (2010) to 2.3% (2017).

1.7 Modern Computer Vision Tasks

Modern CV applications cover a broad range of tasks, such as:

- **Image Classification**
- **Object Detection and Instance Segmentation**
- **Face Recognition and Emotion Detection**
- **Traffic Sign Recognition**
- **Semantic Segmentation**

These capabilities have enabled mass-market products such as surveillance systems, autonomous vehicles, and smartphone apps.

1.8 Emerging Generative Tasks

Recent years have seen the emergence of generative CV tasks:

- **Image Synthesis:** Using GANs (e.g., StyleGAN3 by NVIDIA).
- **Neural Rendering:** With technologies like NeRF for novel view synthesis.
- **Text-to-Image generation:** Enabled by models from Google and others.
- **Image Captioning:** Generating natural language descriptions of image content.

These advancements have broadened the scope of CV from perception to content generation.

1.9 Current Landscape and Future Trends

- The field is rapidly evolving, with constant developments in datasets, benchmarks, and methods.
- Sites like `paperswithcode.com` provide up-to-date performance metrics and code implementations.
- Computer Vision is now a central component in AI research and practical applications, and will continue to grow in relevance across disciplines.

2 Image Formation and Digitization

2.1 Image Formation: Geometric and Radiometric Models

Images are 2D representations of the 3D world, created by collecting light reflected from objects. The process of image formation involves both:

- A **geometric relationship** between the 3D scene and the image plane.
- A **radiometric relationship** between the brightness of image points and the illumination of scene points.

2.2 The Pinhole Camera Model

The pinhole camera is a simplified imaging model where light rays from scene points pass through a small hole and project onto the image plane. This results in an inverted image and defines a basic geometric model that approximates many real-world devices, despite its impracticality due to poor brightness and long exposure needs.

2.3 Perspective Projection

The **Perspective Projection** model formalizes how a 3D scene point (X, Y, Z) is mapped to a 2D image point (u, v). Using the camera reference system:

$$u = -\frac{fx}{z}, \quad v = -\frac{fy}{z}$$

To avoid the image flipping, we conceptually move the image plane in front of the pinhole, making:

$$u = \frac{fx}{z}, \quad v = \frac{fy}{z}$$

The image coordinates are scaled projections of the scene coordinates, inversely proportional to depth (z). The farther a point is, the smaller its projection.

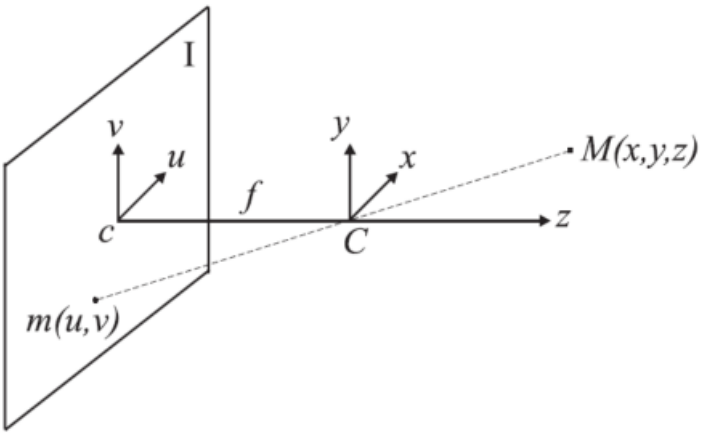


Figure 1: The Image Plane and the Camera Reference System

- "u" is the horizontal axis in the image plane
- "v" is the vertical axis in the image plane
- "X" and "Y" are the respective axis in the 3D reference system

M : scene point
m : corresponding image point
I : image plane
C : optical centre (pin hole)
Optical axis: line through C and orthogonal to I
c : intersection between optical axis and image plane (image centre or piercing point)
f : focal length
F : focal plane

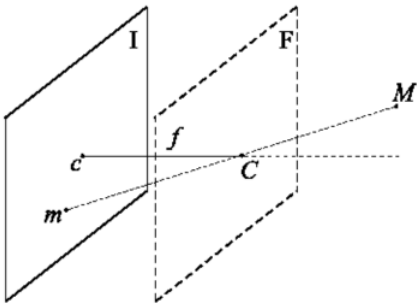
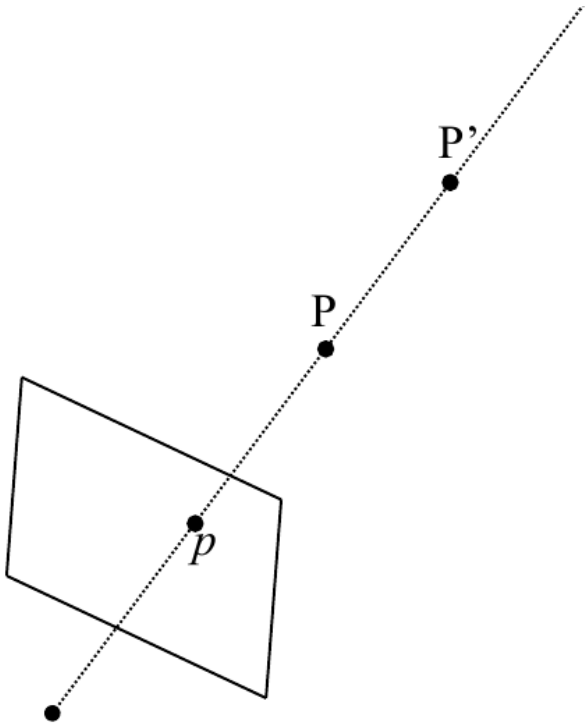


Figure 2: The Image Plane and the Focal Plane in the Perspective Projection Model

2.4 Limits of Perspective Projection



The mapping from 3D to 2D is not invertible: a unique image point corresponds to an entire line in 3D space.
Recovering the depth from a single image is an ill-posed problem, as multiple 3D points may project to the same pixel.
In fact, the pixel p on the image could correspond to any of the points on this straight line.

2.5 Stereo Vision for Depth Estimation

To overcome this problem, stereo vision uses two images captured simultaneously from slightly different viewpoints, similar to how the human visual system works. Given corresponding image points in left and right views, the 3D position can be recovered via **triangulation**.

2.5.1 Stereo Geometry

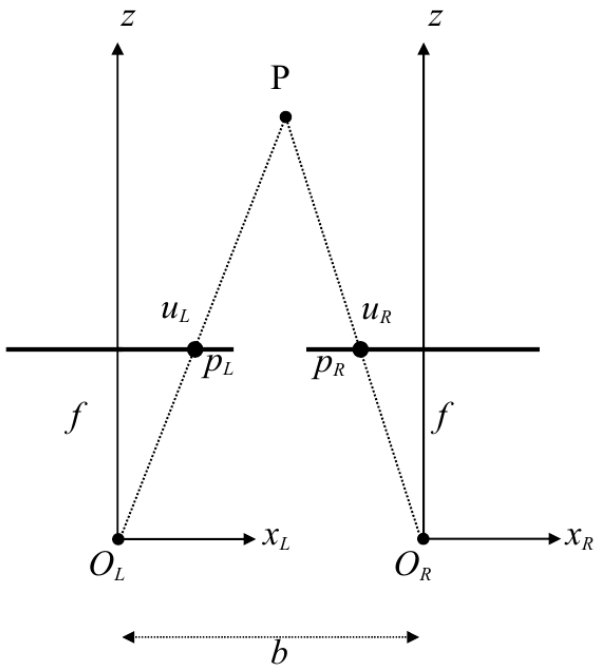
Assuming:

- Aligned cameras with parallel axes.
- Same focal length f.
- A known distance between cameras (baseline) b.

Disparity $d = u_L - u_R$ is the difference between the horizontal coordinates in the left and right images, and depth z can be recovered as:

$$z = \frac{bf}{d}$$

The larger the disparity, the smaller the depth, and vice versa.
The vertical coordinates remain the same in the left and right images.



A point P in the world, the Focal Plane and the Image Plane seen from above.
In the two camera reference systems, point P is mapped to positions with identical vertical coordinates, while the horizontal component changes due to the baseline shift between the cameras.

2.5.2 Epipolar Geometry and Rectification

When cameras are not aligned, the correspondence search becomes complex. The epipolar constraint reduces the search to a 1D line (epipolar line), which may be skewed. **Rectification** transforms images to a canonical form where epipolar lines are horizontal, simplifying correspondence matching.

2.6 Stereo Matching and Correspondence

Finding matching points in stereo images is called **stereo matching**. Common techniques include:

- Block matching (comparing small patches).
- Searching along epipolar lines.

KITTI benchmark datasets are commonly used to evaluate such methods.

2.7 Properties of Perspective Projection

- 3D line segments project to 2D lines.
- Lengths and parallelism are not preserved. The image of a 3D line segment of length L lying in a plane parallel to the image plane at distance z from the optical centre will exhibit a length given by:
$$l = L\frac{f}{z}$$
- The image of parallel 3D lines converges to a **vanishing point**.

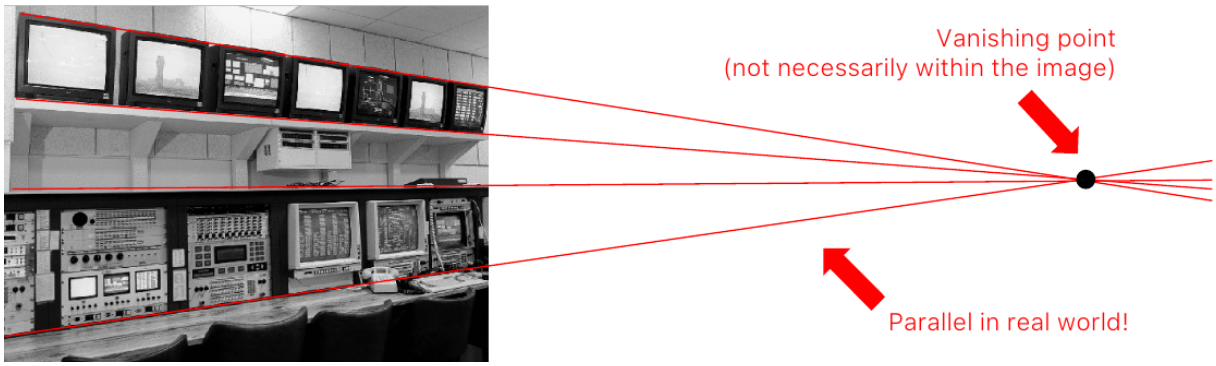


Figure 5: The images of parallel 3D lines intersect at a point, which is referred to as vanishing point.
If the lines are parallel to the image plane they meet at infinity.

2.8 Depth of Field and Lenses

In a pinhole camera, all scene points are in focus due to the small aperture, resulting in infinite Depth of Field (DOF). However, the light is limited, leading to long exposure times.

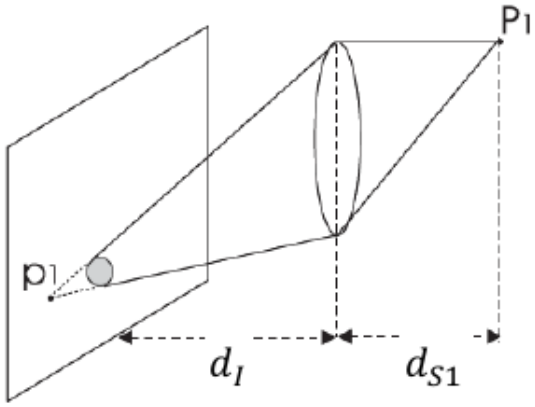
Using lenses allows more light, reducing exposure time and enabling dynamic scene capture. The downside is a **finite DOF**: only points at specific distances are in focus, others appear as blur circles (circles of confusion).

2.8.1 Thin Lens Equation

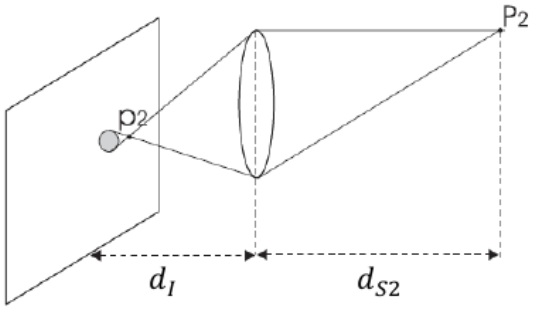
The thin lens model relates object distance d_S, image distance d_I, and focal length f_L:

$$\frac{1}{d_S} + \frac{1}{d_I} = \frac{1}{f_L}$$

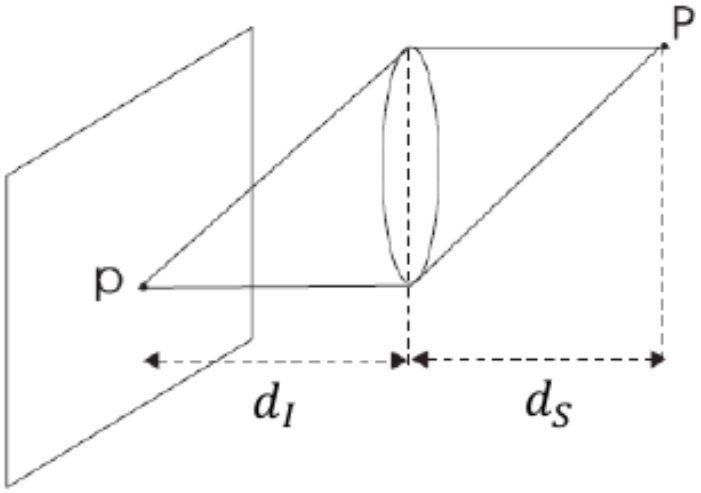
This equation allows controlling focus by adjusting the position of the lens or the image plane.



(a) $p1$ is behind the Image Plane



(b) $p2$ is in front of the Image Plane



(c) p is in focus

Figure 6: A scene point is on focus when all its light rays gathered by the camera hit the image plane at the same point

2.9 Diaphragm and DOF Control

In theory, when imaging a scene through a thin lens, only the points at a certain distance can be on focus, all the others appear blurred into circles. However, as long as such circles are **smaller** than the size of the photosensing elements, the image will still look on-focus (i.e. the light is collected by a single pixel of the camera sensor).

Cameras often deploy an adjustable diaphragm (iris) to control the amount of light gathered through the effective aperture of the lens.

The **diaphragm** controls the effective aperture:

- Smaller aperture ⇒ smaller blur circles ⇒ larger DOF.
- Larger aperture ⇒ more light ⇒ shallower DOF.

A trade-off exists between light sensitivity, motion blur, and DOF.

2.10 Focusing Mechanism

To focus on objects at different distances, lenses can shift along the optical axis. Focusing at infinity occurs when the image plane lies at distance f_L from the lens. Increasing lens-to-sensor distance allows focusing on closer objects.

2.11 Image Digitization

Analog images formed on a sensor must be:

- **Sampled:** to create a grid of pixels.
- **Quantized:** to assign discrete intensity values (gray levels).

If $N \times M$ is the image resolution and m is the number of bits per pixel, the memory required is:

$$B = N \times M \times m$$

Common values:

- $m = 8$ for grayscale $\Rightarrow$ 256 gray levels.
- $m = 24$ for color (8 bits per RGB channel).

Higher resolution and more bits per pixel improve image quality but increase storage cost.

2.12 Sensor Technology

Modern cameras use:

- **CCD** (Charge Coupled Devices): better image quality, higher SNR (*Signal to Noise Ratio*), higher DR (*Dynamic Range*).
- **CMOS** (Complementary Metal-Oxide-Semiconductor): lower power, cheaper, on-chip circuitry.

Sensors directly convert light into electric signals during exposure.

2.13 Color Sensors and Bayer CFA

Sensors are not inherently color-sensitive. To produce color images a **Color Filter Array (CFA)** is placed over the sensor. In the most common, *Bayer CFA*, green filters are twice as much as red and blue ones to mimic the higher sensitivity of the human eye in the green range.

To obtain an RGB triplet at each pixel, missing samples are **interpolated** from neighbouring pixels (demosaicking). However, the true resolution of the sensor is smaller due to the green channel being subsampled by a factor of 2, the blue and red ones by 4. True full-resolution color sensors use a beam-splitting prism and three separate sensors, though it's rare and costly.

2.14 Signal to Noise Ratio (SNR)

Signal-to-Noise Ratio (SNR) – The intensity measured at a pixel under perfectly static conditions varies due to the presence of random noise (i.e. a pixel value is not deterministic but rather a random variable).

Noise sources include:

- *Photon shot noise*: The time between photon arrivals at a pixel is governed by a Poisson statistic and thus the number of photons collected during exposure time is not constant.
- *Electronic circuitry noise*:It is generated by the electronics which reads-out the charge and amplifies the resulting voltage signal.
- *Quantization noise*: related to the final ADC conversion (in digital cameras).
- *Dark current noise*: a random amount of charge due to thermal excitement is observed at each pixel even though the sensor is not exposed to light.

The SNR can be thought of as quantifying the strength of the "true" signal with respect to the unwanted fluctuations induced by noise. (Higher SNR = better image quality). It's expressed in decibels or bits.

2.15 Dynamic Range (DR)

DR measures the range between the minimum and maximum detectable illumination:

$$DR = \frac{E_{max}}{E_{min}}$$

- E_{max} is the saturation level of the sensor.
- E_{min} is the minimum detectable light above noise.
- Higher DR enables capturing both dark and bright regions simultaneously.
- It's usually measured in decibel or bits, like SNR.

HDR imaging techniques use multiple exposures to create extended DR images.

3 Spatial Filtering and Image Denoising

3.1 Understanding and Visualizing Noise

In image acquisition, noise is introduced due to sensor imperfections or environmental conditions. It is often modeled as additive and independent from the signal:

$$I_t(p) = I(p) + n_t(p)$$

where:

- $I(p)$: true intensity at pixel p
- $n_t(p)$: noise function (e.g. a zero-mean Gaussian noise $\mathcal{N}(0, \sigma^2)$)
- t: temporal index

To reduce noise, a temporal average over multiple images can be computed:

$$O(p) = \frac{1}{T} \sum_{t=1}^{T} I_t(p) \approx I(p)$$

This averaging reduces the variance of the noise component and reveals the underlying signal.

3.2 Spatial Averaging from a Single Image

When only one image is available, spatial averaging can be used instead:

$$O(p) = \frac{1}{|S|} \sum_{q \in S} I(q)$$

This replaces the value at p with the mean of a neighborhood S around it, acting as a local denoising strategy.

3.3 Definition of Image Filters

An **image filter** is an operator applied to each pixel p based on its neighborhood S_p:

$$O(p) = f(S_p)$$

Filters perform tasks such as:

- Denoising (e.g., mean, Gaussian, median, bilateral)
- Sharpening (edge enhancement)
- Smoothing

3.4 Linear and Translation-Equivariant (LTE) Operators

An operator T is **linear** if:

$$T[\alpha i_1(x,y) + \beta i_2(x,y)] = \alpha T[i_1] + \beta T[i_2]$$

with α, β constants

It is **translation-equivariant** if:

$$T[i(x - x_0, y - y_0)] = o(x - x_0, y - y_0)$$

If both conditions hold, the operator can be represented as a **2D convolution**:

$$o(x,y) = (i * h)(x,y) = \iint i(\alpha,\beta)\, h(x-\alpha, y-\beta)\, d\alpha d\beta$$

where $h(x,y)$ is the **impulse response** (kernel) of the operator.

3.5 Properties of Convolution

- **Commutative:** $f * g = g * f$
- **Associative:** $f * (g * h) = (f * g) * h$
- **Distributive:** $f * (g + h) = f * g + f * h$
- **Differentiation:** $(f * g)' = f' * g = f * g'$

3.6 Correlation vs Convolution

Correlation does not involve reflection of the kernel:

$$(i \circ h)(x,y) = \iint i(\alpha,\beta) h(x+\alpha, y+\beta)\, d\alpha d\beta$$

If $h(x,y) = h(-x,-y)$ (i.e., symmetric), then convolution and correlation coincide. However, correlation is **not commutative**.

3.7 Discrete Convolution and Practical Implementation

In practice:

$$O(i,j) = \sum_m \sum_n K(m,n) \cdot I(i-m, j-n)$$

To compute the result:

- Slide the kernel over the image.
- Compute local sums.
- **Do not overwrite** the input image.

Border handling options:

- **CROP:** exclude borders (image shrinks)

- **PAD:** extend image with:
  - Zero-padding
  - Replication (copy edge values)
  - Reflection (mirror across edge)
  - Reflection_101 (invert first pixel after edge)

The choice of border padding affects the quality and visual continuity of the filtering, especially near image edges.

3.8 Mean Filter

Example: Mean Filter in OpenCV

Listing 1: Mean Filter with OpenCV

```
import cv2
import numpy as np

img = cv2.imread("noisy_image.png", cv2.IMREAD_GRAYSCALE)
mean_filtered = cv2.blur(img, (3, 3)) # 3x3 mean filter

cv2.imshow("Mean Filter", mean_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The **mean filter** computes the average over a neighborhood. Example 3×3 kernel:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

It performs **low-pass filtering**, removing noise and small features. Fast to compute (only sums), but blurs edges.

3.9 Gaussian Filter

Example: Gaussian Filter in OpenCV

Listing 2: Gaussian Filter with OpenCV

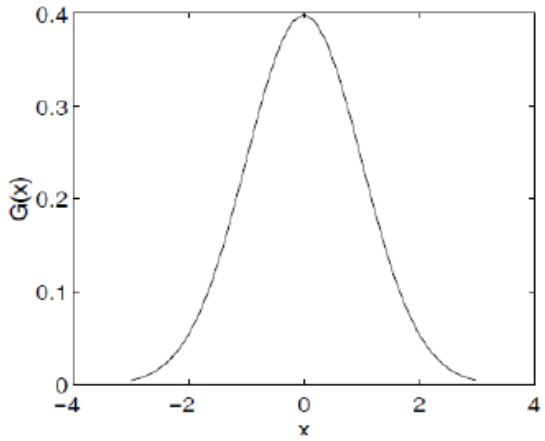
```
import cv2
import numpy as np

img = cv2.imread("noisy_image.png", cv2.IMREAD_GRAYSCALE)
gaussian_filtered = cv2.GaussianBlur(img, (5, 5), sigmaX=1.0)

cv2.imshow("Gaussian Filter", gaussian_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The **Gaussian filter** has a kernel:

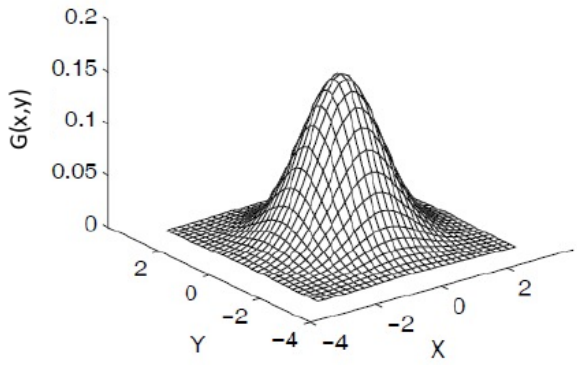
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Key points:

- Circularly symmetric.
- Requires finite-size sampling.
- Suggested size: kernel with $k = 3\sigma \Rightarrow (2k+1) \times (2k+1)$.



(a) 1D Gaussian



(b) 2D Gaussian

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \qquad G(x,y) = G(x)G(y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Figure 7: Gaussian operator visualized.

3.9.1 Separability

A 2D Gaussian can be decomposed into two 1D convolutions, This allows for more efficient convolution, reducing computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(2n)$.

$$G(x,y) = G_x(x) \cdot G_y(y)$$

Speed-up factor: $\frac{2k+1}{2+\frac{1}{k}} \approx \frac{(2k+1)}{2}$ for large k.

3.9.2 Effect of σ

Higher $\sigma \Rightarrow$ more smoothing, larger features retained, finer details lost. The parameter sets the "scale" of the filter.

3.10 Impulse Noise

(a) Original image (b) Image corrupted with impulse noise

Figure 8: **Impulse (salt-and-pepper) noise:** random black/white pixels.

Linear filtering is ineffective toward impulse noise (and blurs the image).

3.11 Median Filter

Example: Median Filter in OpenCV

Listing 3: Median Filter with OpenCV

```
import cv2
import numpy as np

img = cv2.imread("noisy_image.png", cv2.IMREAD_GRAYSCALE)
median_filtered = cv2.medianBlur(img, 3)

cv2.imshow("Median Filter", median_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The **median filter** replaces a pixel with the median of its neighbors. It is:

- Non-linear
- Effective against impulse noise, as outliers (i.e. noisy pixels) tend to fall at either the top or bottom end of the sorted intensities
- Better at preserving edges
- **Not effective for Gaussian noise.**

3.12 Bilateral Filter

Example: Bilateral Filter in OpenCV

Listing 4: Bilateral Filter with OpenCV

```
import cv2
import numpy as np

img = cv2.imread("noisy_image.png", cv2.IMREAD_GRAYSCALE)
bilateral_filtered = cv2.bilateralFilter(img, d=9, sigmaColor=75, sigmaSpace=75)

cv2.imshow("Bilateral Filter", bilateral_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The **bilateral filter** is a non-linear edge-preserving denoising filter. Each neighbor q contributes based on:

$$H(p,q) = \frac{1}{W(p)} G_\sigma(||p-q||) \cdot G_\rho(|I(p)-I(q)|)$$

Where:

- G_σ: spatial proximity weight
- G_ρ: intensity similarity weight
- $W(p)$: normalization term $\sum_{q \in S} G_{\sigma_s}(d_s(p,q))\, G_{\sigma_r}(d_r(p,q))$, with S being the pixels in the operator

Near edges, dissimilar pixels contribute little to the average.

3.13 Non-local Means Filter

Example: Non-local Means Filter in OpenCV

Listing 5: Non-local Means Filter with OpenCV

```
import cv2
import numpy as np

img = cv2.imread("noisy_image.png", cv2.IMREAD_GRAYSCALE)
nlm_filtered = cv2.fastNlMeansDenoising(img, h=10, templateWindowSize=7,
    searchWindowSize=21)

cv2.imshow("Non-local Means Filter", nlm_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Exploits patch-based similarity over the entire image:

$$O(p) = \sum_{q \in S} w(p,q) I(q)$$

Where the weights $w(p,q)$ depend on similarity between patches N_p and N_q:

$$w(p,q) \propto \exp\left(-\frac{||N_p - N_q||^2}{h^2}\right)$$

Very effective against Gaussian noise, preserves texture and detail, but is computationally expensive.

(a) Image corrupted by Gaussian Noise ($\sigma = 20$)

(b) Gaussian Filter

(c) Non-local Means Filter, N=7x7, S=21x21, h=10σ

Figure 9: Visual comparison between Gaussian Filter and Non-local Means Filter

3.14 Filter Comparison Table

Filter	**Type**	**Noise Type**	**Pros/Cons**
Mean	Linear	Gaussian	Fast; blurs edges
Gaussian	Linear	Gaussian	Smooths details; separable
Median	Non-linear	Impulse	Preserves edges; ineffective on Gaussian
Bilateral	Non-linear	Gaussian	Preserves edges; moderate cost
Non-local Means	Non-linear	Gaussian	Best quality; slow

Table 1: Comparison of common spatial filters

---

4 Edge Detection

4.1 Introduction

Edge or contour points are local features of the image that capture important semantic information. They are widely used in image analysis tasks such as foreground-background segmentation, template matching, stereo matching, and visual tracking.

Edges are defined as pixels lying between regions of differing intensity, effectively separating distinct regions.

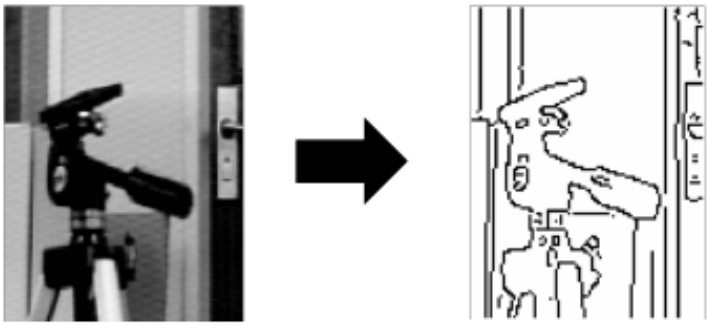


Figure 10: Edges are pixels that can be thought of as lying exactly in between image regions of different intesnity, or, in other words, to separate different image regions.

4.2 1D Step-Edge

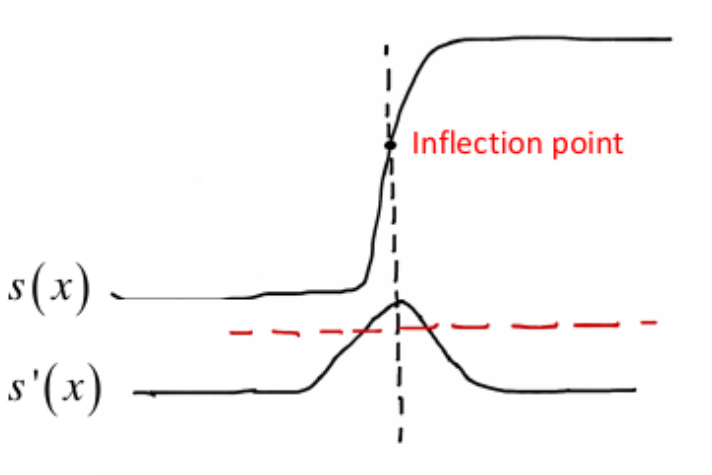


Figure 11: The simplest edge-detection operator relies on thresholding (red line in the image) the absolute value of the derivative of the signal.

A sharp change in a 1D signal represents a step-edge. In such cases:

- The first derivative becomes large in the transition region.
- The extremum of the derivative corresponds to the inflection point.

Edge detection in 1D can be implemented by thresholding the derivative.

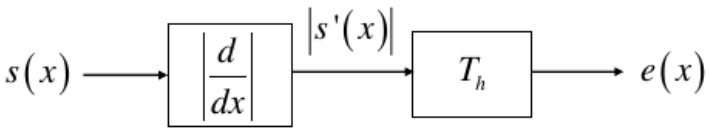


Figure 12: Pipeline of a simple 1D edge detection.

4.3 2D Step-Edge

Edges in 2D have both magnitude and direction. The **gradient** vector encodes:

- Horizontal and vertical intensity changes ($\partial I/\partial x$, $\partial I/\partial y$).
- The direction of maximum variation.

The directional derivative is computed using the dot product between the gradient and a unit vector.

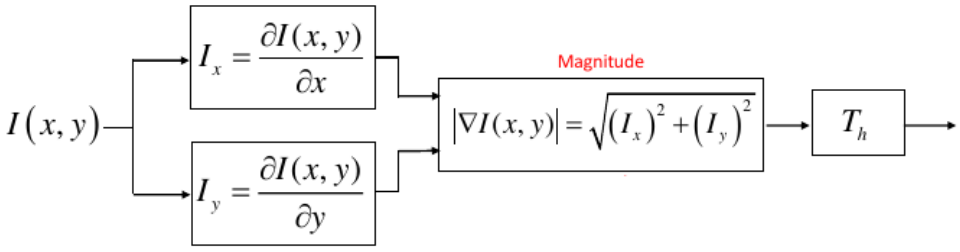


Figure 13: Pipeline of a simple 2D edge detection.

The direction of an edge is determined by the orientation of the image gradient, which is computed using the partial derivatives of the image:

$$I_x = \frac{\partial I(x,y)}{\partial x}, \qquad I_y = \frac{\partial I(x,y)}{\partial y}$$

Given the gradient vector $\nabla I = [I_x, I_y]$, the angle θ indicating the direction of the edge can be computed in two ways:

- Using the arctangent function:

$$\theta = \arctan\left(\frac{I_y}{I_x}\right)$$

  This returns values in $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, and may not correctly distinguish between directions in different quadrants.

- Using the `atan2` function:

$$\theta = \text{atan2}(I_y, I_x)$$

  This function returns values in $[0, 2\pi]$, and correctly determines the angle by taking into account the signs of both I_x and I_y, allowing full orientation information.

`atan2`, unlike the standard arctangent, provides both direction and sign (i.e., correct orientation in the full plane).

4.4 Edge Detection via Gradient Thresholding

The gradient magnitude is:

$$|\nabla I(x,y)| = \sqrt{I_x^2 + I_y^2}$$

The gradient direction is:

$$\theta(x,y) = \text{atan2}(I_y, I_x)$$

4.5 Discrete Gradient Approximation

We can approximate gradients using **backward differences**, **forward differences**, or **central differences**, even though central difference is often preferred because it ensures isotropy, meaning that the edge response does not depend on the edge orientation.

The kernel for central differences for the x-direction is: $[-1\ 0\ 1]$. While the kernel for the backward or forward differences for the x-direction is: $[-1\ 1]$.

There are several ways to approximate the magnitude $|\nabla I|$ of the gradient:

- **Exact magnitude:**

$$|\nabla I| = \sqrt{I_x^2 + I_y^2}$$

  This is the most accurate computation, but it is also the most computationally expensive due to the square root operation.

- **Manhattan (or L_1) approximation:**

$$|\nabla I| \approx |I_x| + |I_y|$$

  This method avoids the square root and provides a fast approximation, though it overestimates the gradient compared to the Euclidean norm.

- **Maximum approximation:**

$$|\nabla I| \approx \max(|I_x|, |I_y|)$$

  This is the simplest and fastest approximation, and it's invariant with respect to edge direction. It's the most isotropic of the three (horizontal, vertical, and diagonal magnitudes are equal).

4.6 Gradient and Noise

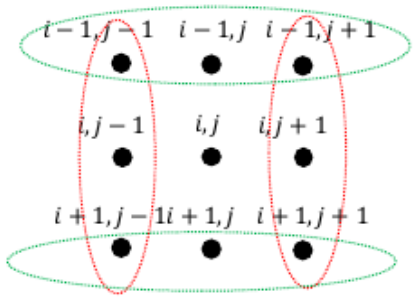
Taking derivatives amplifies noise. To deal with noise:

- Smooth the image before differentiation.
- Combine smoothing and differentiation in orthogonal directions.

4.7 Prewitt and Sobel Operators

Prewitt uses uniform weights for smoothing, while Sobel increases the weight of the center pixel:

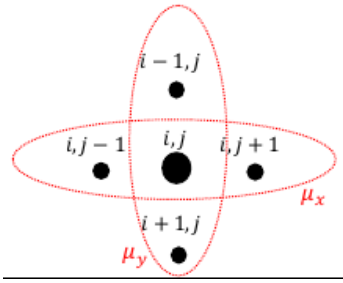
- Better edge localization.
- Still uses central differences.



$$\tilde{I}_x(i,j) = \mu_y(i,j+1) - \mu_y(i,j-1) \quad \Rightarrow \quad \frac{1}{3}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\tilde{I}_y(i,j) = \mu_x(i+1,j) - \mu_x(i-1,j) \quad \Rightarrow \quad \frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 14: **Prewitt operator**: Given the same smoothing, one might wish to approximate partial derivatives by central differences.



$$\mu_x(i,j) = \frac{1}{4}[I(i,j-1) + 2I(i,j) + I(i,j+1)]$$
$$\tilde{I}_x(i,j) = \mu_y(i,j+1) - \mu_y(i,j-1) \quad \Rightarrow \quad \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\mu_y(i,j) = \frac{1}{4}[I(i-1,j) + 2I(i,j) + I(i+1,j)]$$
$$\tilde{I}_y(i,j) = \mu_x(i+1,j) - \mu_x(i-1,j) \quad \Rightarrow \quad \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 15: **Sobel operator**: The central pixel can be weighted more.

Example: Prewitt Operator with OpenCV (manual)

Listing 6: Prewitt Edge Detection

```
import cv2
import numpy as np

img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
kernelx = np.array([[ -1, 0, 1],
                    [ -1, 0, 1],
                    [ -1, 0, 1]], dtype=np.float32)
kernely = np.array([[ 1, 1, 1],
                    [ 0, 0, 0],
                    [ -1, -1, -1]], dtype=np.float32)

grad_x = cv2.filter2D(img, cv2.CV_64F, kernelx)
grad_y = cv2.filter2D(img, cv2.CV_64F, kernely)
magnitude = cv2.magnitude(grad_x, grad_y)

cv2.imshow("Prewitt Magnitude", magnitude / magnitude.max())
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Example: Sobel Operator with OpenCV

Listing 7: Sobel Edge Detection

```
import cv2
import numpy as np
```

```
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
grad_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
grad_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
magnitude = cv2.magnitude(grad_x, grad_y)

cv2.imshow("Sobel Magnitude", magnitude / magnitude.max())
cv2.waitKey(0)
cv2.destroyAllWindows()
```

4.8 Non-Maxima Suppression (NMS)

Instead of simple thresholding:

- Retain only local maxima of gradient magnitude.
- Suppress non-maxima across gradient direction.
- Estimate off-grid values via linear interpolation.

To reduce angular aliasing, gradient directions are quantized into finer bins (e.g., 8 directions) when comparing neighboring pixels.

4.9 Edge Detection Pipeline

1. Smooth the image. 2. Compute gradient. 3. Apply NMS. 4. Threshold the result to remove weak edges.

4.10 Canny Edge Detector

The Canny detector is derived from an optimization framework that maximizes signal-to-noise ratio (SNR), ensures good localization, and minimizes multiple responses to a single edge.

Canny optimizes edge detection using:

- **Good Detection**
- **Good Localization**
- **Single Response per Edge**

Steps:

- Gaussian smoothing
- Gradient computation
- NMS along the gradient direction
- Hysteresis thresholding (dual threshold): a pixel is taken as an edge if either the gradient magnitude is higher than T_h OR higher than T_l AND the pixel is a neighbor of an already detected edge

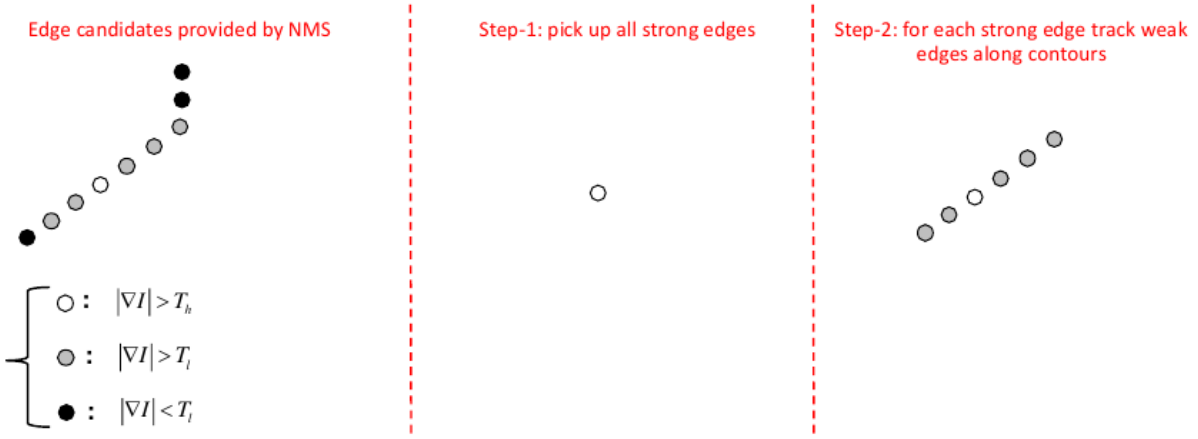


Figure 16: Canny edge detector first localizes strong edges ($> T_h$), and then includes weak edges ($> T_l$) adjacent to strong ones.

Example: Canny Edge Detection with OpenCV

Listing 8: Canny Edge Detector in OpenCV

```
import cv2
import numpy as np

img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
edges = cv2.Canny(img, threshold1=100, threshold2=200)

cv2.imshow("Canny Edges", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

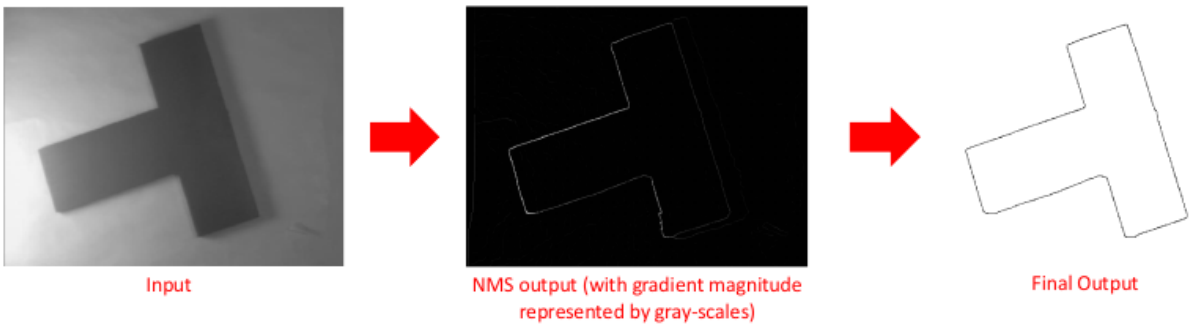


Figure 17: The Canny edge detector accurately identifies edges even in the presence of strong lighting variations. The left side of the image is highly illuminated, yet the object contours remain clearly detected.

4.11 Zero-Crossing

Instead of using the first derivative:

- Use the second derivative and detect zero-crossings.
- Zero-crossings correspond to inflection points in the signal.

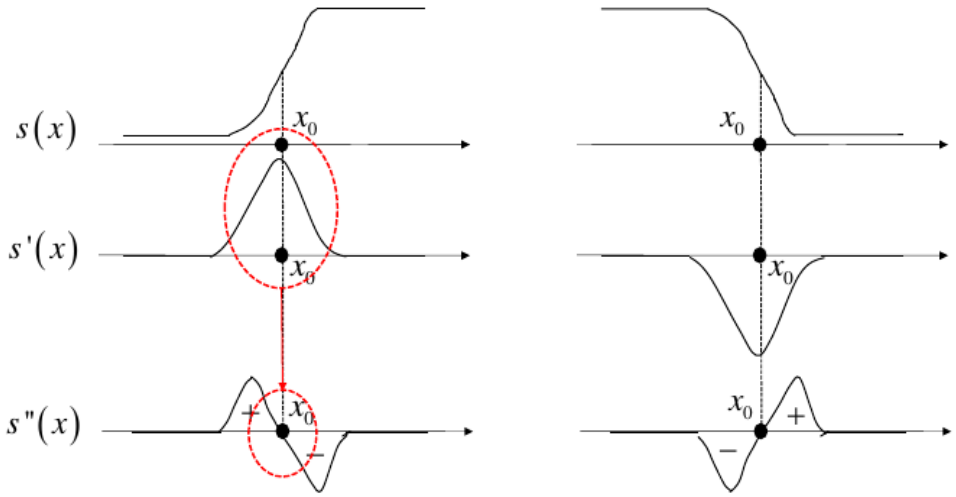


Figure 18: This approach can offer precise localization but generally involves more computation due to the calculation of second-order derivatives, which are also more sensitive to noise.

4.12 Laplacian of Gaussian (LoG)

Marr and Hildreth proposed to rely on the Laplacian as the second order differential operator:

$$\nabla^2 I(x, y) = I_{xx} + I_{yy}$$

To combine noise reduction with second-derivative edge detection, the Laplacian of Gaussian (LoG) operator is used. The process is:

- Gaussian smoothing.
- Second derivative via Laplacian.
- Detect zero-crossings in the result.

The amount of smoothing depends on the Gaussian σ, which controls noise suppression and scale.

$$\nabla^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

It can be shown that the zero-crossing of the Laplacian typically lay close to those of the second derivative along the gradient. Yet, the Laplacian is much faster to compute, i.e. just a convolution by a 3x3 kernel.

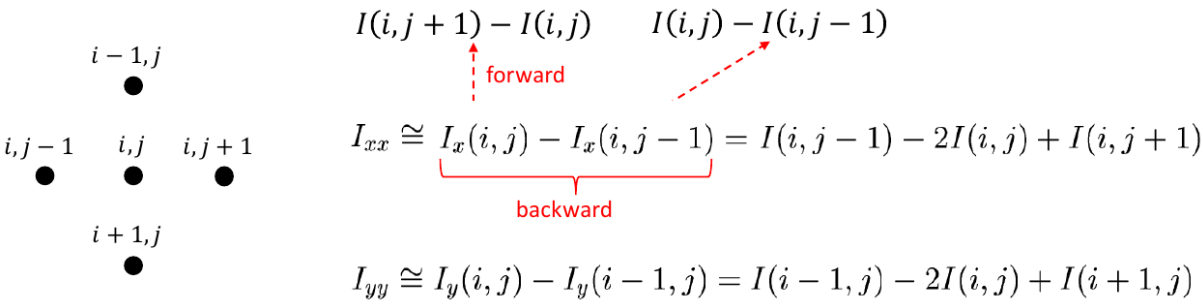


Figure 19: First and second order derivatives can be approximated using forward and backward differences, thus creating the Laplacian Operator.

In practice, the Laplacian of Gaussian can be efficiently approximated using the Difference of Gaussians (DoG) operator, which simplifies computation.

Due to the linearity of convolution and differentiation,

$$\nabla^2(I * G) = I * (\nabla^2 G)$$

Thus, one can convolve the image directly with a **pre-computed** LoG kernel $\nabla^2(I * G) = I * (\nabla^2 G)$, improving efficiency.

When a sign change (zero-crossing) is detected between adjacent pixels, the edge can be localized more precisely:

- At the pixel with the **positive** LoG value (typically corresponding to the darker side of an intensity step).
- At the pixel with the **negative** LoG value (brighter side).
- At the pixel whose LoG value has the **smallest** absolute magnitude among the pair straddling the zero-crossing. This is often preferred as it places the edge closer to the true zero-crossing point.

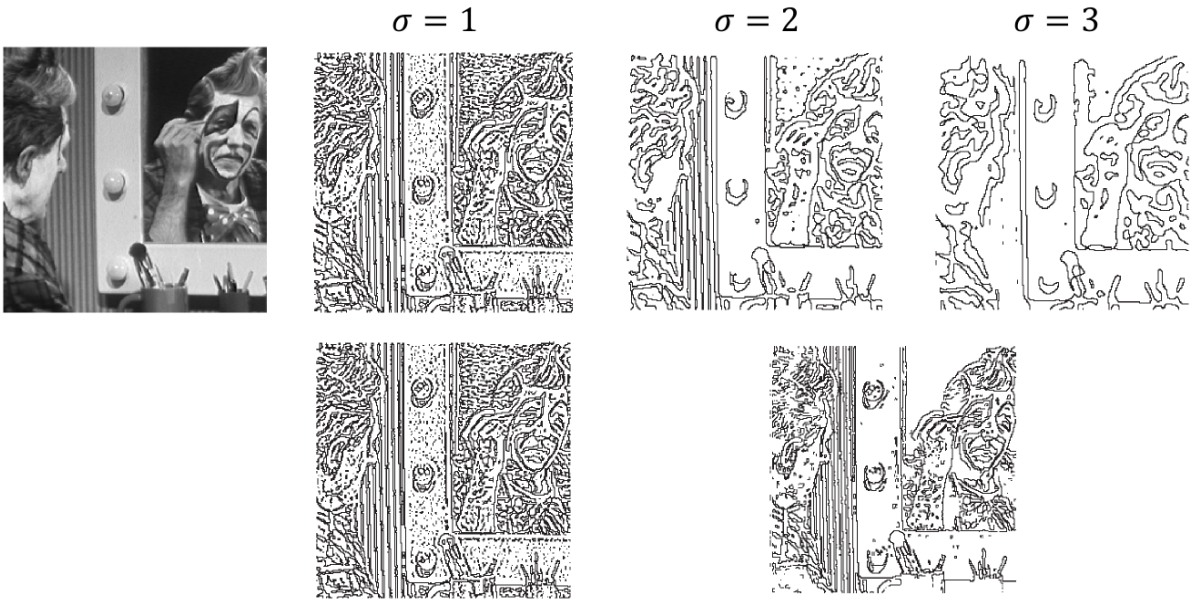


Figure 20: LOG application examples.

5 Local Invariant Features

5.1 Motivation

Local image features are essential for a variety of computer vision tasks including:

- Object detection
- Object recognition
- Image stitching
- Structure from motion

These tasks typically require:

- Robust matching of visual features
- Repeatable detection under different conditions
- Efficient descriptor computation and matching

5.2 Local Feature Detectors

A local feature is a point or region in the image that can be reliably located under a range of transformations. Ideal features should be:

- Repeatable
- Distinctive
- Localized
- Robust to illumination, scale, viewpoint, and noise

5.3 The Local Invariant Features Paradigm

The task of establishing correspondences is split into 3 successive steps:

- **Detection** of salient points (aka keypoints, interest points, feature points...)
- **Description** - computation of a suitable descriptor based on pixels in the keypoint neighbourhood
- **Matching** descriptors between images

Descriptors should be **invariant** (robust) to as many transformations as possible

5.4 Properties of Good Detectors and Descriptors

Speed is desirable for both, and in particular for detectors, which need to be run on the whole image (while decriptors are computed at keypoints only)

Good Detectors Properties

- **Repeatability**: it should find the same keypoints in different views of the scene despite the transformations undergone by the images
- **Saliency**: it should find keypoints surrounded by informative patterns (good for making them discriminative for the matching)

Good Descriptors Properties

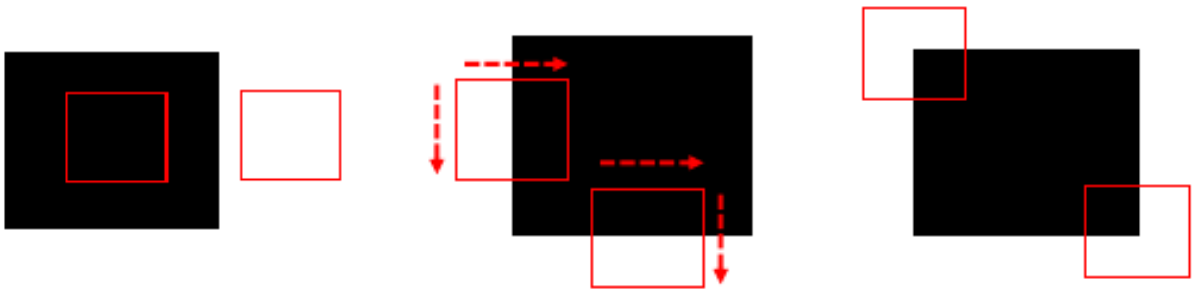
- **Distinctiveness vs. Robustness Trade-off**: the description algorithm should capture the salient information around a keypoint, so to keep important tokens and disregard changes due to nuisances (e.g. light changes) and noise by the images
- **Compactness**: the description should be as concise as possible, to minimize memory occupancy and allow for efficient matching matching)

5.5 Types of Features

- **Corners:** strong changes in intensity in orthogonal directions
- **Blobs:** local maxima/minima over scale
- **Edges:** not suitable as interest point (ambiguous along the direction perpendicular to the gradient)

Generally speaking, pixels exhibiting a large variation along all directions are better for estabilishing reliable correspondences.

5.6 Corner Detection: Moravec Interest Point Detector



(a) **Uniform region**: no change in all directions. C is small.

(b) **Edge**: no change along the edge direction. C is small.

(c) **Corner**: significant change in all directions. C is high.

Figure 21: The cornerness at p is given by the minimum squared difference between the patch (e.g. 7x7) centered at p and those centered at its 8 neighbours. $C(p) = \min_{q \in \backslash_8(p)} \|N(p) - N(q)\|^2$

After computing $C(p)$ for all pixels, one might want to keep only points of higher interest: this is achieved by **thresholding** the resulting image and then by appliyng a **Non-Maxima Suppression**. This removes points under a certain value and then keeps only the ones that represent a local maxima of the image.

5.7 Corner Detection: Harris Corner Detector

Harris corner detector is ideally a *continuous formulation* of the Moravec Interest Point Detector, and it's based on the second moment matrix M that encodes the local structure around the considered pixel:

$$M = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Where I_x, I_y are image gradients.

- The algorithm starts by considering the Sum of Squared Differences (SSD) for an infinitesimal shift $(\Delta x, \Delta y)$:

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x,y) \left(I(x+\Delta x, y+\Delta y) - I(x,y)\right)^2$$

- Using a Taylor expansion, this error function can be approximated as a quadratic form involving the image derivatives I_x and I_y:

$$E(\Delta x, \Delta y) \approx \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} M \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

  where M is the 2x2 structure matrix, which is the weighted sum of the outer product of the gradients in a local window:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Eigenvalues λ_1, λ_2 of M characterize the local structure:

- Flat region: both eigenvalues small
- Edge: one large, one small
- Corner: both large

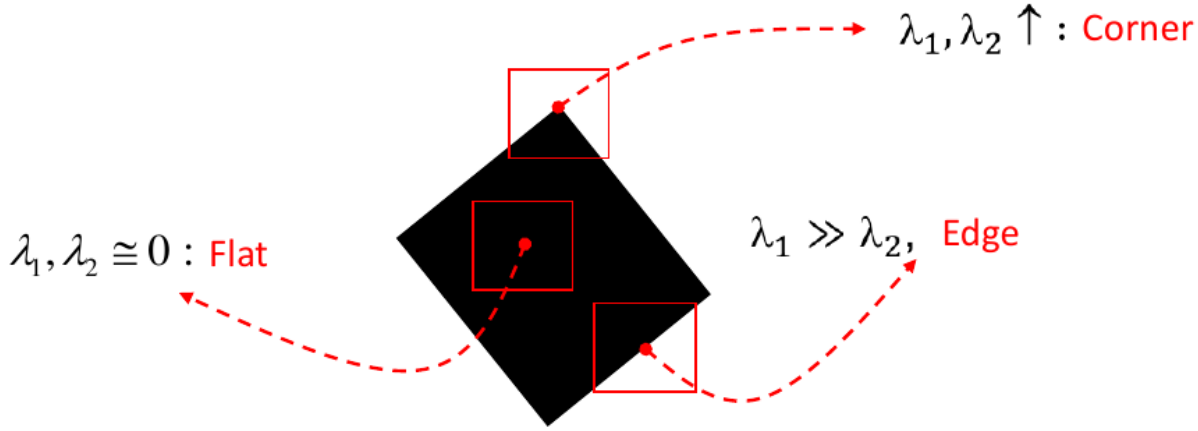


Figure 22: Different values of the eigenvalues of the matrix M in different regions of the image.

Since computing eigenvalues of the matrix M for each pixel is expensive, a more efficient corner response function is used:

$$R = \det(M) - k \cdot \text{trace}(M)^2$$

Typically $k \in [0.04, 0.06]$

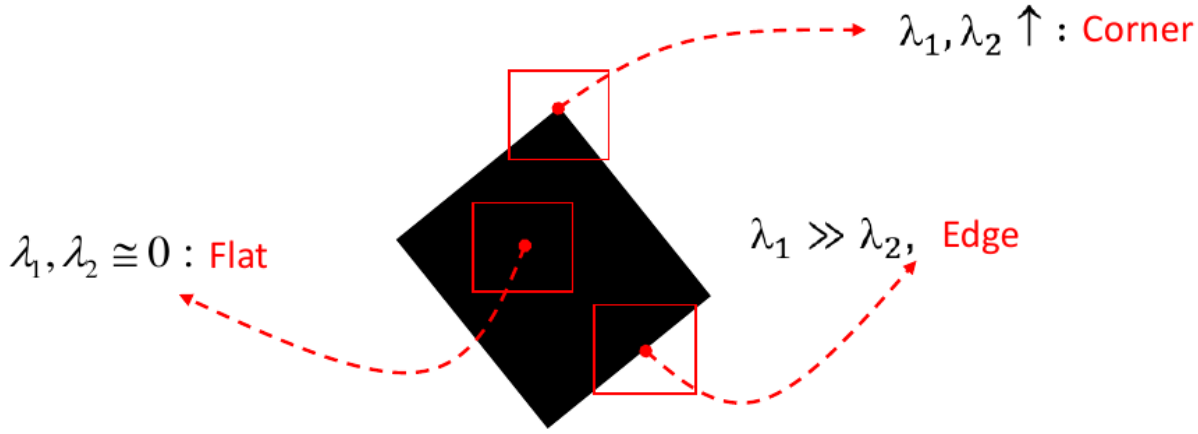


Figure 23: C is near zero in flat regions of the image, < 0 in edges and > 0 in corners.

Harris Corner Detector is **invariant to rotation**, because the eigenvalues of M are invariant to rotation.

However, it's not invariant to all affine intensity changes, such as a variations in the illumination, etc. It's **invariant to additive intensity changes** (since derivatives are unaffected), but **not to multiplicative intensity changes**, since this alters the gradients and the thresholds should be adapted. It is also **not invariant to image scale changes** (a corner might become an edge or a flat region if scaled significantly).

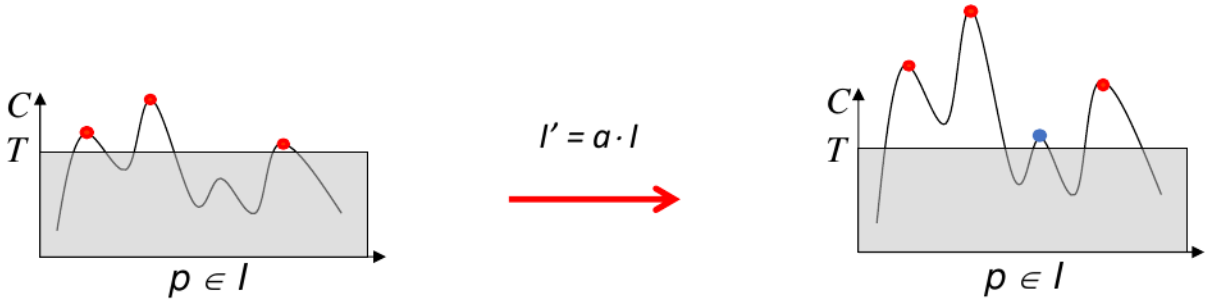
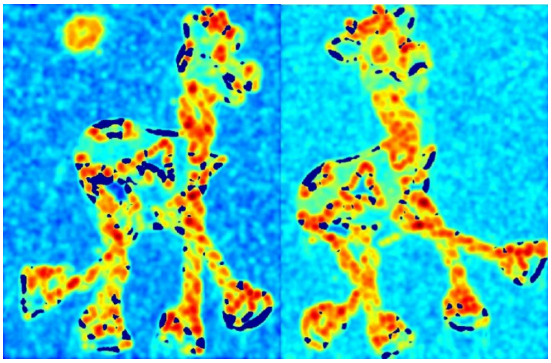
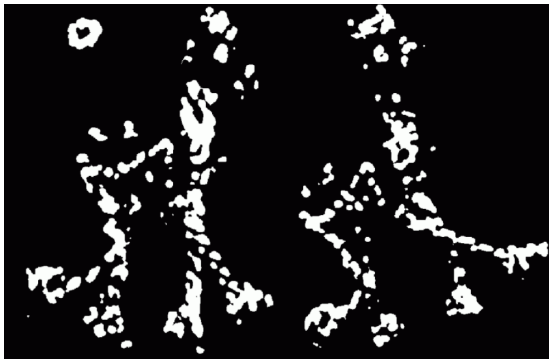


Figure 24: Multiplicative changes in the image alter the detection of corners, unless thresholds are adapted.

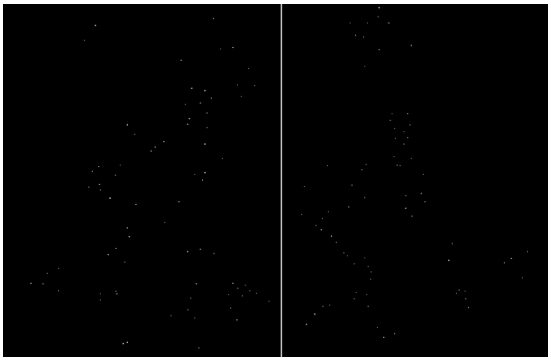
(a) Original image.



(b) Cornerness values for each pixel.



(c) Values thresholded. Only those greater than some value are kept to 1, everything else is set to 0.



(d) Result of Non-Maxima Suppression of the thresholded image.

Figure 25: Example of a Harris corner detection pipeline.

Figure 26: Corners of the images found by the previous pipeline highlighted in red.

Example: Harris Corner Detection in OpenCV

Listing 9: Harris Corner Detector with OpenCV

```
import cv2
import numpy as np

img = cv2.imread('image.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)

dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)
dst = cv2.dilate(dst, None)
img[dst > 0.01 * dst.max()] = [0, 0, 255]

cv2.imshow('Harris Corners', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5.8 Scale-Space representation

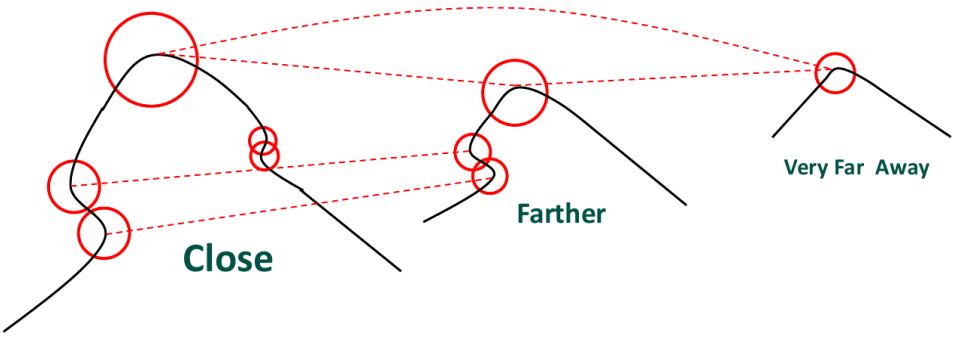


Figure 27: An image contains features at different scales, i.e. points that stand-out as interesting as long as a proper neighbourhood size is chosen to evaluate the chosen interestingness criterion

Scale invariance is a key requirement in modern local invariant feature detection. It is achieved by analyzing the image across multiple scales through a *scale-space* representation, obtained by convolving the original image with Gaussian kernels of increasing standard deviation σ:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

where $G(x, y, \sigma)$ is a 2D Gaussian kernel parameterized by σ. As σ increases, fine-scale details are progressively suppressed, while Gaussian smoothing ensures that no spurious structures (e.g., false extrema) are introduced at coarser scales.

Figure 28: A scale-space pyramid: the image is progressively blurred (increasing σ) and often downsampled to create different octaves. Within an octave, σ increases.

Scale-Space Normalization

While the scale-space represents the image across various scales, criteria are needed to detect salient features and determine their "characteristic scale" – the scale at which a feature is most prominent. Derivatives (used in many feature detectors) tend to weaken as the smoothing scale σ increases. To counteract this and enable fair comparison of feature responses across scales, Lindeberg proposed scale **normalization**: multiplying derivatives by an appropriate power of σ.

Scale-Normalized Laplacian of Gaussian (LoG)

For blob detection (salient image regions detected in both position and scale), the scale-normalized Laplacian of Gaussian (LoG) can be used. The LoG operator combines Gaussian smoothing with the Laplacian operator (∇^2). Its scale-normalized form is:

$$F_{\text{norm}}(x, y, \sigma) = \sigma^2 \nabla^2 L(x, y, \sigma) = \sigma^2 \left(\nabla^2 G(x, y, \sigma) * I(x, y) \right)$$

The σ^2 factor compensates for the natural decay of the Laplacian response with increasing σ, preventing a bias towards detecting only small-scale features. Extrema (maxima or minima) of $F_{\text{norm}}(x, y, \sigma)$ in both space (x, y) and scale (σ) indicate the centers and characteristic scales of blob-like features.

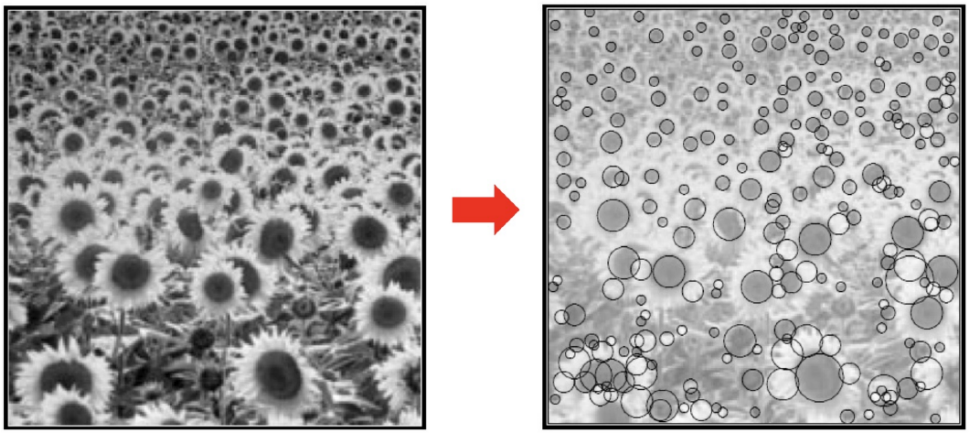


Figure 29: Blobs detected by a LoG-like filter at their characteristic scales.

5.9 Blob Detection: Difference of Gaussians (DoG)

Computing the LoG and its extrema across a finely sampled scale-sapce can be computationally intensive. Lowe, in the SIFT (Scale Invariant Feature Transform) framework, proposed using the Difference of Gaussians (DoG) function as an efficient approximation:

$$\text{DoG}(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$$

where k is a constant factor between the scales of two nearby smoothed images. The DoG function approximates the scale-normalized LoG because the difference of two Gaussian kernels, $G(x, y, k\sigma) - G(x, y, \sigma)$, approximates $(k - 1)\, \sigma^2\, \nabla^2 G(x, y, \sigma)$. Thus, local extrema of the DoG function in space and scale serve as candidate keypoints. The DoG operator, being composed of isotropic Gaussian kernels, is inherently rotation invariant.

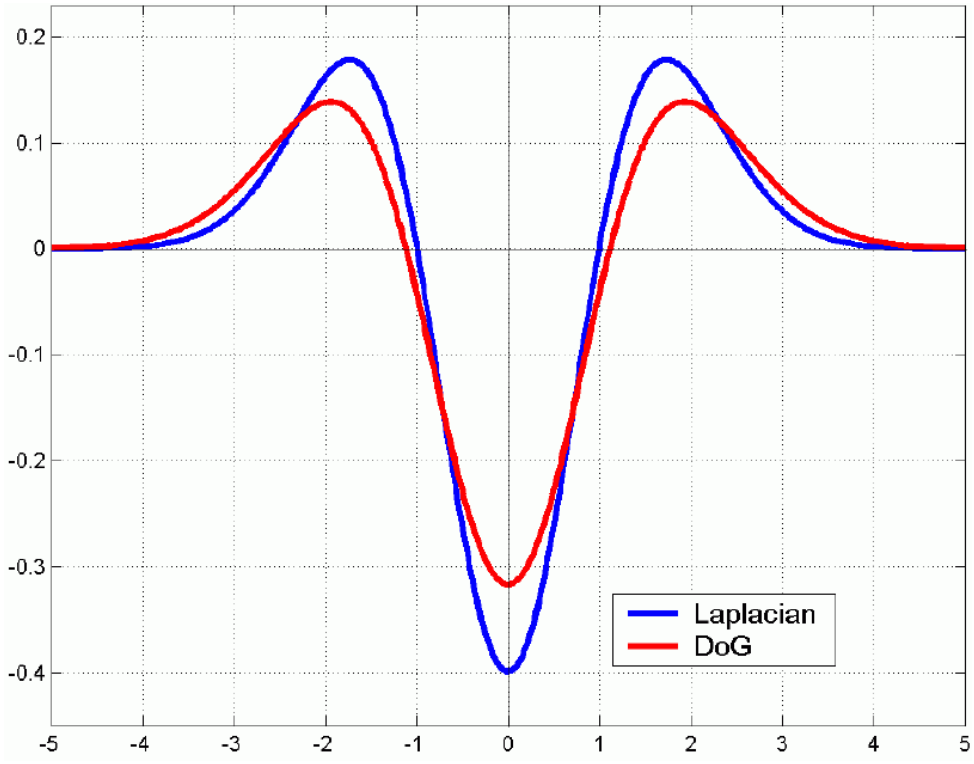


Figure 30: DoG provides a computationally efficient approximation of Lindeberg's scale-normalized LoG: $G(x, y, k\sigma) - G(x, y, \sigma) \approx (k-1)\sigma^2\nabla^2 G(x, y, \sigma)$. DoG is computationally cheaper.

The scale-space is often organized into **"octaves"**, where each octave corresponds to a doubling of σ. Within an octave, several DoG images are computed by subracting adjacent Gaussian-smoothed images. New octaves are created halving the image, taking half the of the columns and half of the rows, instead of doubling σ.

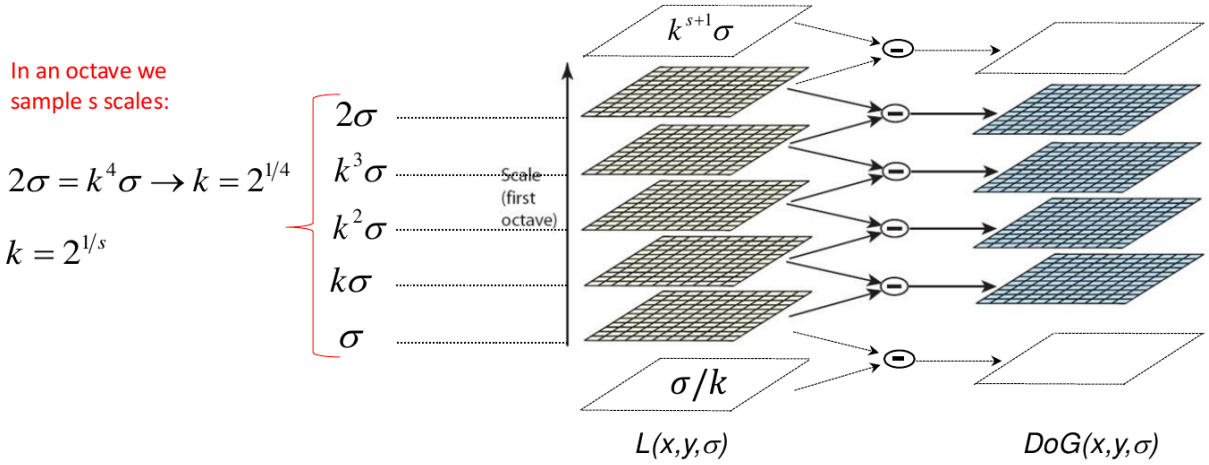
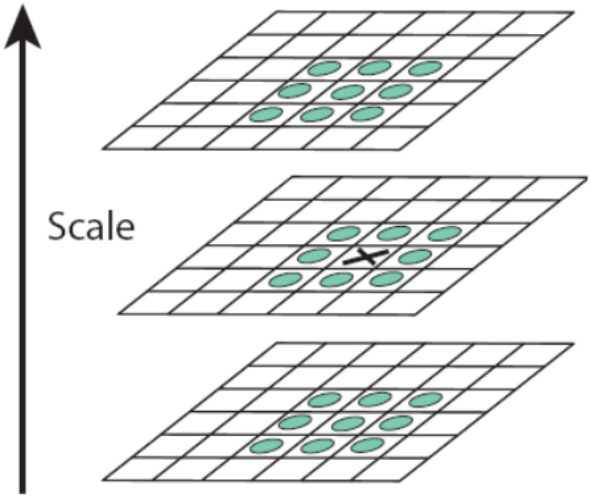


Figure 31: Extrema detection on s DoG images per octave + 2 DoG images for extrema detection on higher and lower scale.

Points (encoded as its horizontal and vertical position and its scale) are considered **extremas** if and only if its DoG is higher (or lower, for minimums) than the one of its 26 neighbours.

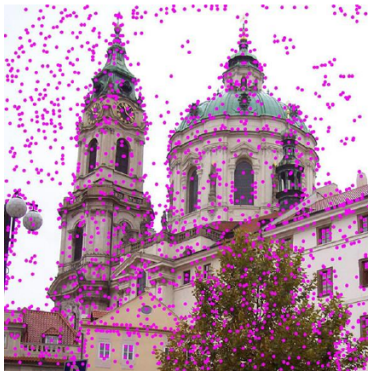
This means that to find extremas in the higher and lower scales of the octave **we need to compute two extra scales**: k^{s+1} and k^{-1}, with s being the number of scales in the octave.



Extremas are found by comparing each point with its 26 neighbours in the (x, y, σ) space: the 8 surrounding pixels in the same scale and the 9 in the same positions in the next scale and the 9 in the previous one.

Extrema Detection and Refinement

SIFT paper suggests specific parameters, such as an initial $\sigma_0 = 1.6$, $s = 3$ images to be detected per octave (requiring $s + 2$ smoothed images), and often starting by up-sampling the input image by a factor of 2 in both dimensions.



(a) **Accurate localization**: Sub-pixel and sub-scale localization is performed by fitting a 3D quadratic function to the DoG values around the candidate.

(b) **Thresholding low-contrast responses**: Keypoints with low DoG magnitudes (low contrast) are dis- carded as they are sensitive to noise.

(c) **Eliminating edge responses**: DoG produces strong responses along edges. These are unstable for localization and are removed using a method similar to the Harris corner detector, by checking the ratio of principal curvatures (eigenvalues of the Hessian matrix of the DoG function)

Figure 33: Refinement of candidate keypoints.

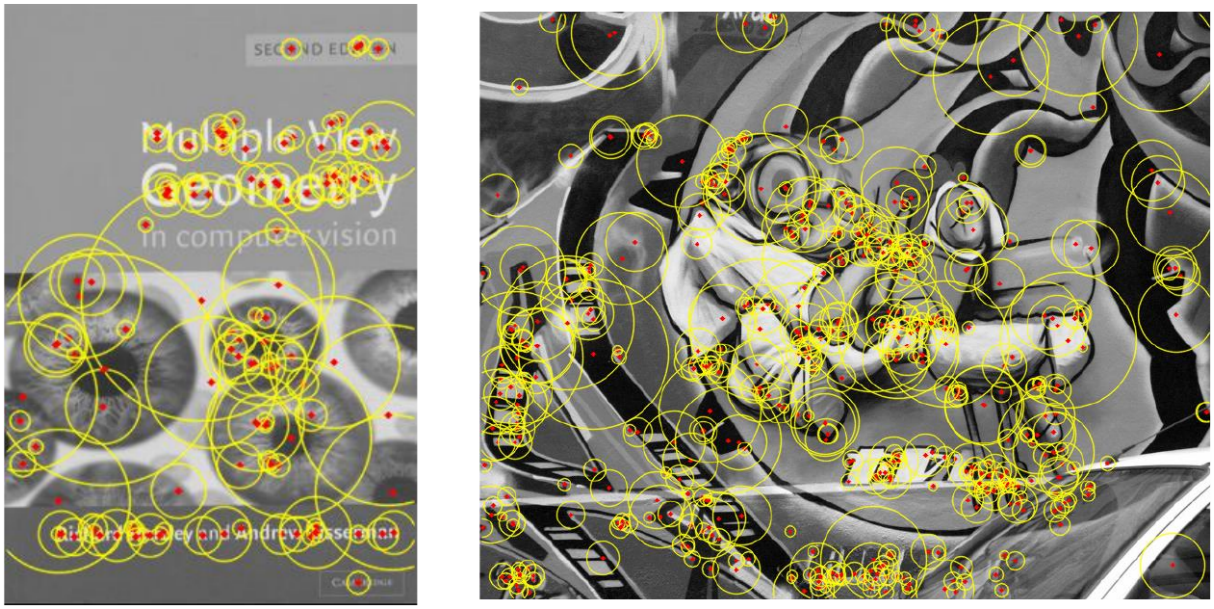


Figure 34: Keypoints detected using DoG. The size of each circle is proportional to the characteristic scale (σ) of the detected feature.

5.10 Scale and Rotation Invariance Description

Once a keypoint (with its position and scale) has been detected, we need to compute a descriptor for it that is also invariant to rotation.

- **Scale Invariance:** This is achieved by taking the image patch used for description from the Gaussian-smoothed image $L(x, y, \sigma)$ that corresponds to the keypoint's characteristic scale σ.
- **Rotation Invariance:** This is achieved by computing a **canonical orientation** for the keypoint. The descriptor is then computed on a patch that has been rotated to align with this canonical orientation. This orientation is defined relative to a new, local reference system, not the main image axes.

5.11 Canonical Orientation

Lowe proposes to compute the canonical orientation for each keypoint as follows:

1. For every pixel in a neighborhood around the keypoint (on the corresponding scale-space image L), compute the gradient magnitude and orientation.

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\theta(x, y) = \arctan^2\left(\frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)}\right)$$

2. Create an **orientation histogram** (with 36 bins, i.e., 10 degrees per bin).
3. Each pixel in the neighborhood **"votes"** for a bin corresponding to its orientation. The weight of the vote is its gradient magnitude, further weighted by a Gaussian centered on the keypoint.

4. The **canonical orientation** is the direction corresponding to the highest peak in this histogram. This is the dominant gradient direction in the patch.

5. To handle ambiguous cases (e.g., highly symmetric patterns), any other peaks that are higher than 80% of the main peak are also kept, creating multiple keypoints at the same location and scale but with different orientations.

6. Finally, a parabola is fit to the three histogram bins around each peak to get a more accurate, sub-bin estimate of the orientation:

    - Identify the histogram bin with the highest value (i.e., the peak of the orientation histogram).
    - Consider the two adjacent bins: one before and one after the peak.
    - Fit a parabola (a quadratic function) through these three points, each point being in the form: (bin position, bin value):
    - Compute the **vertex** of the parabola: this represents **the true orientation peak position**, which may lie between bins (sub-bin precision). This provides a more accurate estimate of the dominant orientation.

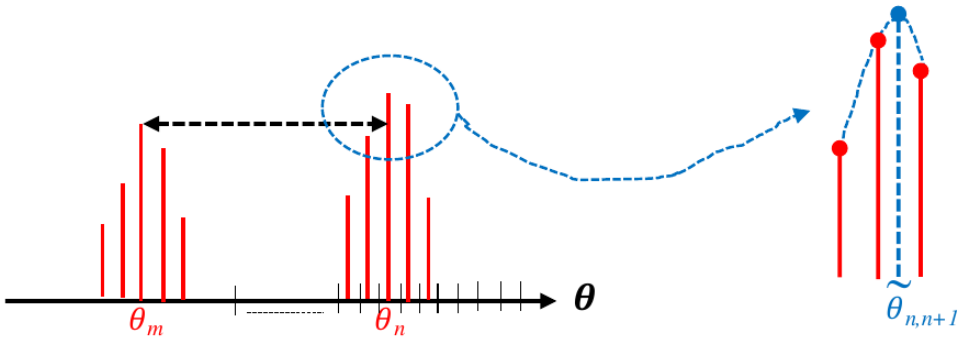


Figure 35: The true orientation of the keypoint is found by interpolating the value of the highest bin and the ones of its neighbors.

5.12 SIFT (Scale Invariant Feature Transform) Descriptor

The SIFT (Scale Invariant Feature Transform) descriptor is computed on the scale- and rotation-normalized patch:

1. A **16x16 pixel grid** is considered around the keypoint, in its canonical orientation and at its characteristic scale.

2. This grid is divided into a **4x4 grid of regions** (each 4x4 pixels).

3. For each of the 16 regions, a **gradient orientation histogram with 8 bins** is created.

4. Each of the 16 pixels within a region contributes to its 8-bin histogram, weighted by its gradient magnitude and a Gaussian centered on the keypoint. The Gaussian down-weights gradients far from the keypoint center.

5. The final descriptor is created by **concatenating** the 16 histograms, resulting in a $4 \times 4 \times 8 = \mathbf{128}$-dimensional vector.

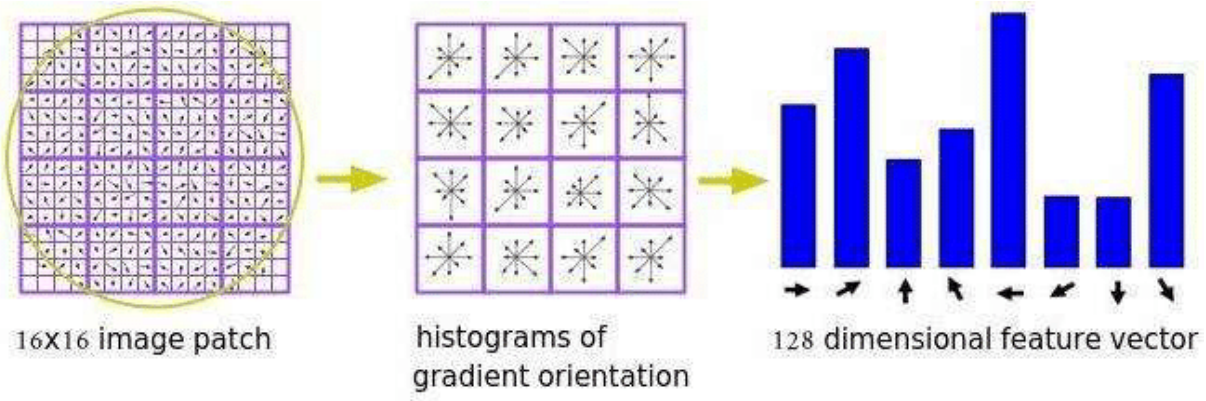


Figure 36: Construction of the descriptor of the keypoint, starting from the 16 regions around the keypoint.

5.12.1 Further Refinements

Trilinear Interpolation: To avoid hard assignments at bin boundaries, a pixel's contribution is distributed bilinearly between the 4 adjacent regions and linearly between the 2 adjacent orientation bins.

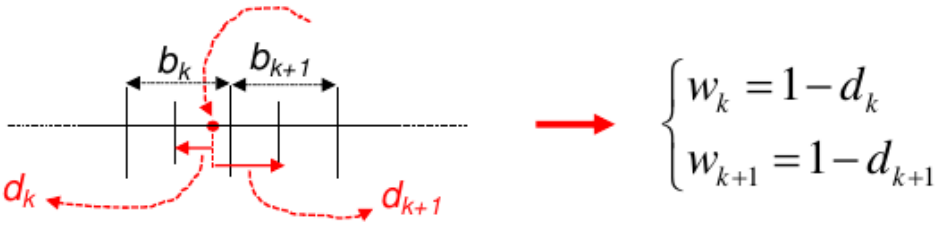


Figure 37: The contribution to two adjacent bins is weighted by the distance of the point to the bin center. Here, the point in red is inside the bin b_k, at a distance d_k from the center of the bin. Its distance d_{k+1} from the center of the bin b_{k+1} is also measured. Then, the contribution (w_k and w_{k+1}) of the point to each of the two bins is higher the nearer it is to the center of such bin.

The pixel's contribution is also **bilinearly interpolated** between the *four spatially adjacent cells* in the grid. This means that if a pixel lies near the boundary between cells, its gradient magnitude is split proportionally among those neighboring cells, including diagonal neighbors. For example, if a pixel is exactly at the corner shared by four cells, its contribution is equally divided (25%) to each of those four cells. Such bilinear interpolation avoids abrupt changes in the descriptor values when a pixel moves slightly, thus enhancing robustness.

Consequently, each pixel's gradient magnitude is distributed to up to four spatial cells **and** two adjacent orientation bins (as explained previously), leading to contributions spread over up to eight elements of the descriptor vector.

Normalization: The 128-dim vector is normalized to unit length to gain invariance to affine illumination changes (contrast). To improve robustness to non-linear changes, values are clipped at 0.2 and the vector is normalized again.

Example: SIFT with OpenCV

Listing 10: SIFT Example with OpenCV

```
import cv2

img = cv2.imread('image.png', cv2.IMREAD_GRAYSCALE)
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(img, None)

img_sift = cv2.drawKeypoints(img, keypoints, None,
                             flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow("SIFT Keypoints", img_sift)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5.13 Validating Matches

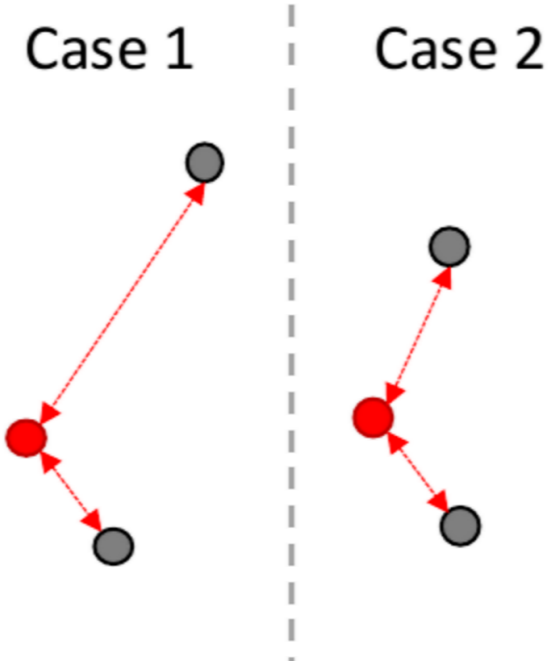
The nearest neighbor does not always provide a valid correspondence, especially in the presence of clutter or occlusion. A criterion is needed to accept or reject matches. A simple distance threshold is often insufficient.

Lowe's Ratio Test:

A much more effective criterion is to compare the distance to the **closest neighbor** (d_{NN}) with the distance to the **second-closest neighbor** (d_{2-NN}).

$$\frac{d_{NN}}{d_{2-NN}} \leq T$$

The intuition is that for a distinctive keypoint, its true match should be much closer than any other incorrect match. A low ratio indicates a good, unambiguous match. Lowe showed that a threshold of $T = 0.8$ can reject 90% of wrong matches while only losing 5% of correct ones.



The second case would be accepted as a match between the *target point* (in red) and the lower reference point (in gray) by a simple distance threshold, but not by the **Lowe's Ratio Test**, since the two reference points are at a very similar distance.
Instead, the first case passes both tests, since the *second-nearest neighbor*, compared to the nearest neighbor, is very far.

5.14 Efficient NN-Search

Exhaustively searching for the nearest neighbor has linear complexity in the number of features, which is slow for large databases.

- **Indexing Techniques** like the **k-d tree** are used to speed up the search.
- A particularly effective approximate variant for high-dimensional descriptor spaces is **Best Bin First (BBF)**.

5.14.1 SIFT Keypoint Extraction: Full Pipeline

SIFT follows these steps:

1. Build DoG pyramid at multiple scales.
2. Detect local extrema in x, y, σ.
3. Fit a 3D quadratic using Taylor expansion to refine keypoint localization at subpixel accuracy.
4. Assign orientation by dominant gradient in local neighborhood.
5. Generate descriptor from 4x4 subregions (each with 8-bin gradient histogram) $\Rightarrow$ 128D descriptor.

Descriptors are normalized to unit length for illumination invariance. After normalization, descriptor values may be thresholded (e.g., clamped at 0.2) to further reduce sensitivity to illumination changes.

5.14.2 A few other major proposals

While SIFT is foundational, other major proposals for local features exist:

- **SURF (Speeded-Up Robust Features):** A faster alternative to SIFT that uses integral images and box filters to approximate Gaussian derivatives.
- **MSER (Maximally Stable Extremal Regions):** Detects entire regions of uniform intensity, not just points.
- **FAST (Features from Accelerated Segment Test):** A very efficient corner detector.
- **Binary Descriptors (BRIEF, ORB, BRISK):** These descriptors are binary strings rather than floating-point vectors. They are very compact and can be matched extremely quickly using the Hamming distance.

5.14.3 Summary: Detector and Descriptor Comparison

Detector	Invariant To	Descriptor	Notes
Harris	Rotation	No	Not scale-invariant
LoG / DoG	Scale, rotation	No	Needs descriptor
SIFT	Scale, rotation	Yes	Robust, 128D
ORB / BRIEF	Rotation (partial)	Yes	Fast, binary

Table 2: Comparison of popular local feature detectors and descriptors

---

6 Instance-level Object Detection

Instance-level object detection aims to localize and recognize a *specific* object instance rather than a generic class. Given a **model image** M containing only the object of interest, and a **target image** I that may contain the object or clutter, the goal is to detect whether the object appears in I and, if so, estimate its **pose** (translation, rotation, and possibly scale).

For simplicity, we often consider the basic case of detecting a single object appearing once in the target image, though generalization to multiple objects or instances is straightforward. Challenges include **pose variations**, **occlusions**, **illumination changes**, and **cluttered backgrounds**.

Due to its *limited variability*—the appearance is assumed to be well represented by a single model image—this problem can often be addressed using classical computer vision techniques, with many applications in industrial vision. In contrast, **category-level object detection** seeks to find objects from a general class (e.g., cars, pedestrians) and typically requires machine or deep learning due to higher variability.

6.0.1 Template Matching

Figure 39: Template matching slides a window over the target image and compares local patches with the model/template.

Matching is based on a similarity or dissimilarity measure between the model M and each patch P in I.

6.0.2 Pixel-based Similarity Measures

- **SSD (Sum of Squared Differences):**

$$\mathrm{SSD}(i,j) \;=\; \sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\big(I(i+m,\,j+n) \;-\; T(m,n)\big)^2$$

- **SAD (Sum of Absolute Differences):**

$$\mathrm{SAD}(i,j) \;=\; \sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\big|\, I(i+m,\,j+n) \;-\; T(m,n)\,\big|$$

SSD and **SAD** are not invariant to intensity changes, such as variations in illumination. This means that they should be used for applications where the *target image* has a controlled illumination or in stable and controlled environments.

- **NCC (Normalized Cross Correlation):**

$$\mathrm{NCC}(i,j) = \frac{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m, j+n)\, T(m,n)}{\sqrt{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m, j+n)^2}\ \sqrt{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} T(m,n)^2}}$$

The numerator is the *dot product* between the template and the image patch (both flattened into vectors), while the denominator is the *product of their Euclidean norms*. This normalization makes the measure **invariant to linear changes in intensity**.

From vector algebra, the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$ can be written as

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\, \|\mathbf{b}\|\, \cos\theta,$$

where θ is the angle between them. Dividing by the product of the norms yields

$$\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\, \|\mathbf{b}\|} = \cos\theta.$$

Therefore, the NCC can be interpreted as the cosine of the angle between the flattened template vector and the corresponding image patch vector in $\mathbb{R}^{MN}$.

- **ZNCC (Zero-mean NCC):**

$$\mathrm{ZNCC}(i,j) = \frac{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} \left(I(i+m, j+n) - \mu(\tilde{I})\right) \left(T(m,n) - \mu(T)\right)}{\sqrt{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} \left(I(i+m, j+n) - \mu(\tilde{I})\right)^2}\ \sqrt{\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} \left(T(m,n) - \mu(T)\right)^2}}$$

with $\tilde{I}$ being the window of the target image at position (i,j) having the same size as T, and $\mu(\tilde{I})$ and $\mu(T)$ being the means of the model and the window $\tilde{I}$

$$\mu(\tilde{I}) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} \tilde{I}(i+m, j+n), \quad \mu(T) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} T(m,n)$$

$ZNCC$ is invariant to **affine** intensity changes (i.e. $\alpha \cdot T + \beta$)

6.0.3 Shape-based Matching

Instead of comparing raw pixel values, we can compare **gradient directions at edge points** of the model image.

First, a set of control points, P_k, is extracted from the model image by an *Edge Detector* and the gradient direction at each P_k is stored. With points positions with respect to the origin and gradient directions, a *template* is composed. Then, at each position (i,j)

of the target image, the **recorded** gradient directions associated with control points are compared to those at their corresponding image points, $P_k(i, j)$.

Note that the edge detection operation is applied only to the model, not on the target image.

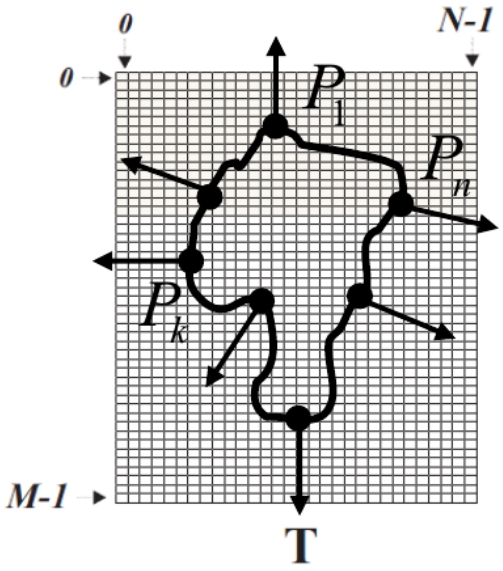


Figure 40: Gradients of the control points (and their position) are saved to create the template.

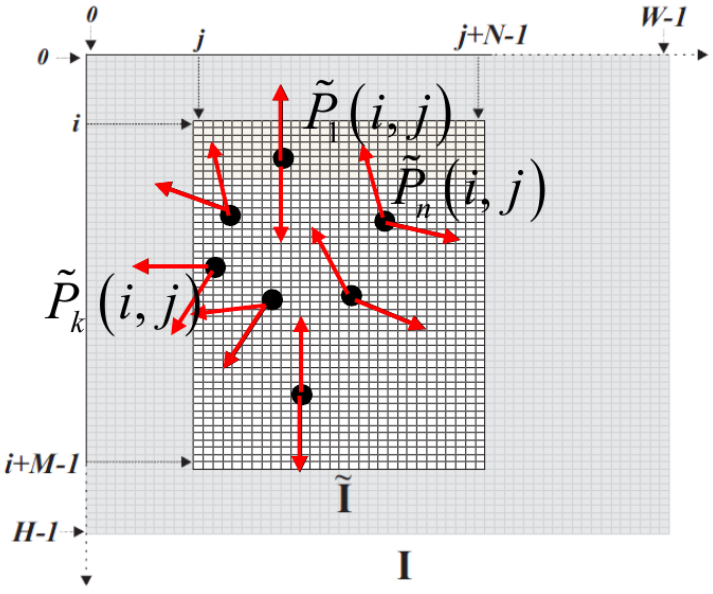


Figure 41: The *template* window is slided on the target image, and the gradients of points associated with control points are compared. Please note that in this image, to create more confusion, both the gradients of the template **and** the ones of the target image are shown in red. Thank you, prof. I lost 30 minutes on this one. Maybe I'm stupid? Could be.

Similarity functions

Similarity functions are used to match template and target points. For example, cosine similarity

$$\mathrm{S(i,j)} = \frac{1}{n}\sum_{k=1}^{n}\cos\theta_k$$

spans in the interval $[-1, 1]$. It takes its maximum value when all the gradients at the control points in the current window of the target image are perfectly aligned to those at the control points of the model image. Choosing a detection threshold, S_{min}, can be thought of as specifying the fraction of model points which must be seen in the image to trigger a detection.

This cosine similarity is robust to changes in contrast and invariant to gradient magnitude.

Certain application settings call for invariance to **global inversion of contrast polarity** along object's contours, as the object may appear either darker or brighter than the background in the target image. This kind of invariance can be achieved by a slight modification to the similarity function defined previously (global):

$$\mathrm{S(i,j)} = \frac{1}{n}\left|\sum_{k=1}^{n}\cos\theta_k\right|$$

The following function is even more robust due to the ability to withstand local contrast polarity inversions (local):

$$\mathrm{S(i,j)} = \frac{1}{n}\sum_{k=1}^{n}|\cos\theta_k|$$

6.0.4 Hough Transform (HT)

Hough Transform is a voting-based technique used to detect parametric shapes that **can be expressed by any equation** (e.g lines, circles, ellipses) based on projection of the input data into a suitable space referred to as **parameter** or **Hough space** (different from the image space).

This technique is robust to **partial occlusion**, **background clutter** and **noise**.

Steps:

1. Detect edge points in image.
2. For each edge point, vote in parameter space using an *accumulator array.*
3. Peaks in the accumulator array indicate likely presence of the object.

6.0.5 Basic Principle of the Hough Transform for Lines

The **HT** was invented to detect lines and later extended to other analytical shapes (circles, ellipses) as well as to **arbitrary shapes** (*Generalized Hough Transform*)

The classical Hough Transform (HT) provides a method to detect analytic shapes, such as straight lines, by mapping image points into a parameter space. In the usual image space interpretation, the parameters of the line equation (e.g., slope m and intercept c)

are fixed, and the equation describes all image points belonging to that line. However, the process can be inverted: by fixing the image coordinates (x, y), the line equation can be interpreted as a mapping from the image point to a curve in the parameter space. This curve represents all possible lines passing through that point.

If multiple points in the image are collinear, the curves corresponding to these points in parameter space will intersect at a single point. This intersection represents the parameters of the line on which the original image points lie. In other words, instead of looking for extended patterns in the image space, the HT searches for intersections in the parameter space, where each intersection corresponds to a potential instance of the sought shape.

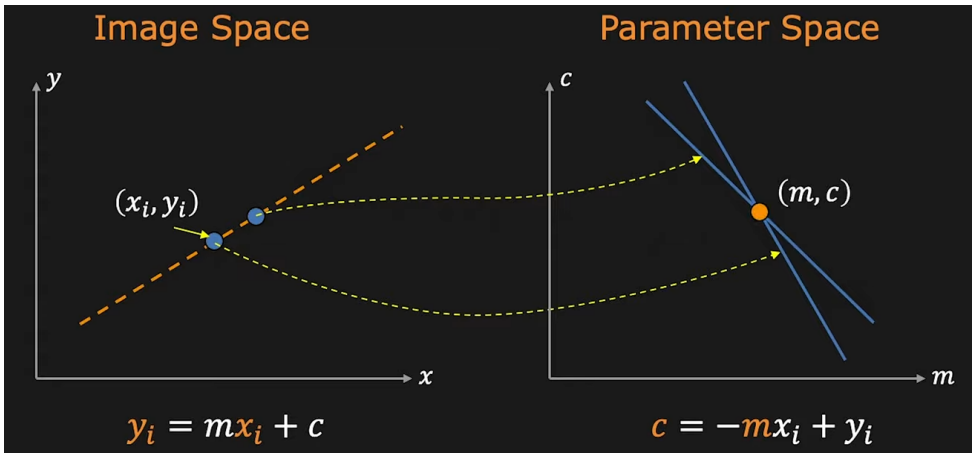


Figure 42: A point in the image space corresponds to a line in the parameter space, and vice versa. All points passing through a line $y = mx + c$ in image space form a bundle of lines all passing through the point (m, c) in parameter space.

Formally, the HT can be described as follows: given a set of image points—often extracted as edge points—the transform maps each point into a curve in the parameter space. The presence of a shape instance is indicated by many such curves intersecting at the same parameter space point. The higher the number of intersections, the stronger the evidence for that shape instance.

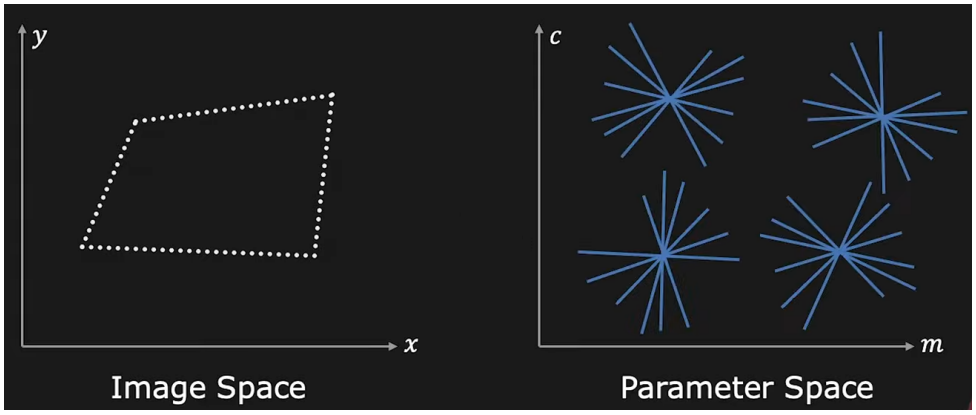


Figure 43: Different lines in image space form different intersections in parameter space.

Practical Implementation: The Voting Process

In practical implementations, the parameter space is quantized and represented as a discrete *Accumulator Array* (AA). Curves are "drawn" into the AA through a voting process:

1. For each image point, the transform equation is evaluated over the discrete parameter space, incrementing the bins that satisfy the equation.
2. Bins corresponding to points where many curves intersect will accumulate more votes.
3. Detecting shapes becomes equivalent to finding peaks (local maxima) in the AA, as these correspond to parameter values supported by many image points.

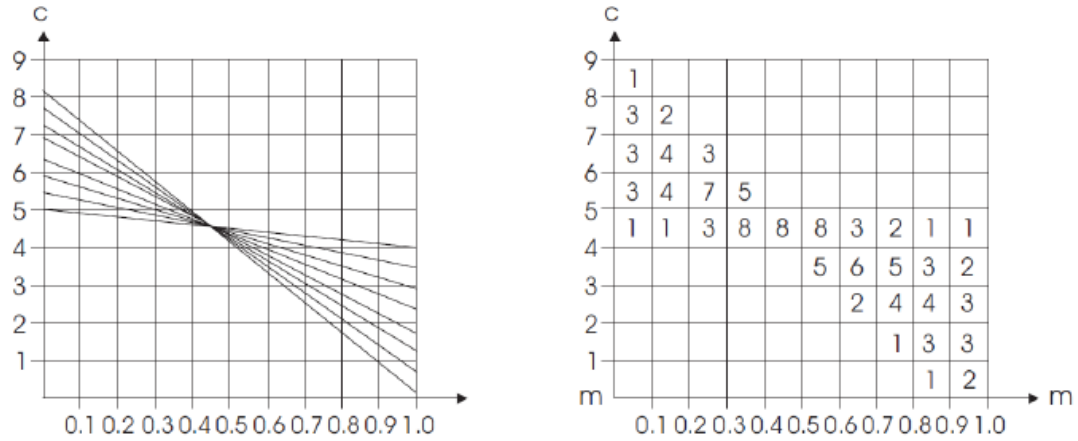


Figure 44: To detect the line more accurately, the AA should be quantized more finely.

The **HT** is **robust to noise** because spurious votes due to noise unlikely accumulate into a bin so as to trigger a false detection. Also, a **partially occluded** object can be detected provided that the threshold on the minimum number of votes required to declare a detection is lowered according to the degree of occlusion to be handled.

Normal parametrization of the line

The usual line parametrization considered so far ($y - mx - c = 0$) is impractical due to m spanning an infinite range.

A line can be parametrized as:

$$\rho = x \cos \vartheta + y \sin \vartheta$$

This representation avoids issues with infinite slopes and allows the accumulator space to be bounded.

Each edge point votes for all (ρ, ϑ) combinations consistent with its position.

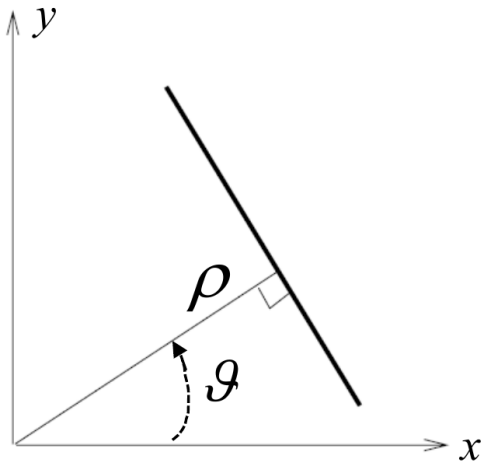


Figure 45: ϑ is the angle that the line makes with respect to the horizontal axis, and ρ is the shortest distance of the line to the origin.

With the normal parametrization: $\vartheta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, $\quad \rho \in [-\rho_{\max}, \rho_{\max}]$
with $\rho_{\max}$ usually being the length of the image diagonal ($N \times N$ pixels $\rightarrow$ $\rho_{\max} = N \cdot \sqrt{2}$)

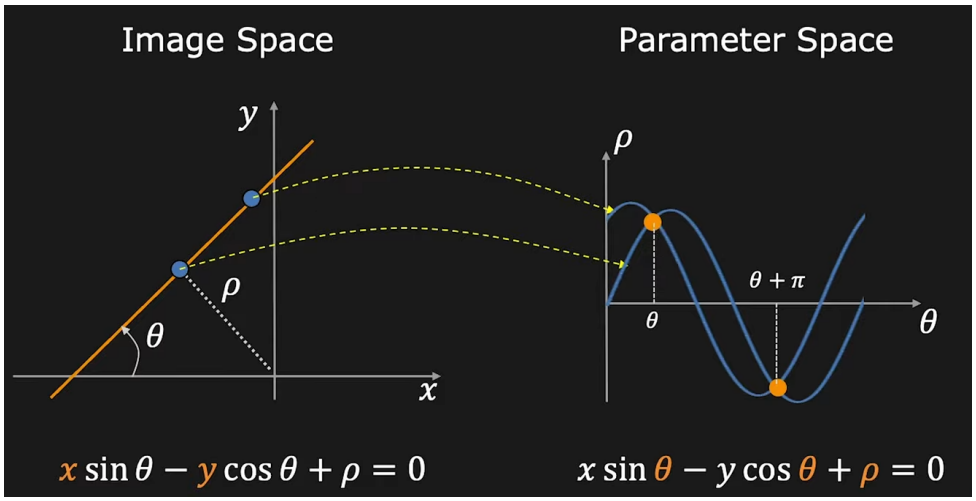


Figure 46: Image points (x, y) are mapped into sinusoidal curves of the (θ, ρ) parameter space.

6.0.6 Generalized Hough Transform (GHT)

The HT has been extended to detect arbitrary (i.e. non analytical) shapes.

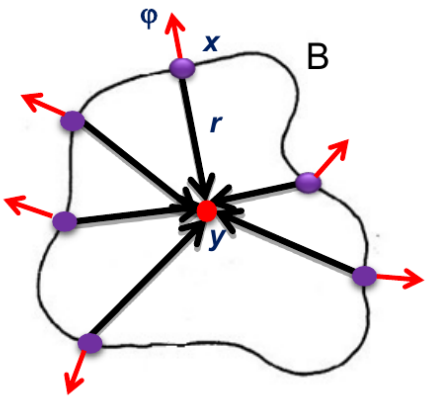


Figure 47: For each point P on the border the gradient and the vector from the reference point P are computed.

Off-line Phase (build the object's model)

1. A reference point y is chosen (e.g. barycentre)
2. For each point x belonging to object's border B:
   (a) Compute gradient direction $\phi(x)$
      i. Gradient direction is quantized according to a chosen step $\Delta\phi$
   (b) Compute vector r from y to x (i.e. $r = y - x$).
3. Store r as a function of $\Delta\phi$ (R-Table)

i	ϕ_i	R_{ϕ_i}
0	0	$\{r \mid y - r = x, x \in B, \phi(x) = 0\}$
1	$\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = \Delta\phi\}$
2	$2\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = 2\Delta\phi\}$
...	...	...

Figure 48: Example of an R-Table. It's indexed by the $\Delta\phi$ step. Each entry can contain more than one r vector.

On-line Phase (object detection)

1. We do edge detection first
2. An image $A[y]$ is initialized as accumulator array. For each edge pixel x of the input image:
   (a) Compute gradient direction ϕ
   (b) Quantize ϕ to index the R-Table. For each r_i vector stored into the accessed row:

i. Compute the position of the reference point $y = x + r_i$

ii. Cast a vote into the accumulator array $A[y] + +$

3. Instances of the sought object are detected by finding peaks of the accumulator array

Note: Traditional GHT uses edge points and gradient directions, whereas modern implementations use local invariant features (e.g., SIFT keypoints) for improved robustness and generalization.

6.0.7 Handling Scale and Rotation

Not knowing the angle at which an object could appear in an image and neither its scale means that to find it we should quantize the angle θ and its scale s and try them **all**. This way, the online phase of the algorithm will use a **4D** accumulator array (x, y, s, θ).

We also need to normalize the *joining vector* r (the one that connects the reference point of the image to each of its border points) by the scale s and the rotation ($R(\theta)$ here is the rotation matrix):

$$r' = s \cdot R(\theta) \cdot r$$

This greatly increases computation and memory requirements, but allows full invariance to similarity transforms.

7 Image Formation (Recap)

7.1 The Perspective Projection Model

Images are fundamentally 2D **perspective projections** of the 3D world. The simplest model to describe this process is the pinhole camera model.

- We consider a point in 3D space, $M = [x, y, z]^T$, whose coordinates are given in the **Camera Reference Frame (CRF)**.
- Its perspective projection onto the image plane I, denoted as $m = [u, v]^T$, is given by a set of **non-linear** equations derived from similar triangles:

$$u = -f\frac{x}{z} \quad \text{and} \quad v = -f\frac{y}{z} \qquad (1)$$

(Note: The negative signs are often omitted by assuming the image plane is in front of the optical center, but the principle remains the same).

The **Camera Reference Frame (CRF)** is a 3D coordinate system defined as follows:

- Its origin is the camera's **optical center**, C.
- The x and y axes are parallel to the horizontal (u) and vertical (v) axes of the image plane.
- The z axis is the **optical axis**, perpendicular to the image plane.

7.1.1 Why Do We Need an Image Formation Model?

A precise mathematical model of image formation is essential for a vast range of computer vision tasks. It allows us to relate 2D image measurements to the 3D structure of the world, enabling applications like 3D reconstruction from images, augmented reality, and robotics. Modern techniques like **NeRF (Neural Radiance Fields)** leverage a deep understanding of this projection model to synthesize photorealistic novel views of a scene by learning its underlying 3D representation from a set of input images.

7.1.2 Towards a More Realistic Camera Model

The simple pinhole projection model is an idealization. To create a more realistic and practical model, we must consider three additional issues:

1. **Image Reference Frame Origin:** In digital images, the coordinate system origin is conventionally in the **top-left corner**, not at the principal point (the intersection of the optical axis with the image plane). We need to account for this translation.

2. **Pixelization:** Real images are not a continuous plane but a discrete **grid of pixels**. The continuous coordinates from the projection model must be converted into discrete pixel indices. This process is called digitization or pixelization.

3. **World Reference Frame (WRF):** In most applications, we do not know the 3D coordinates of a point in the camera's own reference frame (CRF). Instead, we know its coordinates in a generic, external **World Reference Frame**. We need a way to transform points from the WRF to the CRF.

These considerations lead us to the **Complete Forward Imaging Model**, which maps a 3D point in the WRF to a 2D pixel coordinate. This mapping is governed by two sets of parameters:

- **Extrinsic Parameters:** Describe the camera's pose (position and orientation) in the world. They define the transformation from WRF to CRF.
- **Intrinsic Parameters:** Describe the camera's internal geometry (focal length, principal point, pixel size). They define the transformation from CRF to pixel coordinates.

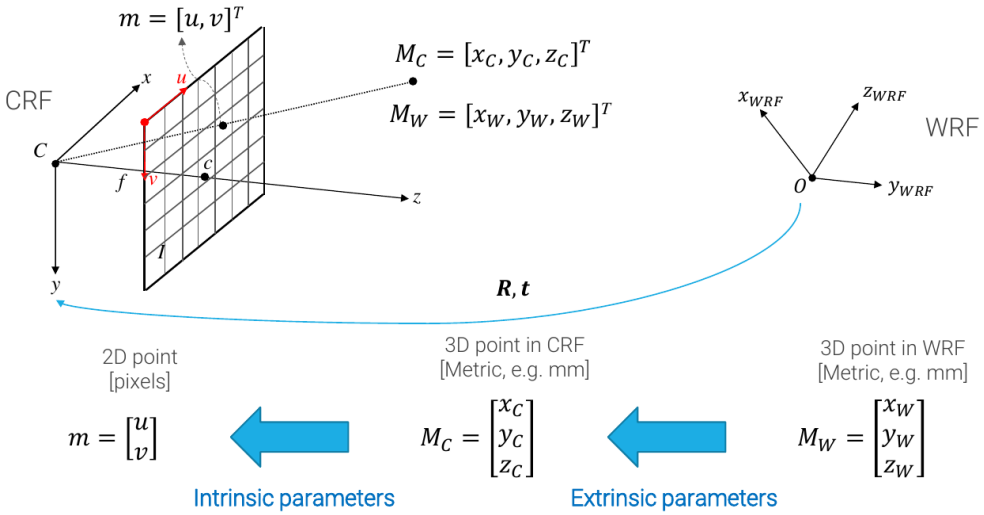


Figure 49: The dual role of camera parameters in the image formation process. Extrinsic parameters (R,t) define the camera's pose in the world (WRF to CRF), while intrinsic parameters model the camera's internal geometry and optics (CRF to 2D pixels).

7.2 Modeling Intrinsic Parameters

7.2.1 Image Pixelization

The conversion from metric units (e.g., mm) on the continuous image plane to discrete pixel units is modeled by introducing the **pixel size**, $(\Delta u, \Delta v)$. These represent the horizontal and vertical size of a single pixel (the quantization step).

- The ideal projection equations give coordinates u, v in mm.
- The pixel coordinates are obtained by dividing by the pixel size.

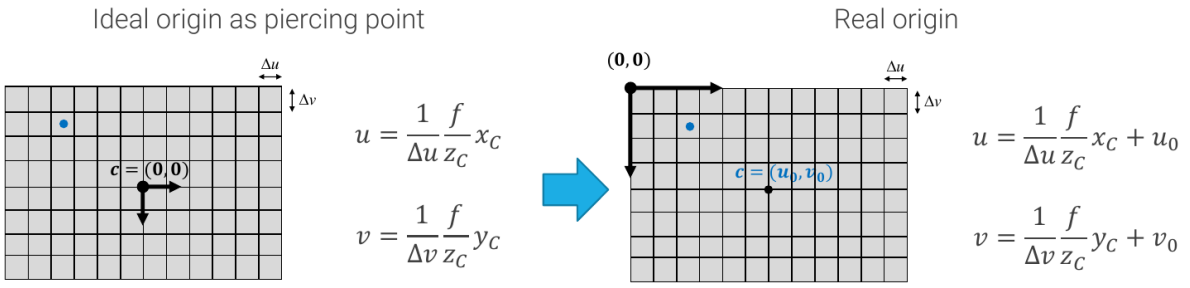
$$u_{\text{pixels}} = \frac{u_{\text{mm}}}{\Delta u} = \frac{1}{\Delta u}\frac{f}{z_C}x_C \quad \text{and} \quad v_{\text{pixels}} = \frac{v_{\text{mm}}}{\Delta v} = \frac{1}{\Delta v}\frac{f}{z_C}y_C$$

(Note: Signs are omitted for clarity, assuming a frontal image plane).

7.2.2 Origin of the Image Reference Frame

Since pixel indices are non-negative, the origin of the image coordinate system is the top-left corner. The ideal origin, the **piercing point** $c = (u_0, v_0)$, is the intersection of the optical axis with the image plane. We must account for the translation from the piercing point to the top-left corner.

$$u = \frac{1}{\Delta u}\frac{f}{z_C}x_C + u_0 \quad \text{and} \quad v = \frac{1}{\Delta v}\frac{f}{z_C}y_C + v_0$$



7.2.3 The 4 Intrinsic Parameters

It is common practice to group the multiplicative factors into two new parameters:

- $f_u = \frac{f}{\Delta u}$: The focal length measured in units of horizontal pixels.
- $f_v = \frac{f}{\Delta v}$: The focal length measured in units of vertical pixels.

The final relationship between 3D coordinates in the CRF and the corresponding pixel coordinates is then:

$$u = f_u\frac{x_C}{z_C} + u_0 \quad \text{and} \quad v = f_v\frac{y_C}{z_C} + v_0$$

This gives us a total of **4 intrinsic parameters**: f_u, f_v, u_0, v_0. These parameters define the camera's internal geometry and are independent of its position in the world.

7.3 Modeling Extrinsic Parameters

The relationship between the external World Reference Frame (WRF) and the Camera Reference Frame (CRF) is a rigid 3D motion, or **Roto-Translation**. This transformation is defined by:

- A **rotation**, represented by a 3x3 rotation matrix R.
- A **translation**, represented by a 3x1 translation vector t.

The transformation of a point M_W from the WRF to M_C in the CRF is given by:

$$M_C = \begin{bmatrix} x_C \\ y_C \\ z_C \end{bmatrix} = RM_W + t = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_W \\ y_W \\ z_W \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$m = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_u \dfrac{r_{11}x_W + r_{12}y_W + r_{13}z_W + t_1}{r_{31}x_W + r_{32}y_W + r_{33}z_W + t_3} + u_0 \\ f_v \dfrac{r_{21}x_W + r_{22}y_W + r_{23}z_W + t_2}{r_{31}x_W + r_{32}y_W + r_{33}z_W + t_3} + v_0 \end{bmatrix} \tag{1}$$

A valid rotation matrix R is an **orthonormal matrix**, meaning its rows and columns are mutually orthogonal and have unit length. This implies that $R^T R = RR^T = I$, so its inverse is simply its transpose.

The rotation matrix R has 9 entries but only **3 independent parameters** (degrees of freedom), corresponding to the rotation angles around the three axes. The translation vector t has **3 independent entries**. This gives a total of **6 extrinsic parameters** that encode the camera's pose relative to the world.

7.4 The Need for a Linear Model: Projective Space

Combining the intrinsic and extrinsic equations results in a **non-linear model**, which is cumbersome to use and estimate.

$$u = f_u \frac{r_{11}x_W + r_{12}y_W + r_{13}z_W + t_1}{r_{31}x_W + r_{32}y_W + r_{33}z_W + t_3} + u_0$$

Furthermore, this Euclidean model cannot handle **points at infinity**. In the real world, parallel lines appear to meet at a **vanishing point** on the horizon. This vanishing point is the projection of the lines' shared "point at infinity".

To address these two goals—(1) creating a linear model and (2) handling points at infinity—we introduce the framework of **Projective Space**.

7.4.1 Homogeneous Coordinates

We move from Euclidean space $\mathbb{R}^n$ to Projective Space $\mathbb{P}^n$ by appending one extra coordinate, called the homogeneous coordinate.

- A 2D Euclidean point $[u, v]$ becomes the 3D homogeneous vector $\tilde{m} = [u, v, 1]$.
- A point in projective space is defined by an **equivalence class** of vectors, where all vectors on the same line through the origin are equivalent.

$$\tilde{m} = [u, v, 1] \equiv [ku, kv, k] \quad \forall k \neq 0$$

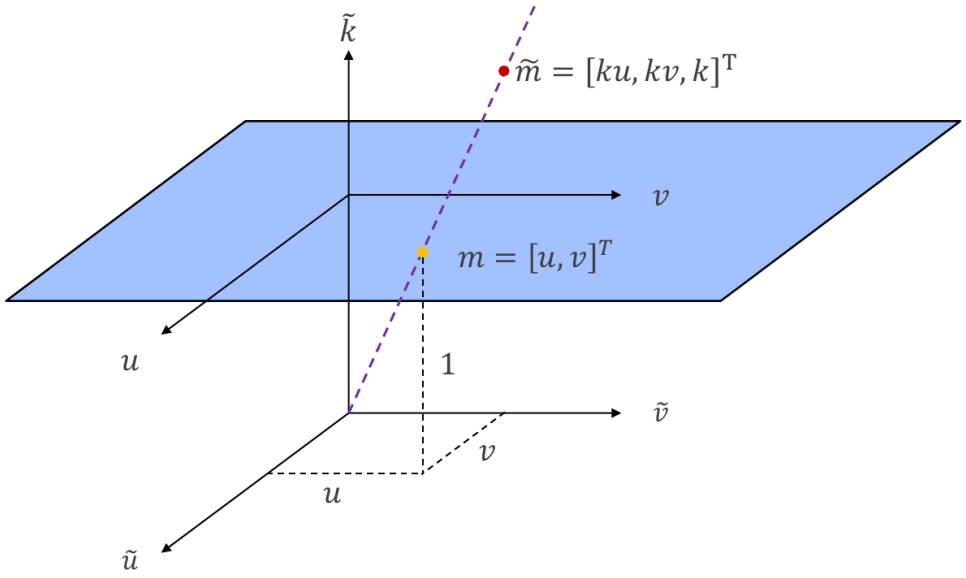


Figure 50: Geometrically, we have mapped the 2D plane onto a canonical plane at $k = 1$ in a 3D space. Every point on the 2D plane now corresponds to a unique ray passing through the origin in this 3D space.

7.4.2 Points at Infinity

This representation gives us a natural way to handle points at infinity.

- A point at infinity corresponds to a direction. It can be found by taking the limit of a point moving along a line.
- In homogeneous coordinates, the point at infinity of a line with direction $d = [a, b]^T$ is $\tilde{m}_\infty = [a, b, 0]^T$.
- All points with the final coordinate $k = 0$ (except for the origin $[0, 0, 0]$) are **points at infinity**. They form the **line at infinity** in $\mathbb{P}^2$.

The same principle extends to 3D space: a Euclidean point $[x, y, z]$ becomes the homogeneous vector $\tilde{M} = [x, y, z, 1] \in \mathbb{P}^3$. Points at infinity have the form $[x, y, z, 0]$ and form the plane at infinity.

7.5 Perspective Projection in Homogeneous Coordinates

The primary motivation for introducing homogeneous coordinates was to linearize the non-linear perspective projection equations.

$$u = f_u \frac{x_C}{z_C} + u_0 \quad \text{and} \quad v = f_v \frac{y_C}{z_C} + v_0$$

If we express the 3D point $M_C = [x_C, y_C, z_C]^T$ and the 2D image point $m = [u, v]^T$ in homogeneous coordinates, the mapping becomes a **linear transformation** via matrix multiplication.

By converting the equations to homogeneous form and multiplying all entries by z_C, we get:

$$\begin{bmatrix} z_C u \\ z_C v \\ z_C \end{bmatrix} = \begin{bmatrix} f_u x_C + u_0 z_C \\ f_v y_C + v_0 z_C \\ z_C \end{bmatrix} = \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_C \\ y_C \\ z_C \\ 1 \end{bmatrix}$$

This can be written as:

$$\tilde{m} \equiv k\tilde{m} = P_{int}\tilde{M}_C$$

where $\tilde{m}$ and $\tilde{M}_C$ are the homogeneous representations of the image and camera points, respectively, and $k = z_C$ is the projective scaling factor. The relation is an equivalence ($\equiv$) because projective coordinates are defined up to a scale factor.

7.5.1 Projecting Points at Infinity (Vanishing Points)

With this linear model, we can now easily project points at infinity to find their corresponding vanishing points on the image plane. A point at infinity in 3D has the homogeneous form $[a, b, c, 0]^T$, representing a direction. Its projection is:

$$\tilde{m}_\infty = P_{int} \begin{bmatrix} a \\ b \\ c \\ 0 \end{bmatrix} = \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 0 \end{bmatrix} = \begin{bmatrix} f_u a + u_0 c \\ f_v b + v_0 c \\ c \end{bmatrix}$$

To get the Euclidean coordinates of the vanishing point m_∞, we divide by the third component:

$$m_\infty = \begin{bmatrix} f_u \frac{a}{c} + u_0 \\ f_v \frac{b}{c} + v_0 \end{bmatrix}$$

This confirms the clear relationship: the **vanishing point** is the projection of the **point at infinity**.

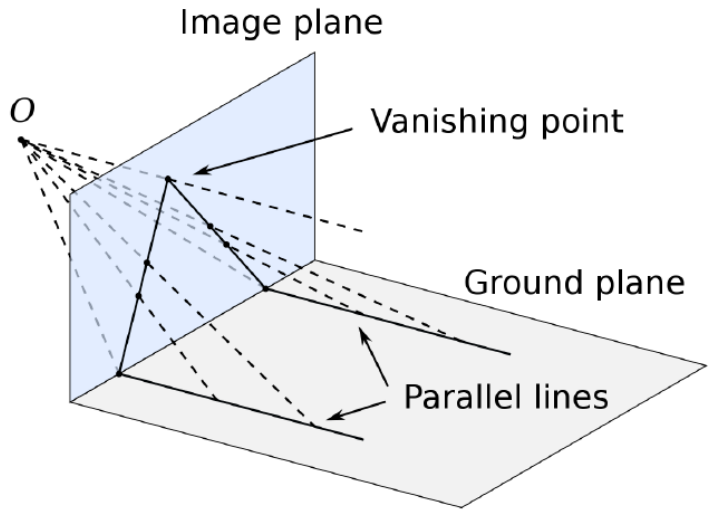


Figure 51: The **vanishing point** is the 2D point on the image where the 3D points at infinity of a set of parallel lines are projected by the camera.

7.5.2 The Intrinsic Parameter Matrix (A)

The 3x3 leftmost part of the P_{int} matrix is called the **intrinsic parameter matrix**, usually denoted as A or K.

$$A = \begin{bmatrix} f_u & 0 & u_0 \\ 0 & f_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix models the intrinsic characteristics of the camera and sensor. It is always an **upper-right triangular matrix**. A more general model could include a 5th intrinsic parameter, the **skew** coefficient, at position A[1,2]. This accounts for non-orthogonality between the sensor axes, but it is usually assumed to be 0 in practice. The projection from CRF to the image plane can now be written concisely as:

$$\tilde{m} \equiv P_{int}\tilde{M}_C = [A|\mathbf{0}]\tilde{M}_C$$

7.5.3 The Extrinsic Parameter Matrix (G)

Similarly, the rigid roto-translation from the World Reference Frame (WRF) to the Camera Reference Frame (CRF) can be expressed as a single 4x4 matrix multiplication in homogeneous coordinates. This is the **extrinsic parameter matrix**, G.

$$\tilde{M}_C = \begin{bmatrix} x_C \\ y_C \\ z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} x_W \\ y_W \\ z_W \\ 1 \end{bmatrix} = G\tilde{M}_W$$

7.6 The Perspective Projection Matrix (PPM)

By putting everything together, we obtain a single, completely **linear model** for the entire image formation process, from a 3D point in the world to a 2D point on the image.

$$\tilde{m} \equiv P_{int}\tilde{M}_C = P_{int}(G\tilde{M}_W) = (P_{int}G)\tilde{M}_W = P\tilde{M}_W$$

The resulting 3x4 matrix P is known as the **Perspective Projection Matrix (PPM)**.

7.6.1 Factorization of the PPM

The PPM can be factorized in several useful ways that reveal the underlying geometric operations:

1. **Canonical Perspective Projection:** The most basic PPM is $P = [I|\mathbf{0}]$. This form performs the core projection operation: scaling the lateral coordinates (x, y) by the distance from the camera (z).
2. **General Factorization:** A generic PPM can be seen as carrying out three fundamental operations in sequence:
   (a) Convert coordinates from WRF to CRF (apply G).
   (b) Perform canonical perspective projection.
   (c) Apply camera-specific intrinsic transformations (apply A).

This leads to the factorization $P = A[I|\mathbf{0}]G$.

3. **Common Factorization:** Substituting the definition of G, we get the most common factorization:
$$P = A[I|\mathbf{0}] \begin{bmatrix} R & t \\ \mathbf{0}^T & 1 \end{bmatrix} = A[R|t]$$

7.7 Lens Distortion

The PPM is based on the ideal pinhole camera model. Real lenses, especially cheap or wide-angle ones, introduce non-linear **distortions**. These are modeled through additional parameters that do not alter the linear form of the PPM but are applied as a separate, non-linear step.

Figure 52: Lens distortion effects are not always highly visible: in this example, straight lines such as the column are slightly curved.

7.7.1 Types of Distortion

- **Radial Distortion:** The most significant type, caused by the "curvature" of the lens. It displaces pixels radially from the image center.
  - **Barrel distortion:** Straight lines in the world appear to bend outwards. Common in wide-angle lenses.
  - **Pincushion distortion:** Straight lines appear to curve inwards. Common in telephoto lenses.

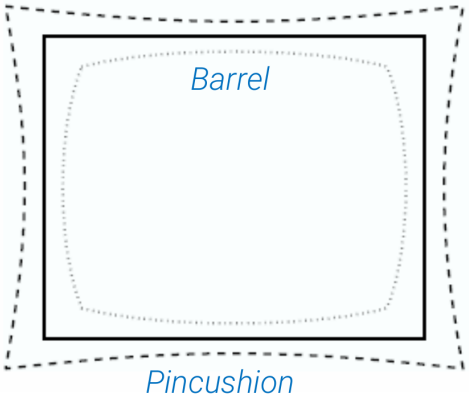


Figure 53: Barrel and Pincushion distortions visualized.

- **Tangential Distortion:** A second order effect caused by misalignments of optical components.

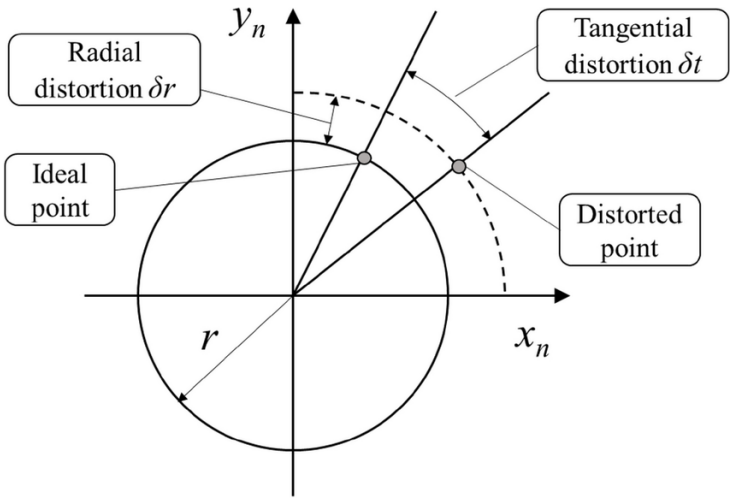


Figure 54: Tangential distortion moves the point along the circumference.

7.7.2 Modeling Lens Distortion

Lens distortion is modeled as a non-linear transformation that maps ideal, **undistorted** image coordinates to the observed, **distorted** coordinates. The total distortion is a sum of radial and tangential components, which are influenced by the distance r from the distortion center (usually the principal point).

- **Radial Distortion Function:** Typically approximated by a Taylor series:

$$L(r) = 1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots$$

- **Tangential Distortion Vector:** Approximated by a more complex polynomial involving both x and y coordinates.

The radial coefficients $(k_1, k_2, \dots)$ and tangential coefficients (p_1, p_2) form the set of lens distortion parameters, which **extends the set of intrinsic parameters**.

7.7.3 Lens Distortion in the Image Formation Pipeline

The non-linear distortion mapping takes place **after** the canonical perspective projection (division by z_C) but **before** the final affine transformation by the intrinsic matrix A.

The complete image formation process with lenses can be summarized in four steps:

1. **Extrinsic Transformation:** Transform 3D points from WRF to CRF using G.

$$\tilde{M}_C = G\tilde{M}_W$$

2. **Canonical Perspective Projection:** Scale by the third coordinate to get ideal, undistorted *metric* image coordinates.

$$\begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix} = \begin{bmatrix} x_C/z_C \\ y_C/z_C \end{bmatrix}$$

3. **Non-linear Lens Distortion:** Apply the distortion model to get the real, distorted *metric* image coordinates.

$$\begin{bmatrix} x \\ y \end{bmatrix} = L(r) \begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix} + \text{tangential\_component}$$

4. **Intrinsic Transformation:** Map the distorted metric coordinates to final pixel coordinates using the intrinsic matrix A.

$$\tilde{m} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

---

8 The Complete Camera Model

To understand camera calibration, we must first define the complete camera model, which describes the entire process of mapping a 3D point in the world to a 2D point in an image. This process involves three different coordinate systems and a series of transformations between them.

- **World Reference Frame (WRF):** A 3D coordinate system (x_W, y_W, z_W) fixed in the world, used to define the metric positions of objects. A point in this frame is denoted as $M_W = [x_W, y_W, z_W]^T$.
- **Camera Reference Frame (CRF):** A 3D coordinate system (x_C, y_C, z_C) attached to the camera, with its origin at the camera's optical center. A point in this frame is denoted as $M_C = [x_C, y_C, z_C]^T$.
- **2D Image Plane:** A 2D coordinate system (u, v) representing the pixel coordinates of the image sensor. A point in this frame is denoted as $m = [u, v]^T$.

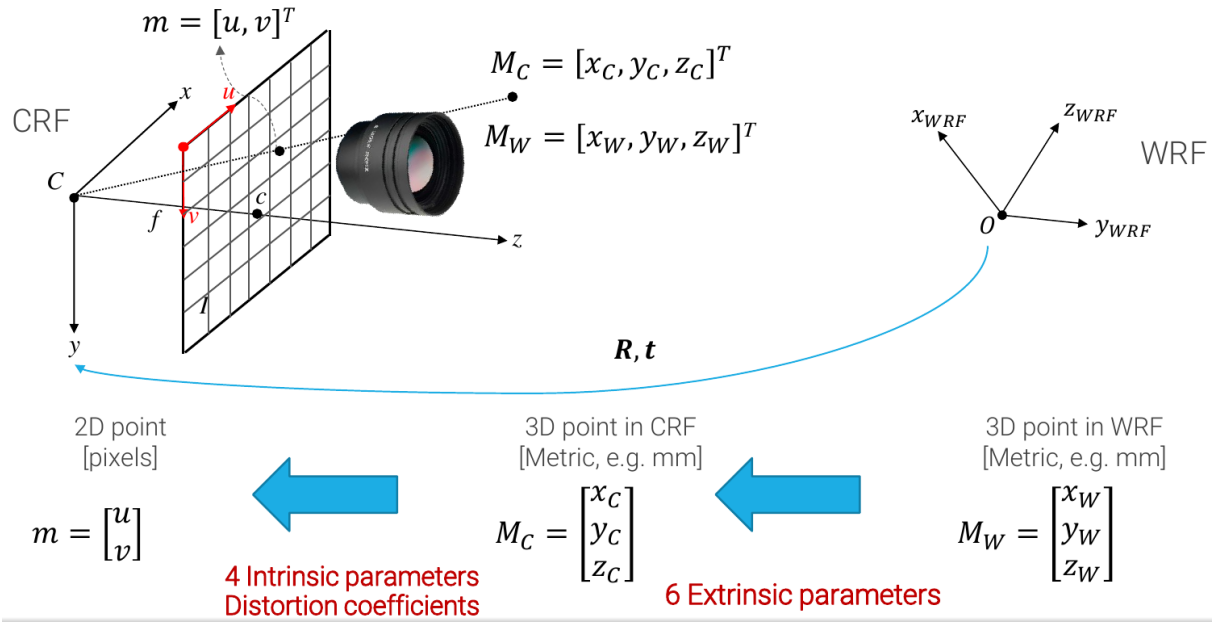


Figure 55: Visual representation of the complete forward imaging model. A 3D point in the World Reference Frame is first transformed into the Camera Reference Frame by the 6 extrinsic parameters (rotation R and translation t). Subsequently, the 3D point in the CRF is projected onto the 2D image plane to produce the final pixel coordinates (m), a process governed by the camera's intrinsic parameters and lens distortion coefficients.

The mapping from a 3D world point to a 2D image point involves two main sets of parameters:

1. **Extrinsic Parameters (R, t):** These **6 parameters** (3 for rotation R and 3 for translation t) define the rigid transformation (position and orientation) from the WRF to the CRF. They describe how the camera is placed in the world.

$$M_C = RM_W + t$$

2. **Intrinsic Parameters (A) and Distortion Coefficients (k, p):** These parameters describe the internal properties of the camera itself.
   - The **intrinsic matrix** A (containing at least 4 parameters: focal lengths f_u, f_v and principal point c_u, c_v) maps the 3D point from the CRF to the 2D image plane (in pixel coordinates).
   - The **distortion coefficients** model the non-linear effects of the camera lens.

8.1 Camera Calibration: The Problem

Camera calibration is the process of estimating all these unknown camera parameters (intrinsics and extrinsics) for a specific camera and setup.

> The formal problem is: **Given a lot of corresponding pairs** of known 3D world points $M_W^{(i)}$ and their measured 2D image projections $m^{(i)}$, **estimate the camera parameters**.

In a typical calibration setup:

- The 3D coordinates (M_W) of points on a known calibration pattern are **Known**.
- The 2D pixel coordinates (m) of these points in an image are **Known** (they are measured).
- The intrinsic and extrinsic parameters are **Unknown** and must be estimated.

8.1.1 Calibration Patterns and Approaches

There are two main categories of camera calibration approaches, distinguished by the type of calibration target used:

1. Using a **single image of a 3D calibration object**. This object must have a known 3D structure, typically featuring at least two planes with a known pattern.

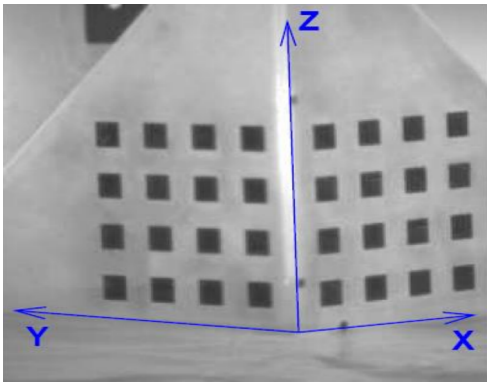
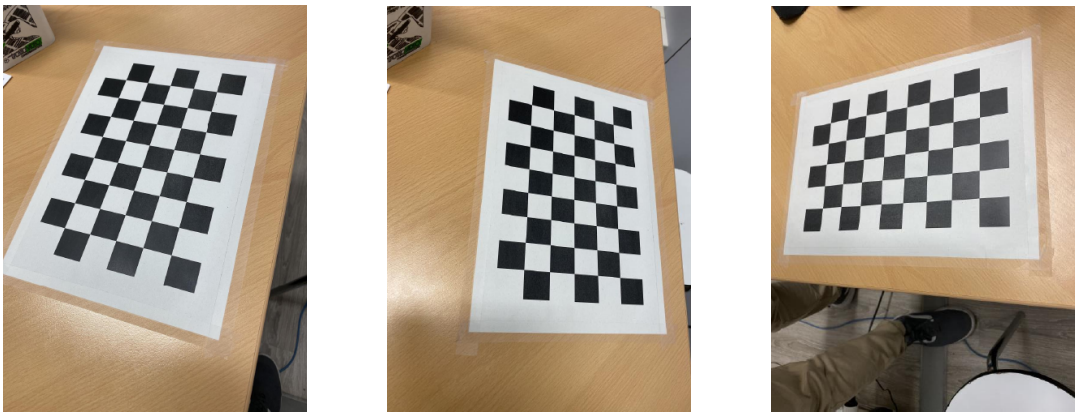


Figure 56: A single image with two non co-planar patterns.

2. Using **several (at least 3) different images of one given planar pattern**. This involves taking multiple pictures of a single flat pattern (like a chessboard) from different viewpoints.



(a) Image 1. (b) Image 2. (c) Image 3.

Figure 57

In practice, building a highly accurate 3D target is difficult and expensive. In contrast, creating an accurate planar target is much easier (e.g., by printing a high-resolution

chessboard). For this reason, the second approach, pioneered by Zhengyou Zhang, is the most widely used.

Implementing calibration software from scratch is a significant effort. Fortunately, major computer vision libraries like **OpenCV ('cv.calibrateCamera')** and the **Matlab Camera Calibration Toolbox** provide robust implementations, largely based on Zhang's method.

8.2 Zhang's Method: A Flexible Planar Approach

Zhang's method is a multi-step process that finds an initial analytical solution for the camera parameters and then refines it using non-linear optimization.

8.2.1 The Overall Workflow

1. **Acquire Images:** Take n images of a planar pattern (e.g., a chessboard) with c internal corners from various viewpoints.
2. **Compute Initial Homographies:** For each of the n images, compute an initial guess for the homography matrix H_i that maps the 3D points on the planar pattern to their 2D image coordinates.
3. **Refine Homographies:** Refine each H_i by minimizing the **reprojection error** (a geometric error metric).
4. **Estimate Intrinsics:** Using the refined homographies, compute an initial guess for the intrinsic matrix A.
5. **Estimate Extrinsics:** Given A and the homographies H_i, compute an initial guess for the rotation R_i and translation t_i for each image.
6. **Estimate Distortion:** Compute an initial guess for the lens distortion parameters k and p.
7. **Global Refinement:** Finally, refine all parameters simultaneously (A, R_i, t_i, k, p) by minimizing the total reprojection error over all images and all corners.

8.3 The Calibration Pattern and Coordinate Systems

A **chessboard pattern** is typically used. For this pattern, we know:

- The **number of internal corners**. To remove rotational ambiguity, the pattern is often designed to have an odd number of corners in one dimension and an even number in the other.
- The **size of the squares** (e.g., in mm).

The internal corners can be detected with high precision using standard algorithms like the Harris corner detector.

A **World Reference Frame (WRF)** can be conveniently attached to the pattern itself:

- The **origin** is placed at a specific corner.

- The x and y axes are aligned with the sides of the chessboard.
- The plane of the pattern is defined as $z_W = 0$.

With these rules and the known square size, we can easily compute the precise 3D coordinates $(x_W, y_W, 0)$ for every corner on the pattern.

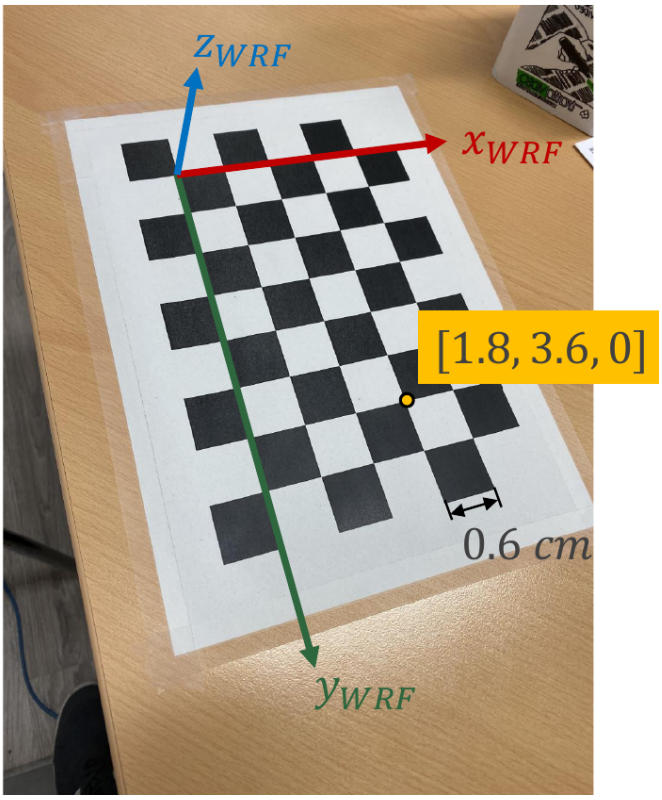


Figure 58: cazzoinculo.

It is important to note that since the pattern moves relative to the camera for each image, we are implicitly defining a different WRF for each view. Consequently, we must estimate a separate extrinsic matrix $[R_i, t_i]$ for each of the n calibration images.

8.3.1 Step 1: Estimating the Homography (H_i)

Since all 3D points on the calibration pattern lie on the plane $z_W = 0$, the general Perspective Projection Matrix (PPM) mapping simplifies. The 3D world point $M_W = [x_W, y_W, z_W, 1]^T$ becomes $\tilde{w} = [x_W, y_W, 1]^T$. The full 3x4 projection matrix P effectively reduces to a 3x3 matrix:

$$k\tilde{m} = P\tilde{M}_W = [p_1, p_2, p_3, p_4] \begin{bmatrix} x_W \\ y_W \\ 0 \\ 1 \end{bmatrix} = [p_1, p_2, p_4] \begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix} = H\tilde{w}$$

This 3x3 transformation matrix H is called a **homography**. It describes a general projective transformation between two planes. H can be thought of as a simplification of the full projection matrix P for the special case where the object is planar.

8.3.2 Solving for H using the DLT Algorithm

For each corner correspondence j in an image i, we have a known 3D point on the pattern $\tilde{w}_j$ and its measured 2D image point $\tilde{m}_{ij}$. The homography equation is:

$$\lambda \tilde{m}_{ij} = H_i \tilde{w}_j$$

where λ is a scale factor. The unknowns are the 9 elements of the matrix H_i.

In a projective space like $\mathbb{P}^2$, two points are equivalent if they lie on the same line from the origin. Therefore, the projected point $H_i\tilde{w}_j$ must be collinear with the measured point $\tilde{m}_{ij}$. This geometric constraint can be expressed algebraically: two vectors are collinear if their **cross product is the zero vector**.

$$\tilde{m}_{ij} \times (H_i \tilde{w}_j) = 0$$

This cross product yields three linear equations in the elements of H_i. However, only **two of these equations are linearly independent**.

For each corner correspondence, we get two linear equations. If we have c corners on our pattern, we can stack these equations to form a homogeneous, overdetermined linear system of the form:

$$L_i h_i = 0$$

where h_i is a 9x1 vector containing the flattened elements of the homography matrix H_i, and L_i is a $2c \times 9$ matrix constructed from the known coordinates.

To avoid the trivial solution $h_i = 0$, we add the constraint that $\|h_i\| = 1$. The problem becomes finding the unit vector h_i that minimizes $\|L_i h_i\|$. This is a classic linear algebra problem that can be solved by finding the **Singular Value Decomposition (SVD)** of the matrix L_i. The solution h_i^* is the last column of the matrix V_i in the SVD decomposition $L_i = U_i D_i V_i^T$, which corresponds to the smallest singular value. This provides an initial algebraic guess for the homography.

8.3.3 Refining the Homography and Estimating Parameters

The homography H_i estimated using the Direct Linear Transform (DLT) algorithm minimizes an **algebraic error** ($L_i h_i = 0$). While this provides a unique, closed-form solution that is cheap to compute, it has no direct physical meaning and may not be geometrically optimal. Therefore, it serves as a good **initial guess** that must be refined.

8.3.4 Non-linear Refinement of H_i

The initial guess for H_i is refined by minimizing a more meaningful error metric: the **reprojection error**. This is a **geometric error** measured in the image plane.

For each corner j in an image i, we compare:

- The measured (observed) pixel coordinates of the corner, m_{ij}.
- The reprojected (predicted) pixel coordinates, $H_i w_j$, obtained by applying the estimated homography to the known 3D world coordinates of the corner.

The goal is to find the optimal homography H_i^* that minimizes the sum of squared Euclidean distances between the measured and reprojected points over all c corners:

$$H_i^* = \arg\min_{H_i} \sum_{j=1}^{c} \|m_{ij} - H_i w_j\|^2$$

This is a non-linear minimization problem, which is solved using an iterative algorithm like **Levenberg-Marquardt**, starting from the initial guess provided by the DLT.

8.3.5 Step 2: Initial Guess for Intrinsic Parameters (A)

Once we have a refined homography $H_i = [h_{i1}\ h_{i2}\ h_{i3}]$ for each of the n images, we can use them to find an initial guess for the intrinsic matrix A.

From the definition of the homography, we know that $H_i = kA[r_{i1}\ r_{i2}\ t_i]$, where r_{i1} and r_{i2} are the first two columns of the rotation matrix R_i. This gives us the relations:

$$kr_{i1} = A^{-1}h_{i1} \quad \text{and} \quad kr_{i2} = A^{-1}h_{i2}$$

Since the columns of a rotation matrix are orthonormal, we can establish two constraints on the intrinsic matrix A for each homography:

1. **Orthogonality:** The dot product of the first two columns must be zero.

$$\langle r_{i1}, r_{i2} \rangle = 0 \implies h_{i1}^T A^{-T} A^{-1} h_{i2} = 0$$

2. **Unit Norm:** The columns must have the same length (norm 1).

$$\|r_{i1}\| = \|r_{i2}\| \implies h_{i1}^T A^{-T} A^{-1} h_{i1} = h_{i2}^T A^{-T} A^{-1} h_{i2}$$

Let $B = A^{-T}A^{-1}$. This matrix B is symmetric and has 6 unique unknown elements. Each homography provides 2 linear constraints on these 6 unknowns. Therefore, if we collect $n \geq 3$ **images** (and thus 3 homographies), we get at least 6 constraints, which is enough to solve for the 6 elements of B using SVD.

Once B is calculated, the intrinsic parameters in the matrix A can be recovered in closed form (this process is called Cholesky decomposition).

8.3.6 Step 3: Initial Guess for Extrinsic Parameters (R_i, t_i)

Once the intrinsic matrix A has been estimated, it is straightforward to compute the extrinsic parameters for each image i:

$$\begin{aligned} r_{i1} &= \frac{1}{k} A^{-1} h_{i1} \\ r_{i2} &= \frac{1}{k} A^{-1} h_{i2} \\ t_i &= \frac{1}{k} A^{-1} h_{i3} \end{aligned}$$

where $k = \|A^{-1}h_{i1}\|$ is a normalization constant. The third column of the rotation matrix can be found using the cross product: $r_{i3} = r_{i1} \times r_{i2}$.

The resulting matrix $R_i = [r_{i1}\ r_{i2}\ r_{i3}]$ may not be perfectly orthonormal due to noise. To fix this, we can find the closest orthonormal matrix to R_i by taking its SVD ($R_i = U_i D_i V_i^T$) and replacing the diagonal matrix D_i with the identity matrix I.

8.3.7 Step 4: Lens Distortion Coefficients

So far, we have assumed a perfect pinhole camera model. However, real lenses introduce non-linear distortions. The coordinates predicted by our model so far, m_{undist}, are the ideal, undistorted pixel coordinates. The coordinates we actually measure in the images, m, are the real, distorted coordinates.

Lens distortion takes place in the metric image space, *before* the affine transformation by the intrinsic matrix A converts metric coordinates to pixel coordinates. The relationship between undistorted metric coordinates (x_{undist}, y_{undist}) and distorted metric coordinates (x, y) can be modeled by a function $L(r)$, where $r^2 = x_{undist}^2 + y_{undist}^2$. Zhang's original method uses a radial distortion model:

$$\begin{bmatrix} x \\ y \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix}$$

We can transform this equation into pixel coordinates to get a linear, non-homogeneous system of equations:

$$Dk = d$$

where $k = [k_1, k_2]^T$ are the unknown distortion coefficients. With c corners in n images, we get $2nc$ equations for our 2 unknowns. This overdetermined system can be solved in a least-squares sense by computing the pseudo-inverse of D:

$$k^* = \arg\min_k \|Dk - d\|_2 = (D^T D)^{-1} D^T d$$

Note: OpenCV uses a more complex model with 3 coefficients for radial distortion (k_1, k_2, k_3) and 2 for tangential distortion (p_1, p_2).

8.3.8 Step 5: Final Refinement via Non-Linear Optimization

At this point, we have initial guesses for all camera parameters. The final step is to perform a global, non-linear optimization to refine all of them simultaneously. This is typically framed as a **Maximum Likelihood Estimate (MLE)**, assuming the measurement noise is independent and identically distributed (i.i.d.) Gaussian.

The goal is to find the set of parameters that minimizes the total reprojection error over all n images and all c corners:

$$A^*, k^*, R_i^*, t_i^* = \arg\min_{A,k,R_i,t_i} \sum_{i=1}^{n} \sum_{j=1}^{c} \|\tilde{m}_{ij} - \hat{m}(A, k, R_i, t_i, \tilde{w}_j)\|^2$$

where $\hat{m}(\cdot)$ is the full projection function that maps a 3D world point $\tilde{w}_j$ to a 2D distorted image point, given all the camera parameters. This large-scale optimization problem is solved using an iterative algorithm like **Levenberg-Marquardt**, using all previously computed values as initial guesses.

8.4 Applications of Camera Calibration

8.4.1 Compensating for Lens Distortion

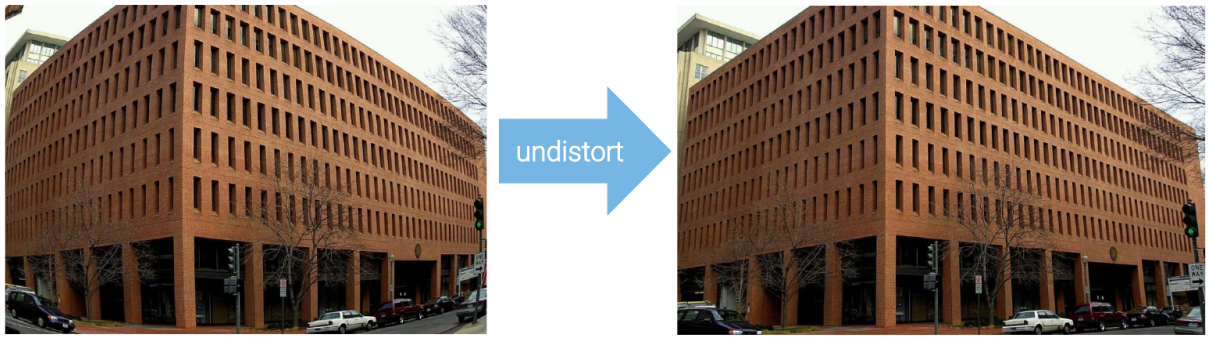


Figure 59

Once a camera is calibrated, we have a precise mathematical model of its lens distortion. Since this distortion is non-linear and makes geometric calculations cumbersome, it is common to **warp** the images to remove the distortion. This process, called **undistortion**, simulates a perfect pinhole camera whose image formation model is purely linear (described by the estimated PPM).

8.4.2 Image Warping

(a) Filtered (Gaussian blur).

(b) Original image.

(c) Warped (rotation & scale).

Figure 60: Image warping refers to transformations of the **spatial domain** of an image, where pixel coordinates are remapped. This is distinct from image **filtering**, which changes the pixel **RGB values** (e.g., Gaussian blur).

If we have a function $w(u, v) = (u', v')$ that maps coordinates from an input image I to an output image I', we can create the warped image by copying the pixel values: $I'(w(u, v)) = I(u, v)$. This transformation w can be a simple rotation, or a full homography.

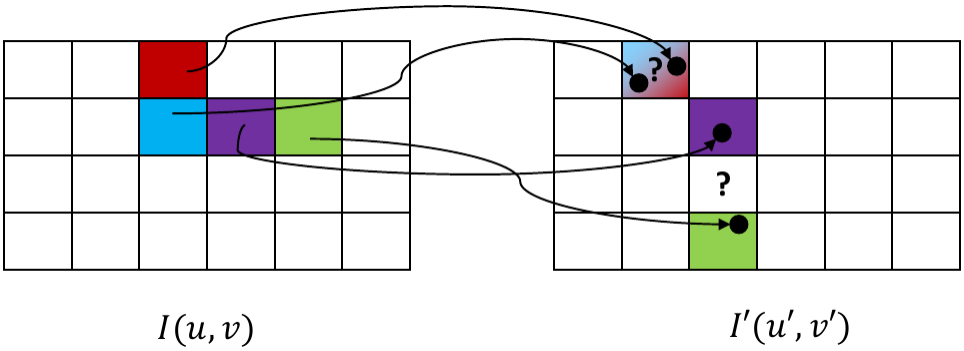


Figure 61: Possible problems with forward mapping.

To avoid problems like holes (pixels in the output that are not mapped to) and folds (multiple input pixels mapping to the same output pixel), image warping is typically implemented using **backward mapping**. We iterate over every integer pixel coordinate (u', v') in the output image, use the *inverse* warp function $w^{-1}(u', v')$ to find the corresponding (continuous) coordinate in the input image, and then use an interpolation method (like **bilinear interpolation**) to determine the correct pixel value.

8.4.3 Changing Point of View and Creating Virtual Cameras

Calibration and homographies enable powerful applications for manipulating viewpoints.

- **Planar Scene Rectification:** Any two images of a planar scene (assuming no lens distortion) are related by a homography. If we know the homography H, we can warp one image to appear as if it were taken from the viewpoint of the other. A common application is **Inverse Perspective Mapping (IPM)**, used in autonomous driving to get a "bird's-eye view" of the road plane.

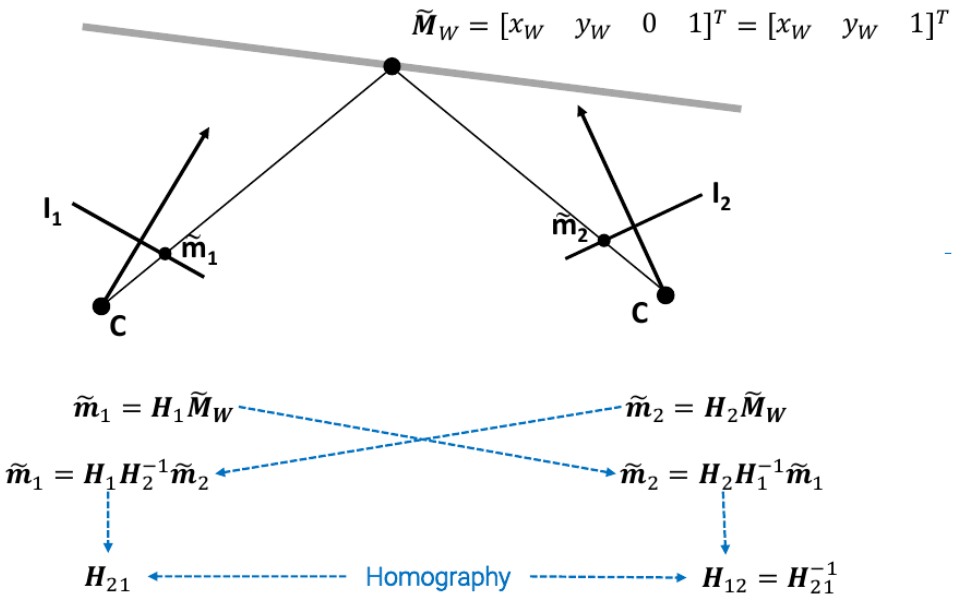


Figure 62: Any two images of a planar scene (without lens distortion) are related by a homography.

- **Creating Rotated Virtual Cameras:** Any two images taken by a camera that is purely *rotating* about its optical center are also related by a homography. This relationship is the foundation for creating panoramic images.

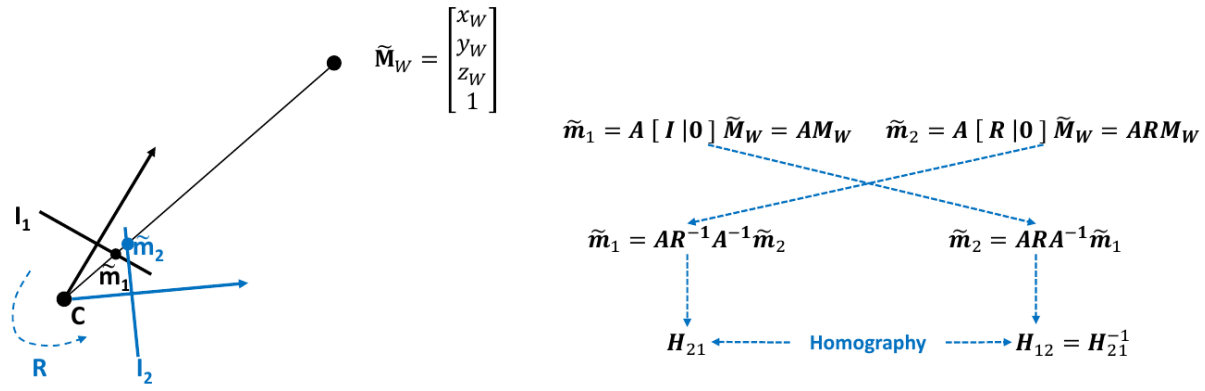
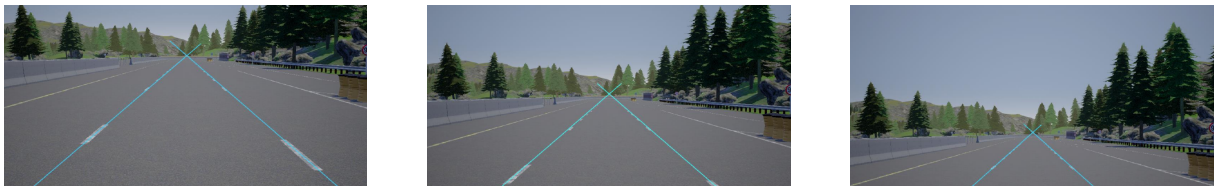


Figure 63: Any two images taken by a camera rotating around its optical center are related by a homography (if lens distortion have been removed).

- **Compensating for Pitch/Yaw:** In autonomous driving, a calibrated camera can be used to compensate for incorrect mounting (pitch or yaw). By detecting the vanishing point of the lane lines, it's possible to compute the camera's rotation relative to the road and then apply a warping homography to generate a corrected, "straight-ahead" view.



(a) Pitch = - 5° (b) Pitch = 0° (c) Pitch = +5°

Figure 64: *Pitch, yaw and roll* are the three dimensions of movement when an object moves through a medium. **Pitch** nose up or tail up. **Yaw**: nose moves from side to side. **Roll**: a circular (clockwise or anticlockwise) movement of the body as it moves forward

9 The Image Classification

9.1 Problem Definition

Image classification is a core task in computer vision. The goal is to assign a single label (a class) to an input image from a predefined set of categories.

- **Input:** An image.
- **Output:** A category from a given list (e.g., Dog, Cat, Bird, Frog, Person).

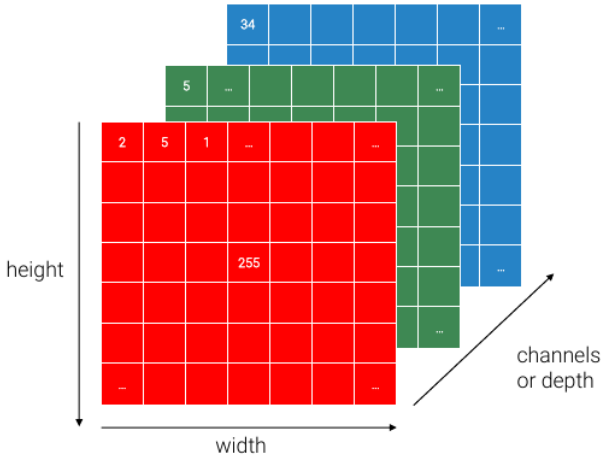
9.2 Semantic Gap and Inherent Challenges

This seemingly simple task is incredibly challenging for computers due to the vast **semantic gap** between the raw pixel data and the high-level concepts we want to identify. Some of the key challenges include:

- **Intraclass variations:** Objects within the same category can look vastly different (e.g., different breeds of cats, from kittens to adults).
- **Viewpoint variations:** An object's appearance changes dramatically depending on the angle from which it is viewed.
- **Illumination changes:** Lighting conditions can drastically alter the colors and shadows in an image.
- **Background clutter:** The object of interest can be difficult to distinguish from a complex or similarly-textured background.
- **Occlusions:** The object may be partially hidden from view.
- **Weirdness of the world...** Unpredictable scenarios, like a cat wearing glasses, defy simple rules.

9.2.1 Image Representation in a Computer

For a computer, an RGB image is not a holistic object but a **tensor** of numbers, typically with three dimensions:



- Height
- Width
- Channels (or depth), which is 3 for a standard RGB image (Red, Green, Blue).

Figure 65

Each pixel in each channel holds a value, usually an integer from 0 to 255.

The classification task, therefore, is to create a function f that maps this large tensor of numbers to a single integer index representing a category:

$$f(\text{ImageTensor}) = \text{CategoryIndex}$$

For example: $0 \rightarrow$ Dog, $1 \rightarrow$ Cat, $\mathbf{2} \rightarrow$ **Bird**.

9.2.2 Limits of "Classic" Computer Vision

How far can classic, rule-based computer vision take us? A common approach might be to first detect edges in an image (e.g., using a Canny edge detector) and then try to define a set of hand-crafted rules based on the resulting edge map to identify an object.

This approach is extremely **brittle**. The enormous intraclass variation (e.g., the vast differences between a hoopoe, a stork, Tweety bird, and a statue of a bird) makes it impossible to define a single, robust set of rules that can identify all "birds".

Traditional computer vision techniques, such as those based on handcrafted rules, require a highly **controlled environment** to be effective. They are usually feasible in **industrial vision applications** (e.g., inspecting parts on an assembly line), but they fail in the uncontrolled and varied conditions of the real world.

9.3 The Machine Learning (Data-Driven) Approach

To overcome the limitations of classic CV, we turn to a fundamentally different paradigm: **(Supervised) Machine Learning**. Instead of a programmer trying to define explicit rules, we provide the computer with a large dataset of examples and let it learn the rules on its own. The goal is to learn a function f that, given a new image, can correctly predict its label.

$$f(\text{NewImage}) = \text{CorrectLabel}$$

9.3.1 Traditional Programming vs. Machine Learning

- **Traditional Programming:** A programmer analyzes the requirements (e.g., "sort an array"), devises an algorithm (the program), and then the computer executes this program on new data to produce an output. The intelligence is in the human-written program.

- **Machine Learning:** A programmer provides the computer with example inputs and their corresponding desired outputs. The **training** (or learning/optimization) process then automatically produces a "program" (the trained model, often treated as a black box). This model can then be used for **testing** (or inference/prediction) on new, unseen inputs to generate outputs. The intelligence is learned from the data.

9.3.2 Consequences of the Data-Driven Approach

This paradigm shift has profound consequences for the field of computer vision.

- **Data and Annotations are Crucial:** In this new paradigm, the quality and quantity of the data and its annotations (labels) are paramount. Knowing the most used datasets and understanding the impact of evaluation metrics is at least as important as knowing the latest machine learning models. The performance of a model is fundamentally tied to the data it was trained on.

- **From Model-centric to Data-centric AI:** As highlighted by Andrew Ng, a leader in the field, a significant portion of machine learning work is data preparation. This has led to a recent push in MLOps (Machine Learning Operations) from a purely "model-centric" view (improving the model's architecture) to a "data-centric" view, which focuses on systematically improving the quality of the dataset. Andrew Ng states: *"If 80% of machine learning is data preparation, we should invest more in it and be more systematic about it."*

9.3.3 Training and Testing Datasets

When applying machine learning, we work with two primary sets of data:

- A **training set** $D^{\text{train}} = \{(x^{(i)}, y^{(i)}) | i = 1, ..., N\}$
- A **test set** $D^{\text{test}} = \{(x^{(i)}, y^{(i)}) | i = 1, ..., M\}$

Here, $x^{(i)} \in \mathbb{R}^f$ is the feature vector representing an item (e.g., a flattened image), and $y^{(i)}$ is the corresponding label.

A core assumption in supervised learning is that both the training and test sets contain samples that are **independent and identically distributed (i.i.d.)** from the **same unknown data distribution** $p_{\text{data}}(x, y)$. This assumption is crucial for ensuring that the model learned on the training set will generalize well to the test set.

9.4 Key Image Classification Datasets

- **MNIST (Modified NIST):** A classic "hello world" dataset in machine learning.

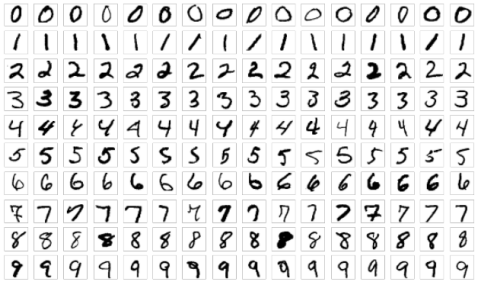


Figure 66: A sample of the MNIST dataset.

  - **Content:** 10 classes of handwritten digits (0-9).
  - **Size:** 50k training images, 10k test images.
  - **Format:** 28x28 grayscale images.
  - **Caveat:** It is a relatively simple dataset. Models that perform well on MNIST may not generalize to more complex, real-world image datasets.

- **CIFAR-10 and CIFAR-100:** Subsets of the 80 million Tiny Images dataset.

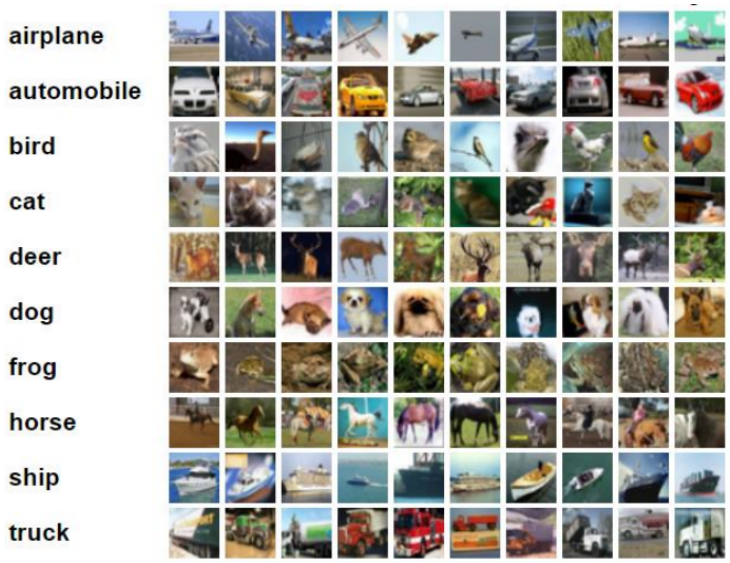


Figure 67: A sample of the CIFAR-10 dataset.

  - **Content:** CIFAR-10 has 10 common object classes. CIFAR-100 has 100 classes, organized into a hierarchy of 20 super-classes with 5 sub-classes each.
  - **Size:** 50k training images, 10k testing images for both.
  - **Format:** 32x32 color (RGB) images.

- **ImageNet / ImageNet 21k:** A massive, large-scale dataset that drove much of the deep learning revolution.

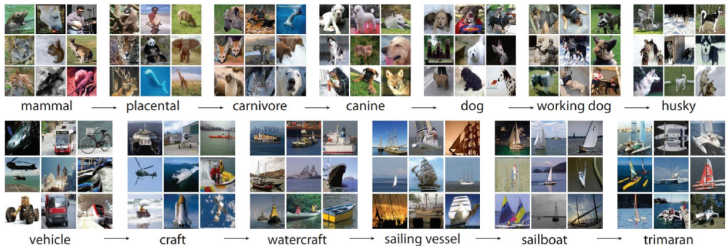


Figure 68: A sample of the ImageNet dataset.

  - **Content:** Modeled on the WordNet hierarchy, containing about 21,000 "synsets" (categories).
  - **Size:** Over 14 million full-resolution RGB images.

- **ILSVRC (ImageNet Large Scale Visual Recognition Challenge):** Often colloquially referred to as "ImageNet". This is a specific competition and dataset derived from the full ImageNet.

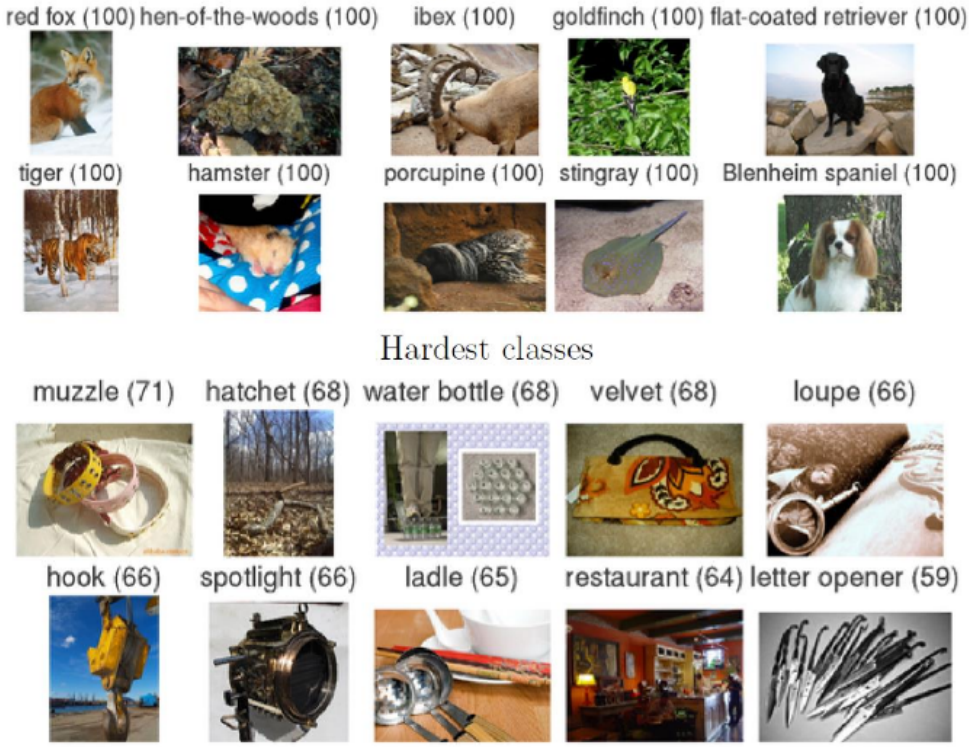


Figure 69: A sample of the ISLVRC dataset.

  - **Content:** 1000 object classes.
  - **Size:** 1.3M training images, 50k validation images, 100k test images.
  - **Format:** Variable resolution color images, often resized to 256x256 for training.

9.5 The Importance of Metrics: Top-5 Accuracy

There is inherent ambiguity in labeling images that contain multiple objects or complex scenes. For this reason, performance on ILSVRC was often reported using **top-5 accu-**

racy. An image is considered correctly classified if the true label is among the top 5 classes predicted by the model. This metric is more forgiving of ambiguity than standard top-1 accuracy. Recent work also explores multi-label accuracy to better handle complex scenes. The choice of metric is crucial, as it defines what we consider a "correct" prediction and can reveal different aspects of a model's performance.

9.6 The Parametric Approach

The data-driven approach can be formalized using a parametric model. We define a score function f that takes an input image x and a set of parameters θ and outputs a prediction.

$$f(x; \theta) = \text{prediction}$$

The learning process consists of finding the optimal set of parameters θ that makes the function's predictions match the training data.

9.6.1 The Linear Classifier

The simplest parametric model is the linear classifier. Here, the score function is a linear mapping from the input pixel data to a set of class scores.

$$f(x; W) = Wx = \text{scores}$$

- x: The input image, flattened into a single column vector. For a 32x32x3 CIFAR image, this is a 3072x1 vector.
- W: The weight matrix, whose parameters are to be learned. For 10 classes and a 3072-dimensional input, W is a 10x3072 matrix.
- 'scores': The output vector, containing a score for each of the 10 classes.

9.6.2 Why a Single Scalar Output is a Bad Idea

One might initially think of having the function output a single integer class index directly. This is a **bad idea** because class labels are categorical variables. Assigning them to integer IDs (e.g., bird=2, cat=3) imposes an artificial ordering and distance metric that is semantically meaningless. A predicted output of 2.5 would have no sensible interpretation ("half a bird and half a cat"), and the proximity of IDs 2 and 3 would not imply that birds and cats are visually similar.

9.6.3 From Scores to Class Labels

The correct approach is to have the classifier output a **score** for each class. These scores, also known as **logits**, represent how confident the model is that the input belongs to each class. The final predicted class is then simply the one with the highest score, found using the 'argmax' function.

9.6.4 Linear Classifier as Template Matching

Each row of the weight matrix W can be interpreted as a "template" for one of the classes. The matrix-vector product Wx computes the dot product (a form of correlation) between the input image vector x and each of these class templates. The class whose template has the "best match" (highest dot product) with the image gets the highest score.

9.6.5 Affine vs. Linear

In machine learning, the term "linear" is often used to describe an **affine** transformation, which includes a bias term b. The full form of the linear classifier is:

$$f(x;\theta) = Wx + b = \text{scores}$$

where the parameters are $\theta = (W, b)$.

9.7 Learning as Optimization

The process of "learning" is formally defined as solving an optimization problem.

- The set of all possible functions that a model can produce is its **hypothesis space** $\mathbb{H}$.
- Learning means finding the "best" function $h^* \in \mathbb{H}$ that minimizes a **loss function** L on the training data.
$$h^* = \arg\min_{h \in \mathbb{H}} L(h, D^{\text{train}})$$
- For parametric models, this is equivalent to finding the "best" set of parameters θ^* that minimizes the loss.
$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta, D^{\text{train}})$$

9.7.1 The Loss Function

Instead of directly optimizing for accuracy (which is non-differentiable and hard to optimize), we use a **proxy measure** called the **loss function** (also known as objective function, cost function, or error function). A good loss function is:

- **Easier to optimize** (e.g., differentiable).
- **Correlated with accuracy:** A low loss should correspond to high accuracy, and a high loss to low accuracy.

The overall loss on a dataset is typically the average of the per-sample losses.

$$L(\theta, D^{\text{train}}) = \frac{1}{N} \sum_i L(\theta, (x^{(i)}, y^{(i)}))$$

9.7.2 The 0-1 Loss (Number of Errors)

A natural choice for a loss function would be the number of misclassifications (L = #errors), also known as the **0-1 loss**. However, this function is very difficult to optimize. It is a step function and its gradient is zero almost everywhere, providing no information about which direction to move the parameters to improve the model. The error rate is insensitive to small (and sometimes even large) changes in the parameters.

9.7.3 From Scores to Probabilities: The Softmax Function

A more effective approach is to transform the raw output scores into a probability distribution over the classes. This is done using the **softmax function**:

$$p_{\text{model}}(Y = j|X = x^{(i)}; \theta) = \text{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp(s_k)}$$

The softmax function takes a vector of arbitrary real-valued scores and squashes it into a vector of values between 0 and 1 that sum to 1. It is a smooth, differentiable approximation of the ‘argmax‘ function (it should be called ”softargmax”). For numerical stability, it is typically implemented by subtracting the maximum score from all scores before applying the exponential.

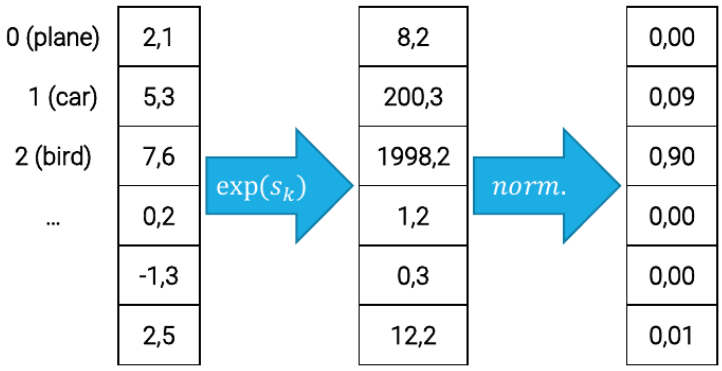


Figure 70: The scores of the classes get transformed into a probability distribution, where the sum of all values is 1.

9.7.4 The Cross-Entropy Loss

Once we have a probability distribution from the model, we can define the loss using the principle of **Maximum Likelihood Estimation (MLE)**. We want to find the parameters θ that maximize the probability of observing the true labels in our training data.

$$\begin{aligned}
\theta^* &= \arg\max_\theta \prod_{i=1}^{N} p_{\text{model}}(Y = y^{(i)}|X = x^{(i)}; \theta) \\
&= \arg\max_\theta \sum_{i=1}^{N} \log p_{\text{model}}(Y = y^{(i)}|X = x^{(i)}; \theta) \\
&= \arg\min_\theta \sum_{i=1}^{N} -\log p_{\text{model}}(Y = y^{(i)}|X = x^{(i)}; \theta)
\end{aligned}$$

This leads us to the **per-sample cross-entropy loss**:

$$L(\theta, (x^{(i)}, y^{(i)})) = -\log p_{\text{model}}(Y = y^{(i)}|X = x^{(i)}; \theta)$$

This loss is low when the model assigns a high probability to the correct class, and high when it assigns a low probability.

Putting it all together, the per-sample loss is:

$$L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_k \exp(s_k)}\right) = -s_{y_i} + \log\left(\sum_k \exp(s_k)\right)$$

The second term, $\log\sum\exp$, is known as the 'logsumexp' function and is a smooth approximation of the 'max' function. Minimizing this loss is therefore approximately equivalent to minimizing $-s_{y_i} + \max_k(s_k)$, which encourages the score of the correct class (s_{y_i}) to be higher than the scores of all other classes. It is a proxy to increase accuracy.

In **PyTorch**, this is implemented in 'torch.nn.CrossEntropyLoss', which conveniently combines a 'LogSoftmax' layer and the 'NLLLoss' (Negative Log Likelihood Loss). It expects raw, unnormalized scores (logits) as input.

9.8 Gradient Descent

Now that we have a differentiable loss function, how do we find the parameters θ that minimize it? The workhorse algorithm is **Gradient Descent**. The process is as follows:

0. (Randomly) initialize the parameters $\theta^{(0)}$.

for $e = 1, ..., E$ epochs:

(a) **Forward pass:** Compute the loss $L(\theta^{(e-1)}, D^{\text{train}})$ over the entire training set.

(b) **Backward pass:** Compute the gradient of the loss with respect to the parameters, $g = \frac{\partial L}{\partial \theta}|_{\theta^{(e-1)}}$.

(c) **Step:** Update the parameters by taking a step in the negative gradient direction: $\theta^{(e)} = \theta^{(e-1)} - lr \cdot g$.

This process requires several important **design decisions / hyperparameters**: the initialization method, the number of epochs, and the learning rate (lr).

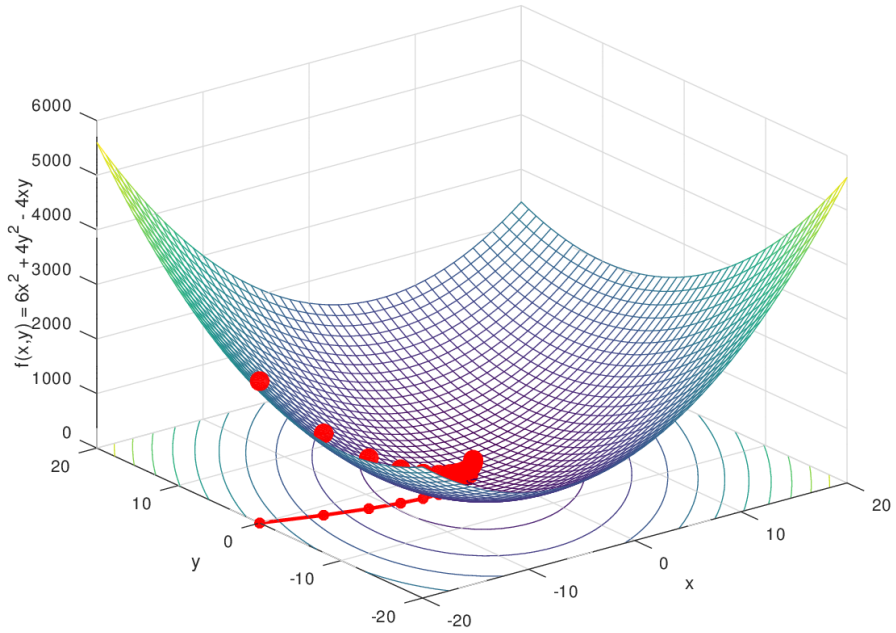


Figure 71: Every step is taken in the direction of the local minimum.

9.8.1 How to Compute Gradients?

Gradients can be computed in several ways:

- **Numerically:** Approximate the gradient using finite differences. Easy to implement, but slow and approximate.
- **Analytically:** Manually derive the gradient expressions using calculus (e.g., the chain rule). This is exact, but slow, tedious, and error-prone for complex models.
- **Automatically:** Use **automatic differentiation**, as implemented by the **backpropagation algorithm** in modern deep learning frameworks. This is the standard method as it is exact and efficient.

9.8.2 Limits of Gradient Descent (Batch GD)

Standard gradient descent (often called Batch Gradient Descent) requires computing the gradient over the *entire* training set to make a single parameter update. For large datasets like ImageNet (1.3M images), this is computationally infeasible, as it would require 1.3M forward and 1.3M backward passes for just one tiny step.

Furthermore, even the gradient computed on the entire training set is still just an approximation of the "true" gradient of the expected loss over the entire data distribution.

9.8.3 Stochastic Gradient Descent (SGD)

A much faster and more practical alternative is **Stochastic Gradient Descent (SGD)**.

- **Online SGD:** Instead of the full dataset, the gradient is computed for **one single training sample** at a time. The parameters are updated after every single sample.

- **SGD with Mini-batches:** A compromise between the two extremes. The gradient is computed over a small **mini-batch** of data (e.g., size $B = 32$ or 64). This is the standard approach in deep learning. The number of parameter updates per epoch is $U = N/B$.

SGD with mini-batches is efficient, allows for parallelization on GPUs, and the noise from the gradient estimation provides a form of regularization that helps the optimizer avoid poor local minima and can improve generalization.

9.8.4 Mini-batch Tradeoffs

The choice of mini-batch size involves several tradeoffs:

- **Larger batches** provide smoother, more accurate gradient estimates but have diminishing returns. They also better exploit parallel hardware like GPUs. Memory requirements scale linearly with batch size.
- **Smaller batches** introduce more noise, which can have a regularizing effect and lead to better generalization, but training takes longer.
- A common rule of thumb is to start with the largest batch size (as a power of 2) that fits in your GPU's memory and experiment from there.

9.8.5 What Does a Linear Model Learn on CIFAR10?

When a linear classifier is trained on CIFAR10, it achieves an accuracy of only about 38%. By visualizing the learned templates (the rows of W), we can see what it has learned. The templates are blurry and seem to have captured the **predominant background color** of each class (e.g., blue for "plane", green for "frog"). A single, fixed template per class is incapable of capturing the rich intraclass variations (like rotated cars). This highlights the fundamental limitation of the linear model: its reliance on a poor input space (raw pixels) and its limited capacity.

This brings us back to the main conclusion: **Don't classify directly in pixel space, learn more effective representations.** This sets the stage for Lecture 4 and the introduction of neural networks.

10 Image Representations

10.1 The Limits of "Shallow" Classifiers

Traditional or "shallow" classifiers, such as k-Nearest Neighbors (k-NN) and linear classifiers, operate directly on the raw pixel values of an image. This approach has significant limitations because of the **low effectiveness of input pixels as data features**.

An example from the CIFAR-10 dataset illustrates this problem vividly. When a k-NN classifier is used, a test image of a "plane" might be correctly classified, but the nearest neighbors retrieved by the algorithm are visually diverse and often belong to incorrect classes (like "frog" or "deer"). This happens because pixel-based distance metrics (like L1 or L2 distance) are not semantically meaningful. A small shift or rotation of an object can lead to a large pixel-wise distance, even though the object's identity is unchanged. The average images for each class, derived from the training set, are blurry and indistinct, further highlighting that raw pixels are a poor representation for complex visual concepts.

10.2 The Importance of Representation

The effectiveness of a classifier is critically dependent on the way the data is represented. A good representation can transform a complex, non-linear classification problem into a much simpler, linearly separable one.

Consider a toy example where data points from two classes are arranged in concentric circles.

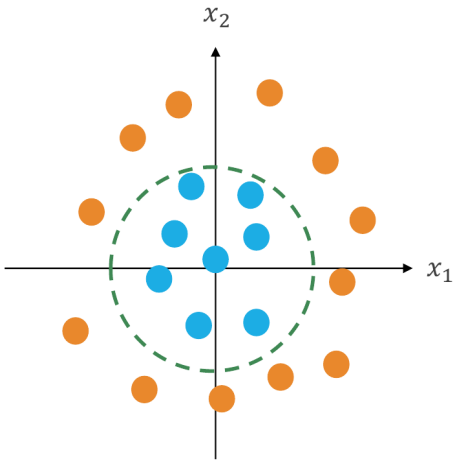


Figure 72

- **In the Input Space (Cartesian coordinates):** The two classes are not linearly separable. The decision boundary required to separate them is a circle, which is a non-linear function.

- **Switching to a Feature Space (Polar coordinates):** By changing the representation from Cartesian coordinates (x_1, x_2) to polar coordinates (ρ, θ), where $\rho = \sqrt{x_1^2 + x_2^2}$ and $\theta = \tan^{-1}(x_2/x_1)$, the problem is transformed. In this new feature space, the two classes become **linearly separable** by a simple vertical line (a threshold on the radius ρ).

This demonstrates a core principle: the goal is to find a feature transformation that makes the classification task easier.

10.3 Bag of Visual Words (BoVW)

The Bag of Visual Words (BoVW) model was the dominant paradigm in computer vision for image classification until the rise of deep learning around 2012. It addressed the limitations of using raw pixels by creating a more robust, mid-level image representation.

10.3.1 Core Concept

Inspired by similar "Bag of Words" techniques in Natural Language Processing, the BoVW model treats an image as a collection of "visual words". The classifier no longer works with the raw image pixels directly. Instead, it operates on a **histogram of codeword frequencies**.

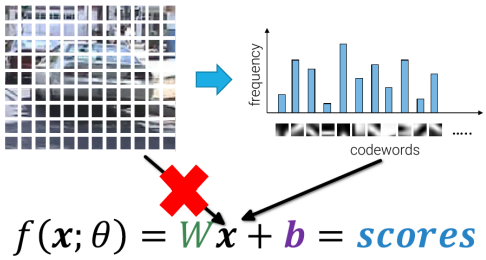


Figure 73: The classifier does not work anymore with the image pixels directly, but with a histogram of codeword frequencies, known as **Bag of (Visual) Words** (BoVW), inspired by similar representations that were popular at the time in Natural Language Processing.

The equation for a linear classifier, $f(x;\theta) = Wx + b$, is conceptually modified. The input x is no longer the flattened pixel vector but is replaced by the BoVW histogram, which serves as a new, more powerful feature vector.

10.3.2 The BoVW Pipeline (as used in ImageNet challenges pre-2012)

The pipeline to create these features was a multi-step process, often with precomputed components for challenges like ImageNet.

1. **Feature Extraction:** First, local descriptors are extracted from the image. A common choice was to compute dense **SIFT (Scale-Invariant Feature Transform)** descriptors at multiple scales across the image.

2. **Codebook (Visual Vocabulary) Creation:** A "visual vocabulary" or codebook is created from a large dataset of descriptors. This was typically done by:

- Randomly selecting a massive number of SIFT descriptors (e.g., 1 million) from the training images.
- Running a clustering algorithm, like **k-means**, on these descriptors.
- The resulting cluster centers (e.g., 1000 of them) form the **codewords** of the visual vocabulary. Each codeword is a prototypical local feature patch.

3. **Image Representation:** For a new image, each of its local descriptors is quantized by assigning it to the nearest codeword in the vocabulary. The image is then represented as a histogram of these codeword counts. This histogram is the BoVW feature vector.

4. **Classification:** This fixed-length histogram is then fed into a standard classifier. A **linear SVM (Support Vector Machine)** trained with SGD was a very popular choice due to its ability to scale to large datasets, a problem for many other classifiers at the time.

In this paradigm, the representation is **fixed**. The creativity of researchers was applied to designing better local features, better encoding schemes, and better classifiers to use with these fixed representations.

10.3.3 The ILSVRC 2011 Winning Entry: An Advanced BoVW System

The winning entry of the ILSVRC 2011 competition was a sophisticated system built upon these principles, showcasing the state-of-the-art before the deep learning revolution. Its pipeline included:

- **Low-level Feature Extraction:** Extracted SIFT (128-dim) and color (96-dim) features from 10k patches per image. These were then reduced to 64 dimensions using PCA.
- **Advanced Encoding (Fisher Vectors):** Instead of a simple BoVW histogram, it used a more powerful encoding method called Fisher Vectors (FV). This involved using a Gaussian Mixture Model (GMM) with N=1,024 Gaussians to create a very high-dimensional signature (520K dimensions).
- **Compression:** The massive FV signatures were compressed to make them manageable.
- **Classification:** A one-vs-all linear SVM was trained using SGD.
- **Fusion:** The final system used a late fusion of the SIFT-based system and the color-based system.

This highly engineered, complex pipeline represents the pinnacle of the "hand-crafted feature" era, which was about to be superseded by a new paradigm.

10.4 Representation Learning

The key shift from traditional methods to deep learning is the move from **fixed representations** to **learnable representations**.

- **Traditional Pipeline (Fixed Representation):** An image is passed through a hand-crafted feature extractor (like BoVW or FV). The resulting feature vector is then used to train a classifier. The representation itself is not learned; only the classifier's parameters are adjusted. The optimization loop (forward/backward pass) only involves the classifier.
- **Deep Learning Pipeline (Learnable Representation):** Deep learning introduces the concept of a **learnable transform**. The entire pipeline, from raw pixels to the final classification, is differentiable. The optimization loop adjusts not only the classifier's parameters but also the parameters of the feature transformation itself. The network learns to extract the most useful features for the task at hand.

10.5 Neural Networks as Learnable Representations

10.5.1 From Linear Classifier to Neural Network

A simple **linear classifier** is defined by the function $f(x;\theta) = Wx + b$, where x is the input vector, W is a weight matrix, and b is a bias vector.

A basic **Neural Network** extends this idea by composing two (or more) linear transformations. It introduces an intermediate, "hidden" representation h. The overall function becomes a composition of functions:

$$f(x;\theta) = W_2 h + b_2 = W_2\phi(W_1 x + b_1) + b_2$$

This formulation introduces two new key **hyper-parameters**:

1. The **dimension of the inner representation C**: This is the size of the hidden vector h, which controls the "width" of the hidden layer.
2. The **activation function** ϕ: A non-linear function applied element-wise.

The parameters to be learned are now $\theta = (W_2, b_2, W_1, b_1)$.

10.5.2 The Role of Activation Functions

Why do we need to insert non-linear activation functions? If we were to stack linear layers without any non-linearity, the entire network would collapse back into a single linear classifier.

$$\begin{aligned} f(x;\theta) &= W_2(W_1 x + b_1) + b_2 \\ &= (W_2 W_1)x + (W_2 b_1 + b_2) \\ &= W_{21}x + b_{21} \end{aligned}$$

This shows that a composition of linear functions is itself a linear function. The non-linear activation function ϕ is what gives the network its ability to learn complex, non-linear mappings and decision boundaries.

10.5.3 Common Activation Functions

- **(Logistic) Sigmoid:** A function that squashes its input to the range (0, 1).

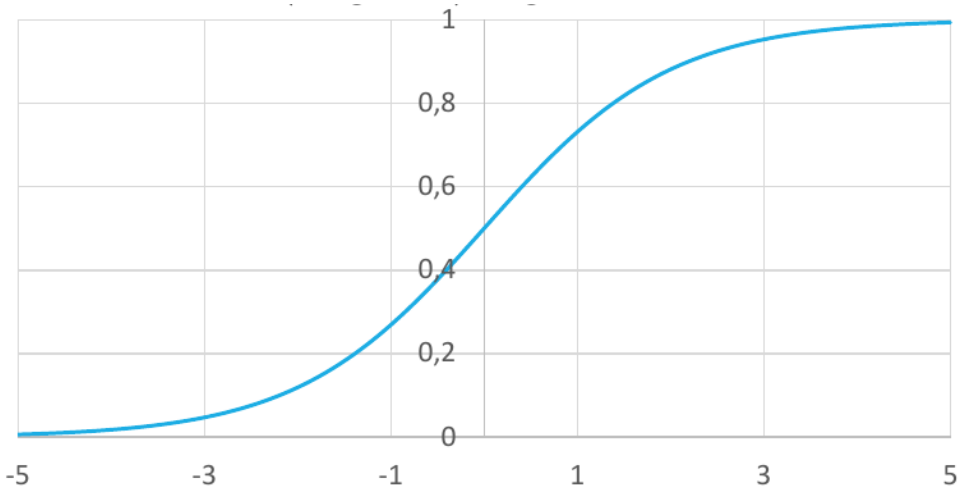


Figure 74: $\phi(a) = \sigma(a) = \frac{1}{1+\exp(-a)}$

- **Rectified Linear Unit (ReLU):** A simple and highly effective function that outputs the input if it is positive, and zero otherwise.

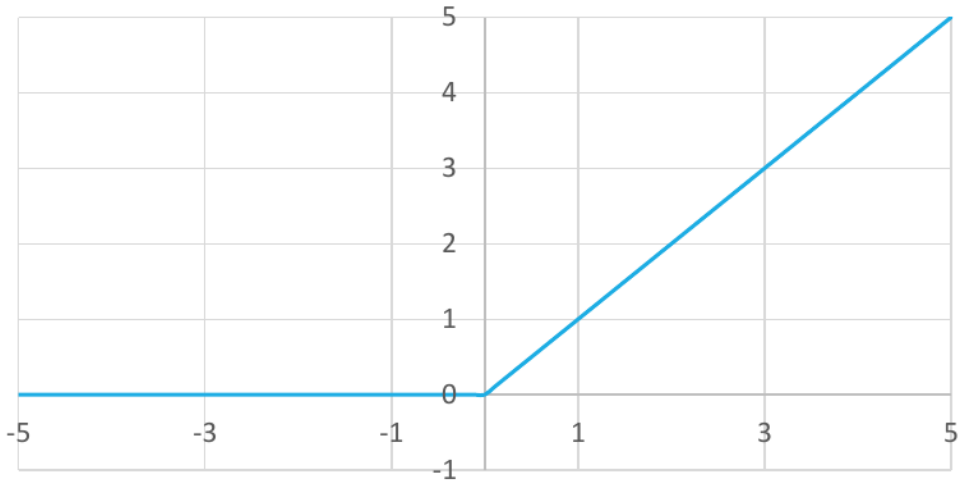


Figure 75: $\phi(a) = \text{ReLU}(a) = \max(0, a)$

- **Leaky ReLU:** To mitigate the "dead neuron" problem, Leaky ReLU introduces a small, non-zero constant gradient for negative inputs (e.g., 0.01). This ensures that the neuron can always recover.

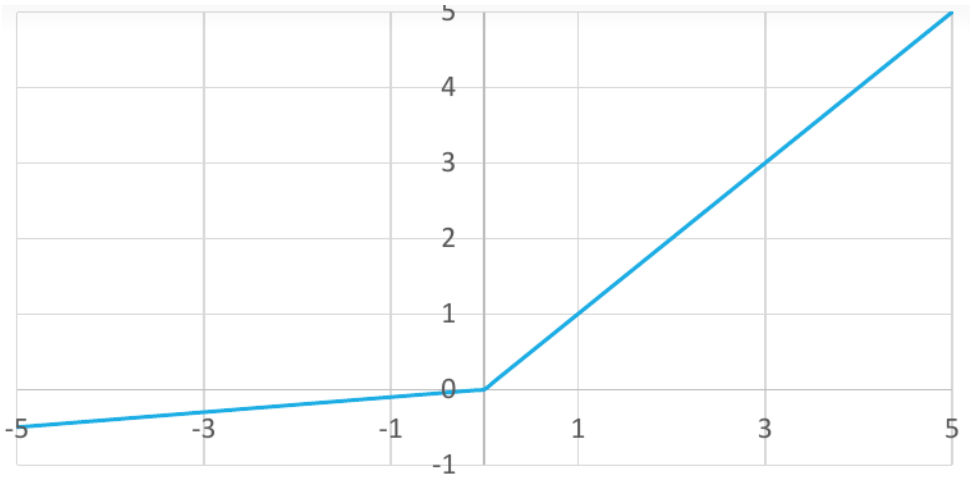


Figure 76: Leaky ReLU$(a) = \begin{cases} a & \text{if } a \geq 0 \\ 0.01a & \text{otherwise} \end{cases}$

10.5.4 Gradients and Training Dynamics

The choice of activation function has a profound impact on the training process due to the nature of their gradients.

- **Sigmoid Gradient:** The gradient of the sigmoid function, $\sigma(a)(1-\sigma(a))$, **saturates** for large positive or negative inputs, meaning the gradient becomes very close to zero. This "vanishing gradient" problem makes it very difficult to train deep networks, as the error signal struggles to propagate back through the layers.

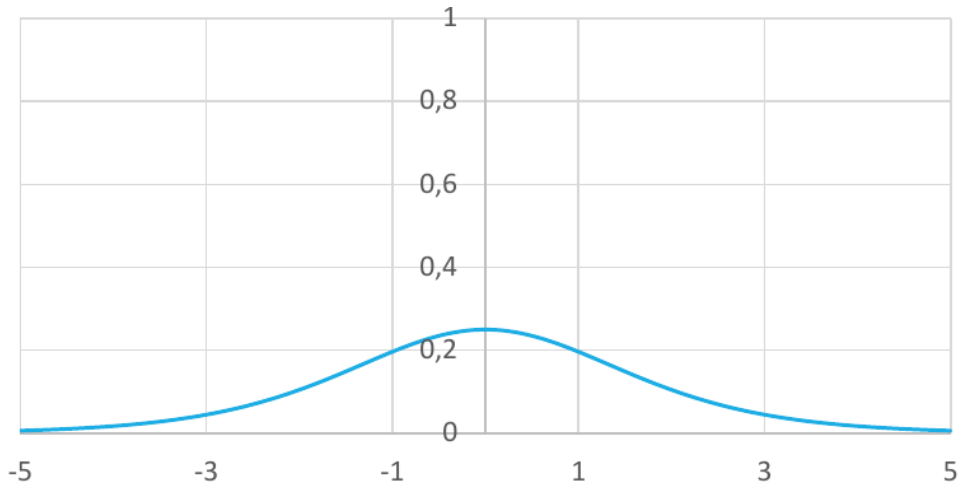


Figure 77: $\frac{d\,\sigma(a)}{da} = \sigma(a)\,(1-\sigma(a))$

- **ReLU Gradient:** The gradient of the ReLU function is extremely simple: 1 for positive inputs and 0 for negative inputs. The constant gradient of 1 for positive values prevents saturation and makes training much faster and more stable. However, ReLU can suffer from the **"dead neuron" problem**: if a neuron's input is consistently negative, it will always output zero, and the gradient flowing through it will also always be zero, effectively "killing" it from participating in learning.

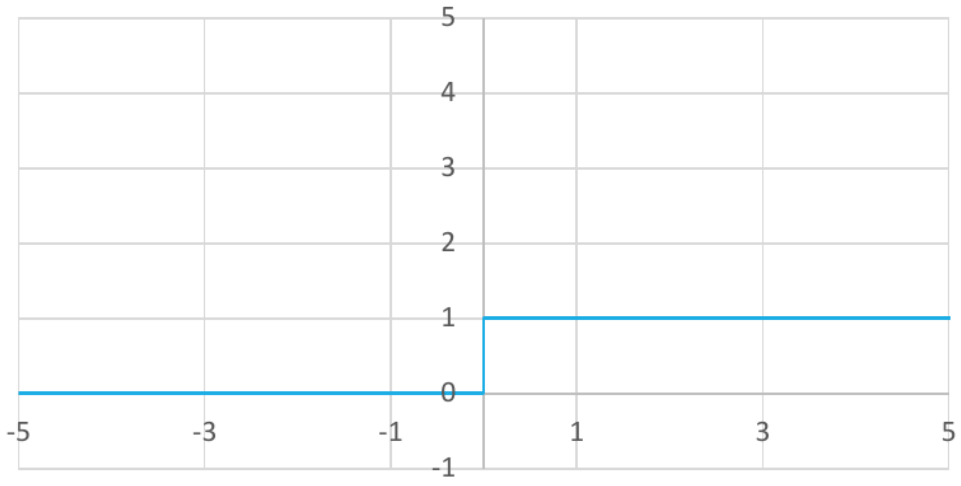


Figure 78: $\frac{d\,\mathrm{ReLU}(a)}{da} = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{otherwise} \end{cases}$

- **Leaky ReLU Gradient:** The gradient of the Leaky ReLU function is similar to ReLU but with a small non-zero slope for negative inputs. Specifically, the gradient is 1 for positive inputs and a small constant α (typically $\alpha = 0.01$) for negative inputs. This small slope helps mitigate the "dead neuron" problem of standard ReLU, allowing the neuron to continue learning even for negative inputs. While Leaky ReLU prevents neurons from being entirely "killed," it may still suffer from issues like slower convergence compared to ReLU, depending on the value of α.

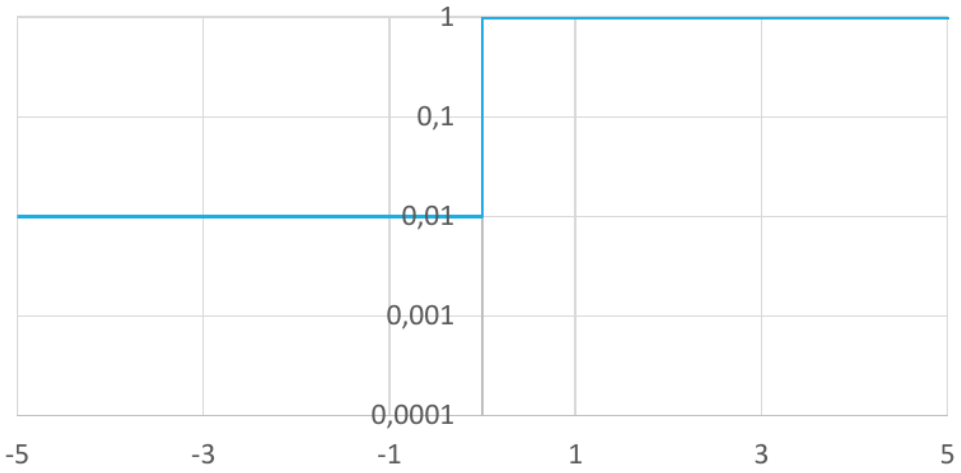


Figure 79: $\frac{d\,\mathrm{LeakyReLU}(a)}{da} = \begin{cases} 1 & \text{if } a \geq 0 \\ 0.01 & \text{otherwise} \end{cases}$

10.5.5 Is ReLU Enough to Create Non-Linear Separability?

A single layer of a neural network performs an affine mapping ($h = Wx + b$) followed by a non-linear activation ($\max(0, h)$).

An affine mapping alone **does not change linear separability**. It can only stretch, rotate, and shift the data points, but if they weren't linearly separable in the input space, they won't be in the hidden space h before the activation.

The ReLU activation, $\max(0, \cdot)$, is what performs the crucial non-linear transformation. It "folds" the space by collapsing all points in certain quadrants onto the axes or the origin. This re-arranges the data points in such a way that they can become linearly separable in the post-activation hidden space.

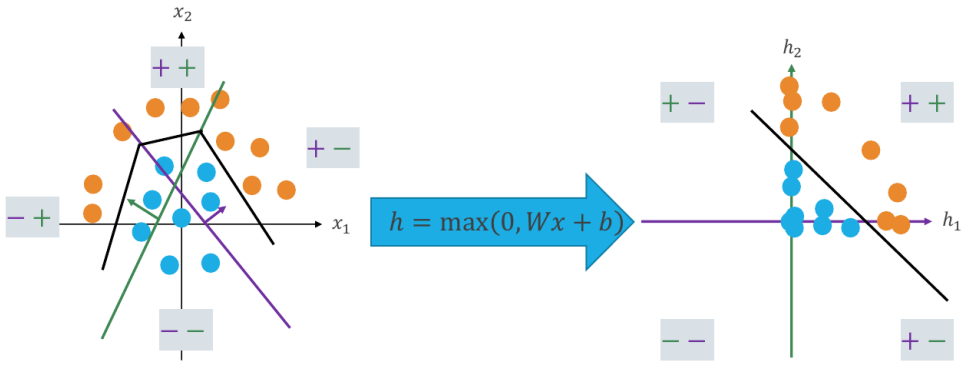


Figure 80: On the left, the *input space*, where points aren't separable by a linear classifier. On the right, the *hidden space*, where points become linearly separable.

Therefore, a linear classifier applied in this new representation space effectively creates a complex, **non-linear classifier in the original input space**.

10.5.6 Terminology of Neural Networks

- **Input/Activations/Parameters:** x is the input tensor, h and s are intermediate **activations**, and W_i, b_i are the learnable **parameters**.
- **Fully Connected (FC) Layer:** A layer where every element of the input influences every element of the output. Also known as a linear layer.
- **Multi-Layer Perceptron (MLP):** A neural network with 2 or more layers.
- **Depth and Width:** The **depth** of a network is the number of layers (a network with $L > 2$ is considered "deep"). The **width** of a layer is the number of activations it computes (i.e., the length of the vector h_i).

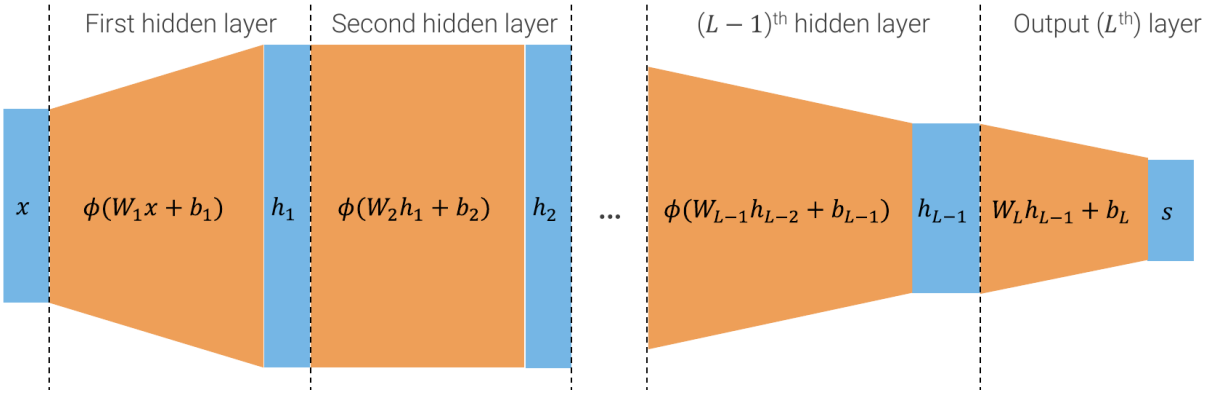


Figure 81

10.6 The Limits of Fully Connected Layers for Images

While MLPs are universal function approximators, they are poorly suited for image data. Let's assume we want the first layer of an MLP to learn to detect local features like vertical edges.

- **Massive Parameter Count:** For an input image of size $H \times W$, an FC layer that detects vertical edges would require approximately H^2W^2 parameters and $2H^2W^2$ FLOPs. For a small 224x224 image, this is over **2.5 billion parameters** and **5 Giga FLOPs**, which is computationally infeasible.
- **Lack of Spatial Structure:** The FC layer's weight matrix W_1 is a massive, unstructured matrix. To detect a vertical edge, it would have to learn the same sparse filter pattern (e.g., '[-1, 1, 0, 0, ...]') independently for every single possible position in the image. This is extremely inefficient.

10.7 Convolutions as a Solution

In traditional image processing, we use **convolution** or **correlation** with hand-crafted filters (kernels) to find local features. Neural networks adopt this operation but with learnable filters. Convolutions are built on powerful **inductive biases** about the structure of images:

1. **Locality:** Informative patterns in images are local. A convolution has a **local receptive field**, meaning each output unit is connected only to a small, local patch of input units. This drastically reduces the number of connections compared to an FC layer.

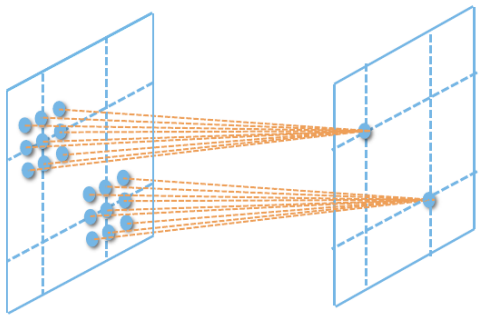


Figure 82: Each point on the output image (on the right) is formed only using a small patch of the input image (on the left).

2. **Stationarity:** The same local patterns (like an edge or a corner) can appear anywhere in an image. Convolutions implement this through **parameter sharing**. The same kernel (set of weights) is applied at every spatial location. This means the network doesn't have to re-learn to detect an edge at every single position.

These two properties make convolutional layers vastly more efficient than FC layers for image processing. A vertical edge detector implemented as a convolution requires only **2 parameters** and performs orders of magnitude fewer FLOPs (150 K flops for a 224x224 image).

10.7.1 Convolution vs. Correlation

Mathematically, a true convolution involves flipping the kernel. The operation used in neural networks does not flip the kernel and is technically called **(cross-)correlation**.

- Convolution: $[I * K](i,j) = \sum_l \sum_m I(l,m)K(i-l, j-m)$
- Correlation: $[K \star I](i,j) = \sum_l \sum_m K(l,m)I(i+l, j+m)$

At every location, correlation can be seen as the **dot product** between the kernel and the underlying image patch.

10.7.2 Properties of Convolutional Layers

A convolutional layer can be interpreted as a special type of matrix multiplication, where the resulting matrix is a **linear operator** that is:

1. **Parameter-sharing:** The same parameters are repeated across the rows of the matrix.
2. **Sparse:** Each output is connected to only a few inputs.
3. **Equivariant to Translation:** This is a critical property. If you translate the input image and then apply the convolution, the result is the same as applying the convolution first and then translating the output. $T(\text{conv}(x)) = \text{conv}(T(x))$. This equivariance, enabled by parameter sharing, is what gives CNNs their data efficiency. The network does not need to see an object at every possible location to learn how to detect it.

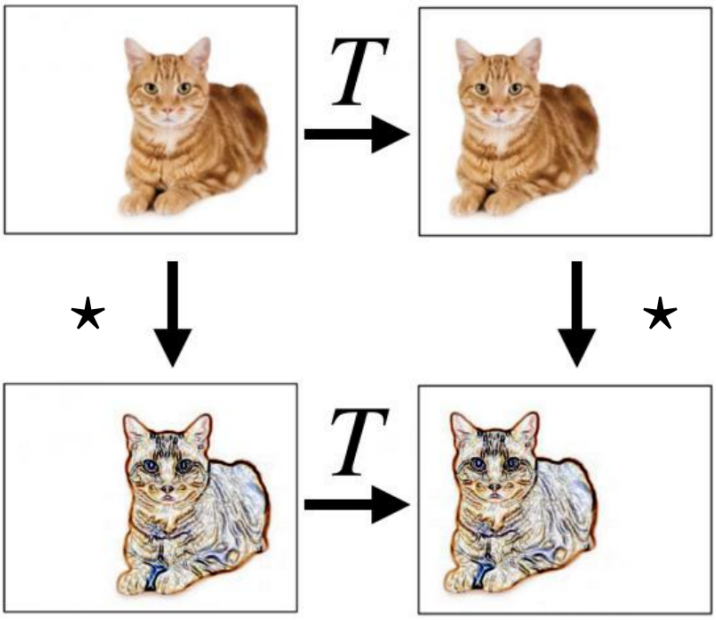
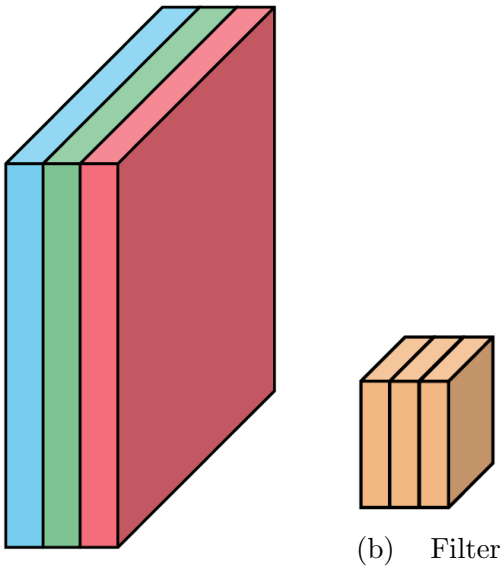


Figure 83: The same result is obtained both by performing first the *convolution* ($\star$) and then the *translation* (T), and in the inverse order.

Note that **correlation is not equivariant** with respect to rotation or scale.

10.7.3 Convolutional Layers in Practice

- **Multiple Channels:** Images have multiple input channels (e.g., 3 for RGB). A convolution kernel must have the same depth as its input. So, for an RGB image, a "5x5" kernel is actually a $3 \times 5 \times 5$ tensor. The convolution is still a 2D operation that slides over the spatial dimensions, computing a dot product over all channels simultaneously. A bias term is also added.



(a) Input image, e.g. $3 \times 32 \times 32$.

(b) Filter or kernel, $3 \times 5 \times 5$

Figure 84: Filters and input depth always match, and the third dimension of a filter is usually implicit, i.e. we refer to this convolution as a "5 by 5 convolution", but it has $3 \times 5 \times 5 = 75$ parameters (76 with the bias), not 25.

The formula for the convolution becomes:

$$[K * I](j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} K_n(m,l)\, I_n(j-m, i-l) + b$$

This is still a 2D convolution, but over vector-valued functions, not a 3D convolution (notice we do not slide over channels). Also, as usual, we compute an affine function, so we also have a bias term.

- **Output Activations:** Sliding a single filter over the input produces a single-channel output activation map, also called a **feature map**.

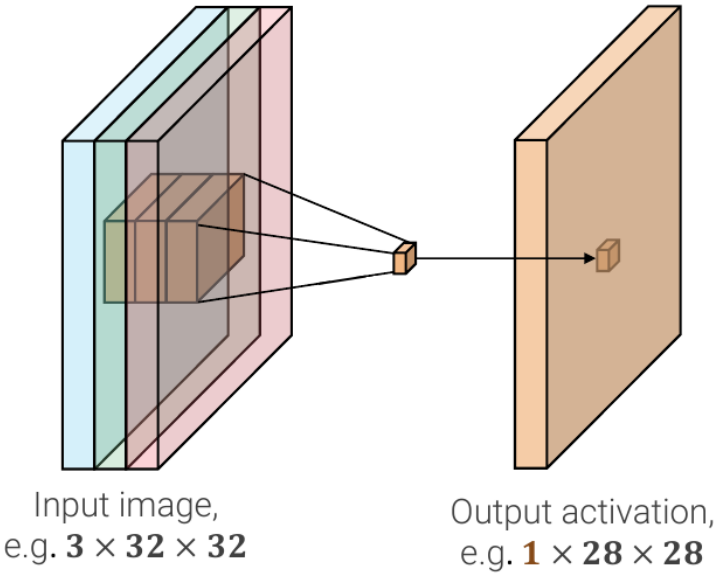


Figure 85: When the filter is at position (j, i) on the input image, we compute one number, i.e. the dot product between the kernel and the aptch of the input image plus the bias.

- **Multiple Output Channels:** A convolutional **layer** consists of multiple filters (e.g., 4 filters). Each filter is applied independently to the input, producing its own feature map. The resulting feature maps are stacked together to form the multi-channel output activation of the layer.

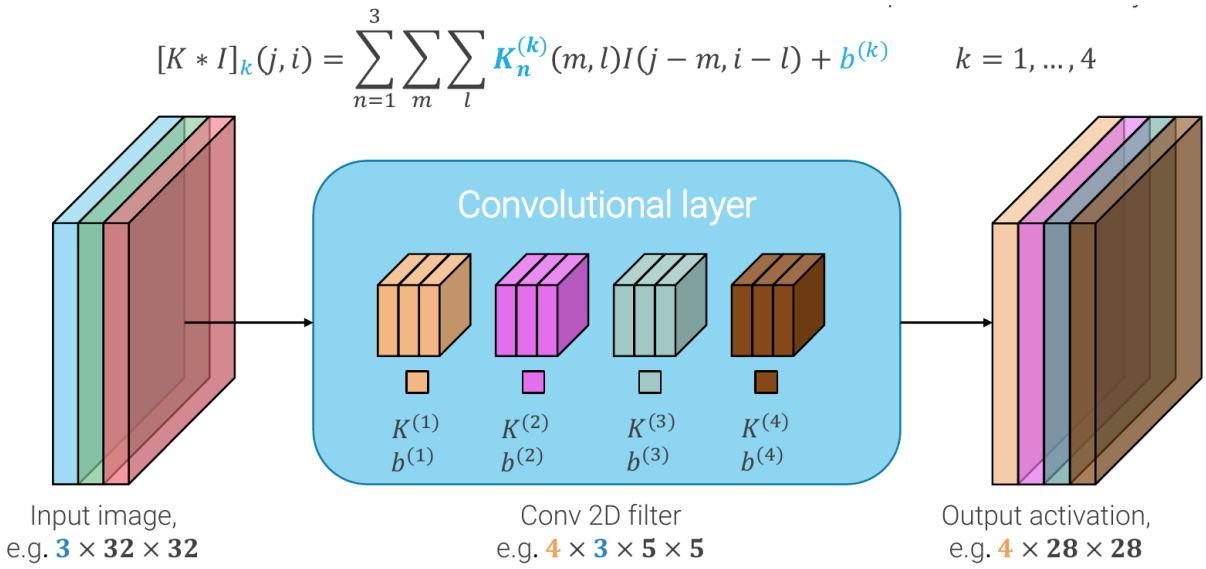


Figure 86

- **Stacking Layers:** Since convolutional layers are a constrained form of linear layers, they must be interleaved with **non-linear activation functions** to be meaningfully composed into a deep network.

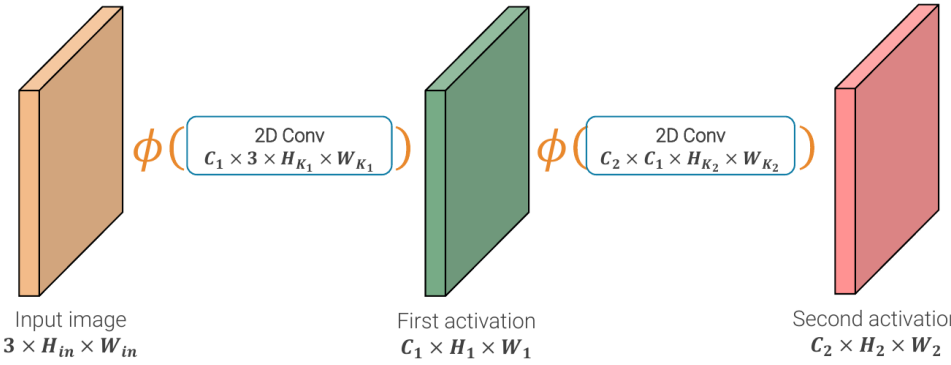


Figure 87

10.7.4 Spatial Dimensions and Receptive Fields

- **Shrinking Feature Maps:** Without padding, a convolutional layer shrinks the spatial dimensions of its input. The output size is $H_{out} = H_{in} - H_K + 1$.
- **Zero Padding:** To maintain the spatial resolution, we add a border of zeros (**zero padding**) around the input image. Using a padding of $P = (H_K - 1)/2$ for a kernel of size H_K results in an output of the same size as the input (often called "same" padding).

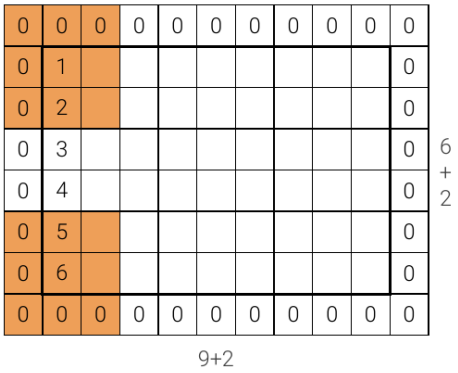


Figure 88: Adding a padding of one pixel on each side of the image lets the output image be of the same size as the input image, when using a 3×3 kernel.

- **Receptive Field:** The receptive field of a hidden unit is the region of the input image that affects its value. In a stack of convolutional layers, the receptive field grows linearly with depth.

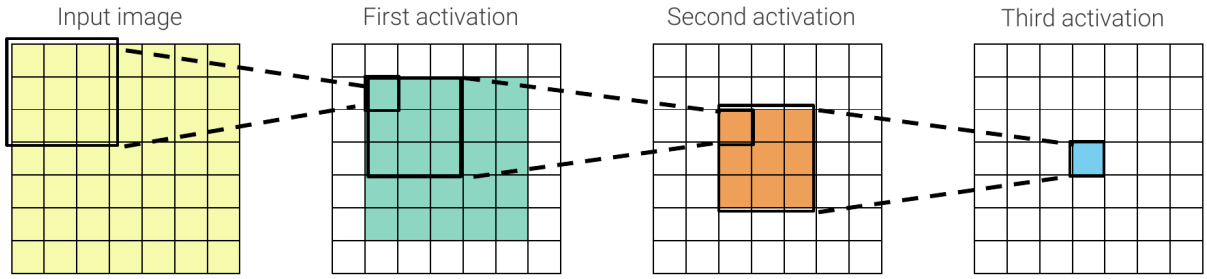


Figure 89: For instance, if we apply a $H_K \times W_K$ kernel size at each layer, the receptive field of an element in the L-th activation has size $r_L = [1 + L(H_K - 1)] \times [1 + L(W_K - 1)]$.

- **Strided Convolutions:** To achieve larger receptive fields more quickly and to down-sample the activations, we can use a **stride** greater than 1. A convolution with stride S skips $S-1$ pixels at each step. This is a common way to reduce the spatial dimensions inside the network.

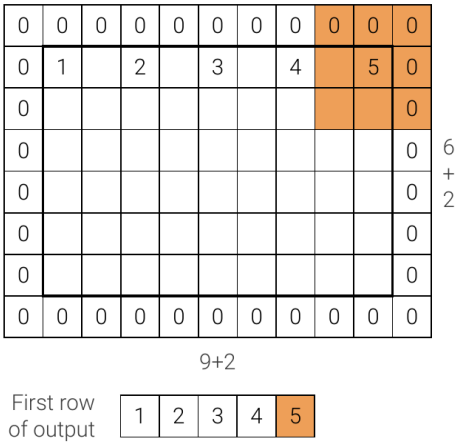


Figure 90: In general, $H_{out} = \lfloor \frac{H_{in} - H_K + 2P}{S} \rfloor + 1$, with P being the padding and S the stride. And the same goes for the width.

The size of the receptive field grows **exponentially** with the number of strided layers, making them very powerful for capturing large-scale context efficiently.

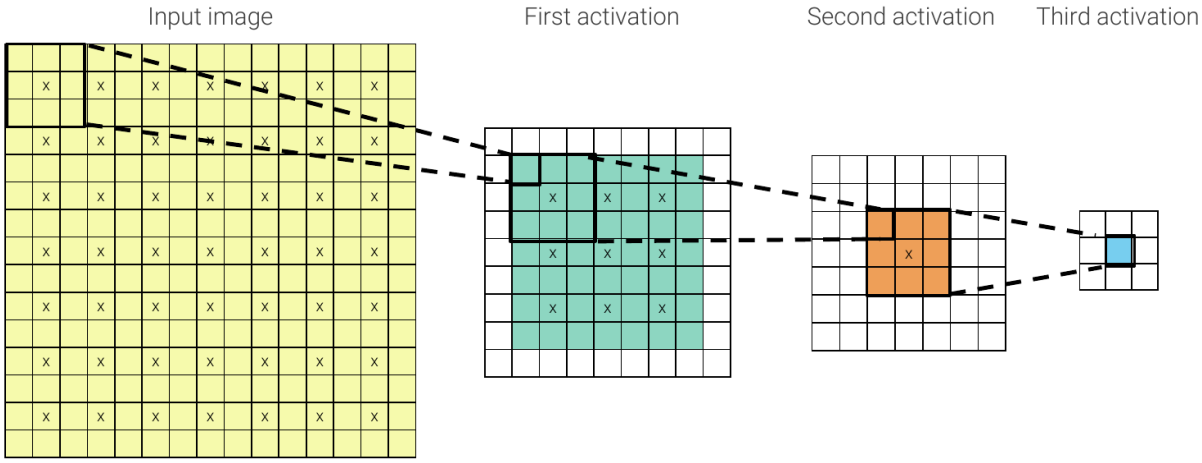


Figure 91: Receptive field with $S = 2$.

10.7.5 Pooling Layers

Pooling layers are another common component of CNNs used for downsampling.

- **Function:** A pooling layer aggregates values in a local neighborhood into a single output value using a pre-specified, non-learned kernel (e.g., 'max', 'average').
- **Key Difference from Convolution:** Each input channel is aggregated **independently**. The operation is purely spatial, and the number of output channels is the same as the input channels ($C_{out} = C_{in}$).

- **Max Pooling:** A common choice is a 2x2 max pooling operation with a stride of 2. It takes the maximum value within each 2x2 patch.

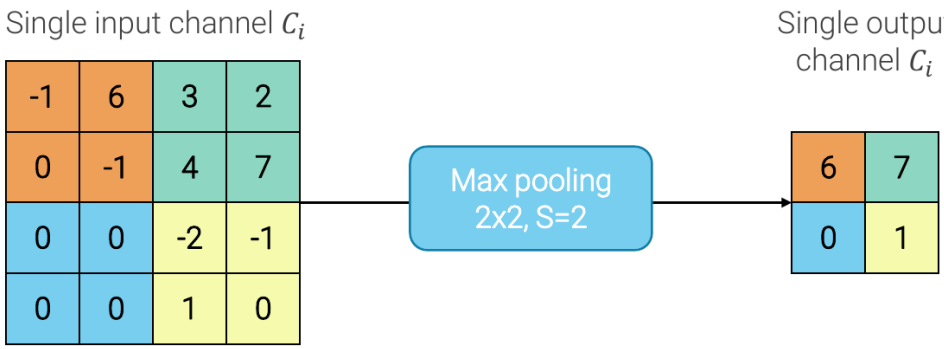


Figure 92

- **Properties:** It has no learnable parameters. A key benefit is that it provides a small amount of **invariance** to small spatial shifts of features. However, Hinton has famously noted that pooling is a "big mistake" that destroys valuable spatial information, even though it works well in practice. Downsampling can be achieved just as well with strided convolutions.

10.7.6 Batch Normalization for Convolutional Layers

The concept of Batch Normalization can be extended to convolutional layers.

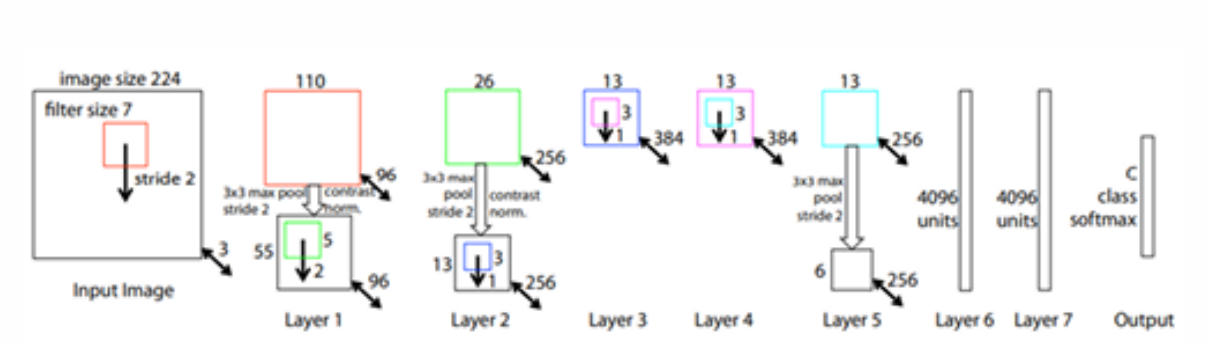
- **Spatial Batch Norm (BatchNorm2D):** Instead of normalizing along just the mini-batch dimension, we normalize along the mini-batch **and spatial dimensions** (height and width).

- **Rationale:** This respects the inductive bias of convolutions. Since the same kernel is applied to all elements of a feature map, it is logical to normalize all these elements in the same way. The idea behind Batch Normalization is to insert a layer to normalize the output of a layer during training, so that **each dimension has zero mean and unit variance in a batch**. The mean and variance are computed over all batch samples and all spatial locations for a given channel. The learnable scale (γ) and shift (β) parameters are per-channel.

11 Successful Architectures

11.1 ZFNet / Clarifai (Zeiler & Fergus, 2013)

ZFNet, which won ILSVRC 2013, can be seen as a thoughtful refinement of AlexNet. The authors aimed to move away from a "trial and error" approach to network design by introducing powerful visualization techniques (using Deconvolutional Networks, or Deconvnets) to understand what the network was learning in its hidden layers.



11.1.1 Key Findings and Improvements

Through visualizations and ablation studies, they identified several issues in the first layer of AlexNet:

- The aggressive 11x11 kernel with a stride of 4 resulted in **dead filters** (filters that learned no meaningful features).
- This aggressive downsampling also led to missing high-frequency information in the learned filters and created **aliasing artifacts** in the activations of the second layer.

To counteract these problems, they proposed key modifications to the initial "stem" layer of the network:

- They replaced the 11x11 conv with stride 4 with a smaller **7x7 convolution with stride 2**.
- They also used a stride of 2 in the subsequent 5x5 convolutional layer.

This more gentle initial downsampling allowed the network to learn more diverse and meaningful features in the early layers, leading to a significant improvement in performance over AlexNet.

11.2 VGG: Very Deep Convolutional Networks (Simonyan & Zisserman, 2014)

VGG, the runner-up in ILSVRC 2014, explored the effectiveness of increasing network depth using an extremely simple and regular design.

11.2.1 Design Philosophy

The core idea behind VGG was to use a homogeneous architecture built from a minimal set of components. This allowed the authors to isolate the effect of depth on performance. The design rules are very strict:

- **Only 3x3 convolutions** are used throughout the network (with stride 1, padding 1).
- **Only 2x2 max-pooling** is used for downsampling (with stride 2, padding 0).
- The number of channels **doubles after each max-pooling layer**.

This very deep, regular structure made the network easy to understand and adapt. The most famous variants are **VGG-16** and **VGG-19**, referring to the number of weight layers.

11.2.2 The Concept of Stages

VGG introduces the idea of designing a network as a repetition of **stages**. A stage is a sequence of layers that processes activations at the **same spatial resolution**. In VGG, a stage consists of a block of convolutional layers followed by a pooling layer that reduces the resolution for the next stage.

- **Stage Composition:** VGG stages are composed of two, three, or even four consecutive 3x3 convolutional layers.
- **Receptive Field vs. Parameters:** A stack of two 3x3 convolutions has the same effective receptive field as a single 5x5 convolution, but with fewer parameters ($18C^2$ vs $25C^2$) and an extra non-linearity (ReLU), which increases the network's expressive power.
- **The "No Free Lunch" Principle:** While this design is parameter-efficient, it comes at a cost: after each stage, the spatial resolution is halved, but the number of channels doubles. Since activation memory is proportional to $H \times W \times C$, the memory required for activations roughly doubles at each stage.

11.2.3 VGG-16 Summary

- **Much Bigger than AlexNet:** VGG-16 has **138 million parameters** (2.3x AlexNet), with the vast majority again concentrated in the final fully connected layers.
- **Computationally Intensive:** It requires approximately **4 TFLOPs** to process a mini-batch of 128 images (14x AlexNet), which equates to **31 GFLOPs per image**. This high computational cost is mainly due to the convolutional layers.
- **Memory Hungry:** The memory required to store the activations for a mini-batch is enormous, around **16.5 GB** (12x AlexNet). This is largely because, unlike AlexNet, VGG does not have an aggressive "stem" layer to quickly reduce the spatial resolution. The first layers operate on large 224x224 feature maps, leading to huge memory consumption.

- **Training Challenges:** Batch Normalization had not been invented yet. Training such a deep network was difficult and required careful pre-initialization of deeper networks using weights from shallower, pre-trained versions. The full network was trained on 4 GPUs for 2-3 weeks.

11.3 GoogLeNet: Inception v1 (Szegedy et al., 2014)

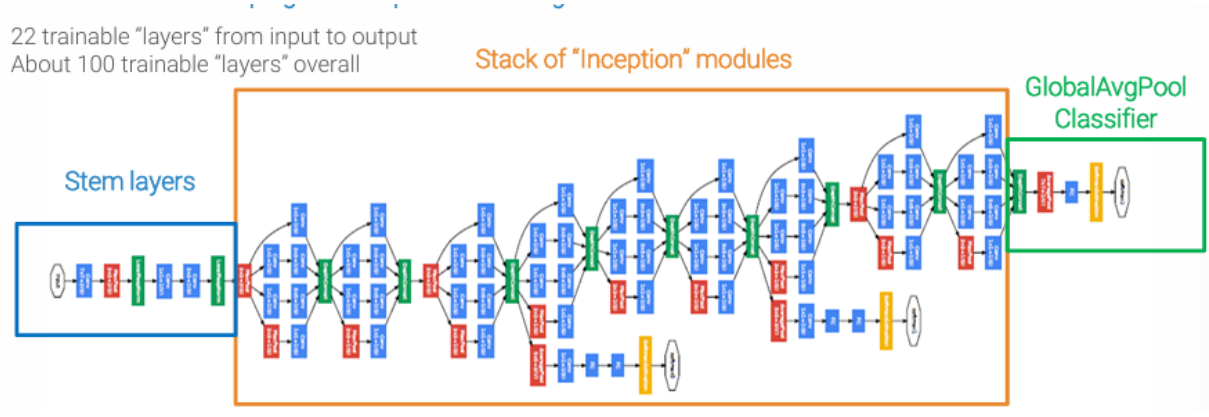
GoogLeNet, the winner of ILSVRC 2014, took a completely different approach from VGG. Its main hallmark was the **improved utilization of computing resources**.

> "The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a **carefully crafted design** that allows for increasing the depth and width of the network while keeping the **computational budget constant**."

11.3.1 Architectural Overview

GoogLeNet is a deep network (22 trainable layers, 100 layers in total) but is computationally very efficient. It is composed of three main parts:

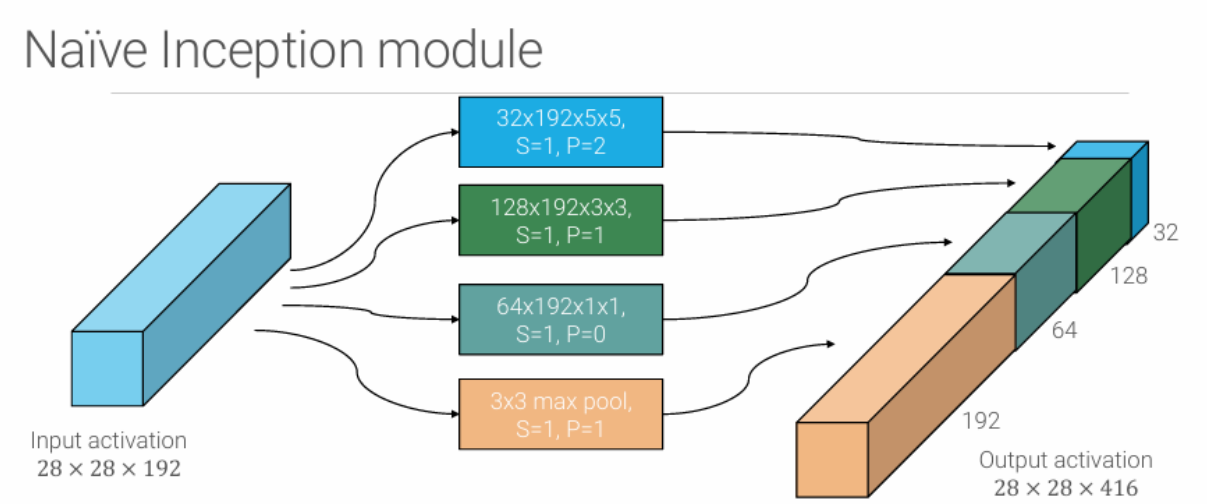
1. **Stem Layers:** An initial sequence of convolutional and pooling layers that aggressively downsamples the input from 224x224 to 28x28. This downsampling is more gentle than AlexNet's but much more efficient than VGG's, requiring only 5 layers, 130 GFLOPs, and 2 GB of memory.
2. **Stack of "Inception" Modules:** The core of the network is a stack of 9 proprietary "Inception" modules.
3. **Classifier:** Instead of large fully connected layers, it uses Global Average Pooling followed by a single FC layer.



11.3.2 The Naïve Inception Module

The core idea of the Inception module is to perform convolutions with multiple kernel sizes (1x1, 3x3, 5x5) in parallel and concatenate their outputs. This allows the network to learn features at different scales simultaneously.

However, a "naïve" implementation of this idea has two major problems:



1. **Channel Explosion:** The concatenation of features from multiple branches, especially after a max-pooling branch, causes the number of output channels to grow very rapidly.
2. **Computational Cost:** Performing large convolutions (like 5x5 and 3x3) on inputs with a high number of channels (e.g., 192) is prohibitively expensive.

11.3.3 The Power of 1x1 Convolutions

The solution to these problems is the clever use of **1x1 convolutions**, a technique popularized by the "Network in Network" paper. A 1x1 convolution has several key properties:

- It behaves like any other convolution but operates on a 1x1 spatial window.
- Its primary function is to **change the depth (number of channels)** of the activations while preserving the spatial dimensions (height and width).
- It can be interpreted as applying a tiny, linear **fully connected (FC) layer at each spatial location** across the channels.
- A stack of 1x1 convolutions is equivalent to applying a small Multi-Layer Perceptron (MLP) at each location.

11.3.4 The Inception Module with Dimension Reduction

By adding 1x1 convolutions as **bottleneck layers** before the expensive 3x3 and 5x5 convolutions, the Inception module becomes computationally feasible.

- **Controlling Time Complexity:** The 1x1 convolution first reduces the channel dimension of the input to the larger convolutions, dramatically cutting down the number of FLOPs. For example, a 3x3 convolution on a 192-channel input is much more expensive than a 1x1 conv (192 $\rightarrow$ 96 channels) followed by a 3x3 conv (96 $\rightarrow$ 128 channels). This simple trick reduced the computational cost of the 3x3 branch from 350M FLOPs to 202M FLOPs in the example.
- **Controlling Output Channels:** A 1x1 convolution is also added after the max-pooling branch to reduce its channel depth, preventing the channel explosion problem.

11.3.5 Global Average Pooling (GAP)

To further reduce the number of parameters, GoogLeNet replaces the massive fully connected layers of AlexNet and VGG with **Global Average Pooling (GAP)**.

- **Method:** Instead of flattening the final feature map (e.g., 7x7x1024) into a huge vector and feeding it to an FC layer, GAP averages each channel's feature map down to a single value. A 7x7x1024 activation becomes a 1x1x1024 vector.
- **Benefit:** This gets rid of the spatial dimensions and drastically reduces the number of parameters needed in the final classifier. GoogLeNet uses GAP followed by a single FC layer, resulting in only **1 million parameters** for the classifier (compared to over 100 million in VGG).
- **Adaptive Pooling:** If the pooling kernel size is computed automatically to match the input activation size (as in PyTorch's 'AdaptiveAvgPool2d'), the network becomes able to process input images of any size.

11.3.6 GoogLeNet Summary

GoogLeNet is a landmark architecture due to its efficiency:

- **Only 7 million parameters** in total.
- **390 GFLOPs** per mini-batch (3 GFLOPs/img).
- Requires only **3.3 GB of memory** for a mini-batch.
- Achieved a 7.89% error rate on ILSVRC 2014 with aggressive cropping.

11.4 Inception v3 (Szegedy et al., 2015)

Inception v3 builds upon the principles of GoogLeNet, further improving computational efficiency and accuracy by introducing the idea of **convolution factorizations**. The goal is to reduce the number of parameters, making the model more disentangled and thus easier to train. This architecture achieved a 4.48% top-5 error on ILSVRC12.

The key idea is to replace larger convolutions with smaller, asymmetric ones:

- **Factorizing large convolutions (Factorization A):** A 5x5 convolution is replaced by two stacked 3x3 convolutions. This was already a principle in VGG, but Inception applies it more broadly.
- **Factorizing into asymmetric convolutions (Factorization B):** A standard $n \times n$ convolution is replaced by a stack of a $1 \times n$ convolution followed by an $n \times 1$ convolution. For example, a 3x3 convolution can be factorized into a 1x3 and a 3x1 convolution. This factorization is particularly effective on mid-scale feature maps (e.g., 17x17).
- **Factorization C** applies similar principles for coarse-scale activations (e.g., 8x8).

These factorizations significantly reduce computational cost and the number of parameters while maintaining or even improving the expressive power of the network.

11.5 Residual Networks (ResNet) (He et al., 2015)

ResNet represents one of the most significant breakthroughs in deep learning, enabling the training of networks that are orders of magnitude deeper than what was previously possible.

11.5.1 The Degradation Problem

The lesson from VGG was that increasing network depth generally improves performance. However, experiments showed that simply stacking more layers on top of a "plain" deep network led to a surprising problem: the performance would saturate and then rapidly **degrade**.

Crucially, this was not just overfitting. The experiments showed that a 56-layer plain network had a **higher training error** than a 20-layer plain network. This indicates a **training problem**: the optimization process was struggling to find a good solution for the deeper network, even with Batch Normalization.

This is paradoxical because a solution exists by construction: a deeper model should be able to achieve at least the same performance as a shallower one. If a 20-layer network achieves performance X, one could construct a 56-layer network by taking the 20-layer network and stacking 36 **identity layers** (layers that just pass their input through) on top. This deeper network would have the exact same performance X. The fact that SGD is **not able to find this solution** reveals that optimizing very deep networks is fundamentally hard.

11.5.2 The Solution: Residual Blocks

The proposed solution is to change the network's structure to make learning identity functions trivial. This is achieved by introducing **residual blocks**, implemented with **skip connections** (or shortcut connections).

Instead of forcing a stack of layers to learn a desired underlying mapping $H(x)$, we let it learn a **residual mapping** $F(x) = H(x) - x$. The output of the block then becomes:

$$y = F(x) + x$$

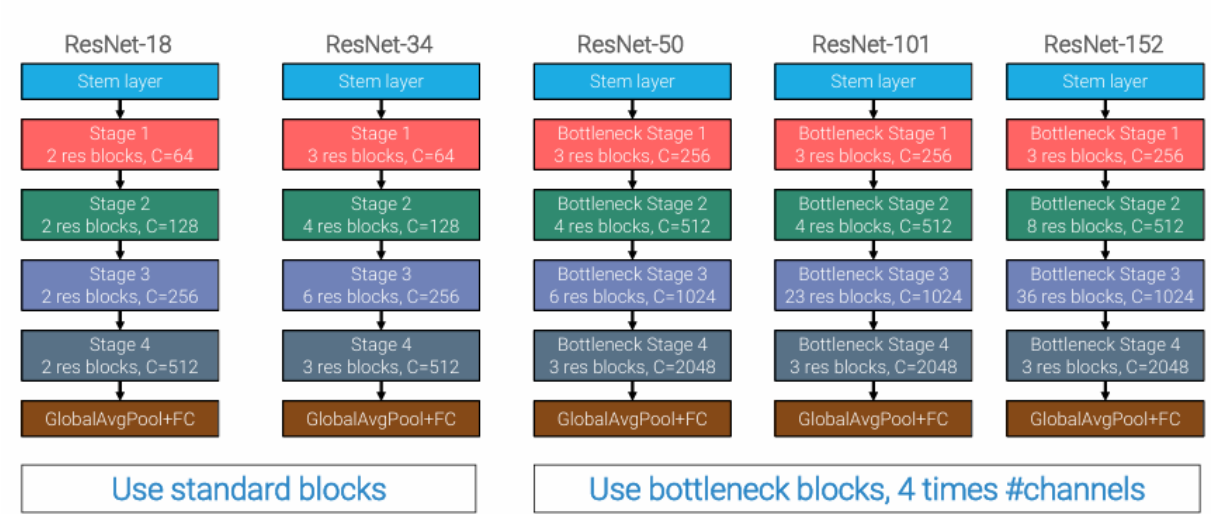
If the identity mapping is optimal, the optimizer can easily achieve this by driving the weights of the layers in $F(x)$ to zero. The network then learns an "optimal" perturbation of the identity function. This reformulation makes the optimization problem much easier.

- **Implementation:** A typical residual block consists of two 3x3 convolutional layers. The input x is added to the output of the second layer before the final ReLU activation.
- **Initialization:** The weights are usually initialized to be very small (or biases to 0), so the network starts close to an identity function and learns the residual perturbations from there.
- **Batch Normalization:** ResNet makes heavy use of Batch Normalization after each convolution.

11.5.3 ResNet Architecture

Inspired by VGG's regular design, a full ResNet is built as a stack of **stages**.

- Each stage is a stack of several residual blocks.
- The first block of each new stage is responsible for downsampling: it halves the spatial resolution (using a stride-2 convolution) and doubles the number of channels.
- The architecture uses a **stem layer** and **Global Average Pooling** at the end, similar to GoogLeNet.
- The naming convention follows VGG: **ResNet-X**, where X is the number of layers with learnable parameters (e.g., ResNet-18, ResNet-34, ResNet-50).



11.5.4 Handling Dimension Mismatches in Skip Connections

When a residual block downsamples (e.g., at the start of a new stage), the input x to the skip connection has different spatial dimensions and channel counts than the output of the convolutional path. The authors tried two solutions:

1. Zero-pad the input to match the increased channel dimension (no extra parameters).
2. Use a **1x1 convolution with stride 2** on the skip connection to perform the downsampling and increase the channels.

The second option was found to perform slightly better and is the standard approach.

11.5.5 Bottleneck Residual Blocks

For very deep networks (like ResNet-50 and beyond), a more efficient **bottleneck block** is used to reduce computational cost. Instead of two 3x3 convolutions, the block uses a stack of:

1. A **1x1 convolution** to reduce the number of channels (the "bottleneck").

2. A **3x3 convolution**.

3. A **1x1 convolution** to restore the number of channels.

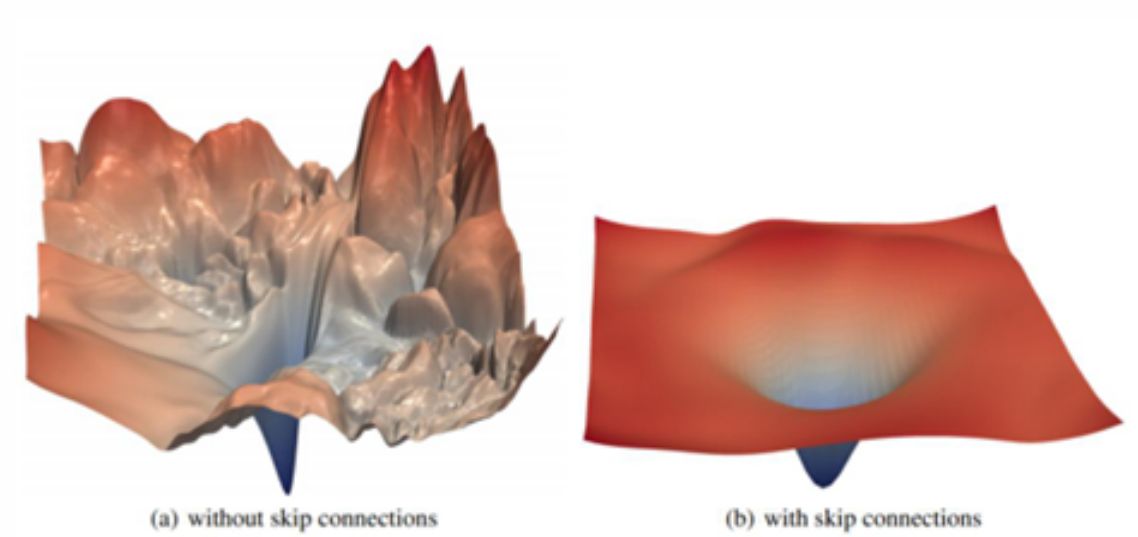
This design allows for building much deeper networks with approximately the same number of parameters and FLOPs as a standard block, enabling faster increases in depth without altering the computational budget. ResNet-50 and deeper variants use these bottleneck blocks, typically with a channel expansion factor of 4.

11.5.6 Effects and Interpretation of Residual Learning

- **Results:** ResNet was a resounding success, winning 1st place in all five main tracks of the ILSVRC & COCO 2015 competitions. The "ultra-deep" 152-layer version demonstrated that properly trained deep networks outperform shallower ones, as expected. ResNet remains a standard backbone for most computer vision tasks today.

- **Smoother Loss Landscape:** Visualizations show that residual connections dramatically smooth the loss landscape. A network without skip connections has a very rugged, chaotic loss surface, while the ResNet equivalent has a smooth, convex-like basin, making optimization much easier.

- **Ensemble of Shallow Networks:** One interpretation is that a ResNet does not behave like a single, ultra-deep network but rather as an implicit **ensemble of many relatively shallow networks**. The skip connections create a multitude of paths of varying lengths through the network. Most of the gradient flow is concentrated through the shorter paths, meaning the network relies more on these shallower ensembles.



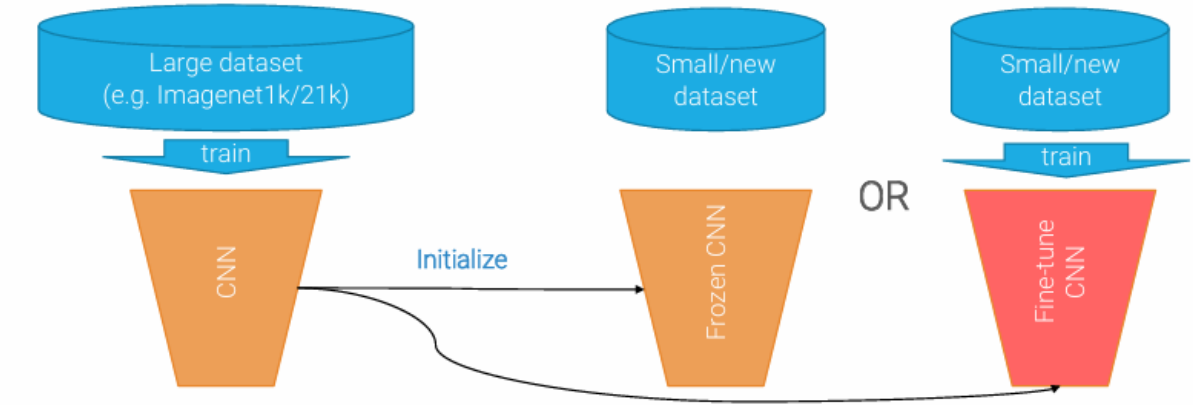
11.5.7 Further Model Tweaks ("ResNet v2")

Subsequent research proposed further refinements to the ResNet architecture:

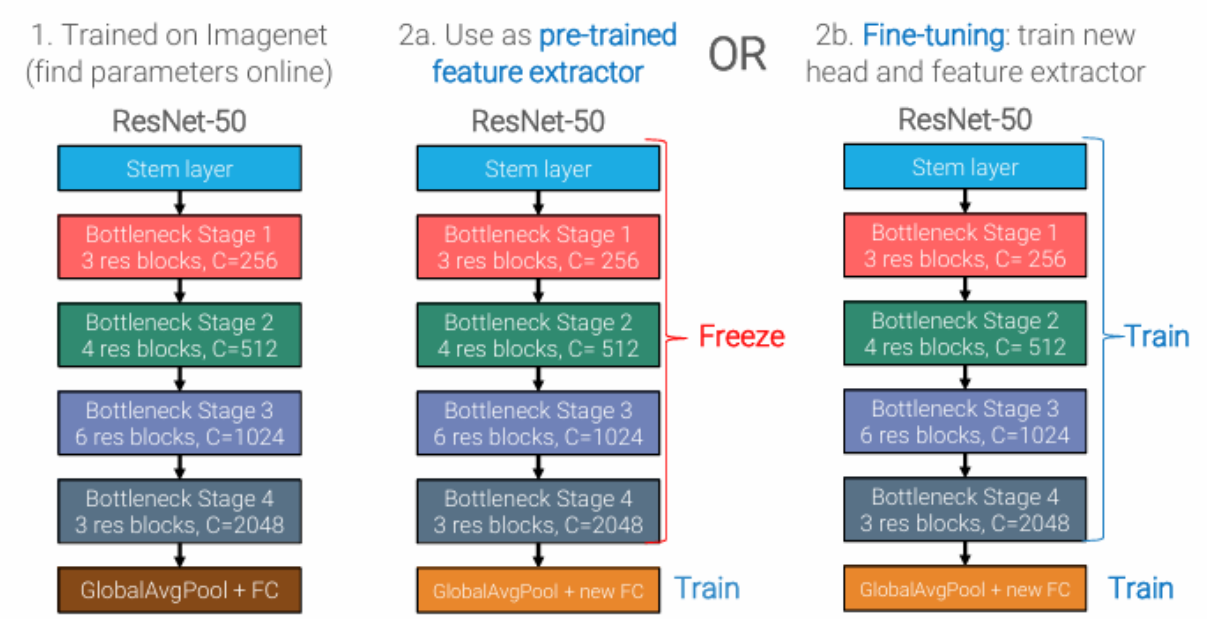
- **ResNet-B:** Moves the stride-2 downsampling from the first 1x1 convolution to the 3x3 convolution in the bottleneck block, ensuring that no spatial information is ignored.
- **ResNet-C:** Replaces the aggressive 7x7 stride-2 stem layer with three stacked 3x3 convolutions (the first with stride 2), which is more efficient.
- **ResNet-D:** Fixes an issue where the 1x1 stride-2 convolution in the skip connection only "sees" 3/4 of the input activation. This is solved by placing a 2x2 average pooling layer with stride 2 before this convolution.

11.6 Transfer Learning

From a practical standpoint, one of the most important features of these powerful, pre-trained architectures is that their learned representations can be effectively **transferred** to new datasets and tasks.



- **Process:** We typically pre-train a network on a very large dataset (e.g., ImageNet). Then, we adapt this pre-trained network for a new, often smaller, dataset.
- **Two Main Strategies:**
  1. **Frozen Feature Extractor:** The convolutional base of the pre-trained network is "frozen" (its weights are not updated). A new, small classifier is added on top and trained on the new dataset. This is fast and effective for small datasets.
  2. **Fine-Tuning:** The entire network (or parts of it) is trained on the new dataset, but with a much smaller learning rate than was used for the original pre-training. This allows the network to adapt its learned features to the new task.
- **Fine-Tuning Recipe:** A common recipe for fine-tuning is to start by training only the new classifier head, then "unfreeze" the rest of the network and train it with a smaller learning rate, possibly using progressive LRs (even smaller LRs for lower layers).


1. Trained on Imagenet
(find parameters online)
ResNet-50
Stem layer
Bottleneck Stage 1
3 res blocks, C=256
Bottleneck Stage 2
4 res blocks, C=512
Bottleneck Stage 3
6 res blocks, C=1024
Bottleneck Stage 4
3 res blocks, C=2048
GlobalAvgPool + FC
2a. Use as pre-trained feature extractor
OR
ResNet-50
Stem layer
Bottleneck Stage 1
3 res blocks, C= 256
Bottleneck Stage 2
4 res blocks, C= 512
Bottleneck Stage 3
6 res blocks, C=1024
Bottleneck Stage 4
3 res blocks, C=2048
GlobalAvgPool + new FC
Freeze
Train
2b. Fine-tuning: train new head and feature extractor
ResNet-50
Stem layer
Bottleneck Stage 1
3 res blocks, C= 256
Bottleneck Stage 2
4 res blocks, C= 512
Bottleneck Stage 3
6 res blocks, C=1024
Bottleneck Stage 4
3 res blocks, C=2048
GlobalAvgPool + new FC
Train
Train

12 Regularization and Training Recipes

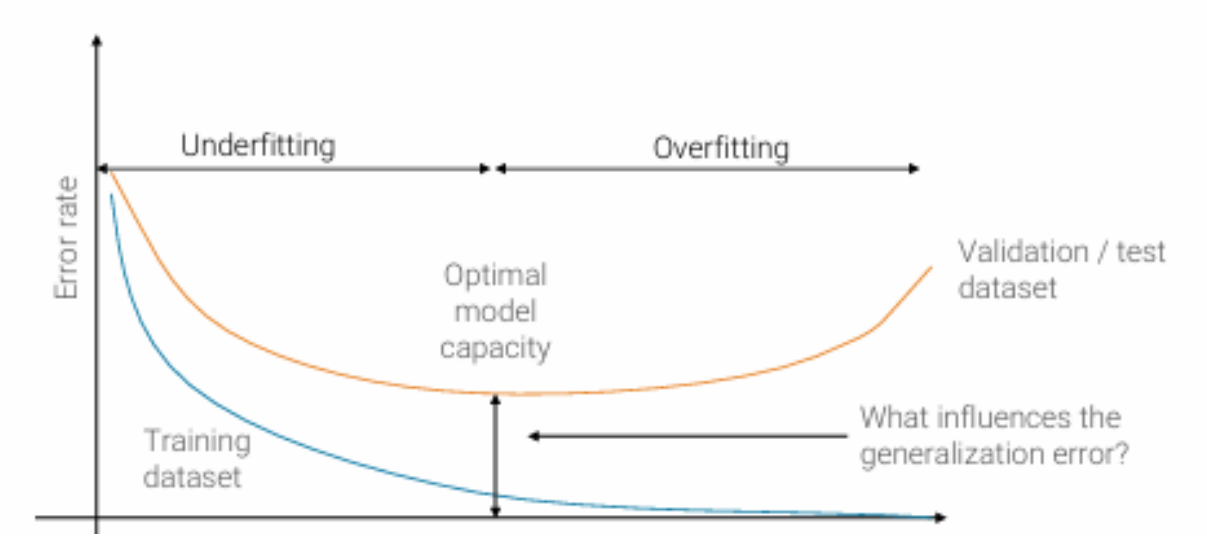
12.1 Generalization Error and Model Capacity

12.1.1 Understanding Generalization Error

The primary objective of training a machine learning model is not merely to achieve high accuracy on the training data, but to **generalize** effectively to new, unseen data. The **generalization error** (or test error) quantifies a model's performance on this unseen data.

The relationship between a model's performance and its complexity, or **capacity**, is typically illustrated by plotting the training and validation error rates against model capacity:

- **Training Error:** As model capacity increases, the training error consistently decreases. A model with sufficient capacity can eventually memorize the entire training set, driving this error towards zero.
- **Validation/Test Error:** This error follows a U-shaped curve. It initially decreases as the model learns the true underlying patterns in the data. However, beyond a certain **optimal model capacity**, the validation error begins to rise. This indicates **overfitting**, where the model starts fitting the noise and specific quirks of the training data, harming its performance on new data.



Model capacity

The region to the left of the optimal point is known as **underfitting**, where the model is too simple to capture the data's structure. The region to the right is **overfitting**. The difference between the training and validation error curves is referred to as the **generalization gap**.

12.1.2 Theoretical vs. Effective Capacity

Theoretical Capacity is an intrinsic property of a model's architecture. It is determined by design choices such as:

- Network width (number of channels or neurons per layer).
- Network depth (number of layers).
- The resolution of the input images.
- The specific architectural design itself.

In modern deep learning, we typically design architectures with very high theoretical capacity, often far beyond what is needed to simply fit the training data.

However, the capacity we leverage in practice is the **Effective Capacity**. This is not fixed but is dynamically controlled by the entire training process. Specifically, **optimization hyperparameters** and **regularization** techniques can constrain a high-capacity model, preventing it from using its full theoretical power and thus reducing overfitting. Key factors influencing effective capacity are:

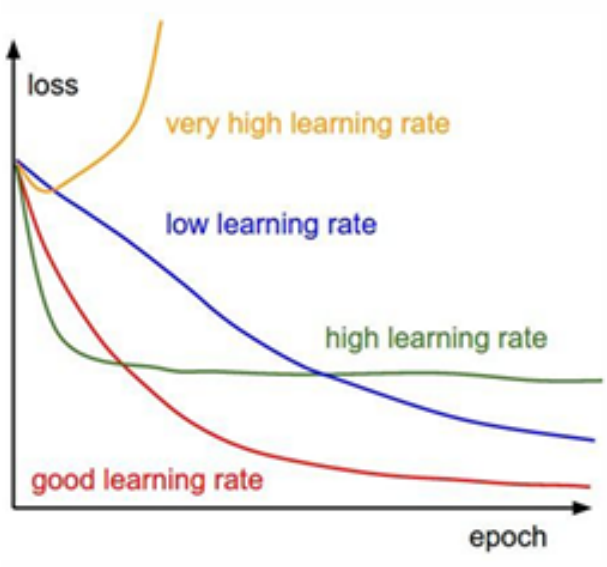
- Learning rate and its schedule.
- Methods that favor small parameter values (e.g., weight decay).
- Total training time (e.g., early stopping).

The concept is clearly demonstrated with an example of generalized linear regression, fitting a noisy cubic function with polynomials of varying degrees (K):

- A **linear model** ($K = 1$) is in the **underfitting** regime. Its capacity is too low to capture the cubic trend.
- A **cubic model** ($K = 3$) represents the **"right"** model. Its capacity matches the underlying data-generating process, achieving low error on both training and test sets.
- A **high-degree polynomial model** ($K = 9$) is an "interpolating" model that **overfits**. It has excessive capacity, perfectly fitting every training point (including noise) but failing disastrously on the test set, as shown by its huge negative R^2 score.

- **Very High LR:** The optimization process becomes unstable, and the loss may diverge (increase) or oscillate wildly.
- **High LR:** The loss will often **decrease very fast in the first few iterations** but then **get stuck**, plateauing at a high value. The optimizer bounces around a wide area of the loss landscape without being able to descend into a good minimum.
- **Low LR:** Convergence is reliable but extremely slow. There's also a risk of getting trapped in a poor local minimum that a higher LR might have escaped.
- **Good LR:** The loss decreases steadily and efficiently, converging to a low value.

12.2 The Learning Rate: A Key Hyperparameter



The learning rate (LR) is one of the most critical hyperparameters for successful training. Its value directly impacts the convergence of the loss function:

- **Very High LR:** The optimization process becomes unstable, and the loss may diverge (increase) or oscillate wildly.
- **High LR:** The loss will often **decrease very fast in the first few iterations** but then **get stuck**, plateauing at a high value. The optimizer bounces around a wide area of the loss landscape without being able to descend into a good minimum.
- **Low LR:** Convergence is reliable but extremely slow. There's also a risk of getting trapped in a poor local minimum that a higher LR might have escaped.
- **Good LR:** The loss decreases steadily and efficiently, converging to a low value.

Finding a single, fixed "good" LR is hard. The practical solution is to use a **learning rate schedule**, which dynamically adjusts the LR during training, often by starting high and decreasing it over time.

12.2.1 Intuition: Favoring Wide Minima and the Role of Schedules

A primary goal of using a learning rate schedule is to guide the optimizer towards a **wide, flat minimum** rather than a sharp, narrow one. The intuition, first proposed by Hochreiter and Schmidhuber (1997), is that flat minima generalize better.

- **Sharp Minima:** A model in a sharp minimum is sensitive to small changes. The loss landscape for the test set is slightly different from the training set; if the training minimum is sharp, the corresponding point on the test landscape could have a much higher loss.
- **Flat Minima:** A model in a flat minimum is more robust. A small shift between the training and test loss landscapes doesn't drastically change the loss, leading to better generalization.

The lecture notes also allude to a more complex, ongoing research debate (Dinh et al., 2017; Keskar et al., 2017), suggesting that the relationship between flatness and generalization is not always straightforward and that some sharp minima may generalize well. However, for practical purposes, favoring wide minima is a successful and widely adopted heuristic.

A high LR early in training encourages exploration, allowing the optimizer to "jump out" of poor, sharp minima. A subsequent lower LR allows it to settle into a good, wide minimum.

12.2.2 Learning Rate Schedules

12.2.3 Step Decay

This is a simple, powerful, and widely used schedule, notably in the original ResNet paper.

- **Method:** Start with a relatively high learning rate (e.g., 0.1). Train for a fixed number of epochs, or until the validation error stagnates. Then, divide the learning rate by a constant factor (e.g., 10). This process is repeated a few times during training.
- **Effect:** The sharp drops in LR help the optimizer, which might be oscillating around a solution, to take smaller steps and converge more precisely.

12.2.4 Cosine Decay

This schedule offers a smooth, continuous alternative to the discrete drops of step decay.

- **Method:** For a training process of E epochs, the learning rate lr_e for epoch e follows a cosine curve from an initial rate lr_0 to a final rate lr_E.

$$lr_e = lr_E + \frac{1}{2}(lr_0 - lr_E)\left(1 + \cos\left(\frac{e\pi}{E}\right)\right)$$

- **Advantage:** It provides a similar optimization path to step decay but is fully automatic, with fewer hyperparameters to tune (no need to decide the exact epochs for decay).

12.2.5 Warm-up

While most schedules start with a high LR, this can be problematic for some networks.

- **Problem:** A high initial learning rate can cause instability, especially in very deep networks or those with random initialization. This may manifest as the model's accuracy remaining at chance levels for several epochs, which is a symptom of poor initialization.
- **Solution: Warm-up.** The training begins with a **very low learning rate for a few epochs**. This allows the model's weights to adapt gently from their initial state. After this warm-up period, the learning rate is increased to its target high value, and the main decay schedule begins. This is beneficial even for shallower networks, as it helps navigate potentially difficult parts of the initial loss landscape.

12.2.6 One-Cycle Schedule

This advanced schedule, a key component of the "Super-Convergence" training methodology, modifies the LR **at every iteration (mini-batch)**, not just every epoch.

- **Method:** It consists of two main phases within a single cycle that spans the entire training duration.

  1. **Ramp-up Phase:** The LR increases from a low initial value (lr_0) to a high maximum value (lr_{max}). This phase typically lasts for a fraction p of the total iterations (e.g., $p = 0.3$).
  2. **Ramp-down Phase:** The LR decreases from lr_{max} down to a near-zero final value (lr_{min}).

  The slide notes that the original proposal had three phases, but modern implementations (like in PyTorch) typically use a two-phase cycle, often realized with **cosine segments** for both the ramp-up and ramp-down for a smoother transition.

- **Intuition:** The initial ramp-up acts as an efficient form of exploration. The long ramp-down allows the optimizer to converge deeply into a stable, wide minimum. This method can lead to much faster training times.

12.3 Regularization

Regularization is any modification made to the training process (or "recipe") with the primary goal of **reducing the generalization (test) error**. Crucially, this is done often at the expense of the training error, meaning a regularized model may fit the training data less perfectly but will perform better on unseen data. The aim is to close the generalization gap.

12.3.1 A General Template for Regularization

Many powerful regularization techniques, including Dropout and Batch Normalization, follow a general template:

- **Training time: Add randomness.** The model's output for a given input is stochastic, influenced by a random component.

$$\text{scores} = f(x; \theta, m) \quad (\text{where } m \text{ is a random variable})$$

- **Test time: Average out the randomness.** The stochastic component is removed, typically by marginalizing it out through expectation. This makes the model's output deterministic.

$$f(x; \theta) = \mathbb{E}_m[f(x; \theta, m)] = \sum_{\text{all masks } m} p(m) f(x; \theta, m)$$

For example, in **Batch Normalization**, an activation $a_j^{(i)}$ is normalized using the mean μ_j and variance ν_j of the *current mini-batch*. Since the mini-batch is random, the normalization is stochastic. At test time, μ_j and ν_j are fixed to the running averages seen during training, making the operation a deterministic linear transformation.

12.3.2 Parameter Norm Penalties

This classic regularization technique adds a penalty term to the loss function to discourage large parameter values. The total loss is:

$$L(\theta; D_{\text{train}}) = \underbrace{L_{\text{task}}(\theta; D_{\text{train}})}_{\text{Data Loss}} + \lambda \quad \underbrace{L_{\text{reg}}(\theta)}_{\text{Regularization Loss}}$$

Commonly used norms are L_2 **Regularization** ($L^{\text{reg}}(\theta) = \sum_i \theta_i^2$) and L_1 **Regularization** ($L^{\text{reg}}(\theta) = \sum_i |\theta_i|$). By penalizing large weights, these norms limit the model's effective capacity.

12.3.3 L_2 Regularization and Weight Decay

In deep learning, L_2 regularization is widely known as **weight decay**. In plain SGD, its gradient update rule demonstrates this name:

$$\theta^{(i+1)} = \underbrace{(1 - lr \cdot \lambda)\theta^{(i)}}_{\text{Decayed parameter}} - lr \cdot \nabla_\theta L_{\text{task}}(\theta^{(i)}; D_{\text{train}})$$

At each step, the weight is first shrunk (decayed) toward zero before the task gradient is applied.

12.3.4 Early Stopping

The duration of training is a hyperparameter controlling effective capacity. **Early stopping** involves monitoring performance on a validation set and halting training when the validation error stops improving, thereby preventing the model from overfitting in later epochs.

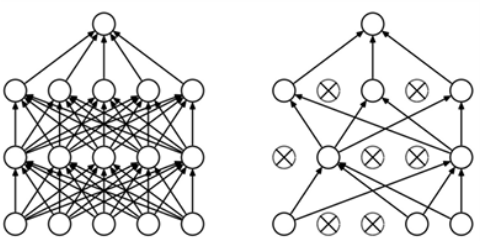
12.3.5 Label Smoothing

Standard cross-entropy loss with one-hot targets encourages the model to become overconfident. **Label smoothing** mitigates this by using "soft" targets. For a small ϵ, the target for the correct class becomes $1-\epsilon$ and other classes get $\epsilon/(C-1)$. This discourages extreme logit values and improves generalization. This can be implemented in modern frameworks directly or via KL Divergence loss for custom soft targets.

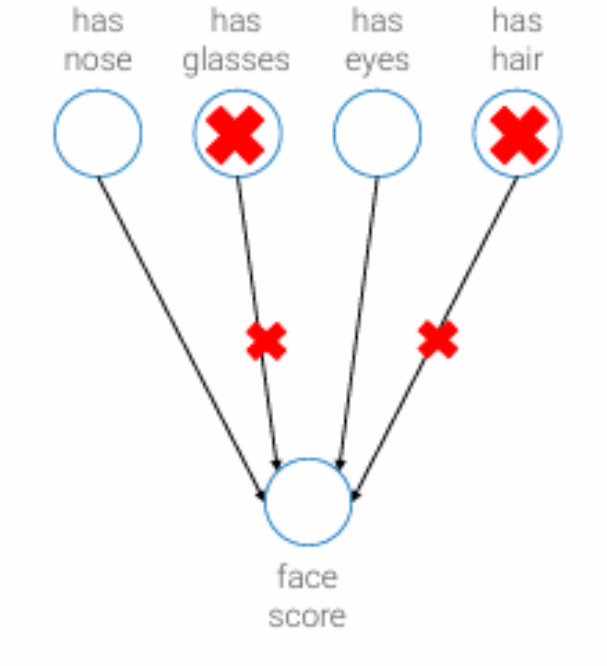
12.3.6 Dropout

Dropout is a powerful regularization technique with several interpretations.

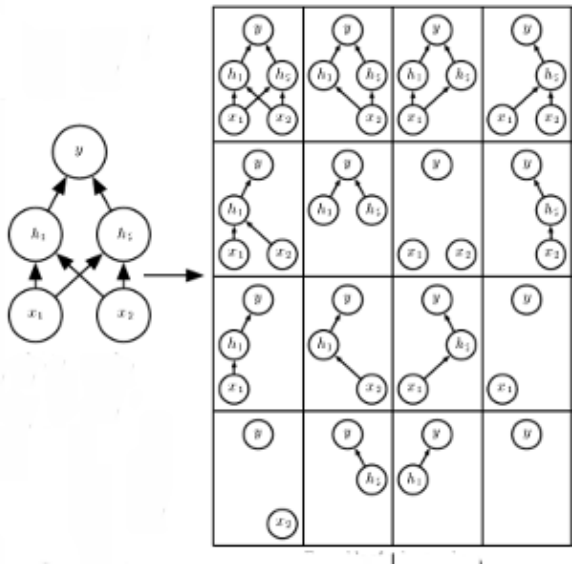
- **Method:** During training, neuron activations are randomly set to zero with some probability $1 - p$. It's usually applied to layers with many parameters, e.g. all but last FC layers in AlexNet, VGG, between GlobalAveragePooling and last fc layer in Inception-ResNet-v2. It's not used in ResNets and variants, for its unclear interactions with BatchNorm. EfficientNet uses it again, and scales p linearly from 0.2 to 0.5 going from B0 to B7



- **Anti-Conspiracy Strategy:** Hinton described dropout as an "anti-conspiracy" mechanism. It forces each neuron to learn useful features independently, without relying on specific other neurons. It acts as **disruptive noise** that regularizes units to be useful in many contexts and can destroy selective information, forcing the network to build more robust representations.



- **Ensemble Interpretation:** Dropout can be seen as training a massive ensemble of thinned-out "sub-networks" that all **share weights**. It is a special form of **bagging**, a concept also central to Random Forests.



- **Test Time:** The randomness is removed. The standard implementation uses **inverted dropout**, where activations are scaled by $1/p$ during training, leaving the test-time forward pass unchanged.
- **Usage:** Dropout is typically applied to layers with many parameters (like FC layers). Its interaction with Batch Normalization can be complex, and it is often not used in standard ResNets but has seen a resurgence in architectures like EfficientNet.

** Dropout makes predictions at training time stochastic, as the output depends not only on the network's parameters θ but also on a randomly sampled mask m:

$$\text{scores} = f(x; \theta, m)$$

However, at test time, we need the output to be deterministic. The principled solution is to "average out" this stochasticity by computing the expected value of the output over all possible random masks. This is equivalent to averaging the predictions of the entire ensemble of thinned networks:

$$\text{scores} = f(x; \theta) = \mathbb{E}_m[f(x; \theta, m)] = \sum_{\text{all masks } m} p(m) f(x; \theta, m)$$

While this is the correct theoretical approach, directly computing this sum is computationally infeasible due to the astronomical number of possible masks. As the slide notes, we could approximate this sum by running a test image through the network multiple times with different masks and averaging the results, but this would make inference prohibitively slow. This practical limitation necessitates a more efficient approximation, which will be addressed next.

12.3.7 Stochastic Depth

This technique applies a dropout-like idea to entire blocks in ResNet architectures. During training, the convolutional path of a residual block is randomly dropped with some probability, effectively shortening the network for that forward pass. The survival probability typically decreases with depth. At test time, the full, unmodified network is used.

12.4 Data Augmentation

Data augmentation creates new training samples by applying **label-preserving** transformations to existing data. It is a highly effective method for improving generalization.

- **Multi-scale Training:** Involves randomly resizing an image and then taking a random crop. This trains the model to be robust to changes in object scale and position.
- **FixRes:** A strategy to fix the train-test resolution discrepancy caused by data augmentation. It involves fine-tuning the classifier on larger crops at the end of training to better match the test-time distribution.
- **Cutout (Random Erasing):** A random rectangular region of the input image is masked out. This acts as "dropout in the input space" and forces the network to learn from a more diverse set of features.
- **Mixup:** Linearly interpolates between two random training images and their one-hot encoded labels. This forces the model to have smoother decision boundaries.
- **CutMix:** Cuts a patch from one image and pastes it onto another. The labels are mixed according to the area of the patch. It has proven to be a very strong regularization strategy.

12.5 Training Recipes and Best Practices

12.5.1 Random Hyper-parameter Search

Finding the best hyperparameters (like learning rate, weight decay) is crucial.

- **Grid Search:** Exhaustively tries all combinations on a predefined grid. This is inefficient if some hyperparameters are more important than others.
- **Random Search:** Samples hyperparameter values randomly from a distribution (e.g., log-uniform). This is more efficient as it explores the parameter space more effectively, dedicating more trials to finding good values for important parameters. However, one must be aware of the curse of dimensionality.

12.5.2 Test-Time Good Practice: Ensembles

A reliable way to boost performance by 1-2% is to use an ensemble.

1. Train multiple models (e.g., 5-10) with the same architecture but different random initializations.
2. At test time, run the input through all models and average their predictions (e.g., average the logits before the final argmax).

This works because different models, even with similar error rates, tend to make different mistakes. The main downsides are the high computational cost for both training and inference.

12.5.3 Exponential Moving Average (EMA)

EMA (or Polyak averaging) is a cheaper alternative to building full ensembles. Instead of averaging predictions from different models, it averages the weights of a single model over time.

- During training, a separate copy of the model's parameters is maintained.
- This copy is updated at each step as an exponential moving average of the main training parameters.
- This averaged set of weights is then used for inference at test time.

12.5.4 A "Recipe" for Training Neural Networks

The lecture highlights Andrej Karpathy's practical "recipe" for training neural networks, emphasizing that success comes from being **thorough, defensive, paranoid, and obsessed with visualizations**. The process is iterative and requires patience and attention to detail.

1. **Become one with the data:** Look at your data, understand its properties, and identify potential issues.
2. **Set up the end-to-end skeleton + get dumb baselines:** Build the full training and evaluation pipeline, test it with simple models, check the initial loss, and ensure it can overfit a small subset of the data.
3. **Overfit:** Start with a known, powerful model (e.g., ResNet) and a good optimizer (e.g., Adam) and try to achieve a very low training error.
4. **Regularize:** Once the model can overfit, apply a suite of regularization techniques (data augmentation, weight decay, dropout, etc.) to reduce the validation error.
5. **Tune:** Use random search to find better hyperparameters around the configuration that worked in the previous steps. Use learning rate schedules.
6. **Test-time optimizations:** Squeeze out final performance gains with techniques like ensembles or EMA.

A PyTorch case study on training a ResNet50 demonstrates that applying these techniques cumulatively leads to massive gains in accuracy, but the process is not linear and involves significant backtracking and experimentation. The key takeaway is that the **training recipe matters tremendously**.