Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Computer Vision Systems

Notes

Márton Szemenyei

February 23, 2023

Contents

Part I

Traditional Vision

1 Introduction to Computer Vision

1.1 What is computer vision?

Computer vision systems can be considered widespread in several walks of life. This is mainly due to the increased availability of computational power, more cost-efficient manufacturing technologies, and the advancement of image processing algorithms. These algorithms are used among others in robotics, process automation, virtual (VR) and augmented reality (AR) systems, not to mention their increased popularity in community service applications.

Computer vision, as a scientific domain, aims to extract relevant information from images or image sequences for decision-makers (individual or automated), with the help of algorithmic tools. The biggest challenge of the domain is that even one image consists of millions of pixels, moreover, the number of possible pixel configurations is several orders of magnitudes larger. For this reason, efficient algorithms and a high amount of computational power are needed to accomplish such tasks.

To further illustrate the difficulties of computer vision tasks, we should also consider that our mind, which is able to extract crucial information from only one image, carries out the vast majority of its processing tasks subconsciously. Thus, making vision the most important sense. Hence, an exact formulation, therefore, the development of algorithms like our vision system, is rather complicated. Furthermore, the human visual system also exploits the information gained with different senses, thus implementing an information fusion system. These complex aspects make it essential to use different heuristics, e.g., machine learning, numeric optimization methods, which are not guaranteed to perform equally well in all situations.

These algorithms can be categorized in several manners, one of the most widespread distinguishes between image processing and computer vision. Image processing aims to produce an image that is more advantageous for our purposes. Image processing is often used to prepare images for further analysis (i.e., more processing steps can be stacked after each other) or to help human users to recognize crucial details more easily.

On the other hand, computer vision has the goal to transform the input image to a higher abstraction level, thus to extract and represent its content in a more compact manner. To further differentiate, the term machine vision is used if these methods are applied in embedded systems (e.g., cars, robots, production machines, telephones, etc.). If the input is an image sequence or image stream, then it is called video analytics.

Due to recent results in machine learning-aided computer vision applications, these algorithms form a separate domain called learning vision. The most dynamically advancing branch of learning vision utilizes deep learning. Although the techniques not mentioned until now also represent very diverse approaches, they are called collectively traditional vision.

Traditional vision applications consist of four main steps, the first of which is the recording of images/image sequences. As a second step, there is - generally - an image enhancement stage, which serves as a means for reduction of the effects of speckles or noise. In this step, it is also possible to carry out image conversion, which can help the processing stage. In the third step, task-relevant features are extracted, i.e., such properties that can help solve the given problem. In the last step, called decision-making, the algorithm uses the available data to determine the final output.

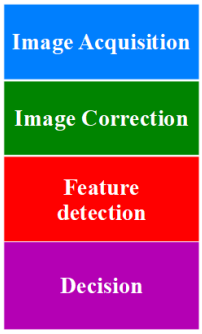

Figure 1.1: *The four steps of traditional vision.*

1.1.1 Essentials - tasks and pitfalls

The essential goal of computer vision is to extract high-level information from images. The most straightforward realization of this goal is, e.g., a classification task, where one label is assigned to each image, which describes the category of the depicted object. In some cases, not only a label but also a bounding box is assigned, this is called localization. In real situations, one image can contain different types of objects, and of course, several instances of them, i.e., in this case, each of them should be classified and localized, respectively. This task is called detection.

Special tasks can require even more fine-grained processing, e.g., besides the position of objects also orientation and shape information is needed. In this case, we can use the approach of semantic segmentation, which means a pixel-wise classification, which results in a mask where each pixel is assigned to a class. One of the difficulties emerges from contiguous objects of the same category, which can be solved, e.g., with the assignment of a secondary label (containing an object identifier besides the class label too), i.e., the task is now regarded as object segmentation.

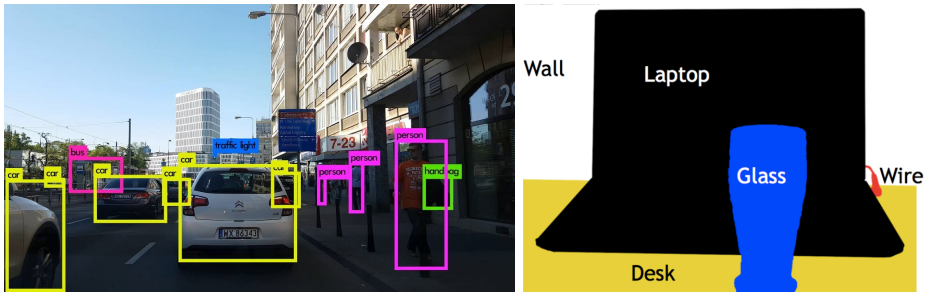

Figure 1.2: *Object detection and segmentation.*

Of course, the high-level detection of objects is not the final stage, it can also be managed to extract and organize the properties of objects (e.g., gender, age, sentiment), or even their connections (e.g., is-part-of, geometric relationships, activities). I.e., a system can be realized which can support well-founded decisions, based on visual information, this is called situation interpretation.

If the goal is the extraction of high-level semantic information, there are several difficulties to face. First of all, the image of the same object will be significantly different if the illumination changes, i.e., the pixel values will be not even close to each other (concerning multiple images of the same object). On the other hand, transformations like rotations, scaling, perspective distortion also contribute to significant changes. The property of some object to be able to alter its shape (e.g., reversible and irreversible deformations) will also result in different descriptors in image space. Real tasks often are supposed to manage scenarios where objects can be (partially) hidden, i.e., only a part of the object can be used for detection.

In the case of object classification and detection, our goal is not restricted to an object, we intend to determine a semantic class. The range of the objects belonging to the same class can be rather

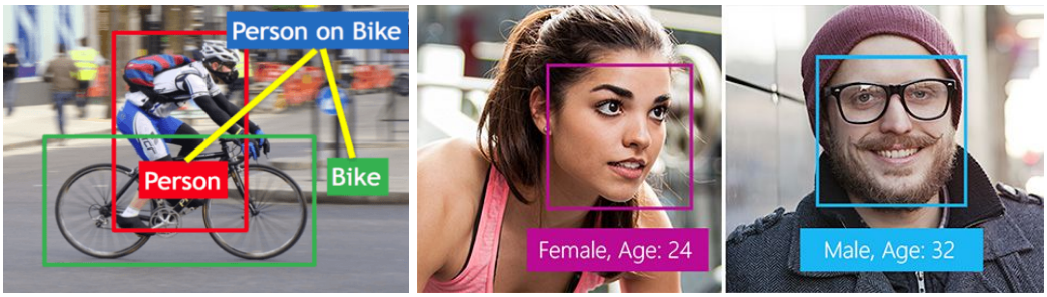

Figure 1.3: *Object relationships (left), and face detection combined with regression for determining age (right).*

broad, i.e., even substantial differences are imaginable between different instances. In practice, the situation is even more complex because often physical or functional properties determine whether an object belongs to a class or not. E.g., although serving the same purpose, there are several different realizations of a chair (shape, number of legs, color, material, etc.). This phenomenon, the greatest pitfall of semantic classification, is called interclass variation.

The difficulties mentioned above are far from the whole picture, e.g., the so-called semantic gaps further complicate the situation. Semantic gaps describe the irreconcilable differences between the digital representation and human interpretation of an image. As an example, consider the fact that humans perceive an object (e.g., a banana) as its yellow color, curved shape, but for computers, it is only an array of pixels. This phenomenon de facto makes it impossible to formulate complex computer vision tasks as simple algorithms.

1.2 Imaging

The essential task of computer vision is the acquisition of images. Although cameras are considered as everyday appliances, the knowledge of their functioning and most important properties is unavoidable to be able to choose proper devices. This chapter discusses the most widely used sensor types and their differences.

The similarity between the human eye and the arrangement of digital cameras is without a doubt. The concept is the following: the human eye consists of several optical subsystems that direct parallel rays to the point of clear vision, called the fovea, which is a spot on the retina, a light-sensitive layer at the back of the eye. To reach the retina, each ray travels through different media. First, the cornea refracts the light rays that are directed through the pupil, which is an opening into the cavity holding the crystalline lens and behind that the vitreous humor. Thus, rays get focused, resulting in sharp images.

The retina is covered with two types of receptive cells, which make it possible to transduce light signals into electric ones, thus providing interpretable input for the central nervous system. Rods are only able to detect light intensity, having the same sensitivity in the whole visible spectrum. On the other hand, cones are the receptors for colored vision, they sense either red, green or blue, i.e. they are frequency-selective.

1.2.1 Sensor types

The structure of cameras is analogous to that of the human vision system: their geometry is constructed to focus incident rays to a sensor array, the measured values of which compose the image. The geometric structure of cameras will be discussed in more detail in one of the following lectures. Besides geometric construction, the type of the photovoltaic sensor in the sensor array is also significant. Light-sensing is generally done with the help of photodiodes, which operate based on the principle of the photoelectric effect, i.e., if the P-N junction is targeted with photons (given their energy is above a threshold), an electron-hole pair is generated. If that phenomenon

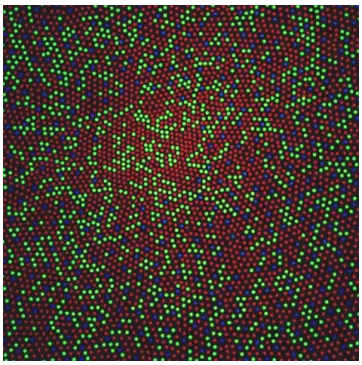

Figure 1.4: *Cones on the retina.*

happens in the depletion region of the diode, then the electric field moves them away, thus current is generated.

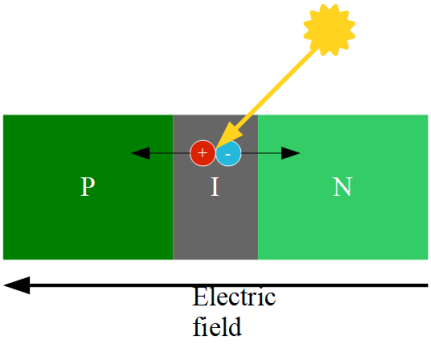

Figure 1.5: *The working principle of a photodiode.*

The two most widely used photosensors are the CCD (charge-coupled device) and the active CMOS (complementary metal-oxide semiconductor) sensors. The elements of a CCD are analog devices, which can store charge in the case a photon is absorbed. After the exposition time is over, the outermost cell in each row transmits its charge to the output amplifier, and then the cells shift their charge outwards. I.e., the reading process of a CCD is done row by row, and in each row element by element.

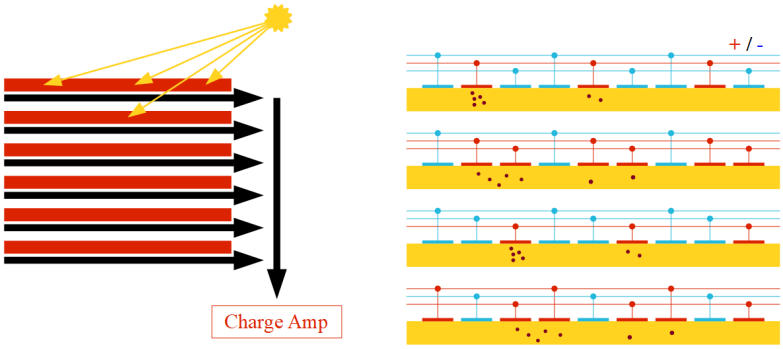

Figure 1.6: *The operation principle of a CCD sensor.*

In the case of CMOS devices, each cell has its separate amplifier, i.e., the readout process of the sensor array is faster. The tradeoff we face is that amplifiers also require space on the silicon; this means that fewer photons can be captured. To solve this problem, microlenses are mounted onto each cell, which direct photons - that would target the amplifier originally - to the photosensitive cells. Because CMOS sensors have smaller dimensions, less energy consumption, and not least are

cheaper, they are dominating the market of digital cameras, while CCD sensors can be found in high-end camcorders.

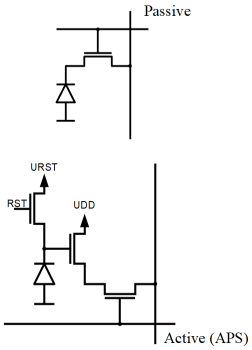

Figure 1.7: *The structure of passive and active CMOS sensors.*

The cheapest and most widely-used solution for color sensing in cameras is the Bayer-filter, which is an array with color-selective elements. This means that the parts are only for specific colors transmissive. If we use a Bayer-filter in front of the sensor array, then the CCD or CMOS sensors can also just register these colors. Since the human eye is more sensitive to the green color, the Bayer-filter contains twice as much green- as red- or blue-transmissive elements.

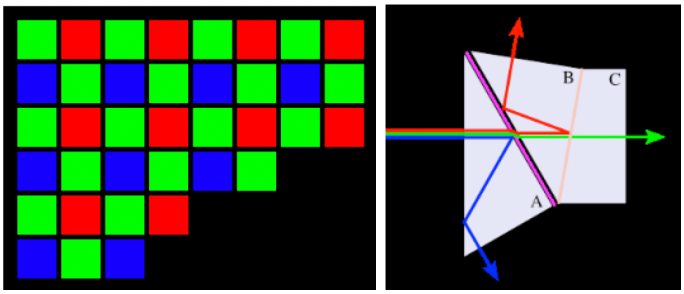

Figure 1.8: *The Bayer-filter (left) and the 3CCD (right).*

An alternative to the Bayer-filter, the 3CCD sensor uses a prism to direct each color component to a separate sensor array, which results in more sharp color differentiation. In low illumination the advantage of the three different sensor arrays is also apparent, of course, this technology has a higher price.

In recent years, different types of special sensors have spread, which contain not only color information, but also the distance of each pixel from the sensor. These devices are called depth cameras.

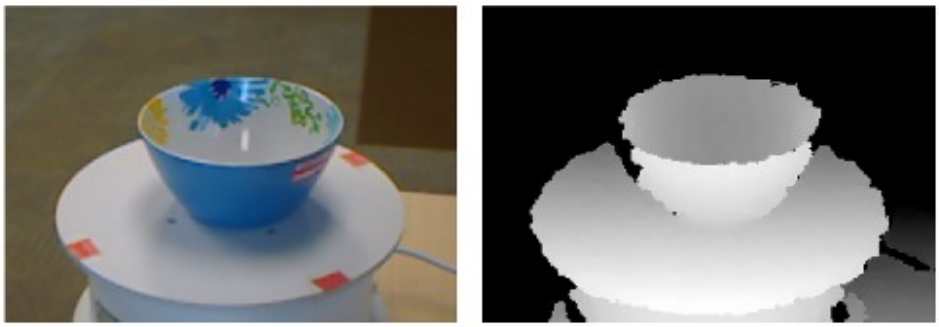

Figure 1.9: *Color and depth image.*

These cameras have two types: the first of which is called stereo. Stereo cameras consist of two

separate cameras mounted in a common chassis, the distance of them is fixed. The distance of the pixels can be calculated with the help of image correspondence between the two devices. These cameras are calibrated generally during the fabrication process; the calculations are carried out on the embedded processing unit.

On the other hand, infrared-based depth cameras have three parts: an RGB-sensor, an infrared projector, and an infrared sensor. The operating principle is the following: a specific pattern is projected in the - for humans invisible - infrared domain, which is perceived - including the distortions caused by the spatial structure of the observed objects - by the infrared sensor. Because the distortion stores information about the distance of the objects, the depth image can be calculated. The infrared-based depth cameras are more widespread due to their more accurate results and less need for data processing. Nevertheless, other infrared sources can disturb their operation.

There is a third type of sensor, called LIDAR. Similar to radar technology, LIDAR also infers the distance of objects from the delay and frequency of electromagnetic waves, the main difference is that LIDAR uses laser instead of radio waves. Due to the significant advancements of the response time of digital processing units, nowadays it is possible to measure distances in the order of cm.

1.3 Image properties

After readout, the image is represented as a two-dimensional array of numbers, the elements of which describe the light intensity at that specific position - these elements are called pixels. Generally, pixels are represented with an 8-bit number, where 0 stands for black and 255 for white. In the case of color images each position is represented with three numbers (red, green and blue, abbreviated as RGB).

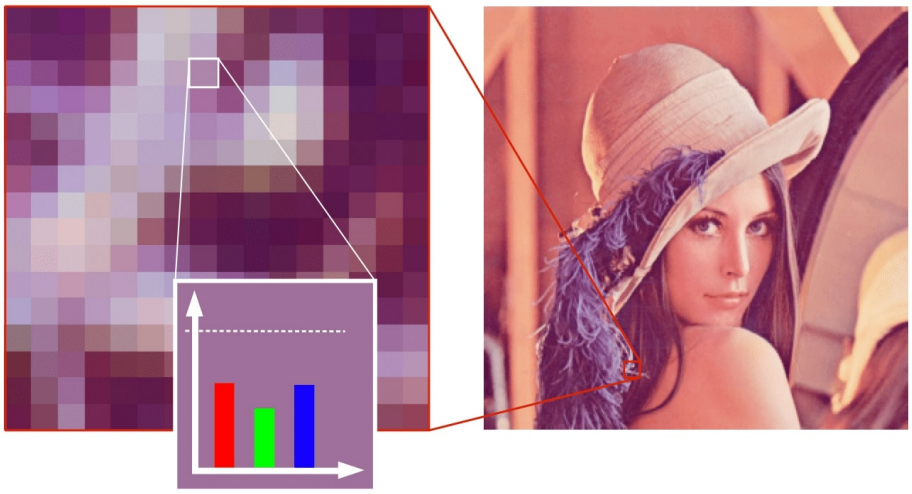

Figure 1.10: *The structure of an image.*

Digital images have numerous important parameters. The resolution determines the dimensions of the number array, increasing it results in a more detailed image - but the storage space required is also bigger. On the other hand, bit depth describes the number of bits, which are used to represent the intensity value of a pixel (for color images, it refers to each channel separately). Most often, 8 bits are used, but floating-point representations require 32 bits, while some compression methods can use less than 8 bits per pixel. In the special case, when the bit depth is 1, the images are called binary.

1.4 Compression and storage

In the case of storing images, a problem may arise, namely storage availability. The size of an average image taken with a mobile phone contains information in the order of several dozens of

Figure 1.11: *The effects of decreasing resolution and bit depth.*

MB; for FullHD videos, it is hundreds of GB (or even TB). This enormous amount of information cannot be efficiently stored given the capacity of (commercially) available hard drives. To handle this problem, it is essential to discuss the topic of data compression, for - based on the characteristics of human vision - it is, e.g., possible to store one pixel on 2 bytes in a way, that the bit depth assigned to each channel is compressed in a different way.

In the case of images, there exist several formats, one of the most essential is BMP, which is an uncompressed format. There are lossless compression methods, e.g., PNG - which is a raster-graphic format - uses such an algorithm; it is used first of all for the compression of text and figures. The formats EPS and SVG are also worth to mention because they use vector graphic representation, meaning they can be scaled basically without loss of resolution.

The most widely used lossy compression formats are JPEG and MPEG, for images and videos, respectively - these formats were typically created for real-world data. For videos, the H.264 and H.265 formats are also often used; they are the newer versions of the MPEG compression method.

1.5 Description of color information

Computer vision often utilizes besides the intensity information found in grayscale images the additional information of color images. Several simple detection algorithms search for similarities in the color space. Not to mention, many problems can arise, e.g., we need to ensure that the pixel values are able to represent the similarity of colors - a prerequisite for a reliable, robust implementation of color-similarity-based search methods.

The most commonly used color representation in cameras (RGB) is not suitable for that task. It is due to the fact that the geometric distance of color-descriptor points in the RGB color space is not expressive regarding the similarity of the colors as perceived by humans. On the other hand, the change in illumination modifies all values, although the color of the original object remains unchanged.

Several transformations exist, which were aimed to overcome this shortage of the RGB representation, i.e., they were designed to describe - without loss of information - color-similarity and to be robust for the change of illumination. These transformations (often given by a transformation matrix) also define a new color space.

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

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

The most common conversion is the process of making a grayscale image from a colored one. In that case, all three channels are combined to create one intensity value. The most often, the luma (Y), intensity (I), value (V), and luminance (L) transforms are used, which are described 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 widely used is the YCbCr-family, which consists of many, slightly different variants. The Y channel from the three is the separated luminance component, which represents the lightness of the given color. The Cr and Cb channels describe the hue of the color. This color space is often used in digital video systems, or in the JPEG and MPEG standards. It is important to note, that the YCbCr color space is different from the YUV color space, which has a similar approach, but is used in analog systems.

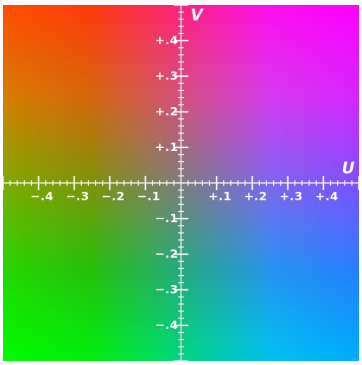

Figure 1.12: *The UV plane of the LUV color space.*

The YCbCr transformation matrix is the following:

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

Another popular color space family is the HSV/HSI/HSL-family. The common in these color representations is that the chrominance is coded with a hue and saturation value. They differ mainly in their representation of lightness/intensity. They can be easily represented in the form of a cone or cylinder. Both alternative color spaces are often used in color-based processing methods.

The transform into the HSV color space is non-linear, i.e., it cannot be described with one matrix. The transformation rules are the following:

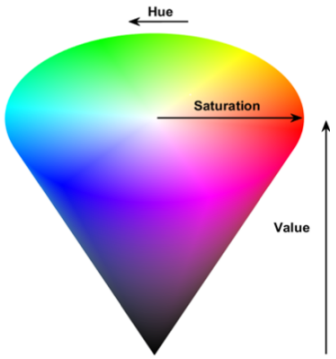

Figure 1.13: *The HSV color space represented with a cone.*

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

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

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

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

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

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

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

Although the calculation of the saturation and hue values seems to be complicated, the principle is rather straightforward: the saturation is proportional to the difference between the most significant and the least significant color components. After that, the three components divide a circle into three equal parts: red is at 0, green at 120, and blue at 240 degrees. Last, the two least significant color components are ”pulling” the actual color into their direction.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, “The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover),” *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

2 Image enhancement and filtering

2.1 Intensity transformations

One of the simplest methods of image enhancement is the application of intensity transformations. Using such methods, the given transform is applied to each pixel. In most cases, the new value of a pixel is only dependent on its old value (i.e., it does not generally depend on the intensity of neighboring pixels).

These transforms have different variants: if a constant is added to the pixel values, the lightness of the image can be changed (increased or decreased, depending on the sign of the constant). Of course, it is also possible to multiply the pixels with a constant, which increases/decreases the contrast of the image - in that case, a constant is also added after the multiplication step to center the new intensity values. It is important to note that these transforms assume that pixel values will be saturated, i.e., clamped to the range of 0-255 in the case of under- or overflow.

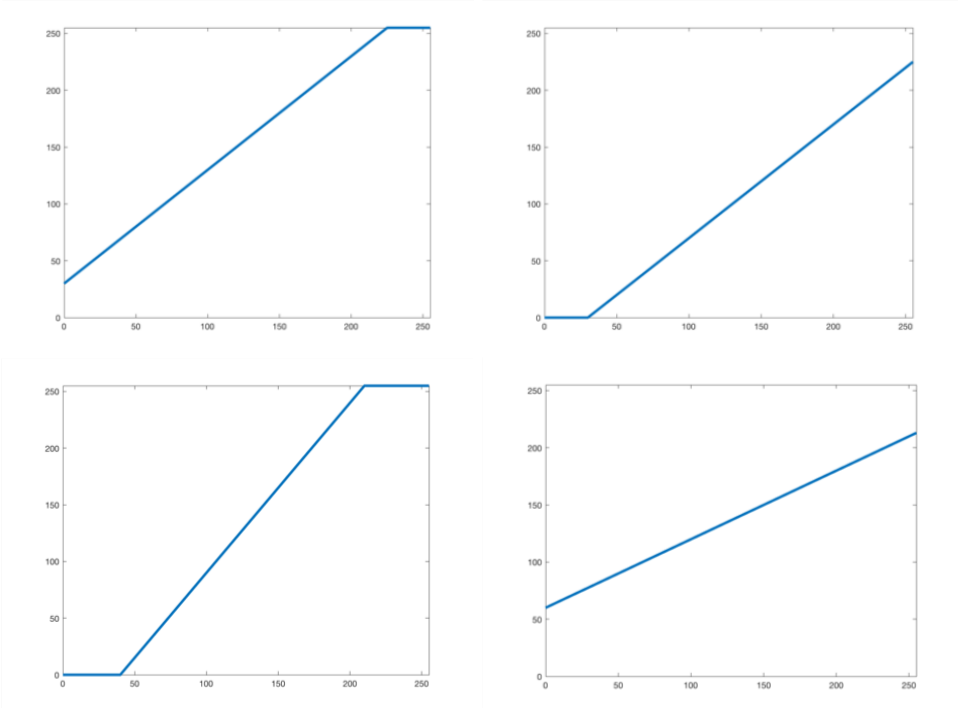

Figure 2.1: *Intensity transformations: increasing/decreasing brightness (above), and increasing/decreasing contrast (below).*

A particular case of intensity transformations is thresholding, which means that pixels with intensity values below a given limit are clamped to 0; on the other hand, values above the threshold are set to 1 - i.e., the result is a binary image. If we use two thresholds, then either value inside or outside the specified interval is set to 1. One main advantage is that pixels with a given intensity can be highlighted, and based on their quantity, size, or position, conclusions can be made. An important note is that predefined threshold values give different results if illumination changes. To avoid that, in practical problems often adaptive - i.e., calculated based on the intensity distribution of the image - thresholds are used.

Figure 2.2: *The result of thresholding.*

Intensity transformations can also be used on color images; in that case, they are applied to each channel independently - which also means that it is possible to use different transforms for different channels. This approach can help to change the relative dominance or the relationship of the colors in the image. In the case of thresholding, it is also possible to highlight specific colors, which can enable the detection of an object with a given color.

2.2 Histogram

Although intensity transforms are simple and fast methods, their robustness leaves much to be desired. Due to this fact, often histogram-based transformations are used. The histogram describes the relative occurrence (frequency) of intensity values in an image or image channel - i.e., the histogram is the empirical probability density function. The histogram can help us to detect and correct the defects of image acquisition (first of all that of illumination or exposure).

Underexposed images were recorded with too fast exposure times, i.e., it was not enough time for the photons to illuminate the pixels; this means that the histogram will have a positive skew (the values are concentrated in the lower part), i.e., the details will be only hard to see. In the case of overexposed images the phenomenon is similar, only this time dark pixels will be too bright (the values are concentrated in the upper part of the histogram).

This kind of problem is handled with the histogram equalization algorithm, for the histogram of under- and overexposed images has a significant deviation compared to the uniform distribution. This algorithm aims to provide a better approximation for the uniform distribution - for that infrequent intensity values are merged, and frequent ones moved. Unfortunately, due to the merging, some information is lost, so this algorithm is only used to achieve better visibility properties (for humans, further processing disregards this step).

2.3 Image noise, noise types

Capturing images with real (i.e., not ideal) devices means that noise and speckles will be present, which makes image processing tasks more challenging. Having different sources, the types of noise present can also be diverse, the most common one is the Gaussian-noise, which is the consequence of the noisy nature of the imaging sensor and the surrounding electronics. Gaussian-noise is typically additive and also pixelwise independent.

Other common noise type is the so-called salt and pepper noise, which occurs sparsely, but changes the value of the pixels in a significant manner. I.e., in dark regions white spots, and in bright regions dark pixels can be observed. Salt and pepper noise is mainly due to errors in the analog-digital conversion process or during informationtransmission (bit errors). Other noise types worth mentioning are the quantization error - a consequence of analog-digital conversion - and the periodic noise, which is due to occasional electromagnetic disturbances.

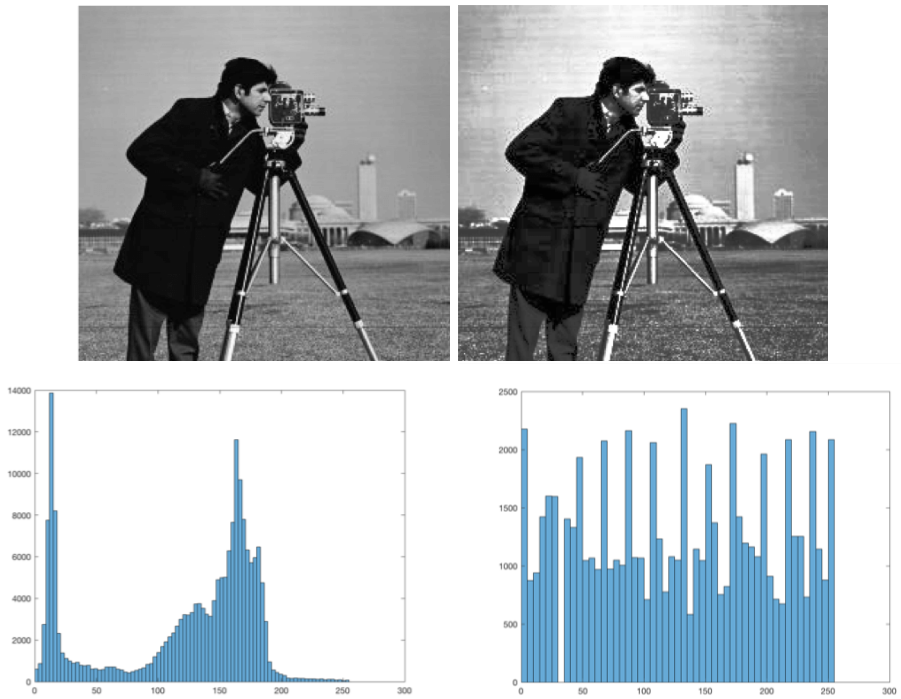

Figure 2.3: *Histogram equalization: the original image and its histogram (left) and the equalized image (right).*

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

2.4 Convolutional filtering

The most essential methods of image enhancement are the different filtering algorithms, which are intended to correct image errors and noise. They are based on convolutional filtering, i.e., a small filtering window (also called the convolutional kernel) is slid through the image, and the value of each pixel is set to the result of the convolution with the pixel itself and its neighborhood. The convolution is described with 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 in row y at position x. According to the formula, convolution is the weighted average (weighted with the filter weights) of the pixel in the neighborhood defined by the kernel. The filter weights are always chosen to sum up to one, otherwise, the whole image would become darker or lighter. Although the operation of convolution should be carried out in the reverse order (note the different directions of the filter and the image), but in practice, this is often

omitted. This means that we calculate not the convolution but the cross-correlation, nevertheless we call it convolution (both are the same only if the kernel is central symmetric).

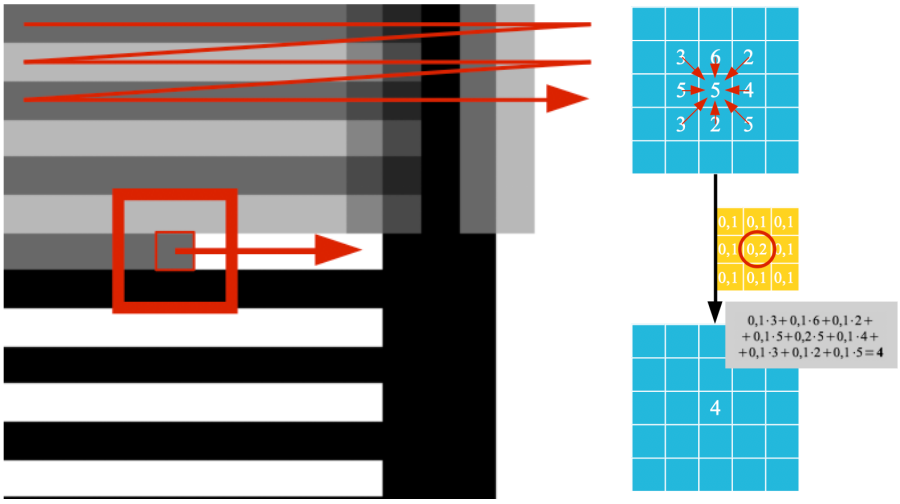

Figure 2.5: *The principle of convolutional filtering (left), and the convolutional operation at a specific position (right).*

2.4.1 Linear filters

The convolutional kernels used for noise filtering are called smoothing filters, e.g., the simplest variant of them is the averaging filter. A sophisticated version is the Gaussian-kernel, which weights farther pixels (from the center of the kernel) less, and weights are determined according to a Gauss function. Changing the standard deviation of the bell-shaped curve has a direct impact on the smoothness - it is also possible to specify different values in each direction and so creating different effects in the x- and y-direction.

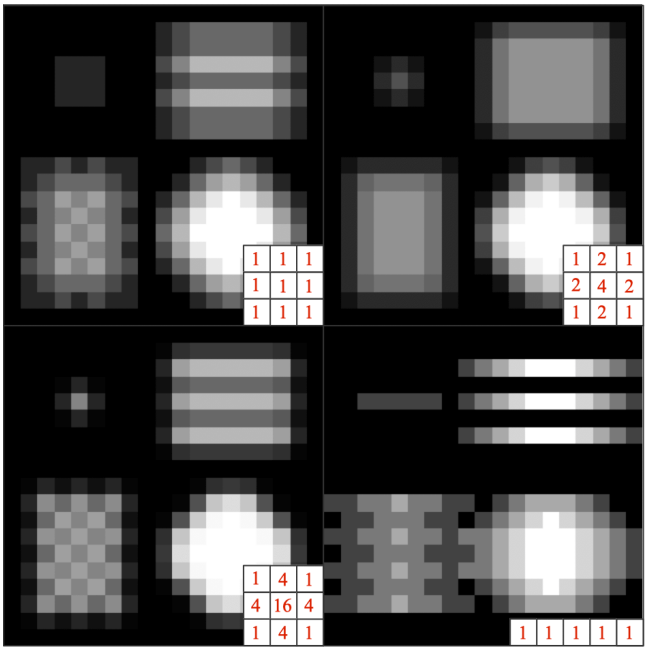

Figure 2.6: *Typical convolutional kernels: averaging (above left), Gauss with two different standard deviation values (above right, below left) and a horizontally averaging kernel (below right).*

In the case of smoothing filters, each element of the kernel is non-negative, and they sum up to one. If the sum differs, then a brightening/darkening step also occurs beside smoothing. Due to the linear nature of convolution, several operations can be merged, or in some cases, it is also possible to split the kernel into parts. This means that the 2D-filtering is carried out as two separate 1D convolutions, this results in significant savings in computational power, i.e., instead of N^2 only $2 * N$ operations are required - note that this is only possible if the rank of the kernel is one.

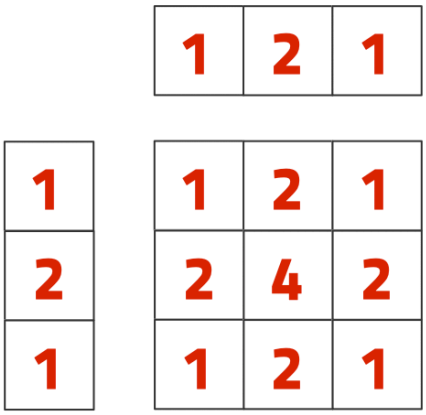

Figure 2.7: *Splitting up a 2D kernel into two 1D filters.*

There are two main problems regarding convolutional kernels: first, although with averaging noise can be reduced, but also some image details are blurred (mostly edges), resulting in unsharp images. On the other hand, due to the averaging step, significantly different values (salt and pepper noise) are also incorporated into the average, i.e., salt and pepper noise is not eliminated but smeared.

Figure 2.8: *Image with white noise before (left) and after (right) smoothing.*

2.4.2 Rank filters

The problems mentioned above can be solved by rank filters, which also take a small neighborhood of a specific pixel into consideration, but instead of convolution, they order the pixels based on their intensity values. According to the choice of this ranking (i.e., the new value of the pixel), maximum, minimum and median (most widely used for image processing) are distinguished.

Median filters substitute the pixel value with the median (i.e., the middle element) of the ranking. The main advantage of this filtering method is the fact that edges will remain untouched, but salt and pepper noise is filtered magnificently. The reason for which is that unlike the average, the median statistics is very robust regarding extreme values. Rank filters have a great computational disadvantage compared to convolutional filtering: the ranking operation is rather computationally intensive, so these filters are much slower. To make the situation worse, some high-level acceleration techniques are only applicable to convolutional kernels.

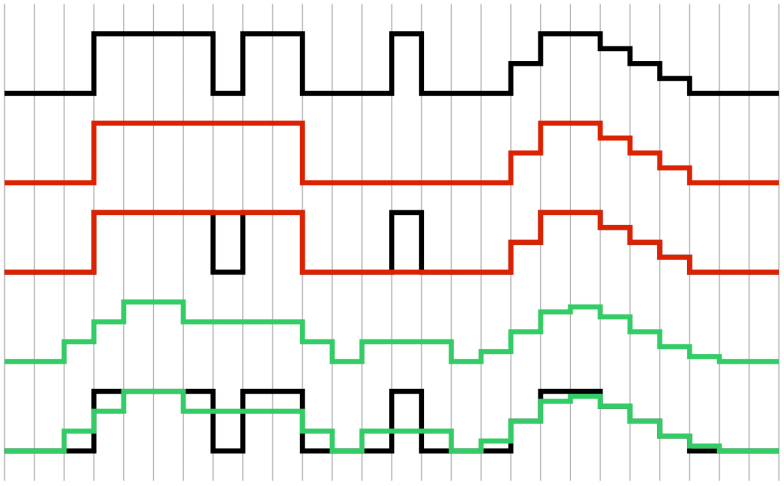

Figure 2.9: *The difference between the median (red) and averaging (green) filter.*

Figure 2.10: *Image with salt and pepper noise (left) and the result of the median filter (right).*

2.5 Sharpening, edge detection

The definition of an edge is a unidirectional, high-value intensity change between neighboring pixels. There must be only a change in one direction, in the other pixel values are constant, and that the change is sharp. Of course, due to image distortions, speckles, noise, and finite resolution, the transition will be smoothened and not in every case unidirectional.

2.5.1 Derivative-based edge detection

The most straightforward way to detect edges is to calculate the derivatives in each direction, which can be realized similarly in a numeric manner to that of convolutional filtering. In each position, two differences are calculated (x- and y-direction), one between the pixel and its right and one between the pixel and its below neighbor. The squared sum of both gives a metric that characterizes how much is the pixel edge-like.

Although being a fast and straightforward method, it suffers from some problems, e.g. due to random noise, the edge image will be noisy too, i.e., several false detections are included due to the random nature of noise. This can be avoided by applying a Gauss-filtering step, thus smoothing the noise. Since both differentiation and the Gauss-filtering are linear operations, they can be merged into one operation, i.e., the filtering step can be carried out with the DoG (Derivative of Gaussian) kernel, which means that this realization will be faster.

Besides the Gauss-kernel, other convolutional edge detection kernels are popular such as the Prewitt- and Sobel-operators, which are direction-dependent edge detectors, otherwise their working principle is similar to that of the Gaussian. To detect edges in every direction, both variants

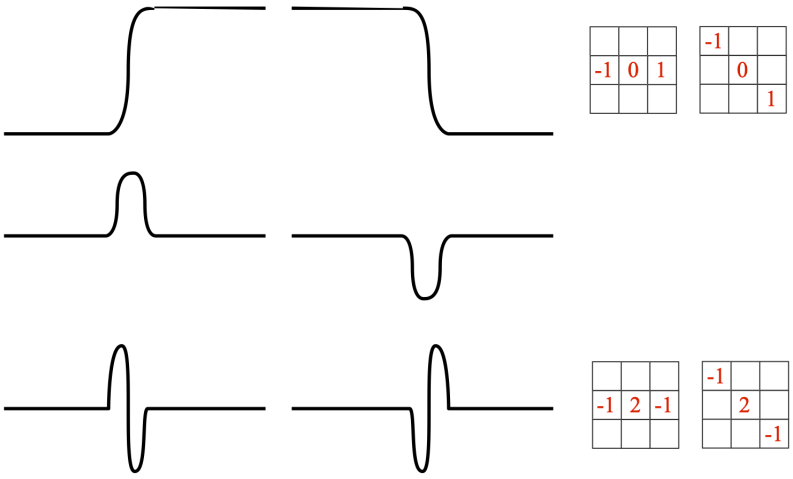

Figure 2.11: *The effect of derivative-based filters.*

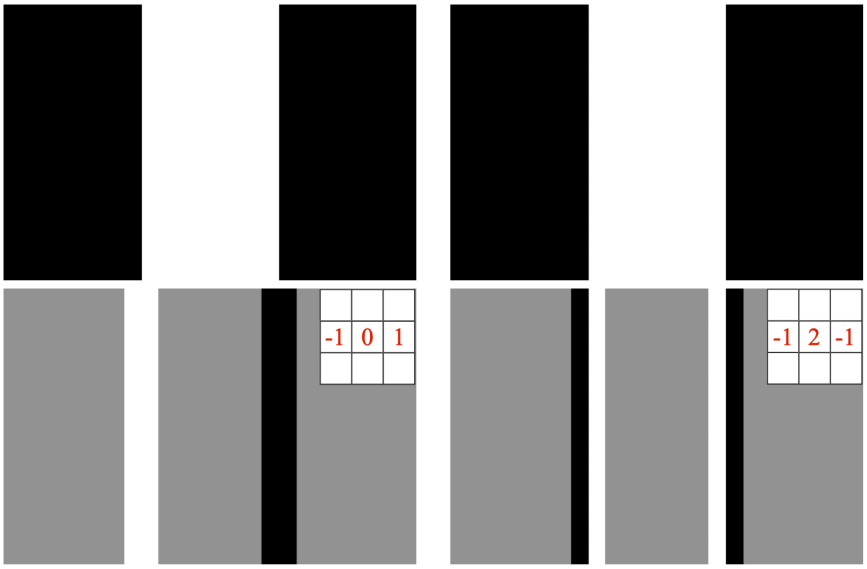

Figure 2.12: *First (left) and second (right) derivatives of an edge. In the case of the latter, the edge is located at the null transition.*

of the Prewitt- or Sobel-operators should be applied to the image. The main difference between them is the fact that the Sobel-operator also smoothens in the direction perpendicular to the edge, i.e., it is less sensitive to noise.

1	0	-1
1	0	-1
1	0	-1

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

Figure 2.13: *The effect of the Prewitt- (left) and Sobel-operators (right) on vertical edges.*

Using first-order derivatives for edge detection has the drawback that due to smoothing, the edge will be smeared, i.e., its localization will be difficult. To solve this problem, we can derivate the image and search for zero transitions - of course, this is carried out in one step, with a second-order derivate convolutional kernel called Laplace-filter (also called Sombrero-filter due to its shape).

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

-1	-1	-1
-1	8	-1
-1	-1	-1

Figure 2.14: *The Laplace-filter for 4-connected (left) and 8-connected (right) neighborhoods.*

In practice, the Laplace-kernel may be substituted with the difference of two Gaussian distributions (the variances are different), which is (also) called the DoG-filter (but here, it means Difference of Gaussians). If the filtering should be carried out with multiple DoG-kernels on the same image, then it is also an excellent trick to calculate the effect of each filter separately then subtract the images - again, due to linearity, the result will be the same.

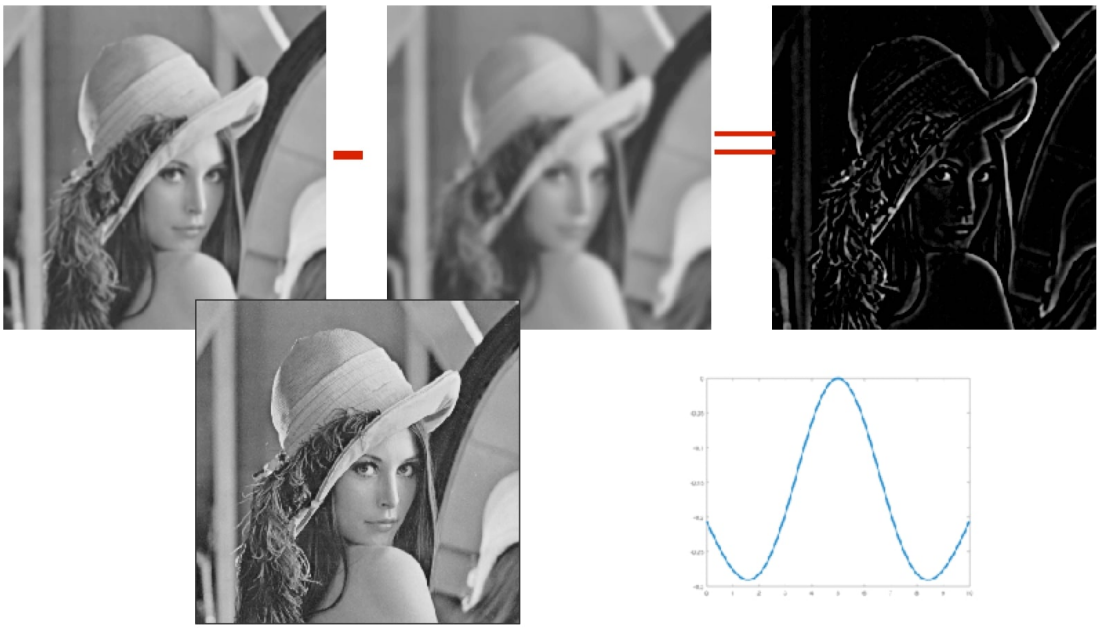

Figure 2.15: *The DoG principle, blue shows the shape of the filter in 1D.*

Application: in several situations it could come handy to automate the image acquisition about an object, e.g., if it cannot be controlled when and where will the object appear, or the number of images would be enormous, i.e., manual image capturing would not be an option. In those cases, it cannot be guaranteed that the object will be in focus, which can make further processing more difficult.

To avoid the problem, automatic focus detection can be used, which determines a quality measure of being focused from the image, and the focus will be modified until the maximum of that measure is nor found. Because in blurred images the majority of the edges are smeared, their gradients will also be smaller. I.e., a trivial measure for a focused image would be the number of pixels where the

norm of the gradient is above a threshold. Note, that this method is only capable of comparing images with identical content.

2.5.2 Sharpening filters

The coefficients/weights of smoothening filters sum up to 1, in the case of edge detector filters, the sum will be 0. There are such filters, which are similar to edge detector filters regarding the structure of their elements (i.e., negative and positive elements on different sides), but sum up to 1 - these filters are called sharpening filters. They are able to highlight fine details, changes, so making the sensation of the image sharper. Thanks to this, these filters are often used for photography applications.

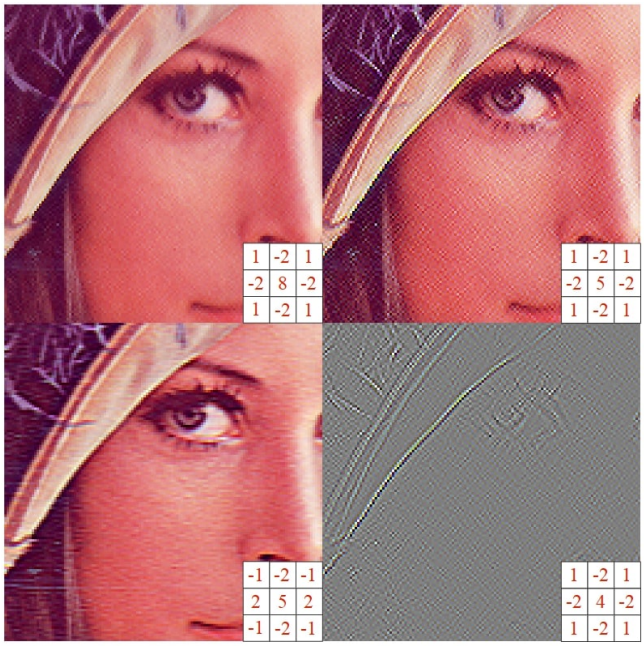

Figure 2.16: *Sharpening filters and an edge detector filter (below right), note the different sum of the weights.*

2.5.3 Canny algorithm

The edge detection algorithms discussed above are based on different derivative calculation methods and resulted in metrics that described how much a pixel could be considered to be part of an edge. From now on, it is the responsibility of the designer to decide the threshold value, above which a pixel is considered to be an edge. Choosing a too high value can result in not detecting lower-contrast parts of edges; on the other hand, a too-small value would result in the false detection of non-existing edges - the consequence of noise and other disturbances. The goal is to find the compromise - but that also will not be perfect.

The state-of-the-art edge detection algorithm, called the Canny algorithm, intends to solve this problem. It consists of multiple steps, the first of which is the calculation - with simple derivation filters - of vertical and horizontal derivatives, based on that information, the norm and the direction of the image gradient (biggest direction of intensity change) are calculated.

After that, starting from each edge-like point, following the direction of the gradient, the pixels with the biggest gradient value are determined. Those pixels are kept, while their neighbors with smaller gradient values are set to zero. After this step, the originally blurred edge image will be as sharp as it would have been calculated with second-order derivatives.

The next step is to threshold the image containing gradient norms, for which two separate threshold values are used, and thus two binary images will be the result. The image created with the smaller threshold probably contains all real edges, but several non-edge pixels will be marked as edges - this is the effect of noise. In the case of the image created with the help of the bigger threshold, not all real edge pixels will be contained, but the amount of noise will be also not so significant.

Figure 2.17: *Edge images used by the Canny algorithm.*

The main advantage of the Canny algorithm is that with the help of both binary images, the result will be a better edge image than the result of them separately. To achieve that, edge points are added to the image created with the higher threshold from the other made with the lower threshold if they are adjacent to the pixels found in the more sparse image. I.e., the Canny algorithm can utilize the different edge images to correct the deficiencies in the more sparse image, while noise not connected to real edges is disregarded.

2.6 The mathematics of an image

For images are two-dimensional image arrays, so they are equivalent to matrices, i.e., the powerful toolbox known from linear algebra can also be applied for images. Although some operations have not much sense (factorizations, matrix multiplication), others are useful, e.g., intensity transformations can be realized with scalar operations: addition changes brightness, while multiplication the contrast.

Using images with equal sizes, we also have several possibilities. The addition (rather averaging) of images can be used to blend images, while element-wise multiplication results in texturing. Nevertheless, one of the most useful operations is the subtraction of two images, because so the differences between the images can be highlighted, which can make several processing tasks easier. This is often used for movement detection or for the separation of static backgrounds.

2.7 Interpolation methods

During image processing, it often occurs that the image is not available in the format required. Often, we would like to have an image with higher resolution; to achieve that, we should be able to determine the values of the new pixels - without extra information. I.e., we can only rely on the old pixel values and our assumptions (e.g., continuity, smoothness of the image). If geometric transformations should be carried out, the situation is the same, because the pixel grid after the transform will not perfectly match the original one.

To determine the new pixel values, interpolation methods are used. They are based on the approach that given known pixels the image is approximated with a defined (continuous) function, which is

Figure 2.18: *Texturing.*

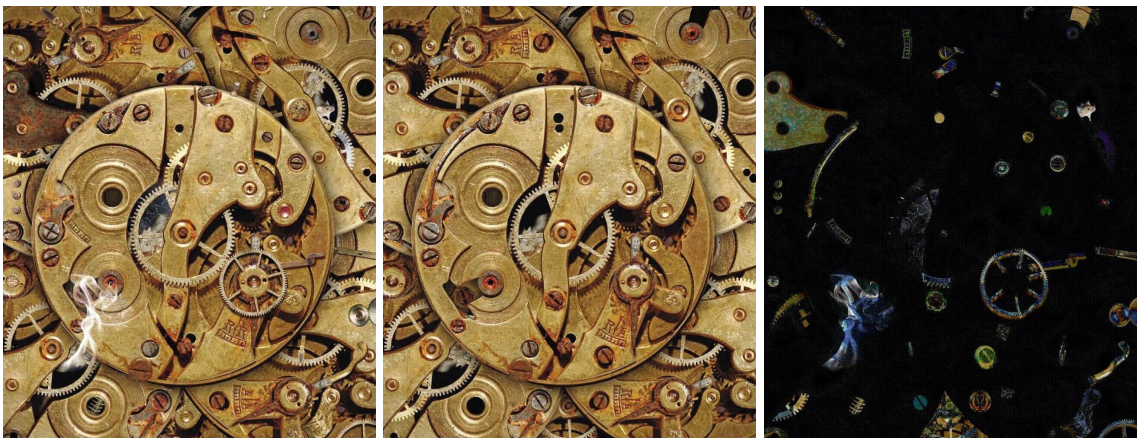

Figure 2.19: *The difference of two images.*

then sampled to determine the values of the new, unknown pixels. An important remark is that with interpolation, new information is not created.

There are several different interpolation techniques; they differ in the set of functions used. The most straightforward approach is the nearest neighbor approximation, where the image signal is approximated with a piecewise continuous function. In that case, the value of an unknown pixel will be equal to the value of its nearest neighbor. Slightly more complicated is the linear interpolation method, where the approximation is done with a piecewise linear function so that the new value will be the weighted average of the two neighbors. Of course, the order of the polynomial can be further increased, e.g., for cubic interpolation, the four nearest neighbors are used to construct a third-order polynomial.

Because images are two-dimensional, the interpolation should also be extended to two dimensions; fortunately, the methods mentioned above can be extended to be applicable for images. Nevertheless, the number of needed neighbors grows exponentially with the number of dimensions. As in the one-dimensional case, for images also the nearest neighbor method is the simplest, but it is only a feasible choice for artificial images (printed text, geometric shapes) or matrices that are not images.

One of the most widely-used interpolation methods for images is the bilinear interpolation, which is the extension of the linear interpolation. Bilinear interpolation proceeds as follows: first, two 1D

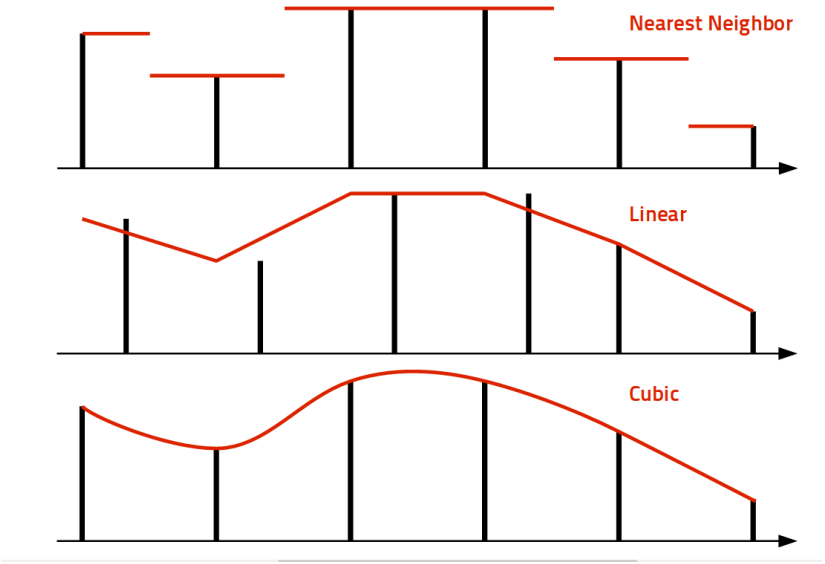

Figure 2.20: *1D interploation methods.*

interpolations are done in one direction, the result of which are two points (one of their coordinates is equal to that of the point we are looking for). The second step is to interpolate between those temporary points in the other direction, thus we get the value of the pixel. This method has two main advantages: first, it is simple; second, due to its linear nature, it cannot cause overshoot. The drawback of this technique is, nevertheless, that the human eye sees its result as somewhat blurry, and the gradient of intensity is also changed radically, thus reducing the smoothness of the image. Bilinear interpolation can be calculated as follows:

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

Of course, other, e.g., cubic interpolation techniques can also be expanded to two dimensions, but for that, we need 16 (instead of 4) pixels; this is the so-called bicubic interpolation. The third-order surface used for approximation can be described with the following formula:

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

Where a_{ij} denotes the polynomial coefficients. To obtain all of the 16 coefficients, for each pixel in the 2x2 neighborhood of the pixel to be interpolated, 4 equations are determined. On the one hand, the intensity value in each pixel is prescribed; on the other hand, the partial intensity derivatives in both x- and y-directions are also prescribed. In the end, the equation for the partial derivative with the mixed directions (i.e., for x and y) is determined; thus, we have 16 equations for the same number of parameters. Generally, they have the form stated below:

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

Note that the derivatives are generally not known from beforehand but can be determined from the intensity values of the pixels as follows:

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

The main advantage of bicubic interpolation is that the result seems sharper than in the case of bilinear interpolation, and the smoothness of the image also will be retained, for derivatives on the image borders are prescribed. Nevertheless, bicubic interpolation can result in overshooting, i.e., ring-like errors can be observed near edges.

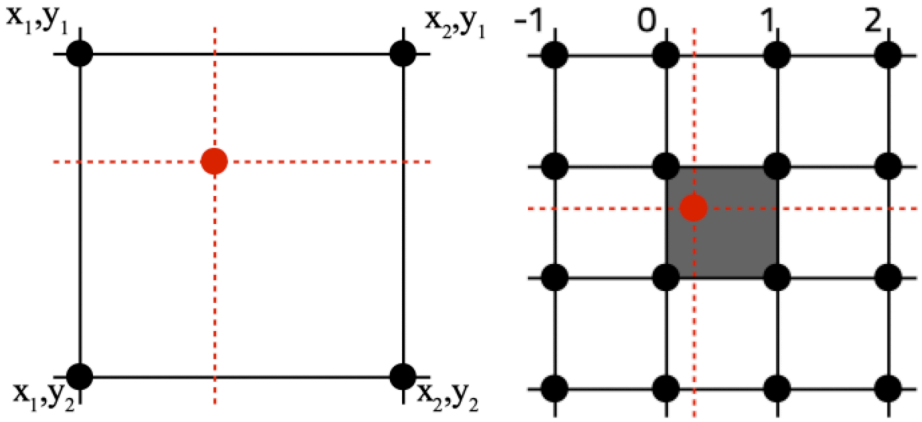

Figure 2.21: *Bilinear (left) and bicubic (right) interpolation.*

A great compromise is the so-called Lanczos-interpolation, which uses the weighted average of the pixel neighborhood (the kernel is called the Lanczos-kernel) for determining the new pixel value. The result of this method also has a good sensation of sharpness but minimal overshoot.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

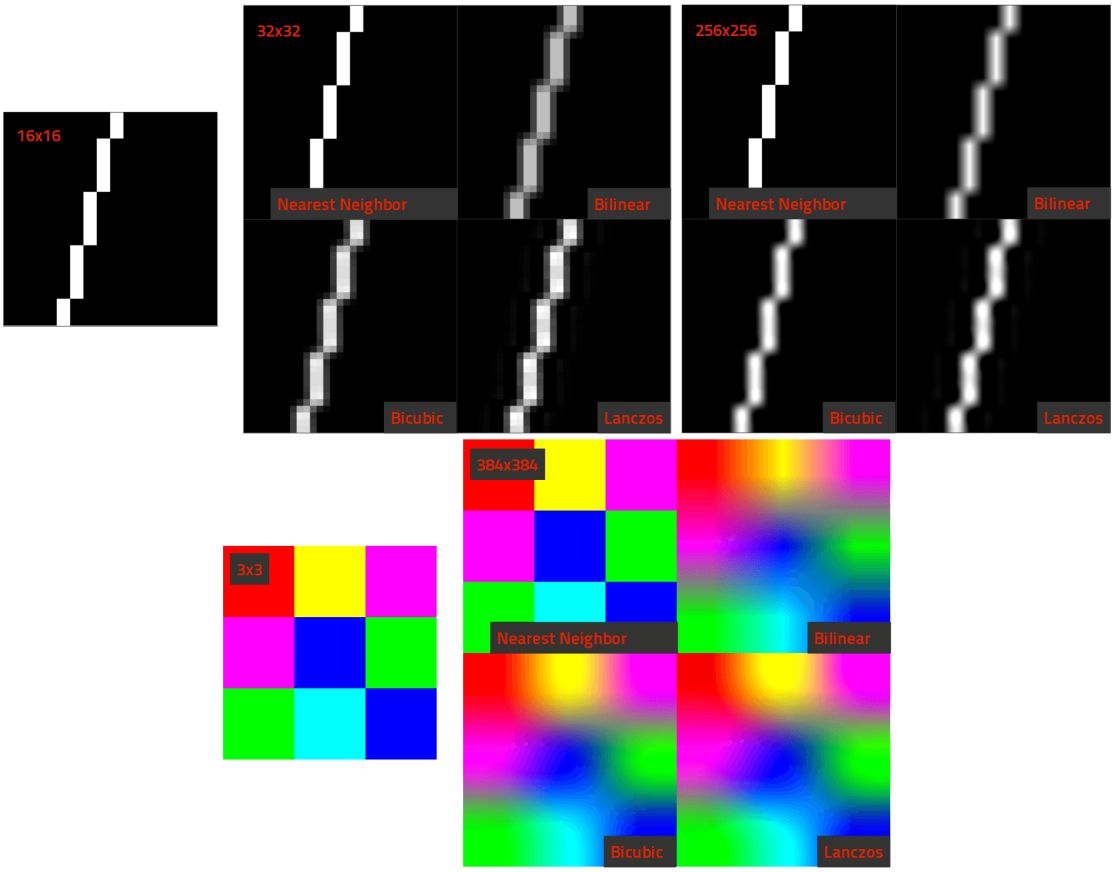

Figure 2.22: *The result of different interpolation techniques with different scale factors.*

[3] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986. DOI: 10.1109/tpami.1986.4767851. [Online]. Available: https://doi.org/10.1109%2Ftpami.1986.4767851.

3 Image processing in the frequency domain

3.1 Introduction

It seems obvious to plot functions of space or time w.r.t. these variables, nevertheless, this is not the only way to represent them. The theorem of the Fourier series states that each periodic function can be decomposed into the sum of sines and cosines of different frequencies, where each frequency has its own amplitude and phase coefficients. These are called the spectrum of the signal and can be represented as a complex number. Displaying an image w.r.t. to the Fourier spectrum is called the frequency-domain representation. The formula of the Fourier series is stated below:

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

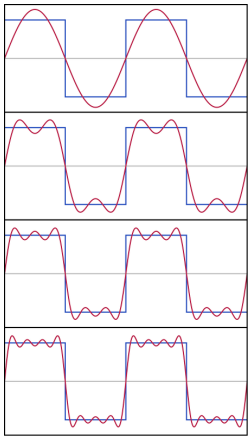

Figure 3.1: *The decomposition of a square wave into the sum of sines.*

3.2 Fourier transform

The element of the Fourier series with the smallest frequency is called the first harmonics, and its frequency the fundamental frequency has the frequency of the original signal. The frequency of further harmonics is an integer multiple of the fundamental frequency. So if the period of the original signal goes to infinity (i.e., the function is aperiodic), then its spectrum will be a continuous function, which is called the Fourier transform of the signal. The Fourier transform can also be carried out on sampled (i.e., discrete) signals; in this case, the result will be periodic, i.e., above a specific frequency it does not contain additional information. The reason for this is that the sampled signal cannot change arbitrarily fast.

In the scientific domain of computer vision, a two-dimensional image is often interpreted as a function of the two dimensions of the image plane, which is determined with the x (horizontal) and y (vertical) coordinates. The image is defined in this plane as the set of discrete impulses, the position of which are determined - the value of the impulses is given by the pixel values. Fortunately, the Fourier transform can be extended to arbitrary (so for two too) dimensions in the given way:

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

Where $I(x,y)$ is the pixel value in row y and column x, u and v are the horizontal and vertical frequency components, respectively, while H and W the height and width of the image.

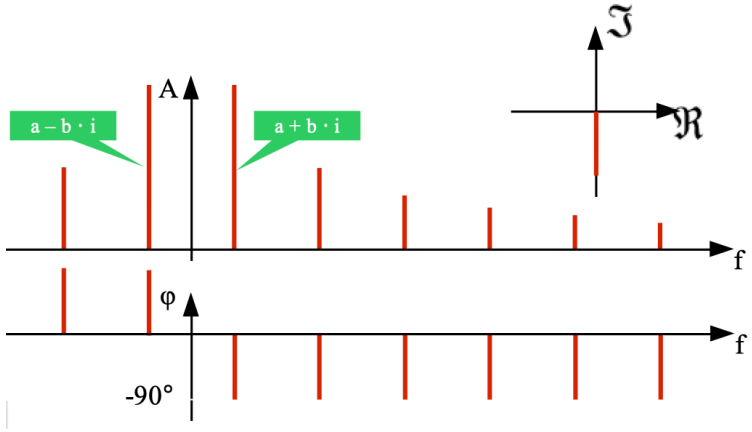

Figure 3.2: *Amplitude and phase spectra in the case of the Discrete Fourier Transform.*

Since images are two-dimensional, the components (i.e., periodic signals) have not only a specific frequency and phase, but also a direction, so the Fourier transform is also two-dimensional. I.e., the displayed amplitude spectrum can help us to determine - besides the frequency values - the direction of the dominant components.

As mentioned above, the Fourier transform of periodic signals is discrete, in the case of discrete functions, it will be periodic. The latter is a useful property because - as images are discrete - only one period of the image spectra should be stored, i.e., without the loss of information, it is possible to transform images to and from the frequency domain. Nevertheless, computers can only store the spectrum as a series of discrete values, i.e., all operations in the frequency domain will assume that the image repeats itself outside its borders. This behavior changes how - theoretically equivalent - algorithms perform depending on whether they are applied in the spatial or the frequency domain.

3.2.1 Phase distortion

It is essential to note that in the case of frequency-domain analysis, the phase spectrum is quite neglected compared to the amplitude spectrum because the physical interpretation of the latter is simpler. Nevertheless, it does not mean that the phase spectrum is not of crucial importance as well. The distortion or errors of it - assumed that the amplitude spectrum is the same - can result in a significant change of the signal shape. The reason for which is that in the case of identical phases, signal components add up to significant peaks.

This can be observed in the following image: if the harmonics of the square wave are added with a 90 degrees phase shift, then the approximation will be satisfactory, if there is no phase shift, the signal will be rather different. For real images it can result in the overflow of pixel values, i.e. in the loss of information.

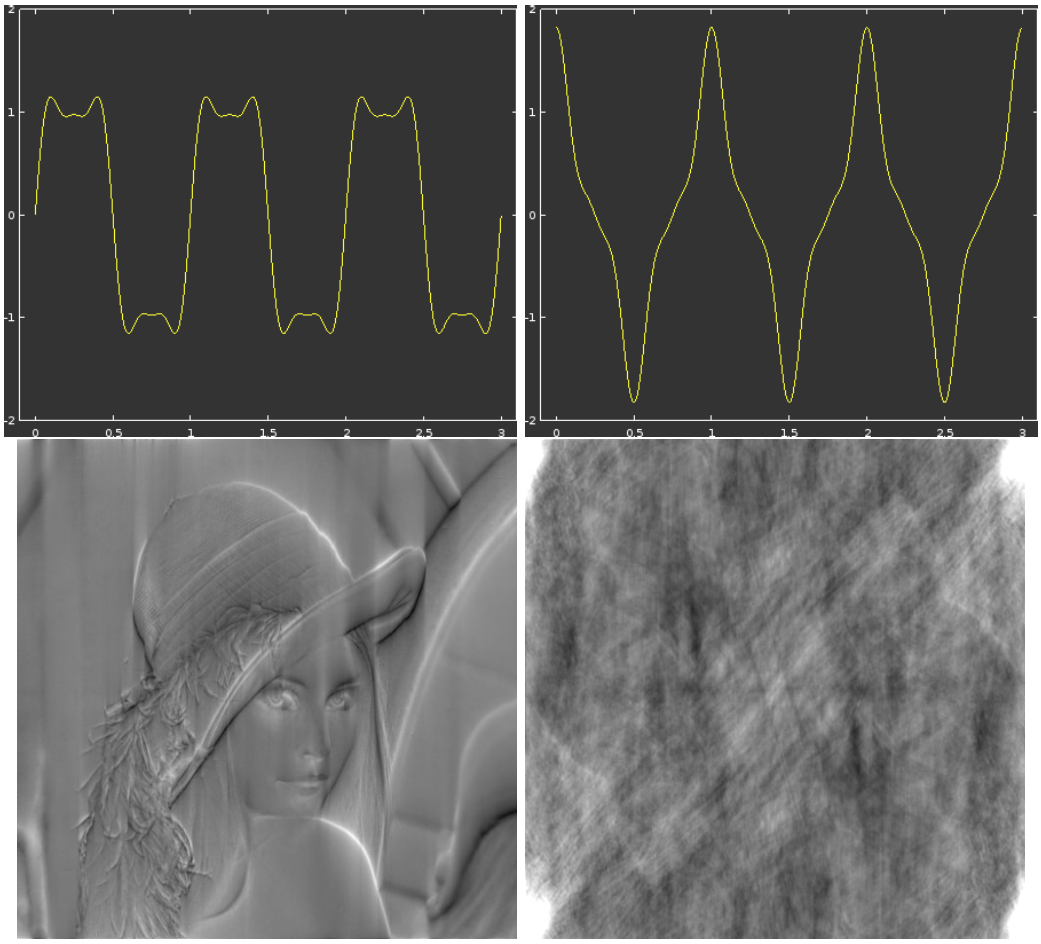

Figure 3.3: *The effect of phase distortion for a square wave (above) and a real image (below).*

3.2.2 Fast Fourier Transform

For the calculation of the spectrum the Fast Fourier Transform is used most often, the principle of which is that the 1D-function consisting of N elements is split up recursively to parts of length $\frac{N}{2}$, so that odd and even indices are in a separate group. This is continued until a group consists of only two elements, then the so-called butterfly operation is carried out.

$$\begin{aligned} y_0 &= x_0 + x_1\omega^k \\ y_1 &= x_0 - x_1\omega^k \end{aligned} \tag{3.3}$$

After that the recursion is traversed in reverse direction and the butterfly operation is carried out on ever longer (according to the powers of 2) data series, the end result will be the N FFT values. The algorithm reuses the temporary results of the butterflies, so the Fourier transform can be calculated instead of N^2 with only $N\ \ log(N)$ steps. As in the case of DFT, the FFT can also be extended to two dimensions, namely, the 1D-variant should be carried out in each direction once.

3.2.3 Filtering

Above we discussed the question of image representation in the frequency domain, although no reason was mentioned why this could be useful. In the following, we discuss one main reason, namely the efficient implementation of filtering algorithms.

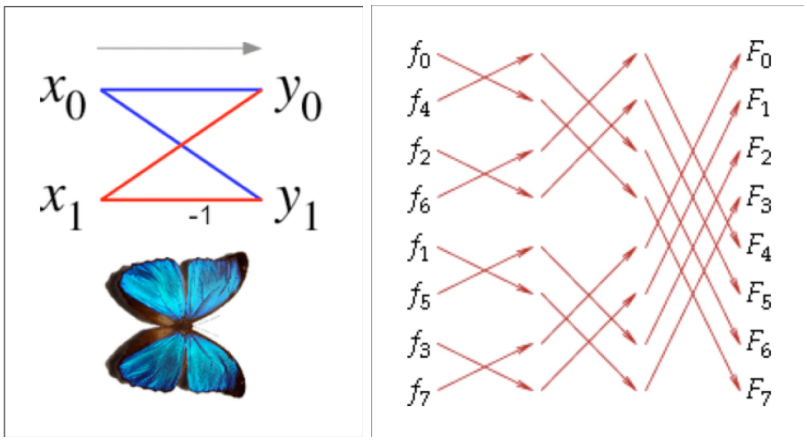

Figure 3.4: *The butterfly operation (left) and the FFT on 8 data points (right).*

3.2.4 Ideal filters

If we consider the image as the linear combination of sines and cosines with different frequencies, then it is evident that elements with lower frequencies correspond to image features that change slowly, i.e., are smooth, while those with higher frequencies to that of fast-changing details. This relationship can be exploited in different ways, e.g., image noise is pixelwise independent, i.e., they cause high-frequency changes - so using a low-pass filter, we are able to reduce image noise. On the other hand, edges can be found with the help of a high-pass filter, for edges correspond to rapid, sharp changes in the intensity function. If high-frequency details are only highlighted but also low-frequency components remain, then we speak about image sharpening.

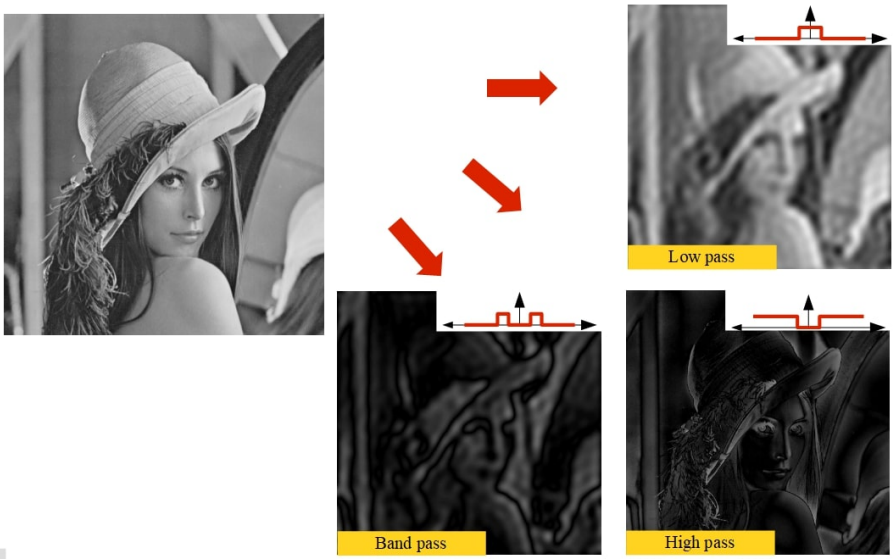

Figure 3.5: *Different filtering methods in the frequency domain.*

Interestingly, because it is also a two-dimensional array of coefficients, the Fourier transform of an image can also be plotted as an image. Because each frequency has its corresponding amplitude and phase, and the amplitude information is for humans easier to interpret, often just the amplitude spectrum is plotted. In the center is the constant component, from where to the borders frequency increases. Due to the properties of the Fourier transform, the image is centrally symmetric.

In the above image, the main drawback of the low-pass filter is clearly to see. Although being one of the easiest to implement filters (in contrast with control engineering, causality is not a problem, since due to the lack of the time dimension, its concept cannot be interpreted), it causes

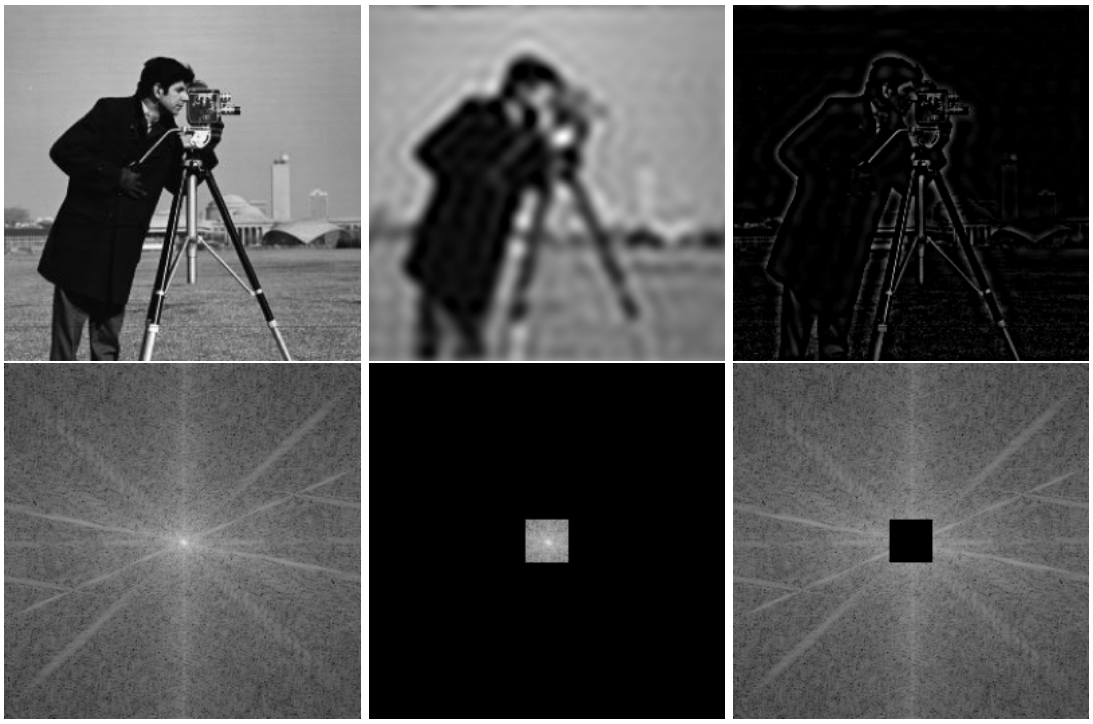

Figure 3.6: *Filtering and their spectra. Original image (left), filtered with low-pass (center), filtered with high-pass (right).*

ring-like artifacts. The phenomenon is easy to interpret: as in the example at the beginning with the square wave, not using high-frequency components means that arbitrary rapid changes cannot be approximated satisfyingly, i.e., we will see the image of the wave-like components.

To solve this problem, a filter can be used which has a more sophisticated spectrum, e.g., the trapezoid-filter or "its smoothed version," the Butterworth-filter.

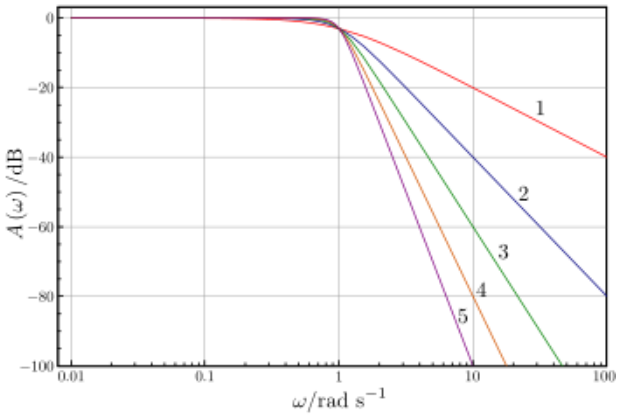

Figure 3.7: *The amplitude spectrum of the Butterworth-filter for different filter orders.*

3.2.5 Convolutional filters

The Convolution Theorem, which states that convolution in the time domain corresponds to elementwise (i.e., pixelwise) multiplication in the frequency domain, is of great significance. This means that convolutional filters should be applied in the frequency domain to save computational power. In the case of multiple filters to apply, the gains are even better, because the Fourier transform and its inverse should only be carried out once.

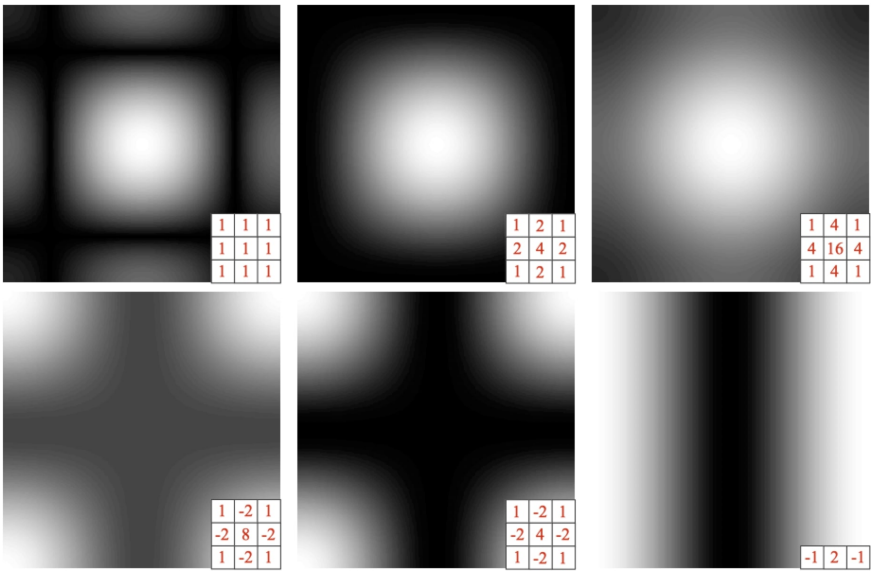

Figure 3.8: *The spectrum of different convolutional kernels.*

The problem of periodic noise was mentioned before; they are generally the result of electromagnetic interference, and unfortunately, cannot be eliminated with smoothing filters. Nevertheless, in the frequency domain, they are easy to find and filter, namely, only the corresponding component in the spectrum should be suppressed - even the filtering can be constrained to the proper frequency direction, not both of them should be suppressed if only one is present.

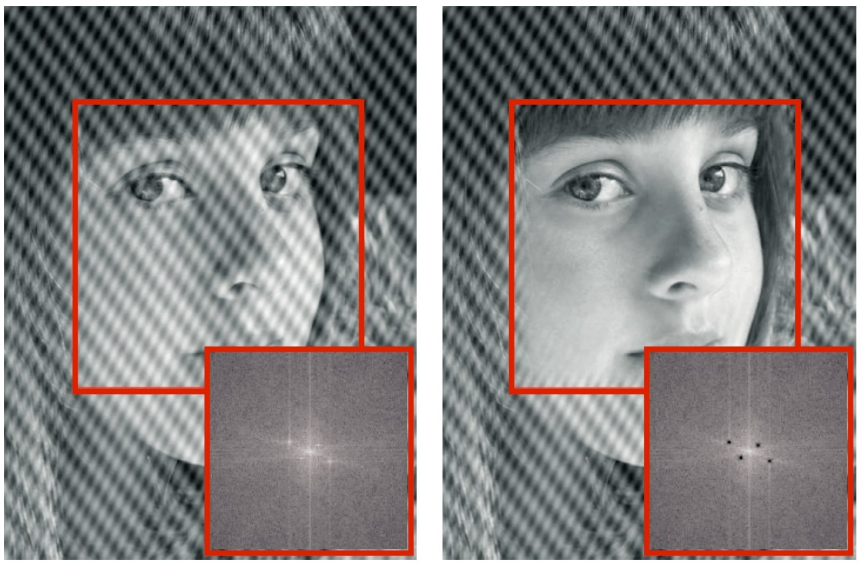

Figure 3.9: *Frequency-domain filtering of periodic noise.*

Application: frequency-domain analysis can also be used to check the orientation of documents (first of all, printed text). Namely, the change of darker rows and brighter gaps defines a dominant frequency component, the direction of which can be detected with ease in the spectrum. Based on that, a decision can be made whether the text is aligned horizontally or vertically, which can help to improve the accuracy of optical character recognition algorithms.

3.3 Cosine transform

The Discrete Cosine Transform (DCT) is similar to the DFT, i.e., it also interprets the image as a periodic function, but outside the borders, a reflectively repeating image is assumed - this will be the starting point for the DCT. Since the signal is - according to the assumption - symmetric, the spectrum will be real.

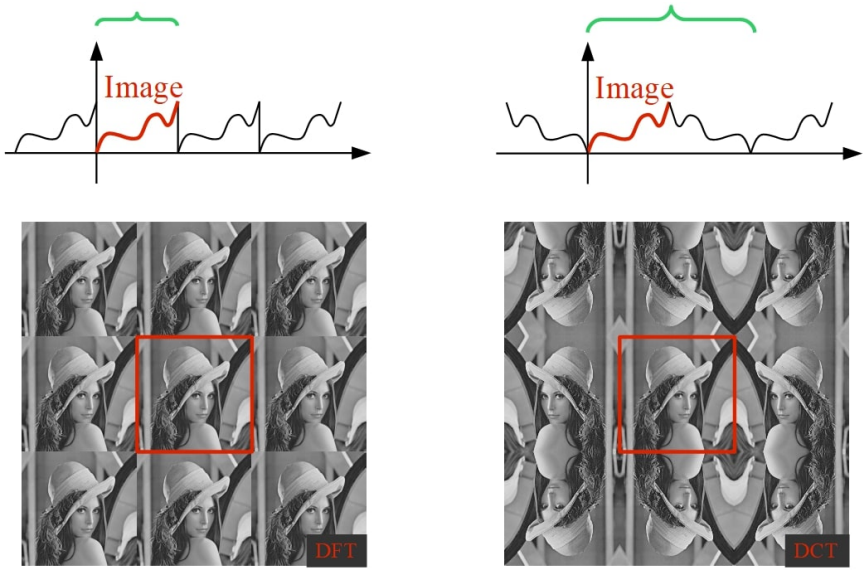

Figure 3.10: *The difference between DFT and DCT.*

This property means several advantages, namely storing real values (instead of complex ones) requires only the half of the storage. On the other hand, the reflections at the border ensure that no jumps are introduced to the image signal - which would occur in the case of DFT as high-frequency components. Practically speaking, the formula for calculating the DCT will also be much simpler:

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

3.3.1 FCT

If we would like to accelerate the calculation of the DCT, we can use the FFT algorithm, but as a prerequisite, the reflected - symmetric - image function should be calculated. After applying the FFT algorithm, building the real part results in the DCT of the image. Note that this modus operandi also does several redundant operations (basically, at the end half of the results is discarded), to handle this, there are algorithms which omit the redundant operations.

3.3.2 JPEG

One of the most successful image compression standards, JPEG, also uses DCT, precisely speaking the variant II of it. JPEG is a lossy compression procedure that can achieve compression ratios up to 1:10 while maintaining for human vision practically the same quality. For that, the specialty of the human vision is exploited: that we cannot perceive rapid intensity changes very well.

Furthermore, the sensitivity of human vision regarding rapid changes is different for both color and intensity vision. Since the density of the color-sensitive cones is less than that of the light-sensitive

rods on the retina, the color-resolution of the eye is also less. So it would be reasonable to store color information with smaller resolution than color/contrast. In the case of RGB, this is not possible, to achieve this JPEG uses the YCbCr color space so, that the resolution of the Cb and Cr components are halved in one (or most often in both) direction.

After that the image is split up into 8x8 (if the resolution is decreased for color channels, 16x16) blocks - the average will be subtracted, then the DCT is carried out on each block, resulting in the spectrum.

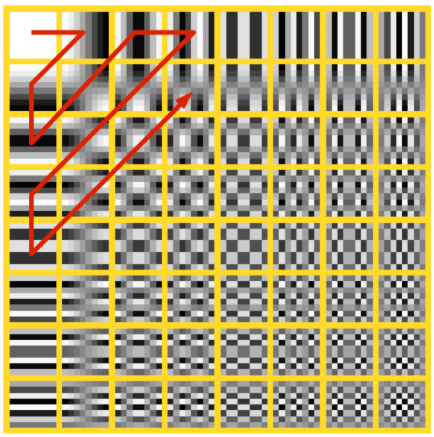

Figure 3.11: *The principle of JPEG-compression.*

After that, the floating-point number resulting from the previous step is divided by a for each 8x8 DCT value predefined constant (which is engineered to give the best results based on the characteristics of human vision), then the values are rounded to the nearest integer; thus 8 bits suffice for each variable. Due to quantization, several components encoding high-frequency components will equal zero, thus no need to store them. To optimize storage further, the indices are calculated diagonally; thus high frequencies are left to the end, i.e., only some of the first coefficients should be stored.

An important remark: due to the fact that high-frequency components are mainly omitted, the JPEG algorithm is inclined to blur edges, and like in the case of the ideal low-pass filter, artifacts are observable around the edges. Although being hard to discover them with the eyes, an edge detection algorithm is likely to find false edges. Thus, in the case of artificially generated images (geometric shapes, printed text, etc.) the use of the png, eps or similar compression methods is encouraged.

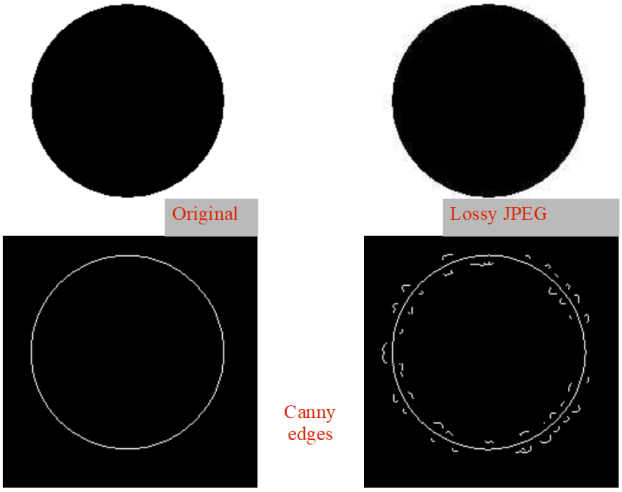

Figure 3.12: *The effect of JPEG compression on the edges.*

3.4 Applications

Besides filtering and compression, the frequency-domain image representation has several other applications, e.g., halftone or mosaic images can be converted into real images, meaning that the mosaic effect, due to its easy to identify period in the frequency domain, can be filtered with ease.

3.4.1 Shape recognition

With the help of the Fourier transform, it is also possible to detect simple shapes. E.g., optical character recognition (OCR) operates like this, because printed characters have specific shapes and a so unique spectrum, which can be used for their recognition.

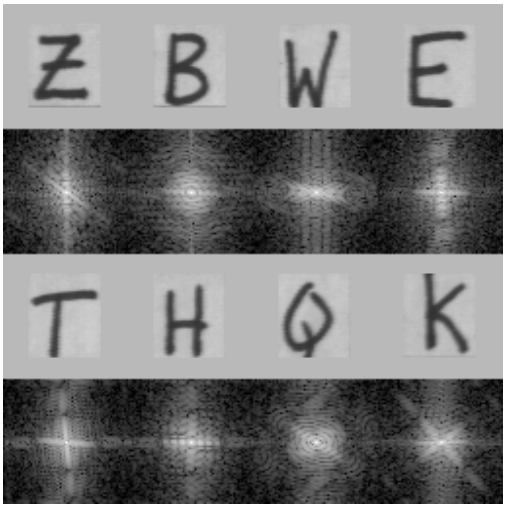

Figure 3.13: *The spectra of different printed characters.*

3.4.2 Deconvolution

Frequency-domain interpretation gives us means to invert convolution with ease; this operation is called deconvolution. During deconvolution, the spectrum of the image will not be multiplied but divided by the spectrum of the filter. With the help of deconvolution, the smoothing effect of e.g., improperly set focus can be corrected. A more sophisticated version is the so-called Wiener-deconvolution, which applies besides deconvolution a frequency-dependent filtering step as well, thus suppressing noise.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

[4] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–297, May 1965. DOI: 10.1090/s0025-5718-1965-0178586-1. [Online]. Available: https://doi.org/10.1090%2Fs0025-5718-1965-0178586-1.

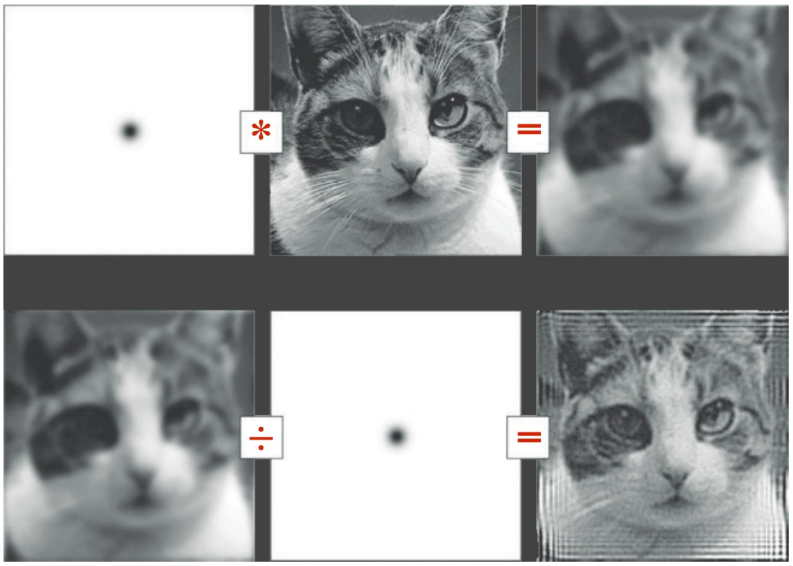

Figure 3.14: *The deconvolution principle and its result.*

4 Image features

4.1 Image fitting, feature types

The main task is during computer image processing to extract such image features that can be used to carry out high-level decisions. Several types of such properties exist, e.g., edges mentioned in the previous lecture. Today, more complicated and so more computationally intensive features will be reviewed.

Figure 4.1: *The applications of image fitting: object identification and relative position prediction (left), panorama fitting (right).*

4.2 Intensity

The simplest feature types are the intensity or color values, to determine them, we do not need separate algorithms, i.e., they have de facto no computational cost. Of course, these features are the least robust, so they can be feasible only in particular situations.

4.2.1 Template matching

Nevertheless, our task can be constrained to a controlled environment (static background and illumination), where a known, non-changing object should be detected. For such tasks, the algorithm of template matching can be utilized, which is rather simple. First, a reference image is acquired of the object (this is our template), which is matched to the image in all positions. To quantify the result, a cost function is evaluated each time, the extrema of which will mean that detection was successful.

In the case of template matching, generally, two different cost functions (similarity measures) are minimized in practice, the first of which is the squared error (squared distance or L2-norm). A more interesting choice is the convolutional/correlational matching, where the convolution between the template and the image is calculated. The property of convolution is that its value will have a high absolute value if the underlying image part resembles the kernel or its inverse. To illustrate this feature, recall the kernels used for edge detection, they are similar to edges indeed.

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

In case of the convolutional filter, if the detection is done for both minima and maxima, then the inverse of the template is also detected - the only difference between the two cost functions. This can be useful e.g., for character recognition, so a template with white background and black character can also detect samples of whites characters on a black background. One important drawback of template matching is that it is rather sensitive to scaling, rotation, and distortions. Any of the above being present means that new variants of the template should be created. I.e., the algorithm is needed to be carried out with each of them, thus resulting in a slower speed.

Application:OCR (Optical Character Recognition), where printed characters should be recognized, is one of the most important fields of utilizing template matching. The task consists of the execution of the template matching given a document with proper orientation, which results in the text in an editable format. The applications of OCR are diverse, from converting scanned documents to editable files to automatic ID scanning.

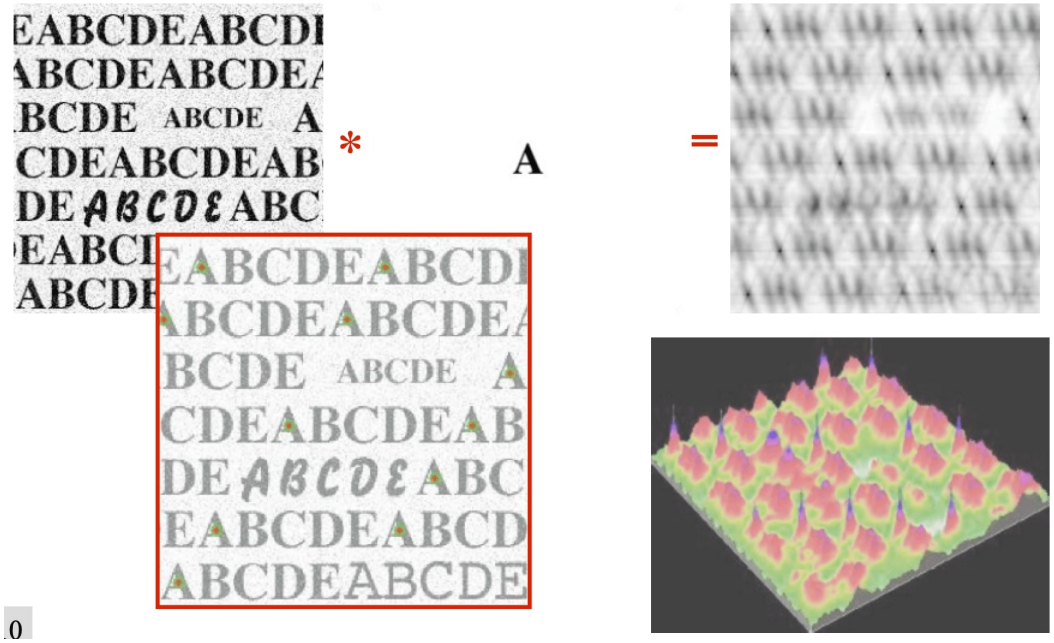

Figure 4.2: *Template matching for OCR in use.*

Other important application area is the scientific domain of virtual and augmented reality, where input devices are often realized in a manner that a special (generally black and white) pattern is mounted on them. These patterns are chosen so that they do not occur on the original objects, i.e., marker-localization can be solved easily and robust, with the help of template matching. This approach is often used in tangible augmented reality systems, the base principle of which is to enable interaction with the real world utilizing real objects.

4.3 Edges

First, edges were discussed, which are simple, easy to extract, nevertheless moderately robust descriptors of the objects in the image. The main drawback of edges is the fact that the change in the image is locally only in one direction significant, so if the edge moves in a perpendicular direction, then it cannot be detected.

To solve this problem, image corners can be used instead of edges. Image corners are by definition such pixels, from which in each direction, the intensity of the image changes in a significant manner. I.e., independent from the direction of the movement of the object, the corner can be tracked; meaning that the object that the corner belongs to can also be tracked or detected.

One of the most widely-used image corner detectors is the Kanade–Lucas–Tomasi- or just the KLT-detector. This algorithm works along with the principle of searching such image details that

change in all directions, i.e., the tracking of them will be simpler. To select such pixels, we can do the following: we calculate the squared difference between the local neighborhood (defined by a local window) of the image and the same window slightly moved in each direction - if the difference is big, regardless of the change direction, then an image corner is found.

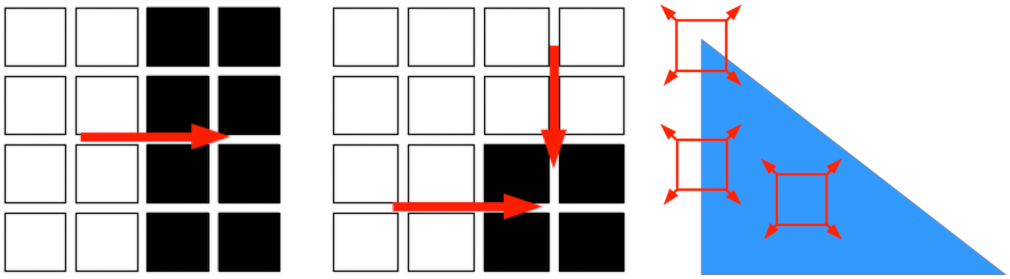

Figure 4.3: *Ideal image edge (left), corner (middle) and the principle of corner detection (right).*

4.3.1 Local structure matrix

Of course, the KLT-detector does not work in the above-sketched way, because that method would require a too high computational cost. Instead of that, the derivatives in x- and y-direction are calculated, then for each pixel, the derivatives in an $n \times n$ neighborhood are ordered into a matrix of dimension $n^2 \times 2$. Namely, the error in each direction can be formulated as follows:

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{4.2}$$

So the cumulative error term is:

$$||E||^2 = \vec{u}^T I^T I \vec{u} = \vec{u}^T H \vec{u} \tag{4.3}$$

Where I_{x1} and I_{y1} are the derivatives of the first pixel (the ordering is irrelevant for the result) in x- and y-direction, H is the so-called local structure matrix of the pixel. Note that if we assume that the expected value of the derivatives is zero (which assumption is well-founded, because derivatives has an equal probability to be positive or negative), then the local structure matrix is proportional to the covariance matrix of the derivatives of the image part.

4.3.2 KLT, Harris

The eigenvectors of the local structure matrix will give us the directions of most significant and smallest image change, the measure is for both the eigenvalues (λ_1, λ_2). Based on that, a method can be developed for corner detection: those points where the smallest change is big enough (i.e., the smallest eigenvalue is big enough) can be considered as corners.

The KLT-detector calculates the local structure matrix for each pixel and based on the smallest eigenvalue, a measure of "cornerness" is assigned, which is then used to determine whether a pixel is a corner or not. Of course, that measure will also be big for the pixels near the corner since corners are not perfect points - an important enhancement step could be to consider local maxima as image corners.

In practice, often the Harris-detector is used instead of the KLT-detector, which also uses the local structure matrix, but the measure of cornerness is as follows:

$$R = det(H) - k * tr(H)^2$$
$$k \in [0.04 - 0.06] \tag{4.4}$$

The main advantage of this formula is that its calculation is much faster than determining the eigenvalues. The Harris-criterion is also applicable for the detection of edge-like image parts; namely, the result of the Harris-criterion will be a large negative number. Another difference is between both detectors is that the KLT has a result more similar to human perception.

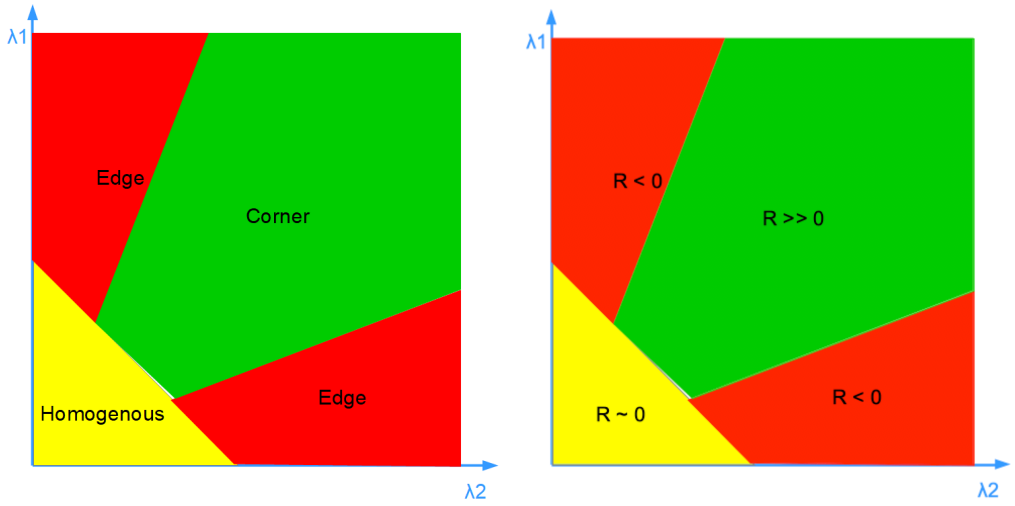

Figure 4.4: *Classification of the KLT- (left) and Harris-criterion (right) values.*

4.4 Invariances

Concerning image features, it is important to discuss the robustness requirements those should fulfill. The reason for them is to be able to detect the same object on different images, independent from the transformations between the images.

The simplest image transform is the intensity change, which is most often due to the change of illumination - two types of which exist. One is the additive change when a constant is added to the pixels, the other is the multiplicative change when the pixels are multiplied with a constant. Another frequently experienced transformation is rotation or the change of scale, which is due to images taken from different perspectives and different distances. Change of perspective is also responsible for perspective distortion, which results in e.g., the deflection of parallel lines.

Both the KLT- and Harris-operators are only for two of the transformations mentioned above invariant: on the one hand, for additive intensity change, the reason for which is that this additive disturbance disappears due to the derivation step. On the other hand, for rotation, because the eigenvalues of the matrix are rotation-invariant. The multiplicative change of intensity results in the multiplicative change of the derivatives (with the same factor), i.e. the criterion values are also changed. Scaling influences the result in a rather significant way, namely due to magnification a corner-like pixel will be just a rounded edge-like object.

4.5 Complex image features

To handle the problem of transformation-invariance, the SIFT-algorithm (Scale Invariant Feature Transform) can be used, which is - except to perspective distortion - invariant to each transformation, and so it provides a robust local region-descriptor, and for this property, is widely-used. The principle is to search for corner-like pixels, nevertheless, not only one metric is assigned to them but an invariant descriptor for the local neighborhood of corner points, which can be used for match with features on other images.

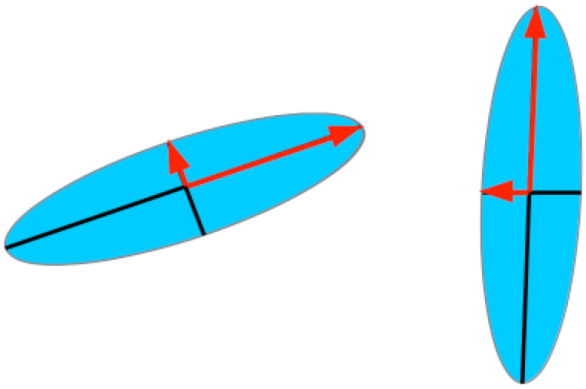

Figure 4.5: *The effect of rotation on the eigenvalues.*

4.5.1 SIFT detector

The main principle of the SIFT detector is that detection is done with the help of different scale factors, and each feature also gets a scale variable assigned to it, which will be used for the descriptor code, thus ensuring scale invariance. Corner detection is done with the DoG-kernels (Difference of Gaussians) introduced in chapter 3. Although the DoG-kernel was referred to as edge-detector, the response of it will be maximal in the case of impulse-like changes. Important to note that if several DoG-kernels are applied to an impulse, then the maximal response will be of that filter that matches the best the size of the impulse.

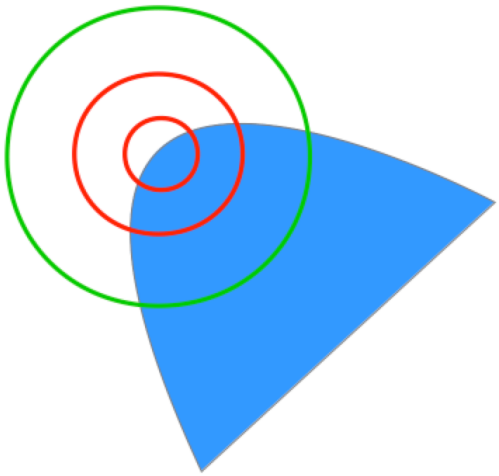

Figure 4.6: *The measure of cornerness of an image feature (blue) for different scales.*

The first step of the SIFT-algorithm is to apply DoG-kernels of different sizes to the image, the response of which is stored. After that, the maximum of local filter responses will be determined, but not only regarding x- and y-coordinates but also regarding the filter size. The coordinate providing the maximal response will be the position of the image corner, the size of the filter resulting in the maximal response will be the scale factor assigned to that point. Interestingly, SIFT is not constrained to the quantized set of positions (given by the discrete pixel grid and filter sizes) while determining the position; namely, a polynomial is fitted on the result, the maximum of which will be calculated. With this method, the corner position can be determined with subpixel accuracy (i.e., the resolution is finer than pixel size), which can be a requirement for several applications.

Several tricks are used to accelerate the SIFT-algorithm: instead of DoG-kernels Gauss-kernels are applied, the difference of which gives the result. Otherwise, when the filter size grows twice as big

as the original, the algorithm switches back to the original, which is applied to the image with half-resolution, so the same result is obtained with a fourfold reduction regarding computational cost.

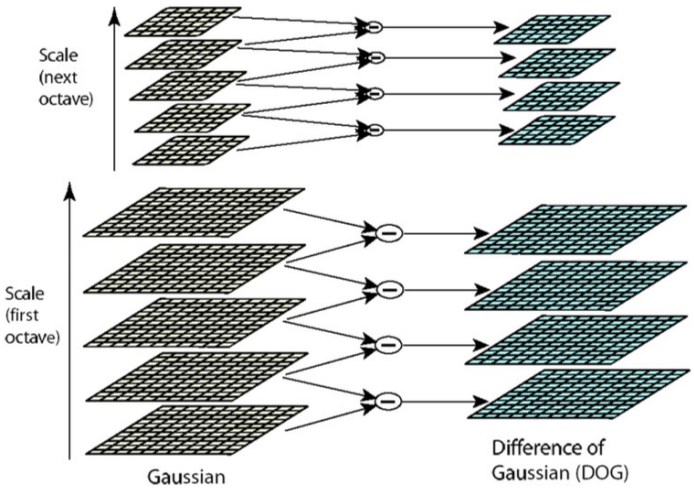

Figure 4.7: *The filtering scheme of the SIFT detector.*

4.5.2 SIFT descriptor

The second step of the SIFT-algorithm is to generate the descriptor code for the image corners, this code consists of 128 numbers - it describes the 16x16 pixel neighborhood of the corner. The neighborhood is always selected from the same image scale as the corner is, thus ensuring scale-invariance. Since the descriptor is made from the intensity gradients (containing the partial derivatives of the x- and y-direction) not from the intensity values, the invariance regarding additive intensity change is ensured, like in the case of the KLT-algorithm.

The most revolutionary idea of the SIFT-algorithm is the incorporation of rotational invariance; to achieve this property, a histogram is made from the gradients of the neighborhood of the pixel, which depicts the magnitude of the gradients into each direction. The histogram divides the gradients into 36 bins, i.e., the resolution is 10 degrees. If a gradient is on the border of two bins, then it is split up equally between both. An important nuance that gradients of pixels further from the corner are considered with a smaller weight.

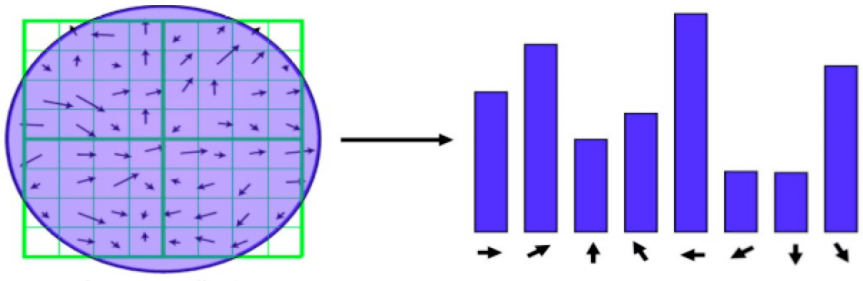

Figure 4.8: *Creating the gradient histogram (in the example it has 8 elements)*

The maximum of the gradient histogram (i.e., the direction, which points to the neighborhood with the greatest change) is set as the orientation of the corner. After that, the image is rotated so that the maximum should show into a predefined direction (e.g., upwards). The final descriptor code will be created from the 16x16 neighborhood of the rotated (and formerly scaled) image, thus ensuring rotational invariance. If the maximum of the gradient histogram is ambiguous, then a descriptor is created for each possible orientation.

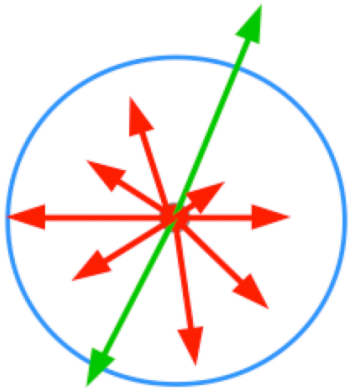

Figure 4.9: *Determining the SIFT orientation (not all 36 elements shown).*

For the descriptor, the 16x16 pixel neighborhood is split up into 16 subsets, 4x4 pixel each, then for each of them a gradient histogram is created as described above. The only difference is the number of bins, i.e., the resolution, namely only 8 (instead of 36) bins are used, resulting in a resolution of 45 degrees. The histogram values are summarized in a vector of 128 elements. This vector will be normalized, thus ensuring invariance to multiplicative intensity changes, namely the constant, which multiplies the gradients will be canceled out during the normalization step. This normalized vector with 128 elements will be used as the feature descriptor.

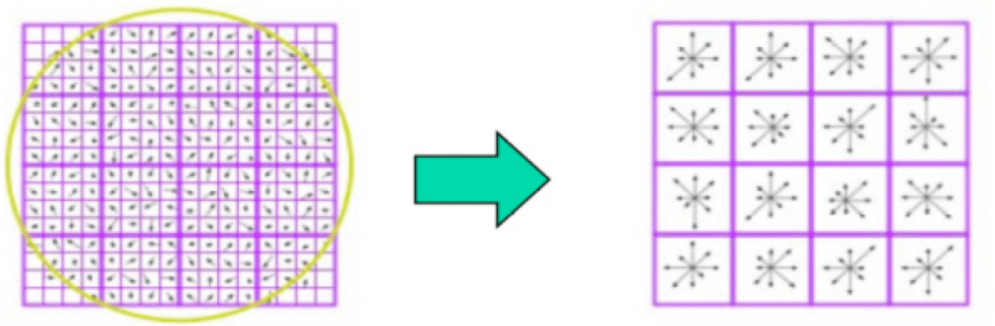

Figure 4.10: *Constructing the final SIFT descriptor.*

Application: local image features can be used in several walks of life, e.g., for image matching, which is useful for creating panorama images or for 3D-vision discussed later on. Nevertheless, one of the most obvious applications is the detection of rigid objects based on a reference image. Our goal is to detect objects which can not be deformed and have not many variants. During the process, image features based on the local gradient histogram will be identified in the reference image and compared to that found in the actual image (a pairing also occurs according to the similarity of the codes). In the case of several matches, the detection is regarded as successful.

This method is used if such objects should be identified, which cannot be modified by the system designer, i.e., they cannot be made easier recognizable. E.g., it is often used for virtual or augmented reality systems that operate with real objects, as well as in camera-based surveillance and tracking systems (e.g., CCTV for public safety or traffic).

4.5.3 ORB detector

The SIFT-algorithm provides such a feature detection and description procedure which is invariant to the transformations mentioned above, nevertheless, it has some practical drawbacks, namely, it is rather computationally expensive, modern PCs are not always able to run it in real-time (luckily,

the patent expired in 2020). Since its publication, several other algorithms - based on SIFT - were designed, which can be executed in real-time. Among them, SURF should be highlighted, because it provides similar robustness but great parallelization opportunities for GPUs, and ORB, which can run in real-time (invariances are retained).

ORB (Oriented FAST and Rotated BRIEF) is the combination of two other algorithms (somewhat they were also extended). ORB uses the FAST algorithm, which is a fast corner detector, for keypoint detection. FAST defines in the neighborhood of the analyzed pixel a circle, the pixels on which are selected for analyzing cornerness. After that, it is counted how many of the selected 16 pixels have a greater intensity difference w.r.t. the pixel being evaluated (for that, a threshold is used); if the number is high enough, then a corner is detected.

Important to note that if the threshold is 12 or higher than not corner-like pixels can be easily excluded with a check for the cardinal points, otherwise to ensure rapid exclusion of non-corner-like pixels a decision tree should be used.

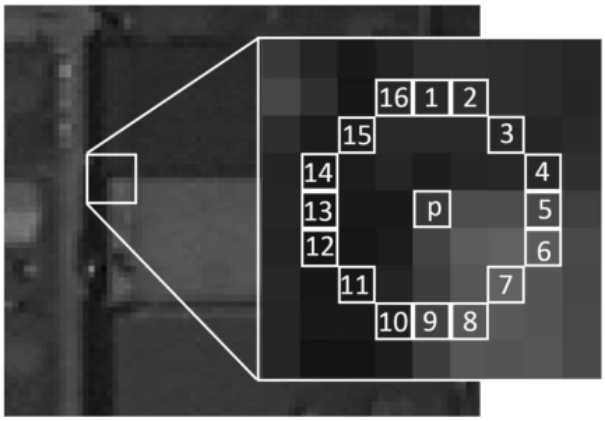

Figure 4.11: *The operating principle of the FAST detector.*

In this case, also an image pyramid is used for ensuring scale invariance, i.e., to be able to carry out corner detection for different scale factors. Similar to SIFT, the final descriptor is created with the help of the image belonging to the scale factor selected for the image corner. Note that FAST is inclined to detect edge-like key points, so the detected pixels should be filtered with the KLT-criterion. In this case, only for some points should the calculation of the criterion be carried out, which means a negligible amount of computation.

The main limitation of the FAST detector is that it cannot assign an orientation to the key points. To resolve this, a separate orientation measure should be defined. The FAST algorithm calculates for that the center of mass of the pixels in the neighborhood of the key point, where their intensity gives the weight of each pixel. The formula of the calculation is as given below:

$$x = \frac{\sum xI(x,y)}{\sum I(x,y)}; \qquad y = \frac{\sum yI(x,y)}{\sum I(x,y)} \tag{4.5}$$

The orientation of ORB will be the direction of the vector pointing from the key point to the center of mass.

4.5.4 ORB descriptor

To generate the descriptor vector, ORB uses the BRIEF descriptor method, which uses (in contrast to the SIFT descriptor) a vector of binary (not floating point) elements. For that, the algorithm uses n_d (in the case of ORB, it is 256) predefined point pairs in the neighborhood of the key point, and for each of them, it checks whether the intensity of the first element of the pair is higher. So the BRIEF descriptor has the following form:

$$\left[sign(I(x_1^{(1)}) - I(x_2^{(1)})) \quad sign(I(x_1^{(2)}) - I(x_2^{(2)})) \quad \dots \quad sign(I(x_1^{(n_d)}) - I(x_2^{(n_d)}))\right] \tag{4.6}$$

The question is for which transformations is the BRIEF descriptor invariant? Due to the fact that it uses only the intensity difference of the pixels, it is trivially invariant to both types of intensity change. Scale invariance is ensured with the multiscale detection and descriptor generation (borrowed from SIFT). The problem is only the rotational invariance; namely, the predefined point pairs will coincide with other pixels if the image part is rotated.

The latter problem could be solved if the image part would be rotated, but this is a computationally intensive operation. It is much simpler if the orientation is quantized per 12 degrees (i.e., we have 30 bins) and not the image part, but the coordinates of the selected point pairs are rotated. Because the coordinates of the point pairs are selected while writing the algorithm, so the rotated variants could also be calculated previously, which means that in runtime, only a lookup operation is needed from the list suitable for the orientation of the image feature.

In the case of the ORB algorithm, the comparison of descriptors should not be compared with the help of the squared distance. Namely, the result will not be in every case satisfactory. Since the ORB descriptor is binary, the comparison is calculated with the Hamming-distance.

4.6 Dimension Reduction

Another frequent sub-task of computer vision is dimension reduction. In that case, the input data has rather high dimensionality, but the variables may not be independent or relevant. Images are perfect examples, namely each pixel is a separate variable, but neighboring variables are strongly correlated, and several pixels do not possess relevant information (mostly near the image borders). So it would be reasonable to merge dependent variables and reject irrelevant ones for the sake of significantly reducing the amount of information needed to be processed. Note that such a process can be able to find a concise description of a local image part.

4.6.1 Principal Component Analysis

An essential method for dimension reduction is called Principal Component Analysis (PCA), which assumes a normal distribution with zero mean. The latter can be easily satisfied with subtracting the mean of the expected value of the variables. After that, PCA transforms data into an orthogonal coordinate frame, the basis of which is given by the eigenvectors of the $(\Sigma = (\mathbf{X} - \mu)^T(\mathbf{X} - \mu))$ covariance matrix of the input data. The resulting basis vectors are called principal components, the variables defined with them will be statistically independent.

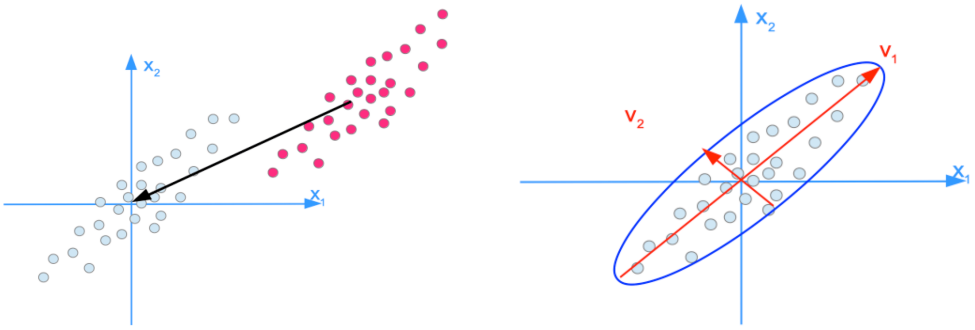

Figure 4.12: *The dataset to be reduced (left), and the principal components (right).*

The variables with the smallest variance are omitted until the variance is not below a predefined percentage of the variance of the whole dataset. The variance of each principal component is given by the eigenvalues corresponding to the eigenvectors. Note that PCA is the optimal reduction

algorithm for normal distributions. In the case of different units for the input variables, the result can be distorted, thus scaling is an important preprocessing step.

Application: an interesting application is face detection with so-called eigenfaces, which are the most important eigenvectors of a database consisting of human faces. These vectors can be used to understand the variance of human faces or to classify them. Namely, if an image is near the subspace spanned by human faces, then it can be considered as face-like; otherwise, it cannot.

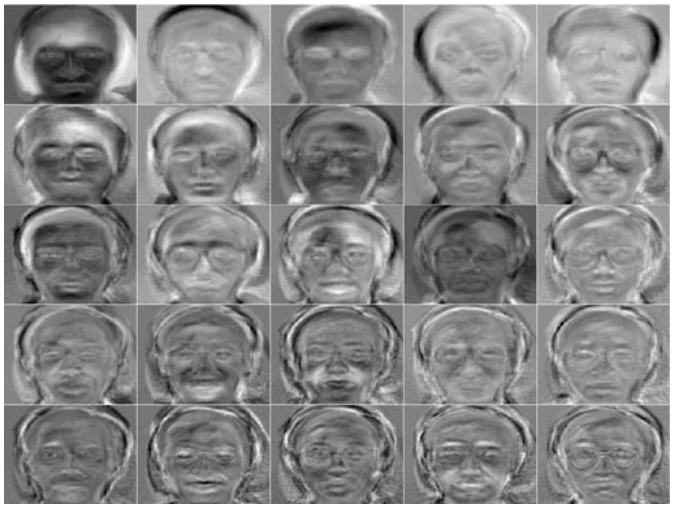

Figure 4.13: *The significant principal components of a database of faces (i.e. face-like images). Note that these images result after subtracting the mean of the pixels.*

PCA and similar methods are used for a compact description of small image parts. From that point of view, PCA can be interpreted as an image feature detector - the description of image features is optimal from specific aspects.

4.6.2 Linear Discriminant Analysis

Note that there is a variant of PCA which can be used for supervised learning, called Linear Discriminant Analysis (LDA), which includes labels corresponding to the dataset (two or more classes are contained). The goal of LDA is to reduce dimensionality in a way that the information useful for the separation of classes should remain. Mathematically speaking, LDA intends to maximize the variance of inter-class variance (not the variance of the whole dataset).

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

[5] J. Shi and Tomasi, "Good features to track," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*, IEEE Comput. Soc. Press, 1994. DOI: 10.1109/cvpr.1994.323794. [Online]. Available: https://doi.org/10.1109%2Fcvpr.1994.323794.

[6] C. Harris and M. Stephens, "A combined corner and edge detector," in *Procedings of the Alvey Vision Conference 1988*, Alvey Vision Club, 1988. DOI: 10.5244/c.2.23. [Online]. Available: https://doi.org/10.5244%2Fc.2.23.

[7] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. DOI: 10.1023/b:visi.0000029664.99615.94. [Online]. Available: https://doi.org/10.1023%2Fb%3Avisi.0000029664.99615.94.

[8] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*, IEEE, Nov. 2011. DOI: 10.1109/iccv.2011.6126544. [Online]. Available: https://doi.org/10.1109%2Ficcv.2011.6126544.

5 Detection and Tracking

5.1 Classification and detection methods

One of the simplest tasks in computer vision is the classification problem, i.e., when a label is assigned to the category of the object shown in the image. Generally, a classification system in computer vision consists of the sequence of different algorithms (this structure is called the algorithmic pipeline). The first steps are the acquisition and digitization of the image, which is followed by a preprocessing, image enhancement (noise filtering, intensity transformations) block. Feature extraction is the next step, to transform the information represented by the pixel intensity values to a space of higher abstraction (spanned by higher-level image features). These features are designed to be able to separate relevant information from disturbances in the space of the image features. The last step is to make a decision, i.e. based on image features the algorithm assigns a label to the image.

In previous lectures, the image correction and feature detection steps have been discussed extensively. However, we did not detail the last step of traditional vision: making decisions based on the extracted image features.

5.1.1 Viola-Jones

The Viola-Jones (also known as Haar Cascade) detector is a general object recognition method, which is used most often for face detection. It uses a special image feature called Haar feature, which analyzes an image part with a binary window so that the sum of the pixel intensities covered by black pixels is subtracted from the intensity sum of the pixels covered by the white pixels of the binary window. I.e., we get a signed number representing the similarity of the image part. A big positive number means similarity, while a big negative result means opposition, a result around zero indicates the total lack of similarity.

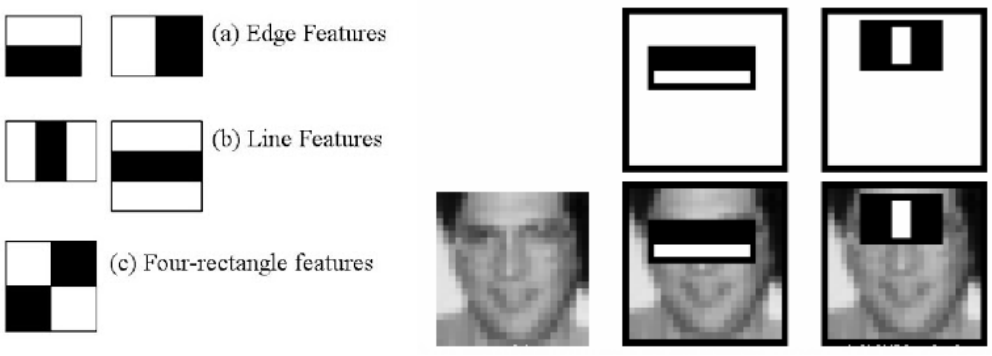

Figure 5.1: *Haar features (left) and their use for face detection (right).*

The main drawback of Haar features is that even for as small image parts as, e.g., 24x24, we could define more than 160,000 different features, the calculation of which would mean an enormous computational load. To reduce this number several different techniques could be used, e.g., the integral image can be calculated, with the help of which the calculation of the response of the features can be accelerated in a significant manner.

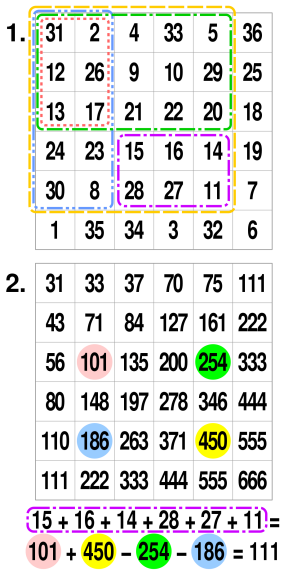

Figure 5.2: *The integral image: The original image can be seen on top, while the generated integral image is below. To compute the sum of pixels in the purple rectangle, we subtract the sum of the green and blue rectangles from the yellow rectangle. However, we subtracted the red region twice, so we add it back. This way, we can compute the sum of pixels in any area, using only three additions.*

Nevertheless, we have lots of features, the majority of which probably does not help to detect objects. To select the useful features, we use a training set, each element of which is classified in the following way:

$$label(I) = sign\Big(\sum_{j=1}^{M} \alpha_j h_j(I)\Big) \quad where \quad h_j(I) = \begin{cases} s_j & \text{if } f_j \geq \theta_j \\ -s_j & \text{if } f_j < \theta_j \end{cases} \tag{5.1}$$

Where α_j is the relative weight of the jth feature, f_j is the feature response and θ_j is the corresponding threshold value. The threshold values for the features and the s_j values are determined separately so that the classification based on one feature should have minimal error. After that, the weight of the given feature is determined to be inversely proportional to the false classification results. Note that not all classification of the images are taken equally into consideration, namely the weight of images classified correctly based on other features is reduced.

With this method, neglecting features with small weights, it is possible to reduce the number of features from 160,000 to about 6,000. Nevertheless, this number is still enormous. Namely, each feature should be assessed in every position. If the features are laid out in a cascade architecture, we can further accelerate the algorithm: at the first level, there are two features, which are able to detect 100% of face-like images while rejecting 50% of the not face-like images. The second filter layer is executed on the remaining images (it consists of 10 filters, which can filter out the majority of the not face-like images). So we can realize that this method is able to reject a significant amount of negative image parts (with not too much computation), i.e., the whole classifier should only be executed on a few image parts.

5.1.2 Bag of (Visual) Words

One drawback of the Viola-Jones detector that the features used are not invariant to the image transforms introduced previously, i.e., by using them, the algorithm can only compensate to a certain degree. For that problem, local image features can provide a suitable solution - besides detecting corner-like points, they can also provide a descriptor code for each detected feature, which is invariant to several transformations. These features encode both the look and structure of each

detected feature; their relative position lets us infer the global shape. Their main drawback is their computational cost, which means, e.g., in the case of the SIFT-algorithm an order of magnitude of seconds for an image of average size on a PC.

For image classification based on local gradient histogram, in most cases, the Bag of Visual Words approach is used. Using a Bag of Words for classification originates from text classification, which uses the idea that texts can be classified based on the relative frequency of the words in them. Note that this approach uses only the relative frequency, not the order or relative position of the words.

This ansatz can also be used for image classification if local image features are interpreted as visual words. During encoding, we face a problem: for textual data, there was a dictionary available that was used for encoding (e.g., by the sequence number of a word). A letter-by-letter match indicated match with a word in the dictionary - if we also had a synonym dictionary then we could assign the same code to words with the same meaning.

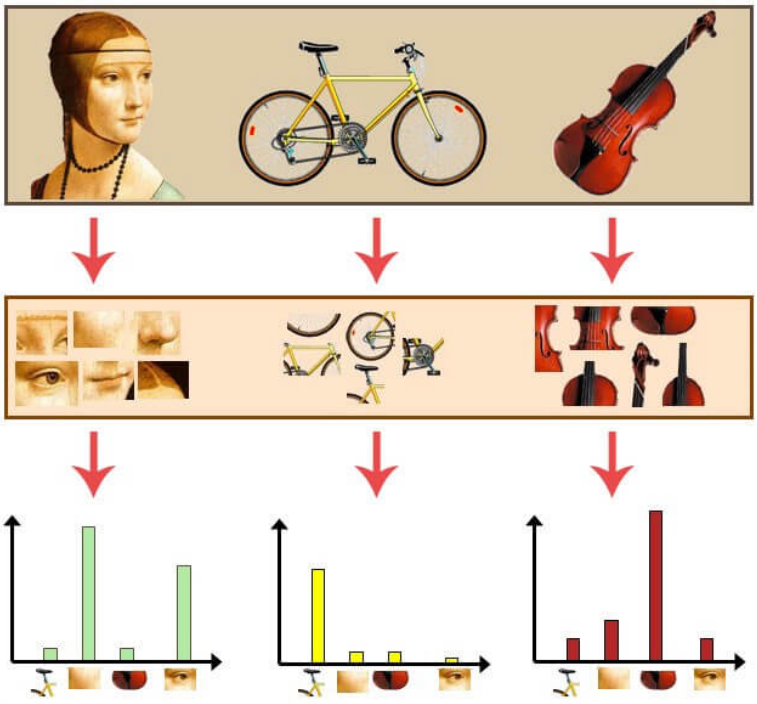

Figure 5.3: *The model of visual words.*

In the case of visual words, we do not have such a dictionary. On the other hand, visual words are represented by a sequence of floating-point numbers, which will never be equal elementwise. So the question is how can we construct a visual dictionary from an image database and how is it possible to assign the visual words found in the image to the words of the dictionary.

For that, we can use clustering - used in previous lectures for segmentation -, an essential method of unsupervised learning. Clustering divides the Bag of Visual Words into clusters to minimize the distance from the centers of the clusters. The number of clusters is a design parameter representing a compromise between decreasing the compactness of the clusters and increasing the number of clusters.

If clustering results in a visual dictionary, the assignment of local image features to its visual words can be carried out by minimizing the squared error calculated from the center of each word. After that, the histogram (relative frequency) of the visual words can be constructed - we also have the choice to weigh the words based on the distance from the dictionary entries. I.e., a detection with high confidence will result in a bigger weight.

The histogram can be used to assign images to classes - for the decision, several methods can be used, mainly one from the domain of machine learning, which - generally - extracts the optimal

decision function from a database, with the help of a statistical or optimization method. One of the simplest examples is the kNN method. For the decision, the k nearest neighbors of the current image are looked up in the image database (from here on: the training database); in this case, the distance is not calculated based on the pixel values but based on the Bag of Visual Words histogram. The label is decided with voting (the majority of the votes is needed) of the k nearest neighbors. Note that for machine learning we generally need to know the labels of the database.

5.1.3 Deformable part models

Besides classification we also may be interested in the position of the object (we want to do detection as well) - this is generally also possible with Visual Bag of Words, but the extracted position information will be somewhat inaccurate because it does not use any absolute or relative position information for classification.

Figure 5.4: *The deformable part model.*

Nevertheless, this can be remedied with some extra components - which are incorporated in deformable part models. The main principle of them is that classes are described not as one set containing visual words but a graph of words, where the edges describe geometric transformations between words. These transformations are somewhat flexible to allow for the deformations of the object.

The features used by deformable part models are basically convolutional filters. Two different types of those filters are root and part filters; the former has a big response for the center, the latter at the position of the before mentioned object parts. Further free parameters of the model are the relative positions of the part filters w.r.t. the root filter, which means two extra parameters for each part filter - they are learned from the training set.

But there is a problem: during training, we only have information about the position of the object, the position of the parts is not specified, i.e., it is not clear what to train the part filters for, because the position of their maximum response is unknown. Formulated otherwise, the expected maximum position of the filters and their relative positions from the root filter are so-called latent parameters.

To solve this problem, the following iterative procedure is used:

1. Random initialization of the filter parameters and the relative displacements.
2. Repeat until convergence:

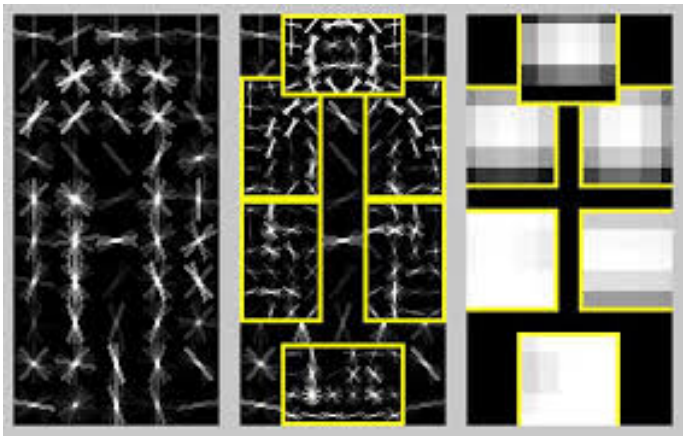

Figure 5.5: *The root (left), part filters (middle) and the deformations (right).*

(a) Part filters are executed near positive detections, the maximum of their response is prescribed as the expected correct response.

(b) Finetuning of the model parameters using the simplified task.

Note that due to the fact of random initialization, the part filters will likely converge to the detection of different parts. We can also note that both steps of the learning process are analogous to the k-Means and Expectation-Maximization algorithms - this is obviously not accidental, namely in the case of clustering we also used latent variables.

It is essential that as the method of the Bag of Visual Words required that the images in the training set should be labeled according to the corresponding classes for constructing the dictionary, deformable part models also need the position of the objects. Such databases are constructed first of all manually, after that the algorithm will be able to carry out the process on other (unknown) images.

5.2 Tracking

In computer vision situations where the input of the processing stage is not a still image, but a video, special care is taken. The tasks executed on videos are quite similar to those of targeting images. Segmentation, detection, classification can be used as well, but there are additional tasks such as movement detection, tracking, and the detection of specific events.

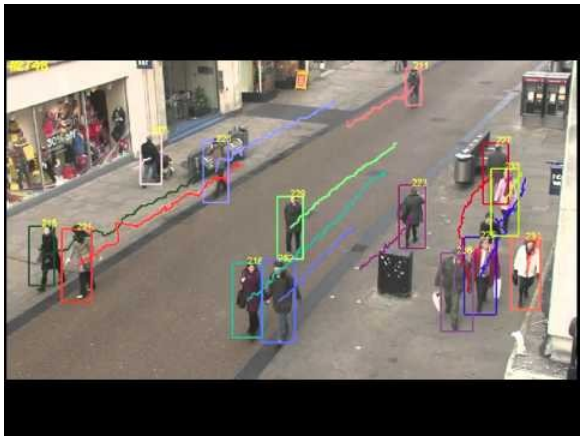

Figure 5.6: *Object tracking.*

To be able to carry out the processing step, first, the representation of videos should be cleared. In most cases, videos are just treated as a sequence of images, which are processed in the order of their arrival. Our algorithms are generally designed to be causal, i.e., they do not use the information stored in future image frames - note that it is not impossible, acausal methods can be easily realized if not real-time processing is required; otherwise it is of course not possible. Besides a sequence of images, we can treat the video as a base image and a sequence of difference images, but this approach is seldom used in computer vision.

There are different variants of tracking, e.g., pixel- and object-level tracking. In the latter case, it is important to identify each object between image frames, but this is a complicated task, namely the object itself can change between the frames or even other complications (hiding, non-linear movements, etc.) make the task more difficult.

Figure 5.7: *Pixel- and object-level tracking.*

5.3 Optical flow

The recognition and segmentation of movement often also need magnitude and direction, which can be calculated by the widely-used optical flow algorithm. The principle of which is that the image is approximated locally in each pixel with a linear, plane surface, then using the steepness of the plane surface and the intensity difference between two image frames, it deduces the amount of movement. The main drawback of it is the so-called aperture problem: namely, for specific types of image parts the movement can be not determined from the local image part.

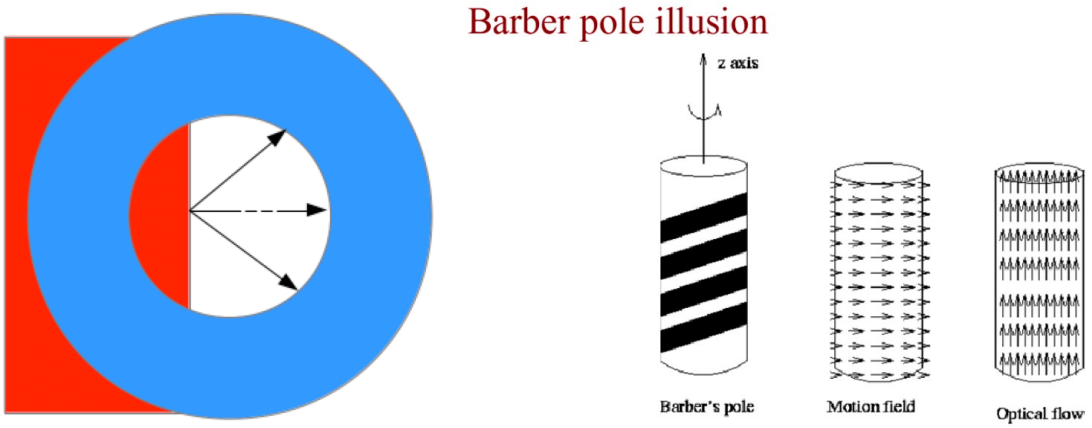

Figure 5.8: *The aperture problem (left). Local movement detection also has its role in human vision: a good example for that is the Barber-pole illusion, the askew stripes rotated around are perceived as the stripes would move upwards, while the movement is sideways. As we will see, optical flow behaves in a similar manner.*

Optical flow assumes that the intensity of objects does not change between images; there is only displacement. Using a Taylor-series approximation, it can be formulated as stated below:

$$I(x,y,t) = I(x+dx, y+dy, t+dt) = I(x,y,t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \tag{5.2}$$

Where $I(x,y,t)$ can be eliminated:

$$I_x dx + I_y dy + I_t dt = I_x \frac{dx}{dt} + I_y \frac{dy}{dt} + I_t = I_x u + I_y v + I_t = 0 \tag{5.3}$$

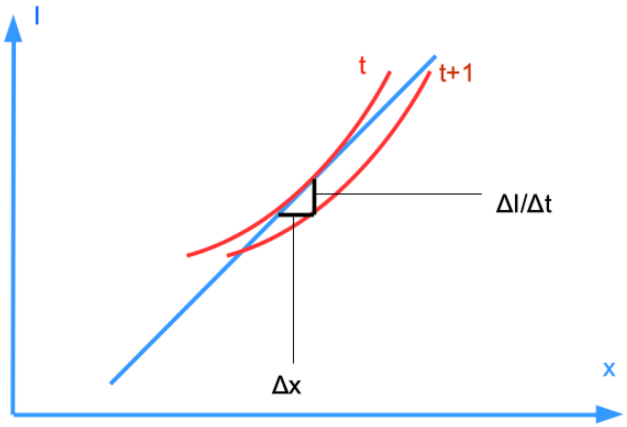

Figure 5.9: *The calculation of optical flow in one dimension: the image is approximated with a local line, so given the local(temporal) change of the specific pixel, the displacement can be estimated in a simple way.*

Where I_x, I_y és I_t are the derivatives of the image in each direction and by time, while u and v are the velocity components along the coordinate axes. The derivatives are calculated numerically (with filters and difference calculation); nevertheless, there is only one equation for two variables, which implies that the optical flow equation does not have a unique solution. In reality, the simple optical flow method is only able to determine one component of the movement vector, namely the component pointing into the direction of the gradient.

$$v = -u\frac{I_x}{I_y} - \frac{I_t}{I_y} \tag{5.4}$$

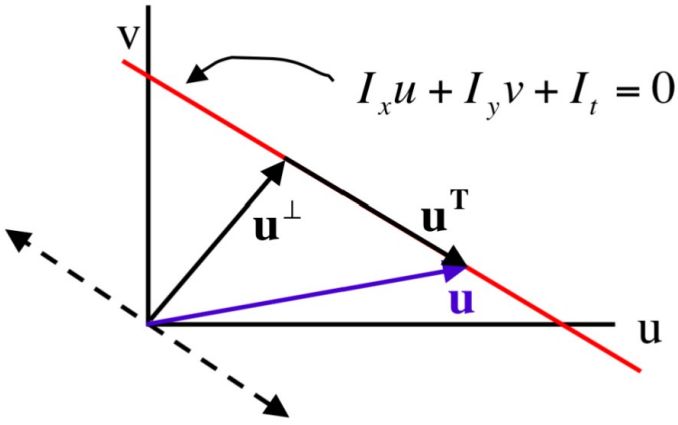

Figure 5.10: *The solution set determined by the optical flow equation (red line), the components of the solution.*

5.3.1 Lucas-Kanade method

To solve this problem, in practice most often the Lucas-Kanade optical flow algorithm is used. It uses one more assumption, namely that for neighboring pixels, the probability is high that they move with each other. I.e., the Lucas-Kanade algorithm intends to solve the optical flow equation in the neighborhood of the specific pixel (not only at one point), for that the method of Least Squares is used as follows:

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tn} \end{pmatrix} \tag{5.5}$$
$$X\vec{u} = Y$$

The derivatives in the equation are known, the displacement vector is unknown so that the error term above should be minimal. To achieve this, we solve the system of equations with the help of Least Squares, which gives the following formula for the optimal displacement:

$$\vec{u} = (X^T X)^{-1} X^T Y \tag{5.6}$$

The $X^T X$ matrix is the same as the local structure matrix in the KLT corner detector (unsurprisingly, the scientists Lucas and Kanade there and here are the same). If we recall the analysis of the local structure matrix, then its eigenvalues describe the magnitude of the biggest and smallest change. The matrix will only be invertible if the eigenvalues are significantly different, i.e. only in the case of corner points. Due to this, this methods is also known as the sparse optical flow algorithm, because it can be calculated only for a few pixels.

5.3.2 Farneback method

But there is a dense optical flow algorithm, which is able to determine the direction of the flow unambiguously, and can be calculated for each pixel. The method was proposed by Gunnar Farneback, thus its name. The main principle of this algorithm is that it uses a second-order polynomial to approximate the image locally (and not a plane). The assumption of the original version is utilized here as well, i.e., that there is no intensity change between two frames, only displacement. So calculating the polynomials for both frames, then comparing them gives us the displacement of the pixels.

$$I_1(x) = x^T A_1 x + b_1^T x + c_1; \quad I_2(x) = x^T A_2 x + b_2^T x + c_2 \tag{5.7}$$

The assumption, that both images are the same except to a displacement d, is used.

$$I_2(x) = I_1(x - d) = (x - d)^T A_1 (x - d) + b_1^T (x - d) + c_1 \tag{5.8}$$

Ordering by the powers of x:

$$I_2(x) = x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - B_1^T d + c_1 \tag{5.9}$$

From that, the equality of the b coefficients of the polynomial can be noted down, which results in the following formula for d:

$$b_2 = b_1 - 2A_1 d \quad \rightarrow \quad d = -\frac{1}{2} A_1^{-1} (b_2 - b_1) \tag{5.10}$$

In practice, a slightly modified version is used: namely, the image cannot be approximated globally with a second-order polynomial, so this is used only locally, and the displacement is determined in each position. Note that the Farneback method uses the neighboring pixels for the polynomial prediction, too, which results in a more robust Least Squares, thus eliminating the effect of noise.

5.3.3 Iterative and pyramid methods

Optical flow methods do not provide a good result if the displacement is so significant that higher-order terms cannot be neglected, i.e., the error of movement prediction will be significant, resulting in false movements in the long term. To handle this problem, an iterative approach is used, the principle of which is that the image is transformed (determined by the movement vectors calculated) after calculating the first optical flow. After that, a new optical flow prediction is calculated between the temporary and the second image. These two steps are repeated until the relative displacement is above a threshold. At the end, temporary movements are summed up to get the final result.

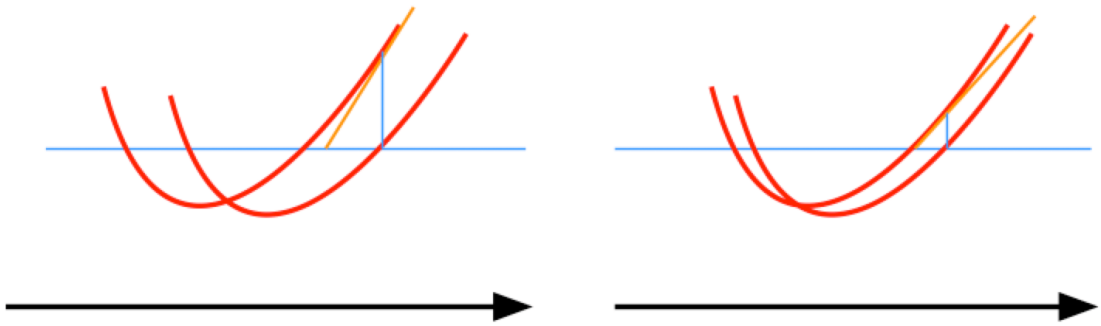

Figure 5.11: *The principle of iterative optical flow: the first iteration (left) brings the image nearer to the displaced version, so in the second iteration (right) the prediction will be good.*

The iterative optical flow algorithm is often used together with the pyramid method. In that case, the first step is to create an image pyramid of coarse-fine resolution with resizing the image. This is followed by the iterative method to predict the optical flow in the image with the smallest resolution. The resulting flow will be somewhat inaccurate, but big movements are easy to calculate; namely, they will be (measured in pixels) smaller with an order of magnitude (compared to the original). After that, the iterative method is applied for the images with consecutively higher resolution, so that the already calculated flow image will be used as a starting condition. With this trick on each level, the coarse result of the last step is made finer.

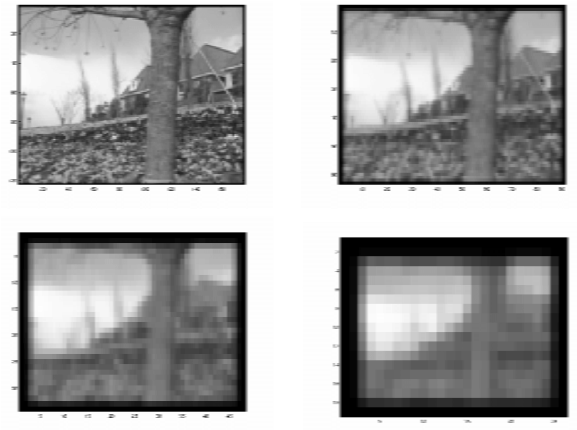

Figure 5.12: *The image pyramid.*

Application: the method of optical flow is used in several fields, an important application area is 3D-reconstruction with a moving camera. With this algorithm, we would be able to create 3D images with equipment originally not suitable for that. To achieve that, a video is recorded, in which the camera moves around the object, then the displacement of the pixels is calculated with optical flow. We can deduce the distance from the camera if the displacement is known (pixels of objects far away have smaller velocity than that of closer objects).

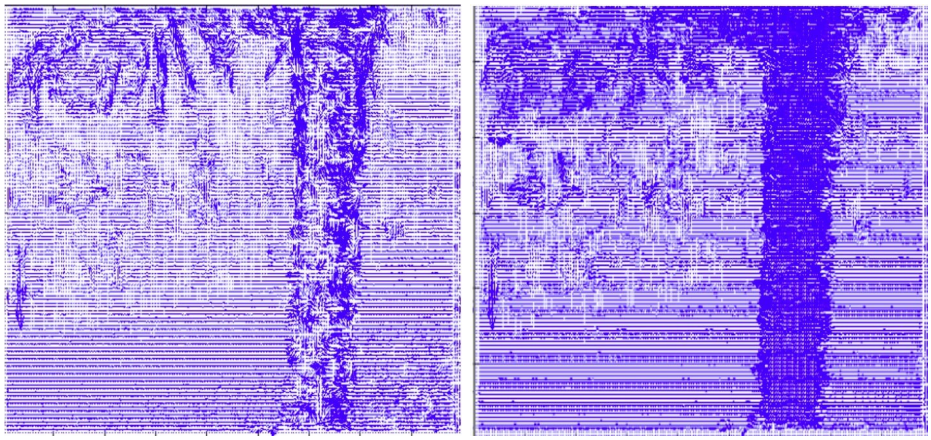

Figure 5.13: *Optical flow with (right) and without(left) the pyramid approach.*

Another interesting approach is the classification of movement events. Namely, moving objects can be assigned to different classes based on the optical flow image (e.g., moving in different directions, rotating, getting closer/further, etc.). This can turn out to be very useful if collisions with or the approximation of the object with the camera should be avoided.

5.4 Hidden Markov Model

Optical flow can be used to determine the displacement of objects by calculating the relative displacement w.r.t. the previous frame. It would also be reasonable to use the previous position for object tracking based on traditional identification techniques to achieve better results with that information. For that purpose, Hidden Markov Models (HMM) are often used.

Hidden Markov Models are based on Markov processes. A process is called a Markov process if the next state depends only on the actual state, i.e., the states encode perfectly the whole history of the process. Discrete Markov Models with a finite state space can be described with a state graph.

In the case of Hidden Markov Models, the states cannot be observed directly, only their consequences are observable. A good example would be the following: we would like to deduce the weather (sunny, rainy) based on the activities of a human (walking, shopping, cleaning) - here the hidden states will be the weather states, the observations will be the activities.

Two typical problems for Hidden Markov Models can be solved. The first is to determine the probability of a sequence of a hidden state. If the state transition probabilities are known, this task is trivial to solve. The second, more interesting task is to find the most probable sequence of hidden states if an observation sequence is available - the solution can be obtained, e.g., with the Viterbi-algorithm.

For object tracking, the hidden state is the position of the object, state transitions can be chosen in multiple ways. It is possible to choose a uniform distribution, but it is reasonable to choose a (quite wide) normal distribution because bigger movements are less probable. The observations of the model are the predicted position (obtained with a method described previously). We can assume that the probability of describing the relationship between the state and the observation is also Gaussian-like. In that case, it would be reasonable to use a distribution with a wider tail (e.g., the Student T distribution).

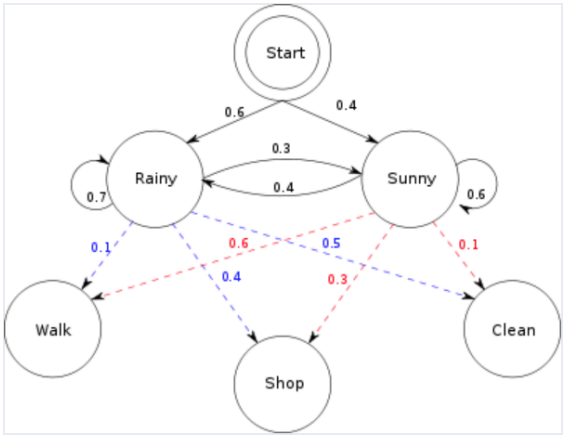

Figure 5.14: *Hidden Markov Model: here hidden states describe the weather, observations the activities of an individual.*

5.5 Kalman-filter

It may happen that for the object position, we have predictions from two different sources. It is assumed that both have a Gaussian distribution (the variances are different in general). In that case, we would like to exploit the additional information to obtain a result with smaller variance. This can be done as follows:

$$\begin{aligned} \mu &= \frac{\sigma_1^2\mu_0 + \sigma_0^2\mu_1}{\sigma_0^2 + \sigma_1^2} = \mu_0 + k(\mu_1 - \mu_0) \\ \sigma^2 &= \frac{\sigma_0^2\sigma_1^2}{\sigma_0^2 + \sigma_1^2} = \sigma_0^2 - k\sigma_0^2 \\ k &= \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \end{aligned} \tag{5.11}$$

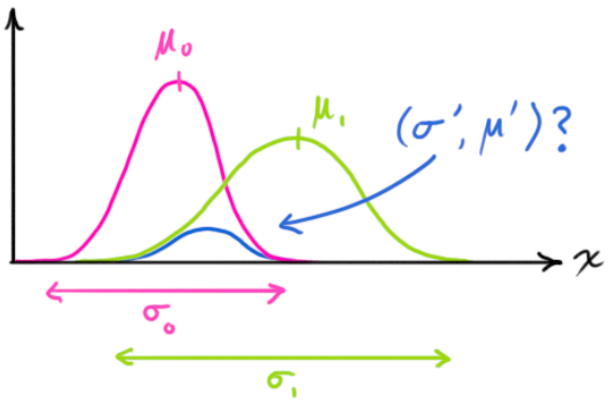

Figure 5.15: *The combination of predictions with Gaussian distributions.*

The intuition behind the formulas is that the new expected value is the weighted average of the expected values of both predictions, where weights are proportional to the reliability of the predictions. The reliability of a prediction is inversely proportional to its variance, thus the change of indices in the numerator. The reliability of the resulting prediction is the sum of the reliabilities, i.e., the standard deviation can be calculated with the replus operation. For multivariate Gaussian distributions, the only difference is that instead of variances the covariance matrix is used.

$$\begin{aligned} \vec{\mu} &= \vec{\mu}_0 + K(\vec{\mu}_1 - \vec{\mu}_0) \\ \Sigma &= \Sigma_0 - K\Sigma_0 \\ K &= \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \end{aligned} \tag{5.12}$$

The k and K coefficients are the so-called Kalman-gains.

In the case of the Kalman-filter, we have two estimates, one is from measurements, the other is from the prediction step, which uses the (known) system dynamics. For linear systems, the system dynamics can be given with the following state equation:

$$\begin{aligned} x_k &= F_k x_{k-1} + B_k u_k \\ y_k &= H_k x_k \end{aligned} \tag{5.13}$$

If the estimation of the previous state is known $N(\hat{x}_{k-1}, \Sigma_{k-1})$, then the prediction will be the following:

$$\begin{aligned} \hat{x}_k &= F_k x_{k-1} + B_k u_k \\ \Sigma_k &= F_k \Sigma_{k-1} F_k^T + Q_k \end{aligned} \tag{5.14}$$

Where Q_k is the uncertainty of the prediction. Since the output of the system often cannot be measured, we need an output estimation (using the state estimation results), this can be formulated as given below:

$$\begin{aligned} \hat{y}_k &= H_k \hat{x}_k \\ \hat{\Sigma}_k &= H_k \Sigma_k H_k^T \end{aligned} \tag{5.15}$$

Thus the measured output $N(z_k, R_k)$ and the estimation can be combined using the formula from above:

$$\begin{aligned} K_k = H_k \hat{\Sigma}_k H_k^T (H_k \hat{\Sigma}_k H_k^T + R_k)^{-1} \quad &\leftarrow \quad a\ Kalman\ gain \\ H_k x_k = H_k \hat{x}_k + K_k(z_k - H_k \hat{x}_k) \quad &\leftarrow \quad calculating\ \mu \\ H_k \Sigma_k H_k^T = H_k \hat{\Sigma}_k H_k^T - K_k H_k \hat{\Sigma}_k H_k^T \quad &\leftarrow \quad calculating\ \Sigma \end{aligned} \tag{5.16}$$

At this point, however, we estimated the output y and not the state x. Note that H_k can be eliminated by multiplying with the inverse of it from the left (one of the H_ks is in the formula of K), this is the case for H_k^T from the right in the equation of the covariance matrix. So the final estimation is expressed as:

$$\begin{aligned} \hat{K}_k &= \hat{\Sigma}_k H_k^T (H_k \hat{\Sigma}_k H_k^T + R_k)^{-1} \\ x_k &= \hat{x}_k + \hat{K}_k(z_k - H_k \hat{x}_k) \\ \Sigma_k &= \hat{\Sigma}_k - \hat{K}_k H_k \hat{\Sigma}_k \end{aligned} \tag{5.17}$$

5.6 Tracking multiple objects

Until now, only one object was tracked; for multiple objects, it can happen (first of all if the paths of them are crossing), that it is ambiguous how to assign the detected objects to the known objects from previous frames. The problem can be formulated as an optimization task:

$$max_x \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_{ij} \quad where \quad \begin{cases} \sum_j x_{ij} = 1 & \forall i \\ \sum_i x_{ij} = 1 & \forall j \\ x_{ij} \in \{0, 1\} \end{cases} \tag{5.18}$$

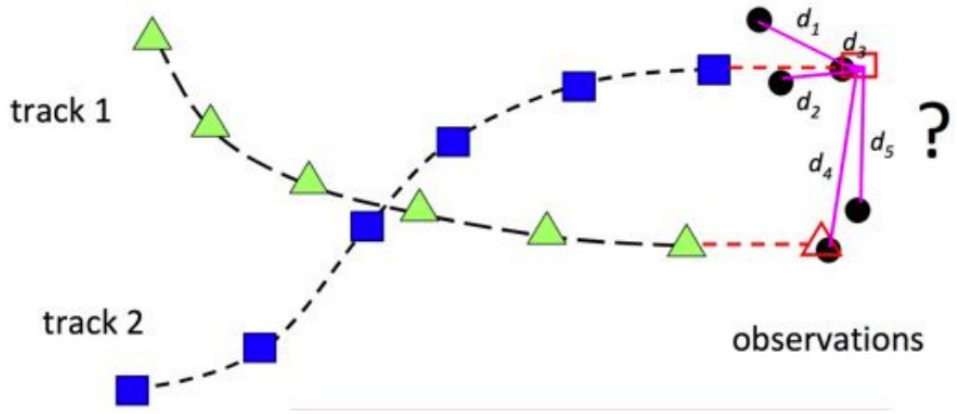

Figure 5.16: *Tracking multiple objects.*

Where w_{ij} is inversely proportional to the difference between the properties of the previously known and the in the current frame detected objects. For identification, several features can be used; the most simple are the position, bounding rectangle, or ellipse. It is also possible to use the local image features discussed in Lecture 3 or the binary descriptors of Lecture 5. The affinity can be defined in every case, which describes the similarity between the features used for the description of two objects.

$$A(x,y) = e^{\frac{-1}{2\sigma^2}||f(x)-f(y)||^2} \tag{5.19}$$

Where $f(x)$ is the feature vector describing x. This measure can be used as a weight in the formula above.

This task, which is also known as the pairing problem, can be solved with the Hungarian algorithm if we have only one frame. For multiple frames, the optimal solution is not guaranteed - in that case, the paths of the objects can be described with a directed, weighted graph. Optimal trajectories can be determined with the min cut-max flow algorithm, which intends to cut the graph into n parts so that the weights of the cut edges are minimal - while in each n subgraphs there should be connections between the starting and final vertices and the weights of these edges should be maximal.

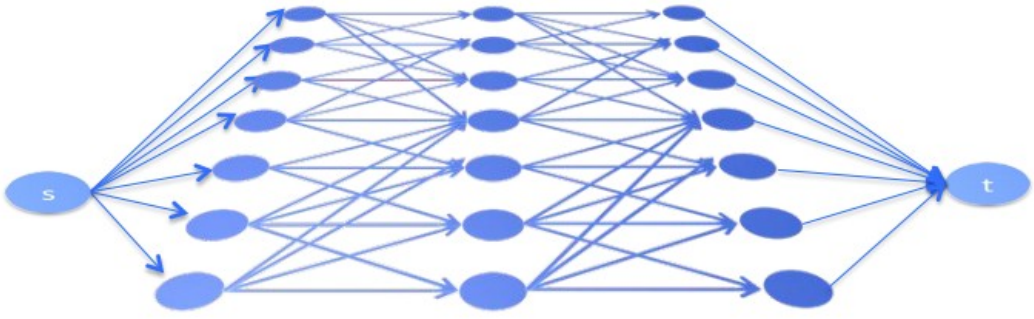

Figure 5.17: *The graph of tracking through multiple frames. Each level of the graph belongs to a different frame, each vertex corresponds to an object detection.*

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

[9] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, IEEE Comput. Soc. DOI: 10.1109/cvpr.2001.990517. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2001.990517.

[10] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, Sep. 2010. DOI: 10.1109/tpami.2009.167. [Online]. Available: https://doi.org/10.1109%2Ftpami.2009.167.

[11] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972. DOI: 10.1145/361237.361242. [Online]. Available: https://doi.org/10.1145%2F361237.361242.

[12] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967. DOI: 10.1109/tit.1967.1054010. [Online]. Available: https://doi.org/10.1109%2Ftit.1967.1054010.

[13] H. W. Kuhn, "The hungarian method for the assignment problem," in *50 Years of Integer Programming 1958-2008*, Springer Berlin Heidelberg, Nov. 2009, pp. 29–47. DOI: 10.1007/978-3-540-68279-0_2. [Online]. Available: https://doi.org/10.1007%2F978-3-540-68279-0_2.

6 Segmentation

6.1 Introduction and methods

In practical computer vision applications, it is often required to assign each pixel in an image to a specific object, i.e., to carry out image segmentation. The output of which is generally a binary or multistate image (i.e., multiple values are present), this can be used as a mask to highlight or crop out the original object from the image. Segmentation also can be used for object recommending methods, mentioned previously.

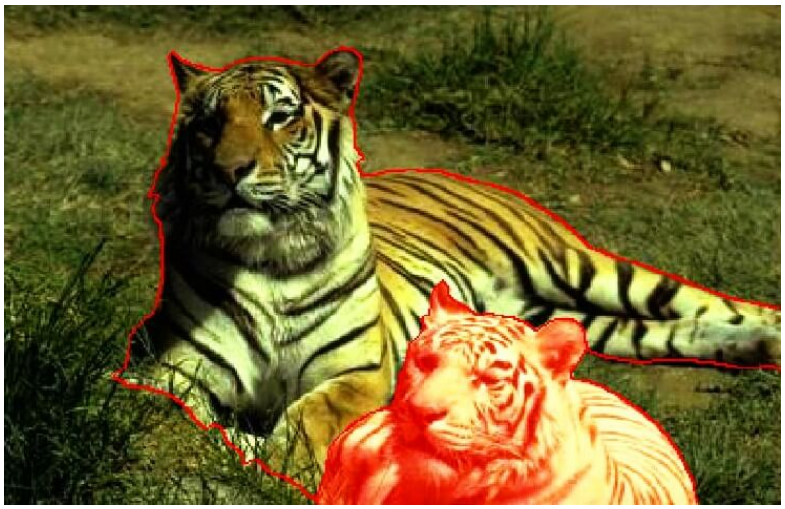

Figure 6.1: *The segmentation task .*

Based on the input (i.e., the information we used during the process) and the output, several different variants of segmentation can be distinguished. If the pixels belonging to a known object should be found, then it is called foreground-segmentation. In the case of traditional segmentation, there is no particular foreground object, because our goal is to separate the image based on the different objects present. If segmentation does not occur on an object, but on an object class/category basis, then the procedure is called semantic segmentation - which means that two adjacent objects s belonging to the same class are merged into one segment. If besides object classes the instances are also important to distinguish, then we use methods called instance segmentation.

Several segmentation methods exist, the simplest of which are using color or intensity, such as thresholding or clustering. Nevertheless, there exist procedures which use homogeneity criteria (i.e ., rules based on which it is unambiguous whether a pixel belongs to a segment or not), these are the region-based methods. Other methods include edge-, movement- or graph cut-based solutions.

6.2 Thresholding

Foreground-segmentation can be carried out - in the simplest way - based on color or intensity information. In controlled, simple environments, it can often be guaranteed that the object of interest is the only of that color, which makes it possible to use only color channel-based thresholding methods to achieve satisfying segmentation results. Of course, the binary image resulting from

Figure 6.2: *The result of thresholding (left) and multivalued thresholding (right) .*

this step should be filtered with binary image processing algorithms. It is important to note, both color- and intensity-based segmentation uses only HSV or YCbCr color spaces (or their variants).

In the case of thresholding, the threshold values are often chosen by the designers, which may result in a suboptimal solution. To avoid this problem, the method of histogram-backprojection is often used. At the beginning of this procedure, a histogram is made from the original image, which is compared during the process to the processed image. After backprojection, a probability is determined for each pixel, indicating how probable it is that the pixel is from the reference histogram. Thresholding this results in a probability mask containing the pixels which belong to the original image.

It is also possible to use the histogram of the image to be segmented with traditional segmentation, i.e., without a reference object. In this case, valleys and peaks are determined in the histogram, splitting up the pixels into different groups, which can also be done in multiple dimensions in a parallel manner. This means that both color and intensity information is used at once.

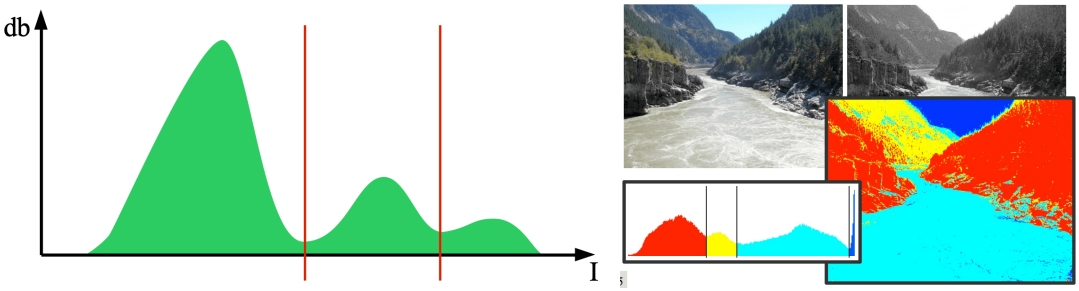

Figure 6.3: *The principle of histogram-based thresholding (left) and its result (right) .*

Application: several virtual and augmented reality system use gestures to implement one way of human-computer interaction. Nevertheless, there are such solutions that utilize special gloves (equipped with sensors); on the other hand, camera-based methods are at least as popular way to recognize gestures. For the proper function of the latter, the segmentation of the hand is needed, which can be easily realized with the help of the histogram-backprojection algorithm (using skin tones as the target color). One constraint of this solution is that the cameras should not see other bare body parts. In general, this can be guaranteed for head-mounted vision systems (e.g., glasses equipped with cameras).

One big problem regarding color-based segmentation is the in some situations subjective interpretation of colors. A suitable illustration of the problem is the picture called Dress, which conquered the Internet in 2015. This image depicts a dress with two different colors, surprisingly, for several

people, it has white and gold, for the aothers blue and black colors. The explanation of the phenomenon is not clear until now, the most widely accepted hypothesis argues that the individually different compensation of the color-distorting effect of natural light is the reason behind that.

Figure 6.4: *A Dress .*

6.3 Clustering

A similar principle is used for clustering-based segmentation algorithms as in the above mentioned, histogram-based case. Earlier, in the case of the visual bag of words classification, clustering was already mentioned, the idea behind which is to assign the points of a set in an arbitrary space to subsets (i.e., clusters) so that they optimally satisfy some compactness criterion. Although several algorithms exist, the k-means algorithm is the most widely used.

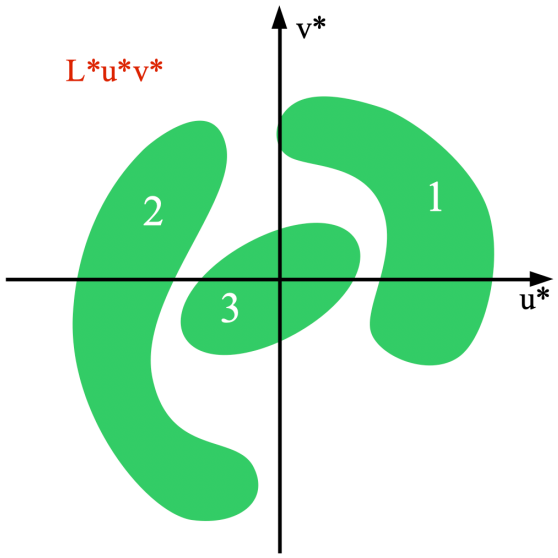

Figure 6.5: *The principle of clustering .*

6.3.1 K-means

The k-means algorithm assigns the points into k clusters so that the squared distance of each element s of the cluster w.r.t. the center of the cluster is minimal. To achieve that, an iterative algorithm is used, which is initialized by selecting k centroids in the space defined by the points. After that, the following two steps are repeated until convergence: first, each point is assigned to the nearest centroid, second, the centroids are recalculated as the arithmetic mean of the points of the cluster. The consequence of the second step is that the position of the centroids is changed, and so is the mapping of the points to the clusters .

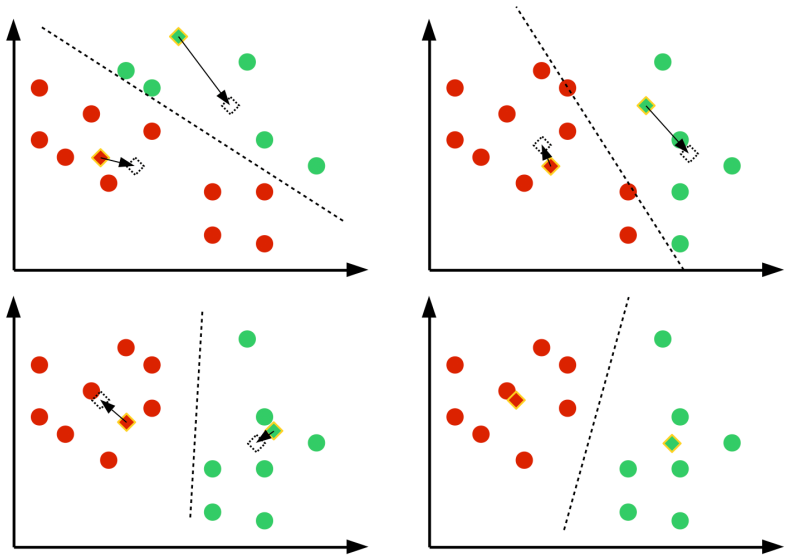

Figure 6.6: *The principle of the k-means algorithm .*

The iteration is continued until convergence, which is proven to be independent of the initialization value. The stopping criteria can be e.g., how many points are assigned to different clusters compared to the previous step or a measure, which determines how much the position of the centroids is changed. It is important to emphasize that the k in the name of the method, i.e., the number of clusters is a design parameter, the choice of which has a significant effect on the performance of the algorithm. Namely, the error can be decreased with a higher value of k (up to N, the number of points, when the error will be zero) - nevertheless, it makes no sense to assign each pixel of an image to a different cluster. To obtain the best value, it is worth plotting the error w.r.t. k, a reasonable choice would be the k value at the elbow point of the curve, i.e., where the error does not reduce in a significant manner if k is further increased.

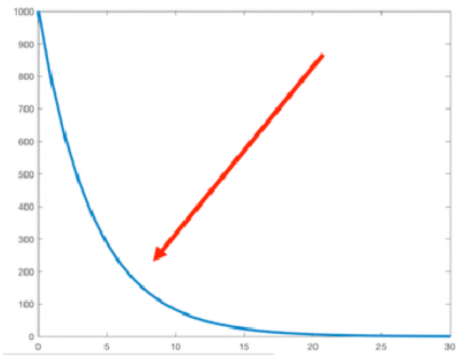

Figure 6.7: *The selection of the number of clusters. Although the error decreases monotonously, the reasonable choice would be the elbow point (marked with the red arrow), because the error values decrease significantly until that point.*

6.3.2 Mixture of Gaussians

In the case of the k-means algorithm, each point is assigned to one cluster, although it could also be possible that some points belong to multiple clusters with different probability. To handle this situation, soft or fuzzy clustering can be applied, such as the Mixture of Gaussians (MoG) algorithm. The MoG algorithm describes the point set with k independent Gauss-distributions. The aim is to determine the parameters of the distributions so that the probability of the point set is maximal; to achieve that, the so-called Expectation-Maximization (EM) algorithm is used.

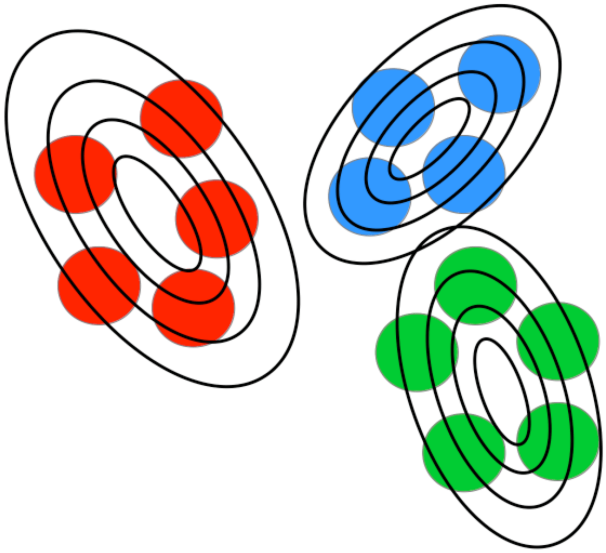

Figure 6.8: *The principle of MoG clustering. The figure is a contour plot of the Gauss-distributions .*

The EM is a two-step iterative algorithm, starting with randomly initialized distributions and iterating until convergence. The steps are the following:

1. **Expectation:**Each point is assigned to that normal distribution which assigns the highest probability to it.
2. **Maximization:** The parameters of each distribution are predicted with the Maximum-Likelihood (ML) method, based on the assigned points .

The parameters of the normal distribution, containing the points X, can be predicted in the following way (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 concluded that the steps of EM and k-means are quite similar, although there is a crucial difference: MoG can result in clusters with different width and shape, which is not possible for k-means.

6.3.3 Mean Shift

The last important clustering algorithm is called mean shift, for which first a weighting kernel is determined. In the case of image segmentation, it is generally a Flat- or Gauss-kernel. Besides the type of kernel, its size also should be determined, which influences the fineness of the result.

It is important to note that the number of clusters should not be chosen for mean shift because it is determined automatically during the process - which is a clear advantage compared to the previous methods.

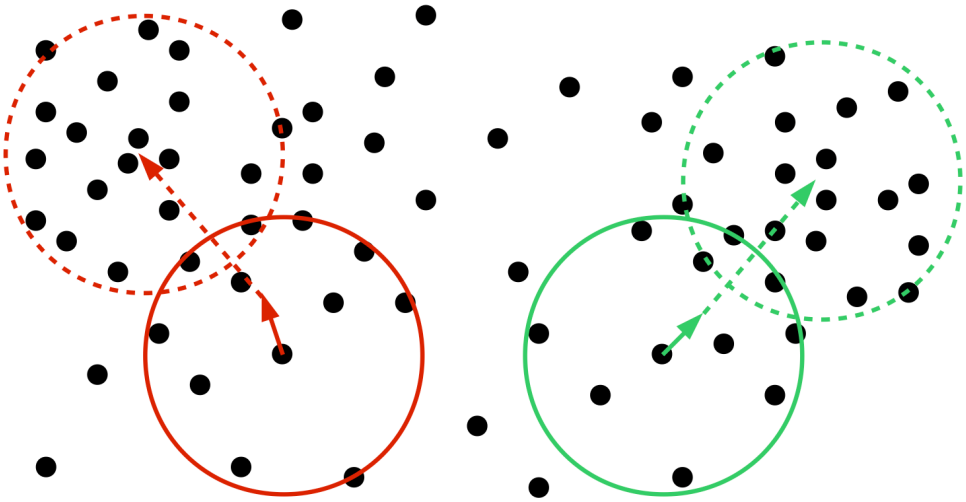

Figure 6.9: *The principle of mean shift clustering .*

During the process, the kernel is slid over the points, and in each position, the average of the covered points is calculated, weighted with the kernel. As a result, the point is shifted to the position of the previously calculated mean. This step is repeated until convergence.

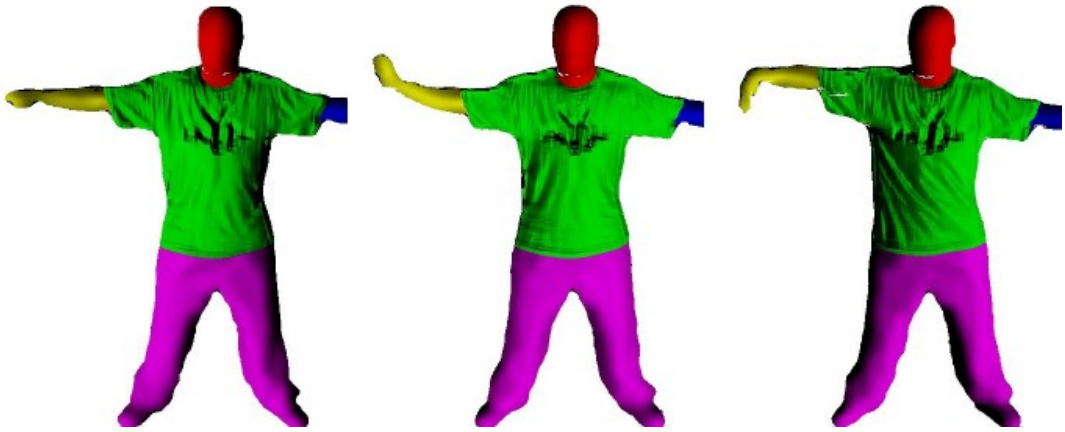

Figure 6.10: *The result of the mean shift algorithm .*

6.4 Region-based methods

The methods mentioned above use only either color or intensity information, which is their biggest disadvantage. Because the majority of objects is contiguous, it is crucial to have contiguous segments as a result. To justify this criterion, region-based methods are used, such as the algorithm of region growing and region splitting or the graph cut-based segmentation algorithm.

6.4.1 Region growing

In the case of region growing, the starting step is the selection of some core points in the image. Those region cores can be chosen based on their intensity, or they can be equally spaced on a grid

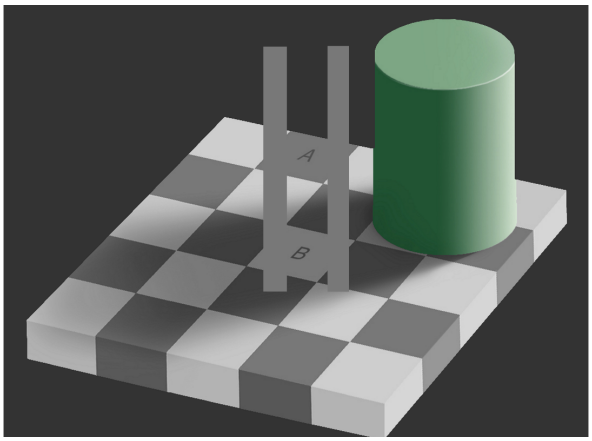

Figure 6.11: *The disadvantage of using only color/intensity: due to distortions, regions belonging to different clusters have the same color/intensity, thus they are assigned to the same cluster .*

- they will be the starting points for the regions. After that, the unlabeled neighbors are examined by evaluating a region-membership function. If the result is positive, the point is assigned to the region, otherwise not. The algorithm terminates if no further points can be added to the regions.

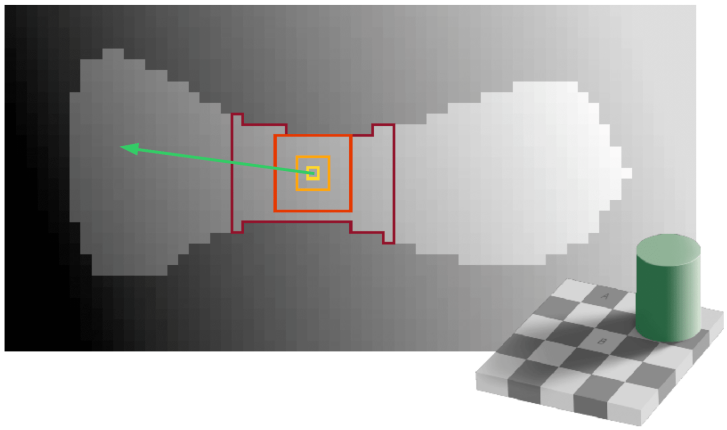

Figure 6.12: *The principle of region growing .*

The result of region growing is influenced by several factors, e.g., the selection of the starting points - a feasible choice would be to select them based on the histogram of the image. If the algorithm results in some small regions, they should be neglected, because they are probably the consequence of local noise or error. The most important factor is the region-membership criterion. Generally, the intensity-difference of neighboring pixels or that of the previous centroid value and the inspected pixel is chosen (using a threshold value). In special cases, similarities of color or texture can also be used.

6.4.2 Split & Merge

To solve this problem, the method of Split & Merge can be used, which - as a starting point - interprets the whole image as one region. After that, in an iterative process, each segment is evaluated based on a homogeneity criterion (determined by the designer). If the segment passes the test, it remains one; otherwise, it is split into four equally sized regions. After that, the evaluation is continued recursively on the new segments, until each segment satisfies the homogeneity criteria.

During the split step, it is probable that homogeneous regions are split up (due to the fact that while determining the smaller regions, possible boundaries were not taken into consideration). So

after splitting, a merging step is also carried out, which utilizes, similarly to the splitting step, a homogeneity criterion. These two steps together form a fast algorithm that is based on global information and is, more importantly, much more robust regarding noise and almost invariant to the choice of neighborhood interpretation. The disadvantage of the split & merge method is that - due to the square-based splitting step - the border of the segments will not always be perfect.

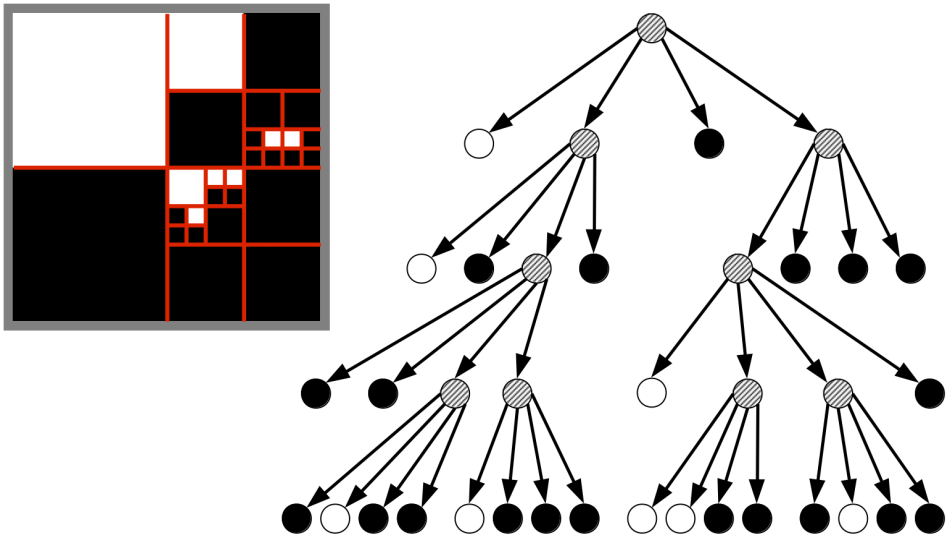

Figure 6.13: *The principle of the split & merge algorithm .*

6.5 Movement segmentation

In the case of videos, the detection and segmentation of movements is an important task, which can be interpreted as one variant of foreground-segmentation, where segmentation is based on movement, not on color or intensity. One of the simplest methods for movement detection is the calculation of the difference image between two images, taken generally not immediately after each other but with a 0.5–1 s delay. To highlight the pixels where the change was significant , thresholding can be used. The decision-making is based on the area of the ”1” pixels in this binary image.

The above-mentioned, rather simple method has several drawbacks, e.g., in the case of a moving object, movement is detected in both (i.e., the previous and the new) positions, thus resulting in ghosting. Another disadvantage of the method is that in the case of changes, which does not imply movement but modify the intensity values, a false positive detection will be made. A typical example is the change of illumination, which is a significant disturbing factor for outdoor videos (in the indoor case, the automatic white balance function can result in the same consequences).

These disadvantages can be eliminated with the use of a Gauss background model. This method aims to separate the background (treated as being static) from the moving foreground. To achieve that, a statistical model of the background is built (with moving averaging); for each pixel mean and standard deviation are calculated. After that, the difference image is formed between the actual and the background image, which is thresholded. The result of the algorithm contains no ghosting, and additionally, slow intensity changes (e.g., the sun begins to shine) can be incorporated into the background model, without a false positive detection.

The standard deviation values of the pixels can be used for choosing the threshold value adaptively. So it is possible to design an algorithm that is less sensitive in such areas, where movement is frequent, and more sensitive, where it is rare. This approach can be useful if large areas are containing frequent but irrelevant movement, e.g., bushes or trees in windy weather.

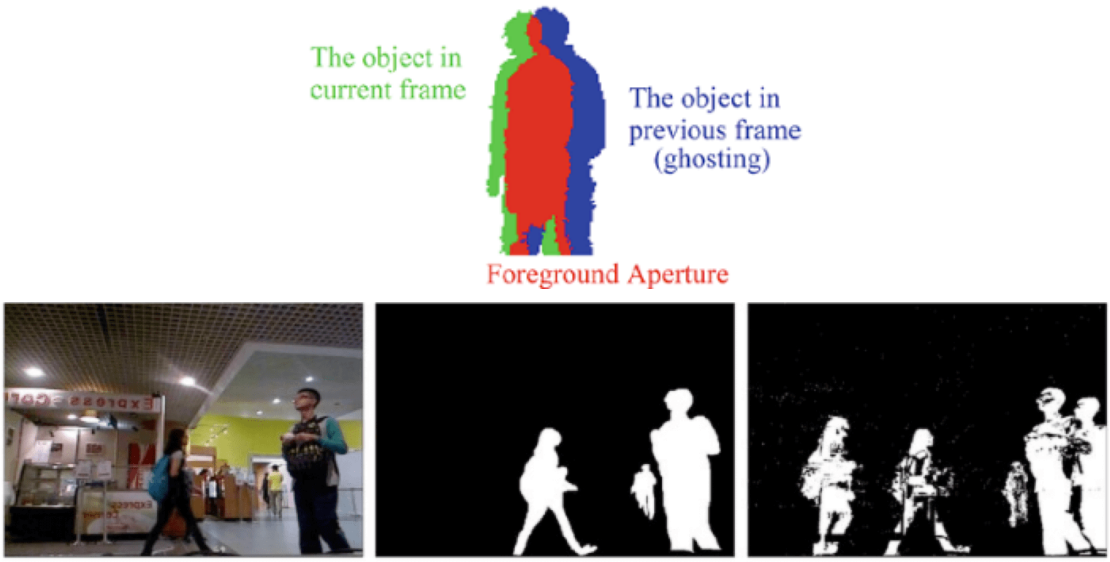

Figure 6.14: *Ghosting .*

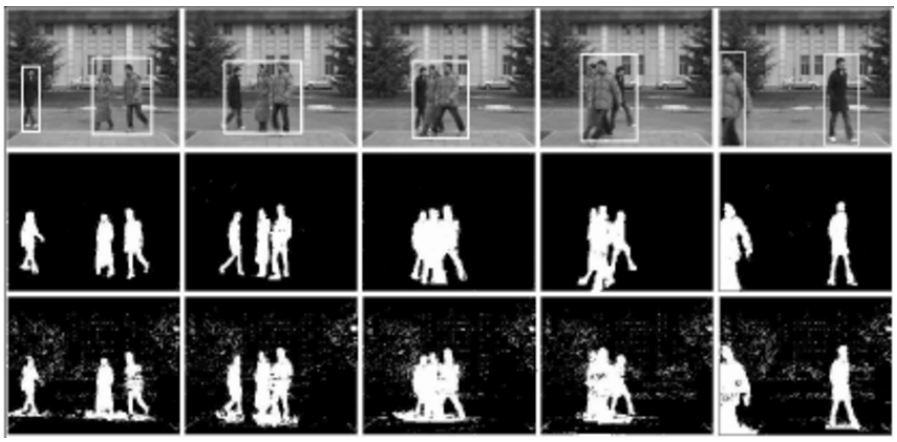

Figure 6.15: *The principle of the Gauss background model-based movement segmentation. In the middle row, the images were calculated with the variance-based difference threshold, in the bottom row the threshold used was constant .*

Application: Surveillance and safety applications should be able to store the recorded footage for future use. Storing everything would result in an enormous amount of storage, mainly due to the size of the videos and the prescribed redundancy. Significant savings can be achieved if only those footage parts are stored where movement was detected (static scenes are irrelevant for us). The functioning of these systems can be significantly enhanced with the more noise-robust background model method; first of all, this can help significantly in the case of outdoor footages.

6.6 Graph Cut

Graph cut algorithms represent a different approach, so they are also worth mentioning. The common of these methods is that they describe the image as a weighted, undirected graph, where vertices correspond to pixels/local pixel groups. The edges of the graph are only given between adjacent pixels/pixel groups; the weights of them express the difference between pixels/pixel groups. The principle of such methods is that the constructed graph is cut into more parts so that the cost of the cuts should be minimal. The cost is determined based on the weights of the from the graph deleted vertices.

Application: Segmentation methods can also be used for so-called superpixel-segmentation . A

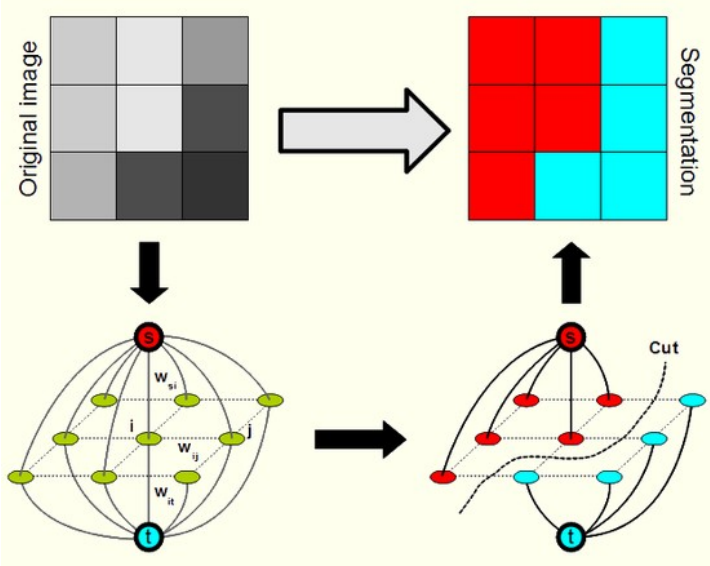

Figure 6.16: *The principle of graph cut segmentation .*

superpixel is a relatively small, compact image detail, which is homogenous (from a specific point of view). Big, complicated objects consist of generally several superpixels. For this task, the same algorithms can be used; the only difference is that the parameters should be finetuned in a manner that they should find small segments. Of course, there are specific methods designed for this task, as well. Superpixels are very useful in several applications, e.g., in the case of manual image labeling (which is required for training machine learning models, for the training datasets), this method can significantly accelerate the process.

6.7 Watershed

The last algorithm discussed here is called the watershed method, the principle of which is that intensity values are interpreted as a contour plot (lighter pixels mean higher levels). After local minima are found, the image is "flooded" starting from them. When the flows originating from multiple minima become adjacent, then the separation line (i.e., the border of the segments) is found.

The main advantage of the watershed algorithm is that makes manual intervention easily possible because the starting points can also be determined manually, which means that even for structurally complicated images, good results can be achieved.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

[14] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995. DOI: 10.1109/34.400568. [Online]. Available: https://doi.org/10.1109%2F34.400568.

[15] R. Nock and F. Nielsen, "Statistical region merging," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, pp. 1452–1458, Nov. 2004. DOI: 10.1109/tpami.2004.110. [Online]. Available: https://doi.org/10.1109%2Ftpami.2004.110.

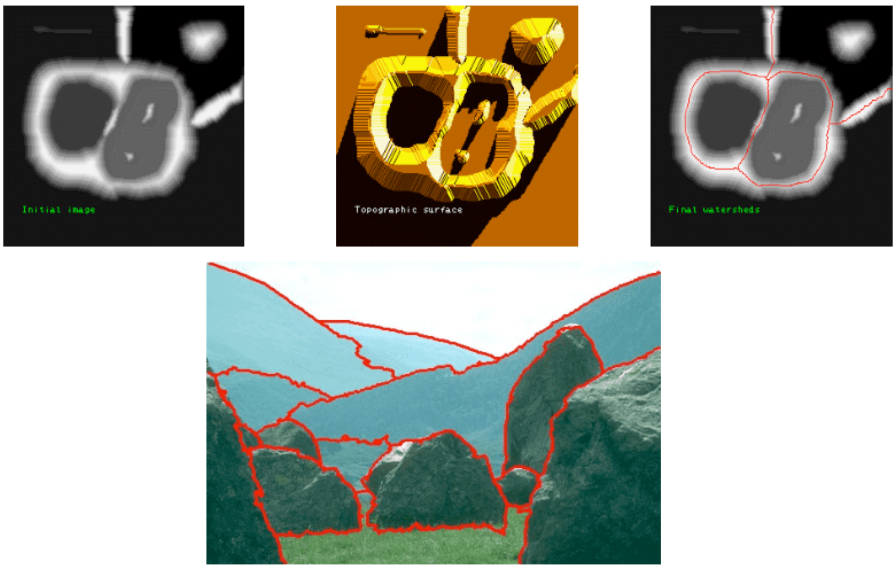

Figure 6.17: *The principle of the watershed algorithm .*

Figure 6.18: *Watershed algorithm with markers .*

[16] C. Couprie, L. Grady, L. Najman, and H. Talbot, "Power watershed: A unifying graph-based optimization framework," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 7, pp. 1384–1399, Jul. 2011. DOI: 10.1109/tpami.2010.200. [Online]. Available: https://doi.org/10.1109%2Ftpami.2010.200.

[17] L. Najman and M. Schmitt, "Watershed of a continuous function," *Signal Processing*, vol. 38, no. 1, pp. 99–112, Jul. 1994. DOI: 10.1016/0165-1684(94)90059-0. [Online]. Available: https://doi.org/10.1016%2F0165-1684%2894%2990059-0.

7 Binary images

7.1 Introduction

As mentioned earlier, several different types of images exist, most often grayscale (one channel) and color images (three channels) are used. Nevertheless, for some tasks, the use of binary images is the standard, e.g., in the case of edge detection (i.e., edges have the value of 1, the rest the value of 0). SSuch binary images can, of course, be the result of different operations, e.g., color detection or complex tasks like object detection.

In the current subchapter, such binary images will be analyzed where the value 1 represents the relevant part of the image (i.e., the object), while 0 represents the background. Please note that although for display purposes for the binary values, the black and white colors are used, the assignment of them to the values is not consistent in the literature - here, white is used to denote the object.

7.2 Morphology

One important family of algorithms is called morphology, which handles the shapes of objects. Theses algorithms are first of all used to carry out image enhancements.

7.2.1 Erosion and dilation

In practice - independent of the degree of sophistication of the algorithm used - results are never perfect, thus - as it is in the case of grayscale and color images - image enhancements are needed. Since the errors occurring in binary images are different, so are their corrections. Two types of errors can occur in this case: either background pixels are labeled as an object, or some of the object pixels are labeled as background. As a result of the former, we may see objects not present in reality or separate objects are merged. Due to the latter, we may experience holes inside some objects, or a contiguous object is mistakenly separated into parts.

To correct the above-mentioned two types of binary errors, the operations of dilation and erosion can be used. For that, the definition of a structuring element is needed, which a window of arbitrary shape, which is - like in the case of convolution - matched in each position to the image, and a logical operation is carried out on it. In contrast to convolution kernels, the values of the structuring element can only be 0, 1, or undefined (”Don’t care”). Between binary images or a binary image and a structuring element, the traditional logical operations can be defined:

The result of erosion is only then a 1 if the structuring element matches perfectly the underlying part of the image, i.e., each pixel value is equal to that of the structuring element. In the case of dilation, the result will be 1 if there is at least a match in one position. If the structuring element used for erosion consists of only 1 values, then the borders of an object will be set to 0, i.e., the size shrinks. For dilation, the result is the opposite; the object will be bigger.

Of course, these operations can use an arbitrary structuring element, which can be used to achieve special filtering effects. A narrow, rectangle-shaped structuring element, e.g., removes edges which are perpendicular to its longer side, nevertheless, edges parallel to the structuring element are retained. Similarly, with a structuring element of a given shape, objects can be filtered out which do not match the shape of the structuring element. E.g., this approach can be used to highlight corners on a binary image.

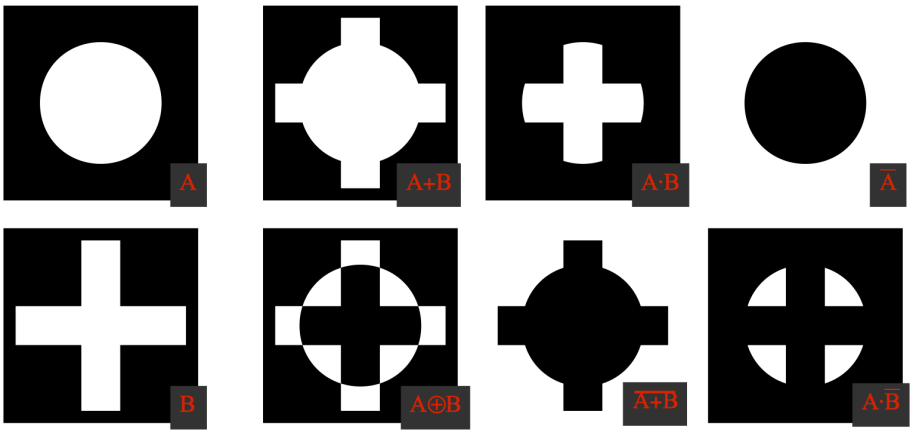

Figure 7.1: *Logical operations on binary images. These are carried out most often in a pixelwise manner .*

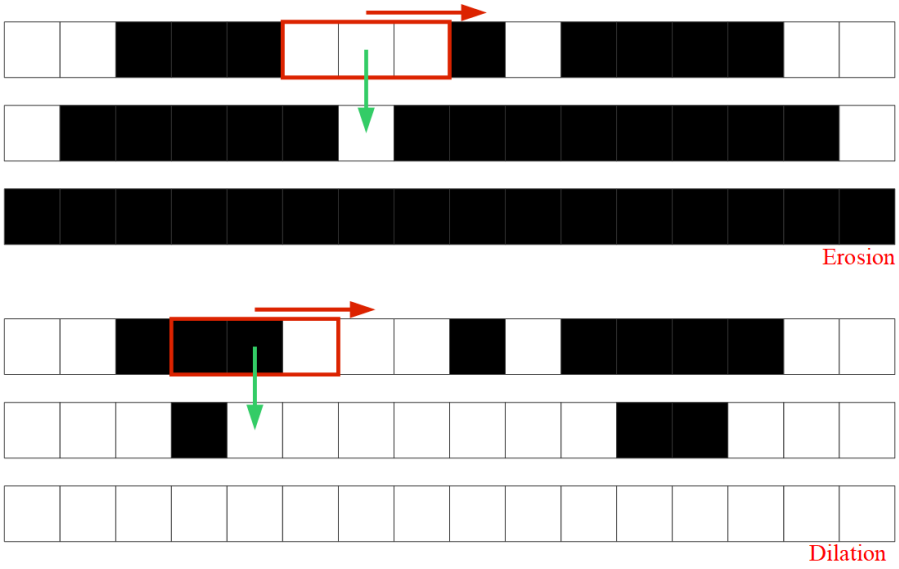

Figure 7.2: *The principle of erosion and dilation .*

Figure 7.3: *The operation of dilation and erosion can be applied to grayscale images, too. In that case, dilation works as a maximum (middle), erosion as a minimum filter (left) .*

7.2.2 Opening and closing

The main drawback of erosion and dilation is that they modify the size of the objects (thus, the scale of their shape will also be modified), so measurements taken after the application of them

results in inaccurate results. To handle this problem, we never use them individually, but we combine them; the most widely-used combinations are the operation sof opening and closing. In the case of opening first erosion is applied to the image (for a given number of times), which is followed by the same number of dilations. The erosion step eliminates small, noise-like objects, while the dilation step regrows large objects to their original size. Important to note that the dilation operation used in opening avoids the merger of objects, i.e., a pixel with value 0 is only set to 1 if the number of independent components will remain the same after the operation. So it can also be achieved that objects which are slightly merged will be separated as a result of opening.

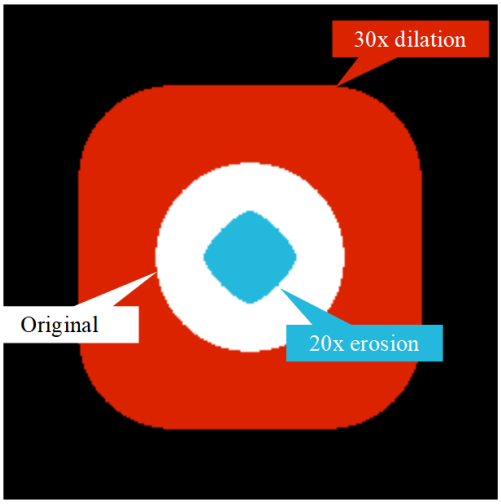

Figure 7.4: *The original (white) object with a circular shape suffers a significant change w.r.t size and shape after applying erosion and dilation to it with a square structuring element .*

The operation of closing is the opposite of opening. First, a given number of dilations are carried out, which fills in small holes inside the object, after that the same number of erosion is done to restate the original size of the objects. If both operations are applied after each other, both types of errors can be corrected.

7.3 Topology

After some initial enhancements we have a quite good-quality binary image, which can be used for measurements about the relevant objects. For this, we can use the results of topology, which describes the relationships between objects.

7.3.1 Pixel connectivity

Before starting with that, the topic of pixel connectivity should be investigated. To decide whether two pixels are connected to each other generally, two conventions are used: 4- and 8-connectivity. In the case of 4-connectivity, each pixel has 4 neighbors (up, down, left, right); for 8-connectivity, four additional neighbors are added to the four of the 4-connectivity case, namely the diagonal pixels.

Regardless of the choice of the pixel connectivity from the above two, it will hurt the Jordan-property of the image plane, which states that a plane is divided by a contiguous, closed curve into exactly two parts. This property seems to be obvious; thus, it would be reasonable to be true for the image plane. Nevertheless, as depicted in the first figure, if using 4-connectivity, the white pixels are not connected; nevertheless, they separated the background (denoted by the black pixels) into two parts. For the 8-connectivity case (middle figure), the white pixels form a contiguous, closed curve, but the background pixels also remain interconnected.

To satisfy the Jordan-property of the image plane, for one of the objects and the background 4-connectivity should be used, while 8-connectivity for the other. In the figure on the right, 8-connectivity is used for the white object pixels, so they form a closed curve, while 4-connectivity is used for the background, i.e., the pixel in the middle will not be a neighbor of any of the other black pixels; thus the background is separated into two parts. We assume that the background is repeated outside the image border, so the black pixels in the corners belong to the same contiguous part.

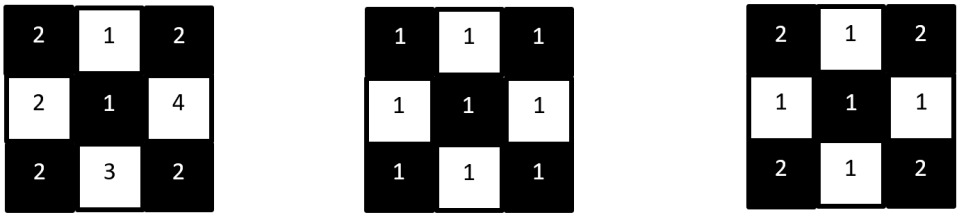

Figure 7.5: *Visualizing the Jordan-property. On the left (4-connectivity) we have 2 background and 4 object components. In the middle (8-connectivity) both the background and the objects are contiguous. On the right (background: 4-connectivity; foreground: 8-connectivity) the foreground is contiguous while the background is split into two parts .*

Note that not only the pixel interconnectivity but also the distance metric should also be specified to carry out measurements (first of all length). For that, we have several possible solutions: we can choose the 4-, 8-distance, or the Euclidean distance. Although the latter is more computationally intensive, the former two are quite inaccurate.

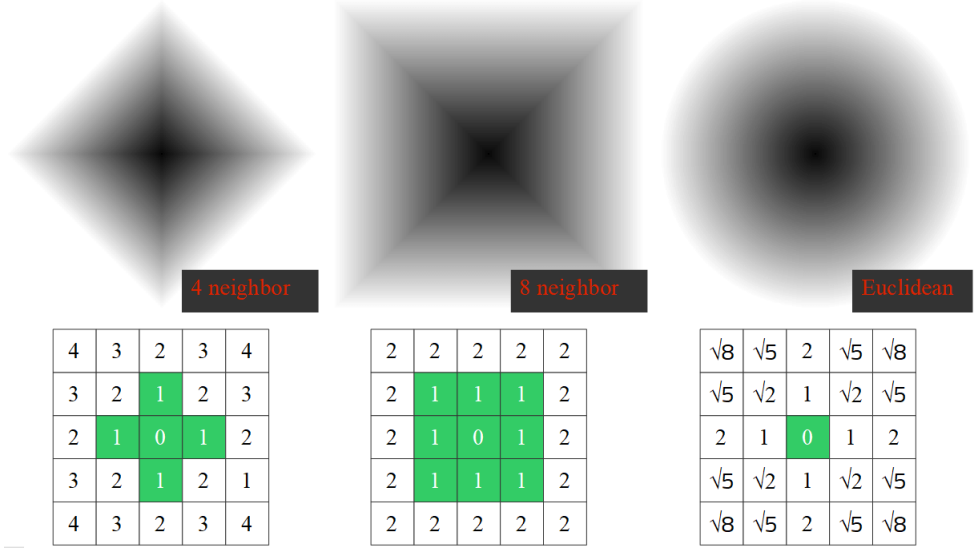

Figure 7.6: *Distance measures based on pixel interconnectivity and the Euclidean distance.*

7.3.2 Skeletonizing

Given the connectivity of the pixels, several image properties can be determined, one of which is the skeleton of the image. The skeleton is such a representation, where each pixel with value 1 has exactly one or two neighbors with value 1, from a topological point of view, it is equivalent to the original object. An alternative interpretation of the skeleton is the set of centers of circles with maximal radius, which can be placed into the object.

Most often, the skeleton is calculated with an iterative erosion algorithm (thinning). In that case, erosion is carried out only if critical points will not be deleted. Critical points are endpoints or

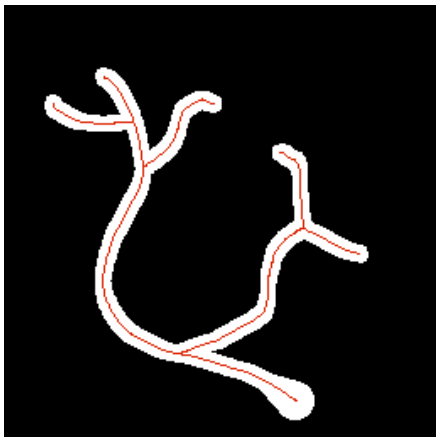

Figure 7.7: *The skeleton of a binary object.*

such points, the removal of which would separate the object into two. Fortunately, these types of pixels can be detected in local windows, which makes the algorithm simpler.

7.3.3 Object labeling and counting

The purpose of labeling and counting objects in a binary image is to assign a label to separate objects, and in the end, the value of the biggest label will equal the number of objects. For object labeling, two widely-used algorithms exist: one is recursive, the other sequential.

The steps of the simple recursive object labeling algorithm are below:

1. The first unlabeled pixel with value 1 has to be found and labeled with label L.
2. The assignment of label L to each neighbor of the evaluated pixel, which has a value of 1, and the call of the second step for each of them.
   (a) If there is no unlabeled pixel with value 1, the algorithm terminates.
3. Jump to step 1 and increment L.

Although the steps are rather simple, some problems may occur, e.g., if big contiguous objects are labeled, then the recursion depth may reach the depth of the stack, which is, first of all, a real problem for lightweight processing units. In those cases, it is more feasible to use the more complicated sequential algorithm. This solution evaluates each pixel one by one (it proceeds row by row), for each pixel of value 1, the following rules are evaluated:

- If only the above or left neighbor of the pixel is labeled, then that label is assigned to the current pixel.
- If the above and left neighbors have the same label, it gets copied.
- If the labels of the above and left neighbors differ, then the label of the above neighbor is assigned, and the equality of both labels is stored.
- If the pixel does not have a labeled neighbor then a new label will be assigned to it.

At the end of the sequential algorithm, a last run over all pixels is utilized to assign a common label to the pixels which belong to the same object (for that, the label equalities stored during runtime are used). This algorithm also can be used to count objects, namely, when a new label

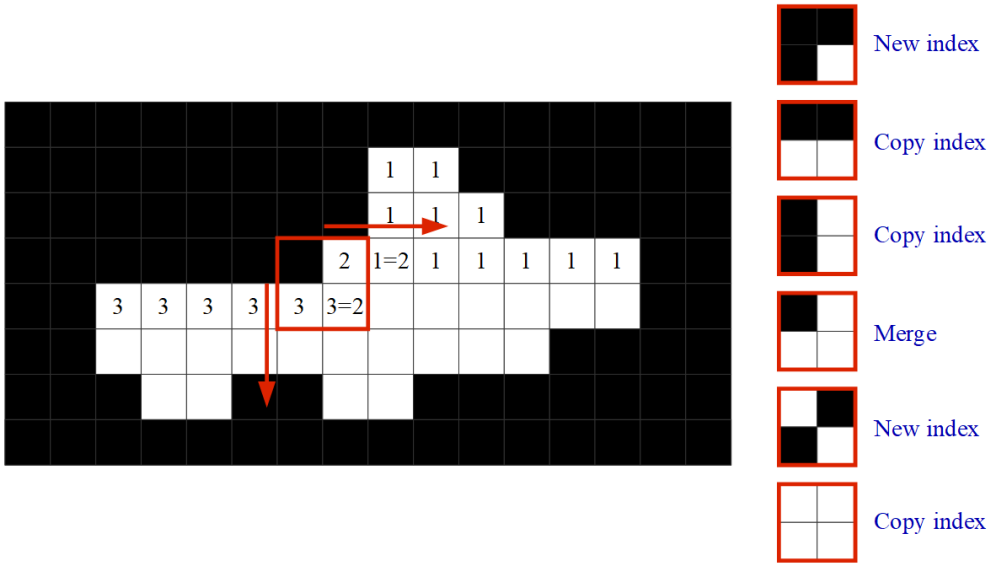

Figure 7.8: *The principle of the sequential object labeling algorithm.*

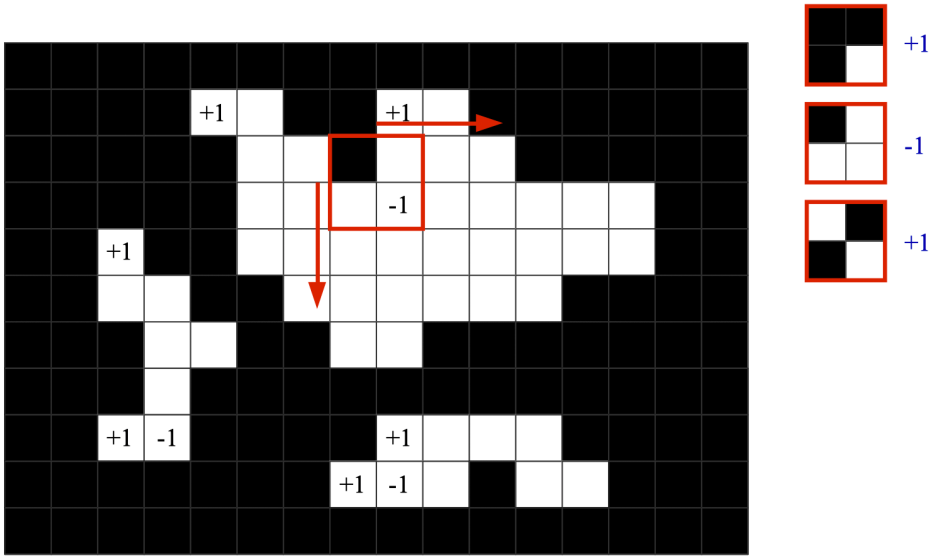

Figure 7.9: *The principle of the sequential object counting algorithm .*

was applied in the previous version, the number of objects will be incremented; when labels are merged, the number will be decremented.

Application: in practice binary image processing is used for the automatic counting of coins, which can accelerate such transactions. For that, a separate device is manufactured, where coins can be placed onto a sheet of a given color, which is observed with a camera from a known distance. Because the background color is given, thresholding can be used to binarize the image (coins will be marked with the value 1). The separation of coins and the elimination of holes can be done with a consecutively executed opening and closing step. With the help of labeling, the objects are separated, the area of which will help us to determine the denomination (i.e., value) of the coins. Of course, the authenticity of the coins can be determined with template matching too.

7.4 Object properties

After the separation of binary objects, the next step is to determine such properties for each object, which can be used to describe and identify them. To calculate them, often, the projections of the objects can be used. Projections are such concise representations, where the 1 values are counted in each column/row. The shape of the original object can be restored from several different projections - this is the base of tomography procedures used in medical imaging.

7.4.1 Position, orientation

Another important object descriptor is the momentum of objects. Momenta can have different orders; for us, the zeroth and first orders will be of interest. The zeroth-order momentum gives the sum of pixel intensities, which - for binary images - equals the area of the objects. The formulas to calculate the area and center of mass from the zeroth and first orders are as follows:

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

Momenta can be used to determine the orientation of objects too, to achieve that, the axis of least momentum should be identified - which is unambiguous if second-order momenta are known, even if the object is not symmetric. The formula for the orientation is:

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

Of course, there are several other ways to describe the orientation of objects, which generally are easier to compute. If the bounding rectangle of the object is known, then the ratio and size of the sides of it can be used to characterize orientation. The orientation of the maximal in-object distance or that of the maximal distance from the center of mass can be determined in the same way.

7.4.2 Further measures

We can also use descriptors to describe the shape of binary objects, one method of which is the chain code. The chain code assigns a number to each direction between [0-3] or [0-7], based on pixel interconnectivity. After that, from a starting point, each point of the object border is investigated one by one, for each step, its direction is stored. After visiting every pixel and terminating in the starting point, the descriptor of the border of the object is determined.

The chain code can be used to calculate the perimeter of the object, but we should be aware that (especially in the case of 4-connectivity) the perimeter is traversed in a zigzag pattern, which artificially increases its length. If the Euclidean distance is used to calculate the length of the steps, then the result will be more accurate. Previous operations can be used to obtain an image, which

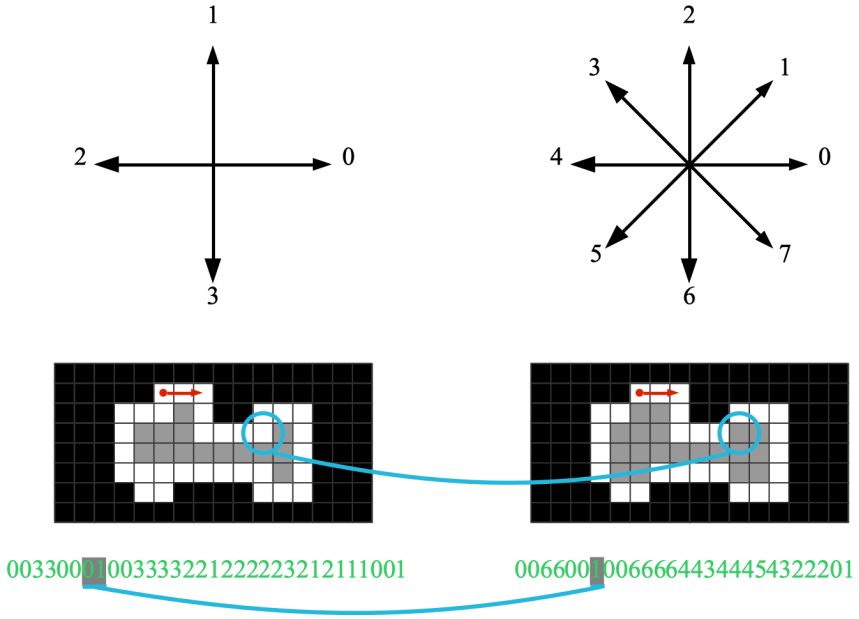

Figure 7.10: *The calculation of the chain code .*

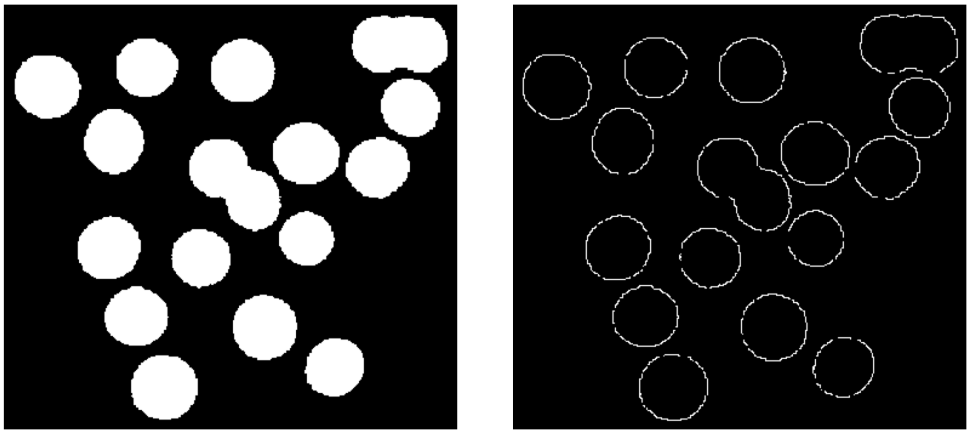

Figure 7.11: *The contours of binary objects.*

only consists of the contour of the objects. For that, erosion should be applied to the original, the result of which should be subtracted from the original image - the result is called the contour image.

If we have the perimeter, area, and center of mass of an object, then other descriptors can be added too, with the help of which they will be identifiable, classifiable. For that, there are several variants, a simple example is the ratio of the perimeter and area.

Widely-used choice is the imprint of the contour, which can be calculated in the following way: from a starting point, with a given direction, the function of the distance between the center of the object and the actual contour point is calculated and used as a descriptor of the shape of the object. The starting point is consequently associated with the largest distance, so the descriptor is made invariant to rotation. If the descriptor is normalized, it will be scale-invariant too.

The Euler-number is also often used, which is the difference of the contiguous regions of the object and the holes inside it. The Euler-number can be used e.g., if the shape of the objects can undergo significant deformation - the most frequently occurring deformation types do not change this property; thus, objects can be distinguished from each other.

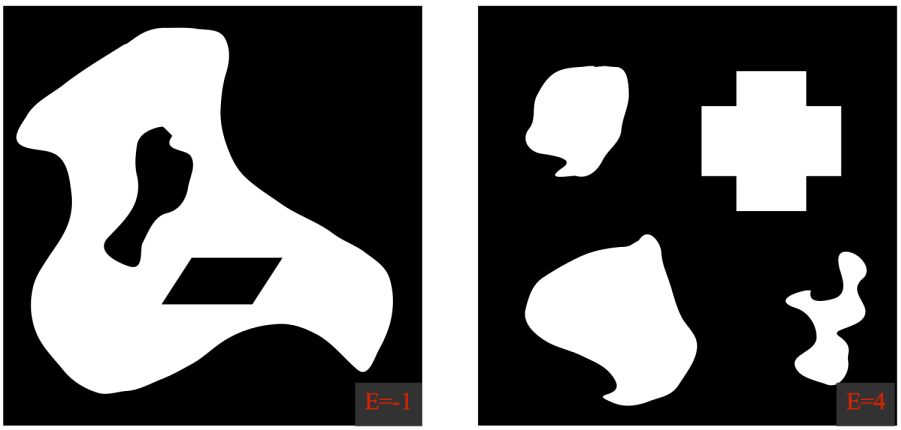

Figure 7.12: *Calculating the Euler-number.*

7.5 Hough-transform

In the case of binary images it is often the task to identify the contour of simple shapes (rectangle, circle), so it would be reasonable to fit a line model to the edge points found - this way the pixels in the line-like composition can be described with a parametric model, which can make object detection simpler. Nevertheless, it should be considered that only one part of the edge points found will fit on a line and they will fit on different lines too, i.e., the task is twofold: the points fitting on a specific line and the parameters of the lines should be determined at once (if the answer for one of the questions would be known then the solution would be trivial).

The Hough-transform is a widely-used algorithm to solve the problem (shape detection can be based on this algorithm). The Hough-transform is a coordinate transform, which transforms the pixels from the common (x, y) coordinate frame to the space determined by the parameters of the lines, denoted by (r, θ). In that space, each point corresponds to a line, one element of which (r) is the length of line section orthogonal to the line itself, starting from the center of the coordinate system. The second parameter is the angle between the line and the x-axis (θ).

It is much more interesting how the image points get mapped to the Hough-space. The mapping of a single point will be a set of all lines, which it is incident to. Although there is an infinite number of lines a pixel fits, it does not fit all lines, so the image of a point in the space defined by (x, y) will be in the Hough-space - spanned by the tuple (r, θ) - as follows:

$$x\ cos\theta + y\ sin\theta = r \tag{7.3}$$

I.e., a pixel is mapped to a sine curve in the Hough-space, if two such curves have an intersection point, it means that they fit the line defined by the coordinates of the intersection point (which defines a line in the Hough-space).

The consequence is that if we have several points which intersect each other, this means that many points are on the same line in the image space. Realizing that, the line-detection algorithm with the help of the Hough-transform can be described as follows: first, all pixels are mapped into the Hough-space, then the lines with the highest number of incident points are determined. After that, those points are removed from the Hough-space, and a search is conducted for the line with the most incident points - this step is repeated until a line can be found which has at least a given number (a threshold should be specified) of incident points.

The Hough-transform can be used for the detection of other simple shapes, which can be described with a parametric model. Using different types of the Hough-transform, several different shapes can be detected, e.g., it is often used for the detection of circles and ellipses. There is also a generalized version of the Hough-transform, which can be used to detect generalized shapes.

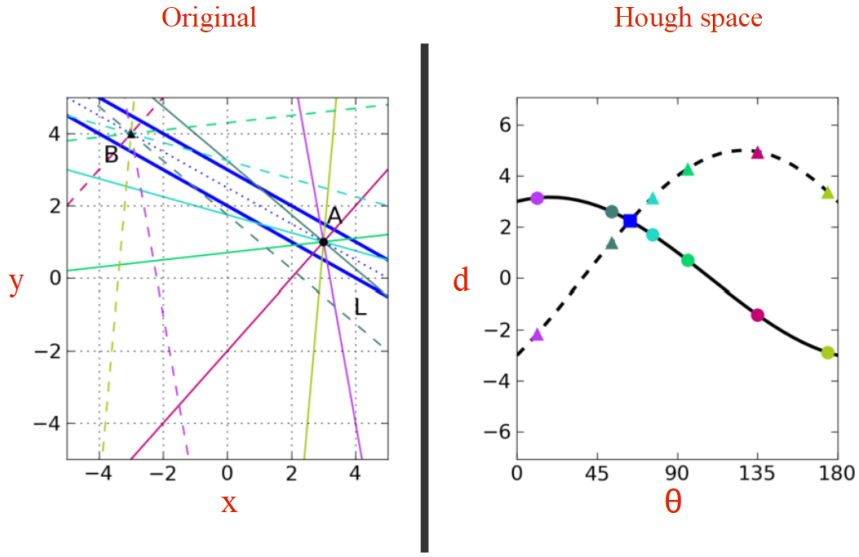

Figure 7.13: *The principle of the Hough-transform .*

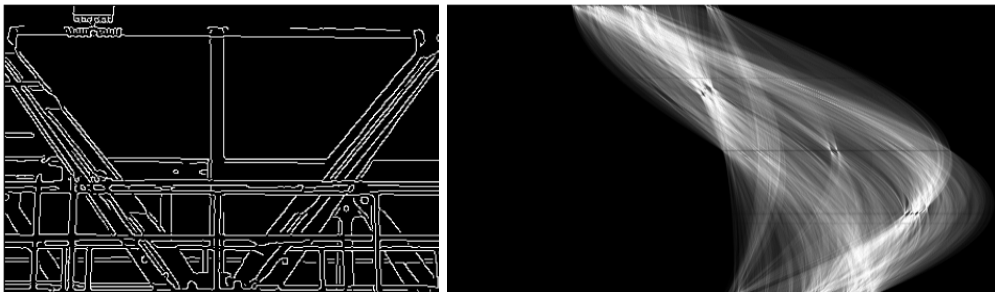

Figure 7.14: *A binary image and its Hough-transform.*

Application: the Hough-transform is often used to detect simple shapes consisting of line segments, e.g., this method can be used to recognize ID cards automatically, which can be considered as a fundamental task of smart public services. The contours of an ID card on an image are generally easily distinguishable, which can be detected without complications using the Canny-algorithm. The Hough-transform follows this step, i.e., edges are converted to lines, and finally, using this result, the rectangle defined by those lines is calculated.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[2] L. Ross, "The image processing handbook, sixth edition, john c. russ. CRC press, boca raton FL, 2011, 972 pages. ISBN 1-4398-4045-0(hardcover)," *Microscopy and Microanalysis*, vol. 17, no. 5, pp. 843–843, Sep. 2011. DOI: 10.1017/s1431927611012050. [Online]. Available: https://doi.org/10.1017%2Fs1431927611012050.

[11] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972. DOI: 10.1145/361237.361242. [Online]. Available: https://doi.org/10.1145%2F361237.361242.

Part II

Learning Vision

8 Neural networks

8.1 Introduction

The main target of computer vision is to extract high-level information from images, but problems discussed in the introductory lecture can make this intention very difficult. At this point, we may wonder whether it would be possible to adapt the abilities of human intelligence into computer vision systems, making problem-solving doable. The domain of artificial intelligence is enormous; several different algorithms are available; the majority of them are exact, i.e., they can be formulated as a sequence of rules and logical conditions - they exploit the intelligence/knowledge of their designer. Due to the lack of understanding of the whole process of human vision, this approach is not applicable to us.

Another type of artificial intelligence consists of learning algorithms, which provide a general, parametrizable model to solve the problem, the parameters of which are tuned during the training process (which is carried out with the help of a training set) - this way the parameters are set to a value that the general model will be able to solve the specific problem at hand. The main advantage, in this case, is that even such problems can be solved, where we do not know the method of the solution - unfortunately, this is only achievable if we can create a proper training dataset.

A disadvantage to consider is the black-box nature of the model we get, i.e., with its help, we do not get closer to the understanding of the problem. This also implies that the reasons behind errors, misclassification, and how to avoid them are also difficult to understand. Because machine learning methods use in general statistics or numeric optimization to obtain the parameter values, no guarantees can be provided generally. Despite the drawbacks mentioned before, they significantly surpass the performance of traditional algorithms regarding computer vision, or, more generally, perception.

8.2 The structure of learning algorithms

In the case of machine learning, the following notation is used consequently: the input is denoted by x, the output by y, and the parameters by ϑ. Using the following notation, the model of a learning algorithm can be given as below:

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

Each learning algorithm has its corresponding cost function (other names are loss, error, or - more generally - objective function), which assigns an error value to the output of the algorithm, which is used to characterize its performance. Furthermore, learning algorithms also consists of one or more optimization method(s), which are used to minimize the loss function by tuning the parameters. Hyperparameters are also part of the algorithm, they are such parameters which influence the quality of the result, but they cannot be determined with the optimization method - typically, such parameters are the properties of the optimization methods or that of the model structure.

8.2.1 Learning types

There are several ways to categorize the methods of machine learning; the first is based on the output type. In the case of regression, the output is a continuous number; otherwise, if it is binary

or an integer from a finite set, we speak about classification. The baseline of classification is binary classification, namely multiclass classification can be built from the composition of binary classifiers - this can be interpreted as if for each class there would be a binary classifier that can distinguish that specific class from the others. The final result is obtained based on the confidence of each classifier. Another possible realization would be that classifiers can decide between two classes, the final result is determined like in sports competitions, i.e., with a ranking method.

We can also classify based on the properties of the training data. The simplest case is called supervised learning, which means that training data is composed of input and correct output pairs, i.e., we know the output w.r.t. each input and expect that the algorithm will be able to determine the correct results for the most input entries or to do it very accurately. If the correct output is only partially known, then it is denoted by the name of semi-supervised learning.

Supervised learning has some significant drawbacks. First, constructing labeled databases is a time- and money-consuming process. Second, the quality of labeling determines the quality of the result. Least, the use of exact algorithms is only possible if the solution of the given task is understood, for supervised learning, the criterion is that we should be able to solve that task. Thus, our cognitive problem-solving ability is the bottleneck in that case.

There is also the so-called unsupervised learning, where only input data is available, the correct output is not known. Constructing such datasets is rather cheap due to the possibility of automating the process. In such situations, we expect that the learning algorithm will be able to find some internal structure in the data, thus it will be able to describe it compactly. The goal of the algorithm is to explain the input with the help of a compact model, i.e., to construct a description of the inner dynamics of it. Such algorithms are clustering methods introduced before, such as k-Means or MoG, which are unsupervised clustering algorithms. The TLS method mentioned shortly can also be interpreted as the unsupervised variant of linear regression.

The third important type is reinforcement learning, which differs regarding two things from the previous two. The first is that for reinforcement learning tasks, the algorithm should be able to make a sequence of decisions, but there is generally no feedback after each decision. On the other hand, the feedback only describes the quality of the steps taken; it does not provide the correct decisions. For reinforcement learning tasks, typical examples are vehicle control tasks or different games (e.g., chess, go, computer games).

8.2.2 Difficulties

At first glance machine learning may seem to be like magic which gives us tools to solve every problem, but practically it also has several limitations and traps which are sometimes difficult to avoid. Important to remember that these algorithms use training data, i.e., they will only be able to describe that piece of the world the data is referring to, which means that training data should cover all possibilities because we do not know how the algorithm would behave for unseen input.

Another point is that e.g., learning vision systems are not able to learn such relationships that need the ability to move or other sensory organs. As an example, consider the following: the definition of a chair could be ”an object we can sit on” - but an algorithm that is unable to move is unable to create the concept of sitting from independent images.

Another pitfall is the complexity of the learning algorithm. Our intuition suggests that if the result is not accurate enough, then making the model more complex will help to improve the performance. Model complexity can be increased in several ways, e.g., we can use an input with more dimensions or simply more parameters. On the other hand, the hyperparameters of the model can also influence the complexity. Building more complex models is not a universal solution strategy: it depends on the tasks, whether it improves or makes the results worse.

It can happen that the algorithm is not complex enough to solve a specific task, in that case, the error on the training set is quite large - if we test it on a validation set (i.e., on data which was not fed into the model during training), the result will be quite the same, this phenomenon is called underfitting. In that case, increasing model complexity will decrease both training and validation

errors. Nevertheless, after a while, the validation error will begin to increase (the training set error will further decrease) - this is called overfitting.

A reason for overfitting is the imperfect nature of the training set. On the one hand, it is only finite, i.e., it is the task of the algorithm to generalize from the data given. On the other hand, both inputs and outputs are noisy, so the training error will not be zero, even in the case of perfect generalization. This means that the algorithm is only able to decrease the training error further if it begins to memorize the input-output pairs; thus, it will be similar to associative memory - implying that for data not represented in the training set, the predictions will be worse. The more complex the algorithm, the easier for it to memorize a large dataset.

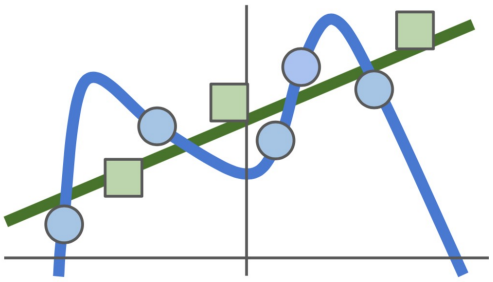

Figure 8.1: *Overfitting in the 1D case. The model (blue) learns the noise in the training set, so its performance on the test data (green) will be rather worse.*

8.3 Image classification

One of the simplest forms of computer vision is the task of classification, i.e., assigning a label to an image that encodes the category of the object depicted in the image. Generally speaking, a computer vision solution for classification consists of carrying out several algorithms sequentially - called the algorithmic pipeline. The first step of which is the capturing and digitization of images, which is followed by preprocessing and enhancing (noise filtering, intensity transformations) blocks. After that, features are extracted to transform the information represented by the pixel intensities to a space spanned by image features of a higher abstraction level. These features are designed to be able to separate task-relevant information easily from disturbances. The last step is decision-making, i.e., assigning a label to the image based on the image features.

8.3.1 Nearest neighbors

We can demonstrate the necessity of using image features with the help of an example. It is also possible to do image classification solely based on image intensity or color, e.g ., with the help of the kNN (k Nearest Neighbors) algorithm, with k as a design parameter. This algorithm searches for the k nearest neighbors of the image in the database (where the labels are known). The neighbors decide (majority vote) the label of the image. The image distance is, in the case of kNN, mostly defined as the absolute or squared difference.

The main problem of the solution is that the difference in intensity or color values does not correspond to the similarity of images, which is especially true for semantic classes. E.g., if the intensity (or the background) changes, the difference of images will be rather different, independently from the class of the object depicted. The problem is much worse if interclass variance results in different colors (animals, vehicles, humans). So color information is only used for classification if the visual properties of the environment can be controlled (e.g., part recognition at an assembly line), including virtual and augmented reality.

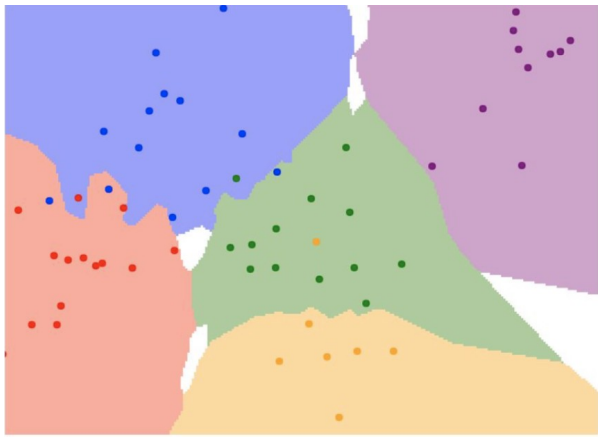

Figure 8.2: *Decision areas of the kNN algorithm for $k = 3$.*

Figure 8.3: *The modified images above have the same squared difference from the original (left above).*

8.3.2 Linear regression

Another essential parameter estimation algorithm is the linear regression or the method of linear Least Squares (LS). The principle of this method is to approximate the relationship between the inputs and outputs of the training dataset with a linear equation. I.e., a line (a hyperplane if multidimensional) is fitted while minimizing the square of the error. The model used is as follows:

$$X\vartheta = Y \tag{8.2}$$

Where each row of the matrix X is a training entry, the ϑ vector consists of the parameters of the hyperplane, and Y is the vector of expected outputs. To obtain the optimal solution, we do not need any optimization algorithm; namely, it can be formulated in a closed-form . Note that in the columns of X, not only different variables but different powers of the same variable can appear. In that case, not a line, but a polynomial is fitted to the dataset, which is called polynomial regression.

$$
\begin{aligned}
||E||^2 = (Y - X\vartheta)^T(Y - X\vartheta) = Y^TY - 2\vartheta^TX^TY + \vartheta^TX^TX\vartheta \\
\frac{\partial||E||^2}{\partial\vartheta} = -2X^TY + 2X^TX\vartheta := 0 \\
\hat{\vartheta}_{LS} = (X^TX)^{-1}X^TY
\end{aligned}
\tag{8.3}
$$

8.3.3 SVM

The linear approximation can be extended to classification too, which can be interpreted that we intend to separate the classes with a line (hyperplane). Generally, an infinite number of such planes exist, so a criterion should be formulated to obtain the best possible result. E.g., we can try to find that specific hyperplane, which separates both classes with the highest certainty, i.e., the nearest point to the hyperplane should be as far as possible. The nearest data entry to the hyperplane is called the support vector, its distance to the hyperplane is the margin.

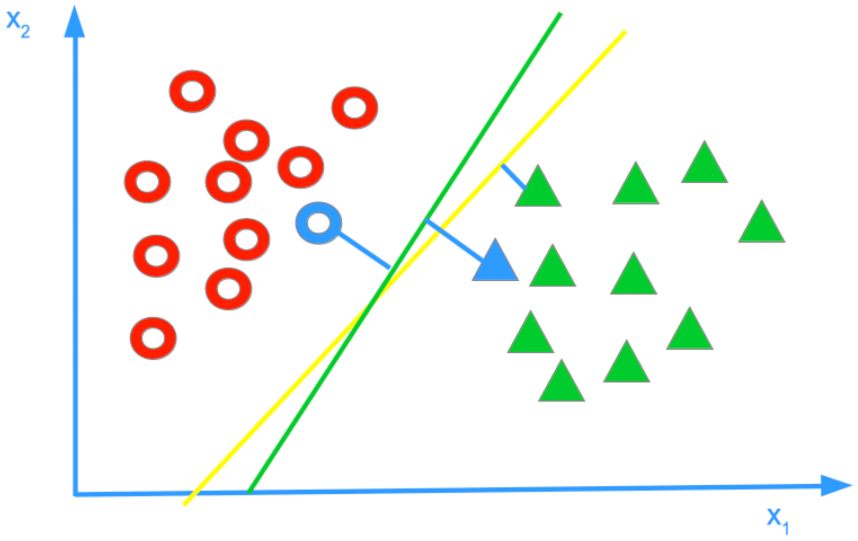

Figure 8.4: *A binary classification task: both classes, the separating hyperplane, the support vectors and the margin .*

The SVM (Support Vector Machine) algorithm is quite popular in computer vision, which determines the hyperplane, which has the highest margin. The decision function of the SVM is stated below:

$$
\hat{y}(x) = \sum_{i}^{N} \alpha_i y_i K(x_i, x) \tag{8.4}
$$

Where N is the number of training samples, y_i and x_i are the in- and output of the ith training sample, α_i is the weight of the ith training entry learned by the SVM , while K is a kernel function. A kernel function is a similarity measure between the input to be classified and the training samples - this means that the output of each entry influences the decision proportional to how similar two data points are. These methods are called kernel methods, which can be treated from a specific aspect as a generalization of the nearest neighbor method.

An important property of the SVM method is that only the coefficients of the support vectors are different from zero, i.e., the above sum should only be calculated for these entries. The choice of the kernel function is also crucial, which significantly influences the result. The default is a linear kernel, which gives the scalar product of two input vectors - in that case, the result will be hyperplane with the highest margin.

For several real problems there is no such linear surface which is able to separate both classes, a good example for that is the XOR-task (of course there are several such examples). A useful property of the SVM method is that if using nonlinear kernel functions, then classes can be separated by nonlinear curves, i.e., the method is easy to extend. The two most-widely used nonlinear kernel functions are the polynomial and the RBF (Radial Basis Function).

$$\begin{aligned} K_{Lin}(x_1, x_2) &= x_1^T x_2 \\ K_{Poly}(x_1, x_2) &= (x_1^T x_2 + c)^k \\ K_{RBF}(x_1, x_2) &= e^{-\gamma ||x_1 - x_2||^2} \end{aligned} \tag{8.5}$$

Where c, k, and are the parameters controlling the complexity of the method. The common property of kernel functions is that they are symmetric and positive semidefinite. Each kernel calculates - as the linear kernel does - the scalar product between two inputs, but before that, a nonlinear transformation is applied (all of this in an implicit manner). I.e., the SVM fits in every case a hyperplane; the only difference is that the parameter space in which fitting occurs is a nonlinearly transformed version of the original one, where this hyperplane is a curve.

8.3.4 The Perceptron model

An essential part of deep learning systems is the linear classification algorithm, which has several names. It is often called perceptron or neuron, despite its classification nature, logistic regression is also in use. The working principle is that the pixels are ordered in a vector, which is multiplied with a weight matrix, the result (output vector) having as many elements as the number of classes. Each element can be interpreted as an indicator of the degree of belonging to that class (the bigger, the better). The formal description of the perceptron is as follows:

$$s = Wx \tag{8.6}$$

Where x is the input, s the degree of belonging to a class, W the matrix of parameters or weights (this notation is used in this chapter consequently). Classifying this way can be interpreted in a way that the ith row of the W matrix selects the direction of the space of pixels along which the score of the ith class increases. Based on that, the decision boundaries between the classes consist of line segments (in the binary case, only one line/hyperplane).

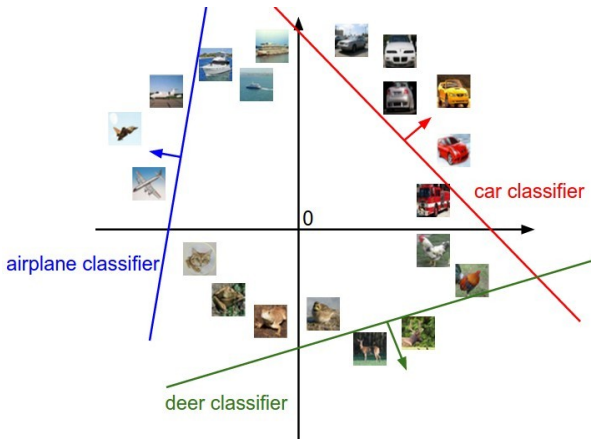

Figure 8.5: *In the case of linear classification the score of a class increases in one direction (all other direction leaves it constant). This direction is given by the row of the weight matrix corresponding to that class .*

8.4 The training process

After defining the algorithmic model of perceptron and analyzing its way of functioning, we should mention how to determine the elements of the W weight matrix. The essential question is how to modify/determine the weights for a better classification result, and which loss function to choose that can measure the performance of the algorithm well.

8.4.1 Cost functions

First, we can characterize the classification quality with the ratio of correctly classified training data, but this approach cannot distinguish between different classifiers with the same accuracy and different variance. Due to this fact, we will define novel loss functions between model output and expected output, which gives the average error for the whole dataset. It should be reasonable to use the squared error between expected and predicted output, which is the most widely-used one for regression problems. The numeric approximation of the output value in the case of classification is not very practical; the squared error can be used, but we can easily construct much better ones.

A frequently used loss function is the so-called Hinge or SVM error function, the main principle of which is to define a measure, which is called the gap, if the score of the correct class is bigger at least with than the other scores, then the error value is 0. Otherwise, the error increases linearly. This loss function can be interpreted as a "safe " separation criteria. The SVM error can be formulated as follows:

$$Li = \sum_{j \neq y_i} max(0, s_j - s_{corr} + \Delta) \tag{8.7}$$

Where s_j is the score of the jth, and s_{corr} is the score of the correct class.

Figure 8.6: *The Hinge loss function in an expressive representation.*

Another practically often used loss function is based on a probabilistic interpretation instead of a geometrical one, it uses the concept of entropy. Entropy formulates that if we have events with different probabilities, then we do not want to encode every one of them with the same amount of bits - it would be reasonable to use a small number of bits for events with high probabilities, and a higher amount for less probable ones. That way, we could minimize the number of bits used for encoding. An event with probability p should be coded with a number of bits inversely proportional to its reciprocal. Thus, entropy gives the expected value of the number of bits used for all events, i.e., the formula for entropy is stated below:

$$H(p) - \sum_i p_i log p_i \tag{8.8}$$

If we do not know the probability distribution of p but an approximation q, then the result will be bigger than the optimal, which is expressed by the measure of cross entropy. The nearer q is to p, the less the cross entropy is. The difference of both entropy values is called KL-divergence, which is a strictly non-negative similarity measure often used for probability distributions.

$$H(p, q) - \sum_i p_i log q_i \tag{8.9}$$

Cross entropy can be used as a classification loss function as follows. First, the output values will be mapped to probabilities with a normalization function called SoftMax, which converts each value to the $[0, 1]$ interval so that the sum will be exactly 1. The formula for SoftMax is as stated below:

$$q_{k,i} = q(y_k|x_i) = \frac{e_{s_k}}{\sum_j e_{s_j}} \tag{8.10}$$

The loss function is defined as the cross entropy between the expected distribution of the labels and the predicted q distribution. Cross entropy is minimal if both distributions are the same, thus minimizing this function will result in distributions closer to the prescribed ones. The prescribed distribution can be constructed by setting the probability of the correct class to 1, the others to 0, thus cross entropy is simplified to:

$$\begin{aligned} L_i = H(p_i, q_i) &= -\sum_k p_{k,i} log q_{k,i} \\ L_i &= -log q_{true,i} \end{aligned} \tag{8.11}$$

Where $p_{k,i}$ and $q_{k,i}$ are the prescribed and expected probabilities of the ith training sample belonging to the kth class, $q_{corr,i}$ is the predicted probability of the correct class. An advantage of using cross entropy as a loss function is that probabilities are used instead of scores which are hard to interpret. A drawback compared to the SVM loss is that cross entropy will never be 0; thus, the SVM loss "saves more": it is satisfied with safe separation and lets the model classify other points correctly. Nevertheless, the difference in both loss functions is hard to show in practice.

8.4.2 Regularization

Both loss functions have an essential problem: if the weight matrices are multiplied with a constant then the value of them will decrease; thus the limit of the value of the weights will tend towards infinity, causing numeric problems and resulting in a confident model - while having the same classification accuracy.

To avoid that, these loss functions are used with a penalty (regularization) term, which constrains the norm of the weight matrix. Often the L1- or L2-norms are used - or even the weighted average of them (elastic regularization). The final loss function can be described as follows:

$$\begin{aligned} L &= \sum_i^N L_i + \lambda R(W) \\ R_{L1}(W) &= \sum_k \sum_l |W_{k,l}| \\ R_{L2}(W) &= \sum_k \sum_l W_{k,l}^2 \\ R_{EL}(W) &= \sum_k \sum_l \left(\beta W_{k,l}^2 + |W_{k,l}|\right) \end{aligned} \tag{8.12}$$

Where L_i is the error for the ith training sample, N the number of training samples, R the regularization term, the λ hyperparameter is the relative weight of the regularization term. In the case of multilayer networks (discussed in the next subchapter), an increasing matrix norm can cause overfitting; thus, regularization can be used to avoid it.

8.5 Optimization

In the previous lecture we did not mention how to minimize the loss function of the perceptron model, although it is a key point. Due to the fact that a closed-form solution does not exist for the perceptron, we will use iterative methods .

8.5.1 Gradient-based optimization

Since both the model equations and the loss function are differentiable, gradient-based methods are applicable. The gradient of the loss function w.r.t. the weights gives us the measure of modification needed in the weights to increase the loss function as much as possible - we need to minimize the loss function thus, we choose the opposing direction (think about it as an "inverse mountaineer"), which will result in a local minimum.

Our goal is to minimize the loss of the whole database; thus, it should be evaluated after each step on the whole dataset. Considering the quite slow (linear) convergence properties of the gradient-based approach. We split the training set to equal subsets (chosen randomly, they are called minibatches, and have a size according to the powers of 2, due to implementation reasons only), and the steps are repeated after each minibatch - this algorithm is called SGD (Stochastic Gradient Descent). The gradient calculated for a minibatch does not equal to that of the whole dataset; nevertheless, the approximation is close enough to result in a satisfying direction and much less computational costs. The update of the weights is calculated as follows:

$$W_{k+1} = W_k - \alpha \frac{\partial ||E||^2}{\partial W} \tag{8.13}$$

Where is the learning rate, a hyperparameter influencing the speed and quality of the training process in a significant way. The proper choice for the learning rate will be elaborated in a later chapter.

A disadvantage of the gradient method is that it can get stuck in local minima; thus, we switch from our "inverse mountaineer" to a downhill rolling stone (yeah, just one) - meaning that we add inertia to the gradient, called momentum. From an implementation point of view, the update step is constructed as a linear combination of the previous update (its weight is in [0.1-0.9]) and the (current) negative gradient.

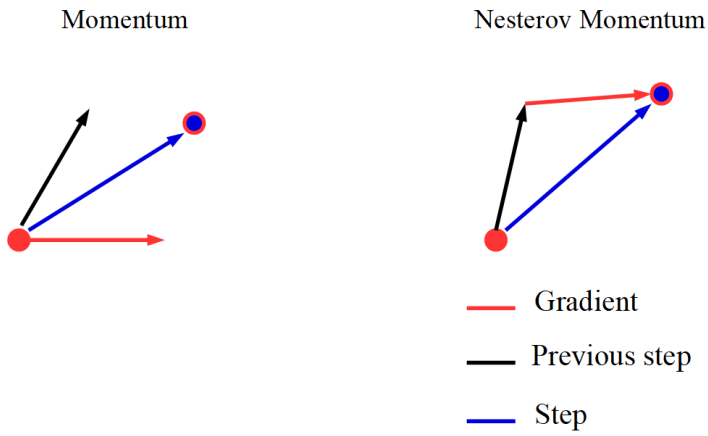

Figure 8.7: *Momentum method (left) and the Nesterov-momentum (right). The latter first performs a step to the momentum direction and the gradient will be calculated at the new position, resulting in a somewhat more stable operation.*

The main drawback of SGD is that it considers each direction of the multidimensional parameter space as equal, although the loss function may be rather steep in a specific direction; thus, small

steps should be chosen for bigger ones may result in "jumping over" the minimum. While in another direction we can have a flat hypersurface and a far minimum, thus a large step size is needed, in contrast to the former case which requires a small one - resulting in controversial requirements (thus we need a compromise to proceed).

Fortunately, the solution is delivered by algorithms, e.g., AdaGrad or RMSProp. The principle of both is to scale the step size in each direction with the inverse of the gradient into that direction, thus providing different step sizes. One of the most widely-used optimization algorithms is called Adam, which is a combination of momentum and gradient-scaling.

8.5.2 Higher-order derivatives

Gradient-based methods have two quite significant drawbacks: the fix step size causes oscillations near the minimum (i.e., the step size specifies the resolution of the approximation). On the other hand, the method is quite slow, especially when starting far from the minimum.

We may conclude that approximating the objective function with a quadratic one, the minimum can be calculated analytically; thus, one step will suffice to reach it. The Newton-method works based on that principle, which uses a second-order Taylor-series approximation for the objective function in x_N as stated below:

$$f(W_k + \Delta W) \approx f(W_k) + f'(W_k)\Delta W + \frac{1}{2} f''(W_k)\Delta W^2 \tag{8.14}$$

Which is derived w.r.t. ΔW, the minimum position can be calculated by setting this expression equal to zero:

$$\begin{aligned} 0 &= f'(W_k) + f''(W_k)\Delta W \\ \Delta W &= -\frac{f'(W_k)}{f''(W_k)} \\ W_{k+1} &= W_k - \frac{f'(W_k)}{f''(W_k)} \end{aligned} \tag{8.15}$$

Since the objective function is not quadratic, we use the above rule within an iteration. In a multivariate case the gradient and the Hesse-matrix (matrix consisting of second-order partial derivatives) is used instead of the first and second derivatives, respectively:

$$W_{k+1} = W_k - H_W^{-1} \ \nabla_W f(W_k) \tag{8.16}$$

Nonetheless, this approach also has its drawbacks: first, calculating the Hesse-matrix is often complicated, and its size increases quadratically w.r.t. the number of parameters, resulting in enormous structures for machine learning algorithms with millions of parameters. Second, the Hesse-matrix may not be invertible (i.e., at least one eigenvalue is near zero) - a consequence of a function which is plain in a direction or, more possibly, we are in a saddle point.

To mitigate that, we use the Levenberg-Marquardt algorithm, which uses a slightly modified update rule:

$$W_{k+1} = W_k - [H_W + \lambda I]^{-1} \ \nabla_W f(W_k) \tag{8.17}$$

I.e., we add a weighted identity matrix to the Hesse-matrix, thus ensuring positive definiteness (with a proper scaling factor). This modification can be interpreted as follows: if the Hesse-matrix is "small", the identity matrix will dominate (resulting in the gradient method), while if it is "big" , we get the Newton-method. As a consequence, the algorithm can cross problematic spots - even if slowly - of the objective function.

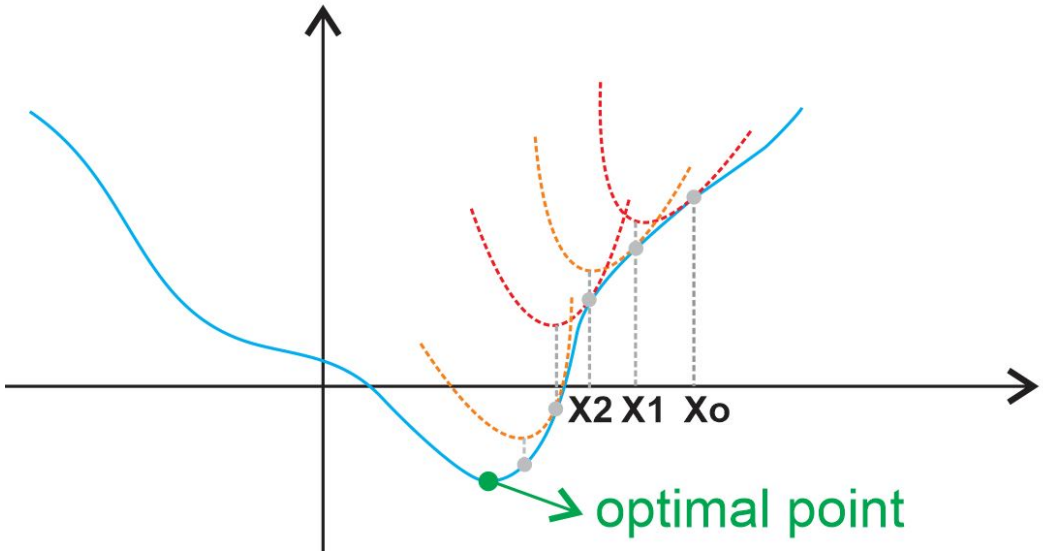

Figure 8.8: *The principle of the Newton-method.*

8.5.3 Backpropagation

As mentioned before, the capabilities of linear models are limited; thus, they are seldom used for images. Nevertheless, they can be used to construct nonlinear learning algorithms. If the neurons introduced in the previous subchapter are connected, we get a multilayer, feedforward neural network. Each layer has its corresponding (unknown) weight matrix, which can be determined with the help of loss functions and optimization methods introduced beforehand.

The only question is how to calculate the derivative of the loss function w.r.t. the weights. Since a neural network can be interpreted as a computational graph, where the nodes implement functions, which can be analytically derived, we can calculate the output of all nodes given the inputs and the node functions - this is called the forward pass. If we know the node functions and the derivative of the loss function w.r.t. the output of the nodes, we can calculate the derivatives w.r.t. the node input and weights by using the chain rule.

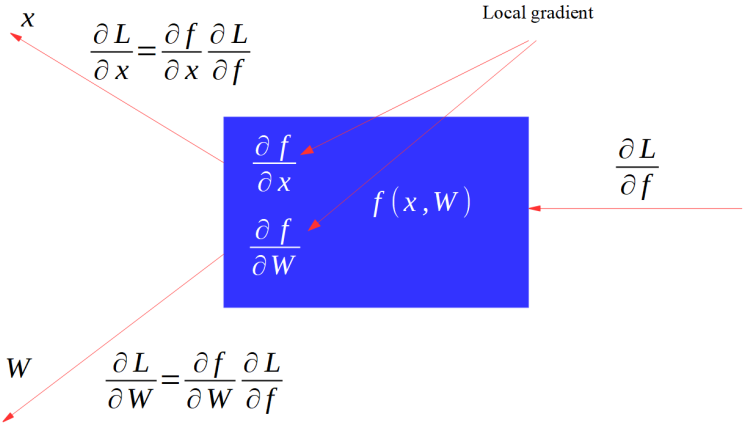

Figure 8.9: *Illustrating the chain rule.*

Where L is the loss function, f and x are the in- and output of a specific layer, and W is the weights of that layer. Using this approach from the end to the front of the network, we can calculate the gradients of all inputs and weights - this method is called backpropagation. The only question is how to get the derivative of the loss function w.r.t. the output of the last node of the computational graph. Note that the last operation of the forward pass is calculating the loss function; thus, the

output of the last node is the cost function itself, the derivative of which w.r.t. itself is 1, so we have everything we need to calculate the gradients.

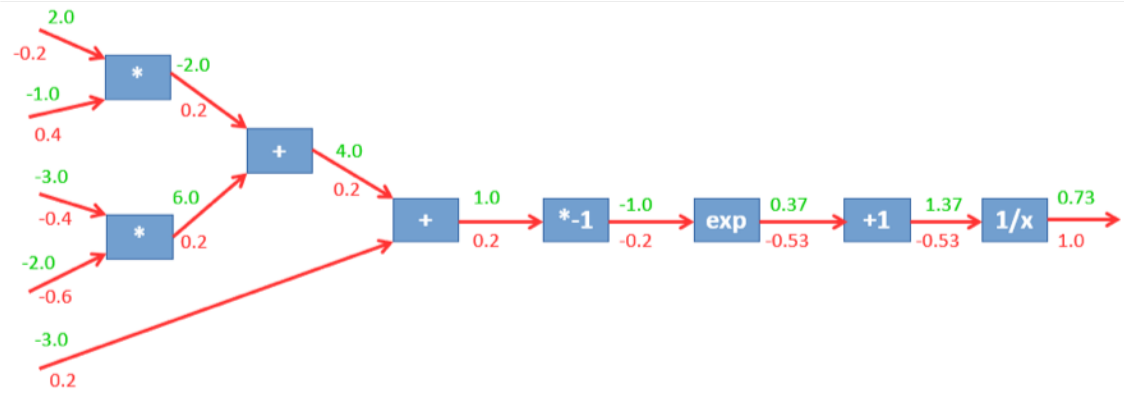

Figure 8.10: *Doing backpropagation on a computational graph. The activations are green, the derivatives are red .*

Of course, the neurons are not constrained to the perceptron model, several different layer types exist, all of which implement functions which can be derived, thus if the forward and backward passes are implemented, our hands are no further bound, we can create new layer types.

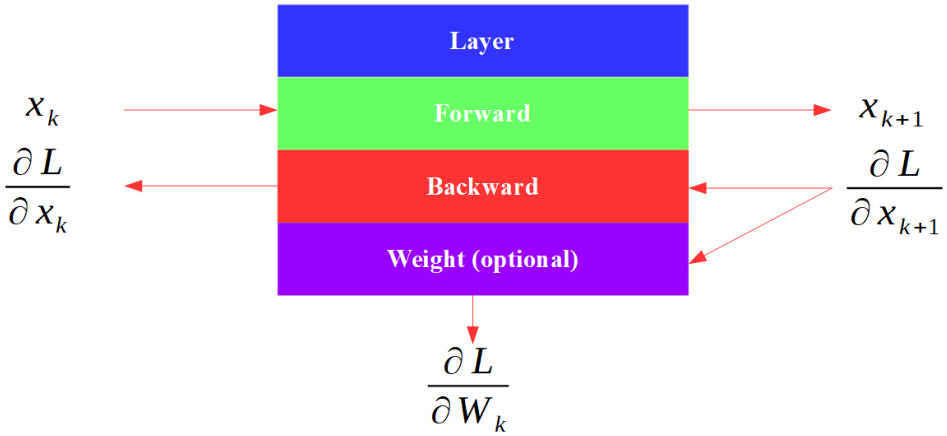

Figure 8.11: *The uniform interface of a layer, which consists everything needed for training and inference .*

Further Reading

[18] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007%2Fs10710-017-9314-z.

[19] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. Springer New York, 2013. DOI: 10.1007/978-1-4614-7138-7. [Online]. Available: https://doi.org/10.1007%2F978-1-4614-7138-7.

[20] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, Jun. 1963. DOI: 10.1137/0111030. [Online]. Available: https://doi.org/10.1137%2F0111030.

9 Convolutional Neural Networks

9.1 Introduction

In the last lecture, we discussed the essentials of machine learning and got to know the method of linear classification. Nevertheless, those algorithms turned out not to be sophisticated enough for accurate image classification - but they can be used as building blocks to build bigger models. Today, we will discuss deep neural networks built from linear blocks.

9.2 Convolutional Neural Networks

Although linear neurons can be found in computer vision applications, they do not represent the typical case - the reason of which is that linear (also called fully connected) layers establish a connection between every in- and every output, thus they have many parameters. As a consequence, storage may be problematic, and overfitting will occur more often. Furthermore, fully connected layers do not exploit the spatiality of images.

9.2.1 Convolutional layer

The convolutional layer can solve the above problems (its name originates from its similarity to the convolutional filters discussed in previous lectures). Such a layer executes N convolutional filters on the input image (which generally has 1-3 channels), the result of which is a filtered image with N channels. Thus, following convolutional layers work with more (N) channels. The size of the kernels and the number of channels (i.e., the depth of the layer) are typical hyperparameters. To retain the original image size after convolution, generally, images are zero-padded (a border from zeros is constructed around the image).

Two additional parameters should be mentioned, which are called stride and dilation. If stride equals 1, then the filter is executed at every position, if it is 2, then only at every second. Dilation (Fig. 18) specifies with which image pixel (how much is the offset) is the next (i.e., shifted with 1 pixel) weight of the kernel multiplied. If it is set to 1, it gives the traditional function; higher dilation results in a ”drawn-apart” filter scheme. This parameter is used to increase the receptive field (the area a layer can perceive) of the layers.

9.2.2 Pooling

Note that if several convolutional layers are stacked after each other (with increasing depth), the size of the activation array at the output of the layers will be huge. To avoid this phenomenon, after a few layers, size reduction is incorporated. Besides using a stride bigger than 1, we also can use the operation of pooling, which is also a sliding-window operation. It substitutes the whole window with one number - most often, the maximum or the average is used.

9.2.3 Activations

The last essential building block of neural networks is the activation layer/function, which follows - in general - each linear and convolutional layer. For both layer types carry out linear operations,

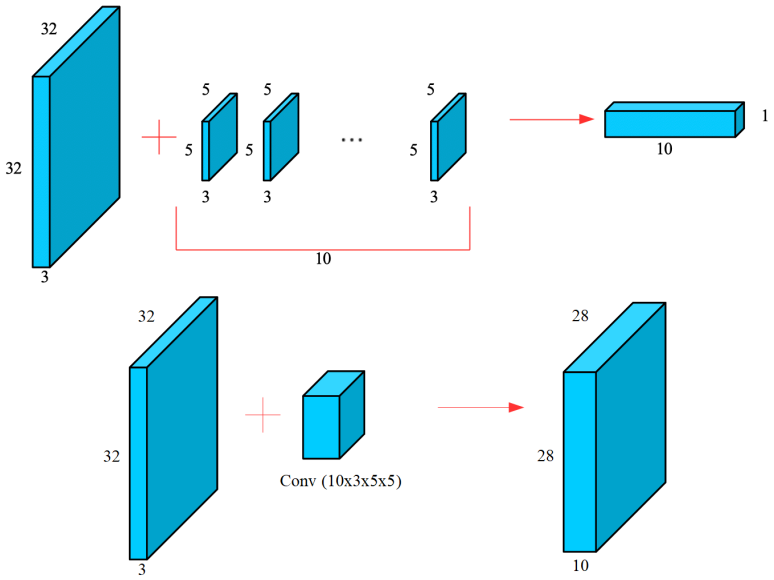

Figure 9.1: *Executing convolution on an image array (above) and the equivalent convolutional layer (below).*

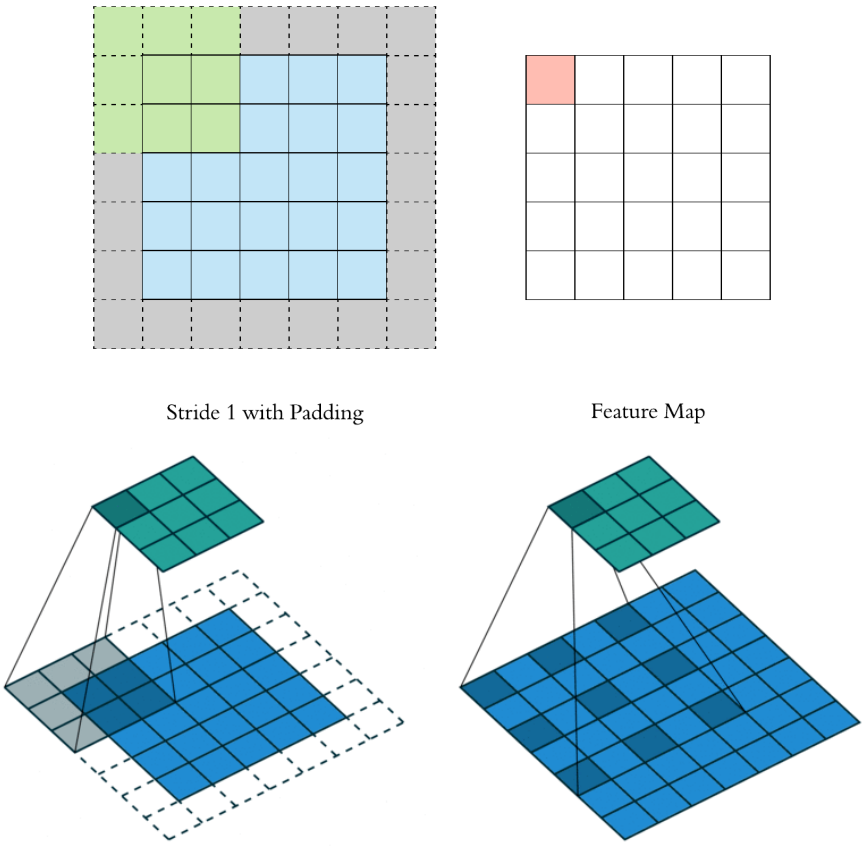

Figure 9.2: *The effect of padding (above), stride (left below) and dilation (right below) on convolution.*

the composition of them will also remain linear. Thus, we insert nonlinear functions between them, which are executed in an elementwise manner. In the case of traditional neural networks, the common choice included the sigmoid or the hyperbolic tangent (abbreviated as tanh), but they are only in the fraction of their definition interval steep; thus, their derivative is almost everywhere

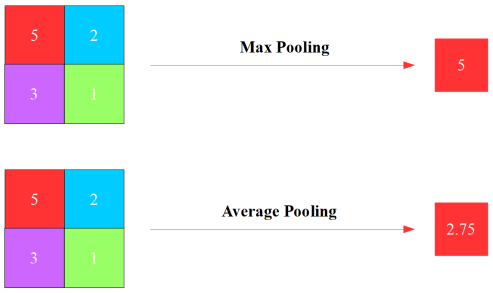

Figure 9.3: *Max (above) and average (below) pooling.*

0. For several layers - the effect of the chain rule - the majority of the derivatives will equal 0 (referred to as the problem of vanishing gradients); thus, the weights will not be modified, and the training can be unsuccessful.

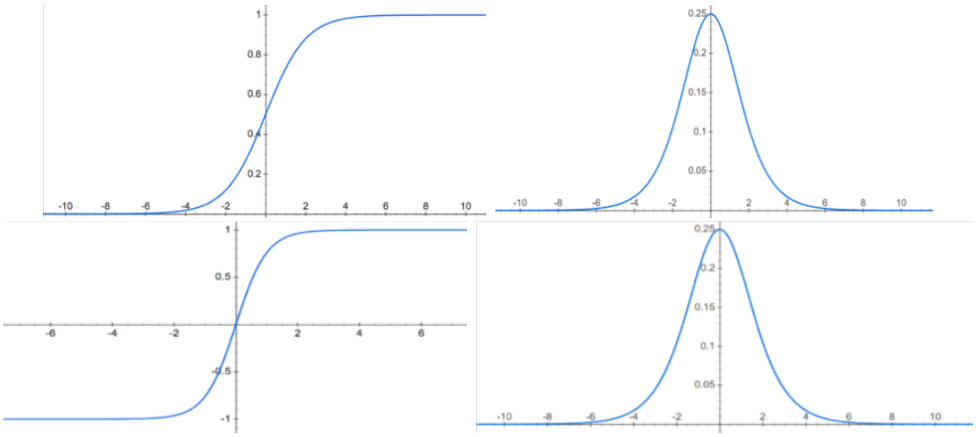

Figure 9.4: *The sigmoid and its derivative (above) and the same comparison for tanh (below).*

For recent architectures, the most popular choice is the ReLU (Rectified Linear Unit), which is a piecewise linear function, it sets negative inputs to 0, but positive ones are retained. The derivative of ReLU is 0 in the negative and 1 in the positive domain. Thus its disturbing effect on derivatives is significantly less. Nonetheless, in the case of using ReLU, we can also experience that weights got stuck - in those cases, the PReLU (Parametric ReLU) or the Leaky ReLU can be used. The latter two variants differ from the vanilla ReLU that they do not set negative activations to 0, but multiply them with a small constant (a hyperparameter for Leaky ReLU, a tunable one for PReLU, i.e ., it is learned with SGD). Another variant is the so-called ELU (Elastic Linear Unit), which is completely smooth, i.e., everywhere differentiable - in contrast to the original version.

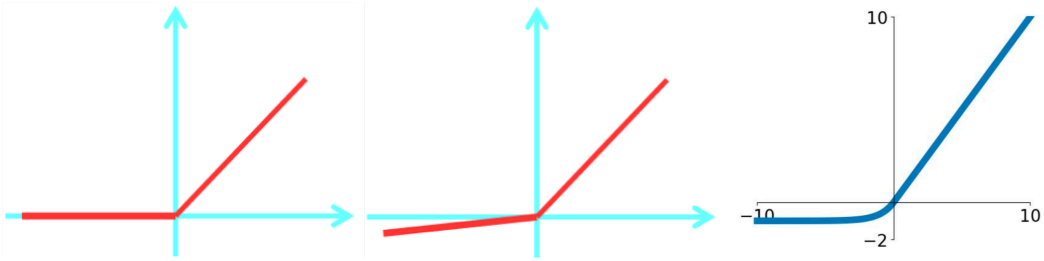

Figure 9.5: *ReLU (left), Leaky ReLU (middle) and ELU (right) activations.*

9.3 Architectures

Neural networks discussed in this chapter are called convolutional neural networks, which are used first of all for computer vision. An important advantage of convolutional layers compared to linear ones is the significantly fewer parameters they possess. A convolutional architecture consists of a sequence of convolutional layers (with activation for each of them) and downscaling operation after a few convolutional layers (stride greater than 1 or pooling). In the end, linear layers (one or a few) calculate the final output.

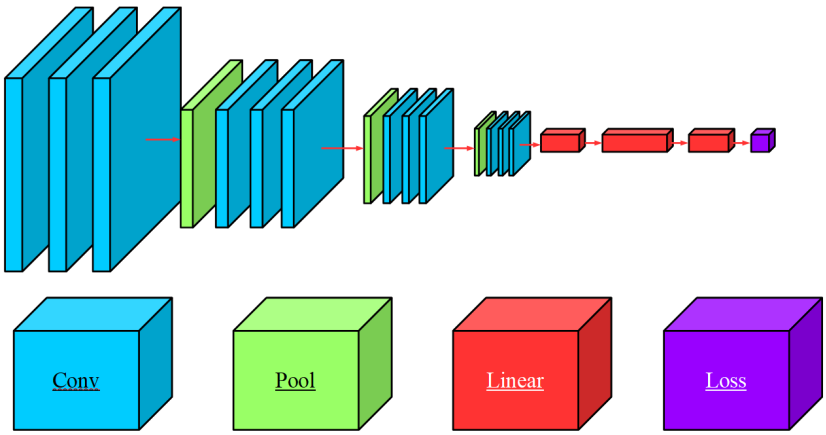

Figure 9.6: *A typical convolutional net.*

It is worth considering what exactly happens as a result of images flowing through a sequence of convolutional layers. Convolution can be interpreted as a simple feature detector, which will be active for some input combinations, and inactive for others. I.e., the output of the first convolutional layer gives us the pixels that activated the kernels. The next layer gets this activation map as an input, i.e., the kernels are not activated anymore for a combination of pixels but that of low-level features. Thus a multilayer convolutional architecture will be able to detect high-level features in layers further away from the input - this approach is quite similar to that of the composition character of human vision.

First successful convolutional networks implemented a similar architecture, but several, more advanced ones exist nowadays - the motivation is mainly twofold: first, the problematic topic of convergence in deep networks should be addressed. Second, our goal is to design architectures that are able to capture complex relationships with fewer parameters, thus decreasing the probability of overfitting. In the following, the motivation behind recent architectures is discussed.

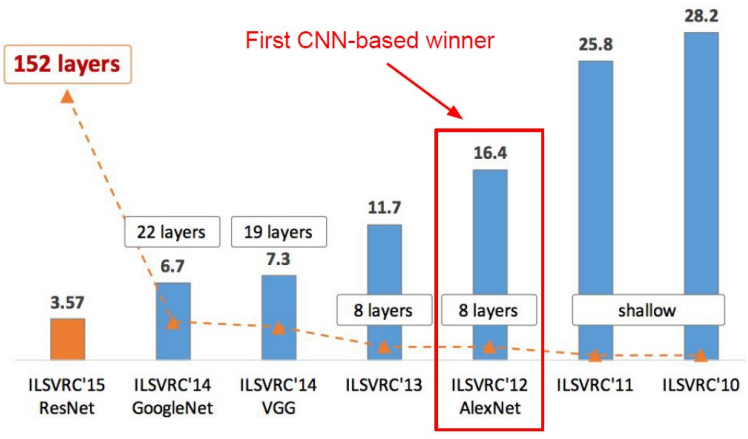

Figure 9.7: *The winner networks of the ImageNet classification competition and their corresponding layer numbers.*

9.3.1 AlexNet

One of the first successful convolutional architecture was the AlexNet, which was the first to use a layer number around 10. The reason behind its success was the application of ReLU activations and normalization layers.

9.3.2 VGG

VGG is one of the most popular traditional neural network model, which has variants with layer numbers between 16 and 19. Compared to AlexNet, it has a more regular architecture; namely, it consists only of convolutional layers with 3x3 kernels. The motivation for which is that three consecutive 3x3 convolutional layers have the same receptive field as one with a 7x7 kernel, but in the former case, we have 2 additional nonlinearities and about half the parameters. I.e., more complex models can be learned while reducing the probability of overfitting (due to fewer parameters). It is worth noting, however, that VGG is not a particulary good architecture to avoid overfitting, as the enormous parameter count of the final linear layer offsets any gains made here.

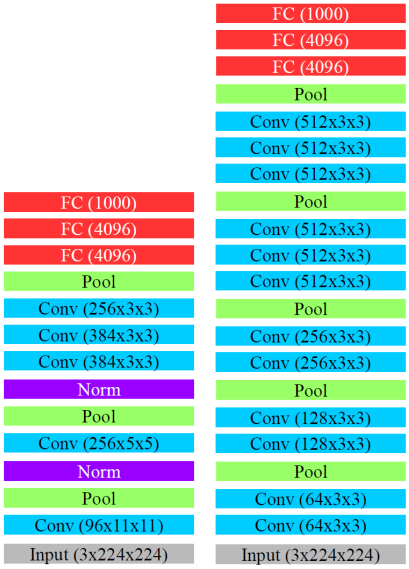

Figure 9.8: *AlexNet (left) and VGG (right) .*

The figure below shows the layers of VGG and their corresponding memory requirements and parameter numbers. It is clear to see that the first part is more memory-intensive, while the near-output part is more parameter-intensive. VGG is considered as wasteful regarding both memory and parameters compared to modern architectures.

9.3.3 Inception

The Inception layer is a good example of how to reduce the number of parameters. It uses the fact that convolutional layers can exist not only sequentially, but also parallel to each other. The parallel layers carry out generally convolutions of different sizes, thus resulting in a structure which is more capable of handling variations of the object scales.

The neural network called GoogLeNet consists of several Inception blocks, besides it has several helper classifiers coming from lower levels, which intend to help to achieve convergence by channeling the gradient to the lower levels.

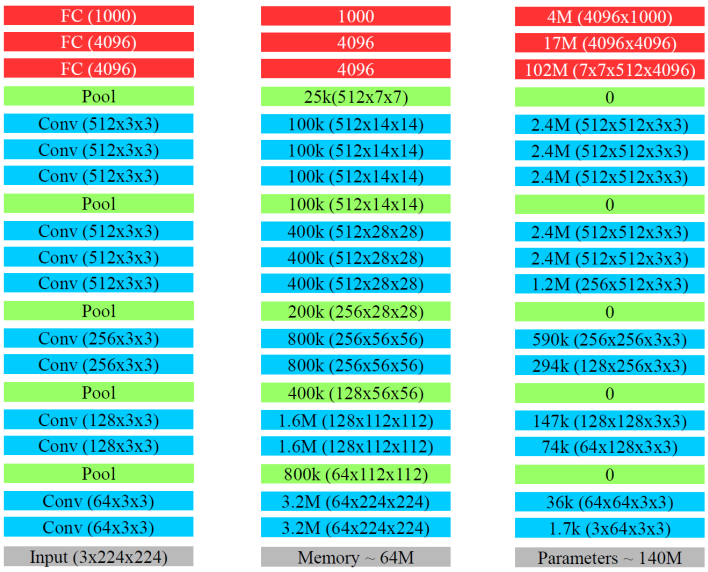

Figure 9.9: *The number of parameters and memory footprint for the VGG architecture.*

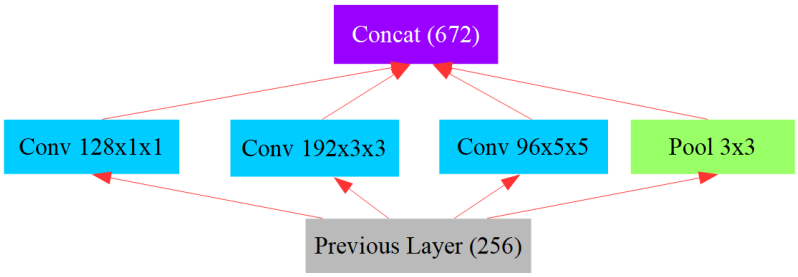

Figure 9.10: *A naive Inception layer.*

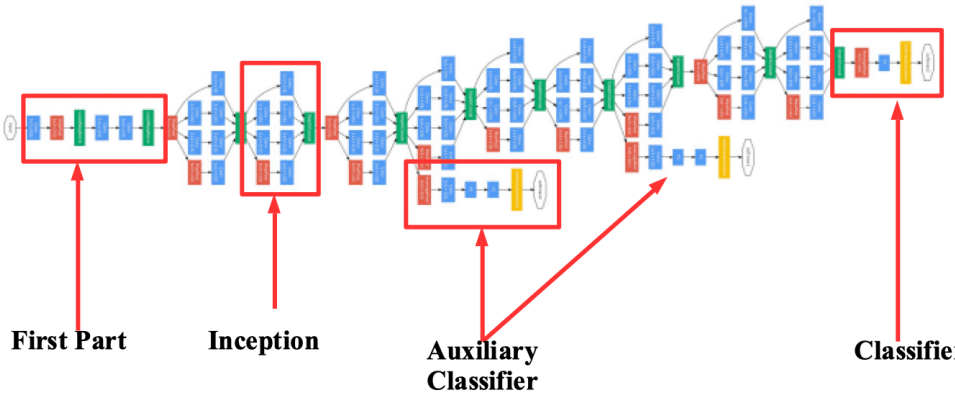

Figure 9.11: *The GoogLeNet model.*

9.3.4 ResNet

The residual block is a good example of how to help convergence too. It achieves that by adding the input to the output of the convolution, thus the layer is supposed to construct the difference between the expected in- and output. The advantage of this design trick is that through the residual connections, the derivative of the loss can be easily backpropagated - note that the derivative of the additional nodes will be 1; thus, numeric problems will be avoided with a higher probability. Residual blocks made it possible to construct networks with depths of 100-200 layers - instead of

the former maximum of about 30.

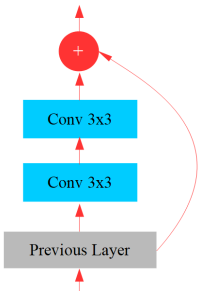

Figure 9.12: *The residual block.*

There is another layer type worth mentioning: bottleneck layers are used to compress the dataflow - both Inception and residual blocks use them. The motivation for them is that 2D-convolutions are rather computationally expensive if the number of the channels of activation maps is big. Thus, a 1x1 convolution is executed before each 2D-convolution, which reduces the number of channels of the activation maps to its fraction. A 2D-convolution (3x3 or 5x5) follows, but in the end, the original channel number is reconstructed with another 1x1 convolution, resulting in the reduction of both parameter numbers and execution time.

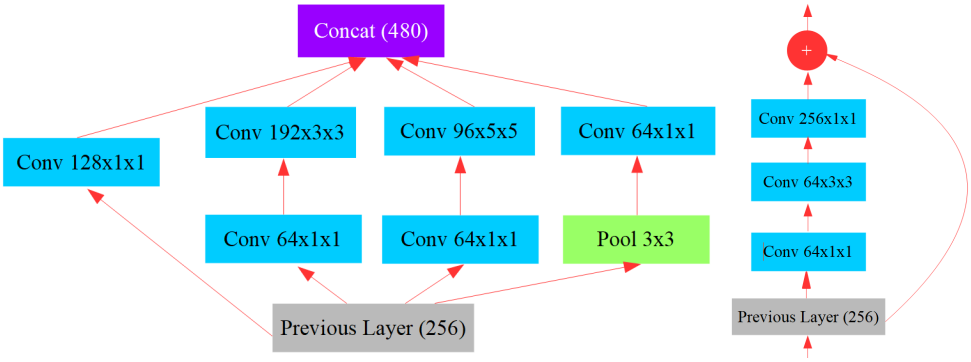

Figure 9.13: *Inception (left) and residual (right) blocks using a bottleneck.*

9.3.5 DenseNet

DenseNet is based on the ResNet architecture, which uses connections within a dense block not only between consecutive layers but also between all layer pairs within the block (thus the name dense block). This modification made it possible to halve the number of parameters and operations compared to ResNet while approximately maintaining its accuracy.

9.4 Visualization

Previous lectures discussed the structure, training, validation, and installation of deep neural networks in detail. Nonetheless, the topic of debugging was not mentioned yet, which is - indisputably - a crucial question, for the trial and error approach costs much time (even up to weeks). The problem we are facing is the fact that neural networks are black-box models; thus, a complete understanding is almost impossible, thus questions like "What?" and "Why?" remain generally unanswered. If we are able to visualize the inner dynamics of the network, we may get a useful glimpse of what is going on (not so) well.

Note that backpropagation can be used to calculate gradients between two arbitrary points within the graph - this can be used in different ways. First, we can determine the pixels that are responsible

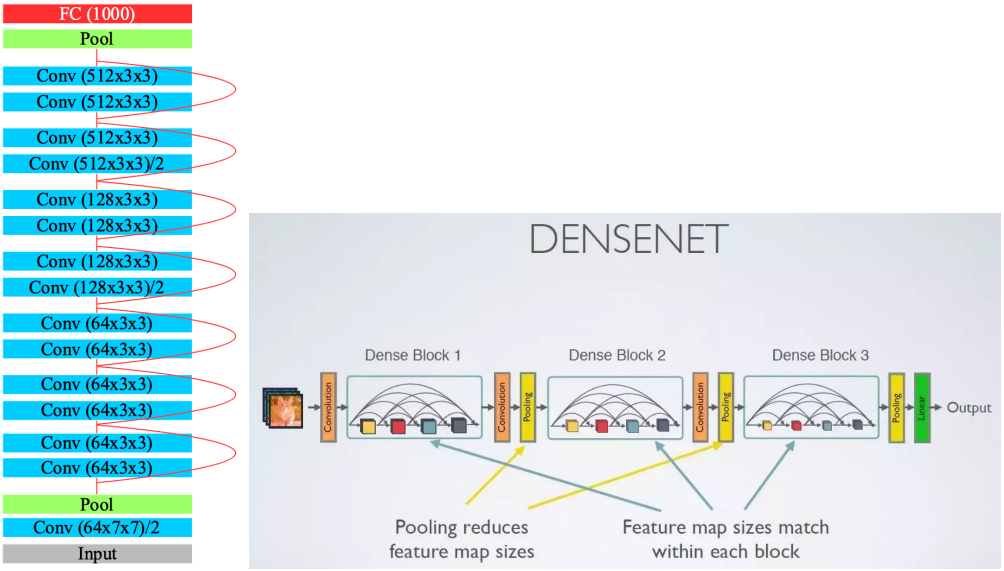

Figure 9.14: *ResNet (left) and DenseNet (right) models.*

for the scores during classification - a method applicable for object segmentation, tracking or for merely helping the engineer identify what image part is responsible for the erroneous classification result.

For the debugging usage, we should only set all output gradients but that of the selected class to 0 (the latter is set to 1), after that, backpropagation is carried out, but in this case, we determine the gradient w.r.t. the input image (not that of the weights). Selecting the maximum along the channel dimension after calculating the absolute value we get a grayscale image, where intensities are proportional to the effect on the scores (this image is called saliency).

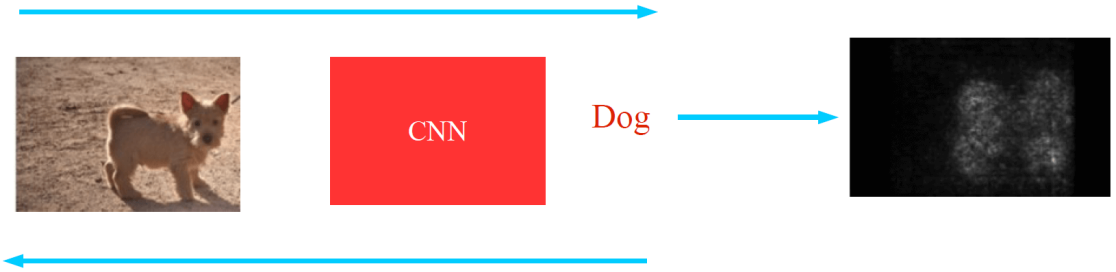

Figure 9.15: *Calculating the saliency.*

9.4.1 Guided backpropagation

If we would like to use this method for segmentation or visualization, then we will face the following problem: the saliency will be significant at locations that influence the output negatively (e.g., "where there is no dog"). To avoid that, we should suppress the effect of such features during backpropagation, which have a counter-effect regarding the class investigated. This problem can be solved with ease; namely, if the corresponding derivative is negative, the feature decreases the effect of the investigated effect. I.e., such gradients should be set to 0 in each layer, the most straightforward way to implement it is to apply ReLU on the gradients during the backward pass of the ReLU activation - the resulting method is called guided backpropagation.

Guided backpropagation can be used - besides getting good-quality saliency - to visualize arbitrary neurons, which is used to determine the image representing the maximal activation for that neuron. For that, the input image is initialized to zeros; then, it is used for the forward pass up to the

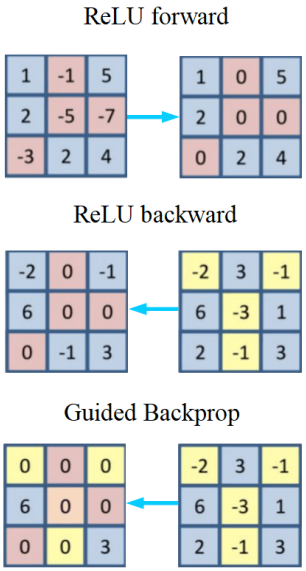

Figure 9.16: *Applying guided backpropagation to the ReLU nonlinearity.*

layer investigated. In the next step, the output gradients of the layer are prescribed, as mentioned above. Then, backpropagation is carried out up to the image. The gradients will be added to the image, then the last two steps are repeated until the change is not negligible.

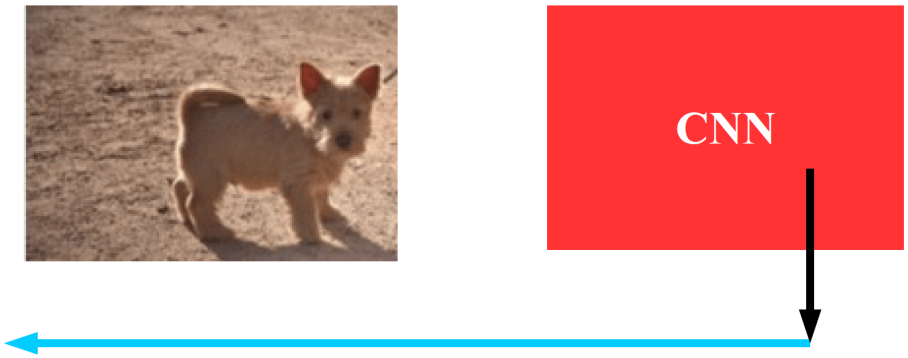

Figure 9.17: *The principle of guided backpropagation.*

Without enhancement, the resulting image cannot be interpreted, and pixel values will increase, and the algorithm will not terminate. Thus, it is necessary to apply any (generally L2) regularization and smoothing filters (for filtering out noise). Small pixel and gradient values during backpropagation are worth setting to 0 manually.

9.4.2 Adversarial examples

An essential discovery regarding the applications of backpropagation was that it is possible to modify small (for humans unnoticeable) details of correctly classified images, which result in an incorrect classification with neural networks used in computer vision. These images are called adversarial examples, and - according to the recent status quo of science - it is rather hard to create an effective defense strategy against them. The best we can do to feed such images also to the network while training. Of course, humans also have such mistakes, the adversarial examples of human vision are called illusions - but those for neural networks seem to be significantly different.

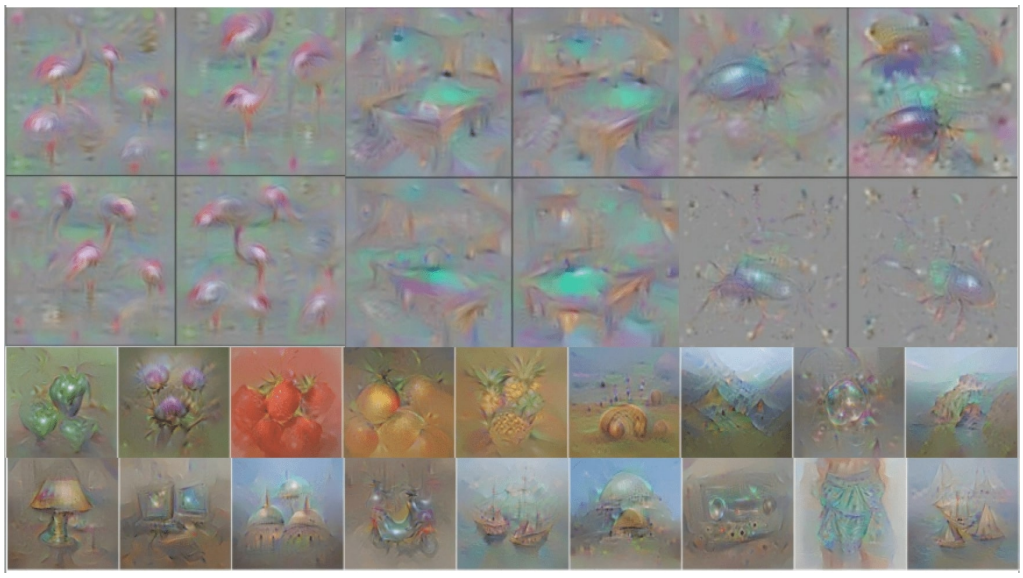

Figure 9.18: *The result of guided backpropagation.*

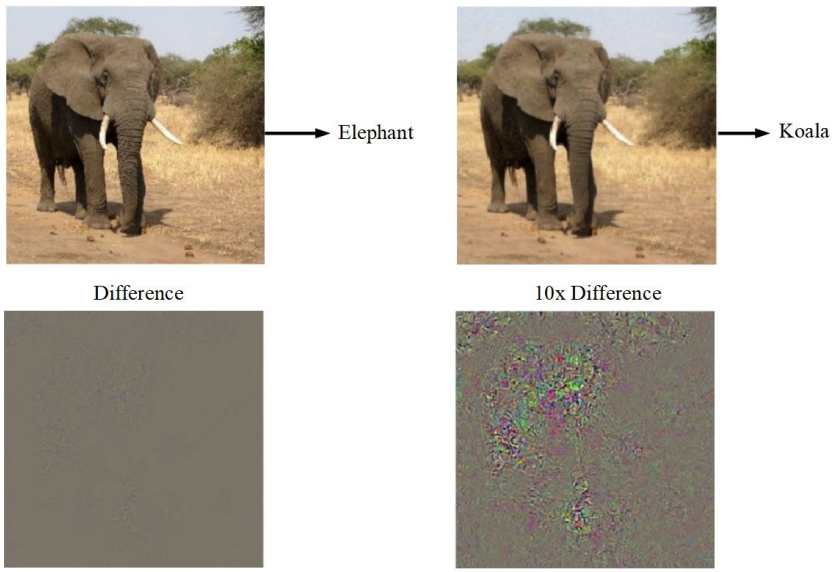

Figure 9.19: *Adversarial examples.*

9.4.3 DeepDream

Guided backpropagation has another, rather interesting use. This application differs from visualization slightly, because here the input image is not empty but an arbitrary real image. Nevertheless, we also use it for the forward pass, but the output gradient is prescribed differently: it will be set equal to the layer activation. This will result in amplifying the details that were strongly detected and suppressing the ones that were not seen by the network - formulated otherwise; we create positive feedback between image and the activations of a specific layer. The result is a creative, dream-/nightmare-like image. Thus the practical use of this method is negligible, but it gives us a sort of empirical evidence that convolutional neural networks may have association abilities similar to humans.

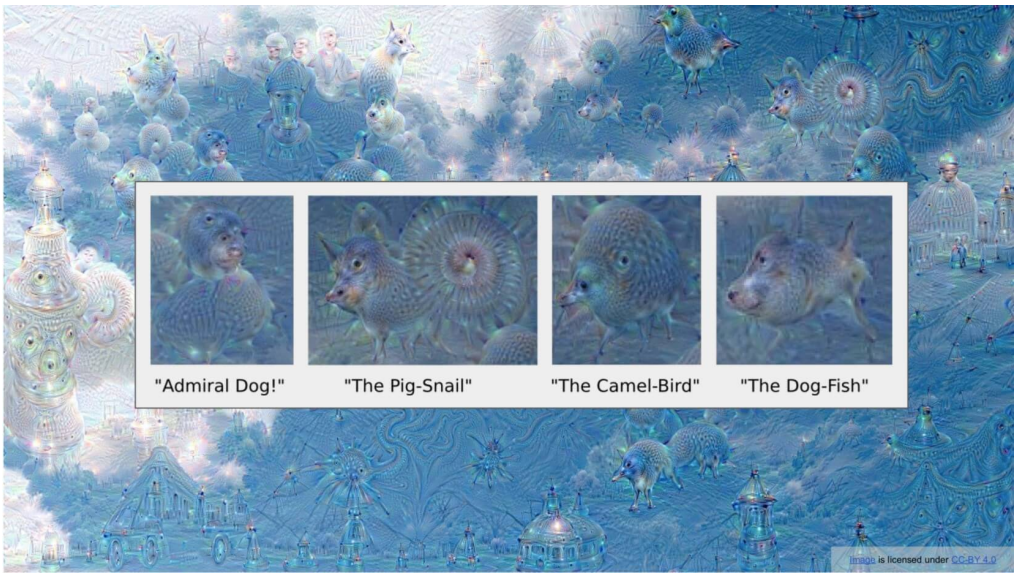

Figure 9.20: *Deep dream .*

Further Reading

[18] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007%2Fs10710-017-9314-z.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015. DOI: 10.1109/cvpr.2015.7298594. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2015.7298594.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016. DOI: 10.1109/cvpr.2016.90. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2016.90.

[23] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jul. 2017. DOI: 10.1109/cvpr.2017.243. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2017.243.

[24] L. A. Gatys, A. S. Ecker, and M. Bethge, *A neural algorithm of artistic style*, 2015. eprint: 1508.06576. [Online]. Available: http://www.arxiv.org/abs/1508.06576.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, 2015. eprint: 1412.6572. [Online]. Available: http://www.arxiv.org/abs/1412.6572.

10 Deep Learning in practice

10.1 Introduction

In previous lectures, we discussed the essentials of deep learning and convolutional neural networks, also a detailed introduction was given to process sequences and to the unique architectures intended to carry out particular tasks. Although theoretically, the methods seem to be easy to use, nonetheless, several difficulties arise in practice. Thus, this lecture aims to summarize those considerations and tricks that are essential to create working neural networks.

The four most important factors making training difficult are:

1. Numeric problems, which prevent the optimization algorithm from convergence.
2. The phenomenon of overfitting, which restricts the practical applicability of the trained model.
3. The nonexistence and the creation of the required amount (meaning a lot) of labeled data.
4. The proper hyperparameters needed for training and generalization.

10.2 Convergence problems

The gradient method and the backpropagation algorithm discussed previously work always in textbooks (thus, here too) correctly. Nevertheless, the finite resolution (and value interval) of floating-point numbers causes several problems regarding convergence. Generally speaking, floating-point arithmetic works well (i.e., it is stable) if the distribution of the numbers used is 0-centered and its standard deviation equals 1.

The reason for that is that during backpropagation several matrix multiplications are carried out, if our numbers are greater than 1 we diverge to infinity, if they are less then 1, we will reach 0 at the end. As a result, the gradient will either go to 0 (vanishing gradient) or diverge to infinity (exploding gradient) - the problem is more probable for the layers near the input because the chain of matrices is quite long there. To stay in the domain of nonzero but finite values, we need numbers near 1.

The above also holds for matrices; in that case, the matrix norm should be near 1. One of the most widely-used matrix norms is the so-called Frobenius-norm, which equals the sum of the squared elements - if the mean of the elements is 0 then it equals the variance.

10.2.1 Initialization

In the first subchapter regarding deep learning, we introduced the gradient method, which updates the weights with the help of the derivative of the loss function to get better results. Nonetheless, it was not mentioned how to obtain initial weights. Because we do not know at the beginning what kind of parameters can solve the task (if we would, the training process would be unnecessary), so we use a uniform initialization scheme. Note that it does matter how the variance is chosen (zero mean seems to be the obvious choice). If weights are big, activation values will be big too; thus, gradients can vanish more probably. The implication of that is that the algorithm makes

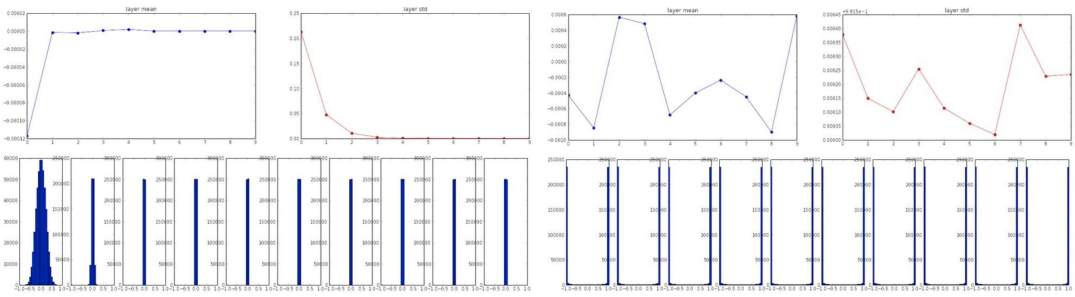

Figure 10.1: *The distribution of layer activations for small (left) and big (right) initial weight values.*

large steps, so it is probable to jump over the minimum. On the other hand, small weights result in small gradients; thus, the network can get stuck in the initial values.

To avoid the above problems, the variance of the weights should be selected with care (not too big, not too small). The most widely-used methods are the Xavier and He initialization schemes, where the initial standard deviation values are determined 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 denote the number of in- and outputs of the given layer. Note that these formulas would result in approximately normally distributed activations and gradients, but the ReLU activation introduces a slight distortion.

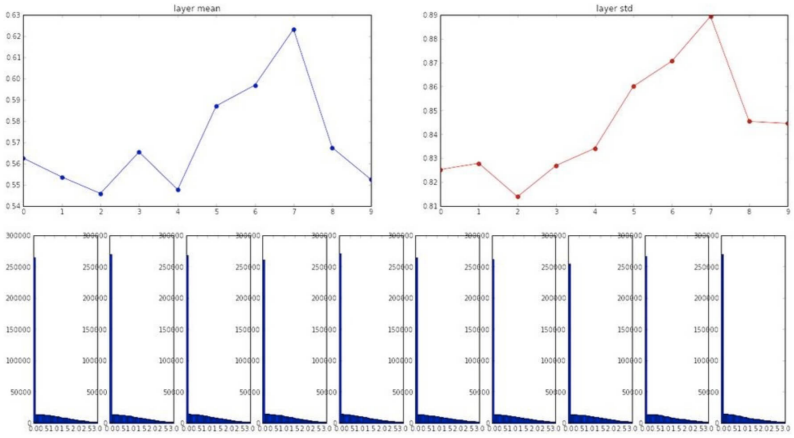

Figure 10.2: *Weight distribution in the case of using Xavier initialization.*

10.2.2 Data normalization

Due to similar considerations, it is necessary to transform the input images. As discussed before, for convergence reasons, it is essential to have an approximately normal distribution of pixels in the input. Namely, regardless of a proper initialization, if the input values are big, the result will be similar, as in the case of having big weights. Having 0 mean is also essential because having only positive or negative values will constrain the possible directions of the gradient.

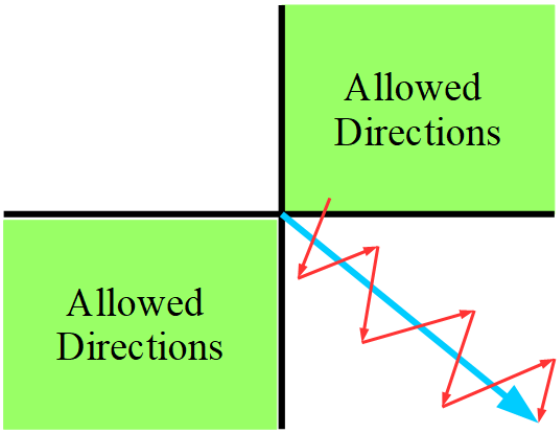

Figure 10.3: *For positive-only input values the gradient will be positive, for negative ones only negative, thus the proper direction can only be approximated in a zigzag manner.*

10.3 Validation and regularization

We mentioned before that as a result of the training we might suffer from overfitting, to detect this phenomenon we use two separate databases, one for training and one for validation - which can be subsets of the same database (a common choice is about 80%-20%). The training database is used for training only while the validation database is used to monitor overfitting, based on which hyperparameters can be tuned. It is important to note that the validation set is **NEVER** used for training.

In practice, two databases do not suffice; namely, the training set is used for determining the weights while the validation set is used for selecting the best possible hyperparameters. In doing so, we are unable to state anything about the performance of the model on a new dataset. Thus, a third dataset - called the test set - is used (it is rather small; approximately 10% of the whole database is chosen). To ensure the reliability of the method, we should separate these three databases (and do not use them for anything but only for training, validation, or testing.

Figure 10.4: *Splitting up a database into training, validation and test sets.*

Overfitting can also be characterized by the activation distribution of the layers. Namely, in the case of overfitting, the network does not try to capture general relationships between the input, but it memorizes the proper answer for each data point. This means that for each input, only a few activations will be maximal (which remember that specific input sample), the other will hold the opposite extreme value. As discussed before, such extreme activation values can occur easily if the weights grow too big, thus regularization methods which intend to slow down the growth of weights.

10.3.1 Dropout

To avoid unwanted activations, two methods are common, the first of which is the Dropout procedure, which sets some (selected based on a probabilistic scheme) of the activations in the forward pass to 0, and further activations are calculated considering those 0s (Fig. 34). This approach significantly decreases overfitting; thus, the network is forced to be redundant - note that Dropout is not used during inference; thus, activations should be scaled w.r.t. the probability chosen for Dropout.

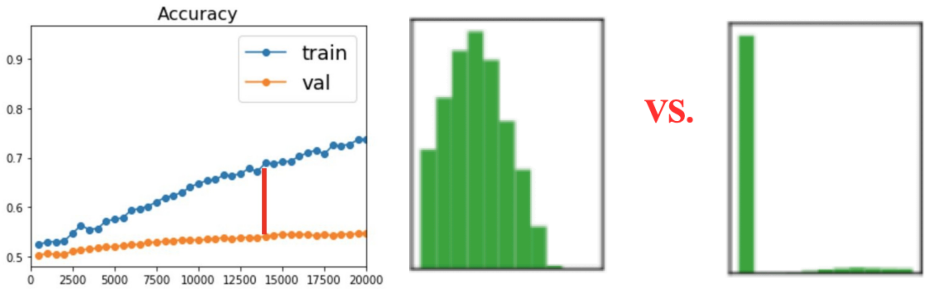

Figure 10.5: *Training and validation loss (left) in the case of overfitting and the distribution of activations in the normal case (middle) and in case of overfitting (right).*

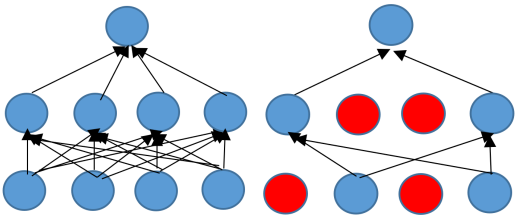

Figure 10.6: *Applying Dropout.*

10.3.2 Batch Normalization

The other solution is called Batch Normalization, i.e., a normalization operation is carried out after the layers selected. As mentioned previously, we evaluate not only one image but a minibatch (generally a multiple of 32) of them in a parallel manner. The main principle is that the mean and standard deviation of each activation is calculated in each iteration, and they are normalized based on the statistics calculated. Besides decreasing overfitting, numeric convergence properties are also improved - the formula is as stated below:

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

Where μ and σ/Σ are the estimated mean and standard deviation/covariance matrix of the inputs x, while α and β are parameters to learn. The success of Batch Normalization is shown by the fact that it can be considered quite essential - meaning each convolution layer is followed by one. Batch Normalization can be combined with Dropout; nevertheless, the majority of experiments show only marginal improvements. The advantages of Batch Normalization are the following:

- Normalizing the distribution of activations reduces overfitting.
- Due to normalization, the activations of the layers are approximately in the same order of magnitude, which improves convergence.
- Since the weights of all layers are changed simultaneously during optimization, the distribution of the inputs changes for every layer (except the first), forcing the layers to adjust continuously. Batch Normalization cancels this, making the optimization more stable (and consequently faster) as a result.

10.3.3 Data augmentation

Overfitting can be decreased with another approach too; the intuition for that is the following: in the case of overfitting, the network memorizes each training sample, thus feeding more data into

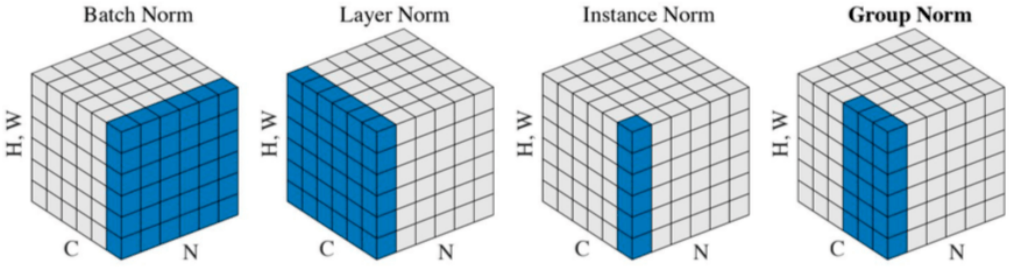

Figure 10.7: *Note that besides Batch Normalization several other variants exist: the whole activation array is normalized separately on an instance base (layer normalization). Instance normalization normalizes each channel separately along the spatial dimensions. The compromise of both is called group normalization, where some channels are treated together.*

the network makes overfitting more difficult. As a result, using more training data almost always reduces overfitting. Creating bigger training databases is rather expensive; thus, it would not be feasible. In the case of images (for other data types, it is also possible to a specific extent), we can generate (partially) new data points, and this is called data augmentation. E.g., with the help of reflections, random cropping, rotations, scaling, intensity transformations, we can artificially enlarge the available dataset - these operations do not modify the image labels; thus they can be carried out with ease. It is important that Batch Normalization, regularization, and data augmentation are used all at once.

10.4 Hyperoptimization

A further difficulty we face while training neural networks is to choose proper values for the hyperparameters, which can be a lot, and we do not have another way than trial and error - thus the training process can last a long time (from a few hours up to weeks), so each trial is expensive. Thus, we can choose hyperparameters while we only use a small subset of the training set for training; this method can also help us to discover bugs.

As a result, only a few hyperparameters remain to tune for the case when we use the whole training set. For that, we can choose methods like grid search, meaning that we sample the hyperparameter space from an equally spaced grid. A better approach is if we use the same number of hyperparameter sets sampled randomly from the same space because it will have a better resolution. The latter is, first of all, in the case useful if some hyperparameters have a more significant influence on the result.

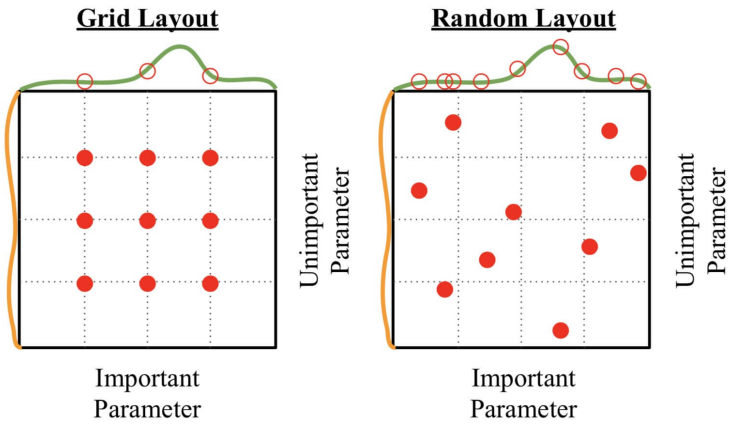

Figure 10.8: *The schemes used for hyperoptimization.*

10.4.1 Learning rate

Below, we will discuss the role of the learning rate in more detail because it determines the step length (i.e., the scaling of the gradient). Our goal is to reach the deepest point of the valley of the loss function with consecutive steps in the direction of the steepest descent. If the step size is too small, we will proceed very slowly, on the other hand, choosing a big learning rate can result in jumping over the minimum (although its neighborhood is reached fast), resulting in an endless oscillation, or worse, in divergence.

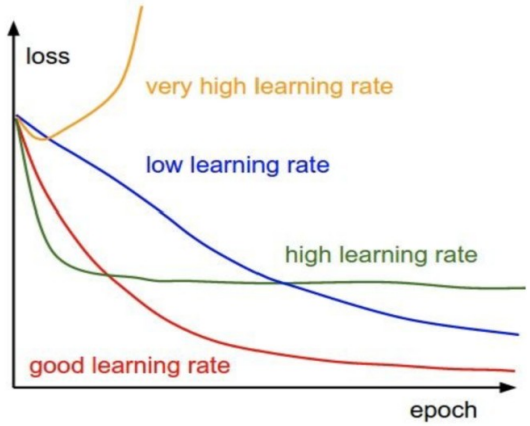

Figure 10.9: *The effect of choosing different values for the learning rate.*

Thus, in practice, we do not use only one learning rate. In the beginning, a higher value is chosen (the highest that shows a stable decrease w.r.t. the loss), thus we reach the neighborhood of the minimum relatively fast. However, we decrease the learning rate after a while to reach the optimum as closely as possible. This can be interpreted as a combination of a coarse optimization and a finetuning process. The most widely-used method is to reduce the learning rate with a fixed factor after a given number of steps.

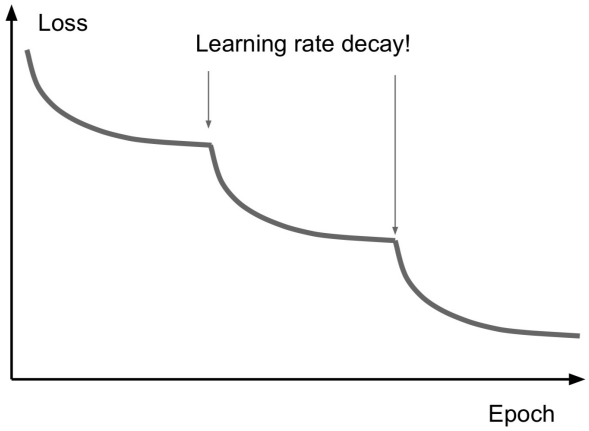

Figure 10.10: *Adaptively changing the learning rate.*

Of course, we can monitor the learning speed and change the learning rate adaptively according to that. In this case, we decrease the learning rate if the previous best result has not improved in the last - given number of - steps. Cosine annealing is also useful, which modulates the learning rate based on the first half period of the cosine function. Thus at the time the half period ends, the learning rate is increased significantly (to its maximum value specified as a constant), which can help us to get out of local minima and to reach a better one.

10.5 Constructing databases

The third difficulty is to obtain enough labeled data to train a neural network. This problem can be mitigated to a certain extent with the help of data augmentation; furthermore, we can use the approach of semi-supervised learning, meaning that only a small part of the database is labeled. Thus, we expect that the network will assign labels, which will be similar if the data entries are similar to the labeled entries.

10.5.1 Transfer learning

A breakthrough of deep learning can mitigate this problem. As we have seen, the first layers of a convolutional neural network (consisting of convolutional and pooling layers) detect different image features. The further away the layer from the input (thus the nearer to the output), the more complex the features they are responsible for. Nevertheless, the first layers can be used for other tasks; thus, we only need to retrain the last few layers - training fewer layers (i.e., fewer parameters) means we need fewer data.

This method is called transfer learning and is a widely-known approach in deep learning. In most cases, it is even enough to retrain only the last, linear layer of the network. In other situations, fine-tuning may be needed for near-input layers; however, even this process would require significantly fewer data.

10.6 Installation

The last topic of this chapter is the topic of preparing neural networks for practical use (called installation). During installation, we face two problems: the networks have millions of parameters; thus, the size is rather big. Not to mention that the execution also needs billions of operations; thus, they are quite slow, which means a constraint for their applicability in low-performance devices.

10.6.1 Pruning

Among the solutions for the problems mentioned above, the most important is called pruning, which means that the least significant part of the weights is selected and neglected. For ranking the weights, there are several methods; the simplest is to consider their absolute value. The weights labeled as insignificant are forced to 0 while finetuning the network. Iteratively applying the above two steps can lead to the deletion of 90% of the weights while only losing a few

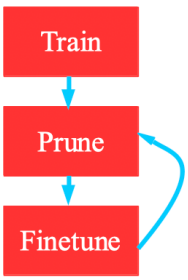

Figure 10.11: *Pruning.*

10.6.2 Weight sharing

A similarly efficient method is called weight sharing, which is as follows: the weights are clustered, and they are substituted with the centroid of the corresponding cluster. The centroids are finetuned, then both steps are iteratively repeated. Weights of an average neural network can be divided into 16 clusters; thus, we only need to encode them with 4 bits, resulting in an eightfold memory saving compared to the 32 bits floats generally use.

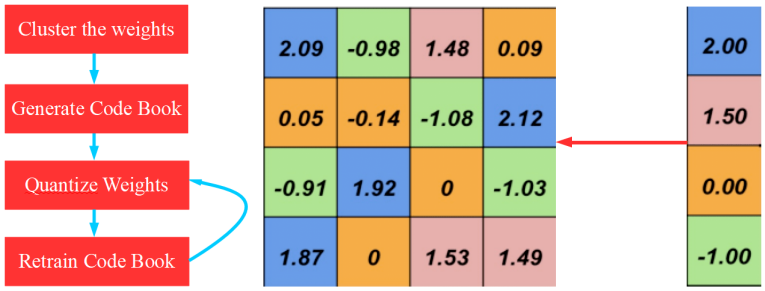

Figure 10.12: *Weight sharing (left) and the scheme of weight quantization (right).*

10.6.3 Ensemble

Another interesting approach is using model ensembles, i.e., several different network architectures initialized differently are used. The result is obtained by averaging the results of the members of the ensemble, which - as practical solutions show - can improve the performance of the best network up to 2-3%. This approach is also called the method of expert systems.

Further Reading

[18] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007%2Fs10710-017-9314-z.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Dec. 2015. DOI: 10.1109/iccv.2015.123. [Online]. Available: https://doi.org/10.1109%2Ficcv.2015.123.

[27] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. eprint: 1502.03167. [Online]. Available: http://www.arxiv.org/abs/1502.03167.

11 Detection and segmentation

11.1 Introduction

In the introductory lecture, several important tasks of computer vision were mentioned, but only the implementation of classification was discussed until now. In the following, we proceed with object detection and segmentation.

11.2 Semantic segmentation

The nearest task to classification is semantic segmentation, where all pixels of an image are classified. Of course, this can be carried out with a neural network trained for classification using a sliding window, but given the average size of an image (with hundreds of thousands or even millions of pixels), the execution would be rather slow.

Figure 11.1: *The task of semantic segmentation.*

11.2.1 Fully convolutional networks

Thus, it would be feasible to parallelize the process so that it could be carried out by one neural network. For that, FCNs (Fully Convolutional Network) are used, which are composed of the sequence of convolutional and activation layers. The last layer is also convolutional; the number of output channels equal to the number of classes, thus the elements of the output activation map represent the classification of the pixels. The problem of the architecture is that convolution for the original resolution is rather expensive, thus downscaling should be incorporated - but in that case, the output will have fewer dimensions, which is undesirable.

FCNs used in practice consist of an up- and downscaling part, which are more or less the mirror image of each other. That way, the output will have the same resolution as the input. FCNs consist of short circuit connections in general between the up- and downscaling units of the same resolution, which helps the propagation of gradients, thus the convergence of the training process. Another advantage of those connections is that they help to propagate the activations of high-resolution low-level features from the beginning of the network, thus helping to determine the borderline of the classes (those would get lost during downscaling).

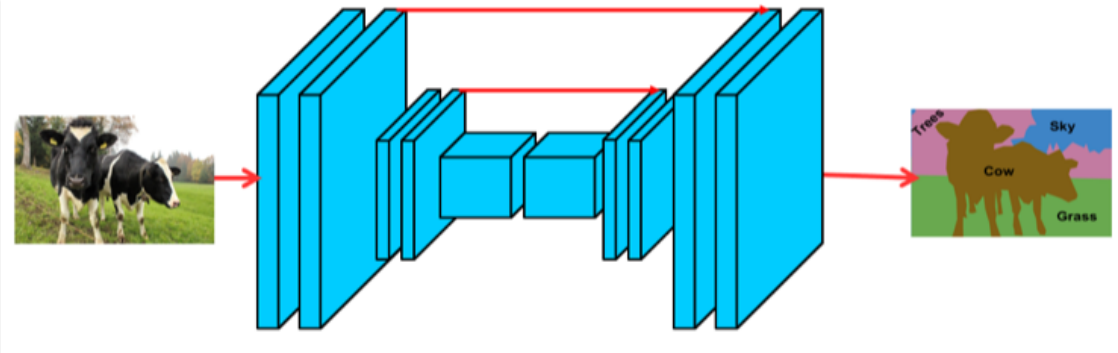

Figure 11.2: *A typical FCN architecture .*

The performance of FCNs can be increased with several other methods, one of the simplest among which is the usage of residual or dense blocks in the downscaling part. That way, a deep network with a big effective field of view (FoV) can be used for segmentation without complicating convergence.

Another possibility is to use dilated convolutional layers (sometimes referred to as atrous convolution, i.e., convolution with holes). The effect of dilation is that the effective FoV of the filter will be enlarged - while the number of parameters stays the same. Thus, the same network can investigate a bigger context, resulting in improved consistency of segmentation. Similarly, using Spatial Pyramid Pooling can be used, which carries out multiple pooling operations on the activation map at the end of the network in a parallel manner. The resulting activations are concatenated, and the classification is done on the concatenated activations. This approach has the advantage of being less sensitive to the scale of objects. Spatial Pyramid Pooling is sometimes also done in a dilated manner - based on similar considerations.

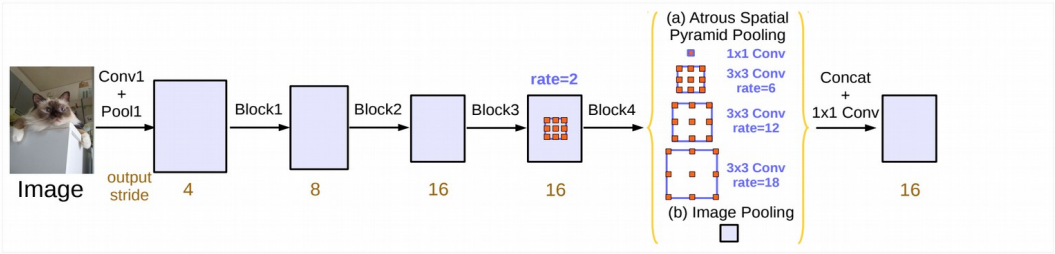

Figure 5. Parallel modules with atrous convolution (ASPP), augmented with image-level features.

Figure 11.3: *An architecture using dilated Spatial Pyramid Pooling.*

11.2.2 Upscaling methods

The only question left unanswered is how to implement upscaling in convolutional neural networks. One of the simplest ideas is called unpooling, which stores the position of the maximum during the maximum pooling done in the downscaling part; in the upscaling part that position will get the value copied from the lower level, while the others are set to 0.

Another widely-used operation is the transposed convolution, which is basically the inversion of strided convolution. The name comes from the fact that convolution can be described in terms of matrix multiplication, transposed convolution is the multiplication with the transposed of that matrix. The main advantage is that upscaling is learnable, which improves the quality of segmentation significantly.

The matrix the name originates from, is as follows (i.e. the matrix representation of the convolution):

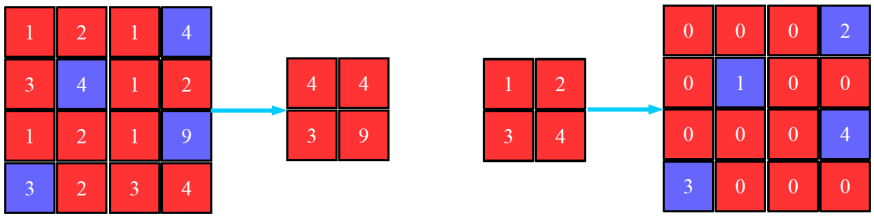

Figure 11.4: *Max unpooling.*

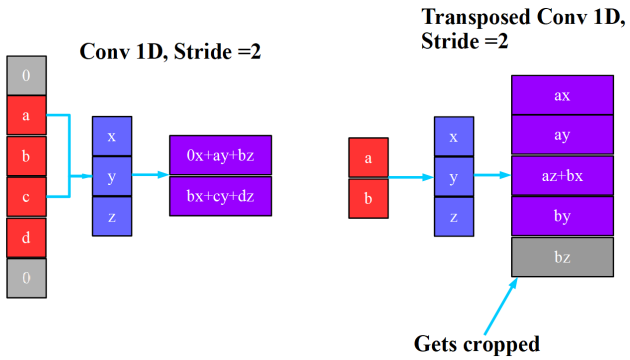

Figure 11.5: *Trsnposed convolution in 1D.*

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

Transposed convolution equals the multiplication with the transposed of that matrix. Note that transposed convolution is often referred to as deconvolution, which is not a proper name because the inverse and the transposed matrices are not (generally) the same.

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

The third widely-used method is called DUC (Dense Upsampling Convolution), which increases the channels of a given activation layer fourfold with the help of a convolutional layer. After that, we reshape the intermediate result so that the activation map should have a twofold size compared to the original, while the number of channels stays the same as that of the original image. This method can also be learned, but it has more parameters, which makes it possible to learn more complex transformations while accepting slower operation.

11.3 Detection

After finishing semantic segmentation, we will discuss the topic of object detection. In that case, the extracted feature is somewhat simpler (generally bounding boxes), but it is also possible to separate each object.

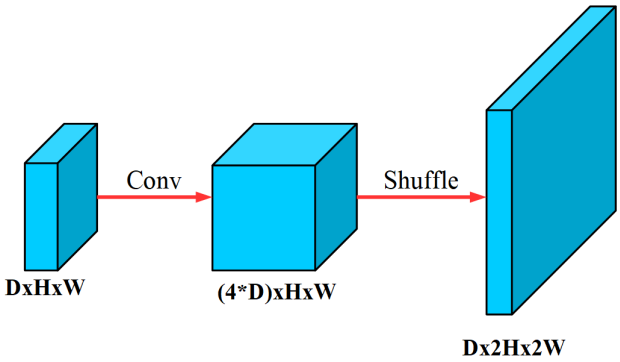

Figure 11.6: *Dense Upsampling Convolution (DUC).*

11.3.1 Localization

One of the most straightforward variants is the localization problem when the classification task is extended with determining the bounding box of the object. This task can be easily solved with the help of neural networks; we only need to add four additional outputs to the classification network. The latter four outputs are prescribed to output the four parameters of the bounding box as accurately as possible. This is a regression problem; thus, the MSE loss function should be used for determining how the parameters fit. The whole loss of the localization network will be the sum of the losses of classification and regression.

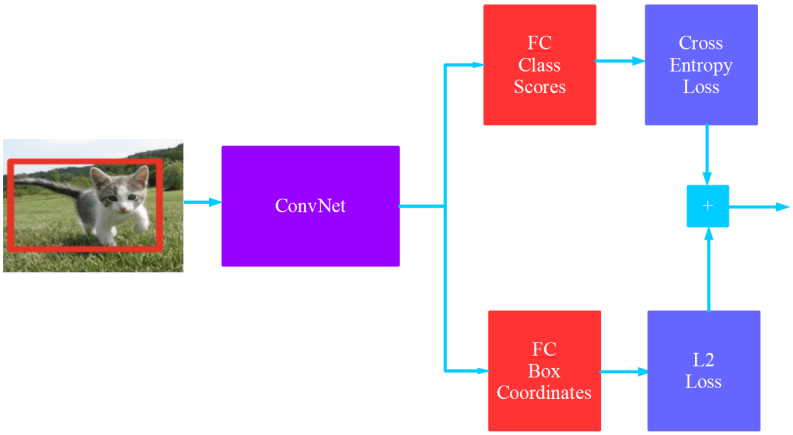

Figure 11.7: *The architecture implementing localization.*

11.3.2 Region-CNN

The detection task is much harder, namely any number of objects corresponding to any classes can occur, which should be reflected by the architecture too. Of course, we can give a coarse upper bound for the number of objects; thus, we could create a convolutional network, which has the same number of classification and bounding box prediction outputs. This would mean for maximum N objects and C classes $N*(C+4)$ outputs, which can be countless, considering that N can be some dozens, while C is generally in the range of hundreds or even thousands.

An alternative would be to use region proposal methods. These methods are traditional segmentation methods, which can be used to have region proposals. Given the region proposals, we run a convolutional neural network on each of them separately - this method is known as R-CNN (Region Convolutional Neural Net). Note that the classification output of the localization network should contain a "none of them" output to be able to filter out region proposals which do not contain any class.

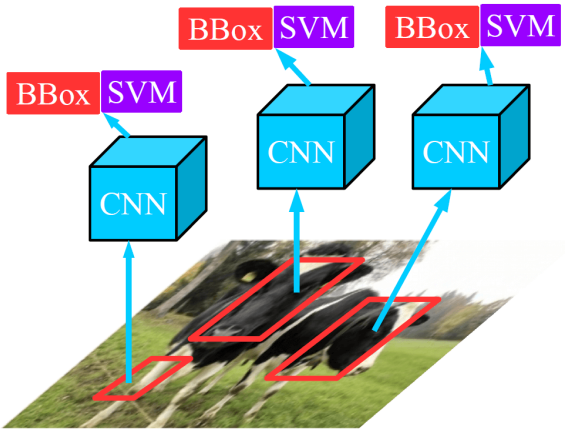

Figure 11.8: *The R-CNN architecture.*

The main drawback of the R-CNN method is that the neural network is executed separately on each region proposal, which is a waste of resources. The improvement of R-CNN is called FastR-CNN, which executes on the whole image a network consisting only of convolutional and downscaling layers, the region proposals are determined on the activation map of that. These proposals are scaled to the same dimensions with the help of a special pooling operation (traditional pooling scales only with a specific factor). After that, a small neural network - consisting only of linear layers; this network is responsible for the classification and bounding box localization - is utilized on the region proposals (Fig. 32.). The speedup gained is about 10-20 times.

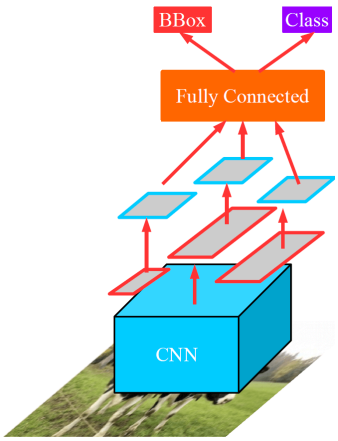

Figure 11.9: *The Fast R-CNN architecture.*

The slowest part of FastR-CNN is constructing the region proposals (about 90% of the runtime); thus, another improvement was introduced, which uses the RPN (Region Proposal Net) neural network for region proposals. This architecture constructs a fixed number of region proposals from the activation map after the convolutional part at the beginning, each of which undergoes binary classification (object/not an object). The latter is needed because - as the number of outputs is fixed - the number of proposals is also fixed. Besides training the main detector network, the RPN is trained to predict the bounding boxes and object-like nature of the regions with the highest accuracy. This modification means another factor of 10 w.r.t. FastR-CNN.

11.3.3 YOLO

Not only region proposals can be used to make object detection efficiently, but a perfect example for which is also the YOLO (You Only Look Once) architecture - do not mistake it for the adage with the same abbreviation. YOLO is similar to the solution of having lots of localization outputs,

discussed at the beginning of the detection part. YOLO feeds the image through a network consisting only of convolutional and downscaling layers, which gives the activation map used for the final prediction.

Which is done in the following way: the image is split up with the help of an NxN grid, and B bounding boxes are predicted for each cell of the grid. Each bounding box has its corresponding classifier with C outputs and a binary classifier, which predicts the object-like nature of the content of the bounding box. Thus for each cell, the number of outputs equals Bx(5+C), which is determined with a 1x1 convolutional filter. Note that YOLO predicts the position of the bounding boxes w.r.t. the upper left corner of the cell, so for the detection of each object that specific cell is responsible which holds the center of the object.

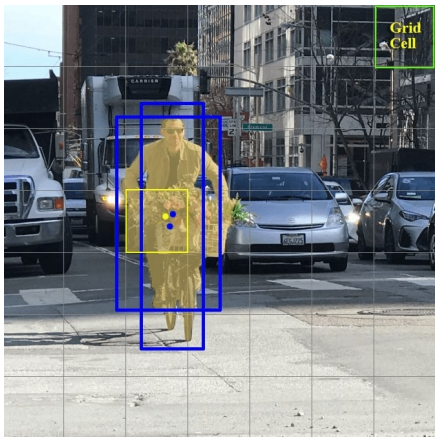

Figure 11.10: *The grid of YOLO and the predictions.*

To predict the width and height of the bounding box, YOLO uses one of B anchor boxes (one for each prediction). The base widths and heights of the anchor boxes are determined by clustering the boxes in the training dataset using B clusters. It is also worth noting that YOLO may find a single object multiple times during detection, therefore if we have overlapping detections of the same class, only the one with the highest confidence value is kept, the rest is discarded. This step is called non-maximum suppression.

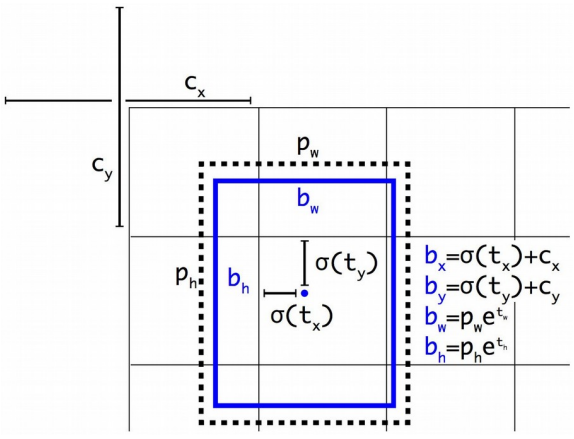

Figure 11.11: *YOLO bounding box prediction. The center of the box is predicted relative to the top left corner of the grid cell, while the width and height are predicted relative to the anchor box.*

YOLO has multiple versions; for instance, anchor boxes were introduced in version 2. Version 3 introduced bounding box prediction at multiple scales, which is achieved by upscaling the lower resolution activation maps using the methods discussed in the segmentation chapter. This trick drastically improves YOLO' s ability to accurately detect small objects, which is sometimes struggling to achieve. The main advantage of YOLO over region-based detection is its higher speed,

allowing it to run real-time on certain dedicated hardware solutions. This is especially true for the Tiny YOLO variant, which sacrifices even more accuracy for speed.

11.3.4 Mask R-CNN

At the end of the subchapter, we concentrate a bit more on the last task, which is object segmentation. In this case, the image should not only be segmented semantically, but the objects corresponding to the same class should also be distinguished. Although being the most complex task, such an architecture can be understood based on the previous ones, which led us to this point. During RPN-based object detection, the image parts consisting of the objects were calculated; thus, our current task differs only in the determination of a binary mask for each object instead of a bounding box. This binary mask can be calculated with the help of the upscaling network part known from the semantic segmentation task. This architecture is referred to as Mask R-CNN.

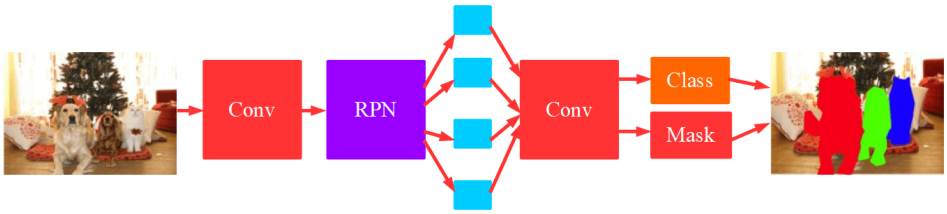

Figure 11.12: *The Mask R-CNN architecture .*

Further Reading

[18] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007%2Fs10710-017-9314-z.

[28] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015. DOI: 10.1109/cvpr.2015.7298965. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2015.7298965.

[29] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-CNN: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017. DOI: 10.1109/tpami.2016.2577031. [Online]. Available: https://doi.org/10.1109%2Ftpami.2016.2577031.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016. DOI: 10.1109/cvpr.2016.91. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2016.91.

[31] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask r-CNN," in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2017. DOI: 10.1109/iccv.2017.322. [Online]. Available: https://doi.org/10.1109%2Ficcv.2017.322.

12 Recurrent networks

12.1 Introduction

Until now we have discussed methods which can process images which are independent of each other. Nonetheless, processing image sequences has several important applications, among others video classification, i.e., event detection. As the identification of objects contained in images is needed for some applications, it could be more useful if we can identify events or actions. A somewhat different application is image captioning, which assigns not only a label but a complete sentence to images (resulting in a more elaborate description) - in this case not the input but the output of the network can be interpreted as a sequence.

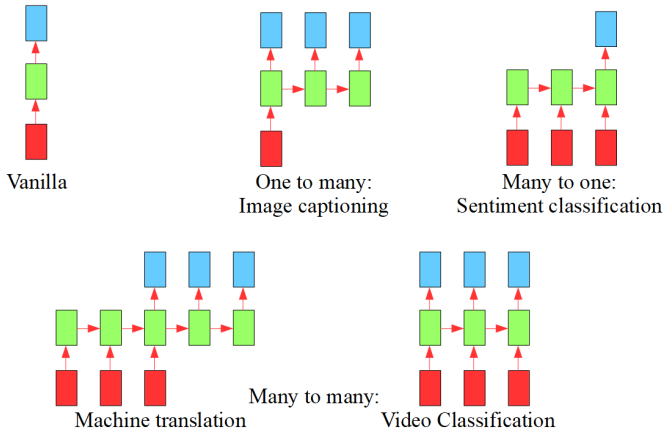

Figure 12.1: *Different sequence processing tasks.*

12.2 Recurrent neural networks

Feedforward convolutional networks do not have memory; thus, they are not suitable for processing time sequences. Thus we need a new architecture that has an inner state - these networks are called RNNs (Recurrent Neural Networks). The value of the current state is calculated based on the actual input and the previous state, and the output depends on the actual value of the inner state; thus the equation of an RNN cell is:

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

Where h is the inner state, t is the actual time step. Thus an RNN cell is the composition of three linear and one activation layers. The new architecture implies the question of how gradients can be calculated in this case. Namely, backpropagation does not work for recurrent architectures, but we can solve this issue with a small trick: a recurrent network can be transformed into a feedforward one if we unroll it in time. This means that the states of the RNN layer are considered as separate consecutive layers for different time steps.

An RNN cell has an in- and output in each timestep, thus each layer of the unrolled network will also have one output and loss, the sum of which gives the aggregate loss value. From this point, the

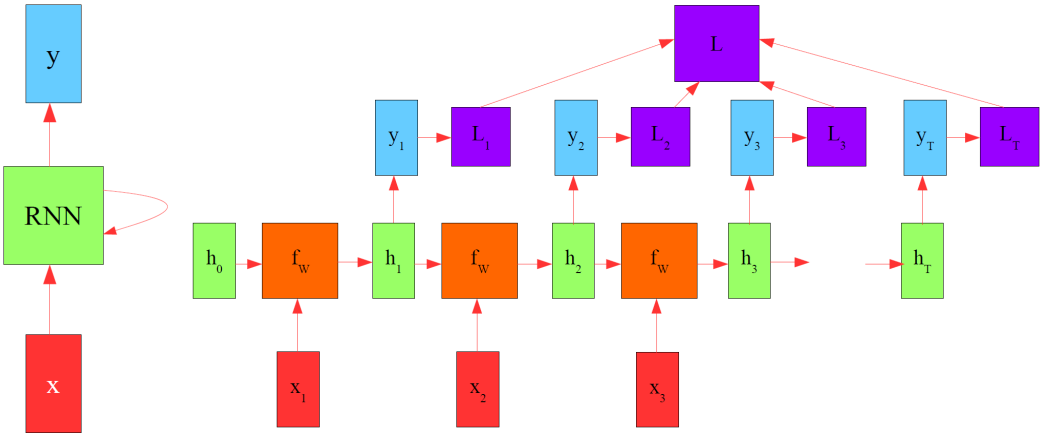

Figure 12.2: *The structure (left) and unrolling (right) of an RNN cell.*

backpropagation algorithm can be used as before. However, there are two important differences: first, as we proceed through time, the size of the network grows; thus, training will be slower. Because we do not need to (and are not able to) have infinite memory, we constrain the length of the network by deleting the oldest layers and inputs from the unrolled network.

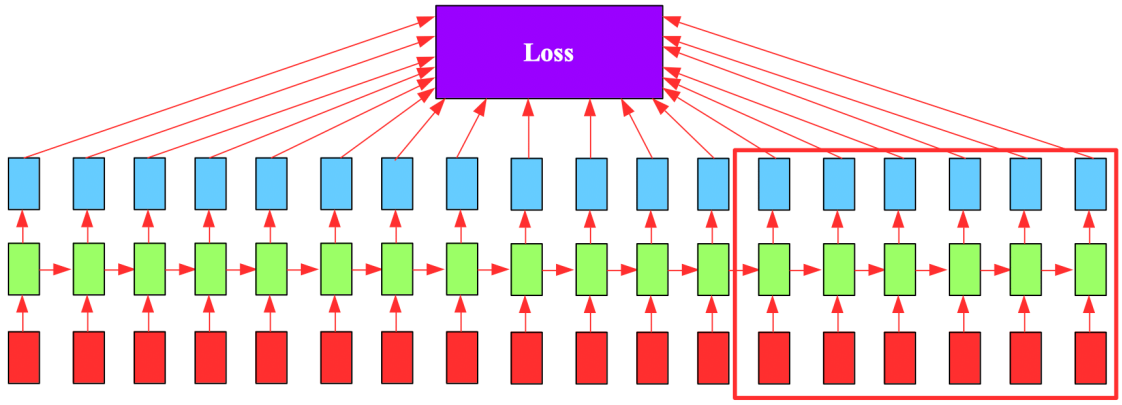

Figure 12.3: *The principle of Backpropagation through time (BPTT). When we are at the end of the network, backpropagation is only executed until the part with the red bounding box, or the problem size would grow towards infinity .*

12.2.1 LSTM

The second difference is that each weight matrix is the same for the unrolled multilayer network; hence we have only one layer. Thus, as we calculate the gradient with the chain rule, we will have a long sequence in the chain. In this case, every element of the product is the same, i.e., it is a power, which will be either zero or infinity for a big enough power except if the number the power of which we calculate is exactly 1. Thus the gradients of an RNN cell can easily explode or vanish, which makes the learning process almost impossible.

The only solution for this problem is to construct such a structure, where the derivative of the inner state w.r.t. to the previous state is about one. This is accomplished in the LSTM (Long Short-Term Memory) cell. The name has its origin in the intention of its designers to create such a short-term memory cell, which can remember in the long term in contrast to the RNN cell. While an RNN cell consists of three linear units, LSTM has four activation functions, which are called gates .

Without the effect of the gates, the previous value of the c cell state will be copied without modification into the actual state; thus, the derivative between them equals one. However, the

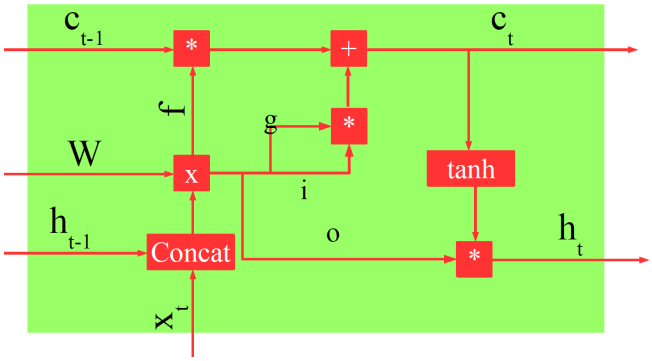

Figure 12.4: *The structure of an LSTM cell.*

cell state may be modified by the gates. The first such gate is the f forget gate, which is a vector with the same dimensions as the cell state - the values of which are between 0 and 1 due to the sigmoid nonlinearity. The elementwise product of that vector with the cell state results in partial forgetting of the state values.

The next gate is the gate g, which is intended to extract that features from the actual value of the input and the previous values of the output that should be remembered in the cell state. After that, the vector of the i input gate will be used for elementwise multiplication with the vector of the gate (analogous to the forget gate), thus selecting the relevant parts from the features to remember, the result is added to the cell state. The last step is to produce the actual output of the LSTM cell, which will be the cell state filtered with the o output gate.

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

Where $*$ denotes elementwise multiplication. Note that there are several variants of the LSTM cell, which are slightly different, and also of cells that differ more significantly, but the base principle is the same. To the latter belongs the GRU (Gated Recurrent Unit). Also note that the idea to help to propagate the gradients backward with "short circuit" connections is not here used at first, e.g., consider the residual block discussed in the previous chapter, which had a similar base principle.

12.2.2 Applications

Besides classifying videos, there are lots of other different applications of sequence processing methods, from which image captioning can be highlighted. This means that a description of 1-2 sentences is assigned to the input image. In that case, the recurrent connection is established concerning sentence generation. Another important application area is the implementation of different (visual) question-answering systems. In that case, the algorithm is supposed to answer a question asked in connection with an image or a video (E.g., "What is the color of the hat the man with sunglasses is wearing?"). We can also mention the implementation of different systems with explicit memory (e.g., Turing machine).

A rather interesting application of recurrent networks is the so-called soft-attention model, which means that a recurrent cell - based on the content of the image - produces a weight value for each image part, which are then taken into consideration with those weights weighted. The weights are newly generated in each step; thus, the focus of the network in the image changes continuously. During image captioning, it can be observed that the networks focus on that spot, which indicates the object corresponding to the word generated.

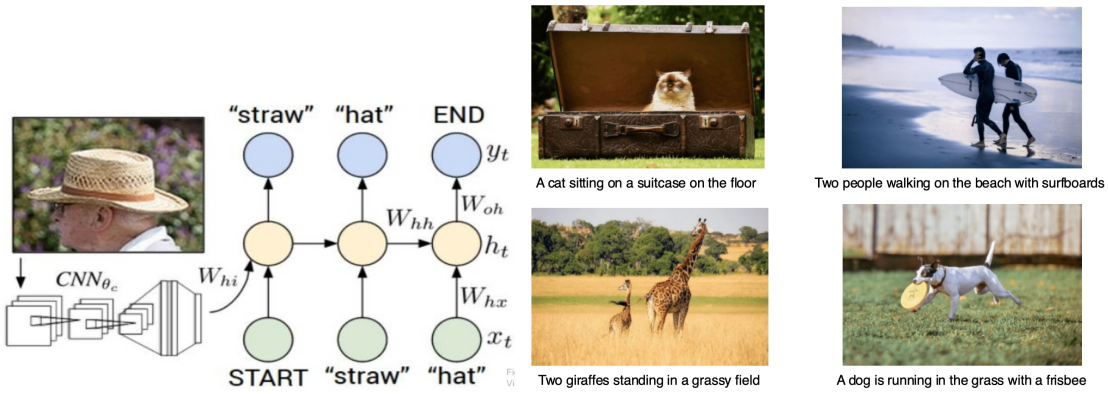

Figure 12.5: *The principle (left) and result (right) of image captioning.*

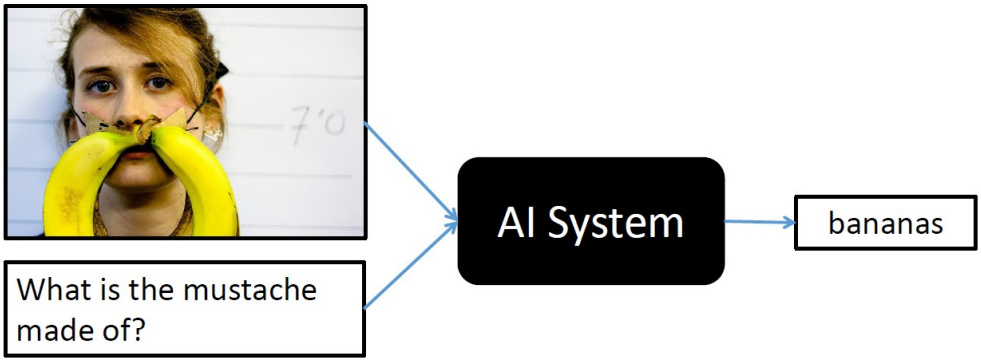

Figure 12.6: *The problem of the visual question-answering system.*

Further Reading

[18] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007%2Fs10710-017-9314-z.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162%2Fneco.1997.9.8.1735.

[33] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio, *Show, attend and tell: Neural image caption generation with visual attention*, 2016. eprint: 1502.03044. [Online]. Available: http://www.arxiv.org/abs/1502.03044.

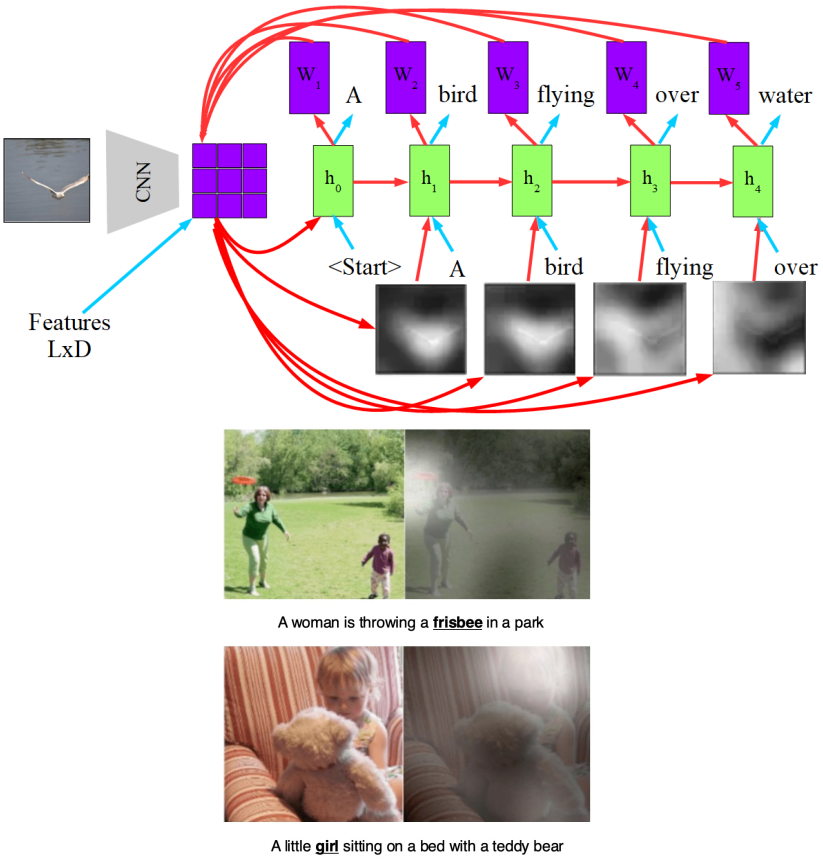

Figure 12.7: *The soft-attention principle (left) and its effect (right) onto image captioning. In the images the attention weights are shown as the underlined words are generated.*

Part III

3D Vision

13 Camera model and Calibration

13.1 Introduction

The main goal of computer vision is to get information automatically from the real world, based on camera images, with the help of a computer algorithm. The majority of the used imaging systems constructs a 2D projection of the 3D world, which means loss/distortion of data. In a typical image, the distance of the pixels from the camera is not retained; thus, they can only be estimated. Above that, due to the properties of projection, several - for our purposes important - geometric features get also distorted. On the other hand, objects with identical size will be different in the image if their distance from the sensor is different.

During projection not only sizes but also angles will be modified, which leads to the distortion of geometric shapes. A representative example is the concept of the vanishing point. As we know, parallel lines intersect each other in infinity - but if they are projected to an image with a camera, then this intersection point infinitely far away will also be contained in the image. I.e., the projection of a point infinitely far away can be finite, which is called the vanishing point.

Figure 13.1: *Distortion of 3D geometric properties and the vanishing point (right).*

As we can see, some important features of an object cannot be determined from one image (although shadows can be used for estimation). If we need that information, then we can use such devices that are able to measure and store also depth information for each pixel (RGB-D sensors). The problem is that those sensors are significantly more expensive than traditional cameras (the difference is about fivefold given the same quality), thus choosing them is not always possible/feasible.

Fortunately we have another way, namely, as in the case of human vision, we can exploit the fact that using two or more images, a part of the 3D scene can be reconstructed. This is the domain of 3D computer vision, which will be discussed in this chapter.

The requirement of at least two images originates from the following: given the same scene, if it is captured from different points of view, then those images will contain additional information. If the correspondence between the content of both images can be determined, we will be able to reconstruct the information distorted and lost during each projection. For that, we do not need two cameras under any circumstances, two images from different positions will suffice - given that the scene has not changed between the acquisitions, or the result will be incorrect. If the movement/change of objects cannot be controlled, the feasible choice would be to use two synchronized cameras.

13.2 Pinhole camera model

Thus, the goal of 3D computer vision is to reconstruct the information distorted/lost during projection. Before that, we need to understand the projection itself; thus, we need a mathematical model. After that, measurements should be recorded with a specific camera system to determine the unknown parameters of the model we built. This step is called camera calibration. In the subchapter below, we will investigate two cases of it: in the first case, the parameters of the projection of one camera, in the second, the relative position of cameras in camera systems will be determined.

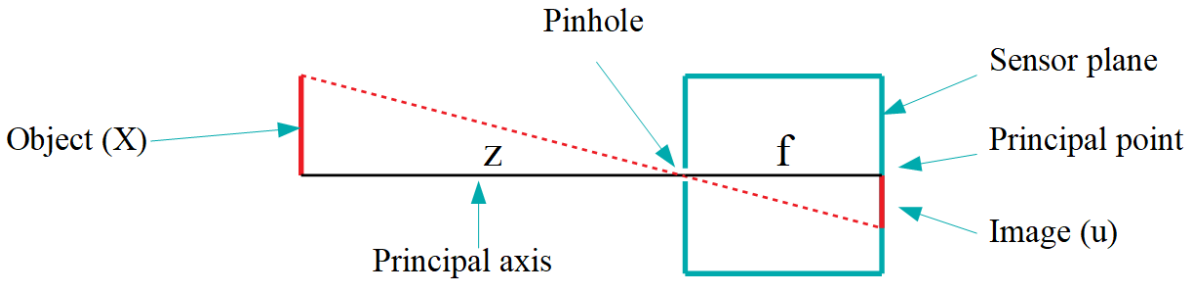

Figure 13.2: *The physical pinhole camera model.*

In computer vision, the pinhole camera model is used quite frequently, which can be imagined as a box, on one side of which there is a small pinhole, which is the opening where light can come in. As a consequence, at the opposite side of the box, a reversed image will show up - where in real cameras, the light sensor is. Another difference of real cameras is that they use a lens instead of a pinhole, which focuses parallel rays to a point; thus, it can substitute the pinhole. The application of the lens has the advantage to pass more light through the pinhole, but it causes geometric distortions. The pinhole camera model can be described with the following equations:

$$u = f_x \frac{x}{z} + p_x; \quad v = f_y \frac{y}{z} + p_y; \tag{13.1}$$

Where u and v are the pixel coordinates, x, y, and z the spatial coordinates of the object, f the focal length of the camera, and p is the principal point. Before discussing the camera model further, we need to make a detour. During camera calibration, we will numerically estimate the parameters (focal length, principal point, etc.) of the pinhole camera model. For the numeric estimations, it is crucial to formulate the problem in the way which is the most suitable for the estimation task at hand. Thus, for calibration - and generally, for problems of geometric nature - the so-called geometric coordinates are used.

13.2.1 Homogeneous coordinates

The use of homogeneous coordinates is popular in the so-called projective geometry, which is the expansion of Euclidean geometry enabling significantly more transformations. While the former is restricted to rigid transformations (translation, rotation), the latter also includes direction-dependent scaling, shearing, and projection. In the case of projective geometry, the Euclidean space is expanded with one additional dimension; thus, the projective plane can be described with 3, while the projective space with 4 dimensions. The conversion between the two geometries is determined with the equations below:

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \rightarrow \begin{pmatrix} \frac{X}{W} \\ \frac{Y}{W} \end{pmatrix} \qquad \begin{pmatrix} aX \\ aY \\ aW \end{pmatrix} = \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \tag{13.2}$$

The first two equations describe the conversion from/into Euclidean into/from projective geometry. An interesting consequence of these rules is that the multiplication of a point in homogeneous

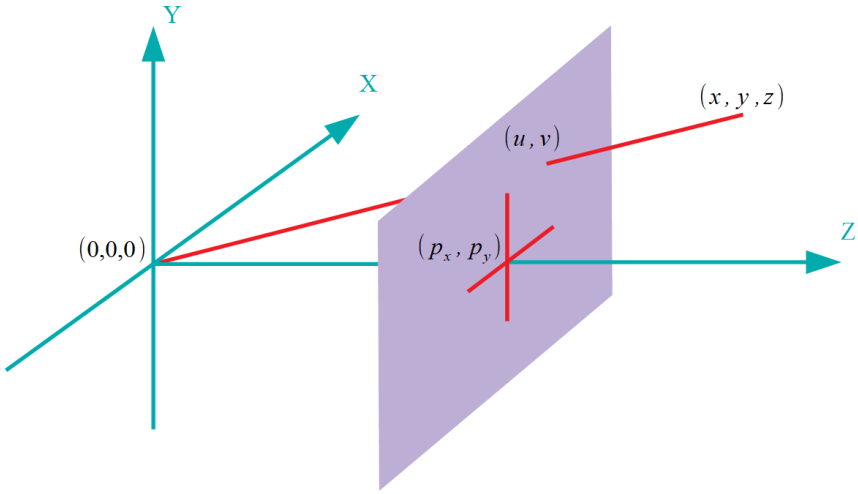

Figure 13.3: *The geometric model of the pinhole camera. Note that in this case the image plane is in front of the camera, thus the image is not reversed - of course, this is only a mathematical abstraction for simpler usage.*

coordinates with a nonzero scalar will result in the same point in the Euclidean space. This scale invariance is described with the third equation; thus the homogeneous coordinates corresponding to the same Euclidean point form a line, which intersects the W=1 plane at the Euclidean coordinates of the point.

Using homogeneous coordinates has two main advantages; the most important one is that the relationships of the pinhole camera model will be linear. The second is scale invariance, which helps a lot regarding the numeric estimation procedure. An interesting fact of homogeneous coordinates is that the $(x, y, 0)$ point exists, which is at infinite distance in the Euclidean plane, but in the projective plane, it can be described with finite coordinates. This "directed infinity" point is called an ideal point and has an important role for self-calibration processes.

Transforms

As noted before, homogeneous coordinates are used to describe geometric transformations. Before going further, it is reasonable to discuss what types of geometric transformations exist. The simplest of these is the so-called Euclidean or rigid transform, which allows the operations of rotation and translation only. As a consequence, Euclidean transforms always preserve the size and angles of objects. Arguably, however, these properties are distorted during the imaging process; therefore, a less restrictive transformation is needed. The next transformation type is the similarity transform, which - in addition to rotation and translation - also allows for uniform (equal in all directions) scaling. This transformation no longer preserves sizes, but still keeps angles. The two types can be described by the equations below:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} ar_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & ar_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & ar_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{13.3}$$

Where r_{ij} and t_i are elements of the rotation matrix and the translation vector, respectively, while a is the parameter of uniform scaling. Notably, the column vectors of the rotation matrix have unit length and are pairwise perpendicular, or, in other words, it is an orthogonal matrix.

The next transformation type is the affine transform, which introduces two new operations: non-uniform scaling, and shear. This type no longer preserves angles; however, it still preserves parallelism, making this transformation type more restrictive than imaging. The last transformation

type is the projective transform, which also includes perspective projection. This transformation no longer keeps parallelism; it preserves intersections only. The equations describing the last two transforms are as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} wx' \\ wy' \\ wz' \\ w \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{13.4}$$

Unlike previously, the matrices are composed of general elements a_{ij} .

13.2.2 Projection matrix

With the help of homogeneous coordinates the projection of the pinhole camera model can be described with one linear matrix multiplication:

$$\begin{pmatrix} wu \\ wv \\ w \end{pmatrix} = \begin{pmatrix} f_x & 0 & p_x \\ 0 & f_x & p_x \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = Ax \tag{13.5}$$

Where f is the focal length, p the principal point, u and v the pixel coordinates, A the so-called camera matrix. Note that the division with the z coordinate will only occur when we return to the Euclidean plane. Note that the equation is only true if the spatial coordinates of the point are given in the coordinate system of the camera. The center of the coordinate system of the camera is typically the pinhole, the x and y axis lay in the image plane, while the z axis is in the direction of the principal axis.

In the 3D space, there is another coordinate system, which is called the world coordinate system, and the coordinates of 3D points are generally given in that coordinate system. The world coordinate system generally depends on the specific application and situation; thus, it may be fixed to a physical object, but sometimes it can be chosen freely. In the latter case, it is often worth to choose it to be the same as the camera coordinate system, but there are a lot of exceptions.

$$\begin{pmatrix} u \\ v \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} wu \\ wv \\ w \end{pmatrix} \quad \leftarrow \quad A \begin{pmatrix} y \\ y \\ z \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} R & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ W \end{pmatrix} \tag{13.6}$$

Generally speaking, the world and camera coordinate systems are not the same; thus, the transformation between them should also be determined during calibration. Fortunately, the conversion between two coordinate systems is a simple Euclidean transform consisting of a rotation R and a translation t. Note that the unknown parameters can be divided into two separate groups. The first of which includes all the elements of the camera matrix A, which depend only on the internal structure of the camera, thus they are called intrinsic parameters.

The second group includes the elements of R and t; thus, it is independent of the internal camera properties, but not of that of the given setup; thus, they are called extrinsic parameters. Both parameter groups determine their geometric transformations, the composition of both is the projection matrix (P)

$$P = A[R \;\; t] \tag{13.7}$$

13.2.3 Real optical systems

In real cameras, a slightly modified setup is used because the pinhole is too small; thus, the amount of light passing through is not enough for quality imaging. Increasing the diameter of the pinhole would result in a blurred image due to increased geometric spreading (i.e., the circle of confusion, CoC, is bigger). Thus, a lens is used at the input opening of the camera, which substitutes the pinhole but has a sufficient size to get enough light onto the sensor.

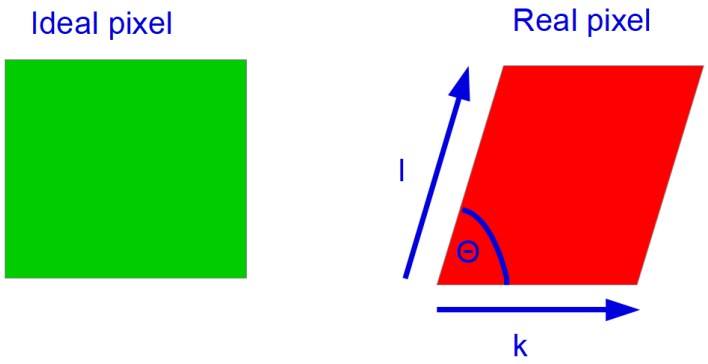

Figure 13.4: *Real pixels are not always squares, the aspect ratio may have deviations (first of all, in the case of old video cameras) or the angle between the sides can differ from 90 degrees.*

Adding a lens causes complications: light rays coming from the rim of the field of view of the camera will not be parallel, thus the refraction will be different. This phenomenon is called radial distortion, a common problem, first of all, for wide-angle lenses.

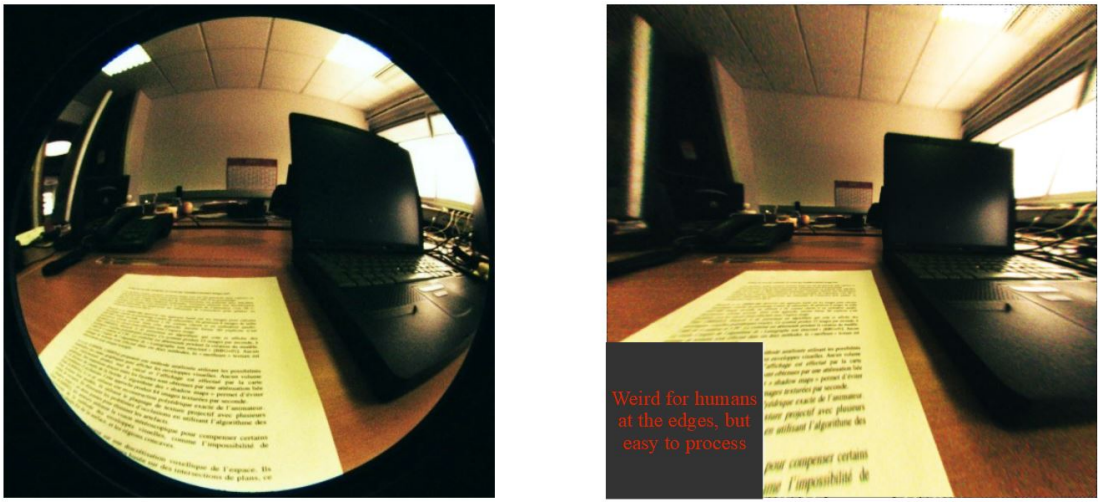

Figure 13.5: *Radial distortion (left) and undistorted image (right).*

13.3 Calibration

For 3D reconstruction, we first need to obtain the parameters of camera projection. From the analysis of projection, we concluded that it could be divided into two parts; the first depends on the position of the camera in the world, while the second on the intrinsic properties of the camera. This subchapter discusses mainly (but not exclusively) how to obtain the latter.

The approach is the following: as we know, the relationship between the points in space and their projection, after conducting several measurements, can be estimated numerically. In practice, we

use a reference object for calibration, which has easily distinguishable markers on it (in known positions). Capturing the calibration object and identifying the markers on the image, the estimation can be done from the corresponding point pairs in the image and in the real world.

Based on the reference object we use, different forms of calibration can be distinguished. The - from a mathematical point of view - simplest case is when the markers are not collinear or coplanar. Nevertheless, there are two- and one-dimensional calibration objects too - but as the markers are positioned in a special way, the mathematics behind it also gets more complicated.

A special case is called self-calibration, i.e., no calibration object is present, the correspondence between images is used to obtain the calibrated state. In this case, easy to identify and detect objects, called natural markers, are used. The main limitation of self-calibration is that it is constrained to the intrinsic parameters; external ones cannot be determined. Another drawback is that at least three different images are required, while with calibration objects, one suffices.

This chapter discusses calibration with 3D and 2D calibration objects, in both cases, the steps are the same:

1. Capturing several different images of the calibration object
2. Identification of the markers in the images
3. Determining the projection matrix
4. Separating intrinsic and extrinsic parameters
5. Prediction refinement

In most cases, calibration objects contain repeating shapes (e.g., squares), where the markers are the corners defined by the repeating shapes. For 2D calibration, almost always a chessboard-like object is used; thus, it is often called chessboard calibration. The main advantage that an accurate calibration object can be created at a small cost with the help of high-quality printers. For 3D reconstruction, chessboards are also used - in this case, two of them perpendicular to each other -, unfortunately, the accuracy is worse because the alignment of the boards is challenging to set up with high accuracy. Another advantage of using repeating shapes is that after defining the center of the coordinate system (e.g., the upper left corner of the chessboard), then the spatial coordinates of the markers can be calculated without complicated measurements.

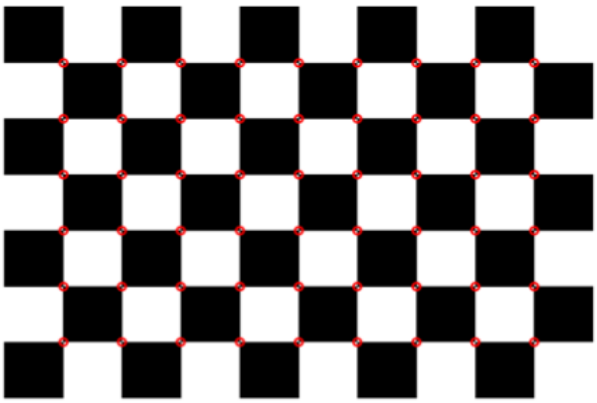

Figure 13.6: *A chessboard-like calibration object and its markers.*

Several images of the calibration object are captured from different positions - as we use numerical procedures to estimate the parameter values, the more measurements, the better. Note that this is only true for the intrinsic parameters, namely, as the images are taken from different positions, the extrinsic parameters vary, thus multiple images cannot be used for refinement. For the camera

itself does not change, the intrinsic parameters also stay the same. If the estimation of extrinsic parameters should be more accurate, then more markers should be used.

Markers are in general corner-like points on the reference object, thus using a corner-detection method (e.g., KLT, Harris) would be reasonable. The main drawback of them is that they do not have subpixel accuracy, which can result in big deviations if the problem is ill-conditioned. Thus, for chessboard-like references, it is common to detect edges, from which lines are extracted with the Hough transform. As it gives us a parametrized form, intersection points can be calculated with subpixel accuracy.

13.3.1 Calculating the projection

As in the $\vec{u} = P\vec{X}$ linear equation system $\vec{u}$ and $\vec{X}$ are known, only calculating P is left. Our problem is that for the solution, we need that $\vec{X}$ is unknown and not P as it is in our case. Nevertheless, the solution is easy: reordering the equation terms and determining the equation system for each point, we can use the method of Least Squares. First, we express the u coordinate as follows:

$$\begin{pmatrix} wu \\ wv \\ w \end{pmatrix} = \begin{pmatrix} p_1^T \\ p_3^T \\ p_3^T \end{pmatrix} \vec{X} \quad \rightarrow \quad u = \frac{wu}{w} = \frac{p_1^T \vec{X}}{p_3^T \vec{X}} \tag{13.8}$$

Where p_1^T is the first row of the projection matrix P. Similarly, the v coordinate is expressed , resulting in the equation system below:

$$\begin{aligned} up_3^T \vec{X} &= p_1^T \vec{X} \\ vp_3^T \vec{X} &= p_2^T \vec{X} \end{aligned} \tag{13.9}$$

Which has a form with matrix notation:

$$\begin{pmatrix} \vec{X} & \vec{0} & -u\vec{X} \\ \vec{0} & \vec{X} & -v\vec{X} \end{pmatrix} \begin{pmatrix} p_1 \\ p_3 \\ p_3 \end{pmatrix} = G\vec{p} = \vec{0} \tag{13.10}$$

Determining the equation system for the other point pairs, the G matrix is expanded with further rows. Note that during LS-estimation, we are forced to arbitrarily choose the norm of the P matrix, or the solution will not be unique. But as P operates on homogeneous coordinates, it is invariant to scalar multiplication, thus independently from choosing the matrix norm, the result will represent the same geometric transform for each scale factor.

The result of the step above, the matrix P is then split up into an A camera matrix and a $[Rt]$ rigid transformation. In the case of 3D calibration, the decomposition can be done due to the special properties of the rotation and the camera matrices with the QR decomposition method.

Note that using a chessboard-like reference object results in the fact that P cannot be determined, as the markers are coplanar, thus the problem is ill-conditioned. To circumvent the problem, we can estimate the so-called homography matrix H instead of the projection matrix P - for that we assume that the chessboard is in the $z = 0$ plane; thus the projection equation will give us the result:

$$\vec{u} = \begin{pmatrix} r_1 & r_2 & r_3 & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} r_1 & r_2 & t \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = H\vec{X} \tag{13.11}$$

From this point, H can be estimated with the LS-method, nevertheless, for getting the intrinsic and extrinsic parameters, a significantly more complicated mathematical toolbox is required than the QR-decomposition algorithm.

13.3.2 Linear regression

As mentioned above, the values of the P or H matrices are determined using the Least Square s (LS) estimation method. As a reminder, the model used is as follows:

$$X\vartheta = Y \tag{13.12}$$

Where each row of the matrix X contains the known parameters from a single equation, the ϑ vector consists of the unknown parameters, and Y is the vector of expected outputs. To obtain the optimal solution, we do not need any optimization algorithm; namely, it can be formulated in a closed form as follows:

$$\begin{aligned} ||E||^2 = (Y - X\vartheta)^T(Y - X\vartheta) = Y^TY - 2\vartheta^TX^TY + \vartheta^TX^TX\vartheta \\ \frac{\partial||E||^2}{\partial\vartheta} = -2X^TY + 2X^TX\vartheta := 0 \\ \hat{\vartheta}_{LS} = (X^TX)^{-1}X^TY \end{aligned} \tag{13.13}$$

Notably, in the case of camera calibration, Y is the zero vector - this is called Total Least Squares. This formulation also gives us the zero vector as a solution for the parameter vector, which is generally of no use. Thus, to solve this problem, we prescribe that the norm of the solution should be one to avoid the trivial solution. This means that that specific vector of unit length is to be determined, which is - after multiplication with the matrix X - nearest to the zero vector. This can be interpreted as the direction of the smallest "gain" of the matrix X, which is given by the singular vector corresponding to the smallest singular value.

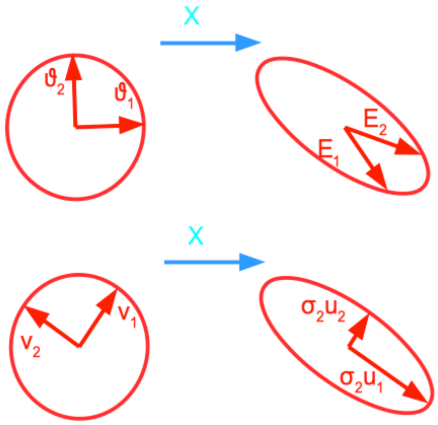

Figure 13.7: *Singular vectors and values: the TLS algorithm intends to find that vector of the unit circle, the length of which will be minimal after matrix multiplication. Assume an orthogonal vector system within the unit circle, then each ellipsoid (above) consisting of all transformed vectors can be constructed by the linear combination of that vector system. If a vector system would exist which is orthogonal both before and after the matrix transformation, then it can be concluded to be equal to the main axes of the ellipsoid. The components of this vector system are called singular vectors, which are the generalization of eigenvectors.*

Singular vectors and values can be defined as follows:

$$
\begin{gathered}
Xv_i = \sigma_i u_i; \quad X^T u_i = \sigma_i v_i \\
\sigma_i \geq 0; \quad v_i^T v_j = \delta_{ij}; \quad u_i^T u_j = \delta_{ij} \\
X = \begin{pmatrix} u_1 & u_2 & \dots & u_n \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_1 & \dots & 0 \\ \vdots & 0 & \dots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{pmatrix} \begin{pmatrix} v_1 & v_2 & \dots & v_n \end{pmatrix}^T = U \Sigma V^T
\end{gathered} \tag{13.14}
$$

Where σ_i is a singular value, u_i and v_i are the corresponding right and left singular vectors, δ_{ij} is the Kroenecker-delta .

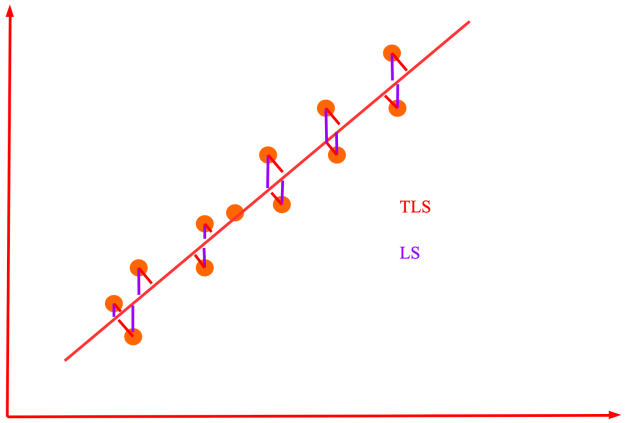

Figure 13.8: *The difference of the error definitions of the LS and TLS algorithms. Note that in the case of a vertical line the error of the LS method will go towards infinity.*

13.3.3 Geometric error

Note that as extrinsic parameters differ from image to image, these estimation steps should be carried out for each of them, each of which will result in identical camera matrices - apart from noise and errors - but the average will give us a quite good estimate. Note that estimated values come from solving an algebraic equation system while minimizing its error - but this algebraic error is difficult to interpret, i.e ., to identify from which aspect is the solution optimal.

Thus, we introduce the geometric error, which gives a measure of how far the projections calculated from the known spatial points are from the markers found in the image. This gives us an intuitive and easy to understand formulation, unfortunately, with a strongly nonlinear formula:

$$
E_G = \sum_i ||\vec{u}_i - f(X_i, P)||^2 \tag{13.15}
$$

Where u_i and X_i are the ith 2D-3D point pair, f the exact function of projection. Fortunately, the minima of both algebraic and geometric errors are near each other; thus, the camera matrix minimizing the algebraic error can be used as a good initial value for an iterative, gradient-based optimization algorithm. Such methods are very efficient even for complicated functions if they can start near the optimum. In the refinement step of camera calibration, the gradient, Newton, and Levenberg-Marquardt methods are often used.

13.3.4 Determining the lens distortion

Note the critical effect of geometric distortions, which originate from the properties or errors of the lens; thus, they belong to the intrinsic camera parameters. As their effect is nonlinear, they

cannot be incorporated into the camera matrix, and their estimation is also difficult. In the last, refinement step of camera calibration, we search for the minimum of a complicated, nonlinear function, which the effect of lens distortions can be added simply to; thus the refinement step can be used to obtain the distortion parameters too.

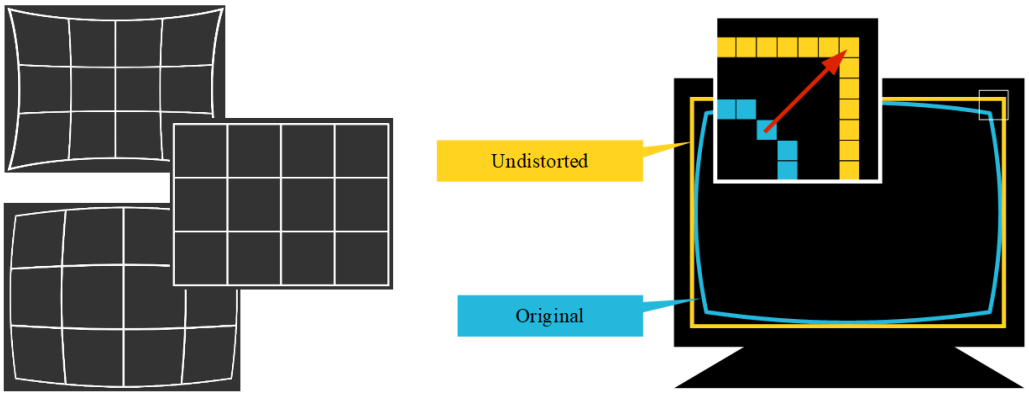

Figure 13.9: *Lens distortions are often corrected with the help of a lookup table. After determining the value of distortion and the position of each pixel in the distortion-free case, we can easily remove distortion with the help of the map obtained.*

13.4 Stereo calibration

For 3D reconstruction, we need multiple cameras or images acquired in different positions. Besides that, we need the geometric relationship between the cameras or images. In some situations, we may obtain that relationship from other sources (as is it a frequent case in mobile robotics), but often we only have access to the images. This subchapter discussed how to determine the relative position of cameras/images with the help of calibration methods.

Before exploring the methods, note that the nature and consequently the difficulty of the task depends significantly on the arrangement of the scene. The two cases - distinguished as before - are the one where we have two cameras at different locations and the other where the camera moves between images. In the first case, we have a fix camera system, where cameras do not move relative to each other; thus, calibration is only needed once at the beginning. In the second, as the camera moves continuously, movement estimation is needed between each pair of images used.

In the case of fixed camera systems, the internal calibration (which is needed) can be carried out at once for every camera. The only thing to pay attention to is that the calibration object should be visible for each camera. Remember that as a result of any method that uses a calibration object we get not only the intrinsic but also the extrinsic (reference object - camera transformation) parameters, for each calibration image.

If we know the transformation between the cameras and a selected point, we can easily calculate the transformation between the cameras. Furthermore, as the relative position of the cameras in the system stays constant, every image can be used for estimation. Thus for fix camera systems, a separate step for calibrating the system is not needed, as the relative positions can be obtained from the internal calibration.

13.4.1 Epipolar geometry

For moving cameras, the situation is somewhat different. In this case, the camera is calibrated at the beginning; after that, we move the camera in the scene we intend to reconstruct. This method does not involve any calibration object, which can make our task much more manageable. This means that the displacement estimation can only be calculated from natural markers contained

in the image, which is a problematic task, namely, their position is not known - would they be known, the reconstruction would be ready. Thus we only can rely on finding the same marker in two images, which can be used to formulate an equation for the point pair.

For that we need to examine first the geometric arrangement, for the sake of simplicity, we constrain the discussion to the stereo camera arrangement, where the moving camera is displayed at two locations, which are called the left and right cameras. The center of the left camera is O_L; the image plane of which is intersected by the spatial point X at X_L. For the right camera, the center is O_R, the intersection of the image plane with X is X_R. The displacement between both camera centers is denoted by the t vector, while the rotation between the camera planes is the R matrix.

Note that if we know O_L and X_L but not X (as the case in reality is), then a direction can be determined where X lies from the left camera, but the distance cannot. Thus we have an infinite number of possibilities along a spatial line for X. However, the projection maps a spatial line into a line (if not perpendicular to the image plane) on the image plane, hence the pair of X_L lies along a line on the image plane of the right camera - these lines are called epipolar lines. All epipolar lines intersect in a point called the epipolar point (e_L and e_R); furthermore, this is the point where the line between the camera centers intersect with the image planes.

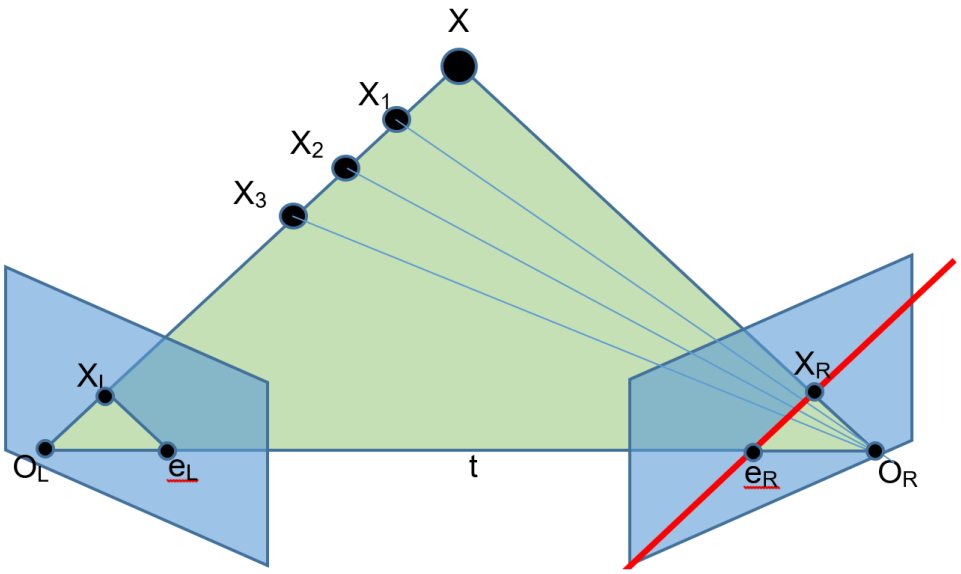

Figure 13.10: *The epipolar arrangement.*

The vector which connects the cameras (t), that one pointing from O_L to X_L and the third from O_R to X_R are coplanar - which is only true if X_L and X_R are the images of the same X. This means that selecting two of the three vectors and calculating the cross product of them; the result will be perpendicular to the third, which can be formulated as follows:

$$x_L^T(t \times R^T x_R) = x_L^T([T\times]R^T)x_R = x_L^T E x_R = 0 \tag{13.16}$$

Where $[T\times]$ is the matrix of the cross product with t and E the essential matrix . Note that the X_R vector was transformed into the coordinate system of the left camera with the rotation matrix; namely, all vectors should be in the same coordinate system to get a valid expression. Thus we get an equation, where we need a point pair between the cameras and the unknowns are the rotational and translational parameters. A small problem that the coordinates of X_L and X_R are not in pixels, but they are given as vectors in the camera coordinate system. The camera matrix gives the transformation between both :

$$x_L = A_L^{-1} u_L \tag{13.17}$$

This is substituted:

$$u_L^T A_L^{-T} E A_R^{-1} u_R = u_L^T F U_R^T = 0 \tag{13.18}$$

Where F is the fundamental matrix; if we have determined the camera matrices during internal calibration, E can be obtained from F. Nevertheless, after that, the rotational and translational parameters should be extracted from E, which is not trivial. During decomposition we have two variants for rotation and one for translation (but with unknown sign); thus we have four combinations, from which, fortunately, one can be selected with simple geometric constrains.

One problem remains, namely, we are not able to determine the magnitude of the displacement vector; thus, we have the direction of the translation and rotation of the camera, but the amount is what we cannot say. Nevertheless, we will be able to carry out the 3D reconstruction, but we will not have information about the scale of it.

13.4.2 Calibration methods

The next important step is determining F from measurements, for which we identify natural markers on both images, and they are matched with a feature descriptor (e.g., SIFT). Given enough pairs, we have a linear equation system, from which the parameters of F can be calculated with the LS-method. Two widely-used methods are the 8- and 7-point algorithms, which use 8 and 7 point pairs, respectively for calculating F.

The 7- and 8-point methods suffer from the same problem as internal calibration: the problem formulation results in an equation system that is not easy to solve. The unknowns are in F not in the vector at the right, but we can reformulate as follows:

$$Af = \begin{pmatrix} u_L^1 u_R^1 & u_L^1 v_R^1 & u_L^1 & v_L^1 u_R^1 & v_L^1 v_R^1 & v_L^1 & u_R^1 & v_R^1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_L^N u_R^N & u_L^N v_R^N & u_L^N & v_L^N u_R^N & v_L^N v_R^N & v_L^N & u_R^N & v_R^N & 1 \end{pmatrix} \begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix} = 0 \tag{13.19}$$

Where f contains the elements of F. In the case of the 8-point method, the matrix has 8 rows, one for each point pair. The problem with that is for satisfying the original equation, F should be singular, but it will not be with the introduced estimation procedure. As a solution, we determine the singular matrix which is nearest to F, for that, we only need to set the smallest singular value of F to 0.

The 7-point algorithm uses the singularity constraint for the initial estimate; thus, 7 point pairs suffice. Thus after the initial estimate, we get a 2D solution set but can use the singularity constraint to select the correct solution. Unfortunately, the calibration algorithms are rather sensitive to noise and scaling. Thus the coordinates of the points used are worth being specified after normalization (i.e., subtracting the mean and after that dividing with the standard deviation).

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

14 3D Reconstruction

14.1 Introduction

The previous lecture discussed the essentials of imaging geometry and calibration methods, which are crucial parts of 3D reconstruction (also true for any reconstruction system). This lecture goes into the details of the implementation of a given reconstruction and the detail of the required computer vision algorithms.

3D reconstruction can be divided into four steps, one of which was discussed previously. This first step is the calibration of each camera and the camera system. After that, the images used for reconstruction are transformed with the rectification transform (obtained as a result of the calibration of the camera system). Then we proceed with solving the correspondence problem, which is followed by the calculation of the spatial points with the help of the calibration result and the point pairs (obtained from the correspondence problem). The reconstruction of 3D space can be followed by several different processing steps; thus, the goal of reconstruction is spatial processing. The algorithms used for that will only be discussed in the following subchapter.

14.2 Rectification

After determining the transformation between the cameras, we have a new opportunity. Recall the epipolar lines introduced for the stereo arrangement, and they specify the line along which the pair of a point lies in the image of the other camera (of course, they differ for each point). There is a distinguished camera arrangement, where all epipolar lines are horizontal, called stereo arrangement - i.e. when the displacement between cameras is only horizontal, and the image planes are parallel.

The significant advantage of this arrangement is that the pairs of each image point should only be searched for in a row, resulting in an acceleration of 2-3 orders of magnitude. In practice, only industrial devices make it possible to construct such an arrangement - but the calibration process can be used to artificially create it, which is called rectification. This means that both images get distorted with a linear transformation in a way that epipolar lines should be horizontal. As lens distortions make epipolar lines curved, this also should be corrected during rectification. Thus both steps are merged, and in the literature often this is also referred to as rectification.

14.3 Disparity

Identifying the correspondence between the acquired images is an essential task of reconstruction. Namely, the information that spatial points are at different locations in images taken from different positions is the one needed for reconstructing the loss of projection. Thus the accuracy of matching the points to their corresponding pairs determines the quality of the final reconstruction. The requirements are to find the pairs as accurate as possible and to find the pair for as many points as possible - the latter is unfortunately not possible entirely due to the finite dimensions of the image and partial coverage.

Rectification helps a lot to make the correspondence problem more manageable, as the pairs will be in the same row in the other image. Thus a disparity value can be assigned to each pixel of one image, which gives how many pixels away its pair is - i.e., the disparity is the horizontal distance

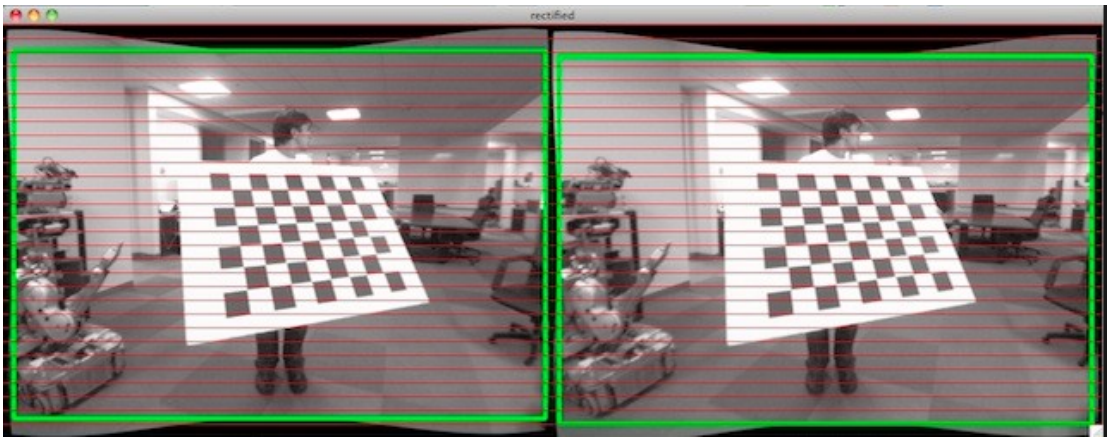

Figure 14.1: *Rectification.*

between a pixel and its pair. Sometimes cameras in the stereo arrangement are above each other; thus the disparity is vertical.

The disparity image contains the disparity values for each pixel and can be represented as a grayscale image, which can be interpreted by the human eye, as the disparity value depends on the distance from the camera. As you may have tried it at least once, as you observe your stretched hand with one eye closed, switching to the other, you will perceive as your hand would be at another location. The closer our hand, the more significant the difference - thus the bigger the disparity, the closer the object.

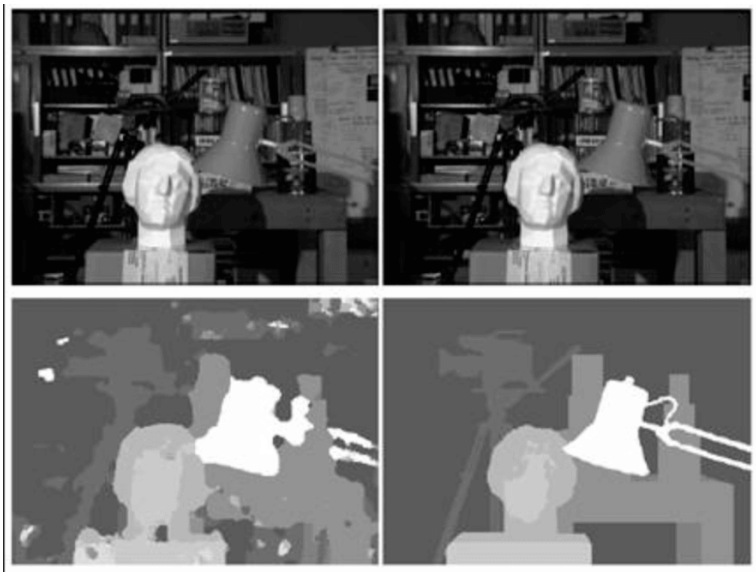

Figure 14.2: *Stereo image pair (above) and the corresponding disparity images (below). The left is the result of an algorithm, the right is the ground truth.*

Disparity can be determined with the help of several methods, which have two big groups: sparse and dense. The latter calculates disparity for about each pixel, while the former only in some distinguished positions. Although our goal is to calculate the disparity as densely as possible, sparse methods often deliver much more accurate results - which can be made denser with interpolation techniques.

14.3.1 Block Matching

Among dense disparity methods, an important family of algorithms is based on the BM (Block Matching) algorithm, which is an essential algorithm of computer vision; thus, it has several applications, and sometimes a slightly different name. It works by comparing the neighborhood of a pixel to each neighborhood of the row of the other image - the measure of similarity can be different, but most often, MSE is used. BM assigns the position where the loss is minimal as a pair to the pixel.

BM determines disparity values independently; thus the result will be quite noisy - in reality, it should be smooth with rare jumps between adjacent pixels, because 3D objects are continuous.

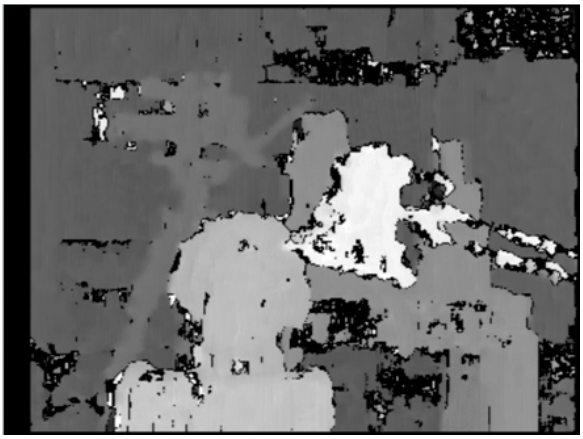

Figure 14.3: *The result of the BM algorithm.*

14.3.2 SGBM

Considering the properties of the real world, we can improve the result significantly, for which the loss function (MSE) is often extended with a penalty term prescribing the smoothness of disparity - this is incorporated into SGBM (Semi-Global Block Matching). The two additional factors in the loss function are as follows: the first investigates the given neighborhood of a pixel, and if the disparity difference is exactly 1, then the loss is incremented with P1 (a constant), if it is greater, we add P2 (another constant). Although the constant values can be chosen arbitrarily, a rule of thumb is to choose P2 as four times P1.

$$E(D) = \sum_{p} E(p, D_p) + \sum_{q \in N(p)} P1T(|D_q - D_p| == 1) + \sum_{q \in N(p)} P2T(|D_q - D_p| > 1) \tag{14.1}$$

14.3.3 Belief propagation

Another method for calculating disparity is called BP (Belief Propagation), which interprets each disparity value as a random variable, for each of which the disparity probability density function is known, the maximum of which gives the belief of that pixel. After that, each pixel sends a message to all of its neighbors, containing the probability density of the neighbors' disparities. As all pixels think that their neighbors have a similar disparity, we get a normal distribution, where the expected value is the belief of the pixel. After receiving messages from all neighbors, a new density is created based on the previous and received ones, the new belief will equal the maximum position of that. This is repeated until convergence (to reach consensus), which is proven for disparity images. The initial density can be generated with the help of the result of BM because it calculates an error for each disparity; thus initial probabilities can be inversely proportional to the errors.

BP introduces other variables besides the disparity, which denote the disparity jumps caused by hiding and object borders. BP has a much better quality result than SGBM, the price for which is a higher computational load, thus it is worth accelerating BP with graphics hardware.

Figure 14.4: *The result of Belief Propagation.*

In the following, such methods are mentioned, which can be used for generating a disparity image, but they are more general methods. Such an algorithm is optical flow; the variants were discussed in detail in previous lectures. As in the case of disparity, optical flow also has its dense and sparse variants; however, it is capable of determining general displacement. If it is not possible to rectify the images then the optical flow is worth trying, or even the local feature descriptor methods can be used for matching; nevertheless, those sparse methods are most often used for determining point pairs for the calibration process.

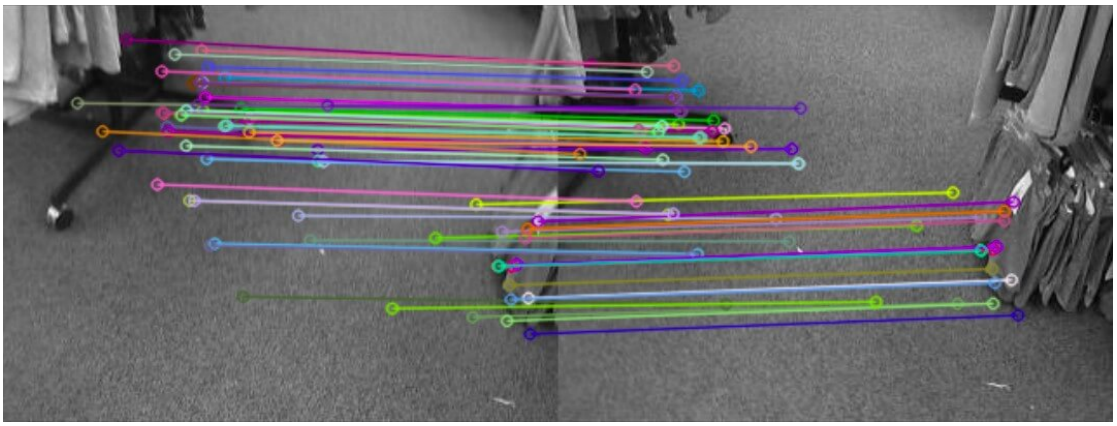

Figure 14.5: *Matching SIFT image features.*

14.4 Reconstruction

The last step of 3D reconstruction is called reprojection, during which the image points are projected back into 3D - for which the principle of triangulation is used.

14.4.1 Triangulation

In the calibration step, we determined the camera parameters; thus, we are able to construct the ray for each pixel along which light has reached the sensor. This is also true for all cameras where we have found a pair for the pixel. Furthermore, we can transform all of them - cameras and rays

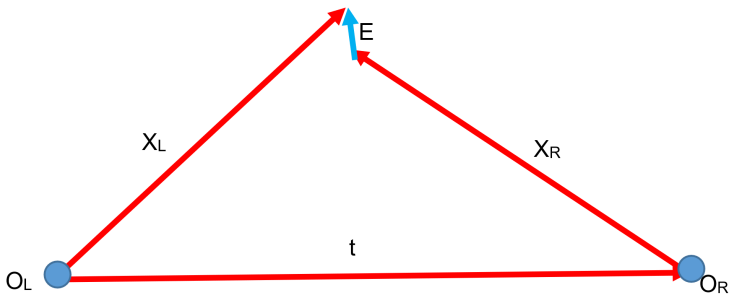

Figure 14.6: *The principle of triangulation.*

- into a common coordinate system, as we know the transformation between the cameras, where the rays corresponding to a point will intersect each other in the position of the real point.

The method discussed above is the base principle of triangulation, which works rather well for 2D problems, but in 3D, the problem is that even small inaccuracies can result in no intersection; thus, the coordinates of a 3D point should be calculated with the LS-method. First, we begin with the projection equation, and then we reorder to get the u coordinate of the image point:

$$\begin{pmatrix} wu \\ wv \\ w \end{pmatrix} = \begin{pmatrix} p_{11}^T \\ p_{12}^T \\ p_{13}^T \end{pmatrix} \vec{X} \quad \rightarrow \quad u = \frac{wu}{w} = \frac{p_{11}^T \vec{X}}{p_{13}^T \vec{X}} \tag{14.2}$$

In this equation u, v, and P are known, but the elements of X are the unknown spatial coordinates. Expressing v in the same way and merging them into an equation system we get the following:

$$\begin{aligned} u p_{13}^T \vec{X} &= p_{11}^T \vec{X} \\ v p_{13}^T \vec{X} &= p_{12}^T \vec{X} \end{aligned} \tag{14.3}$$

Reformulated with matrix notation:

$$\begin{pmatrix} p_{11}^T - u_1 p_{13}^T \\ p_{12}^T - v_1 p_{13}^T \end{pmatrix} \vec{X} = \vec{0} \tag{14.4}$$

, In this case, a known matrix is multiplied from left with a known vector; thus, the formulation is good for us. The problem is that X is 4D (homogeneous coordinates), but we only have two equations - fortunately, we have another camera; thus, we can have two more equations because X has an image also on the second sensor. Thus we can use the LS-method to get the solution. This method can be extended to systems with more cameras; there we would have two more equations for each camera.

Several function libraries including computer vision methods also have a reprojection method suitable for standard stereo arrangements, namely in the case of rectified image pairs, the LS estimation is unnecessary, the spatial coordinates can be calculated with simple matrix multiplication. While calculating rectification, a Q matrix can also be determined, which is then used to get the original spatial coordinates in the form of

$$\vec{X} = Q \begin{pmatrix} u \\ v \\ d \end{pmatrix} \tag{14.5}$$

Where u and v denote the position of the X point in an image and d is the disparity.

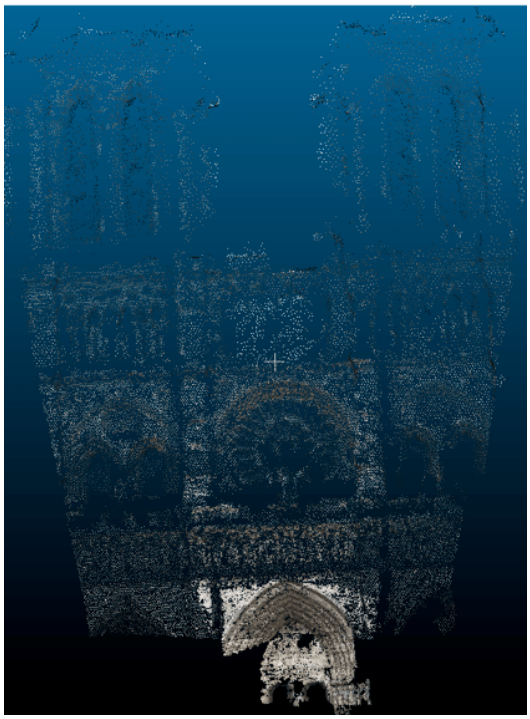

Figure 14.7: *The reconstruction of the Notre Dame.*

14.4.2 Metric reconstruction

Let us recall the original motivation of 3D computer vision, which was to reconstruct the original geometry of 3D space; this is required as camera projection introduces distortions, on the other hand, information was lost as well. Nonetheless, we should be aware of the limitations and operating conditions of these methods. A trivial limitation is that the simple two-camera (stereo) reconstruction will never give a full 3D space because the cameras are near each other; thus they only see the front side of the objects - thus in the reconstruction, the back side of the objects will be missing.

Another critical aspect is the metrics of the reconstruction. The main goal is that the reconstruction should have a known (preferably SI) unit, i.e., it should be metric. Thus we will discuss shortly the prerequisites of metric reconstruction and the consequences of not fulfilling them perfectly. The requirement is that both the intrinsic and extrinsic parameters of the camera system should be fully known.

It can occur - typically in the case of moving cameras - that the extrinsic parameters are only incompletely known (i.e., the displacement between two images cannot be determined). In this case, the reconstruction will be shape-true (i.e., no geometric distortions), but the size of the objects will be unknown. In this case, the unit of the resulting coordinates is the displacement between both cameras. If there is any way to estimate the magnitude of displacement or the size of a known spatial object (e.g., with speedometer, RGB-D sensor, etc.), then the reconstruction can be made metric. Nevertheless, a non-metric reconstruction can also be useful, e.g., in the case of vehicles it is enough to know the distance to other vehicles as a function of the velocity of the vehicle to avoid collisions.

If neither the intrinsic nor the extrinsic parameters are known, then we can carry out the reconstruction process, but the resulting 3D scene will be invariant up to an unknown projective transformation. As seen in the introductory part of the chapter, projective transforms can modify the parallelism of lines or bring a point from infinite distance to a finite one, thus using such transforms can even be dangerous under the constraint of unknown intrinsic and extrinsic parameters.

14.5 One and multicamera methods

In the first subchapter, we discussed the principle and the deficiencies of stereo reconstruction. For the two-camera reconstruction does not cover the full scene, it is often called the ”2.5D” solution. Also, in that case, we have used the minimum number of measurements (i.e., 2) to estimate the spatial coordinates - thus, this approach has the worst SNR, which can be improved with more measurements. Hence, multicamera reconstruction is discussed shortly.

Beforehand we got to know almost everything which is needed for multicamera reconstruction. If we use multiple, fixed cameras that encircle the scene from each direction, then a complete reconstruction is possible. In this case, reprojection can be done with the LS-method with high accuracy. This approach is most often used for creating animations and movies - or in medical imaging for implementing motion capture technologies. The latter means that the movements of a human or animal are captured with high speed in 3D to use this data for different purposes (creating animated characters, diagnostics).

14.5.1 SfM és SLAM

Somewhat more interesting is a reconstruction using only one moving camera. Although it is hardly a multicamera reconstruction, in this case, the reconstruction is made from several images, so this method belongs here. Two widely-used used names in the literature are SfM (Structure from Motion) and SLAM (Simultaneous Localization and Mapping) - the borderline is not exact, but the majority of the literature mentions SfM as an offline and SLAM as a real-time, online method.

Offline solutions look for correspondences among the acquired images (not necessarily in a sequential manner), the reconstruction is started from the most frequent points or from the image pair with the most correspondences - for which robust image feature detection algorithms (SIFT, SURF, etc.) are used.

Figure 14.8: *The reconstruction of the Colosseum based on tourist images.*

In the case of online methods, images arrive as they are taken (i.e., sequentially), each image adds its contribution to the existing reconstruction, while the new position of the camera is estimated - which is generally made w.r.t. the previous image and state. The real-time requirement implies that matching is calculated with the optical flow or other - similarly fast - method. The steps of a typical SLAM algorithm are the following:

1. Feature detection
2. Matching with the features of the previous image
3. Camera pose estimation
4. Estimation of 3D point coordinates

5. Bundle Adjustment

We highlight separately the step of Bundle Adjustment, which occurs for both on- and offline methods. This is somewhat similar to the last enhancement step of internal camera calibration, namely here we also use a numeric optimization algorithm to minimize the geometric error. The difference is that here not the intrinsic camera parameters but the coordinates of the 3D points and the elements of the matrix describing the camera position are refined.

The main problem of SLAM-like algorithms is the drift, which originates from the fact that the position of the camera is estimated w.r.t. the previous position, thus it consists of the sum of relative displacements. For perfect estimation is impossible, the error of each step accumulates causing that the at the beginning entirely accurate estimate moves away from the ground truth.

To solve this problem, we have several algorithms at hand, e.g., we can estimate the displacement not only w.r.t. the previous but also w.r.t. earlier steps, in that way, drift can be decreased. This approach cannot be continued forever because as the camera sees new details of the scene, the new images will not have any correspondence with the old ones. Another solution is called loop closure detection, which detects if the camera revisits a place already visited, and it estimates the position w.r.t. the image taken previously at the same place (or in the near neighborhood). It is generally possible to use the map beside previous images to estimate the position.

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[34] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, Feb. 2008. DOI: 10.1109/tpami.2007.1166. [Online]. Available: https://doi.org/10.1109%2Ftpami.2007.1166.

[35] J. Sun, N.-N. Zheng, and H.-Y. Shum, "Stereo matching using belief propagation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 7, pp. 787–800, Jul. 2003. DOI: 10.1109/tpami.2003.1206509. [Online]. Available: https://doi.org/10.1109%2Ftpami.2003.1206509.

[36] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual SLAM algorithms: A survey from 2010 to 2016," *IPSJ Transactions on Computer Vision and Applications*, vol. 9, no. 1, Jun. 2017. DOI: 10.1186/s41074-017-0027-2. [Online]. Available: https://doi.org/10.1186%2Fs41074-017-0027-2.

[37] C. Wu, "Towards linear-time incremental structure from motion," in *2013 International Conference on 3D Vision*, IEEE, Jun. 2013. DOI: 10.1109/3dv.2013.25. [Online]. Available: https://doi.org/10.1109%2F3dv.2013.25.

15 3D Processing

15.1 Introduction

In previous chapters, 3D reconstruction methods were discussed in detail, but our task is not finished yet, namely reconstruction was aimed to extract data which can be processed further. Thus, these methods should also be discussed; they have some similarities to algorithms described previously, but there are also crucial differences.

The primary tasks of processing 3D structures are the same as in the case of computer vision, the data structure is noisy and erroneous; thus, we need to filter it. After that, we need to separate or identify the relevant part of space, which can be done with segmentation or local image feature-based detection methods.

15.2 The representation of 3D structure

First of all, an essential question should be cleared, namely the form of representation of the spatial structure is not obvious. Traditional computer vision stores 2D images on an also 2D grid, where the elements are the pixels, thus an obvious choice would be a 3D grid with voxels (composed from the words volumetric and pixel), where a small cube is assigned to each voxel, the value of which depends on the object in it.

The solution described above is a bad idea from multiple aspects, the first of which is the curse of dimensionality, as an image of average resolution has 1-4000 pixels in each dimension; thus, it consists of a few million pixels (a few MBs in memory). As a spatial structure has one more dimension, the number of voxels would be a few billions, i.e., GBs of memory would be needed to store them, given the same resolution - not to mention that a reconstruction is composed of multiple images because one image contains only one part of the scene, i.e., a higher resolution is needed.

Furthermore, the problem of the voxel grid is being wasteful, namely, in contrast to images where each pixel gets some light shed onto it, 3D scenes are quite empty. As we look around us, 90-95% of the volume is empty unless you are in a warehouse, of course. On the other hand, reconstructions only include the surface of objects (cameras do not see into them); thus only about 1% of the volume of an average reconstruction is used.

Based on the above, the primary data structure for 3D is the point cloud, which is the (generally unordered) list of 3D object points - often, other information besides position is stored too. The most frequent choice is to store the color of the pixel corresponding to that point. The normal vector of the surfaces, the points are part of, is also often stored.

There are some higher-level description methods than point clouds for 3D objects. Structures can be described with primitives (sphere, plane, cone, cylinder, etc.) or parametric surfaces/curves; thus, a potentially significant set of points can be substituted with a few numbers. An also widely-used solution is to use triangle meshes, which approximate complex surfaces with triangles - this approach is used first of all in computer graphics.

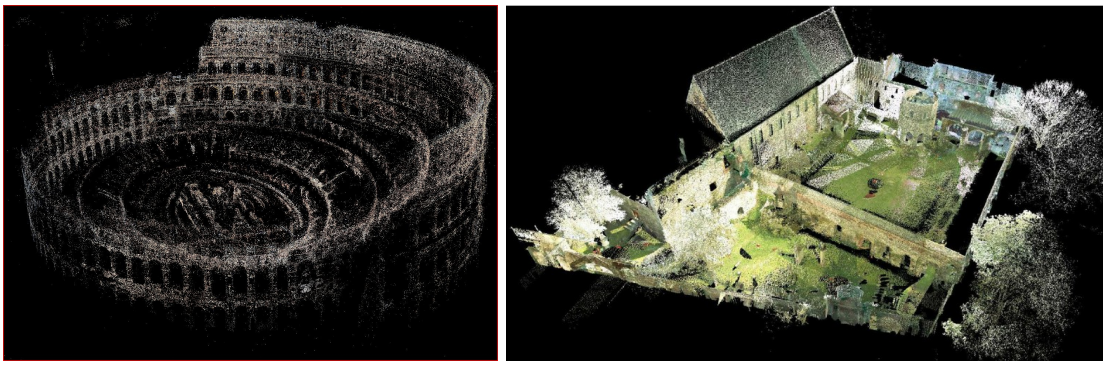

Figure 15.1: *Binary (left) and color (right) cloud points.*

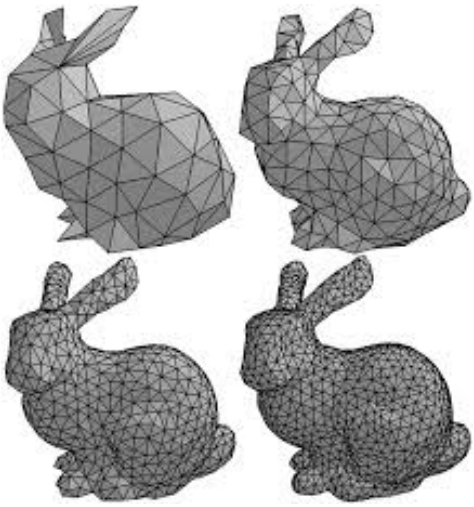

Figure 15.2: *Representing a surface with a mesh (different element sizes used).*

15.3 Filtering methods

As in computer vision, in point cloud processing filtering and enhancement is an important step. For the latter, the simplest solution is the so-called box or voxel filter. It works similarly to the averaging filter for images, but in this case, the window is 3D, which is used to select a local neighborhood for averaging. The box filter is perfect for filtering out position noise and undersampling. Namely, it can occur that due to erroneous matching the same spatial point is included more times with slightly different coordinates - in that case, new images would increase the size of the point cloud towards infinity, which can be avoided with the use of a box filter.

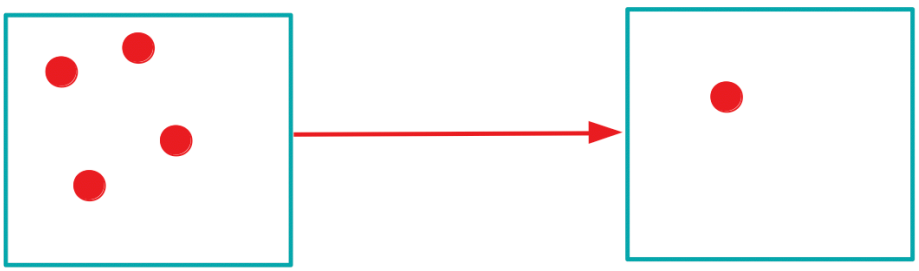

Figure 15.3: *The principle of the box filter.*

There are erroneous points in the point cloud, which can not be corrected by the box filter. Namely, during reconstruction, it occurs often that e.g., due to incorrect matching, some points are totally wrong and far away from the whole point cloud. These points are called outliers, and

they cannot be averaged with adjacent voxels, because there are no adjacent voxels. Nevertheless, this property can be exploited to remove them: we should only search for the k nearest neighbors of each voxel. Calculating the average distance from them, we can identify outliers if this distance deviates significantly from the others.

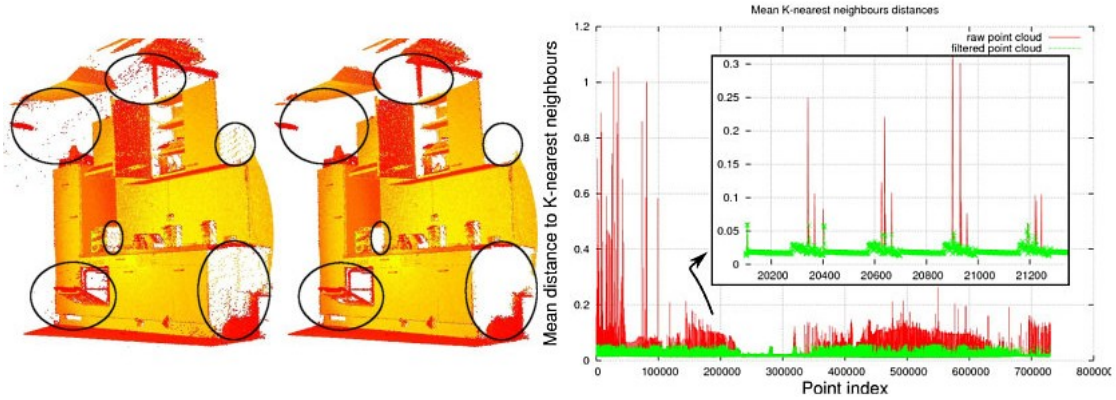

Figure 15.4: *Outliers (left) and the histogram used for filtering them out (right).*

15.3.1 K-d tree representation

For both filtering methods we need near or nearest neighbors - an obvious task for images (adjacent pixels are directly next to the pixel) but not for point clouds as they are not necessarily ordered logically, thus all N points in the cloud should be investigated.

The K-d tree structure tries to solve this problem, which inserts the points of the point cloud into a graph, which consists of a root, and each node has two children - thus, it is a binary tree. In the beginning, we choose a point as root, and then the point cloud is split into two with a plane along an arbitrary dimension (the plane should include the root point). After that, from each sub-point cloud, the nearest points (w.r.t. the plane) are determined and assigned as children of the root. Then the above operations are repeated so that the plane we use should always be perpendicular to that of the parent.

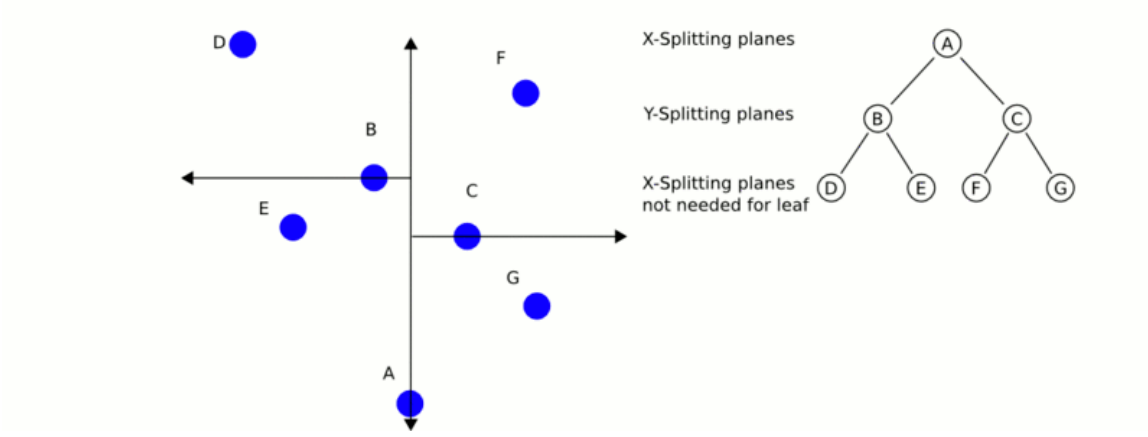

Figure 15.5: *The rule for constructing the k-d tree.*

The main advantage of the k-d tree structure is that after constructing it (which is done only once), the nearest neighbors can be determined as in a binary tree; thus, half of the points can be neglected in each search step. As a result, the search will be proportional instead of N with its base-two logarithm; hence an immense speedup is experienced, first of all for point clouds with millions of points.

15.4 Segmentation

Segmentation is an essential task of point cloud processing, it is used to separate objects from each other or from irrelevant background objects - it is rather common to remove the floor/ground which is in most cases a background object which connects all relevant objects.

In lecture 4, we discussed several segmentation methods, among others, the region-based ones like region growing and slicing, and also clustering-based methods as k-means and MoG or graph-cut based ones. They work almost exactly in the same way for point clouds; the only slight difference is instead of using similarity measures of the pixels, point clouds use measures for spatial membership.

15.4.1 RANSAC

There is an algorithm that is rarely used for images, but for point clouds, it is considered as one of the most popular algorithms. It is called RANSAC (Random Sample Consensus), which is a universal parameter estimation method, which is used among others for camera calibration beside point cloud segmentation.

The base principle is rather simple: randomly selecting some points from a point set, it constructs a candidate for the solution - this step is repeated to get a high number of candidates. After that, it is investigated how many points fit each of them from the whole point set. The points fitting to the candidates are called inliers, which are summed for each candidate - the result will be the candidate with the most inliers.

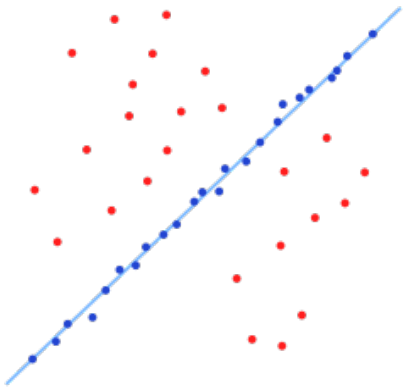

Figure 15.6: *Inlier and outlier points in the case of line fitting.*

In the case of point clouds, RANSAC is typically used to find primitive parametric shapes (planes, cylinders, etc.), for which RANSAC selects some points from the cloud, determines the primitive which fits these points. After constructing several candidates, it is analyzed how much points fit the primitives. Note that because RANSAC is looking for parametric shapes, hence they can be found in a scale- and rotation-invariant manner.

The method has several variants, which are generally different regarding the selection of the random points and the evaluation - in the case of point clouds, the entirely random sampling can be changed to a locally random one. Thus, relatively near points are used to construct the candidates; namely, geometric shapes are local phenomena, i.e., near points are more probable to belong to the same shape. During the evaluation, it is also reasonable to stay within a local neighborhood, based on similar considerations. Generally speaking, sampling and evaluation should be customized for each task.

As RANSAC is a general parameter estimation algorithm, it is a rival of LS-methods - the main advantage of the former is its significant insensitivity to outliers. Namely, LS handles outliers well until the ratio of them is below 50%, above that the estimate will be wrong. The drawback of

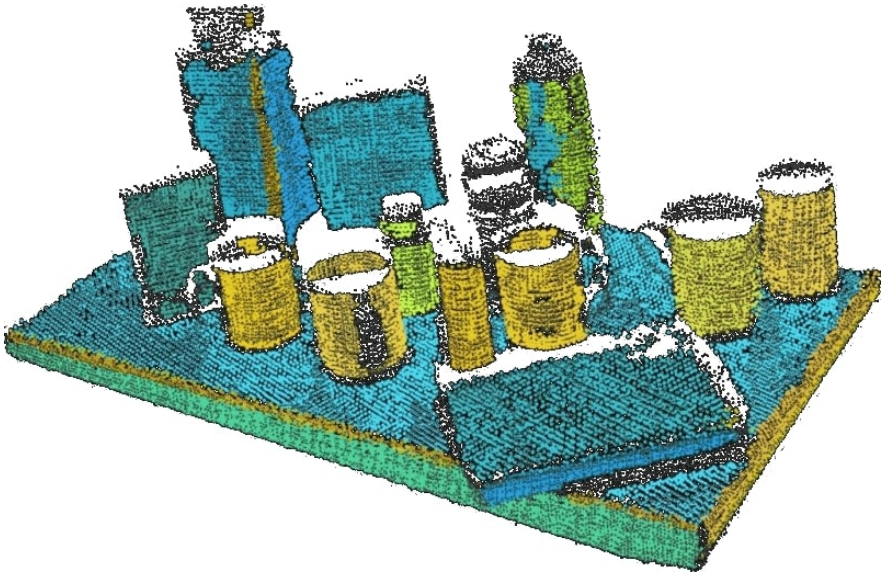

Figure 15.7: *RANSAC shape detection on an image cloud. This model tried to segment planes and cylinders.*

RANSAC is its stochastic nature; thus it cannot be guaranteed that it will find the correct result - to increase the probability, we need more candidates, which results in slower operation (RANSAC is generally not real-time).

15.5 Object recognition

The recognition and detection of objects and classes are one of the most essential tasks of computer vision; thus, it is worth donating a whole chapter to object recognition in point clouds - which is rather similar to that in the case of images. As a first step, a reference object is selected, then its features suitable for recognition are calculated. After that, those features are also calculated for the objects of the point cloud, and then features get matched. The last step is to estimate the transformation between reference and detected objects.

The most critical step is determining the features; thus, these methods are discussed in detail. There are two essential types of point cloud features: local ones describing points and global ones that describe whole objects or segments. Features should be - as in the 2D case - unique to make object detection easier and robust to different transforms.

15.5.1 Local descriptors

One of the most widely-used local features is in the case of point clouds the surface normal, which can be calculated for almost every point and describes the direction of the surface formed by the points near the one being investigated. It is calculated in a rather simple way, namely a surface is a local 2D point set, i.e., the neighbors of the point investigated deviate in two directions, but not in the third one, which is perpendicular to both of them. Thus, determining the covariance matrix of the point set investigated, the eigenvector corresponding to the smallest eigenvalue gives the direction of the least deviation. Note that actually, we get two directions (the eigenvector and the vector in the opposite direction) - we will use the convention that the normal vector should point into the direction of the viewpoint.

Calculating normal vectors is a simple task, but the result will be not unique enough to match points based solely on that - nevertheless, they can be used to construct more complicated features, as it is done in the PFH (Point Feature Histogram) method. This method identifies for each point

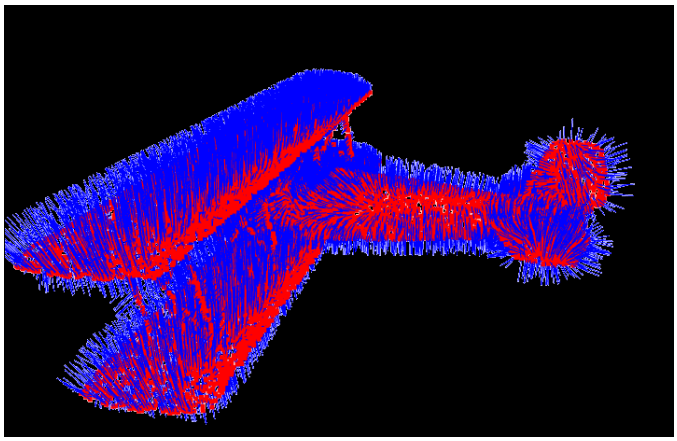

Figure 15.8: *The normal vectors of a point cloud.*

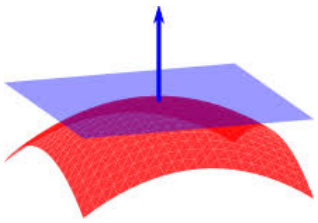

Figure 15.9: *A surface and the direction of the normal vector.*

being described its n nearest neighbors, after that, from this n+1 points all variants of point pairs are generated, for each of which some features are calculated (e.g. distance of the points , the angle between of their normal vectors, the angles between the line segment connecting the points and the normal vectors). After these features are calculated, histograms are created and used as descriptors - if we recall SIFT, we can conclude that it works in a similar way.

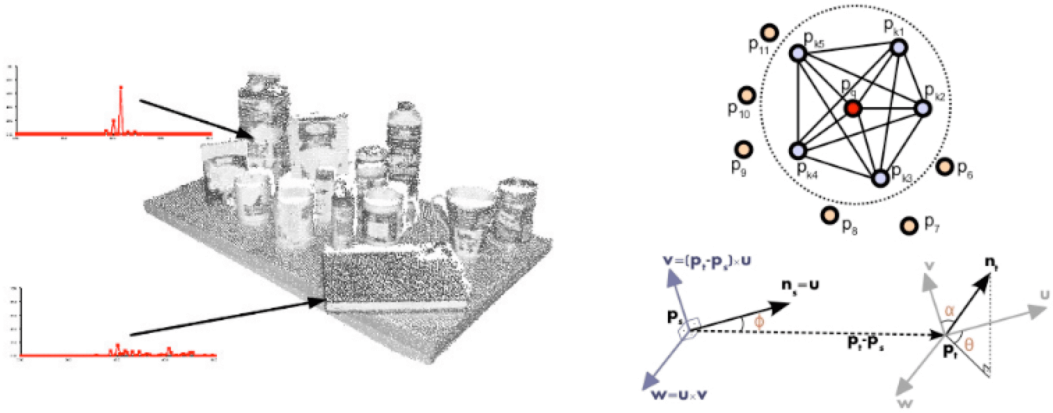

Figure 15.10: *Features between neighboring points used for PFH (right) and some example histograms (left).*

15.5.2 Global descriptors

An example for global descriptors is the GASD (Globally Aligned Spatial Distribution), which calculates the eigenvalues and eigenvectors of the covariance matrix of the object points. After that, the point cloud is rotated so that the eigenvector corresponding to the most significant eigenvalue should point in the x-direction and the smallest in the z-direction. Thus, all objects are rotated in a consistent way, then the scene is split up into an MxMxM grid, and the objects in

each cell are counted. The result is converted into a histogram, which is used as a descriptor. Of course, there are several other methods for generating both local and global descriptors.

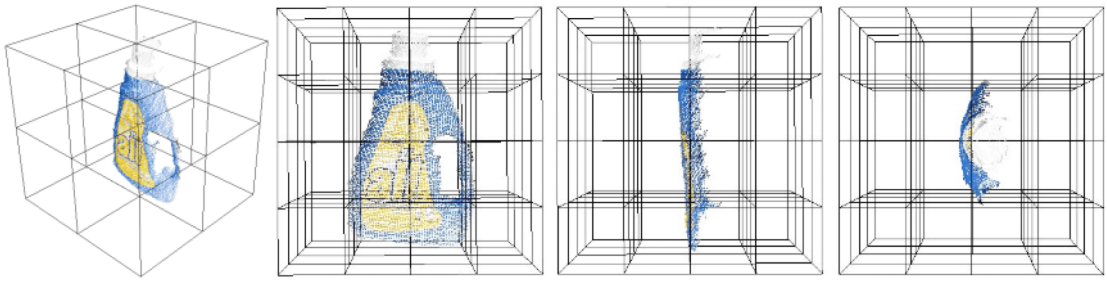

Figure 15.11: *The object being described globally (left) and its variants rotated based on the directions of deviation (right) .*

15.5.3 Registration

Besides object recognition and detection, local descriptors have another application, namely the registration task, during which we have two or more - partially overlapping - point cloud parts of the same scene. The goal is to merge these parts into one point cloud describing the whole scene. During registration, for the detection of overlapping parts, local features are often used.

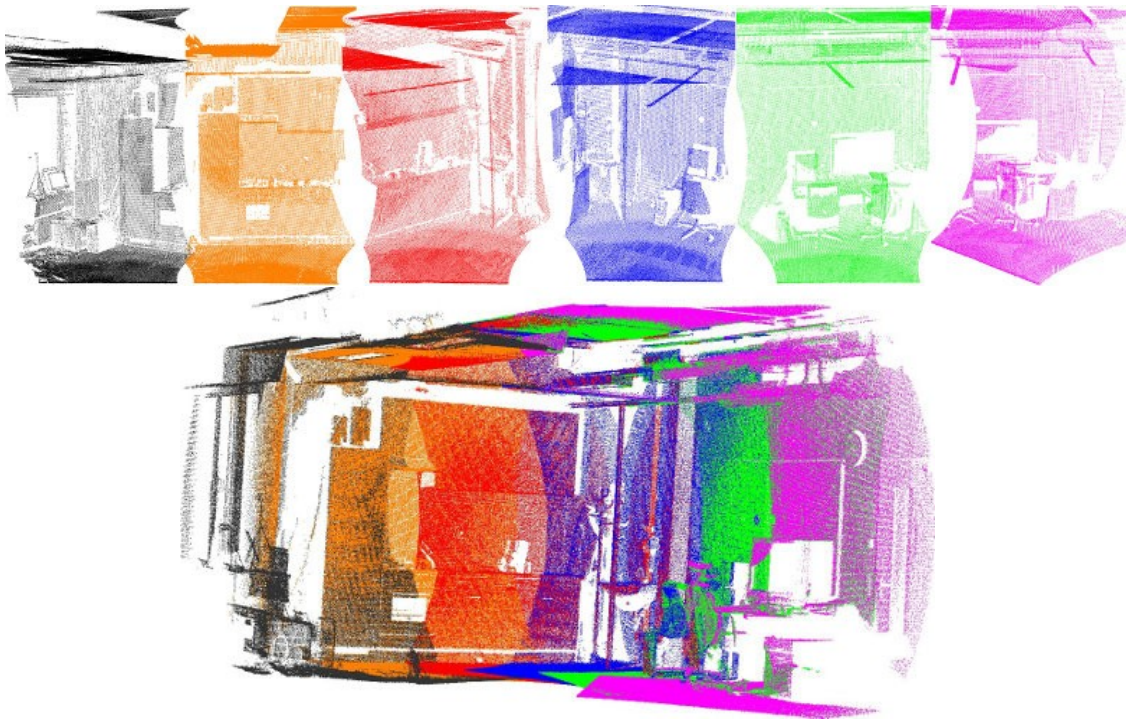

Figure 15.12: *Parts of a point cloud before (above) and after (after) registration .*

15.6 3D Deep Learning

Using the methods of deep learning has a few fundamental difficulties in case of 3D processing, which makes using deep learning in 3D considerably more complicated.

15.6.1 Voxel networks

It would be relatively straightforward to use 3D convolutional networks instead of 2D ones with a voxel grid as input data. However, as discussed earlier, using the voxel representation is highly problematic due to its wasteful use of memory. This is especially serious in the case of deep learning, since the memory requirements of these network is already high enough in the 2D case, and it is one of the major bottlenecks of modern GPUs. Despite this, there are voxel neural networks, although these work at a relatively small resolution, resulting in limited performance.

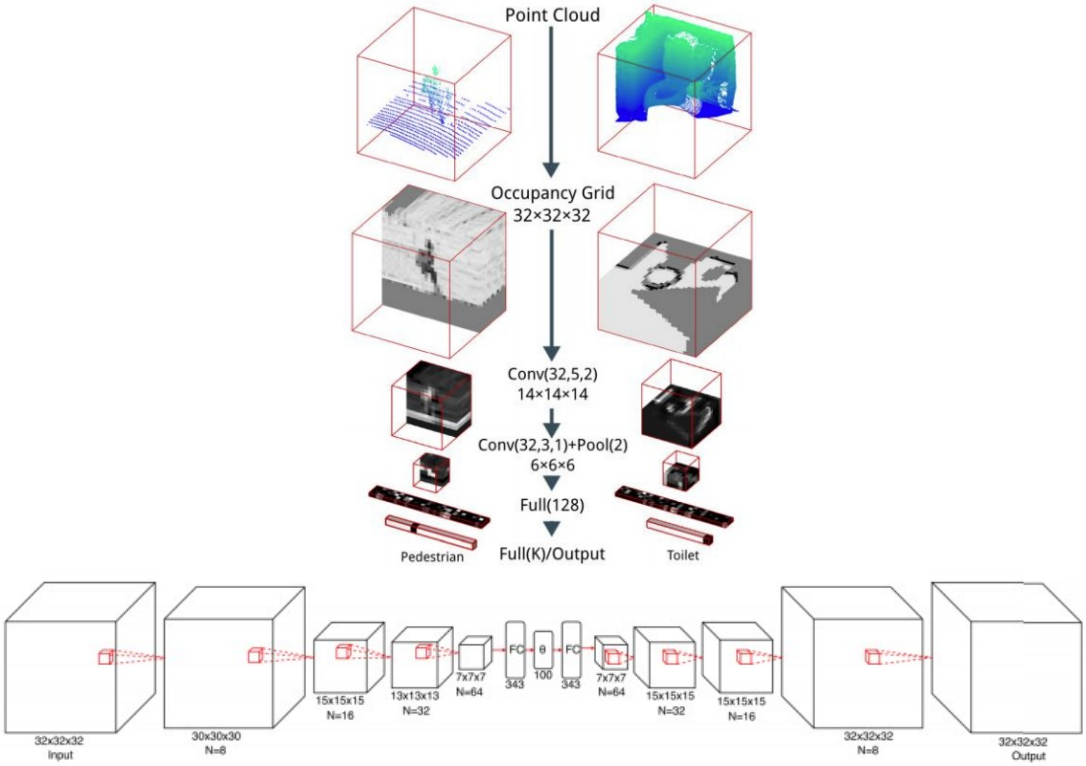

Figure 15.13: *A voxel-based classification (top) and a segmentation (bottom) neural network.*

15.6.2 Projection-based networks

There are also neural networks that attempt to precess a 3D scene by creating 2D projections from randomly generated viewpoints, and then processing these using standard 2D convolutional networks. This way, 3D classification can be reduced to the 2D case, however, this still results in considerably more computation, as more images need to be forwarded through the network. Moreover, these methods are difficult to extend to more complex problems, such as detection or segmentation.

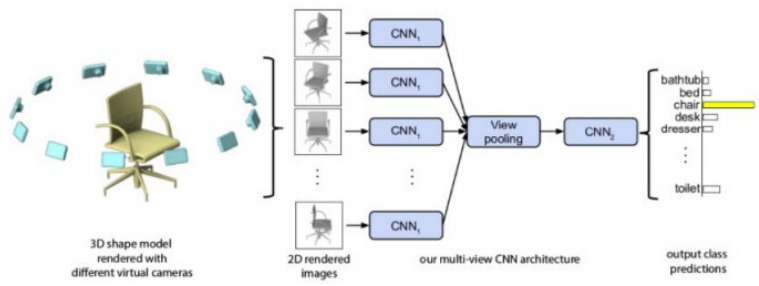

Figure 15.14: *A projection-based deep learning architecture.*

15.6.3 Point cloud networks

Nonetheless, if we attempt to use one of the more appropriate representation methods, we have to face the difficulty of adapting these to convolutional networks. In the case of point clouds we have to solve the problem of ordering invariance: Neural networks work well with ordered data, however, in point clouds, changing the order of the points does not change the 3D structure at all. Neural networks are not invariant to the ordering of its inputs, therefore they produce different outputs for different orderings.

A clever solution to this problem is to transform the points in the cloud onto a grid, and perform the convolution using this grid. This can be done by finding the nearest neighbor of every point in all directions, and consider these as the neighbors of the central point on the grid. This operation is called χ-Convolution. This method can be further modified by taking the distance of the neighbors into account as well, in addition to the features. This solution is employed by the popular PointCNN architecture.

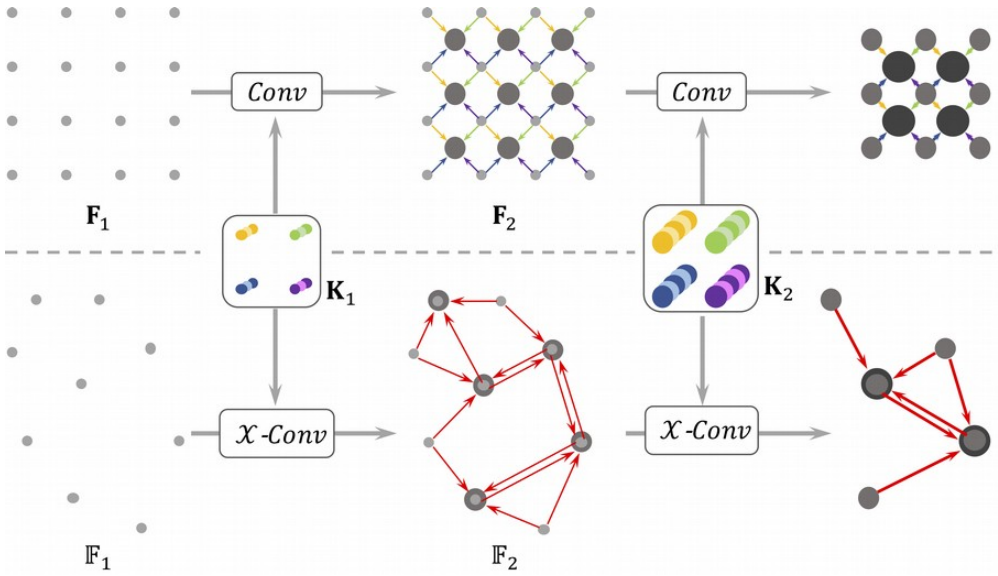

Figure 15.15: *Traditional convolution (top) and the χ-convolution (bottom).*

Further Reading

[1] R. Szeliski, *Computer Vision*. Springer London, 2011. DOI: 10.1007/978-1-84882-935-0. [Online]. Available: https://doi.org/10.1007%2F978-1-84882-935-0.

[38] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. DOI: 10.1145/361002.361007. [Online]. Available: https://doi.org/10.1145%2F361002.361007.

[39] R. Schnabel, R. Wahl, and R. Klein, "Efficient RANSAC for point-cloud shape detection," *Computer Graphics Forum*, vol. 26, no. 2, pp. 214–226, Jun. 2007. DOI: 10.1111/j.1467-8659.2007.01016.x. [Online]. Available: https://doi.org/10.1111%2Fj.1467-8659.2007.01016.x.

[40] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (FPFH) for 3d registration," in *2009 IEEE International Conference on Robotics and Automation*, IEEE, May 2009. DOI: 10.1109/robot.2009.5152473. [Online]. Available: https://doi.org/10.1109%2Frobot.2009.5152473.

[41] J. P. S. do Monte Lima and V. Teichrieb, "An efficient global point cloud descriptor for object recognition and pose estimation," in *2016 29th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, IEEE, Oct. 2016. DOI: 10.1109/sibgrapi.2016.017. [Online]. Available: https://doi.org/10.1109%2Fsibgrapi.2016.017.

[42] Y. Zhou and O. Tuzel, "VoxelNet: End-to-end learning for point cloud based 3d object detection," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, IEEE, Jun. 2018. DOI: 10.1109/cvpr.2018.00472. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2018.00472.

[43] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Dec. 2015. DOI: 10.1109/iccv.2015.114. [Online]. Available: https://doi.org/10.1109%2Ficcv.2015.114.

[44] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, *Pointcnn: Convolution on x-transformed points*, 2018. eprint: 1801.07791. [Online]. Available: http://www.arxiv.org/abs/1801.07791.

Part IV

Real-time Vision

16 Hardware

16.1 Intro

As computer vision needs an enormous amount of data, which is processed in real time, thus discussing the possibilities for accelerating our algorithms is a reasonable decision. Although the performance of processors is increasing, years are needed for significant improvements, which is simply too long during the development phase. Hence, investigating the alternatives for algorithm speedup is essential. There are three main paradigms of acceleration:

- **Distributing the processing task for multiple processors:** this approach uses multiple processing units, hence it is possible to distribute data and tasks among them.
- **Data stream-based processing:** in this case we also have multiple processing units, but they are not independent, thus only data can be distributed among them (the tasks not).
- **Pipeline technique:** hardware using this approach splits the tasks into small subtasks, which are executed in a parallel manner on the data arriving in a sequential manner. Here the data that came in the previous step is one step further in the pipeline. Although using the pipeline technique does not make the processing of one data entry faster, the throughput of the whole processing can experience a speedup of up to one order of magnitude.

16.2 Devices

The majority of the hardware solutions found on the market implements not only one of the three but generally all of them - of course, up to a specific level. Multicore processors, computer networks and supercomputers opt for the first one; while DSPs (Digital Signal Processor), CNNs (Cellular Neural Network), GPUs (Graphics Processing Unit), TPUs (Tensor Processing Unit) use the data stream-based processing paradigm; FPGAs (Field Programmable Gate Array) and ASICs (Application Specific IC) focus on the pipeline approach.

This lecture focuses on GPUs, which are generally less flexible than general-purpose processors, namely several tasks are hardwired in them - nevertheless, the performance can be with orders of magnitude better; they are not as fast as FPGAs or ASICs, but the flexibility of GPUs is higher than that of them.

GPUs are always used in systems with CPUs (Central Processing Unit) - the control program is run on the CPU, which also controls the GPU. Generally, PCI-Express is used for communication between them, as it has quite a high bandwidth, nonetheless, moving data between GPU and CPU is considered as the main bottleneck of processing data on the GPU.

At first, it may not be obvious why to use GPUs for computer vision as they were designed mainly for computer graphics tasks. Computer vision can be thought as the inverse of computer graphics - conceptionally -, both of them are similar regarding the tasks and data structures they use, which states the suitability of GPUs for computer vision.

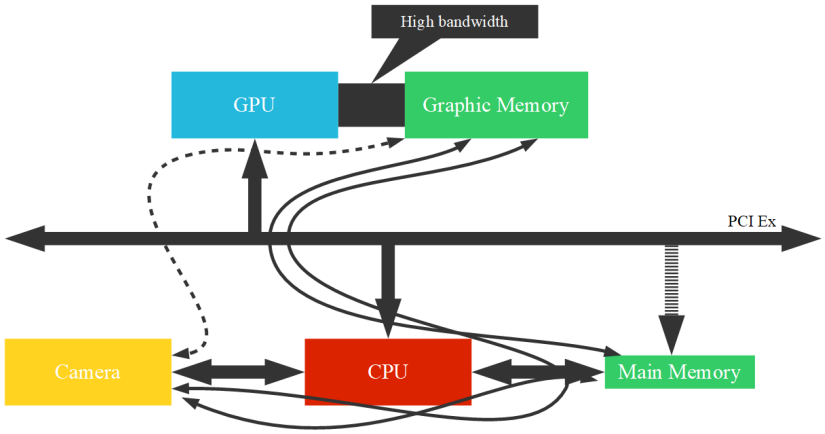

Figure 16.1: *System sketch with GPU.*

16.2.1 Data stream processing

Data processing with CPUs has three essential architecture types:

- **Single Instruction Single Data (SISD):** this is the simplest one, one processing unit processes all the data in a sequential manner.
- **Multiple Instruction Multiple Data (MIMD):** in this case, there are multiple independent processing units which process the data, a typical example is a multicore processor.
- **Single Instruction Multiple Data (SIMD):** here we also have multiple processing units, but they are not independent, because they have a common control unit, thus they carry out the same operation but on different data. The main advantage of this architecture is that much more processing units can be used at the same cost - not to mention the simpler and easier to manage programming paradigm they have compared to the MIMD case.

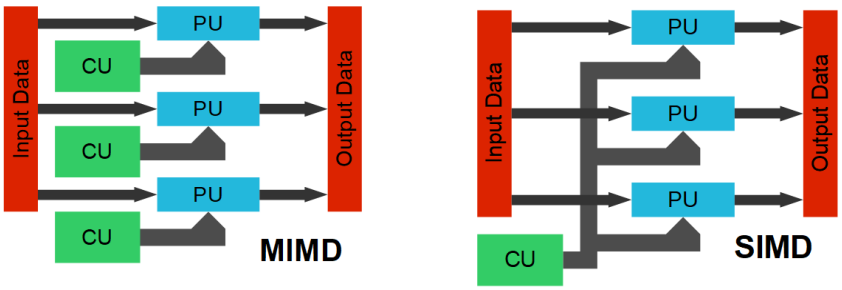

Figure 16.2: *MIMD (bal) and SIMD (jobb) architectures .*

The GPUs and TPUs (discussed later) typically have SIMD architecture; in the former case, data stream processing is supported by advanced memory management, high-bandwidth memories (4096 bits), and pipeline processing. Control units are typically less advanced (and thus, simpler), there is not as much speculation (branch prediction, etc.). An essential element of GPUs is that they have - besides the global memory -, own registers and an advanced cache system for the processing units. Note that modern CPUs also use this principle: SWAR (SIMD Within A Register) architecture makes it possible that a 64 bit ALU can e.g., operate on eight one-byte chars at once. GPUs also support the distribution of multiprocessor-like tasks between GPUs of identical architectures (this is called NVLink technology for NVIDIA).

Thus, data stream processing can be efficiently used if the same operation should be carried out on a large amount of data - for data processing of random nature, e.g., for servers or databases, it is not the best choice. An easy-to-implement operation is called Map when the same operation

should be carried out on the elements of an array. In the case of Reduction, the results of a Map are reduced into a number (typically with sum). Data filtering, sorting, and search are also common; exciting examples are the Gather-Scatter operations: in both cases, we read from (Gather)/write to (Scatter) to an array so that we read the indices of read/write from a smaller array.

16.3 GPU architecture

The general structure of GPUs consists of a PCI interface and a scheduler. Global memory can be accessed by multiple independent memory controllers, which have a common L2 cache. The processing units of a GPU are organized into clusters (GPC - Graphics Processing Cluster), which have their own rasterizing unit (relevant first of all for graphics). GPCs generally consists of 4-8 SMs (Streaming Multiprocessor), in which the processing units reside.

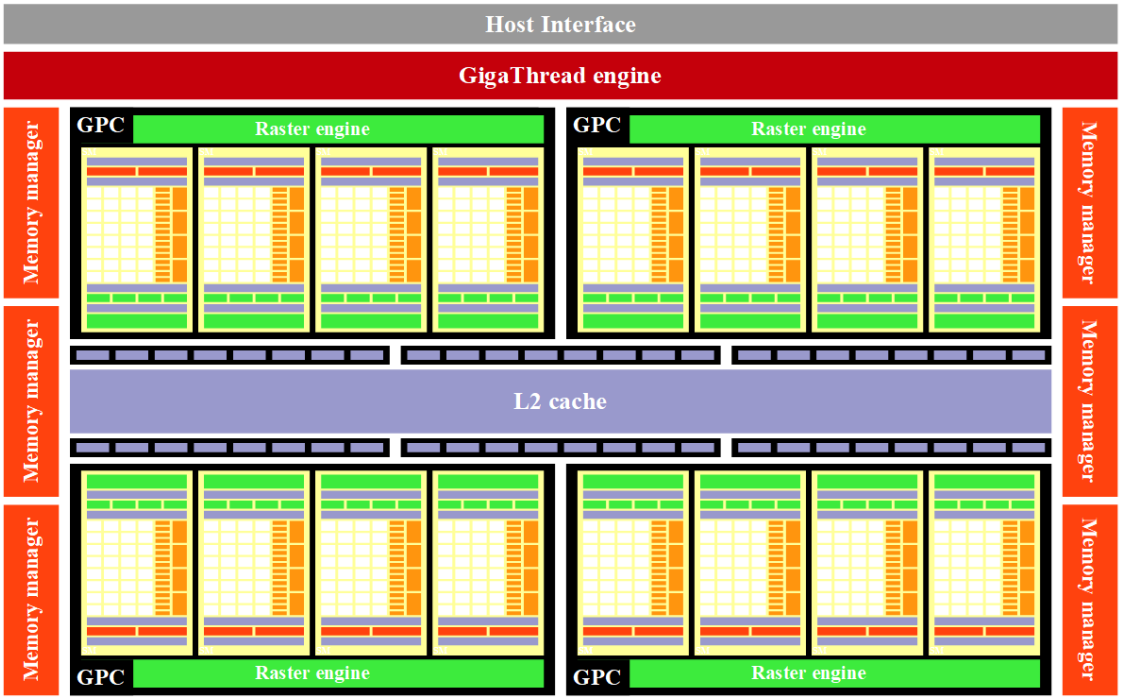

Figure 16.3: *The general structure of a GPU .*

16.3.1 Streaming Multiprocessor

The Streaming Multiprocessor is an essential unit in the GPU as they have a common control unit; an SM has - depending on the architecture - 32/64 cores, which have their registers. SMs have also SFUs (Special Function Unit), which can carry out mathematical operations (trigonometric functions, log, exp) in hardware; SMs also have dedicated memory read/write units (called Load-/Store units), which can access data in the global memory independently from data processing - they are needed because the response time of global memory can be hundreds of machine cycles due to the enormous amount of parallel data requests.

Shared memory is also essential because it can be accessed from all cores of the SM (registers are only available for one core). The SM also has L1 cache (separately for the instructions and for textures), and also texture processing units and dedicated graphics hardware elements (used mainly for computer graphics).

The structure of GPUs and SMs has changed a bit in recent years - for the former, it is mainly the higher number of SMs, but the latter has undergone essential changes regarding its internal architecture. The most important of which is that nowadays GPUs include different core types for

different data types - the majority of processing units works on 32-bit data (floats), but they can (using the SWAR principle) operate on two half-precision numbers (16 bits), which is exploited in the case of neural networks, where the resolution of the numbers is not so important. There are architectures with 64-bit units (for double), or even for integer (and fixed-point) arithmetic.

Figure 16.4: *The SMs of the Fermi (2010, left) and of the Volta (2017, right) architectures.*

16.3.2 Tensor Core

One of the most important advancements is the so-called TensorCore (found in GPUs of the Volta and Turing architectures) - they are dedicated units for matrix multiplication, each of which can carry out the following operation in hardware:

$$C = C + A * B \tag{16.1}$$

Where each operand is a 4x4 floating-point matrix; the advantage of the TensorCore is that we have a dedicated unit for matrix operations (as GPUs were designed for vector operations); thus neural networks (training and inference) experienced a speedup of one order of magnitude.

16.4 Programming model

In the last lecture we discussed the physical structure of GPUs and their runtime models, but their programming model was not mentioned, thus now we focus on the CUDA language.

GPU programming can be divided into two categories: into the first belong different graphic languages, while into the second the GPGPU (General Purpose GPU) languages. The most important from the former are OpenGL and Direct3D, which are mainly used for solutions developed

for graphic applications, but they can also be used for simple image processing tasks (filtering, thresholding, color space conversion). OpenGL is available for the majority of operating systems as a library; it uses the GLSL language, which is the extension of C++. Similar to which is the HLSL language used by Direct3D, which is developed by Microsoft; thus it is only available on Windows.

But for us, GPGPU programming languages are more important, i.e., CUDA and OpenCL, the former is developed by NVIDIA; hence it is only available on their GPUs. CUDA assumes a hardware based on homogeneous processing units, where each device carries out the same operation. It is based on the C/C++ languages , which are extended with some new language constructs. In the CUDA program, a kernel is written for one GPU core, and this is then executed on all cores. CUDA has its own compiler called nvcc.

OpenCL (Open Computing Language) is in contrast to CUDA, an open-source software, which works on all GPUs, theoretically, it works also on heterogeneous devices, but in practice, this is not used. Similar to CUDA it is designed as an extension to C/C++ with its own compiler (it is different for each GPU manufacturer). The philosophy of programming is the same as for CUDA. The second significant difference is that OpenCL programs can be compiled in runtime.

The CUDA language consists of multiple levels, the lowest level of which is the CUDA driver, which controls the computations on the GPU. For the driver, there is a programming interface (API), but it is a really low-level interface; hence it is used rarely. Most often, the CUDA runtime API and the functions built upon that are used. Note that using the driver or the runtime API are mutually exclusive choices.

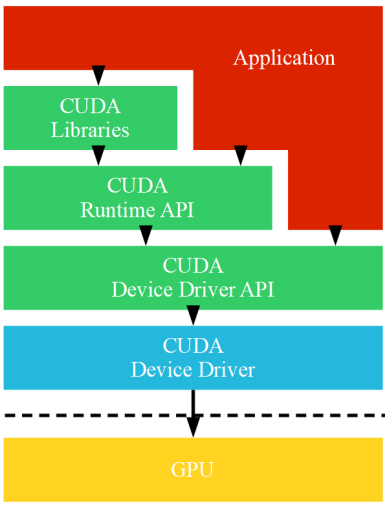

Figure 16.5: *The levels of CUDA.*

Before programming GPUs, we should investigate how GPU programs are executed - it is controlled by the CPU, and the data to be processed is also on the host machine. Thus, at first, the data should be copied to the GPU, then the processing can be started. As moving data and communication between CPU and GPU are expensive, it should be minimalized for the highest speed. As the GPU finishes the parallel program, data is copied back to the host.

16.4.1 Runtime model

The program running on the GPU is called kernel - for the majority of architectures, only one kernel can be launched. The base unit of a kernel is a thread, which is the program running on one core and one data entry; threads are organized into blocks, blocks into grids. The reason behind this two-level hierarchy is that threads in the same block are ensured to run on the same SM, but blocks in a grid are arbitrarily assigned to SMs. This is really important as a thread in a block sees the same shared memory (as it is physically in the SM), while different blocks can access different shared memories.

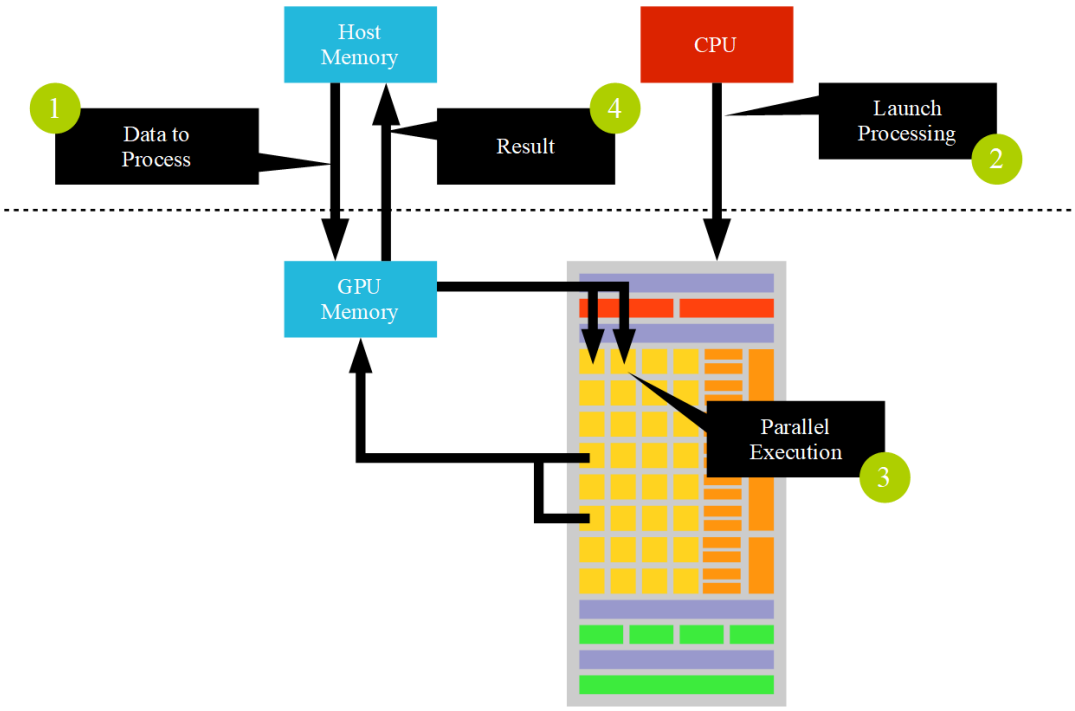

Figure 16.6: *The execution model of GPU programs.*

Another difference is that the size of blocks is finite: they can contain up to 1024 threads, while in a grid, the number of blocks is unlimited. Besides, threads can be organized within a block in one, two or three dimensions, while the grid can be maximum two-dimensional - this does not have physical significance, but organizing the program along multiple dimensions can make programming easier.

The last critical unit of the runtime model is the warp, which summarizes 32/64 threads running on the same SM. Of course, a block can consist of multiple (maximum 32) warps, but at once, maximum 32/64 thread can be running. This concept is important as if the code diverges within a warp (i.e., some threads should execute other instructions) then the rest of the threads should wait; thus divergent code (e.g., if-else) should be avoided within a warp.

Essential special variables are the kernel index variables, which include the serial number of the block the thread belongs to and the serial number of it within the threads of the block. With the help of these, it can be generally specified which element of the array to be processed should be processed by the given thread. For that, we also have variables for the size of the grid and the blocks.

```
dim3 gridDim; // grid dimension in blocks (max 2D)
dim3 blockDim; // block dimension in threads (max 3D)
dim3 blockIdx; // 32 block identifier within the grid
dim3 threadIdx; // 64 thread identifier within the block
dim3 warpSize; // 32 warp size (always 32)
```

A typical example for the use of the above variables is the following function, which sets all elements of an array to a constant value.

```
__global__  void assign( int* d_a, <spa n class="mi">int value )
{
     int idx = blockDim.x * blockIdx.x + <spa n class="s">threadIdx.x;
     d_a[ idx ] = value;
}
```

An SM has several registers; thus, context switches between warps can be made with zero overhead; actually, one SM can run multiple blocks. CUDA cores have pipelines like DSP units, i.e., they can run multiple warps so that into the stages of the pipeline not the consecutive steps of the same program but the steps of different threads are loaded.

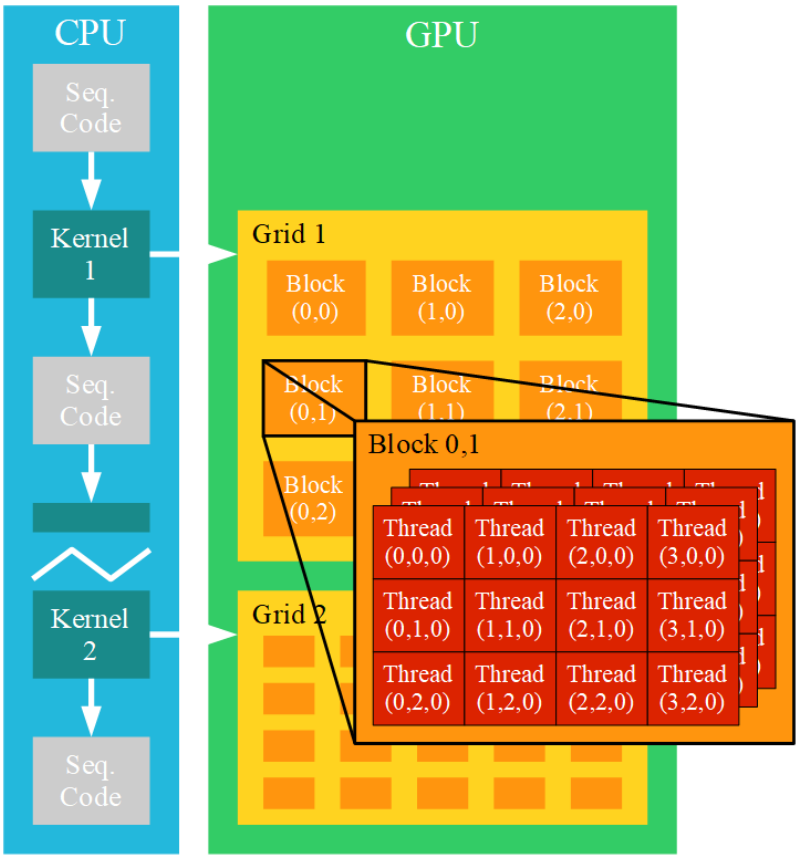

Figure 16.7: *The runtime model of CUDA.*

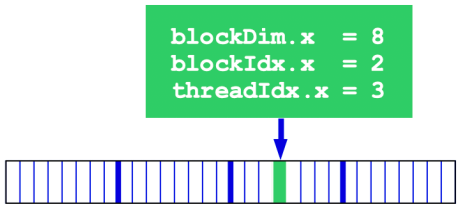

Figure 16.8: *Calculating the kernel index.*

16.4.2 Communication

It is also essential to discuss the inter-thread communication on the GPU, as it is essential for writing efficient programs - the first of them is the synchronization barrier within the block:

```
__shared__ int scratch[blocksize];
scratch[ threadIdx.x ] = value;
__syncthreads__()
leftValue = scratch[ threadIdx.x - 1 ]
}
```

This method is used if the threads within a block write some information into shared memory, which is relevant for all of them, and it is needed for the next operation. In this case, it is crucial to reach that point before letting any of the threads to proceed.

Atomic operations also can be used to implement inter-thread communication, as these operations are ensured to be carried out. A typical example is the implementation of the inner product, during which each thread computes a product then the result is added to the same variable stored in shared memory.

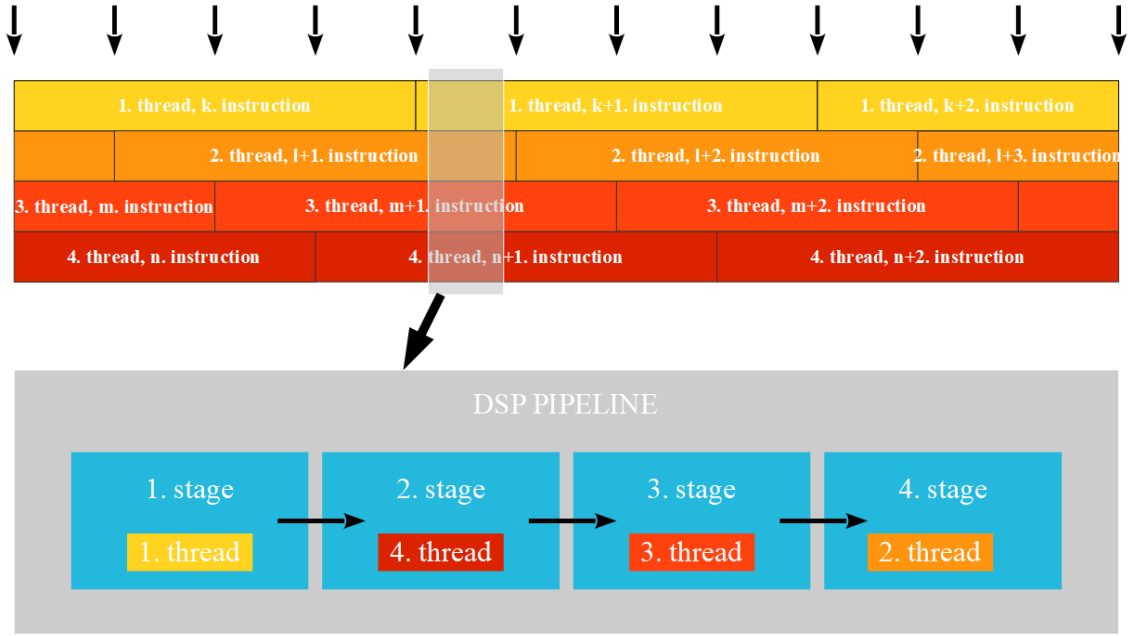

Figure 16.9: *Using pipelines for execution with multiple threads (multithreading).*

Note that the CPU-side program will proceed after the kernel launch in an asynchronous way, thus before using the result synchronization should be explicitly made in the following way:

```
1 cudaThreadSynchronize()
```

This function waits until all threads of the kernel running do not finish - it is automatically called before the new kernel launch, thus in those cases, it can be omitted.

16.4.3 Memory management

It is important to discuss the different memory types available on the GPU and their reasonable use. GPU memory is namely the most critical bottleneck of the processing; the several options we have can seem confusing. The following memory types exist on a GPU:

- **Registers:** private memory area for each thread, which reside physically in the SM (as SRAM, thus it is the fastest type we have), local variables of kernels are stored in them.
- **Local memory:** if registers run out, private data gets stored in local memory, which is located off-chip despite its name; thus it is far away physically; furthermore, it is DRAM, thus quite slow.
- **Shared memory:** it is also SRAM, which can be found in the SM, hence it is very fast. All cores see it in the block; thus, it is slower (first of all, writing) because more complicated access and cache solutions are in place. It is perfect for inter-core communication if it is enough to communicate within the block.
- **Global memory:** it is a separate, dynamic memory chip on the GPU - this is specified on the datasheet of the GPUs, its size is generally between 2-16 GBs. Global memory has a high bandwidth, despite this fact, its access can be very slow, if multiple thousand threads try to read it at once.
- **Constant memory:** physically, it is also off-ship dynamic memory, but it can only be read (in software); thus the very complicated write caching - needed due to using multiple threads - can be neglected, so it is slightly faster than global memory.

- **Texture memory:** a special type of constant memory, which has several useful features, one of which is that it enables us to use 1D, 2D or 3D indexing - the conversion of the indices is done in hardware. Besides that we can use normalized (i.e., between 0 and 1) indexing, the advantage of which is that we can set up that if the texture memory would index between indices, then it should weight the neighboring indices (convolutional filtering) - all of which is also done in hardware.

  Texture memory can be used as a 2D or 3D associative cache, which means that if a value is put into the cache, then its neighborhood is also read into the cache - which is not done based on memory addresses but on the 2D or 3D environment. Above that, we can index out from the texture without an error, for which we have multiple options (e.g., this operation can read a constant or repeating can also be set up).

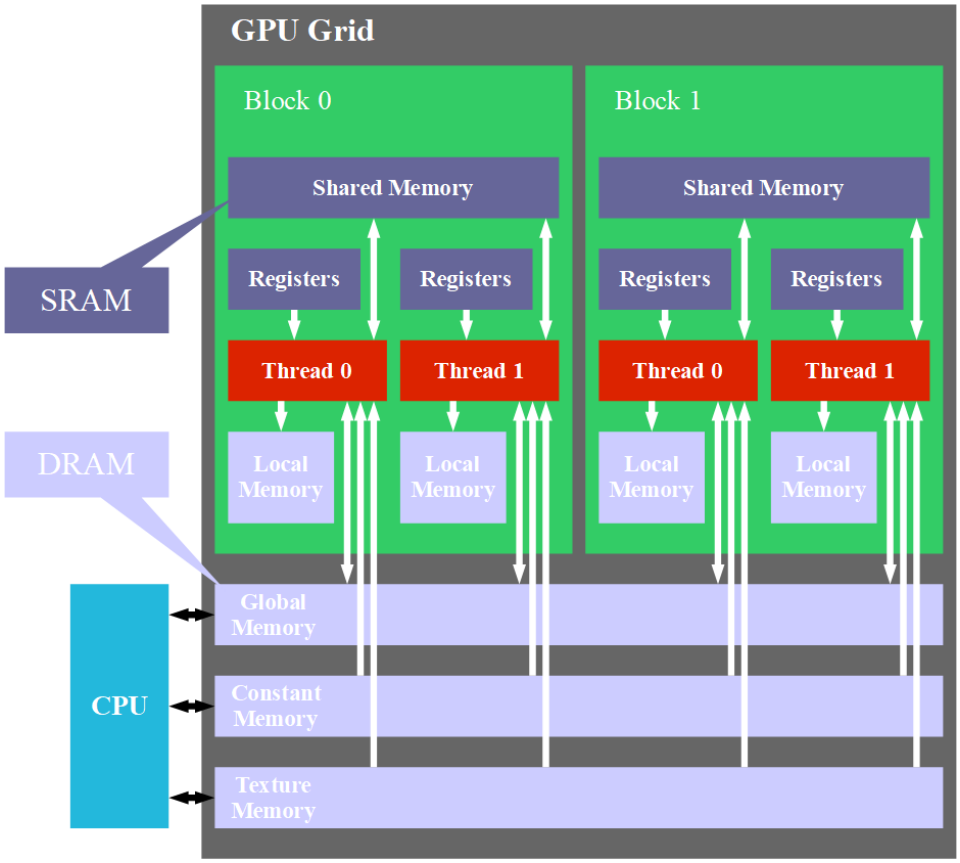

Figure 16.10: *The memory model of CUDA.*

We should also mention the case of writing common memory types at once, shared and global memory operate namely in different ways. While in global memory, each of the writes in the case of parallel writes will happen (their order is not guaranteed), in shared memory, only one write is guaranteed.

16.4.4 Using textures

The use of texture memory should be discussed separately as several settings/decisions should be made to use this. Note that texture memory should be defined from the CPU side, as from the GPU side, it behaves as constant memory. Texture memory can have the following settings:

- **Data type:** the data type of the element of the texture, it can be: 1D-4D vectors, integer or floating-point numbers.

- **Dimension:** the dimension of the texture (and the caching), it can be 1, 2 or 3.
- **Read mode:** specifies the type of indexing, it can be simple array indexing, normalized (between 0 and 1 within the boundaries, floating-point indices are used).
- **Interpolation:** we can set the type of hardware interpolation, which can be nearest neighbor (no filtering) or bilinear interpolation.
- **Boundary handling:** if we index out of the array, we can choose to get a constant (specified before) as a result, or the nearest element in the array. Another option is to set up the texture handler to assume, the texture is periodic outside the image boundaries; even a reflected periodic setting is available.

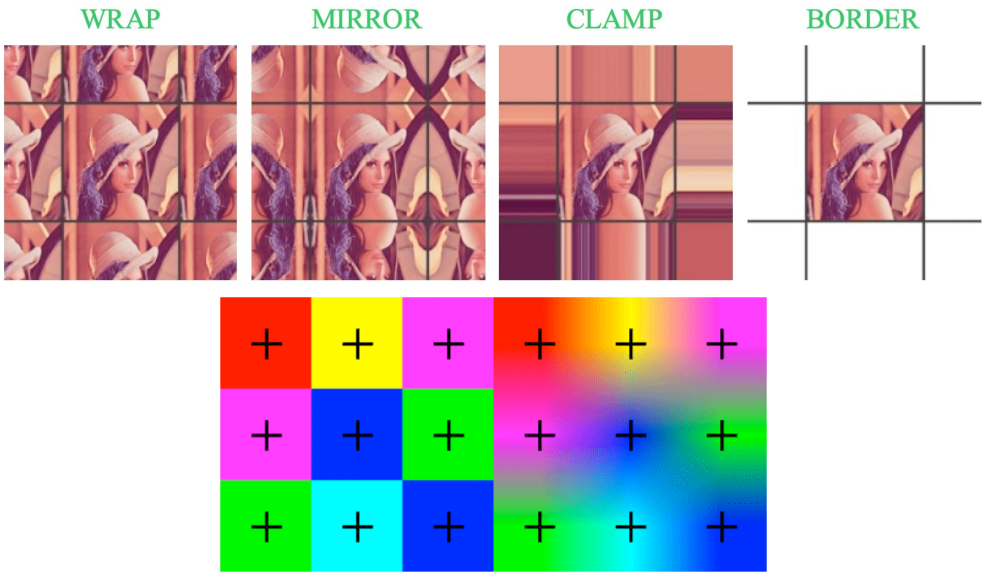

Figure 16.11: *Different boundary handling (above) and interpolation (below) options.*

Further Reading

[45] *CUDA Programming*. Elsevier, 2013. DOI: 10.1016/c2011-0-00029-7. [Online]. Available: https://doi.org/10.1016%2Fc2011-0-00029-7.

17 VPU, TPU, FPGA

17.1 Introduction

In previous lectures, we discussed the structure and programming paradigm of graphics processors; nonetheless, GPUs can be considered quite general-purpose devices, in which some instructions are hardwired. In this lecture, we get acquainted with TPUs (Tensor Processing Unit), designed for the acceleration of matrix operations, and other programmable hardware.

17.2 Vision Processing Unit

It is essential to mention Vision Processing Units (VPU), which are frequently used in computer vision settings. Generally, VPUs are low-consumption devices, equipped with a few RISC cores and several special parallel processing units. These parallel processors usually support low precision floating point (16/32 bit), and fixed point (8/16/32 bit) arithmetic and are further aided by special hardwired operation units. The parallel processors typically use the VLIW (Very Large Instruction Word) instruction set, which aids instruction-level parallelism.

Moreover, VPUs usually have local, on-chip memory units for efficient storage of intermediate values resulting from the computation. Finally, VPUs are also equipped with numerous interfaces for digital cameras. In conclusion, the structure of VPUs is remarkably similar to that of DSPs, except that the latter typically support high precision arithmetic.

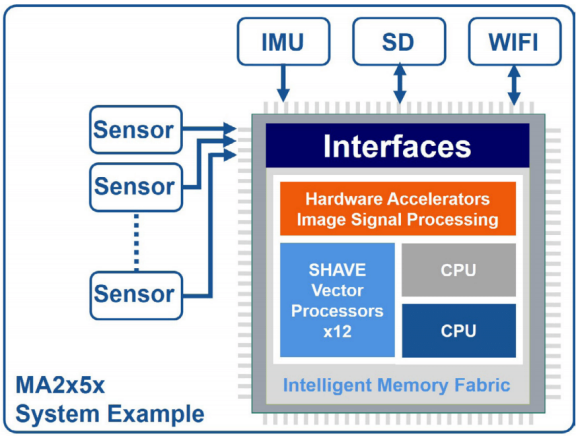

Figure 17.1: *The structure of a typical VPU.*

17.3 Tensor Processing Unit

It is essential to understand that GPUs are designed for vector operations (tensor cores are an exception), the property of which is that they only use their input once for one output. In the case of matrix operations, all inputs are used for multiple outputs; nonetheless, the majority of the output variables should be accumulated during operation execution. For that, in the case of GPU programming, we use shared memory and atomic operations, which slow down the kernel. As deep neural networks need an enormous amount of matrix multiplication (training and inference too), a dedicated hardware could come handy.

17.3.1 Systolic Array

For that, let us investigate matrix multiplication in the most straightforward, 2x2 case, the formula is as follows:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} * \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix} \tag{17.1}$$

This can be implemented with four multiplier and two adder units in the following way:

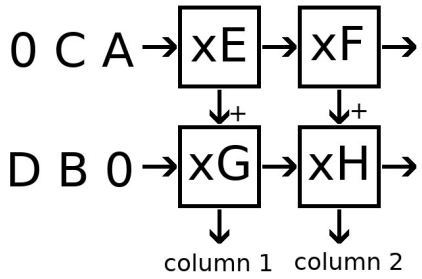

Figure 17.2: *Calculating the matrix product.*

Here multipliers multiply with a constant the data that sequentially comes as input; at the end of the runtime, the elements of the product matrix will be obtained as columns. Note that this method also provides some temporary results at the output, but after 4 cycles, all elements are calculated.

This architecture is called a systolic array, where systolic refers to the intensive usage of spatial parallelization and pipelining. Note that with this technique, matrix multiplication - which has a cubic cost - can be executed in linear time, which is an enormous speedup for huge matrices.

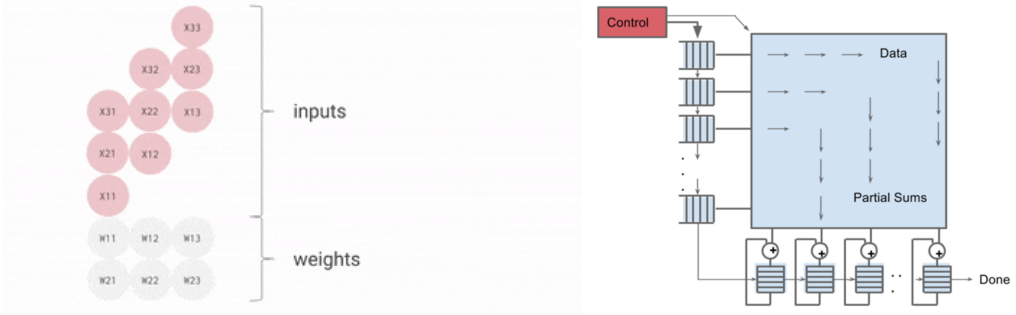

Figure 17.3: *Systolic matrix multiplication (left) and the structure of the TPU's matrix multiplier (right).*

17.3.2 Structure

The central element of the TPU structure is the matrix multiplier, which works according to the principle introduced above. It has a separate weight cache, which is only responsible for loading the weights of a neural network and passing them to the multiplier. The matrix multiplier is followed by activation units containing several activation functions in a hardwired manner and norm/pool units providing hardware acceleration for different downscaling and normalization methods.

It is also crucial that between the outputs of the normalization units and the unified buffer responsible for storing the input data/temporary activations, there is a high-bandwidth buffer. This buffer is connected to the matrix multiplier unit with a systolic preprocessing unit, which is responsible for organizing the data into the proper format. Note that input data is generally sent by

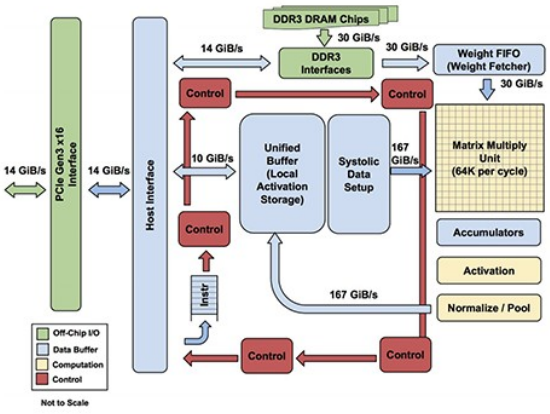

Figure 17.4: *The structure of the TPU.*

the host machine to the TPU, while the weights are locally stored - the reason for which is that the first generation TPU is designed first of all for inference.

The second generation of TPUs have a slightly modified architecture, they have a lot of HBM (High Bandwidth Memory) and the activation, normalization, and pooling units are substituted with general scalar/vector processing units - the reason for this is that second-generation TPUs can also be used for training.

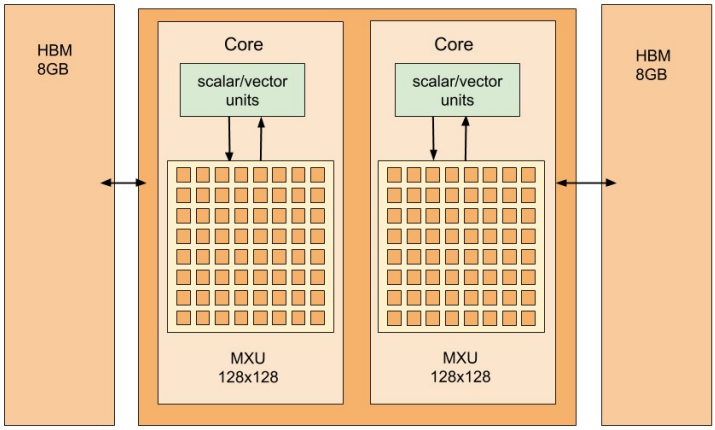

Figure 17.5: *The structure of a second generation TPU.*

17.4 Programmable hardware

GPUs and TPUs fit rather well Deep Learning tasks, but other special vision algorithms are hard to implement with them - first of all such algorithms, which have much conditional execution and branches. In this case, we can configure our task-specific device using programmable hardware.

There are several types of programmable hardware starting with PLAs (Programmable Logic Array) and PROMs (Programmable Read-Only Memory), which are suitable for combinational circuits. For sequential circuits, CPLDs (Complex Programmable Logic Device) can be used; while for complicated hardware or algorithms, most often FPGAs (Field Programmable Gate Array) are used.

FPGAs are special hardware devices, which use mostly pipelining and partially parallel processing for accelerating algorithms. FPGAs are also often used during hardware design, as developing new ICs is very expensive. In the case of ready and functional hardware design, it is also possible (and for mass-production, it is reasonable too) to fabricate it as an ASIC (Application Specific IC).

During FPGA development, the hardware is - at the beginning - designed at the block diagram level, which can be described with a hardware description language. The synthesizer for the FPGA is the program that turns code into real hardware configuration (which is specific for the device we use for the project). During the design process, we generally have the possibility to simulate the described hardware, where most of the errors will be shown. If the simulation is successful, then comes the hardware synthesis step, during which the synthesizer converts our description - using a hardware library - into real hardware.

Then comes the placement and routing stage, which needs the parameters of the device used. During this step will place the units into the blocks of the FPGA; here, it can also be determined what will be the highest speed of the hardware (which depends on the physical distance between the units). At the end of the placement step, we have the final configuration which can be downloaded to the hardware.

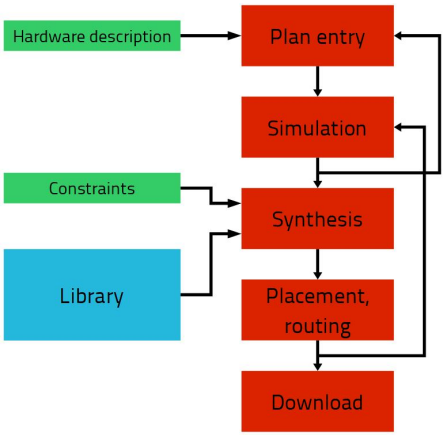

Figure 17.6: *The process of FPGA design.*

17.4.1 Architecture

At the beginning, it is important to understand how hardware can be made programmable: for each programmable element of an FPGA, there is a configuration flip-flop, which can store a one-bit value. They are serialized onto a data wire; the clock is provided by a dedicated clock generator for configuration purposes. During downloading the configuration, the flip-flops get the clock value, while the configuration values come serialized (in the sequence of the flip-flops) through the data wire. Thus, flip-flops behave like a large shift register, so each of them gets its configuration value. In the end, the clock is disabled, hence the flip-flops will - as constant memory - store their configuration value, which is used by different hardware elements as input. Below there are some examples for such hardware elements:

A whole FPGA chip has several configurable elements, which are organized into CLBs (Configurable Logic Block), between which there is a programmable interconnect matrix enabling an arbitrary connection between the CLBs. In FPGAs, there are also block RAM units (serving as on-chip memory) and IO blocks (to enable communication with the outer world).

A CLB consists of two slices, which is the base unit of configurable hardware; they are connected with the slices of adjacent CLBs (independently from the global interconnect matrix), thus the load of the interconnect system is significantly decreased - namely, connections of the CLBs are the bottleneck of FPGA development, so frequently-used connections should be implemented in a hardwired manner.

17.4.2 Slices

Each slice consists of multiple look-up tables, which can store one-bit values. Above that, they also have multiple flip-flops that can be configured arbitrarily, for which initial values, (a)synchronous

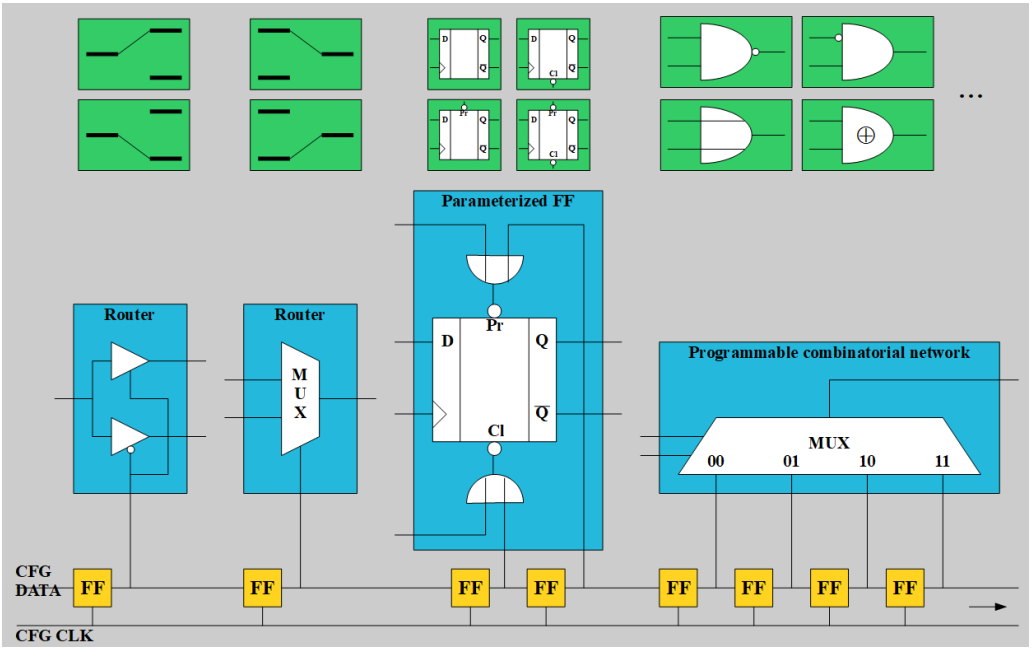

Figure 17.7: *How programmable hardware is constructed.*

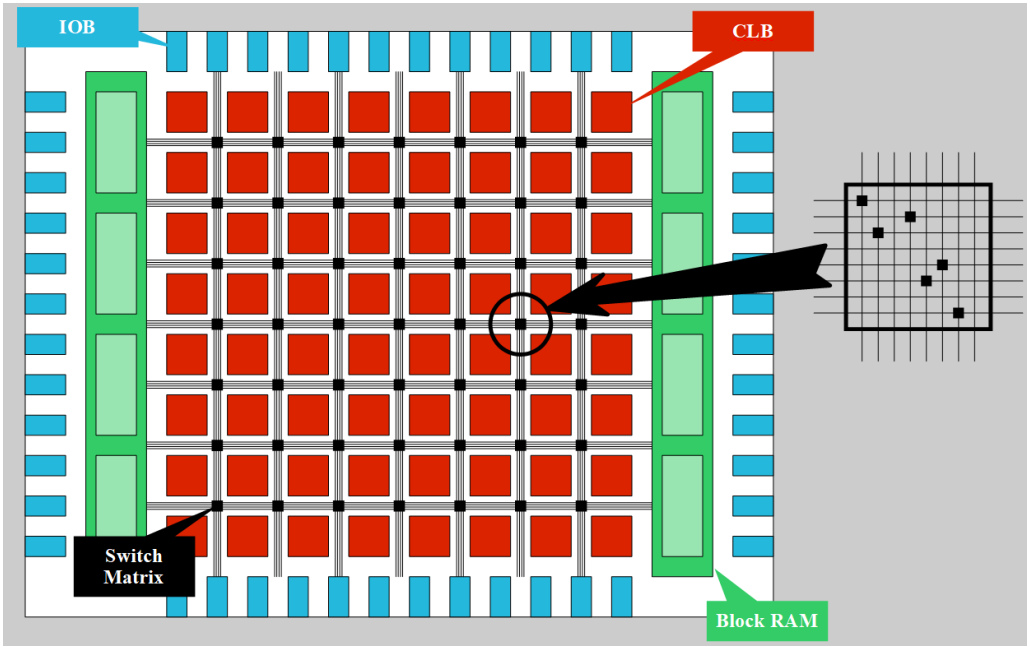

Figure 17.8: *The general structure of an FPGA.*

behavior, clock enable, etc. can be selected. Furthermore, each slice has a supplementary unit for arithmetic operations, e.g., the carry or the counter unit. Also, we have several multiplexers for route selection, supplementing combinational circuits or for other specialized tasks.

It is essential to discuss the operation of the LUT units, which are located within the slices because they are the core of the programmable slice. With the help of a LUT, we can realize simple logic functions (with 5-8 inputs) - for the extension, the multiplexers of the slice can be used. Of course, each slice has a finite number of LUTs; thus, fewer logic circuits with more inputs can fit into a slice. If we load the truth table into the memory elements of the LUT, then it will work as anticipated.

Also note that a LUT can be used as a RAM or (virtually) as a ROM element - they will operate

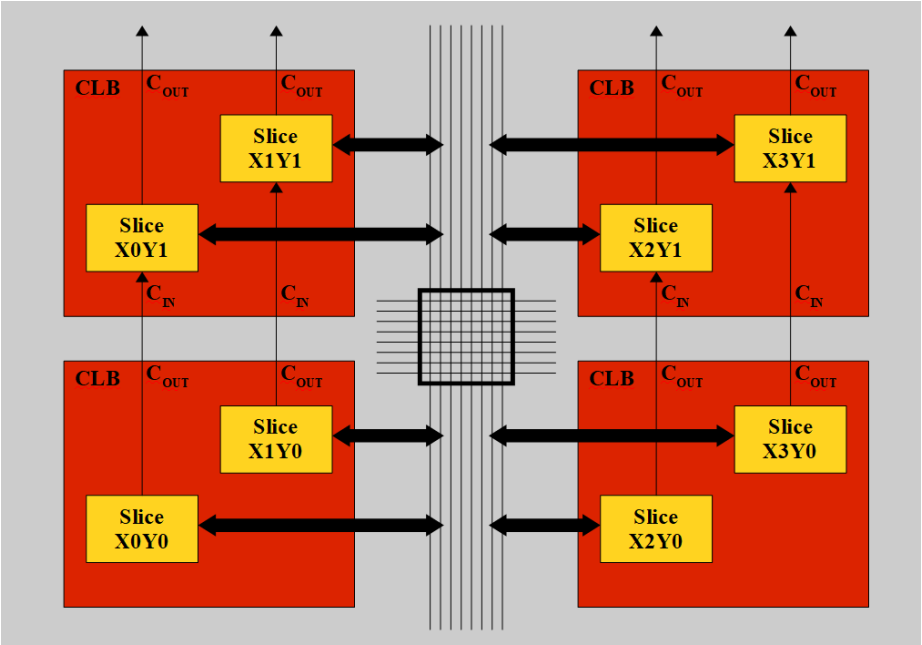

Figure 17.9: *The general structure of CLBs.*

as one-bit memory cells, but accessing them is much faster than accessing the block RAM. LUTs can be used to construct shift registers, which are often used.

Let us analyze some more examples for the usage of slices, the simplest of which is the implementation of combinational circuits: here 6 out of 8 inputs are connected to the inputs of the LUTs, the other 2 controls the multiplexers which select the LUTs:

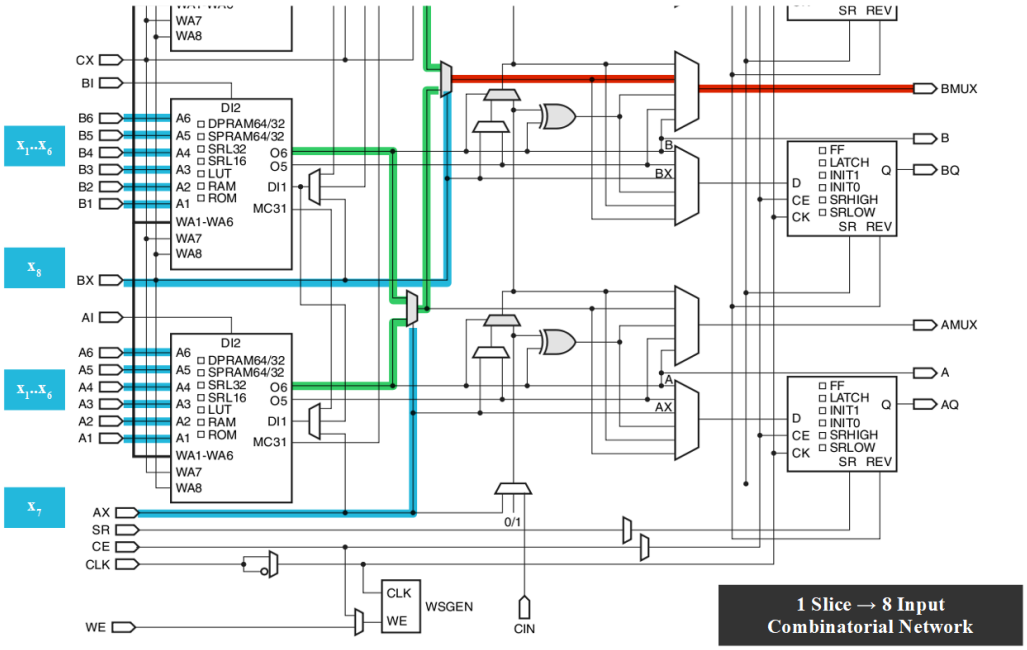

Figure 17.10: *Realizing an 8-input combinational circuit (remark: there are 2 more LUTs outside the image).*

A little more complicated is the adder, here the LUT is used for the XOR operation, while the carry logic is implemented with the supplementary circuits behind the LUT in the following way:

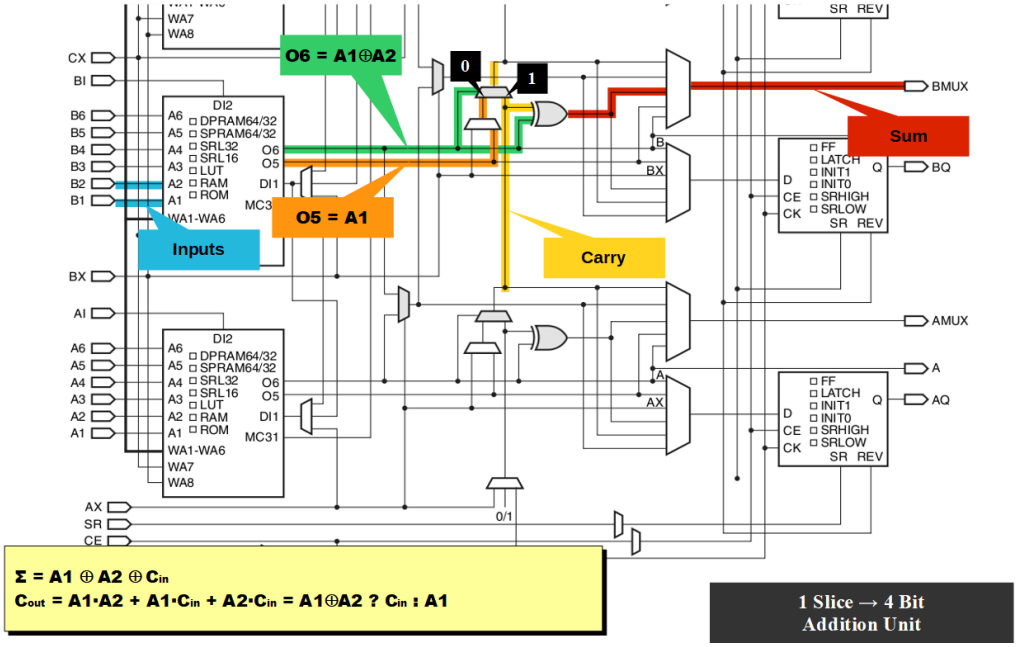

Figure 17.11: *Realizing an adder.*

17.4.3 Memory options

Two types of memory available in FPGAs were discussed shortly: the memory distributed in the slices and the block RAM. The distributed memory is typically organized in a bitwise manner and has a small size: 64-256 bits for each slice - as it is locally available, it is very fast, even asynchronous read can be used.

The block RAM is on the silicon of the FPGA, it is of intermediate size (18-36 kbits) RAM, which supports several types of bit organizations (1,2,4,8,9,18,36 and 72), including the parity bit. It is an important property that both read and write can only happen synchronously. This memory type can also be used as a FIFO. In practice, FPGAs also have an external memory chip, which is a separate DDR memory and can be used commonly with the microprocessor controlling the FPGA. This memory type is the biggest and slowest.

17.5 Examples

At the end let us see some examples for implementing image processing operations in hardware.

17.5.1 Thresholding

The most straightforward task is thresholding, for which we assume that pixels come from the camera in a serialized manner using the YCbCr color space (which is often available in cameras).

17.5.2 Convolution

In the case of convolution, our task is more complicated, as we need not only one pixel but also pixels from the previous and next rows too.

Thus, our hardware consists of a significant number of FIFOs, which delay the pixels with 1-1 image rows. The filter weights are loaded from the configurational unit and passed to the multipliers in a reverse order w.r.t. the pixels. We also need to implement boundary handling; thus products coming from outside the image should be neglected.

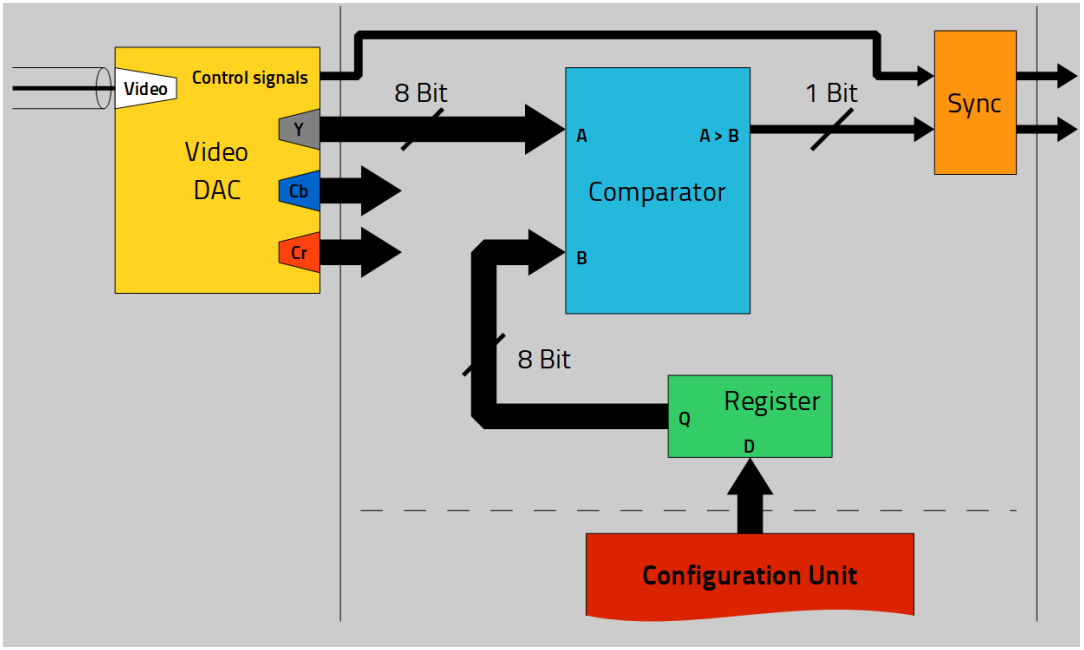

Figure 17.12: *Implementing thresholding with a block diagram.*

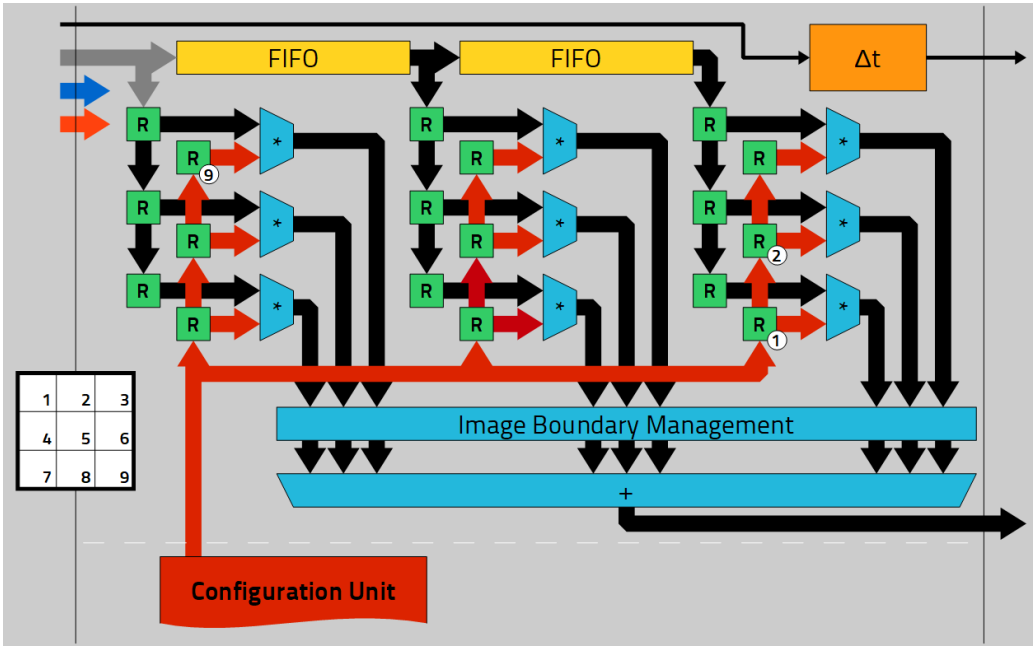

Figure 17.13: *The implementation of convolution with a block diagram.*

17.6 High-Level Design

In the end, we will discuss some of the essential aspects of hardware design with FPGAs. It is important to co-design the hardware and the software during FPGA development. As a first step, the tasks have to be assigned to one of the devices. Sequential algorithms should be implemented on the CPU, while algorithms that can be parallelized and accelerated, in the FPGA. During each step, it is reasonable to simulate hardware and software together.

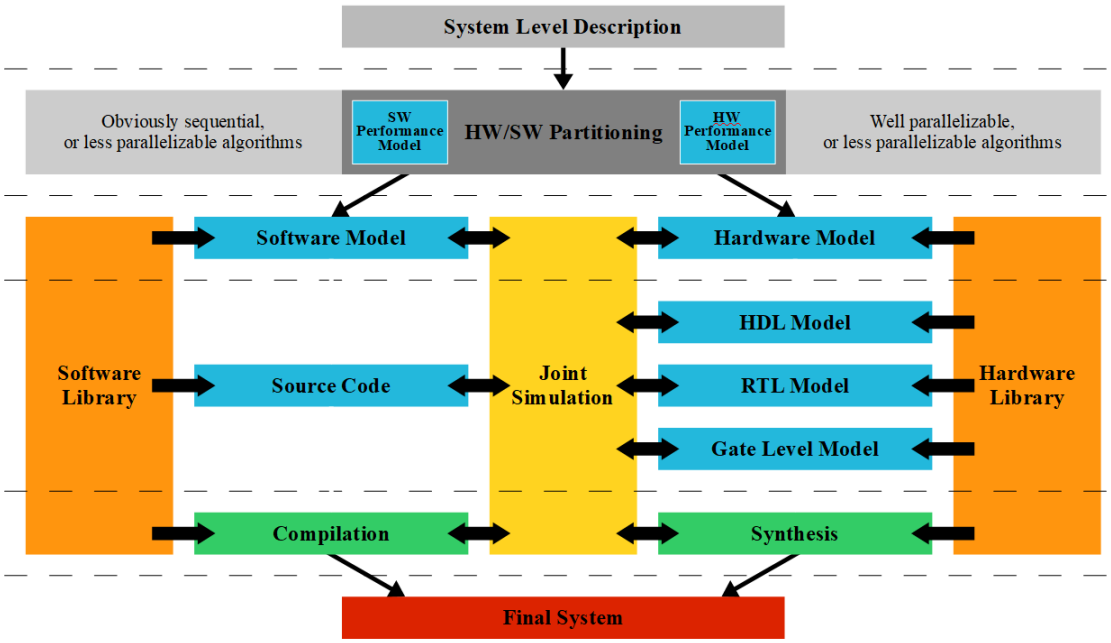

Figure 17.14: *The process of hardware-software co-design.*

17.6.1 The system of data-paths

FPGA-based data processing can be described as a so-called data-path system, where the data elements get processed as they pass through different functional elements. The whole processing is supervised by a separate control unit, which changes the path of the data elements based on the available conditions. This control unit can be a simple sequential circuit or a complete microprocessor system.

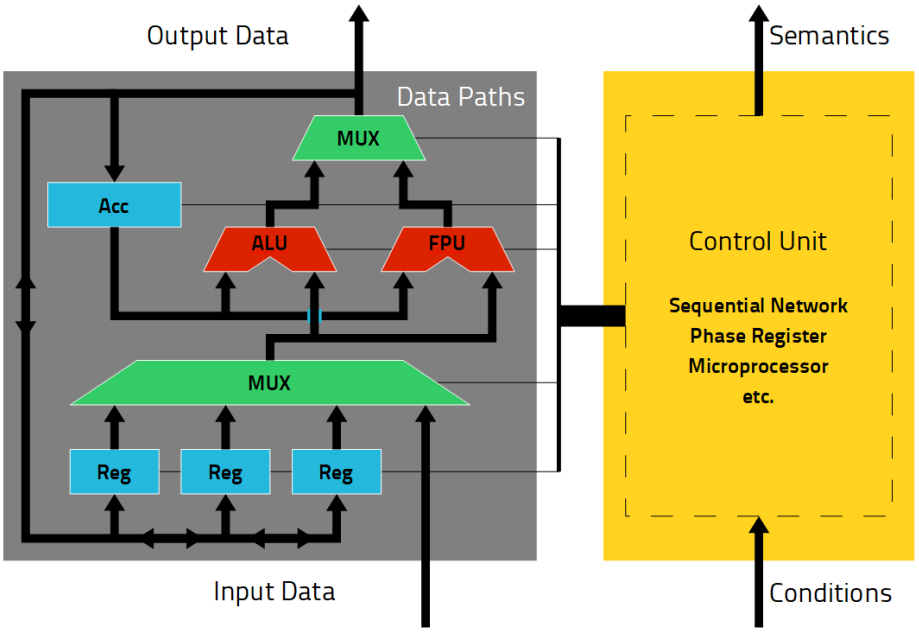

Figure 17.15: *The system of data-paths .*

17.6.2 Pipeline

The essential element of the system of data-paths is the algorithmic pipeline, which is passed by the data elements in a sequential way (one element at a time), which means for images that the operations are carried out pixelwise and serialized. As a result, although the processing of the whole image is not, by all means, fast (here parallelization is not in the game), but if the pipeline is built into the serial pixel-forwarding interface, processing causes only a minimal (in the example below measured only in a few rows and pixels) delay.

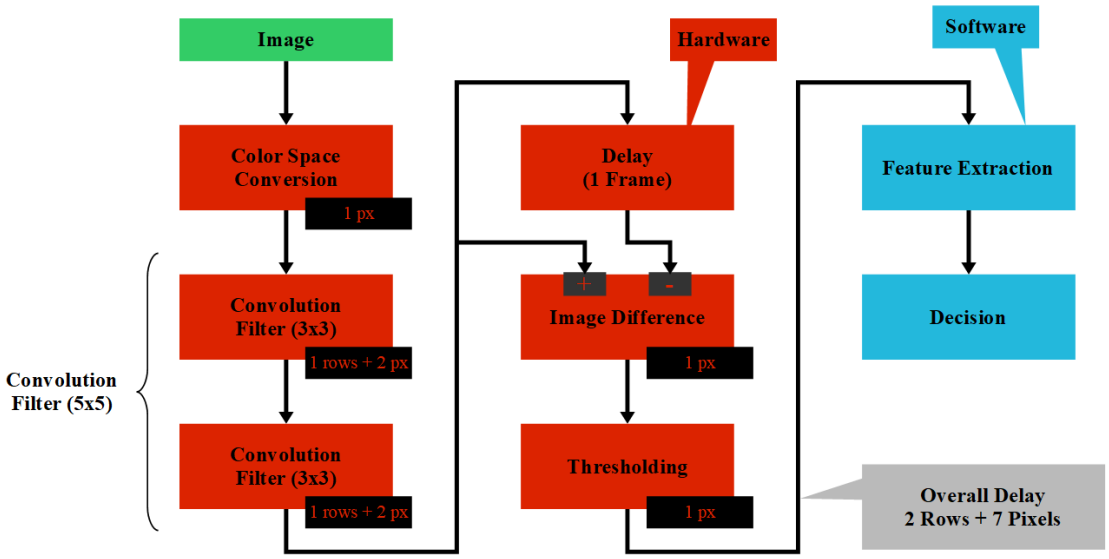

Figure 17.16: *Hardware pipeline realizing a thresholded image difference.*

17.6.3 Reconfiguration

An essential property of the pipeline is to change the path of the data elements, i.e., to have a programmable pipeline - for which a configurable bus system should be placed between the algorithmic blocks.

Besides the simple variant, there is also another type besides the controllable pipeline, called the superscalar pipeline. Superscalar processing means that the hardware manages and uses multiple, parallel pipelines, with the use of which data- or task-level parallelization can be implemented on FPGAs.

Besides that, there is also such a solution, where not only the pipeline but also the whole hardware will be reconfigurable. This can be a necessity if - based on the outcomes of events - the processing task can change, or if we would like to schedule tasks between multiple processing units in an adaptive manner. The latter can be done to optimize workload or energy consumption - it is also possible to do this to allocate tasks with different priorities onto devices with different performance (i.e., here the priority changes in runtime).

Further Reading

[46] S. Kilts, *Advanced FPGA Design*. John Wiley & Sons, Inc., Jun. 2007. DOI: 10.1002/9780470127896. [Online]. Available: https://doi.org/10.1002%2F9780470127896.

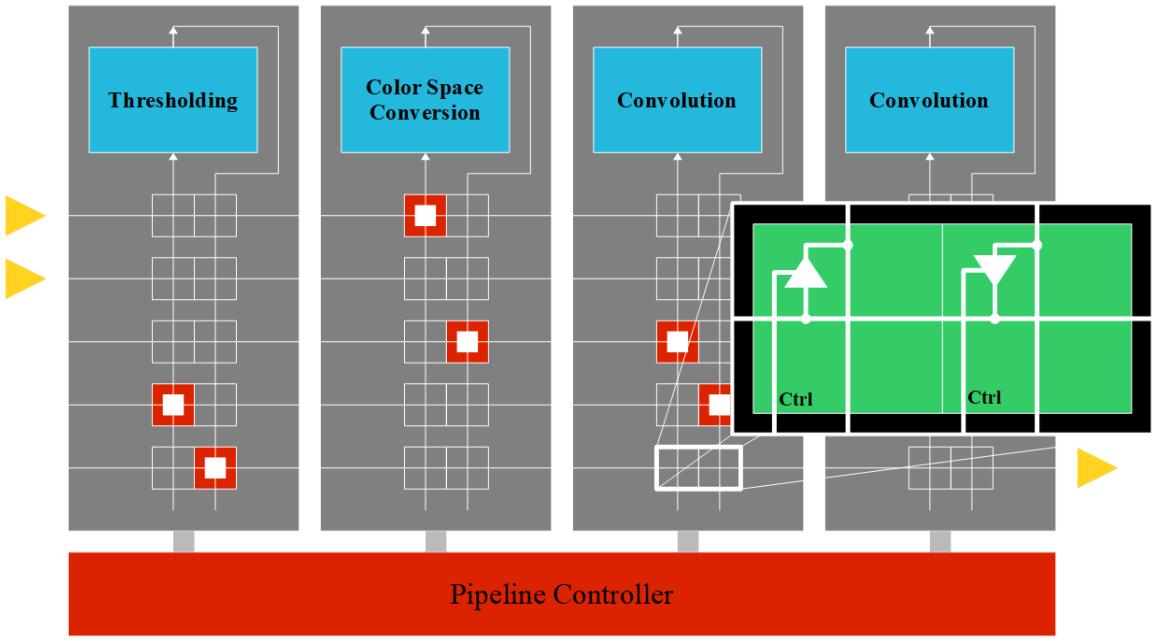

Figure 17.17: *The reconfigurable pipeline.*

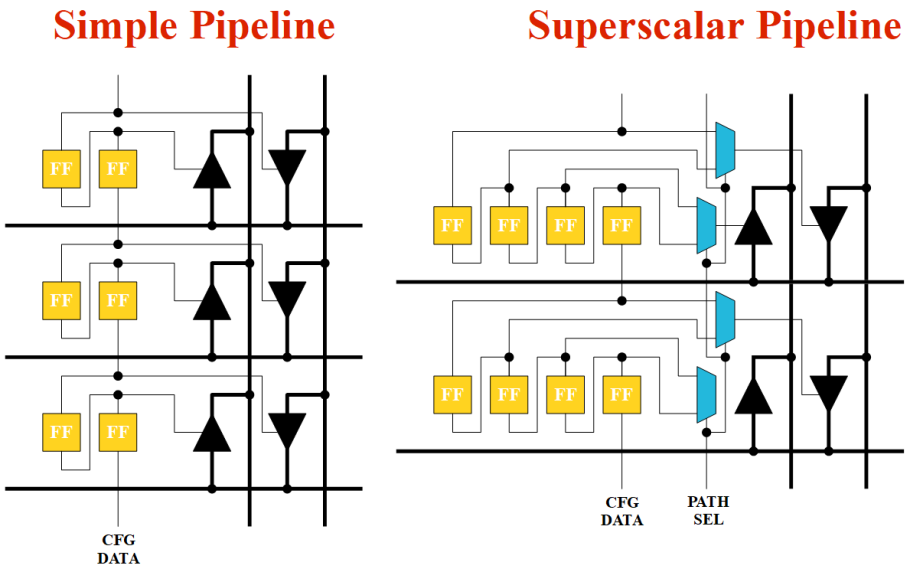

Figure 17.18: *Traditional (left) and superscalar (right) pipeline controllers.*

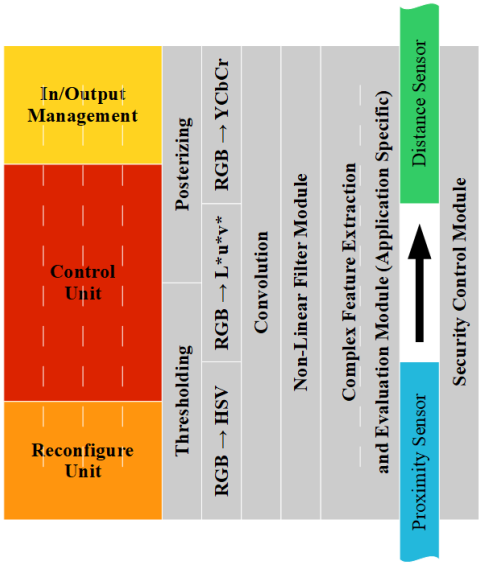

Figure 17.19: *Reconfigurable hardware.*

Acknowledgements

Many thanks to Patrik Reizinger for translating this work to english.

Also, many thanks to Dr. Ferenc Vajda, who developed the curriculum of this course, and created a fair share of the figures.