
**Budapest University of Technology and Economics**
Facutly of Electrical Engineering and Informatics
Department of Contorl Engineering and Information Technology

# Image Processing

Lecture Notes

Márton Szemenyei

September 7, 2024

# Contents

# Part I

# Traditional Vision

# 1 Introduction to Computer Vision

## 1.1 What is Computer Vision?

Various computer vision systems have become widespread in many areas of life. This is due to the rapid growth of available computational power, the proliferation of devices, and significant advancements in the algorithms used in these systems. These algorithms have numerous applications: from robotics and automation to virtual and augmented reality systems, and even the entertainment industry. Furthermore, there is significant potential for applying these methods in public service areas.

The goal of the field of computer vision is to extract information relevant to another decision-making application or a human from images or sequences of images using algorithmic tools. The biggest challenge of the task is that even a single image consists of millions of data points (pixels), which can form images in orders of magnitude more configurations. Processing such a large amount of data requires efficient algorithms and high-performance devices.

Another difficulty in the field is that, although humans can determine vital information from a seen image (and in fact, the eye is the most important human sense), we process most of this information subconsciously, making it challenging to translate these abilities into exact algorithms. This problem is further complicated by the likelihood that many of our vision-based decisions also rely on information obtained through other senses. Due to the above issues, heuristics, machine learning, and mathematical optimization-based methods are often used in the field of computer vision, though their correct operation for every possible scenario cannot be guaranteed.

These algorithms can be categorized in various ways. One of the most common categorizations is based on the goal (output) of the procedures. Here, we distinguish between image processing and computer vision algorithms. In the case of image processing, the goal is to produce a new image that is, in some way, more advantageous than the original one. These procedures are often intended to aid further image processing or improve visibility for humans.

On the other hand, computer vision algorithms output some information extracted from the input at a higher level of abstraction, rather than generating another image. Additionally, we distinguish cases where these procedures are used in an embedded system (machine, car, robot, phone, etc.), in which case we talk about machine vision. A commonly used term is video analytics, where the input to the algorithm is a sequence of images.

Within computer vision, we often distinguish between solutions that use machine learning algorithms during operation, and this field is referred to as learning vision. Special attention is given to solutions that use deep learning, which has become extremely popular in recent years. Despite their diversity, the other methods in this field are referred to as traditional vision.

Traditional vision solutions typically consist of four important steps, the first of which is the acquisition of the images or image sequences to be processed. This is usually followed by an image enhancement step, during which we seek to minimize the impact of any image defects or noise. In this step, other conversions that aid processing may also be performed. The third step is the extraction of features from the image that are useful for solving the given task. In the final step, the algorithm determines the final output based on the extracted features and other data—this step is generally referred to as decision-making.

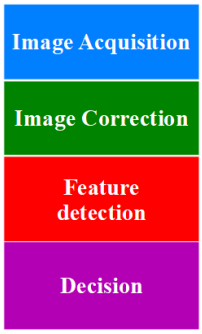


**Figure 1.1:** *The algorithmic steps of traditional vision.*

### 1.1.1 Basic Tasks and Challenges

The primary goal of computer vision is to extract high-level information from images. The simplest form of this is the task of classification, where we assign a single label to an image that encodes the category of the object in the image. In certain cases, a bounding box around the object is also assigned along with the label, in which case we refer to localization. However, in real-world images, multiple objects of different kinds may appear in several instances, requiring the recognition and localization of each relevant object. This task is known as detection.

In some cases, besides the position of objects, we may need to gather information about their shape and pose, for which a bounding box might not be sufficient. In such cases, it may be useful to classify each pixel individually, producing a mask image where the value of each pixel encodes the class of the object to which it belongs. This task is known as semantic segmentation. One drawback of this task is that adjacent objects of the same class may merge, so to avoid this, we can assign both class and object labels to the pixels, leading to the task of instance segmentation.

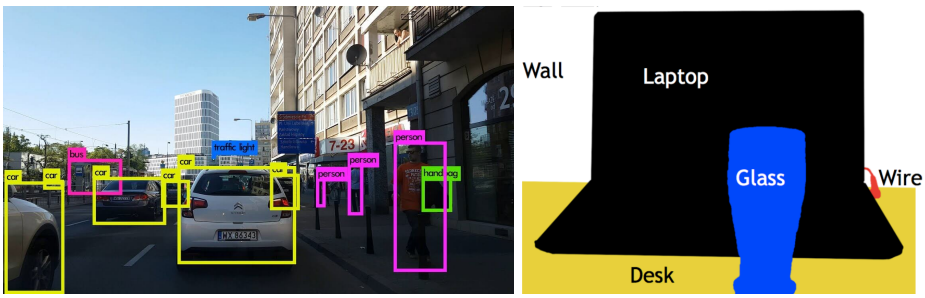


**Figure 1.2:** *The tasks of detection and segmentation.*

We can, of course, go beyond just detecting objects as accurately and in as much detail as possible. We can attempt to extract properties of the individual objects (e.g., gender, age, mood of people) and their relationships (e.g., containment, geometric relations, activities) from the image and organize them into a system. With such complex information, we may be able to make well-informed decisions based on visual information. This task is called scene understanding.

When we aim to extract high-level semantic information from images, we must confront several challenges. The first is that the image of the same object can undergo significant changes due to different lighting conditions, so the numerical value of the pixels belonging to the object will not even approximately match. Similar difficulties arise from rotation, scaling, and perspective distortion, all of which cause considerable changes in the recordings of the same object. Additional problems arise when some objects are deformable, which again alters the values of descriptive features. In real images, it is also common for a significant portion of objects to be occluded, meaning we must recognize them based on only a partial image.

In classification and detection tasks, however, we are not trying to recognize a single specific object but a semantic class. A class can contain many different objects, which can vary significantly

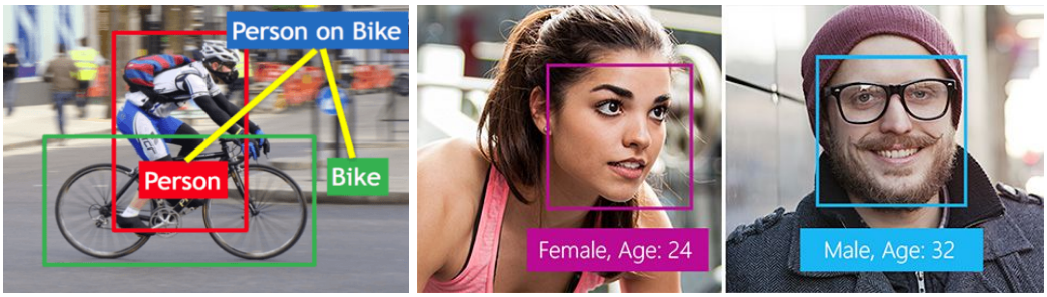


**Figure 1.3:** *Object relations (left) and face detection combined with age regression (right).*

(depending on the class). Furthermore, in the real world, it is common for class members to show no visual similarity, and their belonging to the same class can only be determined based on some physical or functional similarity (consider, for instance, chairs of different designs). This issue is called intra-class variation and is one of the greatest challenges in semantic classification.

Added to the above difficulties is the so-called semantic gap. The semantic gap concept refers to the seemingly insurmountable difference between the digital representation of images and human interpretation. This gap makes it quasi-impossible to formulate more complex computer vision problems as simple algorithms.

## 1.2 Imaging

The most fundamental task in computer vision is creating the images to be processed. Although cameras are well-known, everyday devices, understanding their operation and key properties is essential to choosing the right device for a particular application. In this chapter, we present the different types of sensors and the differences between them.

The operation of cameras is largely based on the principles of the human eye. The human eye is a hollow spherical body with an opening, the pupil, located at the front, through which light enters the lens positioned directly behind it. The lens focuses the parallel incoming light rays onto a point that falls on the retina, the light-sensitive membrane located at the back of the eye. Thanks to the lens, light rays coming from the same direction arrive at the same spot on the retina, ensuring sharp human vision.

On the surface of the retina, numerous light-sensitive cells are located that transmit stimuli to the nervous system. There are two types of light-sensitive cells: the more common rods only detect light intensity, as they have similar sensitivity across the entire range of visible light. In contrast, the less frequent cones are only sensitive to the frequency ranges of blue, green, or red light, enabling the perception of colors.

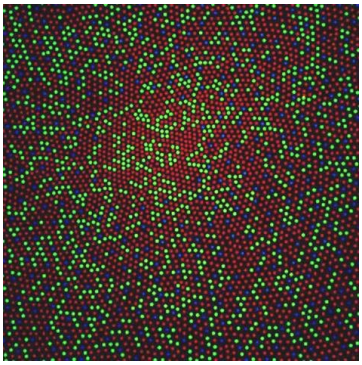

**Figure 1.4:** *The distribution of cones on the retina.*

### 1.2.1 Sensor Types

Cameras are constructed similarly to human vision: due to their geometric design, incoming light rays are projected onto a sensor array, and the measured values create the image. The geometric design of cameras will be discussed in detail in a later lecture. Besides the physical construction of the camera, the type of light-sensitive devices in the sensor array is also crucial. Light detection is typically accomplished using photodiodes. Photodiodes operate on the principle of photoelectricity, meaning that when photons with sufficient energy hit the P-N junction of the diode, they generate an electron-hole pair. If this occurs in the depleted region of the diode, the electric field causes them to move, resulting in an electric current.

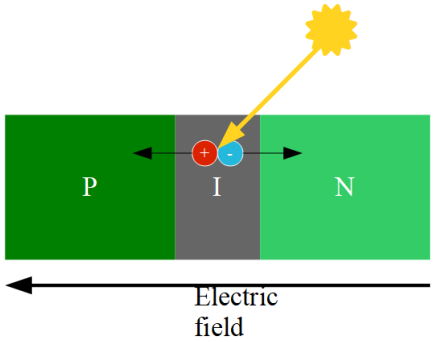


**Figure 1.5:** *The operating principle of a photodiode.*

There are two widespread types of light sensors: CCD (charge-coupled device) sensors and active CMOS (complementary metal-oxide-semiconductor) sensors. The individual elements of a CCD sensor are analog devices that store electric charge when photons strike them. After the exposure time, the outermost cell in each row transfers its charge to an output amplifier, and then the cells pass their charges to neighboring cells. Based on the described principle, the CCD sensor is read out row by row, and within each row, element by element.

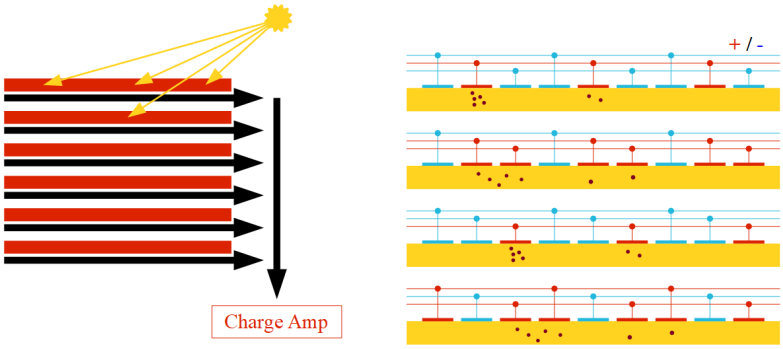


**Figure 1.6:** *The operating principle of a CCD sensor.*

In contrast, in CMOS sensors, each cell has its own amplifier, making the readout of the sensor array significantly faster. However, this comes at the cost of space taken up by the amplifiers within the sensor array, which would otherwise be used for light absorption. To mitigate this drawback, micro-lenses are added to the CMOS cells to direct photons that would otherwise hit the amplifier towards the light-sensitive area. Due to their smaller size, lower power consumption, and more favorable cost, most cameras use CMOS sensors. CCD sensors are primarily used in high-quality video cameras.

There are also several solutions for color detection in cameras, the most common and affordable being the Bayer filter. The Bayer filter is a grid where each element allows only specific colors to pass through. Placing this filter in front of the sensor array ensures that the individual CCD or CMOS sensors only respond to these colors. Most Bayer filters have twice as many green elements as blue and red ones, reflecting the sensitivity of the human eye.

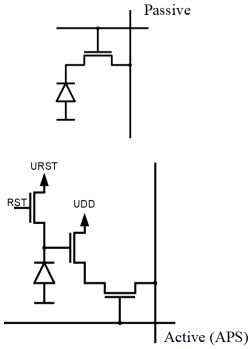


**Figure 1.7:** *The structure of passive and active CMOS sensors.*

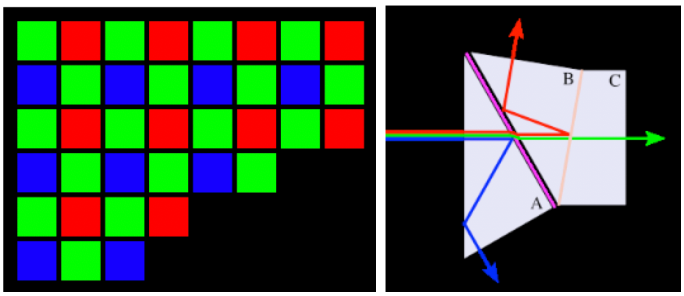


**Figure 1.8:** *The Bayer filter structure (left) and the 3CCD (right).*

An alternative solution is the 3CCD sensor, where a prism splits the three color components and directs them to separate sensor arrays, allowing for sharper color separation. Since this method uses three distinct sensor arrays, 3CCD solutions perform significantly better than Bayer filters in low-light conditions, though at a higher cost.

In recent years, specialized sensors capable of determining not only the intensity of each pixel but also its distance from the sensor have become increasingly common. These devices add a fourth numerical value to each pixel and are known as depth cameras.

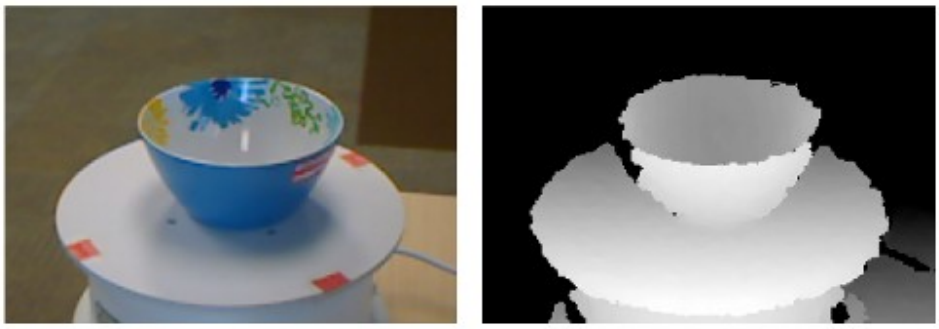

**Figure 1.9:** *A color and a depth image.*

There are essentially two types of these cameras: the first is known as a stereo camera, where two cameras are placed at a fixed distance from each other in a single housing. The distance of each pixel can be calculated from the correspondences between the two captured images. These devices are usually calibrated during manufacturing, and the depth calculation is performed by the processing hardware built into the camera.

On the other hand, infrared-based depth cameras consist of three components: a regular RGB sensor, an infrared projector, and an infrared sensor. They project a predetermined pattern in the invisible infrared spectrum, which is then read back by the infrared sensor. The spatial structure

of the scene is determined based on the distortion of this pattern. Infrared-based sensors are more widespread because they provide better-quality results and require less processing. Their drawback is that other infrared sources in the environment can interfere with the results.

Another type of sensor is the LIDAR sensor. This operates similarly to radar, deducing the distance to objects from the delay and frequency of reflected electromagnetic waves, but it uses laser beams instead of radio waves. Due to the tremendous improvement in the response time of digital processing units, this technology can now measure extremely precise distances (on the order of centimeters).

## 1.3 Image Properties

After being read out, the image captured by the sensors becomes a two-dimensional array of numbers when it enters the computer. Each element of this array represents the intensity of the light arriving at a particular position. These elements are called picture elements or pixels. In most cases, pixels in the computer are represented by an 8-bit number, where 0 represents complete darkness, and the maximum value of 255 represents a fully bright pixel. In the case of color images, each position is associated with three numerical values, commonly referred to by the abbreviation RGB (red-green-blue).

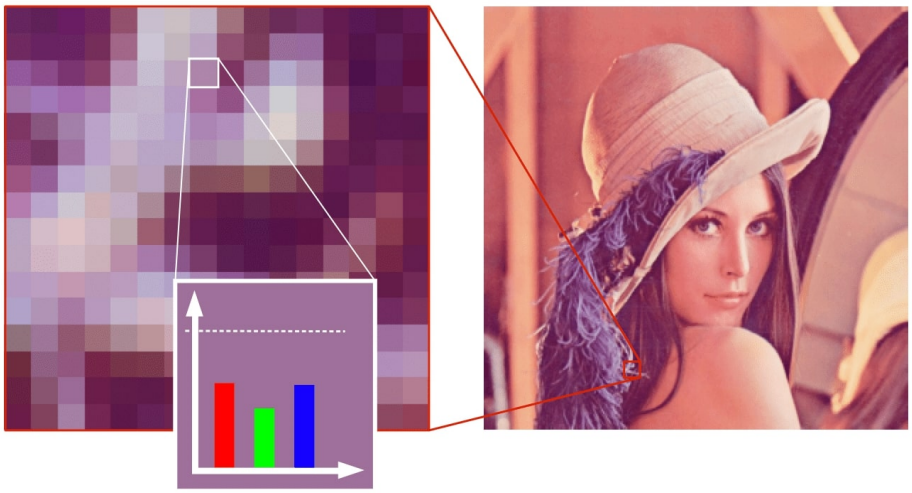

**Figure 1.10:** *Structure of an image.*

Digital images have numerous important parameters and properties. One of the most well-known is the resolution, which defines the dimensions of the numerical array. This parameter significantly determines the image's level of detail, but increasing it also increases the amount of data needed to store the image. Another important parameter is the bit depth of the image, which indicates the number of bits required to store a single pixel. The most common case is an 8-bit depth, but in floating-point representations, this value can be 32. Compression techniques may also use bit depths smaller than 8, which reduces the image's level of detail to some extent. An image with a bit depth of 1 is called a binary image.

## 1.4 Compression and Storage

A significant problem may arise when storing images: for example, an average image taken by a mobile phone contains tens of megabytes of information, and an average FullHD video can be in the range of hundreds of gigabytes or even terabytes. This would impose a load that exceeds the capacity and speed of today's hard drives. For this reason, it is essential to briefly discuss various compression methods. These methods generally exploit the characteristics of human vision; for


**Figure 1.11:** *The effect of reducing resolution and bit depth.*

example, a pixel can be stored in 2 bytes by reducing the number of bits allocated to individual color components in different ways, leveraging the specifics of color perception.

There are many common image formats, one of the most basic being BMP, which is an uncompressed format. There are lossless compressed formats, the most notable of which is PNG, a raster graphic format. This format is mainly used for compressing text and diagrams. It is also worth mentioning the SVG and EPS formats, which primarily use vector graphic representation.

In lossy compression, the most widespread format is JPEG, and in the case of videos, MPEG, which are typically designed for real images and videos. For videos, the H.264 and H.265 formats, which are the latest versions of MPEG compression, are commonly used.

## 1.5 Color Representation

In computer vision, beyond the intensity information contained in grayscale images, we often utilize the additional information carried by color images. Many simple detection algorithms are based on color similarity searches. However, several issues can arise. For example, to make color similarity searches reliable and robust, it is necessary that the pixel values describing the colors allow us to easily express the similarity between colors.

The most commonly used color representation in cameras (also known as the RGB color space) is not suitable for this purpose. The geometric distance between two points describing colors in the RGB color space does not reflect how similar the two colors appear to human perception. Moreover, if the lighting conditions change, all three values in an RGB image change, even though the color of the object in the image has not changed.

The goal of color space transformation procedures is to provide a new color representation that can accurately describe color similarity and be robust to changes in lighting, without losing information. These transformations define a new color space. In certain cases, the transformation can be described using a matrix multiplication.

$$\begin{pmatrix} S_1 & S_2 & S_3 \end{pmatrix} = \mathbf{C} \begin{pmatrix} R & G & B & 1 \end{pmatrix} \tag{1.1}$$

$$\begin{pmatrix} R & G & B \end{pmatrix} = \mathbf{C^{-1}} \begin{pmatrix} S_1 & S_2 & S_3 & 1 \end{pmatrix} \tag{1.2}$$

The most common color space conversion is grayscale conversion, where a single brightness-like value is derived from the three color channels. There are many methods for this, with the most

commonly used being luma (Y), intensity (I), value (V), and luminance (L). These are calculated as follows:

$$\begin{aligned} Y &= 0.3R + 0.59G + 0.11B \\ I &= \frac{1}{3}R + \frac{1}{3}G + \frac{1}{3}B \\ V &= max(R, G, B) \\ L &= \frac{max(R, G, B) + min(R, G, B)}{2} \end{aligned} \tag{1.3}$$

One of the most commonly used color spaces is the YCbCr family, which has several minimally different variants. Of the three channels, Y is a separated brightness component that represents the brightness of a given color. The other two channels, Cr and Cb, encode the hue of the color. This color space is commonly used in digital video systems, as well as in JPEG and MPEG encoding. In practice, the YCbCr color space is often confused with the YUV color space, which operates on a similar principle but is used in analog systems.

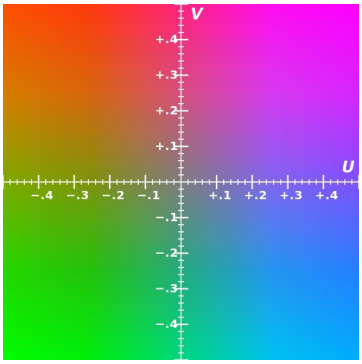


**Figure 1.12:** *The UV plane of the LUV color space.*

The transformation matrix for YCbCr is as follows:

$$C_{YCbCr} = \begin{pmatrix} 0.299 & 0.587 & 0.114 & 0 \\ -0.169 & -0.331 & 0.5 & 128 \\ 0.5 & -0.419 & -0.081 & 128 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.4}$$

Another commonly used family in image processing is the HSV/HSI/HSL family. These color spaces share the characteristic of describing color information using two values, hue and saturation, while the main difference between representations lies in how brightness or intensity is represented. This color space can be easily visualized in the form of a cylinder or cone. Both of these alternative color spaces are frequently used in color-based processing.

The conversion to the HSV color space is nonlinear, meaning it cannot simply be described by a transformation matrix. The method is as follows:

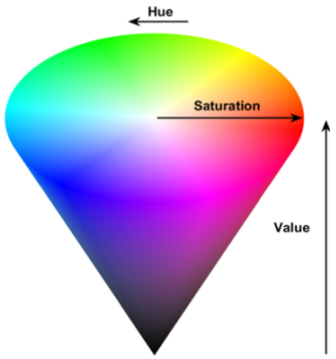


**Figure 1.13:** *The HSV color space can be represented using a cone.*

$$V = max(R, G, B)$$

$$S = \frac{V - min(R, G, B)}{V}$$

$$if \quad S == 0 \quad then \quad H = 0$$

$$c_r = \frac{V - R}{V - min(R, G, B)} \quad c_g = \frac{V - G}{V - min(R, G, B)} \quad c_b = \frac{V - B}{V - min(R, G, B)} \tag{1.5}$$

$$if \quad R == V \quad then \quad H = 0 + 60 * (c_b - c_g)$$

$$if \quad G == V \quad then \quad H = 120 + 60 * (c_r - c_b)$$

$$if \quad B == V \quad then \quad H = 240 + 60 * (c_g - c_r)$$

Behind the seemingly complex saturation and hue calculations lies a simple principle: Saturation is proportional to how much stronger the most dominant color component is compared to the weakest one. Following this, the three color components are placed at three equally spaced points on a circle: red at 0 degrees, green at 120 degrees, and blue at 240 degrees. Subsequently, the two weaker color components pull the current color in their respective directions.

# 2 Image Enhancement, Filtering

## 2.1 Intensity Transformations

One of the simplest methods of image enhancement is the use of various intensity transformations. When applying this method, the algorithm processes the image pixel by pixel, changing the value of each pixel based on a predefined transformation function. In most cases, this transformation function determines the new intensity value solely based on the previous intensity of the given pixel.

There are several versions of these transformations: if a constant is added to the intensity of each pixel, the brightness of the image can be adjusted (increased or decreased depending on the sign of the constant). It is also possible to multiply the pixel intensity values by a constant, which allows adjusting the contrast of the image. When performing a contrast transformation, a constant is also added to shift the new intensity values to the center. It's important to note that these transformations assume that pixel values saturate, meaning they are clipped if the transformation pushes them below the minimum value of 0 or above the maximum value of 255.

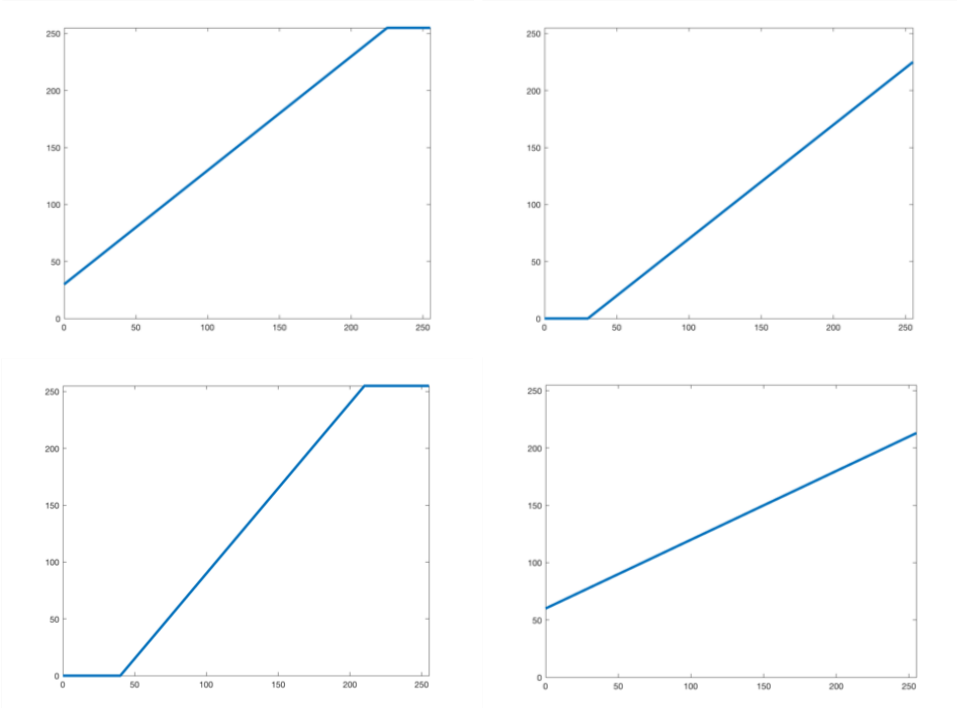

**Figure 2.1:** *Various intensity transformations: Increasing and decreasing brightness (top), and increasing and decreasing contrast (bottom).*

A special type of intensity transformation is thresholding. In this case, pixels on one side of a predefined threshold are set to 0, while those on the other side are set to 1, resulting in a binary image. This operation can also be performed with two threshold values, setting values within or outside the range to 1. Thresholding is particularly useful for highlighting pixels of a certain intensity in an image, and from their quantity, size, or position, further conclusions can be drawn. It is important to note that when using a predefined threshold, changes in lighting conditions can

**Figure 2.2:** *The result of thresholding.*

significantly affect the outcome of the method, so in practice, adaptive thresholds—determined based on the intensity distribution of the image—are often used.

Intensity transformations can also be used on multichannel (i.e., color) images. In this case, the transformation functions are applied independently to each color channel. Naturally, different functions can be used for each channel, allowing the adjustment of color relationships and relative dominance in the image. When using thresholding, certain colors can be highlighted, enabling the detection of objects of a specific color.

## 2.2 Histogram

While intensity transformations are simple and fast methods, their robustness can leave something to be desired. This is why transformations based on the image histogram are often used. A histogram shows the relative frequency of each intensity value in a given image or color channel (in this sense, it is the empirical density function of the intensities). Histograms can often help detect and correct errors in image capture, usually related to lighting or exposure.

In underexposed images, due to the short exposure time, not enough photons hit certain elements of the sensor array, resulting in even the brightest pixel having a relatively low value. Consequently, the intensity range of the image is compressed into the lower half of the full range, making the details of the image difficult to see. In overexposed images, a similar phenomenon occurs, but due to the excessively long exposure time, the dark pixels become too bright, and the image information is compressed into the upper portion of the range.

The histogram equalization algorithm addresses such errors. In underexposed or overexposed images, the histogram significantly deviates from a uniform distribution. The goal of the histogram equalization algorithm is to approximate a uniform distribution by merging rare intensity values and shifting common ones. It is important to note that the application of this algorithm results in some loss of information (due to the merging of certain intensity values), so it is typically used only to improve the visibility of the image.

## 2.3 Image Noise, Types of Noise

Images captured with real devices are always plagued by noise and various errors that complicate processing. These noises come from different sources and, depending on their origin, can be of various types. The most common type of noise in images is Gaussian noise, which results from the internal noise of the pixel sensor and the surrounding electronics. This type of noise is typically additive and independent from pixel to pixel.

Another common type of noise is salt-and-pepper noise, which causes significant deviations in the value of individual pixels, but occurs only rarely, resulting in bright pixels appearing in dark regions

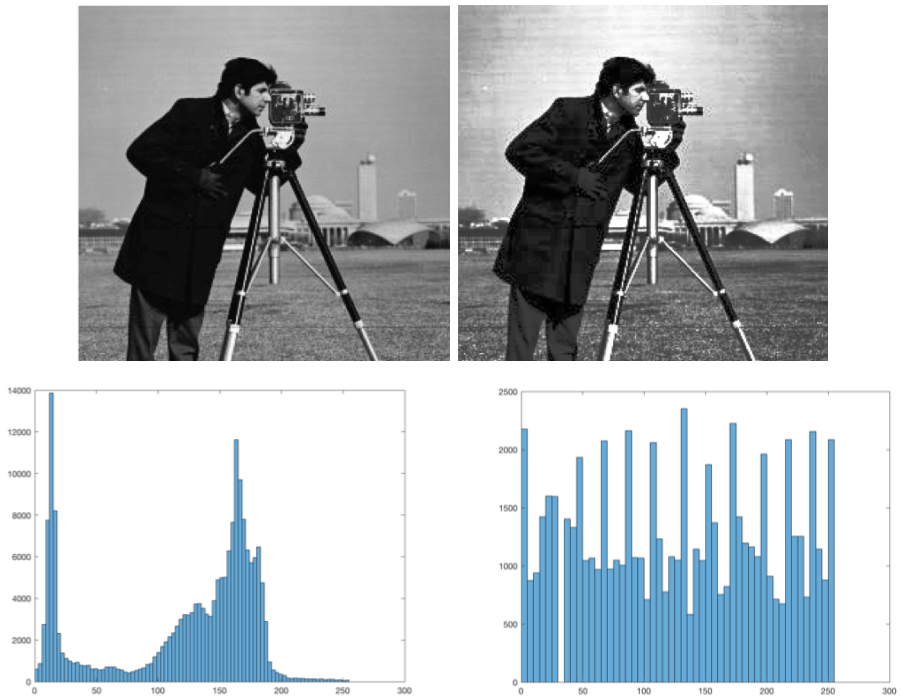

**Figure 2.3:** *Histogram equalization: the original image and its histogram (left), and the equalized image (right).*

(or vice versa). This type of noise is most often caused by analog-to-digital converter issues or bit errors during transmission. Noteworthy are also quantization errors occurring during digitization and periodic errors caused by possible electromagnetic interference.

**Figure 2.4:** *Gaussian noise (left) and salt-and-pepper noise (right).*

## 2.4 Convolutional Filters

The most important members of the image enhancement algorithms family are the various filter algorithms, which aim to improve image errors and noise. These procedures are based on convolutional filtering. By applying a small filter window over the image, the new value of each pixel position is determined by the result of the convolution operation between the filter window and the pixel's local neighborhood. The convolution operation is given by the following formula:

$$(k \circledast I)(x,y) = \sum_{u=-n}^{n} \sum_{v=-n}^{n} k(u,v) * I(x-u, y-v) \tag{2.1}$$

where $I(x, y)$ is the pixel at the x-th column and y-th row of the image. As seen from the formula, the convolution operation is simply the weighted sum of the pixels in the given neighborhood based on the weights from the filter. In practice, filters are always designed such that the sum of the weights equals one; otherwise, the image would be made brighter or darker. It is important to note that although according to the convolution formula, one should move in the opposite direction of the image patch and the filter, in practice, this is not always the case. Therefore, in reality, we compute the cross-correlation operation, but it is still referred to as convolution, although the results match only for centrally symmetric filters.

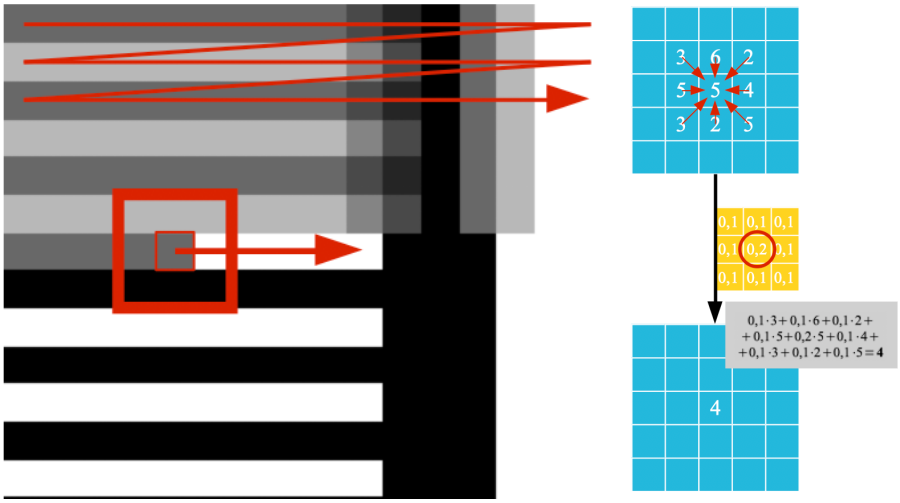


**Figure 2.5:** *The principle of convolutional filtering (left) and the convolution operation at a given position (right).*

### 2.4.1 Linear Filters

Convolutional filters used for noise reduction are called smoothing filters, with the simplest version being the averaging filter. A slightly more sophisticated version is the Gaussian filter, which considers pixels farther from the center with smaller weights, based on a Gaussian bell curve. By adjusting the standard deviation of the bell curve, one can control the strength of the smoothing, and if different standard deviation values are specified for different directions, it is possible to create a Gaussian filter that smooths much more drastically in one direction than in the other.

One of the most fundamental properties of smoothing filters is that every element in the convolutional window is non-negative, and the sum of the elements is exactly 1. If the sum of the weights deviates from this, the filter will not only smooth but also brighten or darken the image. A good property of convolutional filters is that they are linear operations, so successive filtering operations can be easily combined. Additionally, for certain filter windows, it is possible to separate the filter: in this case, a 2D filter can be replaced by two consecutive 1D filters. In this scenario, performing the filter calculation requires $2 * N$ operations instead of $N^2$. However, this can only be done for rank-1 filter matrices.

Convolutional smoothing filters have two main problems: first, while they effectively remove noise, they also blur some details in the image (especially sharp transitions and edges), making the image appear less sharp. Second, since all such filters perform some form of averaging, they are significantly affected by outliers (salt-and-pepper noise), which can distort the average value. As a result, these filters tend to smear rather than eliminate salt-and-pepper noise.

### 2.4.2 Sharpening Filters

In contrast to smoothing filters, where the filter weights are always positive, there exist filters with both positive and negative weights, yet their sum equals 1. These are called sharpening filters and,

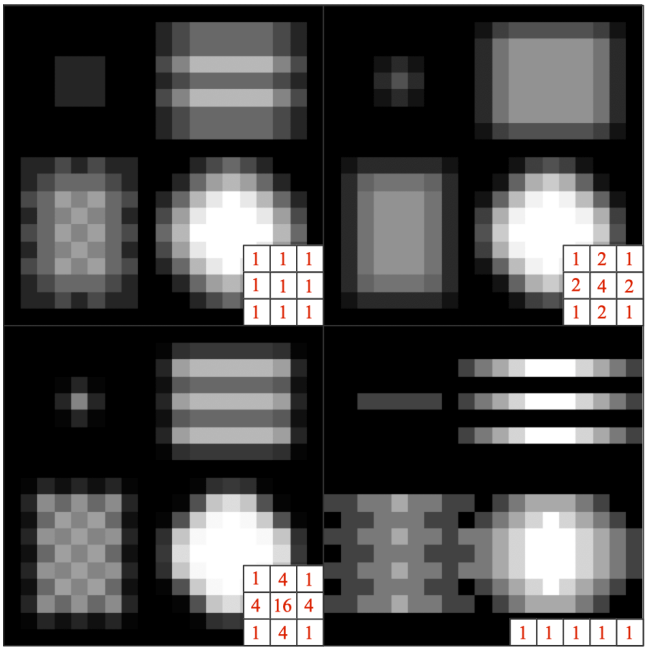


**Figure 2.6:** *Some typical convolutional filters: The averaging filter (top left), the Gaussian filter with two different standard deviation values (top right and bottom left), and a horizontal averaging filter (bottom right).*

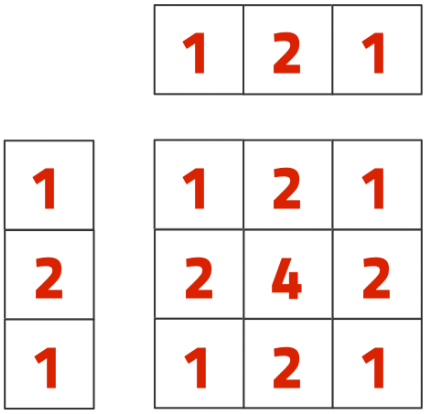


**Figure 2.7:** *A 2D filter decomposed into two 1D filters.*

**Figure 2.8:** *An image contaminated with white noise (left) and the effect of smoothing filtering (right).*

as their name suggests, are designed to enhance fine details and changes in images, making them appear sharper. This type of filter is commonly used in photographic applications. There are also so-called edge-detection filters, which are similar to sharpening filters but with a weight sum of 0.

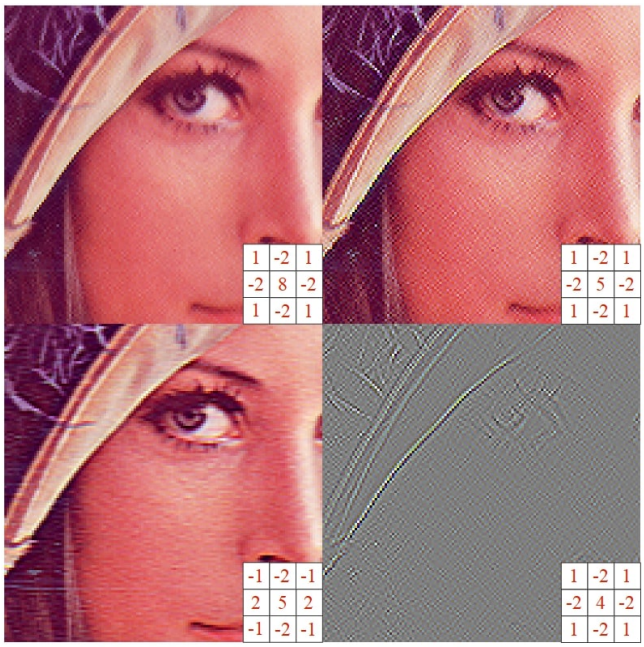


**Figure 2.9:** *Various sharpening filters, as well as an edge-detection filter (bottom right). Note the different filter sums.*

## 2.5 Rank Filters

Rank filters provide a solution to the problems associated with smoothing filters previously mentioned. Rank filters also consider a small neighborhood of the given pixel, but instead of performing convolution, they sort the pixels in the neighborhood by intensity and select a value from this sorted list to assign a new value to the pixel being examined. Among rank filters, maximum and minimum filters are commonly used for various tasks, and median filters are the most prevalent for image filtering.

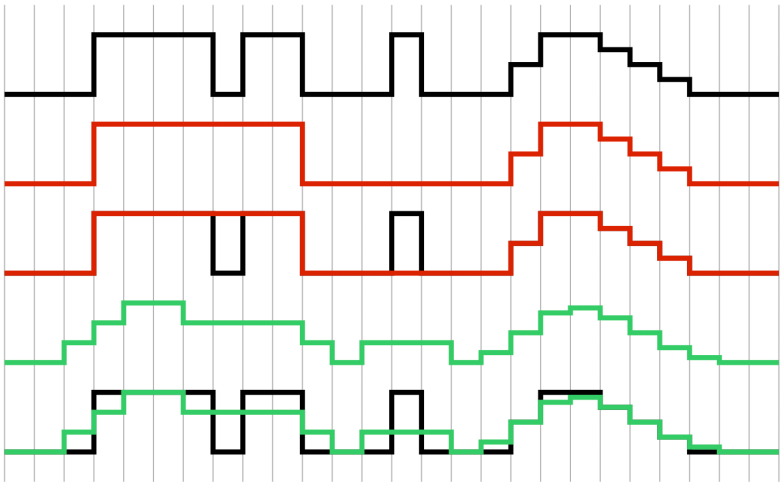

**Figure 2.10:** *The difference between median (red) and averaging (green) filters.*

Median filters replace the pixel in question with the median of the intensities of all the pixels in its

neighborhood, meaning the middle value after sorting. The significant advantage of this filtering is that it preserves sharp boundaries and edges while effectively filtering salt-and-pepper noise. This is because the median statistic is very robust to rare, extreme values compared to the average. The drawback of rank filters is that sorting is computationally expensive compared to convolution, leading to significantly slower performance. Additionally, some high-level acceleration techniques (e.g., separable filters, frequency domain processing) can only be applied to convolutional filters.

**Figure 2.11:** *An image with salt-and-pepper noise (left) and its median-filtered version (right).*

## 2.6 Image Mathematics

Since images are two-dimensional arrays, they are equivalent to matrices known from linear algebra in terms of representation. This means that any operation that can be performed with matrices also works on images. Some of these operations (matrix multiplication, decompositions) have limited utility, while others are particularly useful. Scalar operations can implement intensity transformations discussed in a previous lecture: addition adjusts brightness, while multiplication adjusts contrast.


**Figure 2.12:** *Texturing.*

Operations can also be performed between images of the same size. For example, blending can be achieved by averaging images, while texturing can be realized by element-wise multiplication of two images. One of the most useful operations, however, is subtracting one image from another. This helps highlight the differences between images, which can significantly ease many processing tasks. This approach is commonly used for motion detection or separating a static background.

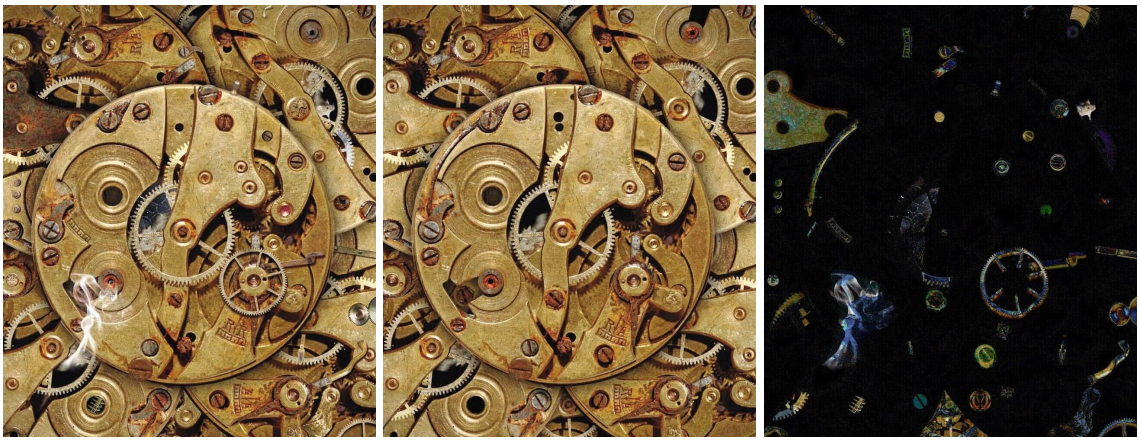

**Figure 2.13:** *Difference imaging.*

## 2.7 Interpolation Techniques

During image processing, it is often the case that an image is not in the desired format. A common situation is when we want to view an image at a larger size than its resolution allows. To increase the resolution, we need to determine the values of the new pixels. Naturally, we have no extra information, so we have to rely on known pixel values and assumptions about the image (continuity, smoothness). We are in a similar situation when performing geometric transformations (e.g., rotation, perspective transformation) on images, as the new, transformed pixel grid will not perfectly overlap with the original grid.

Interpolation techniques are used to determine the new pixel values. These techniques share the property of approximating a signal using known points and then determining the values of new, unknown points by sampling a predetermined function. It is worth noting that this process does not create new information.

There are many different methods of interpolation, which differ based on the family of functions used. The simplest of these is the nearest neighbor approach, which approximates the signal with a constant function over segments. In this case, the new pixel value simply matches the value of the nearest neighbor. A more complex method is linear interpolation, where a linear function is used for approximation over segments, so the new point's value is the weighted average of its two neighbors. The degree of the approximating function can naturally be increased; for example, cubic interpolation uses a cubic polynomial estimated from the four nearest neighbors.

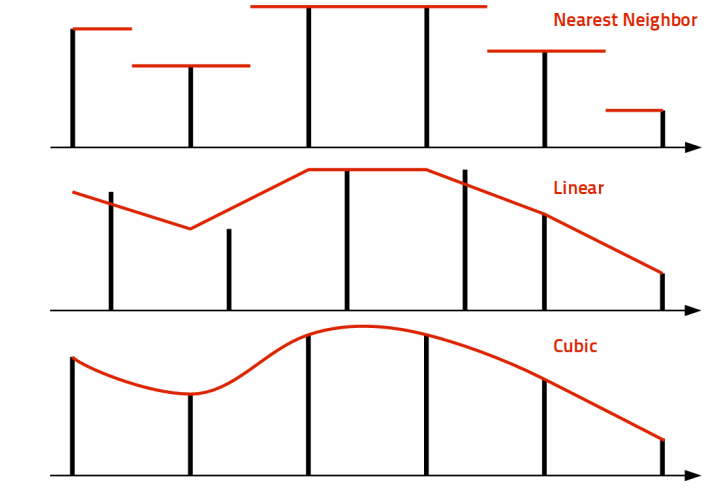


**Figure 2.14:** *1D interpolation techniques.*

However, images are two-dimensional functions, so interpolation must be performed accordingly. Fortunately, the simple interpolation methods mentioned above can be easily extended to arbitrary dimensions. The only limitation is that the amount of neighbors (and computations) required for interpolation grows exponentially with the number of dimensions. In two dimensions, nearest neighbor interpolation is the simplest, but it is only useful for artificial images (geometric diagrams, printed text), or other non-image-based matrices.

One of the most commonly used image interpolation methods is bilinear interpolation, which is an extension of the previously mentioned linear method. The essence is to use the four neighbors of the new pixel to perform two 1D interpolations in one direction. We then get two new points, one of which shares a coordinate with the new point. After that, we simply perform another interpolation in the other direction to calculate the new point's value. Bilinear interpolation has two important advantages: it is simple and does not cause overshooting. However, it tends to make the resulting image appear particularly blurry to the human eye and can cause abrupt changes in intensity gradients, reducing the image's smoothness. Bilinear interpolation is calculated as follows:

$$\begin{aligned} f(x,y_1) &= \frac{x_2-x}{x_2-x_1}f(x_1,y_1) + \frac{x-x_1}{x_2-x_1}f(x_2,y_1) \\ f(x,y_2) &= \frac{x_2-x}{x_2-x_1}f(x_1,y_2) + \frac{x-x_1}{x_2-x_1}f(x_2,y_2) \\ f(x,y) &= \frac{y_2-y}{y_2-y_1}f(x,y_1) + \frac{y-y_1}{y_2-y_1}f(x,y_2) \end{aligned} \tag{2.2}$$

Cubic interpolation can also be extended to two dimensions, using 16 instead of 4 neighboring pixels: this is known as bicubic interpolation. In this method, the image is approximated by a cubic surface, whose formula is:

$$f(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j \tag{2.3}$$

where $a_{ij}$ are the polynomial coefficients. The task of interpolation is to determine these 16 unknown coefficients. To do this, we write 4 equations for each of the 4 elements in the 2x2 neighborhood surrounding the pixel to be interpolated. We specify the intensity value at each point and also specify the values of the intensity's partial derivatives in the x and y directions. Finally, we include the value of the common partial derivative (with respect to x and y), resulting in a total of 16 equations for the 16 parameters, which can then be determined. The general form of the equations to be written is:

$$\begin{aligned} \frac{\partial f(x,y)}{\partial x} &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} i x^{i-1} y^j \\ \frac{\partial f(x,y)}{\partial y} &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} x^i j y^{j-1} \\ \frac{\partial f(x,y)}{\partial x \partial y} &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} i x^{i-1} j y^{j-1} \end{aligned} \tag{2.4}$$

It is important to note that we do not initially know the values of the derivatives, but these can be simply calculated from the image intensity values as follows:

$$\begin{aligned} \frac{\partial f(x,y)}{\partial x} &= \frac{f(x+1,y) - f(x-1,y)}{2} \\ \frac{\partial f(x,y)}{\partial y} &= \frac{f(x,y+1) - f(x,y-1)}{2} \\ \frac{\partial f(x,y)}{\partial x \partial y} &= \frac{f(x+1,y+1) - f(x-1,y) - f(x,y-1) + f(x,y)}{4} \end{aligned} \tag{2.5}$$

One advantage of bicubic interpolation is that it makes the image appear sharper than bilinear interpolation and maintains smoothness due to the prescribed derivatives at the edges. However, its drawback is that interpolation can cause overshooting, which can result in ring-like artifacts around edges.

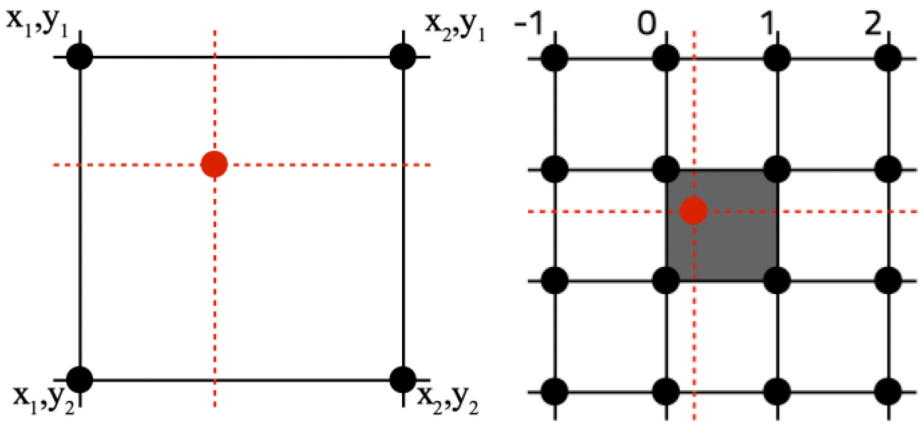


**Figure 2.15:** *The principle of bilinear (left) and bicubic (right) interpolation techniques.*

A very good compromise is the Lanczos interpolation, which uses a weighted average of the neighboring pixels with a so-called Lanczos kernel to determine the new value. This method also provides good sharpness with minimal overshooting.

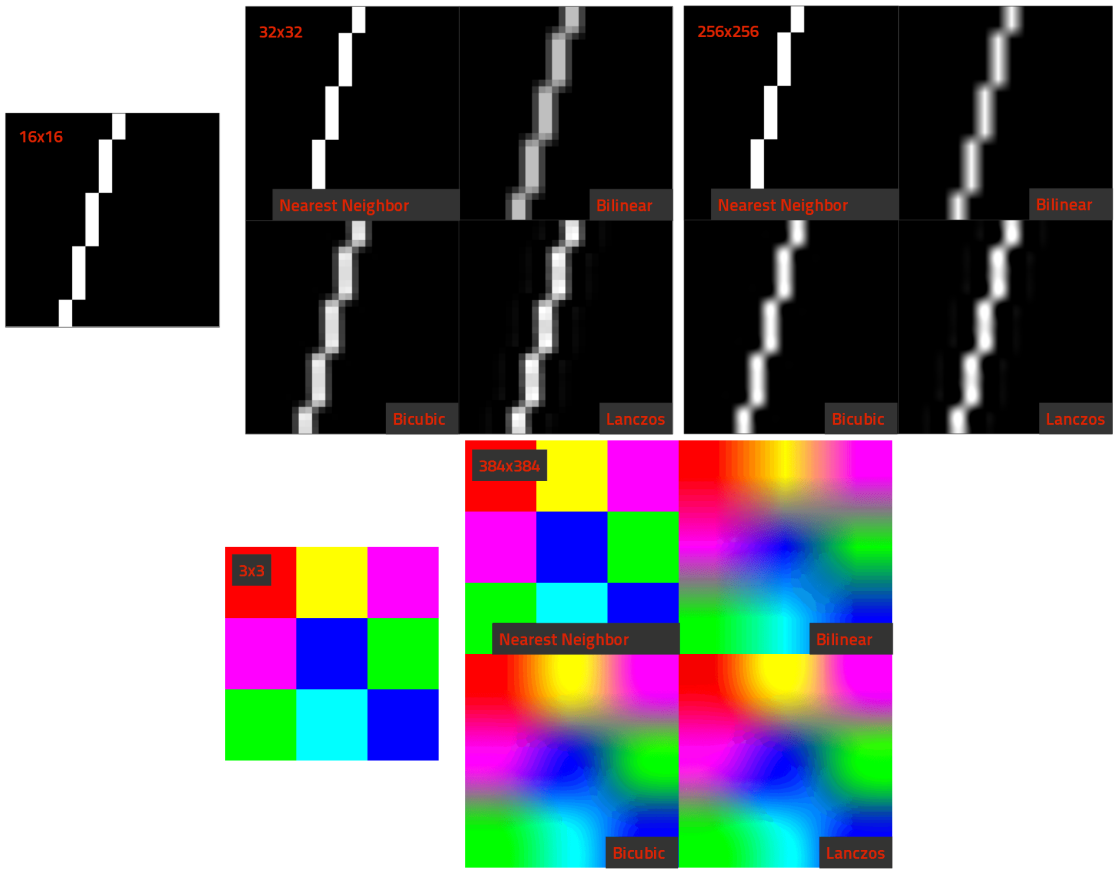


**Figure 2.16:** *Results of different interpolation techniques with varying scale factors.*

# 3 Frequency Domain Processing

## 3.1 Introduction

When dealing with functions, it is almost self-evident to represent spatial and temporal functions directly with these quantities. However, this is not the only way to represent functions. According to the Fourier series decomposition theorem, every periodic function can be expressed as the sum of sine and cosine functions of different frequencies, where each frequency has its own amplitude (magnitude) and phase (shift). The amplitudes and phases associated with these sine-cosine pairs of specific frequencies (which can be expressed as a complex number) are called the spectrum of the signal, and representing the image with these values is referred to as the frequency domain representation of the signal. The Fourier series can be written as follows:

$$f(t) = a_0 + \sum_{k=1}^{N} a_k \sin(k\omega_0 t + \phi_k) \; f(t) = \hat{f}_0 + \sum_{k=-N}^{N} \hat{f}_k e^{ik\omega_0 * t} \tag{3.1}$$

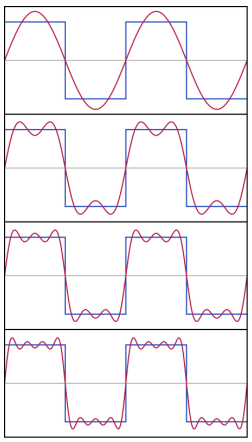

**Figure 3.1:** *Generating a square wave using sine waves.*

## 3.2 Fourier Transform

The lowest frequency term in the Fourier series—the fundamental harmonic—has a frequency equal to the reciprocal of the signal's period, and the frequencies of the other terms—the harmonics—are integer multiples of this fundamental frequency. It is evident that if the period of the signal approaches infinity (i.e., to an aperiodic function), the spectrum will become a continuous function. This continuous complex function is called the Fourier transform of an arbitrary signal. The Fourier transform can also be performed on sampled (discrete) signals, in which case the Fourier transform will be a periodic function—meaning that beyond a certain frequency, it contains no new information. This is because a sampled signal cannot change arbitrarily quickly.

This is just trying to say that if you don't need the higher frequencies to refine the curve becuase it is discreate

In the field of computer vision, it is common to interpret a two-dimensional image as a function of the two dimensions of the image plane. In this horizontal x and vertical y coordinate space, the image can be described as a sum of point-like impulses located at discrete points, where the

magnitude of each impulse is given by the pixel values. Fortunately, the previously introduced Fourier transform can be extended to any (in our case, 2) number of dimensions as follows:

$$F(u,v) = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} I(x,y) * e^{-i(ux\frac{2\pi}{W} + vy\frac{2\pi}{H})} \tag{3.2}$$

where $I(x,y)$ is the pixel value at the y-th row and x-th column, u and v are the horizontal and vertical frequency components, and H and W are the height and width of the image.

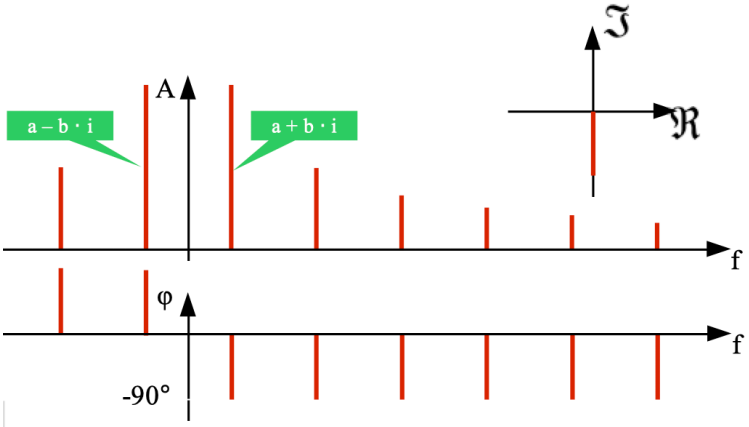


**Figure 3.2:** *Amplitude and phase spectra obtained from the discrete Fourier transform.*

Since images are two-dimensional signals, the periodic signals that make up the image also have directions, not just frequencies and phases (which is why the Fourier transform is two-dimensional). Thus, the displayed amplitude spectrum allows us to determine not only the dominant frequencies in the image but also their directions.

Earlier, we explained that an important property of the Fourier transform is that the spectrum of periodic signals will be discrete, while the spectrum of discrete signals will be periodic. This is fortunate for us because the image is also a discrete signal, so its spectrum will be periodic. This means that it is sufficient to store just one (finite-sized) period of the spectrum, allowing us to convert the image to the frequency domain and then back without losing information. However, due to the physical limitations of computers, the image spectrum can only be stored as a discrete function, sampled, so all our frequency domain operations will assume that the image repeats periodically beyond its edges. This behavior changes how some otherwise equivalent algorithms work depending on whether they are applied in the image or frequency domain.

This is became of the discreate <---> periodic

### 3.2.1 Phase Distortion

It is important to note that when discussing signals in the frequency domain, we usually talk less about the phase characteristic compared to the amplitude spectrum, as the latter has a more tangible physical interpretation. However, this does not mean that the phase characteristic is less important. Phase errors or distortions can cause significant deviations in the signal shape, even if the amplitude spectrum remains the same. This is because different harmonics can add up if they have the same phase, leading to large peaks in the signal shape.

This phenomenon can be observed well in the following image: If the harmonics of a square wave are summed with a 90-degree phase difference, we obtain a good approximation of the square wave, whereas with zero phase shift, we get an entirely different signal. In real images, this can easily lead to pixel value overflow, resulting in information loss.

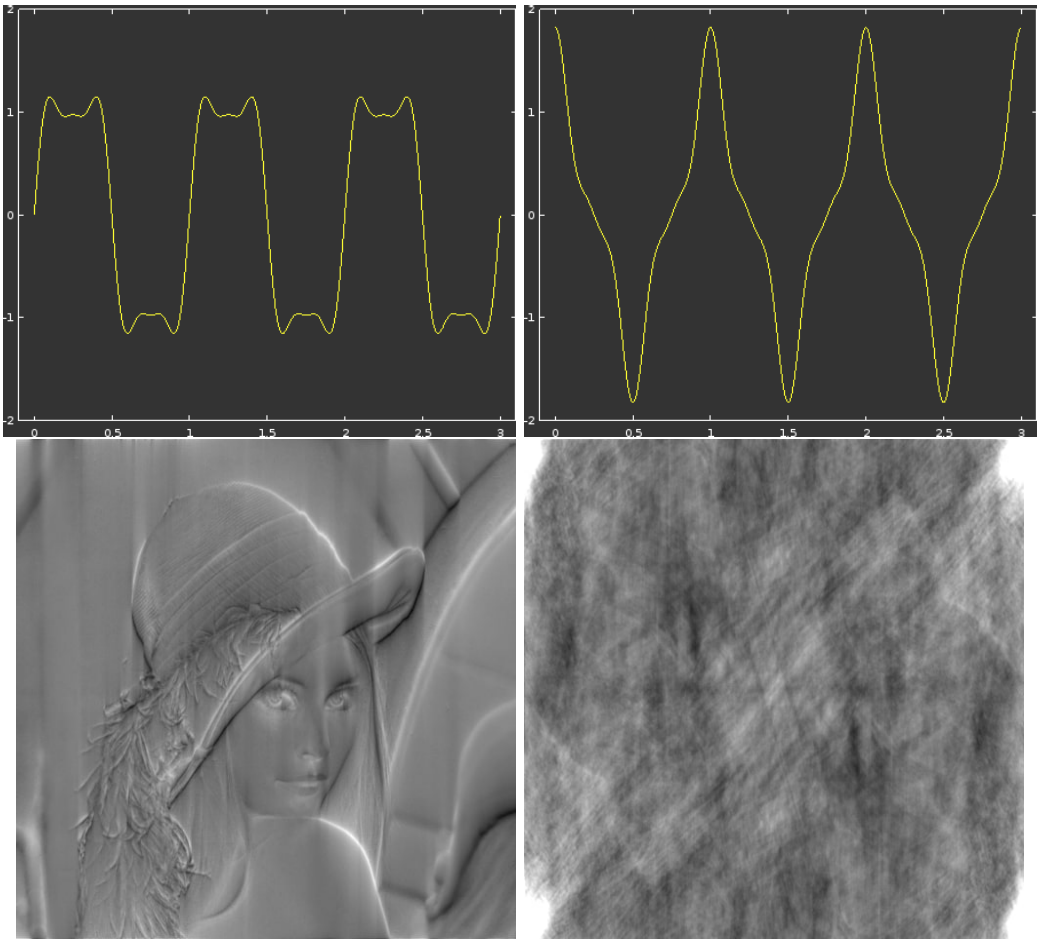

**Figure 3.3:** *The effect of phase distortion on a square wave (top) and a real image (bottom).*

### 3.2.2 Fast Fourier Transform

The image spectrum is usually determined using the Fast Fourier Transform (FFT) algorithm. The essence of the FFT algorithm is that it recursively divides the 1D function with N elements into segments of length $\frac{N}{2}$, placing the even elements in one segment and the odd elements in another. Once there are only 2 elements in a segment, it performs the so-called butterfly operation on them.

$$y_0 = x_0 + x_1\omega^k \;\; y_1 = x_0 - x_1\omega^k \tag{3.3}$$

Then, in the process of recursion, the butterfly operation is repeatedly performed on increasingly longer data sequences (powers of 2), resulting in the FFT values for N points. The algorithm thus achieves that the Fourier transform is computed in $N \;\; log(N)$ steps instead of $N^2$ by reusing intermediate results computed during the butterfly operations. It is worth noting that, similar to the DFT, the FFT can be easily extended to 2 dimensions, which involves performing the transform separately in each direction.

Discreate Forierer Transform

## 3.3 Filters

In the previous sections, we discussed the concept of representation in the frequency domain, but we didn't provide a strong justification for why this is worthwhile. As we will see in the remainder of the lecture, one of the primary applications of frequency domain image processing is the efficient implementation of various filtering techniques.

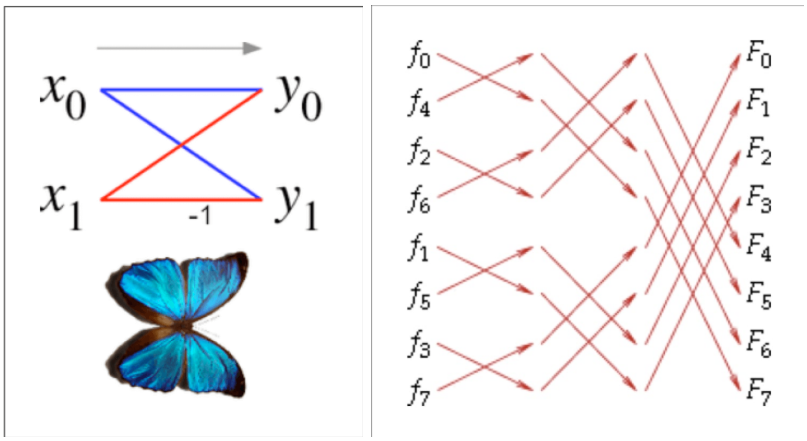


**Figure 3.4:** *The butterfly operation (left) and the FFT applied to 8 data points (right).*

### 3.3.1 Ideal Filters

It is easy to understand that if an image can be represented as the sum of sinusoidal and cosine functions of different frequencies, the lower-frequency functions contain the slower, smoother characteristics of the image, while the higher-frequency components describe the sharp, sudden changes. This relationship can be exploited for several purposes: image noise is typically independent from pixel to pixel, causing sudden changes—so if we suppress these using a low-pass filter in the image spectrum, we can perform noise reduction. Similarly, edges in an image are sudden, sharp changes in intensity, so if we preserve only the high-frequency components using a high-pass filter, we can perform edge detection. If we enhance the high-frequency components of the image spectrum without completely removing the low-frequency components, this corresponds to the sharpening operation.

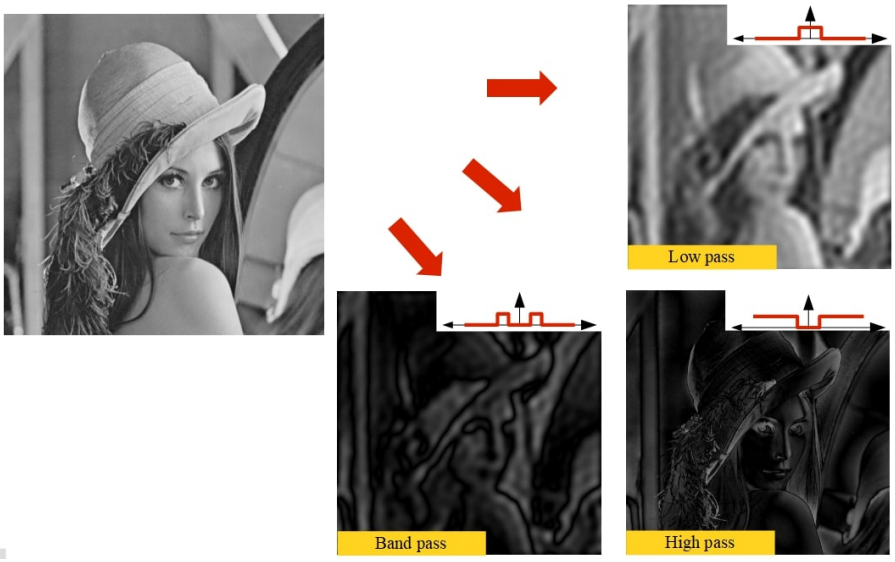


**Figure 3.5:** *Different filtering methods in the frequency domain.*

It is important to note that the two-dimensional Fourier transform of an image is also a two-dimensional array, which can be represented as an image. Since each frequency component has two values—an amplitude and a phase—usually, only the amplitude is visualized, as it contains more easily interpretable information for humans. In this representation, the origin corresponds to the zero frequency or constant component, and moving towards the edges of the image represents higher and higher frequencies. It is important to note that the resulting image is centrally symmetric.

From the images above, we can easily observe a fundamental drawback of applying an ideal low-

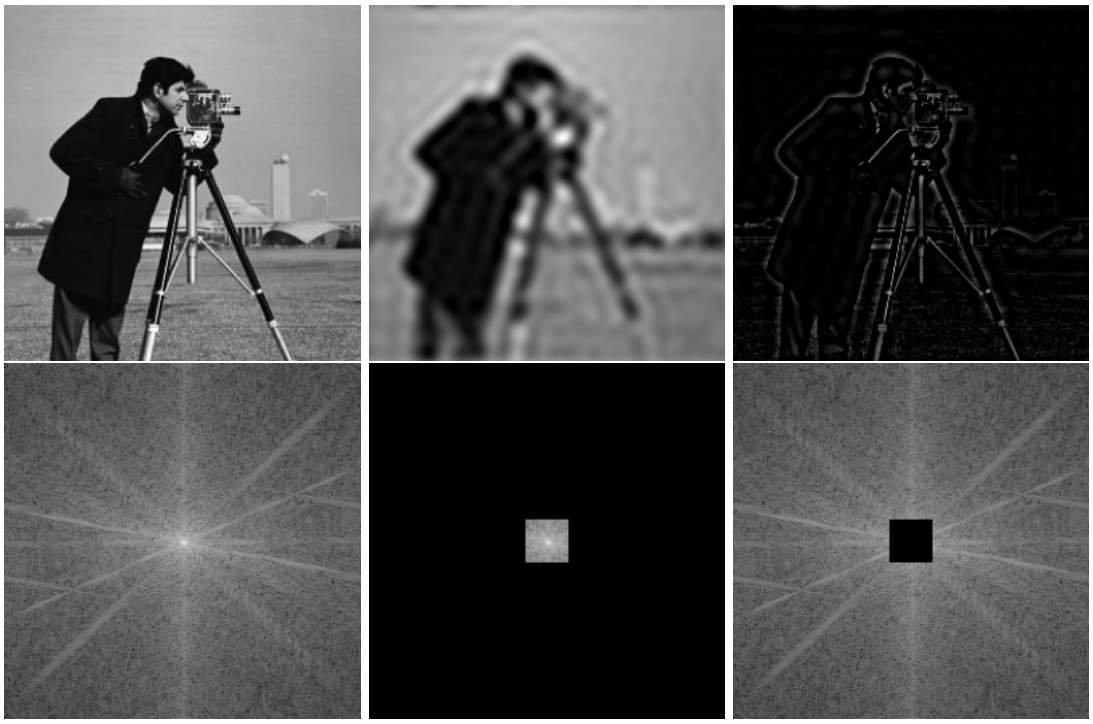

**Figure 3.6:** *Filters and spectra: original image and its spectrum (left), low-pass filtered image and spectrum (middle), high-pass filtered image and spectrum (right).*

pass filter. Although this is one of the easiest filter types to implement for images (unlike in control systems, where the concept of causality cannot be interpreted due to the absence of a time dimension), the resulting image may show wavy, ring-like artifacts. If we recall the square wave example presented at the beginning of the lecture, the reason becomes apparent: by completely eliminating the high-frequency harmonics, we obtain a waveform resembling a square wave composed of only a few sine waves.

To address this problem, more complex filters with smoother spectra are typically used. Examples of this include the trapezoidal filter and its smoother version, the Butterworth filter.

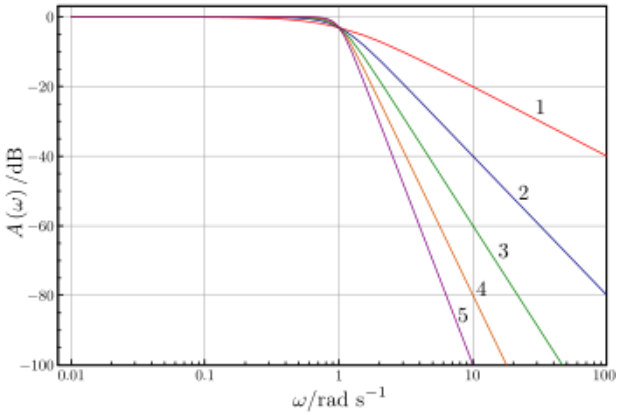


**Figure 3.7:** *The amplitude spectrum of a Butterworth filter at different orders.*

### 3.3.2 Convolutional Filters

A crucial relationship between the Fourier domain and the spatial domain is that convolution in the spatial domain simplifies to a pixel-wise multiplication in the frequency domain. This means that

various convolutional filters can be implemented much more efficiently in the frequency domain. This is particularly advantageous if we wish to apply multiple filters to an image because we only need to perform the Fourier transform and its inverse once, and the computational cost of the inexpensive filtering operations pays off.

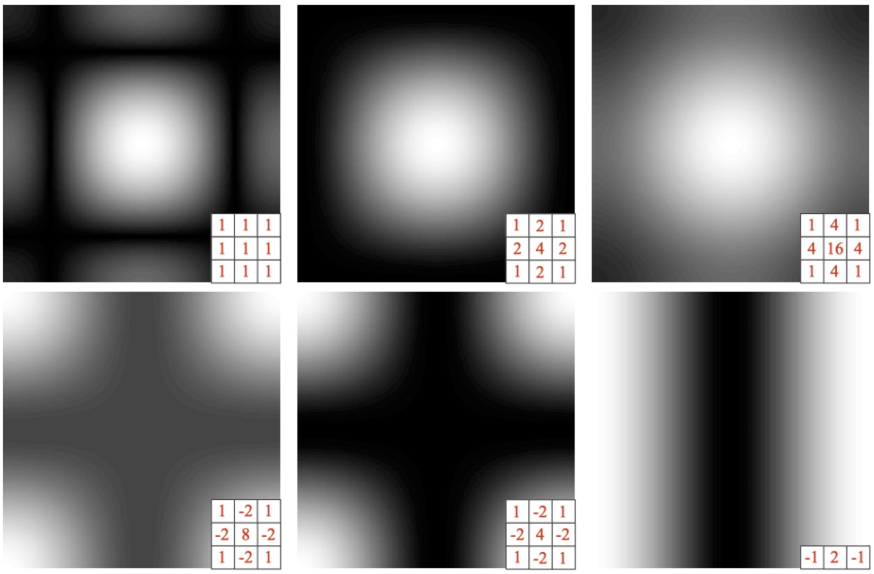


**Figure 3.8:** *The spectra of different convolutional filters.*

Earlier, we mentioned periodic noise, which typically arises from some form of electromagnetic interference in the image. Such noise cannot be filtered out using smoothing filters. However, in the frequency domain, we can easily locate and suppress the frequency responsible for the noise, and it's sufficient to suppress only the appropriate directional component.

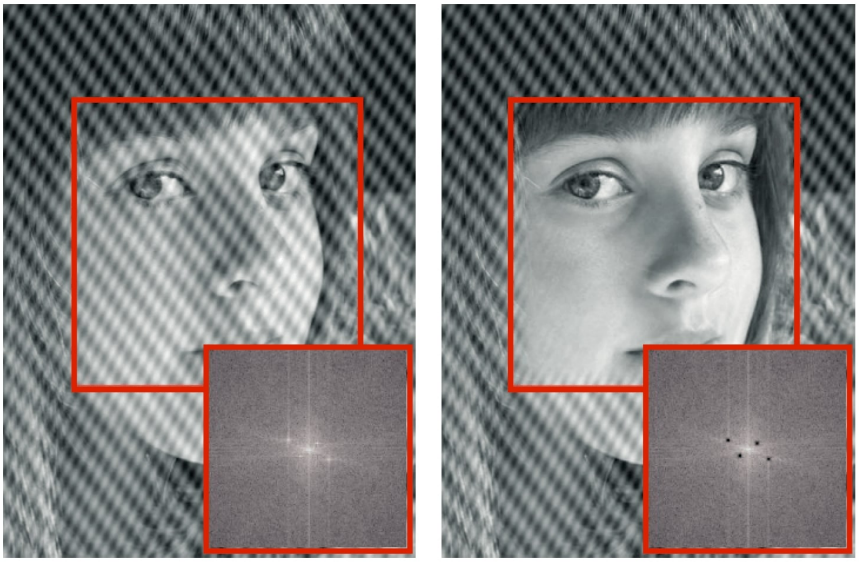

**Figure 3.9:** *Periodic noise filtering in the frequency domain.*

**Application:** Another potential use case is the verification of the orientation of certain documents, especially printed texts. In an image of printed text, the alternating darker lines and lighter spaces between lines will create a dominant frequency that is clearly observable in the image spectrum. By analyzing the direction of this frequency component, we can verify if the text appears horizontally in the image, which greatly enhances the accuracy of optical character recognition (OCR) algorithms.

## 3.4 Cosine Transform

It is important to mention a procedure similar to the Discrete Fourier Transform (DFT) called the Discrete Cosine Transform (DCT). In DCT, we also treat the image as a periodic signal, but we assume that the signal repeats by reflecting at the image edges, and we transform this reflected signal. As a result, our original signal becomes symmetric, and the resulting frequency spectrum will consist of only real values.

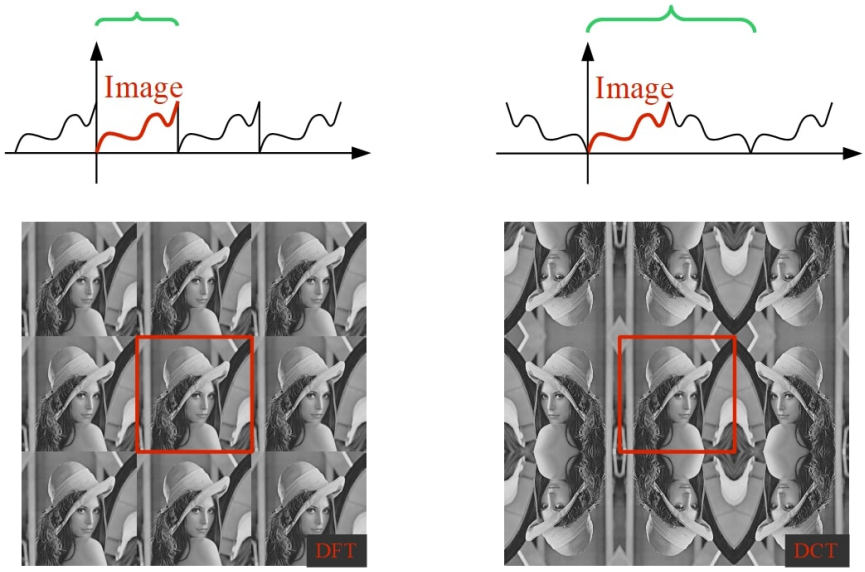


**Figure 3.10:** *The difference between DFT and DCT.*

This property has several advantages: storing real values requires half the memory, resulting in a significantly more compact representation. Another advantage is that no false discontinuities are introduced into the signal due to the reflective repetition. In the case of DFT, these discontinuities would appear as high-frequency components, but with DCT, they do not. Another advantage of this procedure is that the definition of the DCT involves a much simpler formula:

$$D(u,v) = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} I(x,y) * \cos\left[\frac{\pi}{W}(x+\frac{1}{2})u\right] \cos\left[\frac{\pi}{H}(y+\frac{1}{2})v\right] \tag{3.4}$$

### 3.4.1 FCT

It is worth noting that the FFT operation can be used to compute the cosine transform by simply constructing the symmetric image function through reflection and then performing the FFT algorithm on it. Afterward, by taking the real part of the transformation, we obtain the cosine-transformed values. It is worth mentioning that there are specialized algorithms for DCT that avoid performing the unnecessary operations resulting from the symmetry.

### 3.4.2 JPEG

It is worth mentioning that one of the most successful image compression formats, JPEG, uses the DCT algorithm, specifically its Type II variant, to compress images. JPEG is a lossy compression method that often achieves a compression ratio of 1:10 for real images while making the loss of information almost imperceptible to the human eye. This exploits a basic characteristic of human vision: we are less sensitive to sudden intensity changes.

Furthermore, human sensitivity to sudden changes differs between color and intensity perception. Since the cone cells that detect colored light are much more sparsely distributed on the retina than the rods, color vision has an even lower resolution. Therefore, it is more efficient to store the color information at a lower resolution than the brightness/luminance component. However, in conventional RGB images, this component cannot be separated. To address this, JPEG compression uses the YCbCr color space, halving the resolution of the Cb and Cr color components in one or, most commonly, both directions.

After this, the JPEG algorithm divides the image into 8x8 blocks (in the case of reduced resolution for color channels, these are 16x16 blocks), subtracts the average of the pixels in each block, and then performs the discrete cosine transform separately on each block, generating their frequency representation.

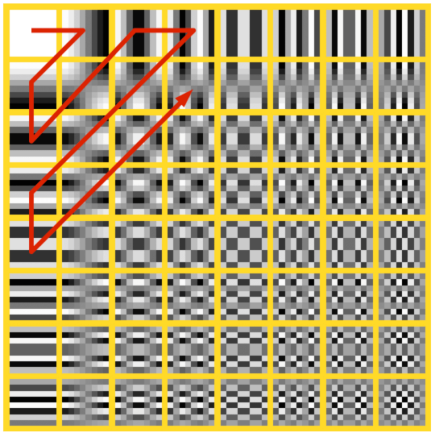

**Figure 3.11:** *The principle of JPEG compression.*

Afterward, the resulting floating-point numbers are divided by a predetermined constant for each element of the 8x8 cosine-transformed block (these constants are based on the characteristics of the human eye, exploiting the fact that contrast sensitivity deteriorates with increasing frequency, thus saving significant space for high-frequency coefficients), and then rounded to the nearest integer, where only 8 bits are needed for each element. However, due to quantization, many higher-frequency elements will be set to zero, meaning they no longer need to be stored. To maximize this, we index through the transformed elements diagonally, always leaving the highest frequencies for last. As a result, it is often sufficient to store only the first few values of the transform.

It is worth noting that because of the omission of high-frequency components, JPEG compression tends to blur the edges to some extent, and, similar to the ideal low-pass filtering, various ring-like artifacts (artifacts) may appear around the edges. Although these are rarely visible to the naked eye, an edge detection algorithm may easily detect false edges in such image regions. Generally, for artificial images (geometric figures, printed text), it is more appropriate to use PNG, EPS, or similar compression formats.

## 3.5 Applications

Beyond filtering and compression, frequency domain representation has many other applications. One such example is the conversion of various mosaic or halftone images into real images. In such cases, the repeating patterns from the mosaic elements are typically very distinct in the frequency domain, making them easy to filter.

### 3.5.1 Shape Recognition

Fourier transformation also provides the possibility of recognizing certain simple shapes. A good example of this is the problem of Optical Character Recognition (OCR). Printed characters tend

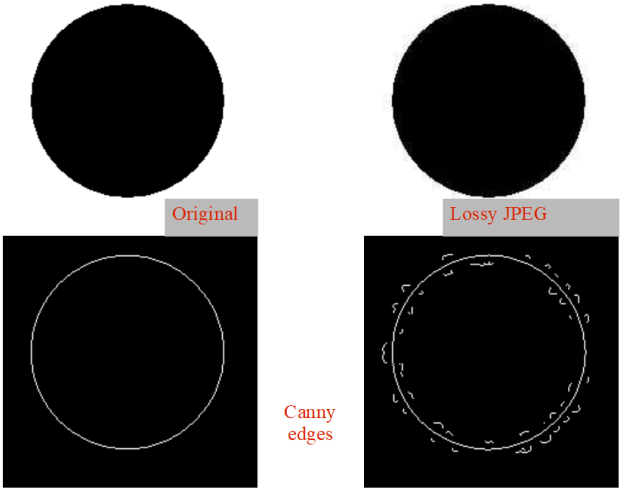


**Figure 3.12:** *The effect of JPEG compression on edges.*

to have well-defined shapes, which result in unique and regular spectra. This can easily be used for recognizing characters.

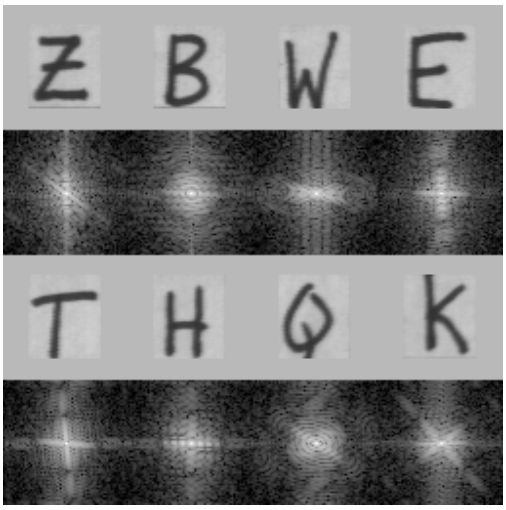

**Figure 3.13:** *The spectra of different printed characters.*

### 3.5.2 Deconvolution

It is important to note that frequency domain representation allows for the simple inversion of the convolution operation, called deconvolution. In deconvolution, the image spectrum is not multiplied but divided by the filter's spectrum. This method can, for example, compensate for the blurring effect caused by a poorly adjusted focus. A more advanced form of deconvolution is Wiener deconvolution, which, in addition to deconvolution, also performs frequency-dependent filtering, thereby suppressing any possible noise that may arise.

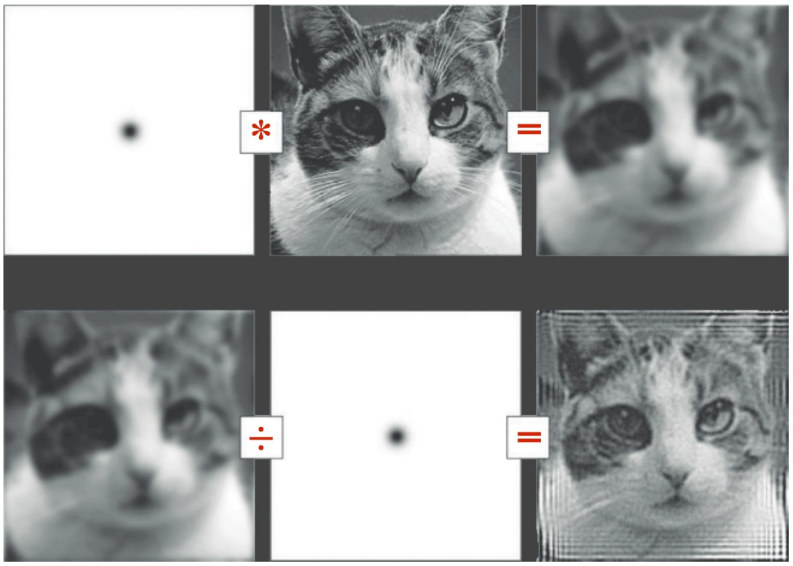

**Figure 3.14:** *The principle and result of deconvolution.*

# 4 Image Features

## 4.1 Image Matching, Feature Types

In computer image processing, our primary task is to capture key details within the image that can later be used for higher-level tasks. There are numerous types of image features, one of which—edges—was discussed in the previous lecture. The topic of the current lecture covers slightly more complex, and consequently, more computationally intensive but more robust features.

**Figure 4.1:** *Applications of image matching: Object identification and estimation of relative position (left). Panorama stitching (right).*

## 4.2 Intensity

The simplest type of image features are the intensity—or color—values themselves. No special algorithm is needed to determine these, making their generation essentially cost-free. Naturally, these features are also the least robust, so they are only useful in special cases.

### 4.2.1 Template Matching

Despite this, it often happens that we have a task where, in a relatively controlled environment (static background and lighting), we need to detect a known, non-changing object. In such cases, the template matching algorithm can be used. This procedure is very simple: first, we create a reference image of the object to be detected, then apply this template to the image in all possible positions, and afterward, we calculate some matching function between the image and the template. Detection is signaled at the positions of the extreme values of the matching function.

In practice, two types of matching functions are commonly used for template matching. One of them is the sum of squared differences between the pixels of the template and the image patch, also known as the L2 distance, where we look for the minimum points. A more interesting solution, however, is convolutional/correlation matching, where convolution between the template and the image is used as the metric. A fundamental property of convolution is that its result will be large in absolute value when the filter resembles the image patch it covers or its inverse. This can be illustrated by edge detection operators, discussed in a previous lecture, which indeed resemble an edge in the image.

$$E_{L2}(x,y) = \sum_{x'}\sum_{y'}(I(x+x',y+y') - T(x',y'))^2$$
$$E_{CC}(x,y) = \sum_{x'}\sum_{y'} I(x+x',y+y')T(x',y') \tag{4.1}$$

The main difference between the two similarity measures is that in the convolution-based solution, if we signal detection at both extremes of the response, we will also detect the inverse of the template. This can be useful, for example, in recognizing letters, where we can identify a black-lettered template on a white background as well as light-colored letters on a dark background. One of the main weaknesses of the template matching method is its sensitivity to rotation, scaling, and distortions, so if these occur, a separate template must be created for each scale and orientation, and the procedure repeated with all templates, negatively affecting speed.

**Application:** One of the most important applications of template matching is Optical Character Recognition (OCR). In this case, using the previously described method, with the text properly aligned, we can run a separate template for each printed character to locate occurrences of the respective letters, thus obtaining the text. The applications of OCR are endless, ranging from converting scanned documents into text to automatic reading of various identification documents.

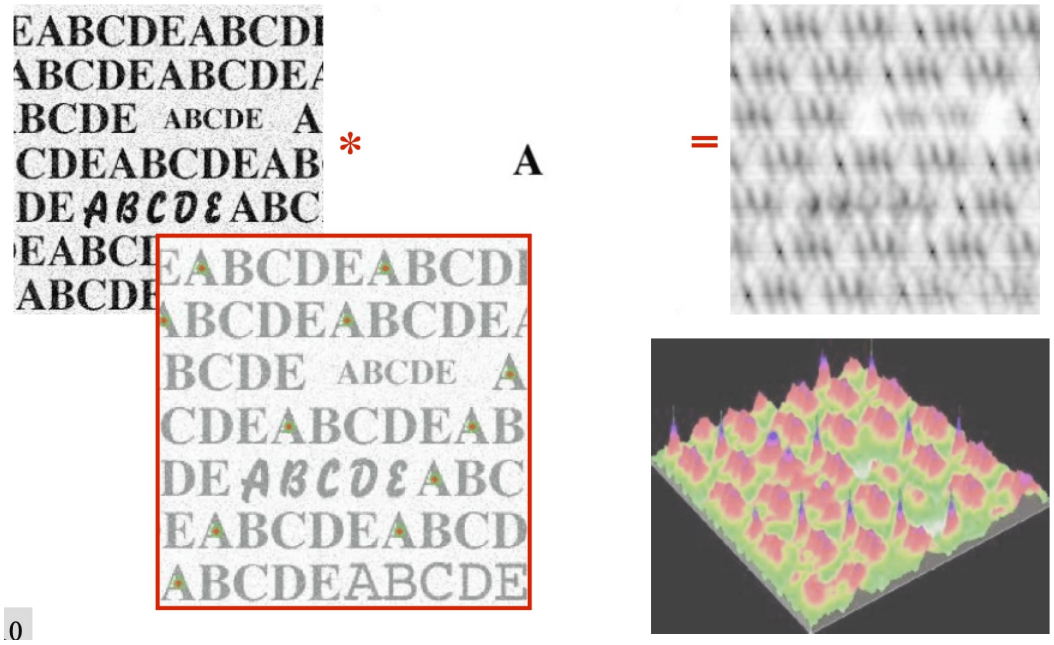


**Figure 4.2:** *Application of TM in OCR.*

Another important application is in the fields of virtual and augmented reality, where input devices are often implemented by placing some special (typically black-and-white) marker patterns on objects. These patterns are chosen such that they do not appear on real-world objects, making marker localization simple and robust using template matching. This method is commonly used in haptic augmented reality systems, where the basic principle is to interact with the virtual environment using specific real objects.

## 4.3 Optical Flow

Motion detection, segmentation, and determination of the magnitude and direction of movement also use intensity values as features in widely used optical flow algorithms. The basic principle of the algorithm is that it approximates the image locally at each pixel with a linear plane, and from the slope of this plane and the intensity changes between two images, it infers the amount of

motion. One important consequence of this is the so-called aperture problem: in certain types of image regions, it is impossible to determine the exact displacement based on local image patches.

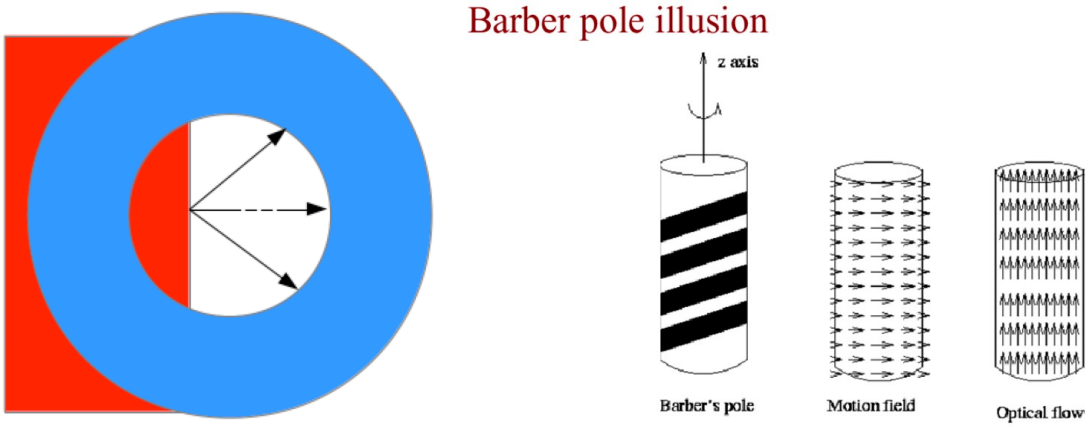


**Figure 4.3:** *The aperture problem (left). Local motion detection also plays a role in human vision, as illustrated by the barber pole illusion (right). Here, when rotating a cylinder painted with slanted stripes, it appears as though the stripes are moving upward, even though the motion is lateral. We will see that optical flow behaves similarly.*

Optical flow assumes that the intensity of objects in the image does not change between two images, only a displacement occurs. This can be expressed as follows, approximating the image with a Taylor series:

$$I(x,y,t) = I(x+dx, y+dy, t+dt) = I(x,y,t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \tag{4.2}$$

Where $I(x,y,t)$ cancels out:

$$\begin{gathered} I_x dx + I_y dy + I_t dt = 0 \\ I_x \frac{dx}{dt} + I_y \frac{dy}{dt} + I_t = I_x u + I_y v + I_t = 0 \end{gathered} \tag{4.3}$$

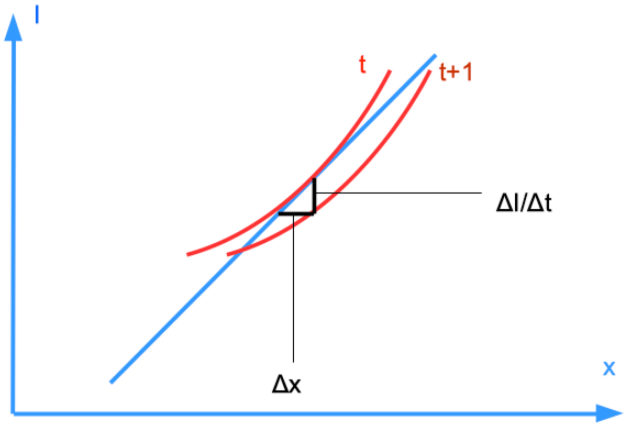


**Figure 4.4:** *Determining optical flow in one dimension: locally approximating the image with a line, the estimated displacement can be easily calculated by knowing the temporal change in a given pixel.*

Where $I_x, I_y$, and $I_t$ are the derivatives of the image in the spatial and temporal directions, and $u$ and $v$ are the velocities in each direction. The image derivatives can be computed numerically

using filters and differencing, but even so, there is only one equation for two unknowns, meaning the optical flow equation cannot be solved unambiguously. In reality, the simple flow method can only determine the component of the motion vector that lies in the direction of the image gradient as follows:

$$d = \frac{I_t}{\sqrt{I_x^2 + I_y^2}} \tag{4.4}$$

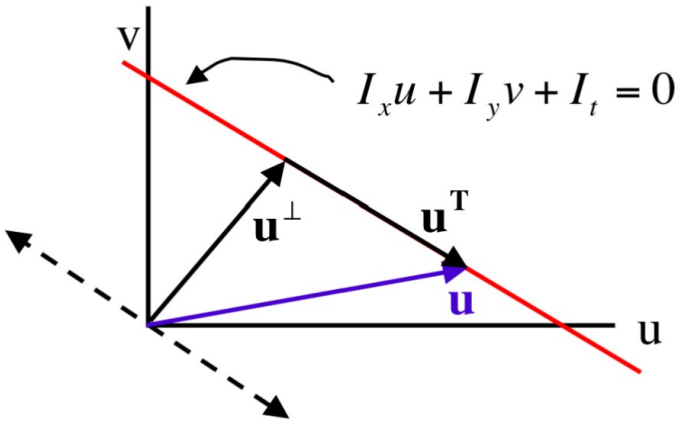


**Figure 4.5:** *The solution space (red line) defined by the intensity flow equation, showing the components of the solution.*

### 4.3.1 Lucas-Kanade Method

In practice, one of the widely used solutions to this problem is the Lucas-Kanade optical flow algorithm. This method makes one more assumption, namely that pixels located close to each other on the image are highly likely to move together. Based on this, the Lucas-Kanade method seeks to solve the optical flow equation not only at a given pixel position but also in its surrounding area. It applies the least squares method as follows:

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tn} \end{pmatrix} \tag{4.5}$$
$$X\vec{u} = Y$$

Here, in the equation above, the derivatives are known, and we are searching for the displacement vector such that the equation is solved with the smallest possible error. To achieve this, we need to solve the system of equations using the least squares method, where the optimal displacement is:

$$\vec{u} = (X^T X)^{-1} X^T Y \tag{4.6}$$

It is important to note that the matrix $X^T X$ in this equation is the same local structure matrix used in the previously described KLT corner detector (this is not surprising, as the "Kanade" in the KLT detector is the same Kanade as in the Lucas-Kanade method). If we recall the discussion on the local structure matrix, we know that the eigenvalues of this matrix express the magnitude of the smallest and largest changes in the image. For this matrix to be invertible, both eigenvalues must differ significantly from zero, meaning that the Lucas-Kanade method can only be evaluated at corner-like points. This is why this method is also known as a sparse optical flow method since it can only be computed at a few points in the image.

### 4.3.2 Farneback Method

There also exists a dense optical flow algorithm capable of determining the flow direction at every point. This procedure is attributed to Gunnar Farneback. The core idea of this method is to approximate the image locally, not with a linear surface (plane), but with a quadratic polynomial. Similar to the original flow algorithm, it assumes no intensity change between the two frames, only displacement. By computing and comparing the polynomials on both frames, the displacement of individual pixels can be determined.

$$I_1(x) = x^T A_1 x + b_1^T x + c_1; \quad I_2(x) = x^T A_2 x + b_2^T x + c_2 \tag{4.7}$$

Assuming that the two images are the same, with only a displacement $d$ between them:

$$I_2(x) = I_1(x - d) = (x - d)^T A_1 (x - d) + b_1^T (x - d) + c_1 \tag{4.8}$$

By grouping this expression by powers of $x$:

$$I_2(x) = x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - B_1^T d + c_1 \tag{4.9}$$

From this, we can express the equality of the $b$ coefficients of the two polynomials, and from this, $d$ can be derived:

$$b_2 = b_1 - 2A_1 d \quad \rightarrow \quad d = -\frac{1}{2} A_1^{-1} (b_2 - b_1) \tag{4.10}$$

In practice, however, a slightly modified version is used: globally, the image cannot be well approximated by a quadratic polynomial, so this approximation is applied locally, and the displacement is determined for each position. Additionally, in Farneback's method, the pixels from the surrounding area are used for the polynomial estimation at each position. This way, using the least squares (LS) method, a much more robust estimate is obtained, minimizing the effects of noise.

### 4.3.3 Iterative and Pyramid Methods

A significant challenge in optical flow methods arises when the displacement is so large that higher-order terms can no longer be neglected. In such cases, the error in motion estimation becomes substantial, leading to completely incorrect motions over time. To address this issue, iterative optical flow estimation is often used. The essence of this solution is that after determining the initial optical flow, the original image is transformed according to the calculated motion vectors, and then a new optical flow estimation is performed between this intermediate image and the second image. This process is repeated until the relative displacement obtained becomes insignificant. By summing the intermediate displacements at the end of the algorithm, the actual displacement is obtained.

Iterative optical flow is often combined with the pyramid method. The first step of the procedure is to create a coarse-to-fine resolution image pyramid using resizing. Then, the flow is estimated iteratively on the image with the smallest resolution. The resulting flow will be somewhat imprecise, but large motions can be easily determined since, due to the lower resolution, they appear significantly smaller on this image. Afterward, the iterative method continues on each higher-resolution image, using the already computed flow as the initial value. In this way, the coarse flow estimate from the previous level is refined at each level.

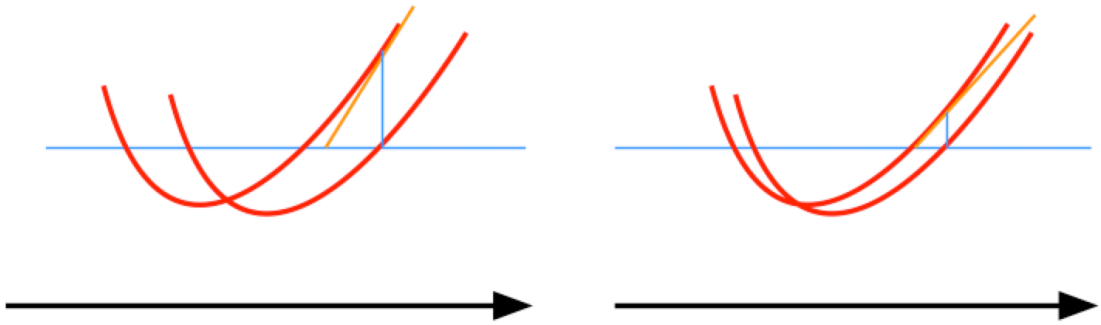

**Figure 4.6:** *The principle of iterative optical flow: after the first iteration (left), the image is brought closer to the displaced version, resulting in a better estimate in the second round (right).*

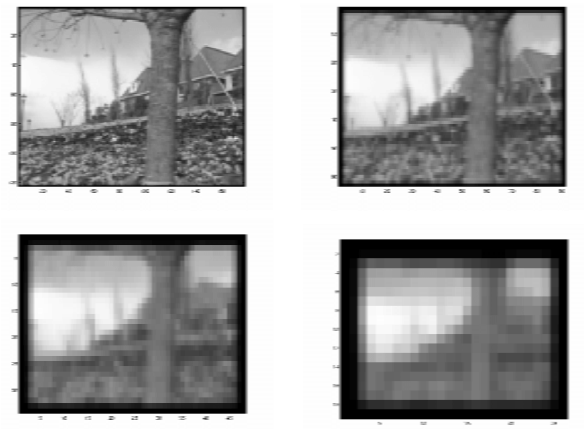

**Figure 4.7:** *The image pyramid.*

### 4.3.4 Application

Optical flow is commonly used in many areas, one of the significant ones being 3D reconstructions with a moving camera. Using the algorithm, it is possible to create spatial recordings with a simple camera that would not otherwise be suitable for this task. To do this, a video of the object is taken while walking around it, and then the displacement of individual pixels is calculated using optical flow. The displacement magnitude can then be used to infer the distance from the camera (pixels from distant objects move more slowly in the image).

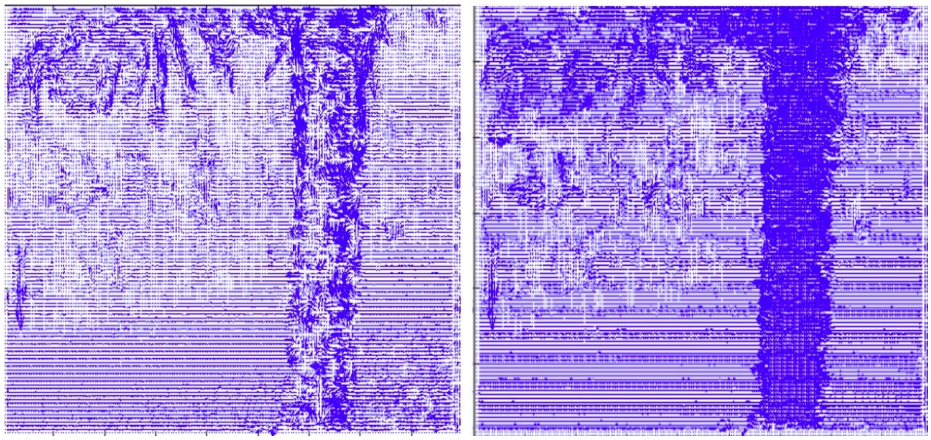

**Figure 4.8:** *Optical flow with (right) and without (left) the pyramid method.*

Another interesting application is the classification of various motion events. Moving objects in the image can be categorized based on their flow patterns into objects moving in different directions, rotating, or approaching or receding from the camera. This is extremely useful when we want to detect or avoid an approaching object that might collide with the equipment housing the camera.

## 4.4 Edge Detection

Edges in an image are defined as significant, unidirectional changes in intensity between adjacent pixels. A key characteristic of edges is that the intensity changes only in one direction, while remaining constant in the other, and that the transition is sharp and abrupt. However, in reality, due to various image defects, noise, and finite resolution, the ideal edges become blurred and transitions occur more gradually. Additionally, local changes in other directions are also possible.

### 4.4.1 Derivative-Based Edge Detection

The simplest method for detecting edges is to compute the derivative of pixel intensities in specific directions, which can be done numerically, similar to convolutional filtering. As we move through the image, we calculate the difference between a pixel and its neighboring pixels to the right and below, yielding the derivatives in the x and y directions. By summing the squares of these two derivatives, we obtain the total magnitude of the derivative, which can be interpreted as a measure of the ”edginess” at that point.

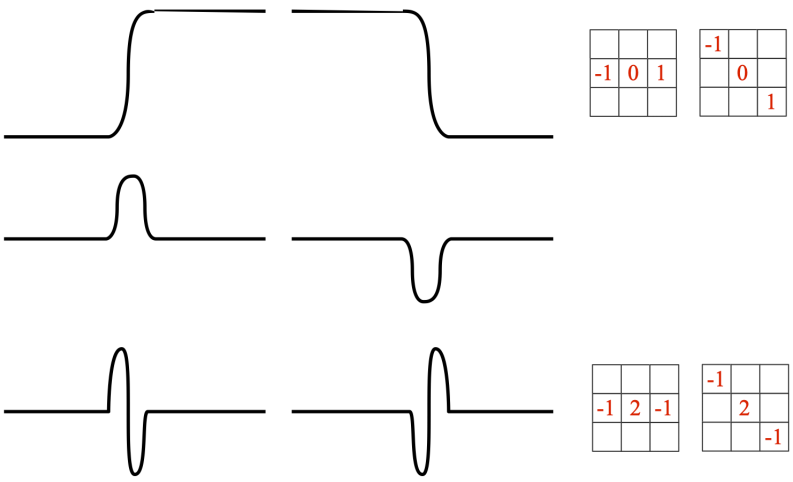


**Figure 4.9:** *Applying derivative filters on an image.*

The method described is fast and simple, but its major drawback is that random image noise can create false derivatives, making the resulting edge map noisy. This can be avoided by pre-smoothing the image with a Gaussian filter to reduce noise. In practice, to speed up the algorithm, we take advantage of the fact that both the derivative and Gaussian filters are linear operations, allowing them to be combined into a single operation. Thus, instead of applying two separate filters sequentially, we perform a single convolution with a filter defined by the derivative of the Gaussian filter, which gives the same result.

In addition to the Gaussian derivative filter, other convolutional edge detection filters are commonly used, such as the Prewitt and Sobel operators, which are directional edge detectors. To detect edges of any orientation, both versions of the operator must be run on the image. The primary difference between the two is that the Sobel operator smooths in the direction opposite to the edge, making it less sensitive to noise.

One of the biggest drawbacks of first derivative-based filters is that the edge will appear blurry due to the smoothing, making precise localization difficult. This can be addressed by taking the second

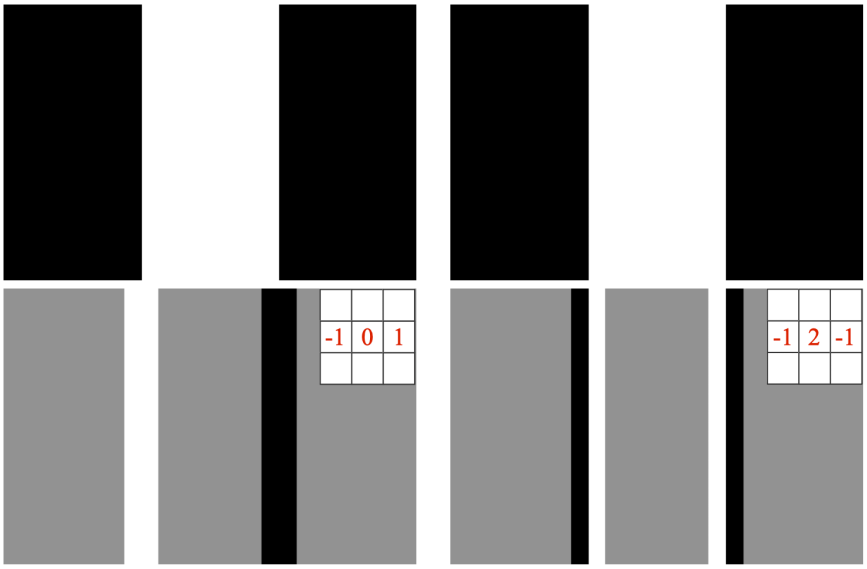


**Figure 4.10:** *First and second derivatives of an edge.*

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

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

**Figure 4.11:** *Prewitt (left) and Sobel (right) operators for vertical edges.*

derivative of the edge image and finding the zero-crossings of the derivatives. This operation is performed in a single step using a second derivative convolution filter called the Laplace filter, which is often referred to as the sombrero hat filter due to its shape.

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 4 | -1 |
| 0 | -1 | 0 |

| | | |
|---|---|---|
| -1 | -1 | -1 |
| -1 | 8 | -1 |
| -1 | -1 | -1 |

**Figure 4.12:** *Laplace filter in 4- and 8-neighbor versions.*

In practice, the Laplace filter is sometimes replaced by the difference of two Gaussian filters with different standard deviations, a solution known as the DoG filter (Difference of Gaussians), which is widely used in many applications. Often, multiple DoG filters of different sizes are applied simultaneously to an image. To speed up the process, the images are first smoothed with different Gaussian filters, and then the filtered images are subtracted from each other. Due to the linearity

of the operations, the result remains the same.

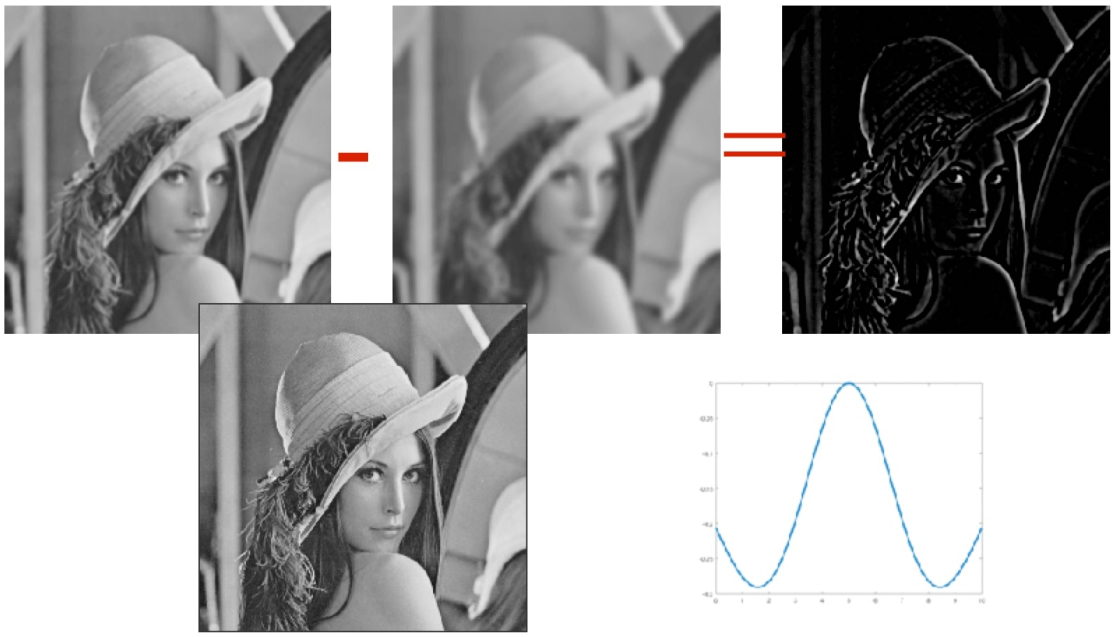

**Figure 4.13:** *Principle of the DoG filter.*

**Application:** In many situations, we may need to automatically capture an image of an important object. Automation is necessary when we cannot control when and where the object appears in the image, or when we need to capture a large number of images. In such cases, we cannot guarantee that the image will be well-focused, which may result in blurred details that hinder further processing.

To avoid this, we can use automatic focus detection, where the focus is adjusted until the sharpness of the image, measured by the number of significant edges, is maximized. Blurred images have fewer strong edges, resulting in smaller gradients. Thus, a simple measure can be derived: the more pixels where the derivative exceeds a certain threshold, the better the focus of the image. It is important to note that this method can only compare images with the same content.

### 4.4.2 Canny Algorithm

Previous edge detection algorithms relied on different methods of calculating derivatives to measure the ”edginess” of each pixel. From that point on, it was up to the designer to decide the threshold above which a pixel would be considered an edge. Setting this threshold too high could miss less contrasted edges, while setting it too low would generate many false edges due to noise. The goal is to find a compromise between the two, but this solution is often imperfect.

The Canny algorithm, a state-of-the-art method for edge detection, addresses this dilemma. It is a multi-step process that begins by computing the horizontal and vertical derivatives of the image. Then, using these derivatives, the magnitude and direction of the image gradient (the direction of maximum intensity change) are determined for each pixel.

Next, starting from every edge-like point and following the gradient direction, the algorithm finds the point with the largest derivative value. These points are kept, while neighboring points with smaller derivatives are suppressed, resulting in sharp edges.

Finally, instead of using a single threshold, the algorithm applies two different thresholds to the gradient magnitude image. This produces two binary edge images. The first binary image, generated with the lower threshold, contains all potential edge points but also includes noise. The

**Figure 4.14:** *Edge images used by the Canny algorithm.*

second binary image, generated with the higher threshold, contains only strong edges but may be incomplete. However, it has minimal noise.

The Canny algorithm's strength lies in its final step, where it combines the two binary images to produce a much better quality edge image. It iteratively fills in the missing edges from the stricter image by adding neighboring edge points from the more permissive image, without copying noise.

## 4.5 Hough Transform

In edge detection, it is quite common to search for the boundaries of various simple shapes (rectangle, circle). In such cases, it may be useful to fit a line model to the detected edge points, allowing us to describe the points in a linear arrangement in the image with a parametric model, which greatly facilitates the detection of shapes. The difficulty of the problem lies in the fact that naturally only a subset of the detected edge points will fit to lines, and among those, many points may fit individual lines. Thus, we need to determine both which points fit a line and what parameters describe this line simultaneously. If we could answer either of these questions, the problem would be trivial.

One of the most popular algorithms for this problem is shape detection based on the Hough Transform. The Hough Transform is a coordinate transformation that maps image points from the conventional image coordinate system $(x, y)$ to a space defined by the parameters of lines $(r, \theta)$. In this space, a line is described by a single point (pair of numbers), where one element is the length of the perpendicular segment from the origin to the line $(r)$, and the other element is the angle of this segment with the x-axis $(\theta)$.

What is more interesting is how the points in the image are represented in the Hough space. This representation is done during the Hough Transform by mapping each point $(x, y)$ in the image to all lines for which the given point fits. Although a given point fits an infinite number of lines, it does not fit all possible lines. Therefore, the image of a specific point $(x, y)$ in the Hough space is given by the following curve:

$$x \ \cos\theta + y \ \sin\theta = r \tag{4.11}$$

That is, the image of a point in the Hough space is a sinusoidal curve. If the curves of two points in the Hough space intersect, it means that both points fit the line described by the intersection point of the two curves (recall: in the Hough space, a point describes a line).

From this relationship, a simple conclusion can be drawn: if there are many curves in the Hough space that intersect approximately at a point, it means that there are many points in the image space that fit the same line. Based on this, the Hough Transform line detection algorithm naturally

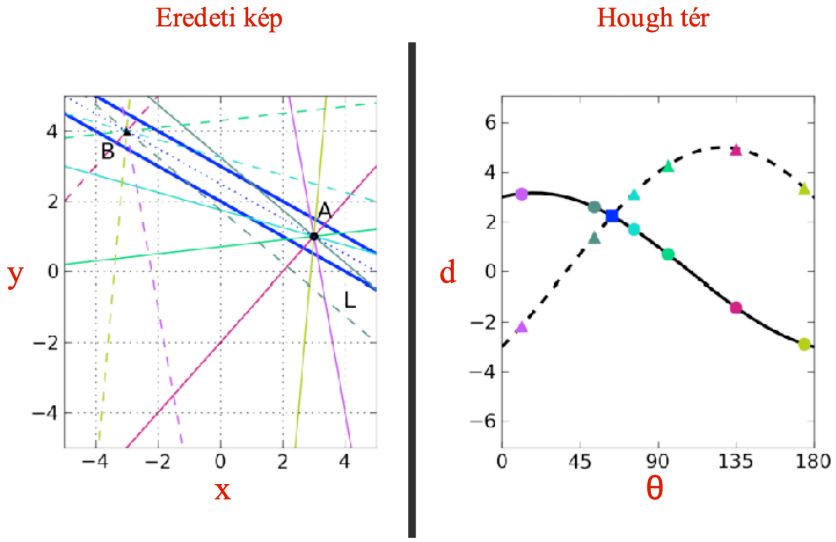


**Figure 4.15:** *The principle of the Hough Transform.*

follows: first, transform all edge points to the Hough space, then find the line on which the most fitting edge points are located. After that, remove these edge points from the Hough space, and then find the line fitting the most remaining points. This process is repeated until a line is found that fits at least a threshold number of points.

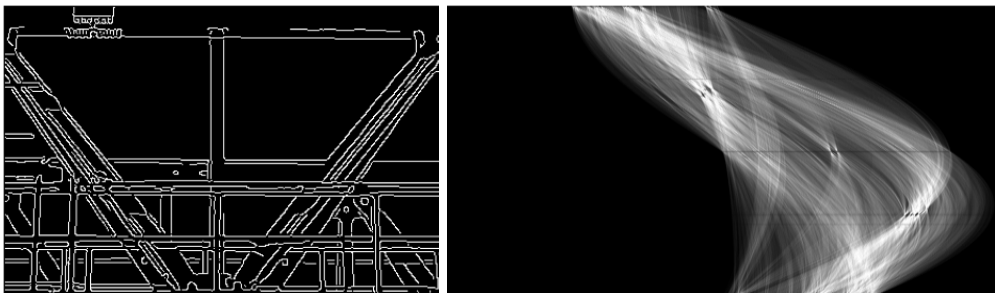

**Figure 4.16:** *A binary image and its Hough Transform.*

It is important to note that the Hough Transform can be applied not only to lines but also to any simple shape described by parameters. By using different types of Hough Transforms, we can detect various simple shapes. Besides lines, the Hough Transform is often used for detecting circles and ellipses. The generalized Hough Transform, which allows the detection of more general shapes, is also worth mentioning.

**Application:** The Hough Transform is frequently used to detect simple shapes composed of line segments in an image. This method can be easily applied, for example, to the automatic recognition of ID cards, which is one of the fundamental tasks in smart administration. In a picture of an ID card, the edges of the card typically create sharp, well-defined, straight edges, which can be easily detected by methods like the Canny edge detector. Subsequently, using the Hough Transform, these edges are converted into lines, and then we search for the rectangular object defined by these lines.

## 4.6 Corners

The first discussed image features were edges, which are simple and quickly extractable but moderately robust descriptors of objects found in images. One of the most significant limitations of

edges is that changes occur locally in only one direction in the image, making it impossible to track if the edge moves perpendicular to this direction.

This problem is addressed by the image feature discussed in this chapter, namely the image corner. Image corners are points in the image plane where significant intensity changes can be observed in all directions. Consequently, regardless of how the object containing the corner moves in the image, we can track the movement of the corner and, therefore, detect and follow the object that is relevant to us in some way.

One of the most widespread methods for detecting image corners is the Kanade–Lucas–Tomasi or KLT corner detector. The principle of this detector is that since we are searching for image segments that change significantly in all directions, we shift the surrounding neighborhood of the pixel being examined in all directions and compute the squared difference between the displaced and the original image segments. If this difference is significant in all directions, we have found a corner-like image segment.

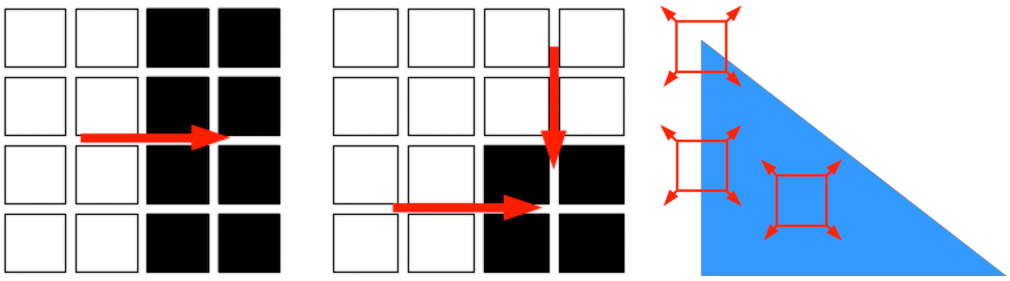

**Figure 4.17:** *Ideal image edge (left), corner (center), and the principle of corner detection (right).*

### 4.6.1 Local Structure Matrix

In practice, the KLT detector does not work as described, because performing this operation for every pixel would be computationally expensive. Instead, the KLT detector calculates the derivatives of the image in the x and y directions, then for each pixel, arranges the derivatives in its $n \times n$ neighborhood into an $n^2 \times 2$ matrix. The error in the displacement of the pixels can be formulated as follows:

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{4.12}$$

From this, the total error is:

$$||E||^2 = \vec{u}^T I^T I \vec{u} = \vec{u}^T H \vec{u} \tag{4.13}$$

Where $I_{x1}$ and $I_{y1}$ are the x and y derivatives of the first pixel in the neighborhood (the order of the pixels is irrelevant for the final result), and $H$ is the so-called local structure matrix of the given pixel. It is important to note that if we assume that the expected value of each derivative in the image is zero (a well-founded assumption since derivatives are just as likely to be negative as positive), then the local structure matrix is proportional to the covariance matrix of the derivatives in the image segment.

### 4.6.2 KLT and Harris

An important property is that the two eigenvectors of the local structure matrix define the two directions in the image plane where the image changes the most and the least. The magnitude of these changes is given by the eigenvalues $(\lambda_1, \lambda_2)$ corresponding to each eigenvector. This relationship provides an excellent method for detecting corners: Points where even the smallest change is still large (i.e., the smallest eigenvalue of the local structure matrix is significant) can be considered corner points.

During the operation of the KLT detector, the local structure matrix is computed for every element of the image, and a corner measure is assigned based on the smaller eigenvalue, which is used to determine whether a pixel is a corner point. It is important to note that the corner measure will be high for pixels around a corner, as image corners are not perfectly point-like phenomena. A simple solution is to consider the local maxima of the corner measures as the locations of the corners.

In practice, instead of the KLT detector, another method called the Harris detector is often used. The Harris detector also uses the local structure matrix to detect corners, but it defines the corner measure using the following formula:

$$\begin{gathered} R = det(H) - k * tr(H)^2 \\ k \in [0.04 - 0.06] \end{gathered} \tag{4.14}$$

The significant advantage of this calculation method is that it is much faster than computing the eigenvalues. Additionally, the Harris corner measure is also suitable for detecting edge-like image segments, as in such cases, the corner measure will be a large negative number. Another key difference between the two detectors is that although slower, the KLT detector's results are closer to human perception.

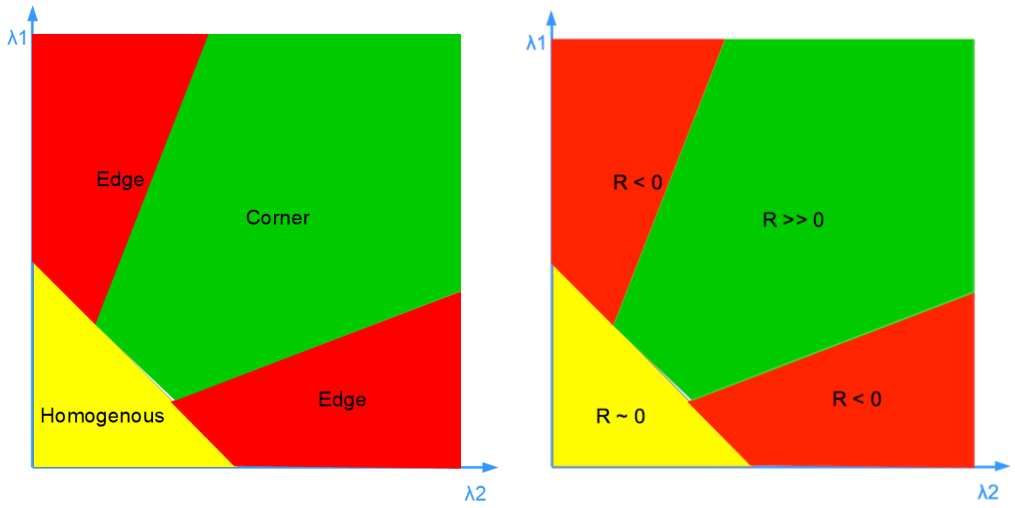


**Figure 4.18:** *Classification principles of KLT (left) and Harris (right) corner measures.*

## 4.7 Invariances

When discussing image features, it is crucial to mention the robustness requirements posed against these features. Our goal with these features is that if we see the same object in multiple images but different transformations occurred between the two images, the image features should be invariant to as many of these transformations as possible so that we can easily detect the same object in different images.

The most basic image transformation is the change in intensity, which is usually due to changes in illumination. There are two types of this transformation. The first is additive, where a constant is

added to the intensity values of the pixels. The other is multiplicative, where the intensity values of the pixels are multiplied by a constant. Other common transformations include rotation and changes in the object's scale, which arise due to images being taken from different viewpoints and distances. Perspective distortion, caused by changes in the viewpoint, can also be observed in images, best exemplified by the convergence of previously parallel lines.

Unfortunately, the KLT and Harris operators are invariant to only two of these transformations: additive intensity changes (since the derivatives of the image eliminate the constant term) and rotation (because rotation does not affect the eigenvalues of a matrix). Under multiplicative intensity changes, the derivatives are multiplied by a constant, so the corner measures change accordingly. Scaling can significantly affect corner detection results, as an object that appears corner-like in one image may appear as a rounded, edge-like object when magnified.

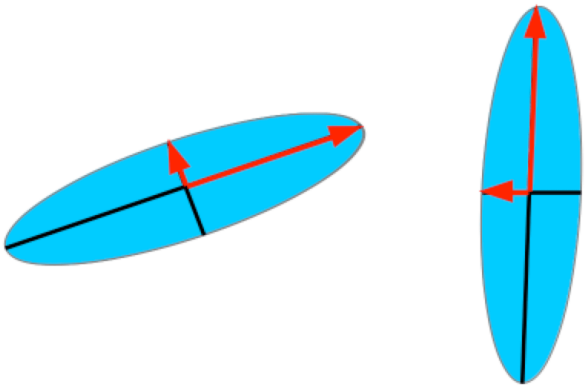

**Figure 4.19:** *The effect of rotation on eigenvalues.*

## 4.8 Complex Image Features

This problem is addressed by the scale-invariant feature transform, or the SIFT algorithm (Scale Invariant Feature Transform), which is invariant to all transformations except perspective distortion, thus producing a robust local descriptor widely used. Its principle is to search for corner-like points in the image, but instead of a single measure, it creates a descriptor code that invariantly describes the local environment of the corner points, allowing for comparison and matching with features found in other images.

### 4.8.1 SIFT Detector

The primary principle of the SIFT algorithm is that corner detection is performed along multiple scale factors, and each feature is assigned a scale variable used in creating the descriptor code, thus ensuring scale invariance. The corner detection is carried out using DoG filters (Difference of Gaussians) as mentioned in Chapter 3. This type of filter was previously known to us as an edge detector, but the response of the DoG filter will be maximal for impulse-like protrusions and indentations. It is important to note that if different-sized DoG filters are applied to a given impulse-like image region, the filter size that most closely matches the width of the impulse will yield the highest response.

As the first step, the SIFT algorithm runs several DoG filters of different sizes across the image and stores the responses. It then searches for local maxima in the filter responses, not only in the x and y coordinates of the image but also according to the filter sizes. The coordinate with the highest response will be the location of the corner point, and the size of the filter with the maximum response will be the associated scale factor. It is worth noting that SIFT does not settle for the maximum position quantized by discrete pixels and filter sizes but fits a polynomial locally

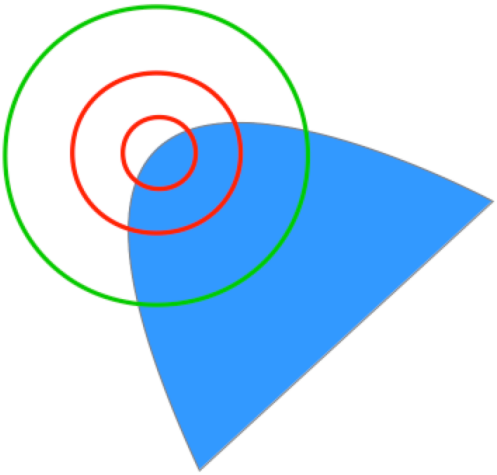

**Figure 4.20:** *Corner-like feature (blue) examined across multiple scales.*

to the response values and finds the maximum location. This method allows determining the corner point position with sub-pixel (i.e., below pixel size) accuracy, which can be a requirement for many applications.

It is worth noting that the procedure speeds up the filtering process in two ways: first, instead of running DoG filters, the algorithm applies Gaussian filters of various sizes to the images and then subtracts them from each other, thereby speeding up the process. Secondly, when the current Gaussian filter size reaches twice the original size, the algorithm runs the original filter on an image with half the resolution, obtaining the same result with approximately a quarter of the computation.

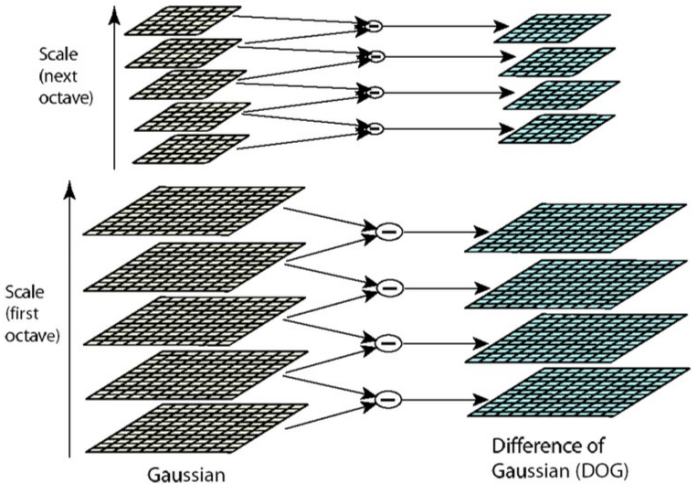


**Figure 4.21:** *The filtering scheme of the SIFT detector.*

### 4.8.2 SIFT Descriptor

The second step of the SIFT algorithm involves generating the descriptor code for the detected corner points. For each corner point, SIFT generates a 128-dimensional vector that describes the appearance of a $16 \times 16$ pixel neighborhood around the corner point. This $16 \times 16$ pixel neighborhood is always taken from an image resized according to the scale of the corner point, thus ensuring the scale invariance of the descriptor code. Since the descriptor code is created not directly from the intensity values of the neighborhood but from their gradients (a two-dimensional

vector formed by the x and y directional derivatives), it also ensures invariance to additive intensity changes, similar to the KLT method.

One of the most revolutionary ideas of the SIFT algorithm is ensuring rotation invariance. To achieve this, a histogram is created for the gradients in the neighborhood of the point, which represents the magnitude of gradients in each direction within this neighborhood. During histogram creation, the image gradients are categorized into 36 directions, dividing the circle into 10-degree intervals. Gradients pointing towards the interval boundaries are evenly distributed among adjacent bins. An important detail is that gradients of points far from the corner are given less weight.

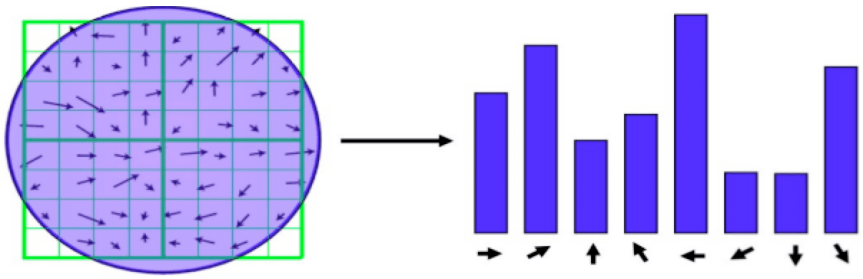

**Figure 4.22:** *Construction of the gradient histogram (the example histogram has 8 bins).*

The position of the maximum of the gradient histogram (i.e., the direction in which the image changes the most in this neighborhood) is found and this direction is assigned as the orientation of the corner point. The image is then rotated so that the orientation of the corner point points in a predetermined direction (e.g., vertically upward), and the final descriptor code is created from the rotated (and previously resized) $16 \times 16$ pixel neighborhood, thus ensuring invariance to rotation. It is important to note that if the maximum value of the gradient histogram is not unique, a separate descriptor code is created for each orientation close to the maximum.

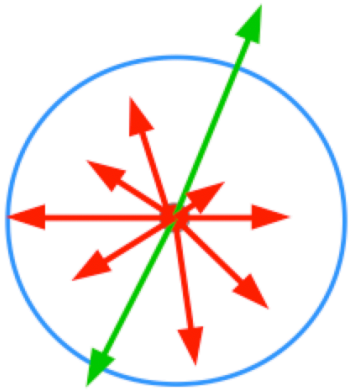

**Figure 4.23:** *Determining the SIFT orientation (not all 36 bins are shown).*

To create the descriptor code, the $16 \times 16$ pixel neighborhood is divided into 16 $4 \times 4$ pixel blocks, and a gradient histogram is created for each block separately as described above. The only difference is that each gradient histogram consists of 8 bins rather than 36, giving a resolution of 45 degrees. The values of these histograms are then concatenated into a single vector, resulting in a total of 128 values. This vector is normalized so that the sum of the squares of its elements equals one. This normalization ensures that multiplicative intensity changes, which would scale the gradients by a constant, do not affect the final descriptor code.

**Application:** Local image features have numerous applications. They are used, for example, for image stitching, which is crucial for panoramic imaging or 3D vision. One of the most straightforward applications is the recognition of rigid objects based on a reference image. In this case, we

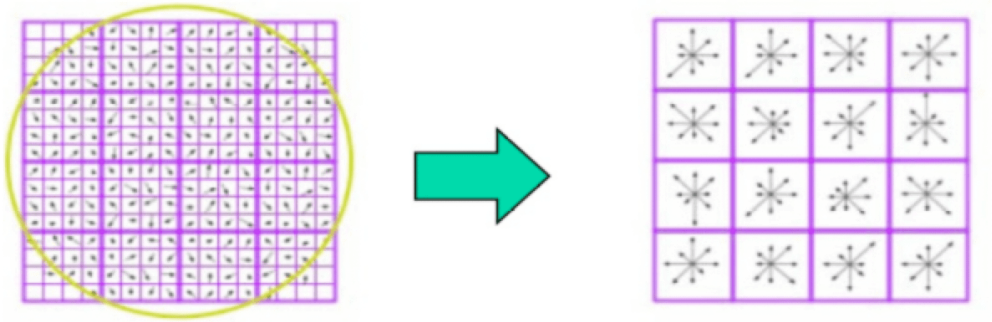

**Figure 4.24:** *Construction of the final descriptor vector.*

aim to recognize and localize objects that do not deform and have not too many variations. The local gradient histogram-based image features are first found in the reference image, and then the features in the current image are compared with these, matching them based on the similarity of their codes. If a large number of matches are found, the sought object has been located.

This method is often used when objects need to be searched for that the system designer can no longer influence, meaning that they cannot be made more recognizable. Such methods are frequently employed in virtual or augmented reality systems that use real-world objects, as well as various camera-based monitoring and tracking systems (e.g., spatial and traffic monitoring camera systems).

### 4.8.3 ORB Detector

By the end of the SIFT process, we have a feature detection and description method that is invariant to the transformations mentioned above. However, the method is computationally intensive and may not always run in real-time even on modern desktop computers (fortunately, the patent expired in 2020). Since the publication of the algorithm, several other methods have been developed that build on the basic idea of SIFT but require significantly less computation. Notable among these is the SURF method, which is highly parallelizable using graphics cards while maintaining similar robustness, and ORB, which can run in real-time while preserving invariance.

The ORB (Oriented FAST and Rotated BRIEF) algorithm results from the combination and enhancement of two other methods. The ORB algorithm uses the FAST (Features from Accelerated Segment Test) corner detector for keypoint detection. The essence of the FAST method is to define a circle in the neighborhood of the point being examined and select the pixels located on this circle for corner detection. The algorithm then counts how many of the 16 selected pixels have intensities that differ from the central pixel by a certain threshold. A high count indicates that a corner-like feature has been detected.

It is worth noting that if the threshold value for different pixels is 12 or higher, non-corner points can be quickly excluded by checking in the four cardinal directions. Alternatively, non-corner pixels can also be quickly excluded using a decision tree.

To ensure scale invariance, a scale pyramid is also used here to perform corner detection at multiple scales. Similar to the SIFT method, the final descriptor is created using the image corresponding to the selected scale factor for the corner point. It is also important to note that the FAST detector tends to detect edge-like features, so it is advisable to filter the detected keypoints using KLT cornerness. In this case, however, only a few points need to have their cornerness calculated, which involves negligible computation.

A significant limitation of the FAST detector is that it cannot assign orientation to the detected keypoints. To address this, a separate orientation metric must be defined. The FAST algorithm solves this by calculating the centroid of the pixels in the keypoint's neighborhood, where each pixel's relative mass is determined by its intensity. The formula for this is:

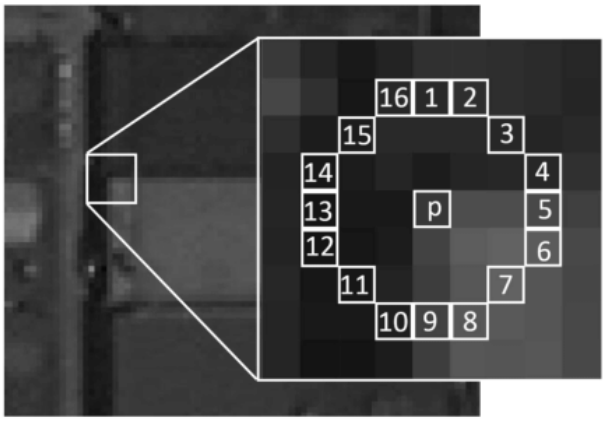


**Figure 4.25:** *The operating principle of the FAST detector.*

$$x = \frac{\sum xI(x,y)}{\sum I(x,y)}; \qquad y = \frac{\sum yI(x,y)}{\sum I(x,y)} \tag{4.15}$$

The ORB feature orientation is then given by the direction of the vector from the keypoint to the centroid.

### 4.8.4 ORB Descriptor

The ORB algorithm uses the BRIEF (Binary Robust Independent Elementary Features) descriptor method for generating the descriptor vector. A fundamental property of BRIEF is that, unlike the SIFT descriptor which uses floating-point numbers, it describes the image patch with a vector of binary elements. To achieve this, the algorithm selects $n_d$ (256 in the case of ORB) predefined point pairs in the keypoint neighborhood and examines each pair to determine if the intensity of the first point in the pair is greater than that of the second point. The BRIEF descriptor is thus represented as:

$$\left[sign(I(x_1^{(1)}) - I(x_2^{(1)})) \quad sign(I(x_1^{(2)}) - I(x_2^{(2)})) \quad \dots \quad sign(I(x_1^{(n_d)}) - I(x_2^{(n_d)}))\right] \tag{4.16}$$

Let us examine the invariances of the BRIEF descriptor. Since the descriptor only considers the difference in intensities between individual pixels, it is trivially invariant to both types of intensity changes. Scale invariance is handled by using a multi-scale detection and descriptor generation method, similar to SIFT. However, rotational invariance poses a problem, as predefined point pairs will fall on different pixels in a rotated image patch.

This problem could be easily addressed by rotating the image patch, but this is computationally expensive. Instead, it is much simpler to quantize orientations in 12-degree increments (yielding 30 possible orientations) and rotate the coordinates of the selected point pairs rather than the image patch itself. Since the coordinates of the point pairs are pre-selected when writing the algorithm, these rotated variants can also be precomputed, so during algorithm execution, we only need to retrieve them from a list corresponding to the orientation of the image feature.

It is worth noting that comparing ORB descriptors is not ideally done using the squared Euclidean distance as in SIFT, as this may not yield good results. Since the ORB descriptor is a binary vector, similarity between two such descriptors is typically measured using the Hamming distance.

## 4.9 Tracking

In the field of computer vision, special attention is given to the fairly common case where the basis of processing is not a still image but a video. The tasks performed on videos are similar to those

done on still images. For example, we can perform segmentation, detection, and classification, but new tasks also arise, such as motion detection and tracking, as well as recognizing various events. The image features discussed earlier are particularly well-suited for tracking tasks.

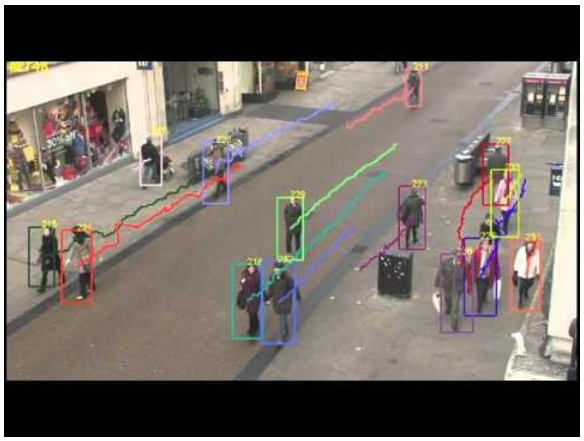

**Figure 4.26:** *Object tracking.*

To process such inputs, we first need to clarify how videos are represented. In most cases, a video is simply interpreted as a sequence of still images, which are processed in the order they are received. It is important to note that our methods are typically designed not to use information from future frames, i.e., frames that follow the current frame being processed. This can be done if real-time processing is not required, but otherwise, it is not feasible. Besides processing as a sequence of still images, a video can also be interpreted as a base image and a series of difference images following it, but this approach is rarely used in computer vision.

Among tracking methods, we can distinguish between pixel-level and object-level tracking. In the latter case, identifying individual objects between frames is crucial. However, this can be challenging since the object itself might change between the two frames, and other complicating factors (occlusions, non-linear motion, etc.) can further complicate identification. In pixel-level tracking, however, the feature tracking methods previously described, such as optical flow, corner detection, or SIFT/SURF/ORB, can be used.

**Figure 4.27:** *Pixel and object tracking.*

## 4.10 Hidden Markov Model

Optical flow can be used to determine the displacement of objects by calculating the relative movement from the previously known position in the previous frame. However, in conventional object tracking based on identification, it can be useful to incorporate the previously known position in some way to provide a better estimate using both pieces of information. Hidden Markov Models (HMMs) are often used for this purpose.

Hidden Markov Models are based on Markov processes. A Markov process is one where the next state depends only on the current state and not on any previous states (i.e., the states of the process fully encode the entire history of the process). Discrete Markov processes with a finite state space can be easily represented using a state graph.

In the case of Hidden Markov Models, however, we cannot directly observe the states of the process, only some consequences of these states. A good example is when we have information about daily activities performed by a person (walking, shopping, cleaning) and want to infer the weather (sunny, rainy) from this. In this case, the hidden states in the Markov model would be the weather states, and the observations would be the activities performed.

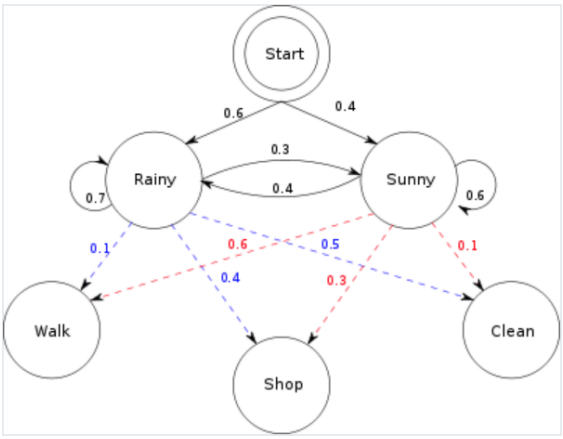


**Figure 4.28:** *Hidden Markov Model: in this example, the hidden states describe the weather, and the observations are the daily activities of a person.*

For Hidden Markov Models, there are two typical problems that can be solved. The first is to determine the probability of a given sequence of hidden states. This task can be solved trivially if the state transition probabilities are known. The second, more interesting problem is to determine the most likely sequence of hidden states given a sequence of observations. This can also be solved, and the most widely used method is the Viterbi algorithm.

In object tracking, the hidden state is the actual position of the object, and the state transition can be addressed in several ways. It is possible to choose a uniform distribution in a certain environment, but it is often preferable to choose a (fairly wide) normal distribution, as larger movements are less likely. The model's observations are the positions estimated using one of the previously described methods. We can assume that the probability between the state and the observation is also given by some Gaussian-like distribution. In this case, it might be advantageous to use a distribution with a heavier tail (e.g., Student's T distribution).

## 4.11 Kalman Filter

It may occur that we have not one but two different estimates for the position of an object. We assume that both estimates are Gaussian distributions with different variances. We might then want to determine a combined estimate with a smaller variance using both. This can be done as follows:

$$\begin{aligned} \mu &= \frac{\sigma_1^2\mu_0 + \sigma_0^2\mu_1}{\sigma_0^2 + \sigma_1^2} = \mu_0 + k(\mu_1 - \mu_0) \\ \sigma^2 &= \frac{\sigma_0^2\sigma_1^2}{\sigma_0^2 + \sigma_1^2} = \sigma_0^2 - k\sigma_0^2 \\ k &= \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \end{aligned} \tag{4.17}$$

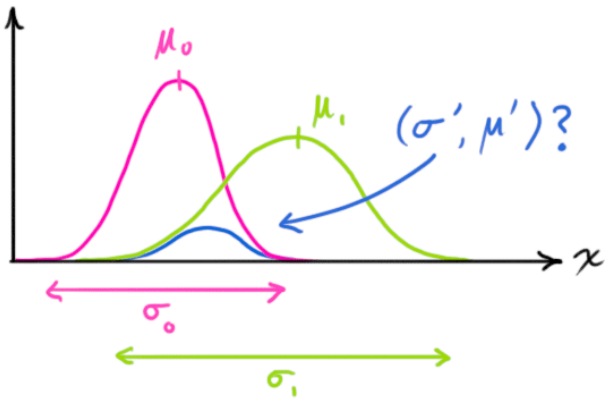


**Figure 4.29:** *Combination of Gaussian distribution estimates.*

The intuition behind the formula is that the new expected value is the weighted average of the expected values of the two estimates, where the weights are proportional to the reliability of each estimate. The reliability of the estimates is inversely proportional to their variances, hence the indices are swapped in the numerator. The reliability of the resulting estimate is the sum of the original reliabilities, meaning the variance results from the replacement operation. The above formula also works for multivariate normal distributions, where instead of variance, we use the covariance matrix.

$$\begin{aligned} \vec{\mu} &= \vec{\mu}_0 + K(\vec{\mu}_1 - \vec{\mu}_0) \\ \Sigma &= \Sigma_0 - K\Sigma_0 \\ K &= \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \end{aligned} \tag{4.18}$$

The quantities $k$ and $K$ in the formulas are known as the Kalman gains.

In the Kalman filter, one of our estimates comes from a measurement, and the other from a prediction based on knowledge of the system dynamics. In the linear case, the system dynamics are given by the following state equations:

$$\begin{aligned} x_k &= F_k x_{k-1} + B_k u_k \\ y_k &= H_k x_k \end{aligned} \tag{4.19}$$

If we know the estimate of the previous state $N(\hat{x}_{k-1}, \Sigma_{k-1})$, the prediction is given by:

$$\begin{aligned} \hat{x}_k &= F_k x_{k-1} + B_k u_k \\ \Sigma_k &= F_k \Sigma_{k-1} F_k^T + Q_k \end{aligned} \tag{4.20}$$

where $Q_k$ represents the uncertainty in the prediction. Since we often cannot measure the system state directly, but only the output, we also need to produce an output estimate from the state estimate as follows:

$$\begin{aligned} \hat{y}_k &= H_k \hat{x}_k \\ \hat{\Sigma}_k &= H_k \Sigma_k H_k^T \end{aligned} \tag{4.21}$$

At this point, we can combine the output measurement $N(z_k, R_k)$ with the estimate using the formula discussed above:

$$\begin{aligned} K_k &= H_k\hat{\Sigma}_k H_k^T(H_k\hat{\Sigma}_k H_k^T + R_k)^{-1} \quad \leftarrow \quad \text{Kalman gain} \\ H_k x_k &= H_k\hat{x}_k + K_k(z_k - H_k\hat{x}_k) \quad \leftarrow \quad \text{estimate of } \mu \\ H_k\Sigma_k H_k^T &= H_k\hat{\Sigma}_k H_k^T - K_k H_k\hat{\Sigma}_k H_k^T \quad \leftarrow \quad \text{estimate of } \Sigma \end{aligned} \tag{4.22}$$

However, at this point, we have estimated the output $y$, not the state $x$. Notice that $H_k$ can be canceled from the left (one $H_k$ can be canceled from the $K$ expression), and $H_k^T$ can be canceled from the right in the covariance matrix equation. Thus, the final estimates are:

$$\begin{aligned} \hat{K}_k &= \hat{\Sigma}_k H_k^T(H_k\hat{\Sigma}_k H_k^T + R_k)^{-1} \\ x_k &= \hat{x}_k + \hat{K}_k(z_k - H_k\hat{x}_k) \\ \Sigma_k &= \hat{\Sigma}_k - \hat{K}_k H_k\hat{\Sigma}_k \end{aligned} \tag{4.23}$$

## 4.12 Multiple Object Tracking

So far, we have discussed tracking a single object. When multiple objects need to be tracked, cases can easily arise (especially when object trajectories cross each other) where it is not clear how to associate the detected objects with those known from previous frames. The problem can be formulated as an optimization problem as follows:

$$\max_x \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_{ij} \quad \text{where} \quad \begin{cases} \sum_j x_{ij} = 1 & \forall i \\ \sum_i x_{ij} = 1 & \forall j \\ x_{ij} \in \{0,1\} \end{cases} \tag{4.24}$$

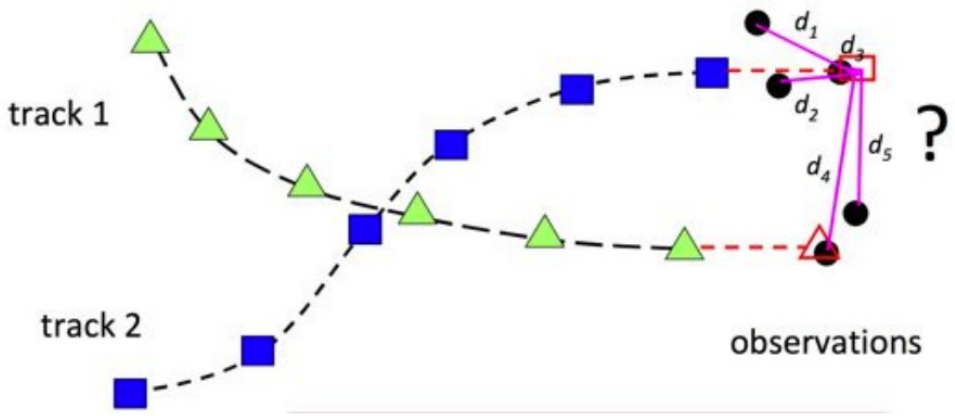


**Figure 4.30:** *Tracking multiple objects.*

Here, $w_{ij}$ is inversely proportional to the error between the previously known and the current detected object properties. Various features can be used for identification. The simplest ones include position, bounding box, or ellipse. However, it is also possible to use local image features introduced in Lecture 3 or different binary descriptor methods discussed in Lecture 5. In every case, we can define affinity, which describes the similarity between the features used to describe two objects:

$$A(x,y) = e^{\frac{-1}{2\sigma^2}||f(x)-f(y)||^2} \tag{4.25}$$

where $f(x)$ is the feature vector describing object $x$. This quantity can be used as weights in the above problem.

This task, also known as the assignment problem, can be solved using the Hungarian algorithm if we are dealing with only one frame. However, with multiple frames, the optimal solution is no longer guaranteed. In this case, the possible trajectories of each object can be described using a weighted, directed graph. Optimal trajectories can be determined using the min-cut max-flow algorithm. The essence of this procedure is to try to partition the graph into $n$ separate parts such that the weights of the cut edges are minimal while ensuring that there is a connection between the initial and final nodes in each of the $n$ subgraphs, and the weights of these edges are maximal.

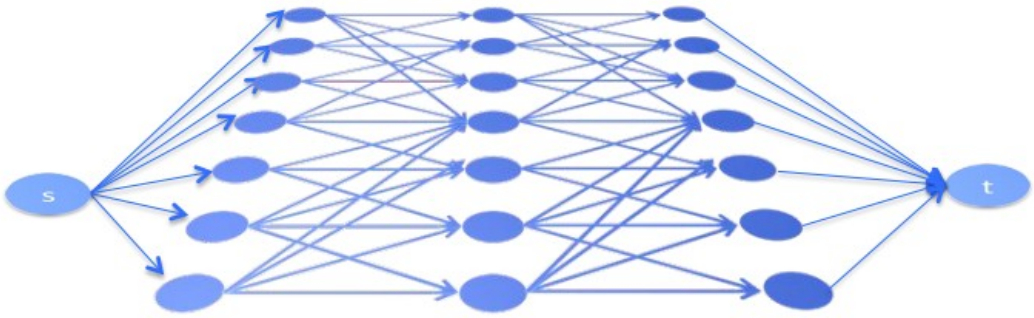


**Figure 4.31:** *The graph for tracking over multiple frames. Each level of the graph corresponds to a frame, and each node at each level represents a detected object.*

# 5 Stereo Vision

## 5.1 Introduction

The fundamental goal of computer vision is to automatically extract information about the real world from a camera image (or images) using a computer algorithm. Most practical imaging systems, however, generate a two-dimensional projection of the real, three-dimensional world, resulting in significant loss or distortion of information. In a typical image, the distance of individual pixels from the camera is not preserved and must be estimated. Additionally, it is evident that due to the projection process, many geometrically important features are distorted. Furthermore, objects of equal size in space will appear to have different sizes in the image if they are at different distances from the camera.

During projection, not only do the sizes change, but the angles also alter, leading to the distortion of geometric shapes and forms. A particularly illustrative example of this distortion is the concept of a vanishing point. As we know, parallel lines intersect at infinity. However, if these parallel lines are projected onto an image using a camera, this point of intersection, which is infinitely far away, will appear on the image. This means that a projection of a point infinitely far away can be finite, and in this case, this projection is called a vanishing point.


**Figure 5.1:** *Distortion of 3D geometric features (left) and the vanishing point (right).*

Thus, it is evident that some significant properties of objects cannot be determined from a single image (although shadows can be used for estimation). If such information is needed, it is possible to use imaging devices that can compute depth information for each pixel in the image (RGB-D sensors). However, these devices are generally much more expensive than standard cameras (about five times the price for the same quality), making them not always a feasible option.

Fortunately, there is another solution: much like human vision, we can restore a portion of the original three-dimensional space using two or more cameras/images. The field of three-dimensional computer vision is focused on solving this task as accurately and efficiently as possible. In this chapter, I will discuss this area.

It is important to note that at least two images are necessary because, by projecting the same scene from different positions, we obtain new information about the original scene's geometry. By matching this information, we can recover the data lost and distorted during projection. It is worth noting that two cameras are not necessarily required—two images taken from different positions

with a single camera can suffice. However, in this case, it is crucial that the scene does not change between the two images, as this would corrupt the reconstruction results. If the motion/change of objects cannot be restricted, then it is advisable to use two synchronized cameras.

## 5.2 Pinhole Camera Model

In three-dimensional vision, the task is to reconstruct the information lost and distorted during the camera's projection. To perform this task, we must first understand the projection process itself and establish a mathematical model. Afterward, in the case of a specific camera system, we need to perform measurements to determine the unknown parameters of the previously established model. This step is called camera calibration. In the following subsections, we will examine two important cases of calibration: in the first, we determine the parameters of a single camera's projection, while in the second, we aim to determine the relative positions of cameras within a camera system.

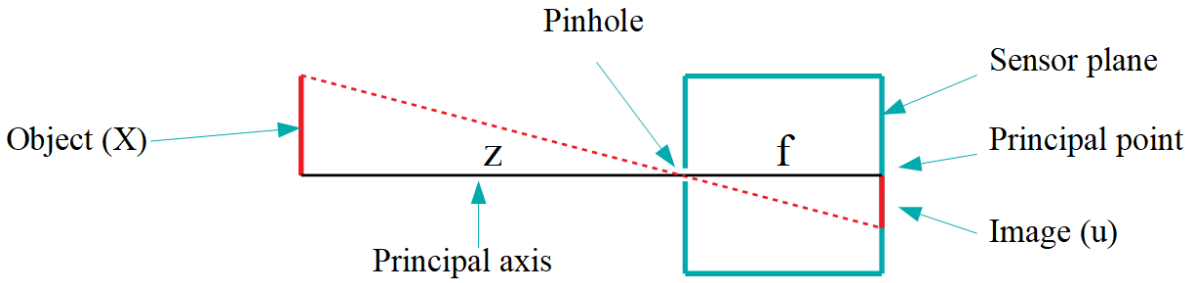


**Figure 5.2:** *The physical model of the pinhole camera.*

The pinhole camera model is widely used in computer vision. A pinhole camera can be simply imagined as a box with a small hole on one side through which light can enter. The incoming light through the hole creates an inverted image on the opposite side of the box. In real cameras, this is where the light sensor is located. Another significant difference in real cameras is that instead of a small hole, a lens is used, which focuses parallel rays of light to a point, thus replacing the pinhole. The advantage of using a lens is that it allows significantly more light to enter than a pinhole, but it introduces geometric distortion in the image. The pinhole camera model can be described by the following equations:

$$u = f_x \frac{x}{z} + p_x; \quad v = f_y \frac{y}{z} + p_y; \tag{5.1}$$

Where $u$ and $v$ are the pixel coordinates, $x$, $y$, and $z$ are the spatial coordinates of the object, $f$ is the focal length of the camera, and $p$ is the principal point. However, before we further discuss the camera model, we need to make a brief detour. During camera calibration, we will essentially perform a numerical estimation of certain elements of the pinhole camera model (focal length, principal point, etc.). When performing numerical estimations, it is crucial to formulate the problem in such a way that makes the estimation as easy as possible. For calibration—and in general for geometric problems—it is customary to use what are known as homogeneous coordinates.

### 5.2.1 Homogeneous Coordinates

The use of homogeneous coordinates is widespread in projective geometry. Projective geometry is an extension of Euclidean geometry that allows significantly more transformations. While Euclidean geometry only permits rigid transformations (translation, rotation), projective geometry allows for directional scaling, shearing, and the operation of projection as well. In projective geometry, the Euclidean plane/space is extended by one additional dimension, so the projective plane can be described with 3 dimensions, and space with 4 dimensions. The transition between the two geometries can be described by the following equations:

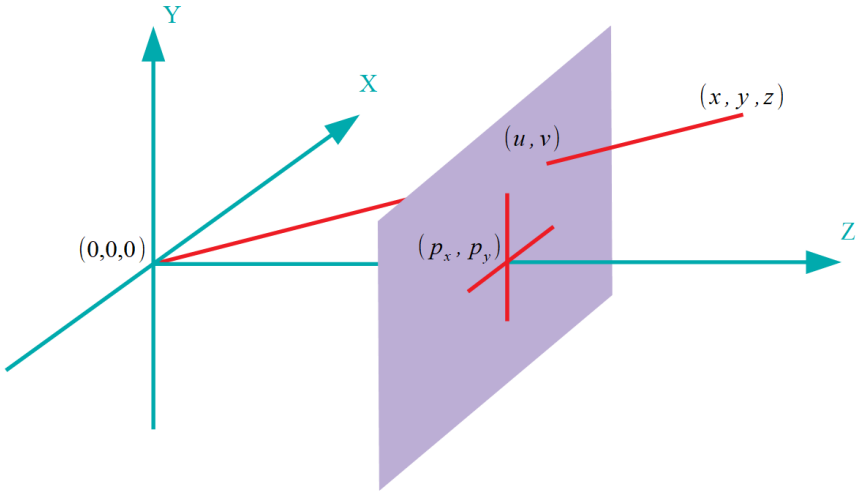


**Figure 5.3:** *The geometric model of the pinhole camera. Note that in this model, the image plane is in front of the pinhole, resulting in an upright image. This is, of course, just a mathematical simplification.*

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \rightarrow \begin{pmatrix} \frac{X}{W} \\ \frac{Y}{W} \end{pmatrix} \qquad \begin{pmatrix} aX \\ aY \\ aW \end{pmatrix} = \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \tag{5.2}$$

The first two equations describe the conversion from Euclidean to projective geometry and back. A curious consequence of the conversion rules is that if a point described in homogeneous coordinates is multiplied by any non-zero scalar, after the conversion back, it will correspond to the same Euclidean point. This scale invariance property is expressed by the third equation. As a result, the homogeneous coordinates corresponding to a Euclidean point lie along a line that intersects the $W = 1$ plane exactly at the point's Euclidean coordinates.

There are two important advantages to using homogeneous coordinates. The most significant is that the relationships of the pinhole camera model become linear in this coordinate system. The other important property is scale invariance, which will greatly simplify numerical estimations. An interesting feature of homogeneous coordinates is that there exists the point $(x, y, 0)$, which is at infinity on the Euclidean plane but still described with entirely finite coordinates in the projective plane. This "directed infinity" point is called an ideal point and plays an important role in self-calibration procedures.

### Transformations

As we mentioned, homogeneous coordinates are used to describe various transformations. However, it is worth discussing exactly what types of geometric transformations exist. The most fundamental of these is the so-called Euclidean transformation, also known as a rigid transformation, as it only allows the operations of rotation and translation. The consequence of this is that during a Euclidean transformation, the sizes and angles of objects remain unchanged. It is easy to see that this is not the case during image formation, so we must resort to more permissive geometric transformations. The next type of transformation is the similarity transformation, which, in addition to the previously mentioned two operations, also allows uniform scaling (equal scaling in all directions). This transformation does not preserve sizes but still preserves angles. These two types of transformations can be expressed using homogeneous coordinates as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} ar_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & ar_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & ar_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5.3}$$

Where $r_{ij}$ is an element of the rotation matrix, $t_i$ is the translation vector, and $a$ is the uniform scaling parameter. Note that the columns of the rotation matrix are unit length and mutually orthogonal, meaning the rotational submatrix is an orthogonal matrix.

The next type of transformation is the affine transformation, which allows two additional operations—directional scaling and shearing. The affine transformation no longer preserves angles but still maintains parallelism, so it is still less permissive than the process of projection, where even parallelism is not preserved. The final type is the projective transformation, which includes the operation of perspective projection, thus only preserving intersections, not parallelism. The equations for these two transformations are as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} wx' \\ wy' \\ wz' \\ w \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5.4}$$

Unlike earlier cases, these matrices are composed of arbitrary elements denoted by $a_{ij}$.

### 5.2.2 Projection Matrix

After introducing homogeneous coordinates, we can describe the pinhole camera model's projection with a single linear matrix multiplication:

$$\begin{pmatrix} wu \\ wv \\ w \end{pmatrix} = \begin{pmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = Ax \tag{5.5}$$

Where $f$ is the focal length, $p$ is the principal point, $u$ and $v$ are the pixel coordinates, and $A$ is the so-called camera matrix. Note that division by the $z$ coordinate will only occur when returning to the Euclidean plane. It's important to note that this equation holds true if the point's spatial coordinates are given in the camera's coordinate system. The origin of the camera's coordinate system is typically the pinhole, its $x$ and $y$ axes coincide with the image plane, and the $z$ axis points in the direction of the principal axis.

However, in three-dimensional space, there exists another coordinate system called the world coordinate system, where the coordinates of 3D points are usually given. The world coordinate system generally depends on the specific application and situation. It may sometimes be fixed to a physical object, but it can often be freely chosen. In such cases, it is often convenient to choose it to coincide with the camera's coordinate system, though there are many exceptions to this rule.

$$\begin{pmatrix} u \\ v \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} wu \\ wv \\ w \end{pmatrix} \quad \leftarrow \quad A \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} R & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \tag{5.6}$$

In practice, the world coordinate system does not necessarily coincide with the camera's coordinate system, so the transformation between the two must be determined during calibration. Fortunately, the transformation between the two coordinate systems is a simple Euclidean transformation, consisting of just a rotation $R$ and a translation $t$. It is worth noting that the unknown parameters obtained in this way are divided into two separate groups. The elements of the camera matrix $A$ form one group: these parameters depend exclusively on the internal construction of the camera and are therefore called intrinsic parameters.

The second group includes the elements of $R$ and $t$, which do not depend on the internal properties of the camera but are instead determined solely by the specific arrangement. Therefore, these are called extrinsic parameters. Each of these parameter groups defines a geometric transformation, and their composition gives the complete projection matrix ($P$):

$$P = A[R \;\; t] \tag{5.7}$$

### 5.2.3 Real Optics

In reality, cameras use a slightly modified setup, as the pinhole is too small to allow enough light for high-quality image formation. If we were to increase the size of the pinhole, the larger angle of incidence of light rays would result in a blurry image. Therefore, a lens is placed at the camera's entrance, focusing parallel incoming light rays into a single point, replacing the pinhole, while its size is large enough to allow sufficient light to pass through.

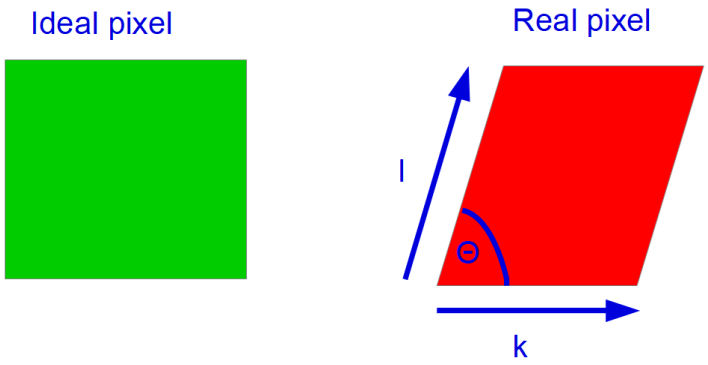


**Figure 5.4:** *In some cameras, real pixels are not necessarily square-shaped. There may be differences in pixel aspect ratios (especially in older video cameras), and the angle between the sides may deviate from 90 degrees.*

However, adding the lens introduces a complication: light rays arriving from the edges of the camera's field of view no longer enter the lens parallel, causing them to refract differently than the parallel incoming rays. As a result, these light rays strike the photosensor array at different positions, causing the image to shift. This phenomenon is called radial distortion and is common, particularly in cameras with wide-angle lenses.

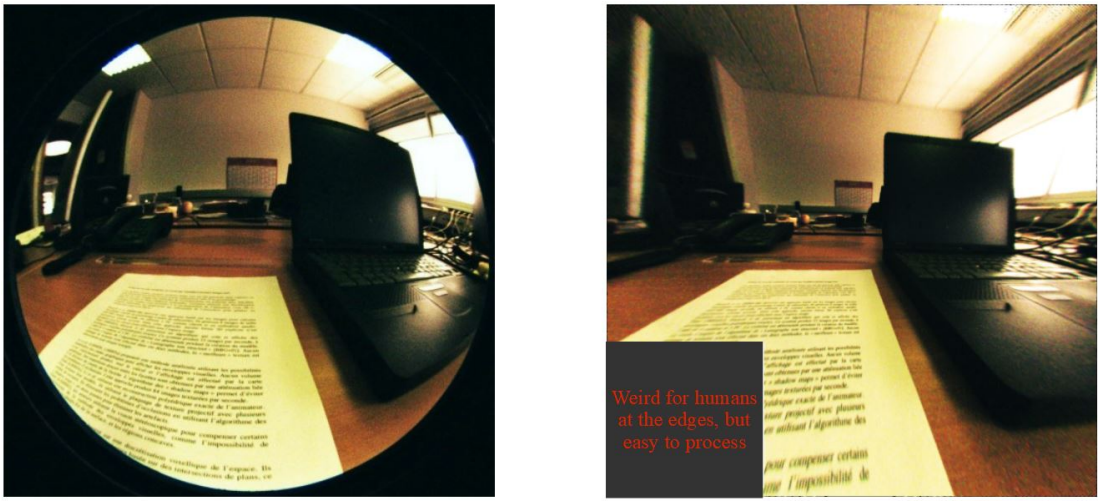


**Figure 5.5:** *Radial distortion (left) and the distortion-corrected image (right).*

## 5.3 Calibration

To perform 3D reconstruction, the first step is to determine the camera's projection parameters. As an introduction, we first needed to understand the mathematics of projection in detail, during which we realized that the projection can essentially be divided into two parts. One part depends on the camera's position in the world, and the other part depends on the camera's internal properties. The methods presented in this subsection are primarily (but not exclusively) used to determine the internal parameters.

However, the question arises: how can these parameters be determined? The answer is quite simple: since we know the mathematical relationship between the 3D points and their projections on the image, we only need to take numerous measurements and numerically estimate the unknown parameters of the projection. In practice, for images, this means creating a calibration object with easily recognizable markers placed in known positions. Then, we take images of the calibration object, identify the positions of the markers in the image, and estimate the parameters based on these point pairs.

Depending on the calibration objects used, there are different forms of calibration. Mathematically, the simplest case is when the markers on the calibration object are arranged in a completely general way, meaning they do not fall on a single plane or line. In this case, we refer to a three-dimensional calibration object. However, there are also two-dimensional and one-dimensional calibration objects. As the arrangement of markers becomes more specific, the underlying mathematics of the calibration becomes more complex.

A special case of calibration is self-calibration, where no calibration object is used, and the calibration is done by matching features between images. In this case, easily recognizable and identifiable natural markers in the images are used. One important limitation of self-calibration is that it can only determine the internal parameters. Another disadvantage is that at least three different images are needed, whereas methods using calibration objects theoretically require only one image.

## 5.4 Stereo Calibration

As established in previous discussions, 3D reconstruction requires multiple cameras or at least multiple images taken from different positions. Additionally, it is necessary to know the geometric relationship between the different cameras or images. In some cases, this relationship can be obtained from another source, such as an external sensor (this is especially common in mobile robotics). However, in many cases, we must rely solely on the images taken by the cameras. In this subsection, we will explore how calibration methods can be used to determine the relative positions of cameras or images.

To explore the possible methods, it is useful to recognize that the nature of the task and, consequently, its difficulty depend greatly on the setup. As discussed in previous chapters, there is a distinction between two scenarios: when two cameras take images from different positions, and when a single camera moves between two images. In the former scenario, we are dealing with a fixed camera system where the cameras do not move relative to each other during image capture, so calibration only needs to be performed once, at the beginning. In the latter scenario, the camera is continuously moving, and the motion between each pair of images must be estimated.

In the case of fixed camera systems, we are in a very fortunate situation, as the internal calibration of the cameras (which must be done anyway) can be performed for all cameras simultaneously. The only requirement is that the calibration object be visible in all cameras. Recall that in all methods using calibration objects, not only the internal parameters of the cameras but also the external parameters (i.e., the transformation between the calibration object and the camera) are computed for each calibration image.

If the transformation between each camera and a designated point is known in a given image, the transformation between the cameras can be easily calculated. Moreover, since the relative positions of the cameras do not change between images, all images can be used to estimate this transformation. This implies that in the case of fixed camera systems, there is no need for separate calibration of the camera system, as the relative positions can be easily determined during the internal calibration, which is already necessary.

### 5.4.1 Epipolar Geometry

The situation is entirely different in the case of moving cameras. Here, the camera calibration is performed once at the beginning of the usage, and then, by moving the camera in the space of

interest, we aim to generate a reconstruction. At this point, there is no calibration object in the space that we can use for simple calibration. This means that the estimation of the movement must be carried out using the natural markers that appear in the image. This is a challenging task because we do not know the spatial coordinates of the natural markers (if we did, the reconstruction would already be complete), so we can only rely on the fact that if we find the same marker in both images, we can set up an equation for these point pairs.

However, before we do this, we need to examine the geometric arrangement itself. For simplicity, let us limit ourselves to the so-called stereo setup, where the moving camera is represented in two positions: these are referred to as the right and left cameras. The center of the left camera is $O_L$, and a spatial point $X$ intersects its image plane at the point $X_L$. The center of the right camera is $O_R$, and the point $X$ intersects its image plane at the point $X_R$. The translation between the two camera centers is described by the vector $t$, and the rotation between the image planes is described by the matrix $R$.

It is worth noting that if we know the points $O_L$ and $X_L$, but not $X$ (as is the case in reality), we can determine the direction in which $X$ is located from the left camera, but not its distance. This means that along a spatial line, there are infinitely many candidates for the point $X$. However, projection maps a spatial line to another line in the image plane, so all possible matches for $X_L$ in the right camera's image also lie along a line. These lines are called epipolar lines. It is important to know that all epipolar lines intersect at a point called the epipole ($e_L$ and $e_R$). Moreover, the epipole is the point where the line connecting the two camera centers intersects the respective image planes.

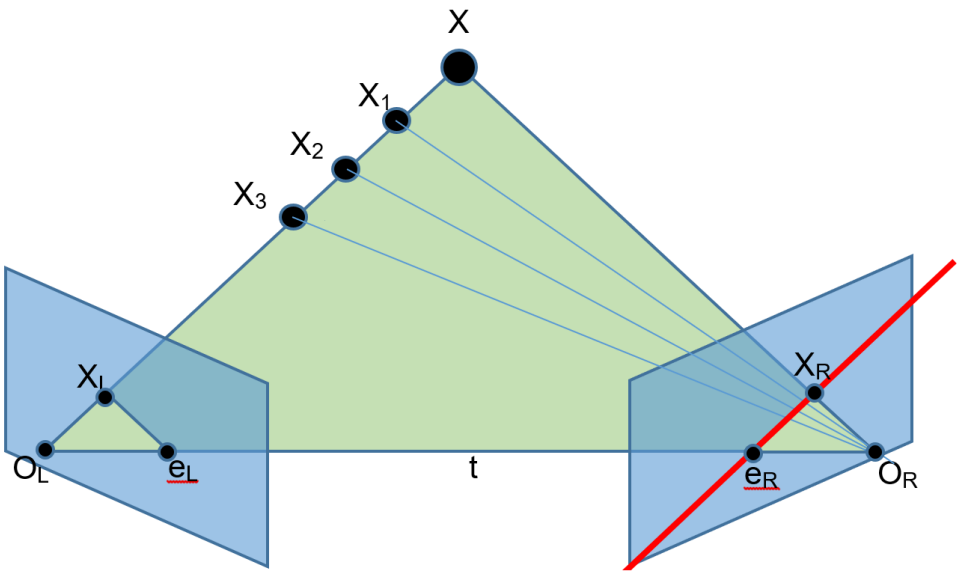


**Figure 5.6:** *The epipolar configuration.*

It is essential to note the following property: the vector connecting the two cameras ($t$), the vector pointing from the left camera center to the point $X_L$, and the vector pointing from the right camera center to the point $X_R$ all lie in the same plane. This is only true, of course, if $X_L$ and $X_R$ are images of the same point $X$. If three vectors lie in the same plane, it means that if we choose any two and compute their cross product, the result will be perpendicular to the third vector. This observation allows us to formulate a system of equations that can be solved numerically, thereby determining the parameters of the cameras.

## 5.5 3D Reconstruction

In the previous lecture, we learned about the fundamental relationships in image formation geometry, as well as the key details of calibration procedures. These are essential components of

3D reconstruction and the foundational steps of building any reconstruction system. However, the current lecture focuses on the actual implementation of the reconstruction and the computer vision algorithms required for this task.

The process of 3D reconstruction can be broken down into four basic steps, one of which we have already discussed in the previous lecture. This first step is the calibration of individual cameras and the camera system. After this, using the rectification transformation obtained during the camera system calibration, we transform the images to be used for the reconstruction. The next step is to search for correspondences between the individual images, and based on the calibration results and the obtained point pairs, we compute the spatial points. Naturally, the reconstruction of the three-dimensional space can be followed by several further processing steps, as the goal of the reconstruction is spatial processing. The algorithms used for this purpose will be the topic of the next subsection.

## 5.6 Rectification

Once the transformation between the cameras is determined, it opens up a new opportunity for us. Recall the epipolar lines introduced during the stereo setup. These lines define the direction along which a point's pair can be found in the other camera's image. Naturally, these lines differ for each point. However, there exists a particular camera arrangement where all epipolar lines are horizontal. This arrangement is called the standard stereo setup, and it occurs when the translation between the cameras is purely horizontal, and there is no rotation between their image planes.

The great advantage of this arrangement is that we only need to search for the pair of each pixel along a single row of the other image, rather than across the entire image. This typically results in a 2-3 order of magnitude speedup. In practice, such an arrangement can only be achieved with industrial tools, but calibration provides the possibility of artificially creating the standard setup, known as rectification. During rectification, the two images are distorted using a linear transformation in such a way that the epipolar lines become horizontal. It is important to note that in the presence of lens distortion, the epipolar lines themselves also curve, so this must be corrected during rectification as well. Consequently, the two operations are often combined, and many sources refer to this combined operation as rectification.

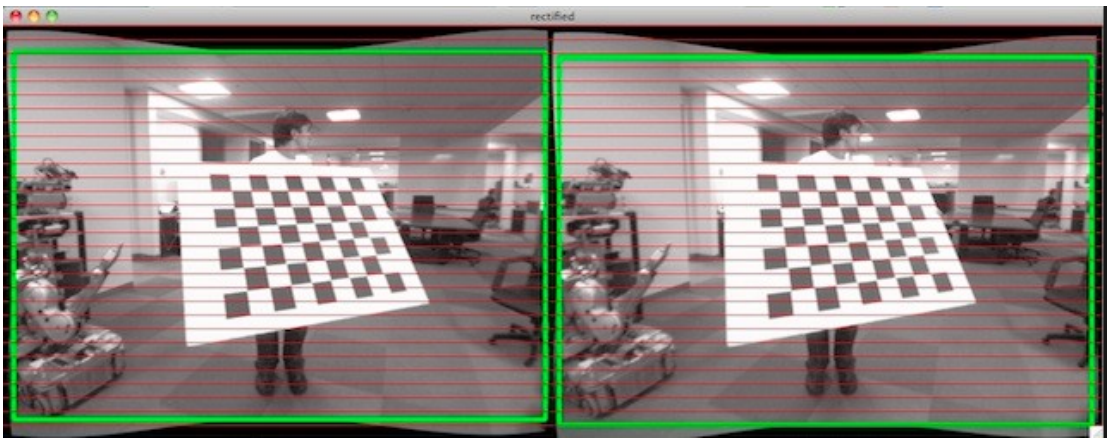


**Figure 5.7:** *The rectification operation.*

## 5.7 Disparity

The search for correspondences between the captured images is the most fundamental operation in reconstruction. The information that the image of the same spatial point is found at different locations in images taken from different positions is precisely the new information we need to restore the information lost during projection. Therefore, the accuracy of matching between images

fundamentally determines the quality of the final reconstruction. We have two requirements for matching: first, the pairs must be found as accurately as possible, and second, the matching should be as complete as possible, meaning we should find pairs for as many pixels as possible. The latter, of course, is not fully achievable due to the finite extent of the image and occlusions.

As discussed in the previous section, performing rectification greatly simplifies the search for correspondences. In the case of a rectified image pair, we can guarantee that a given pixel's pair will be located on the same row in the other image. Consequently, for each pixel in one image, we can assign a so-called disparity value, which expresses how many pixel positions its pair is displaced relative to it. In other words, disparity is the horizontal distance between a pixel and its pair. It is worth noting that there are less common standard arrangements where the cameras are positioned above/below each other, in which case the disparity is vertical.

If we determine the disparity value for each pixel in the image, we obtain a grayscale image called a disparity map. This image is also easily interpretable by the human eye because the disparity value essentially depends on the distance from the camera. Most people have likely experimented with this at least once in their lives, where they close one eye and observe their outstretched hand, then switch to the other eye and notice that their hand appears in a slightly different location. By continuing this experiment, we can observe that the closer the hand is, the greater the difference between the positions perceived by the two eyes. From this, it is easy to understand that the larger the disparity value, the closer the object corresponding to that pixel is to the camera.

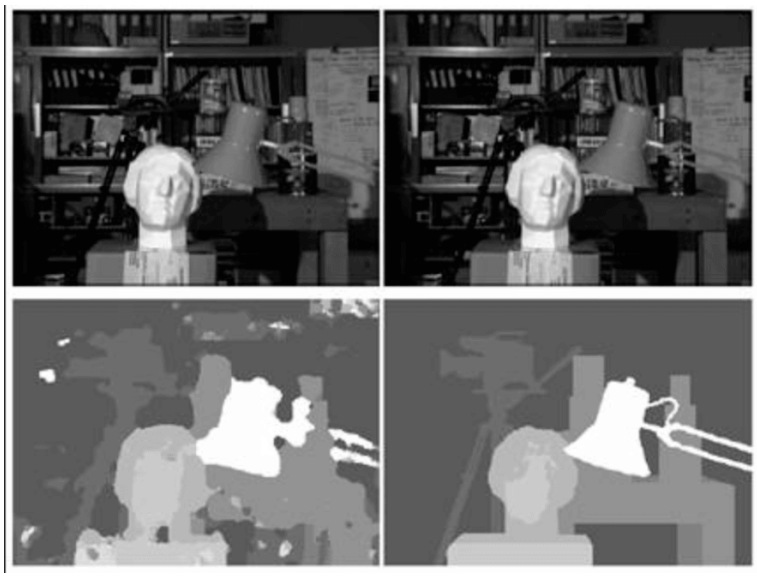

**Figure 5.8:** *A stereo image pair (top) and the corresponding disparity maps (bottom). The left image is the result of an algorithm, while the right image is the true disparity (so-called ground truth).*

There are many methods for determining disparity, which can be categorized into two main groups: dense and sparse disparity methods. Dense methods determine the disparity value for nearly every pixel in the image, while sparse methods only compute it at a few selected positions. Although our goal is to determine disparity for as many points as possible, sparse disparity methods often provide significantly more accurate results. The disparity calculated by sparse methods can be densified using various interpolation techniques.

### 5.7.1 Block Matching

One of the most important families of dense disparity methods is based on the block matching (BM) algorithm. Block matching is a fundamental algorithm in computer vision, with many applications, often mentioned under slightly different names. The essence of the algorithm is that it takes a neighborhood around the pixel under consideration and moves it across all possible positions along

the corresponding row in the other image. At each step, it compares the neighborhood of the original pixel with the neighborhood found at the current position. Various similarity metrics can be chosen, but the sum of squared differences of the individual pixels is most commonly used. The block matching algorithm seeks the position where this sum of squared differences is minimized, and this position will be the match for the original pixel.

It is worth noting that the block matching algorithm determines the disparity of each pixel independently of others. This is important because the disparity map is generally quite "smooth," meaning the disparity values of neighboring pixels are close to each other, with only rare large jumps. This is naturally a result of real three-dimensional spaces where objects are generally spatially connected, and only their boundaries exhibit larger spatial jumps. However, the block matching algorithm does not take this into account, resulting in a disparity map that is often quite noisy.

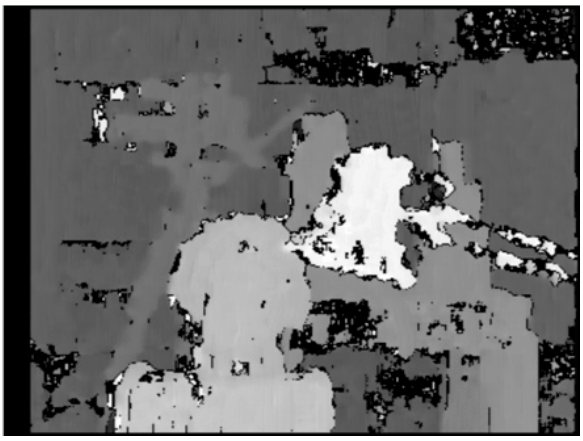

**Figure 5.9:** *The result of the BM algorithm.*

### 5.7.2 SGBM

Taking into account the properties of real spaces, it is possible to significantly improve the quality of the disparity map. One frequently used method is to add a penalty term to the block matching error criterion that enforces smoothness in the disparity map. This solution is applied in the semi-global block matching (SGBM) method. The SGBM method extends the standard block matching error function with two additional terms. In the first term, we examine a neighborhood around the pixel under consideration, and if the disparity values between pixels differ by exactly 1, we increase the error function by a constant $P1$. The second term ensures that if the difference is greater than 1, the error function is penalized by a constant $P2$. The values of the constants are chosen based on the results, although a commonly used rule of thumb is that $P2$ is four times the value of $P1$.

$$E(D) = \sum_{p} E(p, D_p) + \sum_{q \in N(p)} P1T(|D_q - D_p| == 1) + \sum_{q \in N(p)} P2T(|D_q - D_p| > 1) \tag{5.8}$$

## 5.8 Reconstruction

The final step of three-dimensional reconstruction is reprojection, during which the image points are projected back into three-dimensional space. This is done using the principle of triangulation.

### 5.8.1 Triangulation

During calibration, we determine the parameters of each camera, allowing us to generate the ray along which the light that forms the pixel arrived at the camera for each pixel. This can naturally

be done for all cameras where the matching point of the given pixel has been found. Furthermore, since we know the transformations between the cameras, we can transform all cameras and rays into a common coordinate system. In this common reference frame, the projection rays corresponding to the same spatial point will intersect at the original point's position.

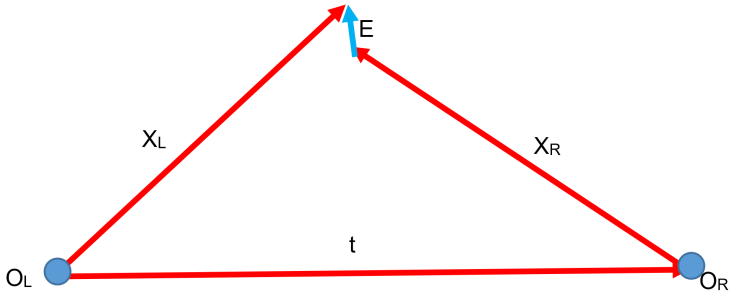


**Figure 5.10:** *The principle of triangulation.*

The method just described is the principle of triangulation, which works particularly well in two-dimensional problems. However, in three-dimensional space, we encounter the problem that even very small inaccuracies can cause the rays from the two cameras to miss each other, resulting in no intersection point. Therefore, it is more practical to determine the original coordinates of the three-dimensional point using the least squares method.

### 5.8.2 Metric Reconstruction

At this point, it's worth recalling the original motivation behind the field of three-dimensional computer vision. Our goal was to reconstruct the geometry of the original three-dimensional space from images distorted and stripped of information by the camera's projection. However, we must be aware of the limitations of the methods described here and the conditions under which they function. One trivial but important limitation is that simple two-camera (stereo) reconstruction will never yield a complete 3D space. The reason for this is that in a stereo arrangement, the cameras are typically positioned close to each other, so they only see the front side of the objects in the scene. The resulting reconstruction will be similar, with the back of all spatial objects missing.

Another important aspect is the question of the metric in the reconstruction. The primary goal is for the dimensions of the resulting reconstruction to be presented in some known (preferably SI) unit of measurement, i.e., for the reconstruction to be metric. It is, therefore, worth briefly discussing the conditions for metric reconstruction and the consequences of incomplete fulfillment of these conditions. Simply put, the condition for metric reconstruction is that the internal parameters of the cameras used, as well as the external parameters of the arrangement, are known completely.

There may be situations (typically with moving cameras) where the external parameters of the arrangement are only partially known (for example, the magnitude of the displacement between the two images cannot be determined). In such cases, the reconstruction will be shape-preserving (free of geometric distortions), but the actual size of the objects will remain unknown. In these cases, the units of the resulting coordinates are typically based on the displacement between the two cameras. It is worth noting that in such situations, if the displacement magnitude can be estimated in any other way (e.g., using a speedometer or an RGB-D sensor) or if the size of a known object in the space can be estimated, the reconstruction can be made metric. Naturally, even if this is not possible, the reconstruction may still be useful—for example, in the case of vehicles, knowing distances in relation to the vehicle's speed may be sufficient to avoid collisions.

However, there may be cases where neither the internal nor external parameters are known. In this case, while the reconstruction process can still be carried out, the resulting three-dimensional scene will be invariant only up to an unknown projective transformation. As mentioned in the introductory section of this chapter, a projective transformation can alter the parallelism of lines or even bring an infinitely distant point into finite proximity, so using such reconstructions can be dangerous.

## 5.9 Single and Multi-Camera Methods

In the previous subsection, we explored the principle of stereo reconstruction and its shortcomings. Since reconstruction with two cameras is incomplete, it is often referred to as a ”2.5D” solution. Moreover, it is worth noting that in stereo reconstruction, the positions of spatial points were determined using the minimum number of measurements (two) required. This implies that two-camera reconstruction is the noisiest type, as the noise in the estimation can be reduced by increasing the number of measurements. Therefore, it is worth briefly discussing the types of multi-camera reconstruction.

The main reason for this brief discussion is that in the previous subsection, we already covered almost everything needed for these solutions. If more fixed cameras are used, the entire space can be reconstructed, provided that the cameras surround the area of interest from all directions. In this case, the reprojection can be computed with great accuracy using the least squares method described in the previous section. These solutions are most commonly used in animation and filmmaking, as well as in medical imaging, for implementing so-called motion capture technologies. Typically, the movement of a person or animal is recorded in three dimensions with high accuracy, and this data is then used for various purposes (creating animated characters, diagnostics).

### 5.9.1 SfM and SLAM

The case of reconstruction using a moving camera is somewhat more interesting. While it is difficult to speak of multi-camera reconstruction in this case, reconstruction with a moving camera is typically performed from multiple frames, so it still falls into this category of methods. Moving-camera reconstruction methods are commonly referred to by two names: SfM (Structure from Motion) and SLAM (Simultaneous Localization and Mapping). The distinction between these two methods is not sharp, although in most literature, SfM methods typically perform the reconstruction offline after the images have been captured, whereas SLAM does so in real-time, online.

In offline solutions, correspondences between existing (not necessarily sequential) images are sought, and reconstruction is typically initiated from the points that appear most frequently or from the image pair with the most correspondences. These correspondences are usually found using robust image feature detection methods (such as SIFT, SURF, etc.).

**Figure 5.11:** *Reconstruction of the Colosseum from tourist photos.*

In online solutions, the images are processed in the order they are captured, and each new image is added to the existing reconstruction while simultaneously estimating the camera’s new position. In this case, the estimation is typically made relative to the previous image and state. Due to the real-time requirement, the matching is usually done using optical flow or similarly fast methods. A typical SLAM algorithm involves the following steps:

1. Feature detection
2. Matching with the features of the previous image

3. Camera pose estimation
4. 3D point coordinate estimation
5. Bundle adjustment

A special emphasis should be placed on the bundle adjustment step, which occurs in both offline and online reconstruction methods. This step is quite similar to the final refinement step in internal camera calibration procedures, as here too, an iterative numerical optimization process is used to minimize geometric errors. However, instead of refining the camera's internal parameters, we optimize the coordinates of the three-dimensional points and the elements of the transformation matrix that describes the camera's position.

One of the fundamental problems of SLAM-type algorithms is the phenomenon of drift. This occurs because the new camera position is always estimated relative to the previous position, so the camera's absolute position is derived from a series of relative displacements. Since perfect estimation is impossible, the errors in successive estimates gradually accumulate, causing the initially small error to grow larger, meaning that the estimated position slowly "drifts" away from the true one.

Several methods exist to address this issue. The first approach is to estimate the relative displacement not only based on the previous image but also in relation to earlier images, thus reducing the amount of drift. Of course, this cannot be done indefinitely, as the moving camera will eventually see new areas, and there will be no overlap with frames that are too old, making it impossible to find common features. Another solution is loop closure detection, where we detect when the camera returns to a previously visited location, and in such cases, the position is estimated relative to the frame captured at that earlier location. Additionally, it is generally possible to estimate localization relative to the already completed portion of the map, in addition to the previous frames.

# 6 Segmentation

## 6.1 Introduction and Methods

In computer vision, we often need to assign each pixel of an image to distinct objects, that is, to segment the image. The output of segmentation is most often a binary or multi-state image that can be used as a mask to highlight or cut out the object found in the original image. The segmentation process can also be used to create object proposal methods mentioned at the end of the previous subsection.

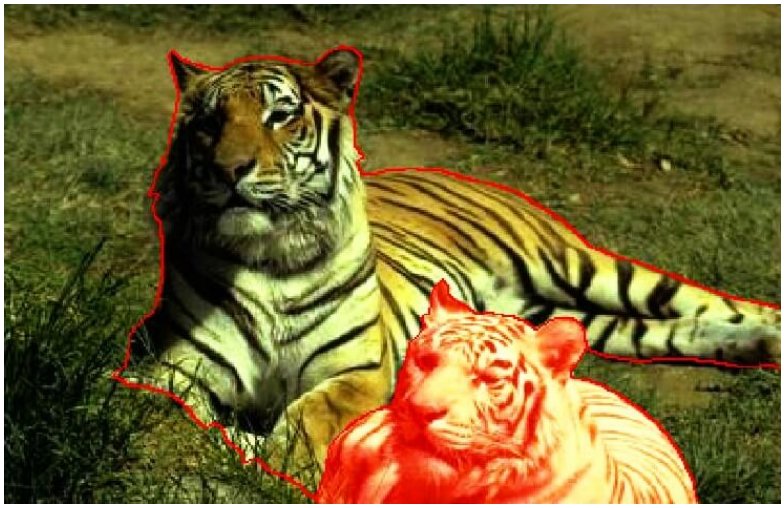

**Figure 6.1:** *The problem of segmentation.*

Based on the information used and the specific output of the process, different types of segmentation can be distinguished. Foreground segmentation refers to when we want to find pixels belonging to an object with known properties. In traditional image segmentation, however, there is no specific foreground object; instead, we want to divide the image according to separate objects. Sometimes the image is segmented not by individual objects but by object categories or classes, in which case we speak of semantic segmentation. This method, however, merges two neighboring objects of the same class into a single segment. If we want to segment not only by classes but also by individual instances within them, we speak of instance segmentation.

There are many methods for segmentation, the simplest of which are color or intensity-based methods such as thresholding and clustering. There are also methods that aim to find contiguous regions based on different coherence criteria; these are called region-based methods. In addition, edge-based, motion-based, and graph-cut-based methods also exist.

## 6.2 Thresholding

Foreground segmentation can be most easily performed based on color or intensity. In simple, controlled environments, it is often guaranteed that the object relevant to our application is the only object with a specific color in the image, so segmentation can be easily performed using thresholding on the color channels. Of course, in reality, the binary image obtained this way still needs to be filtered using the methods described in subsection 4.4. It is important to note

that whether we are dealing with intensity or color-based segmentation, thresholding is almost exclusively performed in one of the HSV or YCbCr color spaces (or their variations).


**Figure 6.2:** *The result of thresholding (left) and multi-level thresholding (right).*

In thresholding-based segmentation, the threshold value is often determined by the designers, which can often be suboptimal. To avoid this, we frequently apply the method of histogram backprojection. At the beginning of the process, we create a reference histogram of the object to be segmented, which is then compared with the image during operation. For each pixel in the image, we determine the likelihood that it comes from the reference histogram during the backprojection. By thresholding the resulting probability image, we can obtain a mask that contains the pixels belonging to the object.

The histogram of the image to be segmented can also be used for traditional segmentation without a reference object. In this case, the algorithm looks for peaks separated by valleys in the image's histogram, thus dividing the pixels of the image into multiple parts. Naturally, this segmentation can also be performed in multiple dimensions simultaneously, allowing both color and brightness information to be utilized.

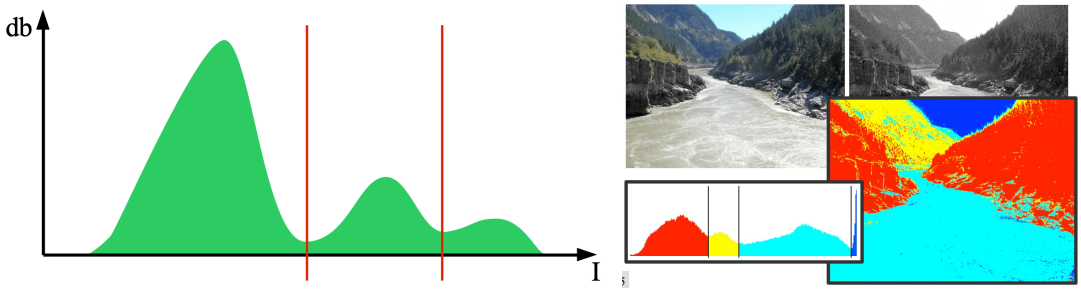


**Figure 6.3:** *The principle of histogram-based thresholding (left) and its result (right).*

**Application:** Many virtual and augmented reality systems use hand gestures to implement one direction of human–machine communication. Although there are solutions where these gestures are detected using gloves equipped with various sensors, there are at least as many camera-based gesture recognition systems. For these to work, hand segmentation is required, which can easily be done, for example, using the histogram backprojection method, based on the color of the skin. It is important to note that this solution is only advisable if the camera does not see other skin surfaces besides the hand, as in that case, those could also be mistaken for a hand. This is usually guaranteed in head-mounted vision systems (such as display glasses with cameras).

One major issue with color-based segmentation is that the interpretation of colors can be subjective in certain cases. This is well illustrated by the image of "The Dress," which took the internet by

storm in 2015. This image shows a dress made up of two colors, which some people see as white and gold, while others see it as blue and black. The exact explanation for this phenomenon is still unknown, though the most popular theory attributes it to individually varying compensation for the color-distorting effects of natural light.

**Figure 6.4:** *The Dress.*

## 6.3 Clustering

A solution very similar to histogram-based segmentation is clustering-based segmentation. We previously mentioned clustering in the context of visual word-based classification, where the essence is to partition a set of points defined in any arbitrary space into several subsets or clusters in such a way that the resulting clusters satisfy some compactness criterion as much as possible. Although many algorithms have been proposed for clustering, one of the most widely used solutions is the k-means algorithm.

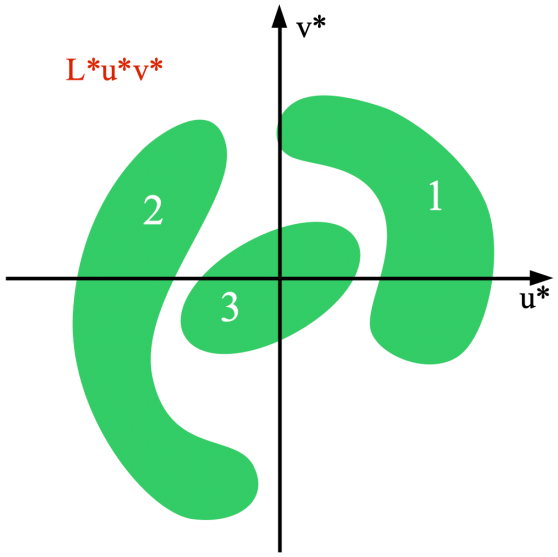


**Figure 6.5:** *The principle of clustering.*

### 6.3.1 k-Means

The k-means method aims to partition the set of points into $k$ clusters in such a way that the sum of the squared distances of the elements of each cluster from the cluster center is minimized. It uses an iterative algorithm, starting with randomly placing $k$ centroids in the space defined by the data points. The algorithm then iterates through the following two steps until convergence. In the first step, each point is assigned to the nearest cluster centroid, which changes the cluster assignments of the points. In the second step, new values for the cluster centroids are computed, which are the arithmetic means of the points assigned to each cluster. This step changes the positions of the centroids, which may change the cluster assignments, and the iteration continues.

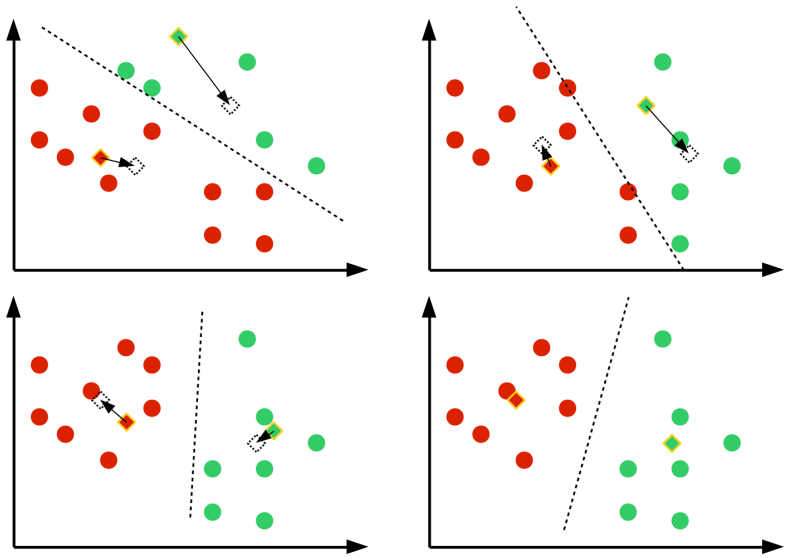

**Figure 6.6:** *The principle of k-means.*

The k-means method is proven to converge regardless of the specific values of the random initialization. A stopping criterion can be the constancy of the cluster centroids or point assignments (or if the change is below a threshold). It is important to note that the value of $k$ in the method's name and operation is chosen by the designer and its careful selection significantly affects the algorithm's performance. The problem is that increasing $k$ will always increase the compactness of the resulting clusters, and indeed, $N$ points can be assigned to $N$ clusters with zero error. This would, however, mean assigning each pixel of an image to a separate object, which has little practical value. Therefore, it is useful to plot the error as a function of the number of clusters and choose the elbow point of this function, which is the number of clusters where the error no longer decreases significantly with increasing $k$.

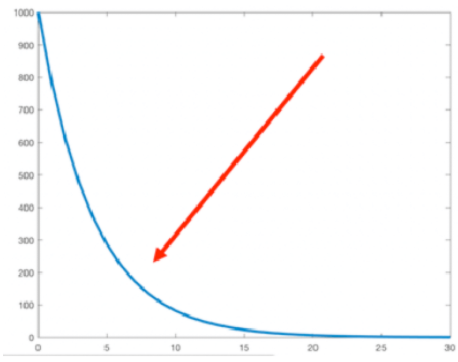


**Figure 6.7:** *Choosing the number of clusters. Although clustering error monotonically decreases, it is advisable to choose the elbow point indicated in the figure, where significant improvement is achieved by increasing the number of clusters.*

### 6.3.2 Mixture of Gaussians

In the k-Means method, each point is assigned to exactly one cluster. However, this does not have to be the case; some points may belong to multiple clusters with different probabilities. In this case, we speak of soft or fuzzy clustering. An example of such an algorithm is the Mixture of Gaussians (MoG), which describes the set of points using $k$ independent Gaussian distributions. The goal of the method is to determine the parameters of the distributions such that the probability of the data set is maximized. It uses the Expectation-Maximization (EM) algorithm for this purpose.

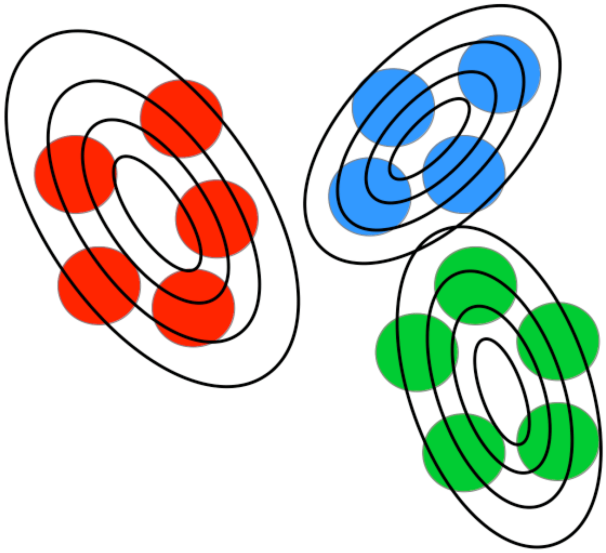

**Figure 6.8:** *The principle of MoG clustering. The figure shows the contour lines of the Gaussian distributions describing each cluster.*

The EM algorithm is a two-step iterative process that initializes the Gaussian distributions randomly and then repeats the following two steps until convergence:

1. **Expectation:** Each point is assigned to the Gaussian distribution that gives the highest probability for that point.
2. **Maximization:** The parameters of each distribution are estimated using the maximum-likelihood method based on the assigned points.

The parameters of the normal distribution containing $X$ points can be estimated as follows (Maximum Likelihood):

$$\begin{aligned} \hat{\mu} &= \overline{X} \\ \hat{\Sigma} &= \frac{1}{N-1}(X-\hat{\mu})^T(X-\hat{\mu}) \end{aligned} \tag{6.1}$$

It can be seen that the steps of EM and k-means are quite similar. The key difference is that the MoG can result in clusters of varying widths and shapes, whereas this is not possible with k-means.

### 6.3.3 Mean Shift

The last important clustering method is mean shift, which begins by defining a weighting kernel. In image segmentation, this is most commonly a flat or Gaussian kernel. It is also necessary to define the size of the kernel, which affects the granularity of the final segmentation. However, it is important to note that with mean shift, there is no need to specify the number of clusters, as this is

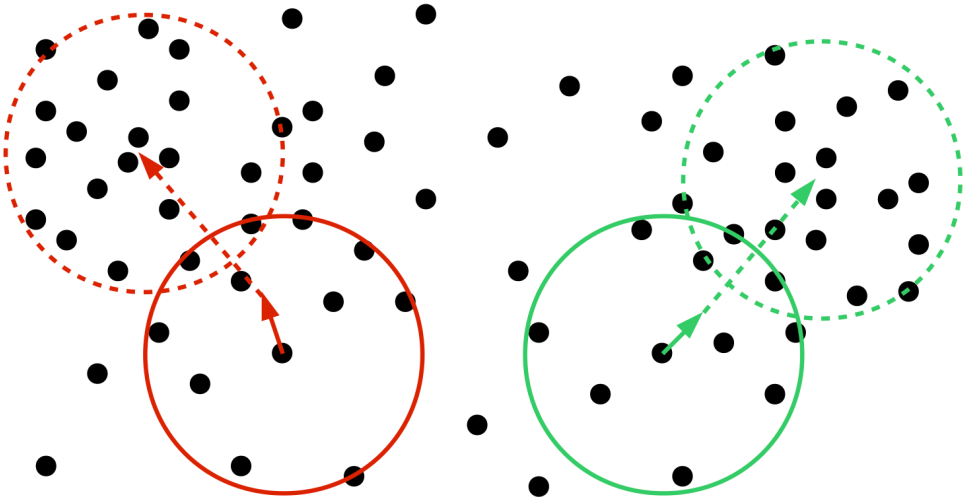

**Figure 6.9:** *The principle of mean shift clustering.*

automatically determined during the algorithm's execution. This presents a significant advantage over the previous two methods.

During the execution of mean shift, we traverse the points with the kernel, then compute the average of the points covered by the kernel (weighted by the kernel weights) at each position. The point is then shifted to this computed average. This process is repeated until no further changes occur.

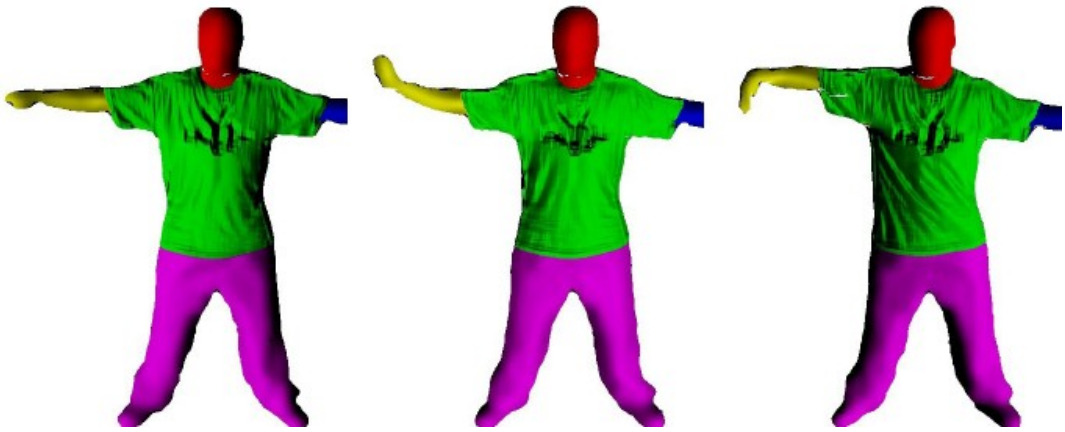

**Figure 6.10:** *The result of the mean shift algorithm.*

## 6.4 Region-Based Methods

One of the major drawbacks of the methods listed so far is that they only consider intensity or color information. Since most objects in the universe are spatially coherent (and hence their images are also coherent), it is a fundamental criterion that the segments obtained during segmentation should also be coherent. Various region-based methods address this problem, such as region growing and region splitting, as well as segmentation methods based on graph cuts.

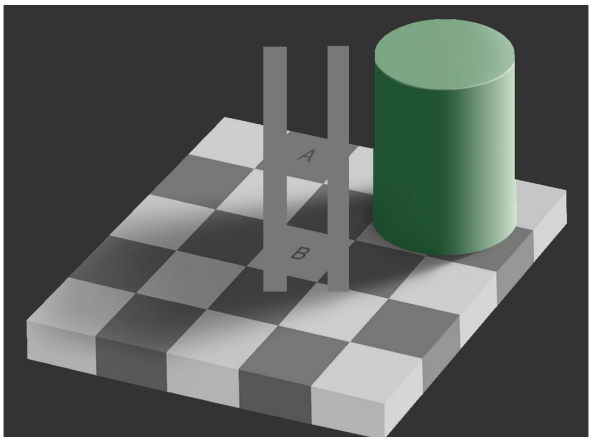


**Figure 6.11:** *The drawback of using intensity/color: Due to various real effects, clearly distinct regions in the image may have the same intensity, leading to them being assigned to the same cluster.*

### 6.4.1 Region Growing

In the region growing process, the algorithm first selects a few seed points from the image. These region seeds can be chosen based on the intensity of the image points or, for example, placed uniformly on a grid. These seeds will serve as the initial values for the regions. Starting from these seeds, the algorithm examines the still unlabeled neighbors of each region and evaluates a region membership function. If this membership function returns a positive result, the examined point is added to the region; otherwise, it is not. The algorithm stops when no more points can be added to any region.

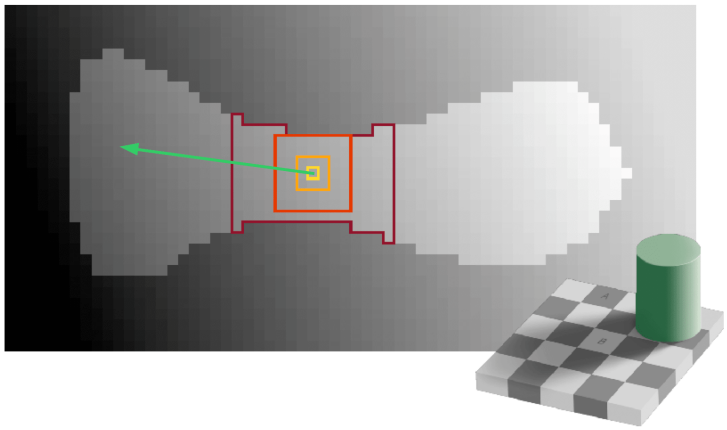

**Figure 6.12:** *The principle of region growing.*

The outcome of the region growing algorithm is influenced by several factors. One of these factors is the choice of seed points, which is advisable to select using the image histogram. It is also worth discarding regions that are too small in the result, as they are likely to be products of some local noise or error. However, the most important factor is the criterion for region membership. Most often, this is decided based on the intensity difference between neighboring pixels or between the old centroid and the examined pixel, using a threshold. For specialized images, color or texture similarities can also be utilized.

### 6.4.2 Split & Merge

The region splitting procedure provides a solution to these problems. In its initial state, it considers the entire image as a single, coherent segment. The algorithm then iteratively traverses all

segments, and if a segment meets a homogeneity criterion specified by the designer, it remains unchanged. Otherwise, the segment is divided into four equal-area parts, and these new segments are examined as well. This process continues until no new segments are created.

However, after splitting, it is likely that many otherwise contiguous regions will be separated. Therefore, a merging phase follows the splitting, which also merges neighboring regions based on a similarity criterion if necessary. This method allows for a fast, globally informed approach that is significantly more robust to noise and almost completely invariant to the choice of neighborhood. The drawback of this method is that the boundaries of the segments are not always entirely accurate due to the square-based splitting.

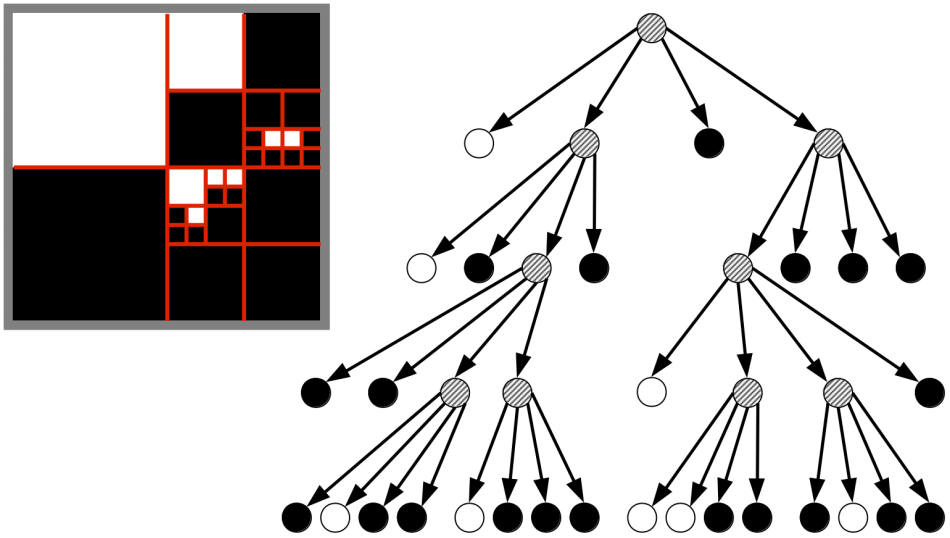

**Figure 6.13:** *The principle of the Split & Merge algorithm.*

## 6.5 Motion Segmentation

In videos, an important task is the detection and segmentation of motion, which can be considered a form of foreground segmentation introduced earlier, where the basis for segmentation is not color or intensity, but motion. One of the simplest ways to detect motion is by creating a difference image from frames taken at different time points (usually not immediately sequentially, but with a 0.5–1 second difference). This highlights pixels that have undergone significant changes between frames. The resulting difference image is thresholded, and by calculating the area of the ”1” pixels in the binary motion image, we can determine if motion has been detected in the image.

The simple method described above has several drawbacks. One of them is that in the case of a moving object, the method will detect motion in both positions of the object in the images, making the resulting motion image ”ghosted,” meaning the moving object will appear in two places. Another issue is that various events in the video may cause significant differences in intensity values between consecutive frames, which are not considered motion but still affect the results. A good example of this is changes in lighting, which is one of the most significant disruptive factors in outdoor videos. In indoor videos, changes such as the camera’s automatic white balance adjustment can also cause such changes.

To address the above issues, motion detection based on the Gaussian background model is used. This procedure aims to separate a static background from a moving foreground. It creates a statistical model of the background using a moving average approach and computes the expected value and standard deviation of each pixel. During operation, the difference image between the current frame and the background image is computed and thresholded. The results obtained do not include ghosting and allow gradual intensity changes (e.g., sunlight) to be incorporated into the background without false motion detection.

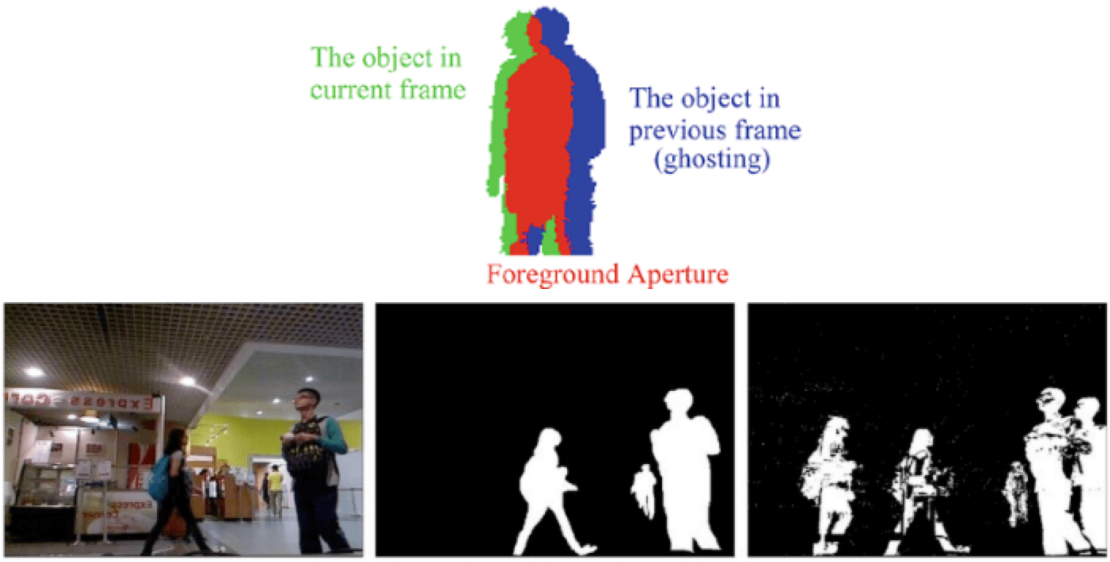


**Figure 6.14:** *Appearance of ghosting.*

The standard deviation values calculated for each pixel in the background model can be used to determine an adaptive threshold. This way, the model can automatically be less sensitive in areas where motion is frequent and more sensitive in areas where it is rare. This solution is useful in cases where many areas in the image might show frequent, irrelevant motion (e.g., bushes and trees in windy conditions).

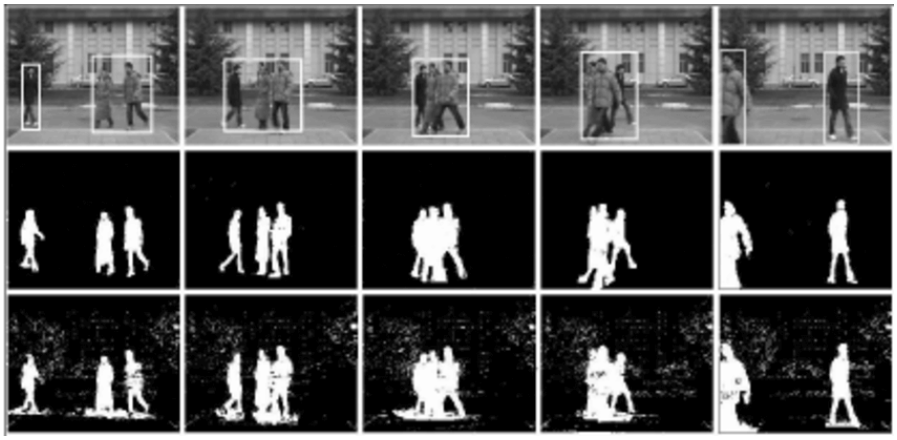

**Figure 6.15:** *The principle of motion segmentation based on the Gaussian background model. The middle row shows images produced with variance-based difference thresholding, while the bottom row shows results with a constant threshold.*

**Application:** In spatial monitoring and security applications, it is important to store the captured footage for future use. This generally requires an enormous amount of storage space due to the size of the videos and the required redundancy. However, the required storage can be significantly reduced by only storing those segments where motion has been detected, as footage where nothing has happened is obviously not valuable. The performance of such systems is greatly improved by using a more robust background model approach, particularly in outdoor footage.

## 6.6 Graph Cut

It is worth mentioning graph cut-based segmentation methods, which come in various forms. A common feature of these methods is that they describe the image as a weighted, undirected graph,

where the graph's nodes represent pixels or local pixel groups. The graph edges are defined only between neighboring pixels or groups, and the weights on these edges express the dissimilarity between pixels or groups. The essence of most such methods is to cut the constructed graph into multiple parts in a way that minimizes the cost of these cuts. The cost of the cuts is determined based on the weights of the edges removed from the graph.

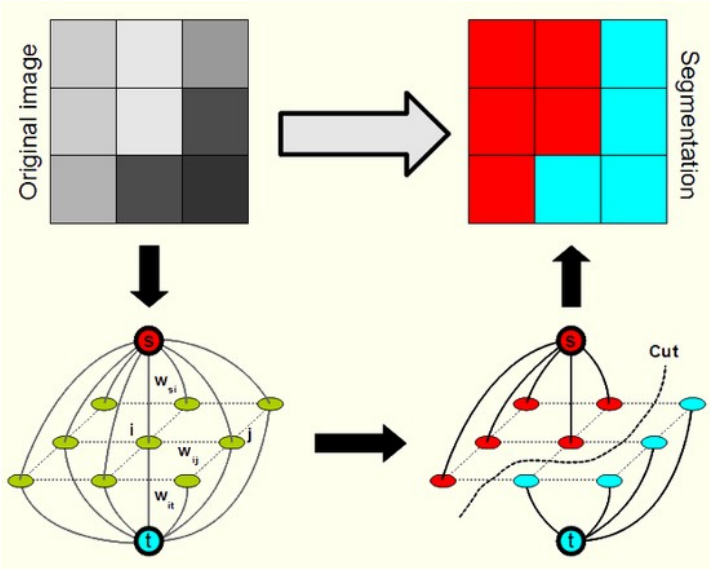


**Figure 6.16:** *The principle of graph cut segmentation.*

**Application:** It is worth noting that segmentation methods are often used for so-called superpixel segmentation as well. A superpixel is a relatively small, compact image region with certain homogeneity properties. Large, complex objects are usually composed of many superpixels. The same algorithms can be used for segmentation in this manner, with parameters adjusted to find many small segments. There are also specifically developed methods for this purpose. Superpixels are extremely useful in various applications, such as speeding up manual image labeling (which is necessary for creating training databases for machine learning) using this method.

## 6.7 Watershed

The last popular algorithm is the watershed method. The essence of this method is to imagine image intensity values as a topographic map (i.e., the brighter a pixel, the higher it is above the image plane). Following this, we find the local minima in the image and start "flooding" the image from each minimum simultaneously. When the floods from two different minima meet, we have found the watershed line, which is the boundary between the two segments.

One major advantage of the Watershed algorithm is that it easily allows manual intervention, as the starting points for the flooding can be defined manually. This method can achieve quite good quality segmentation even in difficult images.

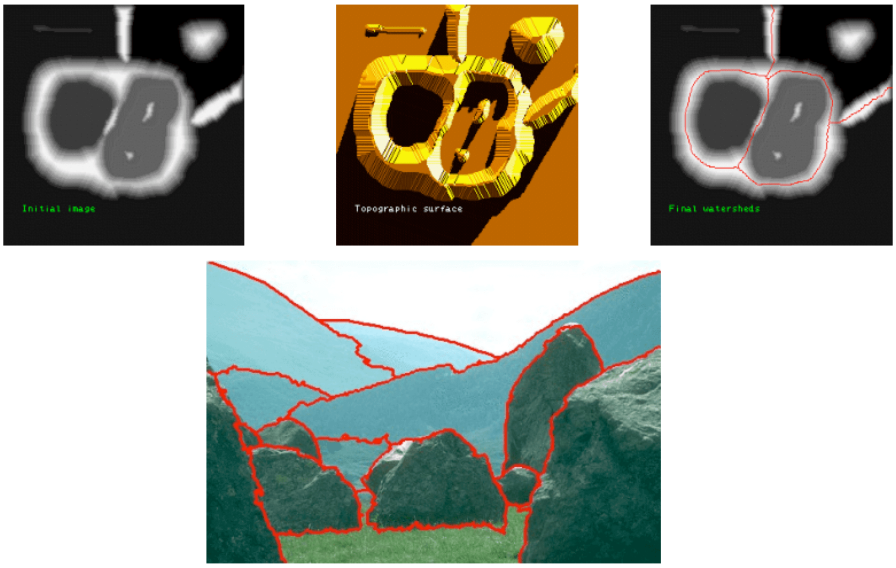


**Figure 6.17:** *The principle of the Watershed algorithm.*

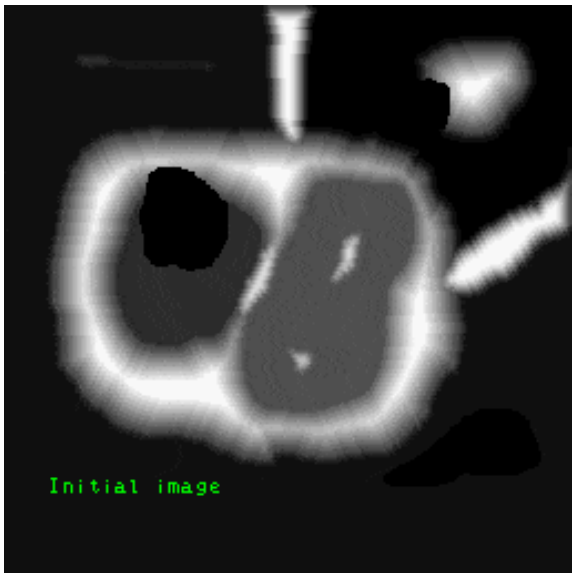


**Figure 6.18:** *The principle of the marker-based Watershed algorithm.*

# 7 Binary Images

## 7.1 Introduction

We have already mentioned that there are several types of images. Among them, the most common are single-channel grayscale images and three-channel color images. However, binary images, which have only two states, are also frequently encountered. For example, in edge detection, pixels corresponding to edges take on one value, while the rest are set to zero. Such binary images can also result from various other operations, such as thresholding, color detection, or complex object detection procedures.

In this subsection, we will examine binary images where one of the two states is ”1”, representing the objects relevant to the application, while ”0” represents the irrelevant background. It is important to note that, for display purposes, the two states of binary images are typically represented by maximum white and minimum black colors. However, there is no universal convention for which color represents the object and which represents the background. In this document, white will consistently represent the object, but other sources may use the reverse.

## 7.2 Morphology

The first important family of algorithms belongs to binary morphology, or morphological operators. These algorithms are primarily used for performing operations on binary images.

### 7.2.1 Erosion and Dilation

In practice, no matter how sophisticated the method used for object separation, it will not be perfect. Thus, as with color and grayscale images, binary images also require improvement. Since binary image errors are of a different nature, the methods used to correct them also differ significantly. Binary images generally exhibit two characteristic types of errors: one is the presence of pixels that actually belong to the background but are labeled as objects, and the opposite is the mislabeling of object pixels as background. The former error can cause false objects to appear in the image, or objects that are separate in reality to be incorrectly merged. The latter can result in holes within objects or mistakenly separate a continuous object.

Erosion and dilation operations can be used to address these two binary image issues. Both operations require the definition of a structuring element. The structuring element is a window of arbitrary shape that is applied to the image in each position similar to a convolution operation, and some logical operation is performed with it. However, unlike convolution kernels, the values of the structuring element can only be ”0”, ”1”, or occasionally undefined (”Don’t care”). Traditional logical operations can be defined between binary images or between a binary image and a structuring element:

In the erosion operation, a pixel in the result image is set to ”1” if the structuring element perfectly fits the input image at that position, meaning all corresponding pixel values match. In contrast, in the dilation operation, the output is set to ”1” if the structuring element and the image match in at least one position. When performing erosion with a structuring element consisting entirely of ”1”s, the edges of objects are set to zero, thereby reducing the size of the objects. Conversely, dilation will increase the size of objects.

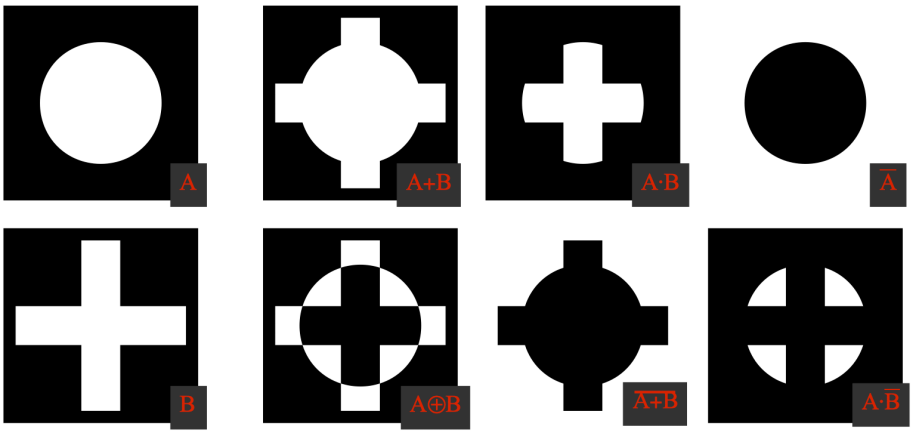


**Figure 7.1:** *Logical operations between binary images. These operations are most commonly performed on a per-pixel basis.*

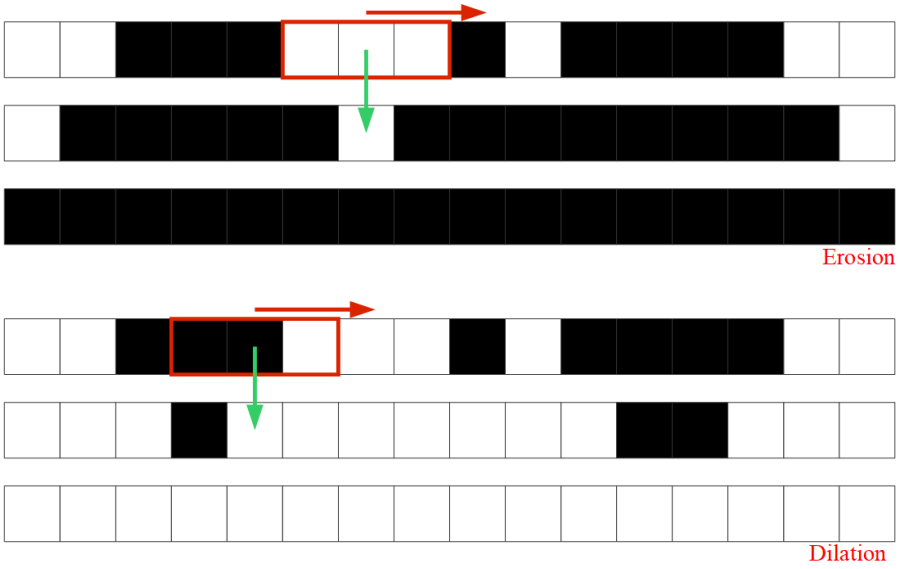


**Figure 7.2:** *The principle of erosion and dilation.*

It is important to note that these operations can be performed with any structuring element, allowing for various specialized filtering on the binary image. For example, a narrow, rectangular structuring element can remove edge-like objects on the binary image that are perpendicular to the longer side of the structuring element, while preserving edges parallel to the element. Similarly, a specially shaped structuring element can filter out all objects whose shape does not match that of the structuring element. This method can be used, for example, to highlight corners in a binary image.

### 7.2.2 Opening and Closing

A major drawback of erosion and dilation operations is that they alter the size and shape of objects, which means that measurements taken after these operations may not be accurate. For this reason, in practice, these procedures are never used in isolation but rather in combination. The two most commonly used operations are opening and closing. In the opening operation, we first perform a predetermined number of erosions, which remove small, noise-like objects, and then perform the same number of dilations, which restore the remaining objects to their original size. It is important to note that the dilation used in opening is erosion-free, meaning that we only set a previously ”0” pixel to ”1” if it does not change the number of independent components. Thus, opening can also

**Figure 7.3:** *Erosion and dilation can also be performed on grayscale images. In this case, dilation behaves as a maximum filter (center), and erosion behaves as a minimum filter (right).*

separate objects that are slightly merged.

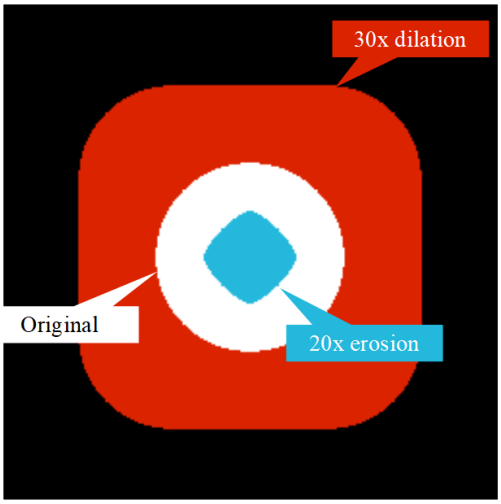


**Figure 7.4:** *The initial (white) circular object undergoes significant changes in size and shape after erosion/dilation with various square structuring elements.*

The closing operation is the opposite of opening. First, a predetermined number of dilations are performed to fill in holes within objects, and then the same number of erosions are performed to restore the objects to their original size. Of course, these two operations can be performed sequentially, thus correcting both types of image errors.

## 7.3 Topology

After improving the binary image, we have a relatively high-quality image of the relevant objects, on which various measurements can be performed. The field of topology, which primarily deals with the relationships between objects, provides significant assistance in this regard.

### 7.3.1 Neighborhood

Before proceeding, it is worth discussing the concept of pixel neighborhood. There are fundamentally two simple conventions to determine if two pixels are neighbors: 4-neighborhood and 8-neighborhood. In the 4-neighborhood convention, each pixel has four neighbors located to its left, right, above, and below. In the 8-neighborhood convention, in addition to the four aforementioned neighbors, each pixel has four more neighbors located diagonally.

Whichever neighborhood convention is used, it will violate the so-called Jordan property of the image plane. The Jordan property states that a plane is divided into exactly two regions by a connected, closed curve. This property is intuitive to humans, and it would be beneficial if it held true in the image plane as well. However, as shown in the figures below, in the case of 4-neighborhood, the white pixels are not neighbors with each other in the given configuration, yet the black-background area is divided into two regions. In the middle figure, using 8-neighborhood, the white pixels form a single connected, closed curve, while the background pixels remain in a single connected region.

To maintain the Jordan property on the plane, we need to use 4-neighborhood for one of the object and background regions and 8-neighborhood for the other. In the right-hand figure, 8-neighborhood is used for the white foreground object, thus these pixels form a closed curve, while 4-neighborhood is used for the background, so the central pixel will not be a neighbor to any black pixel, thus the background is genuinely divided into two regions. In the figures, it is assumed that the background extends to infinity beyond the window boundaries, so the black pixels in the corners always belong to the same connected region.

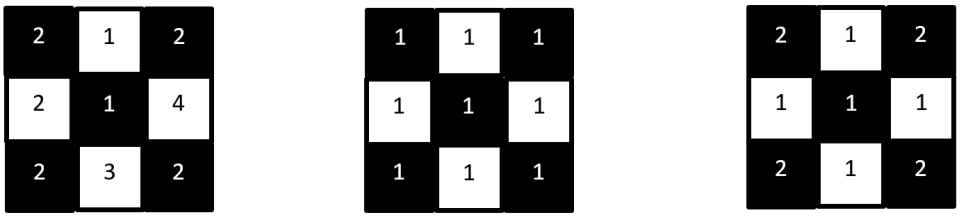


**Figure 7.5:** *Illustration of the Jordan property. In the left case with 4-neighborhood, we have 2 background and 4 foreground components. In the middle case with 8-neighborhood, both the foreground and the background are connected. In the right case, the foreground uses 8-neighborhood and the background uses 4-neighborhood, resulting in one connected foreground and two background regions.*

It is also worth noting that, similar to neighborhood, we need to define a distance metric in image space to perform measurements (especially length). We have several options: we can choose the Manhattan distance, the Chebyshev distance, or the Euclidean distance. Although the latter is computationally more demanding, the former two can lead to significant inaccuracies.

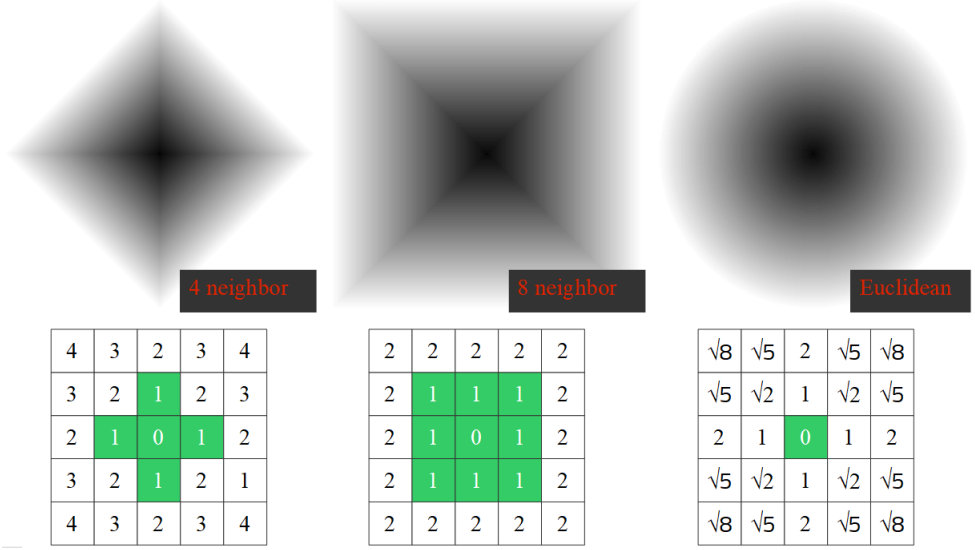


**Figure 7.6:** *Metrics based on different neighborhoods and the Euclidean distance.*

### 7.3.2 Skeletonization

After clarifying this, we can determine various properties of objects in the binary image. The first of these is the object's skeleton. The skeleton is essentially a representation where each ”1” value pixel has exactly 1 or 2 ”1” value neighbors, but topologically it is equivalent to the original object. Alternatively, the skeleton can be imagined as the collection of origins of circles with maximum diameter that can fit into the object.

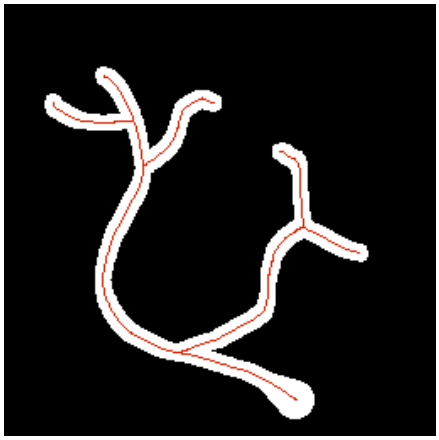

**Figure 7.7:** *The skeleton of a binary object.*

The skeleton of an object is most commonly determined using some iterative erosion procedure (thinning). In this case, erosion is performed only if it does not remove significant points, such as endpoints or points whose removal would split the object. It is important to note that these points can fortunately be detected in local windows, which significantly facilitates this operation.

### 7.3.3 Object Labeling and Counting

The first essential algorithm to perform on binary images is object counting and labeling. The purpose of this procedure is to assign a unique label to each separated object and then determine the number of objects from the highest label number upon completing the labeling. There are fundamentally two common methods for labeling: recursive and sequential methods.

The recursive method is an extremely simple algorithm. Its steps are as follows:

1. Find the first unlabeled ”1” pixel and mark it with label L.
2. Label all ”1” value neighbors of this pixel with label L and invoke step 2 on all neighbors.
   (a) If there are no more unlabeled ”1” value neighbors, the algorithm stops.
3. Jump to step 1 and increment L.

The steps of this method are very simple; however, for large, connected objects, recursion can go very deep, which can cause problems especially with low-performance processing devices. In such cases, it is advisable to use the more complex sequential method. This algorithm scans the image pixels in sequence and assigns a new value to each pixel according to the following rules:

- If only the top or only the left neighbor of the pixel is labeled, then the current pixel receives their label.
- If both the top and left neighbors have the same label, then the current pixel also receives that label.

- If the top and left neighbors have different labels, then the pixel receives the label of the top pixel, and we record the equality of the two label values in a separate storage.
- If the pixel has no labeled neighbors, a new label is introduced for it.

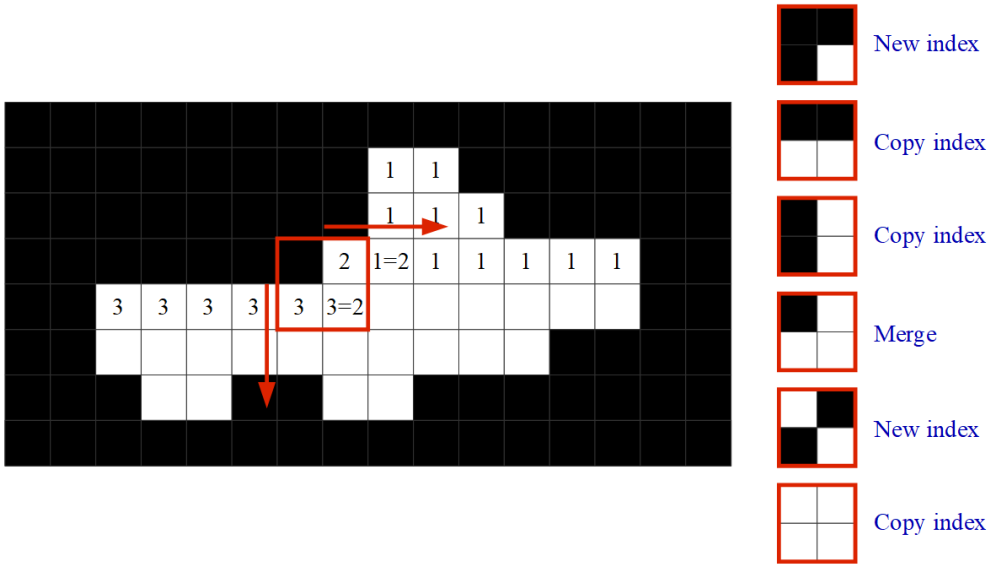


**Figure 7.8:** *The principle of the sequential object labeling algorithm.*

At the end of the sequential algorithm run, we need to pass through the image again to assign a common label to the objects instead of the recorded labels for the same object. It is worth noting that this algorithm can also be used for object counting. In this case, we always increase the count when a new label would be assigned compared to the previous version and decrease it when labels would be merged.

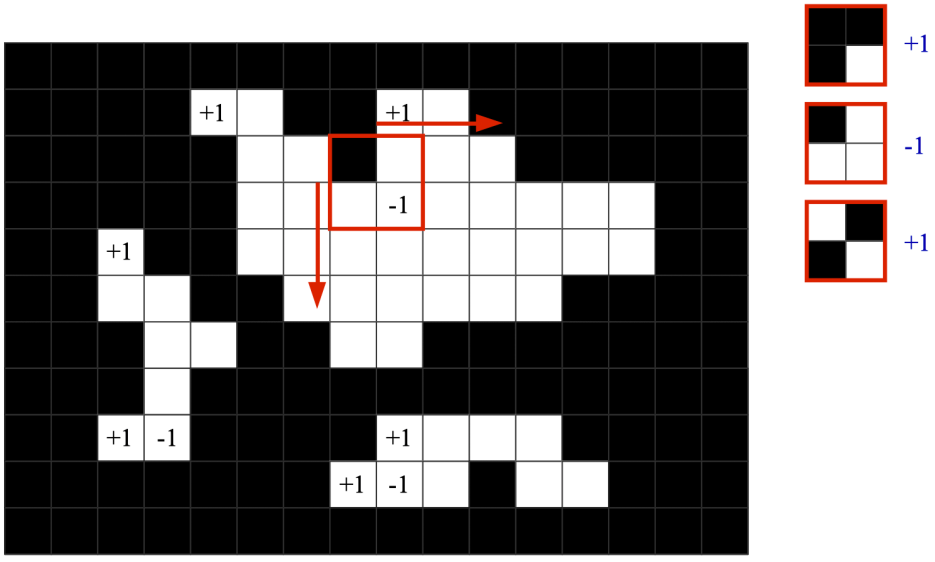


**Figure 7.9:** *The principle of the sequential object counting algorithm.*

**Application:** A practical application of processing binary images is the automatic counting of cash coins, which can speed up transactions involving such payment instruments. For this purpose, it is advisable to create a special device where coins can be placed on a specific colored surface, monitored by a camera connected to a computer at a known distance. Due to the specific background, the image can be easily binarized with thresholding, marking the coins with ”1”. The

separation and hole-free status of the coins can be ensured through sequential opening and closing operations. Subsequently, labeling is used to distinguish individual objects, from which the denomination of the coin is inferred based on its area. Of course, the authenticity of the coins must also be verified through pattern matching.

## 7.4 Object Properties

Once we have successfully separated the different binary objects, the next important step is to determine various characteristics for each object, which will allow us to describe and identify them. These characteristics often make use of the projections of the objects. A projection is a compact representation where we count the number of "1" pixels in each column/row along the two axes of the image. It is worth noting that the original shape of the object can be reconstructed from multiple projections, similar to how tomographic procedures in medicine work.

### 7.4.1 Position, Orientation

An extremely important descriptive quantity for individual objects is the moment, or in technical terms, the momentum. There are several orders of moments, among which the zeroth-order and the first-order moments are particularly useful for us. The zeroth-order moment is simply the sum of pixel intensities. Since we are dealing with binary images here, the zeroth-order moment of an object will give its area. The area and the centroid of the object can be determined using the zeroth-order and first-order moments according to the following formula:

$$\begin{gathered} M_{00} = \sum_{x=0}^{W}\sum_{y=0}^{H} I(x,y); \quad M_{10} = \sum_{x=0}^{W}\sum_{y=0}^{H} x * I(x,y); \quad M_{01} = \sum_{x=0}^{W}\sum_{y=0}^{H} y * I(x,y) \\ A = M_{00}; \quad C = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}\right) \end{gathered} \tag{7.1}$$

Moments can also be used to determine the orientation of an object. Specifically, we need to find the axis with the smallest moment, which can be clearly determined using second-order moments, provided the object is not symmetric (the orientation of a symmetric object is not unique). The orientation can be determined as follows:

$$\begin{gathered} M_{20} = \sum_{x=0}^{W}\sum_{y=0}^{H} x^2 * I(x,y); \quad M_{02} = \sum_{x=0}^{W}\sum_{y=0}^{H} y^2 * I(x,y); \quad M_{11} = \sum_{x=0}^{W}\sum_{y=0}^{H} xy * I(x,y) \\ M_x = M_{20} - \frac{M_{10}^2}{A}; \quad M_y = M_{02} - \frac{M_{01}^2}{A}; \quad M_{xy} = M_{11} - \frac{M_{10}M_{01}}{A}; \\ \Theta = atan\left(\frac{M_x - M_y + \sqrt{(M_x - M_y)^2 + 4M_{xy}^2}}{2M_{xy}}\right) \end{gathered} \tag{7.2}$$

It is worth noting that there are several other metrics for object orientation, which are generally simpler to compute. If the bounding rectangle of the object is known, its dimensions and proportions can be used to characterize the orientation. Similarly, the direction of the largest distance within the object or the largest distance from the centroid can also be determined.

### 7.4.2 Additional Metrics

Binary objects can have additional metrics. One of the first of these is the description of the object's shape. One method for this is chain-code based boundary representation. In chain-code usage, a

number is assigned to each direction [0–3] or [0–7], depending on the neighborhood convention. Then, starting from a specific point, we traverse all the points on the object's boundary, recording the direction at each step. Returning to the starting point gives us a descriptor of the object's contour.

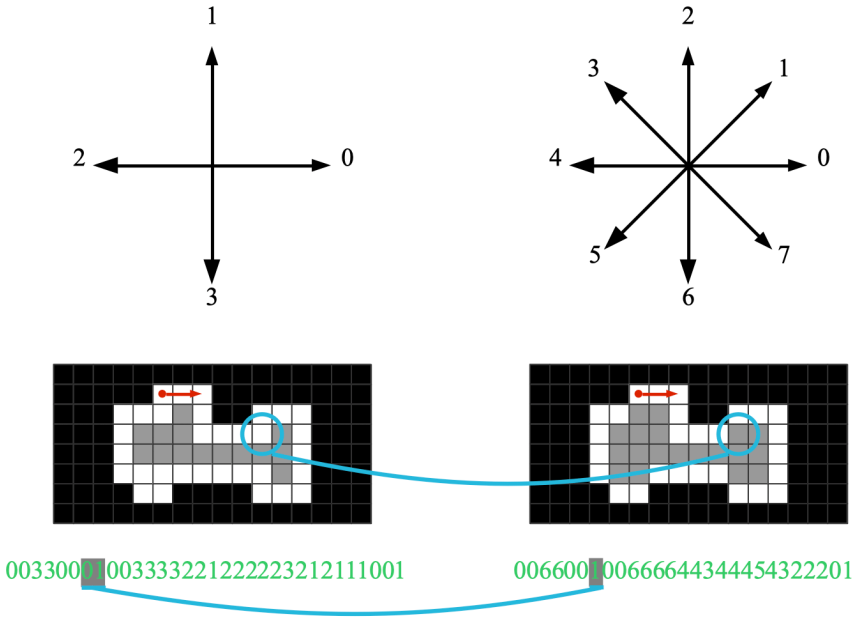


**Figure 7.10:** *Method of generating chain-code.*

Chain-code can be used to determine the perimeter of the object; however, it must be noted that (especially in the case of 4-neighborhood) the boundary traversal is zigzagged, which artificially increases the perimeter length. Therefore, using the Euclidean distance of individual steps provides a much more accurate estimate. It is worth noting that the previously discussed operations can be used to produce a binary image where only the boundaries of the objects are marked with "1". For this, erosion needs to be performed on the image, and by subtracting this eroded image from the original, we obtain the so-called contour image.

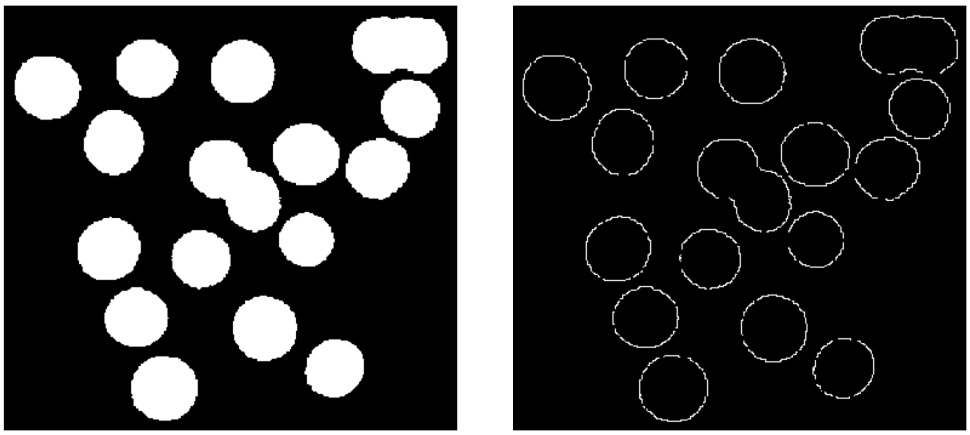

**Figure 7.11:** *Contours of binary objects.*

When the perimeter, area, and centroid of an object are available, many other shape descriptors can be assigned to individual objects, which help in their identification and classification. One of the simplest examples of such descriptors is the ratio of the object's perimeter to its area.

Another common solution is the use of the contour footprint, which can be computed by measuring the distance from the object's centroid to the boundary in all directions starting from a certain

direction, and using this function as a descriptor of the object's shape. The starting point is usually chosen to be the direction with the maximum distance, so the same descriptor is obtained even if the object is rotated. If the resulting descriptor function is normalized, we can obtain a scale-invariant descriptor.

An extremely common descriptor is the Euler number, which is the difference between the number of connected regions in an object and the number of holes within the object. This metric is very useful in cases where the object's shape may be subject to significant distortion, as most common geometric distortions do not change this property, allowing objects with different Euler numbers to be distinguished.

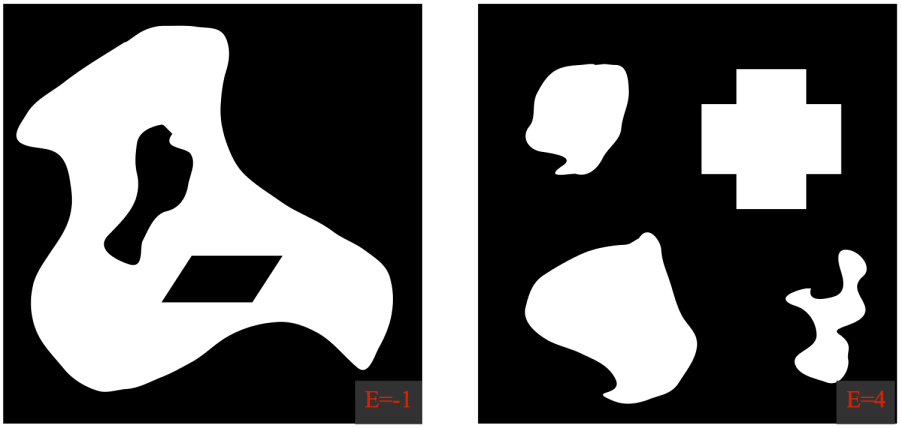


**Figure 7.12:** *Calculation of the Euler number.*

# Part II

# Learning Vision

# 8 Neural Networks

In the previous section, we introduced the steps of traditional vision systems, and in the final phase, we gave a brief insight into the methods of machine learning. Modern vision systems based on artificial intelligence differ from traditional methods not in that they use machine learning (since most traditional methods also employ it in the final step), but rather in that learning-based vision systems are almost entirely composed of learning elements, implementing so-called end-to-end learning.

In the following chapters, we will discuss the theory behind these methods, starting with the learning models used in neural network systems. After that, we will explore the cost functions and the optimization methods used for them. Subsequent chapters will cover convolutional architectures developed for processing image information, followed by practical insights into training neural networks. Finally, we will discuss solving higher-level tasks such as segmentation, detection, and dynamics.

## 8.1 Machine Learning

The fundamental goal of the field of computer vision is to extract high-level information from images. However, as discussed in the introductory lecture, certain problems can make this quite challenging. This naturally raises the question of whether we can somehow integrate the capabilities of human intelligence into computer vision methods, thereby enabling problem-solving. The field of artificial intelligence is vast, and numerous algorithms are available. A significant portion of these algorithms are exact algorithms, meaning they can be expressed as a series of clearly defined instructions and logical conditions. These algorithms essentially exploit the intelligence of their creators to achieve intelligent behavior. Due to the lack of understanding of how human vision works, these algorithms would not help solve the aforementioned problems.

However, there is another class of artificial intelligence methods known as learning algorithms. Learning methods provide a general, parameterized model for solving a problem, and during the learning process, a training dataset is used to adjust these parameters in such a way that the initial general model becomes specialized for solving the given problem. A significant advantage of these algorithms is that they allow us to solve problems whose solutions we do not fully understand, as long as we can generate a suitable dataset for training the algorithm.

A key drawback of machine learning methods, however, is that the resulting model is typically a black box, meaning that examining the model does not necessarily bring us closer to understanding the solution to the problem. This black box nature makes it extremely difficult to understand the reasons for any potential errors or misclassifications, and how to avoid them in the future. It is also important to note that machine learning methods usually determine the parameters during training using statistical methods or numerical optimization, meaning their correct operation cannot be guaranteed. Despite these disadvantages, in the field of computer vision, and sensing in general, these methods significantly outperform traditional algorithms.

## 8.2 Structure of Learning Algorithms

In machine learning, a learning algorithm's input is uniformly denoted by $x$, the output by $y$, and the parameters by $\vartheta$. Using these notations, the model of a learning algorithm can be expressed as a parameterized function:

$$\hat{y} = f(x, \vartheta) \tag{8.1}$$

Every learning algorithm has an associated cost function (often called an error or loss function), which assigns an error value to the output of the learning algorithm, allowing us to evaluate the algorithm's performance. In addition, every learning algorithm includes one or more optimization methods, which are used to minimize the error function by adjusting the parameters. Most learning algorithms also have hyperparameters, which influence the quality of the solution but cannot be determined through the optimization method. These typically include characteristics of the optimization method or the structure of the model itself.

### 8.2.1 Types of Learning

Machine learning methods can be categorized in many ways, one of the most important being based on the algorithm's output. In regression, the output of the learning system is a continuous number. If the algorithm's output is a binary variable or an integer from a finite set, then we speak of classification. The basic case of classification is binary classification, as a multi-class classification system can be constructed from a composition of multiple binary classifiers. This can be visualized as having a binary classifier for each class, which can distinguish the given class from all others, and the final class is determined by the confidence of each classifier. Another approach could be a system where each classifier can decide between two classes, and the final class is determined using a scoring method similar to sports tournaments.

Another important categorization principle is the nature of the training data used for learning. The simplest form of machine learning is supervised learning. In this case, the training data consists of input-output pairs, meaning that for every input, we know the correct output, and we expect the learning algorithm to correctly predict these outputs with as high an accuracy or precision as possible. It may happen that the expected output is only available for part of the training dataset; in this case, we speak of semi-supervised learning.

However, supervised learning has significant limitations. First, the creation of labeled training datasets is a time-consuming and expensive task. Secondly, the quality of the labeling fundamentally limits the quality of the learning algorithm. Lastly, just as exact algorithms can only be used when we consciously understand the solution to a problem, supervised learning algorithms can only be used when we ourselves can solve the given task. This implies that a supervised learning algorithm will never be able to solve tasks that we ourselves cannot.

There is, however, unsupervised learning, where the training dataset consists solely of input values, and the expected output is completely unknown. Such datasets are relatively cheap to generate since, in most cases, the acquisition of new data can be automated. In these cases, we expect the learning algorithm to find some internal structure in the dataset and, thereby, compactly describe it. The goal of these algorithms is to explain the data seen at the input using a compact model and map out its internal structure. Good examples of such algorithms are the clustering methods (k-Means, MoG) introduced in the earlier section, which can essentially be considered the unsupervised versions of classification. The TLS method, discussed later, can also be interpreted as the unsupervised version of linear regression.

The third main type of machine learning is reinforcement learning, which differs from the other two types in two important ways. First, in reinforcement learning, the algorithm must almost always make a series of interconnected decisions, but feedback on the correctness of the decision sequence is typically not provided after every decision. Secondly, during the feedback process, the algorithm only receives an evaluation of the quality of the decisions; it does not learn what the correct decision would have been. Typical examples of reinforcement learning tasks include various games (e.g., chess, Go, video games) as well as various vehicle control tasks.

### 8.2.2 Difficulties

At first glance, machine learning might seem like a kind of magical method that can easily solve all the world's problems. In practice, however, these methods have plenty of limitations and pitfalls, into which one can easily fall. To avoid these pitfalls, it is always important to remember that these algorithms work based on training data, which means they can only interpret that part of the world covered by the training data. This means that the training data we use must cover all possibilities, as we cannot predict how the algorithm will behave in cases it has never encountered before.

It is also worth noting that, for example, learning vision systems typically cannot learn relationships that require the ability to move or other senses. This might seem trivial when stated this way, but just think of the concept of a chair: "an object you can sit on." I can use this definition to recognize a chair because, visually, we can generally determine whether something is sit-on-able. However, an algorithm that cannot move and only receives disconnected images will never develop the concept of sitting.

Another common pitfall of machine learning is the issue of the learning procedure's complexity. A fundamental human intuition is that if the learning algorithm is not accurate enough, making it more complex ("smarter") will improve performance. There are several ways to increase a model's complexity: increasing the number of input variables and parameters are two obvious methods. Furthermore, the hyperparameters of the learning method generally influence complexity as well. However, increasing the model's complexity is a double-edged sword: whether it helps or hurts depends on the situation.

There are cases when the algorithm is not complex enough to solve the given task. In this case, we observe that the error on the training dataset is quite large, and if we then test the algorithm on new data it has not seen during training, we obtain similarly large errors. This phenomenon is called underfitting. In this case, by increasing the complexity, both the training and testing errors decrease. After a while, however, we observe that while the training error continues to decrease, the testing error first stagnates and then starts to increase with further complexity. This phenomenon is called overfitting.

The cause of overfitting is that the training dataset is not entirely perfect. On the one hand, it is finite, meaning the algorithm's task is to learn how to generalize using the dataset. On the other hand, both the inputs and the outputs are subject to noise, so the training error will not be zero, even in the case of perfect generalization. This means that, after a certain point, the algorithm can only continue to reduce the training error by starting to memorize the correct answer for each individual training data point. As a result, instead of solving the problem in general, the algorithm increasingly resembles an associative memory. This means that for inputs not present in the training dataset, it gives worse and worse answers. Moreover, the more complex the algorithm, the easier it is for it to memorize a large dataset.

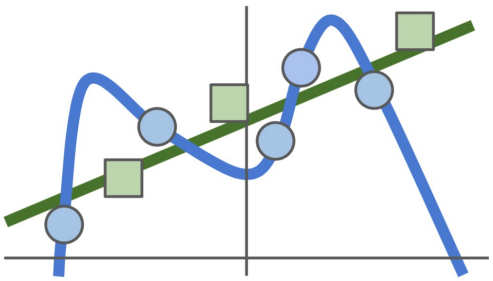

**Figure 8.1:** *Overfitting in the case of one dimension. It can be seen that the blue model learns the noise in the training dataset, so it performs poorly on the test data (green).*

### 8.2.3 The Perceptron Model

The fundamental element of deep learning systems is a linear classifier algorithm, which goes by various names. It is often referred to as a perceptron or neuron, and despite its classification nature, it is also known as logistic regression. The essence of the algorithm's operation is that it arranges the pixels of the image into a single vector, then multiplies this vector by a weight matrix, thereby producing an output vector with as many elements as there are classes to choose from. Each element of this vector can be interpreted as the "goodness" value of one class; the larger the value, the more the image belongs to that class. Formally, the perceptron model can be described as follows:

$$s = Wx \tag{8.2}$$

where $x$ is the input, $s$ is the class goodness, and $W$ is the matrix of parameters or weights (this notation will be consistently used throughout the present chapter). This method of classification can be imagined as the i-th row of the $W$ matrix specifying a direction in pixel space where the goodness of the i-th class increases. Based on this, the decision boundaries between the classes consist of straight segments (in the binary case, a single line/hyperplane).

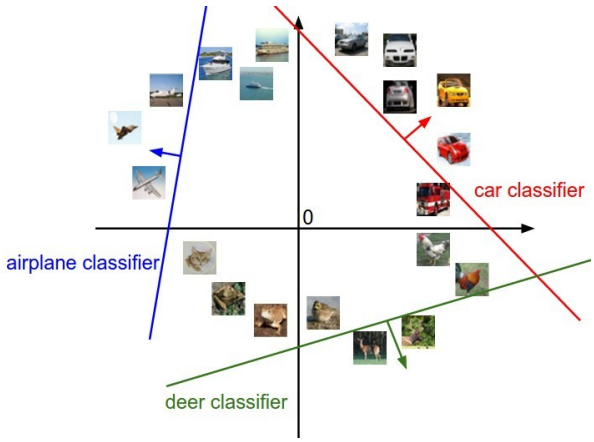


**Figure 8.2:** *In linear classification, the goodness values for each class grow linearly in one direction of the space, while they remain constant in others. The direction of the increase is determined by the row vector of the weight matrix corresponding to the given class.*

## 8.3 The Method of Training

After defining the model of the Perceptron algorithm and analyzing its operation, it is time to discuss the determination of the elements of the weight matrix $W$, which are necessary for the correct functioning of the model. The fundamental question of the method is how to determine the weight values so that the classification is as accurate as possible, and what cost function can be used to effectively measure the algorithm's performance.

### 8.3.1 Loss Functions

At first glance, it may seem practical to measure the quality of the classification by the proportion of correctly identified training data. However, this approach cannot distinguish between models with the same accuracy but different levels of uncertainty. Therefore, we will define entirely new cost functions between the model's output and the prescribed output for the given training data, whose average over the entire training set will provide the overall error measure. One might consider using the squared error between the expected and predicted output, which is the most commonly

used loss function in regression problems. However, approximating output values numerically is not necessarily practical in classification, and while the squared error can be used, better loss functions can often be constructed for these cases.

One frequently used loss is the so-called Hinge or SVM loss function. The principle of this loss function is that we define a margin $\Delta$, and if the score of the correct class is at least this value larger than all other scores, the error is 0. Otherwise, the error increases linearly. This loss function can be understood as a criterion for ensuring "safe" separation. The SVM loss is formally given by:

$$Li = \sum_{j \neq y_i} \max(0, s_j - s_{corr} + \Delta) \tag{8.3}$$

where $s_j$ is the score of the $j$-th class, and $s_{corr}$ is the score of the correct class.


**Figure 8.3:** *Hinge loss visualized.*

There is another loss function commonly used in practice that approaches the problem from a probabilistic standpoint rather than a geometric one. This cost function utilizes the concept of entropy. The idea of entropy is based on the fact that when we want to encode events that occur with different probabilities, it is not efficient to allocate the same number of bits to each event. Instead, more bits should be used for unlikely events and fewer bits for likely ones, minimizing the total number of bits used for all events. The number of bits required to encode an event occurring with probability $p$ is proportional to the reciprocal of the logarithm of $p$. Using this, entropy gives the expected number of bits used for all events:

$$H(p) = -\sum_i p_i \log p_i \tag{8.4}$$

However, if we do not know the true probability distribution $p$ and instead use an approximation $q$, the resulting value will be larger than optimal. This increase is measured by cross-entropy. The closer $q$ approximates $p$, the smaller the cross-entropy. The difference between the two types of entropy is known as the KL divergence, a strictly non-negative function often used to measure the similarity of probability distributions.

$$H(p, q) = -\sum_i p_i \log q_i \tag{8.5}$$

Cross-entropy can be used as a classification loss function in the following way: first, we convert the model's output scores into probability-like values using a normalization function called SoftMax. This function transforms each value into the $[0, 1]$ range, such that the sum of the values is exactly one. The SoftMax function is written as:

$$q_{k,i} = q(y_k|x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \tag{8.6}$$

From here, we define the loss function as the cross-entropy between the expected distribution of labels and the estimated distribution $q$. Since cross-entropy is minimized when the two distributions match, minimizing this function drives the estimated probabilities toward the prescribed values. The expected label distribution is constructed by assigning a probability of 1 to the correct class and 0 to all others. Thus, the cross-entropy loss simplifies to:

$$
\begin{aligned}
L_i = H(p_i, q_i) = -\sum_k p_{k,i} \log q_{k,i} \\
L_i = -\log q_{true,i}
\end{aligned}
\tag{8.7}
$$

where $p_{k,i}$ and $q_{k,i}$ are the prescribed and estimated probabilities of the $k$-th class for the $i$-th training data, and $q_{corr,i}$ is the estimated probability of the correct class. One advantage of the cross-entropy cost function is that it works with probability-like values rather than hard-to-interpret "goodness" values, making the model output more user-friendly. Its downside compared to SVM loss is that the cost is never zero, meaning that SVM loss is more "efficient": it is satisfied with safe separation and allows the model to use the remaining resources for correctly classifying other training data. In practice, however, the difference between the two cost functions is negligible.

### 8.3.2 Regularization

Both loss functions, however, have a fundamental problem. It is easy to see that in both cases, if we simply multiply the current weight matrix of the model by a large number, the value of the loss functions will decrease, while the classification accuracy will remain unchanged since every output score will simply be multiplied by the same constant. Consequently, the norm of the weight matrix will increase indefinitely, leading to numerical issues and an overly confident model.

Therefore, these loss functions are typically used together with a regularization penalty term, which controls the norm of the weight matrix. A common solution is to use the L1 or L2 norm of the matrix as a penalty term. Additionally, there is an elastic regularization method that uses a weighted average of the two types of norms. The final loss function is given by:

$$
\begin{aligned}
L = \sum_i^N L_i + \lambda R(W) \\
R_{L1}(W) = \sum_k \sum_l |W_{k,l}| \\
R_{L2}(W) = \sum_k \sum_l W_{k,l}^2 \\
R_{EL}(W) = \sum_k \sum_l \left(\beta W_{k,l}^2 + |W_{k,l}|\right)
\end{aligned}
\tag{8.8}
$$

where $L_i$ is the loss for the $i$-th training data, $N$ is the number of training data points, $R$ is the regularization term, and $\lambda$ is a hyperparameter that controls the relative weight of the regularization. It is worth noting that in the case of multilayer networks, which will be discussed in the next section, an increase in the norm of the weight matrix can cause overfitting, making regularization a necessary measure to avoid this.

## 8.4 Optimization

One of the biggest unresolved questions from the previous lecture is how to minimize the error function of the perceptron model that we mentioned earlier. Therefore, our first topic will be discussing optimization methods used for minimizing error functions. There is no closed-form solution for finding the optimal weights of the perceptron, so iterative optimization will be necessary.

### 8.4.1 Gradient-Based Optimization

Fortunately, both the model and the cost function are differentiable, allowing us to use gradient-based methods. If we compute the derivative of the error function with respect to the weights (i.e.,

the gradient of the weights), we obtain information on how the weights should change in order to make the error function increase as rapidly as possible. However, by adjusting the weights in the opposite direction of the gradient, we move in the direction of the steepest descent. By repeating this step, akin to "climbing in reverse," we will eventually reach a local minimum.

It is important to note that since we aim to minimize the total error, we must evaluate the error across the entire training dataset at each step. Given that the gradient method only converges after many steps, this is not practical. Therefore, we divide the training dataset into equal-sized, randomly selected subsets (minibatches), and take a step after processing each minibatch. This method is called Stochastic Gradient Descent (SGD). Minibatch sizes are usually powers of two for implementation reasons. Of course, the gradient calculated from a single minibatch will not exactly match the one calculated for the entire dataset, but it will be close enough to guide the method in the right direction. Since executing a single step becomes much cheaper this way, we achieve a significant overall speedup. The gradient method is described by the following formula:

$$W_{k+1} = W_k - \alpha \frac{\partial ||E||^2}{\partial W} \tag{8.9}$$

Where $\alpha$ is the learning rate, a hyperparameter that significantly affects the speed and quality of training. We will discuss how to choose this parameter correctly in a later chapter.

It is also important to note that one of the downsides of the gradient method is that it can easily get stuck in local minima. To address this, we replace the idea of a "reverse climber" with that of a rock rolling downhill. In other words, we add a form of inertia or momentum to the gradient method. In practice, this means that the step taken at any given time is a weighted average of the negative gradient direction and the step taken in the previous time step. This weight typically falls within the range of [0.1-0.9], depending on the application.

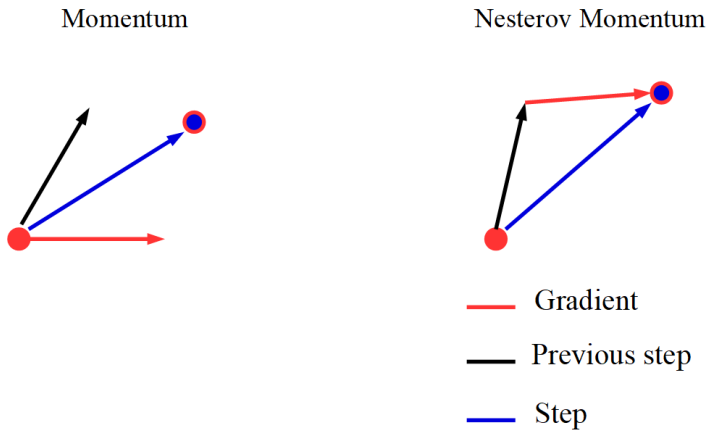


**Figure 8.4:** *The momentum method (left) and Nesterov momentum (right). The latter moves first in the momentum direction, then calculates the gradient there, resulting in slightly more stable performance.*

One of the drawbacks of the stochastic gradient method is that it treats all directions in the multidimensional parameter space as equivalent. However, it might be the case that the cost function is quite steep in one direction, requiring cautious movement to avoid overshooting the minimum with a large step. Meanwhile, the cost function might be flat in another direction, and the optimum could be far away. While the first issue would necessitate reducing the step size, the second would call for increasing it, which is impossible to do simultaneously.

The AdaGrad and RMSProp methods provide solutions to this problem. Both methods track the magnitude of the gradients in different directions and scale the step size inversely to these magnitudes, thus allowing different step sizes for different directions. Among the various optimization algorithms, one of the most popular is the Adam algorithm, which combines the momentum and gradient scaling methods.

### 8.4.2 Backpropagation

As previously mentioned, the capabilities of simple linear models are quite limited, and as a result, they are rarely used for image inputs. However, their introduction was necessary because these simple linear models can be easily integrated into complex nonlinear learning procedures. If we connect the neuron models described in the previous subsection one after the other, we obtain a multilayer, feedforward neural network. Each layer of a neural network has an unknown weight matrix, which can be determined using the previously discussed cost functions and optimization methods.

The only questionable step is the calculation of the derivative of the error function with respect to the weights. A neural network can be envisioned as a computational graph, where each node in the graph implements simple, analytically differentiable functions. If we know the inputs in a computational graph and the functions implemented by each node, we can calculate the output of each node. This is called the forward propagation operation. It is important to note that if we know the functions of the nodes and the derivative of the quantity we are interested in (in this case, the error function) with respect to the node's output, then, using the chain rule, we can easily derive the derivatives with respect to the node's inputs and weights.

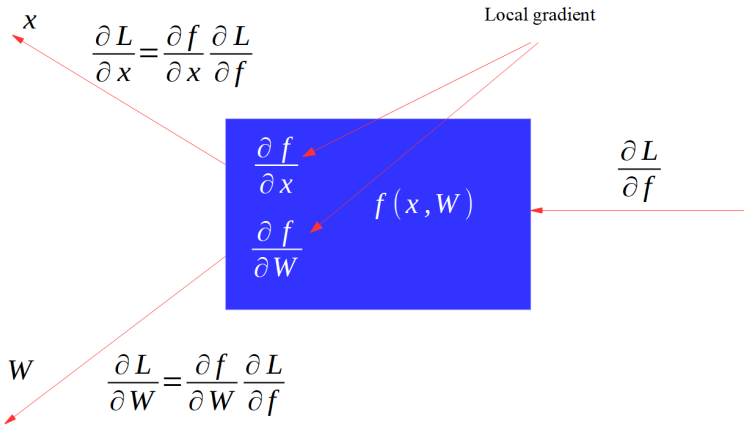


**Figure 8.5:** *Illustration of the chain rule.*

Let $L$ be the error function, $f$ and $x$ the output and input of a given layer, and $W$ the parameters of the layer. With this method, by moving backward through the network, we can determine the gradients of all inputs and weights. This operation is called backpropagation. The only question is how to obtain the derivative of the error function with respect to the output of the last node in the computational graph. Notice that the last operation to be performed during forward propagation is the calculation of the error function, meaning that the output of the graph's final node is the error function itself. The derivative of a quantity with respect to itself is trivially 1. Therefore, everything is in place to compute the gradients of the network.

It is worth noting that the elements of the neural network do not necessarily have to be the perceptron functions we introduced earlier. In practice, neural networks consist of many different types of layers, all of which share the property of implementing differentiable functions. We can even create new types of layers as long as we implement the functions responsible for performing the forward and backward propagation sub-tasks.

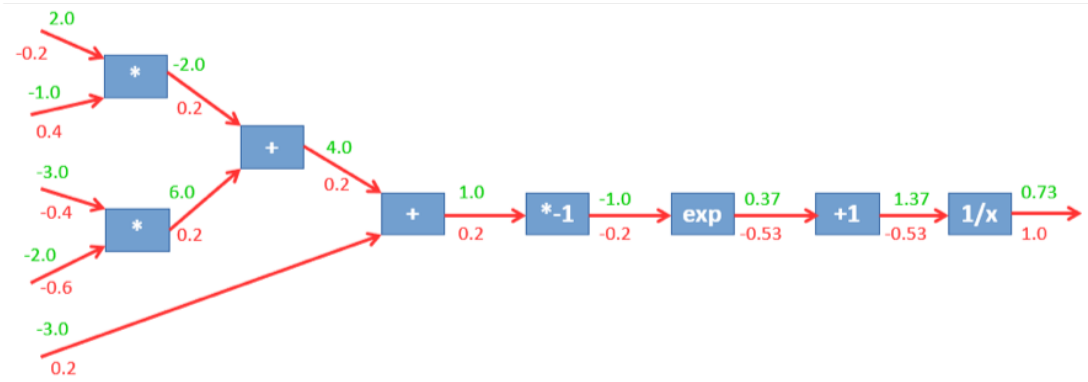


**Figure 8.6:** *Execution of backpropagation on an example graph. Activations are shown in green, derivatives in red.*

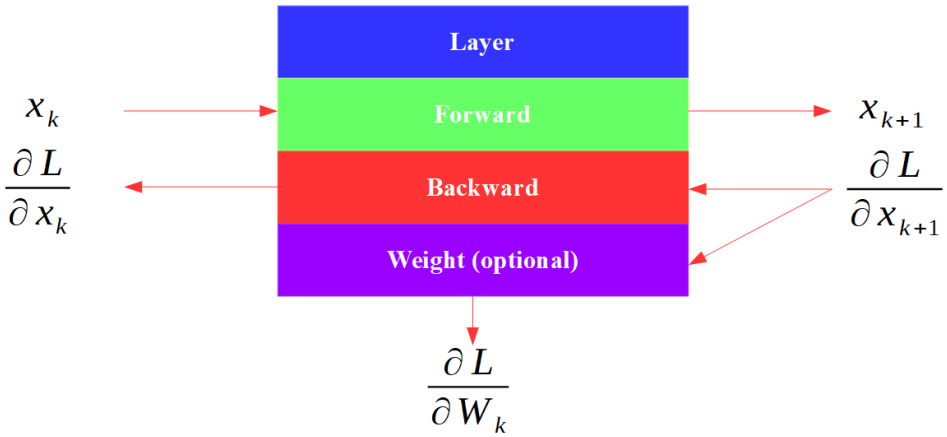


**Figure 8.7:** *A general interface provided by a layer, containing components necessary for both training and prediction.*

# 9 Convolutional Neural Networks

## 9.1 Introduction

In the previous lecture, we became familiar with the basics of machine learning, particularly the method of linear classification. However, we also acknowledged that these algorithms are not complex enough to perform image classification with high quality. Fortunately, these simple classifiers can still be used as building blocks for a larger artificial intelligence model. The topic of today's lecture will be deep neural networks built from simple linear models.

## 9.2 Convolutional Neural Networks

While the linear neuron model introduced earlier does appear in networks used in computer vision, the majority of such networks are not composed mainly of these layers. This is because the linear (also known as fully connected) layer establishes a connection between every input and output, resulting in a large number of parameters, which promotes overfitting and makes the network more difficult to store. Additionally, it does not exploit the spatial structure of images.

### 9.2.1 Convolutional Layer

The convolutional layer provides a solution to these problems, operating similarly to the convolutional filters discussed in the previous volume. A convolutional layer runs $N$ different convolutional filters over the input image (typically 1-3 channels), producing an $N$-channel filtered image. Subsequent convolutional layers operate on input images with $N$ channels. The size of the filters and the number of channels (also known as the depth of the layer) are typical hyperparameters. It is worth noting that in practice, we often pad the edges of the image with zeros to preserve the spatial dimensions.

Convolutional layers have two additional important parameters: stride and dilation. In the case of stride 1, the convolutional filter moves over every pixel, while with stride 2, it moves over every second pixel, and so on. This setting is typically used to reduce the spatial dimensions of the image. Dilation (Figure 9.2, bottom right) defines how many pixels away the next weight of the filter is multiplied with on the image. For a dilation of 1, the filter behaves as usual; with higher values, the filter "stretches" more and more. This setting is used to increase the receptive field of the convolutional filters.

### 9.2.2 Pooling

It is important to note that if we stack more and more convolutional layers with increasing depth, the size of the activation array at the output of the layers will eventually become enormous. Therefore, it is useful to reduce the spatial dimensions of the activation array every few layers. Besides using a convolutional layer with a stride greater than one, this can also be achieved with the so-called pooling operation. Pooling is another sliding-window operation that replaces the covered part of the activation array with a single value. The two most common cases are when this value is either the average or the maximum of the covered values.

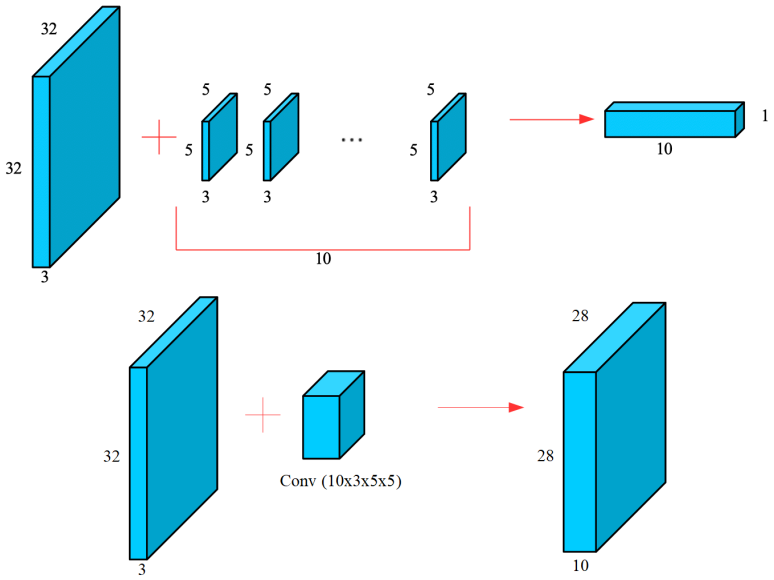


**Figure 9.1:** *Execution of convolution on an image array with multiple filters (top), and the equivalent convolutional layer (bottom).*

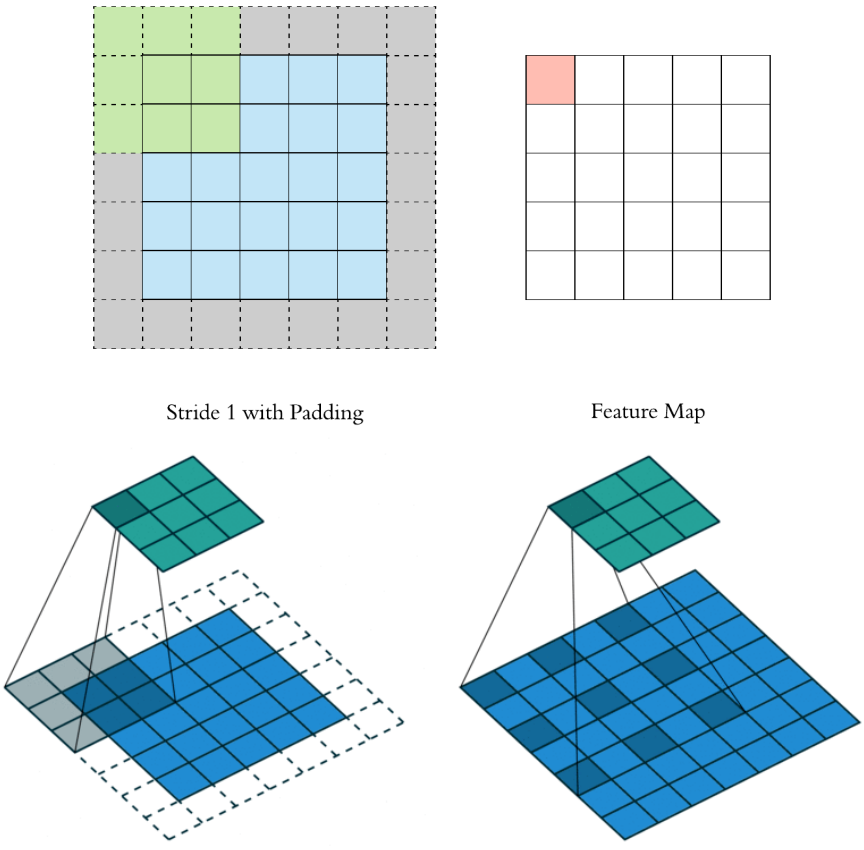


**Figure 9.2:** *The effects of padding (top), stride (bottom left), and dilation (bottom right) on the convolution operation.*

### 9.2.3 Activations

The final essential layer of multilayer neural networks is the activation layer, which typically follows every convolutional and linear layer. These two layers perform linear operations, and their

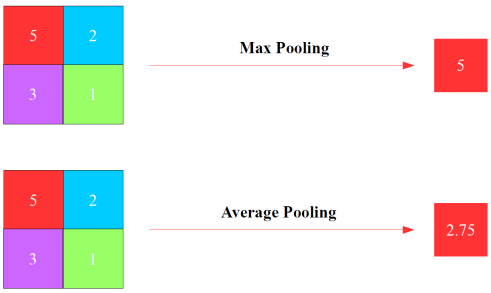


**Figure 9.3:** *Max pooling (top) and average pooling (bottom).*

composition remains linear. Therefore, nonlinear functions are inserted between the layers, which run independently on each element of the activation array. In traditional neural networks, the sigmoid and hyperbolic tangent functions were popular choices. However, these functions have the common drawback that over most of their domain, their shapes are flat, meaning their derivatives are virtually zero. If many such derivatives are placed into a neural network, according to the chain rule, they will eventually nullify most of the gradients. This leads to the early layers of the network becoming stuck and learning failing.

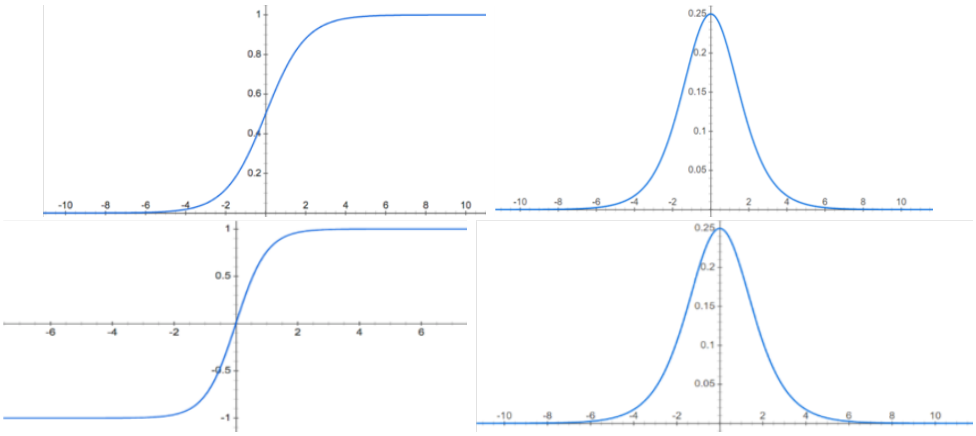

**Figure 9.4:** *Sigmoid function and its derivative (top), and tanh function and its derivative (bottom).*

Currently, the most popular activation function in neural networks is ReLU (Rectified Linear Unit), which is a piecewise linear activation that zeroes out negative inputs but has no effect in the positive domain. Its derivative is 1 in the positive domain and 0 in the negative domain, thus having a significantly smaller disruptive effect on the gradients. However, even with ReLU, early layers can still become stuck, and this can be mitigated by using Parametric ReLU (PReLU) or Leaky ReLU. These differ from the original solution by not zeroing out the activations in the negative domain but multiplying them by a constant less than 1. In the leaky case, this constant is a hyperparameter, while in the parametric case, it can be learned using gradient methods. There is also a smooth version of ReLU, which, unlike the traditional version, is differentiable at every point. This is called the Elastic Linear Unit (ELU).

## 9.3 Architectures

The neural networks containing the layers presented in this chapter are called convolutional neural networks, primarily used in the field of computer vision. The use of convolutional layers is particularly beneficial for image data since a convolutional layer has several orders of magnitude fewer parameters compared to a fully connected layer. In a convolutional neural network, convolutional layers typically follow one another in sequence (with an activation function at each output), with a down-sampling layer (either strided convolution or pooling) inserted after every few convolutional

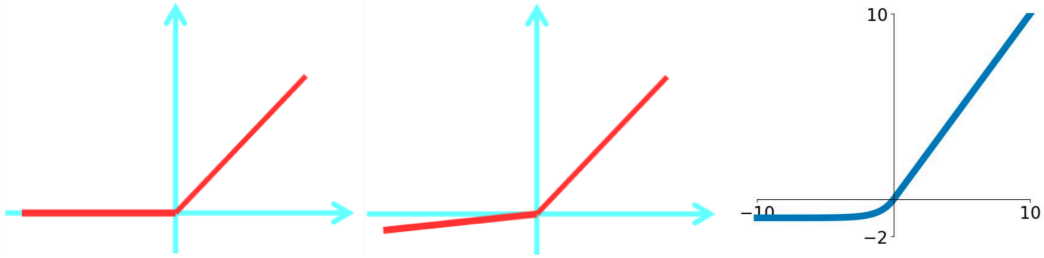


**Figure 9.5:** *ReLU (left), Leaky ReLU (middle), and ELU (right) activations.*

layers. At the end of the network, one or more fully connected layers typically generate the final output.

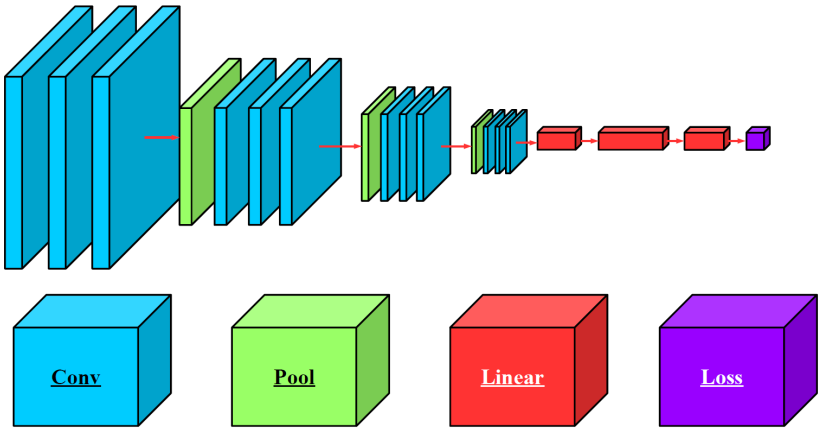


**Figure 9.6:** *A typical convolutional neural network.*

It is worth considering what a series of convolutional layers connected in sequence actually does. A convolution can be thought of as a simple feature detector that activates for certain combinations of its inputs and not for others. Thus, the activation map produced by the first convolutional layer indicates where on the image pixel combinations are present that activate the individual filters. However, the input to the next layer is this activation map, meaning that the filters in this layer activate for certain combinations of these lower-level features rather than individual pixels. Based on this, it is clear that a deep convolutional neural network contains layers in its later stages capable of detecting increasingly complex compositions of primitive image features. This concept closely resembles the compositional nature of human vision.

While the earliest successful convolutional networks implemented an architecture similar to this, there are numerous more advanced variations. These advancements are typically motivated by two primary goals: one, to improve the numerically problematic convergence properties of deep neural networks in some way, and two, to design structures capable of learning more complex relationships with as few free parameters as possible, thereby reducing the likelihood of overfitting. In this chapter, we introduce the motivation behind these architectures.

### 9.3.1 AlexNet

One of the earliest successful convolutional network architectures was AlexNet, which was the first to employ around 10 layers. The main reason for its success was the use of the ReLU activation function and normalization layers.

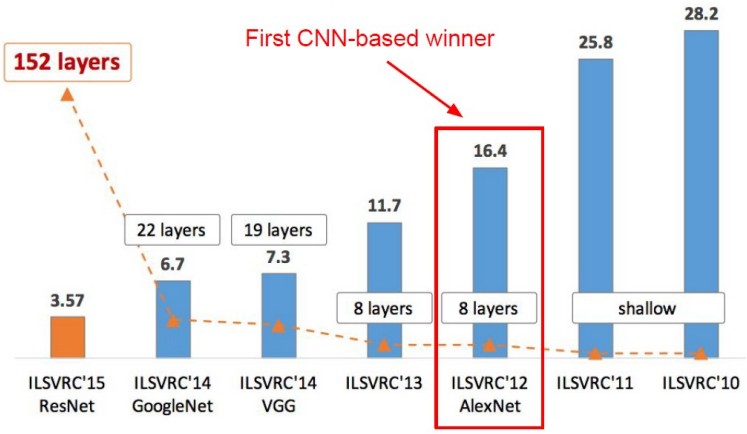


**Figure 9.7:** *The winning networks of the ImageNet classification competition and their number of layers.*

### 9.3.2 VGG

The VGG architecture is one of the most popular traditional neural network models, with variants consisting of 16 and 19 layers. It has a significantly more regular architecture than AlexNet, using only convolutional layers with a kernel size of 3x3. The motivation behind this is that three consecutive 3x3 layers have the same effective receptive field as a single 7x7 layer but with two additional nonlinearities and half the number of parameters. As a result, more complex functions can be learned, while the smaller number of parameters reduces the likelihood of overfitting. However, it is worth noting that this does not mean the VGG architecture is particularly good at avoiding overfitting, as the huge number of parameters in the final 3 fully connected layers neutralizes this advantage.

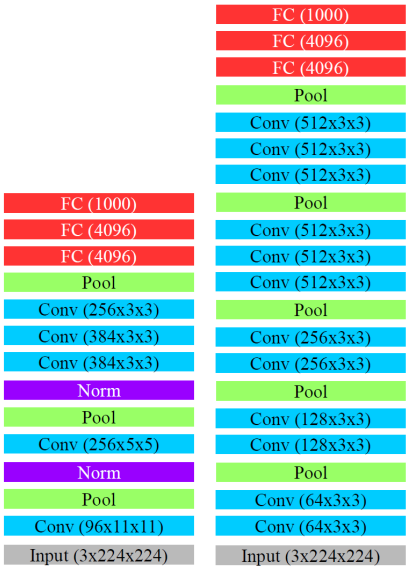


**Figure 9.8:** *The AlexNet (left) and VGG (right) models.*

The following diagram shows the memory size required by each layer of the VGG model and the number of parameters in each layer. It is easy to notice that the beginning of the network is the most memory-intensive part, while the last layers are much more prominent in terms of the number of parameters. It is worth mentioning that compared to modern networks, VGG is extremely wasteful in terms of both parameter count and memory usage.

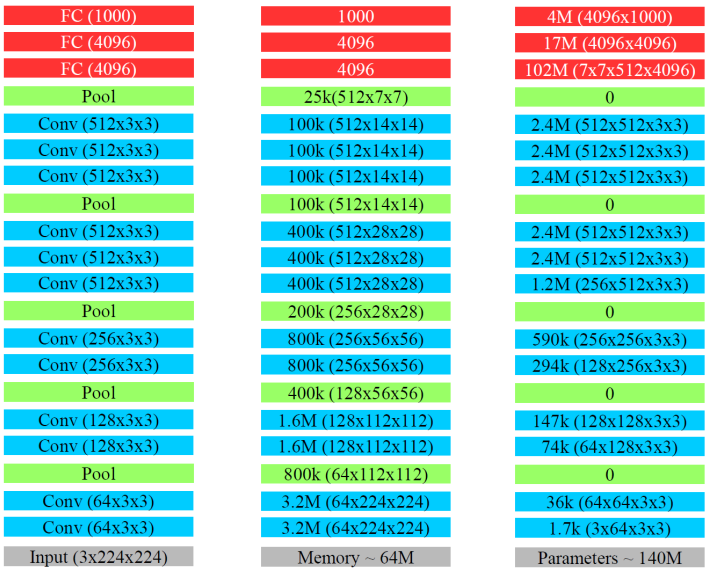


**Figure 9.9:** *The number of parameters and memory size used by the VGG network.*

### 9.3.3 Inception

A good example of parameter reduction is the layer type called Inception. The essence of this solution is that convolutional layers do not exist only in sequence but also in parallel. These parallel layers usually perform convolutions of different sizes, resulting in a network structure that can better handle variations in object scales.

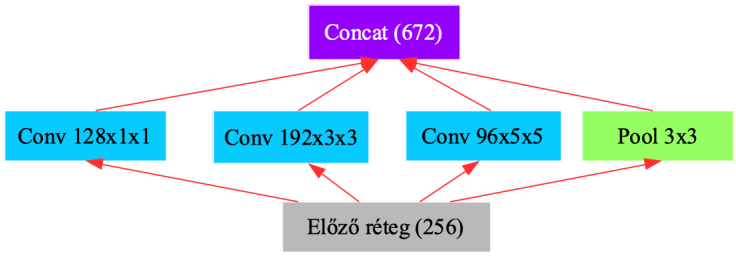


**Figure 9.10:** *The naive Inception layer.*

The GoogLeNet network is built from numerous Inception blocks. Additionally, the network has multiple auxiliary classifiers branching off from lower levels, intended to aid convergence by introducing gradients at lower levels.

### 9.3.4 ResNet

A good example of convergence improvement is the so-called residual block. This layer adds the input to the output after performing the convolution, so the layer essentially only needs to produce the difference between the expected input and output. The benefit of this solution is that in a neural network composed of such blocks, the error derivative has a path back to the front layers of the network through these additions where the derivative remains one. As a result, the numerical problems arising from the numerous multiplications required during backpropagation are addressed. With the introduction of the residual block, the maximum depth of convolutional networks increased from around 30 layers to the range of 100-200 layers.

It is also worth mentioning the so-called bottleneck layers, used in both Inception and residual blocks. Their motivation is that performing two-dimensional convolutions is extremely costly

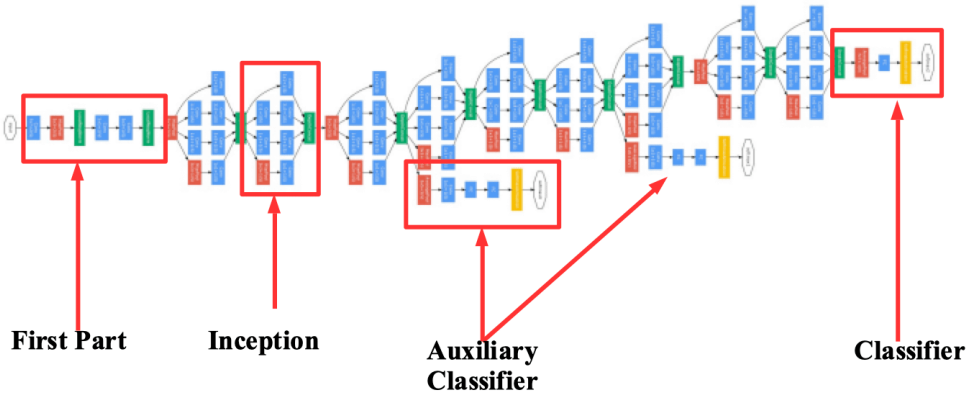


**Figure 9.11:** *The GoogLeNet model.*

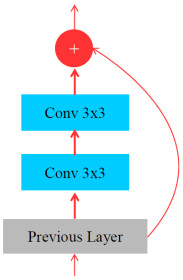


**Figure 9.12:** *The residual block.*

when the number of channels in the activation maps is large. Therefore, in these networks, every two-dimensional convolution is preceded by a 1x1 convolutional layer, which reduces or compresses the number of channels in the input activation map to a fraction of the original. Then, the two-dimensional convolution (3x3 or 5x5) is performed on this compressed map, followed by another 1x1 convolutional layer that restores the original number of channels. This method significantly reduces both the number of parameters in individual layers and the time required to execute the layer.

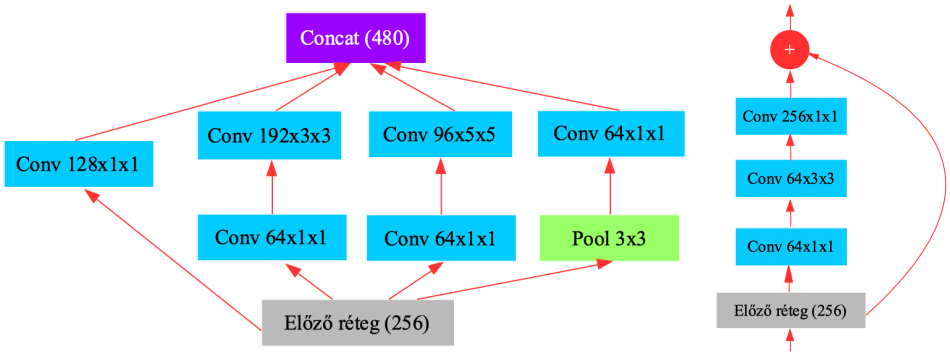


**Figure 9.13:** *Inception (left) and residual (right) blocks using the bottleneck solution.*

### 9.3.5 DenseNet

The DenseNet architecture is an evolved version of ResNet, where within a dense block, there are skip connections not only between consecutive layers but between every pair of layers within a block (hence the name ”dense”). This innovation made it possible to achieve results equivalent to the ResNet model with approximately half the computations and parameters.

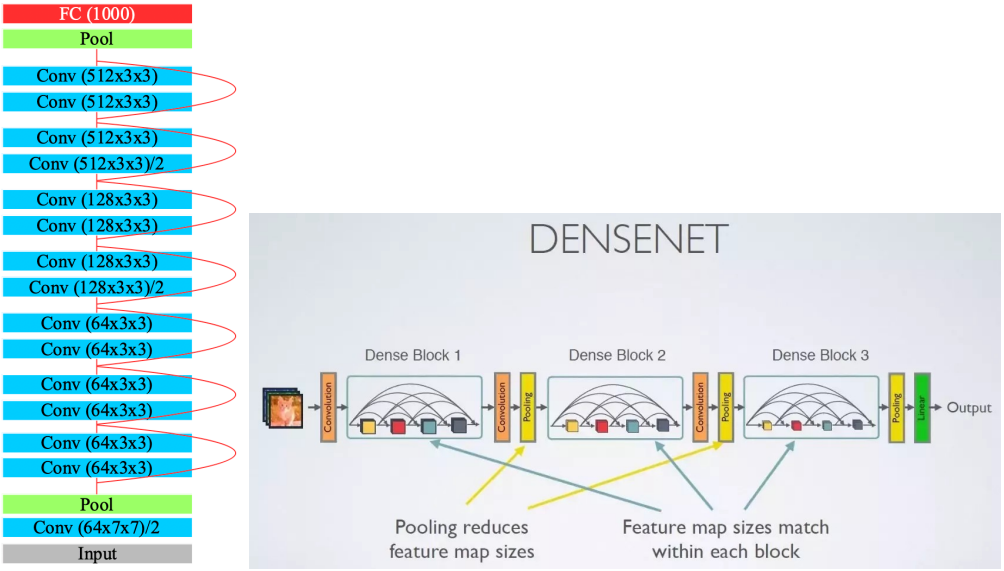


**Figure 9.14:** *The ResNet (left) and DenseNet (right) models.*

## 9.4 Visualization

In the previous and following sections, we discuss in detail the construction, training, validation, and deployment of various deep neural networks. However, one extremely important topic hasn't been covered: the problem of debugging. This is particularly critical in the case of neural networks, as trial and error can be extremely costly, with training a large network taking days or even weeks. The problem is that neural networks act like a kind of black box, meaning we don't fully understand what they are doing and why. This makes it necessary to visualize the inner workings of neural networks to gain some insight into their internal states.

It's worth noting that the backpropagation operation used in gradient computation can generally be applied to calculate the derivative between any two points in the network. This insight can be exploited in various ways. For example, it allows us to determine which pixels influence a class score in a given image. This can be used for object segmentation, tracking, and even to identify which part of an image is responsible for an incorrect classification.

To achieve this, all we need to do is prescribe a value for the output gradient of the selected class's score while setting the gradients of all other classes to zero. We then perform the backpropagation operation, but instead of computing the derivatives with respect to the weights, we compute them with respect to the input image. By taking the absolute value of this and then taking the maximum along the channel dimension, we can generate a grayscale image where the intensity of the pixels is proportional to their impact on the class score. This resulting image is called a *saliency* map.

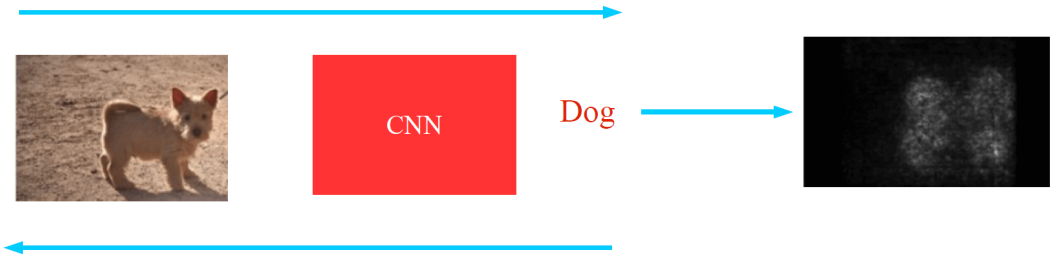


**Figure 9.15:** *Generating a saliency map.*

### 9.4.1 Guided backpropagation

However, if we want to use this method for segmentation or visualization, we encounter a minor issue: the saliency will also be high in regions of the image that negatively influence the output (i.e., "where there is no dog," for example). To address this, we need to suppress the influence of features that act against the class of interest during backpropagation. This can be done easily, as it can be inferred that if a particular feature reduces the class score, then the associated derivative will be negative. From this point on, all we need to do is set these gradients to zero at every layer during backpropagation. The simplest way to achieve this is to modify the ReLU backward step so that the ReLU function is applied to the gradients as well. This operation is called *guided backpropagation*.

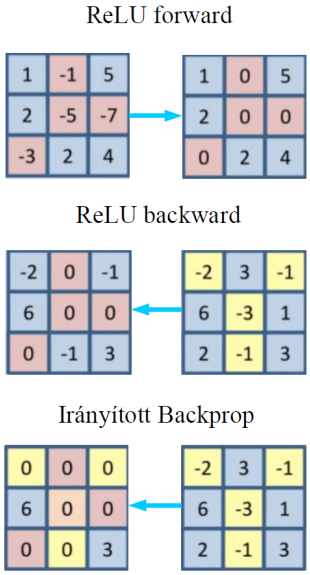


**Figure 9.16:** *Applying guided backpropagation to the ReLU nonlinearity.*

In addition to generating high-quality saliency maps, guided backpropagation can be used to visualize individual neurons. In this case, our goal is to generate the image that maximally activates a given neuron (i.e., convolutional filter). To do this, we initialize the input image with zeros, and then forward it through the network up to the layer containing the selected neuron. We then prescribe the gradients of this layer's output in the manner described earlier and backpropagate all the way to the input image. Afterward, we add the resulting gradients to the image's pixels, repeating these two steps until the image changes significantly.

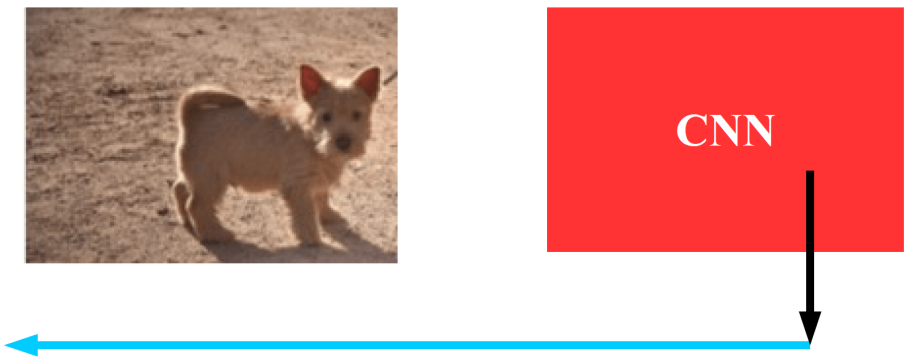


**Figure 9.17:** *The concept of guided backpropagation.*

It's important to note that several refinements need to be made to the resulting image; otherwise, it will become incomprehensible, and the pixel values will continuously increase, causing the algorithm to never stop. Therefore, it's necessary to apply some kind of regularization (usually L2

regularization) to the image, as well as use smoothing filters to reduce noise. It is also advisable to manually zero out overly small pixel and gradient values obtained during backpropagation.

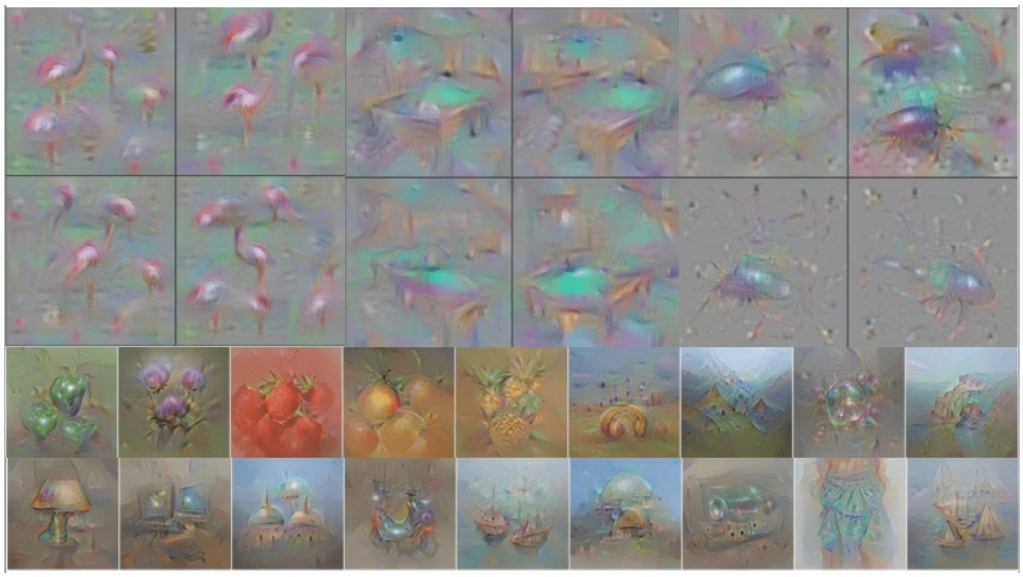

**Figure 9.18:** *The result of guided backpropagation.*

### 9.4.2 Adversarial examples

One significant discovery made during the application of the backpropagation algorithm is that, in typical neural networks used for computer vision, it is possible to generate minimal changes to correctly classified images that are imperceptible to the human eye, yet cause the neural network to misclassify the image. These images are known as *adversarial examples*, and based on our current knowledge, it is quite difficult to defend against them. The best strategy we have is to generate such examples during training and use them to train the network. However, humans are not immune to such errors either—adversarial examples in human vision are known as illusions. It seems, though, that the illusions of neural networks differ significantly from those of humans.

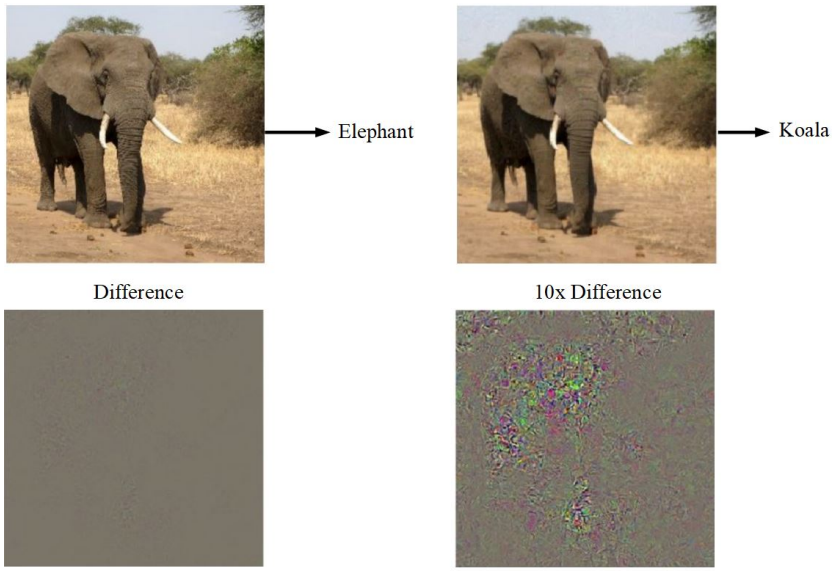


**Figure 9.19:** *Adversarial examples.*

### 9.4.3 DeepDream

There is another interesting application of guided backpropagation, known as *DeepDream*. This method differs from the visualization technique in that the input image is not empty but rather an arbitrary real image. We forward this image through the network to a selected layer, but instead of prescribing the layer's output gradient with a one-hot vector, we simply set it equal to the layer's activations. This essentially creates positive feedback between the image and the activations of the selected layer. The result is a highly creative, dream-like image. Although the practical utility of this method is minimal, it has demonstrated that convolutional neural networks may possess associative capabilities similar to those of humans.

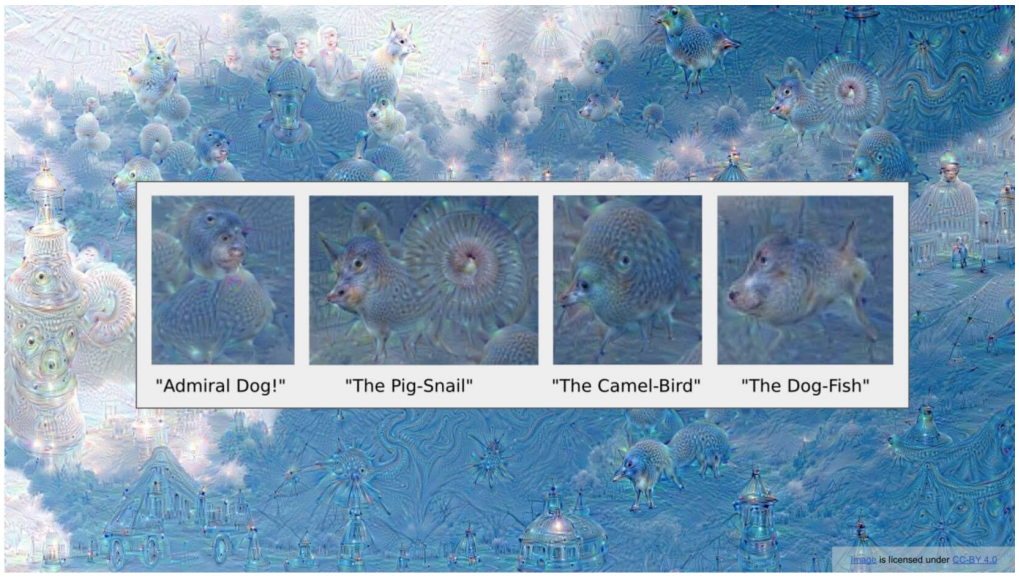


**Figure 9.20:** *The result of DeepDream.*

# 10 Deep Learning in Practice

## 10.1 Introduction

In previous lectures, we covered the basics of deep learning and convolutional neural networks. In the following chapters, we will delve into specialized structures designed for processing sequences and handling more complex visual tasks beyond classification. However, deep learning is one of those areas where methods may seem simple on paper, but in practice, many difficulties arise, making their application challenging. The goal of this lecture is to gather practical considerations and techniques that are essential for building well-functioning neural networks in real-life applications.

The training of neural networks is typically complicated by four main factors:

1. Numerical issues that hinder the convergence of the optimization algorithm
2. The phenomenon of overfitting, which jeopardizes the applicability of the trained model in real-world situations
3. The challenge of generating the large amount of labeled data necessary for training neural networks
4. Determining the hyperparameters that lead to successful training and good generalization

## 10.2 Convergence Issues

The gradient method and backpropagation algorithm described in earlier chapters work perfectly in theory and textbooks (including these notes). However, in practice, due to the finite precision and range of floating-point representation, we encounter many problems that can easily prevent the algorithm from converging. Generally speaking, floating-point arithmetic tends to work more reliably when our numbers are "well-behaved," meaning they are centered around zero and have a standard deviation close to one.

The reason for this is that backpropagation involves a series of matrix multiplications. It is easy to see that when we multiply many numbers, if these numbers are smaller than one, the result will eventually approach zero. Conversely, if they are larger than one, the result will tend toward infinity. As a result, the gradients (especially in the network's early layers, where the matrix product is quite long) will either vanish or explode. This is known as the problem of vanishing or exploding gradients. The product will remain within a reasonable range only if the values of our numbers are relatively close to one.

This principle holds true for matrix multiplication as well, except in this case, the norms of the matrices need to be reasonably small. One of the most widely used matrix norms is the Frobenius norm, which is simply the sum of the squares of the matrix elements. If the average of the matrix elements is zero, this norm equals the variance of the elements.

### 10.2.1 Initialization

In the first chapter on deep learning, we introduced the gradient method, which iteratively adjusts the network parameters using the derivative of the error function to improve performance. However,

we deliberately omitted the topic of how to obtain the initial parameter values. The truth is that we know nothing about the correct parameters at the beginning, so our only option is to initialize them randomly. The standard deviation of these random numbers, however, is crucial (it is evident that the mean should be zero). If the random weights are too large, most of the activation values will become increasingly large, leading to excessively large gradients. As a result, instead of gradually approaching the minimum of the error function, we will take giant leaps in the parameter space, likely overshooting the minimum entirely. If the weights are too small, the activations of the network will be close to zero after just a few layers, which will cause the gradients to vanish, and the network will get stuck at the initial values.

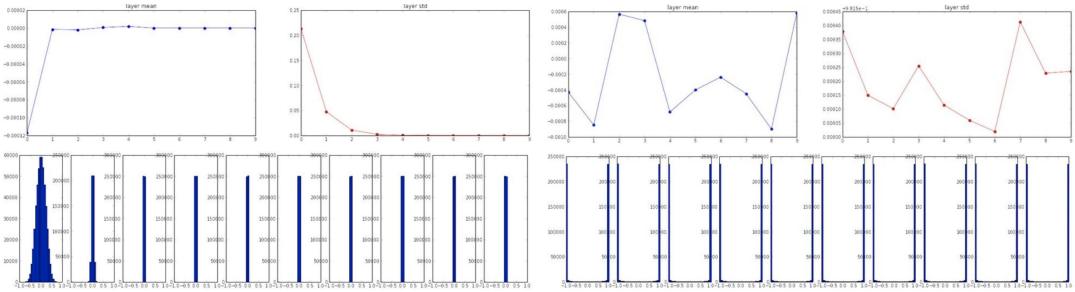

**Figure 10.1:** *The distribution of activations in the network with small random (left) and large random (right) number initialization.*

To avoid this, we must carefully choose the variance of the network's weights, ensuring they are neither too large nor too small. There are several methods to achieve this, with the most common choices being the Xavier and He initialization formulas. These determine the variance of the initial weights for each layer of the network as follows:

$$\begin{aligned} \mu &= 0 \\ \sigma_{Xav} &= \frac{2}{n_i + n_o} \\ \sigma_{He} &= \frac{2}{n_i} \end{aligned} \tag{10.1}$$

Where $n_i$ and $n_o$ represent the number of inputs and outputs of the respective layer. It's worth noting that with these choices, the activations and gradients of the network will approximately follow a normal distribution, although the use of the ReLU activation function distorts this to some extent.

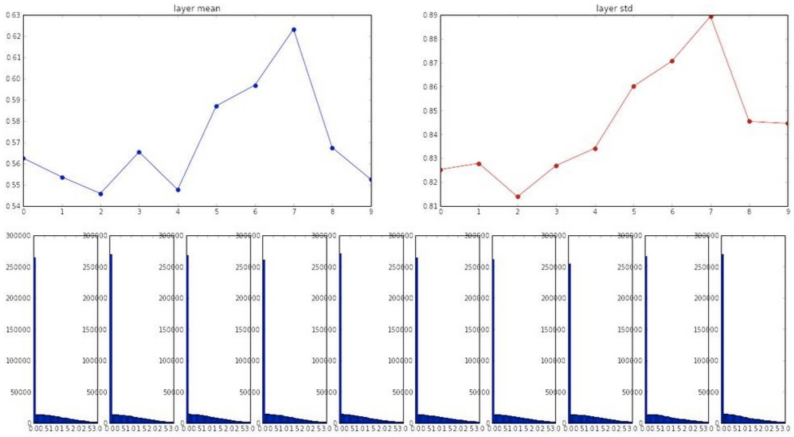


**Figure 10.2:** *The distribution of activations using Xavier initialization.*

### 10.2.2 Data Normalization

For similar reasons, certain transformations must be performed on the images used as input for neural networks. As mentioned earlier, it is crucial for the pixel values of the input image to have a distribution that is approximately standard normal in order to ensure good numerical convergence. Even if the network's weights are well-initialized, if the input values are too large, we will encounter the same problems as with large weights. Additionally, it is important that the pixel values have an average close to zero, because an input that is entirely positive or entirely negative will restrict the possible directions of the gradient.

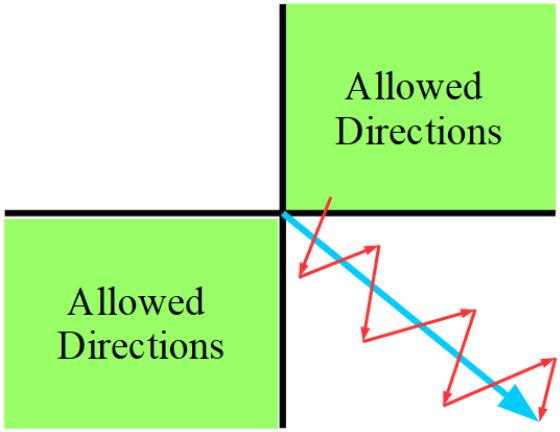


**Figure 10.3:** *With entirely positive input, the gradient will also be entirely positive or entirely negative, thus only allowing movement toward the desired direction in a zigzag pattern.*

## 10.3 Validation and Regularization

As mentioned in the fundamentals of machine learning, during training, the phenomenon of overfitting may occur. To detect this, it is recommended to use two separate datasets: a training set and a validation set. These can be randomly selected parts of the same database (their ratio typically ranges between 80%-20%). The training set is used to perform the training, while the validation set is used to monitor the overfitting phenomenon, which can guide the tuning of hyperparameters. It is crucial to emphasize that the validation dataset **MUST NEVER** be used for training.

However, in practice, two datasets are not enough. The training dataset is used to determine the network's parameters, while the validation dataset is used to tune the network's hyperparameters. This way, we cannot predict how the network will perform on entirely new data. To address this, a third test dataset, comprising approximately 10% of the total data, is recommended. For the method to be reliable, it is essential to completely separate these three datasets and use each for its intended purpose.


**Figure 10.4:** *The distribution of the datasets.*

It is worth noting that the phenomenon of neural network overfitting can also be described by the distribution of activations within the layers. In cases of overfitting, the network starts to learn the correct answer for each individual training sample, rather than general patterns between inputs and outputs. This typically results in only a very few activations being maximized for each training sample in the layers (those that remember the specific training sample), while the other activations take the opposite extreme values. As we have previously discussed, such "extreme"

activations can easily occur when the weights of the individual layers become too large. If we recall the regularization methods discussed earlier, they specifically aim to slow down the growth of weights.

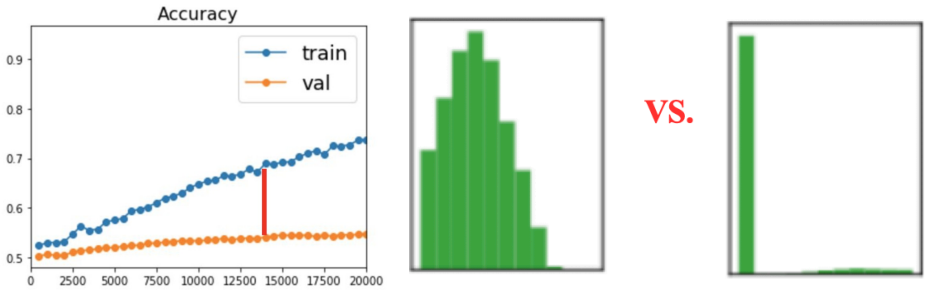


**Figure 10.5:** *The training and validation curves associated with overfitting (left), and the distribution of activations in normal (center) and overfitting (right) cases.*

### 10.3.1 Dropout

There are two other commonly used methods to prevent unwanted activations. The first is a technique called dropout, where during training, a certain fraction of activations in each layer are randomly zeroed out during each forward pass, and the remaining activations are calculated accordingly (as shown in Figure 10.6). It is easy to see that this method significantly reduces overfitting since it forces the network to introduce redundancy. It is important to note that during testing, the random zeroing is no longer performed, but the activations must be scaled according to the dropout probability.

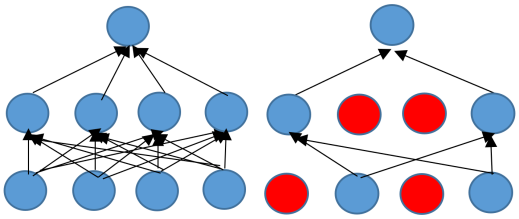

**Figure 10.6:** *Application of dropout.*

### 10.3.2 Batch Normalization

The other solution is called batch normalization, which involves performing a normalization operation after each layer. We previously mentioned that during training, we evaluate not just one image but a batch of images (typically a multiple of 32). The essence of batch normalization is that we continuously compute the mean and variance of the activations during training and normalize the activations accordingly. It is worth noting that this operation not only mitigates overfitting by normalizing the distribution of activations but also improves numerical convergence. The formula for batch normalization is as follows:

$$\begin{aligned} x_{BN} &= \frac{x-\mu}{\sigma^2+\epsilon} \quad \leftarrow \quad vanilla \\ x_{AF} &= \alpha x_{BN} + \beta \quad \leftarrow \quad affine \\ x_{DC} &= \Sigma^{-\frac{1}{2}}(x-\mu) \quad \leftarrow \quad decorrelated \end{aligned} \tag{10.2}$$

Where $\mu$ and $\sigma/\Sigma$ are the estimated mean and variance/covariance matrix of the inputs $x$, while $\alpha$ and $\beta$ are learned parameters. It is worth noting that in modern neural networks, batch normalization is a fundamental operation, so typically, every convolutional layer is followed by a batch

normalization layer. Batch normalization and dropout can even be used together, though most experiments show only minimal performance improvements. The advantages of batch normalization are summarized in the following three points:

- Normalizing the distribution of activations reduces the degree of overfitting.
- Due to normalization, the distributions and gradients of the layers remain approximately in the same range, which aids convergence.
- Since gradient descent changes the weight matrix of each layer simultaneously, the input distribution to all layers except the first changes with each step, which requires re-learning. Batch normalization counteracts this, making convergence significantly more stable (and consequently faster).

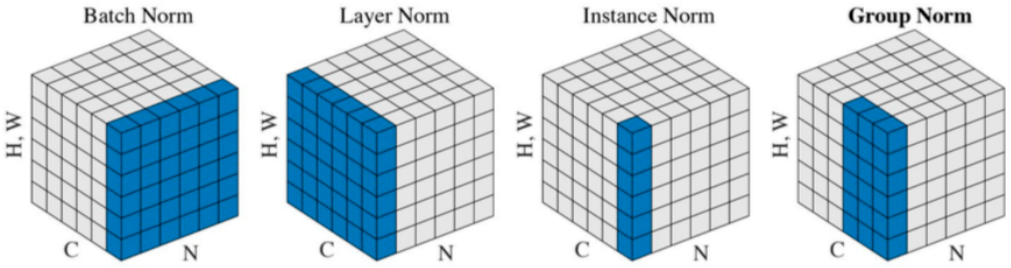


**Figure 10.7:** *It is worth noting that besides batch normalization, there is also layer normalization, where the entire activation block is normalized per instance. In instance normalization, each channel is normalized individually across the spatial dimensions. A compromise between the two is group normalization, where a few channels are normalized together.*

### 10.3.3 Data Augmentation

There is yet another possible way to avoid overfitting: consider that in the case of overfitting, the neural network learns the correct answer for each training example individually. It is evident that the more training data available, the harder it becomes to do this. Therefore, increasing the number of training data almost always helps reduce the extent of overfitting. However, generating training data is extremely costly, so this strategy may not always be feasible on its own. In the case of images, however, we have the option to automatically generate (partially) new training data through data augmentation. The essence of the method is that by flipping, randomly cropping, rotating, scaling, and applying intensity transformations, we artificially increase the size of the dataset. It is easy to see that these operations do not affect the labels of the images, so they can be performed without penalty. It is important to note that batch normalization, regularization, and data augmentation are used together.

## 10.4 Hyperparameter Optimization

Another difficulty in training neural networks is the presence of multiple hyperparameters that may need tuning. There is no better method than trial and error to choose their optimal values. Training a neural network, however, can take a considerable amount of time (from a few hours to several weeks), making each attempt quite costly. Therefore, at the beginning of the training, it is often possible to select approximately correct values for many hyperparameters by training on only a small portion of the entire dataset. This method can also help identify potential program bugs.

During training on the full dataset, usually only a few hyperparameters need to be set within a relatively narrow range. It may be useful to divide these ranges into a uniform grid and perform

separate trainings at each grid point, then compare them. However, a more effective approach is to use a similar number of random hyperparameter combinations for training. This way, the impact of each hyperparameter can be measured with higher resolution. This is particularly useful when one of the hyperparameters significantly affects the quality of training more than the others.

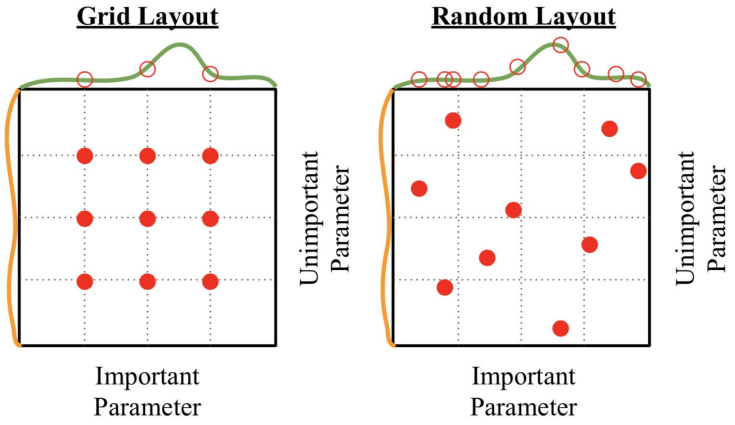


**Figure 10.8:** *Two possible schemes for hyperparameter optimization.*

### 10.4.1 Learning Rate

Among the hyperparameters for training, the learning rate, which determines the size of the steps in gradient methods, deserves special attention. In gradient methods, we aim to reach the deepest point of the "valley" created by the error function by taking a series of steps in the steepest direction of decrease. If the step size is too small, it will take many steps to enter the valley. However, if the step size is too large, we may quickly reach close to the deepest point but overshoot it with the final step, causing the algorithm to "bounce" between the two sides of the valley indefinitely. In extreme cases, a large step size may cause the weights to jump so far that they end up at a higher point on the opposite side of the valley than where we started. Repeating this process will result in climbing out of the valley rather than minimizing it.

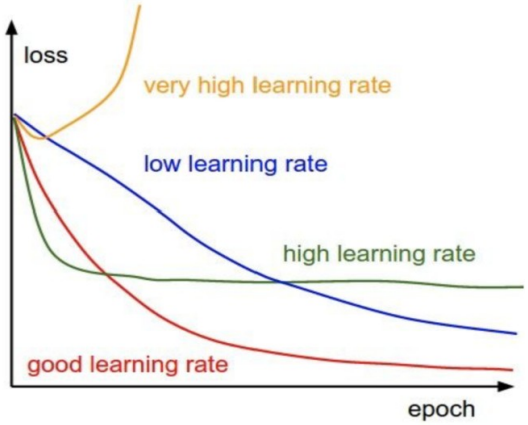


**Figure 10.9:** *The effects of different learning rate choices.*

In practice, due to these considerations, it is not common to use a single learning rate. Instead, optimization typically starts with the largest learning rate that still shows stable decrease in the error value, allowing us to get close to the optimum as quickly as possible. After some time, the learning rate is reduced to enable closer approximation to the true optimum position. This can be viewed as a form of coarse optimization followed by fine-tuning. There are many possible methods for adjusting the learning rate, one of the most common being to decrease the rate by a fixed factor after a certain number of steps.

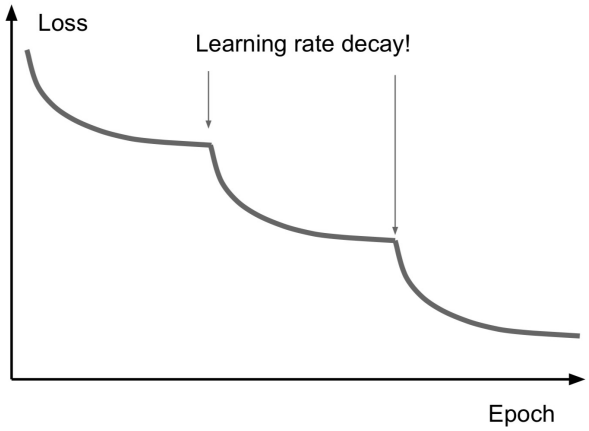


**Figure 10.10:** *Adaptive changes in the learning rate.*

It is also possible to adjust the learning rate adaptively by continuously monitoring the learning speed. In this case, the rate is reduced by a fixed factor if the learning has not exceeded the previous best result for a certain number of steps. Another effective method is to use cosine annealing, which adjusts the learning rate according to the first half-period of a cosine function between a maximum and minimum value. When using cosine annealing, training can continue with the maximum learning rate after the half-period is complete, allowing for additional annealing. The idea is that the suddenly increased learning rate ”pushes” the network weights out of the local minimum, enabling the discovery of a better nearby local minimum during further training.

## 10.5 Database Preparation

The third challenge in training neural networks is the generation of a large number of labeled training data. This problem can be partially mitigated by the data augmentation methods previously discussed. Another notable method is so-called semi-supervised learning, where only a small subset of the database is labeled, and the network is expected to assign similar labels to similar data.

### 10.5.1 Transfer Learning

One of the most significant breakthroughs in deep learning addresses this problem. It is understood that the initial layers of convolutional neural networks detect various image features. As we progress through the layers of the network, these features become increasingly complex and task-specific. Therefore, if we already have a network trained for a specific task, its initial layers can be used for another similar task. Thus, only the final layers of the network need to be retrained for the new task, requiring significantly less data as they contain fewer free parameters compared to the entire network.

This technique is known as transfer learning and is a widespread solution in the field of deep learning. The reasoning above is so accurate that, in many cases, it is sufficient to retrain only the last linear layer of the networks. In other cases, fine-tuning of the initial layers may be necessary, but this also requires orders of magnitude fewer data.

## 10.6 Deployment

The final topic of this chapter is the question of preparing neural networks for practical use (deployment). Deploying neural networks involves two problems: neural networks have millions of parameters, meaning that a deep network can be quite large. Moreover, their execution requires

billions of operations, making them quite slow. This greatly complicates their use in devices with lower performance.

### 10.6.1 Pruning

One of the most important solutions to the above problems is the operation of pruning, where a small percentage of the least important weights are selected and removed from the network. There are several methods for ranking the weights, the most obvious being to use the absolute values of the weights. After this, the network is fine-tuned with the deleted weights set to zero, and this process is repeated several times. Research supports that with iterative pruning techniques, up to 90% of the network weights can be removed with only a few percentage points of performance loss. This results in a tenfold speedup.

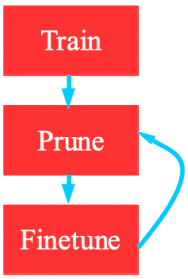


**Figure 10.11:** *The Pruning method.*

### 10.6.2 Weight Sharing

Another similarly effective method is weight quantization, where weights are grouped into a few clusters using a clustering algorithm, and each weight is replaced with the mean value of its cluster. After this, the values of the cluster centroids are fine-tuned, and these operations are also performed iteratively, similar to pruning. An average neural network's weights can be divided into 16 groups with only a few percentage points of performance loss using this method. Since there are only 16 different types of weights in the network, each weight can be represented using just 4 bits, resulting in an eightfold compression compared to the usual 32-bit floating-point representation.

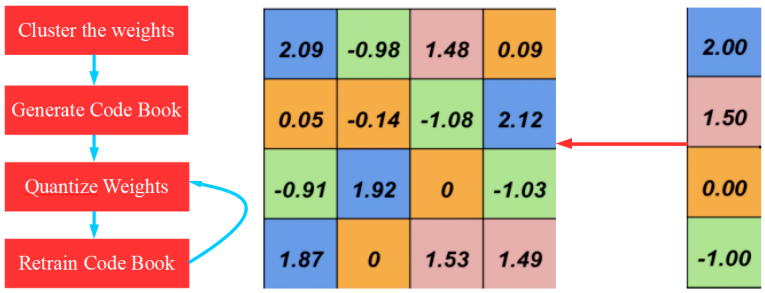


**Figure 10.12:** *The Weight Sharing method (left) and the schema of weight quantization (right).*

### 10.6.3 Ensemble

An interesting approach is the use of ensemble methods. In this case, we generally create a "consensus" estimate by averaging the outputs of several differently structured networks that are initialized and trained in different ways. Based on experience, this method can improve the output of the best network by approximately 2-3%. Such methods are often referred to as expert systems.

# 11 Detection and Segmentation

## 11.1 Introduction

In the introductory lecture on the subject, we listed several important tasks in computer vision, but so far we have only discussed the implementation of classification. In the current lecture, we will examine various types of object detection and segmentation.

## 11.2 Semantic Segmentation

The task closest to classification is semantic segmentation, where we aim to classify every pixel in the image. This can indeed be done using a classifier neural network with a sliding window approach, but performing this for all pixels in an average image, which could be in the order of hundreds of thousands or millions, would take an extremely long time.


**Figure 11.1:** *The task of semantic segmentation.*

### 11.2.1 Fully Convolutional Architecture

Therefore, it is beneficial to parallelize the process so that it can be performed with a single neural network. Fully Convolutional Networks (FCNs) are suitable for this, consisting of a sequence of convolutional and activation layers. The last layer of the network is also convolutional, with its output channels corresponding to the number of classes, so the elements of the output activation map can be considered as classifications for each pixel. The problem with this architecture is that performing convolutions at the full original resolution of the image is expensive, so it is advisable to incorporate down-sampling operations. However, this results in a lower resolution classification, which is undesirable.

In practice, FCN networks consist of an up-sampling and down-sampling part, which are more or less mirror images of each other. This way, we obtain an output of the same size as the original image, but most of the processing is done at a lower resolution, making the runtime acceptable. It is also worth noting that FCN networks typically include skip connections between the up-sampling and down-sampling parts of the same resolution, which aids in the flow of gradients, thus improving the convergence of training. Another advantage of these connections is that low-level features detected early in the network help in more accurately defining the boundaries of classes, which are lost during down-sampling.

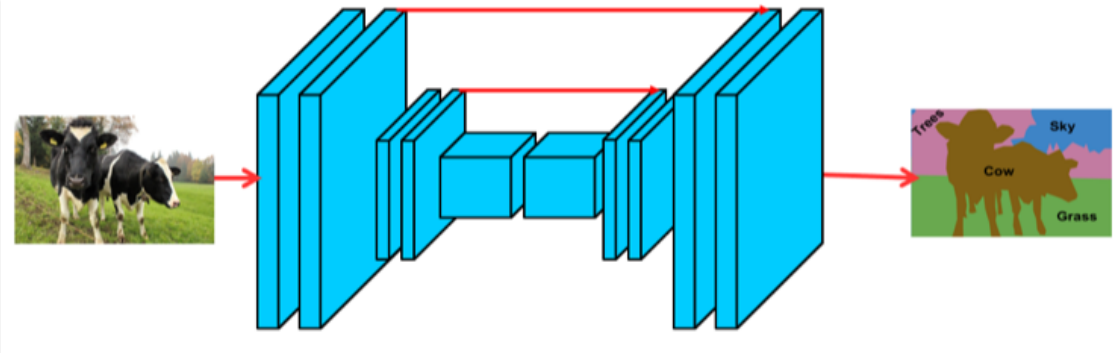


**Figure 11.2:** *A typical FCN architecture.*

The performance of FCN networks can be improved with several additional methods. One of the simplest methods is the use of residual or dense blocks in the down-sampling part of the network. This approach allows us to use deep networks with a large effective receptive field for segmentation without making convergence difficult.

Another option is to use dilated (sometimes referred to as atrous) convolutional filters. The effect of dilation is to increase the effective receptive field of the filter while keeping the number of parameters the same. This allows the network to examine a larger context, improving the consistency of segmentation. Similar improvements can also be achieved with Spatial Pyramid Pooling (SPP). This method performs several pooling operations of different sizes in parallel on the activation map produced at the end of the network and concatenates the resulting activations, performing classification based on these features. The advantage of this solution is that by using activations from multiple scale factors, we reduce the network's sensitivity to scale. The pyramid pooling operation is also often performed in a dilated manner for similar reasons.

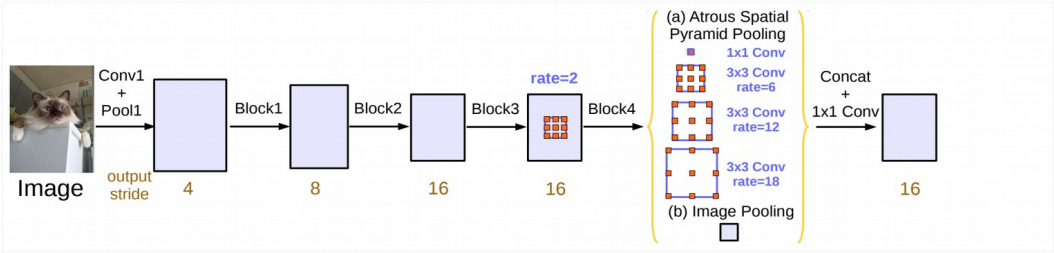


Figure 5. Parallel modules with atrous convolution (ASPP), augmented with image-level features.

**Figure 11.3:** *An architecture using dilated pyramid pooling.*

### 11.2.2 Upsampling Methods

The remaining question is how to implement upsampling in convolutional neural networks. One of the simplest ideas for this is the so-called unpooling operation, where we store the position of the maximum value during down-sampling using maximum pooling, and during up-sampling, we write the lower-level value into this position while keeping the other positions at zero.

Another common solution is transposed convolution, which is essentially the reversal of convolution with stride. The method's name comes from the fact that convolution can be described as a multiplication with a matrix, and transposed convolution is multiplication with the transpose of this matrix. The advantage of this method is that the upsampling is learnable, which significantly improves the quality of segmentation.

The name transposed convolution comes from the fact that convolution can be described as matrix multiplication as follows:

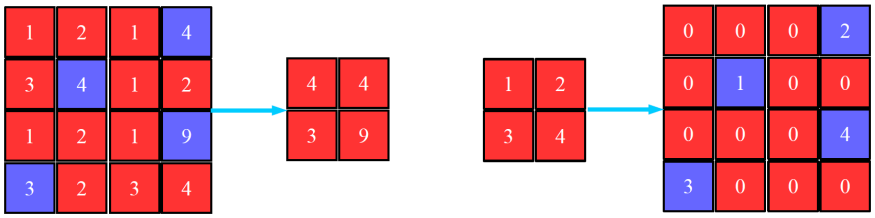


**Figure 11.4:** *The max unpooling operation.*

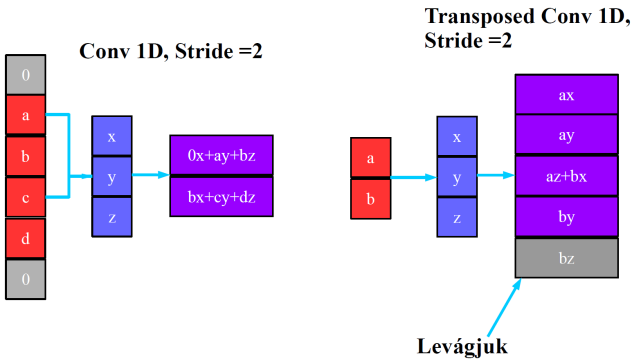


**Figure 11.5:** *Transposed convolution in 1D.*

$$\begin{pmatrix} x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \end{pmatrix} \begin{pmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{pmatrix} = \begin{pmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + by \end{pmatrix} = Xa \tag{11.1}$$

The transposed convolution corresponds to multiplication with the transpose of the previous matrix. It is worth noting that in the field of deep learning, transposed convolution is often incorrectly referred to as deconvolution. This mistake is equivalent to mixing up the transpose and inverse of a matrix.

$$\begin{pmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{pmatrix} = X^T a \tag{11.2}$$

There is also a third common method, which is dense upsampling convolution. The essence of this method is to use a convolutional layer to increase the number of channels in the given activation map by a factor of four, then rearrange the resulting array so that the spatial dimensions of the activation map are twice those of the original, while the number of channels matches the original. This method also allows for learnable upsampling but comes with more parameters, thus learning more complex transformations at the cost of slower operation.

## 11.3 Detection

After finishing the discussion on semantic segmentation, we turn to another important topic: detection. In this case, the extracted features (usually bounding rectangles) are somewhat simpler, but separating individual objects is still achievable.

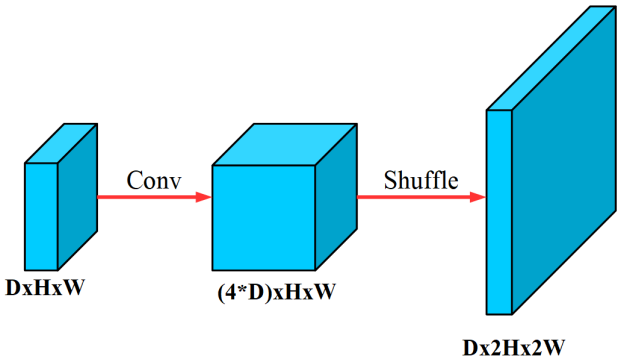


**Figure 11.6:** *Dense upsampling convolution (DUC).*

### 11.3.1 Localization

One of the simplest variations of this is localization, where we enhance the classification task by determining the bounding rectangle of the object of a given class. This task can also be easily performed with neural networks, as it involves simply adding four additional outputs to a classifier network. These outputs are required to accurately provide the four parameters of the bounding rectangle. Since this is a regression problem, the accuracy of the rectangle parameters is ideally measured using a squared error cost function. The total error of the localization network will be the sum of the classification and regression error functions.

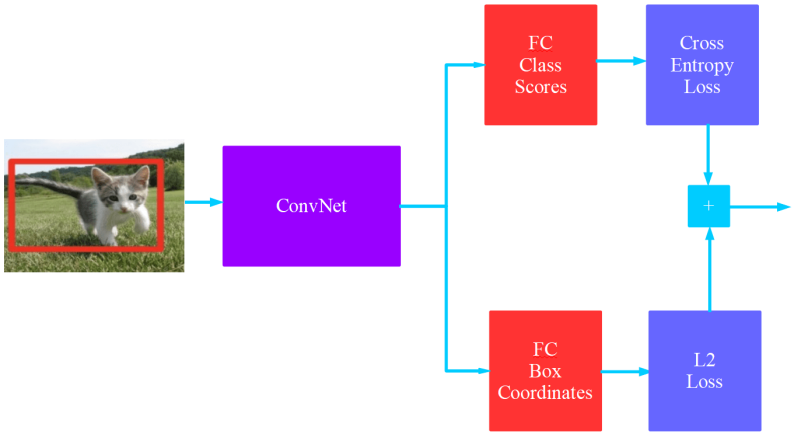


**Figure 11.7:** *Architecture implementing localization.*

### 11.3.2 Region-CNN

In the case of detection tasks, however, we have a significantly more challenging problem. This is because objects of any number and class can occur, and the architecture must be designed accordingly. Of course, we can provide a rough upper estimate of the number of objects, so we could create a convolutional network with exactly that many different classifiers and bounding box estimators. However, for $N$ maximum objects and $C$ classes, this would mean $N \times (C + 4)$ outputs, which could be enormous considering $N$ might be in the dozens and $C$ in the hundreds or thousands.

An alternative solution might be to use the region proposal methods discussed in previous lectures. These methods are essentially traditional segmentation techniques that can produce coherent region proposals. A convolutional neural network used for localization is then run individually on these region proposals. This method is known as R-CNN (Region Convolutional Neural Net). It is worth noting that the classifier output of the used localization network must include a ”none” output in addition to the relevant classes to filter out region proposals that do not contain objects.

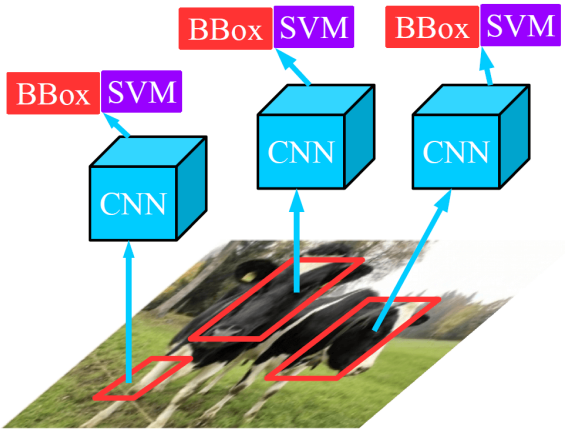


**Figure 11.8:** *R-CNN architecture.*

One of the main drawbacks of the R-CNN method is that the neural network is run separately on each region proposal, which is wasteful. An improved version of the method, Fast R-CNN, runs a network composed only of convolutional and pooling layers on the entire image and then searches for region proposals on the activation map produced. These proposals are then resized to the same size using a specialized pooling operation (traditional pooling operations scale by a given factor). Subsequently, a smaller network with only linear layers is run on the region proposals to perform classification and bounding box estimation (Figure 11.8). This method operates 10-20 times faster compared to the original R-CNN method.

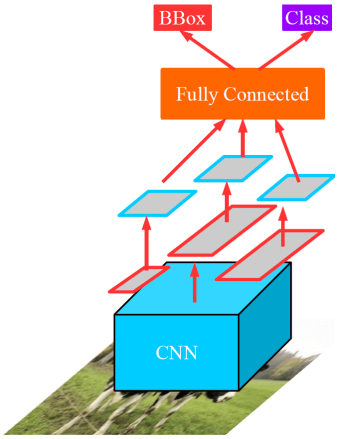


**Figure 11.9:** *Fast R-CNN architecture.*

The slowest part of Fast R-CNN's operation is the generation of region proposals, which accounts for 90% of the runtime. Therefore, an even further improved version was developed, which generates region proposals using a neural network called RPN (Region Proposal Net). This network produces a fixed number of region proposals from the activation map created by the initial convolutional part, and each is binary-classified (object/non-object). This is necessary because the network produces a fixed number of region proposals due to the fixed number of outputs. In addition to training the main detection network, the RPN network is also trained to accurately predict the bounding box and "objectness" of the regions. This method results in another tenfold speedup compared to Fast R-CNN.

### 11.3.3 YOLO

It is important to know that efficient object detection can also be achieved without region proposals. A prime example of this is the highly popular YOLO (You Only Look Once) architecture, which

should not be confused with the similarly abbreviated proverb. The YOLO solution fundamentally resembles the approach suggested at the beginning of the detection discussion, which proposed many separate localization outputs. In operation, YOLO first passes the image through a network consisting purely of convolutions and pooling layers, thus producing the activation map used for the final estimates.

For the final estimation, YOLO divides the image using an NxN grid and estimates B object candidate bounding boxes from each cell. Each bounding box is associated with a C-output classifier and a binary classifier that provides the "objectness" of the image segment falling within the bounding box. Thus, each cell has Bx(5+C) outputs from the network, generated by a 1x1 convolutional filter. It is worth noting that YOLO estimates the bounding box positions relative to the top-left corner of the YOLO cell, so each object detection is handled by the cell in which the object's center lies.

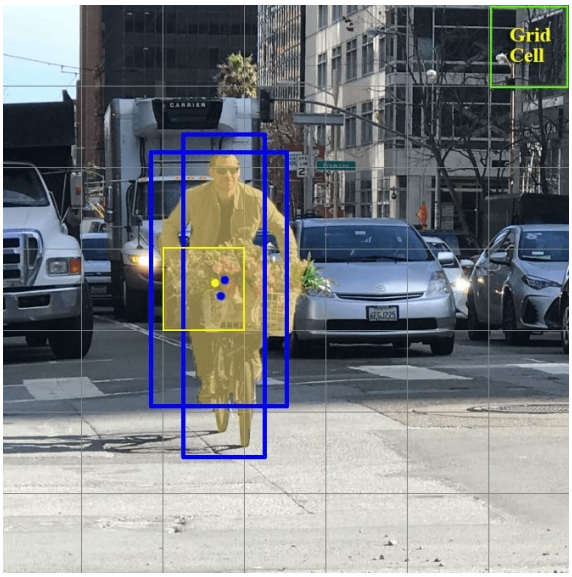


**Figure 11.10:** *Grid and estimates produced by the YOLO model.*

The width and height of the bounding box are estimated relative to a reference bounding box (i.e., anchor box), of which there are B in total (one for each estimate). The dimensions of the anchor boxes are determined by clustering the bounding boxes in the training dataset into B clusters. It is also important to mention that YOLO might detect an object multiple times during detection, in which case the prediction with the highest confidence value is retained while the others are discarded. This step is known as non-maximum suppression.

There are several versions of YOLO, with the second version introducing the use of anchor boxes. The third version performs detection at three different scales, using the scaling and upsampling tricks learned during segmentation. This solution significantly improves YOLO's performance in accurately detecting small objects. YOLO's greatest advantage over region-based detection is its exceptional speed, making it capable of real-time operation with appropriate hardware, especially the TinyYOLO variant.

### 11.3.4 Mask-RCNN

At the end of this subsection, it is worth mentioning the final task of the current topic, which is object segmentation. In this case, we aim not only to segment the image semantically but also to distinguish between different objects of the same class. Although this is clearly the most challenging task, it can still be understood with the knowledge gained so far. During RPN-based object detection, we have already produced image segments containing individual objects, so our task now is only to generate a binary mask for each object instead of the bounding box. This can be easily done using the upsampling network part known from semantic segmentation. This architecture is known as Mask R-CNN.

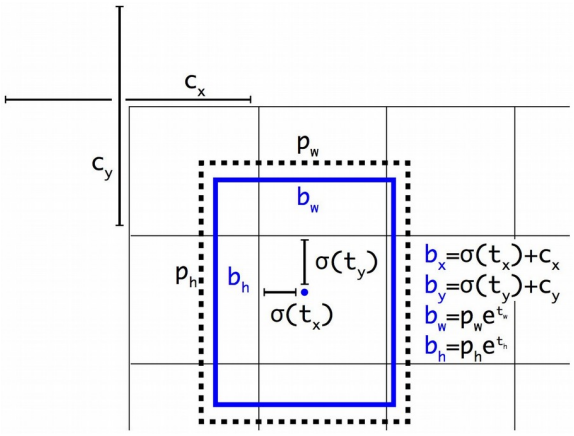


**Figure 11.11:** *YOLO bounding box estimation method. The coordinates of the bounding box are estimated relative to the grid's top-left corner, and its dimensions are relative to the anchor box.*

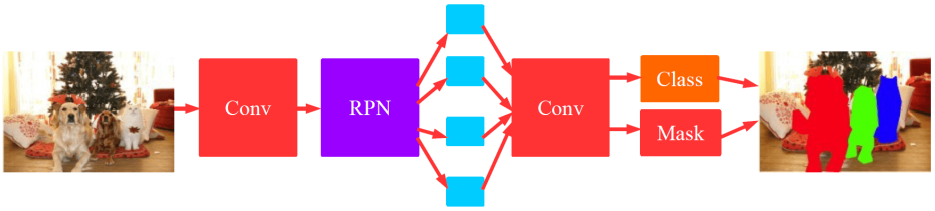


**Figure 11.12:** *Mask-RCNN architecture.*

# 12 Recurrent Networks

## 12.1 Introduction

In the previous discussion, we covered methods suitable for processing static images independently. However, there are many significant applications for processing image sequences, including video classification, or event detection. It is easy to see that while certain applications may require identifying objects in images, recognizing events or actions in videos can be even more useful. A somewhat different application is image captioning, where instead of assigning a single label to an image, we assign an entire sentence, providing a much more complex description. In this case, the output of the network, rather than its input, is interpreted as a sequence.

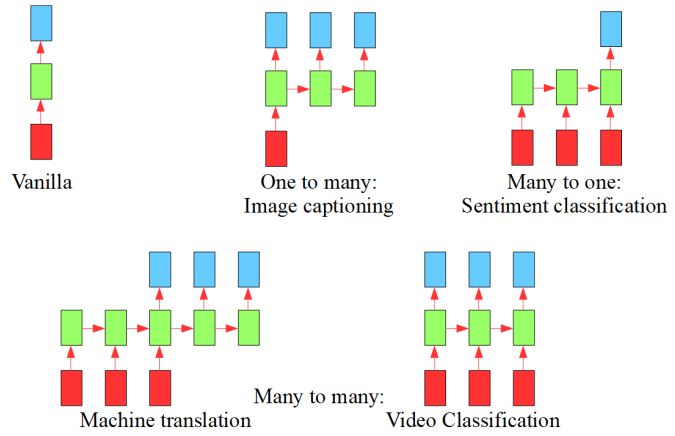


**Figure 12.1:** *Various sequence processing tasks.*

## 12.2 Recurrent Neural Networks

It is evident that feedforward convolutional networks do not have memory elements, making them unsuitable for processing temporal sequences. For effective sequence processing, a new network structure with some form of internal state is needed. Such networks are called Recurrent Neural Networks (RNNs). During the operation of a recurrent layer, the current value of its internal state is calculated based on the current input and the internal state value from the previous time step. The cell's output depends on the recently calculated current value of the internal state. The equations for an RNN cell are given by:

$$\begin{aligned} h_t &= tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ y_t &= \sigma(W_{hy}h_t) \end{aligned} \tag{12.1}$$

where $h$ denotes the internal state and $t$ represents the current time step. It is clear that an RNN cell is essentially composed of three linear layers and an activation function. With the introduction of this new architecture, the question arises about how to determine the gradients of the weights in this structure. The issue is that the backpropagation method does not work with recurrent architectures. Fortunately, this can be addressed with a simple trick: a recurrent network can be transformed into a conventional feedforward network by unfolding it in time. This means that the

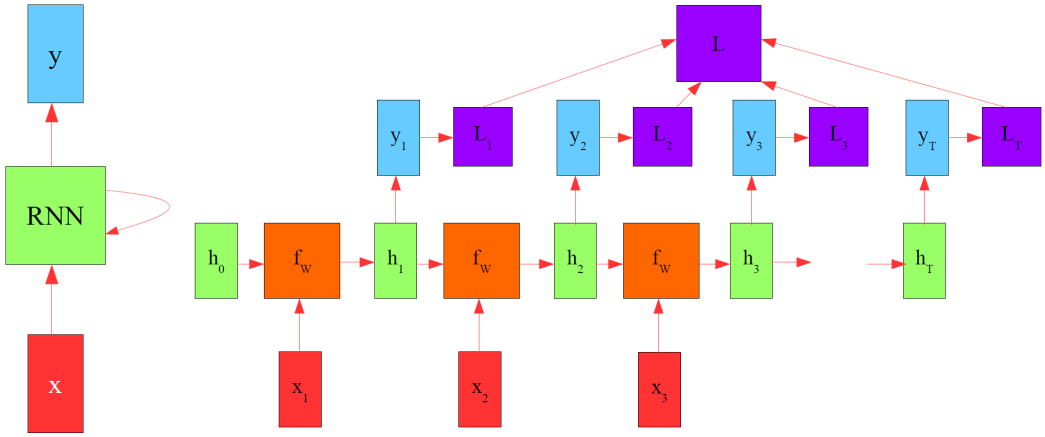


**Figure 12.2:** *Structure of an RNN cell (left) and its temporal unfolding (right).*

single RNN layer's states at different time points are treated as the layers of a traditional network arranged sequentially.

Since an RNN cell has an output and error at each time point, the unfolded network will have an output and error associated with each layer, and their sum represents the total error. From this point, the already known backpropagation algorithm can be used without any issues. However, there are two important differences compared to conventional feedforward networks. Firstly, as we progress through time, the size of the unfolded network increases, which slows down the training process. Moreover, it is not necessary to remember past inputs indefinitely for proper functioning and training. Therefore, during unfolding, the maximum size of the network is limited, and the oldest layers and inputs are removed from the unfolded network.

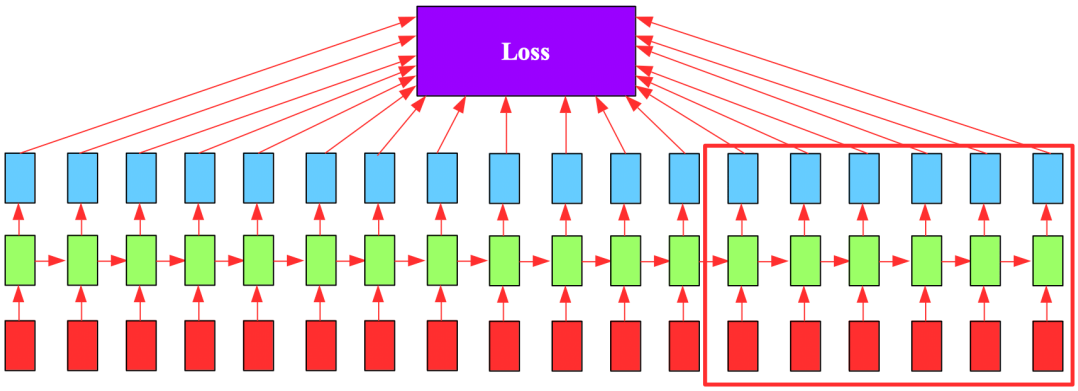


**Figure 12.3:** *Principle of the Backpropagation Through Time (BPTT) algorithm. When reaching the end of the network, we only perform backpropagation up to the red-boxed part, otherwise, the problem size would grow infinitely.*

### 12.2.1 LSTM

Another important difference is that in the unfolded multilayer network, the weight matrices of each layer are identical (since it is effectively a single recurrent layer). This means that when computing derivatives using the chain rule, we need to calculate a long product of derivatives for each layer. In this case, however, every element of the product is the same, so we are essentially talking about a power. It is easy to see that, in practice, any number or matrix raised to a high power is either zero or infinite, except when the number is exactly 1. This implies that the gradients of a recurrent cell can easily vanish or "explode," making training difficult.

The only solution to this problem is to create a structure where the derivative between the current and one-step-previous internal states is approximately one. The Long Short-Term Memory (LSTM)

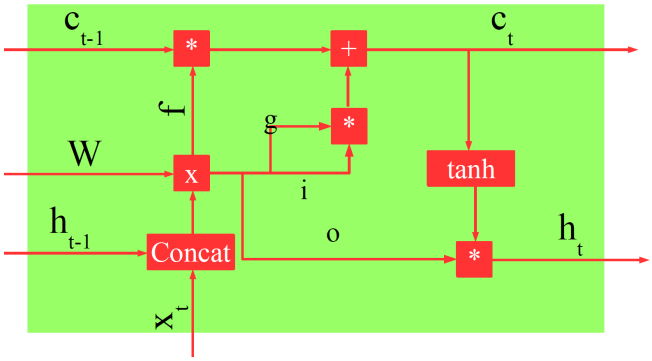


**Figure 12.4:** *Structure of an LSTM cell.*

cell is precisely such an architecture. The cell's name comes from its creators' intention to build a short-term memory cell that can remember information for a sufficiently long time, unlike the standard RNN cell. While the RNN cell consists of three linear units, the LSTM cell comprises four activation functions known as gates.

During the operation of the LSTM cell, in the absence of the effects of individual gates, the previous value of the cell state $c$ is simply copied to the current state, making the derivative between the two exactly one. However, the value of the cell state may still be modified by the effects of the gates. The first such gate is the forget gate $f$, which is a vector of the same size as the cell state, with each element ranging between zero and one due to the sigmoid nonlinearity. By element-wise multiplying the cell state vector by this vector, certain parts of the cell state are forgotten.

The next gate is the so-called input gate $i$, which extracts features from the current input value and the previous cell output value that can be remembered in the cell state. Then, similar to the forget gate, the input gate vector is element-wise multiplied with the output of the input gate, selecting the relevant features to be remembered and added to the cell state. The final step is to produce the cell's current output, which is obtained by filtering the cell state with the output gate $o$.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \tag{12.2}$$
$$c_t = f * c_{t-1} + i * g$$
$$h_t = o * tanh(c_t)$$

where $*$ denotes element-wise multiplication. It is worth noting that there are several variations of the LSTM cell with minor differences, as well as recurrent cells with more significant deviations but based on similar fundamental ideas. For example, the Gated Recurrent Unit (GRU) is a similar concept. It is also important to recognize that facilitating the smooth flow of gradients backward using so-called "shortcut" connections was not first introduced here. The basic idea of the residual block described in the previous chapter was very similar.

### 12.2.2 Application Examples

Besides video classification, there are several other interesting applications for sequence processing methods. Notable among these is image captioning, where short, 1-2 sentence descriptions are assigned to static images provided as input. In this case, recurrence occurs during sentence generation. Another important application is implementing various (visual) question-answer systems. Here, the algorithm must answer questions posed about an input image or video (e.g., "What color is the hat on the man wearing sunglasses?"). It is also worth mentioning the implementation of systems with explicit memory (e.g., Turing machines).

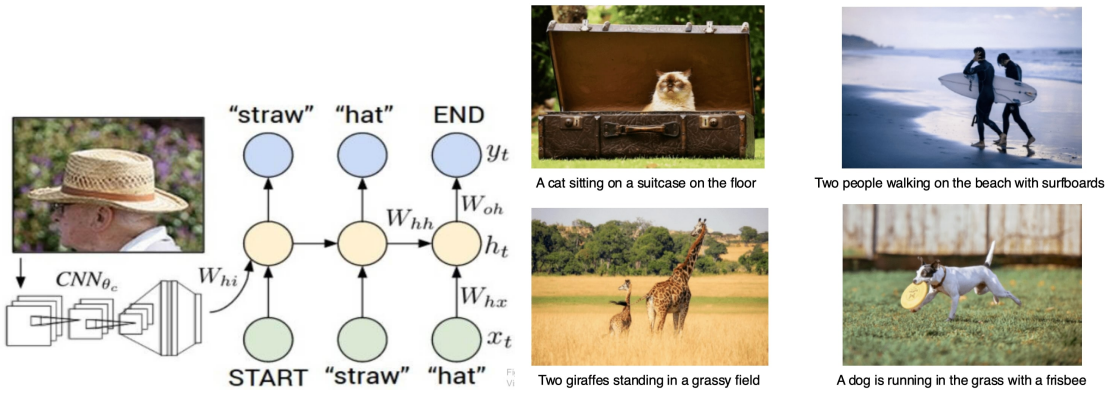


**Figure 12.5:** *Principle of image captioning (left) and results (right).*

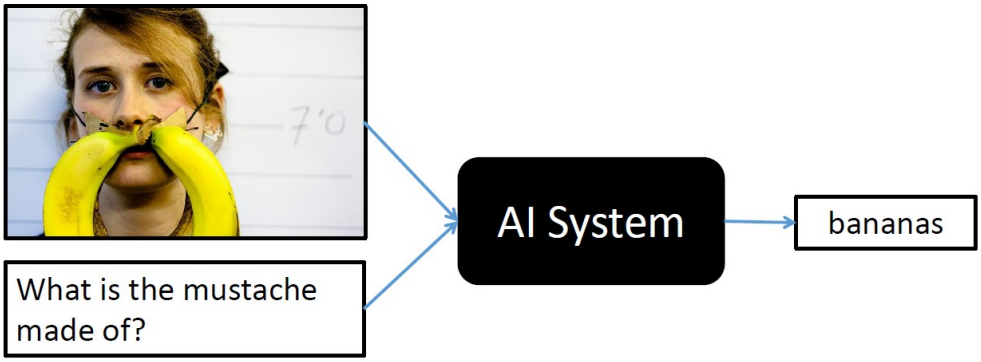


**Figure 12.6:** *Problem of visual question-answer systems.*

A particularly interesting application of recurrent networks is the implementation of the so-called soft attention model. In this model, a recurrent cell assigns weights to different parts of an image based on its content, and at each step, the cell considers the parts of the image with these weights. During operation, the cell generates new weights at each step, continuously changing the areas of the image the network focuses on. In image captioning, it is observed that at the moment of generating each word, the network focuses precisely on the part of the image where the object corresponding to the word is located.

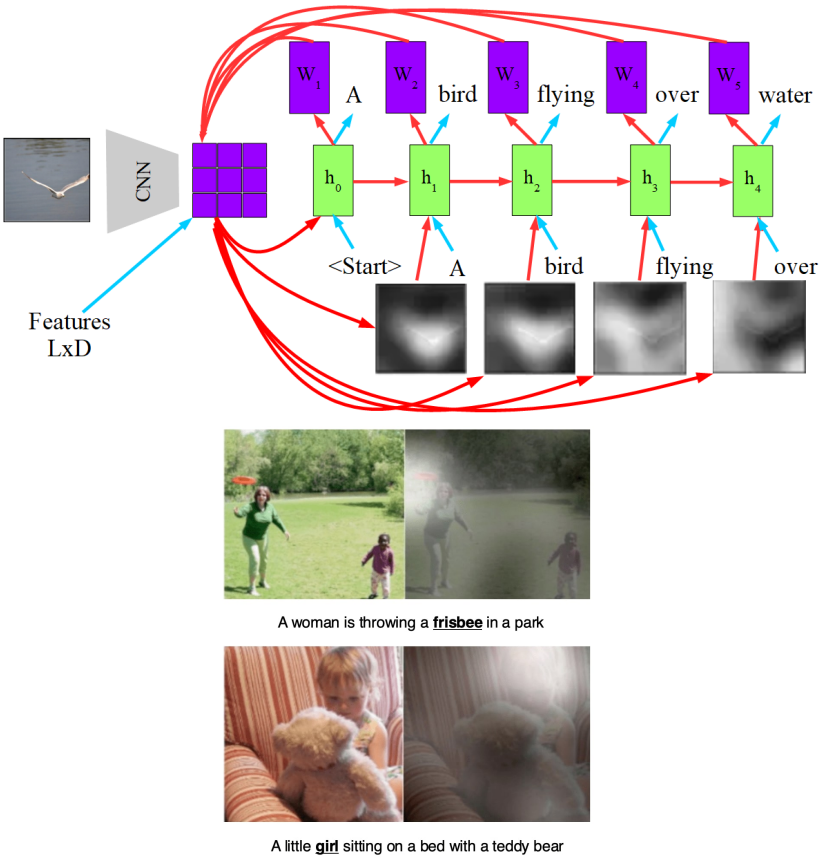


**Figure 12.7:** *Principle of soft attention (top) and its effect on captioning (bottom). These images show the attention weights generated by the network at the moment of generating the underlined word.*

# 13 Unsupervised Learning

## 13.1 Introduction

Thus far, we have discussed supervised learning, meaning all these deep learning algorithms require a dataset with thousands or even millions of hand-labelled examples to learn properly. Not only is this often prohibitively expensive, it also underperforms human capabilities, as we are often able to learn new relationships from only a few examples. It would therefore make sense to try and reproduce human capabilities.

It is important to realize that humans do indeed see million of images durin childhood, however, there images are not 'labelled' (e.g. there is no external feedback associated to them). Instead, humans somehow learn to make sense of visual inputs all by themselves. Think about it, by the time a child asks "what is this?" pointing at a ball, they have already learned how to recognize balls (otherwise, how would they know that it's a thing), they are only asking for the category name.

The big question is, how can we reproduce this ability with neural networks? The answer is in the field of unsupervised learning, which - as its name suggests - aims to learn internal representations of images with unlabelled data. In this chapter we'll attempt to provide a brief overview of these methods, divided into 3 types: generative, reinforcement learning, and self-supervised learning.

## 13.2 Generative Models

As we have seen previously, the visualization of the network architectures discussed so far is feasible, and these networks may be somewhat suitable for image synthesis. However, their capability in this regard is limited because the methods of training discussed so far result in what are known as discriminative neural networks. This means that the neural networks only learn how the expected output depends on the input, for example, how two classes differ visually. In terms of probability theory, this means that discriminative algorithms learn the conditional probability function $P(y|x)$ between the input and output.

This, however, implies that they have no knowledge about the distribution of the inputs (i.e., what the known classes actually look like). To enable the neural network to learn the appearance of images, it needs to learn the joint density function $P(y, x)$. The consequence of this is that the distribution of the images is also known, allowing us to generate realistic images by sampling randomly from this distribution. Networks trained in this way are therefore called generative models. It is worth noting that the joint density function is more complex, so although a generative model can also be used for classification, discriminative models generally perform better.

Generative models can also be trained when labels are not available; in this case, we only teach the network the distribution of the inputs $P(x)$. This is known as unsupervised generative learning.

### 13.2.1 Variational Autoencoder

The basic idea of the Variational Autoencoder (VAE) is that there exist some unknown, hidden (latent) variables $z$ that can compactly describe the content of an image. The goal of the Variational Autoencoder is to learn to generate the corresponding image from these latent variables. It is

important to know that a VAE can be trained without the designers knowing exactly what these latent variables are.

$$P_\Theta(\text{Image}) = \int p_\Theta(z) p_\Theta(x|z)\, dz \tag{13.1}$$

Before discussing the Variational Autoencoder, it is useful to briefly introduce its predecessor, the traditional Autoencoder. These neural network models were popular in the late 2000s when, without the knowledge of the procedures discussed in Lecture 9, we could not reliably train deep neural networks. The essence of an autoencoder is to map an input vector to an internal hidden layer. However, it is important that this hidden layer contains fewer variables than the input. The Autoencoder's task is then to reconstruct the original input from this hidden layer with the smallest possible squared error (with a larger hidden layer, this would trivially be simple).

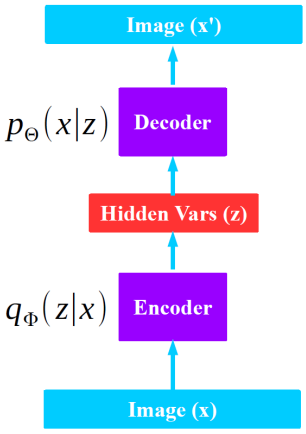


**Figure 13.1:** *Structure of an Autoencoder.*

An Autoencoder can naturally consist of multiple hidden layers; in this case, the layers are added to the architecture individually, each trained to restore the activations of the previous layer. Such an architecture, known as a Stacked Autoencoder, can be divided into two parts: the down-sampling encoder part, which maps the input to the space of hidden variables, and the up-sampling decoder part, which generates the input from the hidden variables. In the early 2000s, the encoder part was frequently used for classification tasks.

One might wonder if we can use the decoder network to synthesize an image from some random latent variable vector. However, there is a major problem: to generate latent variables corresponding to realistic images, we need to know their distribution. Unfortunately, in the case of Autoencoders, we cannot assume that the latent variables are normally distributed. Therefore, one might consider trying to enforce a standard normal distribution on the latent variables during training.

To achieve this, we can use the concept of KL-divergence previously introduced, which measures the difference between the entropy and the cross-entropy of two distributions $p_1$ and $p_2$, or the similarity between them:

$$D_{KL}(p_1 \| p_2) = E_x \left[ \log \frac{p_1}{p_2} \right] \tag{13.2}$$

During the training of Variational Autoencoders, the cost function consists of two parts: the first is the squared reconstruction error known from Autoencoders, while the second is the KL-divergence between the prescribed and the network-generated distributions.

$$L_i(\Theta, \Phi) = E_x \left[ \log p_\Theta(x_i|z) \right] - D_{KL}(q_\Phi \| p(z)) \tag{13.3}$$

where $p_\Theta$ is the distribution represented by the decoder (reconstruction error), $q_\Phi$ is the distribution of the encoder, and $p(z)$ is the prescribed distribution of the latent variables.

It is worth noting that VAE networks do not directly generate the latent variable values and the reconstructed output, but rather their expected values and covariance matrix. During training and testing, we sample from these, and the network is trained with the generated values.

It is also important to know that the KL-divergence criterion applied to the latent variable is applied to the joint distribution of all latent variables generated for all images within a minibatch, not on a per-image basis. This is important because we want the distribution of the latent variables to be normal across the entire dataset but not necessarily the same for each individual image (otherwise, we would not be able to perform the reconstruction).

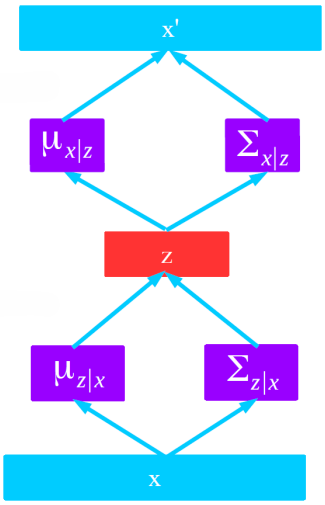


**Figure 13.2:** *Structure of a Variational Autoencoder.*

The Variational Autoencoder obtained in this way is a useful procedure, which is significantly faster than the PixelRNN-based solution, but the generated images are quite blurry, even in the case of convolutional VAE solutions.

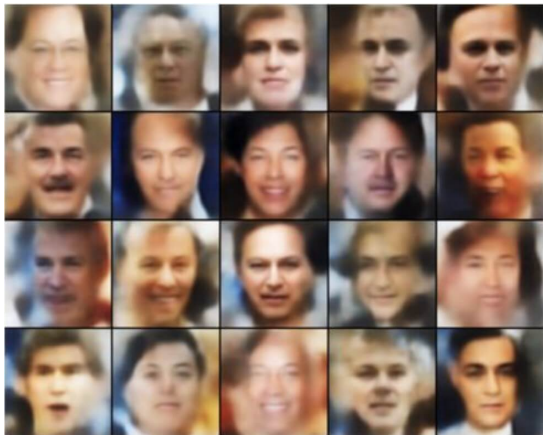

**Figure 13.3:** *Face images generated by a Variational Autoencoder.*

### 13.2.2 GAN

The most well-known generative model is the Generative Adversarial Network (GAN). This family of architectures draws its fundamental idea from game theory rather than probability theory: it is based on two competing neural networks. One is the discriminator network, which is fundamentally a down-sampling binary classifier network. The discriminator's task is to decide whether an image it receives as input is real or a generated fake. Its adversary is the generator network, which is

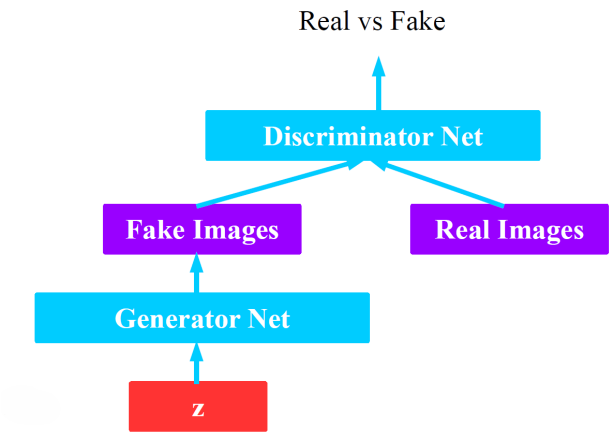


**Figure 13.4:** *Architecture of GAN networks.*

an up-sampling network that attempts to generate an image from a given random noise-like latent variable vector in such a way that it can deceive the discriminator network.

During the training of GANs, we simultaneously optimize the cost of the two networks as follows:

$$\begin{gathered}\max_{\Theta_d}[E_x \log D_{\Theta_d}(x) + E_z \log(1 - D_{\Theta_d}(G_{\Theta_g}(z)))] \\ \min_{\Theta_g}[E_z \log(1 - D_{\Theta_d}(G_{\Theta_g}(z)))]\end{gathered} \tag{13.4}$$

Thus, the discriminator network tries to maximize its output on real images while minimizing its output on generated images. Meanwhile, the generator network aims to minimize this second term. To achieve this, we use the SGA (Stochastic Gradient Ascent) method for the discriminator and the usual SGD method for the generator. However, this presents a significant problem: due to the logarithmic form, the gradient of the cost function is small when the generated images are very poor, which is usually the case at the beginning of training. Therefore, we modify the cost function so that the generator aims to maximize the chance of successful deception rather than minimizing the chance of getting caught. Thus, we also use the SGA method on the generator. As a result, the cost functions change as follows:

$$\begin{gathered}\max_{\Theta_d}[E_x \log D_{\Theta_d}(x) + E_z \log(1 - D_{\Theta_d}(G_{\Theta_g}(z)))] \\ \max_{\Theta_g}[E_z \log(D_{\Theta_d}(G_{\Theta_g}(z)))]\end{gathered} \tag{13.5}$$

A typical training cycle for GANs proceeds as follows:

1. For $k$ steps:
   (a) Generate a minibatch of images
   (b) Request a minibatch of real images
   (c) Train the discriminator
2. Generate a minibatch of images
3. Train the generator

GAN networks continue to be highly successful in generating images today. One reason for this is that they can easily be converted into convolutional versions, typically referred to as DCGANs (Deep Convolutional GANs). These networks generally solve down-sampling and up-sampling using strided convolutional layers. For up-sampling, the convolution is, of course, transposed. It is

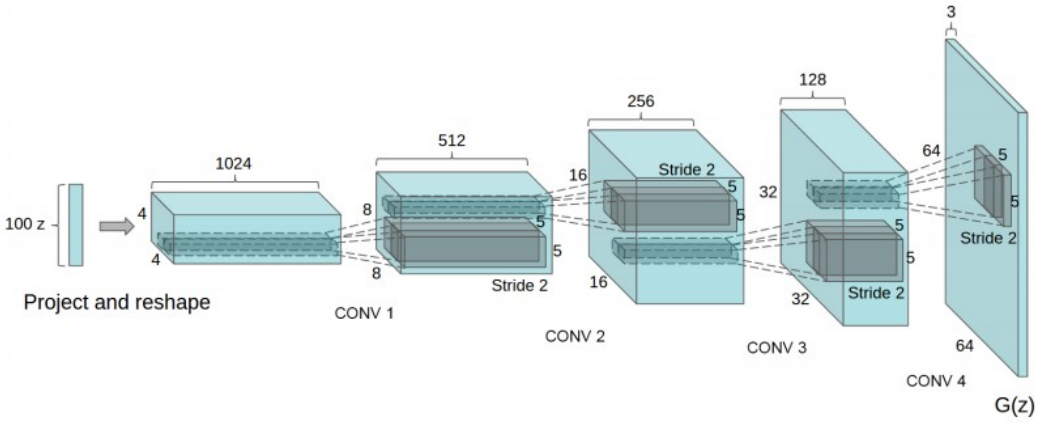


**Figure 13.5:** *Structure of a DCGAN generator.*

also important to use batch normalization and carefully choose the activation function, as GANs often struggle with convergence issues. The discriminator typically uses Leaky ReLU, while the generator uses ordinary ReLU.

It is important to mention that the latent variables used by GAN networks can indeed describe certain properties of the generated images in a way that makes sense to humans. A good example is that mathematical operations applied to latent variable vectors work in a way consistent with human logic, which is not true for raw images. Moreover, interpolating between two arbitrary points in the latent variable space interestingly yields semantic transitions between the contents of the generated images.

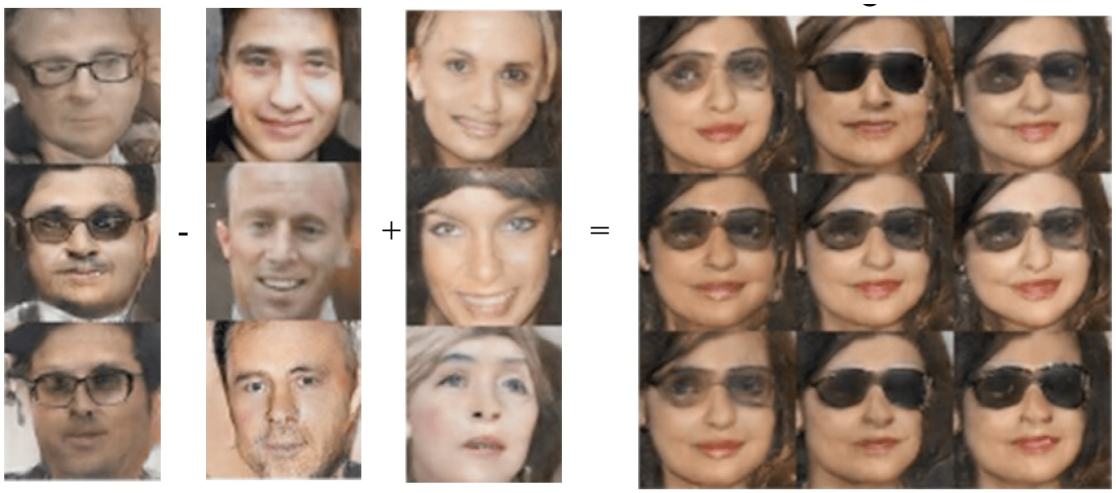


**Figure 13.6:** *Mathematical operations in the latent variable space of GAN-generated images yield semantically meaningful results.*

Although GANs are capable of generating high-quality images, they suffer from many practical difficulties. For example, it is difficult to interpret the meaning of the latent variables, as semantically interpretable variables in the GAN latent space often appear "entangled" with other variables. This means that a variable does not just represent the presence of glasses, smiling, or hair color, but some combination of these attributes.

Training GANs is also extremely challenging because effective learning requires that neither network part wins the competition; otherwise, learning stops. A particular case of this is the so-called Mode Collapse, which occurs when the discriminator wins and can recognize all the images in the database. In this case, the generator tries to adapt to a few examples that the discriminator currently deems realistic. Consequently, the generator will produce the same image (or at least very similar images) for every input.

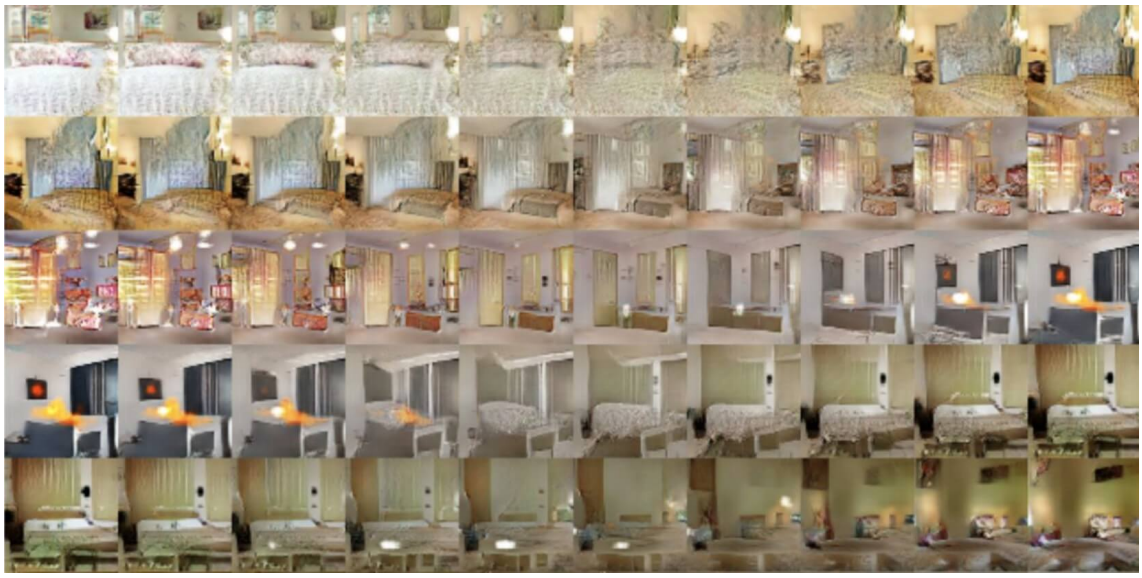

**Figure 13.7:** *Indoor images generated by a DCGAN network (interpolated between random points in the latent vector space).*

## 13.3 Reinforcement Learning

In previous lectures, we explored methods of supervised learning. However, supervised learning has several drawbacks: On the one hand, it often requires a large labeled dataset, which is costly to produce. On the other hand, with these methods, we essentially "copy" our own abilities into the artificial intelligence model, but the algorithm itself does not develop these abilities independently. Thus, the theoretical possibilities of supervised learning are limited by our own constraints.

In reinforcement learning, however, the algorithm (agent) interacts with an external environment, with the goal of independently solving a task within this environment. The problem can be framed such that the environment has states observable by the agent, the agent makes decisions about the actions it will perform based on these observations, and as a result, the environment transitions to a new state. A key element of the reinforcement learning environment is the reward, which is feedback provided by the environment that is observable and related to the quality of task completion. This reward is often rare, occurring infrequently rather than at every time step.

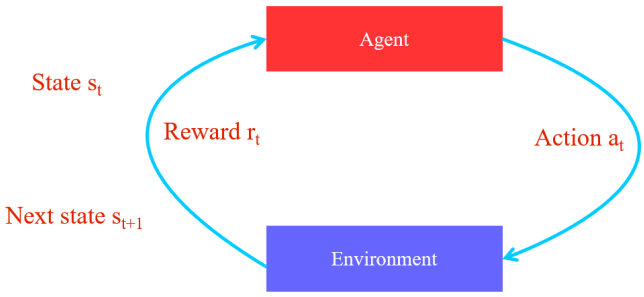


**Figure 13.8:** *Interaction between the agent and the environment.*

It is important to note that, unlike supervised learning, we do not have information about the correct or optimal action. Moreover, if the reward is only realized at the end of a relatively long sequence of decisions, we do not know which actions influenced this final reward. In such an environment, traditional learning paradigms are not applicable.

There are many examples of reinforcement learning, one of the simplest being the Cart-Pole problem, where the state of the environment consists of the position and velocity of the cart, as well

as the angle and angular velocity of the pole. The action can be interpreted as the lateral force applied to the cart, and the reward is 1 at every time step as long as the pole does not deviate too much from the vertical position. Similar problems can be framed in the context of computer games as reinforcement learning problems, where the state is given by the screen pixels and the action space consists of the buttons or controls used to manage the game. Here, the reward is represented by winning the game or the number of points achieved.

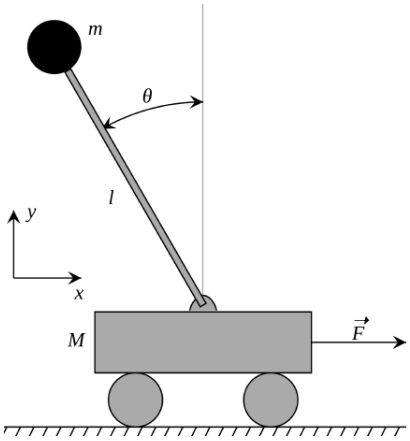


**Figure 13.9:** *The Cart-Pole problem.*

Of course, many other problems can be framed in this manner, ranging from the control of complex robots and vehicles to various board games (chess, checkers, Go) and even aspects of life itself.

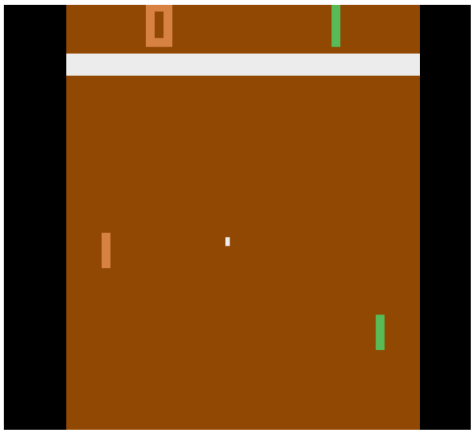

**Figure 13.10:** *An artificial intelligence (right) playing against a computer agent (left) in the Atari game Pong.*

### 13.3.1 Markov Decision Process

To introduce the methods of reinforcement learning, we first need to provide a mathematical description of the problem. Reinforcement learning is commonly described as a Markov Decision Process (MDP). Like the Hidden Markov Model (HMM), the MDP satisfies the Markov property, meaning the current state fully describes the world and its entire history. An MDP can be described with the following structure:

$$(S, A, R, P, \gamma) \tag{13.6}$$

where $S$ is the set of states, $A$ is the set of actions, $R(s, a)$ is the reward distribution for a given state-action pair, and $P(s, a)$ is the distribution of the next state. $\gamma$ is the discount factor used to

exponentially weight past rewards. The purpose of this is to continually reduce the impact of past actions on the current reward. The key characteristic of an agent in the environment described by the MDP is the policy $\pi$, which is a function that assigns an action (or a probability distribution over actions) to each state. The goal of reinforcement learning is to find the policy that maximizes the expected value of the total reward.

## 13.4 Policy Gradients (REINFORCE)

Despite this, it often happens that despite the extremely complex Q-function, the strategy itself is relatively simple. Thus, the question arises whether it is possible to learn it directly. Such a neural network would be quite similar to traditional classifier neural networks (at least in the case of discrete actions), as the goal would be to assign probabilities to each action for a given state, based on which we can make a choice.

During training, we simply need to maximize the expected value of future discounted rewards, provided that we follow the strategy implemented by the network. We can apply gradient-based maximization methods for this.

### 13.4.1 Gradient Calculation

For this, we need to differentiate the cost function with respect to the network parameters. First, we express the cost function as follows:

$$J(\Theta) = E[r(\tau)] = \int_{\tau} r(\tau) p(\tau; \Theta) \tag{13.7}$$

where $\tau$ is the trajectory resulting from the strategy implemented by the network. Its derivative with respect to $\Theta$ is:

$$\nabla_{\Theta} J(\Theta) = \int_{\tau} r(\tau) \nabla_{\Theta} p(\tau; \Theta) \tag{13.8}$$

However, we encounter a significant problem: this integral is unsolvable, even in the discrete case, when it would reduce to a summation. We can somewhat improve this by modifying the equation. Using the derivative identity, we can write:

$$\nabla_{\Theta} p(\tau; \Theta) = p(\tau; \Theta) \frac{\nabla_{\Theta} p(\tau; \Theta)}{p(\tau; \Theta)} = p(\tau; \Theta)\ \nabla_{\Theta} \log p(\tau; \Theta) \tag{13.9}$$

Substituting this back into the original expression:

$$\nabla_{\Theta} J(\Theta) = \int_{\tau} r(\tau) \nabla_{\Theta} \log p(\tau; \Theta)\ p(\tau; \Theta) = E[r(\tau)\ \nabla_{\Theta} \log p(\tau; \Theta)] \tag{13.10}$$

One might think, "Well, now the expression looks much nicer." However, we see that the gradient of the error is the expected value of the reward and the derivative of the trajectory probability. If we can determine the last term of the product, this expected value can be estimated using the Monte Carlo method. The essence of the Monte Carlo method is to approximate the expected value with the average using random sampling. Note that we have already been doing this in mini-batches when applying stochastic gradient methods.

The question is whether we can determine the gradients of the trajectory probability without knowing the transition probabilities. It can be easily written that the probability of the trajectory is:

$$p(\tau;\Theta) = \prod_{t\geq 0} p(s_{t+1}|s_t,a_t)\ \pi_\Theta(a_t|s_t) \tag{13.11}$$

which requires knowledge of the transitions. However, fortunately, we first need to take the logarithm:

$$\log p(\tau;\Theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t,a_t) + \log \pi_\Theta(a_t|s_t) \tag{13.12}$$

Subsequently, if we differentiate this with respect to the network parameters, the transition probabilities drop out, as they do not depend on the network parameters.

$$\nabla_\Theta \log p(\tau;\Theta) = \sum_{t\geq 0} \nabla_\Theta \log \pi_\Theta(a_t|s_t) \tag{13.13}$$

Ultimately, we obtain this formula for the derivative of the cost function:

$$\nabla_\Theta J(\Theta) = \sum_{t\geq 0} r(\tau) \nabla_\Theta \log \pi_\Theta(a_t|s_t) \tag{13.14}$$

That is, the derivative is proportional to the derivative of the logarithm of the network's output (in this case, weighted by the reward). Note that this somewhat resembles the error obtained during classification with the cross-entropy function, where we also had to take the negative logarithm of the correct class probability. The difference here is that, in this case, the correctness is determined by the reward value.

The method obtained in this way is called Policy Gradient, or in some cases, the REINFORCE algorithm. Although its derivation is quite complex, the algorithm is relatively easy to understand: the network performs a series of actions for which it receives some reward. If the reward is good, we modify the network so that it is more likely to perform the same series of actions in similar future cases, that is, we reinforce the network's decision. Otherwise, we modify the network so that the execution of the same series of actions is less likely, meaning we discourage the decision.

### 13.4.2 Noise Reduction and Baseline

One central problem with the Policy Gradient method (and reinforcement learning in general) is the so-called credit-assignment problem. This means that when executing a sequence of decisions, we cannot determine which action was truly responsible for the reward. As a result of this problem, the gradient estimate can be extremely noisy, which can complicate or even prevent convergence.

To address this, several methods are commonly used together. On the one hand, we modify the cost function so that it does not consider the entire trajectory reward for a given action but only the rewards received after the specific action. Additionally, we often discount the rewards so that rewards that are too far in time have less impact on the goodness of the action.

$$\nabla_\Theta J(\Theta) = \sum_{t\geq 0} \Big( \sum_{t'\geq t} \gamma^{t'-t} r_{t'} \Big) \nabla_\Theta \log \pi_\Theta(a_t|s_t) \tag{13.15}$$

### 13.4.3 Actor-Critic Networks

Another problem with the Policy Gradient method is that the reward values received from the environment are usually non-negative. In this case, we always reinforce the algorithm's decision,

just to a greater extent for better strategies. This is not ideal from a convergence perspective, as it means we will also slightly encourage poor steps. Therefore, it would be beneficial to calculate a baseline reward value and subtract this value from the current reward. Thus, we reward performance better than the baseline and punish performance worse than the baseline. The simplest way to compute the baseline reward is to use the moving average of previous rewards. This ensures that we reward the algorithm only if it can surpass its previous average performance.

$$\nabla_\Theta J(\Theta) = \sum_{t\geq 0} \Big( \sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \Big) \nabla_\Theta \log \pi_\Theta(a_t|s_t) \tag{13.16}$$

Another, more sophisticated solution is to consider a reward as good if it is greater than the expected value of the reward achievable from the given state (this is expressed with a so-called value function, denoted by $V$). We can also define the Q-function, which represents the expected future reward achievable with a given state-action pair.

$$\begin{gathered} \nabla_\Theta J(\Theta) = \sum_{t\geq 0} \Big( Q^{\pi_\Theta}(s_t, a_t) - V^{\pi_\Theta}(s_t) \Big) \nabla_\Theta \log \pi_\Theta(a_t|s_t) \\ A^{\pi_\Theta}(s,a) = Q^{\pi_\Theta}(s,a) - V^{\pi_\Theta}(s) \end{gathered} \tag{13.17}$$

The rationale behind this is that if the strategy (and consequently the Q and V value functions) is not optimal (which it is not during training), there will be actions that are better than the state value function. If we take such an action, we reward it; otherwise, we punish it. The advantage function described above can be learned by another neural network via regression.

The method using this solution is called the Actor-Critic method. Based on what has been described, it consists of two separate networks: the Actor network learns the optimal strategy using the Policy Gradient method, while the Critic network attempts to compute the advantage function using regression learning. It is worth noting that it is also useful to use the experience replay principle described earlier with this method.

Actor-Critic networks can achieve quite high-quality performance in reinforcement learning tasks. However, it is worth noting that there are many variations of this method. One of the primary variants is the Asynchronous Advantage Actor-Critic (A3C) model, which runs multiple agents in parallel across different environments. The model parameters are updated stochastically, eliminating the need for experience replay. Another variant is A2C, which performs parameter updates synchronously, ensuring that models do not operate based on outdated parameters. Experiments have shown that the A2C model can achieve better efficiency.

### 13.4.4 Rare Rewards and Curiosity

In reinforcement learning, there are cases where learning is difficult or even practically impossible. One such case is when rewards can only be obtained through very rare and complex action sequences in the given environment. Since these networks act randomly at the beginning of training, the statistical chance of accidentally discovering a good solution is extremely small. This problem can be partially addressed by not always selecting the most probable action according to the policy network's output, but by sampling randomly from the distribution.

For rare external rewards, an innovative and intensely researched solution is to use various intrinsic reward-based learning systems. These intrinsic rewards are often related to curiosity and prediction. The essence of these models is to use the previous state and action to predict the next state as accurately as possible, that is, to better understand the environment's dynamics. In practice, we often estimate not the state directly but a next value of an internally generated descriptive vector from the state. This is done because the state is often very high-dimensional, and much of the information contained in it is not relevant.

One such solution is the so-called World Model architecture. This structure consists of three parts: The Vision module is responsible for providing an internal representation of the image

seen by the network, the Memory module is a recurrent cell responsible for integrating successive representations, and the final module, the Controller, determines the action. An important part of the World Model architecture is that, similar to previously known autoencoders, the network must be able to reconstruct the original state from the internal representation.

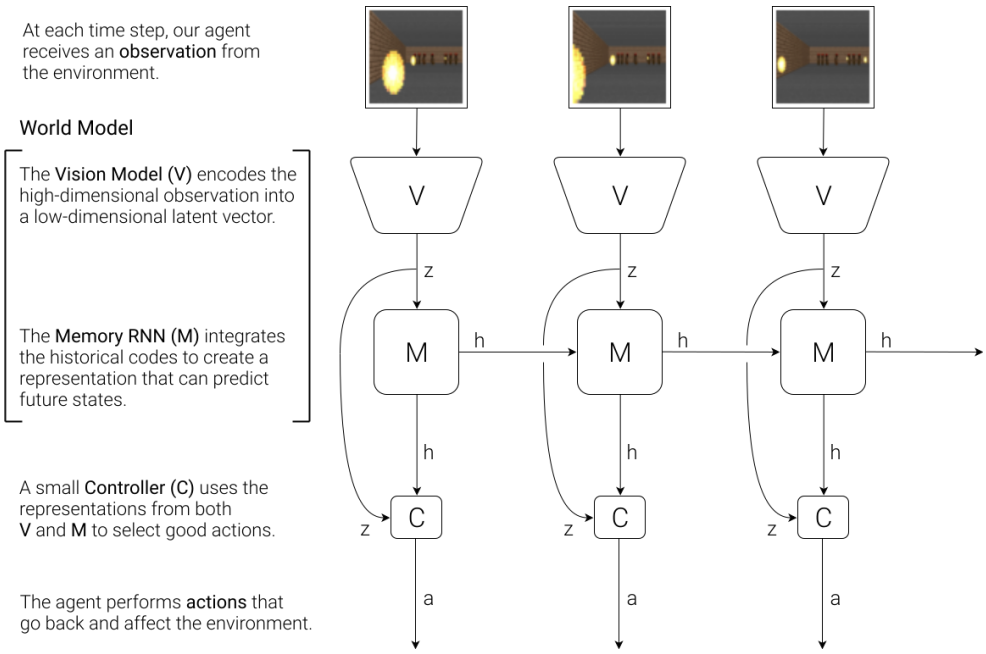


**Figure 13.11:** *The structure of the World Model.*

Such predictive models are often combined with curiosity mechanisms, where we try to reach states we have not yet been to. This is often measured based on the estimated distribution of previous states: unlikely states are considered novel, while probable states are considered known. To better understand the environment, we often estimate not just the next state but also the action leading from one state to another based on two successive states. This is often necessary because the next state can be easily estimated if the network assigns a zero internal representation to every state. However, this makes it impossible to deduce the action between the two states.

In reinforcement learning, another difficulty arises when the environment described by a Markov decision process has hidden states. In such cases, the convergence of algorithms becomes particularly challenging. A typical example is Starcraft (or similar real-time strategy games), where the Fog of War mechanism keeps significant portions of the game space hidden from the algorithm.

## 13.5 Self-Supervised Learning

In previous chapters, we explored two different forms of unsupervised learning, addressing the problems of image generation and task-solving in an environment. When discussing GANs, we highlighted that the internal representation learned by the generator is particularly expressive, and can even be useful for pre-training encoder neural networks. However, training GANs is specifically difficult and cumbersome, leading to the question: Is there a simpler way to pre-train a neural network so that it learns useful features for solving future tasks?

The answer is yes, and this chapter's topic is exactly that method. If we recall the example of transfer learning mentioned earlier, we can see that we solved a very similar problem there, but used a different – generally larger – labeled dataset for pre-training. In this chapter, we focus on methods that can learn from an unlabeled image database by automatically generating tasks from the images, thus the labels can also be produced without human intervention. This solution is referred to as self-supervised learning.

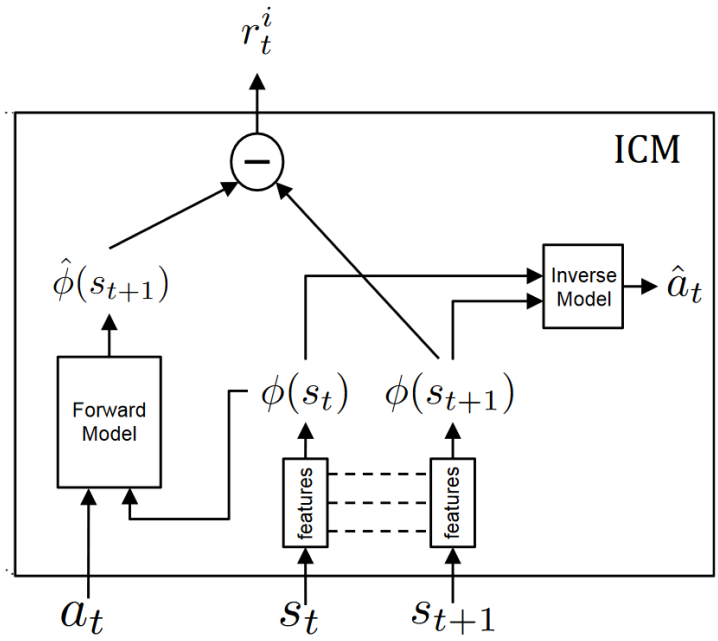


**Figure 13.12:** *The structure of the curiosity model.*

### 13.5.1 Autoencoders

One important family of self-supervised methods is based on Autoencoder architectures. The first and one of the most important methods in this family is the so-called Denoising Autoencoder. This solution is important not only here but also due to its application in diffusion generative models. The denoising autoencoder – as the name suggests – receives a deliberately noisy input and its task is to reconstruct the original noise-free input as accurately as possible. Therefore, it must not only understand what is noise and what is useful signal in an image, but also infer the information lost due to noise from the context.

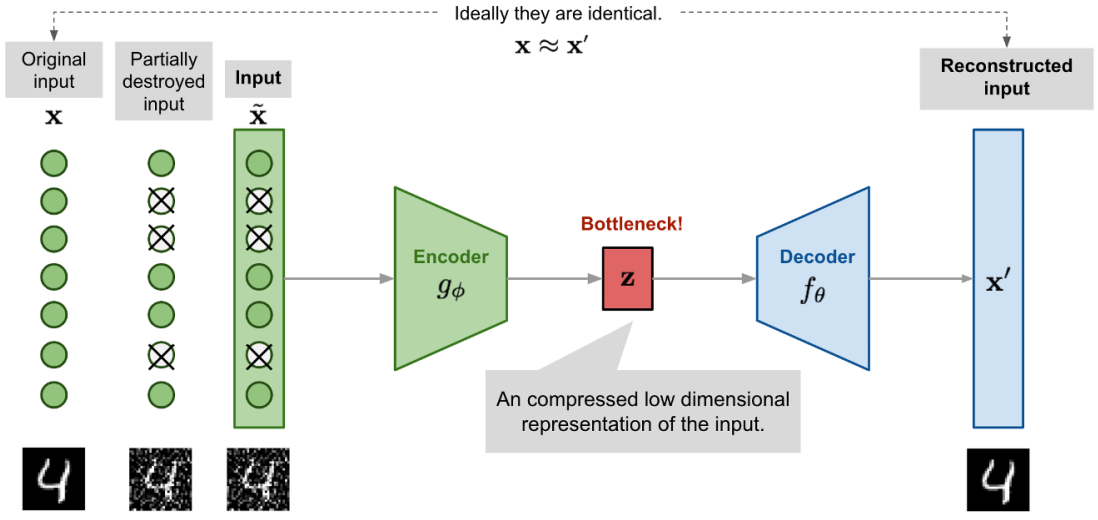


**Figure 13.13:** *The Denoising Autoencoder.*

A method similar to image colorization is the Split-brain Autoencoder, which aims to reconstruct the color channel from a grayscale image. However, this solution not only performs this task but also its inverse: it must also infer the grayscale component from the color channel. Thus, it essentially needs to handle all color conversion tasks simultaneously, which likely enhances its generalization capability.

Another interesting method is the Context Autoencoder, which aims to reconstruct a masked patch from the center of the image as accurately as possible based on the context. A significantly more frequently used variant of this is the so-called Masked Autoencoder (MAE), which not only masks a central patch but divides the image into patches and masks a large majority of them (usually more than 75%). The encoder part of the Autoencoder only receives the remaining patches and its task is to correctly reconstruct the full input.

An important aspect of this method is that the loss function includes not only a reconstruction error term but also a GAN error term. This means that the MAE includes a separate discriminator network trained to distinguish between the original input and the reconstruction by the MAE. Research has found that this extra term significantly improves the performance of the MAE. In summary, the MAE is a particularly effective self-supervised learning algorithm and can even be used as a form of data augmentation/regularization in other networks.

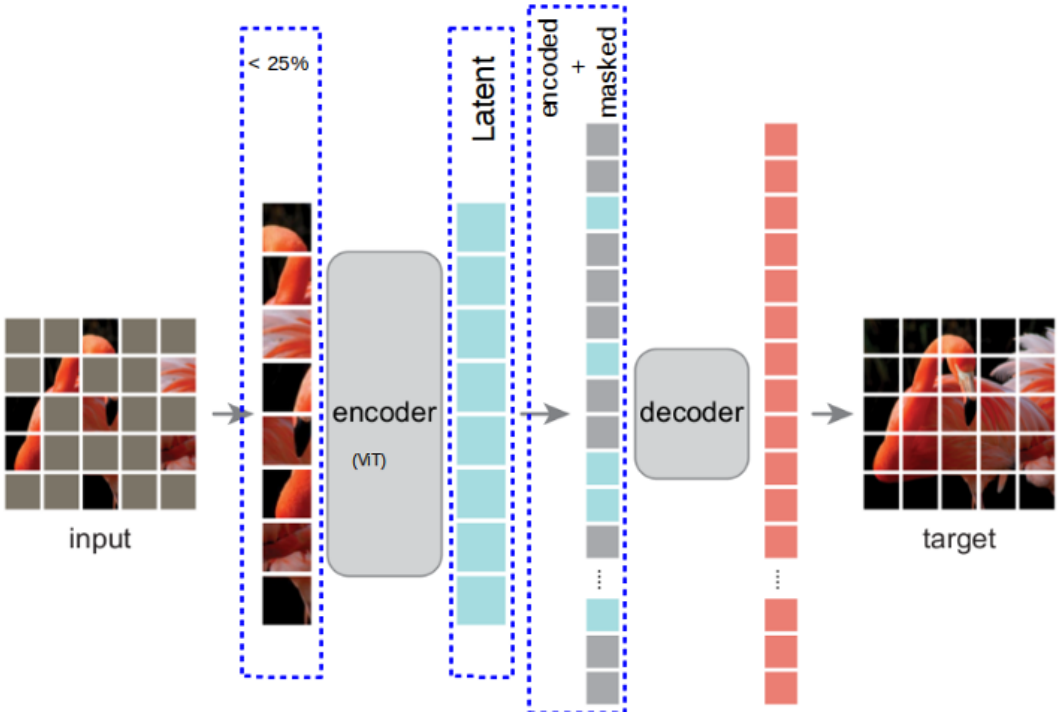


**Figure 13.14:** *The Masked Autoencoder.*

## 13.6 Contrastive Learning

The methods discussed so far have been based on either some explicit pre-text task or some prediction task. The idea was that performing these tasks would require the network to learn generally useful image features, which could then be applied to other tasks. We have already hinted at the fact that this may not always be true when discussing pre-text tasks. Although performing different predictions seems like a more general task, we can still ponder whether it is possible to explicitly train the network to map similar images to similar locations within the learned representation space, while mapping different images to different locations.

The answer is yes, it is possible, and these methods are collectively known as contrastive learning algorithms. Formally, for such an algorithm, there is a reference example $x$, and a set of positive examples $x^+$ and negative examples $x^-$. The learning algorithm is a function $f$ that maps input examples into some abstract feature space, and we want $f(x)$ and $f(x^+)$ to be similar, while $f(x)$ and $f(x^-)$ should be different, i.e.:

$$Score(f(x), f(x^+)) >> Score(f(x), f(x^-)) \tag{13.18}$$

### 13.6.1 Siamese Networks

One of the simplest contrastive learning architectures is the so-called Siamese network, which is essentially a standard convolutional neural network that takes two inputs simultaneously, generating feature vectors from them. These feature vectors are then compared (for example, by calculating their Euclidean distance), and a sigmoid non-linearity is applied to create a classification task. In this case, negative examples (large Euclidean distance) will typically yield values close to 1, while positive examples (small Euclidean distance) will yield values close to 0.5, which makes training relatively straightforward.

A significant problem with this architecture, however, is that comparing two random images provides very little information to the network about what should be similar or different. Therefore, it is common to use the so-called triplet loss, where instead of two images, three images are provided to the network: a reference, a positive, and a negative example. This ensures that in each iteration, the network undergoes both types of training: it is told what should be similar and what should be different.

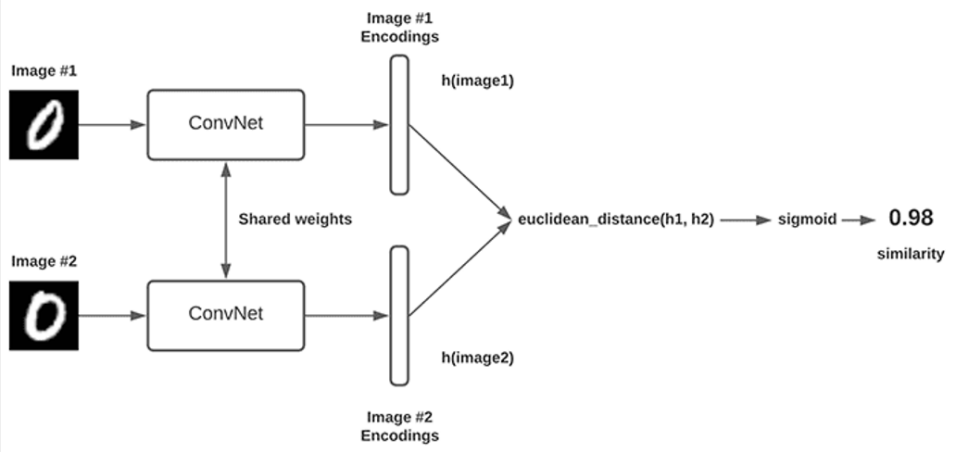


**Figure 13.15:** *The principle of Siamese Networks.*

## 13.7 Few-Shot Learning

Finally, it is worth noting that while one important advantage of generative methods over contrastive methods is that hard negative examples do not pose a problem, contrastive learning has several advantages. One of these advantages was mentioned in the introduction: contrastive networks are explicitly driven to learn useful representations, whereas in other cases, we only assume that the learned features generalize well.

Another advantage is that contrastively learned features can be easily used to achieve a capability with neural networks that traditional supervised learning algorithms struggle with. This capability is known as few-shot learning, which is the ability to recognize a new – previously unseen – class from just a few labeled examples.

Few-shot learning has several sub-cases. In the most challenging case, known as zero-shot learning, the network is not given any examples from the new class but must adapt to it without any training. In one-shot learning, the network receives exactly one labeled image. More generally, few-shot learning algorithms learn about the new class from a few (but more than one) labeled images.

One of the simplest methods for few-shot learning is the so-called Matching Networks, which uses a contrastively trained encoder to generate features from the query image that needs to be classified. There is also a key dataset containing a few images used for training, and their features are also extracted using the encoder. The features of the query image are then compared with the features of the key set, and classification is performed based on the similarities (e.g., by majority voting or weighted voting).

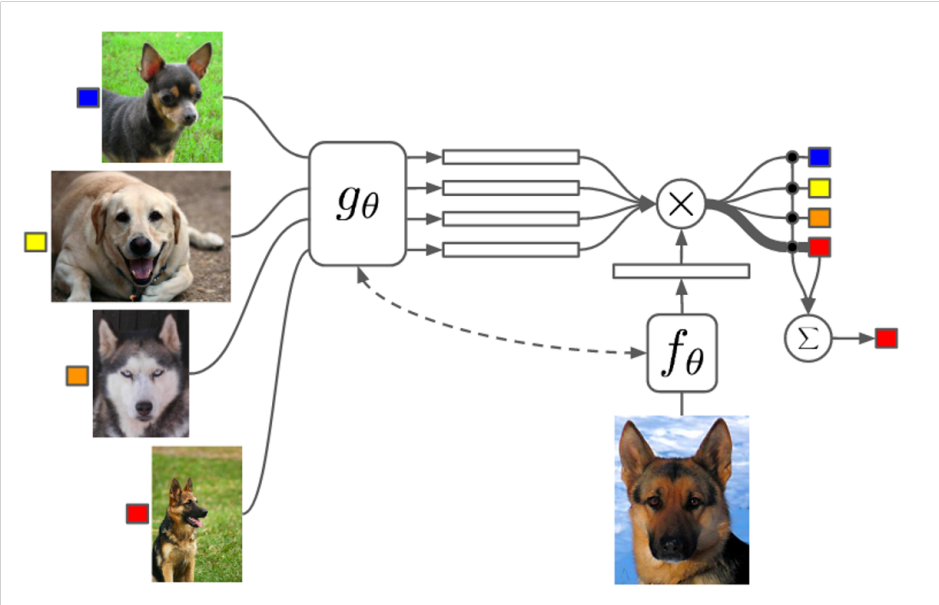


**Figure 13.16:** *The principle of the Matching Network method.*