Computer Vision

Matteo Galiazzo

June 5, 2025

Contents

1 Introduction

The difference between computer vision and image processing is the fact that computer vision is the process of extracting information from images, while image processing aims at improving the quality of images. Quite often image processing helps computer vision. The informations we want to extract from the images could be counting, object orientation, object classification, measurements... Computer vision is challenging since we lose depth from the images (3D information becomes 2D), scale and illumination varies, and there's object occlusion (object hiding other objects).

Computer vision started with hand-crafted decision rules that only required few images as example, and evolved to machine learning where the algorithm learns a decision rule, but the training requires hundreds to thousands of images. The big paradigm shift happened with deep learning (e2e learning) which learned both the image representation and the decision rules, but required thousands to millions of words. Deep learning has been enabled by better networks, better hardware and more data.

2 Fundamentals of Image Processing and Computer Vision

2.1 Images

An imaging device gathers the light reflected by 3D objects to create a 2D representation of the scene.

2.1.1 Pinhole camera model

The "pinhole camera" is the simplest camera model we can define. Light goes through the very small pinhole (to not have saturation) and hits the image plane. Geometrically, the image is achieved by drawing straight rays from scene points through the hold up to the image plane.

This simple geometrical model turns out to be a good approximation of the geometry of image formation. However, useful images can hardly be captured by means of a pinhole camera.

The **geometric model of image formation** in a pinhole camera is known as **perspective projection**.

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

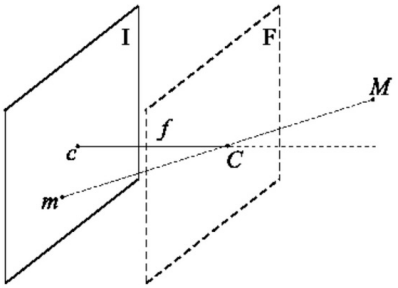


Figure 1: perspective projection

Which, by writing the points as vectors and the plane's coordinates, becomes the geometric model in the image 2

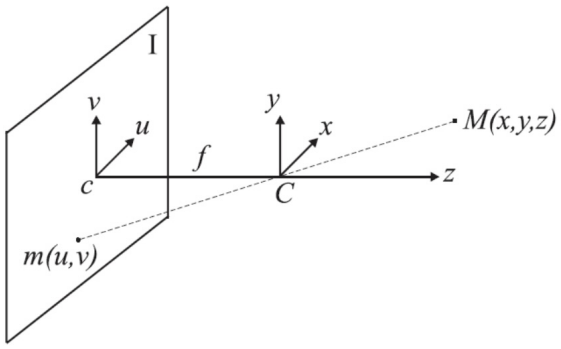


Figure 2: geometric model of image formation

Given the reference frame in the image 2

- 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. It's called the **camera reference system** because it is "attached" to the camera.

For the perspective model these axis must be parallel

The equations to map scene points into their corresponding image points are defined as:

$$\frac{u}{x} = -\frac{f}{z} \rightarrow u = -x\frac{f}{z} \qquad \frac{v}{y} = -\frac{f}{z} \rightarrow v = -y\frac{f}{z}$$

The minus sign means the axis gets inverted (as we can see in the visualization, and it's what happens in the brain). We can get rid of the sign, since the image plane can be though of as lying in front rather than behind the optical centre.

Image coordinates are a scaled version of scene coordinates (function of depth). When z increases, since it's at the denominator in both the equations, the terms gets smaller (object gets smaller in the image). When f increases, since it's at the numerator in both the equations the term gets bigger (object gets bigger in the image)

As we previously said, the image formation process deals with mapping a 3D space onto a 2D space, and so to the loss of depth information. A given scene point is mapped into an image point, but an image point is mapped onto a 3D line. For an image point we can only state that its corresponding scene point lays on a line, but cannot disambiguate a specific 3D point along the line.

2.1.2 Stereo images

We use multiple images to create stereo vision. Given correspondences, 3D information can be recovered easily by triangulation. We can use two cameras, or two cameras and an infrared sensor to project guides to align the images.

For standard stereo geometry there are some assumptions we have to make:

- The cameras have parallel (x, y, z) axes.
- The image planes of both cameras are coplanar and aligned.
- Both cameras have identical focal lengths.

Based on this, the transformation between the two reference frames is just a translation, usually horizontal. For stereo vision is also really important to **sense two images at the same moment**.

The cameras are displaced at a given quantity b called baseline. The **disparity** is the difference between the horizontal coordinates in the left and right images.

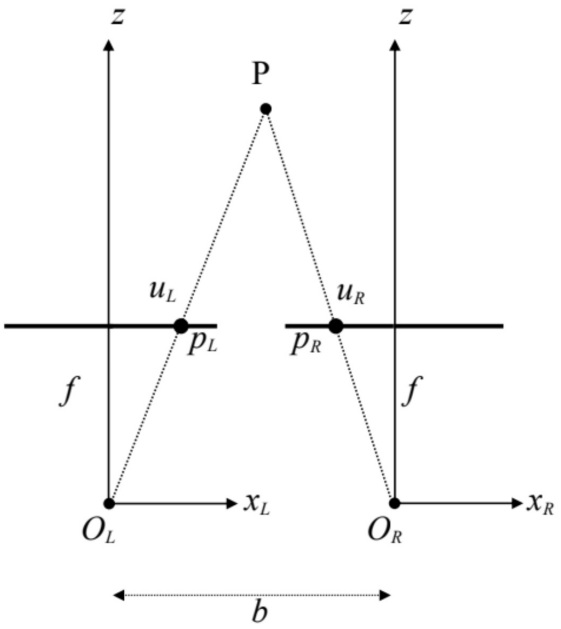


Figure 3: standard stereo geometry

In standard stereo geometry since we are given just two 2D images there is no info about the correspondence between two points in the two images. We can recall that the camera have parallel axes, and so we know that we can search for the correspondence along the horizontal lines. This task is called stereo matching.

2.1.3 Stereo correspondence

In stereo correspondence, given a point in one image, we have to find it in the other image which is the projection of the same 3D point. Such image points are called corresponding points. **Points farther away have a smaller disparity, while close points have a larger disparity**.

Figure 4: Corresponding points look similar in the two images

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:

$$l = L\frac{f}{z}$$

This relationship is more complicated for an arbitrarily oriented 3D segment, as its position and orientation need to be accounted for as well. For a **given position and orientation, length always shrinks alongside distance**.

Perspective projection maps 3D lines into image lines. **Parallelism between 3D lines is not preserved** (except for lines parallel to the image plane). This is the reason why if we look at a really long road into the distance we have the perception that the road becomes thinner, and the lines of the road intersect in the distance. The images of parallel 3D lines intersect at a point, called **vanishing point**, which isn't necessarily within the image.

If the lines are parallel to the image plane they meet at infinity.

2.1.4 Epipolar geometry

What if the two cameras are no longer aligned? Do we need to search through the whole image? We can project the line related to point P_L in the right plane and search across that line. The issue is that this projection can be computed only if the transformation between the two cameras is known (the relative mapping between the two cameras).

It is almost impossible to build a stereo rig which is perfectly aligned horizontally. Searching through oblique epipolar lines is awkward, and computationally is less efficient. What people do in practice is to convert epipolar geometry to standard geometry with rectification/warping. We warp the images as if they were acquired through a standard geometry, then we can compute and apply to both images a transformation known as rectification.

2.1.5 Depth of Field (DOF)

A scene point is on focus when all its light rays, gathered by the camera, hit the image plane at the same point. In a pinhole device this happens to all scene points because of the very small size of the hole, so that the camera features an infinite Depth of Field (DOF).

The drawback is that such a small aperture allows gathering a very limited amount of light. The image is really sharp, but has a really low light. If a point is projected onto a circle instead of a point (bigger pinhole) the image is not sharp (not on focus). If we cannot gather enough light through the aperture we have to integrate through time, by using a longer exposure time.

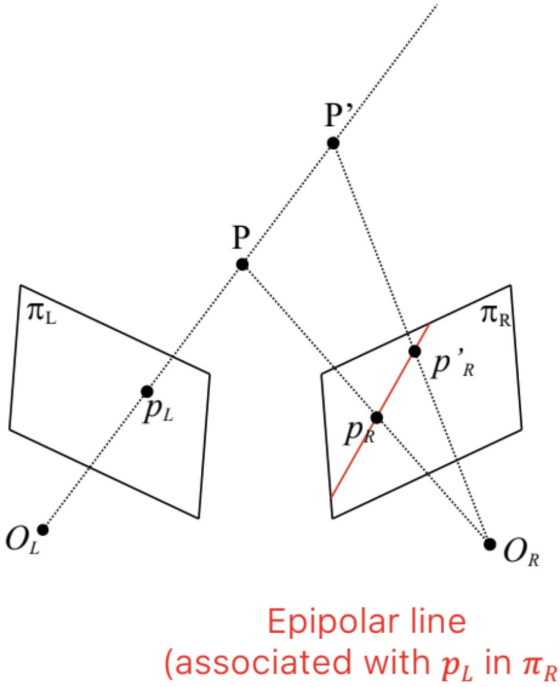


Figure 5: Epipolar line

2.1.6 Lenses

Lenses concentrate light, so we use them to gather more light from a scene point and focus it on a single image point. This enables much smaller exposure times. This way Depth Of Field is no longer infinite, and only a limited range of points can be simultaneously on focus in a given image.

We will consider the approximate model known as thin lens equation, which is useful to graphically determine the position of a focused image point:

- Rays parallel to the optical axis are deflected to pass through F.
- Rays through C are undeflected.

If the image is on focus, the image formation process obeys to the perspective projection model:

- The center of the lens is the optical center.
- The distance v acts as the effective focal length of the projection.

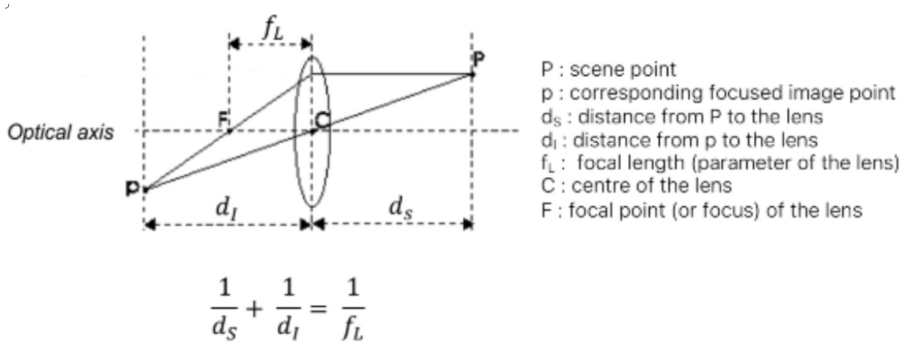


Figure 6: Scheme of a lens

Choosing the distance of the image plane determines the distance at which scene points appear on focus in the image. Scene points in front and behind the focusing plane will result out-of-focus, thereby appearing in the image as **blur circles**, rather than points.

The advantage of lenses is to have a small exposure time for capturing moving objects but we pay in terms of Depth Of Field.

As we can see in 2.1.6 having a small aperture (pinhole camera model) results in everything being on focus, but we need lots of light, or the image will be dark, since few light can enter in the sensor. We could also increase our exposure time to let more light in, but in that case we need the objects in our image to be still. Having a bigger aperture results in more light coming in in a small time fraction, but the lens distortion causes the light to be focused only at certain distances, and creates blur cirlces in other zones, so we get a lower depth

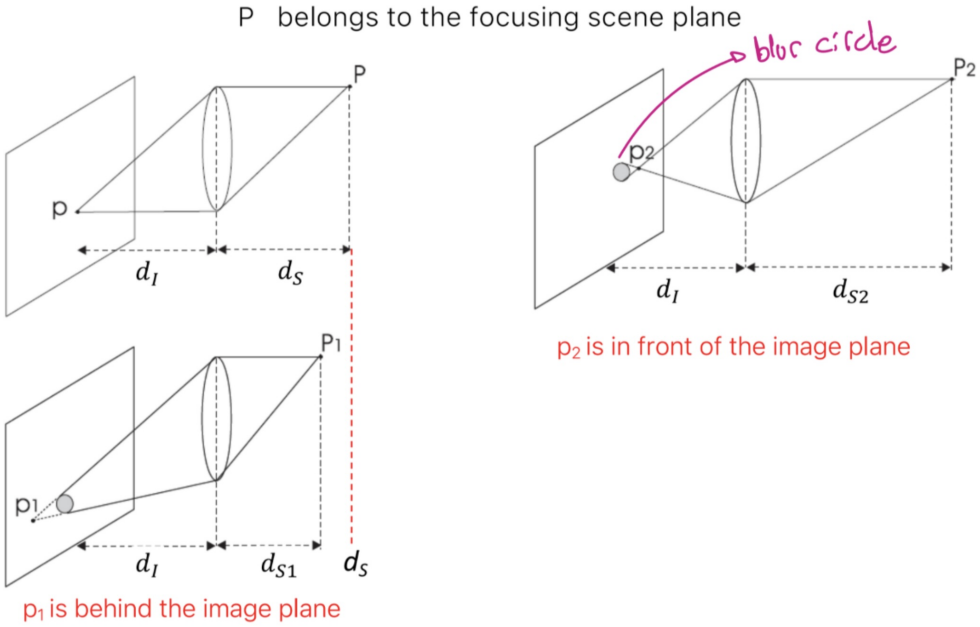


Figure 7: Lens focusing at various distances

of field, as we can see in 7.

2.1.7 Diaphragm

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 the circles are smaller than the size of the photosensing elements (a single pixel), the image will still look on-focus.

The range of distances across which the image appears on focus, due to blur circles being small enough, determines the (Depth Of Field) of the imaging apparatus. Cameras often deploy an adjustable diaphragm (iris) to control the amount of light gathered through the effective aperture of the lens.

- Reduce aperture → less light → smaller blur circle
- More aperture → more light → larger blur circle.
- Close the diaphragm → increase depth of field → not enough light → increase exposure time → moving object → motion blur.

2.1.8 Focusing mechanism (manually changing depth of field)

To focus on objects at diverse distances we need a mechanism that allows the lens (or the lens subsystem) to translate along the optical axis with respect to the fixed position of the image plane.

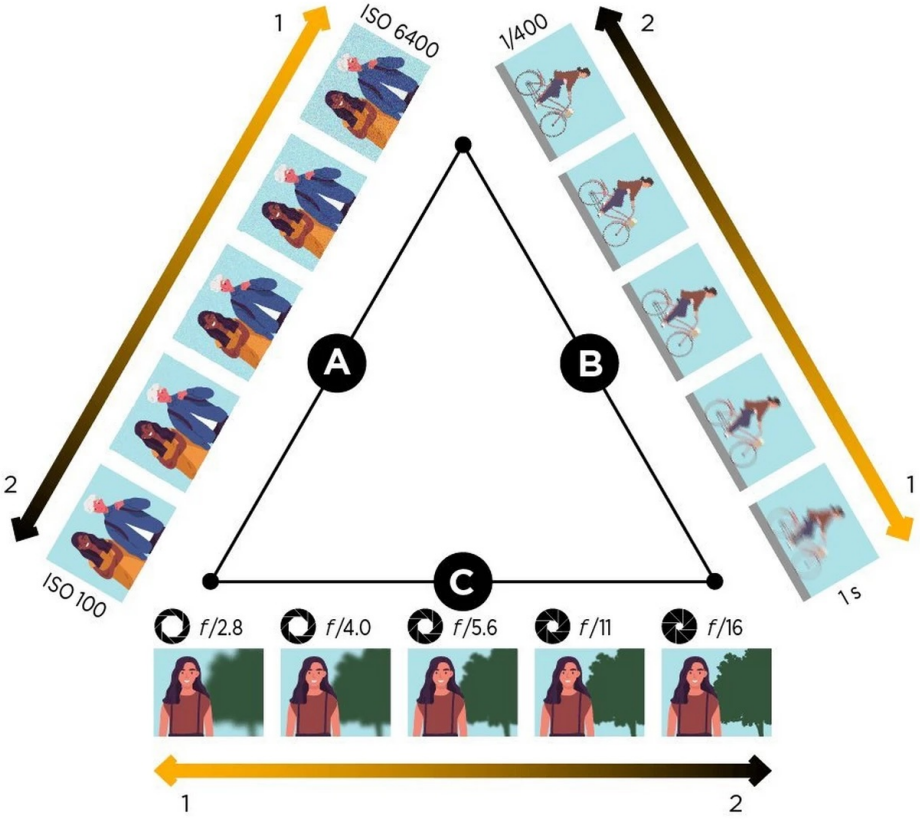
At one end position ($d_I = f_L$) the camera is focused at infinity (objects at inifity are on sharp focus). The focusing mechanism allows the lens to be translated farther away from the image plane up to a certain maximum value, which determines the minimum focusing distance.

2.1.9 Image digitization

How do we convert a continuous image to a discrete one which can be represented on a computer? The process can be divided in two steps: sampling and quanitzation.

- **Sampling**: the planar continuous image is sampled along both the horizontal and vertical directions to pick up a matrix of $N \times M$ samples (pixels).
- **Quantization**: the continuous range of values associated with pixels is quantized into $l = 2^m$ discrete levels known as gray-levels. m is the number of bits used to represent a pixel, with the memory occupancy (in bits) of a gray-scale image given by $B = N \times M \times m$. Coloured digital images are typically represented within computers using 3 bytes per pixels. Both more pixels and more bits per pixel result in a higher quality image.

The more bits we spend for its representation, the higher the quality of the digital image (it becomes a closer approximation to the ideal continuous image). This applies both to sampling and quantization.



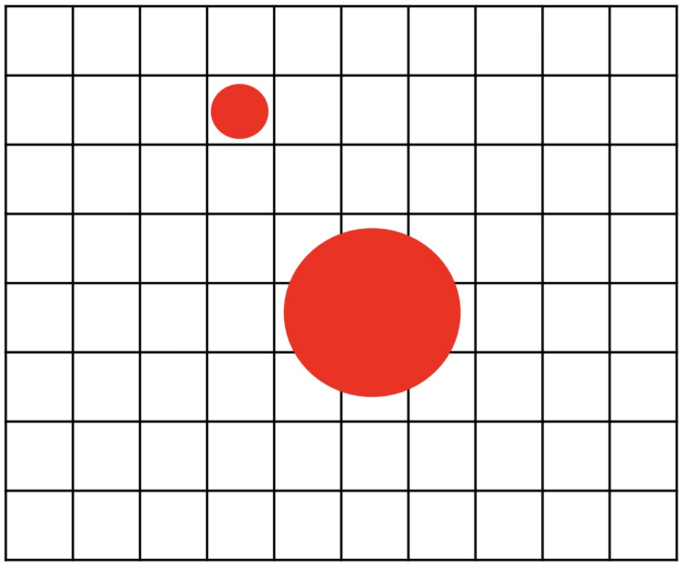


Figure 8: Blurring at pixel level

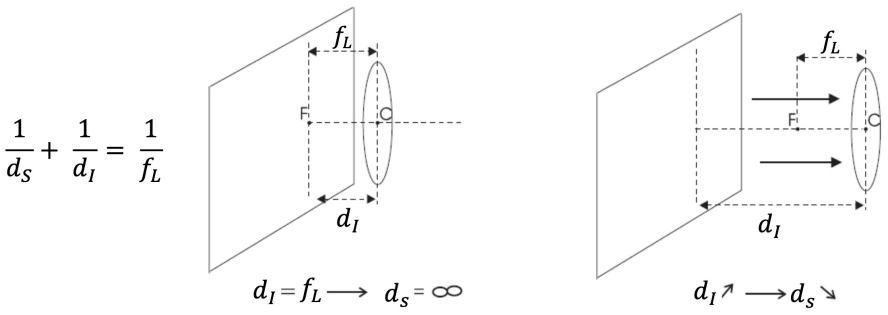


Figure 9: Focusing mechanism

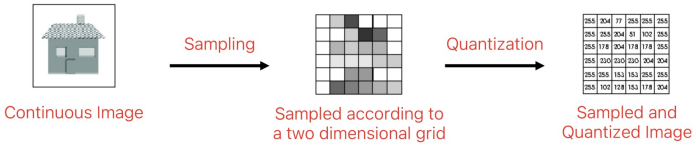


Figure 10: The two steps of the image digitization process

2.1.10 Camera sensors

The sensor is a matrix of photodetectors. During exposure time, each detector converts the incident light into a proportional electic charge. The companion circuitry reads-out the charge to generate the output signal, which can be either digital or analog. For digital cameras the sensor includes the necessary ADC circuitry.

Today, the two main sensor technologies are:

- **Charge Coupled Devices (CCD)**, where the sensor and circuit are separated.
- **Complementary Metal Oxide Semiconductor (CMOS)**, where everything is on the same circuit.

CCD/CMOS sensors can't sense colors, so we place an array of optical filters in front of the photodetectors, to render each pixel sensitive to a specific range of wavelengths.

2.1.11 SNR

The intensity measured at a pixel under perfectly static conditions varies due to the presence of random noise. The main noise sources are:

- **Photon Shot Noise**: the number of photons collected during exposure time is not constant.
- **Electronic Circuitry Noise**: generated by the electronics.
- **Quantization Noise**: related to the ADC conversion.
- **Thermal Noise (Dark Current Noise)**: random charge observed due to thermal excitement.

SNR can be expressed both in decibels and bits.

2.1.12 Dynamic Range (DR)

If the sensed amount of light is too small, the "true" signal cannot be distinguished from noise. Given $E_{\min}$: the minimum detectable irradiation, and $E_{\max}$, the saturation irradiation. The Dynamic Range (DR) of a sensor is defined as $DR = \frac{E_{\max}}{E_{\min}}$, and like the SNR, it is often specified in decibels or bits.

Like SNR, the higher the DR the better it is. A **higher DR** corresponds to the ability of the sensor to simultaneously **capture in one image both the dark and bright** structures of the scene.

2.2 Image Filtering

2.2.1 Noise and image filters

In computer vision we have to deal with noise. Noise is always different, and more noticeable in uniform regions of the image. The simplest thing to reduce noise is to output the average of the pixel color over time, to get **almost** the ideal noiseless value.

$$O(p) = \frac{1}{T}\sum_{t=1}^{T} I_k(p) = \frac{1}{T}\sum_{t=1}^{T}(\tilde{I}(p) + n_t(p))$$

This technique works well on still images.

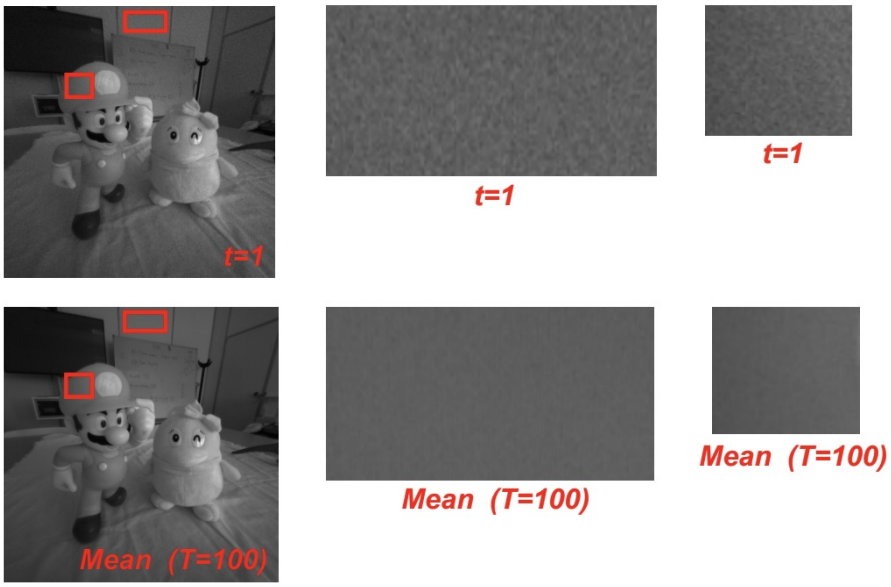


Figure 11: Simple denoising

If we are given a simple image, we may compute a mean across neighbouring pixels, like a spatial rather than temporal mean. The size of the square of the neighbouring pixels is a tradeoff.

This is a very basic denoising filter.

Image filters are image processing operators that compute the new intensity (colour) of a pixel, p, based on the intensities (colours) of those belonging to a neighbourhood (support) of p.

2.2.2 Convolution

An important sub-class of filters is given by **Linear** and **Translation-Equivariant (LTE)** operators.

The **application of filters** in image processing consists in a **2D convolution** between the input image and the impulse response function of the LTE operator.

LTE operators are used as feature extractors in Convolutional Neural Networks (CNNs).

Given an input 2D signal $i(x, y)$, a 2D operator $Ti(x, y)$ is said to be **linear** if and only if:

$$T\{\alpha i_1(x, y) + \beta i_2(x, y)\} = \alpha o_1(x, y) + \beta o_2(x, y)$$

with $o_1 = Ti_1$ and $o_2 = Ti_2$ and α, β are two constants. This means that **scaling and adding images before applying the filter gives the same result as filtering them separately** and then scaling and adding the results.

The operator is said to be **translation-equivariant** if and only if:

$$T\{i(x - x_0, y - y_0)\} = o(x - x_0, y - y_0)$$

This means that **shifting the input shifts the output; the shape stays the same, just in a different location**.

If an operator is both linear and translation-equivariant (LTE), then its output can be written as a convolution between the input signal $i(x, y)$ and the impulse response of the system (which is the output the system gives when you input a single point)

This impulse response is written as $h(x, y) = T\delta(x, y)$, and the output is $o(x, y) = i(x, y) * h(x, y)$.

Mathematically, the 2D convolution is written as:

$$i(x, y) * h(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} i(\alpha, \beta) h(x - \alpha, y - \beta)\, d\alpha d\beta$$

Convolutions have some useful properties:

- **Associative property**: $f * (g * h) = (f * g) * h$ (useful because we can decompose kernels and obtain faster operations).
- **Commutative property**: $f * g = g * f$.
- **Distributive property**: $f * (g + h) = f * g + f * h$.
- **Convolution commutes with differentiation**: $(f * g)' = f' * g = f * g'$.

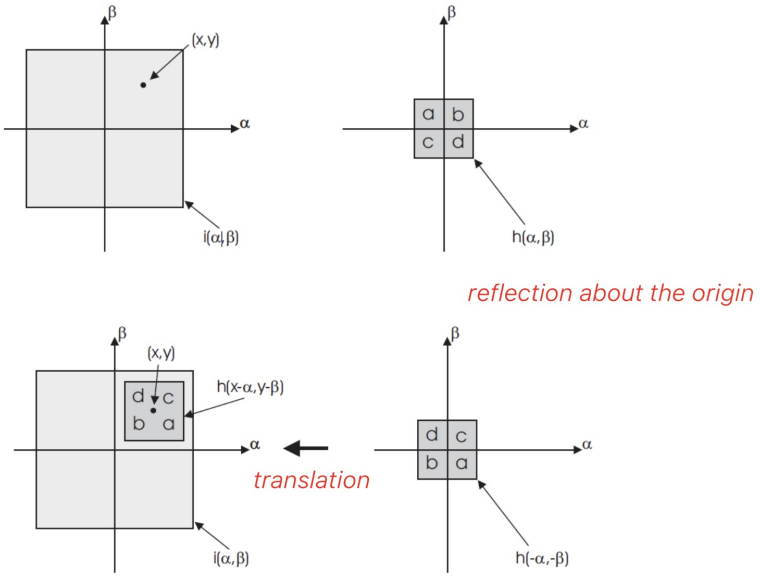


Figure 12: A graphical view of convolution

The correlation of signal $i(x, y)$ with respect to signal $h(x, y)$ is defined as:

$$i(x, y) \circ h(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} i(\alpha, \beta) h(x + \alpha, y + \beta)\, d\alpha d\beta$$

Correlation is not commutative, unlike the convolution.

2.2.3 Discrete convolution

Normal convolution is useful for signal theory, but we want to have a discrete convolution, where we use summations instead of integrals. The four convolution properties highlighted for the convolution hold for the discrete one too.

2.2.4 Practical implementation

The "convolution" operation in many Convolutional Neural Network (CNN) implementations is technically cross-correlation. This is equivalent to classical convolution if the kernel is flipped; the weights learned by the CNN constitute this kernel.

In image processing, both the input image and the kernel are represented as matrices of finite dimensions. The image is typically much larger than the kernel. To generate the output (often called a feature map), the kernel is slid across the input image. At each position, the dot product between the kernel and the overlapping input region is computed.

A challenge arises at image borders, as the kernel may extend beyond the input matrix boundaries. Two primary strategies address this:

- `VALID` convolution (cropping): The convolution is computed only for positions where the kernel fully overlaps the input. This approach is common in some image processing contexts.
- `SAME` convolution (padding): The input image is padded before convolution, often with the goal of preserving the spatial dimensions of the input in the output. This is generally preferred in CNNs. Common padding methods include:
  - Zero-padding: Adding zeros around the border.
  - Replication padding: Replicating border pixel values.
  - Reflection padding: Reflecting pixel values across the border (e.g., reflecting k pixels, where k often corresponds to half the kernel width/height).

Without padding (i.e., using `VALID` convolution), the operation reduces the spatial dimensions of the output.

2.2.5 Mean filter

Mean filtering is the **simplest and fastest way to denoise an image**. It consists in replacing each pixel intensity by the **average intensity over a chosen neighbourhood**. According to signal processing theory, the Mean Filter carries out a **low-pass filtering operation**, which in image processing is **also referred to as image smoothing**. Smoothing is often aimed at image denoising, but sometimes is used to cancel out

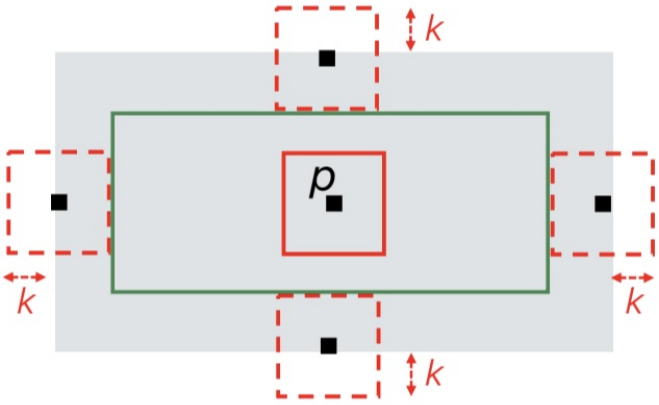


Figure 13: Convolutions crop the image

small-size unwanted details that might hinder the image analysis task. Linear filtering **reduces noise but blurs the image**, so we lose sharpness.

2.2.6 Gaussian filter

The Gaussian filter is an LTE operator whose impulse response is a 2D Gaussian function.

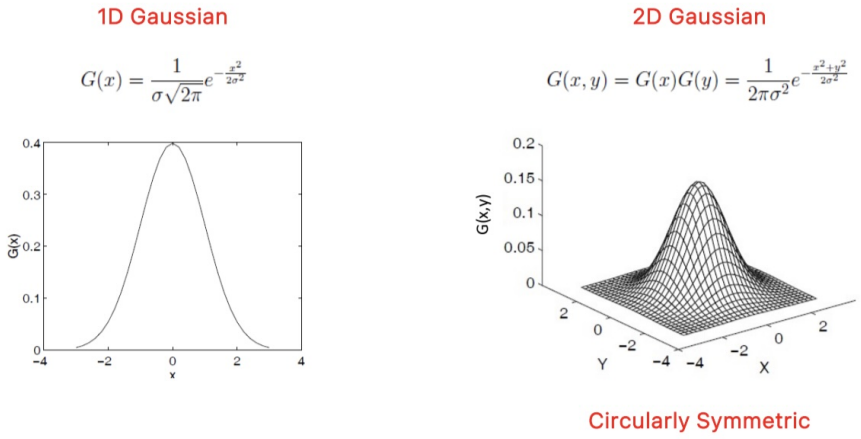


Figure 14: 1D and 2D Gaussian plot

The larger the size of the gaussian kernel, the more accurate the approximation will be, but the computational cost grows with the filter size. We should then use larger sizes for filters with higher σ, smaller sizes whenever σ is smaller. A typical rule is to choose the size of the filter by capturing the interval $[-3\sigma, +3\sigma]$ since it captures 99% of the energy of the Gaussian impulse.

To speedup the filtering we can apply the 2D gaussian by doing 2 1D gaussian filterings.

We also observe that the higher the σ, the higher is the smoothing caused by the filter. We can use this filter to remove details from the image.

2.2.7 Median filter

There is noise that Gaussian filters can't handle well, that is the **salt and pepper noise**. It's usually caused by image corruption (or broken pixels in the sensor). On this noise linear filtering is ineffective, and just blurs the image.

We can use a **non-linear filter**, where each pixel intensity is replaced by the **median over a given neighbourhood**. Median filtering counteracts impulse noise effectively, since **outliers tend to fall at either the top or the bottom end of the sorted intensities**. Median filtering tends to keep sharper edges than linear filters such as the Mean or the Gaussian.

The median filter can effectively denoise the image without introducing significant blur, yet, Gaussian-like noise, such as sensor noise, cannot be dealt with by the median, as this would require computing new noiseless intensities.

Figure 15: Example of the power of the median filter

2.2.8 Bilateral filter

The **bilateral filter** is a non-linear filter that **preserves edges** while reducing noise, such as Gaussian-like noise. This characteristic, also known as edge-preserving smoothing, allows it to denoise images without significantly blurring sharp features.

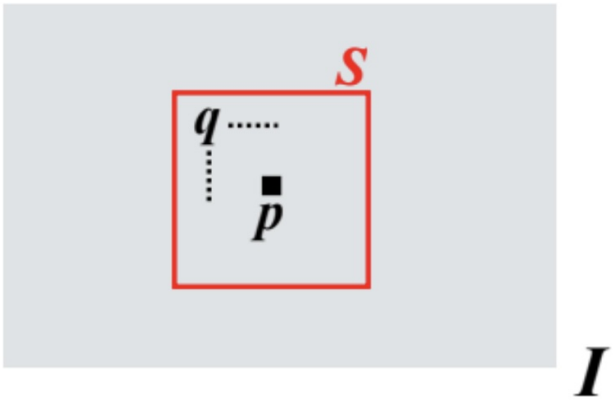


Figure 16: Application of the bilateral filter

The filtered output $O(p)$ for a pixel p is computed as a weighted average of neighboring pixel intensities:

$$O(p) = \sum_{q \in S} H(p,q) \cdot I_q$$

where S is the set of pixels in the neighborhood of p, I_q is the intensity of pixel q, and the weight $H(p,q)$ is given by:

$$H(p,q) = \frac{1}{W(p)} G_{\sigma_s}(d_s(p,q))\, G_{\sigma_r}(d_r(I_p, I_q))$$

Here, G_{σ_s} and G_{σ_r} are Gaussian functions. The components are:

- **Spatial distance**: $d_s(p,q) = ||p-q||_2 = \sqrt{(u_p - u_q)^2 + (v_p - v_q)^2}$, where (u_p, v_p) are the coordinates of pixel p. This term penalizes distance in space.
- **Range (intensity) distance**: $d_r(I_p, I_q) = |I_p - I_q|$. This term penalizes differences in pixel intensities.
- **Normalization factor**: $W(p) = \sum_{q \in S} G_{\sigma_s}(d_s(p,q))\, G_{\sigma_r}(d_r(I_p, I_q))$, ensuring that the weights $H(p,q)$ for a given p sum to 1.

Within the support neighborhood S, a neighboring pixel q is assigned a higher weight $H(p,q)$ if it is both spatially close to p (small $d_s(p,q)$) and has an intensity I_q similar to I_p (small $d_r(I_p, I_q)$).

For a pixel p near an edge, neighboring pixels q on the other side of the edge will have markedly different intensities (I_q vs I_p). This results in a large range distance $d_r(I_p, I_q)$, significantly reducing the value of the range Gaussian G_{σ_r} and thus the overall weight $H(p,q)$. Consequently, these differing pixels contribute minimally to the output $O(p)$, thereby preserving the edge.

A key characteristic is that **the filter weights $H(p,q)$ must be recomputed for each pixel p**, as they depend on the local image content (specifically, I_p).

The bilateral filter is almost like a Gaussian filter, except that the Gaussian is modulated by a function that computes the similarity between the central pixel (where the filter is applied) and a pixel in its neighborhood (that is used in blurring).

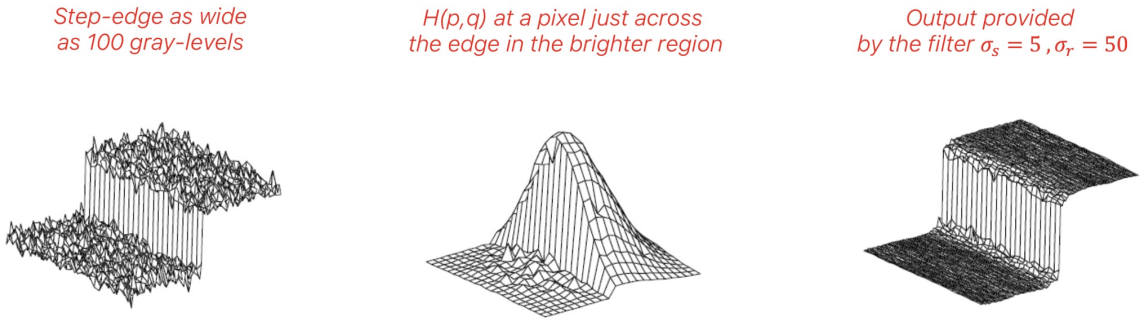


Figure 17: Bilateral filter applied on an edge

- If the two pixel values are very close, it multiplies the Gaussian coefficient by something close to 1, and hence it is equivalent to Gaussian filtering.
- If the pixel values are very different , it will multiply the Gaussian coefficient by a number close to 0, thus turning off the Gaussian filtering for this pixel.

Intuitively, this behaviour yields the following result: Gaussian filtering in uniform areas of the image, no filtering across object borders. The bilateral filter will produce a more pleasant results, because it will avoid the introduction of blur between objects while still removing noise in uniform areas

2.2.9 Non-local means filter

It's another non-linear edge preserving smoothing filter. The key idea is that the **similarity among patches spread over the image** can be deployed to achieve denoising. It's even more expensive than the bilateral filter, since it has to look at more of the image.

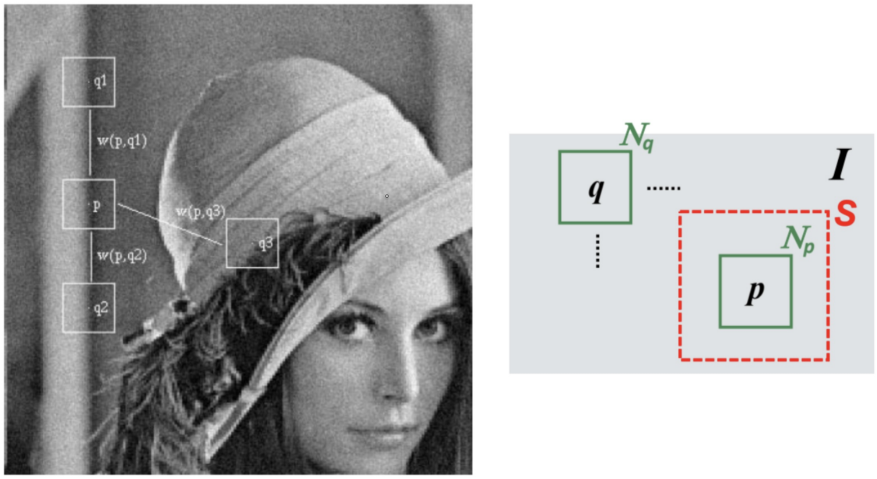


Figure 18: Non-local means filter

Figure 19: Difference between Gaussian filter and non-local means filter

2.3 Edge detection

Edge points are local image features representing significant discontinuities in intensity. They often correspond to object boundaries or changes in surface properties, thus capturing important semantic information. Edges typically occur in transition zones between regions of differing intensities.

1D step-edge In a 1D signal, an edge (or step-edge) manifests as a sharp change in intensity. A basic detection method involves thresholding the absolute value of the signal's first derivative.

2D step-edge A 2D step-edge is characterized by its strength (magnitude) and its orientation. The gradient vector,

$$\nabla I(x,y) = \frac{\partial I(x,y)}{\partial x}\mathbf{i} + \frac{\partial I(x,y)}{\partial y}\mathbf{j}$$

points in the **direction of maximum intensity variation**. In discrete images, partial derivatives are approximated using finite differences (yielding I_x and I_y). The gradient magnitude, indicating edge strength, can be estimated using various approximations. For example, $|\nabla I| \approx \max(|I_x|, |I_y|)$ is computationally efficient but not perfectly rotationally invariant.

Edges and noise Real-world images contain noise, which can cause edges to appear irregular. Computing derivatives on noisy signals is an ill-posed problem because **derivatives tend to amplify high-frequency noise**. To improve robustness, images are often smoothed before differentiation. However, this smoothing also blurs true edges, which can reduce their apparent strength and make precise localization more challenging, as distinct pixel intensities in the original image are averaged.

Prewitt and Sobel The Prewitt and Sobel operators address edge detection in noisy images by convolving the image with small (3×3) kernels that combine smoothing and differentiation. Each operator uses a pair of kernels: one to estimate the gradient component G_x (approximating I_x, responsive to vertical edges) and another for G_y (approximating I_y, responsive to horizontal edges). Each kernel effectively smooths the image in one direction and computes the derivative in the perpendicular direction. After convolving the image with both kernels to obtain G_x and G_y, the gradient magnitude and direction are computed:

$$\text{Magnitude} = \sqrt{G_x^2 + G_y^2}$$

$$\text{Direction} = \arctan^2(\frac{G_y}{G_x})$$

The Sobel operator gives more weight to the central pixels in its smoothing component, generally offering better noise suppression than the Prewitt operator, which uses uniform weights. This can come at the cost of slightly more edge blurring for Sobel.

2.3.1 Non-Maxima Suppression (NMS)

Detecting edges by simply thresholding the gradient magnitude often produces thick or inaccurate edges, partly because **it is difficult to choose a single, globally optimal threshold**. Non-Maxima Suppression (NMS) is a technique to thin wide edges into sharper lines by selecting only pixels that are local maxima of the gradient magnitude. For 2D images, NMS involves:

- Identifying local maxima of the gradient magnitude.
- This search for maxima is performed along the gradient direction at each pixel (which is orthogonal to the potential edge direction).

Since the true gradient direction (estimated locally) may not align perfectly with the discrete pixel grid, gradient magnitudes at sub-pixel locations along this direction are typically estimated using interpolation (e.g., linear interpolation) from neighboring grid pixels. A final thresholding step is often applied to the NMS output. This helps to prune weak edges that might arise from noise or less significant image details, even after suppression.

2.3.2 Canny's edge detector

Canny formulated quantitative criteria for evaluating edge detector performance and sought an optimal filter based on these:

- **Good detection**: High probability of detecting true edges and low probability of falsely detecting non-edges (robustness to noise).
- **Good localization**: Detected edges should be as close as possible to the true edges.
- **Single response**: The detector should return only one point for each true edge (minimizing multiple responses to a single edge).

The Canny edge detection algorithm typically involves the following steps:

1. **Gaussian smoothing**: Reduce noise using a Gaussian filter. The 2D Gaussian convolution $G(x, y) = G(x)G(y)$ is separable, allowing for efficient computation as two 1D convolutions.
2. **Gradient computation**: Calculate gradient magnitude and direction (e.g., using Sobel operators).
3. **Non-Maxima Suppression (NMS)**: Thin wide ridges of gradient magnitude into sharp lines by preserving only pixels that are local maxima along the gradient direction.
4. **Hysteresis thresholding**: This step addresses the issue of "edge streaking" (where an edge contour is broken if its gradient magnitude locally dips below a single threshold). It uses two thresholds: a high threshold T_H and a low threshold T_L.
   - Pixels with gradient magnitude above T_H are immediately classified as "strong" edge pixels.
   - Pixels with gradient magnitude between T_L and T_H are classified as "weak" edge pixels.
   - Pixels with gradient magnitude below T_L are suppressed.

   Strong edge pixels are kept. Weak edge pixels are kept only if they are connected to a strong edge pixel (directly or via other connected weak edge pixels). This connectivity check is often implemented by tracking along contours from strong edge pixels.

Zero-crossing of the second derivative An alternative to finding peaks in the first derivative (gradient magnitude) is to locate edges at the zero-crossings of the second derivative of the signal. 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.

2.3.3 Second derivative along the gradient & Laplacian

The second derivative along the gradient's direction, $n^T Hn$, can be used to find edge locations.

- $n = \frac{\nabla I(x,y)}{||\nabla I(x,y)||}$ is the unit vector in the gradient's direction.
- $H = \begin{bmatrix} \frac{\partial^2 I(x,y)}{\partial x^2} & \frac{\partial^2 I(x,y)}{\partial x \partial y} \\ \frac{\partial^2 I(x,y)}{\partial y \partial x} & \frac{\partial^2 I(x,y)}{\partial y^2} \end{bmatrix}$ is the Hessian matrix.

Computing this directional second derivative explicitly is computationally expensive.

Discrete Laplacian The Laplacian operator, $\nabla^2 I = I_{xx} + I_{yy}$, is a scalar quantity that is rotationally invariant. Its zero-crossings often occur close to the zero-crossings of the second derivative along the gradient direction. Approximating derivatives with finite differences allows for a discrete Laplacian filter, which is much faster to compute than the directional second derivative.

2.3.4 Laplacian of Gaussian (LoG)

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

1. **Gaussian smoothing**: Convolve the image $I(x, y)$ with a Gaussian kernel $G(x, y)$ to get $\tilde{I}(x, y) = I(x, y) * G(x, y)$.
2. **Laplacian operation**: Compute the Laplacian of the smoothed image, $\nabla^2 \tilde{I}(x, y)$.
3. **Zero-crossing detection**: Identify pixels where the $\nabla^2 \tilde{I}(x, y)$ value changes sign. These zero-crossings indicate potential edges.

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 ($\nabla^2 G$), 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.

A larger σ for the Gaussian component results in more smoothing, detecting larger scale edges while suppressing finer details and noise. This is beneficial for noisy images.

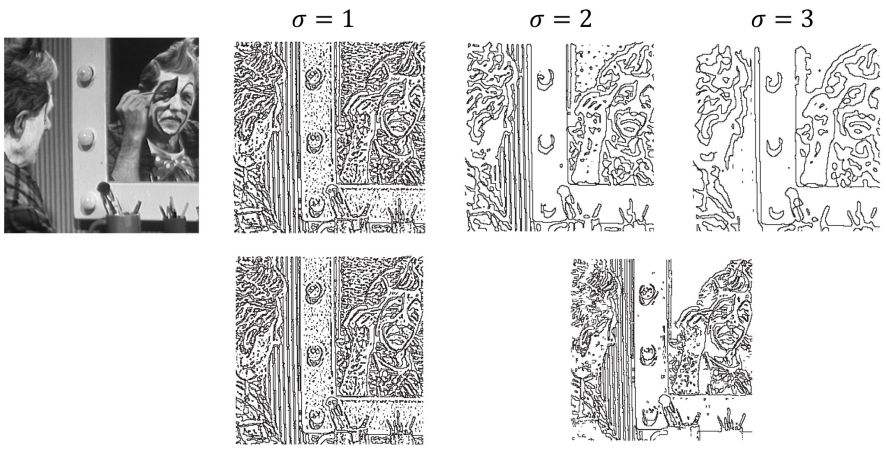


Figure 20: Examples of LoG application

2.4 Feature detection and matching

Feature detection and matching are fundamental for various computer vision tasks, such as image stitching, 3D reconstruction, object recognition, and tracking. The core idea is to identify distinctive points (features) in one image and find their counterparts in other images of the same scene or object. These corresponding image points are projections of the same 3D point viewed from different perspectives. Establishing these correspondences can be challenging because the appearance of points can vary significantly across different views due to changes in viewpoint, illumination, scale, and other factors.

2.4.1 Local invariant features paradigm

The task of establishing correspondences using local features is typically broken down into three steps:

- **Detection**: Identify salient interest points (keypoints) in the image.
- **Description**: Compute a feature descriptor for the local image region around each keypoint. This descriptor numerically captures the appearance of the neighborhood.
- **Matching**: Compare descriptors from different images to find pairs that are most similar, thereby establishing correspondences.

Ideally, descriptors should be **invariant** (or at least robust) to transformations such as changes in scale, rotation, illumination, and viewpoint.

2.4.2 Properties of good detectors/descriptors

- **Detector criteria**:
  - Repeatability: The detector should consistently find the same keypoints in different images of the same scene, despite transformations.
  - Saliency (or distinctiveness): Keypoints should be located in textured regions with informative patterns, making them distinguishable.
- **Descriptor criteria**:
  - Distinctiveness: The descriptor for one keypoint should be significantly different from descriptors of other, unrelated keypoints.
  - Robustness (or invariance): The descriptor for a specific keypoint should remain similar even when the keypoint is viewed under different conditions (e.g., illumination changes, small viewpoint variations). There's often a trade-off between distinctiveness and robustness.
  - Compactness: The descriptor should be as concise as possible to reduce storage and computational cost for matching.
  - Efficiency: Computation of descriptors should be fast.

Overall efficiency is crucial for both detection and description, especially for detectors that operate on the entire image. Descriptors are computed only at detected keypoints.

Edge pixels, while indicative of intensity changes, often suffer from local ambiguity: a small patch centered on an edge looks very similar to patches at nearby points along the same edge (the aperture problem). This makes it difficult to uniquely localize or match edge pixels based solely on their immediate neighborhood.

2.4.3 Moravec Interest Point Detector

The Moravec detector identifies interest points by evaluating how dissimilar a local image patch is from slightly shifted versions of itself. The "cornerness" $C(p)$ at a pixel p is defined as the minimum sum of squared differences

(SSD) between an image patch $N(p)$ centered at p and patches $N(q)$ centered at its 8 immediate neighbors $q \in n_8(p)$:

$$C(p) = \min_{q \in n_8(p)} \sum_{(u,v) \in \text{Patch}} (N_p(u,v) - N_q(u,v))^2$$

where $N_p(u,v)$ is the intensity of a pixel at relative coordinates (u,v) within the patch centered at p. After computing $C(p)$ for all pixels, potential interest points are those where $C(p)$ exceeds a threshold, followed by Non-Maxima Suppression (NMS) to select only local peaks.

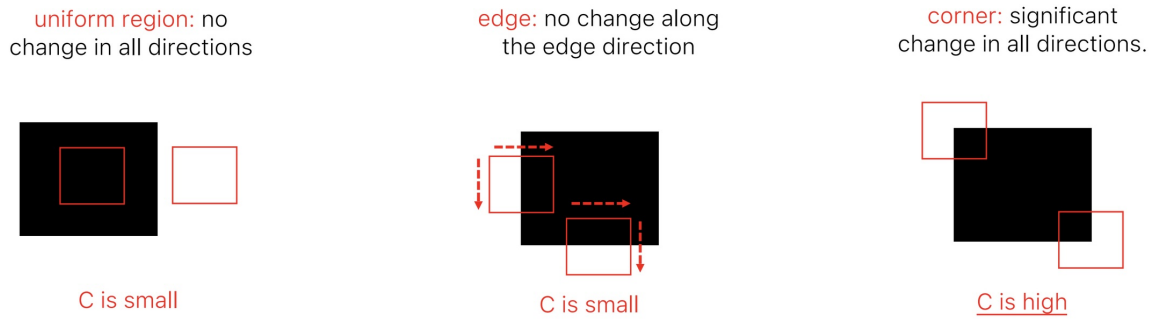


Figure 21: Illustration of patch shifts for cornerness evaluation. A corner (right) shows significant intensity changes in all shift directions. An edge (middle) shows little change along the edge direction. A flat region (left) shows little change in any direction.

2.4.4 Harris Corner Detector

Harris and Stephens refined Moravec's idea by considering infinitesimal shifts $(\Delta x, \Delta y)$ and analyzing the local image structure using a continuous formulation. The sum of squared differences $E(\Delta x, \Delta y)$ for a shift $(\Delta x, \Delta y)$ over a window $w(u,v)$ centered at (x_0, y_0) is:

$$E(\Delta x, \Delta y) = \sum_{u,v} w(u,v)[I(x_0 + u + \Delta x, y_0 + v + \Delta y) - I(x_0 + u, y_0 + v)]^2$$

Using a first-order Taylor expansion for the intensity difference:

$$I(x_0 + u + \Delta x, y_0 + v + \Delta y) - I(x_0 + u, y_0 + v) \approx I_x(x_0 + u, y_0 + v)\Delta x + I_y(x_0 + u, y_0 + v)\Delta y$$

where I_x and I_y are partial derivatives of the image intensity I. Substituting this into $E(\Delta x, \Delta y)$ yields a quadratic approximation:

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

The matrix M, often called the structure tensor or second-moment matrix, is computed for the window around (x_0, y_0):

$$M = \sum_{u,v} w(u,v) \begin{bmatrix} I_x(x_0 + u, y_0 + v)^2 & I_x(x_0 + u, y_0 + v) I_y(x_0 + u, y_0 + v) \\ I_x(x_0 + u, y_0 + v) I_y(x_0 + u, y_0 + v) & I_y(x_0 + u, y_0 + v)^2 \end{bmatrix}$$

The window function $w(u,v)$ is typically a Gaussian or a rectangular window, weighting pixels within the patch.

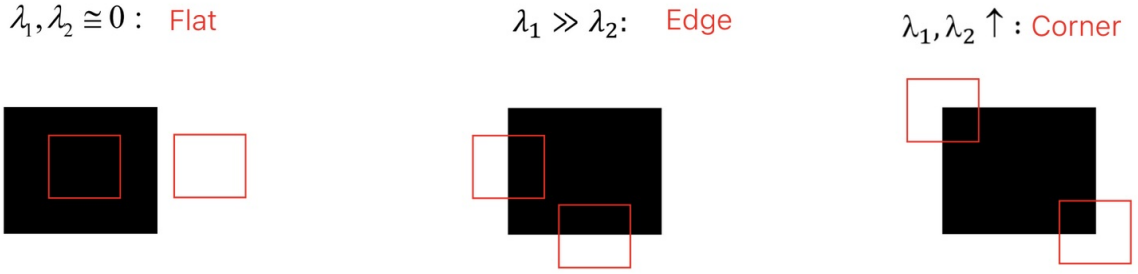


Figure 22: Interpretation of eigenvalues (λ_1, λ_2) of matrix M for Harris corner detection. Corners have two large eigenvalues. Edges have one large and one small eigenvalue. Flat regions have two small eigenvalues.

The matrix M is real and symmetric, so it can be diagonalized by an orthogonal matrix R:

$$M = R \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R^T$$

where λ_1, λ_2 are the eigenvalues of M, and the columns of R are the corresponding orthonormal eigenvectors. These eigenvalues characterize the local image structure:

- Flat region: $\lambda_1 \approx 0, \lambda_2 \approx 0$. $E(\Delta x, \Delta y)$ is small for all shifts.
- Edge: $\lambda_1 \gg 0, \lambda_2 \approx 0$ (or vice versa). $E(\Delta x, \Delta y)$ is large for shifts perpendicular to the edge, small for shifts along the edge.
- Corner: $\lambda_1 \gg 0, \lambda_2 \gg 0$. $E(\Delta x, \Delta y)$ is large for shifts in all directions.

Explicitly computing eigenvalues at each pixel is computationally intensive. Harris proposed a corner response function C_H:

$$C_H = \det(M) - k(\text{trace}(M))^2 = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

where k is an empirical constant (e.g., $0.04 - 0.06$). C_H is large and positive for corners, negative for edges, and small for flat regions.

The **Harris corner detection algorithm** proceeds as follows:

1. Compute image derivatives I_x, I_y.
2. For each pixel, compute the elements of M by summing I_x^2, I_y^2, I_xI_y over a local window (weighted by $w(u, v)$).
3. Compute the Harris response C_H at each pixel.
4. Threshold C_H to find pixels with strong corner responses.
5. Apply Non-Maxima Suppression (NMS) to select only local maxima of C_H as corner points.

Invariance Properties: The Harris detector's response C_H is invariant to image rotation (since eigenvalues are rotationally invariant) and to additive changes in image intensity (as derivatives are unaffected). However, it is not invariant to image scale changes (a corner might become an edge or a flat region if scaled significantly). It is also not invariant to multiplicative intensity changes (e.g., contrast changes), as these scale the gradient magnitudes and thus the eigenvalues, altering C_H unless thresholds are adapted.

2.4.5 Scale-Space representation

Achieving scale invariance is a primary challenge addressed by modern local invariant feature detectors. The core idea is to analyze the image at multiple scales. This is accomplished by creating a scale-space representation, typically by convolving the original image with Gaussian kernels of increasing width (σ).

A **scale-space** is a one-parameter family of images, $L(x, y, \sigma)$, generated from an original image $I(x, y)$ by progressive Gaussian smoothing:

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

where $G(x, y, \sigma)$ is a 2D Gaussian kernel with scale parameter σ. As σ increases, finer details are successively suppressed. Gaussian smoothing is chosen because it does not introduce new, spurious structures (e.g., new local extrema) as the scale increases.

Figure 23: A scale-space pyramid: the image is progressively blurred (increasing σ) and often downsampled to create different octaves. Within an octave, σ increases.

Feature Detection and Characteristic Scale Selection 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 (regions that are brighter or darker than their surroundings), 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:

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

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 $F_{norm}(x, y, \sigma)$ in both space (x, y) and scale (σ) indicate the centers and characteristic scales of blob-like features.

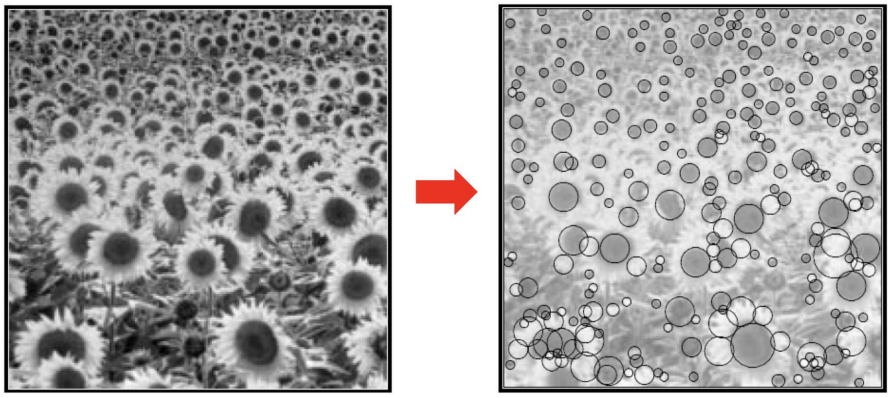


Figure 24: Blobs detected by a LoG-like filter at their characteristic scales.

Difference of Gaussian (DoG) Computing the LoG and its extrema across a finely sampled scale-space 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:

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

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.

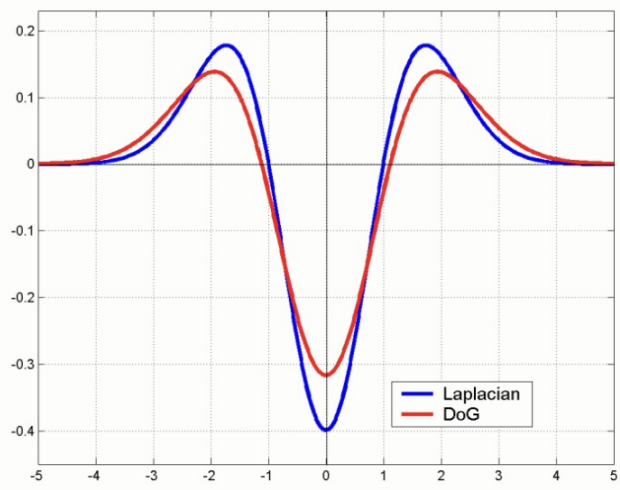


Figure 25: The Difference of Gaussians (DoG) kernel (bottom) as an approximation of the Laplacian of Gaussian (LoG) kernel (top, scaled). The DoG is computationally cheaper.

In practice, 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 subtracting adjacent Gaussian-smoothed images.

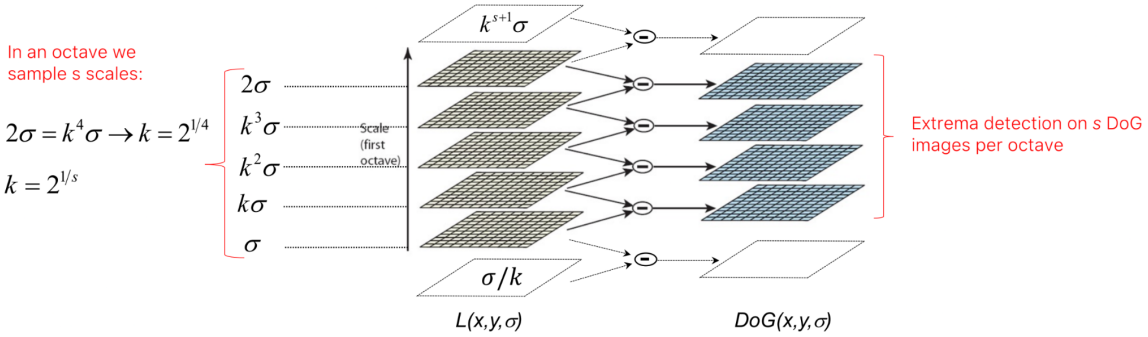


Figure 26: Generation of Difference of Gaussian (DoG) images. An image is repeatedly smoothed (intra-octave scales), and differences between adjacent smoothed images form the DoG layers. The image may also be downsampled to start a new octave.

Extrema Detection and Refinement A point (x, y, σ) is detected as a keypoint candidate if its DoG value is a local extremum (maximum or minimum) compared to its 26 neighbors in the $3 \times 3 \times 3$ region of DoG images (8 neighbors at the same scale, and 9 neighbors each at the scales above and below). Lowe's original SIFT paper suggests specific parameters, such as an initial $\sigma_0 = 1.6$, $s = 3$ images to be detected per octave (requiring $s + 3$ smoothed images), and often starting by up-sampling the input image by a factor of 2.

Candidate keypoints are then refined:

- **Accurate localization**: Sub-pixel and sub-scale localization is performed by fitting a 3D quadratic function to the DoG values around the candidate.
- **Thresholding low-contrast responses**: Keypoints with low DoG magnitudes (low contrast) are discarded as they are sensitive to noise.
- **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).

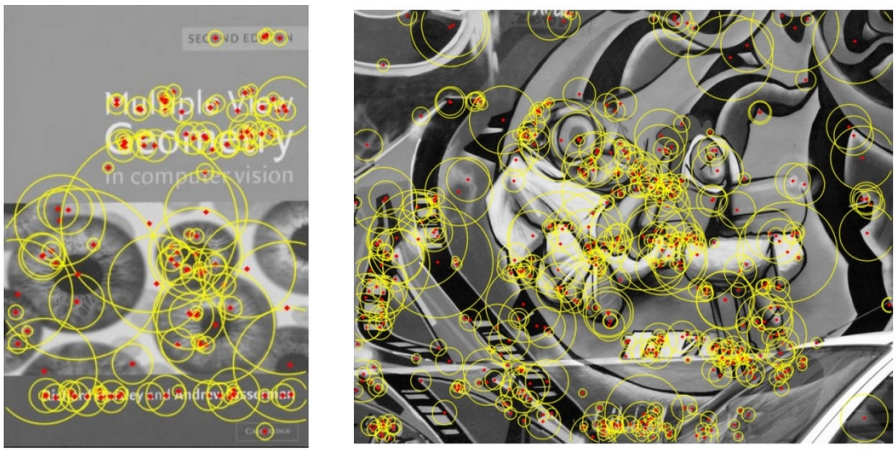


Figure 27: Keypoints detected using DoG. The size of each circle is proportional to the characteristic scale (σ) of the detected feature.

2.4.6 Scale and Rotation Invariant Description

Once stable keypoints are detected along with their characteristic scales, descriptors are computed to capture the local image information. These descriptors must also be invariant to scale and rotation.

- **Scale invariance** is achieved by computing the descriptor on the image patch corresponding to the keypoint's characteristic scale (i.e., from the Gaussian-smoothed image $L(x, y, \sigma_i)$ where σ_i is the scale of detection). The size of the descriptor's support region is scaled proportionally to σ_i.
- **Rotation invariance** is achieved by assigning a *canonical orientation* to each keypoint. The descriptor is then computed relative to this orientation.

Lowe proposed computing the canonical orientation(s) as follows: For each keypoint, consider its corresponding Gaussian-smoothed image $L(x, y, \sigma_i)$.

1. Compute gradient magnitude $m(x, y)$ and orientation $\theta(x, y)$ for pixels in a neighborhood around the keypoint in $L(x, y, \sigma_i)$:

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

2. Create an orientation histogram from these gradients. The histogram typically has 36 bins (covering 360° in 10° increments). Each sample added to the histogram is weighted by its gradient magnitude and by a Gaussian window centered at the keypoint (with σ proportional to the keypoint's scale).

3. The highest peak in the orientation histogram defines the dominant orientation for the keypoint. Any other local peaks within 80% of the highest peak also define additional orientations. This means a single keypoint location/scale can result in multiple feature descriptors if it has multiple prominent orientations (occurring for about 15% of keypoints, often at symmetric locations).

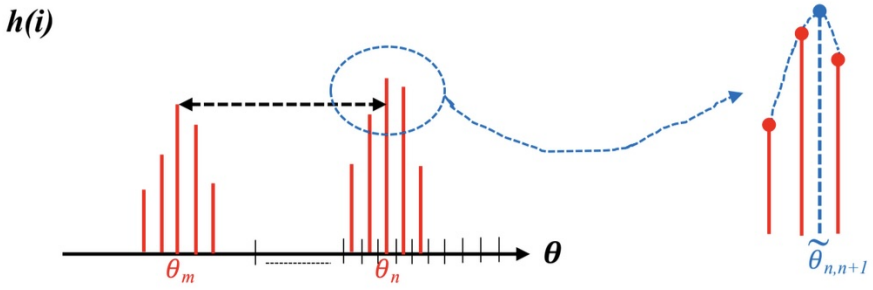


Figure 28: An example of a keypoint with ambiguous orientation, potentially leading to two dominant directions in the orientation histogram.

2.4.7 The SIFT Descriptor

The SIFT (Scale Invariant Feature Transform) descriptor is computed for each keypoint (and each of its canonical orientations) as follows:

1. A 16×16 pixel neighborhood around the keypoint is selected from the Gaussian-smoothed image $L(x, y, \sigma_i)$ corresponding to the keypoint's scale. This patch is rotated to align with the keypoint's canonical orientation.

2. This 16×16 patch is divided into a 4×4 grid of 4×4 pixel subregions.

3. For each 4×4 subregion, an 8-bin gradient orientation histogram is computed. Gradients are relative to the keypoint's canonical orientation.

4. Each pixel in a subregion contributes to its orientation histogram, weighted by its gradient magnitude and a Gaussian window (with σ equal to half the width of the 16×16 grid) centered on the keypoint. This down-weights gradients far from the keypoint center.

The resulting descriptor is a vector of $4 \times 4 \times 8 = 128$ values. This vector is then normalized to unit length to achieve robustness to affine illumination changes (brightness and contrast).

Matching Process To find corresponding keypoints between two images (a query image Q and a reference image R), their SIFT descriptors are compared. This is a Nearest Neighbor (NN) search problem in the 128-dimensional descriptor space. For each SIFT feature f_Q in the query image, its nearest neighbor f_{R1} (most similar descriptor) in the reference image is found using Euclidean distance.

However, the nearest neighbor might not be a correct match (e.g., if f_Q has no true correspondent in R). To improve reliability, Lowe proposed a ratio test: Compare the distance to the nearest neighbor (d_1) with the distance to the second nearest neighbor (d_2). If $d_1/d_2 < T_{ratio}$ (e.g., $T_{ratio} = 0.8$), the match is accepted. Otherwise, it is rejected as ambiguous. This threshold effectively rejects many false matches while retaining most correct ones.

Exhaustive NN search is computationally expensive ($O(NMD)$ where N, M are numbers of features and D is descriptor dimensionality). Approximate nearest neighbor algorithms, such as those based on k-d trees (e.g., Best Bin First search), are used to speed up the matching process significantly for large databases of features.

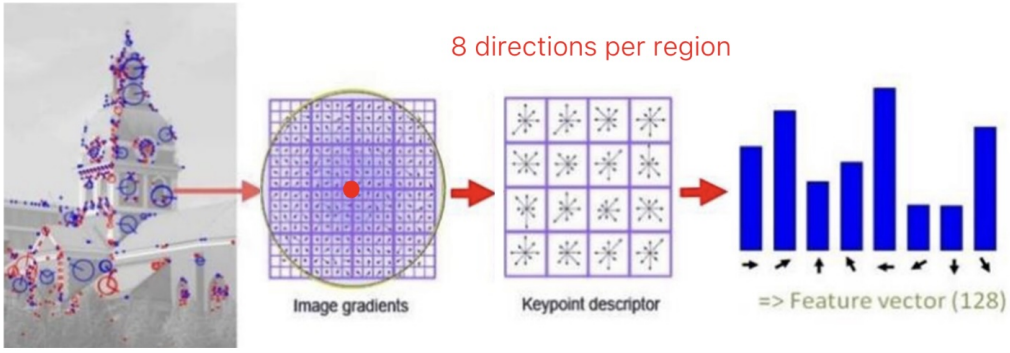


Figure 29: SIFT descriptor computation: A 16×16 window (rotated to canonical orientation) is divided into 4×4 subregions. Each subregion contributes an 8-bin orientation histogram. Total descriptor length is $4 \times 4 \times 8 = 128$.

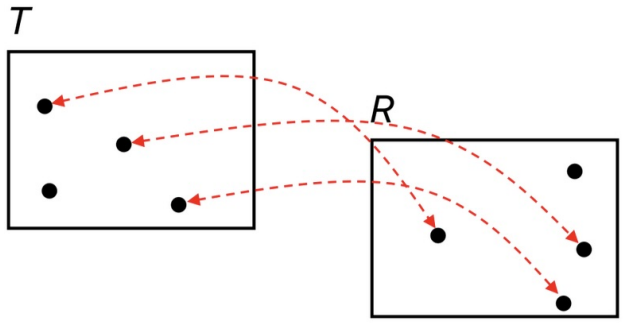


Figure 30: Matching SIFT features between two images. Lines connect corresponding keypoints.

2.5 Camera Calibration

Camera calibration is the process of determining the parameters of a camera model that describes how a 3D scene is projected onto a 2D image. This is crucial for extracting quantitative measurements from images, such as object sizes or distances. We primarily focus on the **perspective projection** model.

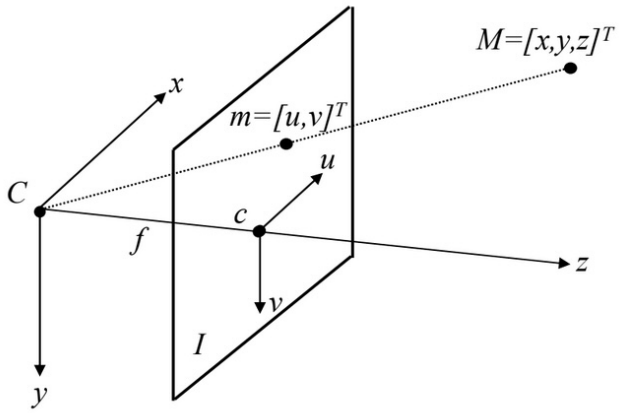


Figure 31: Perspective Projection Model (PPM). A 3D point M in the Camera Reference Frame (CRF) projects to a 2D point m on the image plane I.

2.5.1 From Physical Space to Projective Space

The world we perceive is often modeled as a 3D **Euclidean Space** ($\mathbb{R}^3$). Points are represented by 3D vectors $[x, y, z]^T$ in a chosen reference frame. However, Euclidean geometry struggles with concepts like points at infinity and the projective nature of image formation (e.g., parallel lines appearing to converge).

To handle the geometry of perspective projection more elegantly, we use **Projective Space** (P^3).

- **Homogeneous Coordinates:** A point with Euclidean coordinates (x, y, z) is represented in homogeneous coordinates by an equivalence class of 4D vectors $[\lambda x, \lambda y, \lambda z, \lambda]^T$ for any non-zero scalar λ. A common choice is to set $\lambda = 1$, giving $[x, y, z, 1]^T$.
- **Mapping Back:** To convert from homogeneous coordinates $[X, Y, Z, W]^T$ (where $W \neq 0$) back to Euclidean coordinates, we divide by the fourth component: $[X/W, Y/W, Z/W]^T$.
- **Points at Infinity:** Points at infinity correspond to directions in 3D space. In homogeneous coordinates, they are represented by vectors of the form $[X, Y, Z, 0]^T$, where $[X, Y, Z]^T$ defines the direction. These points cannot be mapped back to the finite Euclidean space.
- **The Origin:** The origin of the Euclidean space $(0, 0, 0)$ is represented by $[0, 0, 0, k]^T$ with $k \neq 0$ in homogeneous coordinates. The vector $[0, 0, 0, 0]^T$ is undefined in projective space.

The main advantage of using homogeneous coordinates is that the non-linear perspective projection equations become **linear transformations** represented by matrix multiplication.

2.5.2 Basic Perspective Projection Matrix (Pinhole Camera)

Let $M = [x, y, z]^T$ be a 3D point expressed in the **Camera Reference Frame (CRF)**, with the origin at the camera's optical center and the z-axis along the optical axis. Let its projection onto the image plane be $m = [u, v]^T$. Assuming the image plane is at $z = f$ (where f is the focal length), the basic perspective projection is: $u = f\frac{x}{z}$ and $v = f\frac{y}{z}$.

Using homogeneous coordinates, we represent the 3D point as $\tilde{M}_{CRF} = [x, y, z, 1]^T$ and the projected 2D point as $\tilde{m} = [u', v', w']^T$, where the final image coordinates are $u = u'/w'$ and $v = v'/w'$. The projection can be written as a linear mapping:

$$w' \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This can be written compactly as $\tilde{m} \propto P_0 \tilde{M}_{CRF}$, where $\propto$ denotes equality up to a non-zero scale factor, and P_0 is the basic 3×4 perspective projection matrix for a pinhole camera centered at the origin, looking along the z-axis. The scale factor here is $w' = z$.

2.5.3 A More Comprehensive Camera Model

The basic model is often insufficient. We need to account for:

1. **Image Coordinate System:** Image coordinates (u, v) are typically measured in pixels, with the origin often at the top-left corner, not the principal point (where the optical axis pierces the image plane).
2. **Pixel Shape:** Pixels might not be square. We need parameters for pixel size/density $(\Delta u, \Delta v)$ or equivalently, focal lengths measured in pixels ($f_u = f/\Delta u$, $f_v = f/\Delta v$).
3. **World Reference Frame (WRF):** 3D points are usually defined in a convenient WRF, not the CRF. We need to relate the WRF to the CRF.

These factors lead to a more complex projection matrix $\tilde{P}$, which can be decomposed into intrinsic and extrinsic parameters.

Intrinsic Parameter Matrix (A) This 3×3 matrix maps projected points from the camera's normalized image plane (where $z = 1$) to pixel coordinates. It encodes the camera's internal geometry:

- f_u, f_v: Focal lengths expressed in units of horizontal and vertical pixel dimensions.
- (u_0, v_0): Coordinates of the principal point (the image center) in pixels. Sometimes denoted (c_x, c_y).

Assuming zero skew, the intrinsic matrix is:

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

These **intrinsic parameters** (at least 4: f_u, f_v, u_0, v_0) are specific to the camera and lens, independent of the camera's position or orientation in the world.

The projection from the CRF using only intrinsic parameters can be written by combining A with a standard projection into the normalized image plane:

$$\tilde{P}_{int} = A[I|\mathbf{0}] = \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix $\tilde{P}_{int}$ maps a 3D point in homogeneous coordinates *relative to the CRF* to its 2D homogeneous pixel coordinates.

Vanishing Points Example Consider a point at infinity $[a, b, c, 0]^T$ representing a direction relative to the CRF. Its projection using $\tilde{P}_{int}$ is:

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

Assuming $c \neq 0$ (the direction is not parallel to the image plane), we can convert to Euclidean pixel coordinates by dividing 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 projected point m_∞ is a **vanishing point** in the image – the point where parallel lines in 3D space with direction $[a, b, c]^T$ appear to converge. If $c = 0$, the direction is parallel to the image plane, and the point projects to infinity in the image plane coordinates (no finite vanishing point).

Extrinsic Parameter Matrix (G) To use points defined in a World Reference Frame (WRF), we need to describe the camera's pose (position and orientation) relative to the WRF. This is done using a rigid body transformation:

- R: A 3×3 rotation matrix describing the orientation of the CRF relative to the WRF.
- T: A 3×1 translation vector describing the position of the CRF origin relative to the WRF origin (expressed in WRF coordinates). Or sometimes, the position of the WRF origin relative to the CRF origin (expressed in CRF coordinates) - convention matters! Let's assume T specifies the CRF origin in WRF for the G matrix below which transforms points *from* WRF *to* CRF.

The transformation from WRF coordinates $\tilde{M}_W = [X, Y, Z, 1]^T$ to CRF coordinates $\tilde{M}_{CRF} = [x, y, z, 1]^T$ is given by:

$$\tilde{M}_{CRF} = G\tilde{M}_W$$

where G is the 4×4 **Extrinsic Parameter Matrix**. A common form relates WRF coordinates to CRF coordinates:

$$G = \begin{bmatrix} R & T \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Here, R rotates the WRF axes to align with the CRF axes, and T translates the origin. This transformation has **6 extrinsic parameters**: 3 for rotation (e.g., Euler angles, axis-angle) and 3 for translation.

The transformation from WRF to CRF coordinates (non-homogeneous) is $M_{crf} = RM_w + T'$, where T' is the translation vector. In homogeneous coordinates, this is often expressed using a 4×4 matrix that incorporates the inverse transformation or directly within the projection matrix.

2.5.4 The Complete Perspective Projection Matrix ($\tilde{\mathbf{P}}$)

Combining the intrinsic and extrinsic parameters, the full projection from a 3D point $\tilde{M}_W$ in WRF homogeneous coordinates to a 2D point $\tilde{m}$ in image homogeneous pixel coordinates is:

$$\tilde{m} \propto A[R|T]\tilde{M}_W$$

Here, the 3×4 matrix $\tilde{P} = A[R|T]$ is the complete **Perspective Projection Matrix (PPM)**.

- R is the 3×3 rotation matrix specifying the camera's orientation.
- T is the 3×1 translation vector specifying the camera's position (specifically, $T = -RC_W$, where C_W is the position of the camera center in world coordinates).

The matrix $[R|T]$ effectively transforms the point from WRF coordinates directly into the CRF, ready for projection and intrinsic transformation by A.

The PPM $\tilde{P}$ encodes all the geometric information of the camera setup:

- Intrinsic parameters (A): Camera's internal geometry (focal length, principal point, pixel properties). (4+ parameters)
- Extrinsic parameters (R, T): Camera's pose (position and orientation) in the world. (6 parameters)

Camera calibration aims to find the numerical values for the parameters in A and, often simultaneously or subsequently, the extrinsic parameters R and T relative to a known calibration target.

2.5.5 P as a Homography

We just need the relation between the camera and the plane, then we can measure everything on that plane.

If the camera is imaging a planar scene, we can assume the z-axis of the World Reference Frame to be perpendicular to the plane such that all 3D points will have their z-coordinate equal to 0. The Perspective Projection Matrix then becomes a simpler transformation defined by a $3x3$ matrix.

$$k\tilde{m} = \tilde{P}\tilde{w} \begin{bmatrix} p_{1,1} & p_{1,2} & \cancel{p_{1,3}} & p_{1,4} \\ p_{2,1} & p_{2,2} & \cancel{p_{2,3}} & p_{2,4} \\ p_{3,1} & p_{3,2} & \cancel{p_{3,3}} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H\tilde{M}$$

Such a transformation, denoted here as H, is known as **homography** and represents a general linear transformation between projective planes. A homography is then a projective transformation between two planes or a mapping between two planar projection of an image.

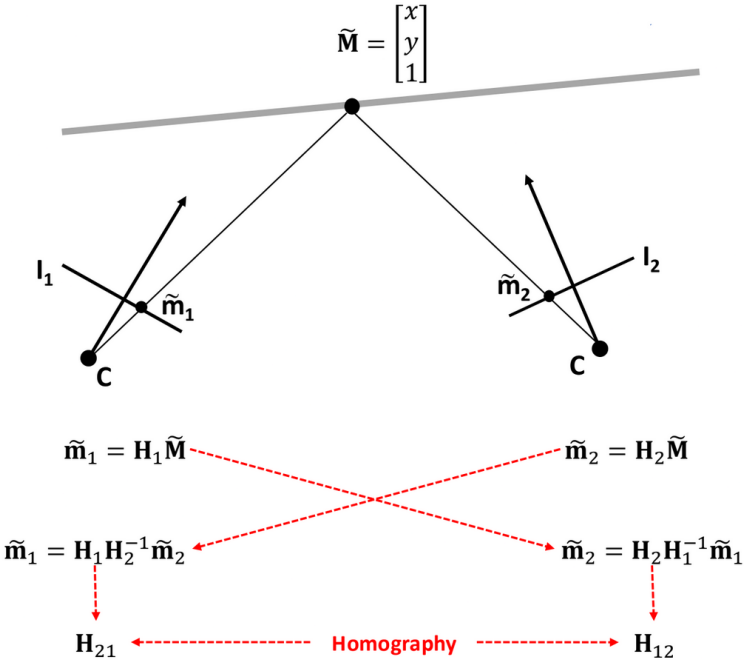


Figure 32: Any two images of a planar scene are related by a homography

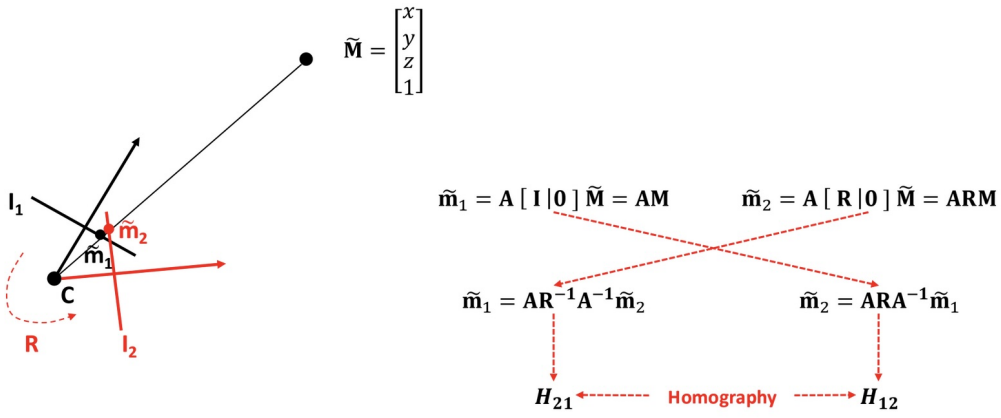


Figure 33: Any two images taken by a camera rotating about the optical center are related by a homography

2.5.6 Lens distortion

The Perspective Projection Matrix is based on the pinhole camera model, however, real lenses introduce distortions with respect to the pure pinhole model. We also have lens distortion, which is modelled through additional parameters that don't alter the form of the Perspective Projection Matrix.

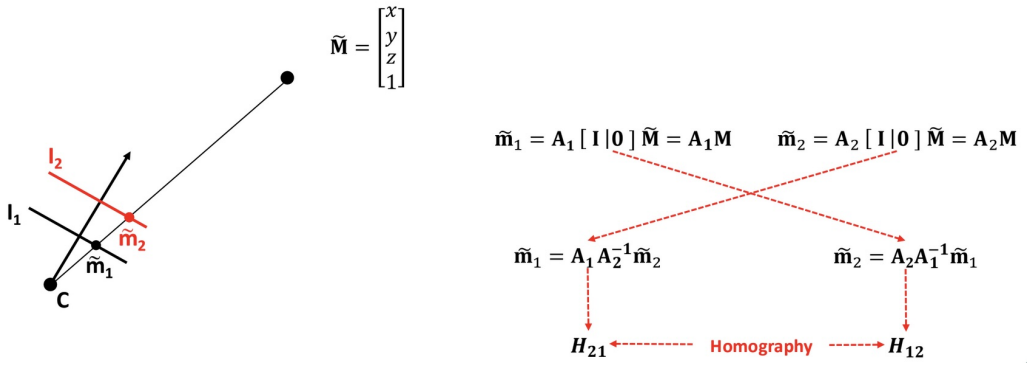


Figure 34: Any two images taken by different cameras in fexed pose are related bu a homography

We have two types of lens distortion: barrel distortion which bends straight lines outwards, and pincushion distortion, which causes straight lines to curve inward.

2.5.7 Calibration

We now have the Perspective Projection Matrix camera model, which can be decomposed in:

- Intrinsic parameter matrix A.
- Rotation matrix R.
- Translation vector T.

Camera calibration is the process where **all parameters defining the camera model are estimated** for a specific camera device. Depending on the application, **either the Perspective Projection Matrix (PMM) only, or also its independent components** (A,R,T) need to be estimated.

There are many camera calibration algorithms, but the basic process always relies on setting up a linear system of equations given a set of known 3D-2D correspondences. To obtain the correspondences we use calibration targets, which have easily detectable features.

The approaches can be split into:

- Those relying on a single image containing a known pattern.
- Those relying on several different images of one given planar pattern.

Since in every frame we change the relative position of the camera and the pattern, we will have a different set of extrinsic parameter for each image. We will have just one set of intrinsic parameter since the camera is always the same across the different pictures.

2.5.8 Zhang's method for Camera Calibration

We use a chessboard pattern of which we know the number of internal corners and the size of the squares. Internal corners can be easily detected by standard algorithms, like the Harris corner detector. Typically the camera is fixed and you move the calibration target in front of the camera. The $[RT]$ are estimated with respect to the reference system attached to the target, but it changes alongside with the pattern. In each image the 3D world reference frame is taken at the top-left corner of the pattern. Each image requires its own estimate of the extrinstic parameters, as they are different from one to the other. Due to the choice of the world reference frame associated with calibration images, in each of them we consider only 3D points with $z = 0$. The Perspective Projection Matrix boils down to a simpler transformation defined by an homography 3×3 matrix, like with P as a homography.

$$k\tilde{m} = \tilde{P}\tilde{w} \begin{bmatrix} p_{1,1} & p_{1,2} & \cancel{p_{1,3}} & p_{1,4} \\ p_{2,1} & p_{2,2} & \cancel{p_{2,3}} & p_{2,4} \\ p_{3,1} & p_{3,2} & \cancel{p_{3,3}} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ \cancel{0} \\ 1 \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H\tilde{M}$$

Estimating H_i (DLT algorithm) Given a pattern with m corners, we can write m systems of 3 linear equations, where:

- Both 3D as well as 2D coordinates are known due to the corners having been detected in the i-th image and the unknowns are thus the 9 elements in H_i.
- H_i (and P_i alike) is known up to an arbitrary scale factor, the independent elements in H_i are 8.

There are plenty of methods for the estimation

The Zhang's method can be summarized as:

1. Acquire n images of a planar pattern with m internal corners.

2. For each image compute an initial guess for homography H_i.
3. Refine eaech H_i by minimizing the reprojection error.
4. Get an initial guess for A given the homographies H_i.
5. Given A and H_i, get an initial guess for R_i and T_i.
6. Compute an initial guess for lens distortion parameters k.
7. Refine all parameters A, R_i, T_i, k by minimizing the reprojection error.

2.5.9 Image warping

After applying the warping function you get real coordinates not discrete ones. During the mapping, some pixels of the destination image may be not rounded perfectly. What if more pixels go to the same position?

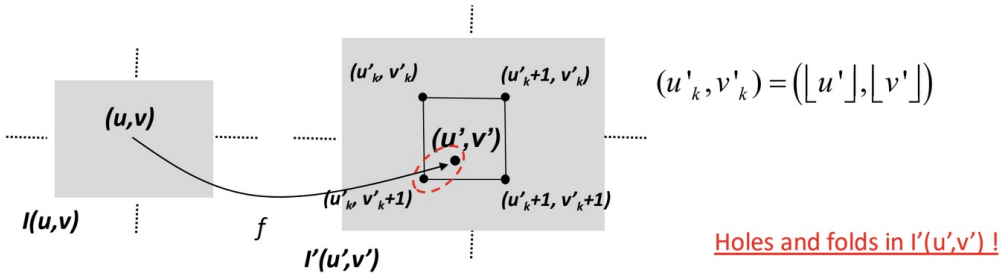


Figure 35: A better choice consists in mapping to the closest point into the destination image

Since the coordinates are always real values the different mapping strategies can be to map the closest point or to interpolate between the 4 closest points.

3 Advanced Topics in Deep Learning for Computer Vision

3.1 Recall on CNNs

In representation learning we try to find good ways of representing data, often by learning a useful transformation of the raw data. Deep learning is a subset of representation learning.

3.1.1 Gradient descent

Neural networks are usually trained using iterative, gradient-based optimizers that drive the cost function to a very low value. The convergence point of gradient descent depends on the initial values of the parameters. For feedforward neural networks it's important to initialize all weights to small random values and to initialize biases to zero or to small positive values. To apply gradient based learning we must choose a cost function and how to represent the output of the model. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. The derivative is useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. A point that obtains the absolute lowest value of $f(x)$ is a global minimum.

To reach the minimum we use the formula $w' = w - \alpha\frac{\partial J(w,b)}{\partial w}$ $b' = b - \alpha\frac{\partial J(w,b)}{\partial b}$, where α is the learning rate, which controls how big are the steps that we take during the gradient descent. If the slope is positive, we subtract the quantity in order to reach the minimum, or else we do the opposite thing.

As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long. Stochastic/Minibatch gradient descent are extensions of the gradient descent algorithm. We can sample a minibatch of examples drawn uniformly from the training set (1 for stochastic). It's like training every time on a different (smaller) training set.

Optimizers: momentum Momentum was designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The size of the step depends on how large and how aligned a sequence of gradients are. The step size is largest when many successive gradients point in the same direction. Momentum smooths the step of the gradient descent in its path to the minimum. We compute: $W^{[l+1]} = W^{[l]} - \alpha v_{W^{[l+1]}}$, where $v_{W^{[l+1]}} = \beta v_{W^{[l]}} + (1-\beta)\frac{\partial J(...)}{\partial W^{[l]}}$. Root Mean Square Propagation (RMSprop) uses an exponentially decaying average to discard history from the extreme past so that it doesn't impede the convergence.

Optimizers: Adam The name derives from "adaptive momentum", and it can be seen as the combination of RMSprop and momentum. It's fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default. There is a first order and a second order Adam. The first order Adam uses the gradient and its moving averages (first moments) to update the parameters. The second order Adam incorporates information about the curvature of the loss surface, using approximations of the Hessian or second order derivatives, which leads to more informed and potentially more efficient updates but requires more computational resources.

3.1.2 Convolutions and filters

An image is characterized by height, width and number of channels.

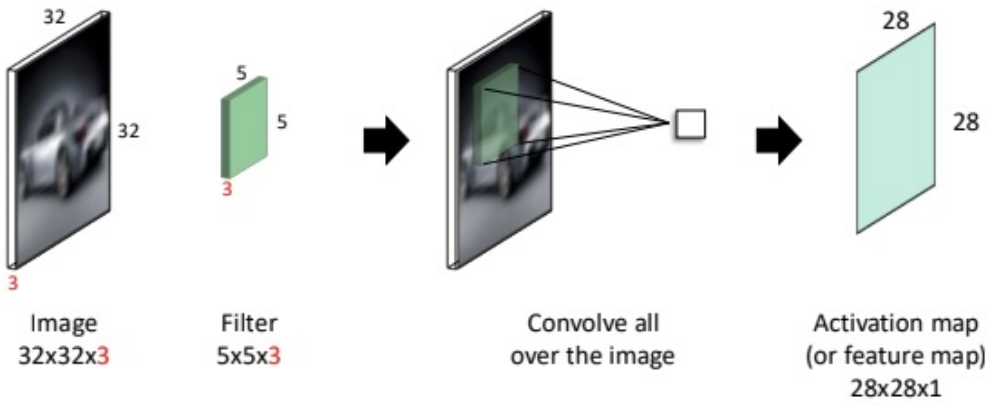


Figure 36: A convolutional filter has the **same depth** of the input volume, so the output dimension is 1.

We can also train more than one filter, where each filter outputs a feature map. By stacking activation maps we get a new volume.

Since convolutions shrink the images we can use padding to preserve the edges. We have always considered a stride of 1 (moving the convolutional filter by one pixel), but we can also use different stride sizes to shrink the image.

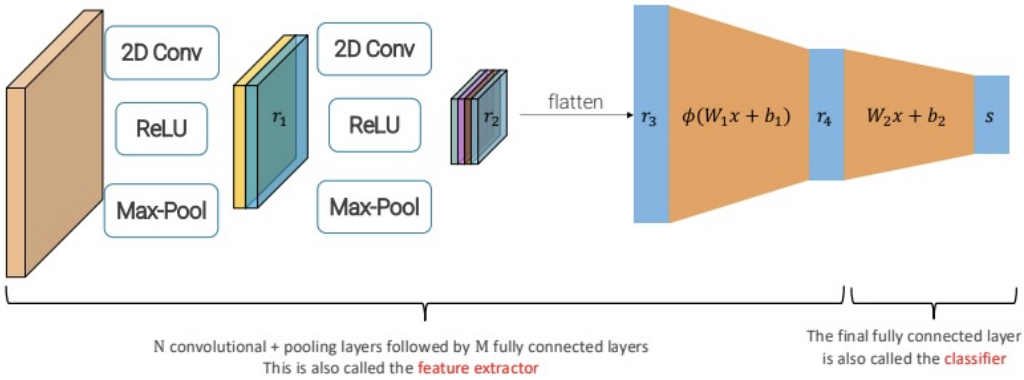


Figure 37: By stacking together convolutional filters, pooling layers and fully connected layers we can design a Deep CNN.

Pooling A pooling layer is used to reduce the size of the representation in order to speedup computation. Pooling comes usually after each conv layer or after a block (or set) of conv layers. It's applied to each activation map independently.

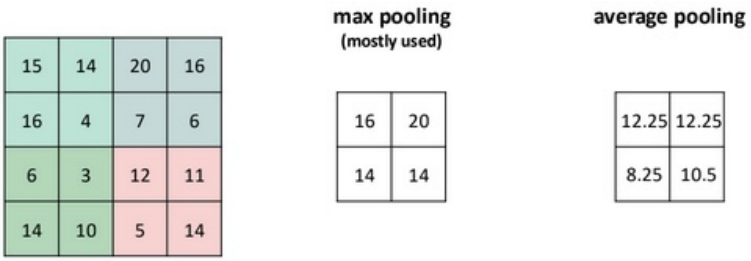
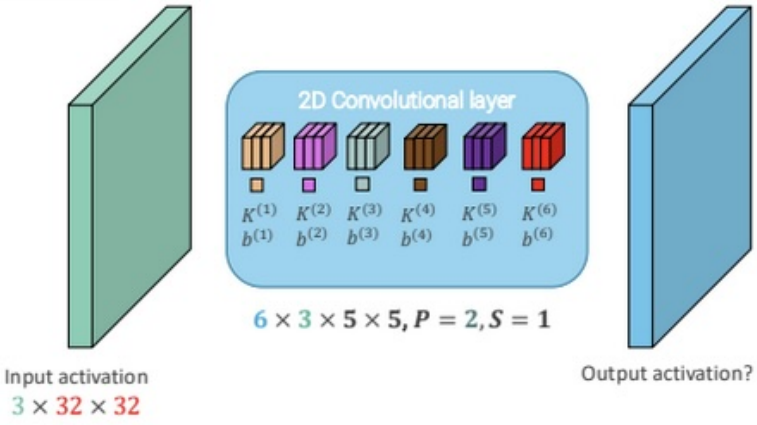


Figure 38: Example: pooling of dimension 2 and stride 2.

Receptive fields The input pixels affecting a hidden unit are called its receptive field. We can encode the information in all the pixels to a single point. We go from spatial to semantic information (we transform spatial information to semantic information).



Convolution parameters and flops As we can see in 3.1.2, the number of learnable parametes for the convolutional layer is $6 \times (3 \times 5 \times 5 + 1) = 6 \times 76 = 456$ (6 convolutional blocks, 3 channels per convolutional block, 5×5 convolution,). The size of the output is $H_{\text{out}} = W_{\text{out}} = 32 - 5 + 2 * 2 + 1 = 32$. Hence, there are $6 \times 32 \times 32 = 6144$ values in the output activation ($\approxeq$ 24KB).

Each of them is obtained as the dot product between the weights and the input, which requires to perform n multiplications and n summations for inputs of size n, i.e. $2n$ flops.

3.1.3 Batch Normalization (BatchNorm)

BatchNorm is a technique designed to stabilize and accelerate the training of deep neural networks by addressing internal covariate shift: the change in the distribution of layer activations caused by updates to preceding layers during training. By normalizing activations at each layer, BatchNorm ensures that gradients are propagated more effectively, enabling coordinated updates across deep networks.

Let $Z^{[l]}$ be a minibatch of activations of the l-th layer to be normalized. To normalize $Z^{[l]}$, we replace it with: $Z^{[l]}_{norm} = \frac{Z^{[l]}-\mu}{\sigma}$, where μ is a vector containing the mean of each unit and σ is a vector containing the standard deviation of each unit. μ and σ are running averages of the values seen during training. BatchNorm helps with speeding up the training, and has a slight regularization effect (but we don't use it for this reason).

We use LayerNorm for fully connected layers because it normalizes per sample, making it batch-size independent. We use InstanceNorm for convolutional layers because it normalizes per sample and per channel.

3.1.4 Dropout regularization

Dropout randomly deactivates a fraction of neurons during training, effectively training an ensemble of smaller sub-networks. For each layer, you set a dropout probability, determining the chance a neuron is ignored in a forward pass. This helps to prevent overfitting because we don't associate a certain pattern to a certain output of the network.

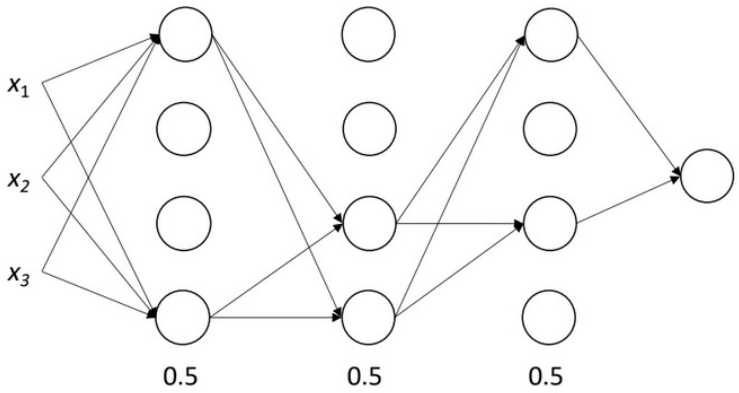


Figure 39: Each layer has a 50% dropout probability

Dropout and normalization are used only at training time. When we test the network we switch off dropout and regularization.

Another powerful regularization technique is just having more data.

3.1.5 Data augmentation

We can create more data by modifying the images we already have. We must only make transformations which keep the labels valid. We can, for example, sample random crops/scales of the data or do color augmentation.

Cutout is the process where we remove a random square region of the input image. This forces the network to use a more diverse set of features, helping generalization. It's gray because we subtract the mean from this pictures, and so the number in the cutout becomes 0.

3.2 CNNs

3.2.1 AlexNet & ZFnet

In 2012 Alex Krizhevsky proposed AlexNet, which had almost a 9% of improvement on SOTA for image classification.

In figure 40, only one half of the network is shown, as the architecture is replicated identically for a 2-GPU setup.

The network begins with a **stem layer**, which is a convolutional layer that quickly reduces the spatial dimensions of the activations to lower memory and computational requirements. The first layer uses two 11×11 kernels to extract features such as corners, edges, and blobs.

However, this design had limitations: the large kernel size (11×11) and stride (4) in the first layer caused a significant **loss of spatial information** early in the network, making it harder to capture fine-grained features.

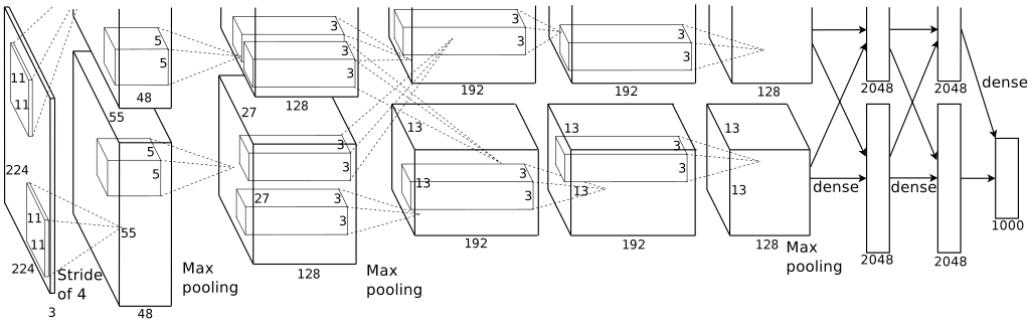


Figure 40: AlexNet architecture

To address these issues, ZFNet replaced the initial 11×11 convolutions with 7×7 convolutions with stride 2 in the first layer, and 5×5 convolutions with stride 2 in the second layer.

3.2.2 VGG

The VGG network architecture is characterized by its simplicity and depth, achieved by exclusively using small 3×3 convolutional filters and omitting a distinct 'stem' layer. The core structure consists of repeated blocks of convolution and pooling layers, followed by a standard flattening layer and a classification head. Training such deep networks presented challenges, including the vanishing gradient problem. As batch normalization was not yet available during VGG's development, pre-initializing network weights from shallower, pre-trained architectures was a crucial technique for enabling effective training.

VGG-16 VGG-19

ConvNet Configuration				
A	B	C	D	E
11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)				
conv3-64	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool				
conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool				
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool				
FC-4096				
FC-4096				
FC-1000				
soft-max				

Figure 41: VGG network architecture

VGG introduced a modular design principle based on repeating stages. Each stage consists of a fixed sequence of layers that process activations while maintaining the same spatial resolution within that stage. VGG employs three main types of such repeating blocks:

- `conv-conv-pool`.
- `conv-conv-conv-pool`.
- `conv-conv-conv-conv-pool` (we can get a more complex convolution by combining 3×3 convolutions).

The use of stacked 3×3 convolutions within a stage, as detailed in the table below, provides an effective receptive field comparable to that of a single larger convolution (e.g., a 5×5 filter). However, this approach requires fewer parameters and less computation while also introducing more non-linear activation functions (ReLUs).

A notable trade-off, also highlighted in the table, is the increased memory required for storing activations when using stacked 3×3 convolutions compared to a single larger filter.

convolutional layer	params	flops	ReLUs	number of activations
$C \times C \times 5 \times 5$, $S = 1$, $P = 2$	$25C^2 + C$	$50C^2W_{in}H_{in}$	1	$C \times W_{in} \times H_{in}$
2 stacked $C \times C \times 3 \times 3$, $S = 1$, $P = 1$	$18C^2 + 2C$	$10C^2W_{in}H_{in}$	2	$2 \times C \times W_{in} \times H_{in}$

The VGG-16 model, for instance, comprises 138 million parameters (approximately 2.3 times that of AlexNet), a significant portion of which are in the fully connected layers. It demands considerable computational resources, around 4 TFLOPs, largely due to its convolutional operations, and requires approximately 16.5 GB of memory.

3.2.3 Inception v1 (GoogLeNet)

A key feature of this architecture is the efficient use of computational resources within the network. This is achieved through a carefully designed structure that allows for increased depth and width while maintaining a fixed computational budget.

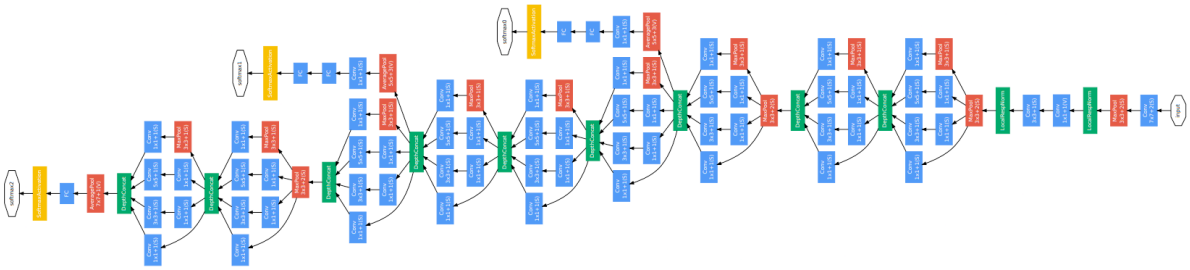
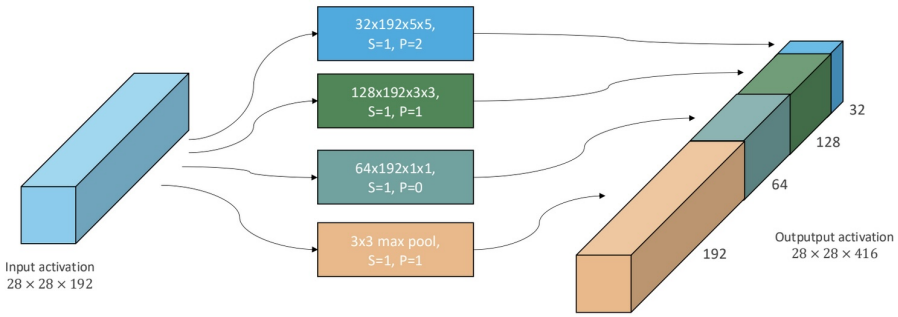


Figure 42: GoogLeNet architecture

The network consists of stem layers that reduce the input size, a series of inception modules that combine various operations, and a final classifier. It includes 22 trainable layers and roughly 100 modules (represented by the blue and red blocks).

Stem layers Stem layers downsample the input aggressively, from 224 to 28 pixels in width/height, within just 5 layers. This approach is slightly more gradual than in AlexNet. For comparison, VGG requires 10 layers to reach a spatial size of 28×28.

Naïve inception module The core idea of the inception module is to perform several pooling and convolution operations of different sizes (e.g., 3×3, 5×5) in parallel, rather than using a single type of filter.



However, this design introduces two main issues:

- The use of max pooling causes the number of channels to grow rapidly when stacking inception modules.
- Performing 3×3 and 5×5 convolutions on many channels becomes computationally expensive as the network deepens.

1×1 convolutions and the inception module To address these problems, 1×1 convolutions are applied before the 3×3 and 5×5 convolutions. These 1×1 convolutions do not capture spatial relationships but operate across channels at each spatial location. They enable us to adjust the depth of activations while preserving spatial resolution, effectively serving as a form of dimensionality reduction.

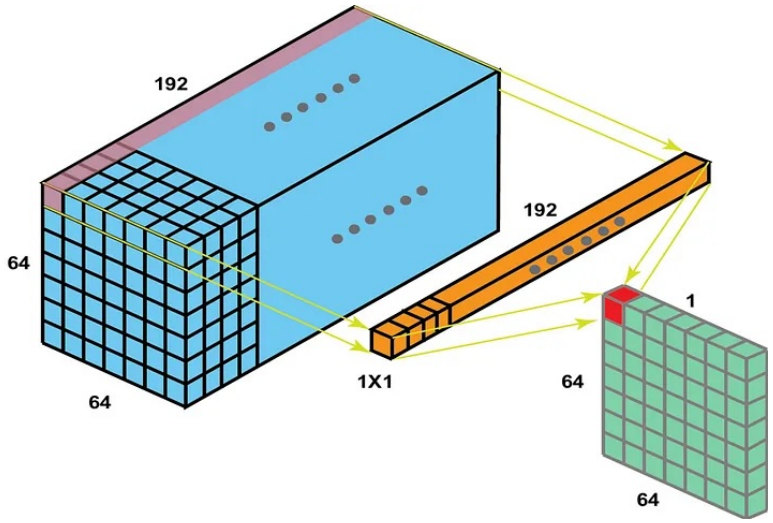


Figure 43: 1×1 convolution

By introducing 1×1 convolutions before the larger convolutions and after pooling, we can:

- **Control time complexity** by reducing the number of channels processed by larger convolutions.
- **Limit the output depth** of the max pooling layers by shrinking the channel dimension.

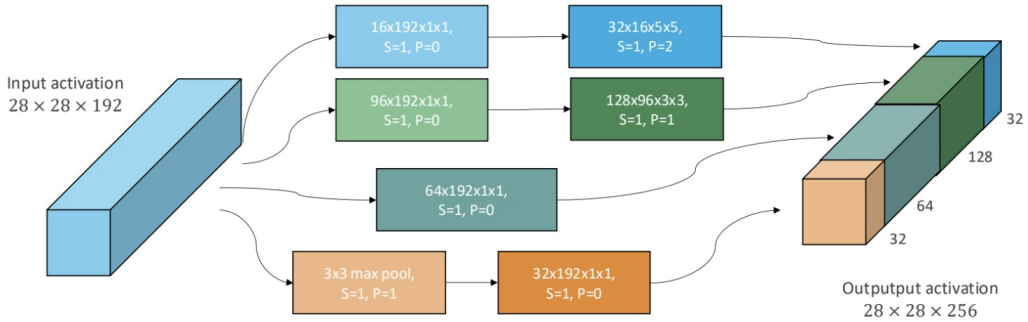


Figure 44: Inception module in GoogLeNet

In Figure 44, we see how Google used 1×1 convolutions to implement an inception module without incurring in excessive computational cost. The approach is to first reduce the number of channels with 1×1 convolutions, then apply the 3×3 and 5×5 spatial convolutions. After pooling operations, another 1×1 convolution reduces the output depth. We reduce dimensionality **after** pooling since compression isn't necessary beforehand.

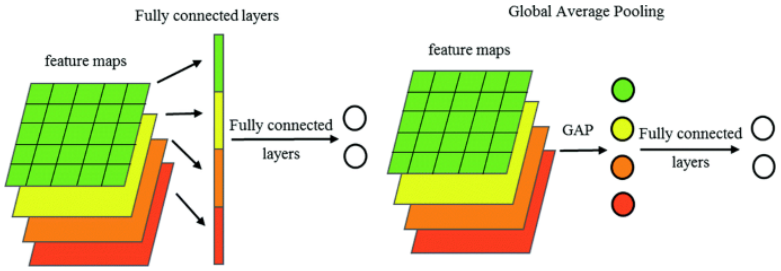


Figure 45: Fully Connected Classifier vs Global Average Pooling

Fully-connected classifier vs global average pooling The final three fully connected layers typically account for a large portion of the parameters due to high-dimensional spatial inputs. To mitigate this, we eliminate the spatial dimensions by averaging across them. This is justified by the assumption that the final activations represent high-level semantic features.

3.2.4 Inception v3

Inception v3 improves computational efficiency and reduces parameter count through convolutional factorization. The two main techniques are:

- Replacing a 5×5 convolution with two sequential 3×3 convolutions (as in VGG).
- Factorizing a 3×3 convolution into a 3×1 followed by a 1×3 convolution.

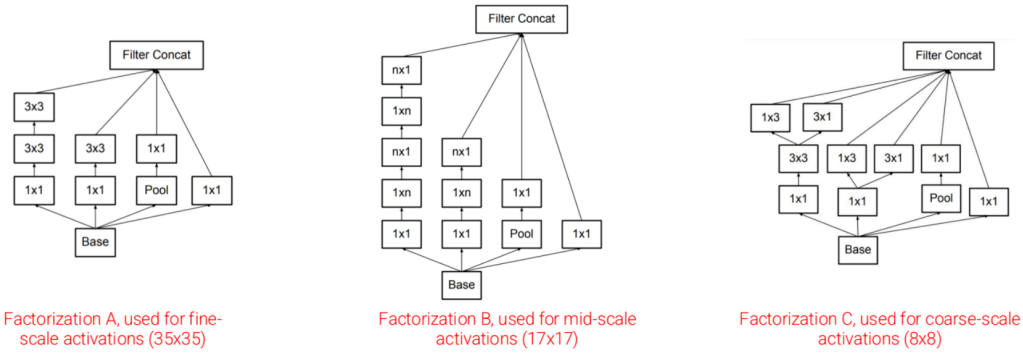


Figure 46: Inception v3 uses three different types of inception layers instead of a single repeated one

3.2.5 Residual Networks (ResNet)

ResNet was designed to solve two major problems in deep neural networks: the vanishing gradient problem and degradation (when adding more layers actually makes performance worse).

The key idea behind ResNet is the **residual block**, which includes a **skip connection**. In standard networks, each layer passes its output directly to the next layer. In ResNet, the input to a layer is also added to its output:

$$H(x) = F(x) + x$$

This means the network learns the residual function $F(x) = H(x) - x$, which is easier to optimize.

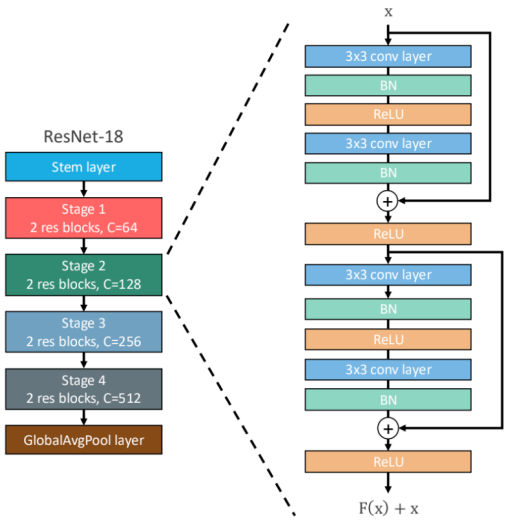


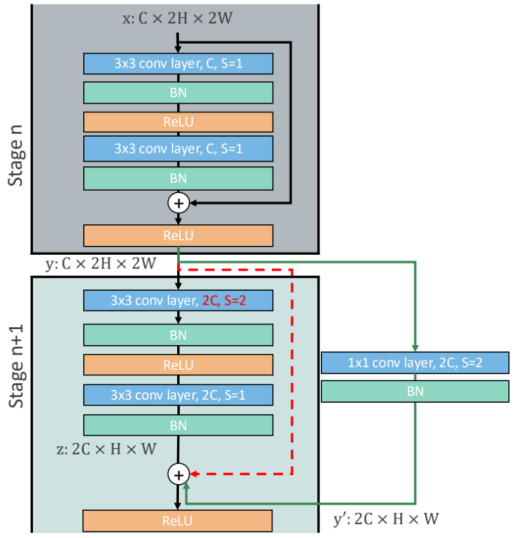
Figure 47: The ResNet architecture

ResNet architecture follows a structured pattern, inspired by VGG:

- Each **stage** is made up of multiple residual blocks.
- Each residual block contains two 3×3 convolutional layers, each followed by batch normalization.
- The first block of each stage reduces the spatial resolution (via stride 2) and doubles the number of channels.
- The network starts with a stem layer and ends with global average pooling, like GoogLeNet.

ResNet remains a standard baseline in many computer vision tasks.

Handling dimension mismatch Residual blocks cannot be directly used at the start of a new stage, since the number of channels and spatial resolution change. To handle this, ResNet uses a 1×1 convolution with stride 2 to match dimensions across the skip connection.



Bottleneck Residual Block For deeper networks like ResNet-50/101/152, a **bottleneck block** is used to reduce computational cost. Instead of stacking two 3×3 convolutions, the bottleneck design compresses and then expands channels:

- 1×1 conv to reduce dimensions from $4C$ to C
- 3×3 conv processes C channels
- 1×1 conv expands back to $4C$

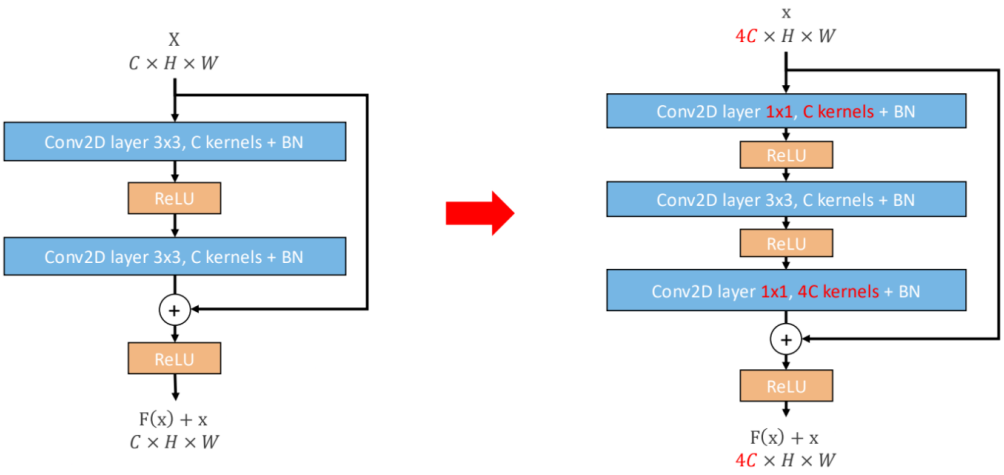


Figure 48: Bottleneck residual block

3.2.6 ResNeXt

ResNeXt builds on ResNet but introduces a new dimension: **cardinality**, which is the number of parallel paths or groups.

Inspired by Inception modules (which split-transform-merge), ResNeXt uses multiple paths inside each residual block. Each block is split into G branches (cardinality), and each branch learns part of the features.

The ResNeXt block is a simplified and efficient version of multi-branch architecture.

Grouped Convolutions Each branch is implemented using **grouped convolutions**:

- The input with C channels is split into G groups, each with $\frac{C}{G}$ channels.
- Each group has separate filters, and processes data independently.
- The outputs of all groups are concatenated, giving K total channels (for K filters).

Structure of a ResNeXt Block

- 1×1 convolution to reduce input channels.
- 3×3 grouped convolution processes features in G groups.
- 1×1 convolution expands channels back.

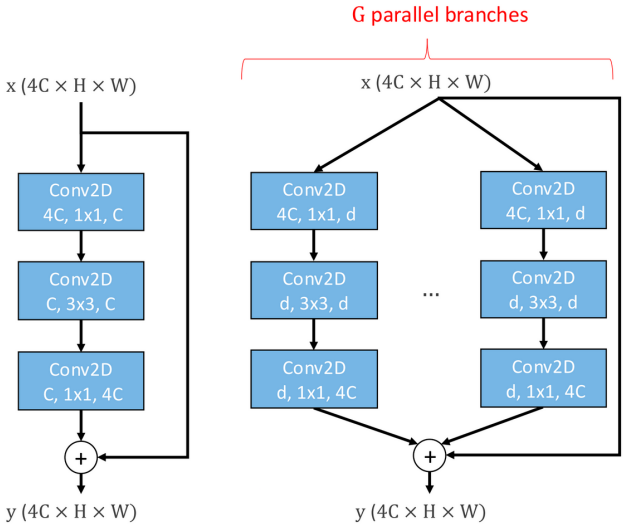


Figure 49: ResNeXt network architecture

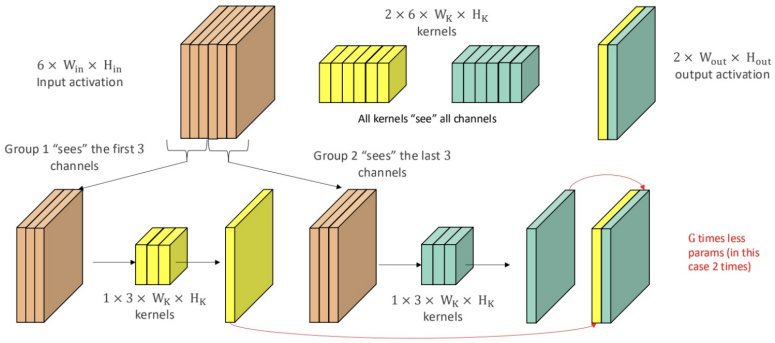


Figure 50: Grouped convolution in ResNeXt with $G = 2$

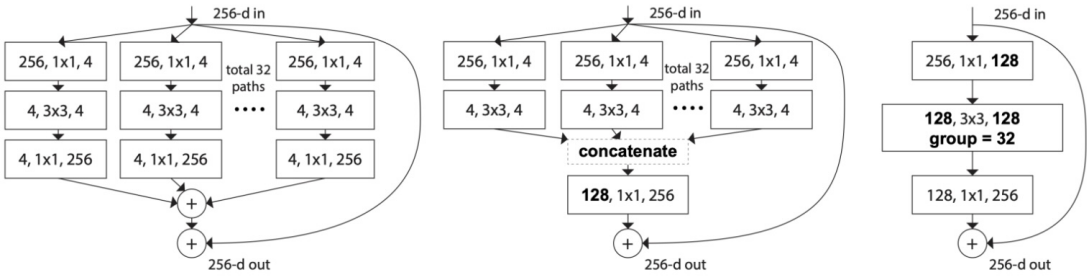


Figure 51: ResNeXt block with grouped convolutions: split, transform, merge.

- Skip connection adds the original input.

Why ResNeXt? It improves performance with fewer parameters and less computation than simply increasing network depth or width. Increasing cardinality G enhances feature learning efficiency without a significant cost in resources.

3.2.7 Squeeze-and-Excitation Networks (SENet)

It's the last paper which won CVPR. It proposed the squeeze-and-excitation module to capture global context and to use it to reweight channels in each block. Given the output U of a block with shape $C \times H \times W$ we do:

- Global average pooling to squeeze.
- General bottleneck formed by two fully connected layers with reduction ratio 16.

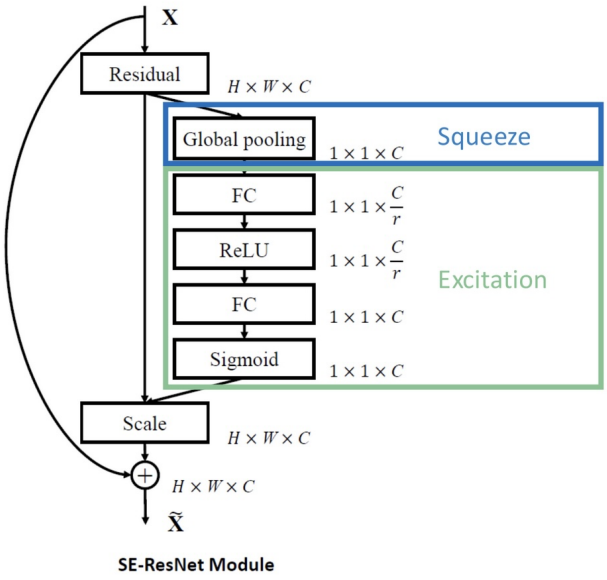


Figure 52: SENet network block diagram

3.2.8 Depthwise Separable convolutions

F. Chollet one year after the SENet paper proposed the Depthwise Separable convolutions. The standard convolutions filter features based on the convolutional kernels and combine features in order to produce new representations. This is very expensive. Depthwise separable convolutions separates filtering and combination, and this aggressively limits the computational cost.

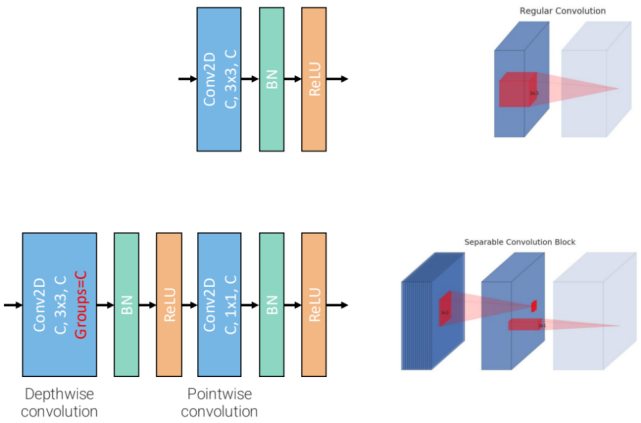


Figure 53: Depthwise Separable convolutions

3.2.9 Inverted residual blocks

MobileNet-v2 introduced the bottleneck residual block to scale up the model depth by increasing the number of layers per block while keeping the computation and number of parameters roughly constant. To this end,

it uses a pair of 1×1 convolutions, where the first compresses the number of channels, while the second one expands them. Hence, the 3×3 convoltuion operates in the compressed domain.

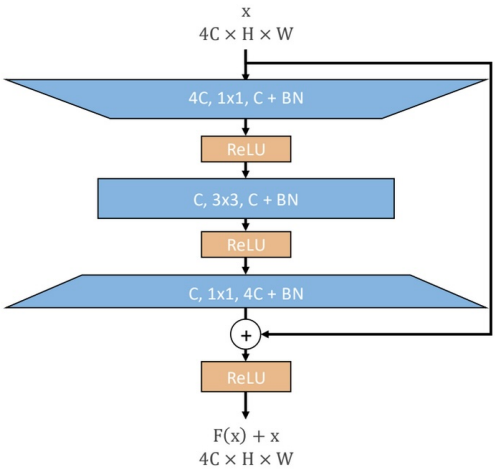


Figure 54: Bottleneck residual block

As we know, compression usually results in information loss, so in MobileNet-v2 inverted residual blocks were proposed. In this blocks, the first 1×1 convolution expands the channels, while the second compresses them back, according to an expansion ratio t. To limit the increase in computation, the inner 3×3 convolution is realized as a depthwise convolution (single filter per input channel instead of applying the filters across all the input channels).

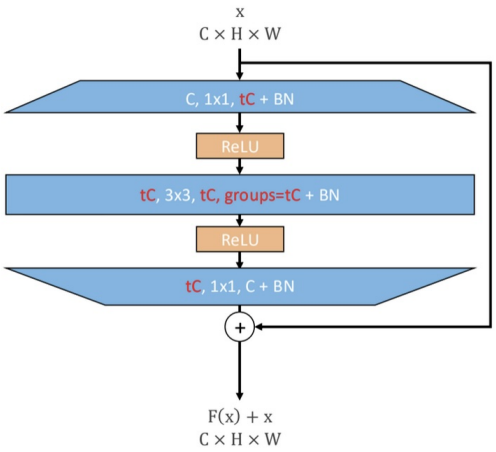


Figure 55: Inverted residual block

3.2.10 MobileNet-v2

MobileNet-v2 is a stack of inverted residual blocks with ReLUs in between. The number of channels grows slowly compared to previous architectures, as we can see in the purple rectangle in figure 56

Whenever spatial dimensions or number of channels do not match between input and output, there are no residual connections.

3.2.11 EfficientNet

EfficientNet proposes designing the network based on available computational resources. There are three main dimensions for scaling a neural network: width, depth, and input resolution.

Larger networks with greater width, depth, or resolution tend to achieve higher accuracy. However, the accuracy gains quickly saturate after around 80%, highlighting a limitation of scaling along a single dimension. **The scaling dimensions are not independent.** Intuitively, when increasing image resolution, we should also increase network depth so that larger receptive fields can capture relevant features spanning more pixels. Similarly, increasing width becomes important at higher resolutions to capture fine-grained patterns present in the additional pixel information.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 56: Mobilenet network architecture

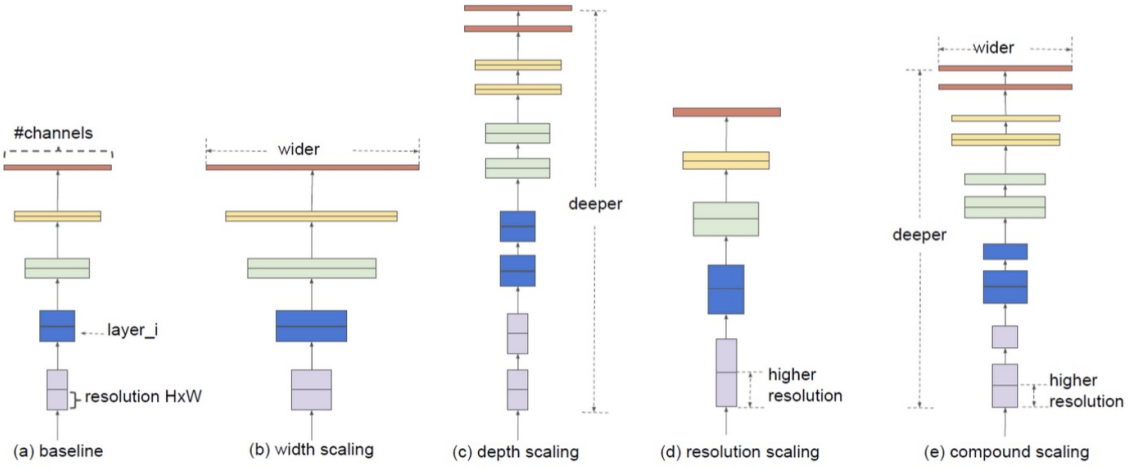


Figure 57: Different types of neural network scaling.

EfficientNet addresses this by using a compound coefficient ϕ to scale all three dimensions in a balanced and principled way.

3.3 RNNs & Transformers

In RNNs, which were used to avoid the problem of fixed size input, the gradient computation involves performing a forward propagation pass moving from left to right through the unrolled graph of the network, followed by a backward propagation pass moving right to left through the graph. The basic problem is that by propagating the gradient over many stages it tends to vanish or explode, so learning long-term dependencies can take a lot of time.

Encoder-Decoder architecture In this type of architecture the encoder processes the input sequence. It emits the context C, usually as a function of its final hidden state. The decoder is conditioned on that fixed-length vector to generate the output sequence. If the context C is a vector, then the encoder RNN is simply a sequence-to-vector RNN, and the decoder is a vector-to-sequence RNN. The **bottleneck problem** arises because the context C outputted by the encoder RNN has a dimension that is too small to properly summarize a long sequence.

Attention Existing models for neural machine translation encode a source sentence into a fixed-length vector from which a decoder generates a translation. Attention provides a solution to the bottleneck problem. We provide to the decoder the learned token start, then we do the dot product between the start token and the hidden states. All the other hidden states give us a probability for each token.

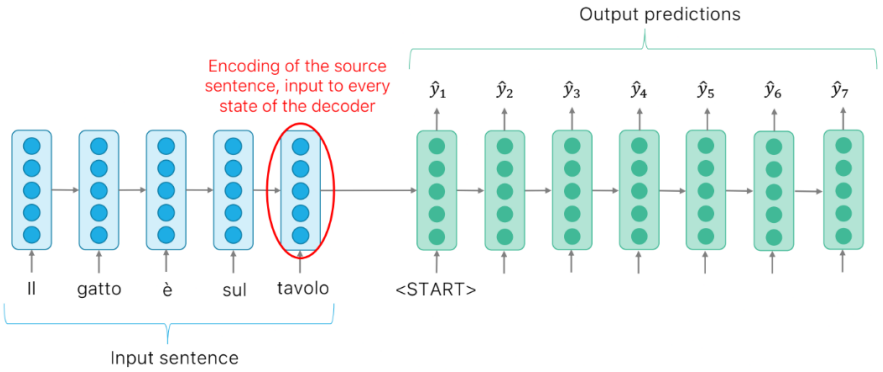


Figure 58: In the classic RNN, the last hidden state is used for all the translation.

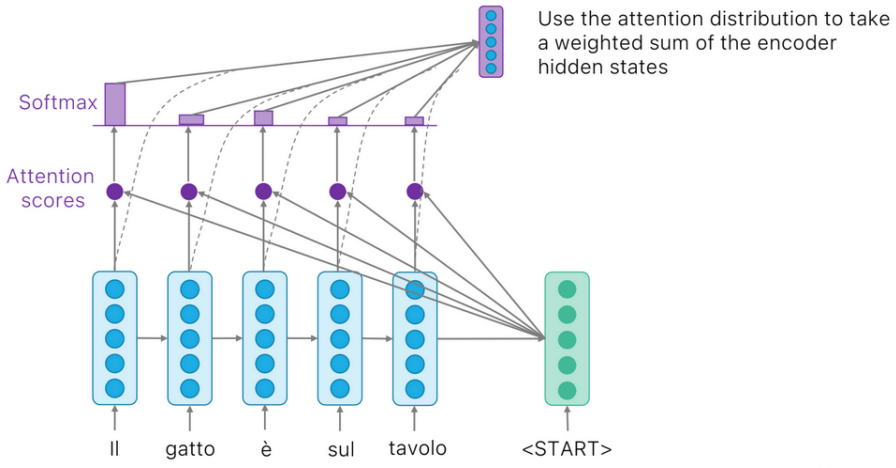


Figure 59: In the attention mechanism, at each step, the current hidden state is compared with all the previous hidden states by computing an attention score.

3.3.1 Transformer architecture

The inherently sequential nature of RNN precludes parallelization within training examples, which becomes critical with longer sequence lengths. The **transformer** is the first model relying entirely on self-attention

(and cross-attention) to compute representations of its input and output without using RNNs or convolutions. It's still an encoder-decoder architecture, where the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given z, the decoder generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is **auto-regressive**, consuming the previously generated symbols as additional input when generating the next.

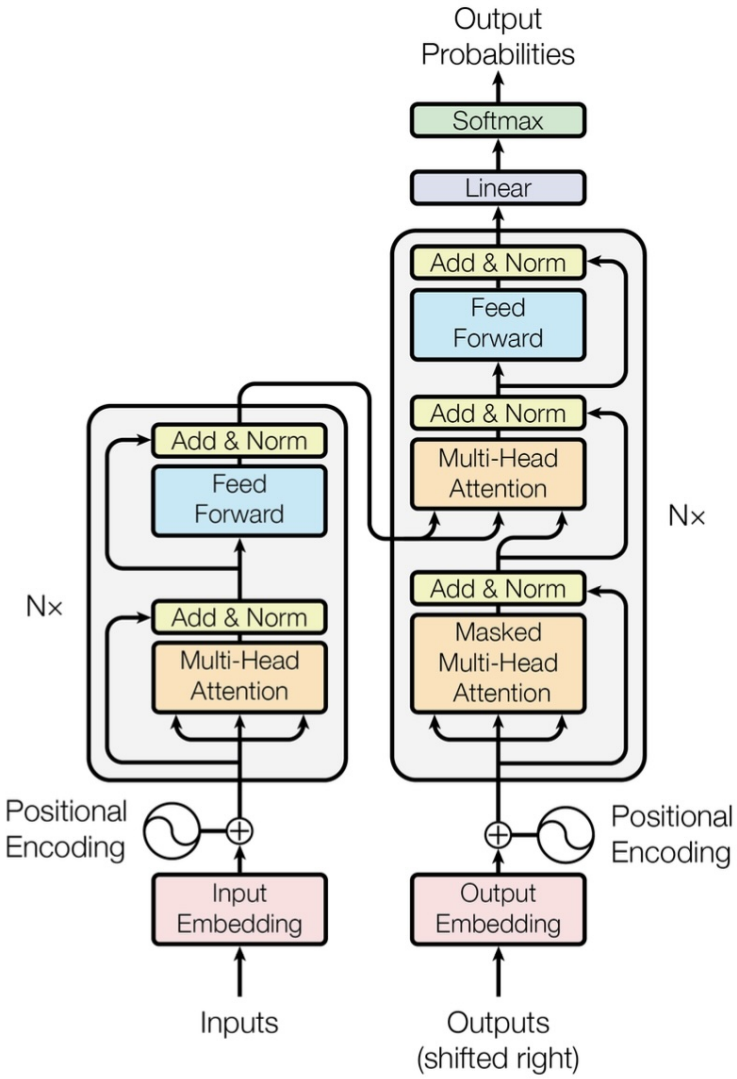


Figure 60: The transformer architecture from the paper "Attention is all you need".

Transformer encoder In the encoder we turn each input word into a vector by using an embedding layer. Each word is embedded into a vector of size d_{model}. If the encoders are stacked, the embedding only happens in the bottom-most encoder. Since we process the words in parallel we lose positional information, so by using positional encoding we keep this information. The positional encoding has the same dimension d_{model} as the embeddings, so that the two can be summed. There are learned and fixed positional encodings; the authors used sine and cosine functions of different frequencies.

Each layer has two sub-layers:

- A multi-head self attention mechanism.
- A fully connected feed-forward network. The exact same feedforward network is independently applied to each position (shared parameters).

There is also a residual connection around each of the two sub-layers, followed by layer normalization. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension d_{model}.

Transfromer decoder The decoder is also composed of a stack of N identical layers. In addition of the two sub-layers of the encoder, the decoder has a third sub-layer, which performs multi-head attention over the output of the encoder stack (cross-attention). As in the encoder there are residual connections and layer normalization.

The self-attention sub-layer in the decoder stack is modified so as to **prevent positions from attending to subsequent positions**. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Self-Attention The first setp in calculating self-attention is to create three vectors from each of the embedded words:

- a Query vector.
- a Key vector.
- a Value vector.

These vectors are created by multiplying the embeddings by three matrices learned during the training process. Their dimensionality is $d_k = \frac{d_{\text{model}}}{h}$, while the embedding and encoder input/output vectors have dimensionality of d_{model}.

The matrices are used to compute the self-attention score.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

As we can see there is a little normalization done with $\sqrt{d_k}$, to try to avoid having the results of the softmax squeezed too much into a single value (one hot). The scores of the self-attention determine how much focus to place on other parts of the input sentence as we encode a word at a certain position.

MultiHead Attention The purpose of multi-head attention is to linearly project the queries, keys and values h times with different, learned linear projections. On each of these projected versions of queries, keys, and values we perform the attention function in parallel. The outputs of different heads are then concatenated and once again projected. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

Cross-Attention in the cross attention layers, the queries come from the previous decoder layer (masked self-attention), and the keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. It's a very powerful mechanism used in modern generative models to introduce variable lengths conditions.

3.3.2 Vision Transformer (ViT)

We split an image into patches and provide the sequence of linear embeddings of these patches as an input to a Transformer. The image patches are treated the same way as tokens in an NLP application.

As an alternative to raw image patches, the input sequence can be formed from feature maps of a CNN. In this hybrid model, the patch embedding projection is applied to patches extracted from a CNN feature map.

To handle $2D$ images, we reshape the image $x \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened $2D$ patches $x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$.

- (H, W) is the resolution of the original image.
- C is the number of channels.
- (P, P) is the resolution of each image patch.
- $N = \frac{HW}{P^2}$ is the resulting number of patches, which also serves as the effective input sequence length for the transformer.

The transformer uses constant latent vector size d_{model} through all of its layers, so we flatten the patches and map to d_{model} dimensions with a trainable linear projection. We also prepend a learnable embedding to the sequence of embedded patches, whose state at the output of the transfromer encoder serves as the image representation. The classification head is implemented by a MLP with one hidden layer. To retain positional information the ViT uses learnable 1D position embeddings.

At the start, the ViT worked very bad. This is because transformers lack some of the inductive biases inherent to CNNs, such as translation equivariance and locality, and therefore do not generalize well when trained on insufficient amounts of data. The convolution in combination with max pooling/striding makes CNNs approximately invariant to translation. When a module is translation invariant, it means that if we apply translation transformation on the input image the output of the module won't change. To solve the inductive bias they gave the network so much data that the classes appear anywhere.

3.4 Object Detection

Previously, we have done object detection with SIFT, but we were just able to say if the object was in the image, not where it was. Now we want to do detection by putting a bounding box around the object. We do

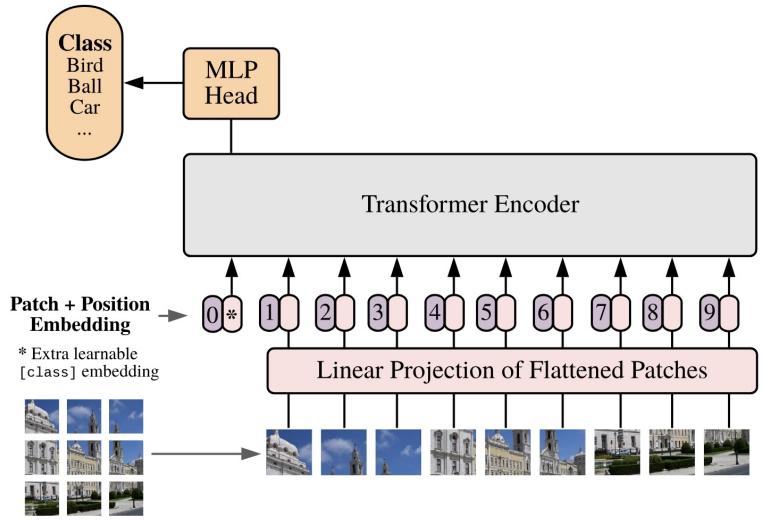


Figure 61: Vision transformer model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence.

this by returning a set of quadruples: $[x, y, h, w, o]_{i=0}^{K}$, where we have the box position, width, height, and class of the output.

The main challenges are the different lengths of the outputs (since we can detect from 0 to K images), the fact that the outputs have both categorical and spatial information, and the fact that the images are usually processed at higher resolution than neural networks for image classification.

3.4.1 Viola-Jones Object Detector

It's a general purpose object detection framework, but it has been mainly applied to faces. We will now see the three main innovations of the algorithm: the AdaBoost algorithm, cascade, and the use of integral images.

AdaBoost algorithm A Weak Learner (WL) is a simple classifier whose error is slightly better than random guessing. Boosting is a way to train and build an ensemble of M weak classifiers to obtain a Strong Learner SL. After training a weak learner, the examples are re-weighted in order to emphasize those which were incorrectly classified by the previous weak classifier.

The **AdaBoost** algorithm steps are:

1. Given N training samples $(x^{(i)}, y^{(i)})$, assign equal weight to each training example $w^{(i)} = \frac{1}{N}$.
2. Iterate for $j = 1, ..., M$ weak learners:
   (a) Fit the best classifier WL_j to the training data by using the current weights.
   (b) Compute the weighted error rate $\epsilon_j = \sum_{i:\, x^{(i)} \text{ is missclassified}} w^{(i)}$.
   (c) Compute $\beta_j = \frac{1-\epsilon_j}{\epsilon_j}$.
   (d) Updates weights $w^{(i)} = w^{(i)}\beta_j$ for wrongly classified examples.
   (e) Re-normalize $w^{(i)}$ to sum to 1.
3. The Strong Learner is given by a weighted majority vote $SL(x) = \sum_j \ln \beta_j WL_j(x) > 0$, con $\ln \beta_j = \alpha_j$.

The idea is that missclassified examples gain weight, forcing subsequent learners to focus on harder cases.

Haar-like features Weak classifiers used to detect faces are simple rectangular filters, composed of 2 to 4 rectangles applied at a fixed position within a 24×24 patch. Even with this simple definition there are over 160 k possible filters in a 24×24 patch. AdaBoost is used to select a small subset of the most effective filters.

Dataset The dataset consisted of 4916 hand labeled faces, scaled and aligned to a base resolution of 24×24 pixels. The non-face windows were collected by selecting random sub-windows from a set of 9500 images which did not contain faces.

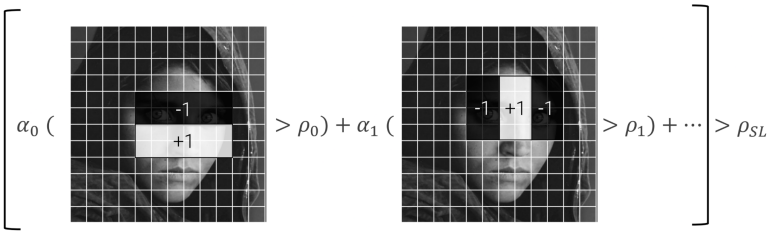


Figure 62: The 2 most effective features selected by AdaBoost on the training set.

Integral images and fast feature computation To speed up the computation of rectangular features, the authors proposed the use of so-called integral images II, where $II(i,j) = \sum_{i' \leq i, j' \leq j} I(i', j')$.

We can compute the value of $II(i,j)$ just by looking at the 3 neighbors: $II(i,j) = II(i,j-1) + II(i-1,j) - II(i-1,j-1) + I(i,j)$, where II are the values in the integral image, and I is the value in the original image.

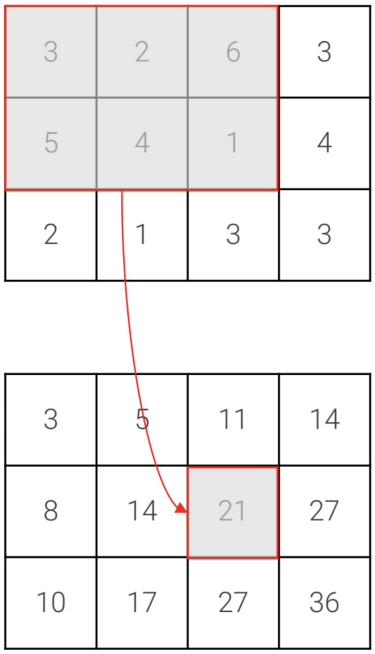


Figure 63: $II(1,2) = II(1,1) + II(0,2) - II(0,1) + I(1,2) = 14 + 11 - 5 + 1$

With integral images we could have an overflow problem if numbers get too big. We can see an example of the calculation in the image 63

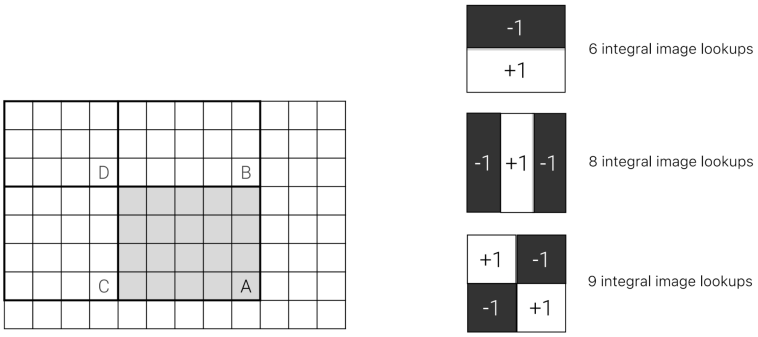


Figure 64: Rectangular filters can be computed in constant time with integral images

Multi-scale sliding window detector At test time, the strong classifier is applied to all spatial locations in the image. Multi-scale detection is necessary, since faces are not necessarily 24×24. To achieve good performance, about 200 features are used to classify each patch. Even if each feature can be computed very fast (thanks to integral images) there are still too many windows in an image to achieve real-time performance.

Faces are far less frequent in an image than background regions. Most of the time is wasted computing a lot of features for background patches. The key idea is then to reject most of the easy background patches with a simpler classifier which can be ran very fast.

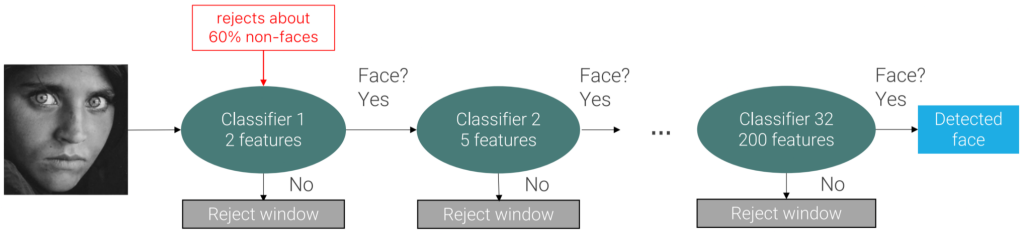


Figure 65: We use more and more features for face detection classifier after classifier

Box overlap There will be several overlapping detections. To check if two boxes overlap we measure the Intersection over Union (IoU) score: $IoU(BB_i, BB_j) = \frac{|BB_i \cap BB_j|}{|BB_i| + |BB_j| - |BB_i \cap BB_j|}$. 0.75 corresponds to a good overlap, 0.90 to a perfect one. To obtain a single detection out of a set of overlapping boxes we perform Non Maxima Suppression of boxes. Non Maxima Suppression considers the highest scoring Bounding Box, eliminates all the boxes with overlap greater than a threshold, and repeats this until all the boxes have been tested.

A detection is considered a True Positive if its overlap with the ground truth BB^{GT} is greater than ρ_{IoU}.

3.4.2 Transfer Learning

If we can assume that only one object is present in the image, object detection simplifies to object localization. Transfer learning is a technique where a model trained on one task is reused (or "transferred") for a different but related task. Instead of training a model from scratch, you start with a pre-trained model and fine-tune it for your specific problem. To solve object detection we can reuse any architecture used in image classification, by adding a regression head for predicting the bounding box.

We can apply a classification CNN as a sliding window detector. We have to add a background class to discard background patches. The problem is that there are too many boxes to try, since we have to try lots of positions with different scales and aspect ratios. The solution is to use region proposals.

3.4.3 Region proposals

Region proposal algorithms inspect the image and attempt to find regions of an image that likely contain an object. The algorithm as a first step oversegments the image into highly uniform regions, and then, **based on similarity scores of color, texture and size** it iteratively aggregates them.

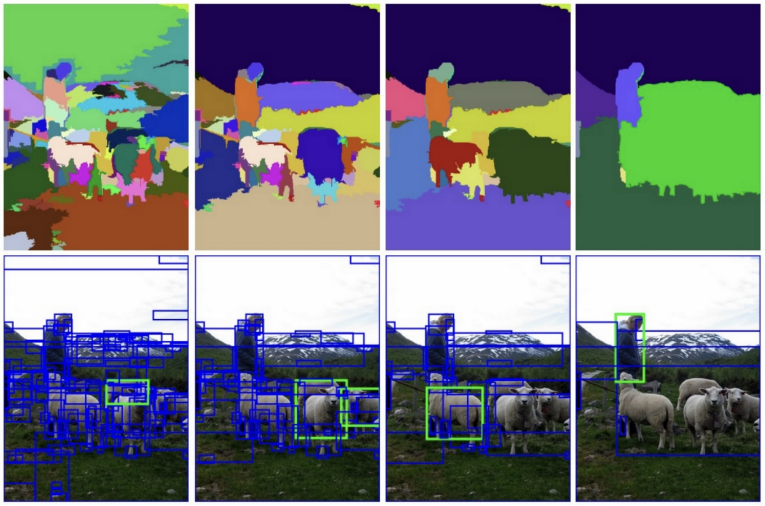


Figure 66: The 2 most similar regions are grouped together and then new similarities are calculated between the resulting region and its neighbors

3.4.4 R-CNN: Region-based CNN

Region-based CNN is the first object detector network. As we can see in 67 we run selective search to get about 2000 proposals, then we anisotropically (not uniformly across all directions) warp these proposals and add some pixels of context into a fixed size, to be able to input them in AlexNet, and for each proposal we get a class and a bounding box correction.

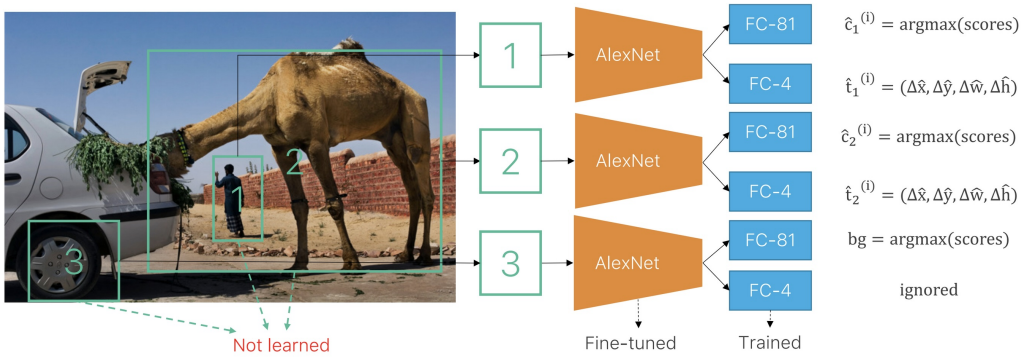


Figure 67: R-CNN network architecture

The problem is that since we have 2 different algorithms (one for the region proposal and one for AlexNet) we can't backpropagate through both of them.

3.4.5 Fast R-CNN

In the Fast R-CNN architecture showed in 68 we can see that the model has been modified to run the first few layers of AlexNet on the original image, apply RoI pooling to crop and warp convolutional features according to proposal, and then run a small per-region network on each region to get the output class and bounding box correction. This results in a faster networks since we don't have to apply 2000 CNN forward passes per image, but we apply most of the CNN before warping.

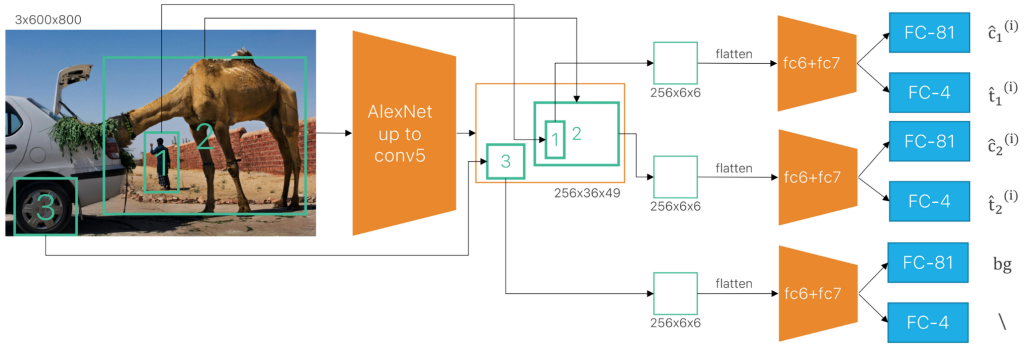


Figure 68: Fast R-CNN network architecture

Fast R-CNN uses the same bounding box correction of R-CNN, but with a smooth L1 loss (equal to L2 near the origin, but smoother elsewhere).

RoI pooling The RoIPool layer converts activations inside Region of Interest, corresponding to rescaled Selective Search regions, into activations with fixed spatial dimensions, which are the ones required by the remaining layers of the network. To do RoI pooling, given a region:

1. Snap the region to the grid.
2. Apply max pooling kernels with approximate size $[H_r/H_o] \times [W_r/W_o]$ and approximate stride $s = [H_r/H_o] \times [W_r/W_o]$.

3.4.6 Faster R-CNN

In Fast R-CNN proposals are not learned, so the selective search is slow. In Faster R-CNN they introduced a RPN (Region Proposal Network), which learns to predict the proposal box and the objectness score.

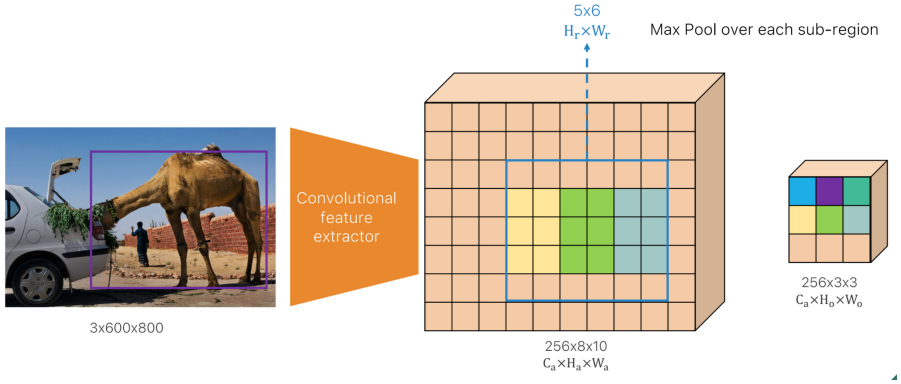


Figure 69: RoI pooling.

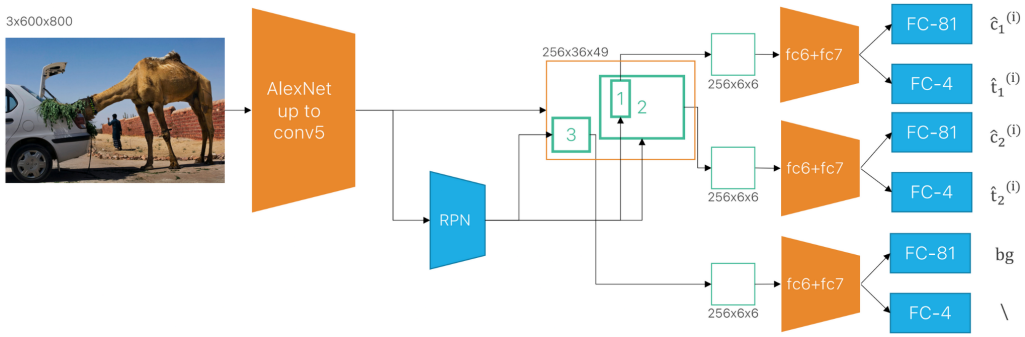


Figure 70: Faster R-CNN network architecture.

Region Proposal Network Region Proposal Networks are applied to a small 3×3 window, which is approximately an object sized window in the original image and doesn't correspond to what we see in 71, predicts the objectness and proposal bounding box.

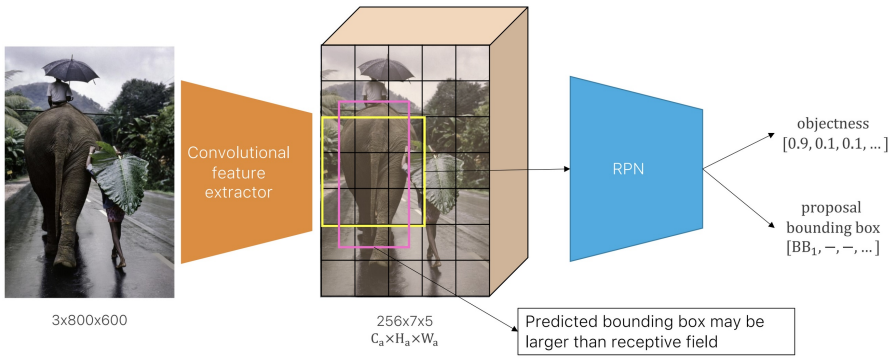


Figure 71: Region Proposal Network architecture.

When the objectness score is low, the localization head produces its output, but it's meaningless and it will be ignored.

Region Proposal Network with Anchors The process of generating object proposals can be simplified. Instead of predicting proposals from scratch, it is easier to refine a set of initial reference boxes, similar to the approach used in R-CNN.

Since objects vary in scale and aspect ratio, we use multiple anchors at each spatial location, each with a different combination of scale and aspect ratio. The RPN predicts k objectness scores and k bounding box refinements for the k anchors at each location.

During training, for a given image and its ground-truth bounding boxes, an anchor is labeled as background if it has low Intersection over Union (IoU) with all ground-truth boxes in the image.

In standard Faster R-CNN, the RPN operates on the final activation layer of the feature extractor. This feature map contains rich semantic information due to its depth in the network, but has low spatial resolution. As a result, even with small-scale anchors, **the RPN might fail to detect objects that are smaller than**

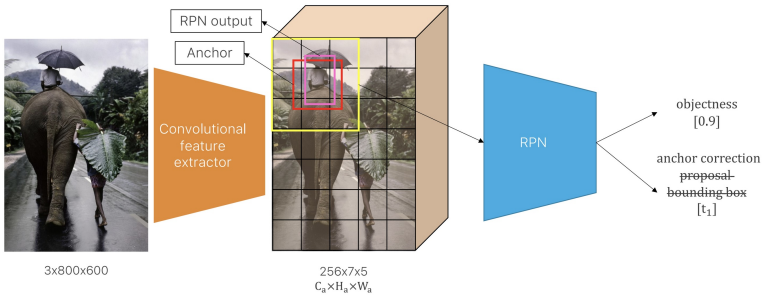


Figure 72: An anchor is a reference box with fixed scale and aspect ratio.

the grid size of the feature map.

3.4.7 Feature Pyramid Network (FPN)

To do object detection at multiple scales, the naive approach is to obtain feature pyramids with CNNs, by running a CNN at each scale of the image, and then perform detection on each activation. This is very bad for inference and training times.

A better approach is to give as input to the Region Proposal Network different stages of a CNN. This is done because CNNs produce a pyramid of features, with different semantic qualities at different depths. But different stages of a CNN have different input sizes (more channels as we go deeper in the network).

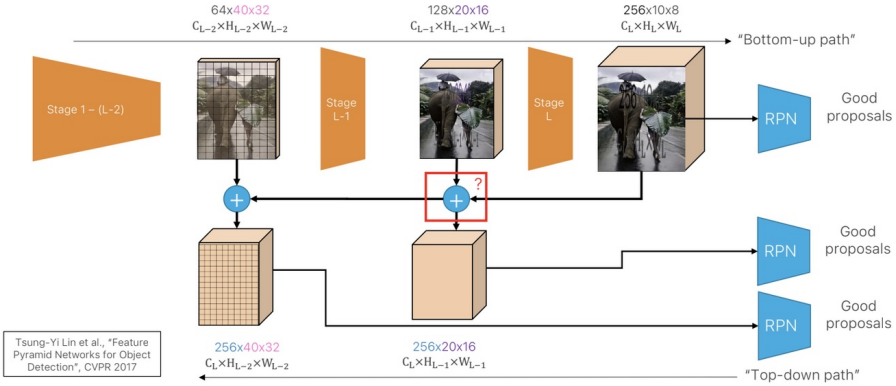


Figure 73: Feature Pyramid Network architecture.

Feature Pyramid Networks overcome this problem by merging coarser but higher level features with less effective but more spatially localized features. This has a limited computational overhead.

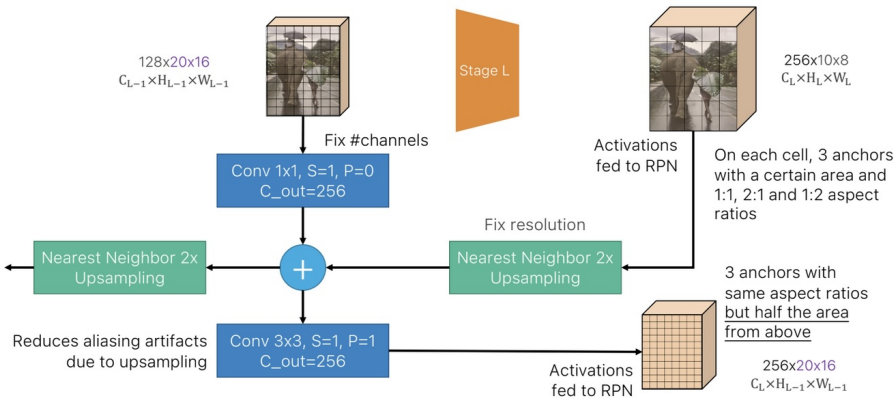


Figure 74: FPN top down path and lateral connections.

In 74 we can see how the FPN sums together different size pictures. To fix the resolution we use Nearest Neighbor upsampling. To fix the number of channels we use 1×1 convolutions.

3.4.8 Faster R-CNN with FPN

We can use the FPN for the detection stage. We use the FPN as a feature extractor which provides a pyramid of activations. Then we can do the usual processing with the RPN and the per-region MLP.

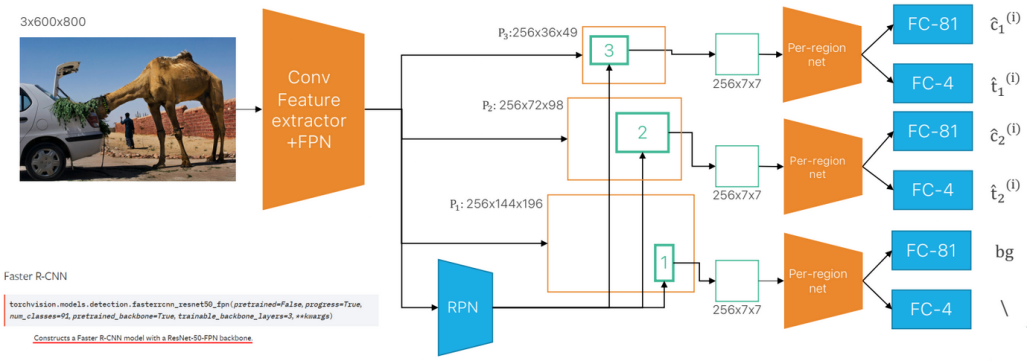


Figure 75: Faster R-CNN with FPN architecture; as we can see we use the FPN together with the convolutional feature extractor.

Introducing the FPN enables better model accuracy but gives no speedup, since we didn't change anything with the broader architecture.

3.4.9 One Stage detectors

As wee can see from the image 75, we have two main stages in the network architecture. The first stage runs the expensive backbone feature extractor with the FPN on the full image, and the RPN generates the proposals for the bounding boxes. This is done one time per each image. The second stage, which is ran once per proposal, has a RoI pool, and a per-region classification and correction.

Instead of a RPN we could try to do all the job by using anchors, since the only things they are missing is learning the anchor dimensions and the class prediction.

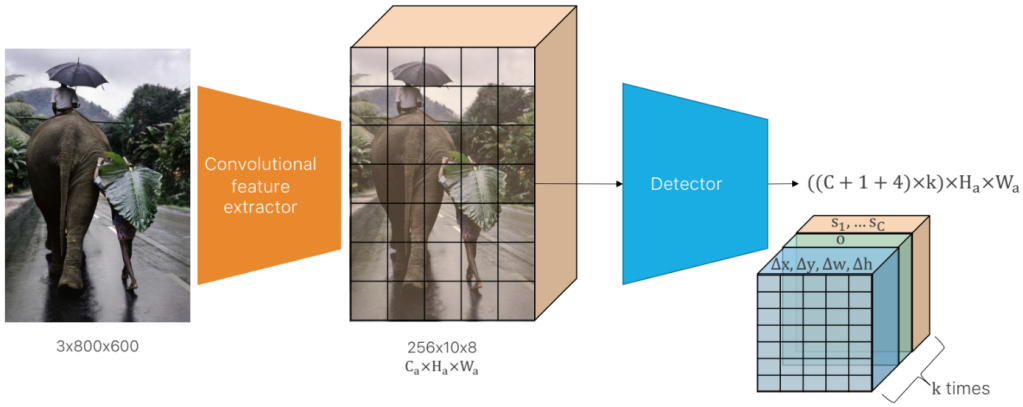


Figure 76: in one-stage detectors we can stack the outputs in a combined single tensor with the appropriate number of channels.

3.4.10 SSD: Single Shot MultiBox Detector

SSD applies multiple detection heads, implemented as 3×3 convolutions, across several feature maps. Each detector outputs $k_s \times (C + 1 + 4)$ channels, where k_s is the number of anchors per location at that scale, C is the number of classes, 1 accounts for the background class, and 4 corresponds to bounding box coordinates.

These detectors are applied both to intermediate feature maps from the backbone network (e.g., VGG) and to additional feature maps generated by stacking extra convolutional layers beyond the backbone.

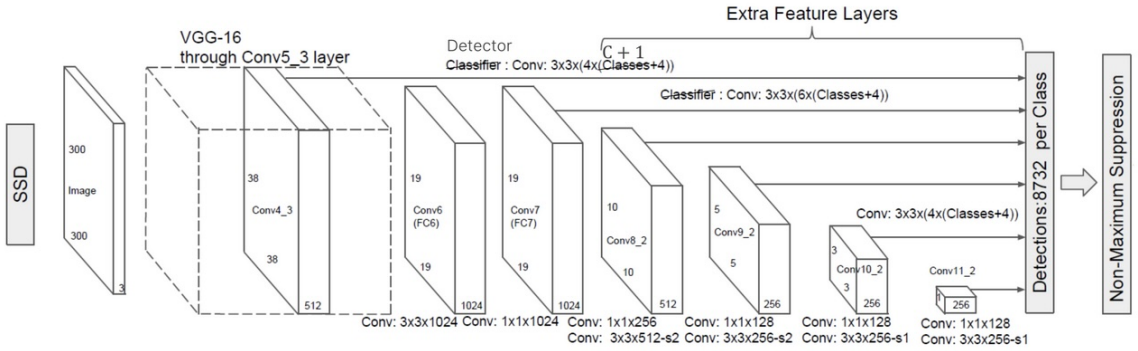


Figure 77: Single Shot MultiBox Detector architecture

3.4.11 YOLOv3

YOLOv3 employs a custom backbone called DarkNet-53, which balances classification accuracy and inference speed. It performs detections at multiple scales by concatenating feature maps from different stages, similar to an FPN (Feature Pyramid Network) but using concatenation instead of addition.

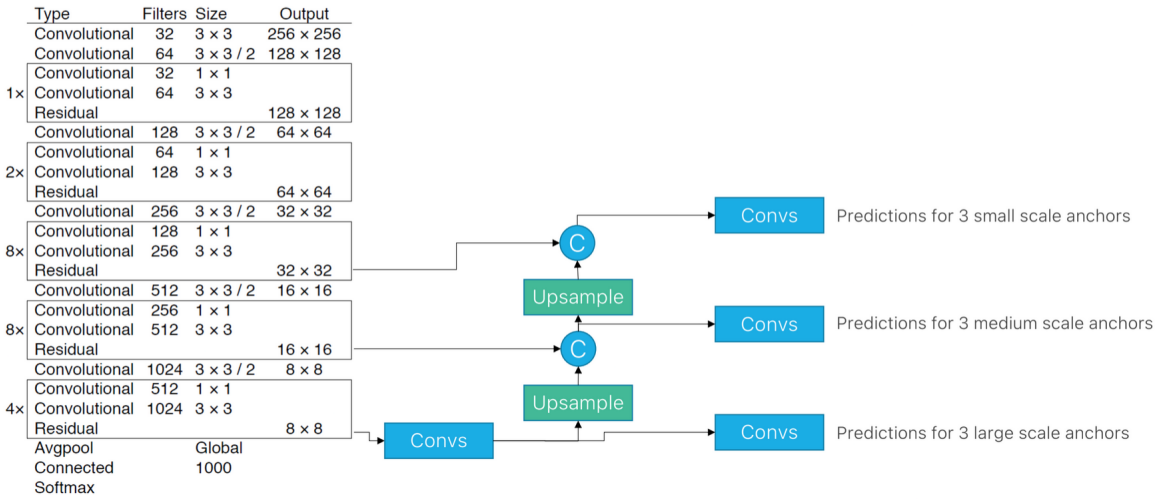


Figure 78: YOLOv3 architecture

Anchor boxes (sizes and aspect ratios) are determined via k-means clustering on the ground-truth bounding boxes of the training dataset. By initializing predictions close to these clustered anchors, the network can converge more quickly. In practice, the resulting anchors tend to be similar across various datasets.

3.4.12 RetinaNet

RetinaNet is a one-stage detector built on top of a standard ResNet backbone with an integrated Feature Pyramid Network (FPN). It has separate classification and regression "heads," each composed of stacked 3×3 convolutional layers. Unlike RPN (Region Proposal Network), these heads do not share parameters, since classification and bounding-box regression require different feature representations.

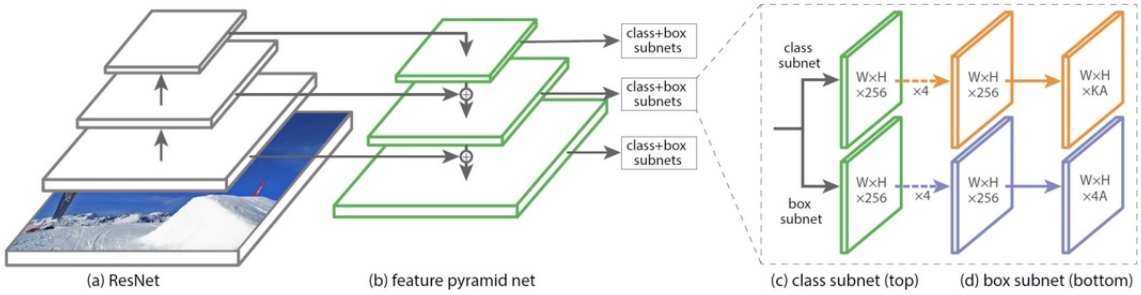


Figure 79: RetinaNet architecture

The class-imbalance challenge In object detection, the number of negative (background) anchors far exceeds the number of positive (object) anchors. This imbalance causes two main issues:

1. "Easy" negatives (anchors with low objectness scores) can dominate the loss, making it hard for the model to learn to discriminate between hard negatives and true positives.
2. Most randomly sampled mini-batches will contain predominantly easy negatives, providing little useful gradient signal and slowing down training.

Two-stage detectors mitigate this by sampling "hard" negatives: only high-scoring region proposals are passed to the second stage. By contrast, one-stage detectors must train on all anchors, including many easy negatives.

Focal loss To address this, RetinaNet uses focal loss, which down-weights easy examples and focuses training on hard examples. For a binary classification task, the standard cross-entropy (CE) loss is

$$\mathcal{L}_{\mathrm{CE}}(p, y) \;=\; -\big[y \ln(p) + (1-y)\ln(1-p)\big],$$

where p is the model's estimated probability that $y = 1$. Defining

$$p_t = \begin{cases} p, & \text{if } y = 1, \\ 1-p, & \text{if } y = 0, \end{cases}$$

we can write $\mathcal{L}_{\mathrm{CE}}(p, y) = -\ln(p_t)$.

The focal loss introduces a modulating factor $(1-p_t)^\gamma$:

$$\mathcal{L}_{\mathrm{FL}}(p_t) \;=\; -(1-p_t)^\gamma \ln(p_t),$$

where $\gamma \geq 0$ is a focusing parameter (commonly $\gamma = 2$). When p_t is large (an "easy" example), $(1-p_t)^\gamma$ becomes small, reducing the loss. For example:

- If $p_t = 0.9$, then $(1-p_t)^\gamma = (0.1)^2 = 0.01$, so the focal loss is $100\times$ smaller than the CE loss.
- If $p_t = 0.6$, then $(1-p_t)^\gamma = (0.4)^2 = 0.16$.
- If $p_t = 0.1$, then $(1-p_t)^\gamma = (0.9)^2 = 0.81$, making the focal loss close to the CE loss.

Class weighting In addition to focal loss, RetinaNet applies standard class weights to balance the relative importance of positive and negative examples. Class weights ensure that errors on the minority class (foreground) receive higher weight, while focal loss focuses learning on hard examples within each class. Combining both mechanisms yields more stable and effective training.

3.4.13 Key differences between YOLOv3 and RetinaNet

- **Backbone and feature fusion**:
  - YOLOv3 uses DarkNet-53 and concatenates feature maps from different stages for multi-scale predictions.
  - RetinaNet uses ResNet with an explicit FPN, which sums feature maps across levels and applies separate heads at each pyramid level.
- **Detection heads**:
  - YOLOv3 predicts boxes and class scores on a grid of three scales directly from concatenated features.
  - RetinaNet has separate stacks of convolutional layers for classification and regression on each FPN level.
- **Loss functions**:
  - YOLOv3 uses binary cross-entropy (BCE) for classification and mean squared error (or IoU-based loss) for bounding-box regression.
  - RetinaNet uses focal loss for classification (to address class imbalance) and smooth L_1 loss for bounding-box regression.
- **Anchor initialization**:
  - YOLOv3 learns anchoring by running k-means clustering on ground-truth boxes of the dataset.
  - RetinaNet uses a fixed set of anchors defined by scale and aspect ratio at each FPN level (often chosen based on prior knowledge).

3.4.14 Multi-label classification

In YOLOv3, class prediction for each bounding box is treated as a multi-label classification task, rather than a multi-class one: a single box can be assigned one or more of the C classes, which are not assumed to be mutually exclusive.

As a result, YOLOv3 does not apply a softmax function to produce a single probability distribution over classes. Instead, it uses C independent sigmoid activations—one per class—to estimate the probability that the box belongs to each class individually.

The classification loss for each box is then the sum of C binary cross-entropy losses, one for each class.

Because the background is implicitly represented by a ground-truth vector of all zeros, there is no need to explicitly introduce a background class. At inference time, a box is considered background if all class scores fall below the detection threshold.

3.4.15 CenterNet

Anchors-based detectors, either two or one stage ones, have some limits:

- Anchors are a brute force approach to detection: we are enumerating a subset of all possible boxes, which is very inefficient. Moreover, the more anchors the better, but it's not clear what the best way to use a subset is.
- We obtain a lot of duplicated entries for an object, which must be post-processed with NMS, so the detector is in practice not end-to-end differentiable.
- Assignment of anchors to ground truths for training is based on manually selected thresholds and hand-crafted rules.

To overcome these limits, CenterNet proposed to represent objects as points (which is also the name of the paper which introduced the network), and regress their size or other properties. It's like the keypoints in Difference of Gaussians, but at a higher level, since the keypoints should be the center of my object.

Given an image of size $3 \times H \times W$, the aim of the network is to produce a heatmap $\hat{Y}$ with values $\in [0, 1]$ and size $C \times \frac{H}{R} \times \frac{W}{R}$ with output stride R. To recover the discretization error caused by the output stride, the network also predicts an offset for each center point. Finally, it also predicts the bounding box size $\hat{S}$ of size $2 \times \frac{H}{R} \times \frac{W}{R}$.

The backbone used to produce the convolutional features are fully convolutional encoder-decoder architectures devised for keypoint detection or semantic segmentation. All the 3 outputs are predicted from the last activation with a dedicated head.

Each ground truth keypoint (object center) $p = (x_p, y_p)$ of class c, is projected in the lower-resolution output heatmap $\tilde{p} = \lfloor \frac{p}{R} \rfloor$. Then, the target heatmap for its class Y_C is set to 1 at $\tilde{p}$ and updated with an unnormalized Gaussian kernel. Points can be seen as a special case of anchor: a single, shape-agnostic, anchor. Detection is performed by finding local maxima in a heatmap, with one box per object (so without using NMS), and based solely on location, not box overlap.

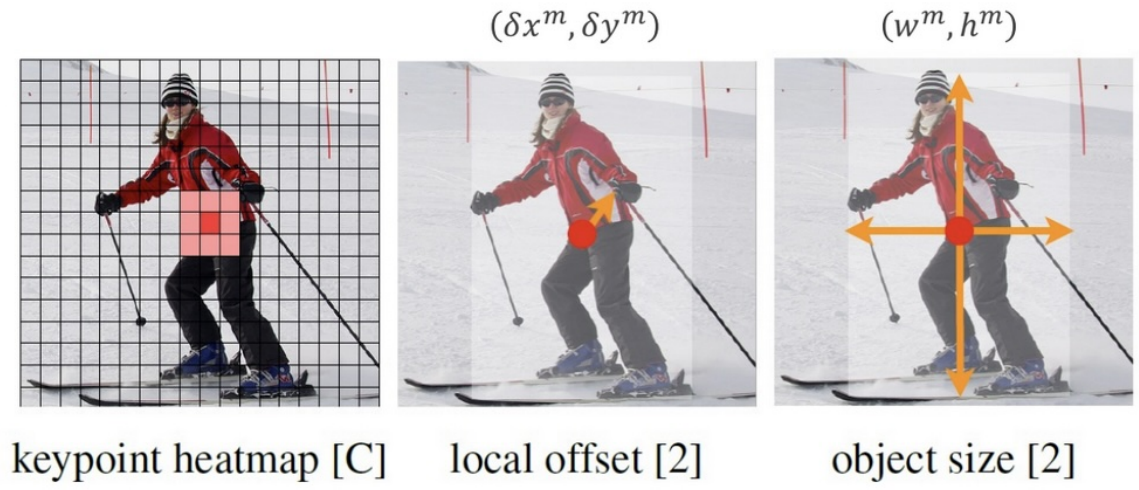


Figure 80: At inference time, given a spatial local maxima in the channel $\hat{Y_C}$ at position (x^m, y^m), the box centered at $(x^m + \delta x^m, x^m + \delta y^m)$ of size (w^m, h^m) and class c is detected, without any further post-processing.

3.5 Image segmentation

In image segmentation we need to do per-pixel labelling, which is a costly task, since the input is the RGB image of size $W \times H$ and the output is **a category c_{uv} for each pixel** $p = (u, v)$. With $c_{u,v} \in [1, ..., C]$ which is a fixed list of categories.

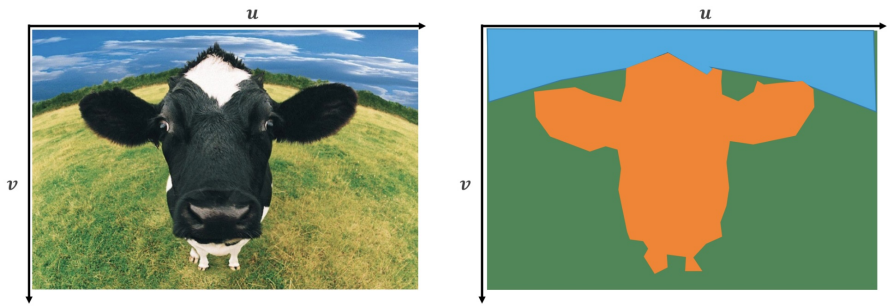


Figure 81: On the right there is the output of the image segmentation task

We can try to use synthetic data to harvest the labels, i.e. with a videogame engine, where we exactly know what the objects are and where they are on the screen. But this has the problem of domain shift (we can't learn to do segmentation on GTA V and then try to apply the same network on the real world).

3.5.1 Generalized IoU and other measures

The intersection over union can be generalized to segmentation masks by counting the pixels. For a class $c = 1, ..., C$ we can define the Intersection over Union for the class as $IoU_C = \frac{\text{area of intersection}}{\text{area of union}}$. To compute the mean IoU score for a dataset, we average IoU_C over classes. $mIoU = \frac{1}{C} \sum_{C=1}^{C} IoU_C$.

The $mIoU$ is the main measure to rank semantic segmentation algorithms. Other measures are:

- Pixel accuracy: the fraction of pixels correctly classified $\frac{\sum_C TP_C}{\#\text{ pixels in the dataset}}$. This has the problem of being biased towards the largest object.
- Mean accuracy: the average of the accuracy for each class $\frac{1}{C} \sum_C \frac{TP_C}{\#\text{ pixels of class } C \text{ in the dataset}}$.
- Frequency weighted IoU: weighted average of IoU_C for each class, with weights given by the frequency of a class in the dataset $\sum_C \frac{\#\text{ pixels of class } C}{\#\text{ pixels in the dataset}} IoU_C$.

3.5.2 Slow R-CNN for segmentation

We can apply the same ideas used in R-CNN: we slide the window at all possible positions. There are no proposals, since we must process each pixel, which is even slower than R-CNN for detection.

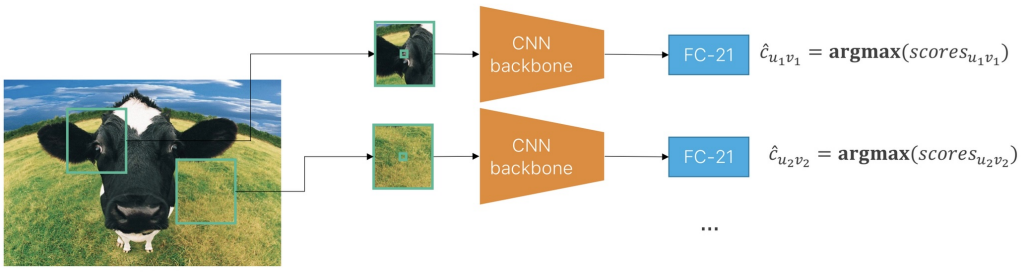


Figure 82: Network architecture for Slow R-CNN for segmentation

As the loss we use the sum of the standard multi-class loss over all pixels:

$$L(\theta, x^{(i)}, y^{(i)}) = \sum_{u,v} CE\left(\text{softmax}(scores_{u,v}), \mathbf{1}(c_{u,v})\right)$$

With $\mathbf{1}$ which is the one-hot encoded ground truth class label for the pixel (u, v).

The idea is that, as we did for detection, we can use a convolutional feature extractor and process the image through a CNN to get some activations and from that compute our output. We have to go from the activations of the extractor to an output that has as many channels as the number of classes and the same resolution as the input.

One way to perform the upsampling can be to use standard, not-learned, image processing operators (like bilinear interpolation).

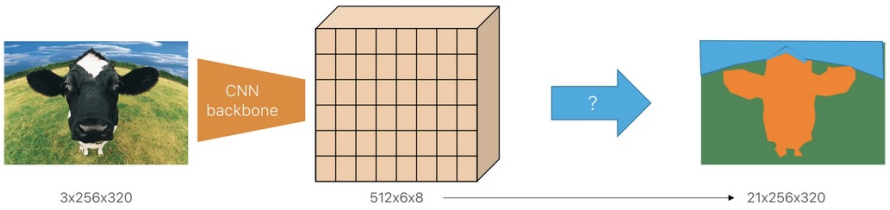


Figure 83: We have to go from 512 channels to 21 channels with a 1×1 conv. We have to go from a resolution of 6×8 to the original resolution of 256×320.

3.5.3 FCN-32s

The FCN-32s (Fully Convolutional Network) is built on top of a CNN backbone. It uses the deepest feature map of the CNN, and then applies a 1×1 convolution to reduce the number of channels to the number of classes. Then the network uses a single upsampling step by a factor of 32 (hence the name 32s) to bring it back to the input image size.

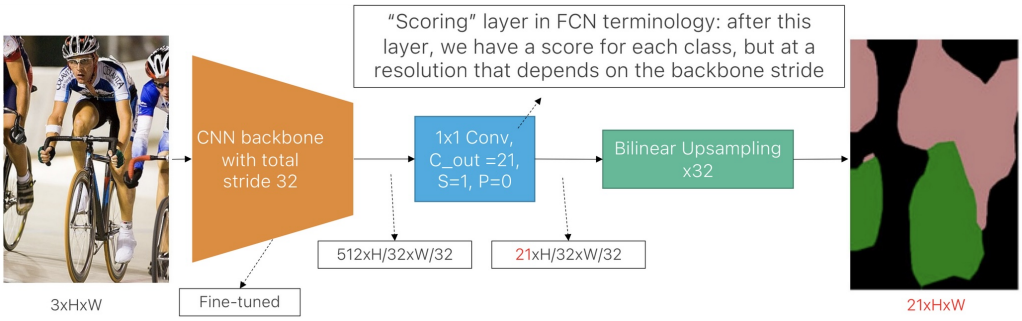


Figure 84: Network architecture for FCN-32s

The problem is that without learning a non-linear upsampling transformation, we can only uniformly spread the coarse information in the final convolutional activation, obtaining very coarse masks. The solution is to upsample multiple activations at different resolutions (like with FPN).

3.5.4 FCN-16s

The FCN-16s adds additional connections between the output and the internal layers, which are referred to as skips. This results to better spatial detail, with more accurate segmentation than FCN-32s.

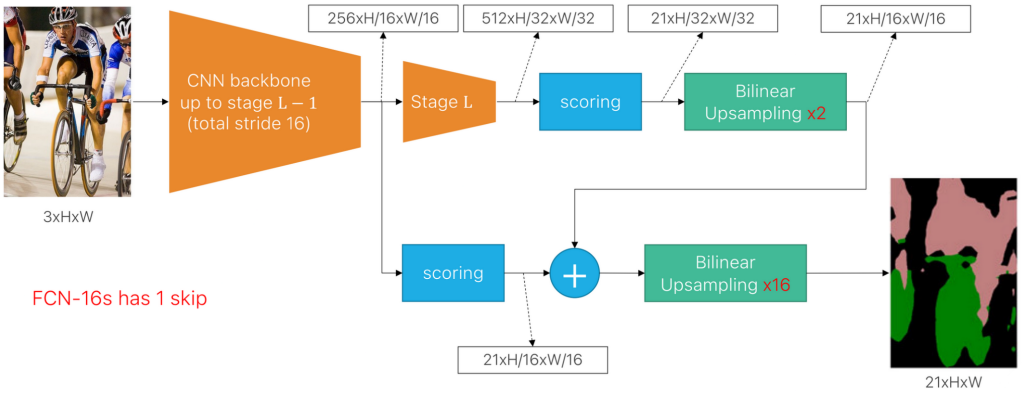


Figure 85: Network architecture for FCN-16s

3.5.5 FCN-8s

In the FCN-8s we can consider one more intermediate layer (2 skip connections) with respect to the FCN-16s.

With the VGG backbone, no more improvements were found after predicting from stride 8 activations (2 skip connections).

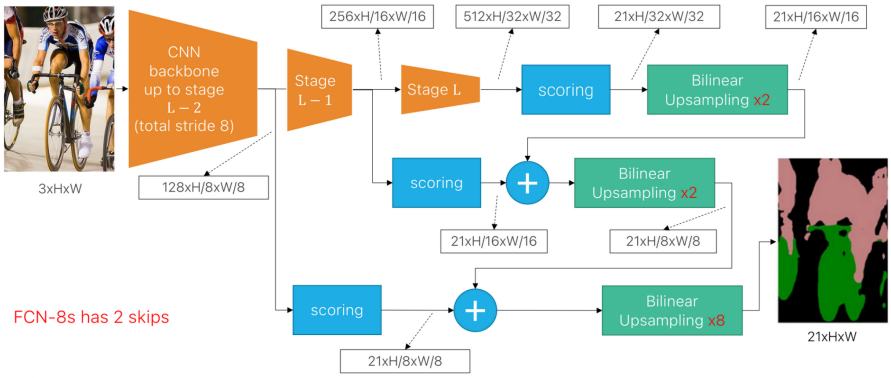


Figure 86: Network architecture for FCN-8s

3.5.6 Transposed Convolutions

Standard convolutions with stride $s > 1$ perform learnable downsampling by skipping $s - 1$ input positions between successive kernel applications. To invert this process and learn upsampling, we can use *transposed convolutions*, sometimes also called *fractionally strided convolutions* or *upconvolutions*.

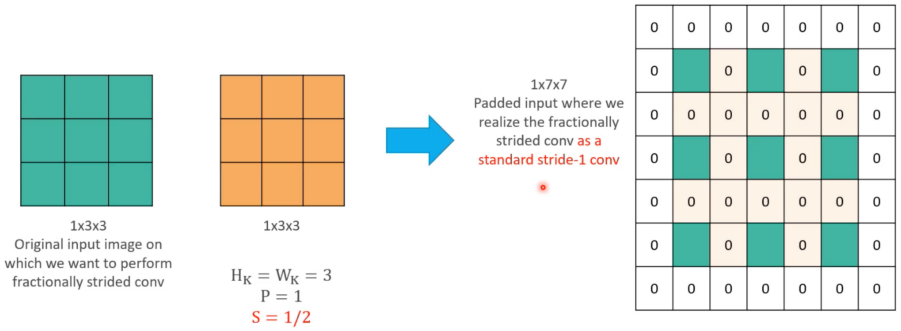


Figure 87: Inserting zeros between input pixels (fractional stride) yields a larger feature map before convolution.

Conceptually, one can imagine upsampling by first "expanding" the input: we insert $s - 1$ zeros between each pair of original pixels (both horizontally and vertically), producing a larger intermediate grid, and then apply a standard convolution with stride 1. For example, inserting zeros around a 3×3 input and convolving with a 3×3 kernel can produce a 5×5 output (Figure 87).

Rather than explicitly inserting zeros, a transposed convolution implements the same computation by *moving the kernel over the output grid*. With stride s, the kernel is shifted by s positions on the output for each move of one position on the input. Each input pixel "casts" a weighted patch (given by the kernel) onto the output grid, and overlapping patches are summed. This reconstructs exactly the same enlarged feature map as the zero-insertion view, but in a single learnable layer.

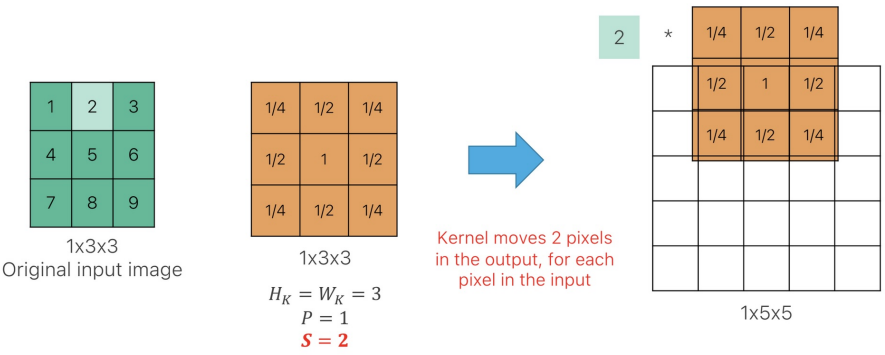


Figure 88: Each input pixel is multiplied by the kernel and "sprayed" onto the output grid with stride 2; overlaps sum to form the final activation.

By learning the kernel weights, transposed convolutions perform upsampling in a way that can adapt to the data. They are widely used in generative and segmentation architectures to increase spatial resolution.

While they can introduce checkerboard artifacts in some settings, these artifacts are usually not problematic in well-designed segmentation networks.

3.5.7 U-net

U-net was one of the first networks used for segmentation. They used transposed convolutions to do upsampling, and skip connections which combine features from the encoder to features from the decoder. Skip connections use concatenation instead of summation. The encoder compresses in a semantically rich representation the input, and the decoder takes the latent representation and goes back to either the original input or a segmentation map.

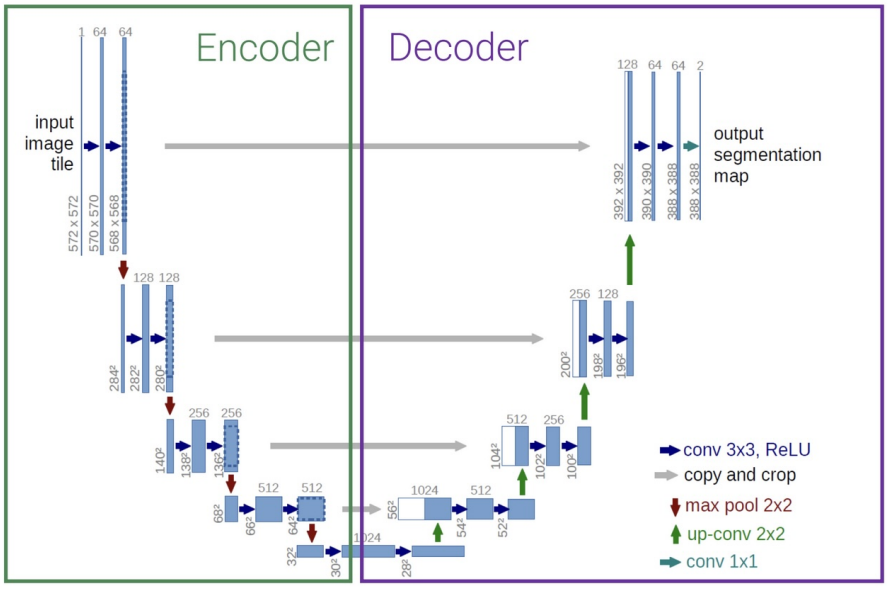


Figure 89: The U-net network architecture.

The operations are mainly three:

- convolutions between features of the same dimensions.
- downsampling with pooling.
- upsampling with transposed convolutions.

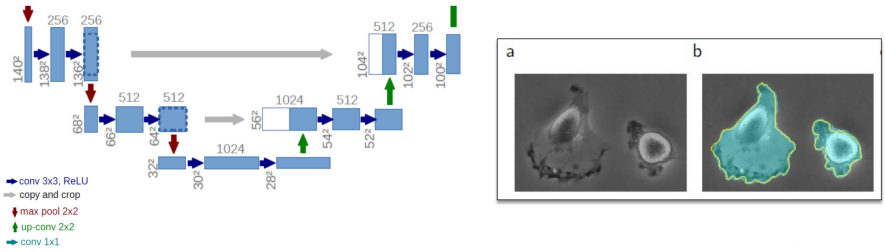


Figure 90: A zoom of the U-net middle layers

3.5.8 Dilated Convolutions

At later stages of a CNN, feature maps become semantically rich because each position captures information from a large receptive field. Earlier layers, however, have smaller receptive fields and therefore less semantic understanding.

Dilated convolutions were introduced to expand the receptive field without increasing computational cost or reducing resolution. This is achieved by inserting gaps (zeros) between kernel elements, allowing the convolution to cover a wider area without increasing the number of parameters.

In ResNet, each stage consists of convolutional blocks followed by batch normalization. Typically, the first block in each stage reduces the spatial resolution (via striding) and doubles the number of channels. Instead, with dilated convolutions, the stride is kept the same, but the dilation rate is increased—allowing the resolution to be preserved across blocks within the same stage.

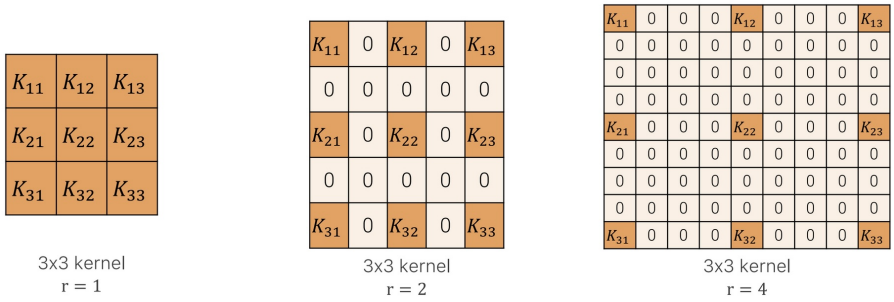


Figure 91: A standard 3×3 kernel has a dilation rate of 1. Increasing the dilation rate to 2 inserts a layer of zeros between kernel elements, expanding the receptive field.

This results in larger feature maps and higher memory usage since the spatial dimensions remain large and the number of channels still increases. However, this is beneficial for tasks like semantic segmentation, where preserving spatial detail is important.

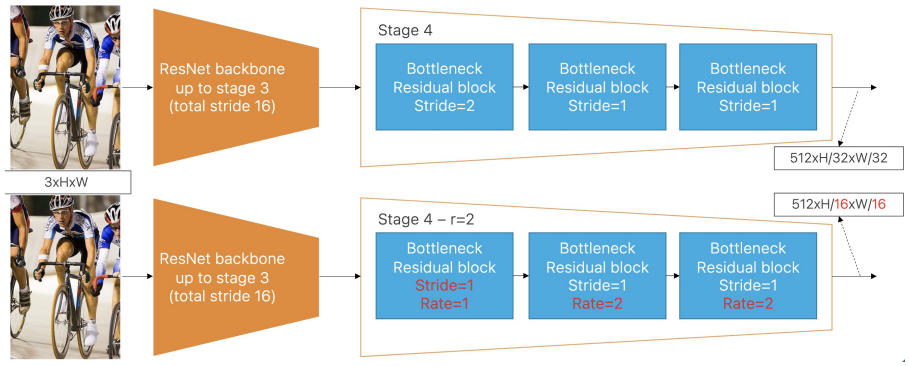


Figure 92: Standard ResNet vs. ResNet with dilated convolutions

Dilated convolutions were applied to specific stages of the ResNet backbone to selectively expand the receptive field. Applying them to every stage would increase memory usage significantly, since the input size remains unchanged and the channel width continues to grow.

3.5.9 DeepLab

DeepLab, a reference model for semantic segmentation, uses ResNet with dilated (atrous) convolutions as its backbone to control the resolution of the output feature maps.

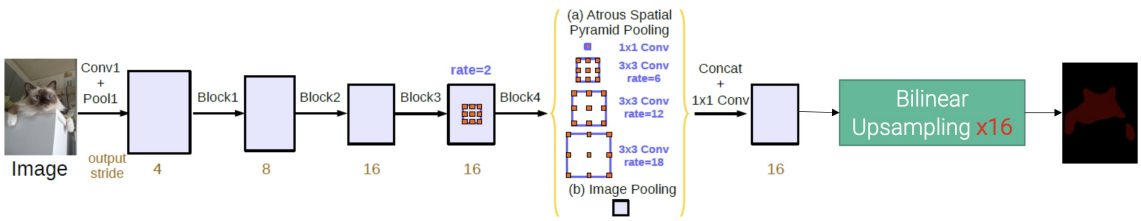


Figure 93: DeepLab v3 network architecture

To address the challenge of detecting objects at multiple scales, DeepLab introduces a module called Atrous Spatial Pyramid Pooling (ASPP), which builds on the concept of spatial pyramid pooling using dilated convolutions.

This idea originated in DeepLab v2, inspired by traditional spatial pyramid pooling. Unlike global average pooling, spatial pyramid pooling preserves spatial structure while aggregating multi-scale context. DeepLab emulates this property using atrous convolutions with different dilation rates.

In DeepLab v2, the ASPP module applies several parallel 3×3 atrous convolutions with increasing dilation rates. These outputs are then concatenated and passed through a 1×1 convolution to produce the final score map. This works effectively since the spatial resolution is preserved throughout the process.

However, as the dilation rate increases, the kernel becomes increasingly sparse—resulting in fewer effective weights being applied. It was observed that a dilation rate of 24 was too large: in many cases, only one weight in the 3×3 kernel was applied, diminishing its effectiveness. As a result, such high dilation rates were discarded.

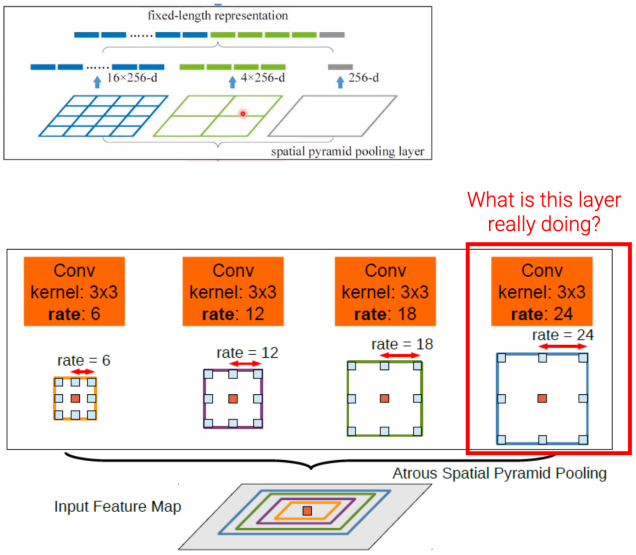


Figure 94: DeepLab v2 network architecture

In DeepLab v3, a global context feature was also added by applying global average pooling over the entire feature map, which further enhances the model's ability to capture large-scale semantic information.

3.5.10 Instance segmentation

Semantic segmentation separates different classes at the pixel level, but doesn't separate different instances of the same class. Object detection separates instances but provides only a crude approximation of the instance shape (the bounding box). The task of instance segmentation lays at the intersection of the two. It can be defined as the task of:

1. Detecting all instances of the objects of interest in an image and classifying them
2. Segmenting them from the background at the pixel level.

Figure 95: difference between object detection, instance segmentation and semantic segmentation.

3.5.11 Mask R-CNN

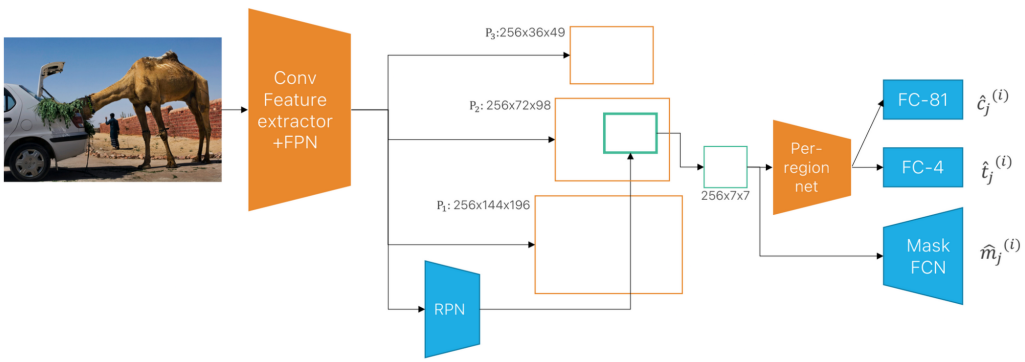


Figure 96: Mask R-CNN network architecture

Mask R-CNN modifies Faster R-CNN, which is a 2 stage detector, by adding a convolutional network called Mask FCN which does segmentation on the proposed region. Another change is the RoI align instead of the RoI pool. Remember that feature maps are spatially continuous, but RoIs have floating-point coordinates after scaling, and we need to align them to the grid. The RoI align doesn't snap the RoI to the grid, but it divides into equally sized subregions, it samples feature values at a regular grid of points within each RoI cell with bilinear interpolation, and it pools samples feature values in each sub-region.

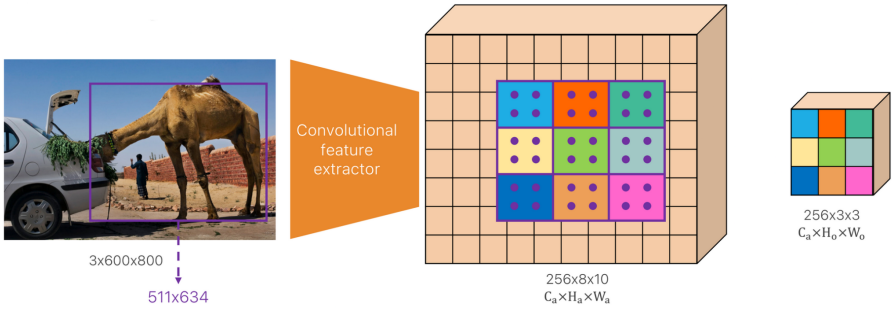


Figure 97: RoI align

3.6 Metric Learning

By removing the classification head of an image classification network we can compute a low-dimensional representation (a low dimensional embedding) of the input images. Performing nearest neighbor search on such embedding vectors is very effective: we get semantically similar images as neighbors.

3.6.1 Face recognition

Given a query face face recognition is a one to many matching problem. A database of faces can have millions of identities, but few images per subject. A match msust be robust to facial expression, changed hair style, aging, glasses, scarfs...

We can't just train a classification network for face images. If we take imagenet as an example, which has 1000 classes and 1.4 M images, we would need 1.4 B images to train if we want to distinguish between 1 M different faces. Face recognition is also an open-world problem, since we could want to add or remove an user. But this requires to throw away the last layer (since we change the output dimension by adding/removing people) and re-train the full network.

The idea is to use transfer learning, which is based on reusing the representation learned once on a reasonably large dataset as feature extractor with a $k-$NN classifier on the embedding to tackle face recognition.

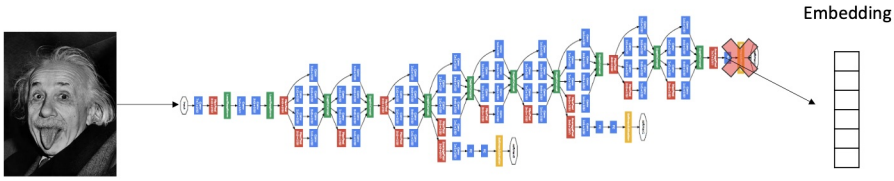


Figure 98: To change the number of images we want to recognize we also have to change the embedding size

Classification embeddings The cross-entropy loss used in classification guides the network to learn high-level and semantically rich embeddings. However, they are an intermediate representation used to classify correctly the input image with the subsequent fully connected linear layer.

In the embedding space the classes are linearly separable. Distances between elements of the same class can be arbitrarily large, and distances between elements of different classes can be arbitrarily small. This is not a great space in which perform nearest neighbor search, since a point can be closer to lots of points of the other classes and not to the ones of the same class.

3.6.2 Face verification

A closely related problem to face recognition is: given two images, confirm that they depict the same person. This is usually solved by learning a similarity function between images and a threshold.

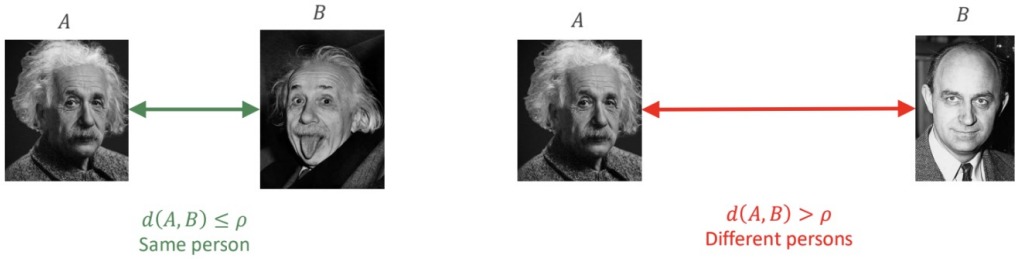


Figure 99: The distance between the images must be under the threshold

3.6.3 Metric Learning

Metric learning, also known as similarity learning, aims to train a model to produce feature embeddings where:

1. The distance between embeddings of the same identity (intra-class distance) is minimized.
2. The distance between embeddings of different identities (inter-class distance) is maximized.

The objective is to learn highly discriminative embeddings:

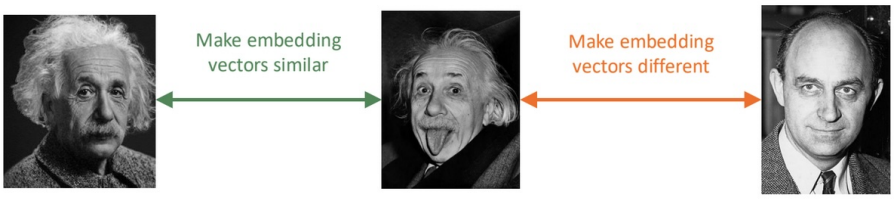


Figure 100: The key concept of metric learning

3.6.4 Siamese Network Training

A Siamese network is designed to compare two input samples. It consists of two identical subnetworks (sharing both architecture and weights) that independently process two inputs, and then compare their outputs to determine their similarity.

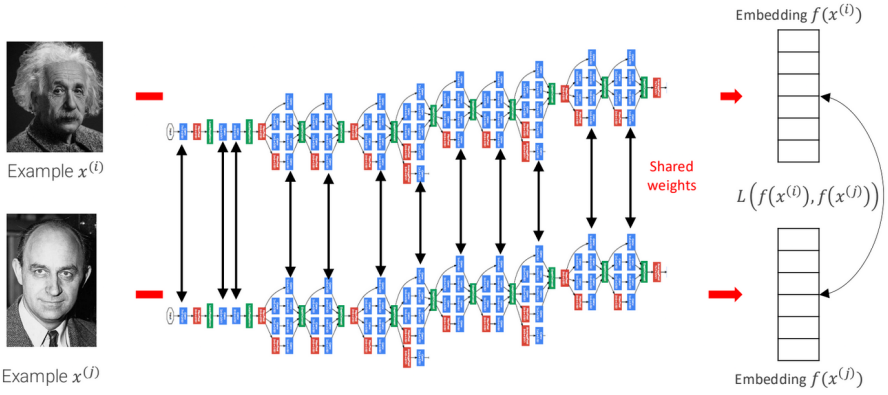


Figure 101: Siamese network architecture

The model is trained by computing a loss function over pairs of examples:

$$L(f(x^{(i)}), f(x^{(j)}))$$

Once trained, the network can embed new images into this learned feature space. A k-nearest neighbor classifier can then be used in this space to classify unseen identities. This approach allows new identities to be added without retraining the model.

Contrastive Loss Standard classification-based approaches may not generalize well in open-world settings like face verification. Instead, contrastive loss directly enforces a clustered structure in the embedding space:

- For matching pairs (same identity), the embedding distance $d(x^{(i)}, x^{(j)})$ should be small.
- For non-matching pairs (different identities), the distance should be large.

Using Euclidean distance $d(x^{(i)}, x^{(j)}) = \|f(x^{(i)}) - f(x^{(j)})\|_2$, a basic contrastive loss can be defined as:

$$L\left(f(x^{(i)}), f(x^{(j)})\right) = \begin{cases} \|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 1 \\ -\|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 0 \end{cases}$$

However, this loss is not well-balanced: the second term is unbounded and can lead to over-separation between classes.

To mitigate this, a **margin** m is introduced to limit the influence of dissimilar pairs once they are sufficiently far apart:

$$L\left(f(x^{(i)}), f(x^{(j)})\right) = \begin{cases} \|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 1 \\ \max\left(0, m - \|f(x^{(i)}) - f(x^{(j)})\|_2^2\right) & \text{if } y^{(i,j)} = 0 \end{cases}$$

Triplet Loss Triplet loss provides a more direct way to enforce the desired structure in the embedding space by considering triplets of images: an anchor A, a positive P (same identity), and a negative N (different identity). The loss encourages:

$$\|f(P) - f(A)\|_2^2 < \|f(N) - f(A)\|_2^2$$

However, this formulation alone is insufficient—it may still allow degenerate solutions (e.g., all embeddings collapse to a constant vector). To address this, a margin m is introduced:

$$\|f(P) - f(A)\|_2^2 + m < \|f(N) - f(A)\|_2^2$$

This ensures that the negative example is not just farther than the positive, but at least by a margin. Compared to contrastive loss, triplet loss does not force all embeddings of the same class to collapse to a single point, making training more flexible.

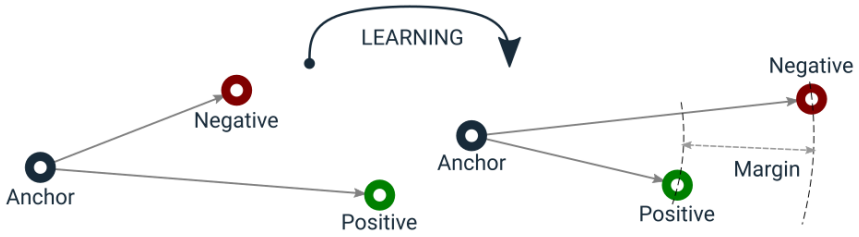


Figure 102: Triplet loss: learning to rank embeddings

In the illustration, the blue and green dots represent the same identity, while the red dot is a different one. The network learns to position the red (negative) farther from the anchor (blue) than the positive (green).

Triplet Selection and Semi-Hard Mining A critical challenge in training with triplet loss is how to form informative triplets. For most random triplets, the margin condition is already satisfied and won't contribute to learning.

To make training effective:

- Large mini-batches are formed by selecting a fixed number of images for each of D identities.
- All anchor-positive pairs are formed within each identity.
- Negative samples are selected based on their distance to the anchor to form meaningful triplets.

There are two important types of triplets:

- **Hard triplets**: where the negative is closer to the anchor than the positive.
- **Semi-hard triplets**: where the negative is farther than the positive, but still violates the margin condition.

Semi-hard negatives are ideal for training. To find them, the following steps are performed at the start of each epoch:

- Compute all embeddings for the training set using the current model.
- For each identity, create all valid (anchor, positive) pairs.

- For each such pair, find negatives N such that:

$$\|f(P) - f(A)\|_2^2 < \|f(N) - f(A)\|_2^2 < \|f(P) - f(A)\|_2^2 + m$$

These are considered *semi-hard* because they fall within the margin.

Avoiding the **hardest negatives** is important since they often correspond to mislabeled or low-quality images, which can degrade training performance.